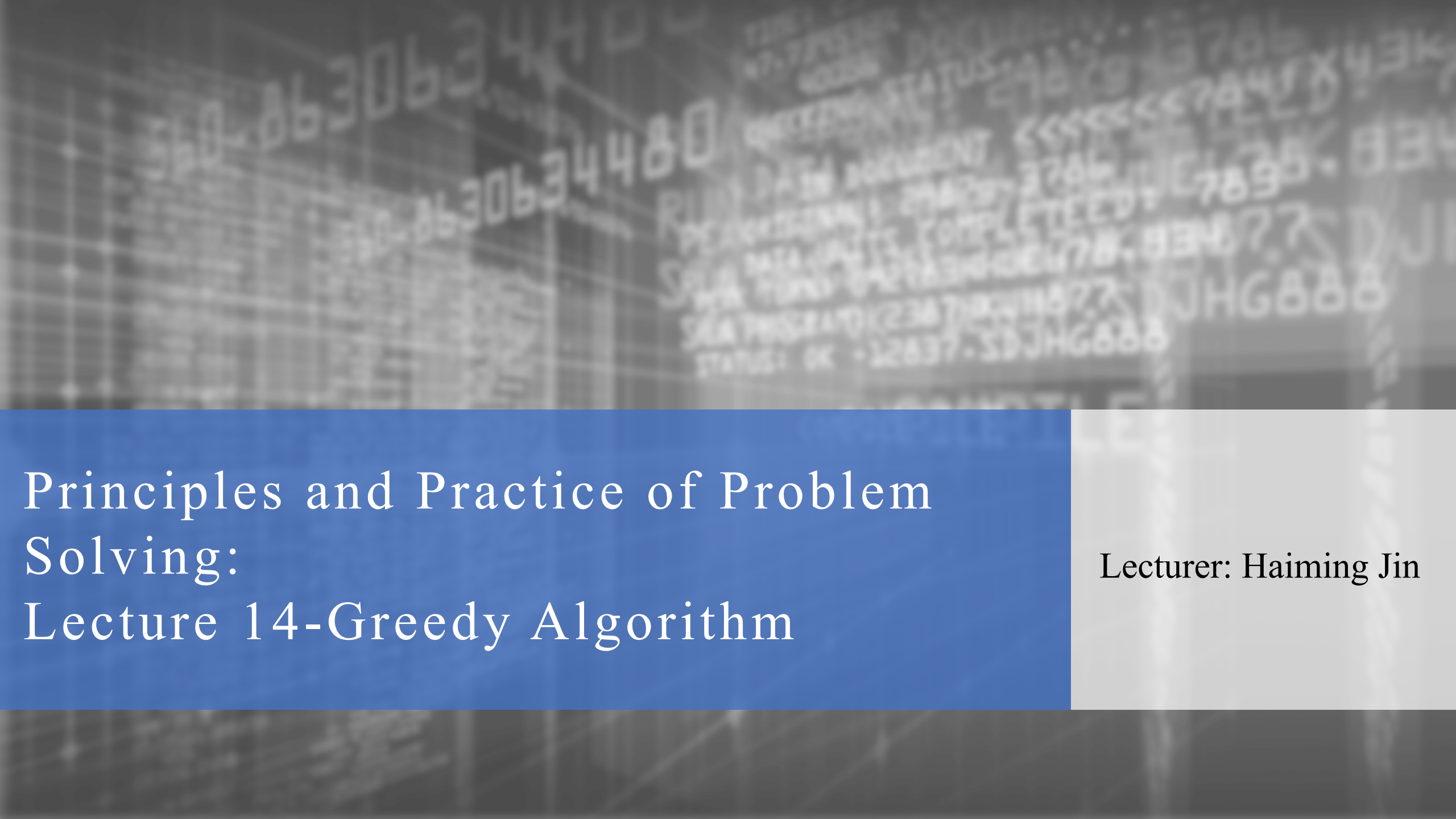


Principles and Practice of Problem Solving: Lecture 14-Greedy Algorithm

Lecturer: Haiming Jin

Outline

- Counting Money Problem
- Greedy Algorithm and Activity Selection Problem
- More Examples

Example: Counting money

- Suppose you want to count out a certain amount of money, using the fewest possible bills and coins.
- A greedy algorithm that 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.

Outline

- Counting Money Problem
- Greedy Algorithm and Activity Selection Problem
- More Examples

Overview

- Like dynamic programming, used to solve optimization problems.
- Problems exhibit optimal substructure (like DP).
- Problems also exhibit the **greedy-choice** property.
  - When we have a choice to make, make the one that looks best *right now*.
  - Make a **locally optimal choice** in hope of getting a **globally optimal solution**.

Greedy Strategy

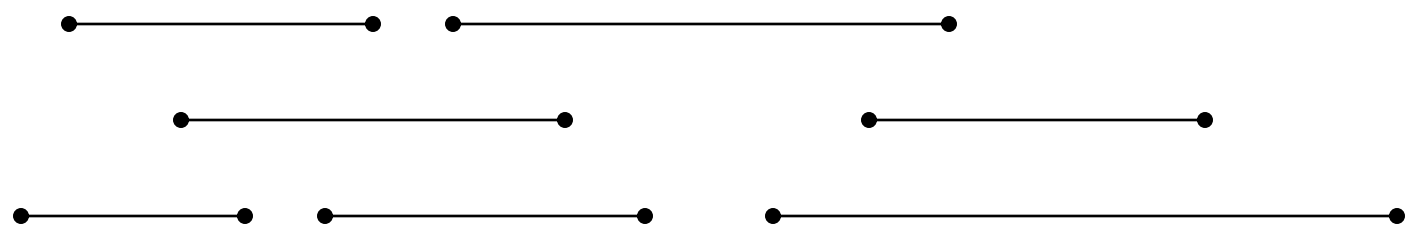
- The choice that seems best at the moment is the one we go with.
  - Prove that when there is a choice to make, one of the optimal choices is the greedy choice. Therefore, it's always safe to make the greedy choice.
  - Show that all but one of the subproblems resulting from the greedy choice are empty.

Activity-selection Problem

- Input: Set S of n activities, $a_1, a_2, \ldots, a_n$.
  - s_i = start time of activity i.
  - f_i = finish time of activity i.
- Output: Subset A of maximum number of compatible activities.
  - Two activities are compatible, if their intervals don't overlap.

Example:

Activities in each line are compatible.



Optimal Substructure

- Assume activities are sorted by finishing times.
  - $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Suppose an optimal solution includes activity a_k.
  - This generates two subproblems.
  - Selecting from $a_1, \ldots, a_{k-1}$, activities compatible with one another, and that finish before a_k starts (compatible with a_k).
  - Selecting from $a_{k+1}, \ldots, a_n$, activities compatible with one another, and that start after a_k finishes.
  - The solutions to the two subproblems must be optimal.

Greedy-choice Property

- The problem also exhibits the greedy-choice property.
  - There is an optimal solution to the subproblem S_{ij}, that includes the activity with the smallest finish time in set S_{ij}.
  - Can be proved easily.
- Hence, there is an optimal solution to S that includes a_1.
- Therefore, make this greedy choice without solving subproblems first and evaluating them.
- Solve the subproblem that ensues as a result of making this greedy choice.
- Combine the greedy choice and the solution to the subproblem.

An Iterative Algorithm for Activity Selection

- 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

A Recursive Algorithm for Activity Selection

Recursive-Activity-Selector (s, f, i, j)

1. $m \leftarrow i+1$
2. **while** $m < j$ and $s_m < f_i$
3. **do** $m \leftarrow m+1$
4. **if** $m < j$
5. **then return** $\{a_m\} \cup$ Recursive-Activity-Selector(s, f, m, j)
6. else return ϕ

Initial Call: Recursive-Activity-Selector (s, f, 0, n+1)

Complexity: $\Theta(n)$

Typical Steps

- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
- Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
- Show that greedy choice and optimal solution to subproblem $\Rightarrow$ optimal solution to the problem.
- Make the greedy choice and **solve top-down**.
- May have to preprocess input to put it into greedy order.
  - Example: Sorting activities by finish time.

Elements of Greedy Algorithms

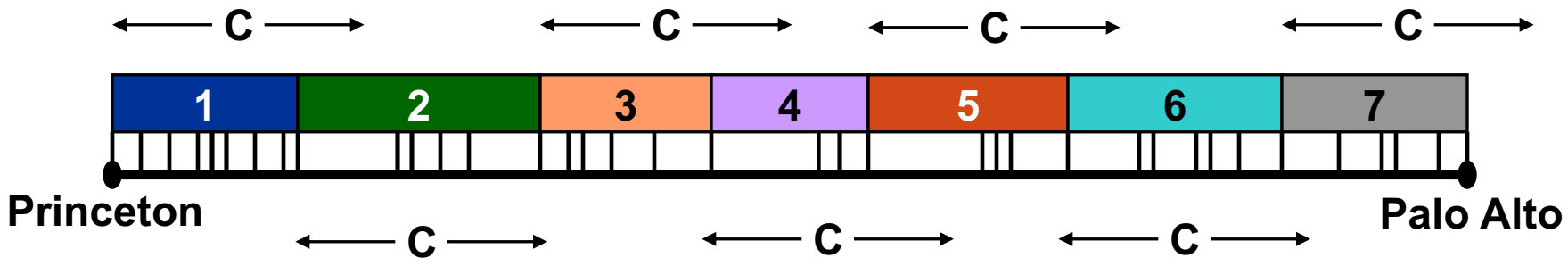
- Greedy-choice Property.
  - A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- Optimal Substructure.

Outline

- Counting Money Problem
- Greedy Algorithm and Activity Selection Problem
- More Examples

Example #1 Selecting Breakpoints

- Minimizing breakpoints.
  - Truck driver going from Princeton to Palo Alto along predetermined route.
  - Refueling stations at certain points along the way.
  - Truck fuel capacity = C.
- Greedy algorithm.
  - Go as far as you can before refueling.



Example #1 Selecting Breakpoints

Greedy Breakpoint Selection Algorithm

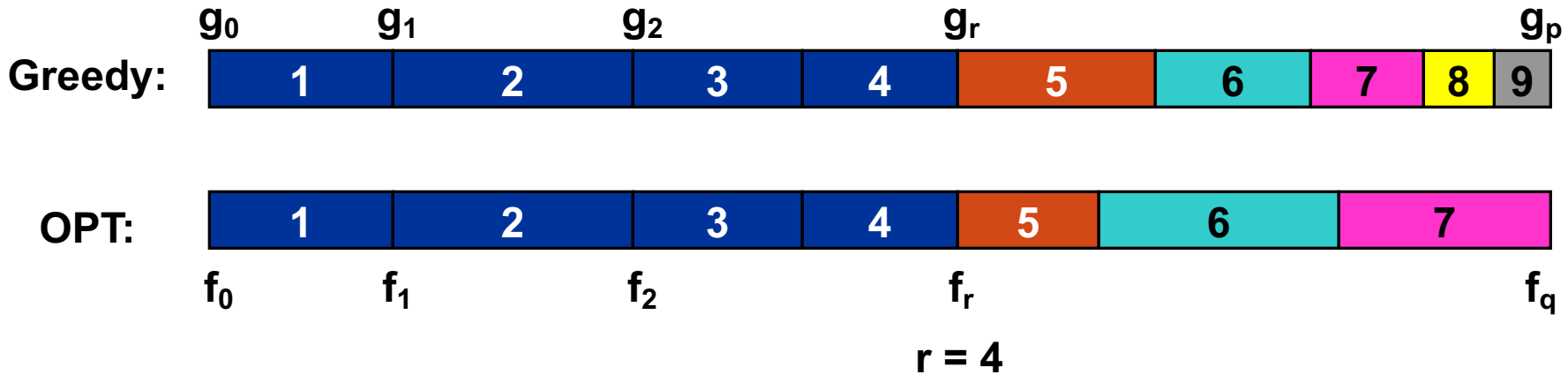
```
Sort breakpoints by increasing value:
0 = b0 < b1 < b2 < ... < bn.

S ← {0}
x = 0
while (x ≠ bn)
   let p be largest integer such that bp ≤ x + C
   if (bp = x)
      return "no solution"
   x ← bp
   S ← S ∪ {p}
return S
```

S = breakpoints selected.

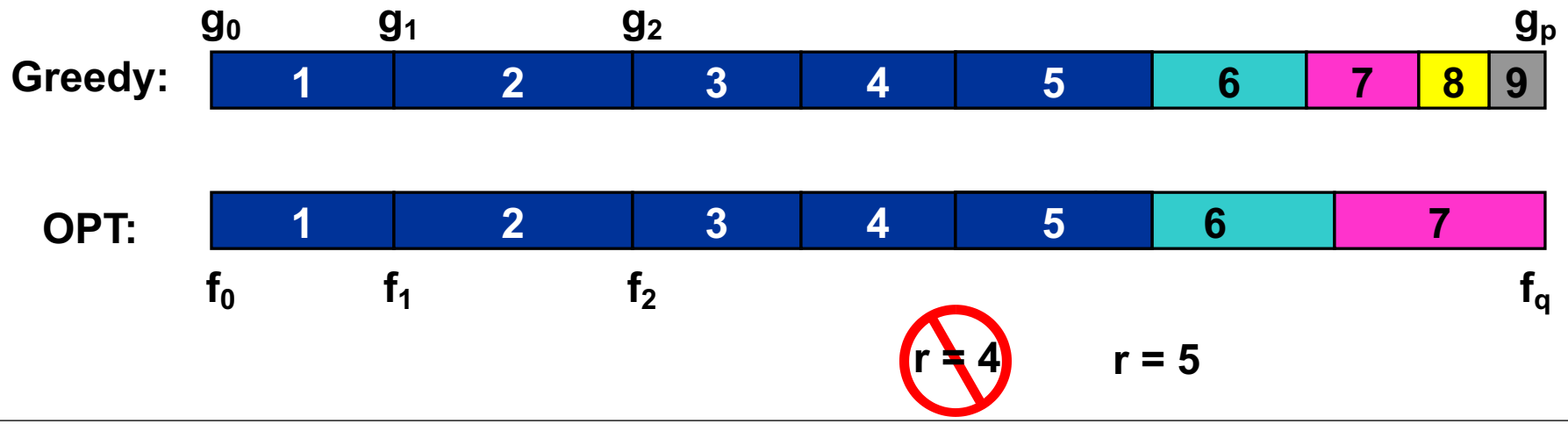
Example #1 Selecting Breakpoints

- **Theorem**: greedy algorithm is optimal.
- Proof (by contradiction):
  - Let $0 = g_0 < g_1 < \ldots < g_p = L$ denote set of breakpoints chosen by greedy and assume it is not optimal.
  - Let $0 = f_0 < f_1 < \ldots < f_q = L$ denote set of breakpoints in optimal solution with $f_0 = g_0$, $f_1 = g_1, \ldots, f_r = g_r$ for largest possible value of r.
  - Note: $q < p$.



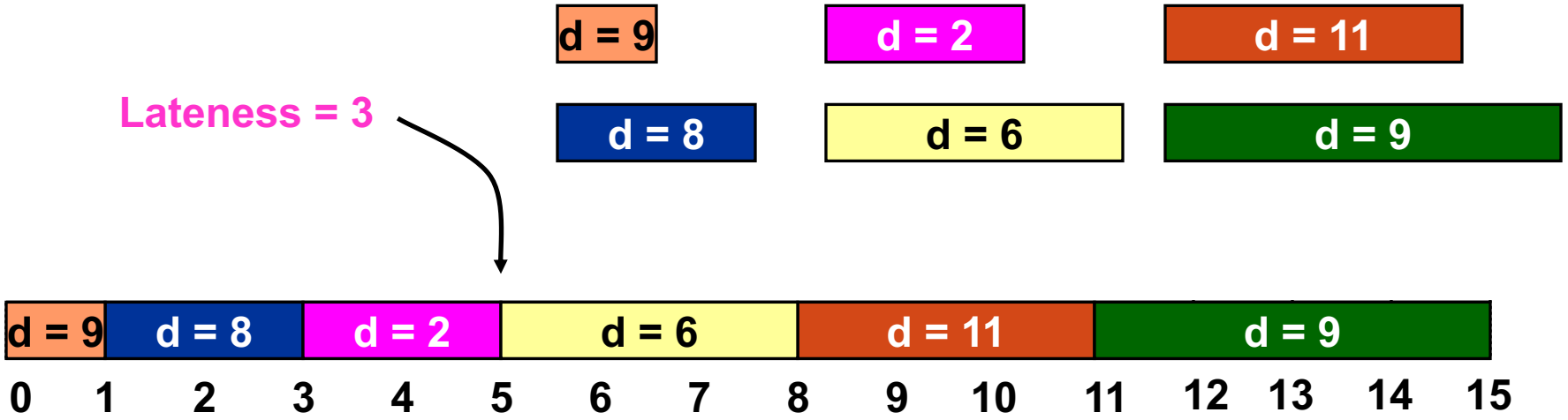
Example #1 Selecting Breakpoints

- **Theorem**: greedy algorithm is optimal.
- Proof (by contradiction):
  - Let $0 = g_0 < g_1 < \ldots < g_p = L$ denote set of breakpoints chosen by greedy and assume it is not optimal.
  - Let $0 = f_0 < f_1 < \ldots < f_q = L$ denote set of breakpoints in optimal solution with $f_0 = g_0$, $f_1 = g_1$, . . . , $f_r = g_r$ for largest possible value of r.
  - Note: $q < p$.



Example #2: Minimizing Lateness

- Minimizing lateness problem.
  - Single resource can process one job at a time.
  - n jobs to be processed.
    - job j requires p_j units of processing time.
    - job j has due date d_j.
  - If we assign job j to start at time s_j, it finishes at time $f_j = s_j + p_j$.
  - Lateness: $\ell_j = \max \{ 0, f_j - d_j \}$.
  - Goal: schedule all jobs to minimize maximum lateness $L = \max \ell_j$.

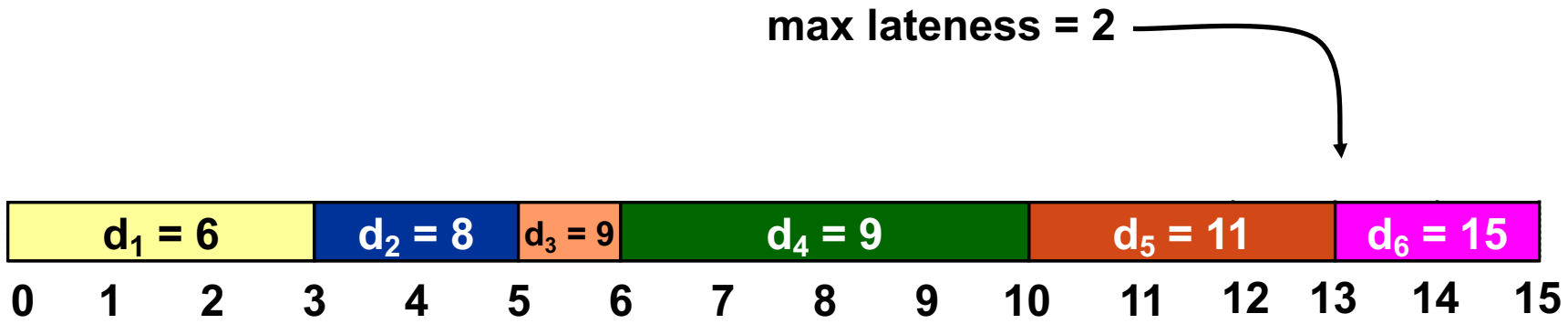


Example #2: Minimizing Lateness

Greedy Job Scheduling Algorithm

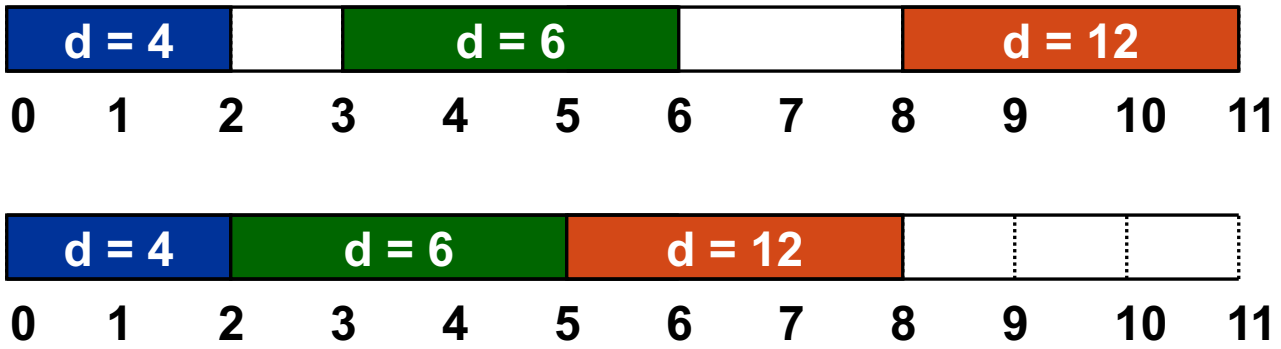
Sort jobs by increasing deadline so that $d_1 \le d_2 \le \ldots \le d_n$.

```
t = 0
for j = 1 to n
   Assign job j to interval [t, t + p_j]
   s_j ← t, f_j ← t + p_j
   t ← t + p_j
output intervals [s_j, f_j]
```



Example #2: Minimizing Lateness

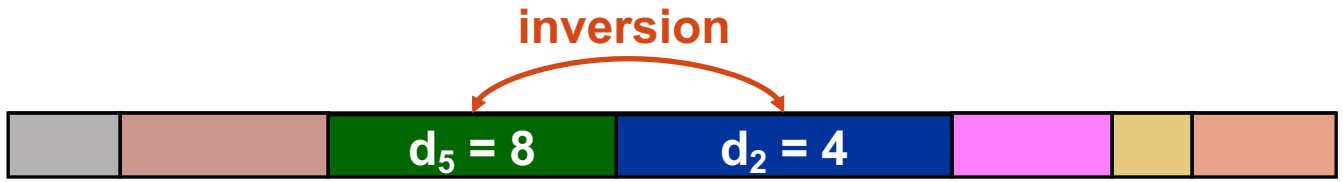
- Fact 1: there exists an optimal schedule with no idle time.



- Fact 2: the greedy schedule has no idle time.

Example #2: Minimizing Lateness

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

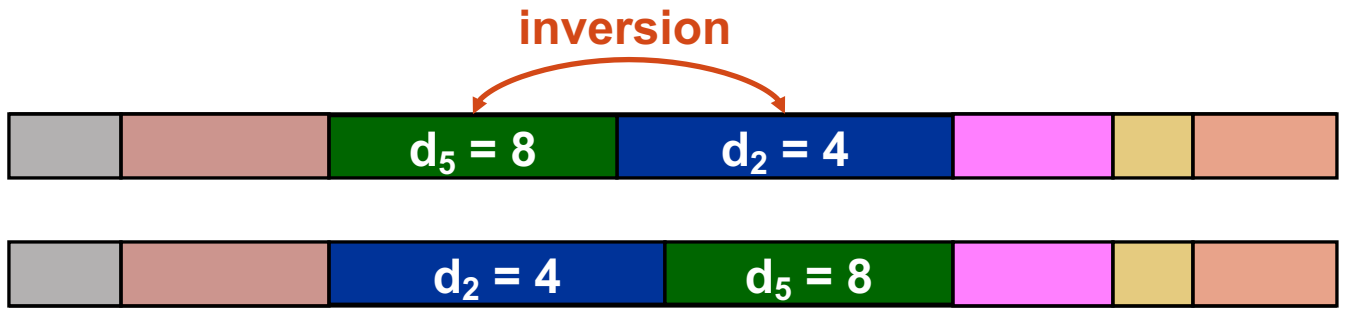


- Fact 3: greedy schedule $\Leftrightarrow$ no inversions.

- Fact 4: if a schedule (with no idle time) has an inversion, it has one whose with a pair of inverted jobs scheduled consecutively.

Example #2: Minimizing Lateness

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

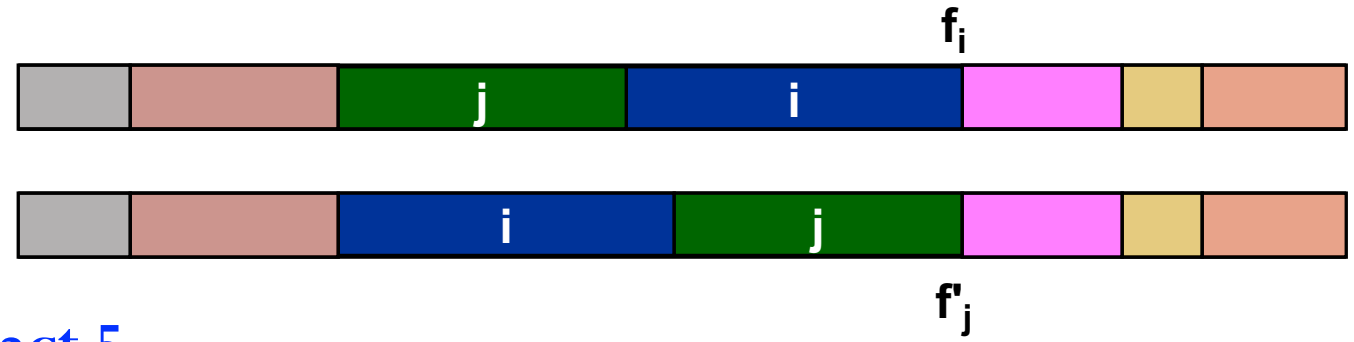


- Fact 3: greedy schedule $\Leftrightarrow$ no inversions.
- Fact 4: if a schedule (with no idle time) has an inversion, it has one whose with a pair of inverted jobs scheduled consecutively.
- Fact 5: swapping two adjacent, inverted jobs:
  - Reduces the number of inversions by one.
  - Does not increase the maximum lateness.
- **Theorem: greedy schedule is optimal.**

How to prove fact 5?

Example #2: Minimizing Lateness

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

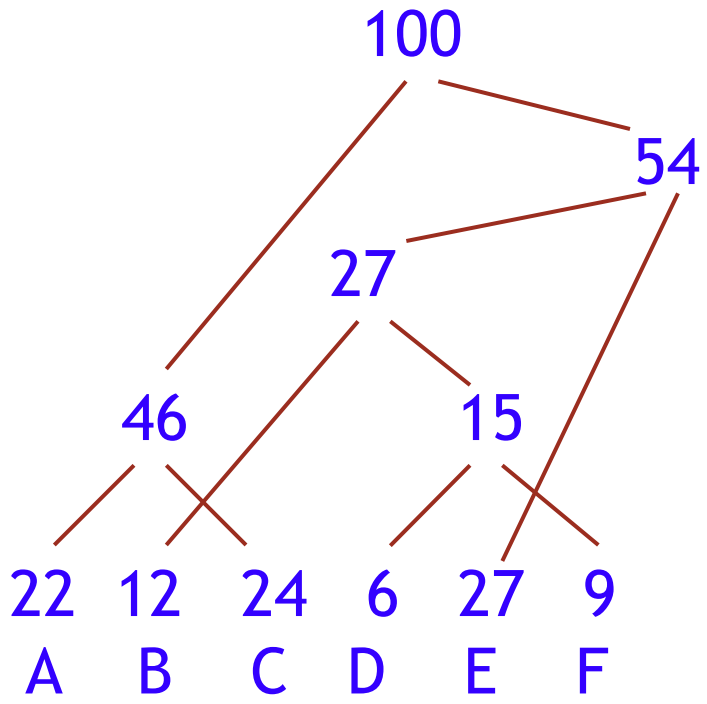


- **Proof of Fact 5**
  - Swapping two adjacent, inverted jobs does not increase max lateness.
    - $\ell'_k = \ell_k$ for all $k \neq i, j$
    - $\ell'_i \leq \ell_i$
    - If job j is late:

$$
\begin{aligned}
\ell'_j &= f'_j - d_j && \text{(definition)} \\
&= f_i - d_j && (j \text{ finishes at time } f_i) \\
&\leq f_i - d_i && (i < j) \\
&\leq \ell_i && \text{(definition)}
\end{aligned}
$$

Example #3: Huffman encoding

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

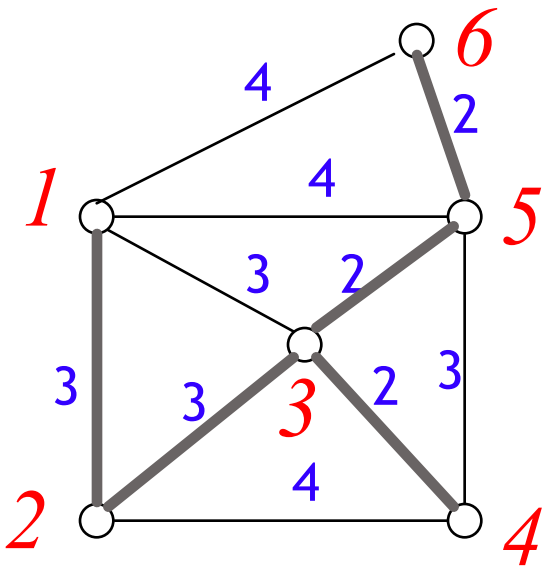


A=00
B=100
C=01
D=1010
E=11
F=1011

- 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

Example #4: Minimum Spanning Tree

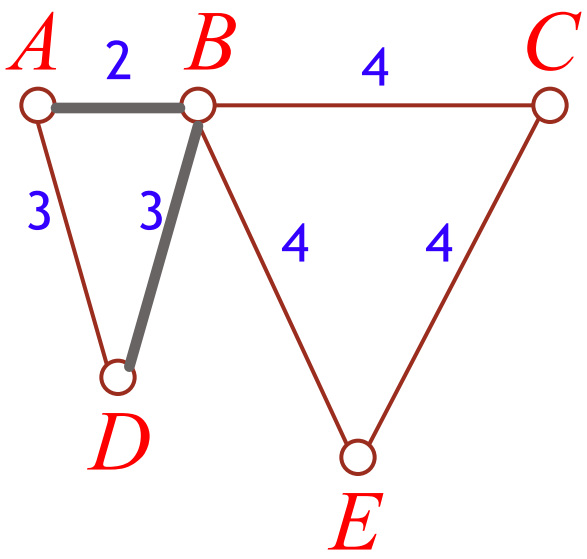
- A minimum spanning tree is a least-cost subset of the edges of a graph that connects all the nodes
  - Start by picking any node and adding it to the tree
  - Repeatedly: Pick any *least-cost* edge from a node in the tree to a node not in the tree, and add the edge and new node to the tree
  - Stop when all nodes have been added to the tree



- The result is a least-cost (3+3+2+2+2=12) spanning tree
- If you think some other edge should be in the spanning tree:
  - Try adding that edge
  - Note that the edge is part of a cycle
  - To break the cycle, you must remove the edge with the greatest cost
    - This will be the edge you just added

Example #5: Traveling salesman

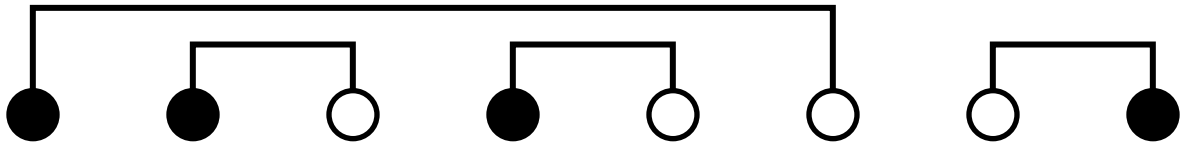
- A salesman must visit every city (starting from city *A*), and wants to cover the least possible distance
  - He can revisit a city (and reuse a road) if necessary
- He does this by using a greedy algorithm: He goes to the next nearest city from wherever he is



- From *A* he goes to *B*
- From *B* he goes to *D*
- This is *not* going to result in a shortest path!
- The best result he can get now will be *ABDBCE*, at a cost of 16
- An actual least-cost path from *A* is *ADBCE*, at a cost of 14

Example #6: Connecting Wires

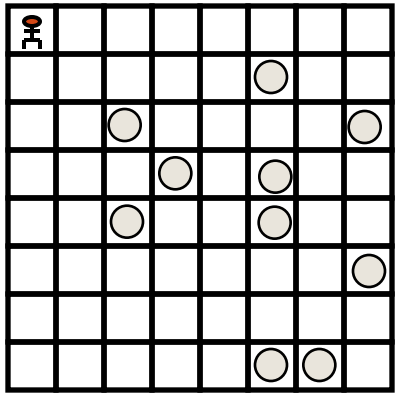
- There are n white dots and n black dots, equally spaced, in a line
- You want to connect each white dot with some one black dot, with a minimum total length of "wire"
- Example:



- Total wire length above is 1 + 1 + 1 + 5 = 8
- Do you see a greedy algorithm for doing this?
- Does the algorithm guarantee an optimal solution?
  - Can you prove it?
  - Can you find a counterexample?

Example #7: Collecting Coins

- A checkerboard has a certain number of coins on it
- A robot starts in the upper-left corner, and walks to the bottom left-hand corner
  - The robot can only move in two directions: right and down
  - The robot collects coins as it goes
- You want to collect *all* the coins using the *minimum* number of robots
- Example:



- Do you see a greedy algorithm for doing this?
- Does the algorithm guarantee an optimal solution?
  - Can you prove it?
  - Can you find a counterexample?

References and Readings

- **Thomas L. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Introduction to Algorithms" (3rd Edition)**
  - Chapter 16 Greedy Algorithms