Theory of Algorithms
Practice problem solutions

Viktória Nemkin, Márta Pintér
Lectored by: Attila Sali

2024

1. Balls are distributed in n boxes, such that in the k^{th} box there are k balls. In one step, we can choose an arbitrary subset of the boxes and take out the same number of balls from each. Give an optimal algorithm that uses the smallest number of steps to empty all the boxes.

Solution:

Think of each box's index as a binary number. During the i^{th} step, we remove 2^i balls from every box whose index has the bit for that step set to 1.

By making iterations from $i = 0$ to $\lceil \log_2 n \rceil$, we removed all balls from all boxes, so the number of steps needed is $O(\log_2 n)$.

We can't have a solution with fewer steps because that would imply we could generate all numbers up to n with just $\lceil \log_2 n \rceil - 1$ different numbers. But with $x = \lceil \log_2 n \rceil - 1$ numbers available, we only have 2^x combinations, which is less than n. This means we would miss at least one number between 1 and n, proving such a solution is not possible.

2. Let M be a natural number of n bits length. Give an algorithm to determine $\lfloor \sqrt[3]{M} \rfloor$ whose running time is polynomial of n.

Solution:

To find the solution x for $x^3 \leq M$ but $(x+1)^3 > M$, we use a binary search approach:

Start with $x = 0$ and guess each bit of x from the most significant bit to the least significant bit. For each bit, try setting it to 1 and check: If x^3 is exactly M, we've found the answer. If it is greater, unset the bit back to 0 if it is less, keep the bit as 1. Proceed until all bits are checked to get x in binary.

Proof of correctness relies on maintaining the invariant property that the already determined bits of x at each step form the correct prefix of the solution. If setting a bit makes x too big, we know that bit shouldn't be set because we can't decrease x later. If it's still too small, that bit needs to be set to reach M, since not setting it would make it impossible to get close enough to M with the remaining bits. For a given bit position b, the value of setting this bit, 2^b, is greater than the sum of all lower bit values combined, which is $2^{b-1} + 2^{b-2} + ... + 2 + 1 = 2^b - 1$.

3. We are given n chips that can test each other in the following way: if two of the chips are connected, then both determine whether the other one is faulty or correct. A correct chip decides correctly about the other, while a faulty chip can give an arbitrary answer. Assume that more than half of the chips are correct. Give an algorithm that uses less than n tests of the above type to find a correct chip.

Solution:

Same as Exercise 13 from Practice session 1, the solution can be read there.

4. The Russian Czar in St. Petersburg wants to send a letter to the Chinese Emperor in Beijing using carrier pigeons. There are n cities in the route, $v_1(=\text{ St. Petersburg}), v_2, \ldots, v_n(=\text{ Beijing})$. It is known for each pigeon from which city to which other city they can fly and how long it takes them to get there. To avoid unnecessary detours, if a pigeon flies from city v_i to city v_j, then $i < j$. Help the Czar to plan the fastest way to deliver the letter.

Solution:

This exercise can be modeled using graphs, the vertices of the graph represent the cities on the route and the edges of the graph represent the pigeons. If there is an edge pointing from v_i to v_j, it means that a pigeon can fly from the city of v_i to the city of v_j. The time it takes for the pigeon to fly is represented by the edge weight. Then, we can use the Dijkstra algorithm to find the shortest path from v_1 to all other vertices, including v_n, which takes $O(n^2)$ time.

Since the graph is a DAG and the vertices are already in topological order, we can also use Dynamic Programming to solve the exercise. We create a DP array where $DP[i]$ is the shortest time to reach city v_i from v_1. For each city, calculate $DP[i]$ by taking the minimum of $DP[j] + w_{j,i}$ for all cities v_j that have an edge pointing to v_i. Initialize $DP[1] = 0$ because the start city's distance to itself is zero. Fill out the DP array in order, leveraging the topological sort that guarantees all predecessors to a city are done before the city needs to use them. The final answer, the shortest path to the last city v_n, is found in $DP[n]$. This solution involves $O(e)$ calculations for the edges and $O(n)$ for finding minimums, where e is the number of edges, for a total of $O(n + e)$ steps.

5. An array $A[1 : n]$ contains red and green elements. We want to rearrange the elements so that all the red elements precede the green elements, or all of them will come after the green elements. A legal step in rearranging them is swapping the position of two adjacent elements. We count swapping two adjacent elements as a step. Give an optimal algorithm for this.

Solution:

Same as Exercise 12 from Practice session 1, the solution can be read there.

6. In a classroom, there is a square board of size $n \times n$, equipped with a total of n^2 lightbulbs. Alongside each row and column, there is a switch that toggles the state of all lightbulbs in that row or column (turning on all the bulbs that are off and turning off all those that are on). Initially, all the lightbulbs were turned off. However, during the break, students tinkered with the switches. What is the minimum number of switch presses needed for the teacher to switch off all the lightbulbs again? What strategy should they use?

Solution:

Pressing a switch more than once is unnecessary, as two presses cancel each other out. Thus, at most, $2n$ presses are sufficient.

If toggling some switches achieves the all-off state, doing the opposite (pressing all unpressed switches and not pressing any previously pressed) will also result in all bulbs being off. This is because a bulb remains unchanged if both or neither of its row and column are pressed and if only one (either row or column) is pressed, pressing the other in the inverse action also results in a change. This means that n presses will be enough. If a solution uses $n + x$ presses, then the inverse will use $n - x$.

The algorithm works as follows (calculate on paper first): We can always find a solution without toggling the first row. Then, for any lit bulb in the first row, we must toggle its column to turn it off. This should result in the other rows being either fully on or fully off. Finally, just turn off any rows that are fully on. If the number of steps taken on paper is more than n, then we instead do the inverse of the steps on the board.

This is optimal, since the all-lightbulbs-on board cannot be solved with less than n toggles.

7. Give a linear running time method that finds the longest *proper* initial segment of the word w, which is also a final segment of w. w is of length n and is given by a character array $w[1 : n]$. (A *proper* segment's length is between 1 to $n-1$.)

Solution:

The difficulty of this task exceeds the standard exam's requirements, it is a bonus task.

Refer to the calculation of the prefix function of the Knuth-Morris-Pratt algorithm `https://cp-algorithms.com/string/prefix-function.html`.

The final solution is in $LPS[n]$, where $LPS[i]$ stores the length of the longest prefix of $w[1:i]$, that is also a suffix of $w[1:i]$.

8. Which of the following functions are $O(n^2)$ and which are $\Omega(n^2)$?
$f_1(n) = 11n^2 + 100000$ $f_2(n) = 8n^2 \log_2 n$ $f_3(n) = 1.5n + 3\sqrt{n}$

Solution:

$f_1(n) \le 12n^2$, if $n \ge 1000$, so by choosing $c = 12$, $n_0 = 1000$ we have shown that $f_1 \in O(n^2)$.
Note: Actually $n_0 = \sqrt{100000} \approx 317$ is a good choice too, but we don't have to find the smallest possible n_0.
(An alternative solution is $f_1(n) \le 100011n^2$, if $n \ge 1$ so $c = 100011$, $n_0 = 1$ is also a good choice.)

Furthermore $f_1(n) \ge 11n^2$, so by choosing $c = 11$, $n_0 = 1$, we have shown that $f_1(n) \in \Omega(n^2)$ (and so $f_1(n) \in \Theta(n^2)$ is also true.)

$f_2(n) \notin O(n^2)$, because if $f_2(n) \le cn^2$ for some c constant, then $8 \log n \le c$, which is only true when $n \le 2^{c/8}$, not for an arbitrarily large n.

Furthermore, $f_2(n) \ge 8n^2$, so by choosing $c = 8$, $n_0 = 1$ we have shown that $f_2 \in \Omega(n^2)$.

$f_3(n) \le 1.5n + 3n = 4.5n$ is always true when $n \ge 1$, so by choosing $c = 4.5$, $n_0 = 1$, we have shown that $f_3(n) \in O(n) \subseteq O(n^2)$.

Furthermore, if $f_3(n) \ge cn^2$ for some c constant, then $cn^2 \le f_3(n) \le 4.5n$ is also true, so then $n \le \frac{4.5}{c}$, which is not true for an arbitrarily large n, so $f_3 \notin \Omega(n^2)$.

9. Assume that in the pseudocode on the right, printing one star counts as one step. Show that the number of steps is $O(n^3)$.

```
for i = 0 to n-1:
    for j = i+1 to n:
        print j copies of *
```

Solution:

The outer loop runs at most n times. Inside of it, the inner loop also runs at most n times. During each iteration of the inner loop, the program prints up to n stars, and this action is considered to take n steps according to the exercise. Together, these nested loops then account for at most $n \cdot n \cdot n = n^3$ steps, which is $O(n^3)$.

10. (a) Show that the running time of Bubble sort is $O(n^2)$. (This was introduced in Prog1, the pseudocode is repeated on the right). Is it also true that the running time is $O(n^3)$?
(b) Further check that the number of steps is $\Omega(n^2)$.
(c) Is it true that the number of steps or the running time is $\Theta(n^2)$?

```
for i = n to 2:
    for j = 1 to (i-1):
        if A[j] > A[j+1]:
            swap A[j] and A[j+1]
```

Solution:

(a) The core of the inner loop takes a constant number of steps, and the inner loop runs at most n times because $i - 1 \le n - 1$, making the number of steps for the inner loop $O(n)$. The outer loop runs at most n times, with its core being the $O(n)$ steps inner loop, thus the outer loop (and therefore the entire code) is $O(n^2)$.

If a function is of order $O(n^2)$, then it is also true that it is $O(n^3)$ based on the definition.

(b) Considering the input where the numbers are in decreasing order, the execution involves $n-1+n-2+\ldots+2+1=\frac{n(n-1)}{2}$ comparisons and the same number of swaps, meaning the total number of steps is $n(n-1)\geq n\cdot\frac{n}{2}=0.5n^2$, provided that $n\geq 2$.

(c) Since both $O(n^2)$ and $\Omega(n^2)$ are true, $\Theta(n^2)$ is also true.

---

11. Put the following functions in order of non-decreasing order of magnitude, i.e., if f_i is immediately followed by f_j in the order, then $f_i(n)\in O(f_j(n))$ holds.
$f_1(n)=8n^3$ $\quad f_2(n)=5\sqrt{n}+1000n$ $\quad f_3(n)=2^{(\log_2 n)^2}$ $\quad f_4(n)=1514n^2\log_2 n$

Solution:

We provide an order and justify its correctness: f_2,f_4,f_1,f_3
$f_2(n)\leq 1005n\leq 1005n^2\log n\leq f_4(n)$, with $c=1,n_0=2$ is good.
$f_4(n)\leq 1514n^3=\frac{1514}{8}f_1(n)$, with $c=\frac{1514}{8},n_0=1$ is good.
Notice that $f_3(n)=(2^{\log n})^{\log n}=n^{\log n}\geq n^3$, if $\log n\geq 3$, hence $c=8$, $n_0=2^3=8$ is good.

12. The InsertionSort of an array $A[1:n]$ (using BinarySearch) consists of $n-1$ loops: in the i^{th} loop, where $1\leq i\leq n-1$, we search for the position of the $A[i+1]^{\text{th}}$ element in the array $A[1:i]$ using BinarySearch. We then swap the $A[i+1]^{\text{th}}$ element with its neighbors until it reaches its desired position.
(a) Show that the BinarySearch algorithm done in the lecture can be modified to obtain the position of the new element when it is not in the array.
(b) If the running time is counted in the number of comparisons and swaps, check that the running time of such an Insertion sort is $O(n^2)$.
(c) Show that the running time is in fact $\Omega(n^2)$ and hence, $\Theta(n^2)$.

Solution:

(a) The BinarySearch algorithm done in the lectures will return "not found" when the element is not in the array. Instead of looking for an equal element, we can look for the smallest index in the array, that contains a number larger than or equal to the searched element. (See for example the `std::lower_bound` function in C++.)

(b) In InsertionSort there are at most n rounds, and in each round we perform a BinarySearch in an array of at most n in size, and after the search, at most n swaps are done to place the current element in its correct position. This means that the number of steps per round is $O(\log n+n)=O(n)$, and since there are at most n rounds, the total number of steps for the entire execution is $n\cdot O(n)=O(n^2)$.

(c) If the numbers in the input array are in decreasing order, then the number of swaps is $1+2+\ldots+n-2+n-1=\frac{n(n-1)}{2}\geq n\cdot\frac{n}{4}=0.25n^2$, if $n\geq 2$. By choosing $c=0.25$ and $n_0=2$ we have shown that $\Omega(n^2)$ is true. Since both $O(n^2)$ and $\Omega(n^2)$ are true, $\Theta(n^2)$ is also true, according to the definition.

13. Let $f_1(n)=1.5\,n!$ and $f_2(n)=200\,(n-1)!$. Which of the following is true?
$f_1\in O(f_2)$ $\quad f_2\in O(f_1)$ $\quad f_1\in\Omega(f_2)$ $\quad f_2\in\Omega(f_1)$ $\quad f_1\in\Theta(f_2)$ $\quad f_2\in\Theta(f_1)$

Solution:

Notice that $f_1(n)=\frac{1.5}{200}\cdot n\cdot f_2(n)=c\cdot n\cdot f_2(n)$. From this, it follows that $f_1\notin O(f_2)$, $f_2\in O(f_1)$, $f_1\in\Omega(f_2)$, $f_2\notin\Omega(f_1)$, therefore $f_1\notin\Theta(f_2)$ and $f_2\notin\Theta(f_1)$.

14. Give a bound for the following functions using the Big-O notation:
$(n^2+8)(n+1)$ $\quad (n\log n+n^2)(n^3+2)$ $\quad (n!+2^n)(n^3+\log(n^2+1))$ $\quad (2^n+n^2)(n^3+3^n)$

Solution:

There are many good solutions, generally we want to place a simple yet as small as possible function into O. Now, we do it in such a way that we hardly need to calculate. For this, we provide estimates for the orders of magnitude of the occurring sums and multiply these estimates together.

$(n^2+8)(n+1) \leq 2n^2 \cdot 2n \in O(n^3)$ (the estimate is true for all $n \geq 3$).
$(n\log n + n^2)(n^3+2) \leq 2n^2 \cdot 2n^3 \in O(n^5)$ (the estimate is true for all $n > 1$).
$(n! + 2^n)(n^3 + \log(n^2+1)) \leq 2n! \cdot 2n^3 \in O(n^3 n!)$ (the estimate is true for all $n \geq 4$, from which $n! > 2^n$).
$(2^n + n^2)(n^3 + 3^n) \leq 2 \cdot 2^n \cdot 2 \cdot 3^n$ (the estimate is true for all $n \geq 4$, because from here both $2^n \geq n^2$ and $3^n > n^3$ are also satisfied).

---

15. There are two algorithms for the same problem, A and B. Let the functions describing their worst case running times be f_A and f_B. It is known that $f_A(n) \in O(f_B(n))$. Does it follow that
a) A is faster on every input than B?
b) except for finitely many inputs A is faster than B?
c) for large enough inputs A is faster than B?

Solution:

The answer is no *for each one, for example, in the case of $f_A(n) = 2n$ and $f_B(n) = n$, it's true that $f_A(n) \in O(f_B(n))$, but $f_A(n) > f_B(n)$ for all n.*

16. Show that if f_1, f_2, g_1, g_2 are positive valued functions, and $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then
(a) $f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n))$
(b) $f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n))$.

Solution:

$\exists c_1, n_1$, such that $f_1(n) \leq c_1 g_1(n)$, when $n \geq n_1$,
$\exists c_2, n_2$, such that $f_2(n) \leq c_2 g_2(n)$, when $n \geq n_2$,
then
$f_1(n) + f_2(n) \leq \max(c_1, c_2)(g_1(n) + g_2(n)) \leq 2\max(c_1, c_2)\max(g_1(n), g_2(n))$, when $n \geq \max(n_1, n_2)$,
and
$f_1(n) f_2(n) \leq c_1 c_2 g_1(n) g_2(n)$, when $n \geq \max(n_1, n_2)$.

17. For which $a, b > 1$ natural numbers are the following true?
$n^a \in O(n^b)$ $\qquad$ $2^{an} \in O(2^{bn})$ $\qquad$ $\log_a n \in O(\log_b n)$

Solution:

In the first case, it is required that $n^a \leq cn^b$, meaning $n^{a-b} \leq c$ for some constant $c > 0$ when $n \geq n_0$. In the case of $a = b$, this is obviously satisfied, $n^a \in O(n^a)$ $\quad (c = 1,\ n_0 = 1)$.
In the case of $a < b$, the exponent is negative, thus $n^{a-b} \leq 1$, so choosing $n_0 = 1$ and $c = 1$ is sufficient.
However, if $a > b$, then the exponent is positive, the function is monotonically increasing, and tends to infinity, so such constant c cannot exist.

For the second case, similar to the previous one, $2^{an-bn} \leq c$ is needed, which is satisfied for $a \leq b$ with the choice of $c = 1$, $n_0 = 1$, but if $a > b$, then no such constant c exists.

In the third case, use the fact that $\log_a n = \frac{\log_b n}{\log_b a}$, thus $c = \frac{1}{\log_b a} > 0$ works, whatever a and b are. ($n_0 = 1$)

18. Assume that n is a power of 2. Show that we can search for the two largest elements in an array of n distinct elements with $n + \log_2 n - 2$ comparisons.

Solution:

First idea: We determine the smallest element, and then we determine the smallest among the rest, which requires $n - 1 + n - 2 = 2n - 3$ comparisons, which is too many.

Second idea: We can do better than this.

Let's think of a single-elimination tournament. The elements are paired in each round and the bigger is eliminated from each pair until there is only one left, the smallest elment. So far, this was done with with $n - 1$ comparisons, since each comparison eliminates 1 element, until we have 1 left from the n.

Observation: The second smallest could only have been eliminated by the smallest! No other element can defeat it.

So the second smallest was one element that competed against the smallest in one of the rounds. Since each round halves the number of elements, until we reach 1, we had to have had k rounds, where $\frac{n}{2^k} = 1$, so $k = \log_2(n)$.

So there are $\log_2(n)$ possible choices for the second smallest element, we can find the smallest among them with $log_2(n) - 1$ comparisons.

Therefore, the total number of comparisons is exactly $n + \log_2 n - 2$.

19. An array $A[1:n]$ contains red and green elements. We would like to rearrange the elements so that all the red elements preceed the green elements, or all of them will come after the green elements. A legal step in rearranging them is swapping the position of two adjacent elements. We count swapping two adjacent elements as a step. Give an optimal (up to a constant factor) algorithm for this.

Solution:

It's clear that it only makes sense to swap elements of different colors. If initially, the first and the last $n/4$ elements are red, then no matter which end we want to sort the reds to, one of the $n/4$ length interval of red elements must swap places with all of the greens, so it may be necessary to take $n^2/8$ steps (the required steps is $\Omega(n^2)$). It can be done with $O(n^2)$ steps since any order can be achieved with fewer than n^2 neighbouring swaps, for example, by using bubble sort (where red < green, if we want to move the reds to the front of the array).

20. We are given n chips that can test each other in the following way: if two of the chips are connected, then both determine whether the other one is faulty or correct. A correct chip always answers properly about the other, while a faulty chip can give an arbitrary answer. Assume, that more than half of the chips are correct. Give an algorithm that uses less than n tests of the above type to find a correct chip.

Solution:

Depending on the two connected chips, the following outputs can be seen:

1st chip	*2nd chip*	*1st output*	*2nd output*
good	*good*	*good*	*good*
good	*bad*	*bad*	*good/bad*
bad	*good*	*good/bad*	*bad*
bad	*bad*	*good/bad*	*good/bad*

Prepare 3 boxes: T=tested, U=untested, G=garbage. Initially, every chip is in the U box. Take two out. If a pair gives a good-good result, then they are the same, either good-good or bad-bad. When this happes put both of them in the T box. If they do not give good-good, then at least one is bad, in which case throw both in the G box.

If both the U and T boxes are not empty, then always pair one from the T box with one from the U box. If they output good-good, then put both in the T box, otherwise in the G box. If the T box is emptied, then take two from the U box again, if there are still two left.

Before the tests and after each test, the following two properties (so-called invariant properties) hold:

(1) In $T \cup U$, there are more good chips than bad ones.

(2) In T, there are always chips of the same kind (either all good or all bad, but it can also be empty).

Because of (1), after the tests $T \cup U$ cannot be empty.

We continue testing as long as there is a test that can be performed. What happens at the end when we can no longer perform any further tests? Then, in U, there are 0 or 1 elements. If there are 0, then because of (1) and (2) T is not empty and consists only of good chips.

If there is only one element in N, then T is empty and because of (1) U can only contain a good chip. In both cases, it's true that in the end, $T \cup U$ contains only good chips.

Initially, we removed 2 elements from U for testing and after that each subsequent test involved taking out at least 1 element from U. Therefore, we conducted no more than $n-1$ tests in total.

21. Sort the sequence $3, 12, 1, 34, 4, 6, 0$ using InsertionSort. How many comparisons were used?

Solution:

There are two types of InsertionSort: using linear and using binary search. The sequences obtained per insertion are the same, but the number of comparisons may differ.

The procedure is to insert the i-th element into the already sorted first $i-1$ elements, so the sequence changes in the following way during the steps (the sorted part is shown underlined, the next element in bold):

$(\underline{3}, \mathbf{12}, 1, 34, 4, 6, 0) \longrightarrow (\underline{3, 12}, \mathbf{1}, 34, 4, 6, 0) \longrightarrow (\underline{1, 3, 12}, \mathbf{34}, 4, 6, 0) \longrightarrow (\underline{1, 3, 12, 34}, \mathbf{4}, 6, 0) \longrightarrow$
$(\underline{1, 3, 4, 12, 34}, \mathbf{6}, 0) \longrightarrow (\underline{1, 3, 4, 6, 12, 34}, \mathbf{0}) \longrightarrow (\underline{0, 1, 3, 4, 6, 12, 34})$.

The number of comparisons for linear search (assuming we search from back to front) is: $1+2+1+3+3+6 = 16$, *for binary search it is:* $1+2+2+2+3+3 = 13$.

22. Sort the sequence $16, 17, 2, 6, 11, 33, 28, 22$ using QuickSort, so that always the first element of the array is chosen as pivot.

Solution:

$(\mathbf{16}, 17, 2, 6, 11, 33, 28, 22) \longrightarrow (\mathbf{6}, 11, 2 \mid 16 \mid \mathbf{17}, 33, 28, 22) \longrightarrow (2 \mid 6 \mid 11 \mid 16 \mid 17 \mid \mathbf{33}, 28, 22) \longrightarrow$
$(2 \mid 6 \mid 11 \mid 16 \mid 17 \mid \mathbf{22}, 28 \mid 33) \longrightarrow (2 \mid 6 \mid 11 \mid 16 \mid 17 \mid 22 \mid 28 \mid 33)$

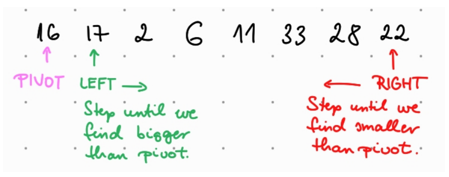
More detailed explanation:

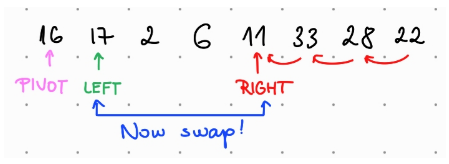
QuickSort sets the pivot to be the first element of the array, then initializes two pointers: left (beginning of the remaining array) and right (end of the remaining array).

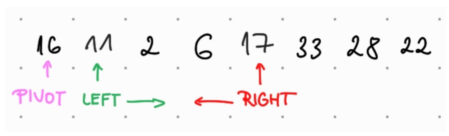
It moves the left pointer to the right until it finds an element larger than the pivot and moves the right pointer to the left until it finds an element smaller than the pivot. When both pointers found such an element, they stop and swap them.

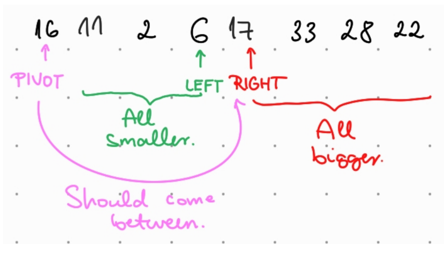
When the pointers meet, everything to the left is smaller and everything to the right is bigger, so the pivot must be swapped in-between.

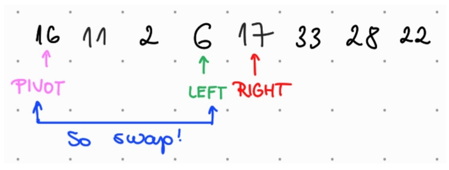
At this point the pivot is in its final position and QuickSort calls itself recursively twice, once on the left and once on right part of the array, to sort those as well.

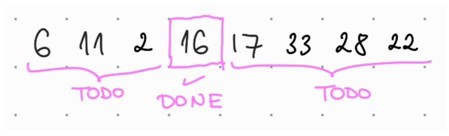

16 17 2 6 11 33 28 22
PIVOT
LEFT
RIGHT
Step until we find bigger than pivot.
Step until we find smaller than pivot.

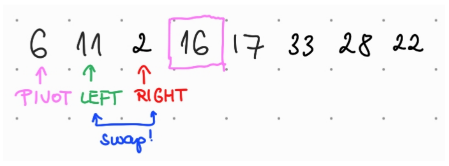


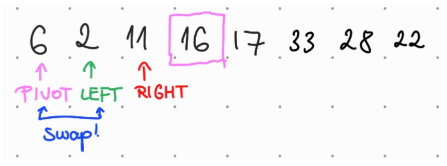


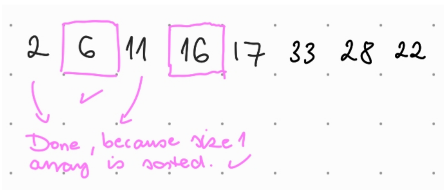


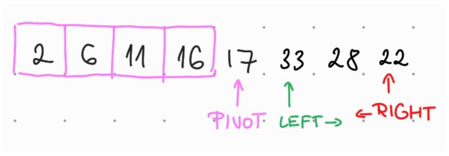


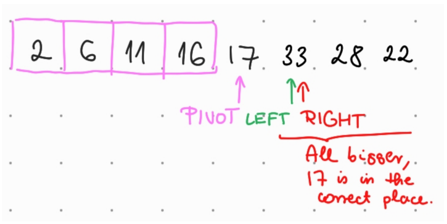


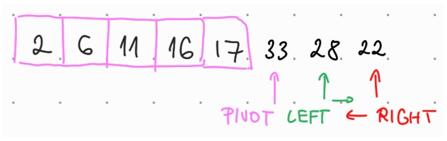


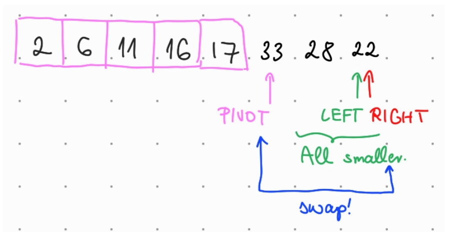


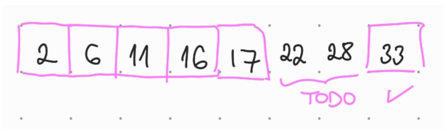


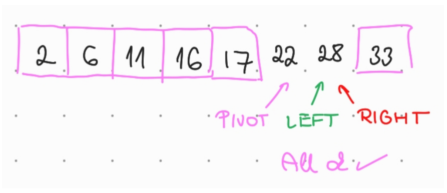




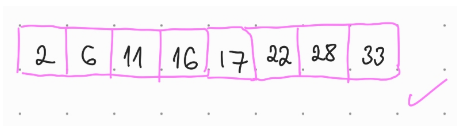








And we are done:



23. Sort the following list using RadixSort: *abc*, *acb*, *bca*, *bbc*, *acc*, *bac*, *baa*.

Solution:

bca, *baa*, *acb*, *abc*, *bbc*, *acc*, *bac*
baa, *bac*, *abc*, *bbc*, *bca*, *acb*, *acc*
abc, *acb*, *acc*, *baa*, *bac*, *bbc*, *bca*

24. Dr. Watson states to Sherlock Holmes that he knows a comparison-based sorting algorithm that sorts an array of any size in such a way that it is guaranteed to compare each element of the array at most 2025 times. How can Sherlock Holmes convince Watson that his algorithm must be incorrect?

Solution:

If every element appears in at most 2025 comparisons, then the algorithm performs a total of $\frac{2025n}{2}$ *comparisons with* n *elements. However, we know that every comparison-based sorting algorithm requires at least* $\log n! = \Omega(n \log n)$ *comparisons, and* $\frac{2025n}{2} \notin \Omega(n \log n)$*, therefore such an algorithm does not exist.*

---

25. Give an $O(n)$ running time algorithm that sorts a sequence of n integer numbers whose elements are from the set
(a) $\{1, \ldots, 3n\}$,
(b) $\{1, \ldots, n^3 - 1\}$.

Solution:

(a) Use BinSort with $3n$ *bins. In this case, the number of steps is* $O(n + 3n) = O(n)$.
(b) Simple BinSort is not sufficient here, use RadixSort instead.

Imagine the numbers written in base n*. Then each number is (at most)* 3 *digits long. If we use RadixSort in this form, we need* 3 *rounds of BinSort, each with* n *bins, so the whole process indeed uses* $O(n)$ *steps.*

Note: Converting to base n can also be done with $O(n)$ arithmetic operations, since every number must be divided by n with remainder, the remainder gives the last digit, the quotient is then divided by n again, this time the remainder gives the next digit, and the quotient gives the first one.

26. We are given an array of n not necessarily distinct integers. Our task is to design an algorithm with running time $O(n \log n)$ that determines the modes (most frequent elements) of the array, that is, the elements which occur the maximum number of times in the array.

Solution:

Begin by sorting the array with MergeSort, because this is the only algorithm we have learned that has a worst-case $O(n \log n)$ complexity. Now identical values are placed next to each other.

Next, find the longest sequence of the same number in the now-sorted array. To do this, iterate through the array while keeping tabs on the length of the longest sequence you've encountered and which number(s) belong to such sequences.

Keep track of the current sequence's length during the iteration, when it ends, check its length: if it's longer than any previously recorded, update your record with this new length and reset the corresponding values to the current one. If it matches the length of your record, just add the number to your list of values. Beyond the sorting phase, this process takes an additional $O(n)$ steps, bringing the overall complexity to $O(n \log n)$.

27. QuickSort is run on an array and the result after the first Partition is $4, 2, 3, 1, 6, 8, 11$. Which could be the pivot element?

Solution:

We need to find an element such that all elements before it are smaller, and all elements after it are larger. Thus, 6, 8, *and* 11 *are suitable, and nothing else is.*

28. It is known for the array $A[1:n]$ that $A[1] \neq A[n]$. Give an $O(\log n)$ running time algorithm that finds an index i such that $A[i] \neq A[i+1]$.

Solution:

We use an algorithm similar to BinarySearch: Let $i = \lceil \frac{n}{2} \rceil$. If $A[i] \neq A[1]$, then the subarray $A[1:i]$ has the same property as the original array, and it suffices to search within it. However, if $A[i] = A[1]$, then obviously $A[i] \neq A[n]$, and we can continue the search in the subarray $A[i:n]$. Thus, at this step, we are guaranteed to get a smaller array if $n \geq 3$.

At the end of the procedure, the array will be reduced to two adjacent, unequal elements, at which point we are done.

The number of steps, similarly to BinarySearch, will be $O(\log n)$.

29. Let the array $A[1:n]$ contain integers and let integer b be given. We would like to decide whether there exist indices $i, j \in \{1, \ldots, n\}$ such that $A[i] + A[j] = b$. Solve this problem in $O(n \log n)$ time.

Solution:

Notice that if we want to try every possible pair of indices, then there are $\binom{n}{2} \in \Theta(n^2)$ possibilities, which is more than the desired number of steps. Sorting the array helps a lot here, so we do that in $O(n \log n)$ steps using MergeSort.

After that, for each of indices $i = 1, 2, \ldots, n$, we search the array for the value $b - A[i]$ using BinarySearch. If we find an index j, such that $A[j] = b - A[i]$, then a good pair of indices in the sorted array is (i, j).

We must report however where the indices were in the original array, before sorting. We can do this if we keep track of the original index as an additional data point for every element or if we do two linear searches for the obtained $A[i]$ and $A[j]$ values in the original array at the end.

Since we performed n binary and 2 linear searches, the total runtime is $O(n \log n + 2n) = O(n \log n)$.

Note: The part after sorting can also be solved linearly using the Two Pointers technique. Starting from indices $i = 1$ and $j = n$, generally, if $A[i] + A[j] < b$, we increment i by one, and if $A[i] + A[j] > b$, we decrement j by one. We stop when $A[i] + A[j] = b$ or if $i > j$—the latter case means there is no solution.

This algorithm is correct, since if $A[1] + A[n] < b$, it means that $A[1]$ even with the biggest possible value in the array, $A[n]$ together still cannot create a large enough sum, so we can increment i by one. Similarly if $A[1] + A[n] > b$, then $A[n]$, even with the smallest possible value in the array $A[1]$ together still cannot create a small enough sum, sum, so we can decrement j by one. And so on for all i and j.

---

30. The value of one element of the originally increasing sequence $a_1, a_2, \ldots, a_n$ is changed. How can the elements be put in increasing order using $O(n)$ steps?

Solution:

Notice that if after the change $a_i < a_{i+1}$ is true for every i, then although we cannot determine which one changed, it is not necessary; the sequence remains ordered.

However, if there is an i such that $a_i > a_{i+1}$, then this occurred because either a_i increased or a_{i+1} decreased.

The simplest (not necessarily the fastest) method is to insert these two problematic elements into the increasing sequence formed by the other elements.

This can be done with linear search, performing $O(n)$ steps. (In binary search, the number of comparisons is logarithmic, but even in that case, the movements may take linearly many steps.)

Note: It can be solved with just one insertion, intead of two. How?

31. Give an algorithm using a small number of comparisons to find the smallest two elements of an array of length n.

Solution:

Same as Exercise 11 from Practice session 1, the solution can be read there.

32. A sequence of pairwise distinct integers is called *bitonic* if it starts as increasing and then from one point on becomes decreasing, or vice versa, starts decreasing and turns into increasing at a point. For example the sequences $(1, 3, 7, 21, 12, 9, 5)$, $(9, 7, 5, 4, 6, 8)$ and $(1, 2, 3, 4, 5)$ are bitonic. Give an $O(n)$ running time sorting algorithm for bitonic sequences of length n.

Solution:

Idea: We split the sequence at the largest element. Each of the two resulting parts is sorted, and we can merge them into a single sorted sequence using the Merge operation we have learned for MegeSort.

Algorithm: By comparing adjacent elements a_i and a_{i+1} in the sequence, we can find the largest element: If $a_{x-1} < a_x$ and $a_x > a_{x+1}$, then a_x is the largest element. If there is no such element, i.e., $a_{i-1} < a_i$ for every i, then there is no further action needed because the sequence is already increasing.

So far, we've used $n - 1$ comparisons. We merge the increasing sequences $a_1, \ldots, a_x$ and $a_n, a_{n-1}, \ldots, a_{x+1}$ using the method we learned, requiring at most $n - 1$ comparisons, thus we are done in at most $2n - 2 \in O(n)$ comparisons. (The algorithm also involves moving elements, but overall this takes $O(n)$ steps.)

We can achieve this without explicitly finding a_x in advance. We can imagine having two sequences, one starting with a_1 and the other with a_n (and initially, we don't know where they end). Let $i = 1$ and $j = n$, and compare a_i and a_j. The smaller one becomes the next element in the sorted list, and if $a_i < a_j$, then we increase the value of i, otherwise we decrease the value of j. We continue in this manner, similar to merging. We finish when $i = j$ (and then this is the largest element). This procedure uses at most $n - 1 \in O(n)$

comparisons.

Note: Both methods work correctly even when the largest element is a_1 or a_n, in which case we originally have a sorted sequence (either increasing or decreasing).

33. The elements of a (not neccessarily sorted) array A of length n are pairwise distinct positive numbers. Give an algorithm that determines an integer $1 \le k \le n$ and chooses k distinct elements of array A, such that the sum of the chosen elements is at most k^3. If there is no such k, the algorithm should report that. The running time of your algorithm must be $O(n \log n)$.

Solution:

Notice that for a fixed value of k, if there is a solution, then the smallest k elements of the array also form a solution. If these are not valid, then there is no solution. Therefore, we need to examine whether there exists a value of k such that the sum of the smallest k elements is at most k^3.

To achieve this, we sort the array in $O(n \log n)$ steps, for example, using MergeSort. Let $k = 1$ and $T = A[1]$. If $T \le k^3$, then we are done; otherwise, we increase k and add $A[k]$ to T. If none of the values of $k \le n$ are suitable, then there is no solution.

The part after sorting uses n comparisons and $O(n)$ additions, so the total number of steps is $O(n \log n)$.

34. We are given n points on the plane whose coordinates are (a_1, b_1), $(a_2, b_2), \ldots, (a_n, b_n)$. We need to find a point $P = (x, y)$ on the plane such that the sum $\sum_{i=1}^{n}(|a_i - x| + |b_i - y|)$ is minimal. Give an $O(n \log n)$ running time algorithm that solves this problem.

Solution:

Notice, that the coordinates are independent from each other. To minimize the written sum, we can separately determine the minimum in the x coordinates and the minimum in the y coordinates.

Therefore, we need to find an x such that the sum $\sum_{i=1}^{n} |a_i - x|$ is minimized and a y such that the sum $\sum_{i=1}^{n} |b_i - y|$ is minimized.

To do this for x, imagine the points a_i on the number line. If x is smaller than the smallest a_i, then increasing x reduces the sum. As long as there are fewer a_i smaller than x than those larger than it, increasing x further reduces the sum. It starts to increase when there are more a_i smaller than x than those larger.

So, the optimal x is the median a_i according to the sorted order, if n is odd. In the case of even n, it doesn't matter where it is within the middle interval, for example, choosing $x = a_{n/2}$ is good.

The same logic works for y as well.

After this, the algorithm proceeds as follows: we sort the a_i values and let x be the $\lceil n/2 \rceil^{th}$ number in the sorted sequence. We do the same for y: sort the b_i values, and let y be the $\lceil n/2 \rceil^{th}$ number in the sorted sequence.

The number of steps for the algorithm, using MergeSort, is $O(n \log n)$, since everything else was a constant number of steps.

35. We are given n intervals $[a_1, b_1], \ldots, [a_n, b_n]$ on the number line. Our task is to calculate the total length covered by these intervals combined (that is, what is the total length of the union $\cup_{i=1}^{n}[a_i, b_i]$). Give an $O(n \log n)$ running time algorithm that solves this problem.

Solution:

The length of an interval is $b_i - a_i$, but the union may be less than the sum of these lengths if the intervals intersect. We need to handle these overlaps, more precisely, find the disjoint intervals that constitute the union.

We sort the endpoints (all $2n$ numbers) while keeping track of which one was the start point of an interval and *which one was the end point. The first part of the union starts at the smallest value (which necessarily is the start of an interval a_j).*

We initialize a counter to 0 and proceed in the sorted sequence, adding $+1$ for every interval start point encountered and adding -1 for every end point. We do this until the counter hits 0 again. This occurs at an endpoint (b_k), and we can see that here ends the first disjoint part of the union, with length $b_k - a_j$. If there are still elements left in the sequence, we continue in the same way, tallying up all the disjoint lengths.

To determine the number of steps, we see that after sorting, we just need to iterate the sequence, counting the start and end points. For determining the length, at most n subtractions are required, and then their results need to be added up. The number of steps for sorting is $O(2n\log(2n)) = O(n\log n)$, followed by $O(n)$ steps, resulting in a total of $O(n\log n)$ steps.

36. Give an algorithm that uses the smallest possible number of comparisons to determine the smallest and the largest elements of an array of length n.

Solution:

Separately, both the smallest and the largest elements can be found with $n-1$ comparisons, meaning we can solve the exercise with at most $2n-2$ comparisons. (In fact, with $2n-3$ as well, since after finding the smallest, we only need to check the other $n-1$ numbers for the largest.)

But, we can do even better than this, as follows: First, pair the elements up and compare the pairs with each other. Then, among each pair, select the smaller element. The smallest among them becomes the overall minimum. Similarly, among the larger elements of the pairs, find the maximum.

If n is even, the number of steps is $\frac{n}{2}$ comparisons among the pairs, then $\frac{n}{2}-1$ comparisons to find the minimum among the smaller pairs and $\frac{n}{2}-1$ comparisons to find the maximum among the bigger pairs, totaling up to $1.5n-2$.

If n is odd, instead of the last pair, consider the last 3 elements to determine the smallest and largest. This can be done with 3 comparisons, so $\frac{n-3}{2}+3$ comparisons first, then $2\cdot(\frac{n-1}{2}-1)$, for finding the minimum and maximum totaling in $1.5(n-1)$.

Note: It can be shown that $1.5n-2$ is indeed necessary. Let B be the number of elements that can potentially be both the smallest and largest, S those that can only be the smallest (we have seen them compared to a larger element), L those that can only be the largest (we have seen them compared to a smaller), and R the rest, for which both smaller and larger elements have been seen.

Initially, $|B| = n$, the others are empty, and in the end, B is empty, $|S| = |L| = 1$.

- *With each comparison between two elements from B, B decreases by two, while S and L each increase by one.*
- *If a comparison involves an element from B and another not from B, B decreases by one and there is always a possibility that either one of S or L increases while the other doesn't decrease (i.e., their union doesn't decrease).*
- *When comparing two elements from S or L, one of them moves to R.*
- *In all other cases, there can always be a result where neither S nor L changes.*

To completely empty B, it requires at least $\frac{n}{2}$ comparisons. Every element from B must first move to S or L, from which only one can move out at a time, so at least $n-2$ comparisons are required to empty them (almost). Hence, there must be at least $1.5n-2$ comparisons in total.

37. Use the adjacency matrix below to determine the number of vertices and edges in the undirected graph.

Determine the degrees of all vertices.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Solution:

There are as many vertices as there are rows, that is, 5. $d(v_i)$ is the number of 1s in the i-th row. $d(v_1) = 2, d(v_2) = 4, d(v_3) = 2, d(v_4) = d(v_5) = 3$. The number of edges is half the sum of the degrees according to the degree sum formula (from the Handshaking lemma), thus 7.

38. (a) How do we know that the adjacency matrix below belongs to a directed graph?
(b) Determine the in and out-degree of each vertex.

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Solution:

(a) The adjacency matrix of undirected graphs is symmetric, this one is not.
(b) $d_{out}(v_i)$ is the number of 1s in the i-th row. $d_{out}(v_1) = 2, d_{out}(v_2) = 1, d_{out}(v_3) = 3, d_{out}(v_4) = 1, d_{out}(v_5) = 0$. $d_{in}(v_i)$ is the number of 1s in the i-th column. $d_{in}(v_1) = 0, d_{in}(v_2) = 2, d_{in}(v_3) = 1, d_{in}(v_4) = d_{in}(v_5) = 2$.

39. What is the out-degree of each vertex in a directed graph whose adjacency list is as follows: **a**: b, c; **b**: e, d; **c**: d; **d**: f; **e**: f; **f**:- ? What is the in-degree of the vertex f? How can this be determined from the adjacency list?

Solution:

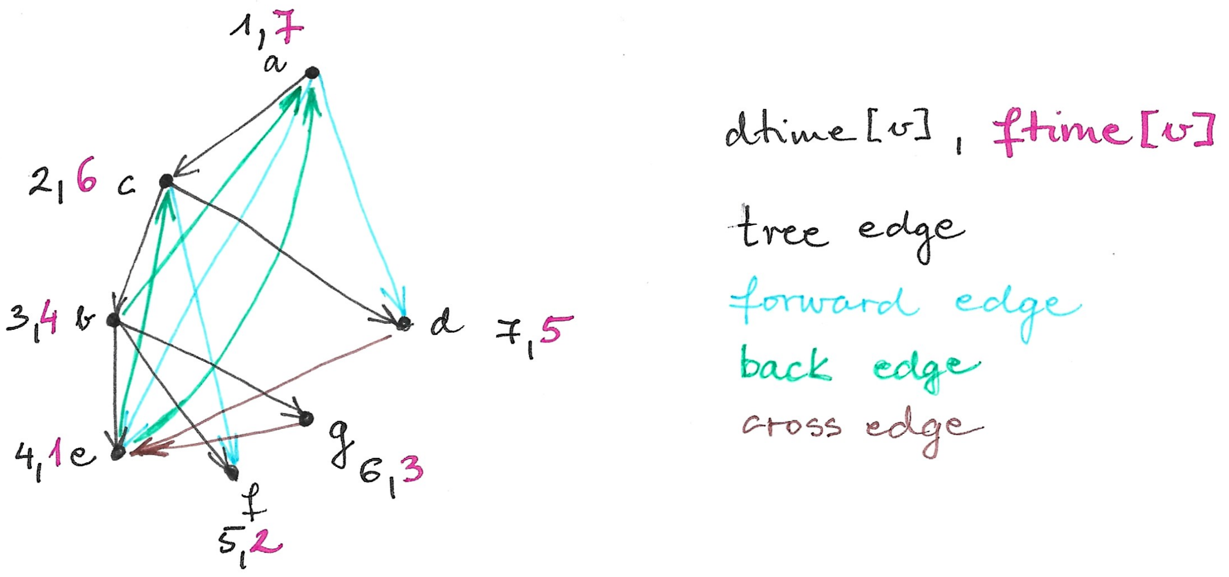
The out-degrees are $2, 2, 1, 1, 1, 0$. The in-degree of f is 2. This is the number of occurences of f in the edge list of the other vertices.

40. Run the DFS-MAIN(G) from vertex 1 in the directed graph below (given by the adjacency list) so that it follows the order given in the edge list when examining the neighbours. Also give the discovery and finishing numbers of the vertices.

$$\mathbf{a} : c, d, e; \quad \mathbf{b} : a, e, f, g; \quad \mathbf{c} : b, d, f; \quad \mathbf{d} : e; \quad \mathbf{e} : a, c; \quad \mathbf{f} : -; \quad \mathbf{g} : e$$

State which edges will be tree edges, forward-edges, back-edges and cross-edges.

Solution:



41. We are given an inverse adjacency list of a directed graph where for each vertex we have a list of the edges entering the vertex. Give an algorithm with running time $O(n+m)$ that produces the outgoing edge list for each vertex from the given edge lists.

Solution:

To construct the reverse edge list, we first copy the array of vertices and assign an empty list to each. This takes $O(n)$ steps so far. This now describes the graph in which the same points are present, but there are no edges yet. Then we go through the original edge list. When we find a new edge (x, y), we add the edge (y, x) to the reverse edge list, meaning we insert vertex x at the beginning of the list of vertex y. Such an insertion requires $O(1)$ steps (as we insert at the beginning of the linked list). Since there are originally m edges, this totals $O(m)$ steps, thus the total number of steps is $O(n+m)$.

42. In a directed graph on six vertices, the order in which vertices are encountered in a run of DFS is a, c, f, e, d, b, while the finishing numbers are a: 6; b: 5; c: 4; d: 3; e: 2; f: 1. Is it possible that there is an edge in the graph (a) from f to e? (b) from d to e?

Solution:

(a) Since dtime$[f] = 3 <$ dtime$[e] = 4$, it is certain that when we traverse f, we have not yet visited e. If there were an edge from f to e, then we could still move on from f (for example, to e), so f could not be the vertex finished first.
(b) This is possible. For example, if the edge list of the graph is $\boldsymbol{a}: c, b$; $\boldsymbol{b}: -$; $\boldsymbol{c}: f, e, d$; $\boldsymbol{d}: e$; $\boldsymbol{e}: -$; $\boldsymbol{f}: -$, then the DFS precisely provides the specified discovery and finishing numbers. Then the edge (d, e) will be a cross edge.

43. We are given a directed graph G with its adjacency list. We are further given an array C indexed by vertices which stores the color of each vertex as red or green. We are also given two red vertices s and t in the graph.
(a) Give an algorithm with running time $O(n+m)$ that decides whether there is a path from s to t that passes only through red vertices.
(b) Give an algorithm with running time $O(n(n+m))$ that decides whether there is a path from s to t that passes through at most one green vertex.

Solution:

(a) First, we transform the graph so that the green vertices are not reachable, then we run the DFS(G_1, s) depth-first search in the obtained new graph G_1 from vertex s. If we reach vertex t during this process, then

such a path exists; otherwise, it does not.

Running time: The transformation of the graph is done by traversing the edge list and deleting every edge where one of the endpoints is green. This can be done with a single traversal of the edge list, and deleting an edge from the adjacency list takes constant time, thus this totals in running time of $O(n+m)$. In the subsequent DFS, we work with a graph with n vertices and at most m edges, so the running time of this part is also $O(n+m)$.

Correctness of the algorithm: There is a path between s and t in the new graph if and only if there is a path consisting only of red vertices in the original graph. And the DFS correctly determines whether t is reachable from s in G_1.

(b) For every green vertex z in the graph, we do the following: we declare vertex z to be red and run the algorithm of question (a). This determines whether there is a path from s to t containing only vertices that were originally red, possibly including z as well. This procedure takes $O(n+m)$ steps for each green vertex, and we do it at most n times, so the total running time is $O(n(n+m))$.

44. Given a directed graph G with an adjacency matrix, we want to determine all vertices from which a given vertex t is reachable by a directed path. Give an algorithm with running time $O(n^2)$ for this task.

Solution:

If t is reachable from s via a directed path, then if we reverse the direction of every edge, s will be reachable via a directed path from t. We can easily obtain the adjacency matrix of this graph from the original adjacency matrix by taking its transpose. Run the DFS(G,t) procedure in the new graph. The running time of both the generation of the transpose and the DFS is $O(n^2)$, hence the total running tiem is also $O(n^2)$.
If the graph were given by an edge list, then using the solution of the 5th problem, it can similarly be seen that we can obtain the desired vertices in $O(n+m)$ steps.

---

45. A novice driver wants to choose the best route to drive to work. The road network is given as an undirected graph where the vertices are the intersections and the edges are the roads. Also, some of the vertices are labelled as difficult intersections to navigate through. (Difficulty of navigation through an intersection does not depend on where he arrives from or where he leave to from it.)

Give an algorithm with $O(n+m)$ running time, to determine the vertices he can drive to from his home at a given vertex such that his path will not have two consecutive vertices labelled as difficult. Assume that you are given the graph with an adjacency list.

Solution:

Let's eliminate those edges from the adjacency list that correspond to paths from difficult intersection to difficult intersection. This can be done in a single traversal of the adjacency list with linear running time in the size of the edge list, thus in $O(n+m)$ steps. Then, in the new graph G' obtained this way, run the DFS(G',v) algorithm, where v is the vertex that corresponds to the home of the driver. G' has $n'=n$ vertices and $m'\le m$ edges, so the DFS requires $O(n'+m')=O(n+m)$ steps. The vertices reached by the algorithm are precisely those reachable from v in G', which are exactly the vertices reachable from v in the original road network corresponding to original graph G without encountering two consecutive difficult intersections in succession.

46. Give an algorithm with running time $O(n+m)$ for Problem 7b.

Solution:

First, we run the algorithm of problem 7(a) to decide if there is a path consisting only of red vertices. If such a path exists, we are done.

Then, in the second part of the algorithm, we determine if there is a path containing exactly one green vertex by identifying those green vertices that are reachable from s and from which t is also reachable. To do this, we *first remove the edges outgoing from the green vertices, which can be done in $O(n+m)$ steps. In the resulting graph G_1, we run the DFS(G_1,s) algorithm and record the green vertices reachable from s in an array T_1 ($T_1 = 1$ if the vertex was reachable and green, otherwise $T_1 = 0$). The running time for this is $O(n + m')$, where m' is the number of edges in the new graph, but $m' \leq m$, the running time is $O(n + m)$. Since we removed the outgoing edges from the green vertices in graph G, the DFS precisely discovers those green vertices that are reachable through red vertices.*

In the next step, we reverse the direction of the edges in G and remove the edges outgoing from the green vertices in the new graph. Let the resulting graph be denoted as G_2. Now, by running the DFS(G_2, t) algorithm, we obtain those green vertices that are reachable from t through directed paths, and we record these in an array T_2. Due to the reversal of edges, there is a directed path to t in the original graph from every green vertex set to 1 in T_2, and due to the removal of outgoing edges, this path only passes through red vertices.

Next, we iterate through the arrays T_1 and T_2, and if we find a vertex v such that $T_1[v] = T_2[v] = 1$, then we can reach the green vertex v from s through directed path containing only red vertices, and then from there to t again passing through only red vertices. If there is no such intersection, then there is no directed path with exactly one green vertex. Since after modifying the graphs ($O(n + m)$) and running DFS twice (each $O(n + m)$), we only need to iterate through the two arrays, which takes $O(n)$ time, the overall running time is $O(m + n)$.

Solution 2: *Let's take another instance of the graph G, let's call this G'. Each vertex v in G will have its counterpart in G', denoted as v'.*

Then we delete each edge (v, w) outgoing from a green vertex in G, and instead, we add the edge (v, w') in G', then we remove all outgoing edges from green vertices in G'. The resulting graph is denoted as H. We run the DFS(H, s) algorithm. If it finds a directed path to t or t', then it will be a good path.

Running time: the number of vertices in H is $2n$, and the number of edges is at most $2m$, because first we duplicate every edge, then we move some, and finally we may delete edges from G'. Running this depth-first search takes $O(2n + 2m) = O(n + m)$ steps.

Correctness: If there is a directed path from s to t, then it must have traversed all the way long through the modified G, as there is no way to return from G' to G. Since there are no outgoing edges from green vertices within G, such a path cannot contain a green vertex. If there is a directed path from s to t', then it must have crossed from G to G', and in doing so, it must have passed through a green vertex. Since it's not possible to go back to G and there are no edges originating from green vertices in G', such a path contains exactly one green vertex. It's easy to see the reverse direction as well: if there is a good path in the original G graph, there will be a good path in H too.

Note: The basic idea of this method can be applied even when searching for paths containing at most $2, 3, \ldots$ green vertices. (In this case, multiple instances of the graph are needed.) Furthermore, with slight modifications, it can also be used to find paths containing exactly $1, 2, 3, \ldots$ green vertices. The running time always remains $O(n + m)$, provided that the prescribed number of green vertices is constant and independent of n.

47. Give an algorithm that either finds a cycle in an undirected graph given by its adjacency list, or shows that the graph is cycle-free. The running time must be $O(n)$ (independently from m, even though it could be much larger than n).

Solution:

The graph is undirected, therefore no forward edges or cross edges will be found during a DFS algorithm's execution.

Only directed graphs can have a cross edge during a DFS run, because it is pointing towards an already discovered and finished vertex. It is crucial for its existence that this edge is only traversable in one direction, otherwise we would have traversed it in the opposite direction when we first discovered its endpoint, creating a tree edge.

Similarly, a forward edge has an endpoint that was discovered later than its starting point. In the undirected case, this edge would have been traversable in the opposite direction too, creating a back edge instead.

So the DFS algorithm in the undirected case will only classify edges as a tree edge or a back edge.

Let's modify the depth-first search algorithm so that we stop if we encounter an already visited vertex along the next edge. If this does not occur, then the graph is cycle-free. If it does, then that edge is a back edge, and it forms a cycle along with the path between its endpoints in the spanning forest.

In any case, the traversal terminates after examining at most n edges, since we have a forest (which can have at most $n-k$ edges, where $1 \leq k$ is the number of components) and possibly one more edge. The breadth-first search can be similarly modified.

Note: In the general case, the complexity of DFS is $O(n+m)$, however with our modification, it will not traverse all edges, only at most n, therefore the runtime complexity becomes $O(n+n) = O(n)$.

48. We are given the adjacency list of a connected directed graph G whose edges have an integer value weight from the set $1, 2, \dots, k$. Let the value of a path be the *maximum* of the weights of the edges on the path. Given two vertices of the graph, x and y, give an algorithm with running time $O(m \log k)$ to determine the path from x to y with the smallest possible value.

Solution:

We will use an algorithm similar to binary search. First, we check if there is a directed path from x to y in the graph obtained by removing edges with weights greater than $\frac{k}{2}$. If there is a path in the new, modified graph from x to y, then we examine whether there can still be a path between x and y with smaller edge weights, so we repeat the process for $\frac{k}{4}$. If removing edges with weights greater than $\frac{k}{2}$ no longer allows for a path between x and y, then we increase the weights to search for. In this case, we delete edges with weights greater than $\frac{3k}{4}$ and continue the algorithm. This procedure surely finds the minimal path of weight l between x and y, because if there were a path of weight at most $(l-1)$, then in the graph obtained by removing edges with weights greater than $(l-1)$, y would be reachable from x, thus the algorithm would not return with the l weight path. The running time is $O(m \log k)$, since for each halving, we perform deletion of at most m edges and then run DFS in $O(n+m)$ steps, and we repeat this process $\log k$ times.

49. We are given the adjacency list of a directed graph G. Each vertex of the graph is given a label (number) from 1 to 100, a label can occur more than once. Find (if there exists) a *colorful* path in the graph in which each label $1 \leq i \leq 100$ occurs exactly once. The running time of the algorithm should be $O(m+n)$.

Solution:

First of all, let's notice that we need to find a path of exactly length 100, where the labels form a permutation of the numbers $1, 2, 3, \dots, 99, 100$. There are $100!$ such permutations, which is a constant.

For each possible permutation $i_1, i_2, \dots, i_{100}$, we do the following:

We transform the graph so that only directed edges remain that correspond to consecutive label pairs in the permutation. That is, an edge from x to y remains in the graph if and only if the label of x is just before the label of y in the permutation.

In the new graph G_1, we need to determine whether there is a path from any vertex labeled i_1 to any vertex labeled i_{100}. We do this by adding a new vertex s connected to every vertex labeled i_1, and then running DFS(G_2, s) in this graph G_2. If there is a vertex labeled i_{100} reached during this process, then the desired path exists; otherwise, it does not.

This procedure is correct because a vertex labeled i_{100} is reachable in the graph G_2 precisely when there exists a colorful *path with labels $i_1, i_2, \dots, i_{100}$ in the original graph G.*

The edge list of G_2 can be prepared in $O(n+m)$ time, as the resulting graph has $n+1$ vertices and at most $m+n$ edges. Therefore, the running time of DFS is $O(n+1+n+m)$, which simplifies to $O(n+m)$.

Since checking one permutation takes $O(n+m)$ time, and there are only a constant number of permutations, the overall running time of the entire process is $O(n+m)$.

50. We are given a graph that is a path on n vertices, where each vertex contains a positive integer. We want to select a subset of vertices that has no adjacent vertices and where the sum of the numbers is as large as possible. Give an algorithm with running time $O(n)$ for this task.

Solution:

This can be solved with dynamic programming. Let w[i] mark the positive integer contained in vertex i and let DP[i] mark the maximal possible such subset from vertices 1 *to* i. *To calculate DP[i] we can observe, that vertex i is either contained in the maximizing subset or not:*

- *If i is contained, then* $i-1$ *cannot be contained, the rest of the vertices that are not adjacent and maximize the sum will come from* 1 *to* $i-2$ *and the maximal possible value for such subset is calculated in DP[i-2].*
- *If i is not contained, then we are maximizing the possible value from vertices* 1 *to* $i-1$, *which is calculated in DP[i-1].*

Therefore

$$\mathrm{DP}[i] = \max(\mathrm{DP}[i-2] + \mathrm{w}[i], \mathrm{DP}[i-1]) \quad i = 2, \ldots n$$

And let $\mathrm{DP}[0] = 0$, *for the value of the empty set and* $\mathrm{DP}[1] = w[1]$, *for the value of the set containing only vertex* 1.

We calculate the values of the DP array from $i = 0$ *to* n, *in each step referencing at most the previous two cells. At the end of the calculation, the solution is contained in DP[n].*

When we are also looking for the elements in the maximizing subset, we can allocate an extra array, named is_ contained, where we set is_ contained[i] = true, if during the calculation of DP[i], vertex i was used. To decode the maximizing subset at the end, we start from i = n and depending on the value of is_ contained[i], we do the following:

- *When is_ contained[i] = true, we print the i and set i = i - 2.*
- *When is_ contained[i] = false, we don't print anything and set i = i - 1.*

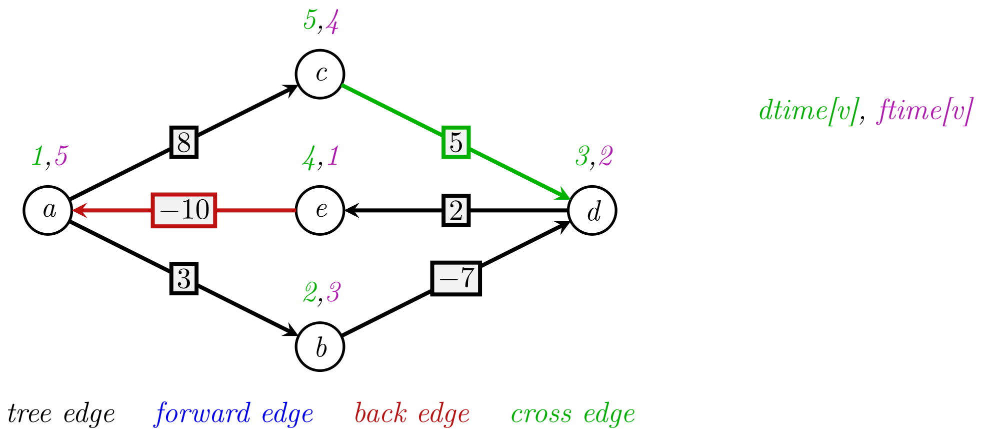
We repeat this until 0<i is true.

During the calculation of DP and is_ contained, we filled $2n$ *cells with a constant number of steps in each iteration and during the solution printing, we have done at most n steps when reading back from is_ contained. Therefore our algorithm is* $O(n)$.

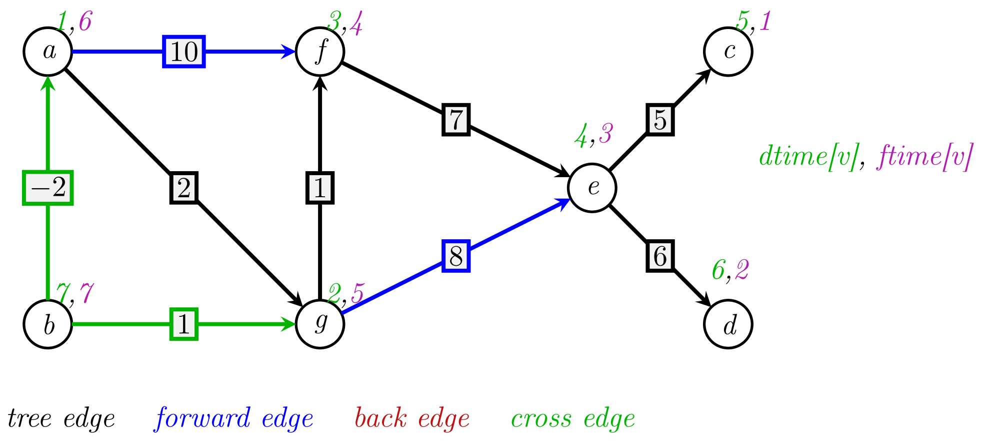
51. We are given the adjacency lists of two directed (edge-)weighted graphs G_1 and G_2.
G_1: **a**: b(3), c(8); **b**: d(-7); **c**: d(5); **d**: e(2); **e**: a(-10);
G_2: **a**: g(2), f(10); **b**: a(-2), g(1); **c**: -; **d**: -; **e**: c(5), d(6); **f**: e(7); **g**: f(1), e(8);
(a) Use the DFS algorithm to determine which of them is a DAG.
(b) For the graph that is acyclic, find a Topological ordering and use that to find the (c) shortest and (d) longest paths from a to all the other vertices.

Solution:

(a) G_1 *is not a DAG, because we have found a back edge during DFS.*

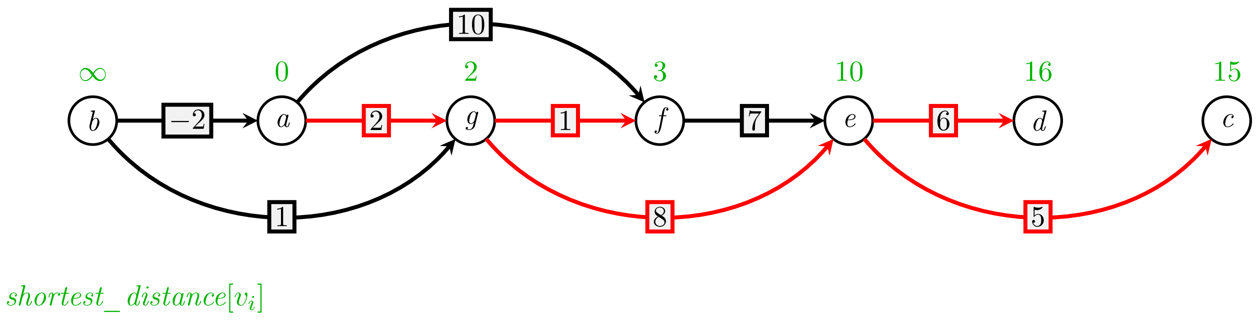


G_2 is a DAG, because there is no back edge.



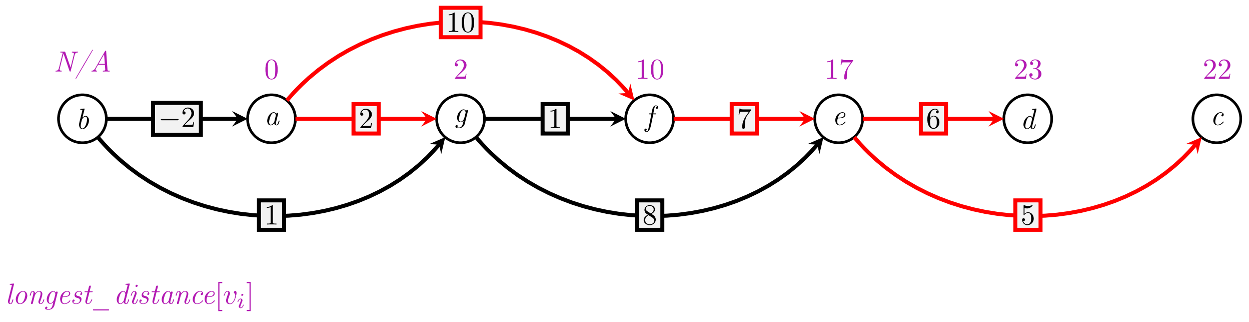
(b) Topological order: b, a, g, f, e, d, c, obtained by running DFS and listing the vertices in decreasing order of their finishing times.

(c) Shortest paths:



Drawn in red the edges we used to obtain the ***minimal*** *value at a specific vertex (e.g. through which we came). At vertex e, the minimal incoming edge could also be fe instead of ge.*

(d) Longest paths:



Drawn in red the edges we used to obtain the ***maximal*** *value at a specific vertex (e.g. through which we came). At vertex e, the maximal incoming edge could also be fe instead of ge.*

52. In a directed graph G given by an adjacency list, some vertices are red, some vertices are blue, and the remaining vertices are colorless. (The coloring is given in an array C indexed by vertices.) Which algorithm can be run once in $O(n+m)$ steps to solve the following problems and how?
(a) We want to decide which blue vertices are reachable from a given red vertex p_1 (via a directed path).
(b) Of all the blue vertices, we want to determine the ones nearest to a given red vertex p_1.

Solution:

(a) Let's run BFS or DFS from p_1 without restarting. Those blue vertices form the desired set, which become visited during this run. This is true because both algorithms traverse exactly those vertices (without restarting) from the starting vertex into which there is a directed path from the starting vertex in the graph. The procedure is $O(n+m)$ because the running time of both algorithms is this much, and then it's just a matter of going through the array of vertices checking which were visited and reading out the visited blue vertices, which takes $O(n)$ steps.

(b) Let's run BFS from p_1 without restarting. By the end of this, we'll have the distance of every vertex from p_1. Then, traversing through the array of distances, we search for the minimum among the values corresponding to the blue vertices, and then we output those blue vertices where this found minimum distance is taken.

This is correct because BFS correctly determines distances when we interpret distance as the number of edges in the path.

The procedure is $O(n+m)$ because the running time of BFS is this much, and afterwards, we only need to traverse through the array of distances once to find the minimum and then once again to print out the blue vertices with this distance. The latter two steps are both $O(n)$, hence the entire running time is $O(n+m)$.

---

53. Let G be an undirected connected graph. Is it true that
(a) For each edge f of G, it is possible to run the DFS on G from some vertex such that f will be a tree-edge.
(b) For each edge f of G, it is possible to run the BFS on G from some vertex such that f will be a tree-edge.
(c) For every spanning tree F of G, we can obtain F as a BFS-tree (that is, all edges of F are tree edges).
(d) For every spanning tree F of G, we can obtain F as a DFS-tree (that is, all edges of F are tree edges).

Solution:

(a and b) The answer is yes to both questions. Let's assume that ff connects vertices uu and vv. The first element of the vertex array of the adjacency list u, and v should be the first element in the list of neighbors of u. Both BFS and DFS will start from u, and they will discover v first.

(c) This is not true, for example, if $G = K_n$ is a complete graph with $n > 3$ vertices and F is a star (every edge is connected to the central vertex). In this case, regardless of the adjacency list, DFS can always move on, and the DFS tree will be a path, not a star.

(d) This is also not true, for example, if $G = K_n$ is again a complete graph, but now F is a path (with maximum degree 2). In this case, regardless of the adjacency list, BFS will immediately discover all other

vertices from the starting point, so the BFS tree will be a star, not a path.

54. We are given the adjacency list of a directed graph G with no directed cycles. Given a vertex s of the graph, we want to determine the number of paths from s to v for all vertices v in the graph. Give an algorithm with running time $O(n + m)$ for this task.

Solution:

Let's denote the graph as $G = (V, E)$. We create an array indexed by the vertices, which we denote as `num`. *Our goal is for $\texttt{num}[v]$ to represent the number of $s \to v$ paths. One possible way to achieve this is as follows:*

We determine a topological ordering of the graph. If vertex v precedes s in the topological ordering, then $\texttt{num}[v] = 0$, because according to the definition of a topological ordering, there cannot be an $s \to v$ path. Furthermore, $\texttt{num}[s] = 1$ (there is exactly one path from s to s, the empty path, since there are no directed cycles in the graph).

With this knowledge, for vertices v that follow s we progress forward in the topological ordering and we use the recursion

$$\texttt{num}[v] = \sum_{u \in V:\ (u,v) \in E} \texttt{num}[u]$$

to determine the number of $s \to v$ paths, where the $\texttt{num}[u]$ values are already known, since if there is a $u \to v$ edge in the graph then u preceeds v the topological ordering. The formula holds because if there is a $u \to v$ edge, then there are exactly as many paths from s to v through u as there are from s to u, thus summing over u gives us the number of paths from s to v (once again using the fact that edges can only go forward in the topological ordering).

The implementation and runnung time of the algorithm: we know that the topological ordering can be determined in $O(n + m)$ steps. To execute the recursion step, we need to know from which u vertices there is an edge to a given v. For this we create the reversed edge list from the original edge list, which takes $O(n + m)$ steps (see previous exercise set, Exercise 5). Then, for each vertex v, we need to look at the `num`*-values of at most as many u vertices as many incoming edges there are to v, and sum these values. Thus this, and the entire algorithm also, takes running time $O(n + m)$.*

Note*: The algorithm defined by the above recursion is essentially dynamic programming, as will be seen later.*

55. Continuation of exercise 2:
(c) We want to decide whether each blue vertex is reachable from at least one red vertex (i.e., there is a directed path to each blue vertex from some red vertex).
(d) We want to find the shortest path in the graph that leads from a red vertex to a blue vertex. (Note: the starting red vertex, and the ending blue vertex are not fixed).

Solution:

(c) Let's add a new vertex v to the graph G. From here, let's introduce (directed) edges to every red vertex in G. The resulting graph will be G_1. G_1 can be obtained from G in $O(n)$ steps. The number of vertices will be $n + 1$, and the number of edges will be at most $n + m$. Then, we start a traversal in G_1 from v. This can be either DFS or BFS. The set K of blue vertices reachable from v will contain precisely those blue vertices that are reachable from a red vertex in G. We mark the vertices in K, and finally, we check - which takes $O(n)$ work - whether K is the set of all blue vertices. The running time of the traversal is $O(1 + n + n + m) = O(n + m)$.

(d) Here, in the previously constructed graph G_1, we perform a BFS starting from v. For each vertex w, we write its distance from v in G_1, let this distance be d_w. This takes $O(n + m)$ work so far. Then, we determine the minimum of the quantities $d_w - 1$, where w goes over the set of blue vertices. This is an additional work of $O(n)$. Thus, the total running time is $O(n + m)$.

56. Circus acrobats stand on each other's shoulders and want to build a tower as big as possible (there will be only one acrobat on each level). For aesthetic and practical reasons, only an acrobat who is both lighter and smaller can stand on the shoulder of another acrobat. In the circus, there are n acrobats, and the height and

weight of each is given.
(a) Give an algorithm with running time $O(n^2)$, that gives the maximum number of people that can stand in a tower.
(b) Give an algorithm with running time $O(n^2)$, that returns a subset of the acrobats that constitute a tallest possible tower.

Solution:

Let's consider a graph G where the vertices represent acrobats, and between any two vertices v_i and v_j, there is a directed edge if acrobat j can stand on the shoulders of acrobat i. Additionally, let's introduce a vertex g (representing the ground) and draw edges from it to every other vertex (any acrobat can stand on the ground). Notice that G is a DAG (due to the conditions of standing on each other) and the paths starting from g correspond to towers.

We need to carefully assign weights to the edges so that the length of such paths corresponds to the height of the towers (as requested). Then both problems can be solved by running a longest path search from g in G.

a) Let every edge have a weight of 1. The tower on which acrobat i stands can be at most `longest`$[v_i]$ *people high (since there cannot exist a path of more than* `longest`$[v_i]-1$ *vertices (acrobats) starting from g and ending at v_i). Let's find the maximum* `longest`$[v_i]$*, which will correspond to the tallest tower. We can determine the tower itself by keeping track, during the longest path search, of the previous vertex on the path at each vertex, enabling us to trace back which acrobats are in the tower.*

b) Let the weight of edge $v_i \to v_j$ be the height of acrobat j, and similarly the weight of edge $f \to v_j$ be the height of acrobat j. In this case, the length of a path will exactly be the sum of the heights of the acrobats on the path. Therefore, the tower on which acrobat i stands at the top will be at most `longest`$[v_i]$ *high (since there cannot exist a longer (or higher) path from g to v_i). Let's find the maximum* `longest`$[v_i]$*, then determine the tower itself in the same way as in part (a).*

Providing the adjacency matrix (or list) of the graph takes running time $O(n2)$ because we have to examine every pair of acrobats. The longest path search takes $O(n^2)$ or $O(n+m)$ steps, depending on how the input is given, and finding the maximum is $O(n)$. Since the graph is simple, $m \in O(n^2)$, so the entire algorithms fit into $O(n^2)$.

---

57. We are given the adjacency list of a weighted graph, where the edges have weights $1, 2, 3$. Give an algorithm with running time $O(n+m)$ to determine the length of the shortest paths from a given vertex s to all other vertices in the graph.

Solution:

Split the edges of weight 2 into 2 edges (each adding a new point), and the 3 weighted edges into 3 edges. The resulting graph $G' = (V', E')$ satisfies $|V'| \le |V| + 2|E|$ and $|E'| \le 3|E|$. Instead of the shortest paths in G, it suffices to use in G' are the paths with the fewest edges. And this, starting from v, can be done by a latitude walk (BFS) from v. Number of steps $= O(|V'| + |E'|) = O(n+m)$.

58. Let us delete the sources ($d_{in}(v) = 0$) and sinks ($d_{out}(v) = 0$) from a directed graph G. Apply this operation repeatedly for the remaining graphs while there are sources and/or sinks in the graph obtained after the deletions. Prove that the final result is the empty graph if and only if G did not contain directed cycles.

Solution:

If G contains a directed cycle, then none of its vertices are sources or sinks, and this remains unchanged even if we remove the sources and sinks from the graph. Therefore, G cannot be emptied by repeating the removal of sources and sinks.

However, if G does not contain a directed cycle, let's take a topological ordering of G. The first vertex in the topological ordering is always a source, and the last one is always a sink. Therefore, we must remove these in the first step of the process (it's possible that these coincide only if the graph consists of a single isolated *vertex). If there are still vertices in the graph after removing the sources and sinks, then the first remaining vertex in the original topological ordering becomes a source because any vertex preceding it in the topological ordering has been removed, so no edges can go into it. Similarly, the last remaining vertex becomes a sink because no edges can go out from it. Therefore, we will also remove these in the next step (along with all the other vertices that became sources or sinks). Thus, in each step, we can remove at least two vertices, except if the graph has only a single (isolated) vertex, which can also be removed. Therefore, by repeating the procedure, the graph can be emptied in at most* $\lfloor n/2 \rfloor + 1$ *steps.*

59. We are given a black and white image of $n \times n$ pixels. We would like to draw boundary line from the upper left corner to the lower right corner of the image in such a way that the sum of the number of black pixels above and to the right of the line and the white pixels below and to the left of the line is as small as possible. Determine this boundary in running time $O(n^2)$.

Solution:

Let's construct a graph (specified by an adjacency list) where the vertices represent pixels (as rectangles) corners and the edges represent the borders of the pixels. The weight of vertical edges should be the sum of black pixels on its right and white pixels on its left. The weight of horizontal edges should be 0. Direct all vertical edges downwards, and horizontal edges to the right. With this definition of edge weights, it can be observed that the total weight of the boundary line moving from the top left corner to the bottom right corner, moving right-downwards, is exactly the number to be minimized, because we count each misplaced pixel exactly once.

Due to the direction of the edges, the graph is a Directed Acyclic Graph (DAG), and we are looking for the shortest path in it.

Number of vertices is $(n+1)^2 = O(n^2)$*, the number of edges is* $m = 2n(n+1) = O(n^2)$*.*

The weight of vertical edges can be determined in $O(n2)$ *steps by iterating row by row from left to right. In each row, we first count the number of black pixels (*$O(n)$*), which becomes the weight of the leftmost edge. The weight of the edge to the right of the ith pixel is less by 1 than the weight of the left edge if the pixel is black, and more by 1 if it is white. Thus, we obtain the weights in* $O(n)$ *steps for each row. In the graph, we can find the minimal weight path in* $O(n+m) = O(n^2)$ *steps. The path obtained exactly meets the requirements.*

60. One thousand (1000) people live in a village. Every day each inhabitant tells every news that she/he learned on the previous day to all of her/his acquaintances. It is known that the network of acquaintances is such, that sooner or alter everybody learns everything. Prove that there are 90 inhabitants such that if they all learn something together, then everybody will learn that within 10 days, the latest.

Solution:

Let's consider an arbitrary spanning tree of the (undirected) graph determined by the acquaintances. (This can be e.g. the BFS or DFS tree.) Such a tree exists because eventually everyone will know everything, so the graph is connected. Choose an arbitrary point v in the spanning tree. (For BFS or DFS, it is advisable to choose the starting point of the search.) In the spanning tree, there is exactly one path to every other point, and the number of edges of this path is the distance of the respective point from v in the tree. (In the case of BFS, this corresponds to the distance in the graph, but in the case of DFS, it is not guaranteed.)

Let V_i *be the set of points in the tree at a distance i from v, these sets are pairwise disjoint. If the people in set* V_i *learn a piece of news, then within 10 days, those in sets* $V_{i+1}, \ldots, V_{i+10}$ *will also learn it (and also those in sets* $V_{i-1}, \ldots, V_{i-10}$ *if such sets exist). For* $k = 0, 1, 2, \ldots, 10$ *define* $W_k = V_k \cup V_{k+11} \cup V_{k+22} \cup \ldots$*. Since the sets* V_i *are pairwise disjoint, the sets* W_k *are also pairwise disjoint. If everyone in set* W_k *learns the news on a given day, then in 10 days, everyone will know it. We know that* $\sum_{k=0}^{10} |W_k| = 1000$*. It cannot be that* $|W_k| \geq 91$ *for all k, because* $91 \cdot 11 = 1001 > 1000$*. Therefore, there must be some k such that* $|W_k| \leq 90$*. They need to be informed about the news.*

61. Give a longest increasing subsequence in the sequence $10, 3, 5, 2, 7, 1, 18, 4, 12, 17, 6$ by the dynamic programming algorithm studied in class. (A subsequence can contain any subset of the elements, it is not necessarily consecutive.)

Solution:

Let us store the original sequence in array A, with length n. We define two auxiliary arrays, L and previous, to help us track information about the increasing subsequences within A:

- *$L[i]$ represents the length of the longest increasing subsequence ending with the element $A[i]$. This subsequence can include any combination of elements from the start of the array up to $A[i-1]$ and it definitely includes the element $A[i]$ itself. (This will be important.)*
- *$previous[i]$ holds the index of the predecessor of $A[i]$ in the longest increasing subsequence ending with $A[i]$. Essentially, it points to the element that comes just before $A[i]$ in this subsequence, helping us trace the subsequence backward.*

We can fill these auxiliary array from left to right. In each step, at index i we look at all of the previous elements in A and select the ones that are smaller than $A[i]$: These are the subsequences A[i] can be appended to as a last element.

$L[i] = \max\{L[j] + 1 \mid A[j] < A[i],\ 0 \le j \le i - 1\}$

$previous[i]$ corresponds to the j index in the equation above that maximized the $L[i]$ value.

It was important to define $L[j]$ in such a way that we definitely know the last element of the subsequence it corresponds to, so we can check for $A[j] < A[i]$ to make sure the subsequence is increasing.

For simplicity, we added an $L[0]$, denoting the length of the empty subsequence.

$i =$	0	1	2	3	4	5	6	7	8	9	10	11
$A[i] =$	-	10	3	5	2	7	1	18	4	12	17	6
$L[i] =$	0	1	1	2	1	3	1	4	2	4	5	3
$previous[j] =$	-	-	-	2	-	3	-	5	2	5	9	3

The length of the longest increasing subsequence is the maximum in L, $\max\{0, 1, 1, 2, 1, 3, 1, 4, 2, 4, 5, 3\} = 5$. The maximum is at index $i = 10$, the exact positions can be found from the previous array recursively: $previous[10] = 9$, $previous[9] = 5$, $previous[5] = 3$, $previous[3] = 2$ and $previous[2] =$ -, so 2 is the first index and a longest increasing subsequence can be formed with elements $A[2], A[3], A[5], A[9], A[10]$, e.g. the numbers $3, 5, 7, 12, 17$.

62. A ladder has n steps. Unfortunately, some of these steps are so rickety that if we step on them, they fall off. Fortunately, we know which steps are rickety and we will not step on these. Assuming that we can in one move cover upto 3 steps on the ladder. Give a dynamic programming algorithm that determines
(a) if we can go from the bottom of the ladder (the floor) to the top step of the ladder.
(b) the number of ways in which we can get from the bottom of the ladder to the top step.
(Assume that the top step is not rickety.) What is the running time of the algorithm?

Solution:

Let NR be the array where $NR[i]$ is true if the i-th step is not rickety.

(a) The subproblems will be whether we can reach the i-th step ($1 \le i \le f$). We fill out the table $F[i]$ in a way that $F[i]$ is true if we can reach the i-th step.

Note that the problem asks for the value of $F[f]$.

Initialization: $F[0] = true$, since the ground is safe; $F[1] = NR[1]$.

The recursion: since we can come to the i-th step from one, two, or three steps below, we can step onto the i-th step exactly when we can reach one of the three preceding steps and the i-th step is not rickety. Therefore, the general recursion is: $F[i] = NR[i] \wedge (F[i-1] \vee F[i-2] \vee F[i-3])$.

It is evident that this only works for $i \geq 3$ cases. For $i = 2$, a similar formula works, except there is no $F[i-3]$, thus: $F[2] = NR[2] \wedge (F[1] \vee F[0])$.

If we fill out the table in the order of $i = 0, 1, 2, \ldots, f$, then at each element, we use a constant number of steps, and the running time of the algorithm is $O(n)$.

(b) Similar to part (a), but now we need a numeric variable. Let $M[i]$ be the number of ways we can reach the i-th step.

$M[f]$ provides the solution.

Initialization: let every $M[i] = 0$ if $i > 0$ and $M[0] = 1$. Since stepping to i can only happen from one of the previous three steps, the number of possible ascents is obtained as the sum of the three preceding numbers. We consider rickety steps by setting the value of M to 0 for those steps, meaning they contribute nothing to the desired count. Thus:

If $NR[1]$ is true, then let $M[1] = 1$, if $NR[2]$ is true, then $M[2] = M[1] + M[0]$ and in general, if $NR[i]$ is true, then let $M[i] = M[i-1] + M[i-2] + M[i-3]$.

This time we need to fill out the table also in the order of $i = 0, 1, 2, \ldots, f$, and then the values we want to use are available to us. For each i, there will be a constant number of additions, which means a total of $O(n)$ additions. For simplicity, we also consider this as the running time.

If we also consider that the numbers to be added are not necessarily of constant size, then the number of steps will be higher. However, it can be easily proven by induction that the largest number is at most 3^n. To describe this, $O(n)$ digits are sufficient. Since addition can be performed with an algorithm that is linear in the number of digits, the total running time of the algorithm is $O(n^2)$.

---

63. We are given $n+1$ positive integers ($n \geq 1$): $s_1, s_2, \ldots, s_n, b$, we want to decide whether the number b is the sum of some subset of the numbers s_i. (Each s_i can be used at most once in the sum.)
Give a dynamic programming algorithm with running time $O(n \cdot b)$ for this problem. (Note: this is called the subset-sum problem). Use the following subproblems:
Let $L(i, c)$ (for $0 \leq i \leq n$ and $0 \leq c \leq b$) denote whether it is possible to express the number c as a sum of some subset of $s_1, \ldots, s_i$.

Solution:

1. *Subproblems: Given in the problem, $L(i, c) = 0$ if c cannot be obtained and 1 if it can be obtained as a sum of some numbers from $s_1, s_2, \ldots, s_i$. (Essentially, we fill out a table of size $(n+1) \times (b+1)$, where $L(i, c)$ stands in the c-th cell of the i-th row.)*

2. *Order: i goes from 0 to n, and within an inner loop, for each i, c goes from 0 to b (we progress row by row in the table from top to bottom, and within each row, from left to right).*

3. *Start: $L(i, 0) = 1$ for every i, because we can always obtain 0 without selecting any number, and $L(0, c) = 0$ if $c > 0$, because if we cannot select any number, then no sum greater than 0 can be obtained. (This means the first column of the table is all 1s, and the rest of the elements in the first row are all 0s.)*

4. *Continue: if $i \geq 1$ and $c \geq 1$: $L(i, c) = 1$ if and only if $L(i-1, c) = 1$ or if $L(i-1, c-s_i) = 1$ (the latter case only applies if $c - s_i \geq 0$). This is correct because c can be obtained either without s_i, just with some of $s_1, \ldots, s_{i-1}$, or using s_i, in which case we need to obtain $c - s_i$ with some of $s_1, \ldots, s_{i-1}$.*

5. *End: we need $L(n, b)$, which determines whether we can obtain b with some of the numbers s_i. (In other words, we want the bottom-right element of the table.)*
6. *Correctness: The formula is correct, because at each step we either use s_i as part of the sum and we must formulate the $c - s_i$ sum from $s_1, \dots, s_{i-1}$ or we don't use s_i and must formulate the entire c sum from $s_1, \dots, s_{i-1}$. In the special cases, where the required sum is 0, that is always considered possible (just don't select any numbers) and when the required sum is not 0, but $i = 0$, we cannot use any numbers, so it is impossible to do.*
7. *Running time: At each subproblem, there are a constant number of steps, and there are $(n + 1) \cdot (b + 1)$, i.e., $O(n \cdot b)$ subproblems, so the running time is $O(n \cdot b)$.*
8. *We need to determine the numbers with which the sum is obtained: it is necessary to determine which s_i's are used in the sum that gives b, if it can be obtained at all. Here, it is sufficient to provide a good subset of the numbers s_i, and this can be done by keeping track of whether c was obtained due to the first (without s_i) or the second (with s_i) case when calculating $L(i, c)$ (if c can be obtained from both cases, we pick one at random). Then, if $L(n, b) = 1$, by traversing backwards, we can determine for each s_i whether it is needed to obtain b or not. At each step, the value of i decreases by one, so this backward traversal (since we are only seeking a good subset, i.e., we only want to find one path) takes $O(n)$ steps.*

64. Consider a chessboard of size $m \times m$. Unfortunately, some holes have been punched into some squares. We want to travel from the bottom left corner to the top right corner, such that in one move we can go from a square to an adjacent square above it or to its right. Provide a procedure using dynamic programming that computes the number of ways to reach the top right corner.

Solution:

Let's define a table T with $m + 1$ rows and columns, where $T[i, j]$ represents the number of ways the cell (i, j) can be reached. (The bottom-left corner is the cell $(1, 1)$, and the top-right corner is the cell (m, m).)

Let the values in the 0-th row and column of T be zeros, and $T[1, 1] = 1$.

For the recursive formula, consider that we can only move to the cell (i, j) from either the cells $(i - 1, j)$ or $(i, j - 1)$. If we set the values corresponding to the forbidden cells (the cells with holes) in T to 0, then the formula

$$T[i,j] = \begin{cases} T[i-1,j] + T[i,j-1], & \text{if } (i,j) \text{ is not forbidden} \\ 0, & \text{if } (i,j) \text{ is forbidden} \end{cases}$$

is correct for $i \geq 1$, $j \geq 1$.

This table can be filled out row by row, as when reaching (i, j), the two values to be used are already available.

Running time: The size of the table is $(m + 1)^2 \in O(n)$, as the input size is $n = m^2$. Constant number of additions are required for each element, so the number of additions is also $O(n)$.

If we also need to consider the size of the numbers, if there are no forbidden cells, then the largest number is $\binom{2m}{m}$. Since $\sum_{i=1}^{2m} \binom{2m}{i} = 2^{2m}$, we have $\binom{2m}{m} \leq 2^{2m}$. The number of digits of this number is $O(m)$. In this case, the running time is $O(m^3)$, or $O(n^{1.5})$.

Note: In the future, for simplicity, we assume that arithmetic operations can be performed in one step regardless of the size of the numbers.

65. Let $s_1 s_2 \dots s_n$ and $t_1 t_2 \dots t_m$ be sequences of characters of lengths n and m respectively. We want that in a matrix A of size $n \times m$, the entry $A[i, j]$ contains the largest number k for which $s_1 s_2 \dots s_i$ and $t_1 t_2 \dots t_j$ end in the same k characters. (If $s_i \neq t_j$ then this would be 0). Give a procedure that fills the array A in $O(nm)$ steps.

Solution:

If $i = 1$ or $j = 1$, then it's simple, since one of the sequences has only 1 element, the value can only be 0 or 1.

$$A[1, j] = \begin{cases} 1, & \text{if } s_1 = t_j \\ 0, & \text{if } s_1 \neq t_j \end{cases} \qquad A[i, 1] = \begin{cases} 1, & \text{if } s_i = t_1 \\ 0, & \text{if } s_i \neq t_1 \end{cases}$$

For the further steps, it should be noticed that if s_i and t_j are different, then obviously $k = 0$; otherwise, it is the extension of the longest common substring obtained for the prefix segments that are shorter by one.

$$A[i, j] = \begin{cases} 1 + A[i-1, j-1], & \text{if } s_i = t_j \\ 0, & \text{if } s_i \neq t_j \end{cases}$$

If we fill the table row by row, we always have the necessary information available. Calculating one element is constant time, and the size of the table is nm, so the overall running time is $O(nm)$.

66. Each element of a table of size $n \times n$ is a positive integer. We want to go from the bottom left corner of the table to the top right corner by taking one step up or one step to the right in the table. We further want that the elements on our path should be in increasing order. Give an algorithm with running time $O(n^2)$ that determines,
(a) how many paths satisfy the rules.
(b) what is the largest value of a path satisfying the rules if the value of a path is the product of the numbers in it.

Solution:

(a) We use dynamic programming: Let A denote the original (given) table. We create an $(n \times n)$ table T, where $T[i, j]$ represents the number of ways we can reach the j-th element of the i-th row of A according to the rules. If $A[1, 1]$ is the bottom-left field and $A[n, n]$ is the top-right field, then $T[n, n]$ provides the desired answer.

Let $T[1, 1] = 1$. Since we can only come from the left or from below to the (i, j) position, there is a valid path if we can reach at least one of these positions, and from there, the last step is valid, i.e. $A[i, j]$ is greater than the preceding element.

Therefore, for the first row if $j > 1$, we have:

$T[1, j] = T[1, j-1]$, if $A[1, j] > A[1, j-1]$, otherwise $T[1, j] = 0$. (The row will be 1 for a while, then eventually becomes 0 if there was a non-increasing step.)

Similarly, for $i > 1$, we have:

$T[i, 1] = T[i-1, 1]$, if $A[i, 1] > A[i-1, 1]$, otherwise $T[i, 1] = 0$.

The general recursion for $i, j > 1$:

- *if $A[i, j] > A[i-1, j]$ and $A[i, j] > A[i, j-1]$, then $T[i, j] = T[i-1, j] + T[i, j-1]$*
- *if $A[i, j] > A[i-1, j]$ and $A[i, j] \leq A[i, j-1]$, then $T[i, j] = T[i-1, j]$*
- *if $A[i, j] \leq A[i-1, j]$ and $A[i, j] > A[i, j-1]$, then $T[i, j] = T[i, j-1]$*
- *if $A[i, j] \leq A[i-1, j]$ and $A[i, j] \leq A[i, j-1]$, then $T[i, j] = 0$.*

After determining the first row and column, if we fill the table row by row, we always have the necessary information available. Thus, calculating each element will require a constant number of operations.

Since the size of the T array is n^2 and filling each element requires a constant number of operations, the running time is $O(n^2)$.

(b) Similar to the previous point, but this time we are storing the maximum value. Therefore, at each valid step, we need to multiply by the current A value, and instead of addition, we choose the maximum value.

Denote this array by S, then:

$S[1,1] = A[1,1]$, since this is the value so far.

Filling the first row and column goes as follows: $S[1,j] = S[1,j-1] \cdot A[1,j]$, if $A[1,j] > A[1,j-1]$, otherwise $S[1,j] = 0$.

Similarly, for $i > 1$:

$S[i,1] = S[i-1,1] \cdot A[i,1]$, if $A[i,1] > A[i-1,1]$, otherwise $S[i,1] = 0$.

In the following,$i, j > 1$, and then:

- *if $A[i,j] > A[i-1,j]$ and $A[i,j] > A[i,j-1]$, then $S[i,j] = A[i,j] \cdot \max\{S[i-1,j], S[i,j-1]\}$*
- *if $A[i,j] > A[i-1,j]$ and $A[i,j] \leq A[i,j-1]$, then $S[i,j] = A[i,j] \cdot S[i-1,j]$*
- *if $A[i,j] \leq A[i-1,j]$ and $A[i,j] > A[i,j-1]$, then $S[i,j] = A[i,j] \cdot S[i,j-1]$*
- *if $A[i,j] \leq A[i-1,j]$ and $A[i,j] \leq A[i,j-1]$, then $S[i,j] = 0$.*

After determining the first row and column, if we fill the table row by row, we always have the necessary information available, so calculating each element will require a constant number of operations. Therefore, the total running time remains $O(n^2)$.

---

67. Given n files, where the i^{th} file's positive integer length is h_i, and two hard drives each with a positive integer capacity of L, our goal is to find the largest number k, such that files with indices between 1 and k can be stored across the two drives. A single file cannot be split between multiple drives, it must be stored in a single drive in one continuous space segment. Give an algorithm that finds the maximal possible k and outputs for each of the first k files which drive it should be saved to. The running time of the algorithm should be $O(L^2)$.

Solution:

Ideas:

The solution will depend heavily on the fact that the files come in a predetermined order and we are only interested in prefixes of this, not any possible subsets. Furthermore, since we are allowed an $O(L^2)$ time complexity, we are able to construct a two dimensional DP table to solve this task.

First solution (inefficient):

Our first solution will use a three dimensional DP table, which will be inefficient. Then, we will optimize this solution with an extra observation to use a two dimensional DP table instead.

Key observation: Let's first focus on the 1st file. We have three possible choices to make with it:

- *We store it on the 1st drive. (If there is enough space left.)*
- *We store it on the 2nd drive. (If there is enough space left.)*
- *We do not store it.*

What is the remaining task that needs to be solved in each of these cases?

- *When we choose storing the file on the 1st drive, the remaining task consists of drives with size $L - h_1$ and L and the remaining files with indices 2 to n.*
- *Storing it on the 2nd drive, the remaining task consists of drives with size L and $L - h_2$ and the remaining files with indices 2 to n.*
- *If we choose not to store this file, we are not allowed to store any later files either.*

The same idea works in the general case. For file i, if we assume that the files with indices 1 to $i-1$ have already been placed and we know the space remaining on the drives, we can make a similar choice whether we want to store the file - and which disk we want to store it on - or whether we should stop.

We can see now that the subtasks that need to be solved are parameterised by the space we used up so far on the two drives and the index of the last file thas was placed.

Let us define $DP[a][b][i]$ to be true if we are able to store the first 1 to i files on the two disks using up a size a space on the first disk and a size b space on the second disk.

Then, to calculate the value of $DP[a][b][i]$, the question is where should we store file i? We either store it on the first or on the second disk. If we store it on the first disk, we will use up h_i space. We know that in total we used up exactly a space from the first disk, so before storing file i, the space used on the disk must have been $a - h_i$. Therefore the remaining subtask to be solved comes from the cell $DP[a-h_i][b][i-1]$. With similar logic, when we store the file on the second disk, the subtask to be solved comes from $DP[a][b-h_i][i-1]$.

Therefore

$$DP[a][b][i] = DP[a-h_i][b][i-1] \vee DP[a][b-h_i][i-1]$$

and the base case is $DP[0][0][0] = true$, since storing no files uses up no space, and $DP[i][j][0] = false$ for all other i and j indices.

Let's also assume, that when DP is indexed by a negative number in any of the 3 dimensions, it returns a $false$, so we don't have to list those as special cases.

We could fill out this table in a tripe nested for loop, however that would take us $O(n \cdot L^2)$ time complexity, which is too much.

Idea for optimization:

However, we can notice that there is redundancy in the three parameters. The total space used on all of the drives will always equal to the total size of the files stored so far. E.g.

$$a + b = \sum_{j=1}^{i} h_j$$

Second solution (optimal, by dropping a parameter):

This means, that one of the parameters can be skipped and calculated from the other two. For example, we can skip parameter b and calculate it with the formula

$$b = \sum_{j=1}^{i} h_j - a$$

So let us drop that dimension and let $DP[a][i]$ mean whether it is possible to store the first i files on the two disks, while using up exactly a space on the first and $\sum_{j=1}^{i} h_j - a$ space on the second.

Now, is the following formula correct?

$$DP[a][i]? = DP[a - h_i][i - 1] \lor DP[a][i - 1]$$

It is not correct, because dropping the dimension b, we are at risk of running out of space on that drive, without noticing it. The drive size $\sum_{j=1}^{i} h_j - a$ must never exceed L and this extra condition must always be checked for both cases.

$$DP[a][i] = (DP[a - h_i][i - 1] \land \sum_{j=1}^{i} h_j - (a - h_i) \leq L) \lor (DP[a][i - 1] \land \sum_{j=1}^{i} h_j - a \leq L)$$

A tiny problem we still have is that calculating the sum of the file sizes from 1 to i takes $O(n)$, we should just precompute that and store it in an array $totalsize[i] = \sum\limits_{j=1}^{i} h_j$.

$$DP[a][i] = (DP[a - h_i][i - 1] \land totalsize[i] - (a - h_i) \leq L) \lor (DP[a][i - 1] \land totalsize[i] - a \leq L)$$

Third solution (by simply skipping the impossible DP cells):

Another way of dealing with a very large, but mostly empty and unused DP table is by doing something called "forward recursion" instead of the typical "backward recursion" we used so far.

Consider the three dimensional $DP[a][b][i]$ table as a graph. The cells are the vertices and the directed edges of the graph point from the two cells that are used in the recursive formula for calculating $DP[a][b][i]$, $DP[a - h_i][b][i - 1]$ and $DP[a][b - h_i][i - 1]$ to the cell $DP[a][b][i]$.

Notice how each $DP[a][b][i]$ cell only has two outgoing edges as well, depending on where we choose to store the file $i + 1$!

We can now change the direction of the recursion by starting from the only true base case, $DP[0][0][0]$ and "move in a forward direction", by placing a true value in cells $DP[h_1][0][1]$ and $DP[0][h_1][1]$, then we recursively continue with each additional file.

Since the file sizes are integer, for a specific i value, at most $L + 1$ number of true values exist, as $0 \leq a \leq L$ and i together determine the value of b too.

So there are only $O(L)$ true values in the DP sub-array $DP[..][..][i]$. The outgoing edges point towards true values in $DP[..][..][i + 1]$. Thus, between each layer, there are $2L$ edges and our solution iterates each edge exactly once. (We make sure not to forward-recurse from a cell that was already done in a previous step.)

Finally, we keep track of the largest i we have seen during the recursion, and stop when we cannot move forward.

(For implementation, a sparse matrix or similar implementation must be used, to only store the non-false values of the table.)

Fourth solution (recognizing how we have already seen a similar task to the third solution):

Notice how the DP graph in the previous solution is a DAG (edges only point towards higher i values) and we are looking for the largest possible i value, which corresponds to the longest path in the DAG.

Therefore, our longest path finding algorithm can be used here, which runs in $O(|V(G)| + |E(G)|)$, where $|V(G)| \leq 2Ln \leq 2n^2$ and $|E(G)| \leq 2L^2 \leq 2n^2$.

68. We are given two strings of length n and m respectively. We want to find the largest matching substring, i.e. if one text is $a_1a_2\cdots a_n$ and the other is $b_1b_2\cdots b_m$, then we want to find the maximum number t such that there are indices $1 \leq i \leq n$ and $1 \leq j \leq m$, such that, $a_ia_{i+1}a_{i+2}\dots a_{i+t-1} = b_jb_{j+1}b_{j+2}\dots b_{j+t-1}$ is satisfied. Give an algorithm for this problem using $O(nm)$ steps.

Solution:

Notice that the task is similar to Exercise 5. The maximum value in the A matrix created there is required t. If the position of this maximum value is cell $A[x][y]$, then substrings $a[x-t+1], \dots, a[x]$ and $b[y-t+1], \dots, b[y]$ are the corresponding largest matching substrings. Runtime complexity is also $O(nm)$, similarly to the previous task, with the addition of finding the maximum value at the end.

69. We are given two $0-1$ sequences $a_1, a_2, \dots, a_n$, and $b_1, b_2, \dots, b_m$ of lengths n and m respectively. We will now fill a matrix T in the following manner:

If $0 \leq i \leq n$, then $T[i, 0] = 0$. If $0 \leq j \leq m$, then $T[0, j] = 0$.

If $1 \leq i \leq n$ and $1 \leq j \leq m$, then $T[i,j] = \begin{cases} T[i-1, j-1] + 1 & \text{if } a_i = b_j \\ \max\{T[i, j-1], T[i-1, j]\} & \text{if } a_i \neq b_j \end{cases}$

What is the meaning of $T[i,j]$? What property of the two series is given by $T[n, m]$?

Solution:

$T[i,j]$ = length of the longest common subsequence in the sequences $a_1\cdots a_i$ and $b_1\cdots b_j$, thus $T[n,m]$ is the length of the longest common subsequence in the two complete sequences.

The recursive formula shows that in the longest common subsequence, either $a_i = b_j$, and then the longest one is obtained by extending the common subsequence found in the two sequences $a_1\cdots a_{i-1}$ and $b_1\cdots b_{j-1}$ with a_i and b_j respectively, hence the length is $T[i-1, j-1] + 1$. If $a_i \neq b_j$, then either b_j or a_i is not in the longest common subsequence, which is indicated by the second possibility.

Note: For such tasks, it is advisable to examine what happens with a simple example, based on which one can guess the rule (which of course needs to be proven).

70. (a) Give a dynamic programming algorithm that, given an integer n as input, computes F_n, the n^{th} Fibonacci number in $O(n)$ steps. The sequence of Fibonacci numbers is defined by $F_0 = F_1 = 1$ and the recursion $F_n = F_{n-1} + F_{n-2}$ if $n \geq 2$.
(b) Consider the following recursive algorithm to compute the n^{th} Fibonacci number F_n. For $n = 0$ and $n = 1$ it returns 1, while for $n \geq 2$ it call itself recursively with inputs $n-1$ and $n-2$, resp., and returns the sum of the two values obtained. Show that the running time of this algorithm is $\Omega(2^{\frac{n}{2}})$.

Solution:

(a) We fill out the table of size $1 \times (n+1)$, i.e. an array $F[0..n]$ in a way that $F[i]$ is the i^{th} Fibonacci number.

Note that the problem asks for the value of $F[n]$.

Initialization: $F[0] = F[1] = 1$.

Then starting from 2 we use the recursion formula of the Fibonacci numbers: $F[i] = F[i-1] + F[i-2]$. If we fill out the table in the order of $i = 0, 1, 2, \dots, n$, then by the time we need the value of $F[i-1]$ and $F[i-2]$ for $F[i]$, we have already determined them.

At each element, we use a constant number of steps, and the running time of the algorithm is $O(n)$.

(b) Let's denote by $L(n)$ the running time of the algorithm if the input size is n. Then $L(0) = L(1) = 1$, and

$L(n) \geq L(n-1) + L(n-2)$, if $n \geq 2$, thus for $k \geq 2$

$$L(2k) \geq L(2k-1) + L(2k-2) \geq L(2k-2) + L(2k-3) + L(2k-2) \geq 2L(2k-2),$$

and the same is true also for $k = 1$, since then $L(2k) = L(2) \geq L(1) + L(0) = 1 + 1 = 2L(0) = 2L(2k-2)$. Therefoer if $n \in \mathbb{N}$ is even, then

$$L(n) \geq 2L(n-2) \geq \ldots \geq 2^{n/2}L(0) = 2^{n/2},$$

while if $n \geq 3$ is odd then

$$L(n) \geq 2L(n-2) \geq \ldots \geq 2^{(n-1)/2}L(1) = \frac{1}{\sqrt{2}}2^{n/2}.$$

Thus $L(n) \geq \frac{1}{\sqrt{2}}2^{n/2}$, if $n \geq 0$ (and this is true also for $n = 0$ and $n = 1$: $L(0) = 1 \geq \frac{1}{\sqrt{2}} \cdot 1$ and $L(1) = 1 \geq \frac{1}{\sqrt{2}} \cdot 2^{1/2} = 1$), So, for example, choosing $c = 1/\sqrt{2}$ and $n_0 = 1$ shows that $L(n) = \Omega(2^{n/2})$ (moreover $n_0 = 0$ is good as well).

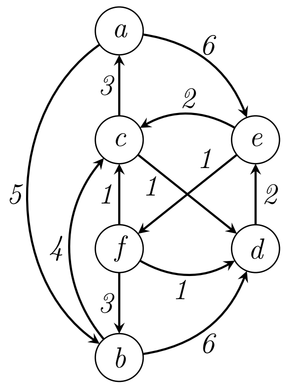
2. solution: We can prove also using induction. Choose such a $c > 0$ value, that $L(n) \geq c2^{n/2}$ holds for $n = 0, 1$. Since $L(0) = L(1) \geq 1$, for example $c = 1/2$ is good. Assume that $L(i) \geq c2^{i/2}$ for all $0 \leq i \leq n$. We prove that it also holds for $n + 1$.

$$L(n+1) \geq L(n) + L(n-1) \geq c2^{n/2} + c2^{(n-1)/2} \geq c2^{(n-1)/2}(2^{1/2} + 1) \geq c2^{(n-1)/2} \cdot 2 = c2^{(n+1)/2}.$$

Thus for example the choices $c = 1/2$ and $n_0 = 0$ show that $L(n) = \Omega(2^{n/2})$.

71. We are given the following adjacency list of a directed graph:
$a : b(5), e(6)$, $b : c(4), d(6)$, $c : a(3), d(1)$, $d : e(2)$, $e : c(2), f(1)$, $f : b(3), c(1), d(1)$.
Using Dijkstra's algorithm, determine the lengths of the shortest paths from a to all other vertices. (No need to justify in words, but one should be able to see the steps, how the safe set S updates. Please also show the table for final distances and the parents.)

Solution:



D	a	b	c	d	e	f	*SAFE*
0.	0	∞	∞	∞	∞	∞	$\{\}$
1.	0	5	∞	∞	6	∞	$\{a\}$
2.	0	5	9	11	6	∞	$\{a, b\}$
3.	0	5	8	11	6	7	$\{a, b, e\}$
4.	0	5	8	8	6	7	$\{a, b, e, f\}$
5.	0	5	8	8	6	7	$\{a, b, e, f, c\}$
6.	0	5	8	8	6	7	$\{a, b, e, f, c, d\}$

P	a	b	c	d	e	f
0.	–	–	–	–	–	–
1.	–	a	–	–	a	–
2.	–	a	b	b	a	–
3.	–	a	e	b	a	e
4.	–	a	e	f	a	e
5.	–	a	e	f	a	e
6.	–	a	e	f	a	e

Dijkstra's algorithm can find the shortest path from a given vertex to all others in a graph, with the condition that the graph ***cannot contain a negative weight edge****. In many real life scenarios, this is true (think of cities and their distances via car). The runtime complexity of the algorithm is $O(n \log n + m) \subseteq O(n^2)$, where n is the number of vertices and m is the number of edges in the graph.*

If the graph contains a negative edge, the algorithm can lead to incorrect results, see Exercise 5 as an example, so we cannot use it. In such cases, other algorithms, such as the Bellman-Ford algorithm can be used, which works correctly, even if there are some negative edges in the graph, as long as there are no negative cycles. But, this algorithm is slower, with time complexity $O(n \cdot m) \subseteq O(n^3)$.

Let's begin Dijkstra! D is the distance vector, P is the parent vector. These are both 1 dimensional vectors, and above we have illustrated how their values change during the execution of the algorithm, each row corresponding to the state of the vectors after one step of the algorithm.

The algorithm works by keeping track of the currently known shortest distances to each vertex from the starting one, vertex a. Initially, we set the distance of vertex a (to itself) to 0 and all others to ∞ (infinity), marking the fact that no path is known at the moment. This value is chosen, because it is greater than any other value, so any path will be shorter than it and we do not have to treat it as a special case.

In each step, we select the vertex with the smallest D value, that is not yet finalized and finalize it, by putting it into the $SAFE$ set. The D values of these vertices are final, it can be proven that no shorter path exists. We will discuss why this is the case later on.

Let's note the selected vertex with v.

Then, we iterate on the outgoing edges of the selected vertex v. For each outgoing edge $v \overset{w}{\to} u$, we form a candidate path to its endpoint u, by appending this edge to the currently known shortest path to the selected vertex, e.g. $a \overset{D[v]}{\rightsquigarrow} v \overset{w}{\to} u$. The length of the candidate path is $D[v] + w$.

This candidate path is then compared to the currently known shortest path to the endpoint, $a \overset{D[u]}{\rightsquigarrow} u$. If the candidate path is shorter, we update the value of $D[u]$, with the length of the candidate path $D[v] + w$, and we also take note in $P[u]$, that the value of $D[u]$ came from the path via v, by setting $P[u] = v$. If the candidate path is longer or the same length, then we just discard it.

We keep doing this process, until all vertices are finalized and in the $SAFE$ set. At the end, the D vector will contain the lengths the shortest paths towards all vertices from the starting one and the P vector will contain all the information necessary to retrieve those paths, which we will show how to do at the end of this exercise.

Let's begin!

In the first step, we finalize vertex a and update the distances along the two outgoing edges:

- *To update along the outgoing edge $a \overset{5}{\to} b$, we consider the candidate path $a \overset{D[a]=0}{\rightsquigarrow} a \overset{5}{\to} b$, towards vertex b, via a, of length 5, which is technically just the edge $a \overset{5}{\to} b$, and compare it to the currently known shortest path $a \overset{D[b]=\infty}{\rightsquigarrow} b$, towards vertex b, of length ∞, which technically just means that no path has been found so far. Since the candidate path's length is shorter, we update by setting $D[b]$ to this length, $D[b] = 5$ and taking note that this value was found via the parent vertex a, by setting $P[b] = a$.*
- *We update along the other outgoing edge $a \overset{6}{\to} e$, and with the exact same logic we update $D[e] = 6$ and $P[e] = a$.*

In the second step, we must choose the vertex that has the smallest D value and not yet finalized, this is vertex b, and finalize it. We then update along its outgoing edges.

- *To update along the outgoing edge $b \overset{4}{\to} c$, we consider the candidate path $a \overset{D[b]=5}{\rightsquigarrow} b \overset{4}{\to} c$, towards vertex c, via b, of length 9, and compare it to the currently known shortest path $a \overset{D[c]=\infty}{\rightsquigarrow} c$, towards vertex c, of length ∞, which technically just means that no path has been found so far. Since the candidate path's length is shorter, we update by setting $D[c]$ to this length, $D[c] = 9$ and taking note that this value was found via the parent vertex b, by setting $P[c] = b$.*
- *In a similar fashion, updating along the other outgoing edge $b \overset{6}{\to} d$, we consider the candidate path $a \overset{D[b]=5}{\rightsquigarrow} b \overset{6}{\to} d$, via b, of length 11 and since it is shorter than the currently known shortest path, $a \overset{D[d]=\infty}{\rightsquigarrow} d$, we update $D[d] = 11$ and $P[d] = b$.*

This is the end of the second step.

In the third step we finalize e, and update with the two outgoing edges:

- *Candidate $a \overset{D[e]=6}{\rightsquigarrow} e \overset{2}{\to} c$ is shorter than $a \overset{D[e]=9}{\rightsquigarrow} c$, so we update $D[c] = 8$ and $P[c] = e$.*

- Candidate $a \overset{D[e]=6}{\rightsquigarrow} e \xrightarrow{1} f$ is shorter than $a \overset{D[f]=\infty}{\rightsquigarrow} f$, so we update $D[f] = 7$ and $P[f] = e$.

In the fourth step we finalize f, and update with the three outgoing edges:

- Candidate $a \overset{D[f]=7}{\rightsquigarrow} f \xrightarrow{3} b$ is longer than $a \overset{D[b]=5}{\rightsquigarrow} b$, so we discard it.

Note here, that b is already in the $SAFE$ set, so its D value is already finalized. This means, that any candidate paths found towards b **will be discarded** by the algorithm. How can we be sure, this will be the case, always? Because b was finalized before f was, we had a shorter path towards b than what we had towards f at the moment. (Remember, we finalize shorted D values first.) The best path towards b is of length 5. There is no possible way to get a shorter path towards b via vertex f, because the best path towards f is of length 7. **Since the graph does not have negative edges, any candidate path towards b via f will be of length at least 7, so it will be discarded in favour of the already known path of length 5.** Therefore every time we finalize a vertex, we can be sure that no further candidate paths via any of the other vertices could give us a shorter path, because the D values of the other vertices are higher and no negative edges exist, so the candidate paths' lengths will be longer too. And this is why Dijkstra's algorithm is correct.

Continuing with the other two outgoing edges from f:

- Candidate $a \overset{D[f]=7}{\rightsquigarrow} f \xrightarrow{1} c$ is the same length as what we already have, path $a \overset{D[c]=8}{\rightsquigarrow} c$, so we discard it. It is enough to keep one of the shortest paths, we don't care about multiple solutions.

- Candidate $a \overset{D[f]=7}{\rightsquigarrow} f \xrightarrow{1} d$ is shorter than $a \overset{D[d]=11}{\rightsquigarrow} d$, so we update $D[d] = 7$ and $P[d] = f$.

Now we finalize vertex c. Note here, that both c and d have a D value of 8. In this case, we can arbitrarily choose any of them as we execute this algorithm on paper. In the official solutions, to keep them consistent, we usually choose vertex with the lexicographically smallest label.

When implementing this algorithm in a programming language, the choice is based on how the minimum finding algorithm on the not-yet finalized vertices is implemented. For example, this could be a linear search on the D vector, in which case the lexicographically smallest vertex label is selected in case of multiple smallest values. Alternatively, it could be implemented more effectively using a combination of a heap (see `std::priority_queue` in C++) and a hash table (see `std::unordered_set` in C++), in which case the D value, that was inserted earlier, will be selected.

This choice is therefore completely up to how the algorithm is implemented (we usually call this implementation detail) and we do not concern ourselves with it, as the correctness of the algorithm does not depend on it.

So let us finalize c and update along its outgoing edges:

- Candidate $a \overset{D[c]=8}{\rightsquigarrow} c \xrightarrow{3} a$ is longer than $a \overset{D[a]=0}{\rightsquigarrow} a$, so we discard it. Notice here, that this is acually a cycle containing the starting vertex a, of length 11.

- Candidate $a \overset{D[c]=8}{\rightsquigarrow} c \xrightarrow{1} d$ is longer than $a \overset{D[d]=8}{\rightsquigarrow} d$, so we discard it.

We then finalize d, which is the last one. Since all vertices are in the $SAFE$ set, no candidate can give us a shorter path towards any of them, the algorithm is finished. In total, we needed $n - 1$ steps, since we do not need to update via the last finalized vertex.

The final distance vector D is:

a	b	c	d	e	f
0	5	8	8	6	7

And the final parent vector P is:

a	b	c	d	e	f
-	a	e	f	a	e

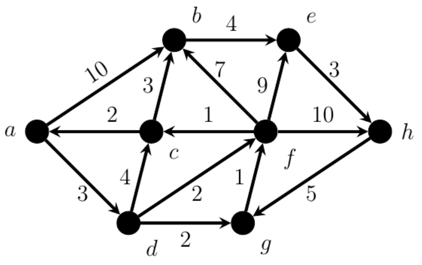
The only question left is how do we recontruct the shortest path to a vertex using the information in P. For example, the shortest path from a to d is of length $D[d] = 8$. Which vertices are along that path?

- *The parent of d is $P[d] = f$, therefore the last edge on the path is $f \xrightarrow{1} d$.*
- *The parent of f is $P[f] = e$, therefore the previous edge on the path is $e \xrightarrow{1} f$, making the path so far $e \xrightarrow{1} f \xrightarrow{1} d$.*
- *The parent of e is $P[e] = a$, therefore the previous edge on the path is $a \xrightarrow{6} e$, making the path so far $a \xrightarrow{6} e \xrightarrow{1} f \xrightarrow{1} d$.*

And since a is the starting vertex, we are done. The shortest path from a to d of length 8 is $a \xrightarrow{6} e \xrightarrow{1} f \xrightarrow{1} d$. Note here, that this decoding procedure was a recursive algorithm.

---

72. Use Dijkstra's algorithm to determine the lengths of the shortest paths from a to all other vertices of the graph. Also provide the shortest paths (as a sequence of vertices). (No need to justify in words, but one should be able to see the steps, how the safe set $SAFE$ and the arrays D and P update. Please also show the table for final distances and the parents.)



Solution:

D	a	b	c	d	e	f	g	h	$SAFE$
0.	0	∞	∞	∞	∞	∞	∞	∞	$\{\}$
1.	0	10	∞	3	∞	∞	∞	∞	$\{a\}$
2.	0	10	7	3	∞	5	5	∞	$\{a,d\}$
3.	0	10	6	3	14	5	5	15	$\{a,d,f\}$
4.	0	10	6	3	14	5	5	15	$\{a,d,f,g\}$
5.	0	9	6	3	14	5	5	15	$\{a,d,f,g,c\}$
6.	0	9	6	3	13	5	5	15	$\{a,d,f,g,c,b\}$
7.	0	9	6	3	13	5	5	15	$\{a,d,f,g,c,b,e\}$
8.	0	9	6	3	13	5	5	15	$\{a,d,f,g,c,b,e,h\}$

P	a	b	c	d	e	f	g	h
0.	–	–	–	–	–	–	–	–
1.	–	a	–	a	–	–	–	–
2.	–	a	d	a	–	d	d	–
3.	–	a	f	a	f	d	d	f
4.	–	a	f	a	f	d	d	f
5.	–	c	f	a	f	d	d	f
6.	–	c	f	a	b	d	d	f
7.	–	c	f	a	b	d	d	f
8.	–	c	f	a	b	d	d	f

- *The shortest path from a to b is $a \xrightarrow{3} d \xrightarrow{2} f \xrightarrow{1} c \xrightarrow{3} b$, of length 9.*
- *The shortest path from a to c is $a \xrightarrow{3} d \xrightarrow{2} f \xrightarrow{1} c$, of length 6.*
- *The shortest path from a to d is $a \xrightarrow{3} d$, of length 3.*
- *The shortest path from a to e is $a \xrightarrow{3} d \xrightarrow{2} f \xrightarrow{1} c \xrightarrow{3} b \xrightarrow{4} e$, of length 13.*
- *The shortest path from a to f is $a \xrightarrow{3} d \xrightarrow{2} f$, of length 5.*
- *The shortest path from a to g is $a \xrightarrow{3} d \xrightarrow{2} g$, of length 5.*
- *The shortest path from a to h is $a \xrightarrow{3} d \xrightarrow{2} f \xrightarrow{10} h$, of length 15.*

73. Determine the directed graph G with the smallest possible number of edges, so that the given table represents the status of the distance array when the Dijkstra's algorithm is run on G. Specify the contents of $SAFE$ at each stage and the parent of each vertex.

D	v_1	v_2	v_3	v_4	v_5	v_6
1.	$*$	2	6	∞	∞	7
2.	$*$	$*$	5	9	∞	6
3.	$*$	$*$	$*$	6	9	6
4.	$*$	$*$	$*$	$*$	8	6
5.	$*$	$*$	$*$	$*$	7	$*$

Solution:

The $*$ symbol represents the vertex being put into the $SAFE$ set.

In the first round v_1 was moved to the $SAFE$ set (it was the starting vertex, hence $D[v_1] = 0$). We were updating along the outgoing edges of v_1 in this round.

- *$D[v_2]$ is set to 2, which was done due to an update with candidate path $v_1 \overset{D[v_1]=0}{\rightsquigarrow} v_1 \xrightarrow{2} v_2$, of length 2. Therefore edge $v_1 \xrightarrow{2} v_2$ must exist.*

- *$D[v_3]$ is set to 6, which was done due to an update with candidate path $v_1 \overset{D[v_1]=0}{\rightsquigarrow} v_1 \xrightarrow{6} v_2$, of length 6. Therefore edge $v_1 \xrightarrow{6} v_3$ must exist.*

- *$D[v_7]$ is set to 7, which was done due to an update with candidate path $v_1 \overset{D[v_1]=0}{\rightsquigarrow} v_1 \xrightarrow{7} v_7$, of length 7. Therefore edge $v_1 \xrightarrow{7} v_7$ must exist.*

In the second round v_2 was moved to the $SAFE$ set. We were updating along the outgoing edges of v_2 in this round.

- *$D[v_3]$ is updated to 5, which was done due to an update with candidate path $v_1 \overset{D[v_2]=2}{\rightsquigarrow} v_2 \xrightarrow{3} v_3$, of length 5. Therefore edge $v_2 \xrightarrow{3} v_3$ must exist. Notice that the weight of the edge is not 5, but $5 - D[v_2]$.*

- *$D[v_4]$ is updated to 9, which was done due to an update with candidate path $v_1 \overset{D[v_2]=2}{\rightsquigarrow} v_2 \xrightarrow{7} v_4$, of length 9. Therefore edge $v_2 \xrightarrow{7} v_4$ must exist.*

- *$D[v_6]$ is updated to 6, which was done due to an update with candidate path $v_1 \overset{D[v_2]=2}{\rightsquigarrow} v_2 \xrightarrow{4} v_6$, of length 6. Therefore edge $v_2 \xrightarrow{4} v_6$ must exist.*

In the third round v_3 was moved to the $SAFE$ set. We were updating along the outgoing edges of v_3 in this round.

- *$D[v_4]$ is updated to 6, which was done due to an update with candidate path $v_1 \overset{D[v_3]=5}{\rightsquigarrow} v_3 \xrightarrow{1} v_4$, of length 6. Therefore edge $v_3 \xrightarrow{1} v_4$ must exist.*

- *$D[v_5]$ is updated to 9, which was done due to an update with candidate path $v_1 \overset{D[v_3]=5}{\rightsquigarrow} v_3 \xrightarrow{4} v_5$, of length 9. Therefore edge $v_3 \xrightarrow{4} v_5$ must exist.*

- *Note here, that we do not know whether a candidate path was considered and discarded for v_6, as its value remained the same. However, the task specifies, that we are looking for the smallest possible number of edges, so we can assume such a candidate path was not available.*

In the fourth round v_4 was moved to the $SAFE$ set. We were updating along the outgoing edges of v_4 in this round.

- *$D[v_5]$ is updated to 8, which was done due to an update with candidate path $v_1 \overset{D[v_4]=6}{\rightsquigarrow} v_4 \xrightarrow{2} v_6$, of length 8. Therefore edge $v_4 \xrightarrow{2} v_6$ must exist.*

- *Similarly to the previous round, we do not know whether a candidate path was considered and discarded for v_6, as its value remained the same. We can assume, that there was no such candidate.*

Finally in the fifth round v_6 was moved to the $SAFE$ set. We were updating along the outgoing edges of v_6 in this round.

- *$D[v_5]$ is updated to 7, which was done due to an update with candidate path $v_1 \overset{D[v_6]=6}{\rightsquigarrow} v_6 \xrightarrow{1} v_5$, of length 7. Therefore edge $v_6 \xrightarrow{1} v_5$ must exist.*

Therefore, the graph with the minimal number of edges corresponding to the Dijkstra run above is given by this adjacency list: $\mathbf{v_1}: v_2(2), v_3(6), v_7(7)$; $\mathbf{v_2}: v_3(3), v_4(7), v_6(4)$; $\mathbf{v_3}: v_4(1), v_5(4)$; $\mathbf{v_4}: v_6(2)$; $\mathbf{v_6}: v_5(1)$.

74. We are given a city's road network as a weighted directed graph: the vertices are the intersections, the edges are the direct paths between the intersections, and the weights of the edges show the average time it takes to travel the road by car.

Two intersections in the city, a and b, will be closed next week for road renovations (these cannot be traversed by car). Given two designated vertices in the graph, S and T, we want to decide whether and by how much the time to get from S to T will increase due to the closure of the nodes a and b. (Assume that the average times assigned to the direct roads between intersections do not change due to the closures.) Which algorithm can be applied if we want to solve this problem in $O(n^2)$ running time (where n denotes the number of intersections)?

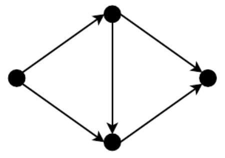
Solution:

Let's assume the graph is given via its adjacency matrix. We first run Dijkstra's algorithm from S on it, which gives the lengths of the shortest paths from S to all other vertices, including T. This takes $O(n^2)$ time. Then, we delete all incoming and outgoing edges of vertices a and b. In the matrix repesentation, this means clearing two rows and two columns and can be done in $O(n)$ time. The resulting graph still has n vertices (two of them isolated now), so running Dijkstra on it still takes $O(n^2)$ time. We shall run Dijkstra a second time from S and check the length of the shortest path to T. We should substract from this number the original result, which tells us how much more time it takes to get from S to T. This difference cannot be negative, and if it equals zero, then the travel time has not changed. This step takes $O(1)$. The entire procedure, considering all previous steps takes $O(n^2)$ time.

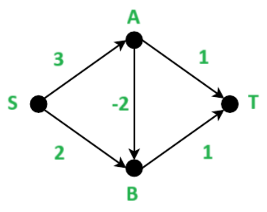
Note: *If the graph was given by its adjacency list, we can convert it to an adjacency matrix format in $O(n^2)$, or we can also run Dijkstra on the adjacency list format directly, with the small correction, that removing all incoming and outgoing edges of a and b now takes $O(n^2)$ time, in particular, to remove all incoming edges, one must iterate the entire adjacency list.*

If we want to consider the edge case, where T is not reachable from S, we must check if $D[T] = \infty$ at the end of any Dijkstra runs and print out "T is not reachable!" (and exit after) for the first run and "T is no longer reachable!" for the second run.

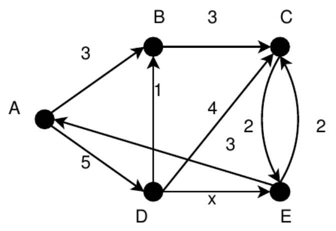
75. Assign edge weights to the edges in the graph shown on the right such that in the resulting graph the Dijkstra's algorithm incorrectly computes the lengths of the shortest paths.



Solution:
Dijkstra may incorrectly compute the shortest paths if we run it on a graph that has at least one negative edge. There are many possible solutions, this is one example. Running Dijkstra on this graph from S will finalize vertex B before A, however, there is an alternative path to B via A that is only length 1, due to the negative edge $A \xrightarrow{-2} B$, which does not get discovered by Dijkstra.



76. Use Dijkstra's algorithm to find the shortest paths from vertex A to all other vertices in the graph shown on the right, as function of positive real parameter x.



Solution:

After the first three steps the SAFE set becomes $\{A, B, D\}$, irrespective of the value of x. The D vector's *values will be $0, 3, 6, 5, x+5$ at this point. After this, in step 4 there are two main cases, depending on whether vertex C or vertex E is placed into the $SAFE$ set. If $x < 1$, then E is placed into the $SAFE$ set, if $x = 1$ then both cases are possible, depending on the implementation, finally if $x > 1$ then C is placed into the $SAFE$ set. Calculating the results in these cases the final values for the D vector are $0, 3, 6, 5, x+5$, when $x \leq 3$, and otherwise when $x > 3$, then $0, 3, 6, 5, 8$.*

See full calculations below:

Calculation for $x < 1$:

D	A	B	C	D	E	$SAFE$
1.	0	3	∞	5	∞	$\{A\}$
2.	0	3	6	5	∞	$\{A, B\}$
3.	0	3	6	5	$x+5$	$\{A, B, D\}$
4.	0	3	$\min\{6, x+7\} = 6$	5	$x+5$	$\{A, B, D, E\}$
5.	0	3	6	5	$x+5$	$\{A, B, D, E, C\}$

Calculation for $x \geq 1$:

D	A	B	C	D	E	$SAFE$
1.	0	3	∞	5	∞	$\{A\}$
2.	0	3	6	5	∞	$\{A, B\}$
3.	0	3	6	5	$x+5$	$\{A, B, D\}$
4.	0	3	6	5	$\min\{8, x+5\}$	$\{A, B, D, C\}$
5.	0	3	6	5	$\min\{8, x+5\}$	$\{A, B, D, C, E\}$

---

77. We are given a city's road network as a weighted directed graph: the vertices are the intersections, the edges are the direct paths between the intersections, and the weights of the edges show the average time it takes a cyclist to cycle along the road.
A bicycle courier at vertex f is given the task of delivering two parcels he has with him to the nodes b and c of the city as quickly as possible (in any order). Which algorithm can be applied to determine, with running time $O(n^2)$, the order in which the courier should deliver the parcels and the shortest time in which he can complete his task?

Solution:

The courier can either start in f, then move to b and finish in node c, or they can start in f, move to c, then finish in node b.

Therefore, we must run Dijkstra's algorithm three times: to calculate the distances from vertex f, vertex b and vertex c to any other vertices in the graph. Let us denote our results with the distance arrays D_f, D_b and D_c.

The shortest delivery time is then either $D_f[b] + D_b[c]$, for visiting the vertices in order f, b, c, or $D_f[c] + D_c[b]$, for visiting the vertices in order f, c, b, the courier must choose the minimum value between the two options.

The three Dijkstra's take $O(n^2)$ time, then choosing the minimum is $O(1)$, for a total of $O(n^2)$.

78. We are given a directed graph G with its adjacency matrix. We know that, with the exception of one negative weight edge, all the other edges have positive weight. There are no negatively weighted cycles in the graph. Give an algorithm with running time $O(n^2)$ to find the shortest paths from a vertex $s \in V(G)$ to all other vertices.

Solution:

Let us denote the negative edge with (u, v) and its negative weight with $w(u, v) < 0$. Since there is no negative cycle, the "shortest path" task is still valid in this graph, but we cannot use Dijkstra directly. (If there was a negative cycle in the task, we could traverse it many times, always decreasing our distance, resulting in a $-\infty$ distance from any vertex from which it is reachable to any vertex we can reach from it.)

Whatever the shortest path from vertex s to another vertex t may be, it either contains the edge (u, v), or it does not.

If it does not contain the edge (u, v), then the path can be found, by first removing the negative edge from the graph, then running Dijkstra on it and reading the result for t.

If it does contain the edge (u, v), then we can similarly remove the negative edge first, then run Dijkstra from vertices s and v, denoting the results with D_s and D_v, then combine the shortest path from the following three parts: The shortest path from s to u, denoted by $D_s[u]$, then the weight of the edge (u, v), denoted by $w(u, v)$ then the shortest path from v to t, denoted by $D_v[t]$. However, the list of edges we got may not form a path, since vertices may repeat in the first and the second half. We must check our result if this is the case or not. If this is the case, the path between those repeating vertices form a cycle, which can be removed from the path, including the negative edge inside of it. The resulting path will be at least as long as the one we found in the first case (it could be longer, if the negative edge we discarded decreased the length by a lot), so we can just discard this case, if we find a repeating vertex.

For the runtime complexity note here, that we only run two Dijsktra's in total, one from s and one from v and calulate the results for all target vertices from these. To check for repeating vertices, we could store the vertices appearing on the shortest path from s to u in a boolean array, then noting if we used one during the second Dijkstra run from v. This lookup is one extra step during the runtime of Dijkstra, so the algorithm is still $O(n^2)$.

79. We are given a simple directed graph G by its adjacency list, with non-negative edge weights. Let v_1 be an arbitrary vertex of G. Give an effective algorithm that finds the 100 closest vertices to v_1.

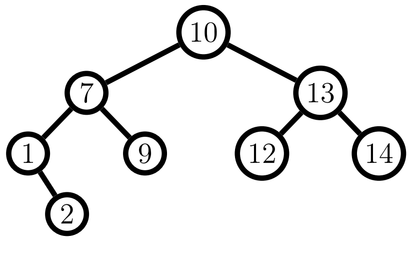
Solution:

Let us run the first 100 *steps of Dijkstra, the version without using heap or keeping track of parents in P. This will place into the $SAFE$ set, the* 100 *closest vertices to v_1, since they are placed one-by-one in increasing order of distance from v_1.*

The runtime complexities are as follows: $O(n)$ for initialization of D, $O(100n) = O(n)$ for finding 100 *minimums, one in each step and $O(100n) = O(n)$, for modifying the D vector* 100 *times. Therefore, the total cost is $O(n)$.*

Note: This is smaller than the size of the input, we don't need to read the entire adjacency list.

80. In what orders list PreOrder, InOrder and PostOrder walks the keys stored in the tree on the right?



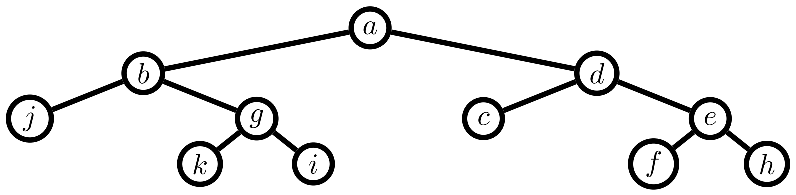
Solution:

PreOrder: 10, 7, 1, 2, 9, 13, 12, 14.
InOrder: 1, 2, 7, 9, 10, 12, 13, 14.
PostOrder: 2, 1, 9, 7, 12, 14, 13, 10.

81. Let the InOrder sequence of the elements of a binary tree be $j, b, k, g, i, a, c, d, f, e, h$, and let the PreOrder be: $a, b, j, g, k, i, d, c, e, f, h$. Use this information to reconstruct the tree.

Solution:

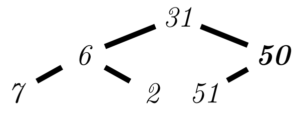
We can see from the PreOrder, that the root is a. From the InOrder, we can see that the left subtree contains j, b, k, g, i, and the right subtree contains c, d, f, e, h. Looking back at the PreOrder, we see that the root of the left subtree is b and the root of the right subtree is d. We can continue the same logic until the tree is fully *reconstructed. The final result is:*



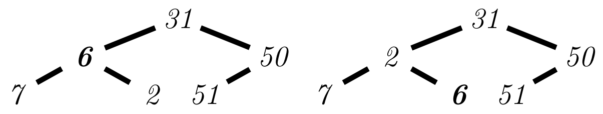
82. (a) Construct a heap using the linear time BUILD-HEAP method from the following array of numbers: $31, 6, 50, 7, 2, 51$.
(b) Insert the numbers 1 and then 5 into the resulting heap.
(c) Perform two consecutive ExtractMin operations on the resulting heap.

Solution:

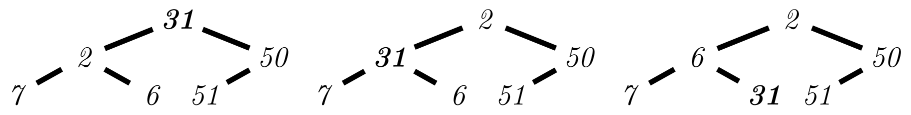
a) We HEAPIFY 50 *first. (No change.)*



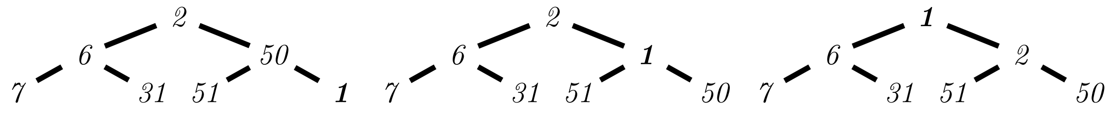
We then HEAPIFY 6*. Among* $6, 7, 2$*, the* 2 *is smallest, so they switch.*



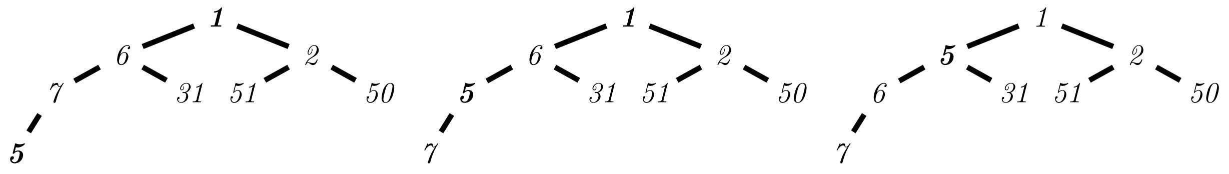
We then HEAPIFY 31. Among $31, 2, 50$*, the* 2 *is smallest, so they switch. Then among* $31, 7, 6$*, the* 6 *is smallest, so they switch.*



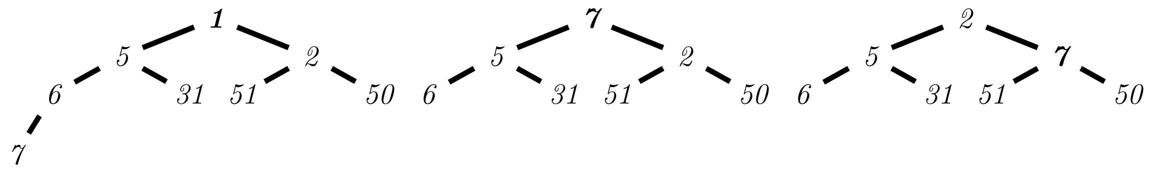
b) 1 *is inserted as the next child, then HEAPIFY is called upwards. Among* $50, 51, 1$*, the* 1 *is smallest so it switches with the parent. Then, among* $2, 6, 1$*, the* 1 *is smallest so it switches with the parent.*



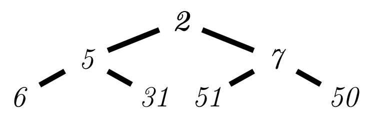
5 *is inserted as the next child, then HEAPIFY is called upwards. Among* $7, 5$*, the* 5 *is smallest so it switches with the parent. Among* $6, 5, 31$*, the* 5 *is the smallest so it switches with the parent.*

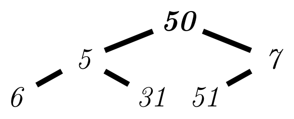


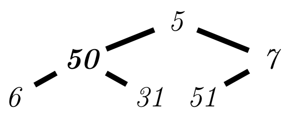
c) First ExtractMin: 1 *is removed and the last leaf,* 7 *is placed as root instead. Then we call HEAPIFY downwards on* 7*. Among* $7, 5, 2$*, the* 2 *is smallest so it switches with the parent. Among* $7, 51, 50$*, the* 7 *is the smallest, so no change occurs.*

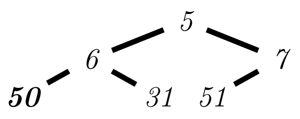


Second ExtractMin: 2 is removed and the last leaf, 50 is placed as root instead. Then we call HEAPIFY downwards on 50. Among 50, 5, 7, the 5 is the smallest, so it switches with the parent. Among 50, 6, 31, the 6 is the smallest, so it switches with the parent.





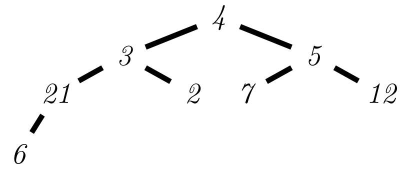




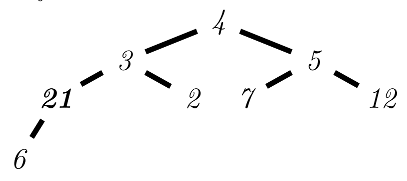
83. (a) Construct a heap using the linear time BUILD-HEAP method from the following array of numbers: $4, 3, 5, 21, 2, 7, 12, 6$.
(b) Insert the number 1 into the resulting heap.
(c) Perform two consecutive ExtractMin operations on the resulting heap.

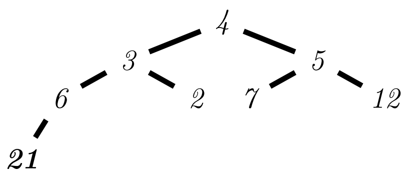
Solution:

a) We start with the array as-is, which represents the Heap, where the node at index i has the left child at index $2i$ and the right child at index $2i + 1$. (And we start indexing from 1.)

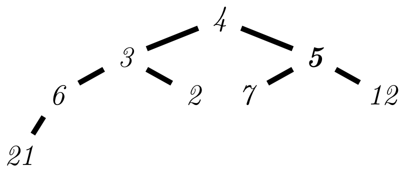


HEAPIFY is called downwards on the non-leaf elements, from bottom-to-up, from right-to-left. We HEAPIFY 21 *first.*

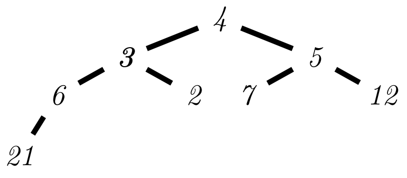


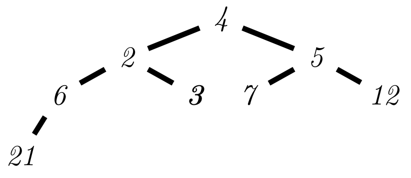


Then we HEAPIFY 5. *(No change.)*

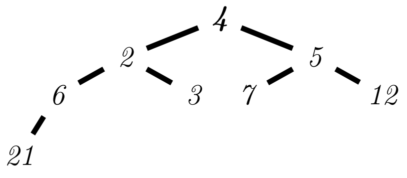


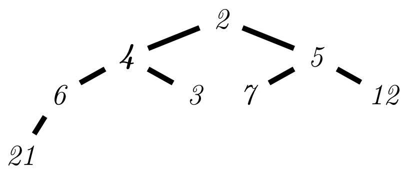
Then we HEAPIFY 3.

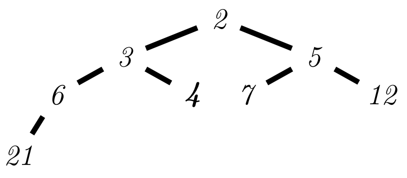




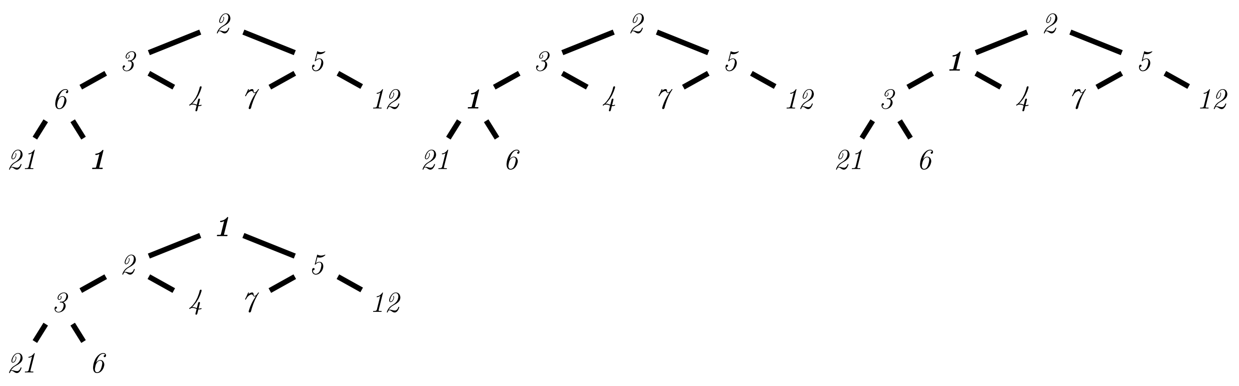
Then we HEAPIFY 4.



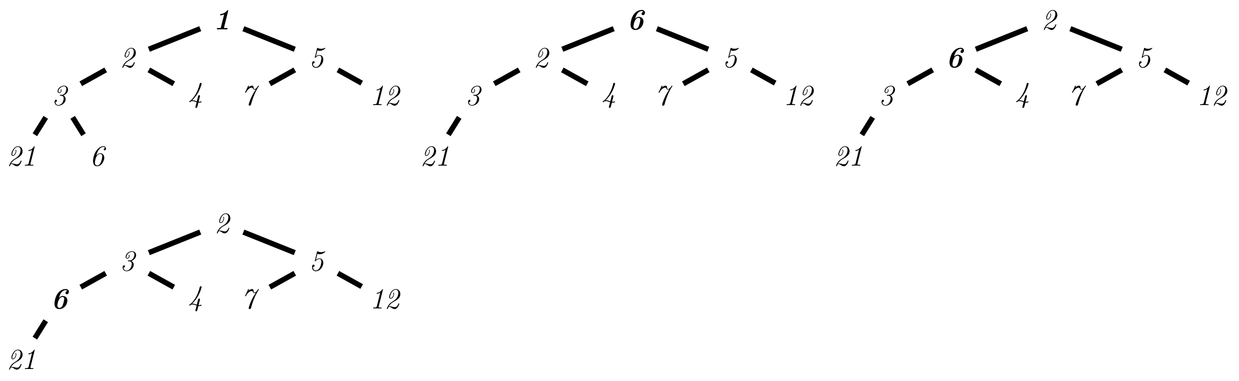




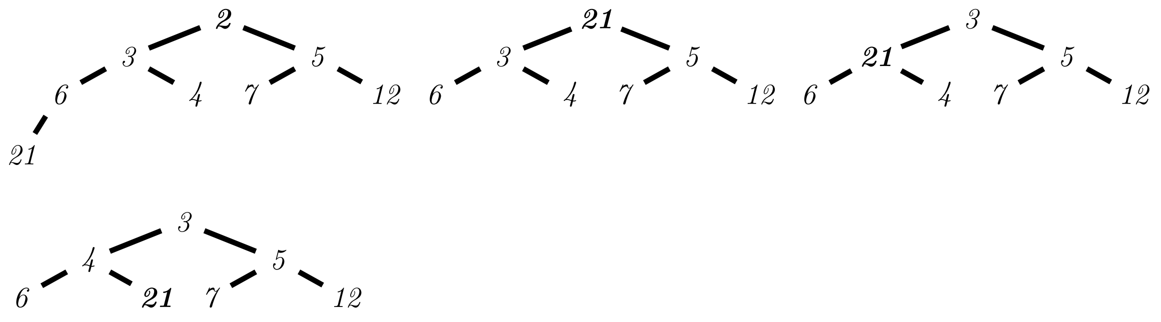
b) Element 1 *inserted as the next leaf and we HEAPIFY it upwards.*



c) The first ExtractMin removes 1 *and the last leaf,* 6 *is placed as root. Then we HEAPIFY* 6 *downwards.*



The second ExtractMin removes 2 *and the last leaf,* 21 *is placed as root. Then we HEAPIFY* 21 *downwards.*



84. Prove that building a heap of n elements requires $\Omega(n)$ comparisons.

Solution:

After the heap has been built, the smallest element can be found without further comparisons (it's in the root). So building the heap requires at least as many comparisons as finding the minimum does, which is $n-1$*.*

85. We are given a heap of n elements and a key k. Give an algorithm that finds the elements of the heap that are smaller than k. If there are m such elements, then the running time must be $O(m)$.

Solution:

We'll use a method similar to PreOrder traversal. We begin by comparing the root value with k*. If the root value is smaller than* k*, we print it. If it is larger or equal, we stop exploring this path since all subsequent nodes will also be larger (in a heap, each parent is smaller than its children, due to the heap property). Next, we recursively apply the same method to the left and right subtrees. The time complexity is* $O(m)$ *because for each element less than* k*, we only consider its two direct children and then stop. Thus, we visit at most* $3m$ *elements, which means the algorithm has* $O(m)$ *time complexity.*

86. (a) Give an efficient algorithm for finding the second smallest element in a heap.
(b) Give an efficient algorithm for finding the tenth smallest element in a heap.

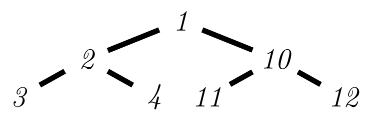
Solution:

(a) The second smallest element is a child of the root, since only the root is smaller than it. So we compare the two children of the root and print the smallest one, which is a constant time ($O(1)$) algorithm.

(b) The distance of the 10th element from the root is at most 9, since only 9 elements can be smaller than it.

The number of such elements is at most $2^{10} - 1 = 1023$, and these are located in the first 1023 positions of the array representing the heap. Finding the 10th smallest among these constant many elements requires $O(1)$ (constant) time.

It's important to note that the children of the root in a heap aren't necessarily smaller than all other elements. For example, this is a heap where one subtree has smaller elements than the other:



Only when moving along a path from the root towards a specific node can we say, that the elements on the path before the node must be smaller, due to the heap property.

87. n keys are inserted into an initally empty heap using succesive Insert operations. Show that the number of comparisons used may be $\Omega(n \log n)$.

Solution:

If we insert the numbers $n, n-1, n-2, \dots, 1$ in this order, then the number of comparisons made is

$$\sum_{i=1}^{n} \lfloor \log i \rfloor \geq \sum_{i=1}^{n} ((\log i) - 1) = \left(\sum_{i=1}^{n} \log i \right) - n \geq \left(\sum_{i=n/2}^{n} \log \frac{n}{2} \right) - n = \frac{n}{2} \log \frac{n}{2} - n \in \Omega(n \log n).$$

88. n elements of an ordered set are stored in a heap. Show that it takes $\Omega(n)$ comparisons to find the largest element.

Solution:

The largest element must be a leaf, since if it had children, they have to be larger than it. To prove that one leaf is larger than all other leaves, each other leaf must be proven to be smaller than at least one other element in the heap. Therefore at least (number of leaves) $- 1 = \Omega(n)$ *comparisons are necessary.*

89. In a doctor's office, you have to check in at the registration desk, where the staff decide whether the patient must be assigned to doctor A or to doctor B or if the patient can be assigned to either one of them. Also, after inpecting the referral note, the patient will be assigned a number indicating the urgency of the case. Assume that a smaller number indicates a greater emergency. When a doctor has finished with a patient, he picks a patient with the smallest emergency number out of all the patients who can/must be treated by him. Assume that the emergency numbers assigned to all patients waiting are different. Describe a data-structure that, in the case where n patients are waiting, allows the registration of a new patient and the selection of the next patient by the doctors in $O(\log n)$ steps.

Solution:

We use 3 heaps, where in the first one the emergency numbers of those patients are placed who can only be treated by doctor A, in the second one those who can only be treated by doctor B, and in the third one those who can be treated by both A and B.

During registration, the insertion of a new patient occurs by inserting their emergency number (using the usual heap insertion) into the corresponding heap, which takes $O(\log n)$ steps, since each of the three heaps contains at most n elements.

Both doctors select the next patient in $O(1)$ steps by reading and comparing the elements in the roots of the heaps containing emergency numbers of patients they can treat. Then, the doctor calls in the patient corresponding to the smaller emergency number (more urgent), and then executes an ExtractMin operation on the respective heap (with the help of the registration software), which takes $O(\log n)$ steps. After the ExtractMin operation, the heap property is restored, so the selection of the next patient can happen in the same way.

90. We want to build a heap from n elements of an ordered set. However, besides the heap property we need that for each node x the keys in the left subtree of x are smaller than the keys in the right subtree of x. Prove that this requires $\Omega(n \log n)$ comparisons.

Solution:

Let's assume there's an algorithm to create the desired structure. Then from the resulting tree, the PreOrder traversal would produce an ordered list in $O(n)$ steps. This implies that at least as many steps are needed to create the desired structure as for sorting, which we have proven to be $\Omega(n \log n)$ on the lecture.

91. There are n points on the plane, such that any two of them differ in both coordinates. Prove that there exists a unique binary tree that stores these n points so that it is a binary search tree for the first coordinates and it has the heap property for the second coordinates of the points. (Note, that the heap property does not mean that the tree is an almost complete binary tree, like the one constructed by BuildHeap studied in class.)

Solution:

Due to the heap property, the root contains the point P with the smallest second coordinate. Due to the search tree property, in the left subtree, there are points whose first coordinate is smaller than that of P, and in the right subtree, there are points whose first coordinate is greater than that of P. Based on this, the statement follows by induction.

92. We are given a binary search tree containing all distinct integers. Is it possible that during a call to SEARCH(x), we see the key-values $20, 18, 3, 15, 5, 8, 9$ in this order along the search path? If it is not possible, give a reason why not, and if it is possible, then specify all the integers x for which this can happen.

Solution:

20 *is the root of the tree. In the next step, we see an* 18*, which is smaller, which means we moved to the left subtree of* 20*. Therefore whatever x may be, it is smaller than* 20*, hence why we choose to move to the left.*

Repeat the same logic for the rest of the path:

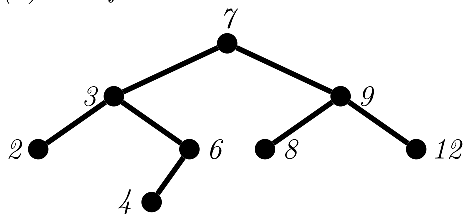
- $18 < 20$*, therefore $x < 20$ as already stated.*
- $3 < 18$*, therefore $x < 18$.*
- $15 > 3$*, therefore $x > 3$.*
- $5 < 15$*, therefore $x < 15$.*
- $8 > 5$*, therefore $x > 5$.*
- $9 > 8$*, therefore $x > 8$.*

Either $x = 9$ and we found it, or we were not successful. Summarizing the above information, we know that $8 < x < 15$*. For any such x, the search path is identical in this tree.*

93. (a) Insert the following elements into an empty binary search tree: $7, 3, 2, 9, 8, 12, 6, 4$.
(b) What is the PreOrder, InOrder and PostOrder of the elements?
(c) Execute Insert(5), Delete(2), Delete(7), Delete(6).

Solution:

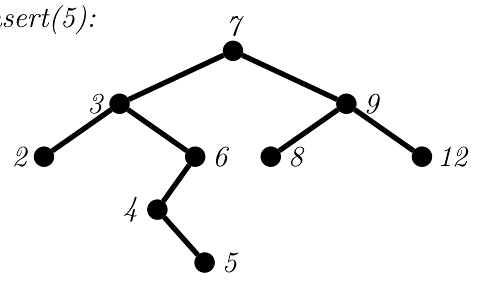
(a) The final tree will look like this:



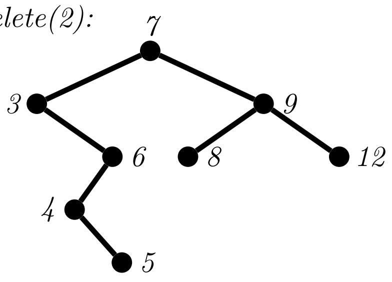
(b) The traversal results:
PreOrder: 7, 3, 2, 6, 4, 9, 8, 12
InOrder: 2, 3, 4, 6, 7, 8, 9, 12
PostOrder: 2, 4, 6, 3, 8, 12, 9, 7

(c)

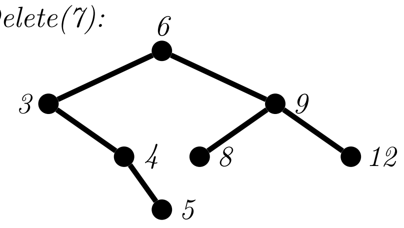
Insert(5):



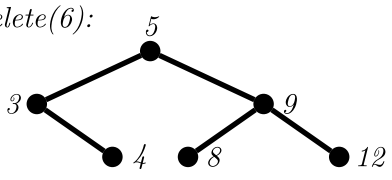
Delete(2):



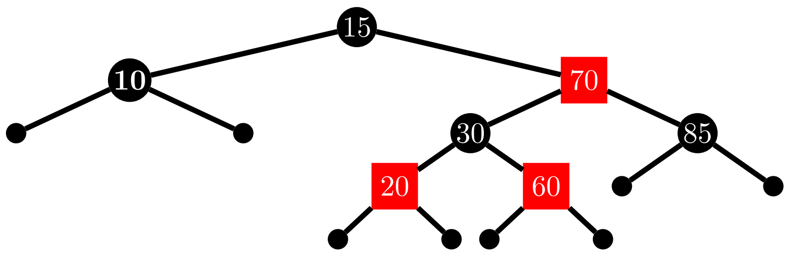
Delete(7):



Delete(6):

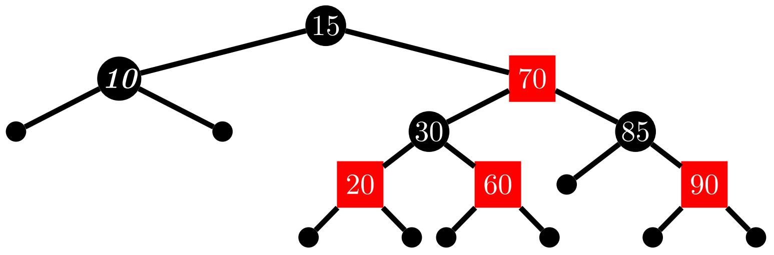


94. (a) Insert 90 into the Red-Black tree on the right.
(b) Insert 50 into the Red-Black tree on the right (that does not yet contain 90).

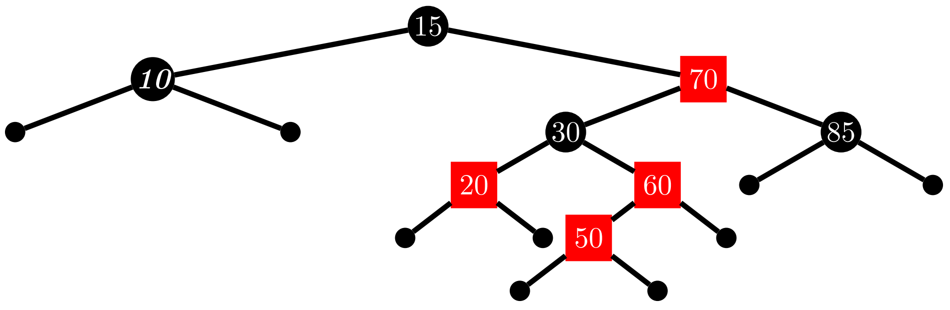


Solution:

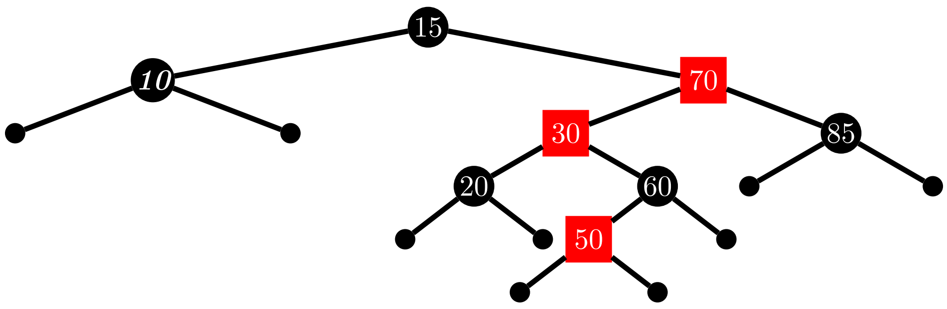
(a) 90 *is inserted as red in its position. No furhter action needed, the red-black tree is valid.*



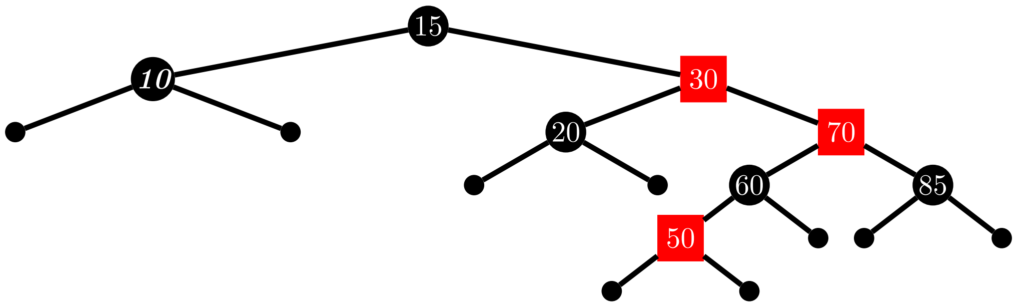
(b) 50 *is inserted at its position. The parent is red, which violates the 7th rule.*



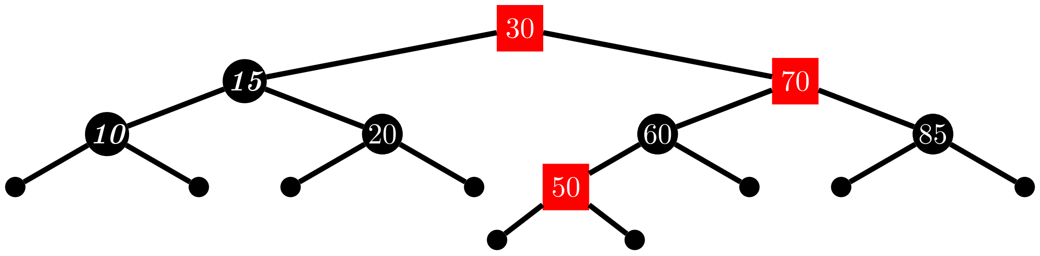
This is Case 2.1 - Red Uncle > Recolor. The uncle (20) *is also red. We recolor the parent* (60) *and the uncle* (20) *to black, while the grandparent* (30) *is colored red, to keep the black heights equal. The issue propagated 2 layers above, between* 30 *and* 70*.*



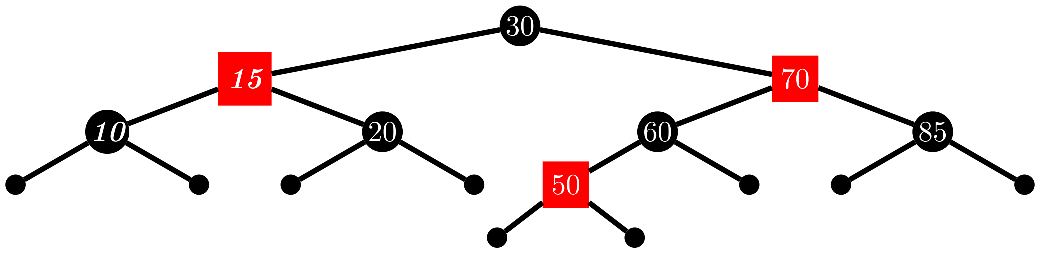
This is Case 2.2.1 - Black Uncle ZigZag > Rotation around parent. Since 70 *and* 30 *do not line up on the same side of the grandparent* (15)*. Rotation around the parent* (70) *is necessary,* 30 *rises up. Then we move on to execute Case 2.2.2 immediately.*



Now we are in Case 2.2.2 - Black Uncle Lineup > Rotation around grandparent. We rotate around the grandparent (15)*,* 30 *rises up.*



In the final step of Case 2.2.2, we recolor the former grandparent (15) *to red and the former parent* (30) *to black.*



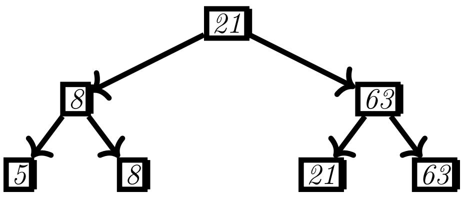
The black heights are equal.

95. Give a 2-3-tree that stores keys $5, 8, 21, 63$, then insert keys $69, 32, 7, 23, 25$ in this order.

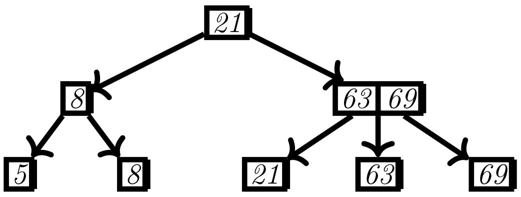
Solution:

In each step, we make sure that for each node, the length of the paths towards all leaves from it are equal, therefore keeping the tree balanced. All operations are formulated with this goal in mind.

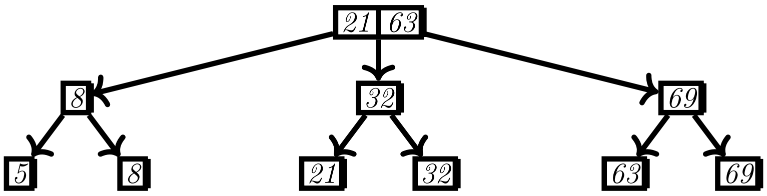
The initial 2-3 tree looks like this. 4 *elements can only be arranged in* 2 *layers in* 2*-nodes.*



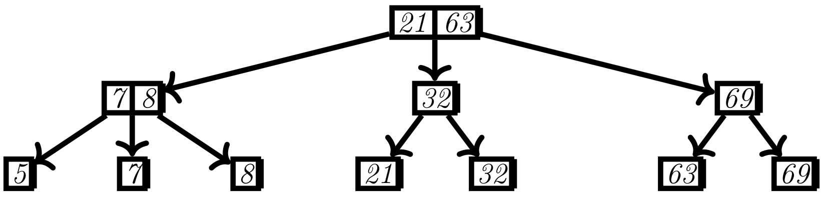
69 *is inserted, the node becomes a* 3*-node.*



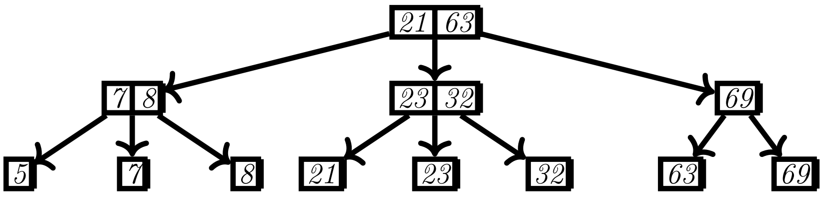
32 *is inserted, the* 3*-node splits into two. The root node becomes a* 3*-node as a result.*



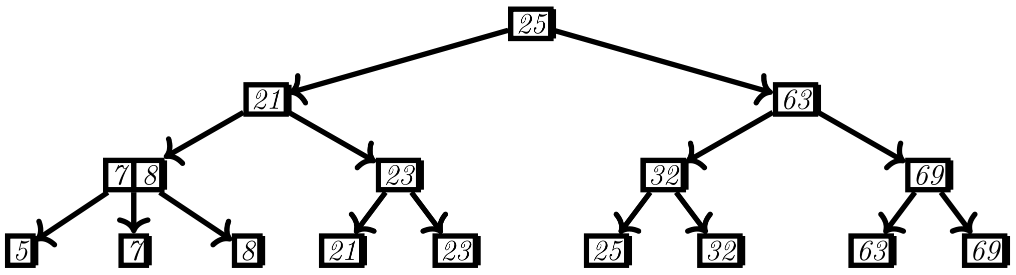
7 *is inserted, the node becomes a* 3*-node.*



23 *is inserted, the node becomes a* 3*-node.*



25 *is inserted, the* 3*-node splits into two* 2*-nodes. The root was a* 3*-node, it cannot store* 4 *elements, so it splits into* 2*. A new root node is created, the height increases by one.*



96. n integers are stored in binary search tree A, while binary search tree B stores m integers. Sort these $n + m$ elements in $O(n + m)$ time.

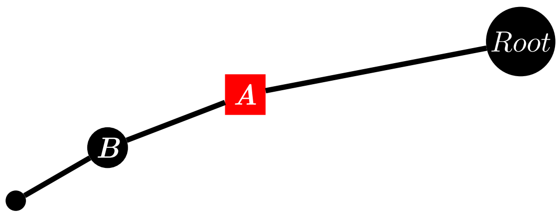
Solution:

We know that InOrder traversal provides the stored elements in increasing order. Let us read out the numbers stored in both trees this way and merge the two sorted lists, using the merge operation of MergeSort. The number of steps for reading (traversal) is proportional to the size of the tree, the merging requires $n + m - 1$ comparisons and $n + m$ moves, so the total number of steps is therefore $O(n + m)$.

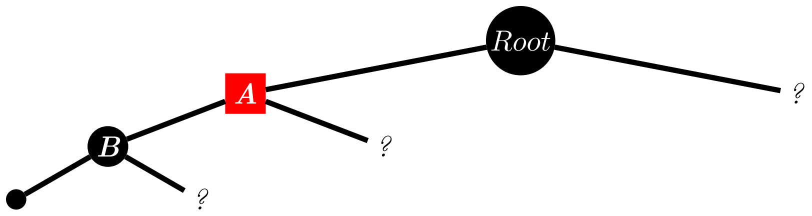
97. In a Red-Black tree the color of the vertices on some path from the root to a leaf are black,red,black,black, in this order. What is the smallest possible number of keys stored in this tree?

Solution:

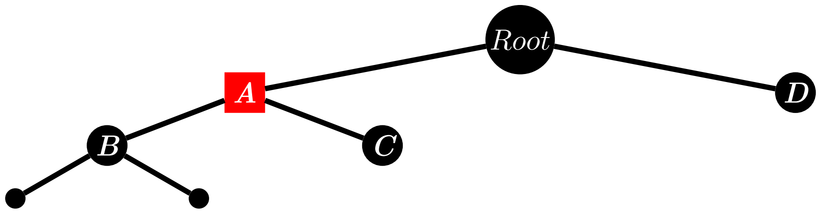
Without loss of generality we can assume the leftmost path was traversed, so this is what we saw:



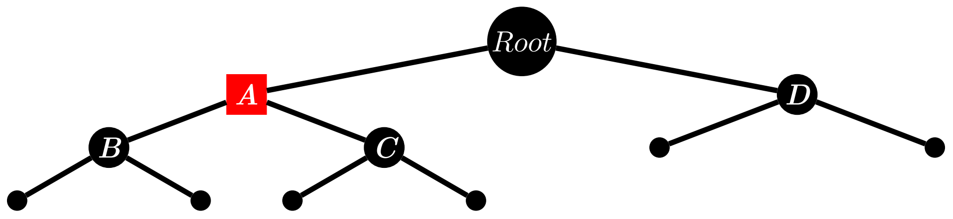
Since in a Red-Black tree each node that has at least one child must have exactly two, the Root, A and B must all have a right child.



The right child of B is also a (black) leaf, so that all black heights are equal from B. The right child of A must be black, since a red node cannot have a red child. The right child of Root could be either color, but the task is asking for smallest number of vertices, so it must be black (to reach equal black heights on both sides of the Root, the reds contribute nothing to it).



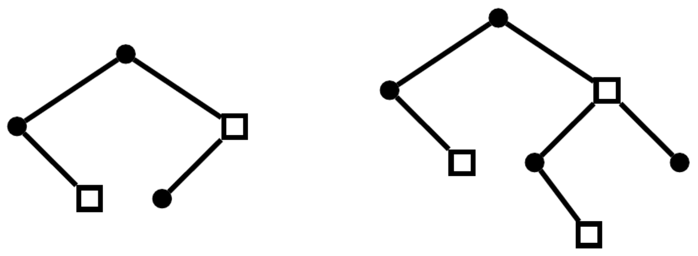
To have equal black heights from A, C cannot be a leaf, it must have children. Similarly, to have equal black heights from Root, D cannot be a leaf, it must have children.



And this Red-Black tree is minimal, all Red-Black tree properties are satisfied.

Hence, the minimal number of elements stored in the tree is 5 *(Root,A,B,C,D). The leaves are not storing elements in Red-Black trees.*

98. Is it possible that the figures below show the vertices of a red-black trees? The empty (NIL) leaves have not been drawn and a black circle indicates a black vertex while a white square indicates a red vertex.



Solution:

In the first case, no, because in the case of the red node on the right side, there are 2 black nodes on the path in one direction (including the leaf, not pictured), but only 1 black node in the other direction (the leaf, not pictured), making the black heights not equal.

The second case is possible, as all properties are satisfied: it is a binary tree, the root is black, the leaves (not pictured) are black, the black heights are equal, no red node has red children, each node is either a leaf or has two children (the leaves are not pictured).

99. What are the minimum and the maximum possible numbers of keys stored in a Red-Black tree whose black height is 3?

Solution:

We know that the number of non-leaf nodes is at least $2^3 - 1 = 7$*, so the tree stores at least this many elements. If the tree contains only black nodes, then it can store exactly 7 elements, making 7 the minimum.*

We also know that its height is at most $2 \cdot 3 = 6$*, when each black node is followed by a red (except the leaves), and a tree of height 6 can store at most* $2^6 - 1 = 63$ *elements. This can be achieved by alternating black and red nodes at each level.*

100. The root of a 2-3-tree has 3 children, the two keys in the root are 40 and 50. What could be the minimum and maximum number of the stored keys if it is known that the tree stores only positive integers?

Solution:

From the conditions, it follows that the root has 3 children: on the left, it stores elements between 1 and 39, in the middle between 40 and 49, and on the right, it stores elements from 50 and above.

The minimal number of elements is obviously obtained if the children of the root are leaves, storing elements x, 40, and 50, where $1 \leq x \leq 39$ is any integer.

To find the maximum number of elements, let's consider what we can say about the height of the tree! Since in this case, the middle subtree can store the least amount of elements among the three subtrees, it will be the limiting factor on the tree height. Its height is at most 4, and to store a maximum of 10 elements, this height is necessary. Thus, the heights of the other two subtrees are also 4. To maximize the number of leaves (stored elements), each subtree branches into 3 parts, storing 27 elements each. This is the allowed number of elements on both sides, so the maximum number of elements is $10 + 2 \cdot 27 = 64$.

---

101. Let x and y denote the two children of the root of a Red-Black tree. It is known, that $bh(x) = bh(y)$, but the the black heights of the two children of x are different. What is the color of node y?

Solution:

Since the node itself is not counted into the black height, the colors of x and y must be the same. In case of red, the root's black height is $bh(root) = bh(x) = bh(y)$. In case of black, the root's black height is $bh(root) = bh(x) + 1 = bh(y) + 1$.

If we name the two children of x: a and b, if their black heights are different $bh(a) \neq bh(b)$, it means that either one of them is exactly 1 larger than the other. Let's name the larger one a and the smaller one b. $bh(a) = bh(b) + 1$. To make the black heights equal at x, this can only happen if a is red and b is black.

Since x has a red child, it cannot be red itself, it must be black. Since y is the same color, it must also be black.

102. Is it possible that the PreOrder of the elements of a red-black tree is : $6, 1, 5, 3, 2, 4$?

Solution:

This means that 6 is the root. Since all elements are less than 6, due to the search tree property, the right subtree of 6 must not contain any of them and the left subtree contains the elements $1, 5, 3, 2, 4$.

So the right subtree of 6 is a single black leaf, therefore the black height of the root is 1. If the left child of the root is black, it must also be a leaf. If the left child of the root is red, its children must be black (since a red node cannot have red children) and they must be leaves as well (as the black heights would not be equal otherwise). Neither case has enough storage capacity for 5 elements.

This is a contradiction, it is not possible.

103. We are given a binary search tree with n vertices, all the values of which are different. For each vertex v of this tree, we want to determine how many elements in the subtree with root v are smaller than v. Give an algorithm that does this in $O(n)$ steps.

Solution:

In a search tree, elements smaller than v can only be in the left subtree of the node v. Our task is to determine the size of the left subtree for each node.

Let us use dynamic programming to accomplish this task!

To calculate the size of a left subtree, we must calculate the size of the entire subtree of our left child. The size of the entire subtree contains the right subtree of that child as well, so we will need this extra information to perform our calculations.

Therefore, let us define two arrays, $L[v]$ and $R[v]$, meaning the size of the left and the right subtrees of v, respectively, for all v vertices of the tree.

When v is not a leaf:

The left subtree of v is the size of the left and right subtrees of the left child of v plus one, for the left child itself, we can see that

$$L[v] = L[\mathit{left\_child}(v)] + R[\mathit{left\_child}(v)] + 1.$$

With similar logic

$$R[v] = L[\mathit{right\_child}(v)] + R[\mathit{right\_child}(v)] + 1.$$

When v is a leaf $L[v] = R[v] = 0$.

If we traverse our tree in PostOrder traversal order to calculate these, all subtrees of a specific vertex will be visited and calculated before the vertex itself is visited, so when the calculations are performed the components will be already available.

Finally, the result is the array $L[v]$.

The number of steps: the traversal is linear, and for each node, we perform a constant number of additional operations, so the total complexity is $O(n)$.

104. A 2-3 tree contains all integers in the interval $[1, 178]$. The root is a 3-node and the first waypoint key it contains is 17. What can be the other one?

Solution:

The second waypoint indicates the smallest number stored in the right subtree, we need to determine this.

Due to the first waypoint, the numbers stored in the left subtree range from 1 to 16.

We know that when the tree stores n nodes (which are stored in the leaves for 2-3 trees) on l levels, then

$$2^{l-1} \le n \le 3^{l-1},$$

or equivalently

$$\log_3 n + 1 \le l \le \log_2 n + 1.$$

(Pay attention to the base-2 and base-3 logarithm switching sides.)

Using the second form, plugging in $n = 16$, for the number of elements in the left subtree we get

$$\log_3 16 + 1 \le l \le \log_2 16 + 1$$

which is calculated as

$$3.524 \le l \le 5.$$

Since the levels must be integer, we get

$$4 \le l \le 5.$$

Therefore, the height of the left subtree is at least 4, and at most 5.

Since we have two waypoints, the root has 3 children. We know that the height of all three subtrees is the same. If this common height is 4, then there are at most $3^{4-1} = 3^3 = 27$ *leaves in both the middle and right subtrees combined. Hence, they can store at most this many values, so all* 178 *elements cannot fit into the tree.*

Therefore, the height of the left subtree, and consequently the other two, must be 5*. In this case, each of the other two subtrees can store at most* $3^{5-1} = 3^4 = 81$ *values. Since* $16 + 2 \cdot 81 = 178$*, it follows that each of the other two subtrees are storing* ***exactly*** *as much as they can possibly store,* 81*.*

At this point, the tree has been uniquely identified.

Every non-leaf node in the middle and right subtrees is therefore at maximum capacity, having exactly 3 *children (while in the left subtree, they all have* 2*), allowing the middle and right subtrees to store* 81 *elements each exactly.*

The tree itself stores all elements between $1-16$ *in the left subtree, all elements between* $(16+1)-(16+81) = 17-97$ *in the middle subtree and all elements between* $(97+1)-(97+81) = 98-178$ *in the right subtree.*

Therefore 98 *is the second waypoint, the smallest element in the right subtree.*

105. A 2-3-tree stores 4 keys. Is the tree unique?

Solution:

Using the formula for the number of levels l *and the number of leaves* n

$$\log_3 n + 1 \leq l \leq \log_2 n + 1$$

substituting $n = 4$ *for the number of elements stored (in the leaves)*

$$\log_3 4 + 1 \leq l \leq \log_2 4 + 1$$

we get

$$2.2618 \leq l \leq 3$$

however l *must be integer, therefore equivalently*

$$3 \leq l \leq 3,$$

so $l = 3$*.*

The tree has precisely 3 *levels. A three-level tree has exactly* 4 *leaves, if each node has exactly* 2 *children. If any node had 3 children, there would be at least* 5 *leaves. Therefore, the tree is a complete binary tree, which is unique.*

Since we also know that the elements grow from left to right in the leaves, and the labels of the internal nodes can be uniquely determined from the leaves, the 2-3 tree is unique.

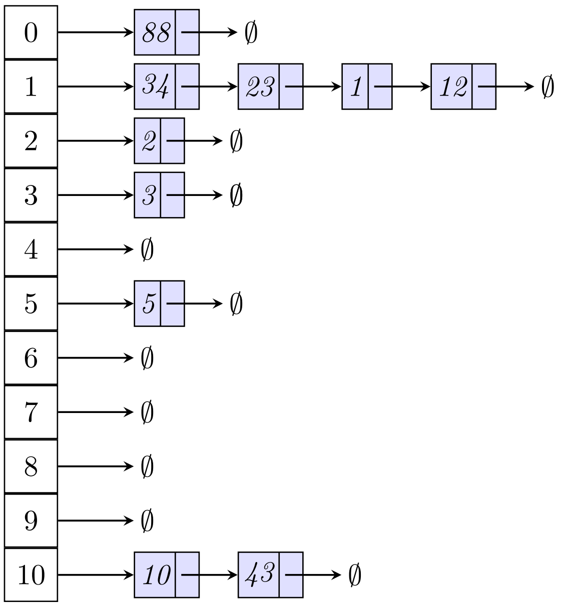
106. We want to store items in a hash table using the hash function $h(k) = k \pmod{11}$, and we will resolve collisions with chaining.
(a) Insert the following elements into an empty hash table: $2, 5, 12, 1, 3, 88, 23, 43, 10, 34$.
(b) Demonstrate how SEARCH(1), SEARCH(20) and SEARCH(45) will work in the resulting table.
(c) Delete the element 23.

Solution:

(a)

The table will look like this after the insertions:

Each collision is resolved by inserting the incoming element at the start of the linked list. The $\emptyset$ *represents a null pointer.*



(b)

SEARCH(1)*: Will linear search the linked list at index* 1 *and find the element* 1 *after* 3 *steps.*
SEARCH(20)*: Will check the linked list at index* 9 *and see it's empty. Therefore* 20 *is not in the hash table.*
SEARCH(45)*: Will linear search the linked list at index* 1*. After reaching the end, it reports that* 45 *is not found in the tabe.*

(c)

DELETE(23)*: First we execute* ***SEARCH(23)****, which will linear search the linked list at index* 1*. During linear search, in each step, we always keep track of the previous element too. When we find the element* 23 *in the second cell we edit the pointer of the previous element (*34*) to point to the next element instead (*1*). Then we destroy the cell containing* 23*.*

107. We want to store items in an open-address hash table using the hash function $h(k) = k \pmod{11}$. After the insertion of $4, 5, 14, 15, 16, 26, 3$, what will the resulting table look like and what are the number of collisions at each insert if collisions were resolved using:
(a) Linear probe (b) Quadratic probe
(c) Double hashing where our second hash function is $h'(x) = 7x \pmod{(10)} + 1$?

Solution:

(a)

The probing sequence is: $-1, -2, -3, -4, \ldots$
The number of collisions are: $0 + 0 + 0 + 2 + 4 + 4 + 4 = 14$.
The table at the end is:

0	1	2	3	4	5	6	7	8	9	10
26	16	15	14	4	5					3

(b)

The probing sequence is: $+1, -1, +4, -4, +9, -9, +16, \ldots$
The number of collisions are: $0 + 0 + 0 + 3 + 1 + 4 + 2 = 10$.
The table at the end is:

0	1	2	3	4	5	6	7	8	9	10
26		3	14	4	5	16		15		

(c)

The probing sequence is: $-h'(x), -2h'(x), -3h'(x), -4h'(x) \ldots$
The number of collisions are: $0 + 0 + 0 + 1 + 1 + 1 + 2 = 5$.
The table at the end is:

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
	26	*16*	*14*	*4*	*5*				*15*	*3*

108. The edges of an undirected graph are as follows:
$ab(2), ac(3), bd(2), bc(2), cd(1), de(2), df(4), ef(1), eg(2), fg(2), fh(1), gh(3)$.
What are the edges selected by the Kruskal's algorithm and in which order? To determine this, use the UNION-FIND data structure.

Solution:

(Note: An extra edge has been added to the graph, relative to how it appeared on the Practice Session originally, so that all possible cases can be presented below.)

Kruskal is an algorithm that can find a ***minimal weight spanning tree*** *in a weighted, undirected graph in* $\mathbf{O(m_G \log n_G)}$ *time, where n_G is the number of vertices and m_G is the number of edges of the graph G.*

The first step is ordering the edges of the graph in increasing order of weights:
$cd(1), ef(1), fh(1), ab(2), bd(2), bc(2), de(2), eg(2), fg(2), ac(3), gh(3), df(4)$

This sorting happens in $O(m_G \log m_G) \subseteq O(m_G \log n_G^2) = O(m_G \cdot 2 \log n_G) = O(m_G \log n_G)$.

Then we iterate this sorted list, building up the spanning forest of the graph. We keep track of the currently connected components of the current spanning forest using the UNION-FIND data structure. If the current edge connects two previously unconnected components we include it, if it runs inside an already connected component we discard it (otherwise, it would create a cycle).

You may notice that this is a greedy algorithm, as it makes local choices in each step, without having to consider how that affects later steps. It can be proven, that the structure of this problem is such, that the greedy algorithm in fact will find the optimal solution. Beware, that ***this is not true in general for greedy algorithms****, the underlying mathematics are incredibly beautiful, and lead to Matroid theory with applications ranging from statics in building construction to Kirchhoff's Current Law in electronic circuit theory.*

The UNION-FIND data structure keeps track of the sets of vertices, that are currently connected assigning a representing vertex to each set and pointing a directed path towards the representing vertex. Two vertices are currently in the same set if and only if they point towards the same representing vertex via a directed path.

The FIND operation traverses the directed path towards the representing vertex and returns it.

To merge two components together we first FIND their representative vertices, then UNION them by pointing one of the representats towards the other. This way all vertices in both sets now point (indirectly) towards the same representant, effectively merging the two sets into one. If we did this naively, we may end up with a single very long path, but if we always point the smaller set's representant towards the bigger one, we can prove that the length of the resulting paths will be $O(\log(n_G))$ on average.

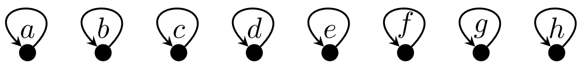
Due to this, $FIND$ takes $O(\log(n_G))$ time and while $UNION$ itself takes $O(1)$, it usually involves two FIND operations beforehand, to find the representants to be unioned. We run it m_G times, for a total cost of $O(m_G \log n_G)$. The initial sorting step is also $O(m_G \log n_G)$, so the entire algorithm runs in $O(m_G \log n_G)$.

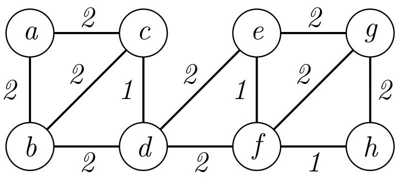
Below, on the left hand side you'll see how the UNION-FIND data structure evolves during the algorithm's run and on the right hand side you'll see the corresponding spanning forest in the graph evolving.

The UNION-FIND data structure, although is a graph too, does not directly correspond to the edges of the original graph, ***i.e. for edge $u - v$, do not expect the new edge in the UNION-FIND data structure to appear between $u - v$ (sometimes it does, in special cases, but do not get confused by this)****. Instead, UNION-FIND represents the spanning forest's connected components via a shared representing vertex at the end of the directed path from the vertices of the graph.*

- *Initially, each vertex is isolated (represents itself) and no edges have been added to the spanning forest yet.*

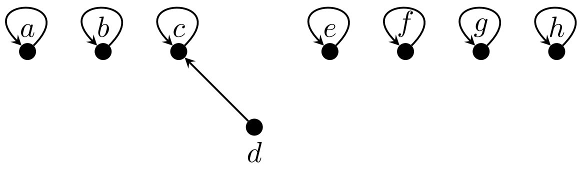
$cd(1), ef(1), fh(1), ab(2), bd(2), bc(2), de(2), eg(2), fg(2), ac(3), gh(3), df(4)$

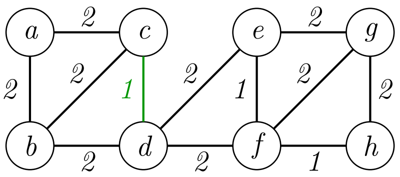




- *The first edge we consider is* $cd(1)$.
- *We first check the endpoints* c *and* d, *if they are in the same component already.*
- *Executing* $FIND(c)$ *it reports that its component's representing vertex is* c.
- *Executing* $FIND(d)$ *it reports that its component's representing vertex is* d.
- *Since* $FIND(c) \neq FIND(d)$, *it means that edge* $cd(1)$ *connects two previously disconnected components, therefore we should add it to the spanning forest.*
- *Since the components of* c *and* d *are now connected, we execute* $UNION(c, d)$, *to keep track of this fact.*
- *Since both of the sets are size* 1, *we can choose which one to point to the other, we choose* d *to point to* c.
- *Now the representant of both vertices* c *and* d *is* c, *therefore they are now in the same connected component set.*

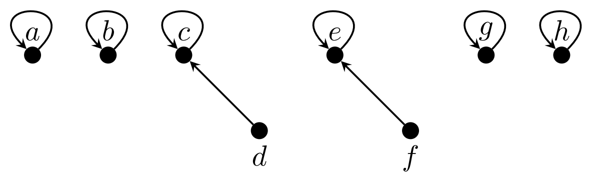
$cd(1), ef(1), fh(1), ab(2), bd(2), bc(2), de(2), eg(2), fg(2), ac(3), gh(3), df(4)$

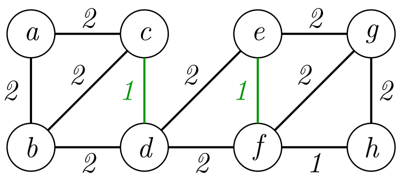




- *The next edge we consider is* $ef(1)$.
- *We first check the endpoints* e *and* f, *if they are in the same component already.*
- *Executing* $FIND(e)$ *it reports that its representing vertex is* e.
- *Executing* $FIND(f)$ *it reports that its representing vertex is* f.
- *Since* $FIND(e) \neq FIND(f)$, *it means that edge* $ef(1)$ *connects two previously disconnected components, therefore we should add it to the spanning forest.*
- *Since the components of* e *and* f *are now connected, we execute* $UNION(e, f)$, *to keep track of this fact.*
- *Since both of the sets are size* 1, *we can choose which one to point to the other, we choose* f *to point to* e.
- *Now the representant of both vertices* e *and* f *is* e, *therefore they are now in the same connected component set.*

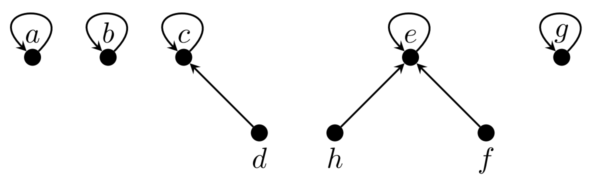
$cd(1), ef(1), fh(1), ab(2), bd(2), bc(2), de(2), eg(2), fg(2), ac(3), gh(3), df(4)$

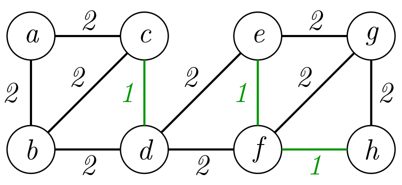




- *The next edge we consider is* $fh(1)$.
- *We first check the endpoints* f *and* h, *if they are in the same component already.*
- *Executing* $FIND(f)$ *it traverses the path* $f \rightarrow e$ *in the UNION-FIND data structure and reports that* f*'s representing vertex is* e.
- *Executing* $FIND(h)$ *it reports that its representing vertex is* h.
- *Since* $FIND(f) \neq FIND(h)$, *it means that edge* $fh(1)$ *connects two previously disconnected components, therefore we should add it to the spanning forest.*
- *Since the components represented by* e *and* h *are now connected, we execute* $UNION(e, h)$, *to keep track of this fact.*
- *Since the component of* e *is size* 2, *while the component of* h *is size* 1, *we must point* h *towards* e, *to minimize the length of the paths appearing. (Pointing* e *towards* h *would result in a path length of* 2*!)*
- *Notice here that due to the graph edge* $fh(1)$ *in the UNION-FIND dataset a* ***directed edge*** $h \rightarrow e$ *appeared!*
- *Now the representant of all three vertices* e, f *and* h *is* e, *therefore they are now in the same connected component set.*

$cd(1), ef(1), fh(1), ab(2), bd(2), bc(2), de(2), eg(2), fg(2), ac(3), gh(3), df(4)$

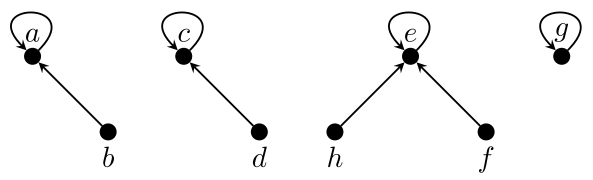


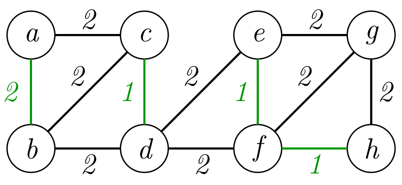


- *The next edge we consider is* $ab(2)$.
- *We first check the endpoints* a *and* b, *if they are in the same component already.*
- *Executing* $FIND(a)$ *it reports that its representing vertex is* a.
- *Executing* $FIND(b)$ *it reports that its representing vertex is* b.
- *Since* $FIND(a) \neq FIND(b)$, *it means that edge* $ab(2)$ *connects two previously disconnected components, therefore we should add it to the spanning forest.*
- *Since the components represented by* a *and* b *are now connected, we execute* $UNION(a, b)$, *to keep track of this fact.*
- *Since both of the sets are size* 1, *we can choose which one to point to the other, we choose* b *to point to* a.

- *Now the representant of both vertices a and b is a, therefore they are now in the same connected component set.*

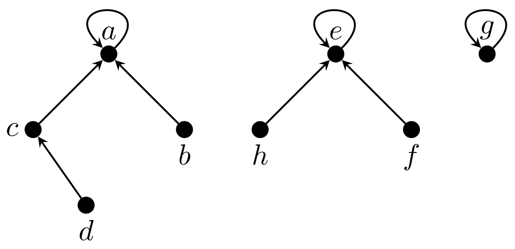
$cd(1), ef(1), fh(1), ab(2), bd(2), bc(2), de(2), eg(2), fg(2), ac(3), gh(3), df(4)$

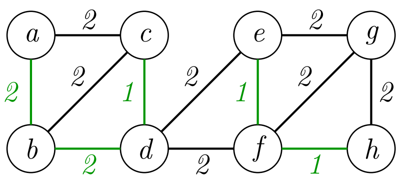




- *The next edge we consider is bd(2).*
- *We first check the endpoints b and d, if they are in the same component already.*
- *Executing $FIND(b)$ it traverses the path $b \to a$ in the UNION-FIND data structure and reports that b's representing vertex is a.*
- *Executing $FIND(d)$ it traverses the path $d \to c$ in the UNION-FIND data structure and reports that d's representing vertex is c.*
- *Since $FIND(b) \neq FIND(d)$, it means that edge bd(2) connects two previously disconnected components, therefore we should add it to the spanning forest.*
- *Since the components represented by a and c are now connected, we execute $UNION(a, c)$, to keep track of this fact.*
- *Since both of the sets are size 2, we can choose which one to point to the other, we choose c to point to a.*
- *Notice here that due to the graph edge bd(2) in the UNION-FIND dataset a* ***directed edge*** *$c \to a$ appeared!*
- *Now the representant of vertices a, b, c, d is a, therefore they are now in the same connected component set.*

$cd(1), ef(1), fh(1), ab(2), bd(2), bc(2), de(2), eg(2), fg(2), ac(3), gh(3), df(4)$

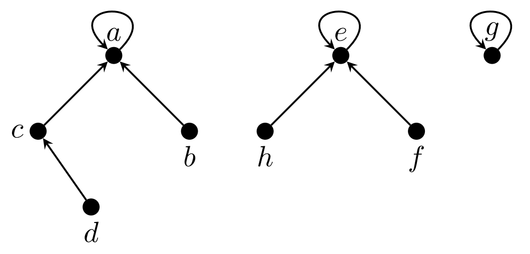


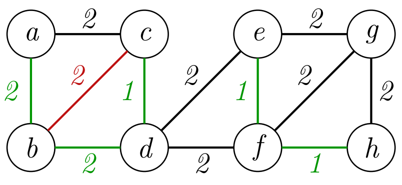


- *The next edge we consider is bc(2).*
- *We first check the endpoints b and c, if they are in the same component already.*
- *Executing $FIND(b)$ it traverses the path $b \to a$ in the UNION-FIND data structure and reports that b's representing vertex is a.*
- *Executing $FIND(c)$ it traverses the path $c \to a$ in the UNION-FIND data structure and reports that c's representing vertex is a.*

- *Since $FIND(b) = FIND(c)$, it means that edge $bd(2)$ runs inside a single component, its endpoints b and c are already connected via a path, therefore adding this to the spanning forest would create a cycle.*
- *So edge $bc(2)$ is discarded, the UNION-FIND dataset is not modified.*

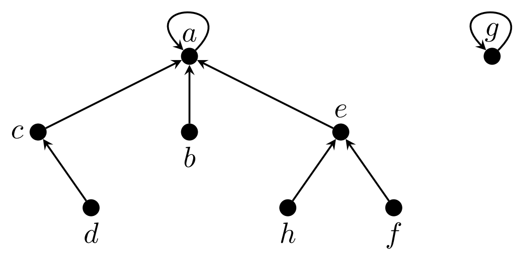
$$cd(1), ef(1), fh(1), ab(2), bd(2), bc(2), de(2), eg(2), fg(2), ac(3), gh(3), df(4)$$

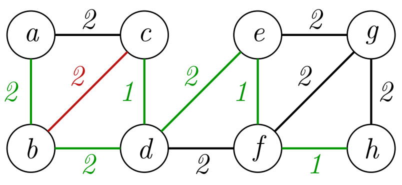




- *The next edge we consider is $de(2)$.*
- *We first check the endpoints d and e, if they are in the same component already.*
- *Executing $FIND(d)$ it traverses the path $d \to c \to a$ in the UNION-FIND data structure and reports that d's representing vertex is a.*
- *Executing $FIND(e)$ it reports that its representing vertex is e.*
- *Since $FIND(d) \neq FIND(e)$, it means that edge $de(2)$ connects two previously disconnected components, therefore we should add it to the spanning forest.*
- *Since the components represented by a and e are now connected, we execute $UNION(a, e)$, to keep track of this fact.*
- *Since the component of a is size 4, while the component of e is size 3, we must point e towards a, to minimize the length of the paths appearing. (Pointing a towards e would result in a path length of 3, while doing it our way, we only get paths of length at most 2!)*
- *Notice here that due to the graph edge $de(2)$ in the UNION-FIND dataset a* ***directed edge*** *$e \to a$ appeared!*
- *Now the representant of vertices a, b, c, d, e, f, h is a, therefore they are now in the same connected component set.*

$$cd(1), ef(1), fh(1), ab(2), bd(2), bc(2), de(2), eg(2), fg(2), ac(3), gh(3), df(4)$$

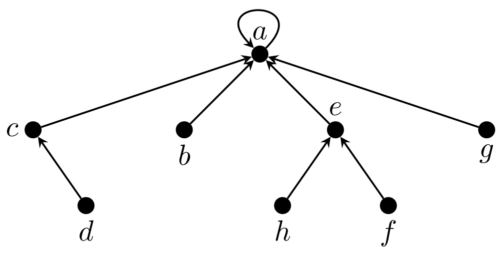


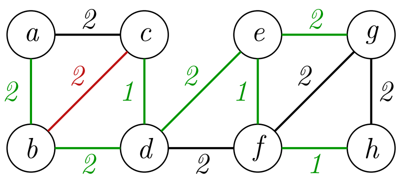


- *The next edge we consider is $eg(2)$.*
- *We first check the endpoints e and g, if they are in the same component already.*

- *Executing $FIND(e)$ it traverses the path $e \to a$ in the UNION-FIND data structure and reports that d's representing vertex is a.*
- *Executing $FIND(g)$ it reports that its representing vertex is g.*
- *Since $FIND(e) \neq FIND(g)$, it means that edge $eg(2)$ connects two previously disconnected components, therefore we should add it to the spanning forest.*
- *Since the components represented by a and g are now connected, we execute $UNION(a, g)$, to keep track of this fact.*
- *Since the component of a is size 7, while the component of g is size 1, we must point g towards a, to minimize the length of the paths appearing. (Pointing a towards g would result in a path length of 3, while doing it our way, we only get paths of length at most 2!)*
- *Notice here that due to the graph edge $eg(2)$ in the UNION-FIND dataset a* ***directed edge*** *$g \to a$ appeared!*
- *Now the representant of every vertex is a, therefore our graph is connected and our spanning tree is complete!*

$cd(1), ef(1), fh(1), ab(2), bd(2), bc(2), de(2), eg(2), fg(2), ac(3), gh(3), df(4)$

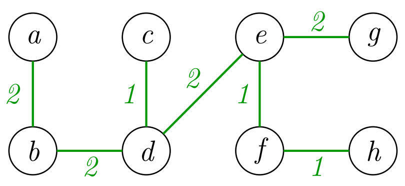




The rest of the edges can be skipped, since our minimal spanning tree is complete.

Our final result, a minimal spanning tree of the graph are the green edges:

Note, that other spanning trees with the same weight are possible (try to find one), hence we say 'a minimal' and not 'the minimum'.



Note here that although the UNION-FIND data structure at the end also happens to resemble a tree, the edges of that do not directly correspond to the edges of a spanning tree in the graph.

---

109. Can it happen that in an open-address hash table of size $n > 3$ with exactly 3 elements in it, but a search takes $O(n)$ steps?

Solution:

Yes, if there were a lot of deletions earlier, it can happen. For example if every single cell that does not contain an element contains the $*$ sign, denoting that something was deleted from there, the search procedure will continue when encountering such an element, therefore the entire table will be traversed.

In practical implementations, when a hash table reaches a state where there are many deleted, but otherwise empty cells, it becomes inefficient to search in it. In such cases, a common approach is to rehash the elements into a new empty table. This process involves reading the contents of the existing table and reinserting them into the new one. We do not discuss the exact techniques for this.

110. For a hash table of size $M = 7$, is the following a good hash function: $h(x) = x^2 \pmod 7$?

Solution:

No. To check, we only have to go through the hash of the possible remainders of 7*:*

- $h(0) = 0 \pmod 7 = 0$
- $h(1) = 1 \pmod 7 = 1$
- $h(2) = 4 \pmod 7 = 4$
- $h(3) = 9 \pmod 7 = 2$
- $h(4) = 16 \pmod 7 = 2$
- $h(5) = 25 \pmod 7 = 4$
- $h(6) = 36 \pmod 7 = 1$

Notice that in general, the hash function is $h(7y + z) = (7y + z)^2 \pmod 7 = 7(7y^2 + 2z) + z^2 \pmod 7 = z^2 \pmod 7 = h(z)$*, hence why only the remainders are checked.*

Since $h(1) = h(6)$*,* $h(2) = h(5)$ *and* $h(3) = h(4)$*, so only the cells with indices* $0, 1, 2, 4$ *are used as possible starting positions (altough the other cells can later be filled, but only after some collisions). It would be better to distribute the incoming keys "more evenly" among the* 7 *posible cells ot the table. This would result in likely less collisions in the average scenario.*

111. Given an open-address hash table of length 11 with hash function $h(x) = x \pmod{11}$ and using linear probe for collision resolution, we perform a few insertions and deletions and obtain the following table ($*$ denotes a deleted entry and not an empty one):

0	1	2	3	4	5	6	7	8	9	10
11	4	*	3	15		6	18			22

There is only one element in the above table that upon deletion and being inserted again, will end up with a new address in the hash table. What is this element? Show the deletion and insertion steps of it to support your answer.

Solution:

The answer is number 4 in the 1st cell.

On the one hand, we can check that all the other numbers in the table will be placed back to their current position after deletion and insertion.

On the other hand, the deletion of 4 starts at the 4th cell due to $h(4) = 4$*, from here we step backwards to the 3rd, 2nd, and then 1st cell, where we find and delete the number* 4 *and write* $*$ *in place of it. The insertion of* 4 *in the resulting table starts again with a search at the 4th cell, where a collision happens and with the linear probing sequence we continue to the* 3*rd cell, then the* 2*nd, where 4 can be written in place of* $*$*, a formerly deleted element.*

The reason why this happens is because 4 *was inserted after what was originally in place* 2 *was inserted but before it was deleted.*

This is why it's important to mark the deleted cell with a $*$, so that we do not lose track of elements that collided with it when they were inserted.

112. A big film production wants to shoot a film in a city. A map of the city is given as an undirected graph, where the vertices are the nodes and the edges are the direct roads between them. The filmmakers would like to close as many roads as possible during the filming, but we charge money (greater than zero) to close each section, we know the specific amount for each road. (The filmmakers have money to close all roads.) Give an algorithm that determines which roads should be closed if we want the city to function during the closure (we need to be able to get from everywhere to everywhere) and we want to maximize our revenue. The running time of your algorithm should be $O(n^2)$, where n is the number of nodes.

Solution:

Since we (the city administration) aim to maximize revenue, we want to offer to close the streets that sum up to the highest possible cost. Therefore we want to keep the roads open that sum up to the smallest possible cost, while keeping the city connected.

This means, that we want to find a minimal spanning tree of our city, which will contain the edges that will keep the city connected while minimizing the total "unpaid" costs, therefore maximizing the total "paid" costs.

We can run the Kruskal algorithm on the undirected weighted graph to find a minimal spanning tree, then return all edges that don't belong to it.

The time complexity of the Kruskal algorithm is $O(m_G \log n_G)$, where n_G is the number of vertices and m_G is the number of edges in the graph.

113. Give an algorithm that determines a maximal weight spanning tree in an undirected, connected, edge weighted graph.

Solution:

Let the input graph be $G = (V, E, c)$, with n_G vertices, m_G edges, and the edge weights given by the function $c : E \to \mathbf{R}$.

Consider the function $c' = -c$ instead of c, and with this weight function c', determine a minimal cost spanning tree T in G. This will be a maximal weight spanning tree with respect to the original weight function.

When T is minimal with respect to c', then for any other possible spanning tree T_2 of G we know that $c'(T) \leq c'(T_2)$ and from this it follows that $-c(T) \leq -c(T_2)$, so $c(T) \geq c(T_2)$, which means that T is maximal with respect to c.

For computation, the Kruskal algorithm can be used, with a time complexity of $O(m_G \log n_G)$.

---

114. In table $T[0 : M]$ we have placed $2n$ elements ($n < M/3$) using some hash function, when we found that the elements were all placed in the first $3n$ positions. If there was no deletion in the meantime and in the resulting table all places with index $3i$ remained empty ($0 \leq i < n$), then there could have been at most how many collisions if (a) a linear probe was used? (b) a quadratic probe was used?

Solution:

(a) The resulting hash table looks like this (it is possibly longer to the right, denoted by ...):

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
	a_1	a_2		a_3	a_4		a_5	a_6		...

From each group of cells with indices $(3i), (3i+1), (3i+2)$, cell $(3i)$ is empty and cells $(3i+1), (3i+2)$ are filled. When using linear probing, upon collision we move to the left and we stop at an empty cell. Therefore collisions only happen with neighbouring elements to the right, so only the content of the cell $3i+1$ could have *collided at position $(3i+2)$. For each group of 3 cells, this results in at most 1 collision, for a total of at most n.*

This can in fact happen, for example when elements $(2), (M+2), (5), (M+5), \dots, (3i+2), (M+3i+2), \dots$ are inserted in this order.

(b) Since everyone has at least one empty neighbor and quadratic probing first checks the two neighbours, there could be at most one collision per group. Thus, there could still be at most n collisions

This can in fact happen, using the same example numbers as in (a), but now with quadratic probing.

115. A hash table of size m already has some elements, but the table is not full. Give an algorithm with running time $O(m)$ that determines the maximum number of collisions that can happen during the insertion of a new element using a linear probe to resolve collisions.

Solution:

The task is to determine the length of the longest occupied block in the table. This can be done simply iteratively. We initialize a counter to 0 and we start iterating the hash table from left-to-right. In each step, when we encounter a cell that is occupied, we increase the counter by 1 and when we encounter a cell that is empty, we reset the counter to 0. During our iterations, in each step, we keep track of the maximal value of the counter we have seen. At the end, we return this value.

116. We are given the adjacency list of a simple, connected graph $G = (V, E)$. The edges of the graph are weighted, the weight function $c : E \to \{-1, 1\}$. Give an algorithm that determines in $O(|V| + |E|)$ steps, the minimum total weight of a subgraph of G that is connected and contains all points of G.

Solution:

Unfortunately, Kruskal is too slow (it runs in $O(|E| \log |V|)$ time), so we cannot use it here.

However, there are only two types of weights, with value -1 and value 1. DFS (and BFS) can find spanning trees in unweighted graphs and its runtime fits into the limits, so we may be able to make it find a minimal spanning tree if we only have 2 types of weights to consider, with some extra logic.

Let us denote the number of positive edges in a minimal spanning tree with s_{pos} and the number of negative ones with s_{neg}. Since in a spanning tree we have exactly $n_G - 1$ edges, we know that $s_{pos} + s_{neg} = n_G - 1$. Furthermore, the total weight of the spanning tree is $s_{pos} - s_{neg}$.

The idea of the algorithm is to maximize s_{neg}, which in turn will also minimize s_{pos}, as their sum is constant.

This can be achieved by running DFS first on a graph that only consists of negative edges. This will find the largest possible connected components using just the negative edges. After this is complete, if we are left with more than one component, we might need to use some positive edges to finish connecting our tree.

Luckily we do not need to return which exact edges crete this minimal spanning tree, we are only interested in the weight sum. Therefore it is sufficient to find the value of s_{pos}.

From the equation $s_{pos} + s_{neg} = n_G - 1$, we have

$$s_{pos} = n_G - s_{neg} - 1$$

Here, if our negative spanning forest is already a tree, $s_{pos} = 0$ is true.

The positive edge connections needed are guaranteed to exist, since the original graph is connected, so we can just plug this into our total weight formula

$$s_{pos} - s_{neg} = n_G - 2s_{neg} - 1$$

and we know exactly the weight of a minimal spanning tree, without having to run any extra steps.

To summarize, our algorithm simply runs DFS on the graph of the negative edges, counts the edges of the resulting spanning forest, by counting the DFS parent array's non-null entries, denoted by s_{neg} and reports back the number $n_G - 2s_{neg} - 1$.

Time complexity wise, we first construct the adjacency list of the negative edges only, which we can do in $O(|V| + |E|)$, by simply iterating the original adjacency list and copying only the negative edges into a new one. Then, we run DFS on the resulting graph, count the number of edges in the spaning tree, denoted by s_{neg} and report back the value $n_G - 2s_{neg} - 1$, which takes $O(|V| + |E|)$ time for the DFS itself and $O(|V|)$ time for counting the resulting parent array's non-null entries.

117. We are given the adjacency matrix of the connected, weighted, undirected, graph of a city's road network: the vertices are the nodes, the edges are the direct roads between the nodes, and the weight of the edges indicates how many workers can clear the given section of road in 1 hour. After a night snow storm, we would like to know the minimum number of workers we need to hire in total, so that all the nodes of the city can be reached after 8am from the main square (which is a vertex in the graph) on roads clear of snow. Assume that the snowfall stopped and the workers can start the work at 7 am.
Which algorithm can be applied, if we want to get an answer in $O(n^2 \log n)$ running time, where n is the number of nodes? Describe why it can be applied and how.

Solution:

Let us apply Kruskal's algorithm to the graph of roads, where the weight of the edges represents the number of workers needed to clean that specific road segment. The output of the algorithm is a minimal spanning tree, which has the property that it is the minimal weight set of edges in a connected graph that make the vertices connected. Since the workers have just 1 hour to finish working, we will need to hire unique workers for each road segment, so the weight of the minimal spanning tree is the total number of workers needed.

The graph is given by its adjacency matrix, where the weights of the edges are written in the cells. We can sort these using merge sort in $O(n^2 \log n^2) = O(n^2 \log n)$ steps. Then, by selecting the edges in increasing order and using the UNION-FIND data structure, we can decide whether to add each new edge in $O(\log n)$ steps for each edge using two FINDs, and update the data structure in a constant number of steps if needed using a UNION. Since there are at most n^2 edges, this totals $O(n^2 \log n)$ steps, including the initial sorting.

118. Prove that the following two languages are in the class NP. Which of these is in P? Which of them is in coNP?
$P_1 = \{G | G$ is an undirected graph with a cycle on exactly 100 vertices. $\}$
$P_2 = \{(G, k) | k$ is a positive integer and G is an undirected graph with a cycle on exactly k vertices. $\}$

Solution:

For** P_1**:

$\mathbf{P_1 \in NP}$

A problem is in P if it has a polynomial algorithm for finding a solution. Roughly speaking, polynomial algorithms are considered "effective", while anything worse than that (exponential, factorial, etc.) are considered ineffective, only good for small enough inputs.

Our goal for the future is to find polynomial solutions to as many problems as we can. Some problems, we haven't been able to solve like this. In cases like this, some form of randomized (nondeterministic) solution can be viable. These are not guaranteed to find the best solution with 100% probability or they are not guaranteed to find the solution fast, but they can perform "good enough" in practice. For any randomized solver to work, we need an effective verification algorithm for a potential solution. These algorithms explore the possible solutions to a problem by some metaheuristic (see Drunktard's Walk, Simulated Annealing, Hill Climbing, Genetic Algorithm, etc) checking in each step if the currently held random solution is correct, or how far away it is from a correct solution, then randomly generating new ones that are (hopefully) closer to it.

Some problems don't even have an effective verification algorithm. If someone were to give us the solution to *such a problem, we would not be able to tell if it is correct or not. Think about the difference between someone telling us the solution to a Sudoku puzzle vs telling us the best next move in a game of Chess. If someone gives me a fully filled Sudoku table, it is easy to check if it is correct, just by checking each row, column and square if they contain unique numbers. If someone tells me the best next move in Chess, I can't verify if they are correct easily, as I would have to explore all possible future moves, of which there are exponentially many possibilities.*

We have much more hope for finding a polynomial solution to a problem, where at least we can verify a correct solution for ourselves, when we are given one, than we have for problems, where we can't even do that. And while a polynomial solution might not exist yet, we can still do away with a randomized alternative, that works "good enough" in practice. Hence, the class NP (nondeterministic polynomial) contains exactly these types of problems.

A language is in NP if it has an effective algorithm for checking a possible solution (more generally, a witness) to the problem. This is formalized via the witness theorem, so we will use this to prove that $P_1 \in NP$.

Imagine you and your friend are given a G graph via its adjacency matrix and you get into an argument on whether or not G contains a 100 *vertex cycle. Your friend assures you it does, but you don't believe them. What proof can you ask from your friend? You can tell them to point out the cycle in question to you: this will be your witness (denoted by t). However, you don't believe them, so you must check for yourself if they are lying to you by trying to fake a good solution: this will be your witness checking algorithm (denoted by* τ*).*

Since the witness checking algorithm must be effective, it has to run in polynomial time relative to its input. Since its input contains the witness itself, the witness must be polynomial in size, relative to the original input in question. We'll check these during our proof.

We first specify what format your friend should give the witness to you. In this case, the cycle can be represented by listing the indices of the vertices in order they appear in the cycle: $v_1, v_2, v_3, \dots, v_x$.

Notice that we did not trust your friend to give us exactly 100 *vertices. It is part of the checking algorithm to make sure that is the case.*

What is the size of the witness? We are given x *numbers, denoting vertex indices in our graph* G*. What is the possible range of a single number? If* G *has* n_G *vertices, then* $1 \leq v_i \leq n_G$ *for all* v_i *in the witness. The number* n_G *can be represented in binary format on* $\log_2(n)$ *bits, which is the upper limit on the width of our numbers. Therefore a valid witness (which has exactly* 100 *vertices) has a size* $O(100 \log_2(n_G)) = O(\log_2(n_G))$*.*

In the case of an invalid witness, the size can be arbitrarily large, our friend could give us any garbage, or the full text of Odyssey by Homer, or their favourite chocolate cake recipe. If our checking algorithm notices, that the witness is too large to be valid, or it has other formatting issues, it can immediately report that and exit. This is the case, for example when $100 < x$ *or when the witness contains something other than a list of numbers.*

Now let's describe the witness checking algorithm:

- *We first check, that we have been given exactly* $x = 100$ *numbers on our input. We can do this by counting them. If we notice* x *to be at least* 101*, we report that failure and exit. This is done in* $O(101) = O(1)$ *steps.*
- *We then check, that the numbers given on our input are indeed vertices that exist in the graph. This requires a range check, for all* v_i *we check* $1 \leq v_i \leq n_G$*. These numbers are in binary format and we use the* $\leq$ *comparison operator, which operates bit-by-bit from the most significant position to the least, until it finds a differing bit. This will be* 200 *comparisons, each taking* $O(\log_2(n_G))$ *steps for the bits in the numbers, for a total of* $O(200 \log_2(n_G)) = O(\log_2(n_G))$ *steps.*
- *Finally, we must check if the edges exist between all adjacent pairs of vertices in the witness. This will be* 100 *lookups in our 2D adjacency matrix of* G*, for the edges* $v_1 - v_2$*,* $v_2 - v_3$*, …,* $v_{99} - v_{100}$ *and don't forget* $v_{100} - v_1$*, since it's a cycle. This takes* $O(100) = O(1)$ *steps for all edges.*

If you have checked all of these, you can be sure, that your friend did not trick you, they indeed pointed out a 100 cycle in this graph. So now, you can believe them.

What was the runtime of this checking algorithm? Totaling the individual steps, we had $O(1) + O(\log_2(n_G)) + O(1) = O(\log_2(n_G))$. The size of the original input, G's adjacency matrix is $O(n_G^2)$. If we denote the size of the input with n, then $n = n_G^2$. The size of the witness is $O(\log_2(n_G))$, which is polynomial relative to the input size n, since $n_G = \sqrt{n}$, so $O(\log_2(n_G)) = O(\log_2(\sqrt{n})) = O(\log_2(n^{\frac{1}{2}})) = O(\frac{1}{2}\log_2(n)) = O(\log_2(n))$. The witness checking algorithm also runs in $O(\log_2(n_G))$, with the same logic, that is also $O(\log_2(n))$, hence polynomial.

Finally, we should make sure that our checking algorithm is correct, as in:

- *there always exists a witness that will pass our checks if G contains such a 100-cycle*
- *no witness exists that could fool our checks if G does not contain such a 100-cycle.*

In the current case, this is a trivial observation, since the 100-cycle itself is the witness. In later exercises, we'll see examples of non-trivial cases.

To summarize the explanation above, focusing on what must be written down for a proper solution:

To prove that $P_1 \in NP$, we use the witness theorem.

Suppose that the input G is given by its adjacency matrix. If G has n_G vertices, the size of the original input is $O(n_G^2)$.

The witness is a description of such a 100-cycle, given by the list of vertices, in order they appear in the cycle.

t (Witness): $v_1, v_2, \ldots, v_x$

The size of the witness is $O(x\log_2(n_G))$, in case of a valid witness, with $x = 100$, that is $O(\log_2(n_G))$. This is polynomial relative to the input size, $O(n_G^2)$.

τ (Witness Checking Algorithm):

- *Count that $x = 100$. Done in $O(101) = O(1)$.*
- *Check that all vertices are valid with a range check on all v_i: $1 \leq v_i \leq n_G$. Done in $O(100\log_2(n_G)) = O(\log_2(n_G))$.*
- *Check that all of the edges $v_1 - v_2$, $v_2 - v_3$, $\ldots$, $v_{99} - v_{100}$, $v_{100} - v_1$ exist in the graph. Done in $O(100) = O(1)$.*

The runtime of the witness checking algorithm is $O(\log_2(n_G))$, which is polynomial relative to the input size, $O(n_G^2)$.

A witness always exists if G contains a 100 cycle, and does not exist, when G does not contain such a cycle, since that cycle is the witness.

$\mathbf{P_1 \in P}$

Is P_1 in P? In other words, do we need our friend telling us the solution, or can we find it ourselves?

Interestingly, we can actually solve P_1 effectively.

Let's iterate all possible subsets of 100 vertices in the graph! There are exactly $\binom{n_G}{100}$ of them. Then, let's try all possible orderings of those vertices. For a specific set, there are 100! orderings. Finally, let's check if that is a cycle, by checking the existence of all 100 necessary edges in the graph. This runs in

$$O(\binom{n_G}{100} \cdot 100! \cdot 100) = O(\binom{n_G}{100})$$

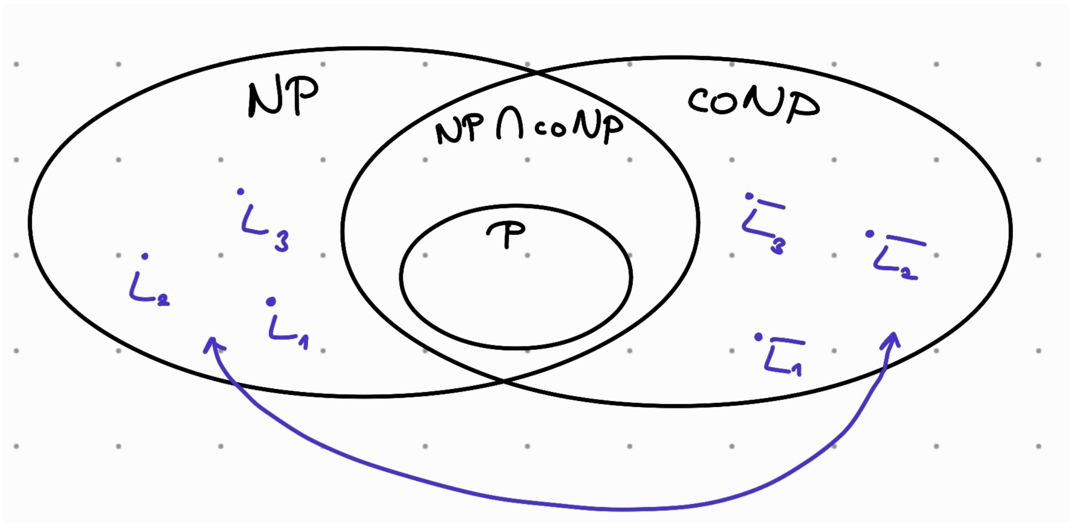
steps. Since

$$\binom{n_G}{100} = \frac{n_G \cdot (n_G-1) \cdot (n_G-2) \cdot \ldots \cdot (n_G-99)}{100 \cdot 99 \cdot 98 \cdot \ldots \cdot 1} \leq n_G^{100} = O(n_G^{100})$$

, the runtime is $O(n_G^{100})$, which is polynomial relative to $O(n_G^2)$, the input size.

$\mathbf{P_1} \in$ **coNP**

coNP is the set of languages who's complementer is in NP. ***The co- prefix** <u>**does not mean**</u> **that** coNP **would be the complementer of** NP.* *The co- prefix is always used to denote a set, which contains the* ***complementers of the <u>elements</u>** from the original set. See below:*



It is still an unresolved, open question in computer science, whether $P \overset{?}{=} NP \cap coNP$ and whether $P \overset{?}{=} NP$.

To prove that P_1 is in $coNP$, we must prove that $\overline{P_1}$ is in NP, which means that instead of giving a witness for the yes answer, we must give a witness for the no answer (that the graph has no 100-cycles). So now your friend wants to convince you that the graph has no 100-cycles. Although we don't have such a construct, which could in general show the non-existence of a 100-cycle in a graph, we don't need it.

Since the problem is solveable in P, we don't need any extra information to figure out if the answer is no to the question. Therefore, the witness is empty, and the witness checking algorithm is the polynomial solver we just discussed.

This logic works in general, for both a witness for the yes and the no answer, hence $P \subseteq NP$ and $P \subseteq coNP$ as we can see on the Venn-diagram above.

For** P_2**:

$\mathbf{P_2} \in$ **NP**

Similarly as before, the witness is now the list of the vertices in the k-cycle. Since the graph has at most n_G vertices, the length of this list is smaller than n_G and the size of the witness is $O(n_G \log(n_G))$. For $n_G < k$ no cycle can exist, therefore any witness of that length is immediately rejected. The checking algorithm is identical to the one for $P1$. Although the runtime calculation is a bit different, it is still polynomial.

Note: We can't prove that $P_2 \in P$ using the same algorithm as we did for P_1, since $\binom{n}{k}$ is no longer polynomial (k is not a constant anymore). For example for $k = \frac{n}{2}$, the algorithm would run in $\Theta(\binom{n}{\frac{n}{2}})$ (the lower bound is also true), which is $\Theta(\frac{2^n}{\sqrt{n}})$ due to Stirling's approximation. It does not follow that the problem is not in P, just that this possible solver algorithm is not polynomial itself. Other algorithms could potentially perform better, but in this course we do not study such an algorithm.

119. We are given a length n sequence of characters of the English alphabet. We want to decide whether n is a prime number. Prove that this problem is in P.

Solution:

To prove that this problem is in P, we must give a polynomial algorithm that can solve it.

For the numbers $i = 2, 3, \ldots, \lfloor\sqrt{n}\rfloor$ we shall do the following: Let's iterate through the characters of the sequence and mark each i'th character with a $*$. (Effectively performing integer division by i.) If we mark the last character with a $*$, it means n is divisible by i, therefore it is not a prime.

Since n's factors come in pairs, where one is smaller than or equal to $\lfloor\sqrt{n}\rfloor$ and the other is larger than or equal to $\lceil\sqrt{n}\rceil$, all possible pairs have been checked.

If no factor is found, n must be a prime.

The number of steps required is $O(\sqrt{n} \cdot n)$, which is polynomial relative to the input size, which is $O(n)$.

Note: We could also convert n into a binary number and perform the commonly known algorithm for integer division for an even more efficient algorithm.

---

120. We want to decide about a positive integer given in its decimal number system form whether (a) it is a complete square (b) whether it is a power of 2. (The input size n is the number of the decimal digits.) Prove that these problems are in P.

Solution:

To prove that these problems are in P, we must give polynomial algorithms that can solve them.

Let us denote the input number with s, therefore the size of the input is $n \in O(\log_2(s))$.

(a) We will use BinarySearch to determine whether there exists a number q such that $q^2 = s$.

Initially, the possible values of q come from the interval $[1, s]$ (since the square of a number cannot be more the number itself).

Then, we pick the mid point of this interval, $q = \lceil\frac{s+1}{2}\rceil$. We then multiply $q \cdot q$ and check whether the result is

- *$q \cdot q = s$, in this case, the answer is yes and we can immediately return.*
- *$q \cdot q < s$, in this case we continue searching the lower half of the interval.*
- *$q \cdot q > s$, in this case we continue searching the upper half of the interval.*

Eventually, we either find the root or our interval of possibilities shrinks to size 0, in which case the answer is no, s is not a complete square.

Multiplications can be performed with decimal representations in $O(n^2)$ steps. The number of questions required for binary search is $O(\log_2 s) \subseteq O(n)$. Therefore, the algorithm's complexity is $O(n^3)$.

(b) Perform integer division on s by 2 with remainder. If the remainder is 1, then s is not a power of 2. If the remainder is 0, divide the quotient by 2 again, and so on. Once the quotient becomes 1 (and the remainder

is 0*), then s is a power of* 2 *and we can stop. One division can be performed in* $O(n^2)$ *steps. The number of divisions is* $O(\log_2 s) \subseteq O(n)$*. Therefore, the algorithm's complexity is* $O(n^3)$*.*

121. We are given a directed graph by its adjacency list. We want to decide whether it can be transformed into a DAG by deleting at most 3 edges. Prove that this problem is in P.

Solution:

Let's iterate through all the possible size 3 *sets of edges (if* G *has* n_G *vertices and* m_G *edges, then there are* $O(m_G^3)$ *many size* 3 *edge sets). In each step of the iteration, remove just those* 3 *edges from the graph temporarily, then run a DFS on the resulting graph. If the DFS does not find a back edge, then the resulting graph is a DAG, in which case we have our solution. If it is not a DAG, we keep trying with the other edge sets (by first putting back the previous* 3 *edges into the graph and taking out the new ones). If neither of the cases gave us a DAG, then there are no suitable edges. The number of steps in the DFS is* $O(n_G + m_G)$*, when the graph is given by its adjacency list. So the complete algorithm runs in* $O(m_G^3(n_G + m_G))$*, which is polynomial relative to the input size, which is* $O(n_G + m_G)$ *in the case of an adjacency list.*

122. For the following languages (of decision problems), prove that the first seven are in NP, the last two in coNP. Which of them are in P?

$P_1 = \{\ (G, k) \mid G$ is a bipartite graph and k a positive integer, and G has a matching of size $k\ \}$
$P_2 = \{\ G \mid G$ is an undirected graph with an Eulerian circuit $\}$
$P_3 = \{\ G \mid G$ is an undirected graph with an independent vertex set of size 2024 $\}$
$P_4 = \{\ (G, k) \mid G$ is an undirected graph, k a positive integer, and G has an independent vertex set of size $k\ \}$
$P_5 = \{\ (s_1, \dots, s_m, b) \mid b$ and all the s_i's are positive integers, and there is a subset of $(s_1, \dots, s_m)$ such that their sum is $b\ \}$
$P_6 = \{\ G \mid G$ is an undirected graph with chromatic number 2 $\}$
$P_7 = \{\ G \mid G$ is an undirected graph with a proper coloring with 2024 colors $\}$
$P_8 = \{\ m \mid m$ is a positive integer and all its non-trivial divisors are smaller than $\dfrac{m}{10}\ \}$
$P_9 = \{\ G \mid G$ is a tree on $n_G \geq 2$ vertices and it has exactly 1 leaf $\}$

Solution:

$\mathbf{P_1 \in NP}$*:*

- ***Input + its size:*** *We assume G is given by its adjacency matrix and the number k is in binary format unless stated otherwise. The adjacency matrix is size* $O(n_G^2)$*, the binary number is size* $O(\log(k))$*, for a total of* $O(n_G^2 + \log(k))$*.*

- ***Witness (t) + its size is polynomial relative to the input:*** *The witness is a matching consisting of k edges, which is given by a list of k vertex index pairs, each of them in binary format. Since the graph has* n_G *vertices, one vertex index has the width at most* $O(\log(n_G))$ *bits, for a total witness size of* $O(k\log(n_G))$*. Although it is not immediately visible that this is polynomial relative to the input size* $O(n_G^2)$*, since a valid matching cannot have more than* $\lfloor \frac{n_G}{2} \rfloor$ *edges, there is an upper bound on k with* $k \leq \lfloor \frac{n_G}{2} \rfloor$*, which means the witness size is in fact polynomial relative to the input size.*

- ***Witness checking algorithm (τ) + its runtime is polynomial relative to the input:*** *The verification algorithm checks whether each edge of the matching exists in the graph (*$O(k)$*), that it contains exactly k edges (*$O(k)$*), and that no two edges share a common endpoint (*$O(k \log k)$*), since duplicates can be found via sorting). This is a* $O(k \log k)$ *algorithm, which is polynomial relative to the input size due to the same upper bound argument we had for the witness size.*

- ***Correctness:*** *A witness always exists for a yes instance (graph with a k-matching) and never for a no instance (graph with no k-matching), since the witness is the matching itself.*

$\mathbf{P_1 \in P}$*: In ITC2, we have seen a polynomial algorithm that finds the maximum size matching. We just need to add an extra step to verify that this maximum is at least k.*

$\mathbf{P_2} \in \mathbf{NP}$*:*

- ***Input + its size:*** *We assume G is given by its adjacency matrix, of size $O(n_G^2)$.*
- ***Witness (t) + its size is polynomial relative to the input:*** *The witness is an Eulerian circuit, i.e. a suitable permutation of the edges. It is given by the list of edges in the order they appear on the circuit, each edge represented by the vertex indices of its two endpoints in binary format. The size of the witness is $O(n_G^2 \log(n_G))$, as we have $O(n_G^2)$ edges, each represented by two $O(\log(n_G))$-width binary number. This witness size is polynomial relative to the input size.*
- ***Witness checking algorithm (τ) + its runtime is polynomial relative to the input:*** *The verification algorithm checks whether each edge of the witness is indeed an edge in the graph and that no edges of the graph has been left out or used more than once. This can be done for example by iterating the edges of the witness and constructing the adjacency matrix of the matrix from them and checking if the result is identical to the original graph. This step is done in $O(n_G^2)$ time. Then, the verification algorithm also checks that each edge in the list shares a common vertex with the next one, forming a walkable circuit. This step is done in $O(n_G^2)$ time. So the total runtime is $O(n_G^2)$, which is polynomial relative to the input size.*
- ***Correctness:*** *A witness always exists for a yes instance (graph with an Eulerian circuit) and never for a no instance (graph with no Eulerian circuit), since the witness is the Eulerian circuit itself.*

$\mathbf{P_2} \in \mathbf{P}$*: We have seen in ITC2 that an undirected graph has an Eulerian circuit if and only if the graph is connected and every vertex has an even degree. Connectivity can be checked with a polynomial algorithm by running a BFS or DFS on the graph once in $O(n_G^2)$ time. By summing the rows of the adjacency matrix, we can calculate the degree of each vertex in $O(n_G^2)$, so the second condition can also be verified in polynomial time.*

$\mathbf{P_3} \in \mathbf{NP}$*:*

- ***Input + its size:*** *We assume G is given by its adjacency matrix, of size $O(n_G^2)$.*
- ***Witness (t) + its size is polynomial relative to the input:*** *A witness is a size 2025 subset of the vertices, $S \subseteq V(G)$, consisting of the independent points. It is given by the list of the vertex indices in binary format. The size of this is $O(2025 \log(n_G)) = O(\log(n_G))$, which is polynomial relative to the input size.*
- ***Witness checking algorithm (τ) + its runtime is polynomial relative to the input:*** *The verification algorithm checks each pair of points in S to see if there is an edge between them. This can be done in $O(2025^2) \subseteq O(1)$ steps, by performing lookups in the adjacency matrix, which is in fact $O(1)$ constant time.*
- ***Correctness:*** *A witness always exists for a yes instance (graph with a size 2025 independent vertex set) and never for a no instance (graph with no size 2025 independent vertex set), since the witness is the size 2025 independent vertex set itself.*

$\mathbf{P_3} \in \mathbf{P}$*: We can check all possible size 2025 subsets of the vertices of G. There will be $\binom{n_g}{2025}$ of them. For each of these vertex sets we must check if they are independent: there can be no edges running between any pairs of vertices in them. this can be done in $O(2025^2) = O(1)$ steps by doing lookups in the adjacency matrix. The total runtime is therefore $O(\binom{n_G}{2025}) = O(n_G^{2025})$, which is polynomial relative to the input size.*

$\mathbf{P_4} \in \mathbf{NP}$*:*

- ***Input + its size:*** *We assume G is given by its adjacency matrix and the number k is in binary format unless stated otherwise. The adjacency matrix is size $O(n_G^2)$, the binary number is size $O(\log(k))$, for a total of $O(n_G^2 + \log(k))$.*

- ***Witness (t) + its size is polynomial relative to the input:*** *A witness is a size k subset of the vertices, $S \subseteq V(G)$, consisting of the independent points. It is given by the list of the vertex indices in binary format. The size of this is $O(k\log(n_G))$. It is not immediately obvious, that this is polynomial relative to the input size, since $O(k)$ is exponential relative to $O(\log k)$. However, since the size of any independent vertex set in G cannot be larger than n_G, the number of vertices, we know that the upper bound $k \leq n_G$ is true. This makes the witness size polynomial relative to the input size.*
- ***Witness checking algorithm (τ) + its runtime is polynomial relative to the input:*** *The verification algorithm checks each pair of points in S to see if there is an edge between them. This can be done in $O(k^2)$ steps, by performing lookups in the adjacency matrix. With a similar argument as for the witness size, this is polynomial relative to the input size.*
- ***Correctness:*** *A witness always exists for a yes instance (graph with a size k independent vertex set) and never for a no instance (graph with no size k independent vertex set), since the witness is the size k independent vertex set itself.*

$\mathbf{P_4} \overset{?}{\in} \mathbf{P}$*: This problem is in fact NP-complete. It is the Maximum Independent Vertex Set (MAXINDEP) problem's decision form. Therefore, if anyone was able to prove that $MAXINDEP$ is in P, $P = NP$ would immediately follow and vice versa. We do not know whether or not this is true at the moment.*

$\mathbf{P_5} \in \mathbf{NP}$*:*

- ***Input + its size:*** *All integers are in binary format unless stated otherwise. The width of the s_i can be upper bound with the maximum s_i's width, therefore the size of the input is $O(m\log(max(s_i)) + \log(b))$.*
- ***Witness (t) + its size is polynomial relative to the input:*** *A witness is a subset of indices of S for which the sum equals b. This is given by the list of indexes in binary format. The size of a single index is $O(\log m)$, and we can choose up to m indexes, therefore the of the witness is $O(m\log(m))$, which is polynomial relative to the input size.*
- ***Witness checking algorithm (τ) + its runtime is polynomial relative to the input:*** *The verification algorithm calculates the sum according to the given indices and compares it with b. The algorithm that performs adding the two numbers, $a + b$ receives the numbers in binary format, with size $O(\log(a) + \log(b))$ on its input and it can perform the calculations in $O(max(\log(a), \log(b))) \subseteq O(\log(a)+\log(b))$ steps using the method we learn in primary school. However, after multiple additions, the size of the result can grow larger and larger. In each step, the result can be 1 bit longer than the operands. After m additions, the size of the result can become m bits longer, i.e. $O(m + \log(\max(s_i))$. Therefore, we can upper bound the number of steps in the additions with $O((m+\log(\max(s_i))^2)$. Finally, to compare the number to b, we will take at most $O(\log(b))$ steps, to check each bit of the result. If the result has a different number of bits, it can be rejected upon discovering this fact.*
- ***Correctness:*** *A witness always exists for a yes instance (a set, which has a subset that sums up to b) and never for a no instance (a set, which has no subset that sums up to b), since the witness is the subset itself.*

$\mathbf{P_5} \overset{?}{\in} \mathbf{P}$*: This problem is in fact NP-complete. It is the Subset Sum (SSP) problem's decision form. Therefore, if anyone was able to prove that SSP is in P, $P = NP$ would immediately follow and vice versa. We do not know whether or not this is true at the moment.*

$\mathbf{P_6} \in \mathbf{NP}$*: The chromatic number being 2 means, that the graph can be colored with 2 colors and cannot be colored with 1 color.*

- ***Input + its size:*** *We assume G is given by its adjacency matrix, of size $O(n_G^2)$.*
- ***Witness (t) + its size is polynomial relative to the input:*** *The witness is a good coloring with 2 colors. This is given by a binary array of 0's and 1's, where $c[i] = 0$ if v_i's color is red and $c[i] = 1$ if v_i's color is blue. The size of the witness is $O(n_G)$, which is polynomial relative to the input size.*

- ***Witness checking algorithm* (τ) *+ its runtime is polynomial relative to the input*:** *The verification algorithm checks that it received exactly n_G number of bits (and only 0's and 1's are allowed). Then, it checks for each edge of the graph, whether its two endpoints received different colors, to make sure this is a valid 2-coloring. Additionally, it must also check if the graph has no valid 1-coloring, otherwise the chromatic number would be 1. This is only the case, if the graph has absolutely no edges, so we iterate at the adjacency matrix until we find an edge (this can be done while doing the first check anyways). The verification can be performed in $O(n_G^2)$ steps, checking each edge, which is polynomial relative to the input size.*

- ***Correctness**: A witness always exists for a yes instance (graph with valid 2 coloring that has no valid 1 coloring) and never for a no instance (graph with no valid 2 coloring), since the witness is the 2-coloring itself and the verification algorithm checks to make sure no valid 1-coloring exists in the graph.*

$\mathbf{P_6} \in \mathbf{P}$*: Graphs that have valid 2-colorings are exactly the bipartite graphs. We have seen an algorithm in ITC2 to check if a graph is bipartite using BFS. The number of steps of this algorithm is polynomial. Then, we check if the graph has at least an edge, to exclude graphs with a chromatic number of 1.*

$\mathbf{P_7} \in \mathbf{NP}$*: Notice, that the task is not asking us to check that there is no valid coloring of less colors, only that a valid 2025 coloring is achieveable.*

- ***Input + its size**: We assume G is given by its adjacency matrix, of size $O(n_G^2)$.*

- ***Witness (t) + its size is polynomial relative to the input**: The witness is a good coloring with 2025 colors. We can number the colors between 1 and 2025. Then, the coloring is given by an integer array, containing numbers between 1 and 2025 where $c[i] = x$ if v_i's color's number is x. A size of a single cell is the width of a binary number between 1 and 2025, $O(\log(2024)) = O(1)$. The size of the witness is therefore $O(n_G)$, which is polynomial relative to the input size.*

- ***Witness checking algorithm* (τ) *+ its runtime is polynomial relative to the input*:** *The verification algorithm checks that it received exactly n_G number of integers between 1 and 2025. Then, it checks for each edge of the graph, whether its two endpoints received different colors, to make sure this is a valid 2025-coloring. Luckily we do not have to check that no valid coloring on less colors exists, because that is an NP-complete problem, for a color number of 3 or larger. Therefore the verification is complete and this can be performed in $O(n_G^2)$ steps, checking each edge, which is polynomial relative to the input size.*

- ***Correctness**: A witness always exists for a yes instance (graph with valid 2025 coloring) and never for a no instance (graph with no valid 2025 coloring), since the witness is the 2025-coloring itself.*

$\mathbf{P_7} \overset{?}{\in} \mathbf{P}$*: This problem is in fact NP-complete. It is the problem of whether a 2025 coloring exists for a specific graph (2025COLOR) in decision form. Therefore, if anyone was able to prove that 2025COLOR is in P, $P = NP$ would immediately follow and vice versa. We do not know whether or not this is true at the moment.*

$\mathbf{P_8} \in \mathbf{coNP}$*: A non-trivial divisor of m is d if it divides m without remainder (e.g. $m \mod d = 0$) and $1 < d < m$. (A prime number has no non-trivial divisors.) Notice that for coNP, we are giving a witness to the **no** answer! So we must prove that a divisor d for which both $\frac{m}{10} \leq d$ and also $1 < d < m$ holds exists.*

- ***Input + its size**: The input is the positive integer m, which is given in binary format unless stated otherwise. The size of the input is therefore $O(\log(m))$.*

- ***Witness (t) + its size is polynomial relative to the input**: A witness for the no answer is a divisor d of m for which both $\frac{m}{10} \leq d$ and also $1 < d < m$ holds. The size of the witness is $O(\log(d))$. Since $d < m$, this is smaller than the input size, and therefore polynomial relative to the input size.*

- ***Witness checking algorithm* (τ) *+ its runtime is polynomial relative to the input*:** *The verification algorithm first computes $\frac{m}{10}$ in $O(\log(m))$, then checks whether the witness is at least this* *large and is also non-trivial, e.g. $1 < d < m$. These comparisons also take $O(\log(m))$. Then, by performing an integer division with remainder, it also checks whether it is indeed a divisor. Since the division can be performed in $O(log_2(m)^2)$ steps, the total runtime of the algorithm is $O(log_2(m)^2)$, which is polynomial relative to the input size.*

- ***Correctness**: A witness **always exists for a no** instance (m with a non-trivial divisor **too large**) and **never for a yes instance** (m with only small enough non-trivial divisors), since the witness is the non-trivial divisor itself that is too large.*

$\mathbf{P_8} \in P$*: If m has a non-trivial divisor larger than or equal to $\frac{m}{10}$, since they come in pairs, their pair has to be smaller than or equal to 10. This is true, because if d is a non-trivial divisor of m, then let $c = \frac{m}{d}$ be its pair, so $m = cd$. Since $\frac{m}{10} \leq d$, therefore $m \leq 10d$ and from $m = cd$, it follows that $m = cd \leq 10d$, so $c \leq 10$. So we can check all possible options between 2 and 10 if they divide m. Checking the primes is enough, so check the remainder of m with $2, 3, 5$ and 7. We can do integer division in $O(\log(m)^2)$ four times, so this algorithm is polynomial relative to the input size.*

$\mathbf{P_9} \in \mathbf{NP}$*: Notice that for coNP, we are giving a witness to the **no** answer! Since all trees have at least 1 leaf, we must prove that the graph has 2 or more leaves.*

- ***Input + its size**: We assume the tree's graph G is given by its adjacency matrix, of size $O(n_G^2)$. (There are better representations for trees, but this is also fine.)*
- ***Witness (t) + its size is polynomial relative to the input**: A witness is two vertices with degree 1. They can be given by their binary indices. The size of the witness is therefore $O(\log(n_G)$, which is polynomial relative to the input size.*
- ***Witness checking algorithm (τ) + its runtime is polynomial relative to the input**: The verification algorithm checks that these vertices are indeed part of the graph by performing a range check in $O(\log(n_G))$ steps, checks that they are not the same vertex, by checking for equality in $O(\log(n_G))$ steps, then it counts the degree of the given vertices in the adjacency matrix and makes sure it is exactly 1, so they are indeed both leaves of the graph. This can be done in $O(n_G)$ steps. The total runtime is therefore $O(n_G)$, which is polynomial relative to the input size.*
- ***Correctness**: A witness **always exists for a no** instance (a tree with 2 or more leaves) and **never for a yes instance** (a tree with exactly 1 leaf), since the witness is 2 different leaves of the graph.*

$\mathbf{P_9} \in \mathbf{P}$*: It was mentioned in ITC2 that every tree with at least 2 vertices has at least 2 degree-1 vertices. Therefore, the answer to the question will always be no. That is, the algorithm will immediately give the no answer without examining anything about the input. The number of steps for this is constant. (For the case of a graph on 1 vertex, if it has no loop we don't consider that a leaf, since its degree is 0, not 1 and if it has a loop, it's not a tree because it has a cycle.)*

123. Assume that we have an algorithm X that can, given a graph G and a positive integer k, tell in polynomial time if G has an independent vertex set of size at least k.
(a) Design an algorithm (using X), that can determine $\alpha(G)$, the size of the maximum independent vertex set of G in polynomial time.
(b) Design an algorithm (using X), that will find an independent vertex set of G of size $\alpha(G)$ in polynomial time.

Solution:

Let us assume, that G is given by its adjacency matrix, of size $O(n_G^2)$ and k is in binary format, unless otherwise stated, of size $O(\log(k))$. Therefore, the size of X's input is $O(n_G^2 + \log(k))$. If we assume X runs in polynomial time relative to its input size, it means that there is some constant $0 < c$, for which it is true that X runs in $O((n_G^2 + \log(k))^c)$.

(a) We know that the possible values of $\alpha(G)$ are between 1 and n_G. We can run X for each possible k in $1 \leq k \leq n_G$ on the graph. The smallest k for which it returns Yes is $\alpha(G)$. The runtime of this algorithm will be $O(n_G(n_G^2 + \log(k))^c)$, which is polynomial relative to the input size.

Note: We can do this even more efficiently, using BinarySearch. Since if X returns yes for any k, it will also return yes for any k' larger than that, therefore our search space is "ordered". For smaller k's the answer is no, for larger k's the answer is yes and we are looking for the divide between them, which can be done using BinarySearch.

(b) First, determine the value of $k = \alpha(G)$ using the method given in (a). Then, let's examine each 'missing' edge from the graph separately. For a specific pair of vertices that is not connected via an edge, let us add an extra edge to the graph and run algorithm X on this modified graph with parameter k. If it reports YES back, it means that the size of (one of) the largest independent vertex set(s) did not decrease with this extra edge, therefore the edge must be connecting vertices where at least one of them is not part of this independent set. In this case, we keep the edge in the graph and continue. However, if X reports NO back, it means that the extra edge has decreased k (it cannot increase it), therefore it must be connecting vertices that are both part of the independent set. In this case let us remove this edge from the graph. Continue this procedure for all missing edges of the graph, until we finish all of them.

Observations about this algorithm:

- *The algorithm does not branch! For each edge it makes a definitive choice on whether we should keep it or throw it away and continues on that graph only.*
- *The resulting graph will contain all of the edges that do not run between points of the independent set, as those are the only ones we skipped. Therefore the independent set is clearly visible, as it will be a large square submatrix full of 0's in the adjacency matrix. We can for example calculate the degree of all vertices in $O(n_G^2)$ and keep the ones that are equal to $n_G - \alpha(G)$.*
- *The runtime of the algorithm is as follows: We first run the algorithm we designed in (a), then we run algorithm X for each of the $O(n_G^2)$ number of edges of the graph, then we run the degree check. Therefore it is $O(n_G(n_G^2 + \log(k))^c + n_G^2(n_G^2 + \log(k))^c + n_G^2)$, which is polynomial relative to the input size.*

---

124. Let FACTORIZATION be the language of the following decision problem:
Input: Two positive integers m and t.
Question: Does m have a divisor d such that, $1 < d \leq t$?

(a) Show that FACTORIZATION is in NP.
(b) Show that FACTORIZATION is in coNP. (You can use the fact that PRIMES is in P.)
(c) Show that if $P = NP \cap coNP$, then RSA can be decrypted fast, because then prime factorization can be done in polynomial time.

Solution:

(a) $FACTORIZATION \in NP$

- ***Input + its size**: The input is the positive integers m and t, which are given in binary format unless stated otherwise. The size of the input is therefore $O(\log(m) + \log(t))$.*
- ***Witness (t) + its size is polynomial relative to the input**: A witness for the yes answer is the divisor d, such that $1 < d \leq t$. The size of the witness is $O(\log(d))$. Since $d \leq t$, this is smaller than the input size, and therefore polynomial relative to the input size.*
- ***Witness checking algorithm (τ) + its runtime is polynomial relative to the input**: The verification algorithm does a range check in $O(\log(t))$, then performs integer division with remainder in $O((\log(m) + \log(d))^2)$ time. Therefore the total time is $O(\log(t) + (\log(m) + \log(d))^2)$, which is polynomial relative to the input size, since $d \leq t$.*

- ***Correctness****: A witness always exists for a yes instance (for m that has a divisor less than or equal to t) and never for the no answer (for m with no such divisor), since the witness is the divisor itself in question.*

(b) $FACTORIZATION \in coNP$

- ***Input + its size****: The input is the positive integers m and t, which are given in binary format unless stated otherwise. The size of the input is therefore* $O(\log(m) + \log(t))$.
- ***Witness (t) + its size is polynomial relative to the input****: A suitable witness for the no answer is the prime factorization of m, i.e.* $m = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_k^{\alpha_k}$, *where the* p_i *are primes and the* α_i *are positive integers.*

  The size of the witness is $O(\sum_{i=1}^{k} \log(p_i) + \log(\alpha_i))$.

  We handle the p_i *and the* α_i *parts separately, as follows:*

$$\sum_{i=1}^{k} \log(p_i) = \log(\prod_{i=1}^{k} p_i) \leq \log(\prod_{i=1}^{k} p_i^{\alpha_i}) = \log(m)$$

  For the α_i, *we can first show that if we switch out the primes in m's factorization to 2, we can only decrease its value, since that is the smallest possible prime.*

$$m = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_k^{\alpha_k} \geq 2^{\alpha_1} 2^{\alpha_2} \dots 2^{\alpha_k} = 2^{\sum_{i=1}^{k} \alpha_i}$$

  Taking the logarithm of both sides we get

$$\log(m) \geq \sum_{i=1}^{k} \alpha_i$$

  Now looking back at our α_i *sum above, since the logarithm of a positive integer is always less than or equal to the number itself, we get*

$$\sum_{i=1}^{k} \log(\alpha_i) \leq \sum_{i=1}^{k} \alpha_i \leq \log(m)$$

  Therefore the size of our witness is $O(\sum_{i=1}^{k} \log(p_i) + \log(\alpha_i)) \subseteq O(\log(m))$, *which is polynomial relative to the input size.*

- ***Witness checking algorithm (τ) + its runtime is polynomial relative to the input****:*

  The verification algorithm first checks if all p_i*'s are indeed prime, using the fact that* $PRIME \in P$, *i.e. the AKS primality check.*

  Since there are k primes to check, we need a suitable upper bound for the value of k.

  Since $1 \leq \alpha_i$ *for all i, we can see that*

$$\log(m) \geq \sum_{i=1}^{k} \alpha_i \geq \sum_{i=1}^{k} 1 = k$$

so this step is polynomial relative to the input size.

Then it calculates $p_1^{\alpha_1} p_2^{\alpha_2} \dots p_k^{\alpha_k}$ and checks if it is equal to m. This requires $O(\sum_{i=1}^{k} \alpha_i)$ multiplications and we have proven above already, that this can be upper bound by $O(\log(m))$. Each multiplication is performed between numbers of size $O(\log(m))$, therefore the runtime is $O(\log(m)^2)$ for one multiplication and $O(\log(m)^3)$ for all of them. The comparison then requires $O(\log(m))$ steps. These steps are therefore polynomial relative to the input size.

Finally it verifies if $t < p_i$ holds for every i. (Remember, this is a witness for the no answer.) This requires $O(k \log(max(p_i)))$ steps. We have already proven above that $k \leq \log(m)$, so this is also polynomial relative to the input size.

All steps are therefore polynomial realtive to the input size.

- ***Correctness**: The prime factorization of a number always exists and it can only pass the checking if m has no divisors smaller than or equal to t, since those would result in prime factors smaller than or equal to t in m's factorization.*

(c) We have just proven in parts (a) and (b), that FACTORIZATION $\in NP \cap coNP$. If we assumed that $P = NP \cap coNP$ (it is unknown to computer science whether this is true or not at the moment), it would follow that FACTORIZATION $\in P$, as in there would exist an algorithm with polynomial number of steps that can decide whether or not number m has a factor smaller than t.

If we actually want to find a factor of m, we could use this algorithm combined with a BinarySearch procedure. If the smallest factor of m is d_1, then for any $t < d_1$ given to this algorithm, it will return no and any $t \geq d_1$ it will return yes. If we find the smallest t where the answer is yes, that will equal to d_1. Then we can set $m_1 = \frac{m}{d_1}$ and continue finding the factors of m_1 in a similar manner, dividing with the next one and creating m_2 and so on, until all of them are found. Since $2 \leq d_i$, in each step the size of m (in binary format) decreases by at least one bit, therefore we run BinarySearch $O(\log(m))$ many times. In each step, the search space is $1 < t < m$, so BinarySearch runs in $O(\log(m))$ steps. If $FACTORIZATION$ has a polynomial solver, we would run it for a total of $O(\log(m)^2)$ times, making this entire algorithm polynomial, relative to the input size.

125. Assume that we have an algorithm X that can determine in polynomial time if a given graph G is 3-colorable. Provide an algorithm, using X, that finds a 3-coloring of a graph G (if it exists) in polynomial time.

Solution:

We first check with X is the graph itself is 3-colorable. If the answer is no, we can return this failure.

Then, we fill up the graph with as many edges we can while keeping it still 3 colorable.

For each "missing" edge from the graph, we add it and check if the graph is still 3 colorable. If the answer is yes, it means that this edge is connecting two vertices in two different color sets, therefore we can keep it. If the graph is not 3 colorable anymore, it means that the only way to color this graph using 3 colors involves assigning the same color to both endpoints of the edge, therefore the edge must be discarded.

This algorithm never branches. In each step we make a decision whether to keep the edge in or discard it and continue with that graph only.

At the end of the procedure, we will retain one possible configuration of 3 independent vertex sets of the graph and all edges running between them will be added by the procedure above, so we can easily distinguish the members of the three sets.

We called X $O(n_G^2)$ times, and X itself is polynomial, therefore this entire procedure is polynomial relative to the input size.

126. There exists two decision problems, A and B, which answer different questions for the same input. It is known that $A, B \in$ P. Prove that if decision problem C answers whether at least one of A or B answers YES for a given input, then $C \in \mathrm{P}$ also holds. How does the running time of the algorithm deciding C depend on the running times of the algorithms of A and B?

Solution:

Let A run in $O(n^a)$ and B run in $O(n^b)$, where n is the size of the common input and a, b positive integer constants.

Let's decide C by running the decider algorithm of A and B both. If either of them reports a yes answer back, we return yes, otherwise no.

The total runtime is $O(n^a + n^b + 1) \subseteq O(n^{a+b})$, which is polynomial relative to the input size n of C.

127. Let B be a polynomial running time algorithm, whose output is not necessarily one bit. Consider procedure C which runs B on input x, then runs B again with the output of the first run as input, and so on, c times altogether. Prove that the running time of C is polynomial if c is a constant. What happens if c could be polynomial in the length of the first input?

Solution:

Let the runtime of B be $O(n^b)$ if the size of its input x is n. Then, the size of the output of B will be $O(n^b)$. (In order for B to calculate the output, it must write it down, which is part of the runtime of B. Therefore the size of the output is asymptotically bound by the runtime of the algorithm.)

Upon refeeding B with the input of size $O(n^b)$, it will return an output of size $O((n^b)^b) = O(n^{b^2})$. Repeating this c times, we get a size $O(n^{b^c})$ result, which is polynomial relative to the original input size n, since b^c is a constant.

However, if B is called $c \in \Omega(n^k)$ times, this is no longer the case. Let us give a simple counterexample: Let B be an algorithm that duplicates the input x word on its output, i.e. it writes xx down, doubling the length. The runtime of B itself is $\Theta(n)$, which is polynomial. After the second call, it will write $xxxx$ down, making the length $4n$, and so on. If C runs B n times, refeeding the output of the previous run as the input of the next, the resulting length will be $\Theta(2^n \cdot n)$, which is not polynomial relative to the input size, even though C only called B a linear (i.e. polynomial) number of times relative to the first input's size.