NP-Complete problems, Approximation algorithms

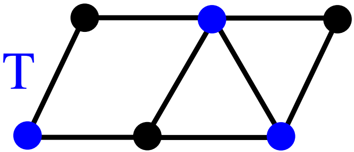
László Papp

BME

21st of April, 2023

Minimum vertex cover

Remainder: In a graph G, $T \subseteq V(G)$ is a vertex cover if for every edge u, v, either $u \in T$ or $v \in T$ or both. The size of the minimum vertex cover is denoted by $\tau(G)$.



Optimization version of VERTEX COVER
Input: Graph G.
Task: Find a minimum vertex cover.

Decision version of VERTEX COVER
Input: Graph G and a number k.
Question: Does G have a vertex cover of size at most k?

Claim
(The decision version of) VERTEX COVER is in NP.

The decision version of VERTEX COVER is in NP.

Input: Graph G and a number k.
Question: Does G have a vertex cover of size at most k?

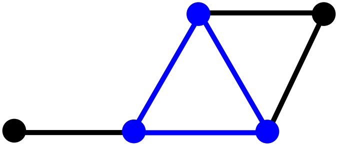
W is a witness for the YES answer if W is subset of $V(G)$, W covers $E(G)$ and $|W| \leq k$.
The verification algorithm checks these three properties:

- Checks that each element of W is a vertex of G and no element of W appears twice or more. It requires at most $O(n^2)$ time.
- For each edge of G it checks that one if its endpoints is contained in W. It requires at most $O(en)$ time.
- Counts the size of W and compare it to k. It require at most $O(n)$ time.

The size of the input is $\Theta(n + e + log(k))$. The size of the witness is $O(n)$. The time complexity of the verification algorithm is in $O(en)$. Both of them are polinomial in the size of the input. Therefore VERTEX COVER is in NP.

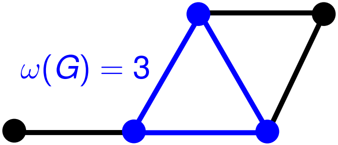
Cliques

Definition: A **clique** in a graph is a complete subgraph. The **clique number** $\omega(G)$ of graph G is the number of vertices of the biggest clique of G.



Cliques

Definition: A **clique** in a graph is a complete subgraph. The **clique number** $\omega(G)$ of graph G is the number of vertices of the biggest clique of G.



In chemistry, bioinformatics and social sciences, finding a maximum clique is an important task. We can define the following optimization problem:

Optimization version of CLIQUE
Input: Graph G.
Task: Find a maximum clique of G.

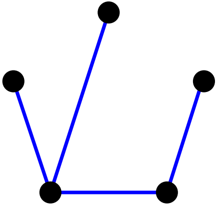
Decision version of CLIQUE

Problem CLIQUE
Input: Simple graph G and a number k.
Question: Is $\omega(G) \geq k$?

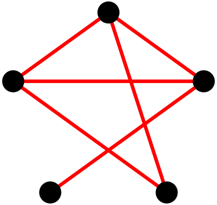
CLIQUE is in NP: If G is an input graph and the answer is YES, then the witness W_G is a subset of $V(G)$, which spans a clique of size k.
So the verification algorithm checks that W_G contains k different vertices and they are pairwise adjacent.
This can be done in $O(k^2)$ time.
The answer cannot be YES if $k > n$. Thus $O(k^2) \subseteq O(n^2)$, and therefore this verification algorithm runs in polynomial time.

Complement graph

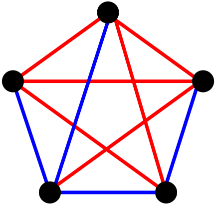
Definition: Let G be a simple graph. The **complement graph** of G, denoted by $\overline{G}$, is the simple graph over the same vertex set which contains edge $\{u, v\}$ if and only if G does not.



G



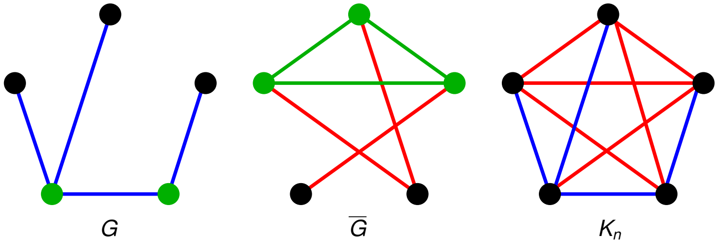
$\overline{G}$



K_n

Complement graph

Definition: Let G be a simple graph. The **complement graph** of G, denoted by $\overline{G}$, is the simple graph over the same vertex set which contains edge $\{u, v\}$ if and only if G does not.



Claim

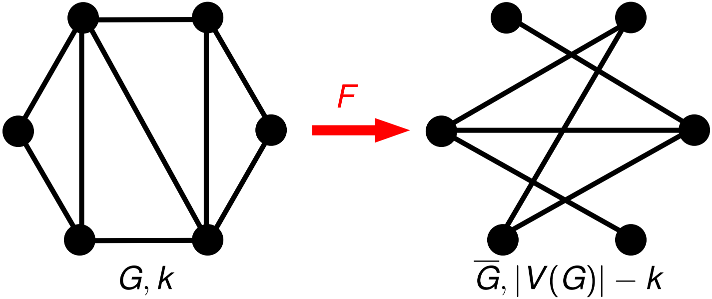
A $T \subseteq V(G)$ is a vertex cover of G if and only if $V(G) \setminus T$ spans a clique of $\overline{G}$.

Corollary

A $T \subseteq V(G)$ is a minimum vertex cover of G if and only if $V(G) \setminus T$ spans a maximum clique of $\overline{G}$.

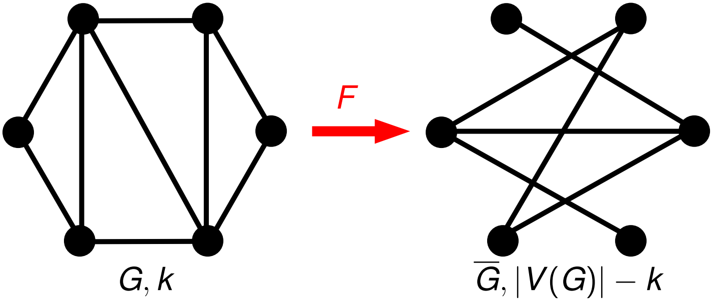
CLIQUE $\prec$ VERTEX COVER

The input is a simple graph G and a number k for CLIQUE.
The Karp-reduction F maps the (G, k) pair to $(\overline{G}, |V(G)| - k)$.
According to the last Claim G has a clique of size k if and only if $\overline{G}$ has a vertex cover of size $|V(G)| - k$. So if the answer of an input I of CLIQUE is YES (NO), then the answer for $F(I)$, which is an input of VERTEX COVER, is also YES (NO).
F can be calculated in $O(|V(G)|^2)$ time, so this is a polynomial time reduction.



CLIQUE $\prec$ VERTEX COVER

The input is a simple graph G and a number k for CLIQUE. The Karp-reduction F maps the (G, k) pair to $(\overline{G}, |V(G)| - k)$. According to the last Claim G has a clique of size k if and only if $\overline{G}$ has a vertex cover of size $|V(G)| - k$. So if the answer of an input I of CLIQUE is YES (NO), then the answer for $F(I)$, which is an input of VERTEX COVER, is also YES (NO). F can be calculated in $O(|V(G)|^2)$ time, so this is a polynomial time reduction.

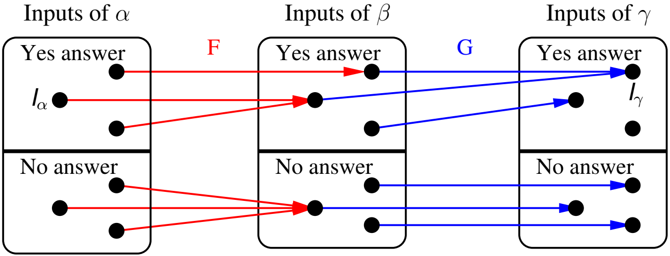


Note that F^{-1} is a VERTEX COVER $\prec$ CLIQUE Karp-reduction.

Transitivity of the Karp reduction

Claim: Let α, β and γ be decision problems. If $\alpha \prec \beta$ and $\beta \prec \gamma$, then $\alpha \prec \gamma$.

Proof: Let F be a Karp-reduction from α to β and let G be a Karp-reduction from β to γ. Then the mapping $G \circ F$ is a Karp-reduction from α to γ.



The size of $F(I_\alpha)$ is polynomial in the size of I_α, beacuse F can be calculated in polynomial time. The composition of two polynomials is a polynomial. Thus $G \circ F$ can be calculated in polynomial time. The answer for I_α is YES if and only if the answer for $I_\gamma = G \circ F(I_\alpha)$ is YES. □

An interpretation of Karp reduction

- A Karp-reduction from α to β is a method which can be used to solve α efficiently if we can solve β efficiently. So $\alpha \prec \beta$ intuitively means that α is not harder than β.
- $\alpha \prec \beta \prec \gamma \implies \alpha \prec \gamma$ matches our intiution: If α is not harder than β and β is not harder than γ then α is not harder than γ.
- We have seen that CLIQUE$\prec$ VERTEX COVER and CLIQUE$\prec$ VERTEX COVER. This can be interpreted as that the VERTEX COVER problem is as hard as the CLIQUE problem in some sense.
- We can use this kind of hardness to define a hierarchy of decision problems.

NP-hard, NP-complete

Definition: If for every problem $\alpha \in$ NP $\alpha \prec \beta$ holds, then we say that β is **NP-hard**.

Note that if we have a polynomial time algorithm for an NP-hard problem, then $P = NP$.

Defintion: A decision problem π is **NP-complete** if it is in NP and it is NP-hard.

So NP-complete problems are the hardest ones among the problems contained in NP. But it is not obvious that such a problem exists.

Cook-Levin Theorem

SAT is NP-Complete.

We are not going to prove this theorem.

CNF-SAT

Definition: In a Boolean formula a **literal** is a variable x_i or its negated version $\neg x_i$. A **clause** is a disjunction of literals. A Boolen formula is **CNF (in conjunctive normal form)** if it is a conjunction of several clauses.

Example: $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_4 \vee x_2) \wedge (x_1 \vee x_3) \wedge \neg x_4$ is CNF but $(x_1 \wedge x_2) \vee (\neg x_2 \wedge x_3) \vee (\neg x_3 \vee x_4)$ is not.

CNF-SAT

Definition: In a Boolean formula a **literal** is a variable x_i or its negated version $\neg x_i$. A **clause** is a disjunction of literals. A Boolen formula is **CNF (in conjunctive normal form)** if it is a conjunction of several clauses.

Example: $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_4 \vee x_2) \wedge (x_1 \vee x_3) \wedge \neg x_4$ is CNF but $(x_1 \wedge x_2) \vee (\neg x_2 \wedge x_3) \vee (\neg x_3 \vee x_4)$ is not.

PROBLEM CNF-SAT
Input: A CNF Boolean formula ϕ
Question: Can we assign values 0 and 1 to the variables of ϕ such a way that ϕ evaluates to 1?

Claim
CNF-SAT is in NP and CNF-SAT is NP-complete.

The witness is a proper assignment of the variables

3-SAT

Definition: A Boolean formula is **3-CNF** if it is CNF and each clause contains exactly three literals.
Example: $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_4 \vee x_2) \wedge (x_1 \vee x_3 \vee \neg x_4)$ is 3-CNF but $(x_1 \vee x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee x_4)$ is not.

Problem 3-SAT
Input: A 3-CNF Boolean formula Φ
Question: Can we assign values 0 and 1 to the variables of Φ such a way that Φ evaluates to 1?

Claim
There is a polinomial algorithm which converts any Boolen formula Φ into an equisatisfiable $3 - CNF$ Boolean formula Φ', so Φ is satisfiable $\iff$ Φ' is satisfiable.

Corollary:

3-SAT is NP-complete.

How to prove that problem π is NP-complete?

Reformulation of the definition: π is NP-complete if it is in NP and for each $\alpha \in$ *NP* $\alpha \prec \pi$.

There are infinitely many problems in the NP class. Therefore showing for each $\alpha \in$NP that there is a Karp reduction from α to π, is hopeless.

How to prove that problem π is NP-complete?

Reformulation of the definition: π is NP-complete if it is in NP and for each $\alpha \in$ *NP* $\alpha \prec \pi$.

There are infinitely many problems in the NP class. Therefore showing for each $\alpha \in$NP that there is a Karp reduction from α to π, is hopeless.
If $\beta \prec \pi$ and β is NP-hard, then for every $\alpha \in$ NP $\alpha \prec \beta$ and the transitivity of the Karp reduction implies that $\alpha \prec \pi$. This means that π is NP-hard.

To prove that π is NP-complete:

- Show that π is in *NP*.
- Find an NP-complete problem β and show that $\beta \prec \pi$.

CLIQUE is NP-complete

Problem CLIQUE
Input: Simple graph G and a number k.
Question: Is $\omega(G) \geq k$?

CLIQUE is in NP: If G is an input graph and the answer is YES, then the witness W_G is a subset of $V(G)$, which spans a clique of size k.
So the verification algorithm checks that W_G contains k different vertices and they are pairwise adjacent.
This can be done in $O(k^2)$ time.
The answer cannot be YES if $k > n$. Thus $O(k^2) \subseteq O(n^2)$, and therefore this verification algorithm runs in polynomial time.

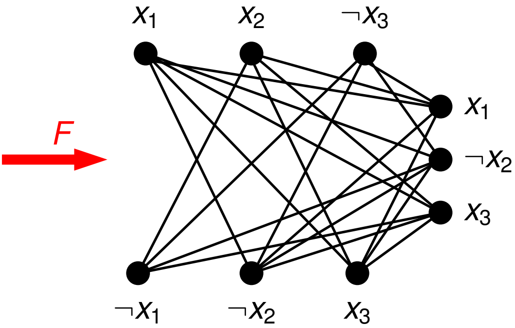
CLIQUE is NP-hard: We will give a 3-SAT$\prec$CLIQUE Karp reduction. Since 3-SAT is NP-hard, the existence of this Karp reduction proves that CLIQUE is NP-hard as well.

Corollary: CLIQUE is NP-complete.

3-SAT $\prec$ CLIQUE

For the input $\phi = (l_1^1 \vee l_2^1 \vee l_3^1) \wedge (l_1^2 \vee l_2^2 \vee l_3^2) \wedge \ldots \wedge (l_1^k \vee l_2^k \vee l_3^k)$
the reduction creates the following graph:
The vertices of the graph are l_j^i (the *j*th literal of the *i*th clause).
l_j^i and l_n^m are adjacent if and only if $i \neq m$ and $l_j^i \neq \neg l_n^m$.
(The corresponding literals are not in the same close and they are not the negate of each other.)
We search for a clique of size $\geq k$ (the number of clauses).

$$(x_1 \vee x_2 \vee \neg x_3) \wedge \\ \wedge(\neg x_1 \vee \neg x_2 \vee x_3) \wedge \\ \wedge(x_1 \vee \neg x_2 \vee x_3)$$

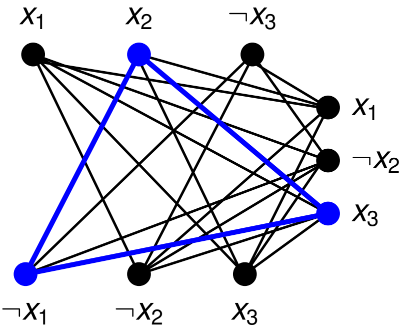


F can be calculated in $O(|\phi|^2)$ time, so in polynomial time.

Correctness of the reduction

Consider an assignment which satisfies ϕ. In each clause there is a literal which is true. These literals span a clique of size k.

$$\begin{array}{l}(x_1 \vee x_2 \vee \neg x_3)\wedge \\ \wedge(\neg x_1 \vee \neg x_2 \vee x_3)\wedge \\ \wedge(x_1 \vee \neg x_2 \vee x_3)\wedge\end{array} \xrightarrow{F}$$



A clique does not contain two literals from the same clause. Therefore if the size of a clique is the number of the clauses, then this clique contains one literal from each clause. Setting these literals to true satisfy ϕ. We can set all of these literals true because none of them is a negate of another one contained in the clique.

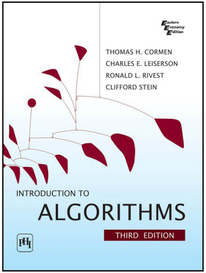
Corollaries

$$\left.\begin{array}{l}\text{3-SAT} \in \text{NP-Complete}\\ \text{3-SAT} \prec \text{CLIQUE}\\ \text{CLIQUE} \in \text{NP}\end{array}\right\} \Longrightarrow \text{CLIQUE} \in \text{NP-Complete}$$

$$\left.\begin{array}{l}\text{CLIQUE} \in \text{NP-Complete}\\ \text{CLIQUE} \prec \text{VERTEX COVER}\\ \text{VERTEX COVER} \in \text{NP}\end{array}\right\} \Longrightarrow \begin{array}{l}\text{VERTEX COVER}\\ \in \text{NP-Complete}\end{array}$$

VERTEX COVER $\prec$ HAMILTONIAN

A Karp-reduction from VERTEX COVER to HAMILTONIAN is a bit harder than the reductions which we have seen, but it exists. If you are interested, you can read it in the book Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms.



Corollary: HAMILTONIAN and the decision version of the TSP are NP-complete.

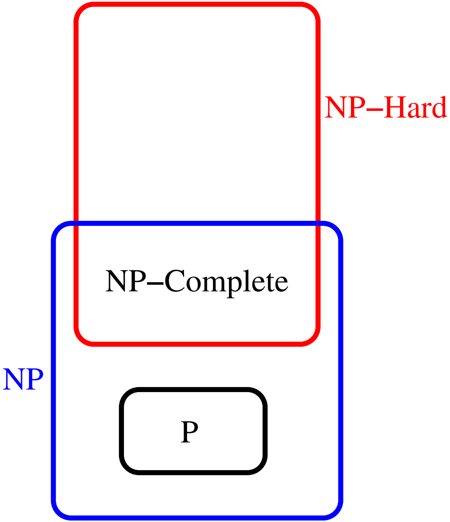
All of these decision problems are NP-complete:

- SAT
- 3-SAT
- CLIQUE
- VERTEX COVER
- HAMILTONIAN
- TSP
- Many other decision problems.

If any of them is in P, then all of them are. Many people tried to give a polynomial time algorithm for them, but no one has been successful so far. Therefore most people believe that $P \neq NP$.

Relations between the classes which we have learnt

If we assume that $P \neq NP$, then the classes look like this:



What to do if our task is to solve an instance of a problem which is NP-hard?

There are many options:

- If its size is small then we can use some exponential time algorithms.
- If the problem is an optimization problem, then we can use heuristics or approximation algorithms.
- Check whether the given instance belongs to a special version of the problem which is not NP-complete.

Example: CNF-SAT is in NP-Complete, but 2-SAT is in P.
Definition: A Boolean formula is **2-CNF** if it is CNF and each clause contains exactly two literals.

Problem 2-SAT
Input: A 2-CNF boolean formula ϕ.
Question: Can ϕ be satisfied?

Bin packing

Problem: We have boxes (bins) of the same size and many objects of different sizes. We want to put all the objects in the boxes. The boxes, their transportation and the storage space which they require cost money. Therefore we want to use as few boxes as possible.

Bin packing as an optimization problem

A list of rational numbers between 0 and 1 is given: $a_1, a_2, a_3 \ldots a_n$, $\forall i \; 0 \leq a_i \leq 1$. These are the sizes of the objects. The size of each bin is 1. We can put a set of objects in a bin only if the sum of their sizes is at most 1. Determine the least number of bins which are required to pack all the objects!

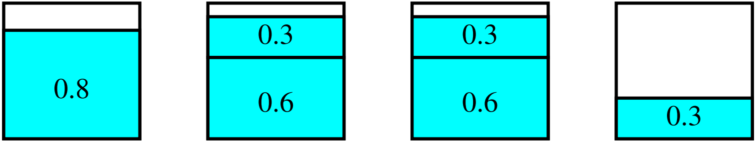
Example: We have six objects of sizes 0.8, 0.6, 0.6, 0.3, 0.3, 0.3.

Bin packing as an optimization problem

A list of rational numbers between 0 and 1 is given: $a_1, a_2, a_3 \ldots a_n$, $\forall i\ 0 \leq a_i \leq 1$. These are the sizes of the objects. The size of each bin is 1. We can put a set of objects in a bin only if the sum of their sizes is at most 1. Determine the least number of bins which are required to pack all the objects!

Example: We have six objects of sizes 0.8, 0.6, 0.6, 0.3, 0.3, 0.3.

We can pack them in 4 bins:

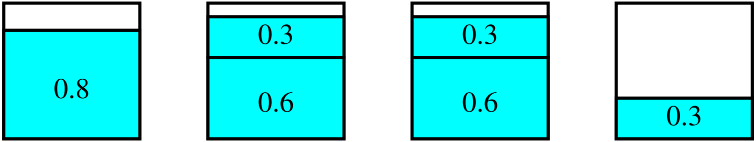


Bin packing as an optimization problem

A list of rational numbers between 0 and 1 is given: $a_1, a_2, a_3 \ldots a_n$, $\forall i \ 0 \leq a_i \leq 1$. These are the sizes of the objects. The size of each bin is 1. We can put a set of objects in a bin only if the sum of their sizes is at most 1. Determine the least number of bins which are required to pack all the objects!

Example: We have six objects of sizes 0.8, 0.6, 0.6, 0.3, 0.3, 0.3.

We can pack them in 4 bins:



However the sum of the sizes is only 2.9 we cannot pack the objects in 3 bins, because the object of size 0.8 has to be packed alone, therefore 0.2 space is wasted in its bin.

Bin packing as a decision problem

Problem BIN PACKING
Input: A list of rational numbers between 0 and 1 are given: $a_1, a_2, a_3 \dots a_n$, $\forall i\ 0 \leq a_i \leq 1$ and an integer k.
Question: Can we pack all the items of given sizes in k bins of size one?
Equivalently: Are there disjoint sets $B_1, B_2, \dots, B_k$, such that $\cup_{i=1}^{k} B_i = \{1, 2, \dots n\}$ and $\forall i \sum_{j \in B_i} a_j \leq 1$?

Claim

BIN PACKING is NP-complete.

It can be shown that BIN PACKING is in NP. A SAT3$\prec$BIN PACKING exists, but we do not discuss it now.

Trying to solve the optimization version of Bin packing

OK Bin packing is hard, it is unlikely that there is a fast algorithm which can find an optimal packing if we have more than 50 items. What to do?

Trying to solve the optimization version of Bin packing

OK Bin packing is hard, it is unlikely that there is a fast algorithm which can find an optimal packing if we have more than 50 items. What to do?

Use an algorithm which does not use much more bins than the optimal packing (which uses the least amount of bins)!

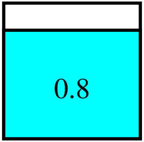
That is what we call an approximation algorithm. We write the formal definition later.

First Fit (FF) Algorithm

We pack the items incrementally. We create a bin B_1 and put the first item in B_1. Assume that, we have packed the first $i - 1$ items and we want to put the ith item in a bin. We do it in the following way:

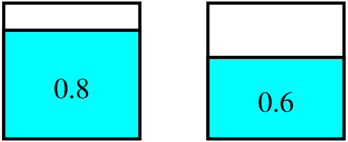
We go through the bins in the order of their creation time and we try to put the ith item in each of them. At the first occurrence when the item fits in a bin we put it there. If no bin has enough free space to store the ith item, then we create a new bin and place the ith item there.

Example: We have six objects of sizes 0.8, 0.6, 0.6, 0.3, 0.3, 0.3.



First Fit (FF) Algorithm

We pack the items incrementally. We create a bin B_1 and put the first item in B_1. Assume that, we have packed the first $i-1$ items and we want to put the ith item in a bin. We do it in the following way:
We go through the bins in the order of their creation time and we try to put the ith item in each of them. At the first occurrence when the item fits in a bin we put it there. If no bin has enough free space to store the ith item, then we create a new bin and place the ith item there.
Example: We have six objects of sizes 0.8, 0.6, 0.6, 0.3, 0.3, 0.3.

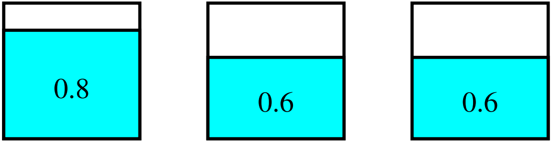


First Fit (FF) Algorithm

We pack the items incrementally. We create a bin B_1 and put the first item in B_1. Assume that, we have packed the first $i-1$ items and we want to put the *i*th item in a bin. We do it in the following way:

We go through the bins in the order of their creation time and we try to put the *i*th item in each of them. At the first occurrence when the item fits in a bin we put it there. If no bin has enough free space to store the *i*th item, then we create a new bin and place the *i*th item there.

Example: We have six objects of sizes 0.8, 0.6, 0.6, 0.3, 0.3, 0.3.

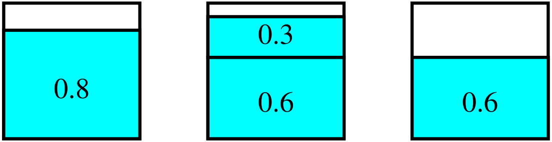


First Fit (FF) Algorithm

We pack the items incrementally. We create a bin B_1 and put the first item in B_1. Assume that, we have packed the first $i-1$ items and we want to put the ith item in a bin. We do it in the following way:

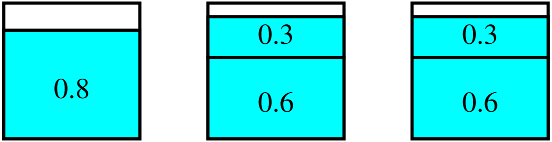
We go through the bins in the order of their creation time and we try to put the ith item in each of them. At the first occurrence when the item fits in a bin we put it there. If no bin has enough free space to store the ith item, then we create a new bin and place the ith item there.

Example: We have six objects of sizes 0.8, 0.6, 0.6, 0.3, 0.3, 0.3.



First Fit (FF) Algorithm

We pack the items incrementally. We create a bin B_1 and put the first item in B_1. Assume that, we have packed the first $i-1$ items and we want to put the ith item in a bin. We do it in the following way:
We go through the bins in the order of their creation time and we try to put the ith item in each of them. At the first occurrence when the item fits in a bin we put it there. If no bin has enough free space to store the ith item, then we create a new bin and place the ith item there.
Example: We have six objects of sizes 0.8, 0.6, 0.6, 0.3, 0.3, 0.3.

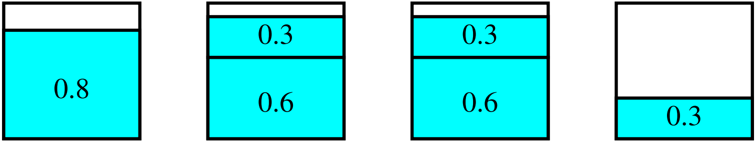


First Fit (FF) Algorithm

We pack the items incrementally. We create a bin B_1 and put the first item in B_1. Assume that, we have packed the first $i-1$ items and we want to put the ith item in a bin. We do it in the following way:

We go through the bins in the order of their creation time and we try to put the ith item in each of them. At the first occurrence when the item fits in a bin we put it there. If no bin has enough free space to store the ith item, then we create a new bin and place the ith item there.

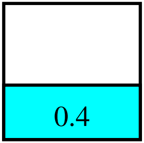
Example: We have six objects of sizes 0.8, 0.6, 0.6, 0.3, 0.3, 0.3.



Does First Fit always find an optimal solution?

Of course not, usually it does not find an optimal solution. Last time we were lucky.

Consider the following input: 4 Bins of sizes 0.4, 0.4, 0.6, 0.6:



Does First Fit always find an optimal solution?

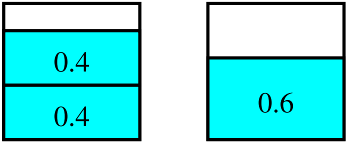
Of course not, usually it does not find an optimal solution. Last time we were lucky.

Consider the following input: 4 Bins of sizes 0.4, 0.4, 0.6, 0.6:

Does First Fit always find an optimal solution?

Of course not, usually it does not find an optimal solution. Last time we were lucky.

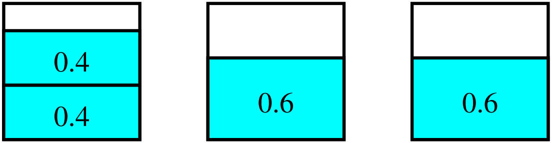
Consider the following input: 4 Bins of sizes 0.4, 0.4, 0.6, 0.6:



Does First Fit always find an optimal solution?

Of course not, usually it does not find an optimal solution. Last time we were lucky.

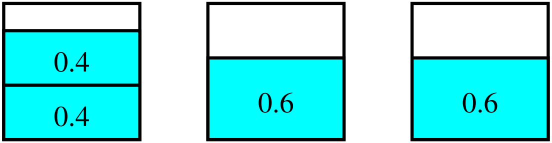
Consider the following input: 4 Bins of sizes 0.4, 0.4, 0.6, 0.6:



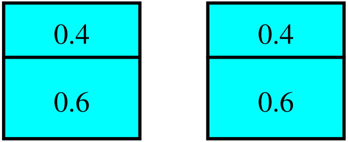
Does First Fit always find an optimal solution?

Of course not, usually it does not find an optimal solution. Last time we were lucky.

Consider the following input: 4 Bins of sizes 0.4, 0.4, 0.6, 0.6:



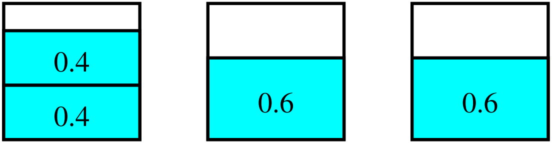
But the optimal solution is:



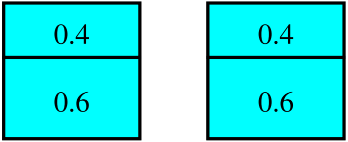
Does First Fit always find an optimal solution?

Of course not, usually it does not find an optimal solution. Last time we were lucky.

Consider the following input: 4 Bins of sizes 0.4, 0.4, 0.6, 0.6:



But the optimal solution is:



Idea: What if we try to pack bigger items first?