Greedy Algorithm

Tanjina Helaly

DESIGNING AND ANALYZING EFFICIENT ALGORITHMS

- important techniques used
  - **divide-and-conquer,**
  - randomization,
  - **recursion.**
  - **dynamic programming (Chapter 15),**
  - **greedy algorithms (Chapter 16),** and
  - amortized analysis (Chapter 17).
- Among these the last 3 are use for optimization
- What is optimization?
  - the action of making the best or most effective use of a situation or resource.

Greedy algorithm

- A greedy algorithm is a mathematical process that
  - looks for simple, easy-to-implement solutions to complex, **multi-step** problems
  - by deciding **which next step** will provide the **most** obvious **benefit**.
- Such algorithms are called greedy because
  - *it always makes **the choice that looks best at*** the moment.
  - the algorithm doesn’t consider the larger problem as a whole.
  - Once a decision has been made, it is never reconsidered.

Greedy algorithm

- A ***greedy*** *algorithm* makes a **locally optimal** choice **in the hope** that this choice will **lead to a globally optimal** solution.
- Greedy algorithms do not always yield optimal solutions, but for many problems they do.

How do you decide which choice is optimal?

- For any optimization there are 2 key things.
  - An objective function
    - Normally maximize or minimize something
  - A set of constraints
    - What resources and limitation we have.

HOW DO YOU DECIDE WHICH CHOICE IS OPTIMAL?

- Assume that you have an **objective function** that needs to be optimized (either maximized or minimized) at a given point.
- A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized.
- The Greedy algorithm has only one shot to compute the optimal solution so that **it never goes back and reverses the decision**.

Steps of Greedy algorithm

- Make greedy choice at the beginning of each iteration
- Create sub problem
- Solve the sub problem
  - How?
  - Continue first two steps until the all the sub-problems are solved.

Activity selection problem

Activity Selection Problem

- The **activity selection problem** is a classic optimization problem concerning the selection of **non-conflicting** activities to perform within a given time frame.
- The problem is to select the **maximum number of activities** that can be **performed by a single** person or machine, assuming that a person can only **work on a single activity at a time**.
- A classic application of this problem is in **scheduling** a room for multiple competing events, each having its own time requirements (start and end time).

ACTIVITY SELECTION PROBLEM- EXAMPLE

- Suppose we have a set of activities $\{a_1; a_2; \ldots ;a_n \}$ ***that wish to use a resource, such as a lecture hall, which*** can serve only one activity at a time.
- Each activity a_i has a ***start time*** s_i ***and a finish time*** f_i***, where*** $0 < s_i < f_i < a$ **.**
- We have to select the **maximum-size subset** of activities that are mutually **compatible**.
- Two activities are **compatible if** their intervals **do not overlap**.

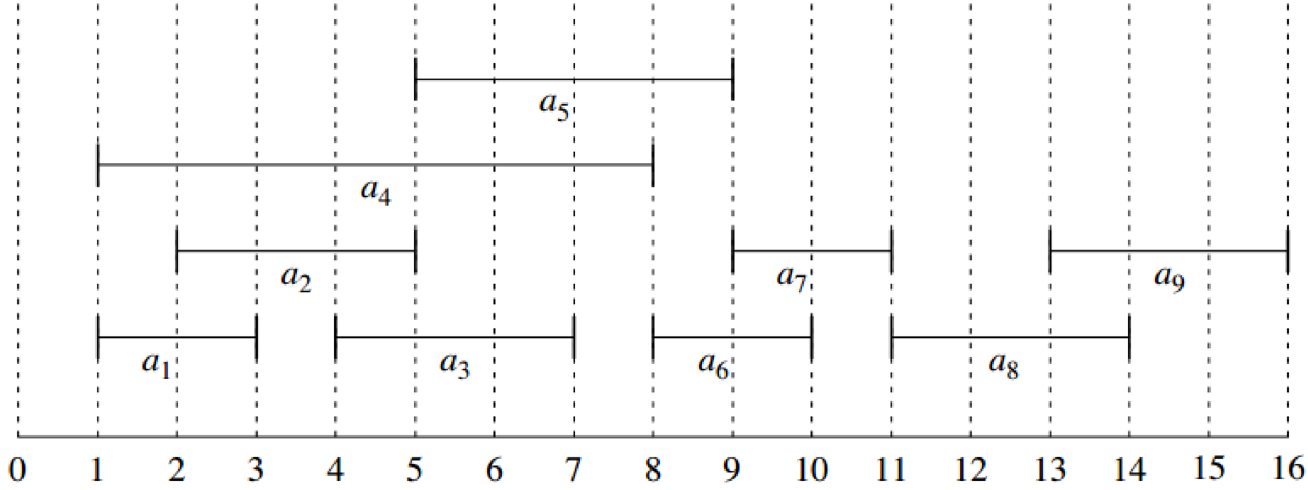
Activity Selection Problem- Example

- We assume that the activities are sorted in monotonically increasing order of finish time:
  - $f_1 <= f_2 <= f_3 <= .... <= f_{n-1} <= fn$:
- Subset $\{a_1; a_4; a_8; a_{11}\}$ is better than the subset $\{a_3; a_9; a_{11}\}$

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Activity Selection Problem- Another Example

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



Maximum-size mutually compatible set: $\{a_1, a_3, a_6, a_8\}$.

Not unique: also $\{a_2, a_5, a_7, a_9\}$.

How to solve?

- Brute force
  - Generate all possible subsets of non-conflicting activities
  - Choose the largest subset
- Greedy algorithm
  - Steps:
    - Make greedy choice at the beginning of each iteration
    - Create sub problem
    - Solve the sub problem

Activity Selection Problem- Greedy Algorithm

- **Make greedy choices:** select a job to start with. Which one?
  - Select the job that finishes first
  - **Assumption:** Jobs are sorted according to finishing time.
- **Create sub problems:** leaving this job, leaves you with a smaller number of jobs to be selected.
- **Solve Sub problems:** Continue first two steps until the all the jobs are finished. (recursion!)

ACTIVITY SELECTION PROBLEM – ITERATIVE SOLUTION

- *s -> the set of start time*
- *f -> the set of finish time.*

```
GREEDY-ACTIVITY-SELECTOR(s, f)
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}
        k = m
return A
```

Running time complexity = O(n)

Activity Selection Problem – recursive solution

- *s -> the set of start time*
- *f -> the set of finish time.*
- *k -> index of last selected activity*
- *n -> number of activities*

Running time complexity = O(n)

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
m = k + 1
while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return Ø
```

- *Note:In order to start, we add the fictitious activity a_0 with $f_0 = 0$, so that subproblem S_0 is the entire set of activities S. The initial call, which solves the entire problem, is RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n)*

Coin changing problem

Coin Changing Problem

- Suppose you have different kinds of coin of quarters(25 cents), dimes(10 cents), nickels (5 cents) , and pennies(1 cent).
- Consider the problem of **making change for n cents using the fewest number of coins**. Assume that each coin's value is an integer.
- So, we need to find the minimum number of coins that add up to a given amount of money.

Coin Changing Problem

- **Goal**: Convert some amount of money **n** into given denominations, using the fewest possible number of coins

- **Input**: An amount of money **n**, and an array of **d** denominations **c** = (c_1, c_2, …, c_d), in a decreasing order of value ($c_1 > c_2 > \ldots > c_d$)

- **Output**: A list of d integers i_1, i_2, …, i_d such that

  $$c_1 i_1 + c_2 i_2 + \ldots + c_d i_d = \mathbf{n}$$

  and $i_1 + i_2 + \ldots + i_d$ is minimal

Solution

- **Make greedy choices:** Select the coin with max value smaller or equal to the amount, this should lead to minimum number of coins.
  - Try the 25 cent first!
- **Create sub problems:** Giving out the first coin, leaves you with a smaller amount.
- **Solve Sub problems**: Continue first two steps until the change is not given. (recursion!)

Coin Changing problem – recursive solution

- *n -> The change needed*
- *v -> the list of coins sorted in* ***descending order****. So, the max value coin will be at first index, then the next smaller and so on. [O(mlogm)]*
- *i -> index*

```
GreedyRecursiveCoinChange(n, v[], i)
if n > 0
    if v[i] ≤ n
        Print(v[i])
        return 1 + GreedyRecursiveCoinChange(n − v[i], v, i)
    else
        return GreedyRecursiveCoinChange(n, v, i + 1)
else
    return 0
```

Running time complexity = O(n)

Running time complexity including sorting = O(mlogm)+O(n)

Coin Changing problem – iterative solution

- *n -> The change needed*
- *v -> the list of coins sorted in **descending order**. So, the max value coin will be at first index, then the next smaller and so on. [O(mlogm)]*

```
GreedyIterativeCoinChange(n, v[])
val = 0, i = 0
while n > 0 and i < v.length − 1
    if v[i] ≤ n
        tVal = ⌊n/v[i]⌋
        Print(v[i] + 'cent : ' + tVal + 'times')
        val+ = tVal
        n = n − tVal ∗ v[i]
    i + +
return val
```

Running time complexity = O(n)

Running time complexity including sorting = O(mlogm) + O(n)

Try the following

- **Case 1:**
  - Make a change for 12 cents when you have only 4 kinds of coins - 10, 8, 4, and 1
  - **Does it give you optimal solution?**
- **Case 2:**
  - You do not have the 5 cent coin. So, the coin set has 25 cent, 10 cent and 1 cent. Now give a change for 30 cent.
  - **Does it give you optimal solution?**
- What can you conclude to?
  - **Is greedy algorithm good for coin changing problem?**

Knapsack problem

What is knapsack problem?

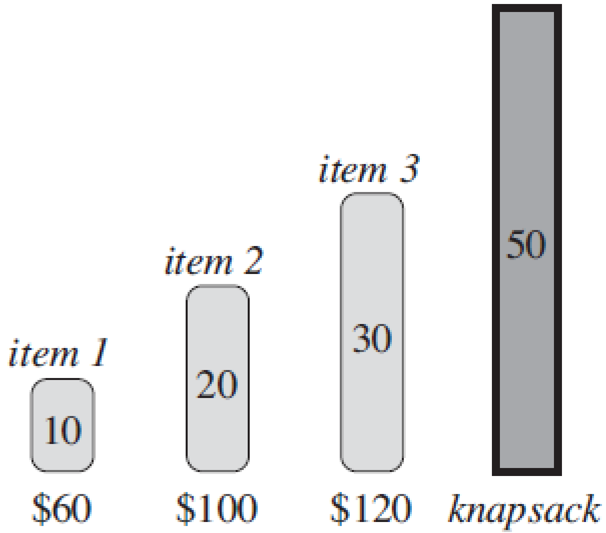
- A thief robbing a store finds n items.
- The i^{th} item is worth v_i dollars and weighs w_i pounds, where i and w_i are integers.
- The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W .
- Which items should he take?

2 TYPES OF KNAPSACK PROBLEMS

- 2 versions of this problem
  - 0-1 knapsack problem
    - for each item, the thief must either take it or leave it behind; he cannot take a fractional amount of an item or take an item more than once.
  - Fractional knapsack problem
    - the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item.

0-1 KNAPSACK PROBLEM

- Assume the following knapsack problem, there are 3 items and a knapsack that can hold 50 pounds.



(a)

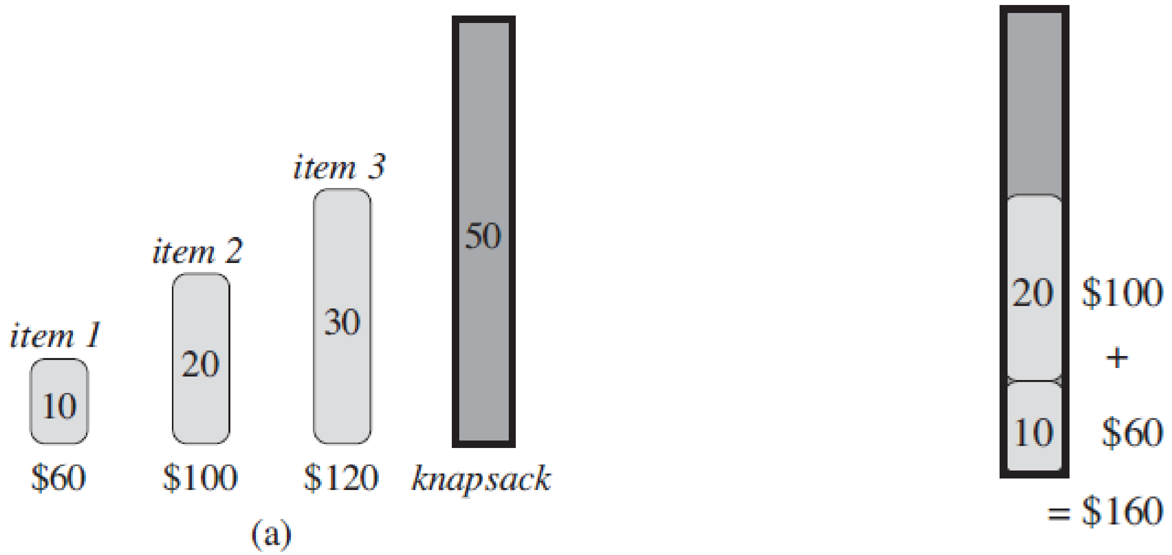
0-1 KNAPSACK PROBLEM

- Here is the value per pound table.
- Greedy choice – first take the item with most value per pound. So, take item 1 first, then item 2 and then item 3.

Item #	Value	Weight	Value/Weight
1	60	10	6(most valuable)
2	100	20	5
3	120	30	4

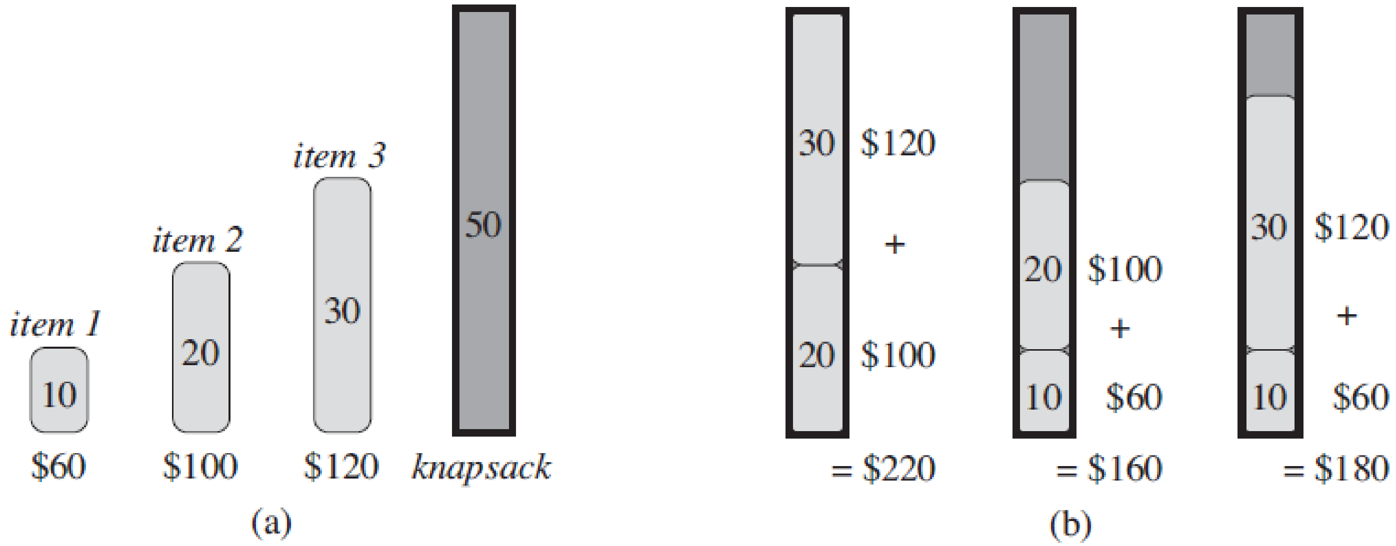
0-1 KNAPSACK PROBLEM

- Greedy choice – first take the item with most value per pound. So, take item 1 first, then item 2 and then item 3.
- What is the total value worth?
  - $160



0-1 KNAPSACK PROBLEM

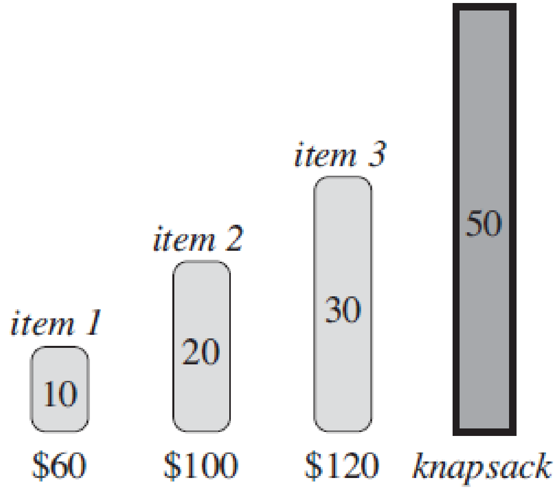
- Lets see what total value we get if we take other items.



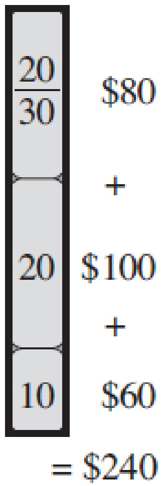
- Conclusion:
  - Including item 1 doesn't give optimal solution. Rather excluding does.
  - **Greedy algorithm doesn't give optimal solution for 0-1 knapsack problem.**

Fractional knapsack problem

- For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.



(a)



(c)

Fractional Knapsack Algorithm - iterative

- *v* → *the set of values of items*
- *w* → *the set of weights of items*
- *c* → *capacity (weight need to be filled in) of the knapsack.*

```
KS(c, v, w)
    sort the item according to v/w in descending order and store in I
    i =0, frac=1;
    tVal=0;
    n = I.length;
    while (c>0 && i<n)
        if(w[i]<=c) frac = 1;
        else frac = c/w[i]
        c =  c – frac*w[i]
        tVal += fract*v[i]
  return tVal;
```

Running time complexity including sorting = O(nlogn)+O(n) = O(nlogn)

Running time complexity without sorting = O(n)

Fractional Knapsack Algorithm - recursive

- *I → sorted items according to v/w in descending order*
- *c → capacity (weight need to be filled in) of the knapsack.*
- *n → number of items*
- *i → current item*

Running time complexity including sorting = O(nlogn)+O(n) = O(nlogn)

Running time complexity without sorting = O(n)

```
KS(c, I, i, n)
    if (c<=0 or i>n) return 0;
    if (c < I[i].weight)
         frac = c/I[i].weight
        return frac* I[i].value + KS(0, I, i+1, n)
    else
        return I[i].value + KS(c-I[i].weight, I, i+1, n)
```

0-1 Knapsack Algorithm - iterative

- *v* → *the set of values of items*
- *w* → *the set of weights of items*
- *c* → *capacity (weight need to be filled in) of the knapsack.*

```
KS(c, v, w)
    sort the item according to v/w in descending order and store in I
    i =0, frac=1;
    tVal=0;
    n = I.length;
    while (c>0 && i<n)
        if(w[i]<=c)
            c =  c – w[i]
            tVal += v[i]
   return tVal;
```

Running time complexity including sorting = O(nlogn)+O(n) = O(nlogn)

Running time complexity without sorting = O(n)

0-1 Knapsack Algorithm - recursive

- *I → sorted items according to v/w in descending order*
- *c → capacity (weight need to be filled in) of the knapsack.*
- *n → number of items*
- *i → current item*

```
KS(c, I, i, n)
    if (c<=0 or i>n) return 0;
    if (c < I[i].weight)
        return KS(c, I, i+1, n)
    else
        return I[i].value + KS(c-I[i].weight, I, i+1, n)
```

Running time complexity including sorting = O(nlogn)+O(n) = O(nlogn)

Running time complexity without sorting = O(n)

Another Example

- Assume you are a busy person. You have exactly T time to do some interesting things and you want to do maximum such things.
- Objective:
  - Maximize the number of interesting things to complete.
- Constraint:
  - Need to finish the works at T time.

Solution of Example

- You are given an array **A** of integers, where each element indicates the time a thing takes for completion. You want to calculate the maximum number of things that you can do in the limited time that you have.
- This is a simple Greedy-algorithm problem.
  - In each iteration, you have to greedily select the things which will take the minimum amount of time to complete.
  - Steps
    - Sort the array **A** in a non-decreasing order.
    - Select one item at a time
    - Complete the item if you have enough time (item's time is less than your available time.)
    - Add one to **numberOfThingsCompleted**.

So when should we use greedy algorithm?

Elements of Greedy Strategy

- An greedy algorithm makes a sequence of choices, each of the choices that seems best at the moment is chosen
  - NOT always produce an optimal solution
- Problems that has the following 2 properties are good candidates for greedy algorithm.
  - Greedy-choice property
  - Optimal substructure

Greedy-Choice Property

- A globally optimal solution can be arrived at by making a locally optimal (greedy) choice
  - Make whatever choice seems best at the moment and then solve the sub-problem arising after the choice is made
  - The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to sub-problems
- Of course, we must prove that a greedy choice at each step yields a globally optimal solution

Optimal Substructures

- A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to sub-problems
  - Example- For Activity Selection Problem
    - If an optimal solution A to S begins with activity 1, then A' = A – {1} is optimal to S'={i ∈S: $s_i \geq f_1$}

Application

- Activity selection problem
- Interval partitioning problem
- Job sequencing problem
- Fractional knapsack problem
- Prim's minimum spanning tree

Try at home

- Why selecting the job with earliest starting time or shortest duration does not work?

Reference

- Chapter 16 (16.1 and 16.2) (Cormen)