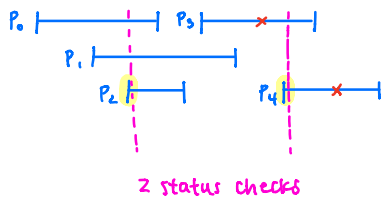
Amy Seo

Dis. 1D
10 - 11:50 AM
Song

1. a)




Algorithm:

- sort the processes by descending start time in a heap H where the root of the heap is the process with the latest start time
- initialize count = 0
- while there are processes left in H:
  - set p = minimum (root) of heap + pop it off
  - set t = start time of process p
  - increment count
  - For every process q in H:
    - if q is running at time t, remove q from H
- return count

Proof: Suppose we have $P_0, P_1, \ldots, P_n$ processes where n is the total amount of processes and C is the number of calls to status_check.

- Base case: when n = 1, there is 1 call to status_check since the heap H only holds a single process.
- Inductive hypothesis: For some value k, our algorithm then produces the minimum number of calls to status_check
- Inductive Step: Let n = k+1 processes.
  - Case 1: status_check needs to be called one more time
    - $P_{k+1}$ is not removed for earlier calls to status_check when handling all other processes
    - This means $P_{k+1}$ is still in the heap after handling all $P_k$ processes
    - since our algorithm produces the optimal solution for $P_k$ processes by the inductive hypothesis, adding one more call to status_check for $P_{k+1}$ still creates the minimum number of calls.
  - Case 2: status_check does NOT need to be called an additional time because $P_{k+1}$ is handled in some other previous process
    - $P_{k+1}$ will no longer be in heap H after processing

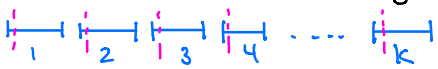
$P_0, \ldots, P_k$ processes

- By the inductive hypothesis, our algorithm produces the minimum number of calls to status-check for $P_k$ processes
- Therefore, our algorithm produces the minimum number of calls to status_check for $P_{k+1}$ processes.

∴ Our algorithm produces the minimum number of calls to status-check for any number of processes by induction.

Time Complexity: For n processes, constructing the heap takes $O(n \log n)$ time. Looping through the heap until all items are removed takes $O(n \log n)$ time.
$O(n \log n) + O(n \log n) = O(n \log n)$ runtime

b) processes where no 2 are running at same time




need k calls to status-check

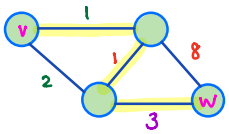
Claim: Supposing $k^*$ is the largest value of k to find a set of k sensitive processes w/ no 2 running at the same time, there must be a set of $k^*$ times at which you can run status-check so that some invocation occurs during the execution of each sensitive process.

Proof: (by contradiction) Suppose there is no set of $k^*$ times that will run status-check to successfully have some invocation occur during the execution of each process.

- But calling status-check at the start of every process will always create an invocation during each process since no 2 processes overlap → there is no way one call can cover more than one process
- Thus, we have found a set of $k^*$ times that satisfies the conditions of having an invocation to status-check occur during the execution of each sensitive process, but this is a contradiction.

∴ There must be a set of $k^*$ times at which you can run status-check so that some invocation occurs during the execution of each sensitive process. by contradiction.

2.




first "search": 2 queries
second "search": 2 queries
third "search": 1 query

- no traveling backwards in time = no negative edges → Dijkstra's

Algorithm: modified Dijkstra's (take queries into account)

- Let V be the set of visited nodes, initially only holding the start node v
- Let d(u) hold the minimum time to get to some node u
  - Initialize d to hold d(v) = 0
- Let P hold pairs of (destination: previous) to keep track of the path
- while V ≠ all nodes in graph G of destinations:
  - select node n not in V with at least one edge from a node u in V such that $f_e(d(u))$ is as small as possible.
  - set d(n) equal to this time:
    $d(n) = \min_{e=(u,n) \mid u \in V, n \notin V} f_e(d(u))$
  - add n to V
  - set P(n) = u
- to recover the fastest path, traverse P(v) → P(P(v)) → ... until you get to w.

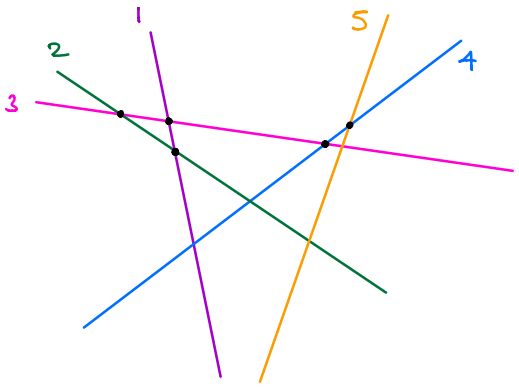
Proof by induction: (G = 1)

- Base case: when there is only one node v, V = {v} and d(v) = 0 and so the time it takes to get to v = 0 so we know the min. distance.
- Inductive hypothesis: suppose G > 1 and our base case holds true, that is, we know the min distance/time it takes to reach w.
- Inductive step: we arrive at the k+1 node in our path from the $k^{th}$ node u we have already visited since we cannot travel backwards in time.
  - It follows that it takes $f_{e=\{u,n\}}(d(u))$ time to get to n from u by the inductive hypothesis where d(u) is the time it takes to reach u, since we cannot arrive earlier by travelling later.
  - Thus, the minimum time to reach n must be $f_e(d(u))$ as the function f is monotone increasing.

∴ Our algorithm will find the minimum time path to get from v to w.

Time complexity: In the worst case, we would have to query every edge because at each node, we want to know the time it'll take to get to all surrounding destinations, so we will query at most O(e) times, where e is the number of edges.

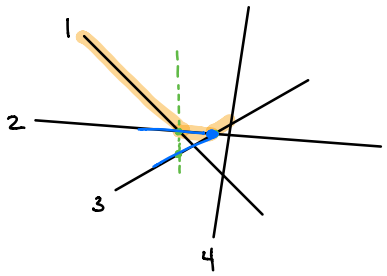
3.




ordering of slopes from smallest to biggest:
1, 2, 3, 4, 5

Algorithm: divide + conquer

- sort lines in increasing order of slope
- keep dividing set of sorted lines in half until each set has $\leq 3$ lines
  - Let $a_0$ be the first line
  - Let $a_1$ be the second line (let $a_2$ be the third line if there is one)
  - denote last line as $a_\ell$ (either $a_1$ or $a_2$ if it exists)
  - Let $(x_i, y_i)$ be the intersection point of $a_0$ and $a_\ell$
  - Note that $a_0$ and $a_\ell$ are always visible*
  - If $a_\ell = a_2$, $a_1$ is visible iff it's intersection point with $a_0$ $(x_i, y_i)$ is to the left of $a_0$'s intersection w/ $a_2$ $(x_j, y_j)$. That is: $x_i < x_j$
    - If $x_i < x_j$:
      $S = \{a_0, a_1, a_2\}$
      $I = \{x_0, x_1\}$
      ($x_0$: intersection between $a_0$ & $a_1$; $x_1$: intersection between $a_1$ & $a_2$) (just x-coordinates)
    - Else:
      $S = \{a_0, a_1\}$
      $I = \{x_0\}$




slopes: $a_1 < a_2$

- merge: Let $S_p$ and $S_q$ be two sorted lists $S_p = \{a_0, a_1\}$, $S_q = \{a_2, a_3\}$ and $I_p = \{x_0\}$, $I_q = \{x_1\}$ be their intersection x-coordinate lists respectively.
  - combine $I_p = \{x_0\}$ and $I_q = \{x_1\}$ into one list of intersections sorted by increasing x-coordinate $I' = \{x_0, x_1\}$
  - Let $S'$ be $S_p$ + $S_q$ that are sorted by slope: $S' = \{S_p, S_q\}$
  - Let the length of $I$ be denoted by $l$
  - $a_0$ and $a_{l-1}$ are visible*
  - For every intersection x-coordinate $x_i$ from $x_0$ up to & including $x_{l-1}$
    - Let $a_{pi}$ be the uppermost line in $S_p$ at $x_i$
    - Let $a_{qi}$ be the uppermost line in $S_q$ at $x_i$
    - Let $x_i'$ be the smallest x-coordinate where $a_{qi}$ lies above $a_{pi}$
      - Let this coordinate of intersection be represented as $(x_i', y_i')$ & add it to $I'$ and remove $x_i$
    - Else
      - remove $a_{pi}$ from $S'$ and $x_i$ from $I'$
- recursively divide & merge until you have one list of visible lines $S$

Time Complexity: $T(n) = \underbrace{2T(\frac{n}{2})}_{\text{dividing}} + \underbrace{cn}_{\text{merging each } S_p, S_q \text{ and } I_p, I_q \text{ per merge}}$

$= 2\left(2T(\frac{n}{2^2}) + \frac{cn}{2}\right) + cn$

$\vdots$

$= 2^i T(\frac{n}{2^i}) + icn$

$\hookrightarrow \frac{n}{2^i} = 1$

$2^i = n$

$1 = \log n$

$T(n) = 2^{\log n} T(\frac{n}{2^{\log n}}) + (\log n)cn$

$= nT(1) + cn\log n$

$\approx n + n\log n$

$= O(n\log n)$ for $n$ given lines

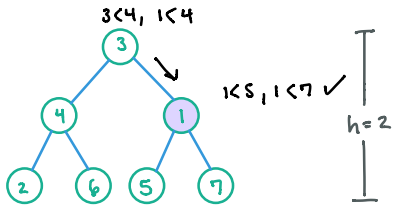
Proof: Suppose our algorithm returns the incorrect list of visible lines $V$ when the correct list is $V'$

- Case 1: $V'$ contains some line $L'$ that is not in $V$
  - By definition, $L'$ must be the uppermost line at some $x$-coordinate $x'$.
  - This $x$-coordinate must be the point of intersection between the uppermost lines of some 2 subsets at $x_0$.
  - It follows that $x_0$ would be in our set $I$ of intersections by the nature of our algorithm, and this intersection at $x'$ would have been added, along with $L'$ to our list of visible lines.
  - $L'$ would only be removed if there is a more uppermost line in the same subset that intersects with $x'$, which is not the case → contradiction.

- Case 2: $V$ contains some extra line $L$ that is not in $V'$
  - By definition of visible line, $L$ must not be uppermost at any $x$ coordinate if it is not in $V'$
  - But our algorithm declared $L$ to be uppermost at some coordinate $x'$ by the intersection of 2 lines in different subsets who are uppermost of $x_0$.
  - Thus, $L$ must be uppermost immediately to the left of $x'$ and some other line (without loss of generality) $L'$ must be uppermost to the right of $x'$.
    → Contradiction as $L$ is not in $V'$

∴ By contradiction, our algorithm always returns the valid list of visible lines.

4.




Observations:
- no going backwards
  - longest simple path in complete B.T. = $2h$
- all distinct values

Algorithm:
- choose an arbitrary node $n$
- ☆ probe $n$ and all of its neighbors
- if $n$ has the lowest value compared to all its neighbors
  ↳ return $n$ as a local minimum
- else
  ↳ choose the neighbor with the lowest value and set it to $n$ and go back to ☆

Proof: Suppose our algorithm returned a node $p$ that is not a local minimum and that a neighbor $q$ in our tree is a valid local minimum.
- Case 1: $q$ has been previously visited
  - Then our algorithm would've chosen $q$ since $q < p$ by definition
  - This is a contradiction
- Case 2: $q$ has not been visited yet
  - Then our algorithm would choose $q$ next to visit as $q < p$ by definition; it would not stop at $p$
  - This is a contradiction

∴ By contradiction, our algorithm always finds the local minimum.

Time Complexity:
- note that our algorithm never backtracks
- longest simple path = $2h$ where $h$ is height of binary tree
- each iteration performs at most 4 probes
  - itself
  - parent (if applicable)
  - left child (if applicable)
  - right child (if applicable)
- total probes = $2h(4) = 8h$, $h \leq \log(n) + 1$

$$8h \leq 8\log(n) + 1$$
$$\downarrow$$
$$O(\log n) \text{ probes}$$

5. 1 3 4 5 7 ↲ 4 5 7 1 3
5 7 1 3 4
↑
k=2

Algorithm: Let arr be the input array
- initialize L = 0 (left pointer)
- initialize R = len(arr) − 1 (right pointer)
- if arr[R] > arr[L], the array is already sorted + not rotated
  ↳ return k=0
- while L < R:
  - let m = L + (R−L) // 2
  - if arr[m] > arr[R]:
    ↳ set L = m+1
  - else
    ↳ set R = m
- k = L
- return k

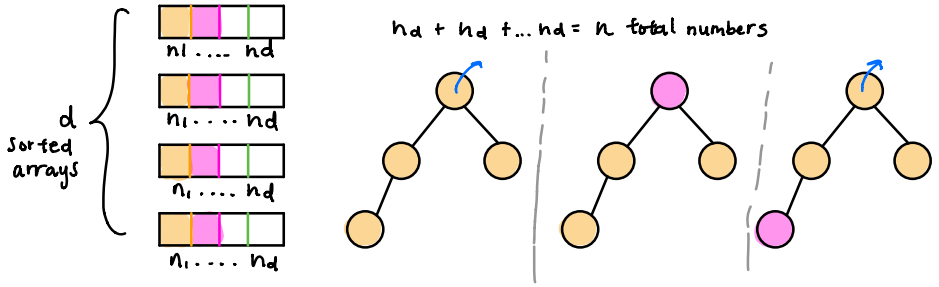
Proof: Suppose our algorithm returns an incorrect K when the right answer is K'
- Case 1: K' < K
  - ☆ Then, arr[K'] < arr[K] by definition
  - It follows that all elements before K' are less than arr[K'] and all elements after K are greater than arr[K]
  - Then at some point, our algorithm must set R = K or set R = X < K to reduce the searching space until L=R=K' as there is no way we would remove K' from the search space by the nature of our algorithm
  - Contradiction as we removed K' to keep K in our search space.

- Case 2: K' > K  4 5 6 1 2 3 (K, K')
  - ☆ still applies
  - It follows that all elements before K are less than arr[K] and all elements after K' are greater than arr[K']
  - Then at some point, our algorithm must set L=m+1 such that L=R and K is now removed from the search space.
  - Contradiction as K cannot be removed if our algorithm returned it

∴ Our algorithm always returns the correct K value.

Time Complexity: this is a modified binary search as we continue reducing our search space in half so it runs in O(log n) time

6.




Algorithm:
- Initialize a min-heap H to hold the first element of all d lists.
- Initialize an empty array A
- ★ pop off the min (root) and add the next element in the list that min belongs to into the heap & reheapify
  - ↳ append min to A
- repeat ★ until H is empty
- return A

Proof: Suppose our algorithm returns an incorrect sorting
- There must be some integer a that is less than b, but our algorithm returns a list with b having a smaller index than a
  [... b ... a]

- Case 1: b and a are in the min-Heap at the same time
  - Since our heap is organized by minimum values, b would not be able to be popped off before a since $a < b$
  - Contradiction as our algorithm had to have popped b off before a to have it return this incorrect ordering

- Case 2: b and a are not in the min-Heap at the same time
  - Since all d lists are sorted, if a and b are not in the heap at the same time, one has to be in the heap before the other.
  - It follows then, that a will be inserted into the min-heap before b as $a < b$, and a will be popped off before b enters the heap by definition of this case.
  - Contradiction as b was popped off before a.

∴ By contradiction, our algorithm always returns the valid sorting.

Time complexity:
- We will add all n elements to the heap
- For each element added to the heap, we have to reheapify, which takes $O(\log d)$ time as we will have a max of d items in the heap
- Runtime = $O(n \log d)$