# UCLA
## Computer Science Department

## CS180– Midterm
## Algorithms & Complexity

10/30/2018

Name: [redacted]
UID: [redacted]

This exam contains 7 pages (including this cover page) and 6 questions.

- Writing has to be legible.
- Express algorithms in bullet form, step by step.

**Distribution of Marks**

| Question | Points | Score |
|---|---|---|
| 1 | 20 | 20 |
| 2 | 20 | 13 |
| 3 | 20 | 12 |
| 4 | 10 | 10 |
| 5 | 20 | 20 |
| 6 | 10 | 6 |
| Total: | 100 | 81 +4 |

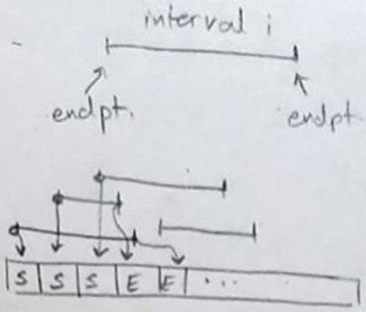
Name(last, first): [redacted] Midterm 10/30/2018

20

1. (20 points) Consider a set of intervals $I_1, I_2, \cdots, I_n$:
   (a) Design a linear time algorithm (assume that intervals are sorted in any manner you wish) that assigns the intervals to the minimum number of processors.
   (b) Prove the correctness of your algorithm.

a) · Calculate maximum density to determine minimum number of processors

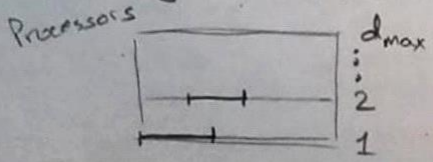
Given → · Sort all endpoints of the intervals, according to time $t$




data on endpt holds:
- time $t$
- whether it's a start of an interval or an end

· Perform a linear scan on these sorted inputs, w/ a variable $d$ (for density)
  → $d$++ if an 'S' is encountered
  → $d$-- if an 'E' is encountered
· Keep track of max($d$). Maximum value of $d$ == min. # of processors

∴ Assign intervals to first available processor, as dictated by interval start time




O(n) time!

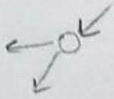
b) Assume to the contrary there exists a solution $S'$ that can solve this problem w/ $< d$ processors.
This means that the number of overlapping intervals is $< d$, otherwise there would be bottlenecking, and this solution would be invalid.
However, the algorithm described in a) keeps track of the highest density of intervals, so $S$ must equal the solution output by the algorithm in a).

17/20

2. (20 points) (a) Design an efficient algorithm that outputs the vertices of a DAG (Directed Acylic Graph), such that if there is an edge $(x, y)$ then $x$ is output before $y$.

(b) Analyze the run time of your algorithm.


a) • Perform topological sort.

e.g. 


① Calculate "in-degrees" and "out degrees" for every vertex

② Identify any vertex v w/ an in-degree of 0 (i.e., a source) and output it.

③ Delete v from the DAG, update in-degrees of neighboring vertices

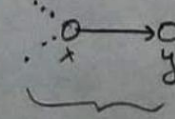
④ Repeat until there are no nodes left*

*Guaranteed to have no nodes left because a DAG has no cycles, so there will always be a source

b) This algorithm runs in $O(e)$ time, where e = # of edges. Each edge is visited exactly once, and you can "charge" the updating of vertex degrees to the deletion of edges when a source is removed, so the runtime of this algorithm is proportional to # of edges, ∴ $O(e)$

3/10

Proof: Assume to the contrary that ∃ an ordering from this algorithm where, in a DAG with edge (x, y), y is output before x. This means there exists a subgraph in the DAG where:



subgraph

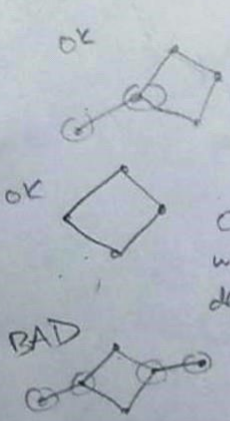
However, y would not be output before x in this case, because vertices are only output if their in-degree == 0, and as long as x exists in the DAG, y's in-degree will be ≥ 1. ∴ y cannot be output before x.

12

3. (20 points) An undirected graph is said to have property $X$ if you can start from a vertex, traverse all edges of the graph exactly once, without removing your pen from the paper.
   (a) Classify the graphs that have property X?
   (b) Design an efficient algorithm for generating a traversal of a graph that has property $X$.

a) Graphs with property X are called Eulerian graphs. Graphs with this property are guaranteed to have $\leq 2$ vertices of odd degree. ✓

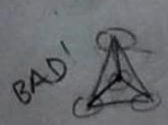
↳ we can prove this by using the pigeonhole principle A vertex with an odd degree means that there eventually has to be a vertex with "only" one edge coming out of it. Since this vertex has to be included in the graph, it must be either a start or an end point. (Can't pass through it since each edge can only be used once). If you assume an Eulerian graph has >2 vertices of odd degree, this means you have 3+ vertices that need to be either a start or end point, which is impossible.



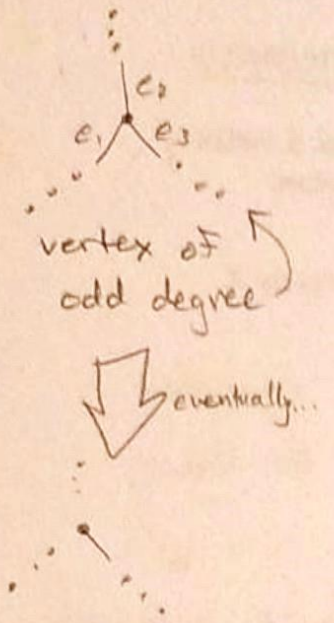

b) Perform BFS on any node! BFS runs in linear time ($O(e+v)$) and will traverse all graphs with this property

X-8

Since $\sum d_i = 2e$ (sum of degrees has to be equal to 2 times the # of edges), even something like the graph shown below wouldn't work. If you delete edges as you traverse a graph (same as forcing to use each edge only once) traversing a node w/ an odd degree means you'll reduce the degree of that vertex by 2 each time




(see next page) → → → →

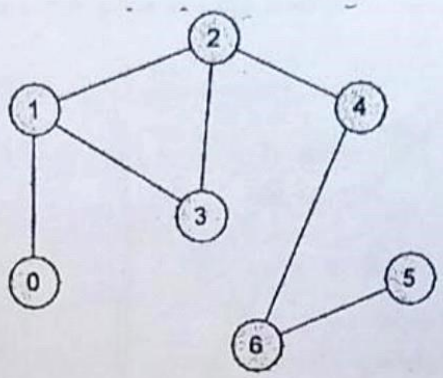


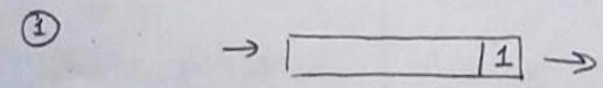

Passing through a vertex like the one on the left means you have to come in on an edge $e_1$, and out on a different edge $e_2$, meaning eventually you'll be left with only one edge to that vertex, reducing the graph down to the situation described earlier in the proof: a vertex with only one edge coming out of it ∴, a graph w/ more than 2 vertices of odd degree CANNOT be an Eulerian graph (by the pigeonhole argument made earlier in the proof). ∎

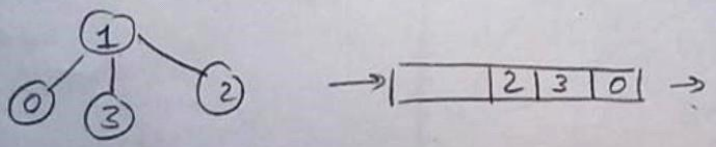
10

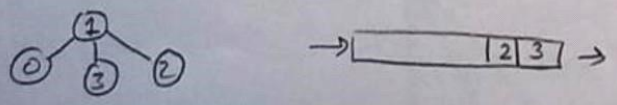
4. (10 points) Consider an unweighted graph $G$ shown below:

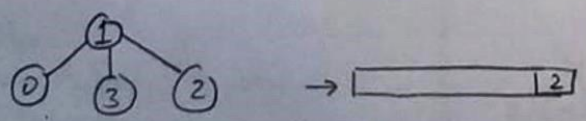
   (a) Starting from vertex 1, show every step of BFS along with the corresponding FIFO next to it.

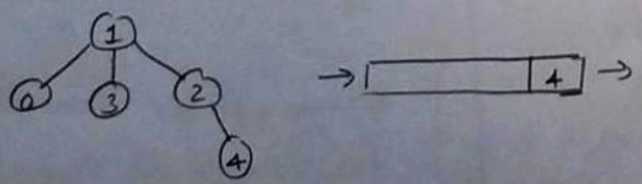


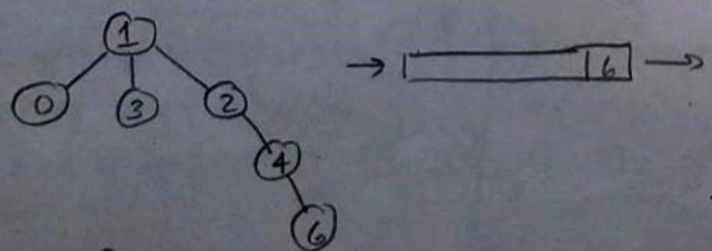

1) 

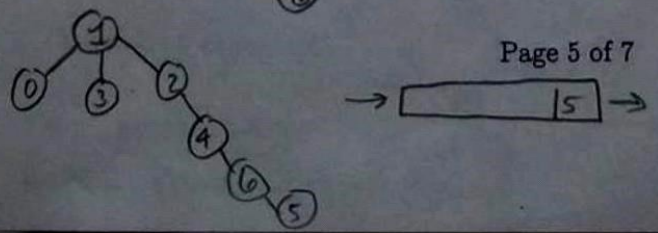

2) 

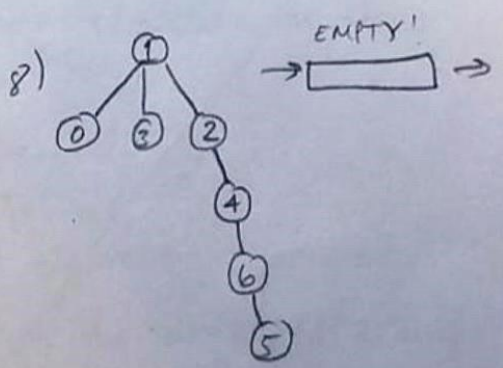

3) 


4) 


5) 

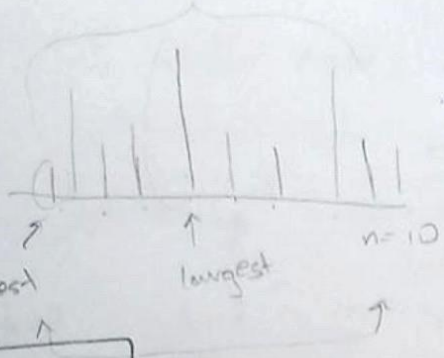

6) 


7) 


8) 

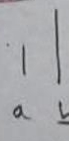
5. (20 points) Consider an unsorted list of integers. You can find the minimum number in the list with $n-1$ comparisons. Similarly, you can find the maximum with $n-1$ comparisons. So you can find both the minimum and the maximum with about $2n-3$ comparisons. Design an algorithm that finds both the minimum and the maximum using about $\frac{3n}{2}$ comparisons.

→ 1.5n




**ANSWER:**

- Each element has the potential to be either the smallest or largest.
- Compare 2 arbitrary values, a and b



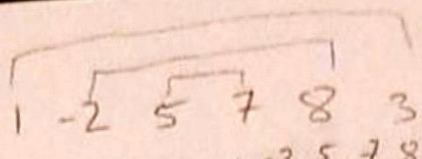

→ if a<b...
a cannot be the biggest
b cannot be the smallest
→ else
a cannot be the smallest
b cannot be the biggest

- Can accomplish this by looping from both ends and comparing values
- After $\frac{n}{2}$ loops, you've halved the problem (similar to the famous problem)

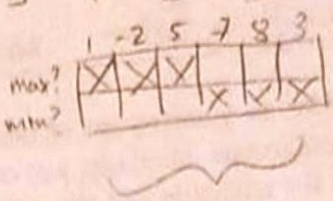
- if n == 0
  return 0,0 // no max or min
- if n == 1
  return that one element as both max and min
- compare first two elements in the array.
  ↳ if first ≤ second
  s = first element
  l = second element
- For each element a in the array (starting from the third element), if it exists...
  if a < smallest
  s = a
  else if a > largest
  l = a
- s = smallest (min) value in the array
- l = largest (max) value in the array



(answer continued on back)

e.g) 


- Repeat process separately for only the max candidates and only the min candidates




After 3 comparisons

- Problem halves each time, until the set of max candidates and/or the set of min candidates is $\leq 3$, at which point you can perform a linear scan to find smallest out of min candidates and largest out of max comparisons

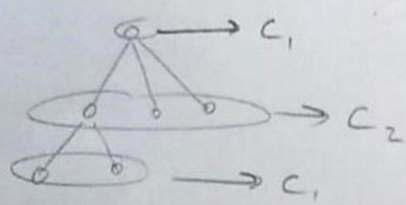
$n = 6, \quad 1.5n = 9$

$9 = 3 + 2 + 2$

- 3: initial comparisons
- 2: find min
- 2: find max

6. (10 points) Give an algorithm to color a graph with 2 colors (assuming it is 2-colorable). A proof of correctness is not necessary.

- Perform BFS, starting on any node
- Assign nodes on even levels (starting w/ the start node and including every other level) color $c_1$
- Assign nodes on odd levels (levels 1, 3, ...) color $c_2$




2-colorable means a graph w/ no odd cycles

-4 Time Complexity