

Greedy Algorithms

CS 4231, Fall 2020

Mihalis Yannakakis

Optimization Problems

- For given instance of the problem, there is a set of “feasible” **solutions**, involving a number of choices (or decisions)
- Every solution has a **cost** or a **value**: metric for evaluating solutions
- Want to find an **optimal** solution: minimum cost or maximum value

Greedy Algorithms

- Form a solution by making a series of choices.
At each point make choice that is locally optimal (looks best) at that point according to some criterion – **greedy choice**

Ingredients of a greedy algorithm:

1. Formulation of solution as a **sequence of choices**
 2. **Criterion** (metric) used to compare choices at each point
- May lead to a globally (overall) optimal solution or to a suboptimal solution

Proving Optimality of Greedy Algorithms

Need to prove two things:

1. There is an optimal solution that includes the first greedy choice
 - Usually involves an **exchange argument**
2. Given the first greedy choice, the value of the overall solution depends monotonically on the value of the solution for the remaining problem \Rightarrow an optimal solution for the remainder of the problem leads to an overall optimal solution:
Optimal substructure property
 - Optimality of the solution follows from these properties by induction on the size of the solution

Activity Selection Problem

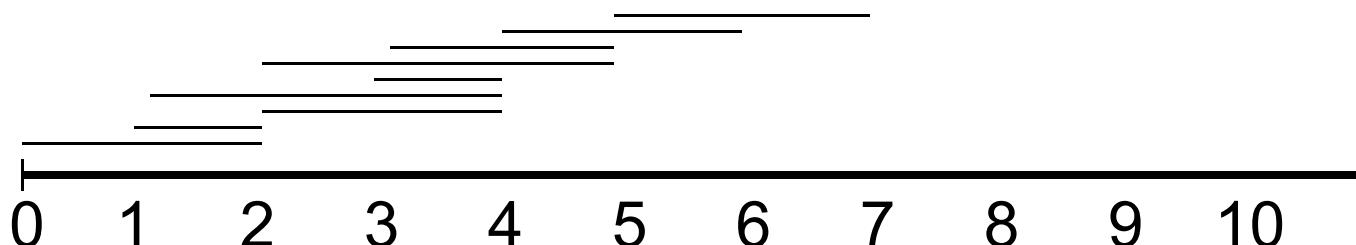
- Given set of desired activities that share a resource, eg. meetings in a classroom
- Each activity has a start time s_i and finish time f_i
 - Interval (s_i, f_i) for each activity
- Find the maximum number of activities that can be scheduled without conflict (intervals don't overlap)

- Example:

A_i: 1 2 3 4 5 6 7 8 9

S_i: 0 1 2 1 3 2 3 4 5

f_i: 2 2 4 4 4 5 5 5 7



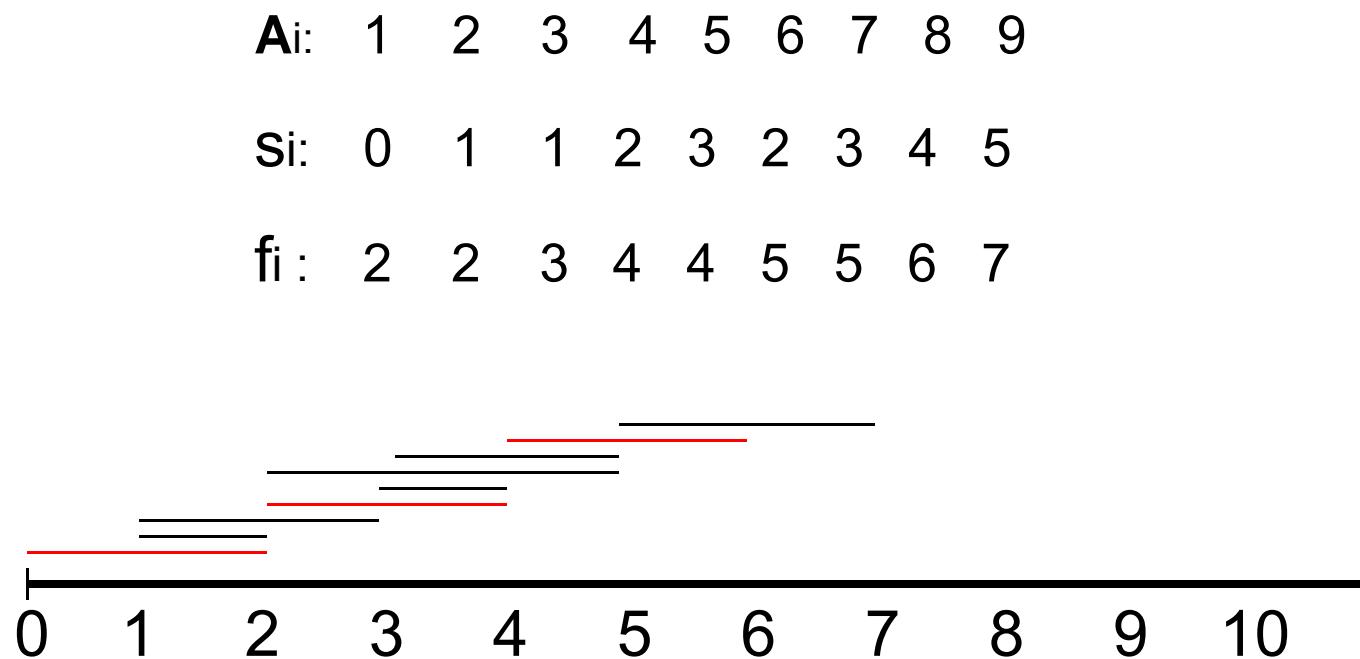
Some greedy criteria that don't work

- **Shortest activity:** choose a shortest activity, remove activities that conflict with it, and iterate
- Counterexample: (1,5), (4,7), (6,12)

- **Earliest start:** choose an activity that starts as early as possible, remove conflicting activities, and iterate
- Counterexample: (1,5), (2,3), (4,7)

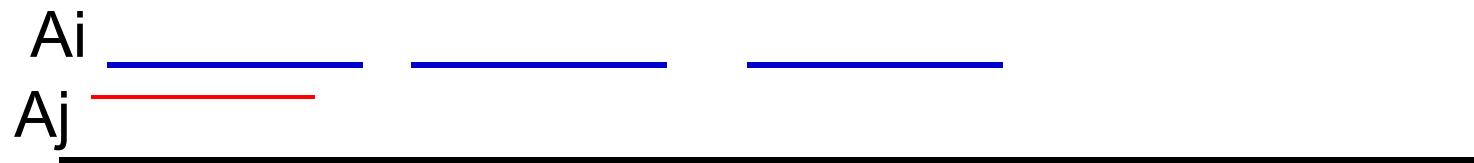
Greedy Algorithm

- First activity: Choose one that *finishes* earliest
- Remove activities that conflict with it
- Iterate with the remaining set



Proof of Optimality

- Property 1:
- Consider an optimal solution $S = \text{set of activities (intervals) that don't overlap}$
- If first interval A_i of S (the one that finishes first) finishes later than first interval A_j of Greedy, then exchanging them (i.e. replacing A_i in S with A_j) yields another feasible (and optimal) solution



Therefore, there is an optimal solution whose first interval agrees with the greedy

Optimality of Greedy

Property 2: Once we chose the first activity, the remaining ones must be chosen among the set S' of those that do not conflict with it.

To get the maximum overall nonconflicting set, we need to pick the maximum possible nonconflicting set from S' (otherwise, we can improve the solution)

Alternative Optimality Proof

- Let Sol^* be an optimal solution that agrees with the greedy solution as long as possible; i.e., among the optimal solutions, we pick a solution Sol^* that agrees with greedy in the first k activities and there is no other optimal solution that agrees in the first $k+1$ activities.
- Show that $\text{Sol}^* = \text{Greedy}$ solution by contradiction:
Argue that if $k < \text{length of Greedy}$, then there is another optimal solution that agrees with Greedy in the first $k+1$ activities. Proof: exchange argument for the $(k+1)$ th activity

Greedy Algorithm (iterative version)

- Sort the activities by finishing time
 - Let $\{ A_i = (s_i, f_i) \mid i=1, \dots, n \}$ be the sorted list
- $Sol = \{ A_1 \}$
 - $k=1$ /* k = index of last activity picked */
- For $i = 2$ to n do
 - if $s_i \geq f_k$ then $\{ Sol = Sol \cup \{A_i\}; k = i \}$
- Return Sol

Time complexity: $O(n \log n)$

If activities already sorted by finishing time then $O(n)$

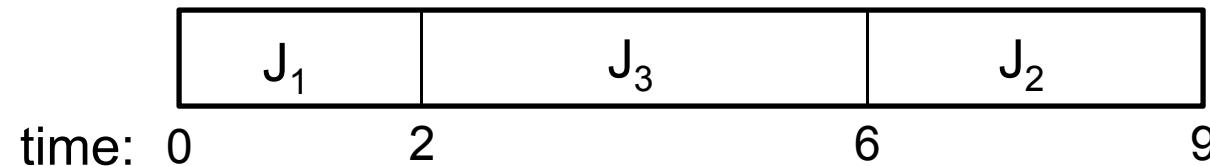
A Scheduling Problem

- **n Jobs:** Job i has processing time p_i , deadline d_i .
- **1 Machine** executes one job at a time according to a schedule, starting from time 0. Each job i starts at some time t_i and finishes at time $f_i = t_i + p_i$.
- **Lateness of a job in a schedule:** $l_i = \max(f_i - d_i, 0)$.
- **Minimum Lateness Problem:** Find a schedule that minimizes the maximum lateness of a job: $\max_i\{l_i\}$

Example

Job	p_i	d_i
1	2	4
2	3	8
3	4	5

a schedule



Lateness of jobs: $l_1 = 0$, $l_2 = 1$, $l_3 = 1$

Lateness of schedule = 1

Some greedy criteria that don't work

- Shortest time first.
- Counterexample: $J_1: p_1=1, d_1=20; J_2: p_2=5, d_2=5$
- Minimum “slack” (d_i-p_i) first.
- Counterexample: $J_1: p_1=1, d_1=2; J_2: p_2=5, d_2=5$

Greedy: Earliest Deadline First

- Schedule the jobs in order of deadline.
- **Theorem:** The Earliest Deadline First algorithm is optimal (minimizes the maximum lateness).
- **Proof:**

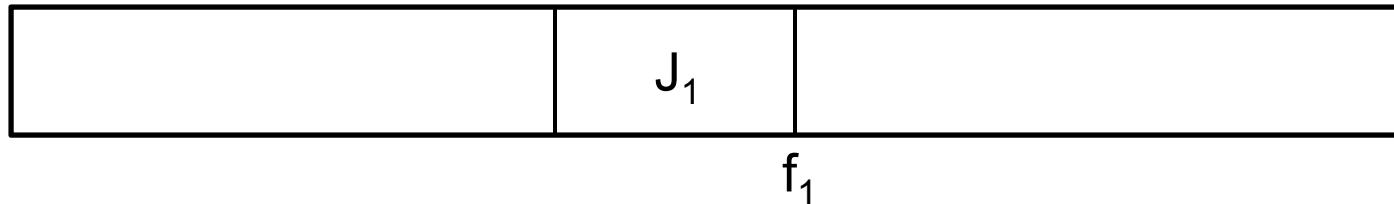
Property 1:

Use exchange argument to show that there is an optimal schedule that starts with the same job as the EDF schedule:

Given any optimal schedule S , we can obtain another schedule S' that starts with the same job as the EDF schedule and such that $\text{lateness}(S') \leq \text{lateness}(S)$.

Exchange argument

Optimal schedule S:



Schedule S': Move the earliest deadline job J_1 to the front



All jobs have deadline $\geq d_1$.

Jobs that were before J_1 in S and are now after J_1 finish by time $f_1 \Rightarrow$ their lateness in S' is $\leq f_1 - d_1 =$ lateness of J_1 in S.

Jobs after J_1 are not affected.

\Rightarrow lateness(S') \leq lateness(S)

Optimal Substructure (Property 2)

- Consider schedules that start with job J_1 with earliest deadline
 - Schedule $S = J_1, \text{schedule } R \text{ of remaining jobs}$
 - $\text{Lateness}(S) = \max\{\text{lateness}(J_1), \text{lateness}(R)\}$
-
- Any schedule R of the remaining jobs that minimizes the $\text{lateness}(R)$ yields a schedule S that minimizes the lateness of the overall schedule S

Huffman Coding

Given a set C of characters, and a frequency $f[c]$ for each character. We want to encode each character c by a binary string $\text{code}(c)$ (not necessarily of same length) so that

- no codeword is a prefix of another codeword
- the encoding minimizes $\sum_c f[c] |\text{code}(c)|$

Optimal variable length prefix code

(can be less costly than fixed length codes)

- Want prefix-free property so that there is no ambiguity when we decode a string

Example: if $\text{code}(a)=0$ and $\text{code}(b)=00$, then 00 could be aa or b

Variable Length Code example

- Variable length code may be less costly than fixed length

Character	a	b	c	d
Frequency	10	2	2	2

Fixed length code: Each character 2 bits → cost 32

Variable length code:	a	b	c	d
	0	10	110	111

Cost: $10 + 4 + 6 + 6 = 26$

Binary tree representation of a code

Frequencies

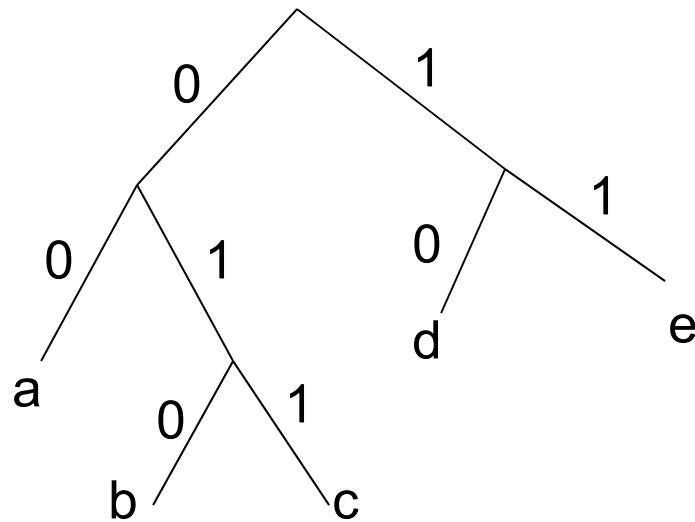
a: 14

b: 6

c: 7

d: 8

e: 12



Code

a: 00
b: 010
c: 011
d: 10
e: 11

Cost of the code: $2f[a]+3f[b]+3f[c]+2f[d]+2f[e]$

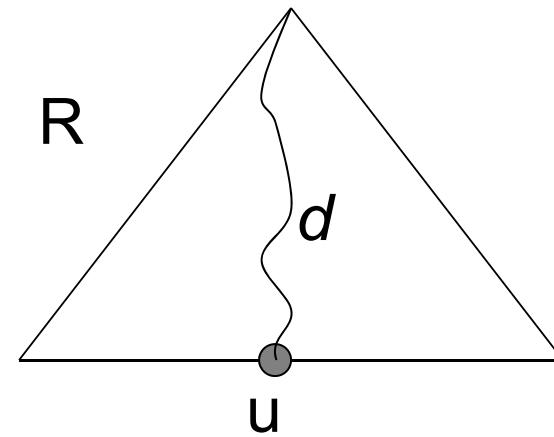
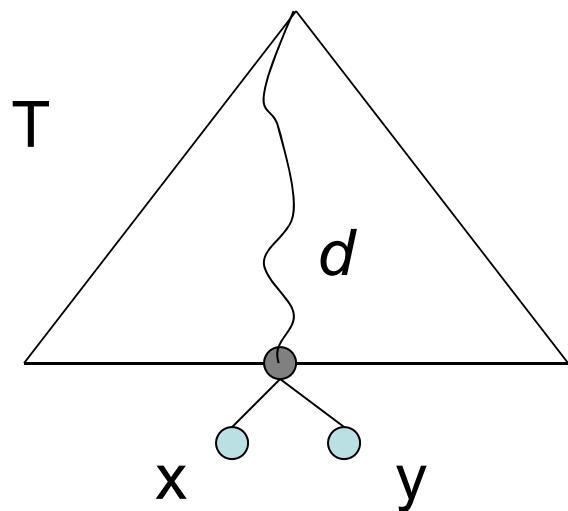
$$=28 + 18 + 21 + 16 + 24 = 107$$

If some internal node has 1 child, then suboptimal

For example if no e, then can use for d code 1 instead of 10

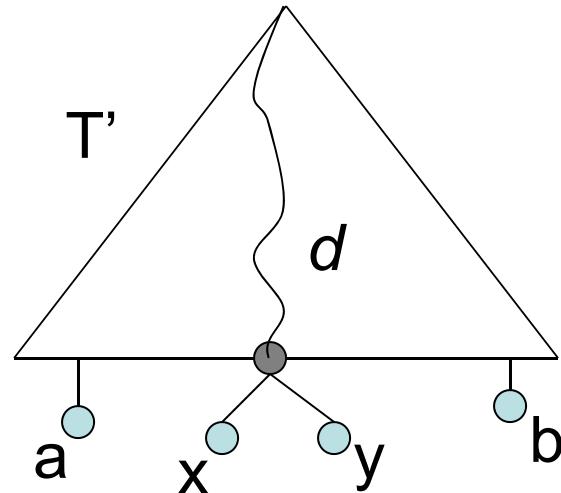
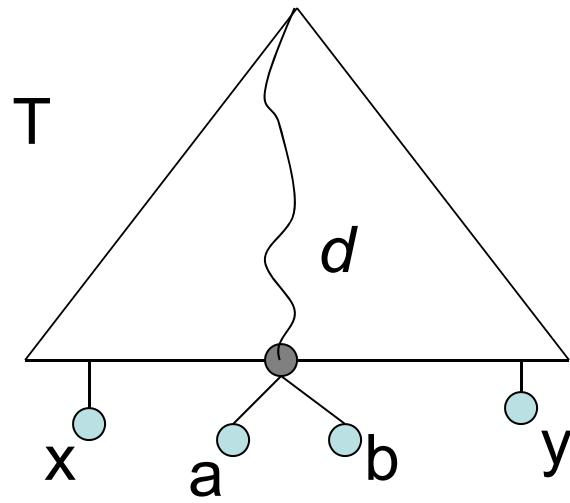
Building binary tree bottom-up

- Sequence of merges of sibling leaves
1. Determine two characters that are sibling leaves in an optimal tree
 2. Merge them and reduce to a problem of size $n-1$



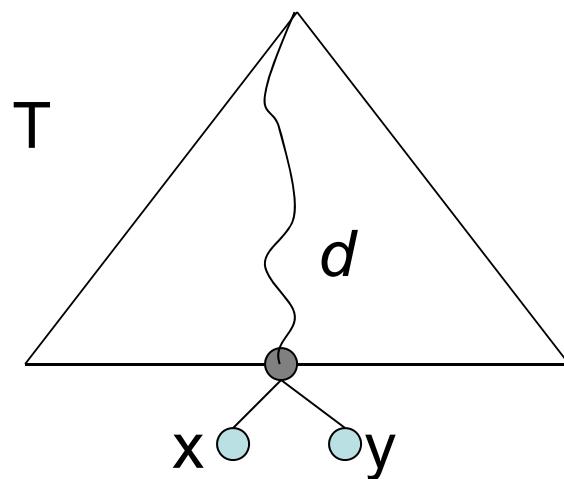
Greedy Choice

- Choose two characters x, y with the lowest frequency
 - Intuition: They should both be deep in the tree
- Proof that there is an optimal tree in which x, y are sibling leaves (property 1):
 - Take optimal tree, let a, b be sibling leaves of maximum depth $f[x] \leq f[a], f[y] \leq f[b]$. Exchange x with a and y with b
New tree T' has no larger cost: $\text{cost}(T') \leq \text{cost}(T)$

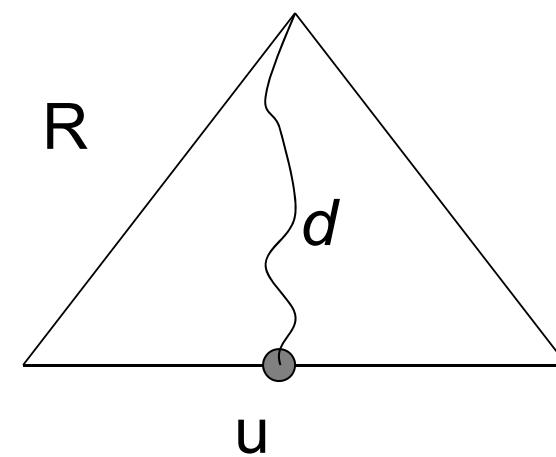


Reduced Subproblem

- Once we determined two sibling leaves x, y of optimal tree, merge the two characters into a new character u with frequency $f[u] = \text{sum of the two frequencies, } f[x]+f[y]$



$$\text{cost}(T) = \dots + (d+1)(f[x]+f[y])$$



$$\text{cost}(R) = \dots + d f[u]$$

$$\text{cost}(T) = \text{cost}(R) + f[x]+f[y]$$

To minimize $\text{cost}(T)$ it is necessary and sufficient to minimize $\text{cost}(R)$ for the reduced problem

Huffman's Algorithm

Create a node for each character c in C

Initialize $Q = C$

While $|Q| > 1$

{ Find two characters x, y in Q with the smallest frequencies

Remove them from Q

Add to Q a new “merged character” xy with $f[xy] = f[x] + f[y]$

Create a new node for xy with children the nodes for x, y
and label the two edges 0, 1

}

Return the final tree T

Codeword $\text{code}[c]$ for each character c = string of labels from
root to leaf corresponding to c

Example

Frequencies

a: 14

b: 6

c: 7

d: 8

e: 12

Code

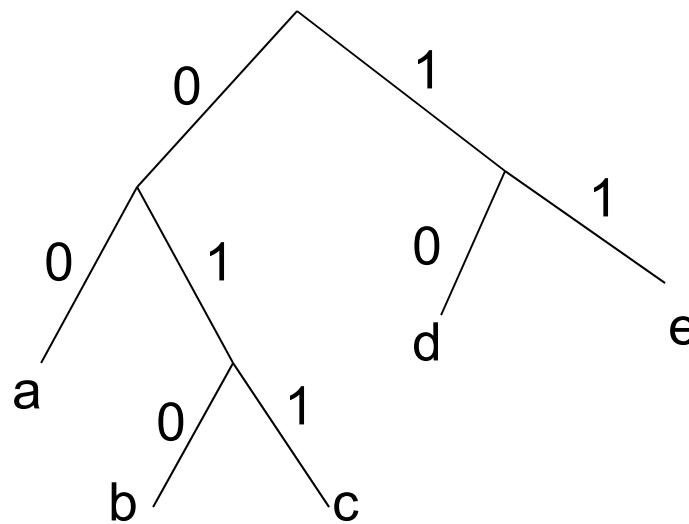
a: 00

b: 010

c: 011

d: 10

e: 11



1. Merge b,c \Rightarrow new character (bc) frequency 13
2. Merge d,e \Rightarrow new character (de) frequency 20
3. Merge a,(bc) \Rightarrow new character (abc) frequency 27
4. Merge (abc),(de) \Rightarrow new character (abcde) frequency 47

Done

Complexity

- Use a min-priority queue (eg. a heap) to maintain the frequencies of the characters in the current subproblem.
- In each iteration, extract the two smallest elements, and insert their sum
- Total time $O(n \log n)$