Greedy Algorithms

Lecture 11

Optimization Problems

- **Optimization Problem**
  - Problem with an objective function to either:
    - Maximize some profit
    - Minimize some cost

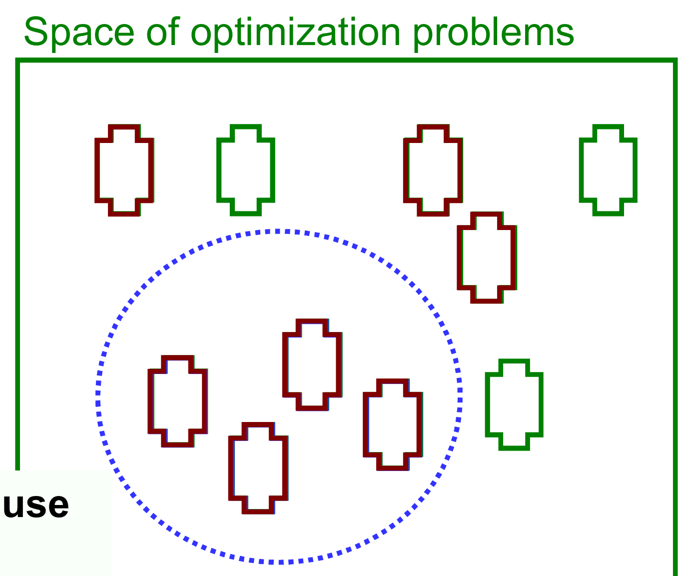
- **Optimization problems appear in so many applications**
  - *Maximize* the number of jobs using a resource **[Activity-Selection Problem**]
  - Encode the data in a file to *minimize* its size [**Huffman Encoding Problem**]
  - Collect the *maximum* value of goods that fit in a given bucket [**knapsack Problem**]
  - Select the *smallest-weight* of edges to connect all nodes in a graph **[Minimum Spanning Tree**]

Solving Optimization Problems

- **Two techniques for solving optimization problems:**
  - Greedy Algorithms ("Greedy Strategy")
  - Dynamic Programming

Greedy algorithms can solve some problems optimally

Dynamic programming can solve more problems optimally (superset)



We still care about Greedy Algorithms because for some problems:

- **Dynamic programming is overkill (slow)**
- **Greedy algorithm is simpler and more efficient**

Greedy Algorithms

- **Main Concept**
  - **Divide the problem into multiple steps (sub-problems)**
  - **For each step take the best choice at the current moment (Local optimal) (Greedy choice)**
  - **A *greedy algorithm* always makes the choice that looks best at the moment**
  - **<u>The hope:</u> A locally optimal choice will lead to a globally optimal solution**
    - For some problems, it works. For others, it does not

Greedy Algorithms

- A *greedy algorithm* always makes the choice that looks best at the moment
  - The hope: a locally optimal choice will lead to a globally optimal solution
  - For some problems, it works

- Dynamic programming can be overkill (slow); greedy algorithms tend to be easier to code
  - Activity-Selection Problem
  - Huffman Codes

The Greedy Method Technique

- **The greedy method** is a general algorithm design paradigm, built on the following elements:
  - **configurations**: different choices, collections, or values to find
  - **objective function**: a score assigned to configurations, which we want to either maximize or minimize
- It works best when applied to problems with the **greedy-choice** property:
  - a globally-optimal solution can always be found by a series of local improvements from a starting configuration.

Elements Of Greedy Algorithms

- **Greedy-Choice Property**
  - At each step, we do a greedy (local optimal) choice

- **Top-Down Solution**
  - The greedy choice is usually done independent of the sub-problems
  - Usually done “before” solving the sub-problem

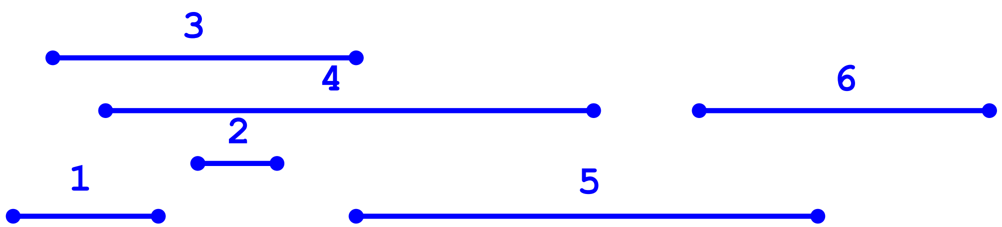
- **Optimal Substructure**
  - The global optimal solution can be composed from the local optimal of the sub-problems

Activity-Selection Problem

- Problem: get your money's worth out of a carnival
  - Buy a wristband that lets you onto any ride
  - Lots of rides, each starting and ending at different times
  - Your goal: ride as many rides as possible
    - Another, alternative goal that we don't solve here: maximize time spent on rides
- Welcome to the *activity selection problem*

Activity-Selection

- Formally:
  - Given a set S of n activities $S = \{a_1, \ldots, a_n\}$
    s_i = start time of activity i
    f_i = finish time of activity i
  - Find max-size subset A of compatible (non-overlapping) activities

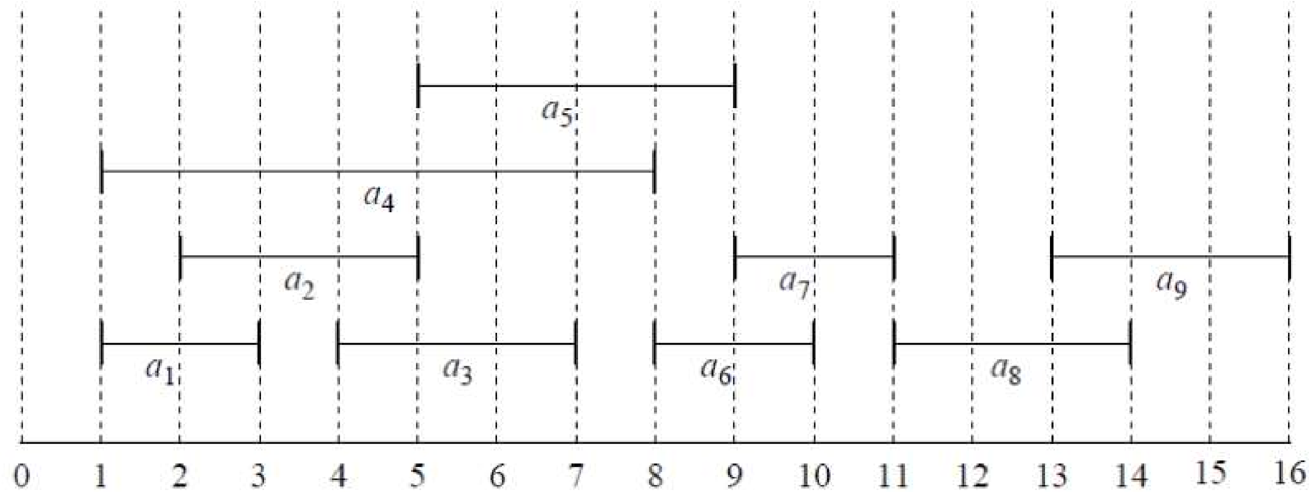


- Assume that $f_1 \leq f_2 \leq \ldots \leq f_n$

Example

S sorted by finish time:

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

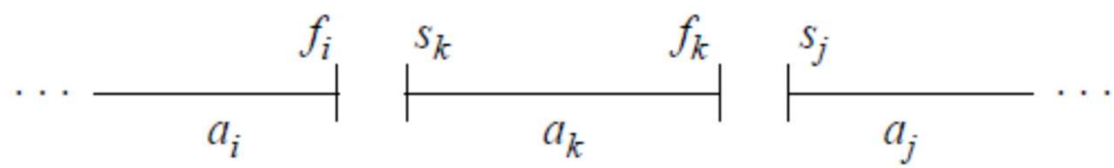


- Maximum-size mutually compatible set: $\{a_1, a_3, a_6, a_8\}$.
- Not unique: also $\{a_2, a_5, a_7, a_9\}$.

Activity Selection: Optimal Substructure

- $S_{ij} = \{a_k \in S : fi \ \leq sk < fk \ \leq sj\}$

  = activities that start after a_i finishes and finish before a_j starts



- In words, activities in S_{ij} are compatible with:
  - All activities that finish by f_i
  - All activities that start no earlier than s_j

Activity Selection: Optimal Substructure

- Let A_{ij} be a maximum-size set of compatible activities in S_{ij}
- Let $a_k \in A_{ij}$ be some activity in A_{ij}. Then we have two sub-problems:
  - Find compatible activities in S_{ik} (activities that start after a_i finishes and that finish before a_k starts)
  - Find compatible activities in S_{kj} (activities that start after a_k finishes and that finish before a_j starts)
- $A_{ij} = A_{ik} \cup \{a_k\} \cup Akj$
- $\rightarrow |A_{ij}| = |A_{ik}| + |A_{kj}| + 1$

Activity Selection: Dynamic Programming

- Let $c[i, j]$ be the size of optimal solution for S_{ij}. Then,

  c[i,j] = c[i,k] + c[k,j] + 1

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \,, \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset \,. \end{cases}$$

Greedy Choice Property

- Dynamic programming
  - Solve all the sub-problems
- Activity selection problem also exhibits the greedy choice property:
  - We should choose an activity that leaves the resource available for as many other activities as possible
  - The first greedy choice is $a_{1,}$ since $f_1 \leq f_2 \leq \ldots \leq f_n$

Activity Selection: A Greedy Algorithm

- So actual algorithm is simple:
  - Sort the activities by finish time
  - Schedule the first activity
  - Then schedule the next activity in sorted list which starts after previous activity finishes
  - Repeat until no more activities
- Intuition is even more simple:
  - Always pick the shortest ride available at the time

Activity Selection: A Greedy Algorithm

- **Select the activity that ends first (smallest end time)**
  - **Intuition: it leaves the largest possible empty space for more activities**

- **Once selected an activity**
  - **Delete all non-compatible activities**
  - **They cannot be selected**

- **Repeat the algorithm for the remaining activities**
  - **Either using iterations or recursion**

Greedy Choice: Select the next best activity (Local Optimal)

Sub-problem: We created one sub-problem to solve (Find the optimal schedule after the selected activity)

Hopefully when we merge the local optimal + the sub-problem optimal solution ➔ we get a global optimal

Greedy Algorithm Correctness

- Theorem:
  - If S_k (activities that start after a_k finishes) is nonempty and a_m has the earliest finish time in S_k, then a_m is included in some optimal solution.

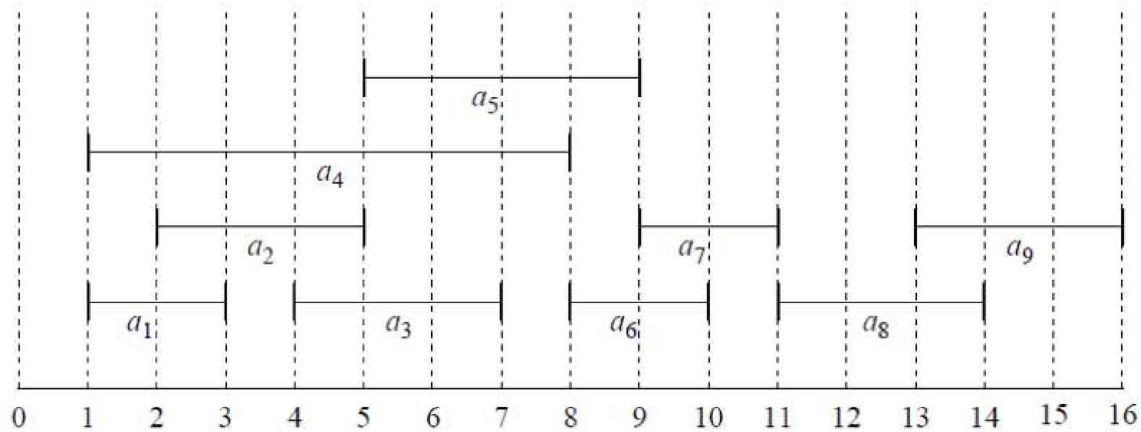
- How to prove it?
  - **We can convert any other optimal solution (S') to the greedy algorithm solution (S)**

- Idea:
  - Compare the activities in *S'* and *S* from left-to-right
  - If they match in the selected activity ➔ skip
  - If they do not match, we can replace the activity in *S'* by that in *S* because the one in *S* finishes first

Example

S sorted by finish time:

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



- S: $\{a_1, a_3, a_6, a_8\}$.
- S':$\{a_2, a_5, a_7, a_9\}$.
- a_2, a_5, a_7, a_9 in S' can be replaced by a_1, a_3, a_6, a_8 from S (finishes earlier)

We mapped S' to S and showed that S is even better

Recursive Solution

Two arrays containing the start and end times
(Assumption: they are sorted based on end times)

The activity chosen in
the last call

The problem size

```
Recursive-Activity-Selection(s, f, k, n)
    m = k +1
    While (m <= n) && ( s[m] < f[k])
            m++;
    If (m <= n)
        return  {A_m}  U Recursive-Activity-Selection(s, f, m, n)
    Else
        return Φ
```

Find the next activity starting after
the end of k

Time Complexity: ***O(n)***
(Assuming arrays are already sorted, otherwise we add *O(n Log n)*

Iterative Solution

Two arrays containing the start and end times
(Assumption: they are sorted based on end times)

```
Iterative-Activity-Selection(s, f)
    n = s.length
    A = {a1}
    k = 1

    for (m = 2 to n)
           if (S[m] >= f[k])
                  A = A U {am}
                  k = m

    Return A
```

Elements Of Greedy Algorithms

- **Proving a greedy solution is optimal**
  - Remember: Not all problems have optimal greedy solution
  - If it does, you need to prove it

  - Usually the proof includes mapping or converting any other optimal solution to the greedy solution