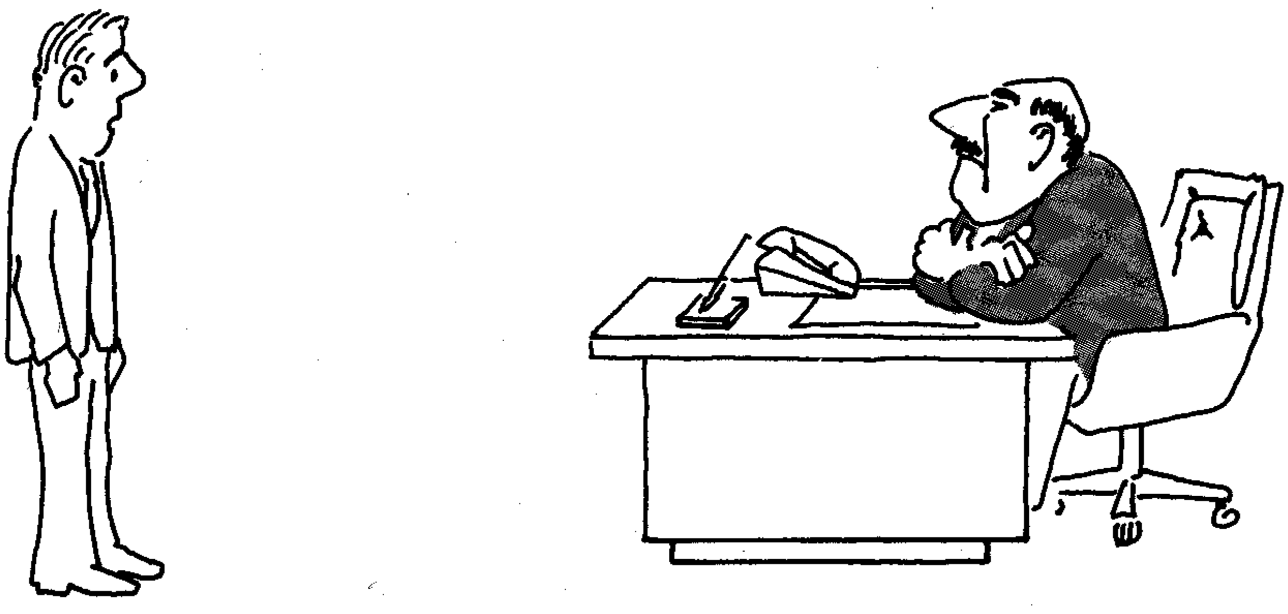
Approximation Algorithms

NP-completeness

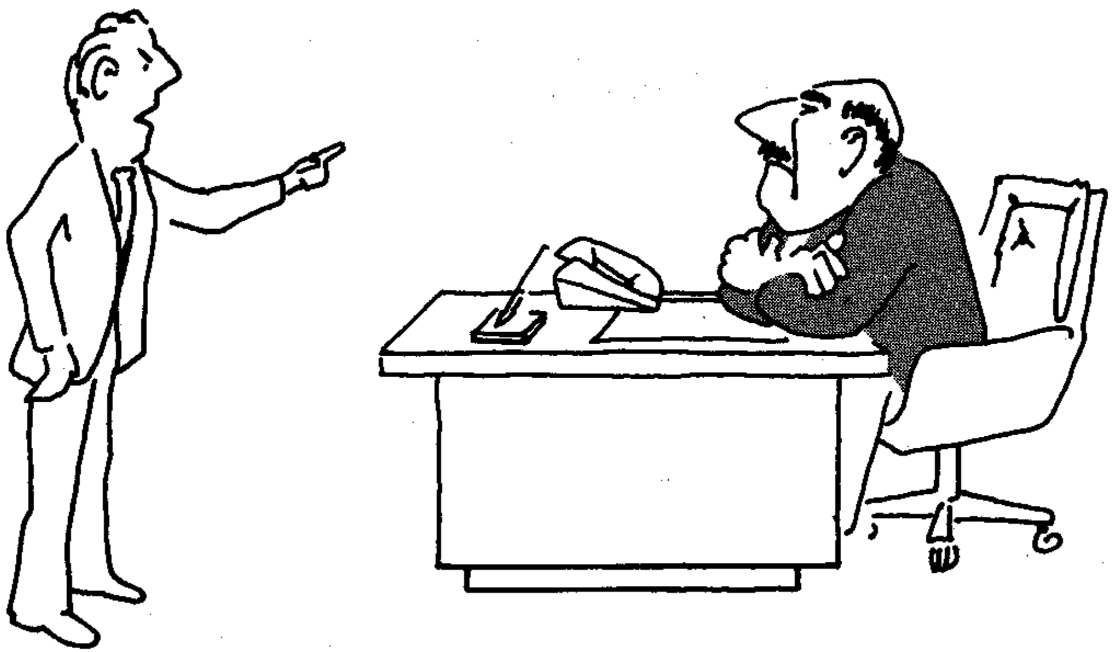


"I can't find an efficient algorithm, I guess I'm just too dumb."

NP-completeness

“I can’t find an efficient algorithm, but neither can all these famous people.”

Million dollar open problem: **Is P=NP?**



"I can't find an efficient algorithm, because no such algorithm is possible!"

Scenario

Your boss gives you a computationally hard problem

- No knowledge about approximation:
  - Spend a few months looking for an optimal solution
  - Come to their office and confess that you cannot do it
  - Get fired
- Knowledge about approximation
  - Show your boss that this is a NP-complete (NP-hard) problem. There does not exist any polynomial time algorithm to find an exact solution
  - Propose a good algorithm (either heuristic or approximation) to find a near-optimal solution
  - Better yet, prove the approximation ratio

Coping With NP-Compleness

Stick with small problems

- Develop clever enumeration strategies.
- Guaranteed to find optimal solution.
- No guarantees on running time, but problem is small.

Special Cases

- Look at specific types of input
- Find an algorithm that runs in polynomial time
- Gives correct solution.
  – Example: vertex cover in bipartite graphs, perfect graphs.

Heuristics.

- Develop intuitive algorithms.
- Guaranteed to run in polynomial time.
- No guarantees on quality of solution.

Coping With NP-Compleness

Approximation algorithms.

- Guaranteed to run in polynomial time.
- Guaranteed to find "high quality" solution, say within 1% of optimum.

Obstacle: need to prove a solution’s value is close to optimum, without even knowing what optimum value is!

Unwilling to relax correctness.

- Solve in exponential time but faster than brute force..
- Example:
  - Dynamic Programming for Knapsack O(nW)
  - Brute Force $O(2^n)$

Average Case .

- Find an algorithm which works well on average.
- Average running time is polynomial.

Approximation Algorithm

- Up to now, the best algorithm for solving an NP-complete problem requires exponential time in the worst case. It is too time-consuming.
- To reduce the time required for solving a problem, we can relax the problem, and obtain a feasible solution “close” to an optimal solution
- One compromise is to use heuristic solutions.
- The word “heuristic” may be interpreted as “educated guess.”

Approximation Algorithm

- An algorithm that returns near-optimal solutions is called an ***approximation algorithm***.

- **We need to find an *approximation ratio bound* for an approximation algorithm.**

Approximation Ratio bound

We say an approximation algorithm for the problem has a ratio bound of $\rho(n)$ if for any input size n, the cost C of the solution produced by the approximation algorithm is within a factor of $\rho(n)$ of the C^* of the optimal solution:

$$\max\{\frac{C}{C^*}, \frac{C^*}{C}\} = \rho(n)$$

This definition applies for both minimization and maximization problems.

Rho-α-ρ-approximation algorithm

- An approximation algorithm with an approximation ratio bound of ρ is called a ρ-approximation algorithm or a $(1+\varepsilon)$-approximation algorithm.

- Note that ρ is always larger than 1 and $\varepsilon=\rho-1$.
- Note some books may use α "alpha" instead of "rho"

Approximate Ratio

- C^* is the cost of optimal solution and C is the cost of an approximate algorithm
- $\rho(n)=\max(C/C^*, C^*/C)$ where n is size of problem input
- If $\rho(n)=1$, then the algorithm is an optimal algorithm
- The larger $\rho(n)$, the worse the algorithm

Greedy Approximations

- Use a greedy algorithm to solve the given problem
  - Repeat until a solution is found:
  - Among the set of possible next steps:
    Choose the current best-looking alternative and commit to it
- Usually fast and simple
- Works in some cases…(always finds optimal solutions)
  - Dijsktra's single-source shortest path algorithm
  - Prim's and Kruskal's algorithm for finding MSTs
- but not in others…(may find an approximate solution)
  - TSP – always choosing current least edge-cost node to visit next

Vertex-cover problem

- Vertex cover: given an undirected graph $G=(V,E)$, then a subset $V' \in V$ such that if $(u,v) \in E$, then $u \in V'$ or $v \in V'$ (or both).
- Size of a vertex cover: the number of vertices in it.
- Vertex-cover problem: find a vertex-cover of minimal size.

Vertex-cover problem

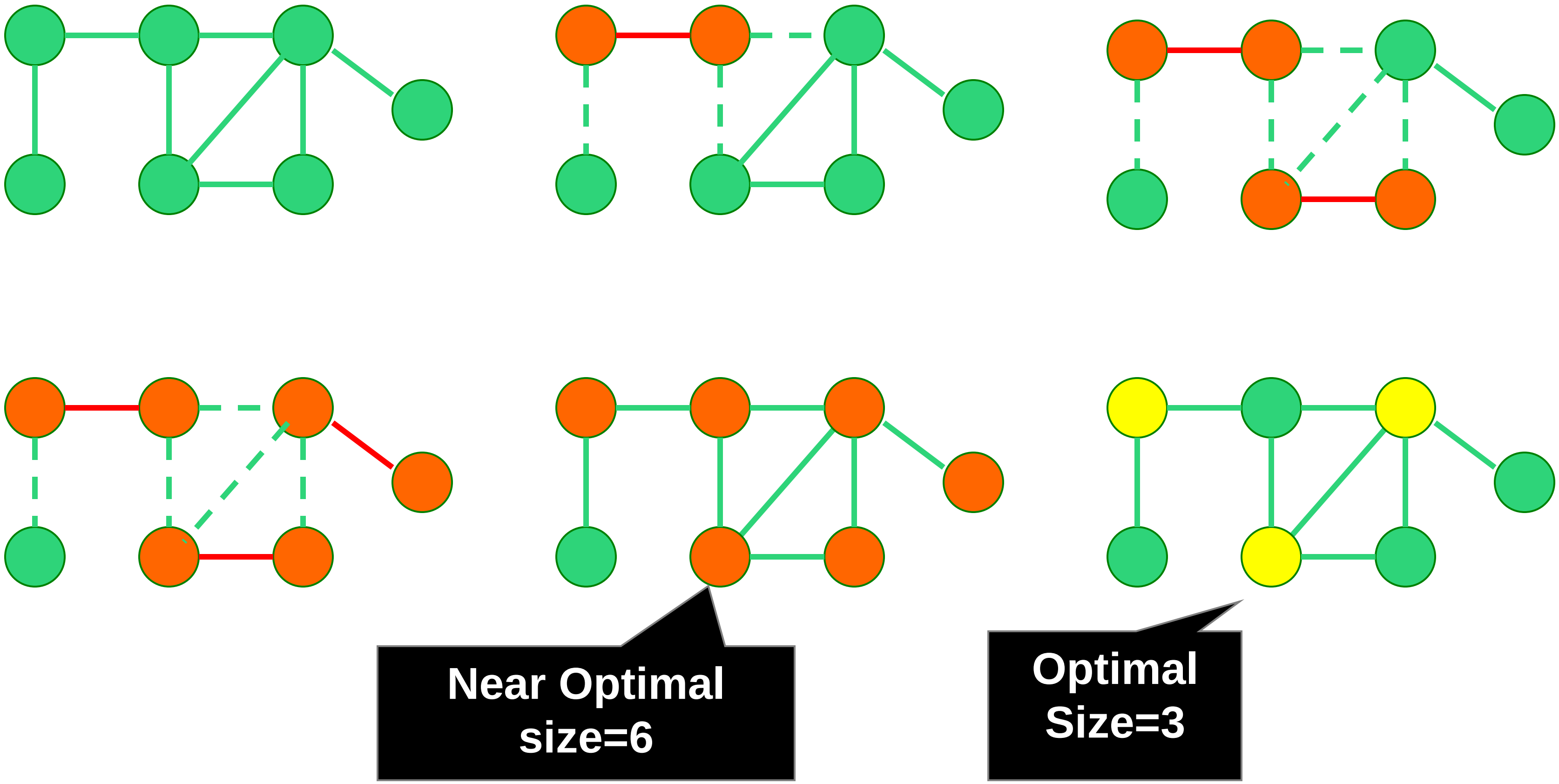
- Vertex-cover problem is NP-complete. (See section 34.5.2).
  - Vertex-cover belongs to NP.
  - Vertex-cover is NP-hard (CLIQUE$\leq_P$vertex-cover.)
    - Reduce $<G,k>$ where $G=<V,E>$ of a CLIQUE instance to $<G',|V|-k>$ where $G'=<V,E'>$ where $E'=\{(u,v): u,v\in V, u\neq v \text{ and } <u,v>\notin E\}$ of a vertex-cover instance.
- So find an approximate algorithm.

Approximate vertex-cover algorithm

```
APPROX-VERTEX-COVER(G)

C ← Ø
E′ ← E[G]
while E′ ≠ Ø
    do let (u, v) be an arbitrary edge of E′
       C ← C U {u, v}
            remove every edge in E′ incident on u or v
return C
```

The vertex-cover problem



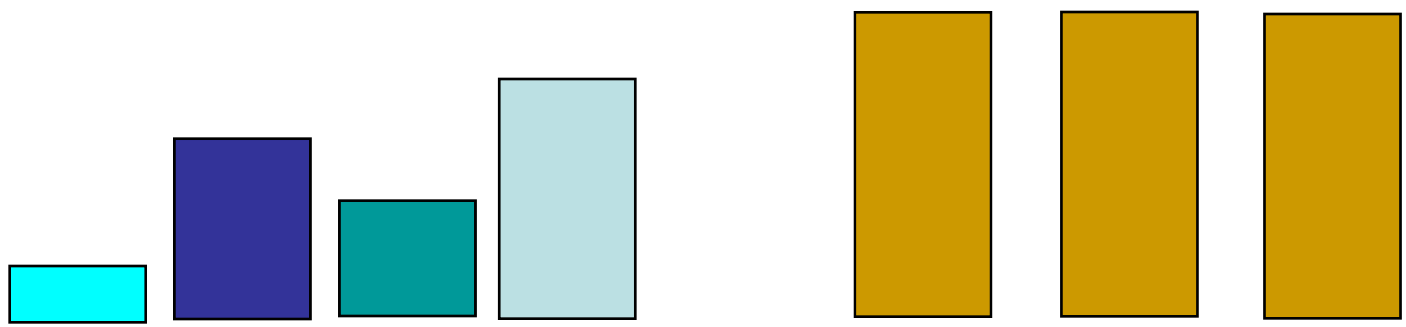
2-approximate vertex-cover

- *Theorem 35.1* (page 1026).
  - APPROXIMATE-VERTEX-COVER is a poly time 2-approximate algorithm, i.e., the size of returned vertex cover set is at most twice of the size of optimal vertex-cover.
- Proof:
  - It runs in poly time
  - The returned C is a vertex-cover.
  - Let A be the set of edges picked in line 4 and C* be the optimal vertex-cover.
    - Then C* must include at least one end of each edge in A and no two edges in A are covered by the same vertex in C*, so $|C^*| \geq |A|$.
    - Moreover, $|C|=2|A|$, so $|C| \leq 2|C^*|$.

The Grocery Bagging Problem

- You are an environmentally-conscious grocery bagger at Fred Meyers
- You would like to minimize the total number of bags needed to pack each customer's items.

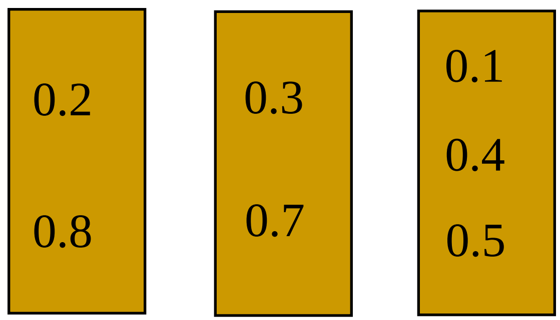
Items (mostly junk food) Grocery bags



Sizes $a_1, a_2, \ldots, a_n$ $(0 < a_i \leq 1)$ Size of each bag = 1

Optimal Grocery Bagging: An Example

- Example: Items = 0.5, 0.2, 0.7, 0.8, 0.4, 0.1, 0.3
  - How may bags of size 1 are required?



Only 3 bags required

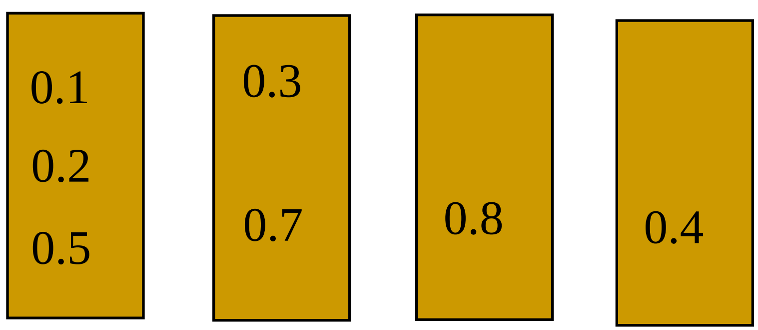
- Can find optimal solution through exhaustive search
  - Search all combinations of N items using 1 bag, 2 bags, etc.
  - Takes **exponential time!**

Greedy Grocery Bagging

- Greedy strategy #1 "**First Fit**":
  1. Place each item in first bin large enough to hold it
  2. If no such bin exists, get a new bin
- **Example**: Items = 0.5, 0.2, 0.7, 0.8, 0.4, 0.1, 0.3

Greedy Grocery Bagging

- Greedy strategy #1 "**First Fit**":
  1. Place each item in first bin large enough to hold it
  2. If no such bin exists, get a new bin
- **Example**: Items = 0.5, 0.2, 0.7, 0.8, 0.4, 0.1, 0.3



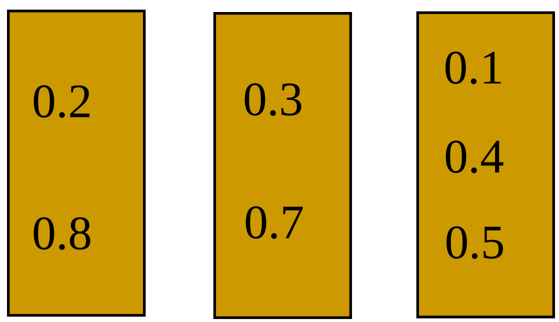
Uses 4 bins
Not optimal

Better Greedy Grocery Bagging

- Greedy strategy #2 "First Fit Decreasing":
  1. Sort items according to *decreasing size*
  2. Place each item in first bin large enough to hold it
- **Example**: Items = 0.5, 0.2, 0.7, 0.8, 0.4, 0.1, 0.3

Better Greedy Grocery Bagging

- Greedy strategy #2 "First Fit Decreasing":
  1. Sort items according to *decreasing size*
  2. Place each item in first bin large enough to hold it
- **Example**: Items = 0.5, 0.2, 0.7, 0.8, 0.4, 0.1, 0.3



Uses 3 bins
Optimal in this case
Not optimal in general

- **Approximation Result**: If OPT is the optimal number of bins, First Fit Decreasing never uses more than **1.2OPT + 4** bins

Bin Packing–Dec. is NP-complete

Bin Packing problem: Given n items of sizes a_1, $a_2, \ldots, a_n$ ($0 < a_i \leq 1$), pack these items in the at most k bins of size 1.

- Bin packing in in NP
  - To verify a solution
    - Add the weights of the items in each bin.
    - Each bin must contain < 1unit.
    - Check that each item is in a bin
    - There are at most k bins used.
  - This can be done in O(n).

- SET-PARTITION reduces to Bin Packing

Bin Packing–Dec. is NP-complete

SET-PARTITION: Given a set of numbers X = { $x_1, x_2, \ldots x_k$ }. Is there a subset of X, B, such that the sum of the elements in B is equal to the sum of the elements in S-B.

Bin Packing : Given n items of sizes $a_1, a_2, \ldots, a_n$ ($0 < a_i \leq 1$), pack these items in the at most k bins of size 1.

SET-PARTITION $\leq_p$ Bin Packing

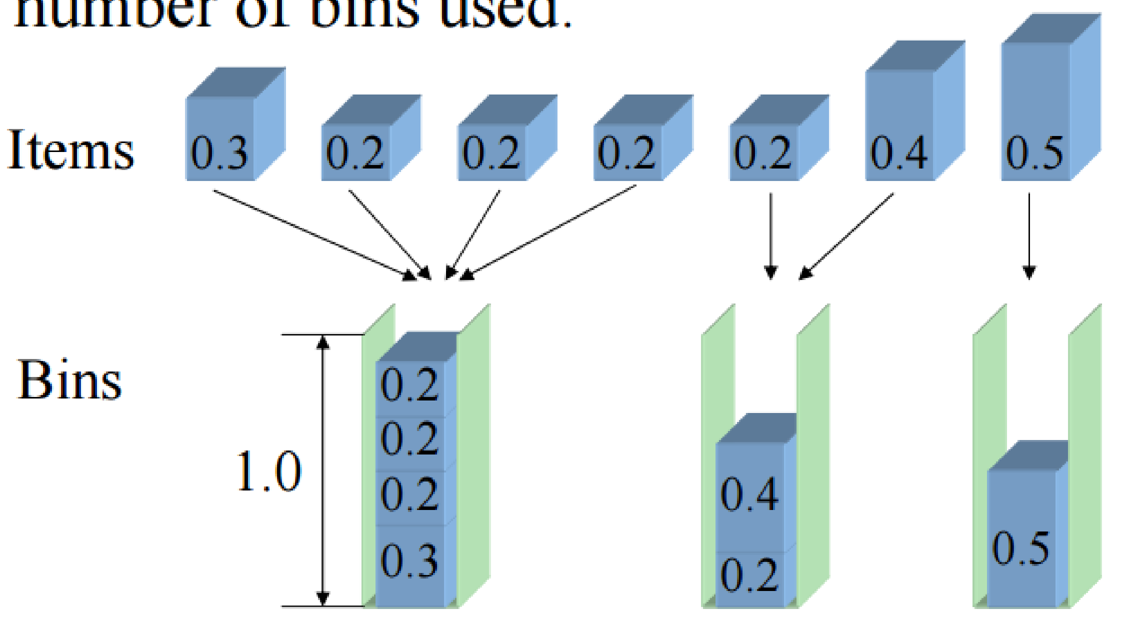
Let sum = $\sum_{i=1}^{k} x_i$. Define S = { $s_1, s_2, \ldots s_k$ }

where $s_i = \frac{2x_i}{sum}$ for i = 1, ..k. Then if { $s_1, s_2, \ldots s_k$ } can be packed into 2 bins, X can be partitioned into 2.

Thus Bin Packing is NP-Complete

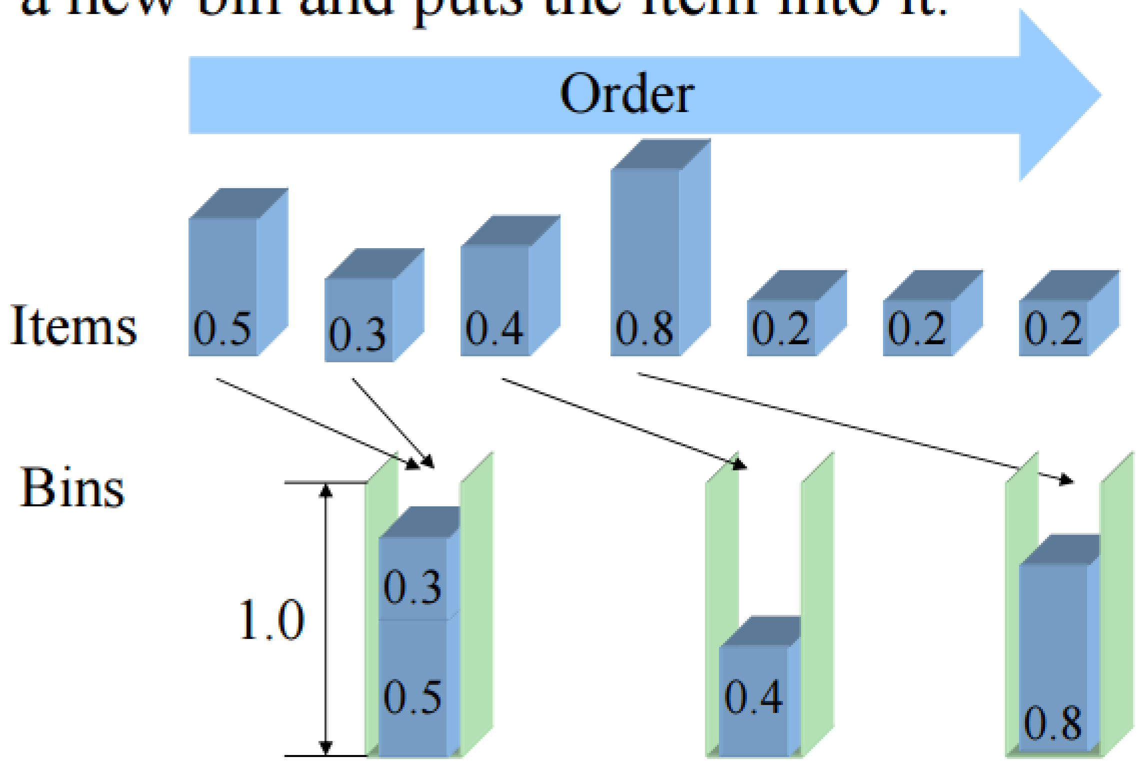
Bin packing

- Input:
  – n items with sizes $a_1, \ldots, a_n$ $(0 < a_i \leq 1)$.
- Task:
  – Find a packing in unit-sized bins that minimizes the number of bins used.



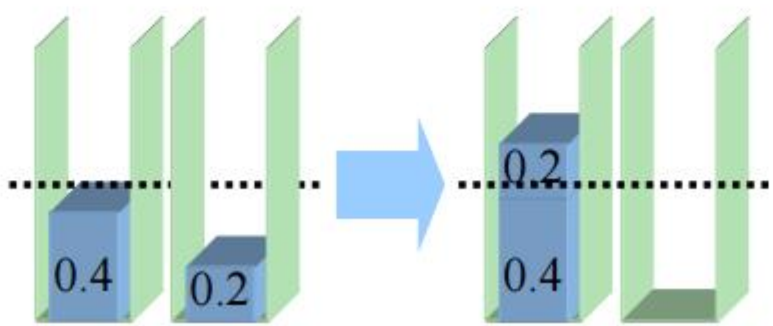
First-Fit Algorithm

- This algorithm puts each item in one of partially packed bins.
  - If the item does not fit into any of these bins, it opens a new bin and puts the item into it.



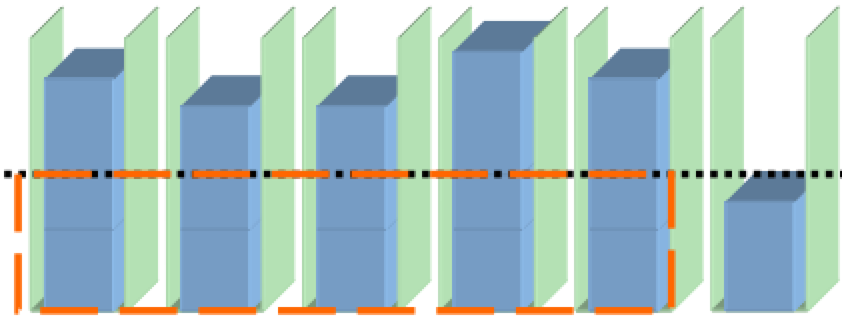
First-Fit finds 2OPT solution

- OPT: # bins used in the optimal solution.
- [Proof]
  - Suppose that First-Fit uses m bins.
  - Then, at least (m-1) bins are more than half full.
    - We never have two bins less than half full.
      - If there are two bins less than half full, items in the second bin can be substituted into the first bin by First-Fit.



First-Fit finds 2OPT solution

- Suppose that First-Fit uses m bins.
- Then, at least (m-1) bins are more than half full.



Sum of sizes of the items

$$\mathrm{OPT} \geq \sum_{i=1}^{n} a_i > \frac{m-1}{2}$$

The size of

$$2\mathrm{OPT} > m - 1$$

Since m and OPT are integers. $\longrightarrow$ $$2\mathrm{OPT} \geq m$$

Knapsack

- Given: a set S of n objects with weights and values, and a weight bound:
  - $w_1, w_2, \ldots, w_n, W$ (weights, weight bound).
  - $b_1, b_2, \ldots, b_n$ (benefit).

- Find: subset of S with total weight at most B, and maximum total value.

Formally: $$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

- Problem is known to be NP-hard

Assumptions

- $\forall i,\ w_i \leq W$ (every item can be added to T)
- $\forall i,\ b_i > 0$. (non-negative benefits)
- benefits, weights, and bound are all integers.

Note:

- This is a maximum problem.
- Define: OPT = The optimal solution.

- We will see a 2-approximation for two versions of knapsack.

Basic Greedy Algorithm

1.

Knapsack Approximation

Greedy Algorithm

- Define ratio: $v_i = \frac{b_i}{w_i}$ for item i
- Sort items in non-increasing order such that

  $v_{(1)} \geq v_{(2)} \geq \ldots \geq v_{(n)}$.
- Greedily pick items in above order until

  A = {item(1) , item(2) , …, item(j) } with weight(A) ≤ W and weight(A) + $w_{(j+1)}$ > W

Knapsack Approximation

- Seems like a quick approximation algorithm but it can be very “bad”
- Consider the small example
  - Item 1: weight 1 and benefit 2, value v = 2/1 = 2
  - Item 2: weight W and benefit W, value = W/W = 1

The greedy algorithm picks the small Item 1with benefit 2 since it has the larger value ratio.

This leaves W-1 empty space in the backpack.

However there is no room for the large item which would have filled the entire backpack and given us total benefit W.

This is W/2 off from optimal.

Knapsack 2-Approximation

Greedy2 Algorithm

- Sort items in non-increasing order such that

  $v_{(1)} \geq v_{(2)} \geq \ldots \geq v_{(n)}$.

- Greedily pick items in above order until

  A = {item(1) , item(2) , …, item(j) } with weight(A) > W and

  weight(A) + $w_{(j+1)}$ > W

- Pick the better of

  A = {item(1) , item(2) , …, item(j_j } and {item(j+1)}

  if TotalBenefits(A) > $b_{(j+1)}$ return A

  else return item(j+1)

Greedy2 is 2-approximation Knapsack

Proof: We used a greedy algorithm so if the solution is suboptimal then we must have some leftover space at the end. Let S be the total weight of the solution A, then the left over space is W – S. Imagine we were able to take a fraction of an item. Then by adding $\frac{W-S}{w_{j+1}} b_{j+1}$ to our knapsack's total benefit, we would either match or exceed OPT (remember OPT is unable to use fractions).

So OPT $\leq \sum_{i=1}^{j} b_i + \frac{W-S}{w_{j+1}} b_{j+1} \leq \sum_{i=1}^{j} b_i + b_{j+1}$ since we couldn't fit the entire (j+1)st item in the knapsack $\frac{W-S}{w_{j+1}} < 1$.

OPT $\leq \sum_{i=1}^{j} b_i + b_{j+1} \leq$ 2 max $(\sum_{i=1}^{j} b_i,\ b_{j+1})$ since Greedy2 solution is 2 max $(\sum_{i=1}^{j} b_i,\ b_{j+1})$

½ OPT $\leq$ Greedy2. or $max\left\{\frac{OPT}{Greedy2}, \frac{Greedy2}{OPT}\right\}$ = 2