UCL

CS 180 Algorithms & Compl

Midterm Total Time: 1.5 hours

Each problem has 20 points.

All algorithm should be described in English, bullet-by-bullet

1 Consider a set of intervals I1, …, In. **a.** Design a linear time algorithm (assume intervals are sorted in any manner you wish) that finds a maximum subset of mutually non-overlapping intervals. **b.** Prove the correctness of your algorithm.

a) Assume each interval I_i is represented by a starting time S_i and a finishing time f_i. Prior to running the algorithm, assume the starting times and the finishing times have been sorted in increasing order.

- While there are remaining intervals
  - Select the interval with the minimum finishing time and add it to the Schedule
  - Delete all intervals who overlap with the interval selected. Any interval whose starting time is less than or equal to the finishing time of the Scheduled interval is considered overlapping.

b) - Suppose there is an optimal schedule O whose intervals are represented by $O_1, O_2 \cdots O_k$. Additionally, suppose the algorithm given above produces the Schedule A whose intervals are given by $A_1, A_2 \cdots A_m$.

It can be shown that for all $r < k$, $f(A_r) \leq f(O_r)$

- By induction, examine the case $r = 1$. A_1 must be the interval with the least finishing time of all intervals. O_1 can either be the interval with the least finishing time or another with a greater finishing time.

In either case, it can be seen that $f(A_1) \leq f(O_r)$

- Examine the case $r = n$, assuming the inductive hypothesis that $f(A_{n-1}) \leq f(O_{n-1})$. We know that O_n is in the set of available intervals for A_n, since $f(A_{n-1}) \leq f(O_{n-1}) < f(O_n)$. In the best case, A_n will be an interval with a finishing time earlier than $f(O_n)$. In the worst case, $f(A_n) = f(O_n)$.
- In either case, $f(A_n) \leq f(O_n)$.

- The number of intervals in A is greater than or equal to the number of intervals in O.
  - By contradiction, suppose $|O| = |A| + 1$. This means there is an interval in O that is not in A.
  - By the previous part of the proof, we know every $f(A_i) \leq f(O_i)$, so this interval must be greater than O_k, we will call this interval O_{k+1}.
  - The finishing time $f(O_{k+1}) > f(O_k) \geq f(A_k)$. Thus, it is available for A to include.
  - The algorithm producing A does not terminate until there are no more available intervals. So O_{k+1} would be included in A. This contradicts the assumption.

20

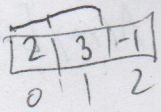
2 . a. Design an efficient algorithm better than O (n^2) to be used in sparse graphs for finding the shortest path between two vertices S and T in a positive weighted graph. **b.** Justify the correctness of your runtime analysis.

a) One efficient algorithm would be an implementation of Dijkstra's algorithm that updates a heap containing the path lengths to each vertex.

- Create a set M for which the minimum path lengths have been finalized. Add the starting vertex S to the set.
- Assuming the graph is connected, consider the set of all vertices in the graph to be part of the set V.
- Create a heap H which will contain the path lengths to every currently reachable node.
- While the ending vertex T is not in M
  - For all vertices not in M that are connected by an edge to a vertex in M, calculate the minimum weight path to S. Store these path lengths in H.
  - Select the minimum-length path in H (which will be at the root of H)
    - Add the vertex it corresponds to into M.
    - If the selected vetex is T, output the path length
    - Delete the path length from H.

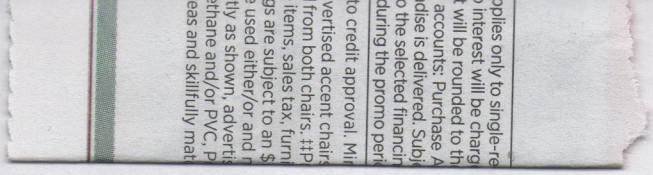
b) The algorithm will run in $O(e \log n)$ time.

- Updates to the heap will be made each time a node is to added to M. Each update involves examining the edges adjacent to the newest added node, and after this these edges are never looked at again. Thus, there are $O(e)$ updates to the heap.
- Making an update to the heap takes $O(\log n)$ time, as up to $(n-1)$ nodes could be stored in the heap at once (the starting node will never be in the heap).
- Since the graph is sparse, the number of edges will be less than $\frac{n^2}{\log n}$.
- The run time will be $O(e \log n)$, and the limit on the number of edges means this will run better than $O(n^2)$.

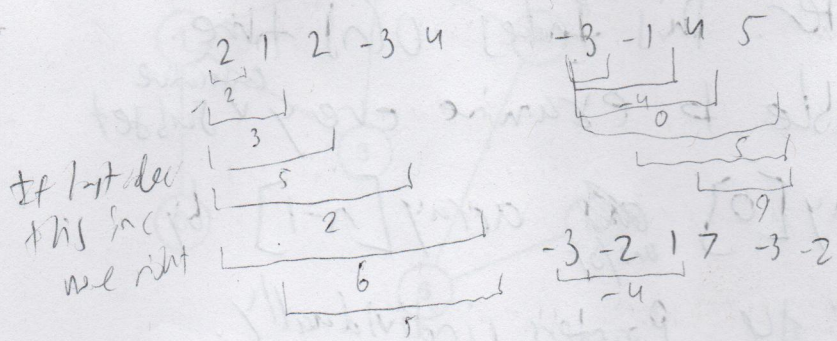


& 5

5



3. Consider a sequence of positive and negative (including zero) integers. Find a consecutive subset of these numbers whose sum is maximized. Assume the weight of an empty subset is zero. **a.** Design a linear time algorithm. **b.** Prove the correctness of your algorithm.
Example: For the sequence 2 -3 5 -12 the maximum sum is 4.



Variable Reference

Indices	S	Start
	F	finish
Sums	C	current Sum
	M	max Sum

5

- Assume the integers are stored in an array.

a) - Designate two indices into the array: Start ~~and finish~~ S and finish F. Initially, both are 0.
- Designate two variables: Current Sum C and max Sum M. Initially, both are 0.
- While F is less than the ~~index of the~~ number of ~~last~~ integers in the array
  - ~~Increment F by one~~
  - Current Sum = Current Sum + Array [F]
  - If (current Sum > max Sum), set max Sum = Current Sum
  - Increment F by one.
- While S is less than the ~~index of the~~ number of ~~last~~ integers in the array.
  - Current Sum = Current Sum − Array [S]
  - If (Current Sum > max Sum), set max Sum = Current Sum
  - Increment S by one.

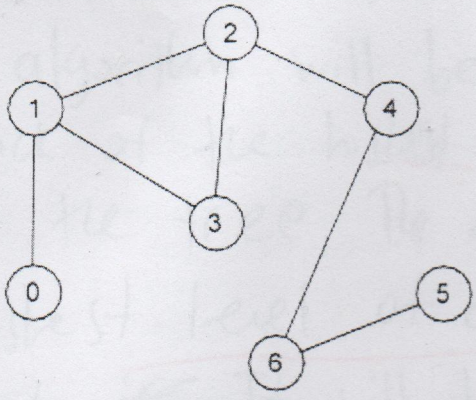
→

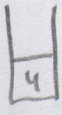
would find the maximum and/or array[n-1] for consecutive subset that includes Array[0] on n-integer array

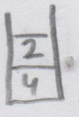
b) • This runs in O(n) time, because it takes n steps to increment the F Pointer and O(n) steps to ~~increment~~ increment the S Pointer. This takes O(n) time.
• The algorithm is able to examine every consecutive subset that includes Array[0] and/or array[n-1] by incrementing each of the pointers individually.

20

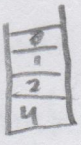
4. Consider an unweighted graph G shown below. **a.** Starting from vertex **4,** show every step of DFS along with the corresponding stack next to it. **b.** What is the run time of DFS if the graph is not connected (no proof is necessary)?



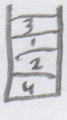
a) 1. Add 4 to the Stack. 

2. Add 2 to the Stack. 

3. Add 1 to the Stack. 

4. Add 0 to the Stack. 

5. Pop 0 off the Stack. 

6. Add 3 to the Stack. 

7. Pop 3 off the Stack. 

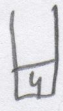
8. Pop 1 off the Stack. 

9. Pop 2 off the Stack. 

10. Add 6 to the Stack. 

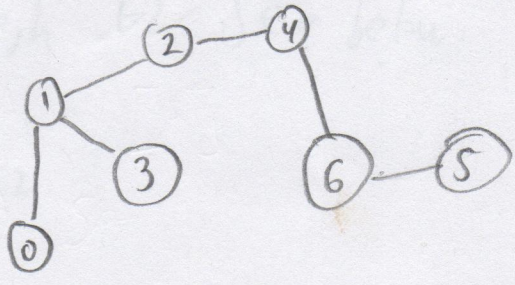
11. Add 5 to the Stack. 

12. Pop 5 off the Stack. 

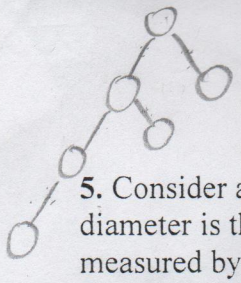
13. Pop 6 off the Stack. 

14. Pop 4 off the Stack. 

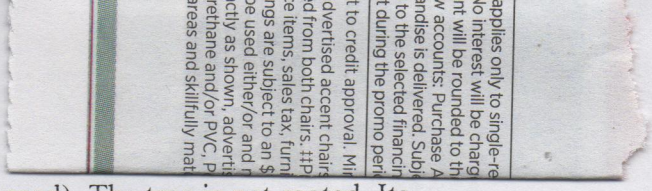
DFS Tree Produced.



b) DFS runs in $O(e+v)$ time.



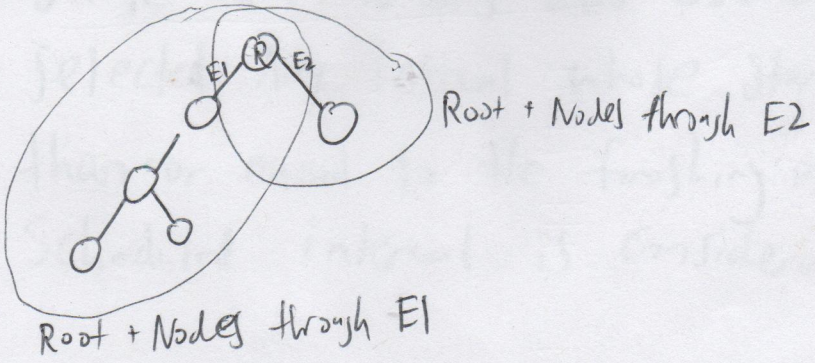
12



5. Consider a binary tree (it is not necessarily balanced). The tree is not rooted. Its diameter is the distance between two vertices that are furthest from each other (distance is measured by the number of edges in a simple path). Design a linear time algorithm that finds the diameter of a binary tree.

The algorithm will be based on DFS, and will keep track of the highest level node in each branch of the tree. The diameter will be the sum of the highest level of each branch. ~~Since it does not~~ ~~Note~~ ~~is~~ It will have the same run time as DFS.

- Select a node with degree = 2 (two edges attached) to be the root R (This step takes O(V) time to scan all the vertices)
- Assign one edge of R to be E1 and the other edge to be E2. Divide the graph in two halves: the root and all nodes reachable through E1 and the root and all nodes reachable through E2. See below.



- Run DFS on both partitions of the graph. (Takes O(e+v))
- The diameter is the sum of the highest level found for each partition.