Final Exam

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on every page of this quiz booklet.
- You have 180 minutes to earn **150** points. Do not spend too much time on any one problem. Read them all through first, and attack them in the order that allows you to make the most progress.
- This quiz booklet contains 13 pages, including this one. Two extra sheets of scratch paper are attached. Please detach them before turning in your quiz at the end of the exam period.
- This quiz is closed book. You may use **three** $8\frac{1}{2}'' \times 11''$ or A4 crib sheets (both sides). No calculators or programmable devices are permitted. No cell phones or other communications devices are permitted.
- Write your solutions in the space provided. If you need more space, write on the back of the sheet containing the problem. Pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. It is sufficient to cite known results.
- Show your work, as partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat.
- Good luck!

Problem	Parts	Points	Grade	Grader		Problem	Parts	Points	Grade	Grader
1	10	30				5	1	20		
2	1	10				6	1	15		
3	1	10				7	1	20		
4	1	20				8	3	25		
						Total		150		

Name: SOLUTIONS

Problem 1. True or False [30 points] (10 parts)

For each of the following questions, circle either True, False or Unknown.

1. After hashing n keys into a hash table of size m that uses chaining to handle collisions, we hash two new keys k_1 and k_2. Under the simple uniform hashing assumption, the probability that k_1 and k_2 are hashed into the same table location is exactly $1/m$ with no dependence on the number of keys n.

   Answer = True False

2. Under the uniform hashing assumption, if we use a hash table of size m with open addressing to hash 3 keys, the probability that the third inserted key needs exactly three probes before being inserted into the table is exactly $\frac{2}{m(m-1)}$.

   Answer = True False

3. We use a hash table of size m with open addressing to hash n items. Under the uniform hashing assumption, the expected cost to insert another element into the table is at most $1 + \alpha$, where $\alpha = n/m$ is the average load.

   Answer = True False

4. There is a polynomial-time algorithm for the Knapsack problem if all items have size in $\{0, 1\}$ regardless of the bits required to describe their values and the size of the knapsack.

   Answer = True False Unknown

5. There exists a polynomial-time algorithm for finding longest simple paths in weighted directed acyclic graphs.

   Answer = True False Unknown

6.Every search problem in NP can be solved in exponential time.

Answer = True False Unknown

7.If there are negative edges in a graph but no negative cycles, Dijkstra's algorithm still runs correctly.

Answer = True False

8.In a shortest path problem, if each arc length increases by k units, shortest path distances increase by a multiple of k.

Answer = True False

9.For any two functions f and g, we always have either $f = O(g)$ or $g = O(f)$.

Answer = True False

f, g oscillate

10.Dijkstra's shortest path algorithm runs in $O(V^3)$ time.

Answer = True False

The question is Big-Oh not Θ!

Problem 2. Data Structures [10 points]

Design a data structure that keeps a sequence of real numbers $S = (x_1, ..., x_n)$, and supports the following operations in $O(\log n)$ time, where n is the current length of the sequence:

- Insert(y, i): inserts y between x_i and x_{i+1}
- Sum(i, j): compute the sum $\sum_{t=i}^{j} x_t$

Assume that the data structure is initially empty.

- We use an AVL tree-inspired data structure. Every node in the tree keeps track of the #nodes in the left subtree, the total sum of the nodes in the left subtree, and its own value
- At any point in time, the tree is an AVL tree keyed on the rank of each number in the current sequence S (Important note: we do not assume that the maintained sequence of numbers is ordered).
- To perform Insert(y,i): we compare i to root.num-left

  if $i \leq$ root.num-left+1 we call Insert(y, i, root.left-child)

  else if $i >$ root.num-left+1 we call Insert(y, i-root.num-left-1, root.right-child)

  etc. until we reach an empty leaf ℓ

  ℓ.value $\leftarrow y$, ℓ.left-child = ℓ.right-child $\leftarrow$ NIL, ℓ.num-left $\leftarrow$ 0

  increase by 1 the num-left values of all nodes where we went left in the recursion, and increase by y the sum-left value

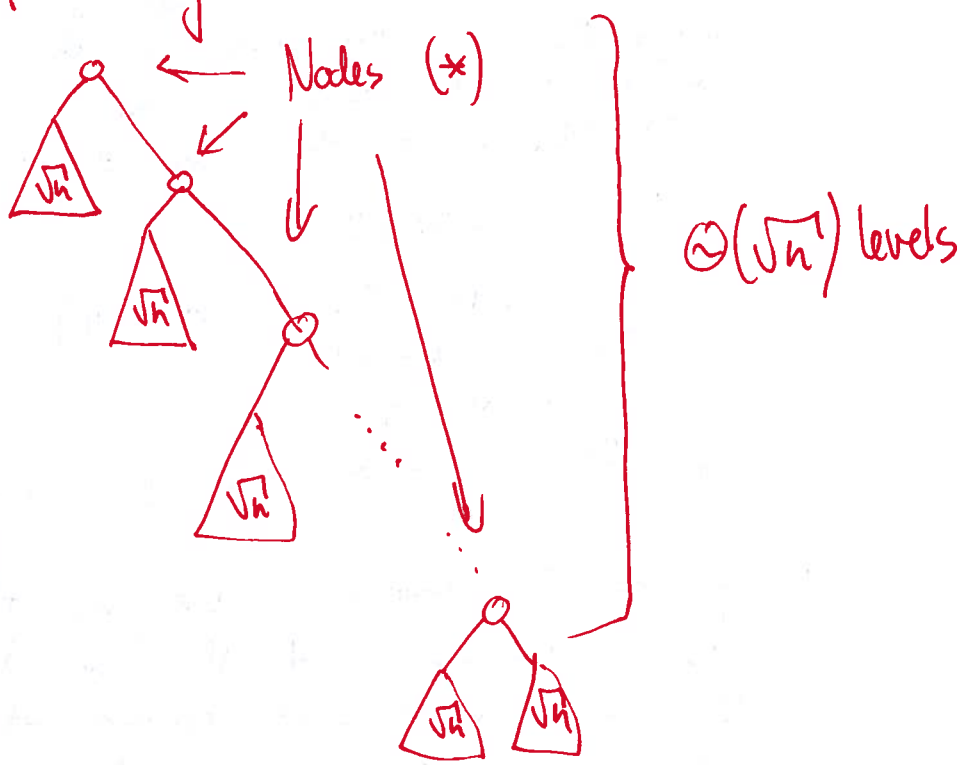
  finally, rebalance the tree, updating left-child, right-child, num-left, sum-left values of nodes as necessary
- To do Sum(i,j): find $\sum_{t=0}^{i-1} x_t$, and $\sum_{t=0}^{j} x_t$ and subtract them

  To compute $\sum_{t=0}^{i-1} x_t$ walk down the tree based on $i-1$ in the same fashion as in Insert operation with index $i-1$ but stop when $i-1 =$ num-left; keep a sum variable initialized to 0 and add left-sum to it for every node where we went right.

Problem 3. Binary Trees [10 points]

Consider the family of binary trees of n nodes with the following invariant for every node. If n_1 and n_2 are the number of nodes in the left and the right subtree respectively, then $\max\{n_1, n_2\} \leq (\min\{n_1, n_2\})^2 + 1$. Is the height of these trees bounded by $O(\log n)$? Justify your answer with a rigorous argument or a counter-example.

Consider the following tree:



Each of $\triangle\sqrt{n}$ is a balanced binary tree on $\sqrt{n}$ nodes. The invariant holds for each node trivially, if for every node, you make sure that

$|n_1 - n_2| \leq 1$.

The invariant does hold for each of the nodes (*) because the smaller tree is the left one, and its size is $\sqrt{n}$, so $(\sqrt{n})^2 + 1 = n + 1$ clearly bounds the size of the right subtree.

The height of this tree is $\Theta(\sqrt{n})$, so the height of the trees is <u>not</u> bounded by $O(\log n)$.

Problem 4. k^{th} **minimum in min-heap** [20 points]

Present an $O(k \log k)$ time algorithm to return the kth minimum element in a min-heap H of size n, where $1 \leq k \leq n$. Partial credit will be given to less efficient solutions provided your complexity analysis is accurate.

Create a new min-heap I which is initially empty. Insert the root of H into I with the pair (H[0], 0), where H[0] is the value of the root and 0 is the index of the root of H. (Heap I is a min-heap with respect to the first element in the pair.) Then,

```
For i=1... K:
    let (v, p) = I.extractMin()
    if i == K:
        return v
    Insert both of node p's children into I.
```

After i iterations of the loop, the i smallest elements ~~have been e~~ of H have been extracted from I and every other element of H is either in I or is a descendant of some node in I. Thus, the min element in I is the next smaller element of H.

The size of heap I never exceeds O(K), so operations on I take O(log K) time. We must add and remove O(K) elements before we are done, so our algorithm is O(K log K).

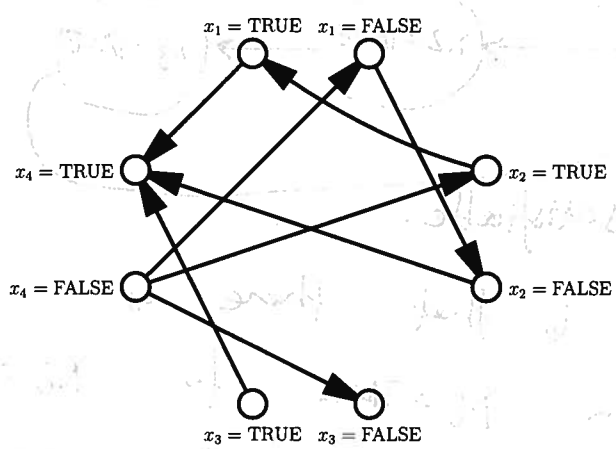
Problem 5. 2-Satisfiability [20 points]

The 2-SAT problem is defined as follows: There are n Boolean variables $x_1, \ldots, x_n$, and a set of m clauses. Each clause has two variables which are either in true (x_i) or complemented ($\overline{x_i}$) form. Here are two examples:

1. $(x_1 + \overline{x_2})(\overline{x_1} + x_4)(\overline{x_3} + x_4)(x_2 + x_4)$ is satisfiable.
2. $(x_1 + x_2)(\overline{x_2} + \overline{x_3})(x_3 + x_1)(\overline{x_1} + x_4)(\overline{x_4} + \overline{x_1})$ is not satisfiable.

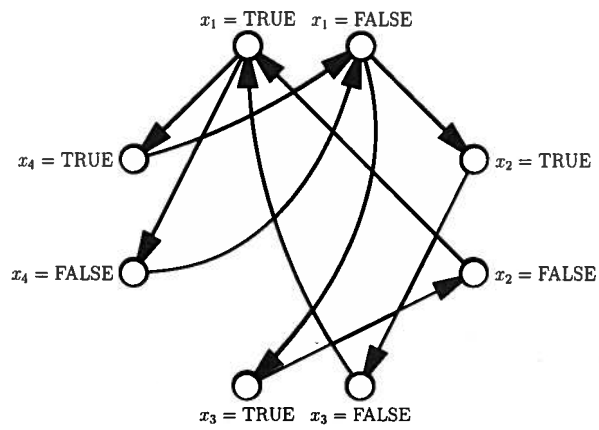
For a 2-SAT formula to be satisfiable, every clause should be satisfiable. To satisfy a clause $(x_1 + \overline{x_2})$ we can set $x_1 = \text{TRUE}$ and/or $x_2 = \text{FALSE}$. If $x_1 = \text{FALSE}$ and $x_2 = \text{TRUE}$, the clause is not satisfied. Example 1 is a satisfiable set of clauses because we can set $x_1 = \text{TRUE}$, $x_2 = \text{TRUE}$, $x_3 = \text{FALSE}$ and $x_4 = \text{TRUE}$ to satisfy all the clauses. There is no satisfying assignment for Example 2. Variable assignments that are required to satisfy some of the clauses conflict with assignments that are required to satisfy other clauses in this case. The 2-SAT problem is to find a satisfying assignment, if one exists. We note that 2-SAT (unlike 3-SAT) can be solved in polynomial time.

Here we are only concerned with a *sufficiency* check for unsatisfiability. That is, we want to devise a graph-based algorithm that checks if a given set of clauses is unsatisfiable. We want this check to be as efficient and as general as possible. To do this, we will represent the 2-SAT problem as a graph. The two graphs for the examples above are shown below in Figures 1 and 2. Each graph has $2n$ vertices: there are two vertices for every variable corresponding to the true and complemented forms, namely, $x_i = \text{TRUE}$ and $x_i = \text{FALSE}$ for each x_i. The edges of the graph represent the implied assignments for variables. For example, for every clause of the form $(x_1 + \overline{x_2})$, we have an edge from $x_1 = \text{FALSE}$ to $x_2 = \text{FALSE}$ and an edge from $x_2 = \text{TRUE}$ to $x_1 = \text{TRUE}$ (If we set $x_1 = \text{FALSE}$, then we require $x_2 = \text{FALSE}$ for this clause to be satisfied, similarly the other case).



$$(x_1 + \overline{x_2})(\overline{x_1} + x_4)(\overline{x_3} + x_4)(x_2 + x_4)$$

Figure 1: Example 1 satisfiable



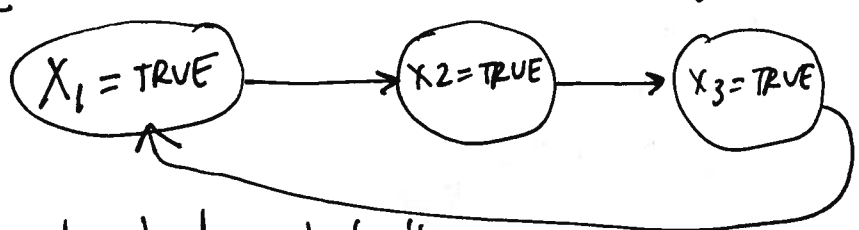
$$(x_1 + x_2)(\overline{x_2} + \overline{x_3})(x_3 + x_1)(\overline{x_1} + x_4)(\overline{x_4} + \overline{x_1})$$

Figure 2: Example 2 unsatisfiable

Devise an algorithm that operates on the graph derived from the 2-SAT problem. Your algorithm should return FALSE if it discovers that the problem is unsatisfiable and UNKNOWN otherwise. You will be graded on both the efficiency and generality of your algorithm.

The generality is defined in the following way. Let $\mathcal{A}$ and $\mathcal{B}$ be two correct algorithms. We say that $\mathcal{A}$ is more general than $\mathcal{B}$ if the set of inputs for which $\mathcal{A}$ returns FALSE is a strict superset of the set of inputs for which $\mathcal{B}$ returns FALSE.

$$(\overline{X_1} + X_2)\ (\overline{X_2} + X_3)\ (\overline{X_3} + X_1)$$



cycle but satisfiable.

The check is that there is a path from X_i = TRUE to X_i = FALSE for some X_i and a path from X_i = FALSE to X_i = TRUE for the same X_i.

[Alternately, there is a cycle that ~~has~~ contains x_i = TRUE and x_i = FALSE for some x_i.]

A naïve algorithm would be to check for each x_i whether such paths exist. A more efficient algorithm is to find strongly connected components of the given graph in $O(V+E)$ time. If any component contains both x_i = TRUE and x_i = FALSE for some i, the graph/2-SAT problem is unsatisfiable.

<u>Aside:</u> This can be shown to be both a necessary and sufficient condition. That is, if the graph does not have a component containing such a pair of vertices, it is guaranteed to be satisfiable.

Problem 6. Second Shortest Paths [15 points]

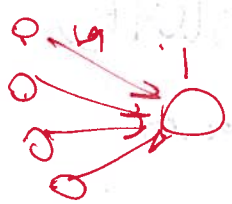
In an acyclic directed weighted graph G with a specified source vertex s, let $\alpha(i)$ be the length of the second shortest path from s to the vertex i. You can assume that all path lengths between any two vertices are distinct. How can we determine $\alpha(i)$ for all vertices in G in $O(V+E)$ time? You will receive partial credit for less efficient algorithms if your complexity analysis is accurate.

$\delta[i]$: shortest path length from s to vertex i

$\alpha[i]$: 2nd shortest path length from s to i

Fill up the values in topologically sorted order.

~~$\alpha[i]$~~



$$\delta[i] = \min_{\forall j \text{ s.t. } (j \to i)} \{ \delta[j] + w(i,j) \}$$

$$\alpha[i] = 2^{nd}\min_{\forall j \text{ s.t. } (j \to i)} \{ \delta[j] + w(i,j), \alpha[j] + w(i,j) \}$$

Problem 7. Common Vertex [20 points]

Given four vertices u, v, s and t in a directed weighted graph $G = (V, E)$ with non-negative edge weights, present an algorithm to find out if there exists a vertex $v_c \in V$ which is part of some shortest path from u to v and also a part of some shortest path from s to t. The algorithm should run in $O(E + V \log V)$ time. Partial credit will be given to less efficient algorithms provided your complexity analysis is accurate.

A vertex V_c is on the shortest path from U to V

iff $d[U, V] = d[U, V_c] + d[V_c, V]$

1. Run Dijkstra on G from vertex U.
2. Run Dijkstra on G from S.
3. Run ~~reverse~~ Dijkstra on G' (edge reversed) from V.
4. Run Dijkstra on G' from t.

Iterate over all $V_c \in V$

if $d[U,V] = d[U,V_c] + d[V_c,V]$

1 1 3

and $d[s,t] = d[s, V_c] + d[V_c,t]$

2 2 4

return V_c.

return FALSE.

complexity 4 Dijkstra + $O(V)$

$O(E + V \log V)$

Problem 8. The Ball Game (3 parts) [25 points]

Professors Devadas and Daskalakis play the following game. N balls are inserted into a tube whose diameter matches the diameter of the balls, and therefore the balls cannot change positions inside the tube. Each ball has a distinct value on it. Let $A_1, A_2, \ldots, A_N$ be the values of the balls in the order they are inserted. The tube is opaque, but it is open at both ends, so only the first ball at each end is visible (its value is visible as well). In each turn, a player removes one ball from the tube from either end and collects as many points as the value of the ball. Players take alternate turns, and each has a goal to maximize his score.

Professor Daskalakis has a reasonable strategy. He always removes the ball with a higher value (out of the two visible balls). Professor Devadas, however, uses his infra-red vision that only people who have been at MIT long enough have been secretly taught. Therefore, he can see all balls and their values through the tube. (Note that this is not regarded cheating at MIT; it is attributed to professor skills.) Of course, he wants to use his power to maximize his score.

Example: $A = (3, 7, 1, 2)$
If Professor Devadas plays first, he will choose 2, then Professor Daskalakis will choose 3, then Professor Devadas will choose 7 and finally Professor Daskalakis will take 1. So the score would be Devadas: 9, Daskalakis: 4. If Professor Daskalakis plays first, he will choose 3, then professor Devadas will choose 7, then Professor Daskalakis will choose 2 and finally Professor Devadas will take 1. So the score would be Daskalakis: 5, Devadas: 8.

Develop an efficient DP algorithm that computes the maximum score Professor Devadas can achieve when he plays first given, the array $A_1, A_2, \ldots, A_N$. Less efficient solutions will be given partial credit provided the complexity analysis is accurate.

(a) [10 points] State the set of subproblems that you will use to solve this problem and the corresponding recurrence relation to compute the solution.

Subproblems: $D[i,j]$ – max score of professor Devadas on Array $A_i, \ldots, A_j$ when he plays first $(1 \le i \le n,\ 1 \le j \le n)$

Init: $D[i,i] = A_i$, $1 \le i \le n$
$D[i,i+1] = \max(A_i, A_{i+1})$, $1 \le i < n$

Recurrence:
$$D[i,j] = \max \begin{cases} A_i + D[i+1, j-1] & \text{if } A_j > A_{i+1} \\ A_i + D[i+2, j] & \text{if } A_j < A_{i+1} \\ A_j + D[i+1, j-1] & \text{if } A_i > A_{j-1} \\ A_j + D[i, j-2] & \text{if } A_i < A_{j-1} \end{cases}$$

for $1 \le i < n-1$
$i+2 \le j \le n$

Solution: $D[1,n]$

(b) [8 points] Describe an iterative (non-recursive) algorithm to compute the maximum score. Analyze the running time of your algorithm.

```
def COMPUTE_SCORE(A):
    for i = 1,...,n:
        D[i,i] = Ai
        if i<n:
            D[i,i+1] = max(Ai,Ai+1)
loop
O(n) →  for l = 3,...,n:
loop O(n) →  for i=1,...,n-l+1:
            j=i+l-1
O(1) →      compute D[i,j] by formula in part a (recurrence)
    return D[1,n]
```

Total running time is $O(n^2)$

(c) [7 points] Modify your algorithm above to print the set of moves made by both professors. Write down the modified algorithm below.

```
def GAME(A):
    COMPUTE_SCORE(A)   //Assume that D matrix is computed by this call
    i=1, j=n
    while i≤j :
        if i==j :
            PRINT "Devadas take", Ai "at" i
            i=i+1
        elif i+1==j:
            print "Devadas take" max(Ai,Ai+1) "at" argmax(Ai,Ai+1)
            print "Daskalakis take" min(Ai,Ai+1) "at" argmin(Ai,Ai+1)
            i=i+2
        else:
            if Aj>Ai+1 and D[i,j]==Ai+D[i+1,j-1]:
                print "Devadas take" Ai "at" i
                print "Daskalakis take" Aj "at" j
                i=i+1, j=j-1
            elif Aj<Ai+1 and Ai+D[i+2,j] ==D[i,j]:
                print "Devadas take" Ai "at" i
                print "Daskalakis take" Ai+1 "at" i+1
            i=i+2
```

flip page

SCRATCH PAPER

```
elif Ai > Aj-1 and D[i,j] == Aj + D[i+1,j-1] :
    print "Devados take" Aj " at" j
    print "Daskalakis take" Ai "at" i
    i=i+1 , j=j-1
else Ai < Aj-1 and D[i,j] == Aj + D[i,j-2]
    print "Devados take" Aj "at" j
    print "Daskalakis take" Aj-1 "at j-1
    j=j-2
```

SCRATCH PAPER