Greedy Algorithms

A greedy algorithm always makes the choice that looks best at the moment and adds it to the current partial solution.

Greedy algorithms don't always yield optimal solutions, but when they do, they're usually the simplest and most efficient algorithms available.

Outline

1. Intro and Simple Problems

2. Interval Scheduling

3. Knapsack

4. Interval Partitioning

General Idea

Greedy algorithms generate a solution to an *optimization problem* through a sequence of choices that are:

- *Feasible*, i.e. satisfying given constraints
- *Locally optimal* (make the choice that looks best at the moment, without considering the future)
- *Irrevocable* (no backtracking)

For some problems, greedy algorithms provide globally optimal solutions for every instance. For other problems, they generate fast approximate solutions.

Often, they involve a sorting step that dominates the total cost.

Exercise on Minimum number of Coins

Input: Amount n, and coins of denominations $d_1 > \cdots > d_m$

Output: Minimum number of coins for amount n

Example: $n = 48c,\ \ d_1 = 25, d_2 = 10c, d_3 = 5c, d_4 = 1c$

Greedy solution: Give as many coins as possible from the largest coin, then as many as possible from second largest and so on.

- 48=1x25+2X10+3X1

Greedy solution may not be optimal for arbitrary coin denominations

- $n = 30c,\ \ d_1 = 25, d_2 = 10c, d_3 = 1c$

Exercise on Max Sum of Non-Adjacent Elements

Given an array of n positive numbers $A[1], A[2], \dots, A[n]$. Our goal is to pick out a subset of non-adjacent elements whose sum is maximized.

For example, if the array is (1, 8, 6, 3, 6), then the elements chosen should be $A[2]$ and $A[5]$, whose sum is 14.

Describe a greedy algorithm and discuss whether it is optimal or not

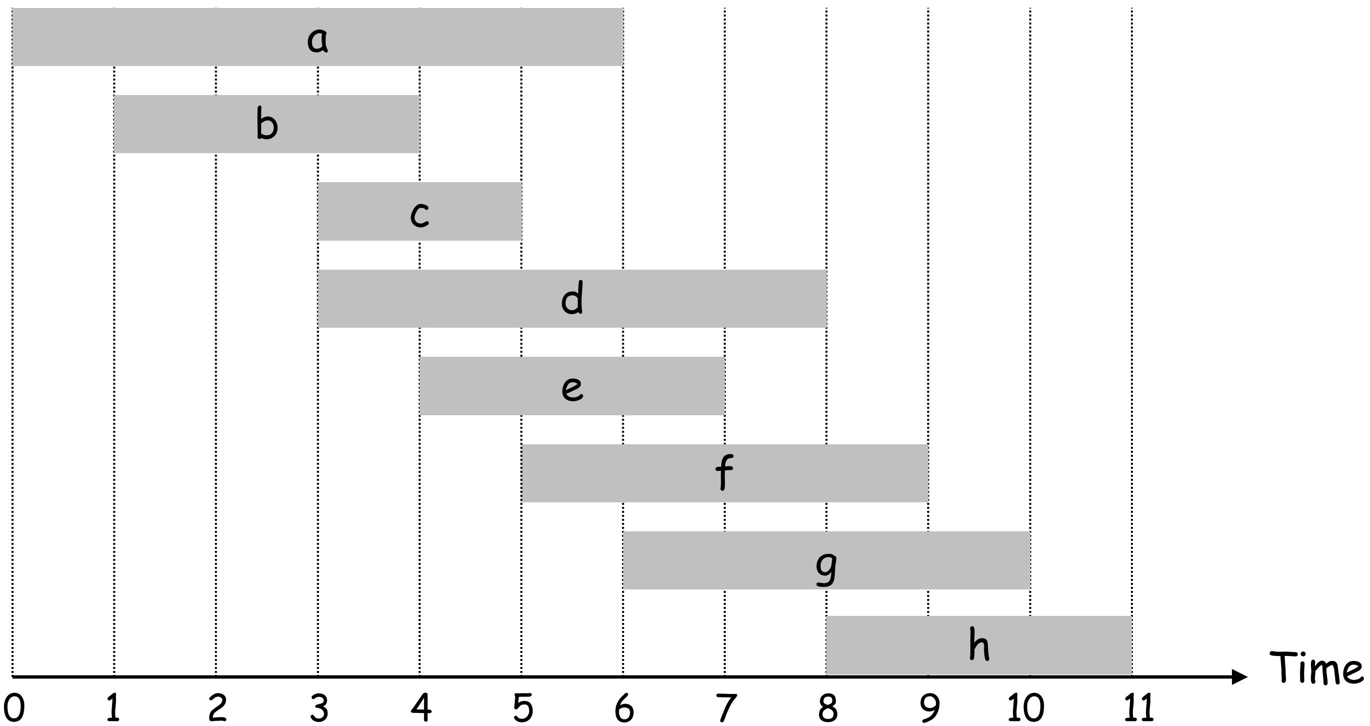
Start with S equal to the empty set
While some elements remain in A
    Pick the largest $A[i]$
    Add $A[i]$ to S
    Mark $A[i]$ and its neighbors as deleted
End while
Return S

Counter-example of optimality. Input: (1, 4, 6, 4).
The above algorithm returns $S = \{1,6\}$ with sum=7.
The optimal solution is $S = \{4,4\}$ with sum=8.

Interval Scheduling

Interval scheduling.

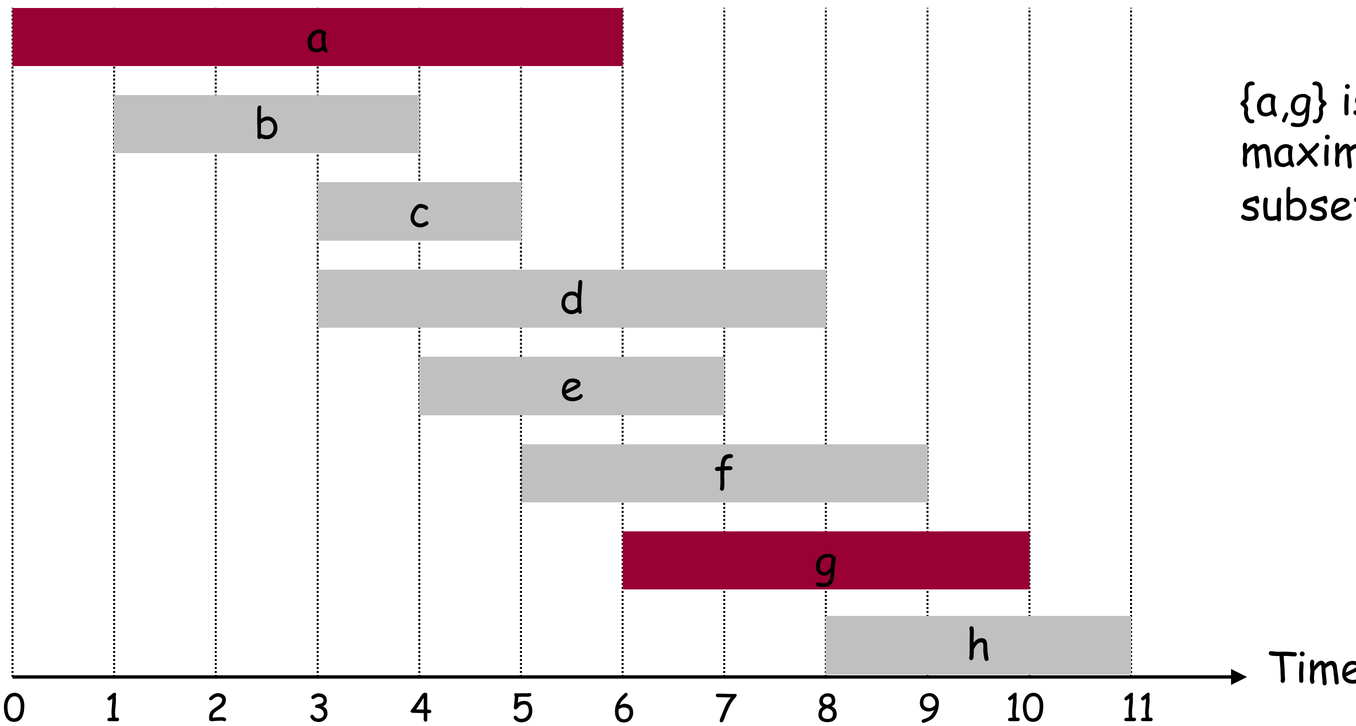
- Job j starts at s_j and finishes at f_j.
- Two jobs are compatible if they don't overlap.
- Goal: find maximum size subset of mutually compatible jobs.



Interval Scheduling

Interval scheduling.

- Job j starts at s_j and finishes at f_j.
- Two jobs are compatible if they don't overlap.
- Goal: find maximum size subset of mutually compatible jobs.

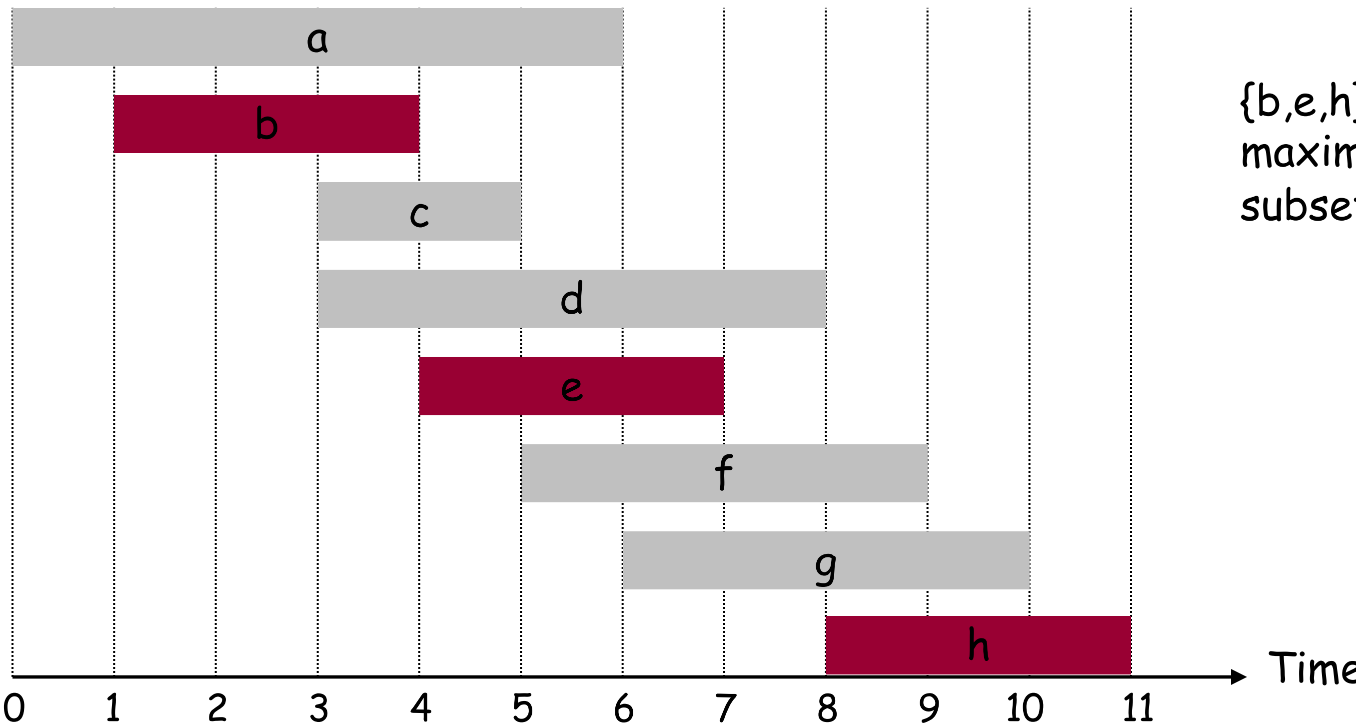


{a,g} is NOT a maximum-size subset.

Interval Scheduling

Interval scheduling.

- Job j starts at s_j and finishes at f_j.
- Two jobs are compatible if they don't overlap.
- Goal: find maximum size subset of mutually compatible jobs.



{b,e,h} is a maximum-size subset.

Interval Scheduling: Greedy Algorithms

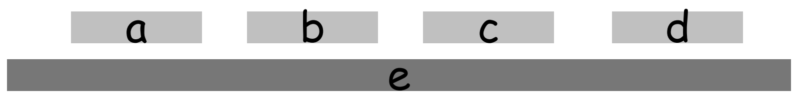
Greedy template. Consider jobs in some order.
Take a job provided it's *compatible* with the ones already taken.

Usually, many compatible jobs can co-exist.
Need to choose a rule specifying which job to choose next.
Three *possible* rules are:

- [Earliest start time]
  Consider jobs in increasing order of start time s_j.
- [Shortest interval]
  Consider jobs in increasing order of interval length $f_j - s_j$.
- [Fewest conflicts]
  For each job, count the number of conflicting jobs c_j.
  Schedule in ascending order of conflicts c_j.

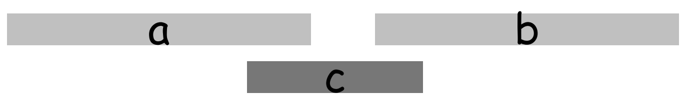
Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order.
Take a job provided it's compatible with the ones already taken.



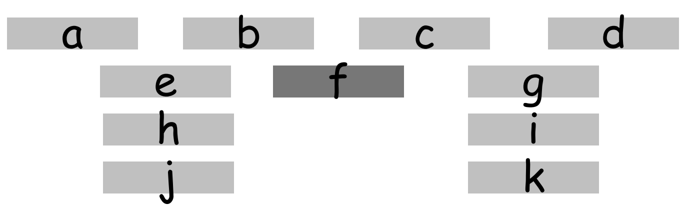
Order on earliest start time

Chooses {e} instead of {a,b,c,d}



Order on shortest interval

Chooses {c} instead of {a,b}



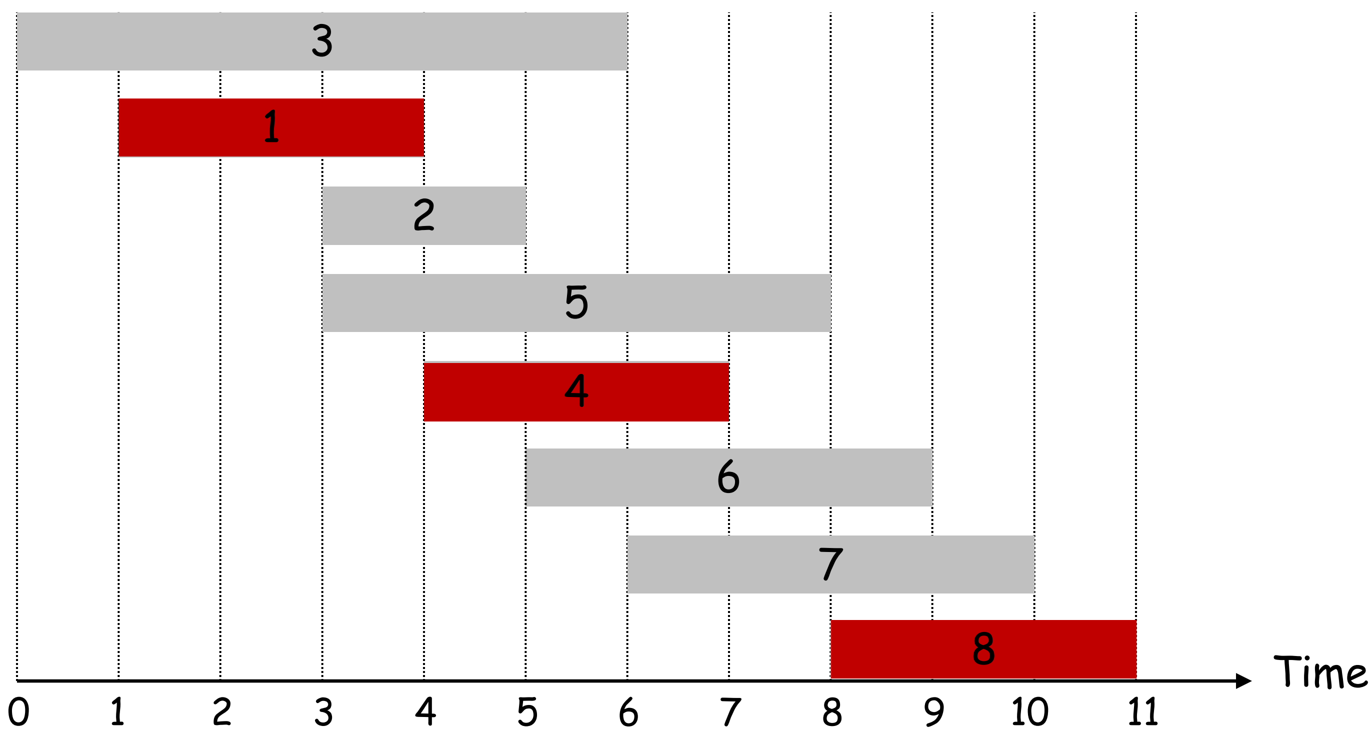
Order on fewest conflicts

Chooses {f} which forces choosing {a,f,d} instead of {a,b,c,d}

Examples above provide ***counterexamples*** for the three proposed rules. For each, there is an input for which the rule yields a non-optimal (i.e., non max-size) schedule.

Interval Scheduling: Greedy Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time.
Take each job provided it's compatible with the ones already taken



Interval Scheduling: Greedy Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

Intuition: leaves maximum interval for scheduling the rest of the jobs.

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn
A ← ∅, last ← 0
for j ← 1 to n
    if sj ≥ last then A ← A ∪ {j}, last ← fj
return A
```

Running time dominated by cost of sorting: $\Theta(n \log n)$.

- Remember the finish time of the last job added to A.
- Job j is compatible with A if $s_j \geq last$.

Remember:

Correctness (optimality) of greedy algorithms is usually not obvious.

Need to prove!

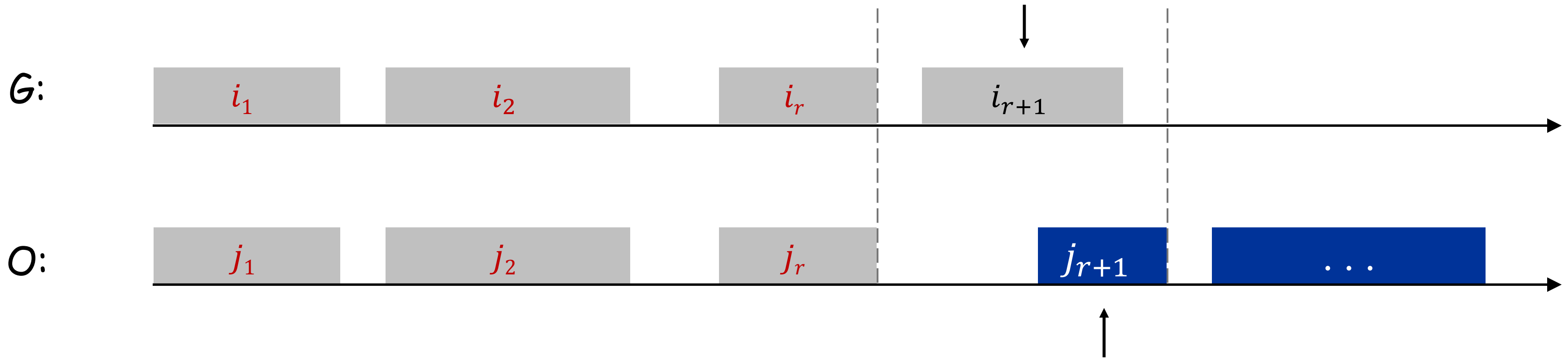
Interval Scheduling: Correctness

Theorem. Greedy algorithm is optimal.

Proof.

- Assume that the solution G of Greedy is different from O, the optimal solution.
- Let $i_1, i_2, \ldots\ i_k$ denote the set of jobs selected by greedy.
- Let $j_1, j_2, \ldots\ j_m$ denote set of jobs in the optimal solution.
  Find largest value of r such that $i_1 = j_1,\ \ i_2 = j_2,\ \ \ldots\ ,\ \ i_r = j_r$

By definition of Greedy, job i_{r+1} finishes before j_{r+1}



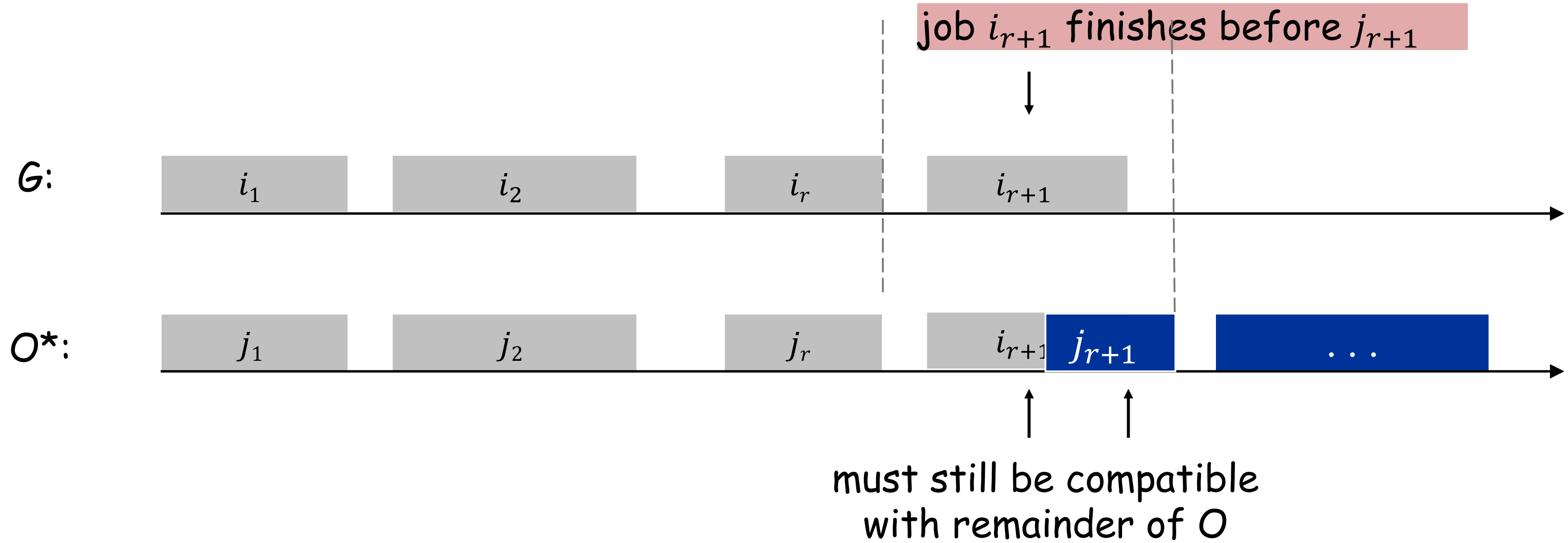
Interval Scheduling: Correctness

Theorem. Greedy algorithm is optimal.

Proof (Continued)

- Assume G is different from O.
- Choose largest r such that $i_t = j_t$ for $t \le r$ and $i_{r+1} \neq j_{r+1}$.
- Create O* from O by replacing j_{r+1} with i_{r+1}.

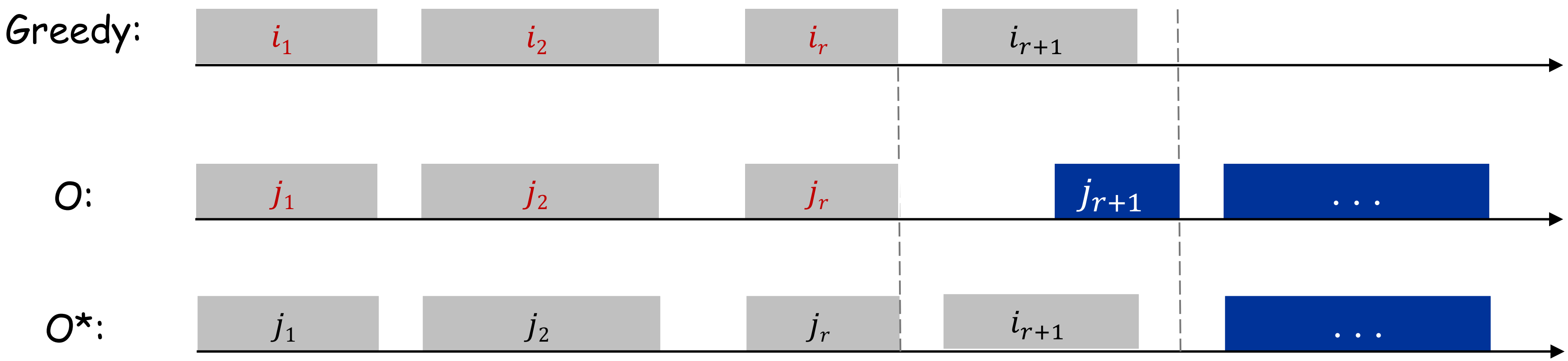
- O* is still a legal solution and has same size as O.
- $\Rightarrow$ O* is also Optimal



Interval Scheduling: Correctness

Proof. So far

- Assumed G≠O.
- Let r be such that O shares first r items with G.
- Process creates new Optimal solution O*, which shares first $r+1$ items with Greedy

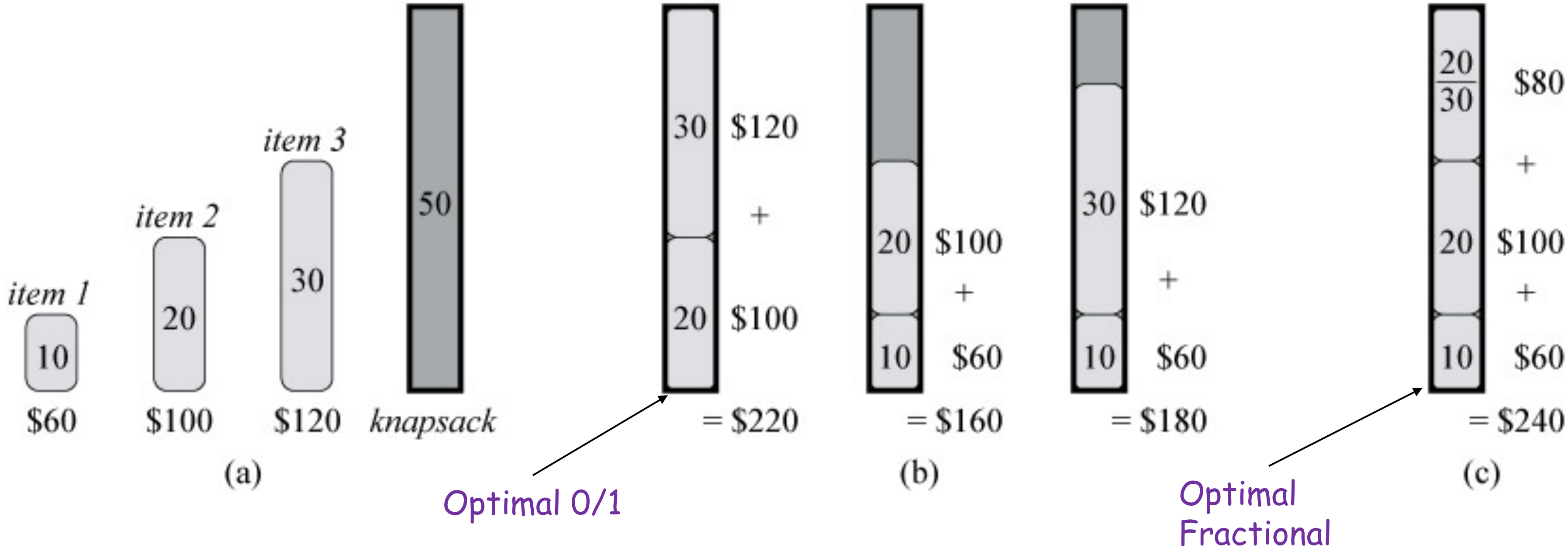


Can repeat this process starting with Greedy and (optimal) O*
Continue repeating this process until O becomes the same as greedy.

Important: Since cost remains the same, final Greedy solution we've created is optimal!

Finished!

The Fractional Knapsack Problem



Input: Set of n items: item i has weight w_i and value v_i, and a knapsack with capacity W.

Goal: Find $0 \le x_1, \dots, x_n \le 1$ such that $\sum_{i=1}^{n} x_i w_i \le W$ and $\sum_{i=1}^{n} x_i v_i$ is maximized.

There are two different versions of this problem:

- The x_i's must be 0 or 1: The 0/1 knapsack problem.
- The x_i's can take fractional values: The fractional knapsack problem

The Greedy Algorithm for Fractional Knapsack

Sort items so that $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$
$w \leftarrow W$
for $i \leftarrow 1$ **to** n
    if $w_i \leq w$ **then**
        $x_i \leftarrow 1$
        $w \leftarrow w - w_i$
    else
        $x_i \leftarrow w/w_i$
        return
return

Idea:

- Sort all items by value-per-pound
- For each item, take as much as possible

Running time: $\Theta(n \log n)$

Note: This algorithm cannot solve the 0/1 version optimally.

Greedy Algorithm: Correctness

Theorem: The greedy algorithm is optimal.

Proof: We assume that $\sum_{i=1}^{n} w_i \geq W$, so knapsack is fully packed. Otherwise the algorithm is trivially optimal.

Let the greedy solution be $\boldsymbol{G = (x_1, x_2, \ldots, x_k, 0, \ldots, 0)}$

- Note: for $i < k,\ x_i = 1$; for $i > k,\ x_i = 0$; $0 \leq x_k \leq 1$.

Consider any optimal solution $\boldsymbol{O = (y_1, y_2, \ldots, y_n)}$

- Note: Since both $\boldsymbol{G}$ and $\boldsymbol{O}$ must fully pack the knapsack,

$$\sum_{i=1}^{k} x_i w_i = W = \sum_{i=1}^{n} y_i w_i$$

Look at the first item i where the two solutions differ.

$$\boldsymbol{G} = x_1\ x_2\ x_3\ \ldots\ x_{i-1}\ x_i\ \ldots\ x_k\ \ldots\ 0\ \ 0$$

$$\boldsymbol{O} = x_1\ x_2\ x_3\ \ldots\ x_{i-1}\ y_i\ \ldots\ y_k\ \ldots\ y_{n-1}\ \ y_n$$

By definition of greedy, $x_i \geq y_i$

- Let $x = x_i - y_i \geq 0$

Greedy Algorithm: Correctness (continued)

We now modify O as follows:

- Set $y_i \leftarrow x_i$ and remove amount of some items in $i+1$ to n of total weight xw_i
- This is always doable because in both O and G, the used total weight of items i to n is the same. $(\sum_{i=1}^{k} x_i\, w_i = W = \sum_{i=1}^{n} y_i\, w_i)$

After the modification:

- The total value of this new O has not decreased, since all the items $i+1$ to n have lesser or equal value-per-pound than item i
- This new O's value can not be greater than before, since O was already an optimal solution,
- => O's value stays the same => O is still an optimal solution
- O's first index that differs from G is now at least $i+1$

Repeating this process will eventually convert O into G

(1st index that differs always increases so the process must stop).

Note that process keeps solution value of O invariant (and optimal).

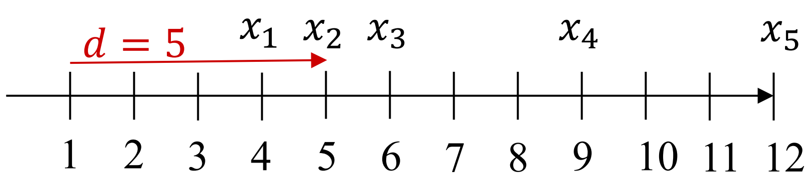
$\Longrightarrow$ at end $G = O \Longrightarrow$ Greedy is optimal

Exercise on Hiking Problem

Suppose you are going on a hiking trip over multiple days. For safety reasons you can only hike during daytime. You can travel at most d kilometers per day, and there are n camping sites along the hiking trail where you can make stops at night.

Assuming the starting point of the trail is at position $x_0 = 0$, the camping sites are at locations $x_1, \dots, x_n$, with $x_1 < x_2 < \cdots < x_n$, where x_n is the final destination.

Design an $O(n)$-time algorithm to find a plan that uses the minimum number of days to finish the trip. For example, if $n = 5, d = 5$, and $(x_1, \dots, x_n) = (4, 5, 6, 9, 12)$, an optimal plan would take 3 days, making stops at camping sites x_1 and x_4. Note that there may be more than one optimal plan (x_2 and x_4 is also optimal); your algorithm just needs to find any one of them. You can assume that $x_{i+1} - x_i \leq d$ for all i (otherwise there is no solution).



Algorithm for Hiking Problem

For each day i, stop at the furthest camping site, i.e. stop at the largest x_j such that x_j minus the start location of day i is at most d.

camping sites = []; $curloc = x_0$;

for $i = 1 \text{ to } n$ do

    if $x_i - curloc > d$ then

        $curloc = x_{i-1}$;

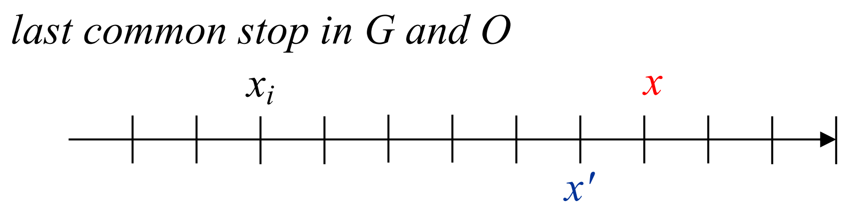
        camping sites.insert(x_{i-1});

return camping sites

In the previous example $x_1 = 4, x_2 = 5, x_3 = 6, x_4 = 9, x_5 = 12$, the algorithm outputs camping sites = $[x_2, x_4]$.

Proof of Correctness (Optimality) for Hiking Problem

- Let G be the solution returned by this greedy algorithm, and let O be an optimal solution.
  - In the previous example G = $[x_2, x_4]$, O = $[x_1, x_4]$
- Consider the first camping site where O is different from G. Suppose the camping site in G is located at x and the one in O is located at x'.
  - In the previous example $x = x_2, x' = x_1$
- By the greedy choice, we must have $x > x'$ because greedy always selects the furthest site within distance d from previous stop. Now replace x' with x in O. The resulting O* must still satisfy the requirement (travel at most d kilometers per day) and contains the same number of stops as O. Thus, it is also optimal.
- Repeatedly applying this transformation will convert O into G. Thus, G is also an optimal solution.

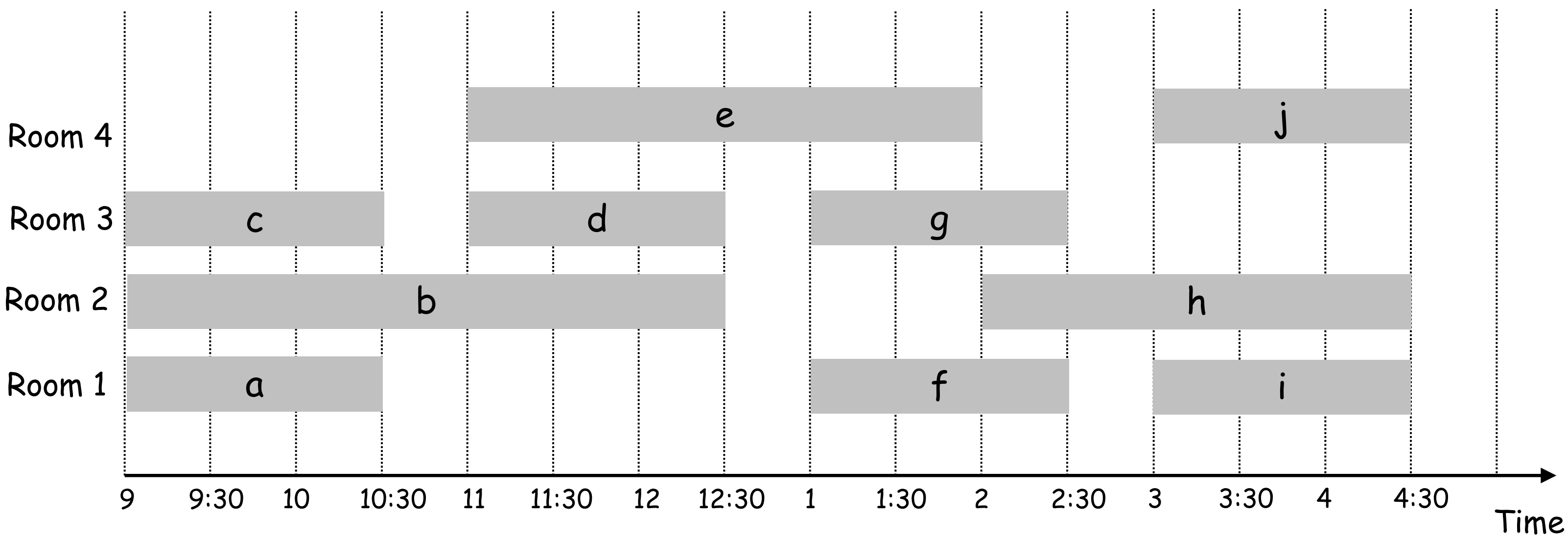


Interval Partitioning

Interval partitioning.

- Lecture j starts at s_j and finishes at f_j.
- Goal: find the 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.

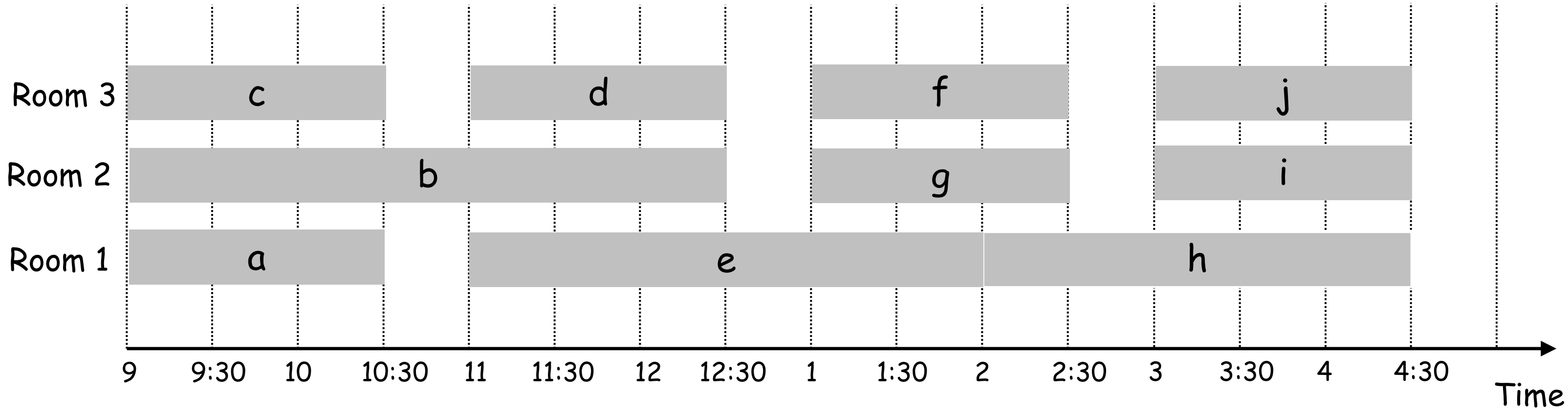


Interval Partitioning

Interval partitioning.

- Lecture j starts at s_j and finishes at f_j.
- Goal: find the 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: Greedy Algorithm

Greedy algorithm. Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
Sort intervals by starting time so that s1 ≤ s2 ≤ ... ≤ sn.

d ← 0  // # classrooms used so far

for j ← 1 to n
    if lecture j is compatible with some classroom k then
        schedule lecture j in classroom k
    else
        allocate a new classroom d + 1
        schedule lecture j in classroom d + 1
        d ← d + 1
```

Greedy! It only opens a new classroom if it is needed.

Need to prove optimality.
More specifically, need to show that processing the lectures **ordered by starting time** implies optimality.

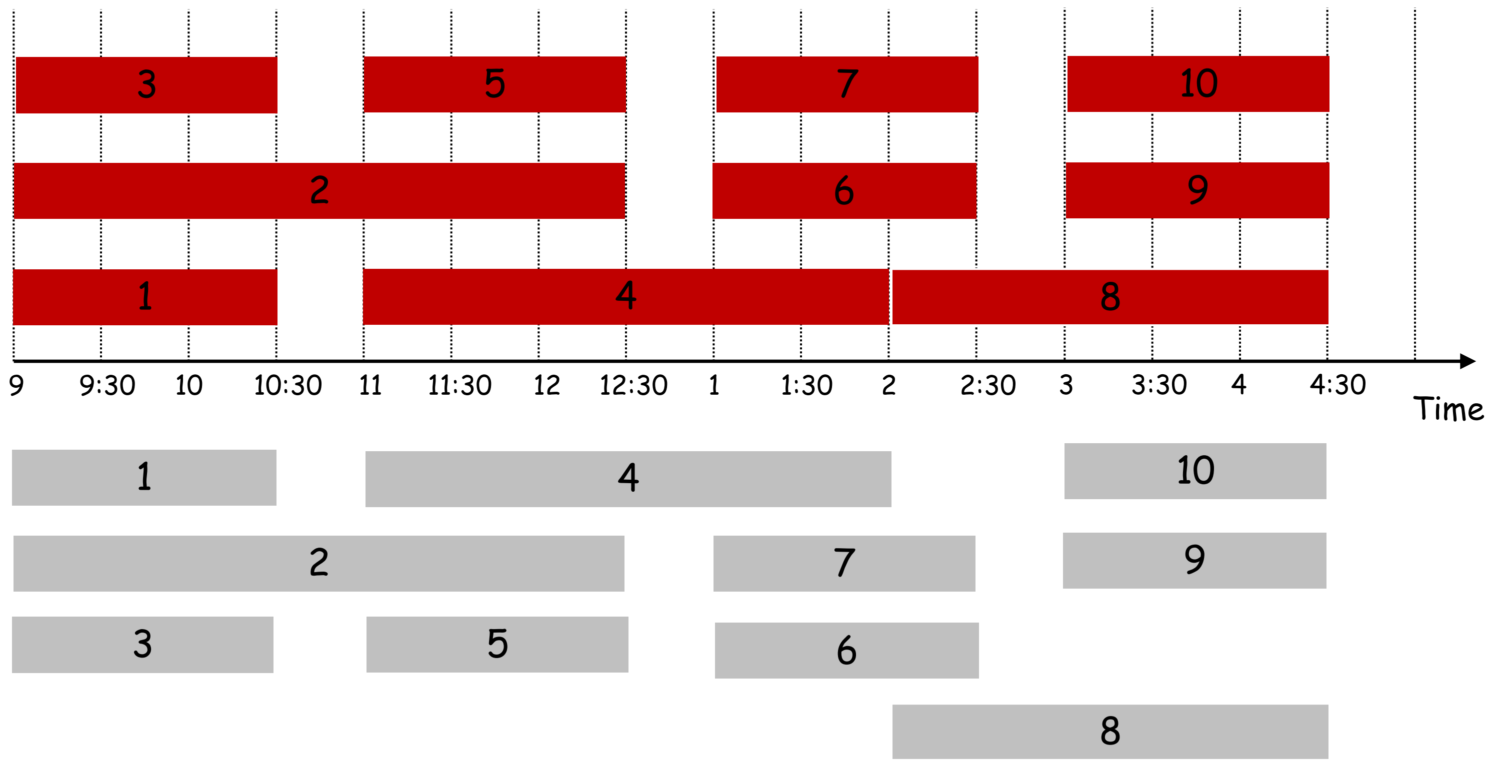
Convince yourself that processing by finishing time does NOT yield optimal solution!

Interval Partitioning

Interval partitioning.

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

ALG: Sort by start time. Insert in order, opening new classroom when needed



Interval Partitioning: Lower Bound on Optimal Solution

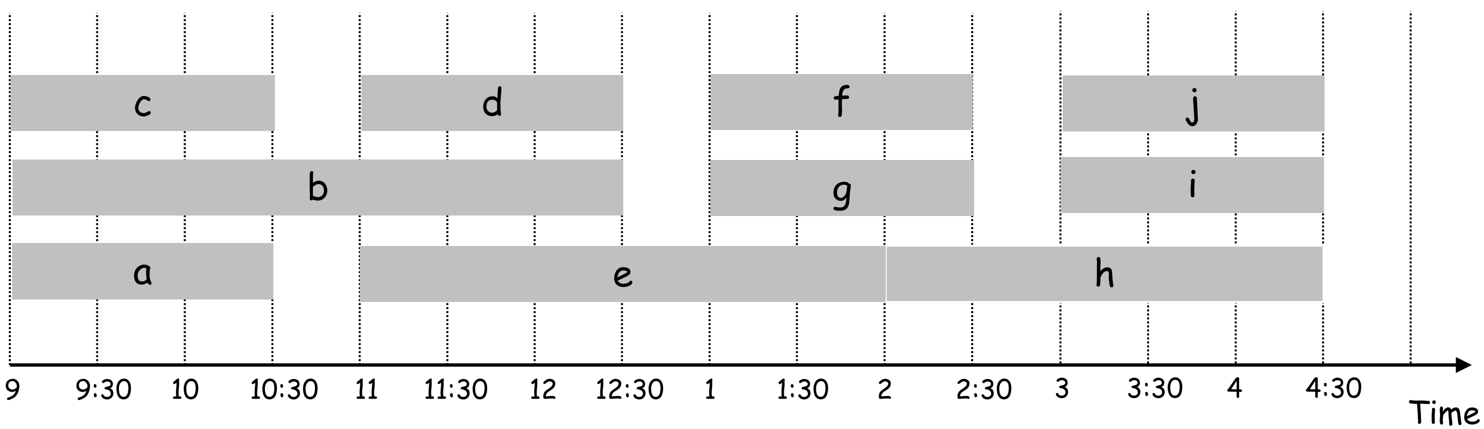
Def. The depth of a set of open intervals is the maximum number that exist at any instant of time.

Intuition: At each second of the day keep track of how many simultaneous classes are being taught at that time. The depth is the maximum number you have seen.

Key observation. Minimum number of classrooms needed $\geq$ depth.

Ex: Depth of schedule below = 3 $\Rightarrow$ this schedule is optimal.

We will show: The # classrooms used by the greedy algorithm = depth.



Interval Partitioning: Correctness

Theorem. Greedy algorithm is optimal.

Pf.

- Let d = number of classrooms opened by greedy algorithm .
- Classroom d is opened because we needed to schedule a lecture, say j, that is incompatible with all $d - 1$ other classrooms.
- Since we sorted by start time, all these incompatibilities are caused by lectures that all start no later than s_j and finish later than s_j.
- $\Rightarrow$ depth $\geq d$.
- Every algorithm uses at least depth classrooms
  $\Rightarrow$ **Greedy is optimal.**

Interval Partitioning: Running Time

```
Sort intervals by starting time so that s₁ ≤ s₂ ≤ ... ≤ sₙ.
d ← 0   // # classrooms used so far
for j ← 1 to n
    if lecture j is compatible with some classroom k then (*)
        schedule lecture j in classroom k   (**)
    else
        allocate a new classroom d + 1
        schedule lecture j in classroom d + 1   (***)
        d ← d + 1
```

- To implement line (*) the algorithm maintains, for each classroom, the finishing time of the last item placed in the classroom.
  It then compares s_j to those finishing times. If $s_j \geq$ one of those finishing times, it places lecture j in the associated classroom
- A Brute-force implementation of line (*) uses $O(n)$ time
  $\Rightarrow O(n^2)$ in total
- Observation: If j is not compatible with the classroom with **the earliest finish time**, then j is not compatible with any other classroom

Interval Partitioning: Running Time

```
Sort intervals by starting time so that s1 ≤ s2 ≤ ... ≤ sn.
d ← 0   // # classrooms used so far
for j ← 1 to n
    if lecture j is compatible with some classroom k then (*)
        schedule lecture j in classroom k   (**)
    else
        allocate a new classroom d + 1
        schedule lecture j in classroom d + 1   (***)
        d ← d + 1
```

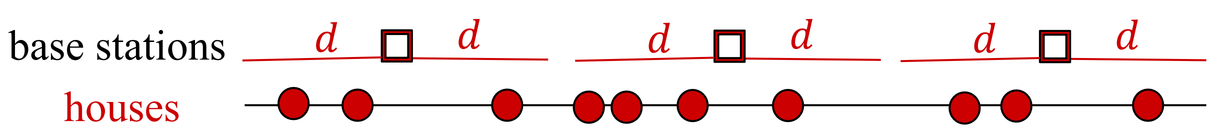
Running time: $\Theta(n \log n)$

- To implement line (*) we can keep the classrooms in a **min heap** using the finishing times of the last class in the room as the key of each classroom
- cost 1: To check whether there is a compatible classroom we find min finishing time f_{min} at the top of the **min heap** and check whether $f_{min} \le s_j$
- $cost\ O(\log n)$: Let k be classroom with f_{min} ($f_{min} \le s_j$). To implement (**), we perform **extract-min** and insert classroom k with new finishing time f_j to min heap
- $O(\log n)$: To implement (***) insert the new classroom j with finishing time f_j to min heap

Exercise on Placement of Base Stations

Consider a long river, along which n houses are scattered. You can think of this river as an axis, and the houses are given by their coordinates on this axis in a sorted order. You must place cell phone base stations at certain points along the river, so that every house is within d kilometers of one of the base stations (d is an input).

Give an $O(n)$-time algorithm that minimizes the number of base stations used, and show that it indeed yields the optimal solution.



Algorithm

Put the first base station at $x + d$ where x is the coordinate of the first house. Remove all the houses that are covered and then repeat if there are still houses not covered.

Correctness: Let G be the solution returned by this greedy algorithm, and let O be an optimal solution.

Consider the first base station where O is different from G. Suppose the base station in G is located at x and the one in O is located at x'. By the greedy choice, we must have $x > x'$ because greedy selects the furthest point possible (at distance *d*) from the first non-covered house.

Now replace x' with x in O . The resulting O* must still cover all houses. Repeatedly applying this transformation will convert O into G. Thus, G is also an optimal solution.