Greedy Algorithms

Kumkum Saxena

Greedy Algorithms

- Introduction
- Knapsack problem
- Job sequencing with deadlines
- Optimal storage on tapes
- Optimal merge pattern
- Analysis of All problems

Optimization problems

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution
- A “greedy algorithm” sometimes works well for optimization problems
- A greedy algorithm works in phases. At each phase:
  - You take the best you can get right now, without regard for future consequences
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum

Example: Counting money

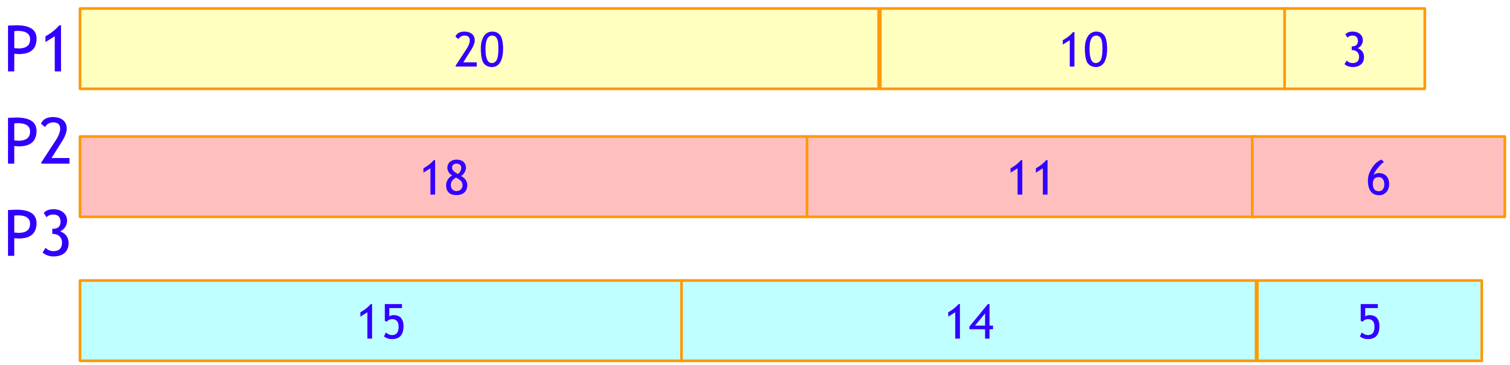
- Suppose you want to count out a certain amount of money, using the fewest possible bills and coins
- A greedy algorithm would do this would be:
- At each step, take the largest possible bill or coin that does not overshoot
- Example: To make $6.39, you can choose:
  - a $5 bill
  - a $1 bill, to make $6
  - a 25¢ coin, to make $6.25
  - A 10¢ coin, to make $6.35
  - four 1¢ coins, to make $6.39
- For US money, the greedy algorithm always gives the optimum solution

A failure of the greedy algorithm

- In some (fictional) monetary system, “krons” come in 1 kron, 7 kron, and 10 kron coins
- Using a greedy algorithm to count out 15 krons, you would get
  - A 10 kron piece
  - Five 1 kron pieces, for a total of 15 krons
- This requires six coins
- A better solution would be to use two 7 kron pieces and one 1 kron piece
  - This only requires three coins
- The greedy algorithm results in a solution, but not in an optimal solution

A scheduling problem

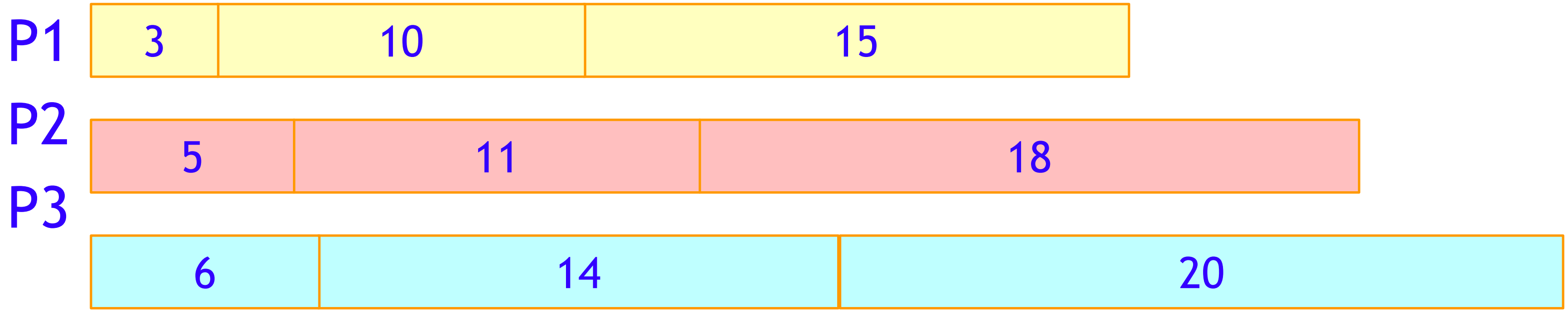
- You have to run nine jobs, with running times of 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes
- You have three processors on which you can run these jobs
- You decide to do the longest-running jobs first, on whatever processor is available



- Time to completion: 18 + 11 + 6 = 35 minutes
- This solution isn't bad, but we might be able to do better

Another approach

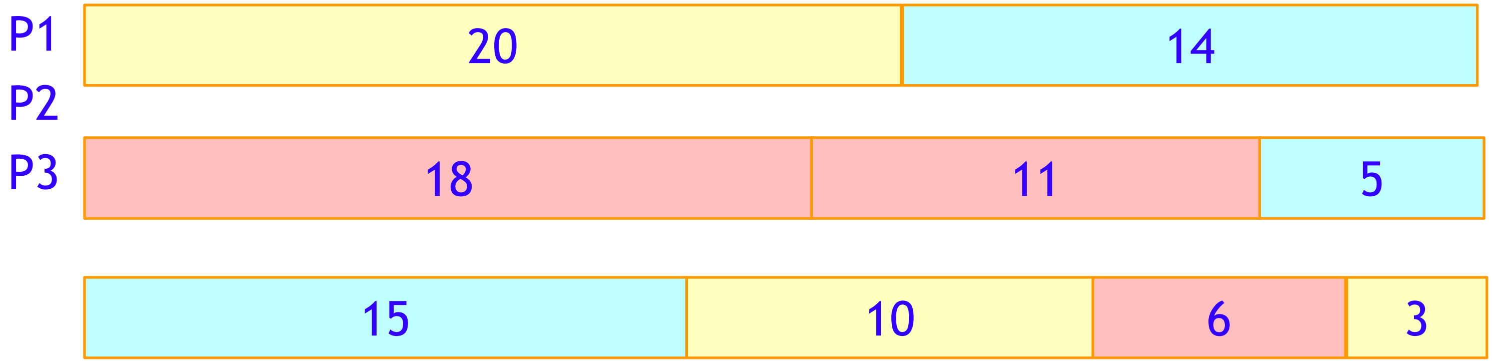
- What would be the result if you ran the *shortest* job first?
- Again, the running times are 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes



- That wasn't such a good idea; time to completion is now
- 6 + 14 + 20 = 40 minutes
- Note, however, that the greedy algorithm itself is fast
  - All we had to do at each stage was pick the minimum or maximum

An optimum solution

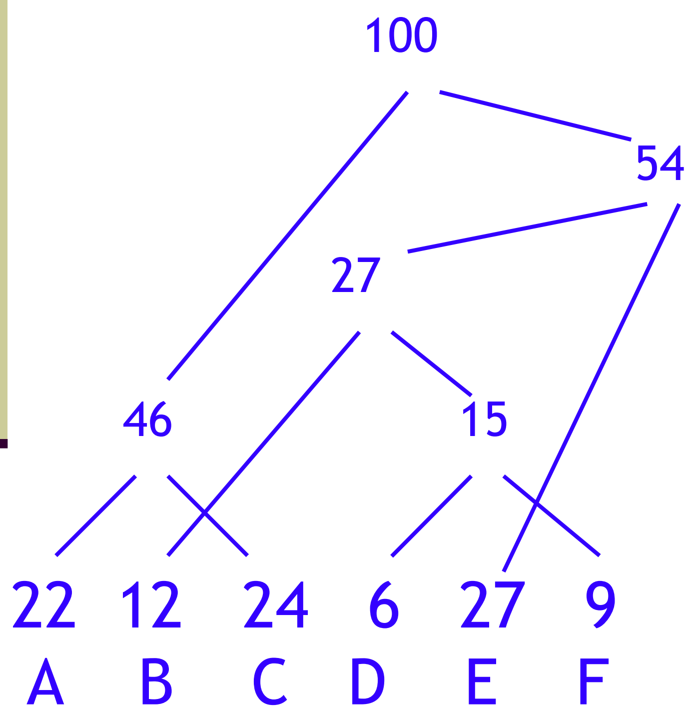
- Better solutions do exist:



- This solution is clearly optimal (why?)
- Clearly, there are other optimal solutions (why?)
- How do we find such a solution?
  - One way: Try all possible assignments of jobs to processors
  - Unfortunately, this approach can take exponential time

Huffman encoding

- The Huffman encoding algorithm is a greedy algorithm
- You always pick the two smallest numbers to combine



A=00
B=100
C=01
D=1010
E=11
F=1011

- Average bits/char:
- 0.22*2 + 0.12*3 +
- 0.24*2 + 0.06*4 +
- 0.27*2 + 0.09*4
- = 2.42
- The Huffman algorithm finds an optimal solution

OPTIMIZATION PROBLEM

- An optimization problem:
  - Given a problem instance, a set of constraints and an objective function.
  - Find a feasible solution for the given instance.
    - either maximum or minimum depending problem being solved.
    - constraints specify the limitations on the required solutions.

SOLUTION FOR OPTIMIZATION PROBLEM

- For some optimization problems,
  - Dynamic Programming is “overkill”
  - Greedy Strategy is simpler and more efficient

DYNAMIC PROGRAMMING VS GREEDY

Dynamic Programming	Greedy Algorithm
At each step, the choice is determined based on solutions of sub problems.	At each step, we quickly make a choice that currently looks best. A local optimal (greedy) choice
Sub-problems are solved first.	Greedy choice can be made first before solving further sub-problems.
Bottom-up approach	Top-down approach
Can be slower, more complex	Usually faster, simpler

GREEDY METHOD

- Characteristics of greedy algorithm:
  - make a sequence of choices
  - each choice is the one that seems best so far, only depends on what's been done so far
  - choice produces a smaller problem to be solved

PHASES OF GREEDY ALGORITHM

- A greedy algorithm works in phases.
- At each phase:
  - takes the best solution right now, without regard for future consequences
  - choosing a *local* optimum at each step, and end up at a *global* optimum solution.

The Greedy Method

- The greedy approach does *not always* lead to an *optimal solution*.
- The problems that have a greedy solution are said to posses the greedy-choice property.
- The *greedy approach* is also used in the context of hard (*difficult to solve*) problems in order to generate an approximate solution.

KNAPSACK PROBLEM

There are two version of knapsack problem

1.0-1 knapsack problem:

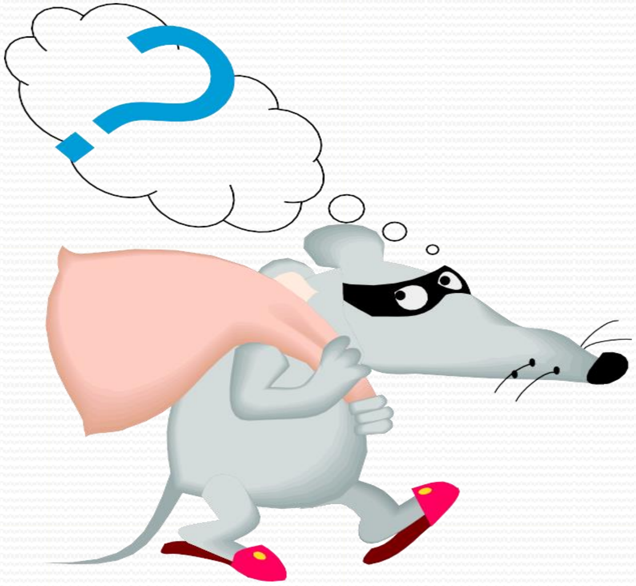
- Items are indivisible. (either take an item or not)
- can be solved with dynamic programming.

2.Fractional knapsack problem:

- Items are divisible. (can take any fraction of an item)
- It can be solved in greedy method

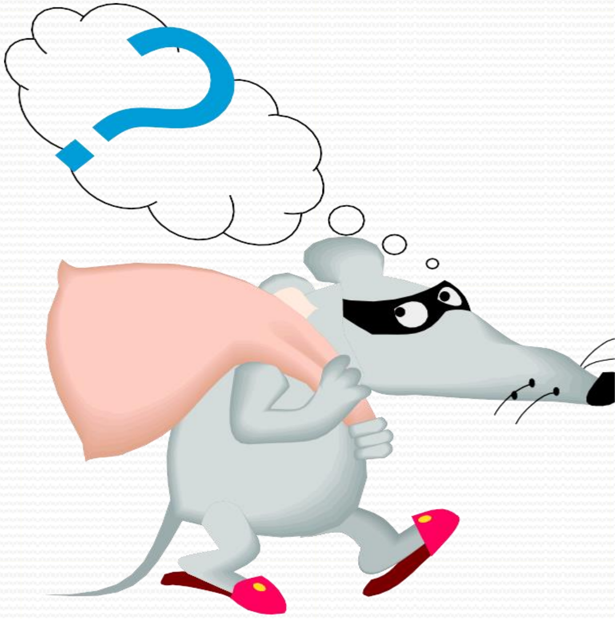
0-1 KNAPSACK PROBLEM:

- A thief robbing a store finds n items.
- i^{th} item: worth v_i value of item and weight of item w_i
- W, w_i, v_i are integers.
- He can carry at most W pounds.



FRACTIONAL KNAPSACK PROBLEM:

- A thief robbing a store finds n items.
- i^{th} item: worth v_i value of item and w_i weight of item
- W, w_i, v_i are integers.
- He can carry at most W pounds.



- He can take fractions of items.

THE OPTIMAL KNAPSACK ALGORITHM

- **Input:**
  - an integer n
  - positive values w_i and v_i such that $1 \leq i \leq n$
  - positive value W.
- **Output:**
  - n values of x_i such that $0 \leq x_i \leq 1$
  - Total profit

THE OPTIMAL KNAPSACK ALGORITHM

- Initialization:
  - Sort the n objects from large to small based on their ratios v_i / w_i.
  - We assume the arrays w[1..n] and v[1..n] store the respective weights and values after sorting.
  - initialize array x[1..n] to zeros.
  - weight = 0; i = 1;

THE OPTIMAL KNAPSACK ALGORITHM

while ($i \leq n$ and weight $< W$) do

if weight $+ w[i] \leq W$ then

$x[i] = 1$

else

$x[i] = (W -$ weight$)$ / w$[i]$

weight = weight + x$[i]$ * w$[i]$

i++

KNAPSACK - EXAMPLE

Problem:

$n = 3$

$W = 20$

$(v_1, v_2, v_3) = (25, 24, 15)$

$(w_1, w_2, w_3) = (18, 15, 10)$

KNAPSACK - EXAMPLE

Solution:

- Optimal solution:
  - $x_1 = 0$
  - $x_2 = 1$
  - $x_3 = 1/2$
- Total profit = 24 + 7.5 = 31.5

Fractional Knapsack Problem

Algorithm FractionalKnapsack(S, W):

Input: Set S of items, such that each item $i \in S$ has a positive benefit b_i and a positive weight w_i; positive maximum total weight W

Output: Amount x_i of each item $i \in S$ that maximizes the total benefit while not exceeding the maximum total weight W

for each item $i \in S$ **do**

  $x_i \leftarrow 0$

  $v_i \leftarrow b_i / w_i$ {***value index*** of item i}

$w \leftarrow 0$ {total weight}

while $w < W$ **do**

  remove from S an item i with highest value index   {greedy choice}

  $a \leftarrow \min\{w_i, W - w\}$ {more than $W - w$ causes a weight overflow}

  $x_i \leftarrow a$

  $w \leftarrow w + a$

Fractional Knapsack Problem

- In the solution we use a heap-based *PQ* to store the items of S, where the *key* of each item is its *value index*
- With *PQ*, each greedy choice, which removes an item with the greatest value index, takes *O(log n)* time
- The *fractional knapsack algorithm* can be implemented in time *O(n log n)*.

Fractional Knapsack Problem

- *Fractional knapsack problem* satisfies the *greedy-choice property*, hence
- Thm: Given an instance of a fractional knapsack problem with set S of n items, we can construct a maximum benefit subset of S, allowing for fractional amounts, that has a total weight W in $O(n \log n)$ time.

Examples

Consider 5 items along their respective weights and values: -

$I = (I_1, I_2, I_3, I_4, I_5)$

$w = (5, 10, 20, 30, 40)$

$v = (30, 20, 100, 90, 160)$

The capacity of knapsack $W = 60$

Find the optimal solution for the fractional knapsack problem making use of greedy approach. Consider-

n = 5

w = 60 kg

(w1, w2, w3, w4, w5) = (5, 10, 15, 22, 25)

(b1, b2, b3, b4, b5) = (30, 40, 45, 77, 90)

- So, our knapsack will contain the items-

< I1 , I2 , I5 , (20/22) I4 >

Now,

Total cost of the knapsack

= 160 + (20/27) x 77= 160 + 70== 230 units

Example: Knapsack Capacity W = 30 and

Item	A	B	C	D
Value	50	140	60	60
Size	5	20	10	12
Ratio	10	7	6	5

- Solution:
- All of A, all of B, and ((30-25)/10) of C (and none of D)
- Size: 5 + 20 + 10*(5/10) = 30
- Value: 50 + 140 + 60*(5/10) = 190 + 30 = 220

Analysis

- A greedy algorithm typically makes (approximately) n choices for a problem of size n
    - (The first or last choice may be forced)
- Hence the expected running time is:
- O(n * O(choice(n))), where choice(n) is making a choice among n objects
    - Counting: Must find largest useable coin from among k sizes of coin (k is a constant), an O(k)=O(1) operation;
        - Therefore, coin counting is (n)
    - Huffman: Must sort n values before making n choices
        - Therefore, Huffman is O(n log n) + O(n) = O(n log n)
    - Minimum spanning tree: At each new node, must include new edges and keep them sorted, which is O(n log n) overall
        - Therefore, MST is O(n log n) + O(n) = O(n log n)