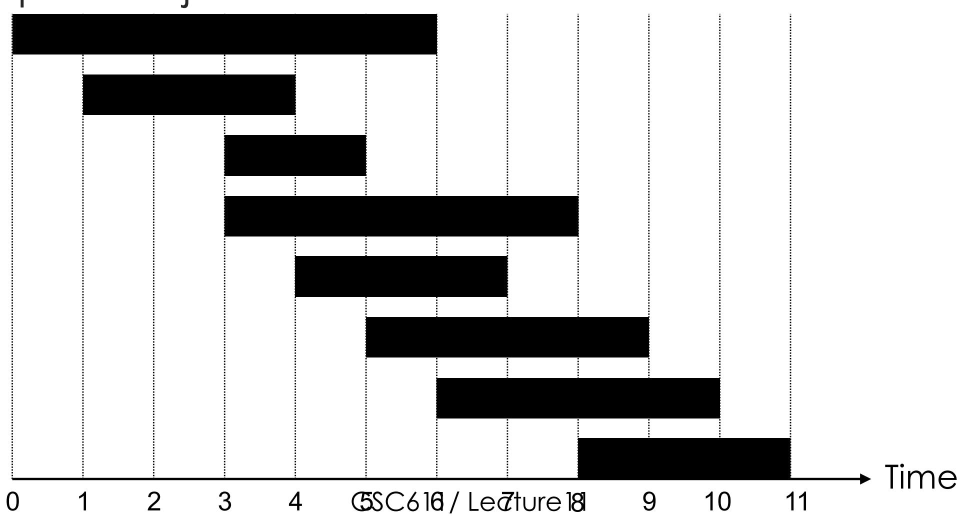
CSC 611: Analysis of Algorithms

Lecture 8

Greedy Algorithms

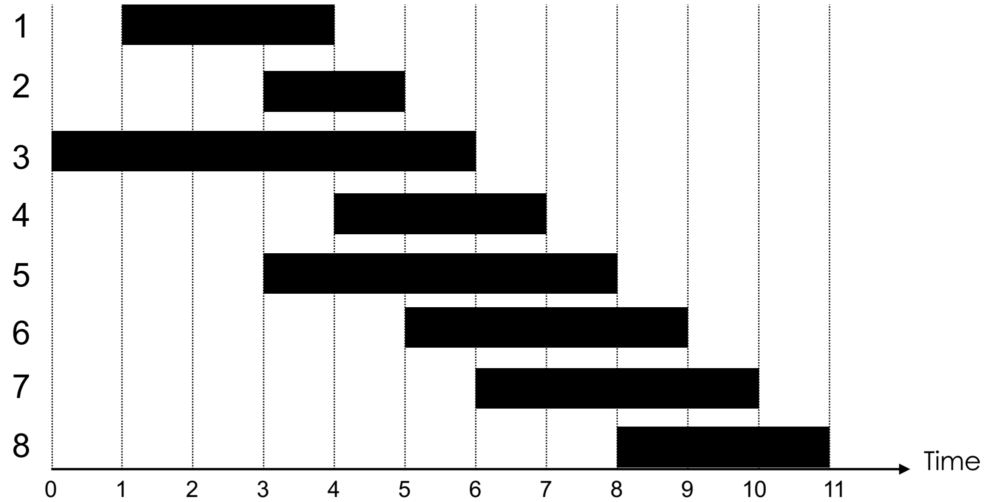
Weighted Interval Scheduling

- Job j starts at s_j, finishes at f_j, and has weight or value v_j
- Two jobs are **compatible** if they don't overlap
- Goal: find maximum **weight** subset of mutually compatible jobs



Weighted Interval Scheduling

- Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$
- Def. p(j) = largest index i < j such that job i is compatible with j
- Ex: p(8) = 5, p(7) = 3, p(2) = 0



1. Making the Choice

- OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j
  - Case 1: OPT selects job j
    - Can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
    - Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

optimal substructure

  - Case 2: OPT does not select job j
    - Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

2. A Recursive Solution

- OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j
  - Case 1: OPT selects job j
    - Can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
    - Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)
    - OPT(j) = v_j + OPT(p(j))
  - Case 2: OPT does not select job j
    - Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1
    - OPT(i) = OPT(j-1)

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Top-Down Recursive Algorithm

WRONG!

Input: n, $s_1,\ldots,s_n$, $f_1,\ldots,f_n$, $v_1,\ldots,v_n$

Sort jobs by finish times so that $f_1 \le f_2 \le \ldots \le f_n$

Compute p(1), p(2), ..., p(n)

```
Compute-Opt(j)
{
  if (j = 0)
    return 0
  else
    return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

3. Compute the Optimal Value

- Compute values in increasing order of j

Input: $n, s_1,\ldots,s_n, f_1,\ldots,f_n, v_1,\ldots,v_n$

Sort jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$

Compute $p(1), p(2), \ldots, p(n)$

```
Iterative-Compute-Opt
{
   M[0] = 0
   for j = 1 to n
      M[j] = max(v_j + M[p(j)], M[j-1])
}
```


Memoized Version

- Store results of each sub-problem; lookup as needed

Input: $n, s_1,\ldots,s_n, f_1,\ldots,f_n, v_1,\ldots,v_n$
Sort jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$.
Compute $p(1), p(2), \ldots, p(n)$

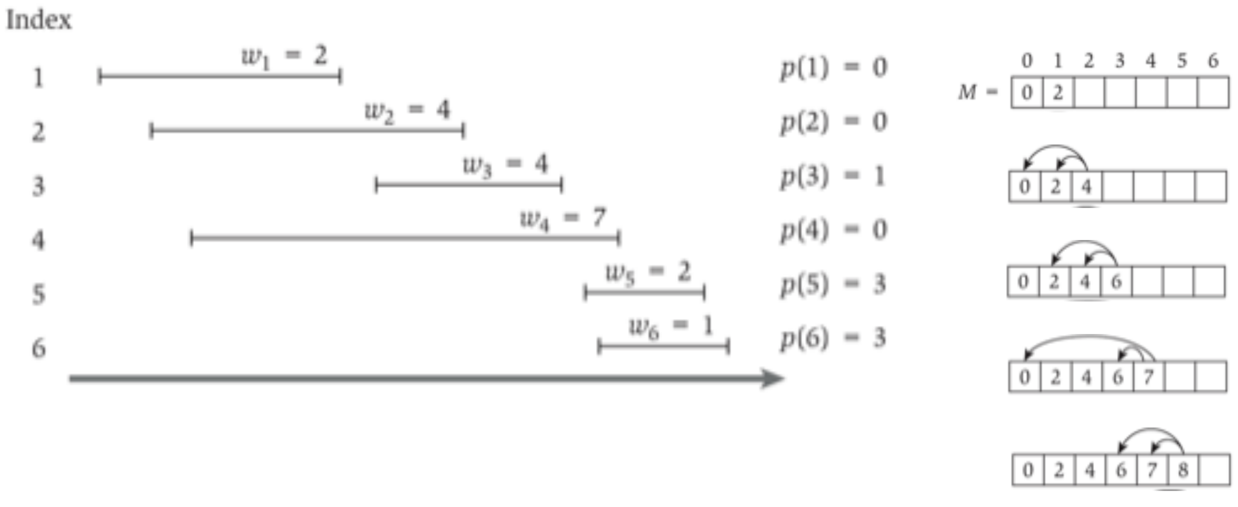
```
for j = 1 to n
   M[j] = empty          ← global array
M[j] = 0

M-Compute-Opt(j)
{
   if (M[j] is empty)
      M[j] = max(v_j + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

4. Finding the Optimal Solution

- Two options
  1. Store additional information: at each time step store either j or p(j) – value that gave the optimal solution
  2. Recursively find the solution by iterating through array M

```
Find-Solution(j)
{
  if (j = 0)
    output nothing
  else if (v_j + M[p(j)] > M[j-1])
    print j
    Find-Solution(p(j))
  else
    Find-Solution(j-1)
}
```


An Example

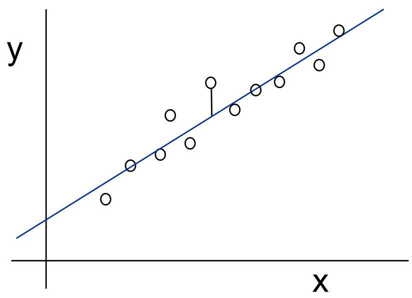


$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{\, v_j + OPT(p(j)),\ OPT(j-1) \,\} & \text{otherwise} \end{cases}$$

Segmented Least Squares

- Least squares
  - Foundational problem in statistic and numerical analysis
  - Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$
  - Find a line y = ax + b that minimizes the sum of the squared error:

$$Error = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$



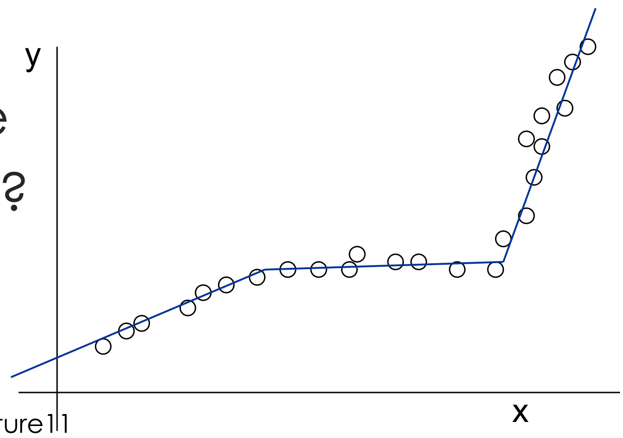
- Solution – closed form
  - Minimum error is achieved when

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Segmented Least Squares

- Segmented least squares
  - Points lie roughly on a sequence of several line segments
  - Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes f(x)
- What is a reasonable choice for f(x) to balance accuracy and parsimony?

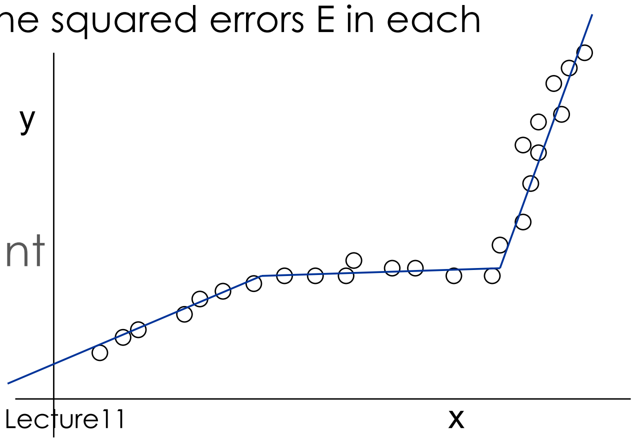
goodness of fit ↑ (accuracy) — number of lines ↑ (parsimony)



Segmented Least Squares

- Segmented least squares
  - Points lie roughly on a sequence of several line segments
  - Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes:
    - the sum of the sums of the squared errors E in each segment
    - the number of lines L
- Tradeoff function:

  E + c L, for some constant c > 0



(1,2) Making the Choice and Recursive Solution

- Notation
  - OPT(j) = minimum cost for points $p_1, p_{i+1}, \ldots, p_j$
  - e(i, j) = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$
- To compute OPT(j)
  - Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some i
  - Cost = e(i, j) + c + OPT(i-1)

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \le i \le j} \{ e(i,j) + c + OPT(i-1) \} & \text{otherwise} \end{cases}$$

3. Compute the Optimal Value

```
INPUT: n, p1,…,pN , c

Segmented-Least-Squares() {
   M[0] = 0
   for j = 1 to n
      for i = 1 to j
         compute the least square error eij for
         the segment pi,…, pj

   for j = 1 to n
      M[j] = min 1 ≤ i ≤ j (eij + c + M[i-1])

   return M[n]
}
```

- Running time: $O(n^3)$
  - Bottleneck = computing e(i, j) for $O(n^2)$ pairs, $O(n)$ per pair using previous formula

Greedy Algorithms

- Similar to dynamic programming, but simpler approach
  - Also used for optimization problems
- **Idea:** When we have a choice to make, make the one that looks best right now
  - Make a locally optimal choice in the hope of getting a globally optimal solution
- Greedy algorithms don't always yield an optimal solution
- When the problem has certain general characteristics, greedy algorithms give optimal solutions

Activity Selection

- Problem
  - Schedule the largest possible set of non-overlapping activities for a given room

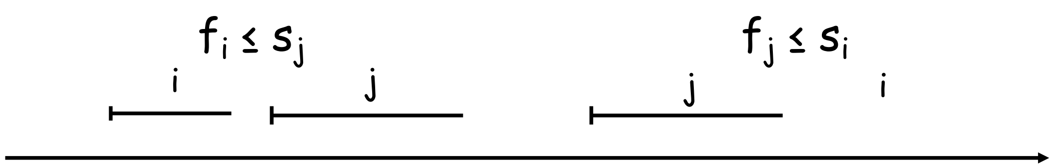
	Start	End	Activity
1	8:00am	9:15am	Numerical methods class
2	8:30am	10:30am	Movie presentation (refreshments served)
3	9:20am	11:00am	Data structures class
4	10:00am	noon	Programming club mtg. (Pizza provided)
5	11:30am	1:00pm	Computer graphics class
6	1:05pm	2:15pm	Analysis of algorithms class
7	2:30pm	3:00pm	Computer security class
8	noon	4:00pm	Computer games contest (refreshments served)
9	4:00pm	5:30pm	Operating systems class

Activity Selection

- Schedule **n activities** that require exclusive use of a common resource

  $S = \{a_1, \ldots, a_n\}$ – set of activities

- a_i needs resource during period $[s_i , f_i)$
  - s_i = **start time** and f_i = **finish time** of activity a_i
  - $0 \leq s_i < f_i < \infty$
- Activities a_i and a_j are **compatible** if the intervals $[s_i , f_i)$ and $[s_j, f_j)$ do not overlap



Activity Selection Problem

Select the largest possible set of non-overlapping (***compatible***) activities.

E.g.:

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	8	9	10	11	12	13	14

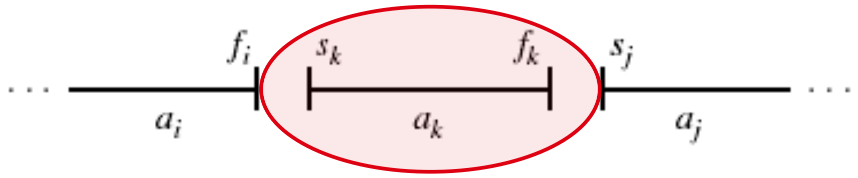
- Activities are sorted in increasing order of finish times
- A subset of mutually compatible activities: $\{a_3, a_9, a_{11}\}$
- Maximal set of mutually compatible activities: $\{a_1, a_4, a_8, a_{11}\}$ and $\{a_2, a_4, a_9, a_{11}\}$

Optimal Substructure

- Define the space of subproblems:

  $S_{ij} = \{ a_k \in S : f_i \le s_k < f_k \le s_j \}$

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



- Add fictitious activities
  – $a_0 = [-\infty, 0)$
  – $a_{n+1} = [\infty, \text{“}\infty + 1\text{”})$
  – Range for S_{ij} is $0 \le i, j \le n + 1$

$S = S_{0,n+1}$ entire space of activities

Representing the Problem

- We assume that activities are sorted in increasing order of finish times:

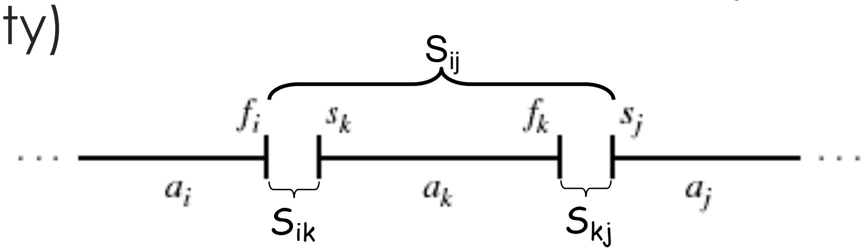
$$f_0 \le f_1 \le f_2 \le \ldots \le f_n < f_{n+1}$$

- What happens to set S_{ij} for $i \ge j$?
  - For an activity $a_k \in S_{ij}$: $f_i \le s_k < f_k \le s_j < f_j$
    contradiction with $f_i \ge f_j$!
  - $\Rightarrow S_{ij} = \emptyset$ (the set S_{ij} must be empty!)
- We only need to consider sets S_{ij} with

$$0 \le i < j \le n + 1$$

Optimal Substructure

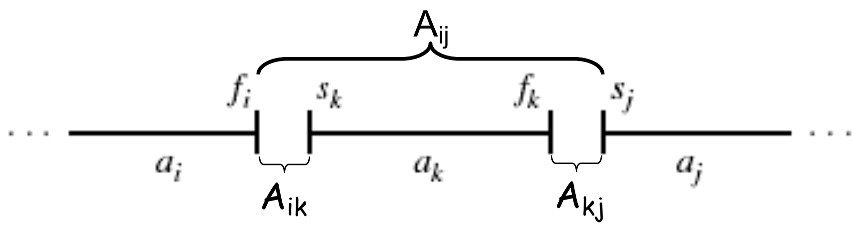
- Subproblem:
  - Select a maximum-size subset of mutually compatible activities from set S_{ij}
- Assume that a solution to the above subproblem includes activity a_k (S_{ij} is non-empty)



Solution to S_{ij} = (Solution to S_{ik}) U $\{a_k\}$ U (Solution to S_{kj})

$|$Solution to $S_{ij}|$ = $|$Solution to $S_{ik}|$ + 1 + $|$Solution to $S_{kj}|$

Optimal Substructure



A_{ij} = Optimal solution to S_{ij}

- **Claim**: Sets A_{ik} and A_{kj} must be optimal solutions
- Assume ∃ A_{ik}' that includes more activities than A_{ik}

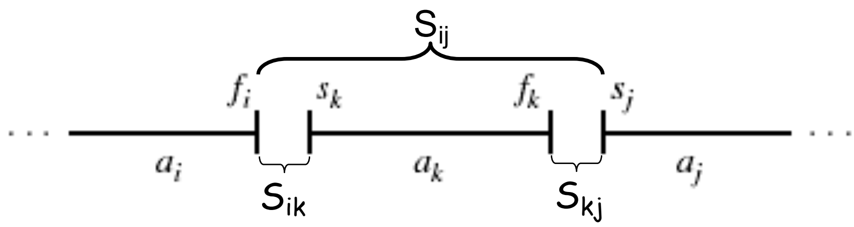
$$\text{Size}[A_{ij}'] = \text{Size}[A_{ik}'] + 1 + \text{Size}[A_{kj}] > \text{Size}[A_{ij}]$$

⇒ Contradiction: we assumed that A_{ij} has the maximum # of activities taken from S_{ij}

Recursive Solution

- Any optimal solution (associated with a set S_{ij}) contains within it optimal solutions to subproblems S_{ik} and S_{kj}
- c[i, j] = size of maximum-size subset of mutually compatible activities in S_{ij}
- If $S_{ij} = \emptyset \Rightarrow$ $c[i, j] = 0$

Recursive Solution



If $S_{ij} \neq \emptyset$ and if we consider that a_k is used in an optimal solution (maximum-size subset of mutually compatible activities of S_{ij}), then:

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

Recursive Solution

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

- There are $j - i - 1$ possible values for k
  - $k = i+1, \ldots, j - 1$
  - a_k cannot be a_i or a_j (from the definition of S_{ij})

    $S_{ij} = \{ a_k \in S : f_i \leq s_k < f_k \leq s_j \}$
  - We check all the values and take the best one

We could now write a dynamic programming algorithm

Theorem

Let $S_{ij} \neq \emptyset$ and a_m the activity in S_{ij} with the earliest finish time:

$$f_m = \min \{ f_k : a_k \in S_{ij} \}$$

Then:

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}
   – There exists some optimal solution that contains a_m
2. $S_{im} = \emptyset$
   – Choosing a_m leaves S_{mj} the only nonempty subproblem

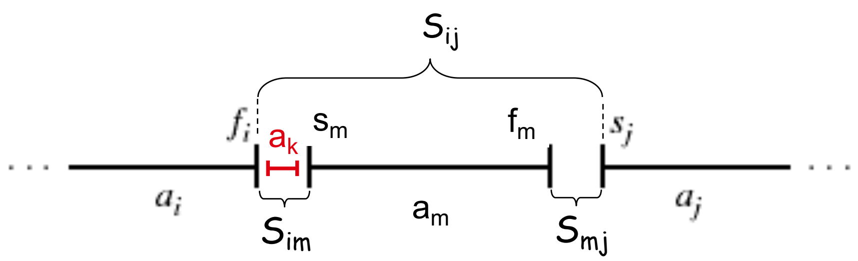
Proof

2. Assume $\exists\, a_k \in S_{im}$

$f_i \leq s_k < f_k \leq s_m < f_m$

$\Rightarrow$ $f_k < f_m$ contradiction !

a_m must have the earliest finish time

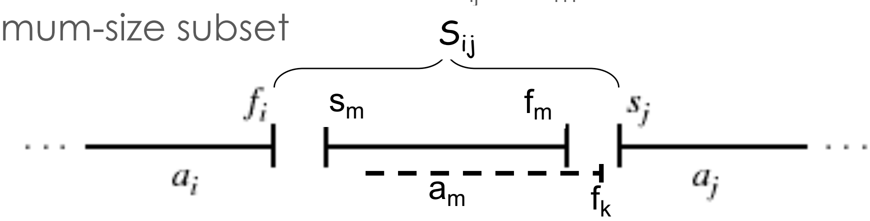


$\Rightarrow$ There is no $a_k \in S_{im} \Rightarrow S_{im} = \emptyset$

Proof: Greedy Choice Property

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}

- A_{ij} = optimal solution for activity selection from S_{ij}
  - Order activities in A_{ij} in increasing order of finish time
  - Let a_k be the first activity in $A_{ij} = \{a_k, \ldots\}$
- If $a_k = a_m$ Done!
- Otherwise, replace a_k with a_m (resulting in a set A_{ij}')
  - since $f_m \le f_k$ the activities in A_{ij}' will continue to be compatible
  - A_{ij}' will have the same size as $A_{ij} \Rightarrow a_m$ is used in some maximum-size subset



Why is the Theorem Useful?

	Dynamic programming	Using the theorem
Number of subproblems in the optimal solution	2 subproblems: S_{ik}, S_{kj}	1 subproblem: S_{mj} ($S_{im} = \emptyset$)
Number of choices to consider	$j - i - 1$ choices	1 choice: the activity a_m with the earliest finish time in S_{ij}

- Making the greedy choice (the activity with the earliest finish time in S_{ij})
  - Reduces the number of subproblems and choices
  - Allows solving each subproblem in a top-down fashion
- Only one subproblem left to solve!

Greedy Approach

- To select a maximum-size subset of mutually compatible activities from set S_{ij}:
  - Choose $a_m \in S_{ij}$ with earliest finish time (greedy choice)
  - Add a_m to the set of activities used in the optimal solution
  - Solve the same problem for the set S_{mj}
- From the theorem
  - By choosing a_m we are guaranteed to have used an activity included in an optimal solution
    - ⇒ We do not need to solve the subproblem S_{mj} before making the choice!
  - The problem has the **GREEDY CHOICE** property

Characterizing the Subproblems

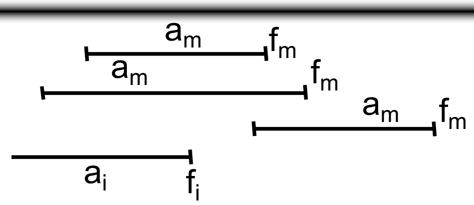
- The original problem: find the maximum subset of mutually compatible activities for $S = S_{0,\ n+1}$
- Activities are sorted by increasing finish time

$$a_0, a_1, a_2, a_3, \ldots, a_{n+1}$$

- We always choose an activity with the earliest finish time
  - Greedy choice maximizes the unscheduled time remaining
  - Finish time of activities selected is strictly increasing

A Recursive Greedy Algorithm

Alg.: REC-ACT-SEL (s, f, i, n)

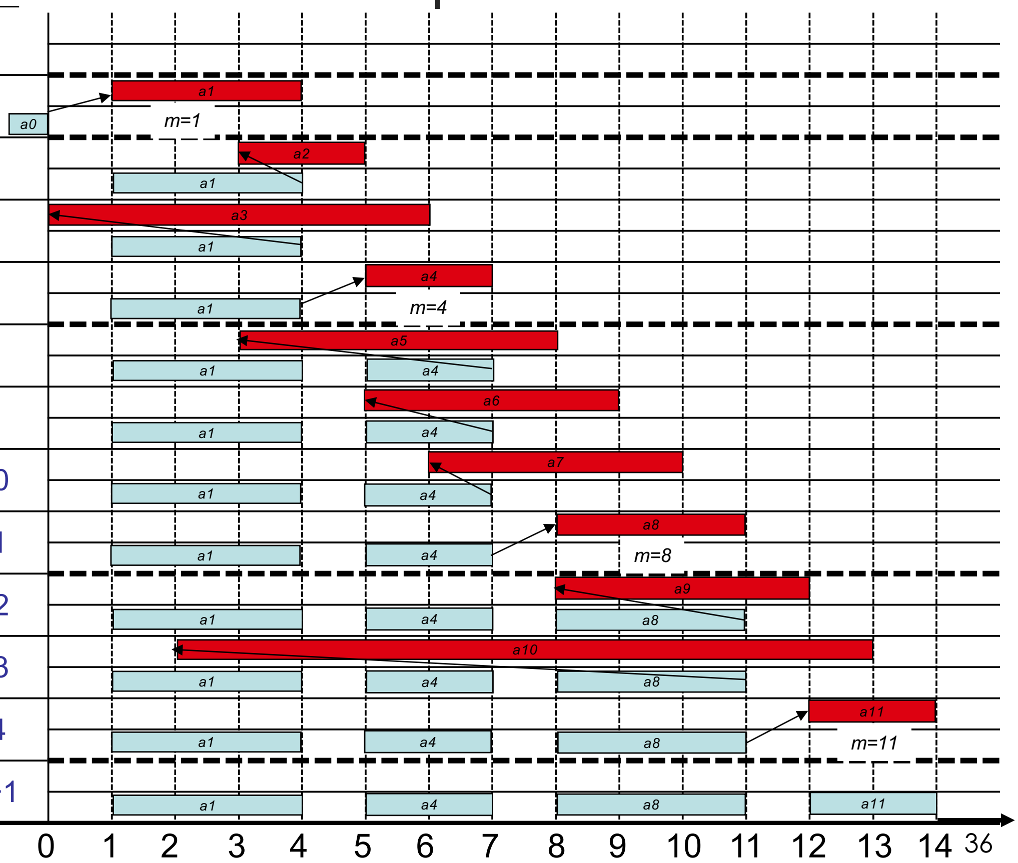


1. $m \leftarrow i + 1$
2. **while** $m \le n$ **and** $s_m < f_i$ ► Find first activity in $S_{i,n+1}$
3. **do** $m \leftarrow m + 1$
4. **if** $m \le n$
5. **then return** $\{a_m\}$ U REC-ACT-SEL(s, f, m, n)
6. **else return** Ø

- Activities are ordered in increasing order of finish time
- Running time: $\Theta(n)$ – each activity is examined only once
- **Initial call:** REC-ACT-SEL(s, f, 0, n)

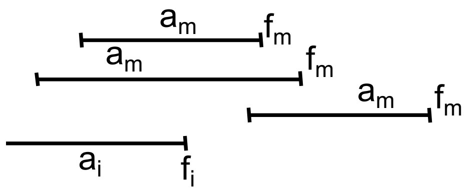
Example

k	s_k	f_k
0	-∞	0
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14
12	∞	∞+1



An Incremental Algorithm

Alg.: GREEDY-ACTIVITY-SELECTOR(s, f, n)



$A \leftarrow \{a_1\}$

$i \leftarrow 1$

for $m \leftarrow 2$ **to** n

    do if $s_m \geq f_i$ ► activity a_m is compatible with a_i

        then $A \leftarrow A \cup \{a_m\}$

            $i \leftarrow m$ ► a_i is most recent addition to A

return A

- Assumes that activities are ordered in increasing order of finish time
- Running time: $\Theta(n)$ – each activity is examined only once

Steps Toward Our Greedy Solution

1. Determined the optimal substructure of the problem
2. Developed a recursive solution
3. Proved that one of the optimal choices is the greedy choice
4. Showed that all but one of the subproblems resulted by making the greedy choice are empty
5. Developed a recursive algorithm that implements the greedy strategy
6. Converted the recursive algorithm to an iterative one

Designing Greedy Algorithms

1. Cast the optimization problem as one for which:
   - we make a (greedy) choice and are left with only one subproblem to solve
2. Prove the GREEDY CHOICE property:
   - that there is always an optimal solution to the original problem that makes the greedy choice
3. Prove the OPTIMAL SUBSTRUCTURE:
   - the greedy choice + an optimal solution to the resulting subproblem leads to an optimal solution

Correctness of Greedy Algorithms

1. Greedy Choice Property
   – A globally optimal solution can be arrived at by making a locally optimal (greedy) choice
2. Optimal Substructure Property
   – We know that we have arrived at a subproblem by making a greedy choice
   – Optimal solution to subproblem + greedy choice $\Rightarrow$ optimal solution for the original problem

Dynamic Programming vs. Greedy Algorithms

- Dynamic programming
  - We make a choice at each step
  - The choice depends on solutions to subproblems
  - Bottom up solution, from smaller to larger subproblems
- Greedy algorithm
  - Make the greedy choice and THEN
  - Solve the subproblem arising after the choice is made
  - The choice we make may depend on previous choices, but not on solutions to subproblems
  - Top down solution, problems decrease in size

The Knapsack Problem

- **The 0-1 knapsack problem**
  - A thief robbing a store finds n items: the i-th item is worth v_i dollars and weights w_i pounds (v_i, w_i integers)
  - The thief can only carry W pounds in his knapsack
  - Items must be taken entirely or left behind
  - Which items should the thief take to maximize the value of his load?
- **The fractional knapsack problem**
  - Similar to above
  - The thief can take fractions of items

Fractional Knapsack Problem

- Knapsack capacity: W
- There are n items: the i-th item has value v_i and weight w_i
- Goal:
  - Find fractions x_i so that for all $0 \le x_i \le 1$, $i = 1, 2, .., n$

    $\sum w_i x_i \le W$ and

    $\sum x_i v_i$ is maximum

Fractional Knapsack Problem

- Greedy strategy 1:
  - Pick the item with the maximum value
- *E.g.:*
  - $W = 1$
  - $w_1 = 100$, $v_1 = 2$
  - $w_2 = 1$, $v_2 = 1$
  - Taking from the item with the maximum value:

    Total value (choose item 1) = $v_1W/w_1 = 2/100$
  - Smaller than what the thief can take if choosing the other item

    Total value (choose item 2) = $v_2W/w_2 = 1$

Fractional Knapsack Problem

- Greedy strategy 2:
  - Pick the item with the maximum value per pound v_i/w_i
  - If the supply of that element is exhausted and the thief can carry more: take as much as possible from the item with the next greatest value per pound
  - It is good to order items based on their value per pound

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$$

Fractional Knapsack Problem

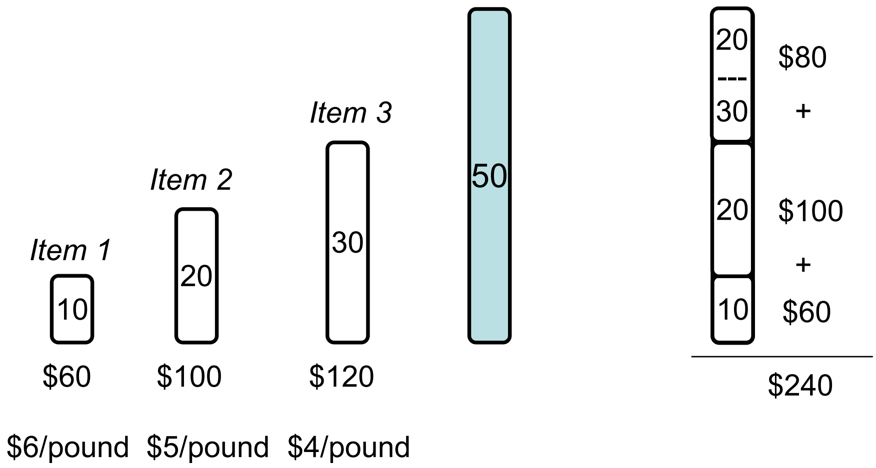
Alg.: Fractional-Knapsack (W, v[n], w[n])

1. w = W
2. While w > 0 and there are items remaining
3. pick item i with maximum v_i/w_i
4. $x_i \leftarrow \min(1, w/w_i)$
5. remove item i from list
6. $w \leftarrow w - x_i w_i$

- w – the amount of space remaining in the knapsack
- Running time: $\Theta(n)$ if items already ordered; else $\Theta(n\lg n)$

Fractional Knapsack - Example

- *E.g.:*



Greedy Choice

Items:	1	2	3	...	j	...	n
Optimal solution:	x_1	x_2	x_3		x_j		x_n
Greedy solution:	x_1'	x_2'	x_3'		x_j'		x_n'

- We know that: $x_1' \geq x_1$
  - greedy choice takes as much as possible from item 1
- Modify the optimal solution to take x_1' of item 1
  - We have to decrease the quantity taken from some item j: the new x_j is decreased by: $(x_1' - x_1)\ w_1\ /w_j$
- Increase in profit: $(x_1' - x_1)\ v_1$
- Decrease in profit: $(x_1' - x_1) w_1\ v_j/w_j$

$$(x_1' - x_1)\ v_1 \geq (x_1' - x_1) w_1\ v_j/w_j$$

$$v_1 \geq w_1 \frac{v_j}{w_j} \Rightarrow \frac{v_1}{w_1} \geq \frac{v_j}{w_j}$$

True, since x_1 had the best value/pound ratio

Huffman Codes

- Widely used technique for data compression
- Assume the data to be a sequence of characters
- Looking for an effective way of storing the data
- ***Binary character code***
  - Uniquely represents a character by a binary string

Fixed-Length Codes

E.g.: Data file containing 100,000 characters

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

- 3 bits needed
- a = 000, b = 001, c = 010, d = 011, e = 100, f = 101
- Requires: 100,000 × 3 = 300,000 bits

Huffman Codes

- Idea:
  - Use the frequencies of occurrence of characters to build a optimal way of representing each character

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

Variable-Length Codes

E.g.: Data file containing 100,000 characters

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

- Assign short codewords to frequent characters and long codewords to infrequent characters

a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100

$(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1{,}000$

$= 224{,}000$ bits

Prefix Codes

- Prefix codes:
  - Codes for which no codeword is also a prefix of some other codeword
  - Better name would be “prefix-free codes”
- We can achieve optimal data compression using prefix codes
  - We will restrict our attention to prefix codes

Encoding with Binary Character Codes

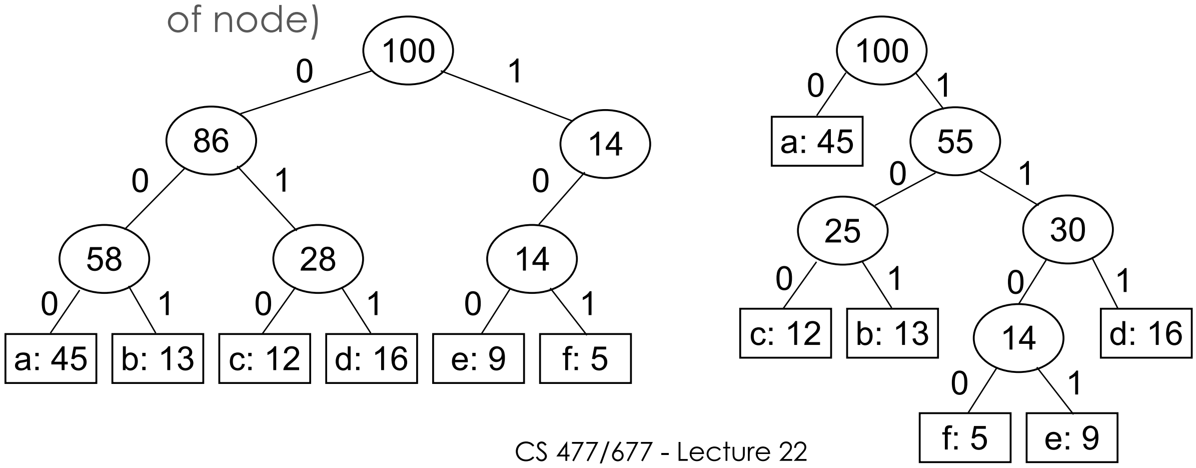
- Encoding
  - Concatenate the codewords representing each character in the file
- *E.g.*:
  - a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100
  - abc = 0 × 101 × 100 = 0101100

Decoding with Binary Character Codes

- Prefix codes simplify decoding
  - No codeword is a prefix of another ⇒ the codeword that begins an encoded file is unambiguous
- Approach
  - Identify the initial codeword
  - Translate it back to the original character
  - Repeat the process on the remainder of the file
- *E.g.*:
  - a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100
  - 001011101 = 0 × 0 × 101 × 1101 = aabe

Prefix Code Representation

- Binary tree whose leaves are the given characters
- Binary codeword
  - the path from the root to the character, where 0 means "go to the left child" and 1 means "go to the right child"
- Length of the codeword
  - Length of the path from root to the character leaf (depth of node)



Optimal Codes

- An optimal code is always represented by a **full binary tree**
  - Every non-leaf has two children
  - Fixed-length code is not optimal, variable-length is
- How many bits are required to encode a file?
  - Let C be the alphabet of characters
  - Let $f(c)$ be the frequency of character c
  - Let $d_T(c)$ be the depth of c's leaf in the tree T corresponding to a prefix code

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

the cost of tree T

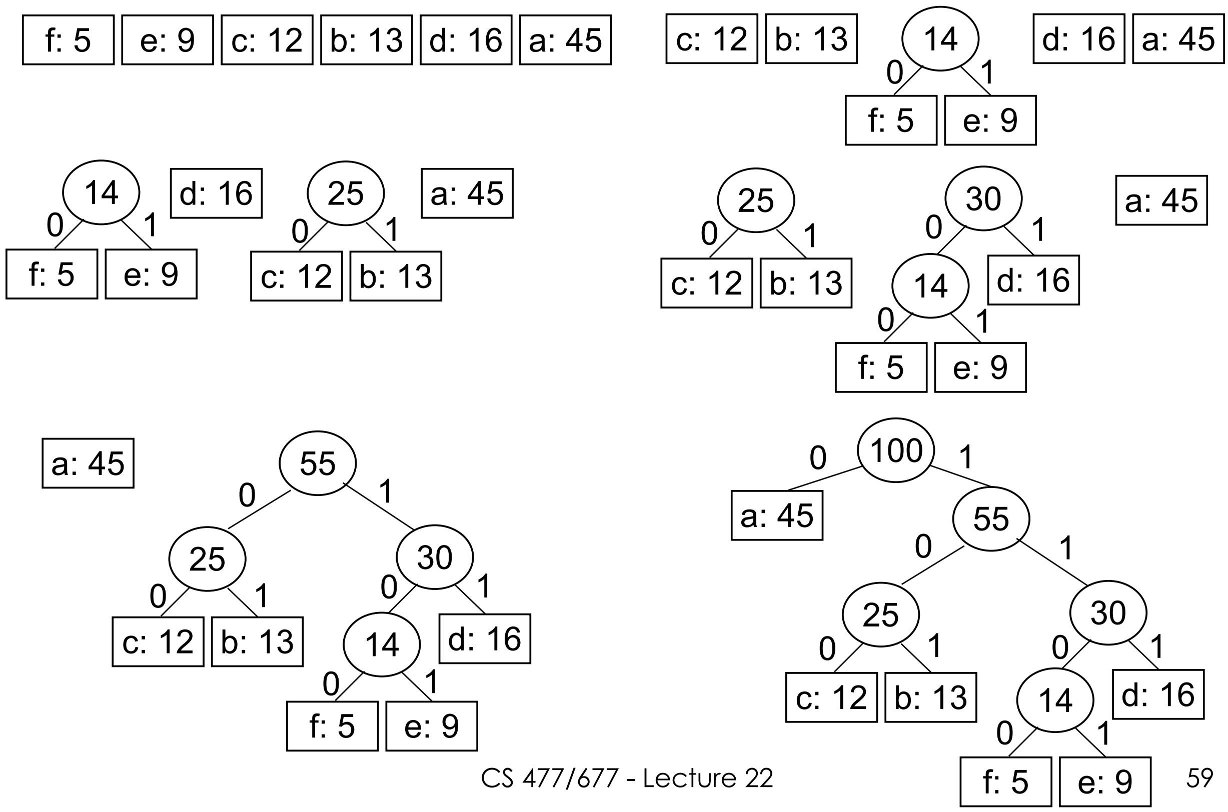
Constructing a Huffman Code

- Let's build a greedy algorithm that constructs an optimal prefix code (called a **Huffman code**)
- Assume that:
  - C is a set of n characters
  - Each character has a frequency $f(c)$
- Idea:

f: 5	e: 9	c: 12	b: 13	d: 16	a: 45

  - The tree T is built in a bottom up manner
  - Start with a set of $|C| = n$ leaves
  - At each step, merge the two least frequent objects: the frequency of the new node = sum of two frequencies
  - Use a min-priority queue Q, keyed on f to identify the two least frequent objects

Example



Building a Huffman Code

Alg.: HUFFMAN(*C*) Running time: **O(nlgn)**

1. n ← |C|
2. Q ← C ← O(n)
3. **for** i ← 1 **to** n - 1
4. **do** allocate a new node **z**
5. left[z] ← x ← EXTRACT-MIN(Q)
6. right[z] ← y ← EXTRACT-MIN(Q)
7. f[z] ← f[x] + f[y]
8. INSERT (Q, z)
9. **return** EXTRACT-MIN(Q)

O(nlgn)

Greedy Choice Property

Let C be an alphabet in which each character $c \in C$ has frequency $f[c]$. Let x and y be two characters in C having the lowest frequencies.

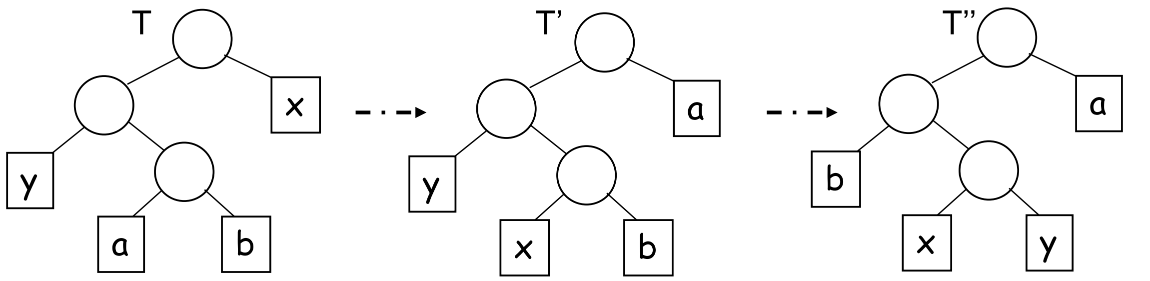
Then, there exists an optimal prefix code for C in which the codewords for x and y have the same (maximum) length and differ only in the last bit.

Proof of the Greedy Choice

- Idea:
  - Consider a tree T representing an arbitrary optimal prefix code
  - Modify T to make a tree representing another optimal prefix code in which x and y will appear as sibling leaves of maximum depth

  ⇒ The codes of x and y will have the same length and differ only in the last bit

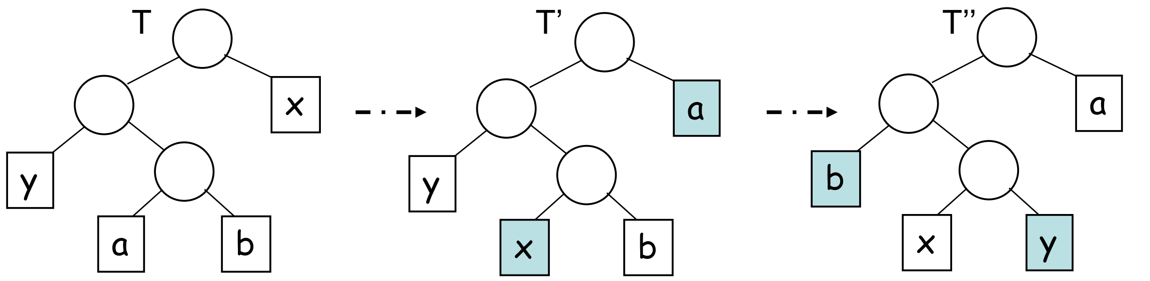
Proof of the Greedy Choice (cont.)



- a, b – two characters, sibling leaves of max. depth in T
- Assume: $f[a] \le f[b]$ and $f[x] \le f[y]$
- $f[x]$ and $f[y]$ are the two lowest leaf frequencies, in order

  $\Rightarrow f[x] \le f[a]$ and $f[y] \le f[b]$
- Exchange the positions of **a** and **x** (T') and of **b** and **y** (T'')

Proof of the Greedy Choice (cont.)



$$B(T) - B(T') = \sum_{c\in C} f(c)d_T(c) - \sum_{c\in C} f(c)d_{T'}(c)$$

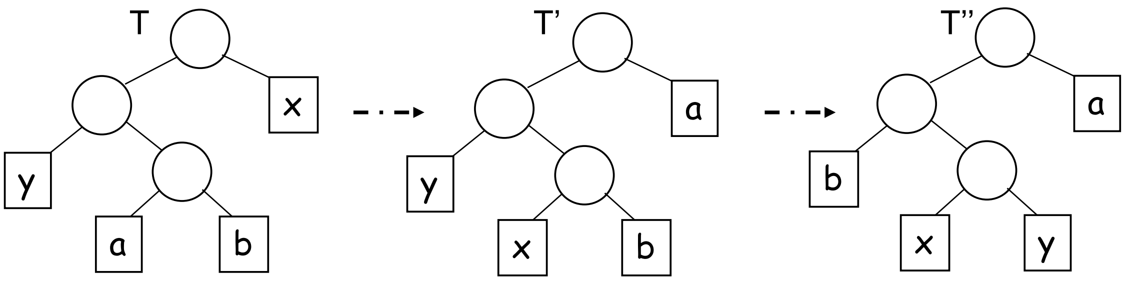
$$= f[x]d_T(x) + f[a]d_T(a) - f[x]d_{T'}(x) - f[a]d_{T'}(a)$$

$$= f[x]d_T(x) + f[a]d_T(a) - f[x]d_T(a) - f[a]d_T(x)$$

$$= \underbrace{(f[a] - f[x])}_{\substack{\ge 0 \\ \text{x is a minimum} \\ \text{frequency leaf}}} \underbrace{(d_T(a) - d_T(x))}_{\substack{\ge 0 \\ \text{a is a leaf of} \\ \text{maximum depth}}}$$

≥ 0

Proof of the Greedy Choice (cont.)



$B(T) - B(T') \geq 0$

Similarly, exchanging **y** and **b** does not increase the cost:

$B(T') - B(T'') \geq 0$

$\Rightarrow B(T'') \leq B(T)$. Also, since T is optimal, $B(T) \leq B(T'')$

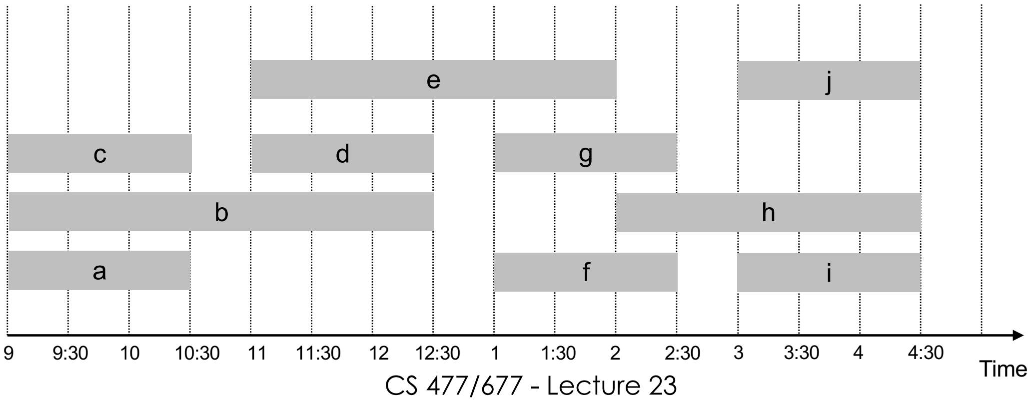
Therefore, $B(T) = B(T'') \Rightarrow T''$ is an optimal tree, in which x and y are sibling leaves of maximum depth

Discussion

- Greedy choice property:
  - Building an optimal tree by mergers can begin with the greedy choice: merging the two characters with the lowest frequencies
  - The cost of each merger is the sum of frequencies of the two items being merged
  - Of all possible mergers, HUFFMAN chooses the one that incurs the least cost

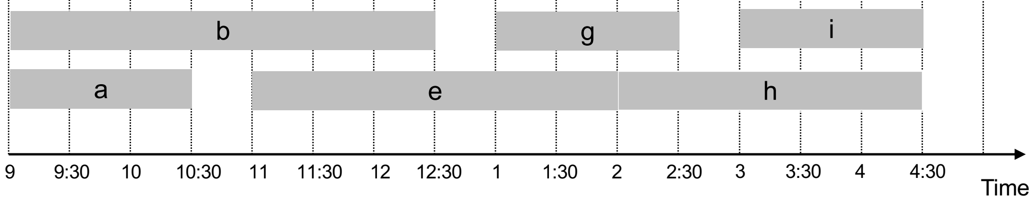
Interval Partitioning

- Lecture j starts at s_j and finishes at f_j
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room
  - Ex: this schedule uses 4 classrooms to schedule 10 lectures



Interval Partitioning

- Lecture j starts at s_j and finishes at f_j
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room
  - Ex: this schedule uses only 3

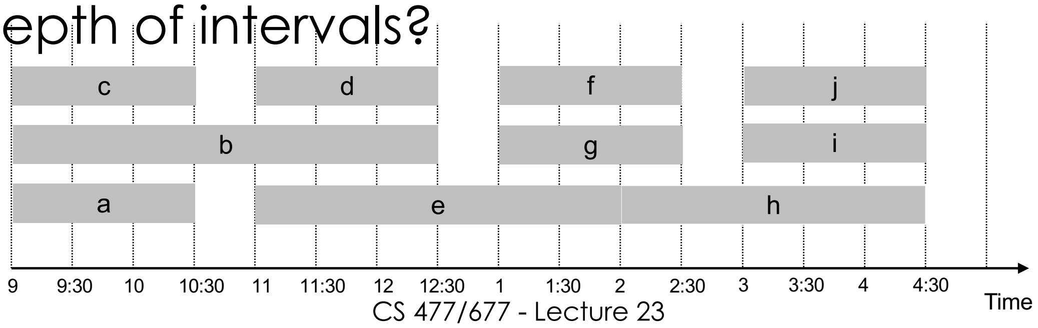


Interval Partitioning: Lower Bound on Optimal Solution

- The **depth** of a set of open intervals is the maximum number that contain any given time
- Key observation:
  - The number of classrooms needed ≥ depth
- Ex: Depth of schedule below = 3 ⇒ schedule below is optimal

a, b, c all contain 9:30

- Does there always exist a schedule equal to depth of intervals?



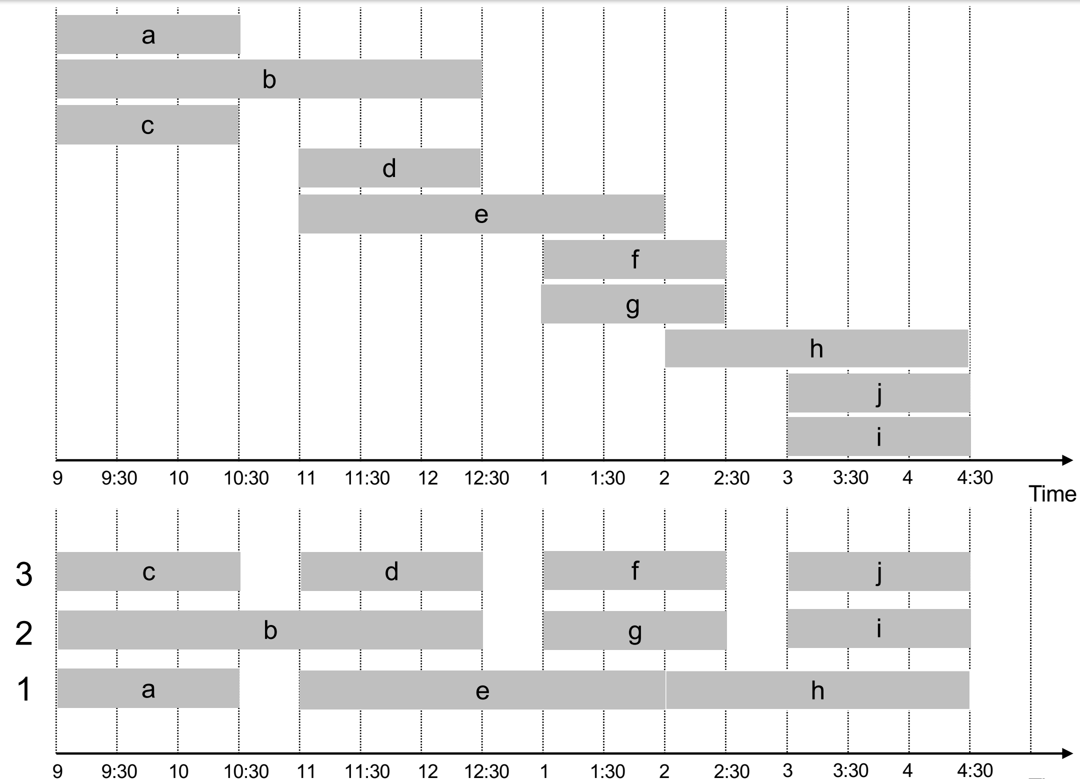
Greedy Strategy

- Consider lectures in increasing order of start time: assign lecture to any compatible classroom
  - Labels set {1, 2, 3, …, d}, where d is the depth of the set of intervals
  - Overlapping intervals are given different labels
  - Assign a label that has not been assigned to any previous interval that overlaps it

Greedy Algorithm

1. Sort intervals by start times, such that $s_1 \le s_2 \le \ldots \le s_n$
   (let $I_1, I_2, .., I_n$ denote the intervals in this order)
2. **for** j = 1 **to** n
3. Exclude from set {1, 2, …, d} the labels of preceding and overlapping intervals I_i from consideration for I_j
4. **if** there is any label from {1, 2, …, d} that was not excluded
   assign that label to I_j
5. **else**
6. leave I_j unlabeled

Example



Claim

- Every interval will be assigned a label
  - For interval I_j, assume there are t intervals earlier in the sorted order that overlap it
  - We have $t + 1$ intervals that pass over a common point on the timeline
  - $t + 1 \le d$, thus $t \le d - 1$
  - At least one of the d labels is not excluded by this set of t intervals, which we can assign to I_j

Claim

- No two overlapping intervals are assigned the same label
  - Consider I and I' that overlap, and I precedes I' in the sorted order
  - When I' is considered, the label for I is excluded from consideration
  - Thus, the algorithm will assign a different label to I

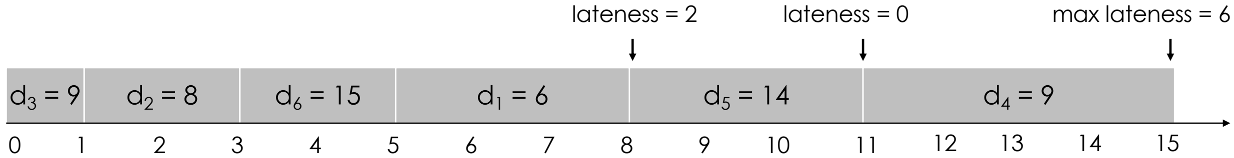
Greedy Choice Property

- The greedy algorithm schedules every interval on a resource, using a number of resources equal to the depth of the set of intervals. This is the optimal number of resources needed.
- Proof:
  - Follows from previous claims
- Structural proof
  - Discover a simple "structural" bound asserting that every possible solution must have a certain value
  - Then show that your algorithm always achieves this bound

Scheduling to Minimizing Lateness

- Single resource processes one job at a time
- Job j requires t_j units of processing time, is due at time d_j
- If j starts at time s_j, it finishes at time $f_j = s_j + t_j$
- Lateness: $\ell_j = \max \{ 0, f_j - d_j \}$
- Goal: schedule all jobs to minimize **maximum** lateness $L = \max \ell_j$
- Example:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Greedy Algorithms

- Greedy strategy: consider jobs in some order
  - **[Shortest processing time first]** Consider jobs in ascending order of processing time t_j

counterexample

	1	2
t_j	1	10
d_j	100	10

Choosing t_1 first: $l_2 = 1$
Choosing t_2 first: $l_2 = l_1 = 0$

  - **[Smallest slack]** Consider jobs in ascending order of slack $d_j - t_j$

counterexample

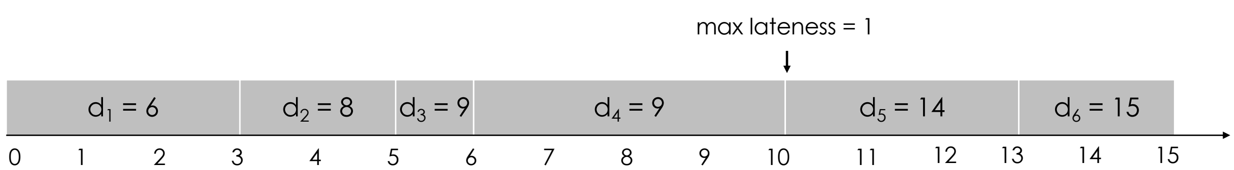
	1	2
t_j	1	10
d_j	2	10

Choosing t_2 first: $l_1 = 9$
Choosing t_1 first: $l_1 = 0$ and $l_2 = 1$

Greedy Algorithm

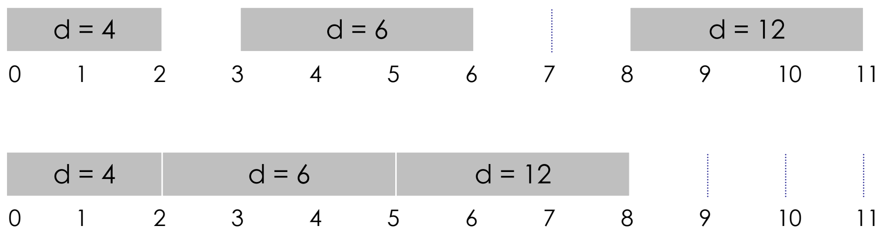
- Greedy choice: earliest deadline first

```
Sort n jobs by deadline so that d1 < d2 <… < dn

t = 0
for j = 1 to n
   Assign job j to interval [t, t + tj]
   sj = t, fj = t + tj
   t = t + tj
output intervals [sj, fj]
```



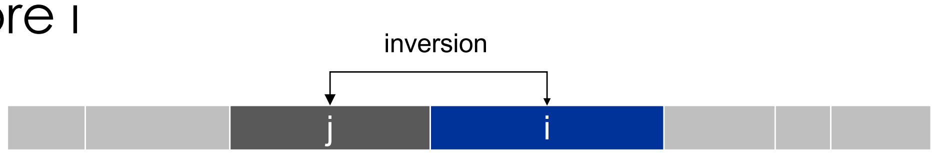
Minimizing Lateness: No Idle Time

- Observation: The greedy schedule has no idle time
- Observation: There exists an optimal schedule with no **idle time**



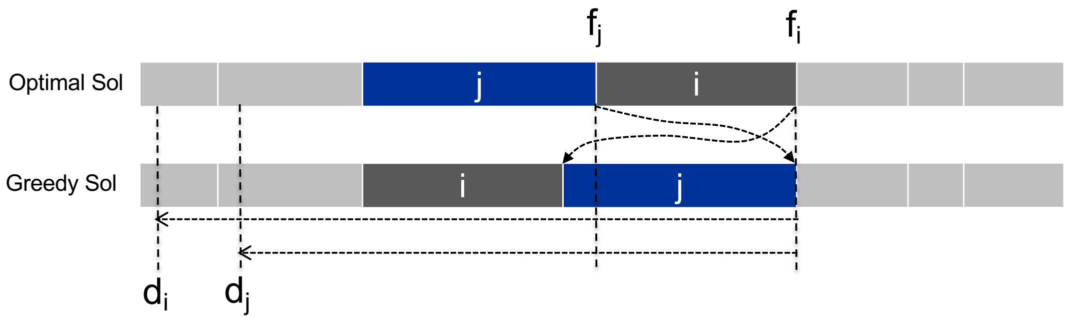
Minimizing Lateness: Inversions

- An inversion in schedule S is a pair of jobs i and j such that: $d_i < d_j$ but j scheduled before i



- Observation: greedy schedule has no inversions

Greedy Choice Property



- Optimal solution: $d_i < d_j$ but j scheduled before i
- Greedy solution: i scheduled before j
  - Job i finishes sooner, no increase in latency

  Lateness(Job j)$_{GREEDY}$ = $f_i - d_j$

  $\leq$ ➔ No increase in latency

  Lateness(Job i)$_{OPT}$ = $f_i - d_i$

Greedy Analysis Strategies

- Exchange argument
  - Gradually transform any solution to the one found by the greedy algorithm without hurting its quality
- Structural
  - Discover a simple "structural" bound asserting that every possible solution must have a certain value, then show that your algorithm always achieves this bound
- Greedy algorithm stays ahead
  - Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's

Coin Changing

- Given currency denominations: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins

- Ex: 34¢

- Ex: $2.89

Greedy Algorithm

- Greedy strategy: at each iteration, add coin of the largest value that does not take us past the amount to be paid

```
Sort coins denominations by value: c_1 < c_2 < … < c_n.

  coins selected
S = {}
while (x > 0) {
   let k be largest integer such that c_k <= x
   if (k = 0)
      return "no solution found"
   x = x - c_k
   S = S  U  {k}
}
return S
```

Greedy Choice Property

- Algorithm is optimal for U.S. coinage: 1, 5, 10, 25, 100

Change = D * 100 + Q * 25 + D * 10 + N * 5 + P

  - Consider optimal way to change $c_k <= x < c_{k+1}$: greedy takes coin k
  - We claim that any optimal solution must also take coin k
  - If not, it needs enough coins of type $c_1, \ldots, c_{k-1}$ to add up to x
  - Problem reduces to coin-changing x - c_k cents, which, by induction, is optimally solved by greedy algorithm

Greedy Choice Property

- Algorithm is optimal for U.S. coinage: 1, 5, 10, 25, 100

Change = Dl * 100 + Q * 25 + D * 10 + N * 5 + P

- Optimal solution: Dl Q D N P
- Greedy solution: Dl' Q' D' N' P'

1. Value < 5
   - Both optimal and greedy use the same # of coins
2. 10 (D) > Value > 5 (N)
   - Greedy uses one N and then pennies after that
   - If OPT does not use N, then it should use pennies for the entire amount => could replace 5 P for 1 N

Greedy Choice Property

Change = Dl * 100 + Q * 25 + D * 10 + N * 5 + P

- Optimal solution: Dl   Q   D   N   P
- Greedy solution: Dl'   Q'   D'   N'   P'

3. 25 (Q) > Value > 10 (D)
   - Greedy uses dimes (D's)
   - If OPT does not use D's, it needs to use either 2 coins (2 N), or 6 coins (1 N and 5 P) or 10 coins (10 P) to cover 10 cents
   - Could replace those with 1 D for a better solution

Greedy Choice Property

Change = Dl * 100 + Q * 25 + D * 10 + N * 5 + P

- Optimal solution: Dl   Q   D   N   P
- Greedy solution: Dl'   Q'   D'   N'   P'

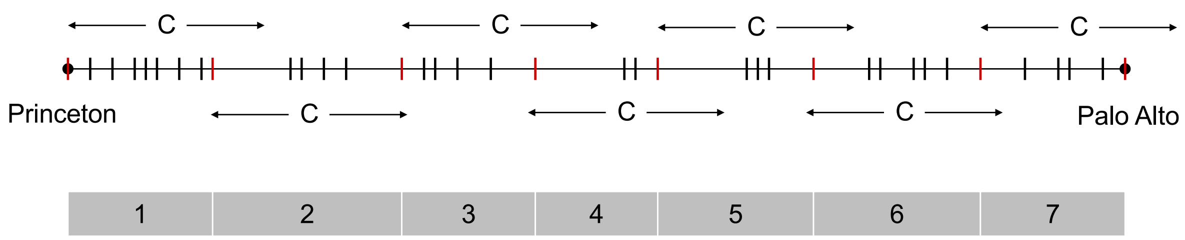
4. 100 (Dl) > Value > 25 (Q)
   - Greedy picks at least one quarter (Q), OPT does not
   - If OPT has no Ds: take all the Ns and Ps and replace 25 cents into one quarter (Q)
   - If OPT has 2 or fewer dimes: it uses at least 3 coins to cover one quarter, so we can replace 25 cents with 1 Q
   - If OPT has 3 or more dimes (e.g., 40 cents: with 4 Ds): take the first 3 Ds and replace them with 1 Q and 1 N

Coin-Changing US Postal Denominations

- Observation: greedy algorithm is sub-optimal for US postal denominations:
  - $.01, .02, .03, .04, .05, .10, .20, .32, .40, .44, .50, .64, .65, .75, .79, .80, .85, .98
  - $1, $1.05, $2, $4.95, $5, $5.15, $18.30, $18.95
- Counterexample: 160¢
  - Greedy: 105, 50, 5
  - Optimal: 80, 80

Selecting Breakpoints

- Road trip from Princeton to Palo Alto along fixed route
- Refueling stations at certain points along the way (red marks)
- Fuel capacity = C
- Goal:
  - makes as few refueling stops as possible
- Greedy strategy:
  - go as far as you can before refueling



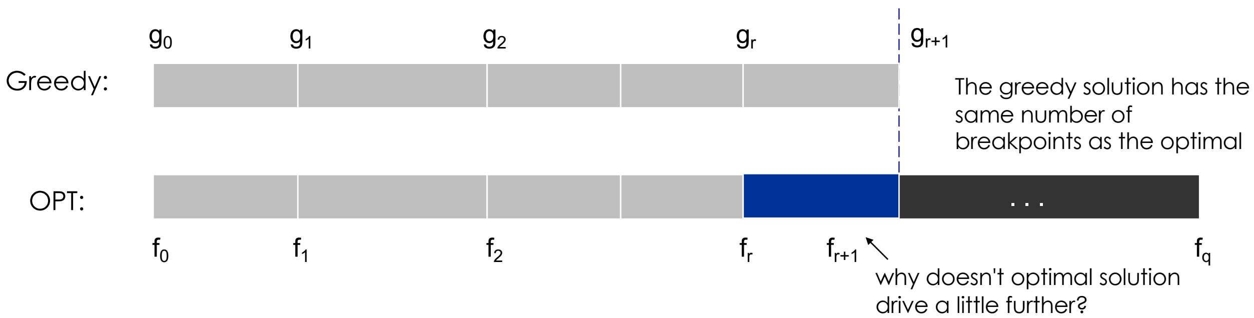
Greedy Algorithm

```
Sort breakpoints so that: 0 = b_0 < b_1 < b_2 < ... < b_n = L

S = {0}          ← breakpoints selected
x = 0            ← current location

while (x < b_n)
   let p be largest integer such that b_p <= x + C
   if (b_p = x)
      return "no solution"
   x = b_p
   S = S  U {p}
return S
```

- Implementation: O(n log n)
  - Use binary search to select each breakpoint p

Greedy Choice Property

- Let $0 = g_0 < g_1 < \ldots < g_p = L$ denote set of breakpoints chosen by the greedy
- Let $0 = f_0 < f_1 < \ldots < f_q = L$ denote set of breakpoints in an optimal solution with $f_0 = g_0$, $f_1 = g_1$, . . . , $f_r = g_r$
- Note: $g_{r+1} > f_{r+1}$ by greedy choice of algorithm



Problem – Buying Licenses

- Your company needs to buy licenses for **n** pieces of software
- Licenses can be bought only one per month
- Each license currently sells for $100, but becomes more expensive each month
  - The price increases by a factor $r_j > 1$ each month
  - License j will cost $100 * r_j^t$ if bought t months from now
  - $r_i < r_j$ for license $i < j$
- In which order should the company buy the licenses, to minimize the amount of money spent?

Solution

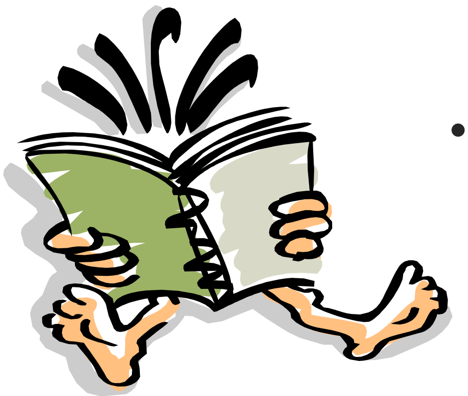
- Greedy choice:
  - Buy licenses in decreasing order of rate r_j
  - $r_1 > r_2 > r_3 \ldots$
- Proof of greedy choice property
  - Optimal solution: …. $r_i\ r_j$….. $r_i < r_j$
  - Greedy solution: …. $r_j\ r_i$…..
  - Cost by optimal solution: $100 * r_i^t + 100 * r_j^{t+1}$
  - Cost by greedy solution: $100 * r_j^t + 100 * r_i^{t+1}$

$CG - CO = 100 * (r_j^t + r_i^{t+1} - r_i^t - r_j^{t+1}) < 0$

$r_i^{t+1} - r_i^t < r_j^{t+1} - r_j^t$

$r_i^t(r_i - 1) < r_j^t(r_j - 1)$ OK! (because $r_i < r_j$)

Readings



- Chapters 14, 15