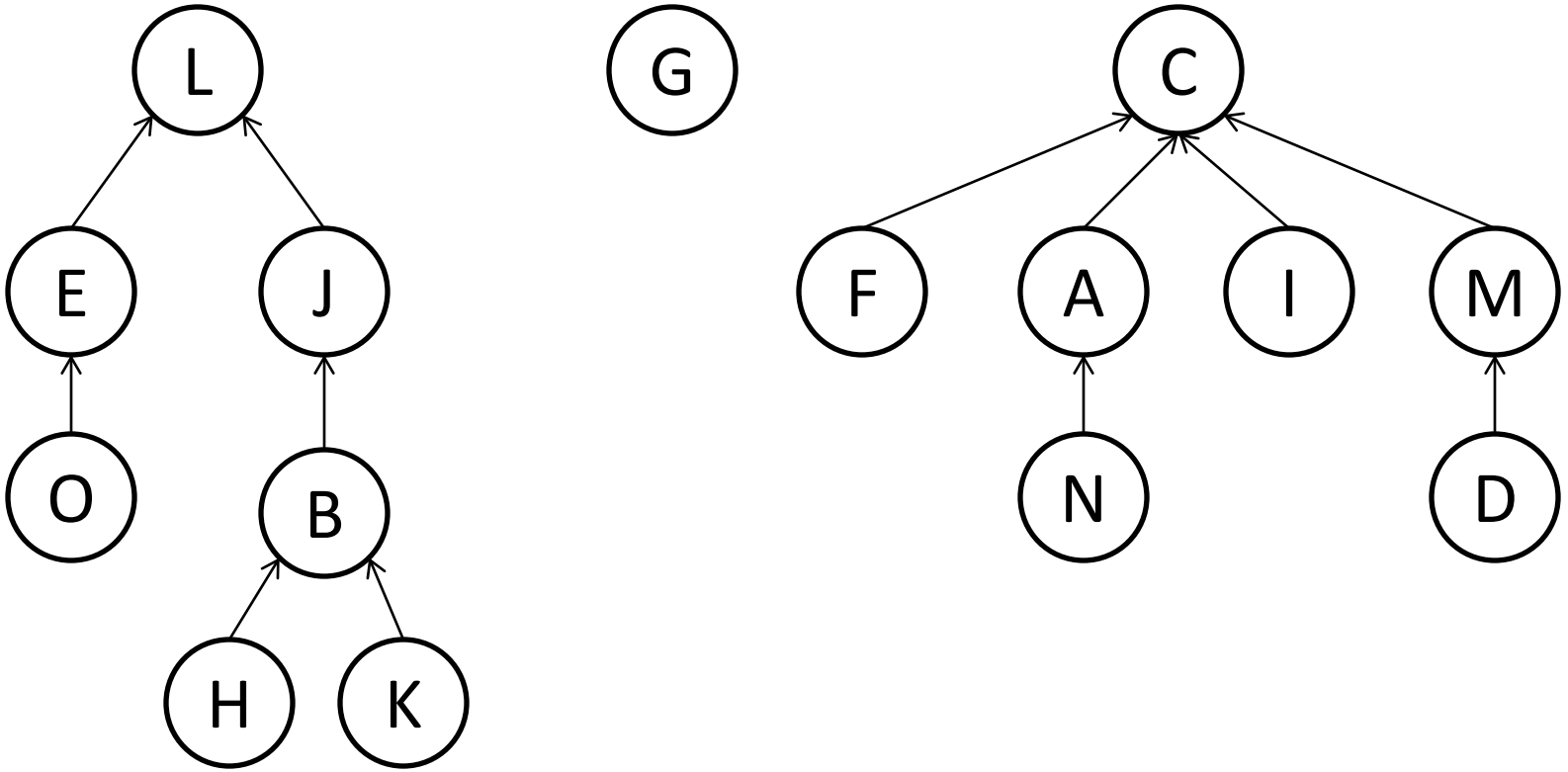
CS 161: Design and Analysis of Algorithms

Greedy Algorithms 3: Minimum Spanning Trees/Scheduling

- Disjoint Sets, continued
- Analysis of Kruskal's Algorithm
- Interval Scheduling

Disjoint Sets, Continued

- Each set represented as directed tree
- Set identified by root of tree



Disjoint Sets, Continued

- makeset(x) = { O(1)
  - Set p(x) = x
  - Set rank(x) = 0
  }
- find(x) = { O(h)
  - Let r = x
  - While p(r) ≠ r, r = p(r)
  - Return r
  }

Disjoint Sets, Continued

- union(x,y) = { $O(h_1 + h_2)$
  - Let x’ = p(x), y’ = p(y)
  - If rank(x’) > rank(y’):
    - p(y’) = x’
  - Else
    - p(x’) = y’
    - If rank(x’) = rank(y’): rank(y’) = rank(y’) + 1

Disjoint Sets, Continued

- rank(x) < rank(p(x))
- Any root of rank k has at least 2^k descendants
- There are at most $n/2^k$ nodes of rank k
- Maximum rank is at most log n

Optimization: Path Compression

- When we call find(x), we will traverse the ancestors of x until we find the root r
- Regardless of union operations, r will always be an ancestor of x
- Can shortcut path to root and set p(x) = r

Optimization: Path Compression

- find(x) = {
  - Let r = x
  - Let s be a stack
  - While p(r) ≠ r:
    - s.push(r)
    - R = p(r)
  - While s isn't empty: p(s.pop()) = r
  - Return r

  }

Optimization: Path Compression

- find(x) = {
  - If $x \neq p(x)$, set $p(x) = find(p(x))$
  - Return p(x)
  }

Running Time?

- Running time = depth of x in tree
- Still O(h), might still be O(log n)
- However, once we call find(x), subsequent calls to find(x) are constant time
- Using amortized analysis, almost linear time on average

Amortized Analysis

- Consider any sequence of n calls to find (might be part of calls to union)
- Running time bounded by number times we follow parent pointers

Amortized analysis

- Path compression does not affect ranks
- If we update a parent pointer, the new parent must have higher rank than the old parent
- If x has a parent, its rank will never be changed again.
- Maximum rank: since n calls to find, largest set has at most n values, so max rank = log n

Amortized Analysis

- Break possible ranks into intervals $\{k+1,\ldots,2^k\}$:
  - Interval 0: {1}
  - Interval 1: {2}
  - Interval 2: {3,4}
  - Interval 3: {5,6,7,…,16}
  - Interval 4: $\{17,\ldots,2^{16} = 65536\}$
  - Interval 5: $\{65537,\ldots,2^{65536}\}$
  - …

Amortized Analysis

- How many intervals needed?
- Maximum rank = log n
- In practice: n never larger than 2^{65536}, so log n never larger than 65536: only need 5 intervals
- $\log^*(k)$ = number of times apply log to k to get something at most 1
- In theory: need $\log^*(n)$ intervals

Amortized Analysis

- How many nodes in {k+1,…,2^k}?
  - At most n/2^i nodes of rank i, so sum from i=k+1 to i=2^k

$$\sum_{i=k+1}^{2^k} \frac{n}{2^i} < \sum_{i=k+1}^{\infty} \frac{n}{2^i} = \frac{n}{2^k} \sum_{j=1}^{\infty} \frac{1}{2^j} = \frac{n}{2^k}$$

- Number of times we update parent pointer, where old parent pointer in same range as x?
  - At most 2^k for each node in {k+1,…,2^k}
  - At most n updates for each range
  - At most n $\log^*(n)$

Amortized Analysis

- When we call find(x), we repeatedly follow parent pointers
- For all except the last pointer, we update the parent pointer
- At most $\log^*(n)$ of the nodes we visit have their parent pointers in the next interval
- Total number of parent pointers followed:
  - Parent pointers not updated: n
  - Updates where parent in next level: $n \log^*(n)$
  - Updates where parent in same level: $n \log^*(n)$

Amortized Analysis

- Total number of parent pointers followed: $n+2n \log^*(n) \leq 3n \log^*(n) = O(n \log^*(n))$
- Time per call to find: $O(\log^*(n))$ amortized
- Therefore, find and union take amortized $O(\log^*(n))$, essentially constant

Kruskal's Algorithm

- Repeatedly add lightest edge that does not form a cycle
- How to find lightest edge? Sort them initially
  - Heap/self-balancing BST O(|E| log |E|)
  - If edge weights are small, can sort in O(|E|)
- Check if adding causes a cycle? Keep track of separate components with Disjoint Sets

Kruskal's Algorithm

- For all v, makeset(v)
- Sort E by increasing weight
- For all edges (u,v) in E in sorted order:
  - If find(u) ≠ find(v):
    - Add (u,v) to MST
    - union(u,v)

Kruskal's Algorithm

- Running Time?
- Time for sort = $O(|E| \log |E|)$ or $O(|E|)$
- Call makeset $|V|$ times, find twice per edge, union at most once per edge
- $|E| \geq |V|-1$, so $|V| = O(|E|)$
- Time after sorting: $O(|E| \log^* |E|)$

Kruskal's Algorithm

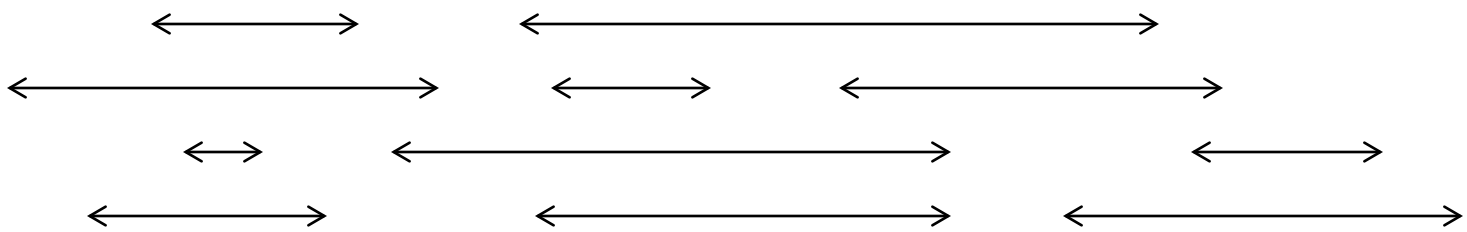
- In general, sorting is bottleneck
- If we can sort in linear time or are given edges in sorted order, $O(|E| \log^* |E|)$
- Otherwise, $O(|E| \log |E|)$
  - Same as Prim's with binary heap

More on $\log^*(n)$

- Called iterated logarithm
- Extremely slow growing
  - At most 5 for all values seem in practice
- Not the slowest:
  - Inverse Ackermann function $\alpha(n)$ asymptotically slower
- Arises occasionally in algorithmic analysis
  - Multiply n-bit integers in time $O(n \log n \, c^{\log^* n})$

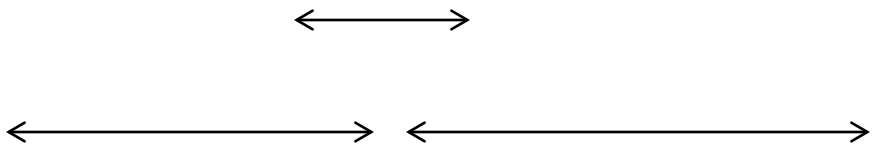
Interval Scheduling

- Suppose there are n jobs
  - Each job i must be worked on and completed in the specified time interval [s(i),f(i)]
  - Can only work on 1 job at a time
  - How do we maximize the number of jobs completed?



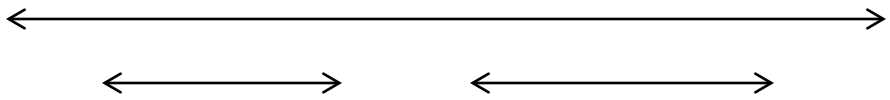
Greedy Attempt 1: Shortest Intervals

- Since we want the most intervals possible, might make sense to repeatedly pick the smallest interval that does not cause a conflict



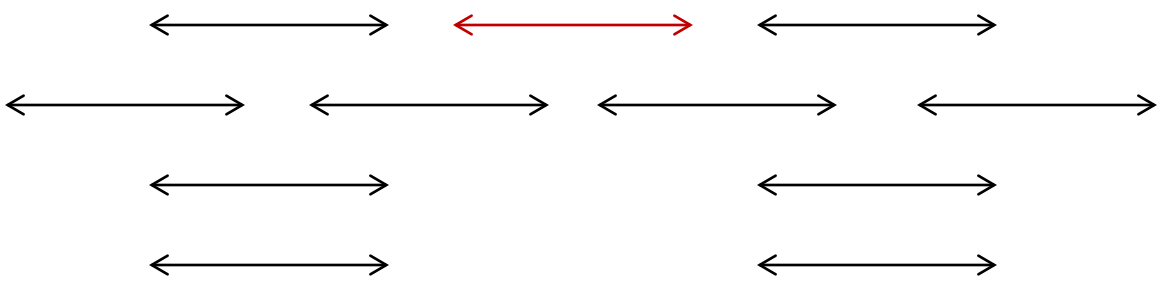
Greedy Attempt 2: Earliest Start

- Since getting started sooner means we can work for longer, might make sense to pick the interval with the earliest start time



Greedy Attempt 3: Fewest Conflict

- Whenever we do one job, we are preventing ourselves from completing any job that overlaps. Therefore, it might make sense to pick the job with the fewest conflicts



Greedy Attempt 4: Earliest Finish

- If we finish the first task as early as possible, will have the most time left for other tasks.
- Turns out to be correct greedy algorithm

Greedy Attempt 4: Earliest Finish

- Sort tasks by their end time f(i)
- Let t = 0
- For each task i in increasing order of f(i)
  - If t > s(i), discard i
  - Otherwise, do task i, and set t = f(i)

Greedy Attempt 4: Earliest Finish

- Running time?
  - Sort in O(n log n)
  - O(1) extra time per task
  - Total time: O(n log n) in general
  - O(n) if we can sort in linear time

Greedy Attempt 4: Earliest Finish

- Optimal?
  - Let $i_1, \ldots i_k$ be the intervals used in the greedy solution G, in order of time completed ($f(i_r)<f(i_{r+1})$)
  - Let $j_1, \ldots j_m$ be intervals in some optimal solution O, in order of time completed
    - Clearly $m \leq k$. We want to show that $m = k$
  - Claim: For each r = 0, …, k, there is an optimal solution O_r where the first r intervals are $i_1, \ldots i_r$

Greedy Attempt 4: Earliest Finish

- Claim: For each r = 0, …, k, there is an optimal solution O_r where the first r intervals are i_1, … i_r
  - Proof: Clearly true for r = 0
  - Given O_{r-1}, construct O_r by changing interval j_r
  - Interval $j_{r-1} = i_{r-1}$, so ends at time $f(i_{r-1})$
  - Interval i_r has earliest end point among intervals starting after $f(i_{r-1})$
  - Can safely replace $j_r = i_r$, arriving at O_r

Greedy Attempt 4: Earliest Finish

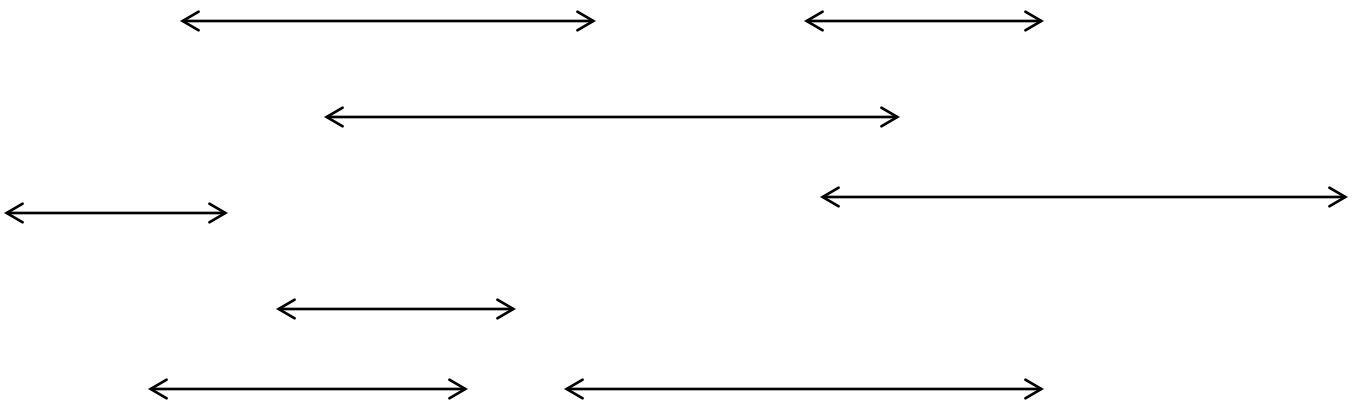
- Claim: Greedy is optimal
  - Proof: Consider optimal solution O_k
  - First k i_1, … i_k, so fist k intervals are greedy solution
  - If Greedy not optimal, must be j_{k+1} occurring after i_k
  - But then after picking i_k, greedy would have had at least 1 interval to choose from, would not have terminated

Related Problem: Interval Partitioning

- Have n tasks, want to complete them all using as few people as possible
- Formally: have n intervals, want to partition them into k subsets so that the intervals in each subset do not overlap, and k is minimized

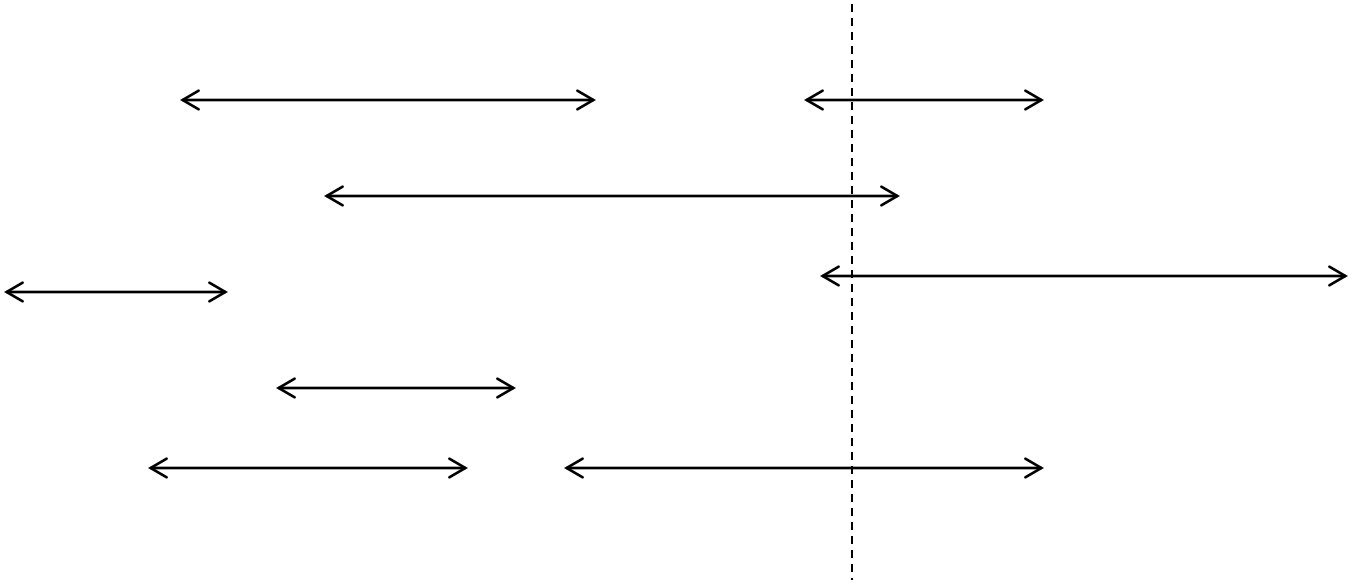
Interval Partitioning

- How many partitions needed?



Interval Partitioning

- How many partitions needed?



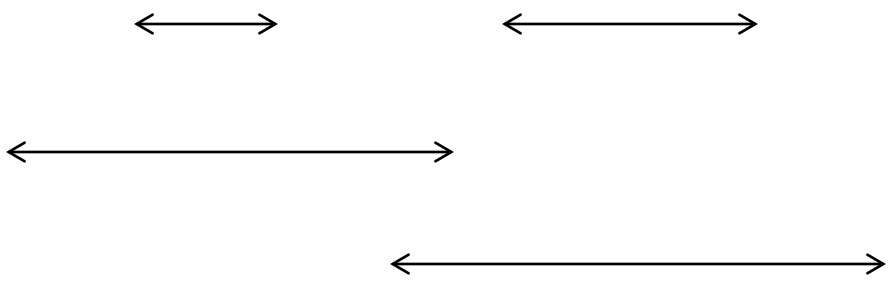
- Clearly at least 4

Interval Partitioning

- Let S be the set of tasks
- Let depth(S) be the maximum number of tasks occurring simultaneously
- Clearly, the number of partitions must be at least depth(S)
- Can we find a partition into depth(S) sets?

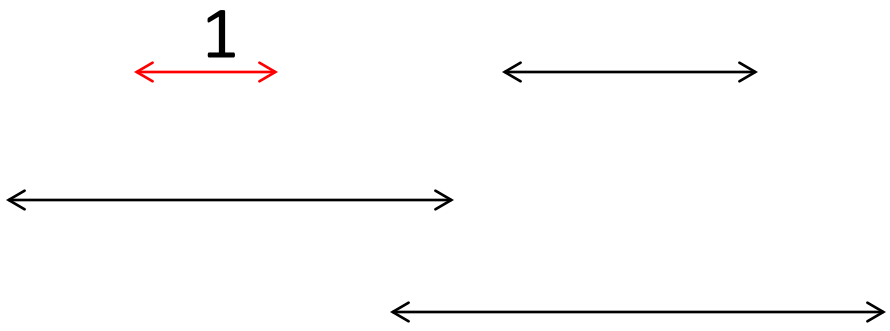
Interval Partitioning

- Idea: take earliest interval i (by start time or finish time?), and add to any set that has no intervals overlapping i
- By finish time?



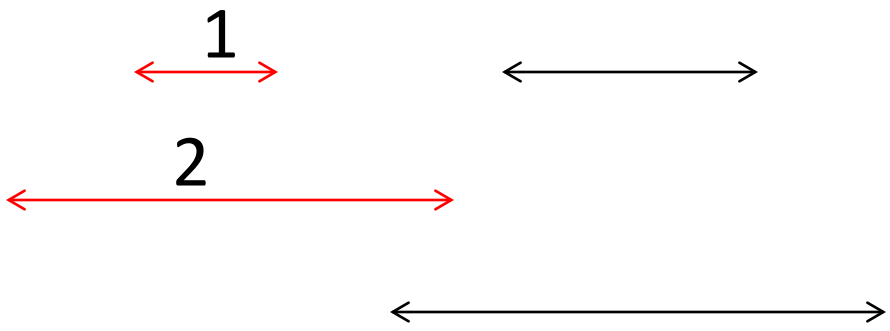
Interval Partitioning

- Idea: take earliest interval i (by start time or finish time?), and add to any set that has no intervals overlapping i
- By finish time?



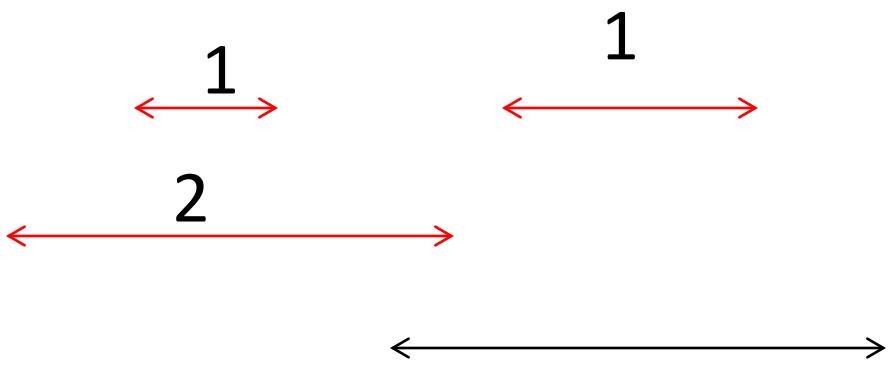
Interval Partitioning

- Idea: take earliest interval i (by start time or finish time?), and add to any set that has no intervals overlapping i
- By finish time?



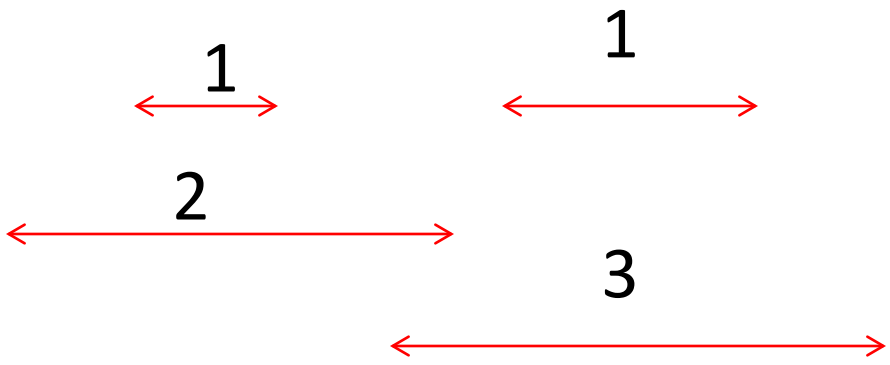
Interval Partitioning

- Idea: take earliest interval i (by start time or finish time?), and add to any set that has no intervals overlapping i
- By finish time?



Interval Partitioning

- Idea: take earliest interval i (by start time or finish time?), and add to any set that has no intervals overlapping i
- By finish time?



- Greedy gives 3, but depth = 2

Interval Partitioning

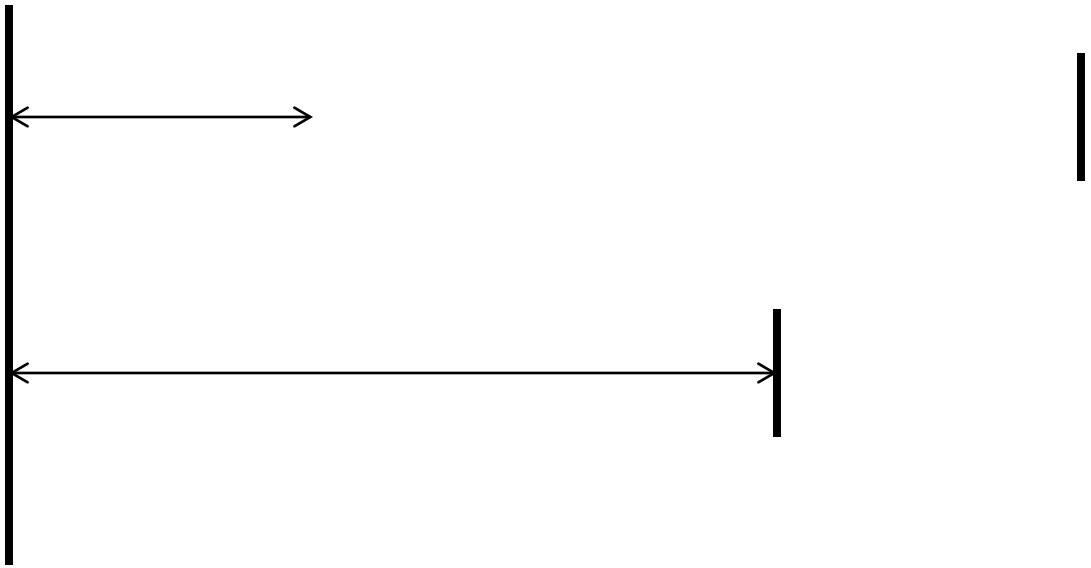
- Take earliest interval i (by start time), and add to any set that has no intervals overlapping i
- Suppose number of sets is > depth
  - When we go to add interval i, all depth(S) sets have an interval overlapping i
  - i must have higher s(i) than any of these intervals
  - But then the number of intervals overlapping at time s(i) is depth(S) + 1, a contradiction

Scheduling to Minimize Lateness

- Back to having one person to work on tasks
- Have n jobs, each with processing time t_i, and deadline d_i
- Devise a schedule to complete the jobs, minimizing maximum lateness:
  - Define the lateness L_i of task i as $\max(0, f_i - d_i)$
  - Want to minimize max of lateness values

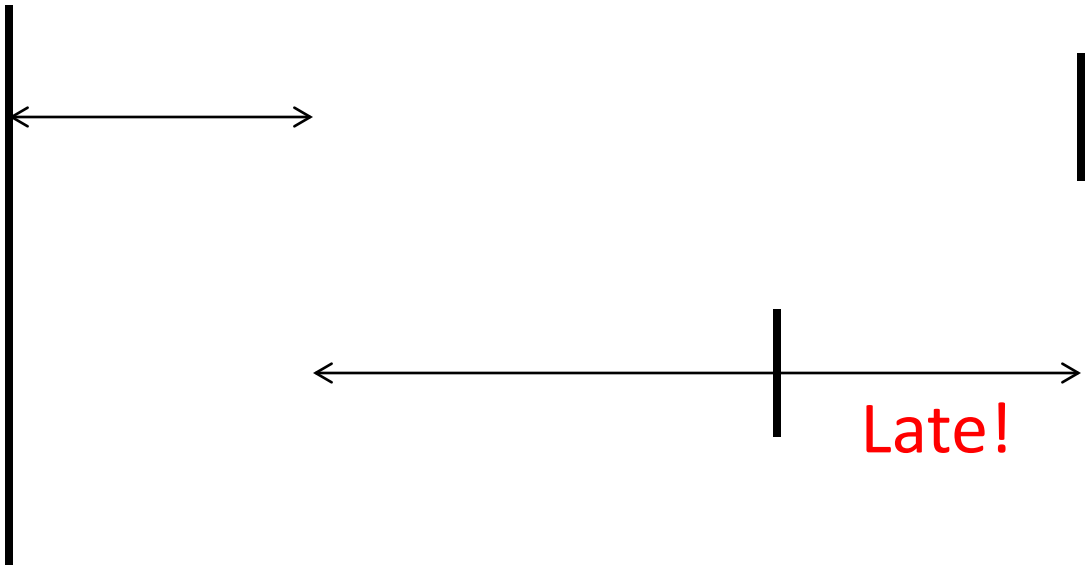
Minimizing Lateness

- How should we prioritize tasks?
  - By processing time?



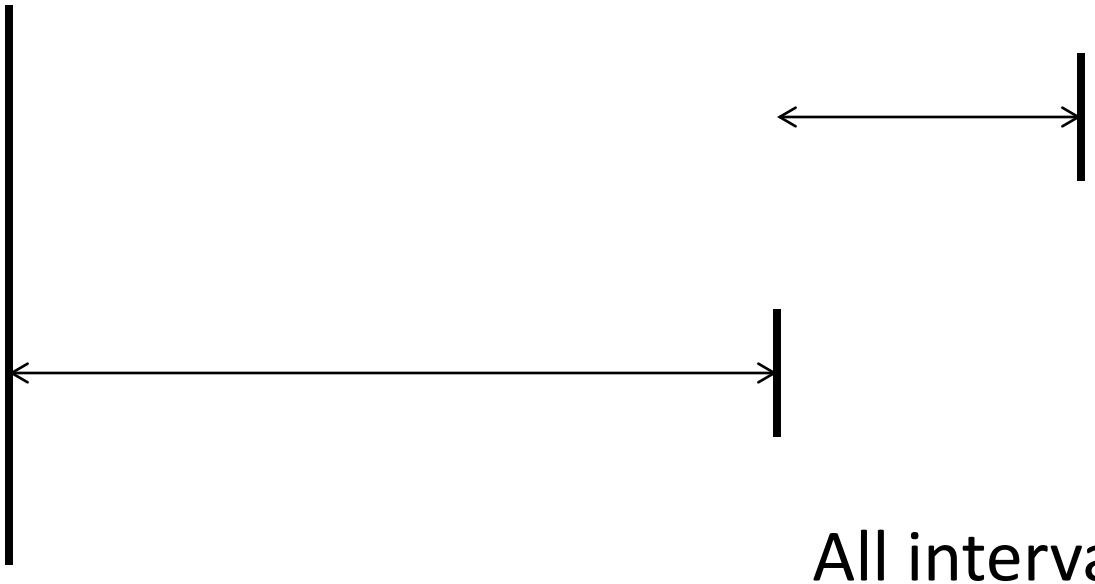
Minimizing Lateness

- How should we prioritize tasks?
  - By processing time?
    - Greedy:



Minimizing Lateness

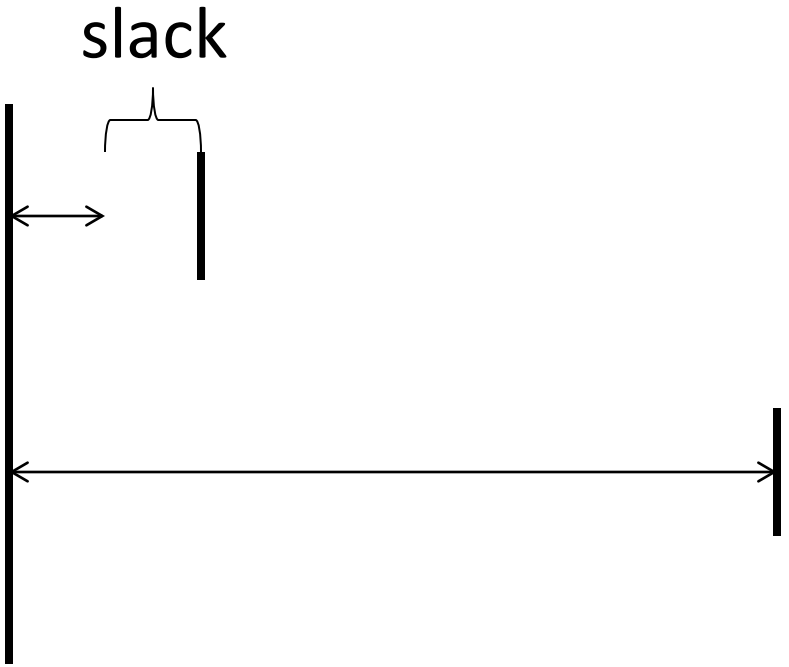
- How should we prioritize tasks?
  - By processing time?
    - Optimal:



All intervals on time

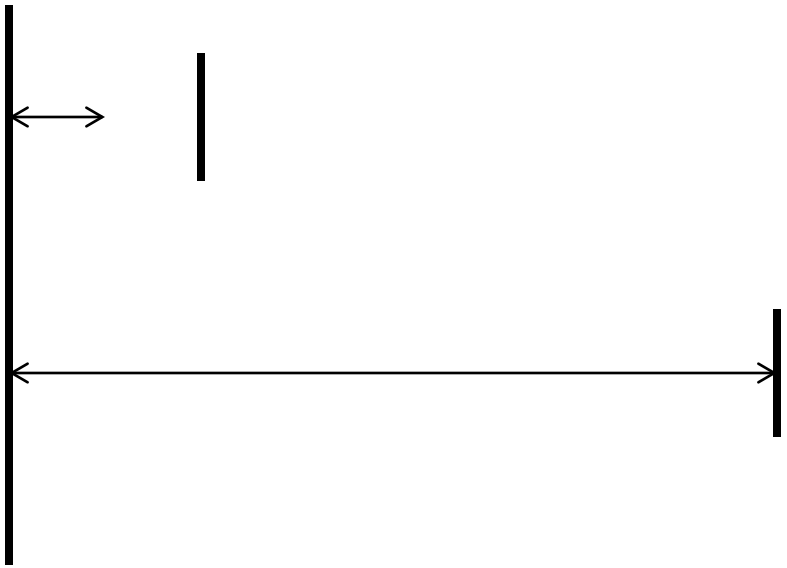
Minimizing Lateness

- How should we prioritize tasks?
  - By slack time $d_i - t_i$?



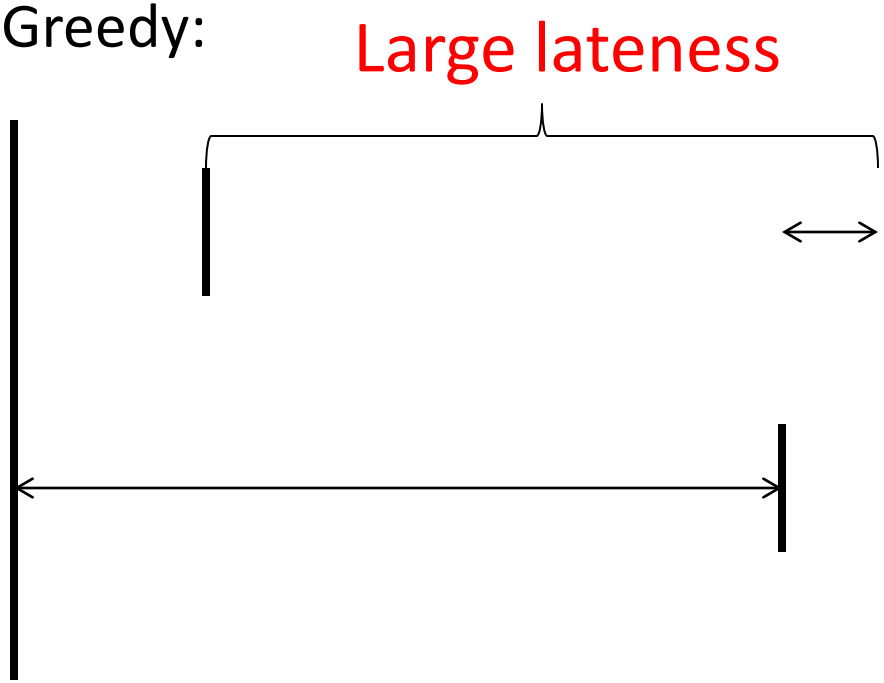
Minimizing Lateness

- How should we prioritize tasks?
  - By slack time $d_i - t_i$?



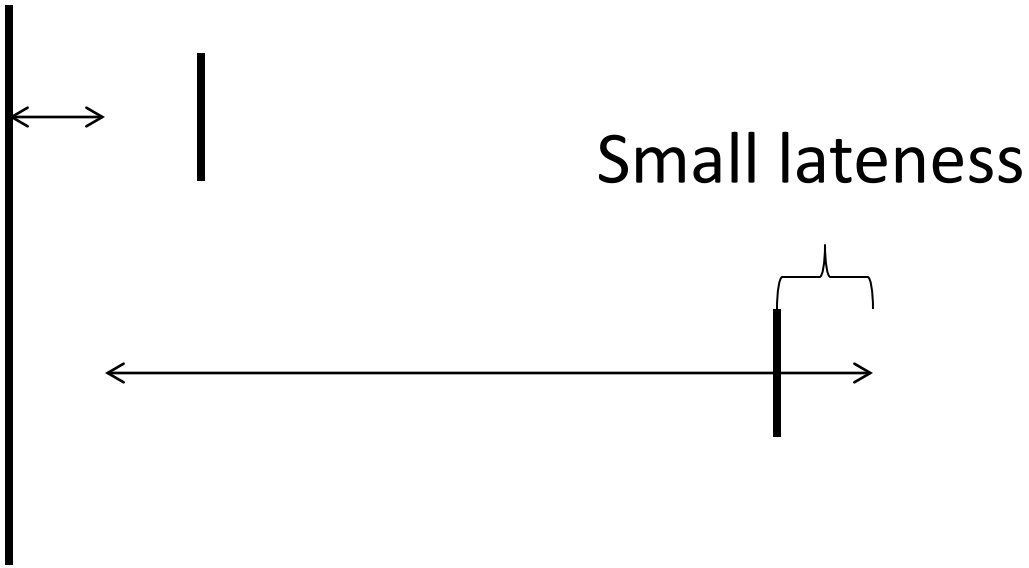
Minimizing Lateness

- How should we prioritize tasks?
  - By slack time $d_i - t_i$?
    - Greedy:



Minimizing Lateness

- How should we prioritize tasks?
  - By slack time $d_i - t_i$?
    - Optimal:



Minimizing Lateness

- How should we prioritize tasks?
  - By deadline?
  - Seems to simplistic, but it turns out to be the right approach

Minimizing Lateness

- Greedy algorithm: schedule tasks with earlier deadlines earlier
- Running time: time to sort deadlines

Minimizing Lateness

- Claim 1: There is an optimal schedule with no gaps between tasks

Minimizing Lateness

- In any schedule, an **inversion** is a pair of tasks where the task with the later deadline is scheduled earlier.
- Claim 2: All schedules with no gaps between tasks and no inversions have the same maximum lateness

Minimizing Lateness

- Proof: If two tasks have no inversions and no gaps, the only difference is in the order of tasks that have same deadlines
- Consider one deadline d
- All tasks with deadline d are scheduled after all tasks with earlier deadlines
- Last task with deadline d ends at same time for both schedules
- Maximum lateness for tasks with deadline d is the same in both schedules

Minimizing Lateness

- Claim 3: There is an optimal schedule with no inversions and no gaps.
  - By Claim 1, there is an optimal solution O with no gaps
  - Idea: repeatedly remove inversions, showing that the maximum lateness can only decrease.

Minimizing Lateness

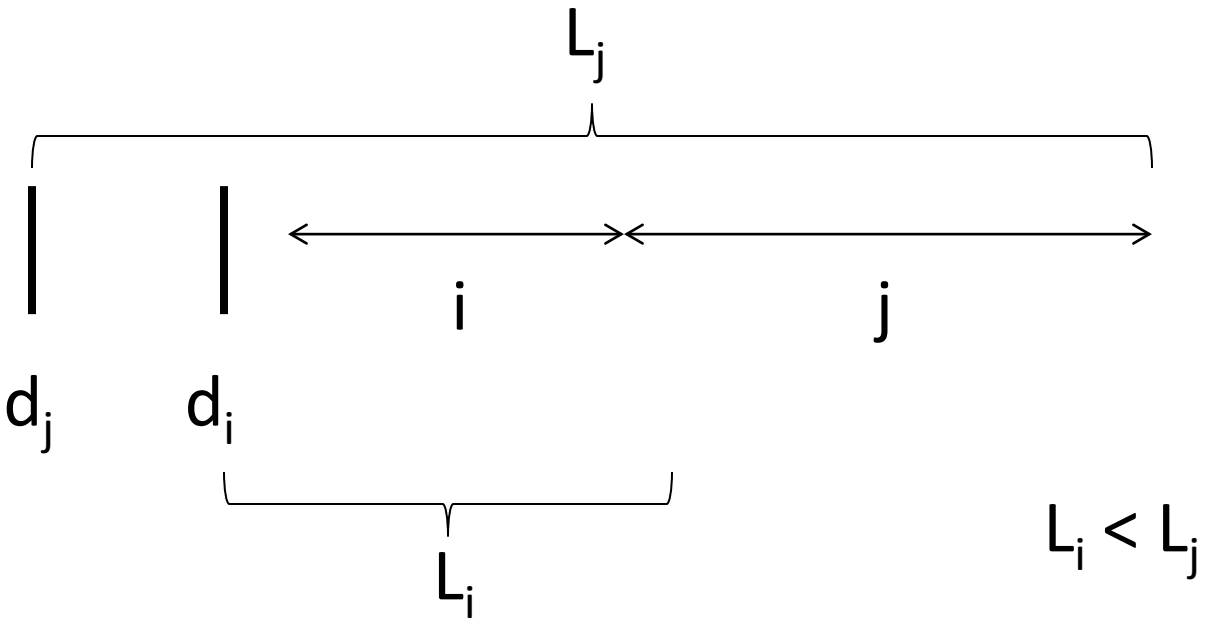
- Fact 1: If O has an inversion, there is a pair of jobs i and j such that j is scheduled immediately after i, and $d_j < d_i$
- Proof: Start with any inversion (a,b).
  - Starting from a, and continuing in the scheduled order, there must see an interval where the deadline decreases for the first time. This interval and the previous are j and i

Minimizing Lateness

- Fact 2: After swapping i and j, O has one less inversion

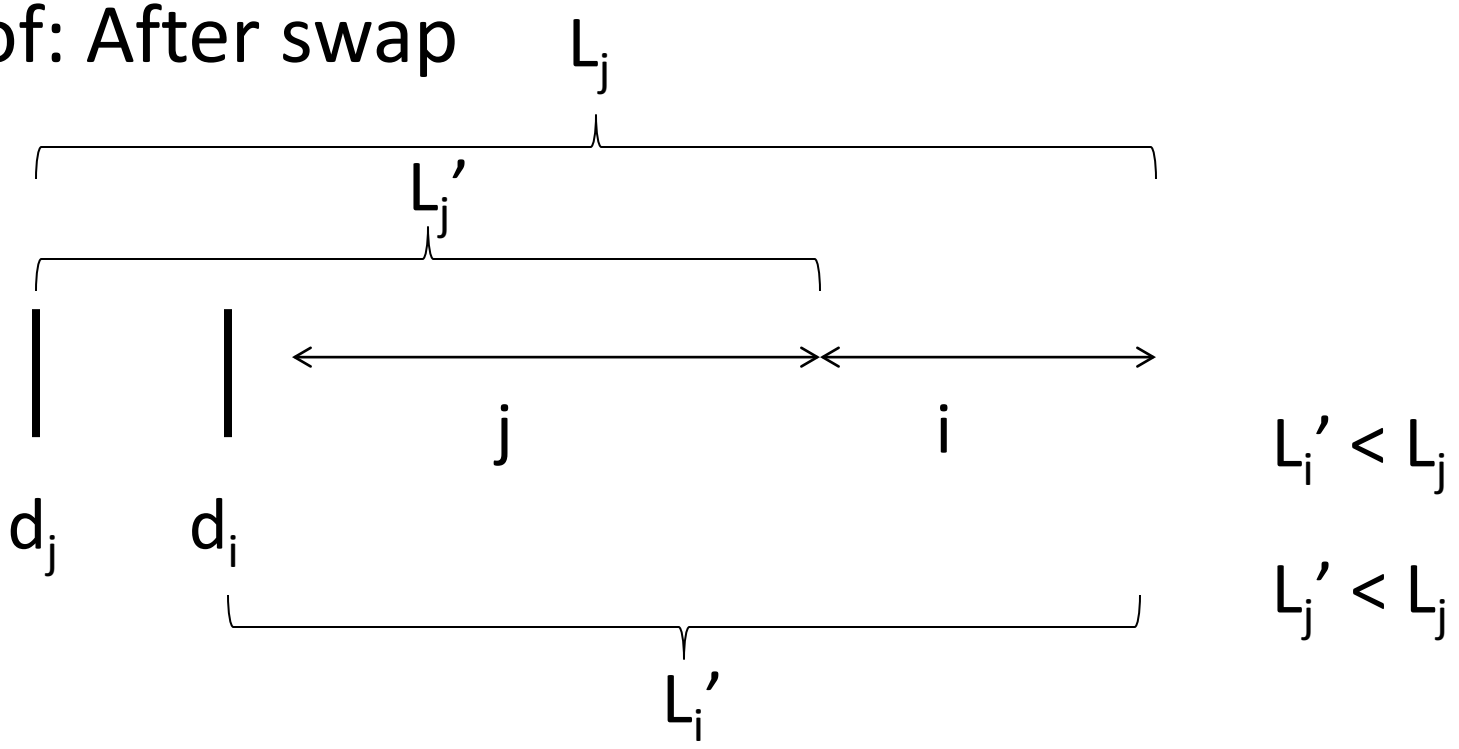
Minimizing Lateness

- Fact 3: After swapping i and j, O's maximum lateness did not increase.
- Proof: Before swap



Minimizing Lateness

- Fact 3: After swapping i and j, O's maximum lateness did not increase.
- Proof: After swap



Minimizing Lateness

- Fact 3: After swapping i and j, O's maximum lateness did not increase
- Proof: After swapping i and j, the maximum lateness between just i and j did not increase
  - Lateness of other intervals did not change
  - Therefore, maximum over all intervals did not increase

Maximizing Lateness

- Since greedy has no inversions or gaps, it must have the same maximum lateness as any other solution with no inversions or gaps, including the optimal solution guaranteed by claim 3.
- Therefore, greedy is optimal