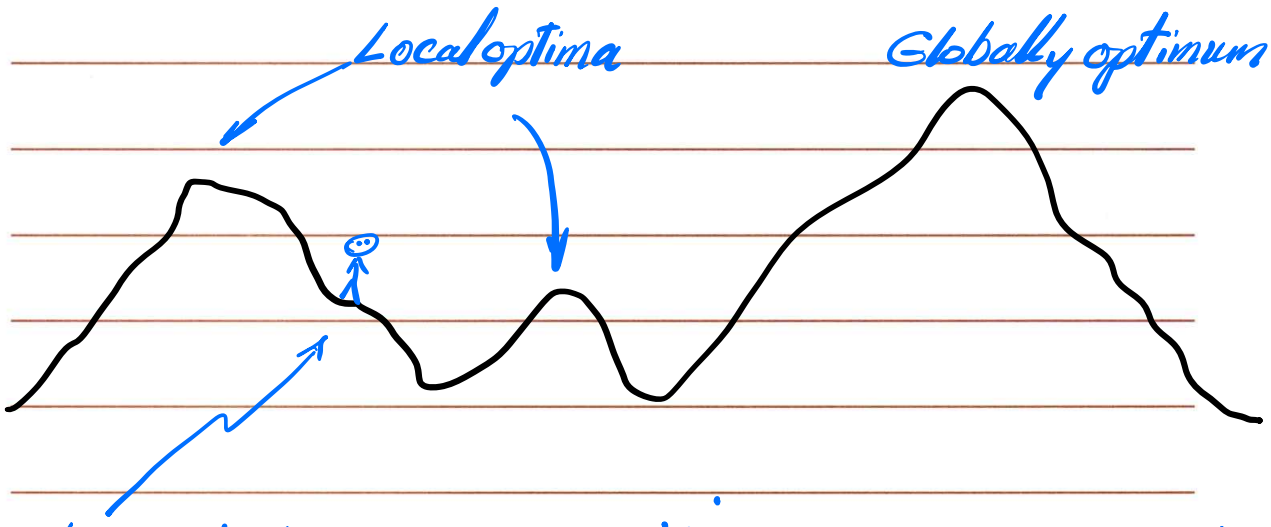
Greedy

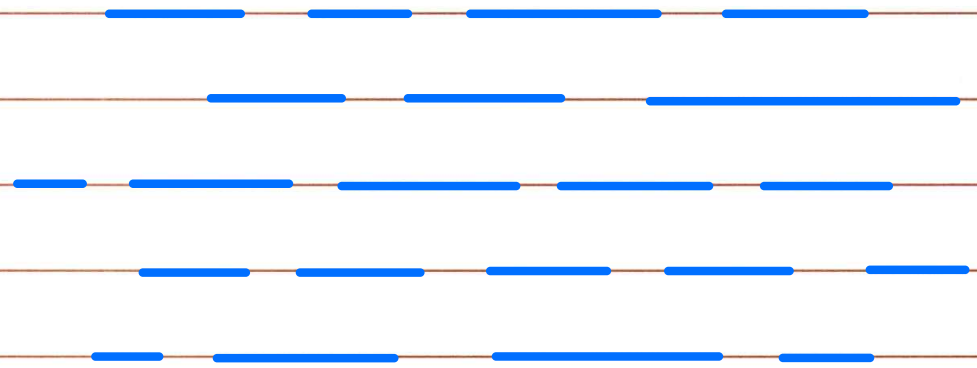


Hiker starting at this position and trying to find the highest peak will find a local optimum solution.

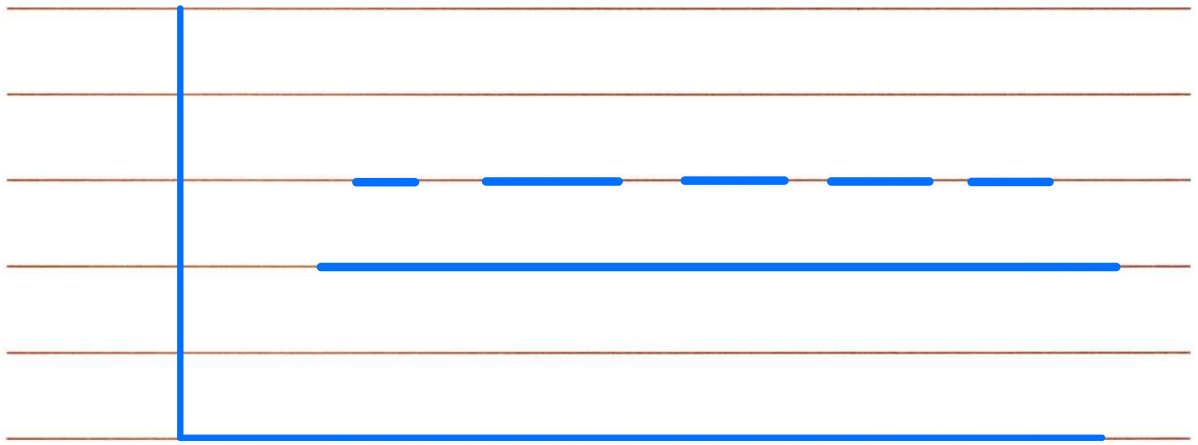
Interval Scheduling Problem

Input: A set of requests $\{1..n\}$, where the i^{th} request starts at $s(i)$ and ends at $f(i)$.

Output: A largest compatible subset of these requests.



Try #1 Earliest Start Time First ✗



Counterexample

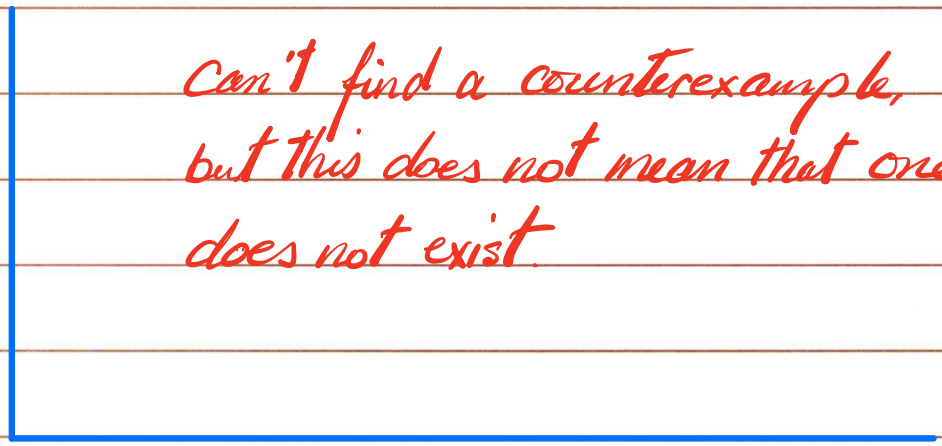
Try #2 Smallest Request First ✗

Counterexample

Try #3 Smallest # of Overlaps First ✗

Counterexample

Try #4 Earliest Finish Time First



can't find a counterexample, but this does not mean that one does not exist.

High Level Solution

Initially R is the complete set of requests and A is empty

While R is not empty

Choose a request $i \in R$ that has the smallest finish time

Add request i to A

Delete all requests from R that are not compatible with i

Endwhile

Return A

Proof of Correctness

1. Prove that A is a compatible set

2. Prove that A is an optimal set

Easy to show #1: Since we always delete all overlapping requests before choosing the next request, we can never end up with overlapping requests in A.

#2: Say A is of size k, and suppose there is an optimal solution O.

Label requests in A: $i_1, i_2, \ldots i_k$

" " " O: $j_1, j_2, \ldots, j_m$

We will first prove that for all indices $1 \leq r \leq k$, we have $f(i_r) \leq f(j_r)$

Proof by mathematical induction

A i_1 ..... i_{r-1} i_r

O j_1 ..... j_{r-1} j_r

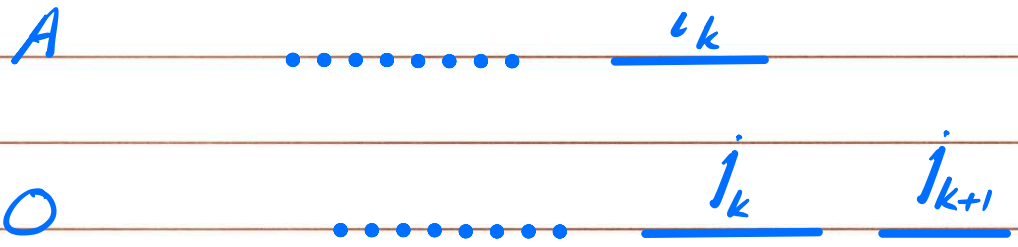
Base Case: i_1 is the request with the earliest finish time, so $f(i_1) \leq f(j_1)$

Inductive hypothesis: We assume that $f(i_{r-1}) \leq f(j_{r-1})$

Inductive Step:
We now need to show that $f(i_r) \leq f(j_r)$.
Since j_{r-1} and j_r are compatible and $f(i_{r-1}) \leq f(j_{r-1})$, then i_{r-1} and j_r are also compatible. So our algorithm can choose either i_r or j_r after i_{r-1}, and since it always picks the one w/ earliest finish time, then i_r must finish no later than j_r.

Having proven that for all indices $1 \leq r \leq k$, $f(i_r) \leq f(j_r)$, we can now prove that $|A| = |O|$

Proof by Contradiction:
We assume that $|O| > |A|$, i.e. the optimal solution contains a request j_{k+1}



Since $f(i_r) \leq f(j_r)$ for all indices $1 \leq r \leq k$, then $f(i_k) \leq f(j_k)$.
Also, since j_{k+1} and j_k are compatible, then i_k and j_{k+1} will also be compatible. So if this request existed our algorithm could have picked it. Contradiction! and

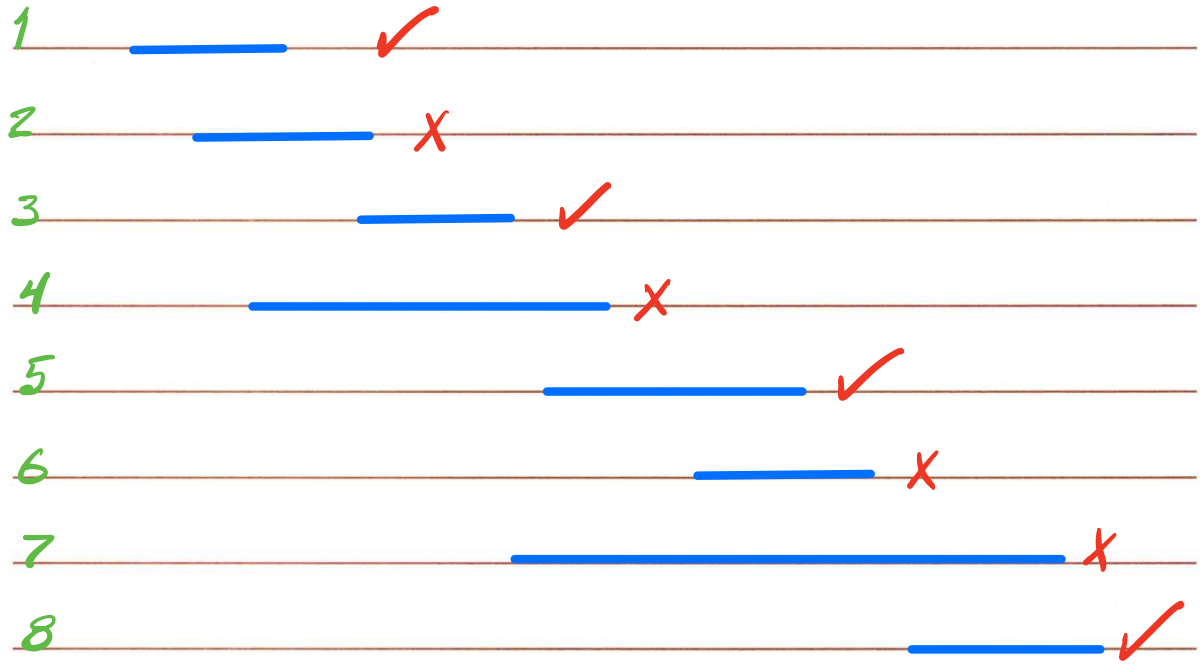
Implementation

$O(n \log n)$ Sort requests in order of finish time and label in this order:

$f(i) \leq f(j)$ where $i < j$

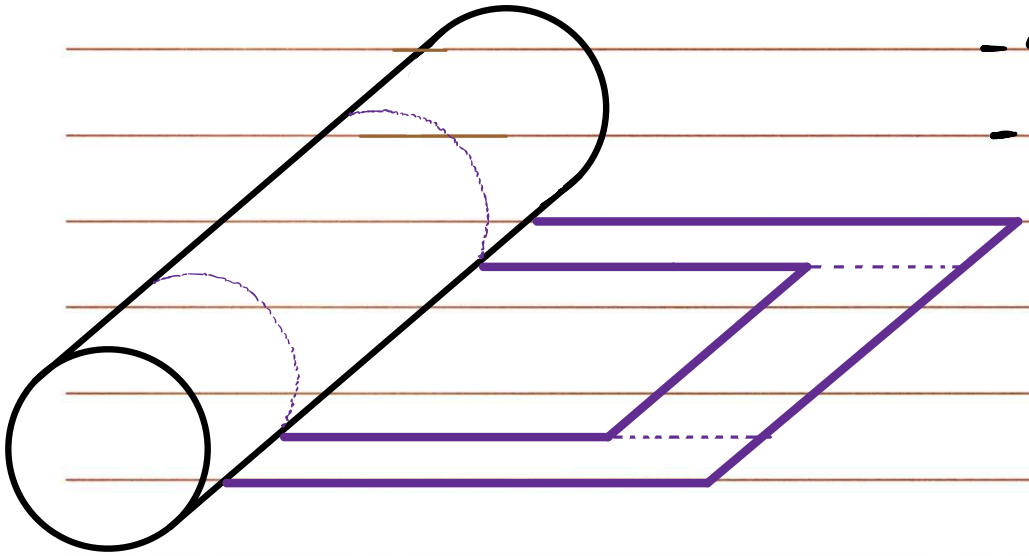
$O(n)$ Select requests in order of increasing $f(i)$, always selecting the first request.
Then iterate through the intervals in this order until reaching the first interval j where $s(j) \geq f(i)$ and then pick j.

Overall Complexity $= O(n \log n)$

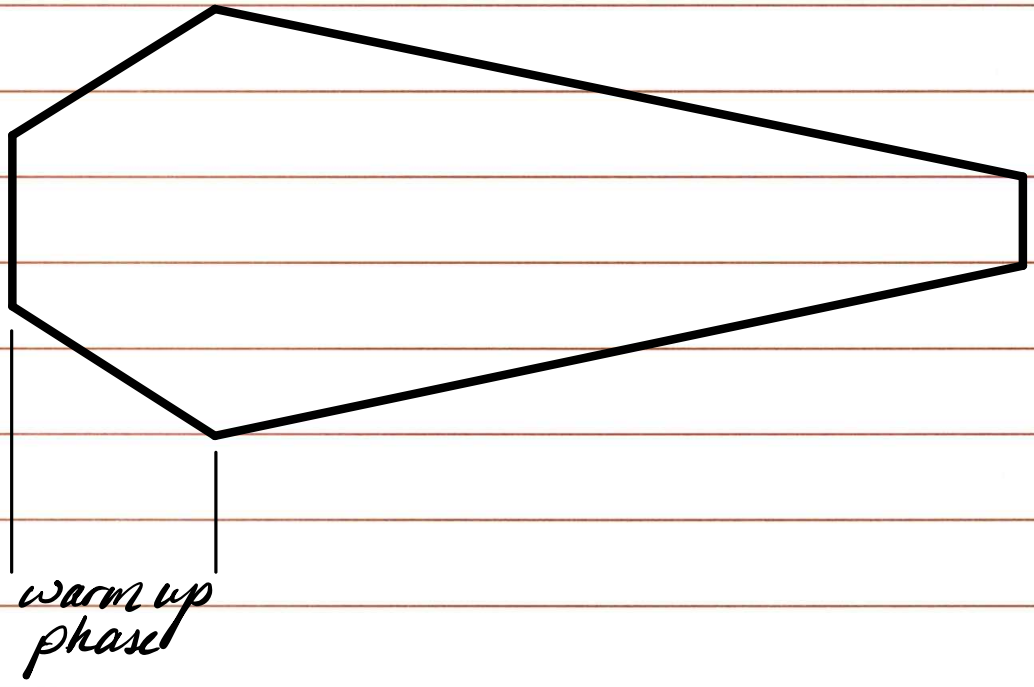


Customer orders specify:

- Quantity
- Width
- Grade



Coffin scheduling



Scheduling to Minimize Lateness

Input: A set of requests $\{1..n\}$, where the i^{th} request has deadline d_i (and time duration t_i.

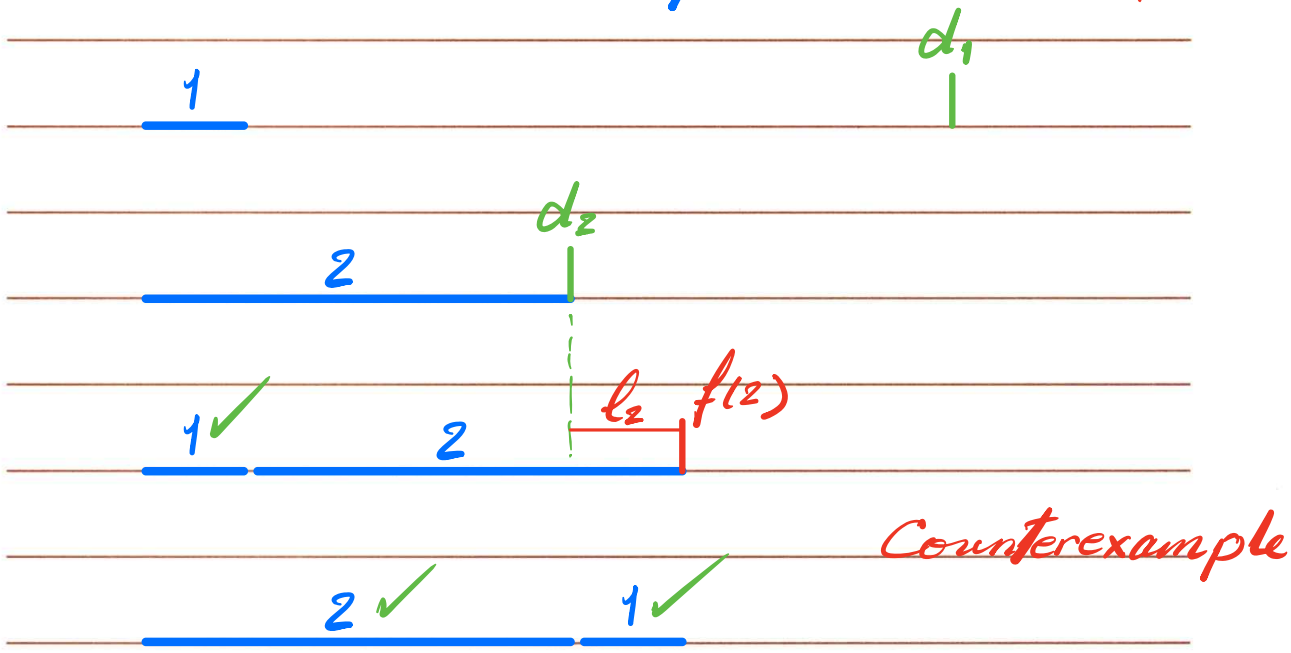
Output: A schedule of the requests that minimizes the maximum lateness. where the lateness for request i is $l_i = f(i) - d_i$

Based on the above objective, which of the following solutions is preferred?

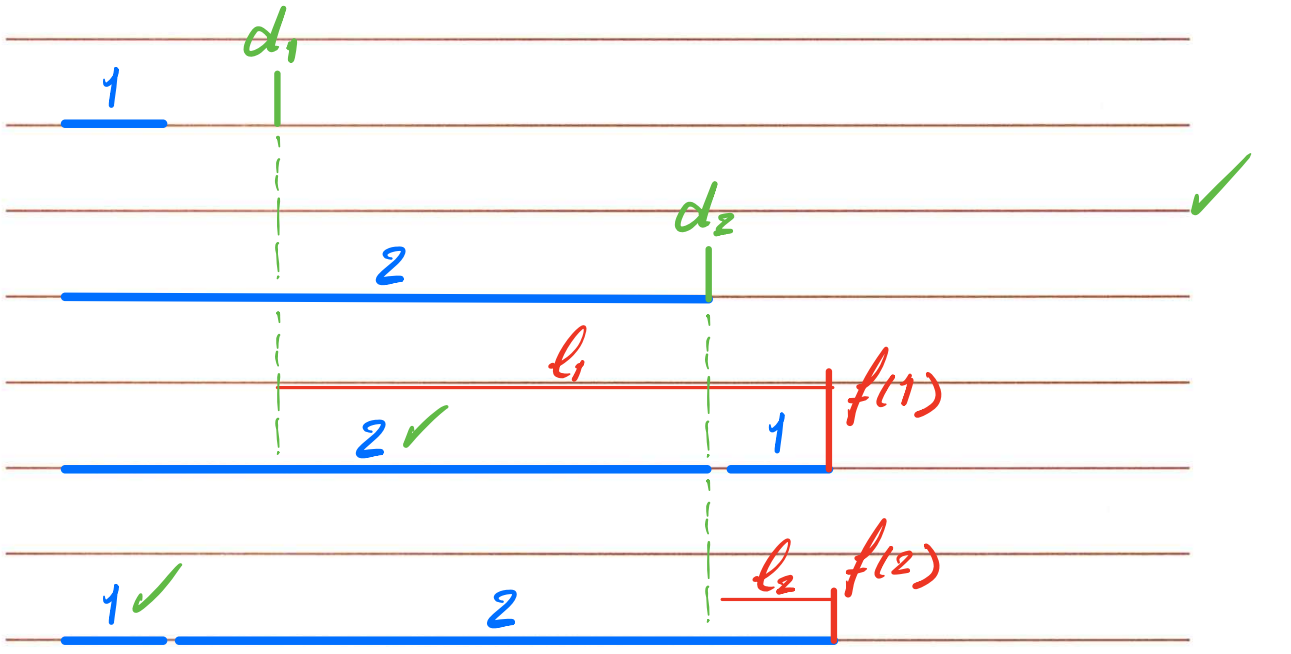
Sol. 1 — job 1 late by 5 hrs; job 2 late by 6 hrs
max. lateness = 6

Sol. 2 — job 1 late by 0 hrs; job 2 late by 7 hrs
max. lateness = 7

Try # 1 Smallest Request First ✗



Try # 1 Shortest Slack First ✗

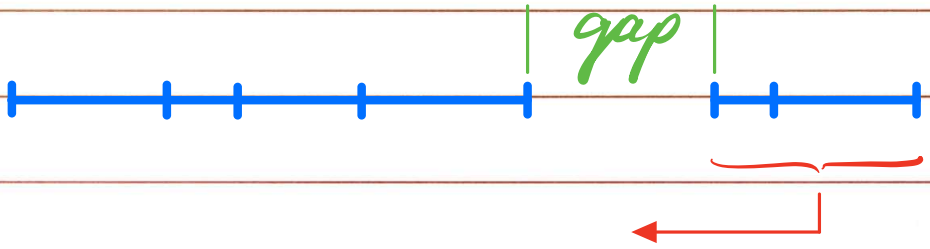


Solution:

Schedule jobs in order of their deadline without any gaps between jobs.

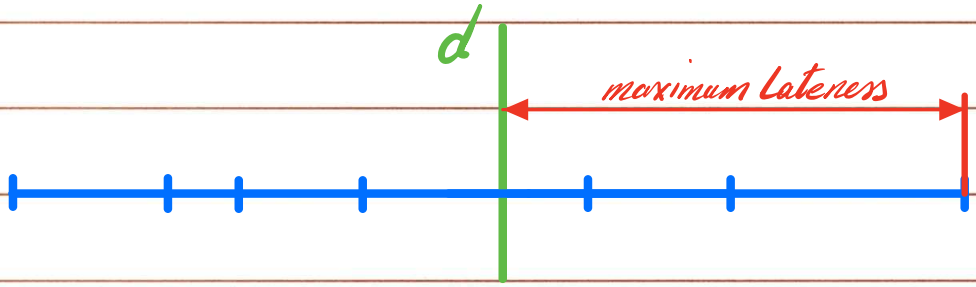
Proof of Correctness:

1- There is an optimal solution with no gaps.



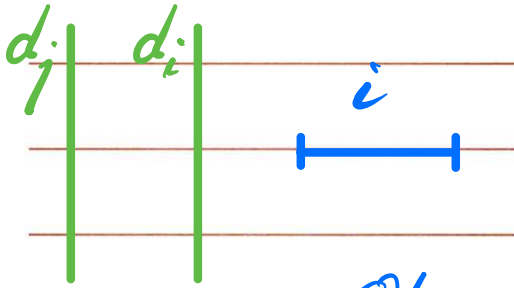
Can always close gaps in an optimal solution to produce an optimal solution with no gaps.

2- Jobs with identical deadlines can be scheduled in any order without affecting maximum lateness.



Maximum lateness will be independent of the order of jobs

3- Def. Schedule A' has an inversion if a job i with deadline d_i is scheduled before job j with an earlier deadline.

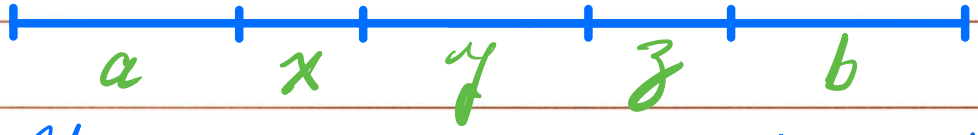


Observation: Our solution contains no inversions.

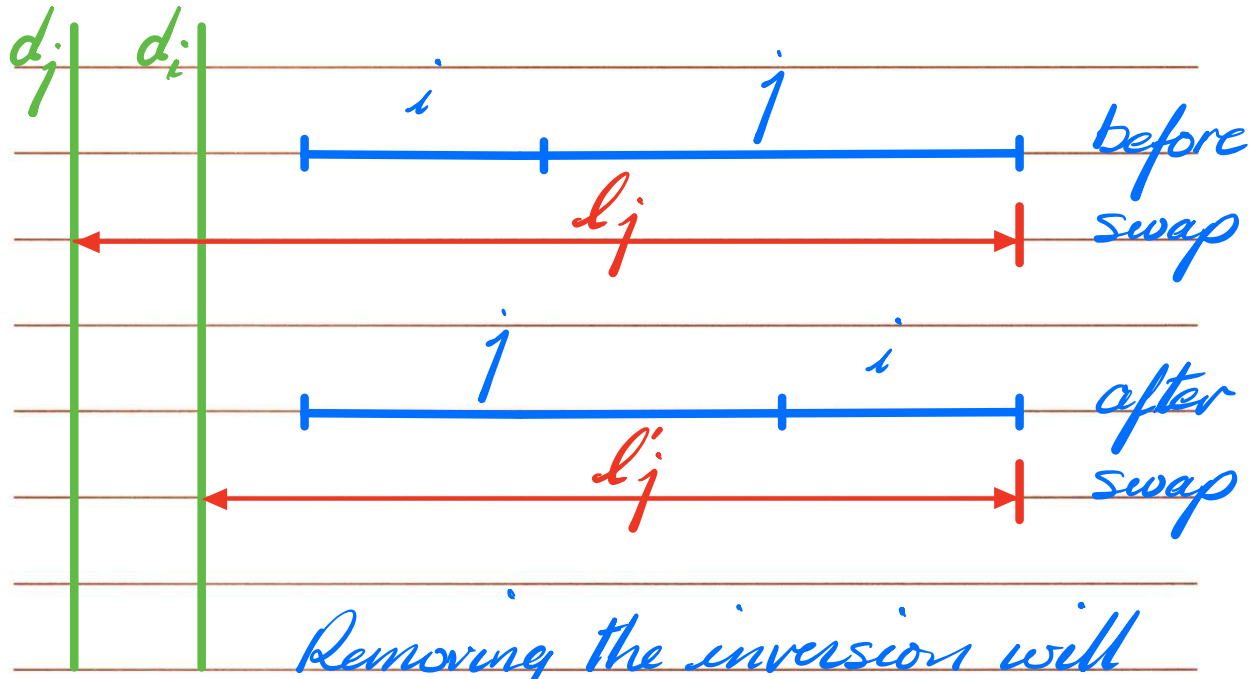
4- All schedules with no inversions and no idle time have the same maximum lateness

If deadlines are distinct, there will only be one such schedule.
And if a subset of jobs have the same deadline, the order of jobs within that subset does not affect their maximum lateness (Fact #2)

5- There is an optimal schedule that has no inversions and no idle time.



If there is an inversion between jobs a and b, we can always find two adjacent jobs between a and b with inversion between them.



Removing the inversion will not increase the maximum lateness

If there is an optimal solution that has inversions, we can eliminate the inversions one by one as shown above until there are no more inversions. This solution will also be optimal.

6_ Proved that there exists an optimal schedule with no inversions and no idle time.

Also proved that all schedules with no inversions and no idle time have the same maximum lateness.

Our greedy algorithm produces one such solution, therefore our solution is also optimal.

Discussion 3

1. Let's consider a long, quiet country road with houses scattered very sparsely along it. We can picture the road as a long line segment, with an eastern endpoint and a western endpoint. Further, let's suppose that, despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.
Give an efficient algorithm that achieves this goal and uses as few base stations as possible. Prove that your algorithm correctly minimizes the number of base stations.

Solution:
Find the first house from the left (western most house) and go four miles to its right and place a base station there. Eliminate all houses covered by that station and repeat the process until all houses are covered.

Proof:
The proof is similar to that for the interval scheduling solution we did in lecture. We first show (using mathematical induction) that our base stations are always to the right of (or not to the left of) the corresponding base stations in any optimal solution. Using this fact, we can then easily show that our solution is optimal (using proof by contradiction).

1- Our base stations are always to the right of (or not to the left of) the corresponding base stations in any optimal solution:
   Base case: we place our first base station as far to the right of the leftmost house as possible. If the base station in the optimal solution is to its right then the first house on the left will not be covered.
   Inductive step: Assume our kth base station is to the right of the kth base station in the optimal solution. We can now show that our k+1st base station is also to the right of the k+1st base station in the optimal solution. To prove this we look at the leftmost house to the right of our kth base station which is not covered by our kth base station. We call this house H. If H is not covered by our kth base station, then it cannot be covered by the kth base station in the optimal solution since our base station is to the right of it. Our k+1st base station is placed 4 miles to the right of H. If the k+1st base station in the optimal solution is further to the right of our k+1st base station, then H is not going to be covered by neither the Kth base station nor the k+1st base station in the optimal solution so the k+1st base station in our solution must also be to the right of the k+1st base station in the optimal solution.
2- Now assume that our solution required n base stations and the optimal solution requires fewer base stations. We now look at our last based station. The reason we needed this base station in our solution was that there was a house to the right of our n-1st base station that was not covered by it. We call this house H. If H is not covered by our n-1st

base station, then H will not be covered by the n-1st base station in the optimal solution either (since our base stations are always to the right of the corresponding base stations in the optimal solution). So, the optimal solution will also need another base station to cover H.

Complexity of our solution will be O(n logn) since we need to sort the houses first from left to right by their x coordinates. The actual positioning of the base stations will require O(n) time.

2. Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of campers. One of his plans is the following mini-triathlon exercise: each contestant must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swims the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins swimming the 20 laps; as soon as he or she is out and starts biking, a third contestant begins swimming, and so on.
Each contestant has a projected *swimming time*, a projected *biking time*, and a projected *running time*. Your friend wants to decide on a *schedule* for the triathlon: an order in which to sequence the starts of the contestants. Let's say that the *completion time* of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathlon, assuming the time projections are accurate.
What is the best order for sending people out, if one wants the whole competition to be over as soon as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible. Prove that your algorithm achieves this.

Solution:
Sort the athletes by decreasing biking + running time.

Proof:
The proof is similar to the proof we did for the scheduling problem to minimize maximum lateness. We first define an inversion as an athlete i with higher (bi+ri) being scheduled after athlete j with lower (bj+rj). We can then show that inversions can be removed without increasing the competition time. We then show that given an optimal solution with inversions, we can remove inversions one by one without affecting the optimality of the solution until the solution turns into our solution.

1- Inversions can be removed without increasing the competition time.
   Remember that if there is an inversion between two items a and b, we can always find two adjacent items somewhere between a and b so that they have an inversion between them. Now we focus on two adjacent athletes (scheduled one after the other) who have an inversion between them, e.g. athlete i with higher (bi+ri) is scheduled after athlete j with lower (bj+rj). Now we show that scheduling athlete i before athlete j is not going push out the completion time of the two athletes i and j. We do this one athlete at a time:
   - By moving athlete i to the left (starting earlier) we cannot increase the completion time of athlete i
   - By moving athlete j to the right (starting after athlete i) we will push out the completion time of athlete j but since the swimming portion is sequential and athlete j gets out of the pool at the same time that athlete i was getting out of the pool before removing the inversion, and since athlete j is faster than athlete i in the biking and

running sections, then the completion time for athlete j will not be worse than the completion time for athlete i prior to removing the inversion.

2- Since we know that removing inversions will not affect the completion time negatively, if we are given an optimal solution that has any inversions in it, we can remove these inversions one by one without affecting the optimality of the solution. When there are no more inversions, this solution will be the same as ours, i.e. athletes sorted in descending order of biking+running time. So our solution is also optimal.

3. The values 1, 2, 3, . . . , 63 are all inserted (in any order) into an initially empty min-heap. What is the smallest number that could be a leaf node?

Solution: since there are $2^6 - 1$ elements in the heap, the heap will consist of a full binary tree with 6 levels. So, the smallest possible value at a leaf node would be 6.

4. Given an unsorted array of size *n*. Devise a heap-based algorithm that finds the *k*-th largest element in the array. What is its runtime complexity?

Solution: build a max heap of size n in O(n) time. Do k extract_max operations to find the kth largest element. The overall complexity will be O(n + k logn)

5. Suppose you have two min-heaps, A and B, with a total of *n* elements between them. You want to discover if A and B have a key in common. Give a solution to this problem that takes time O(n log n) and explain why it is correct. Give a brief explanation for why your algorithm has the required running time. For this problem, do not use the fact that heaps are implemented as arrays; treat them as abstract data types.

Solution:
We can compare the element at the top of A (TA) with the element at the top of B (TB).

If TA < TB	Extract_Min (A)
Elseif TA > TB	Extract_Min (B)
Else	A mut be equal to B. We have found a common key

We repeat the above until we either find a common key or one of the heaps is empty.

The complexity of the solution is O(n log n) since at each iteration we do two extract_min operations at a cost of log n and there could be at most O(n) iterations.