Final Exam Solutions

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

For each of the following questions, circle either T (True) or F (False). **Explain your choice.** (Your explanation is worth more than your choice of true or false.)

(a) T F For all positive $f(n)$, $f(n) + o(f(n)) = \Theta(f(n))$.

Solution: True.

(b) T F For all positive $f(n)$, $g(n)$ and $h(n)$, if $f(n) = O(g(n))$ and $f(n) = \Omega(h(n))$, then $g(n) + h(n) = \Omega(f(n))$.

Solution: True. This follows from $f(n) = O(g(n)) \Rightarrow g(n) = \Omega(f(n))$.

(c) T F Under the simple uniform hashing assumption, the probability that three specific data elements (say 1, 2 and 3) hash to the same slot (i.e., $h(1) = h(2) = h(3)$) is $1/m^3$, where m is a number of buckets.

Solution: False. The above formula only describes the probability of collision in a *fixed* bucket (say bucket number 1). The correct answer is $1/m^2$.

(d) T F Given an array of n integers, each belonging to $\{-1, 0, 1\}$, we can sort the array in $O(n)$ time in the worst case.

Solution: True. Use counting sort, e.g., after adding 1 to all numbers.

(e) T F The following array is a max heap: $[10, 3, 5, 1, 4, 2]$.

Solution: False. The element 3 is smaller than its child 4, violating the max-heap property.

(f) T F RADIX SORT does not work correctly (i.e., does not produce the correct output) if we sort each individual digit using INSERTION SORT instead of COUNTING SORT.

Solution: False. INSERTION SORT (as presented in class) is a stable sort, so RADIX SORT remains correct. The change can worsen running time, though.

3 points for correct answer and explanation
1 point for incorrect answer, but mentioning that radix sort needs to use a stable sort

(g) T F Given a directed graph G, consider forming a graph G' as follows. Each vertex $u' \in G'$ represents a strongly connected component (SCC) of G. There is an edge (u', v') in G' if there is an edge in G from the SCC corresponding to u' to the SCC corresponding to v'.

Then G' is a directed acyclic graph.

Solution: True. If there were any cycles in the graph of strongly connected components, then all the components on the cycle would actually be one strongly connected component.

3 points for correct answer

(h) T F Consider two positively weighted graphs $G = (V, E, w)$ and $G' = (V, E, w')$ with the same vertices V and edges E such that, for any edge $e \in E$, we have $w'(e) = w(e)^2$.

For any two vertices $u, v \in V$, any shortest path between u and v in G' is also a shortest path in G.

Solution: False. Assume we have two paths in G, one with weights 2 and 2 and another one with weight 3. The first one is shorter in G' while the second one is shorter in G.

3 points for correct answer - most people got this right

(i) T F An optimal solution to a knapsack problem will always contain the object i with the greatest value-to-cost ratio v_i/c_i.

Solution: False. Greedy choice doesn't work for the knapsack problem. For example, if the maximum cost is 2, and there are two items, the first with cost 1 and value 2, and the second with cost 2 and value 3, the optimal solution is to take just the second item.

3 points for correct answer

(j) T F Every problem in NP can be solved in exponential time.

Solution: True.

3 points for correct answer

Problem 2. Short answer [40 points] (8 parts)

(a) Rank the following functions by increasing order of growth; that is, find an arrangement g_1, g_2, g_3, g_4 of the functions satisfying $g_1 = O(g_2)$, $g_2 = O(g_3)$, $g_3 = O(g_4)$. (For example, the correct ordering of n^2, n^4, n, n^3 is n, n^2, n^3, n^4.)

$$f_1 = (n!)^{1/n} \qquad f_2 = \log n^n \qquad f_3 = n^{n^{1/2}} \qquad f_4 = n \log n \log\log n$$

Solution: The correct order if f_1, f_2, f_4, f_3

(b) Solve the following recurrences by giving tight Θ-notation bounds. You do not need to justify your answers, but any justification that you provide will help when assigning partial credit.

i. $T(n) = 4T(n/2) + n^2 \log n$

ii. $T(n) = 8T(n/2) + n \log n$

iii. $T(n) = \sqrt{6006} \cdot T(n/2) + n^{\sqrt{6006}}$

Solution:

i. Case 2 of the Master Method - $T(n) = n^2 \log^2 n$.

ii. Case 1 of the Master Method - $T(n) = n^3$.

iii. Case 3 of the Master Method to obtain $T(n) = \Theta(n^{\sqrt{6006}})$, checking that the regularity condition $af(n/2) = \sqrt{6006}\frac{n}{2}^{\sqrt{6006}} \leq cn^{\sqrt{6006}}$ for some $c < 1$. Condition holds for any $c > \sqrt{6006}/2^{\sqrt{6006}}$.

(c) Give a recurrence $T(n) = \cdots$ for the running time of each of the following algorithms, along with the asymptotic solution to that recurrence:

i. Insertion sort
ii. Merge sort
iii. 2D peak finding (the fastest algorithm we've seen)

Solution:

i. $T(n) = T(n-1) + O(n) = O(n^2)$
ii. $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$
iii. $T(n) = T(n/2) + \Theta(n) = \Theta(n)$

(d) Could a binary search tree be built using $o(n \lg n)$ comparisons in the comparison model? Explain why or why not.

Solution: No, or else we could sort in $o(n \lg n)$ time by building a BST in $o(n \lg n)$ time and then doing an in-order tree walk in $O(n)$ time.

(e) Given n integers in the range $0 \ldots k$, describe how to preprocess these integers into a data structure that can answer the following query in $O(1)$ time: given two integers a and b, how many integers fall within the range $a \ldots b$?

Solution: same preprocessing as for counting sort, then numbers in range is $C'(b) - C'(a)$.

(f) Describe how any comparison-based sorting algorithm can be made stable, without affecting the running time by more than a constant factor.

Solution: Tag elements with their original positions in the array, only increase by a factor of 2 at most

(g) In dynamic programming, we derive a recurrence relation for the solution to one subproblem in terms of solutions to other subproblems. To turn this relation into a bottom-up dynamic programming algorithm, we need an order to fill in the solution cells in a table, such that all needed subproblems are solved before solving a subproblem. For each of the following relations, give such a valid traversal order, or if no traversal order is possible for the given relation, briefly justify why.

i. $A(i,j) = F(A(i,j-1), A(i-1,j-1), A(i-1,j+1))$
ii. $A(i,j) = F(A(\min\{i,j\}-1, \min\{i,j\}-1), A(\max\{i,j\}-1, \max\{i,j\}-1))$
iii. $A(i,j) = F(A(i-2,j-2), A(i+2,j+2))$

Solution:

i. Solve $A(i,j)$ for(i from 0 to n: for(j from 0 to n))
ii. Solve $A(k,k)$ for(k from 0 to n) then solve rest in any order
iii. Impossible: cyclic.

(h) Consider an array $A[1\ldots n]$ of integers in the range $1\ldots n^2$. A number a is a ***heavy hitter*** in A if a occurs in A at least $n/2$ times.
Give an efficient algorithm that finds all heavy hitters in a given array A.

Solution: Radix-sort and linear scan.

Problem 3. You are the computer [15 points] (3 parts)

(a) Fill in the following grid with the correct subproblem solutions for this sequence alignment problem with these weights: 0 for mutation, 1 for insertion or deletion, and 3 for a match (the goal is to maximize the sum of the weights). Here "ATC" is the starting sequence and "TCAG" is the ending sequence.

–	–	A	T	C
–	0			
T				
C				
A				
G				

What is the optimal alignment?

Solution: The optimal alignment is to match the "TC" in both sequences.

-	-	A	T	C
-	0	1	2	3
T	1	2	4	5
C	2	3	5	7
A	3	5	6	8
G	4	6	7	9

3 points for correctly filled grid, 2 points for correct alignment of sequences

(b) Draw a max-heap on the following set of integers: $\{2, 3, 5, 7, 11, 13, 17\}$. (You do not need to use the Build-Heap algorithm.)

Solution: multiple possible solutions

5 points for a correct answer - (almost) everyone got this correct

(c) Compute $\sqrt[3]{6006}$ using two iterations of Newton's method, i.e., fill out the following table. Your entry for x_1 should be fully simplified. Your entry for x_2 can be left unsimplified.

i	x_i
0	1
1	
2	

Solution:

i	x_i
0	1
1	$2002.\bar{6}$
2	$2/3x_1 + \frac{2002}{x_1^2}$

5 points for a correct answer

-2 for significant arithmetic/algebraic errors
-3 for errors that led to negative or very large solutions (should have been caught by simple sanity checks)

Problem 4. Rotated array [10 points]

Consider an array $A[1 \cdots n]$ constructed by the following process: we start with n distinct elements, sort them, and then rotate the array k steps to the right. For example, we might start with the sorted array $[1, 4, 5, 9, 10]$, and rotate it right by $k = 3$ steps to get $[5, 9, 10, 1, 4]$. Give an $O(\log n)$-time algorithm that finds and returns the position of a given element x in array A, or returns None if x is not in A. Your algorithm is given the array $A[1 \cdots n]$ but does *not* know k.

Solution: Solution I (to be fixed): You can perform a modified binary search.

```
SEARCH(A, i, j, x)
    first ← A[i]
    middle ← A[(i + j)/2]
    last ← A[j]
    if x ∈ {first, middle, last}:
      then
          return the corresponding index
      else :
          if (x < first and x > middle) or (x > first and x > middle): [xxx Isn't it equivalent to just x > m
            then
                return SEARCH(A, (i + j)/2, j, x)
            else :
                return SEARCH(A, i, (i + j)/2, x)
```

Solution II: Let $k \geq 0$ be the number of elements A was rotated by. We show how to identify the value of k, after which one can simply perform binary search in the "left part" $A[1 \ldots k]$ and the "right part" $A[k + 1 \ldots n]$. To this end, observe that all elements in the left part are not smaller than $A[1]$, while all elements in the right part are smaller than $A[1]$. By binary search we find the smallest $j > 1$ such that $A[j]$ is smaller than $A[1]$ (or set $j = 1$ if no such j exists). We then report $k = j - 1$.

Problem 5. Taxachusetts [10 points]

Suppose you are given a weighted graph $G = (V, E, w)$ of highways, and the state government has implemented a new tax rule whereby the cost of a path gets doubled as penalty if the number of edges in the path is greater than 10. Explain how to reduce finding the shortest-path weight between every pair of vertices (under this penalty) to the usual all-pairs shortest paths problem (as solved by Floyd-Warshall).

Solution: Augment to keep track of path lengths between each pair. We also augment so we keep track of the shortest-path weight between each pair using fewer than 10 edges and using greater than 10 edges.

Problem 6. Does this path make me look fat? [10 points]

Consider a connected weighted directed graph $G = (V, E, w)$. Define the ***fatness*** of a path P to be the maximum weight of any edge in P. Give an efficient algorithm that, given such a graph and two vertices $u, v \in V$, finds the minimum possible fatness of a path from u to v in G.

Solution: There are two good solutions to this problem.

We can see that that fatness must be the weight of one of the edges, so we sort all edge weights and perform a binary search. To test whether or not a path with a fatness no more than x exists, we perform a breadth-first search that only walks edges with weight less than or equal to x. If we reach v, such a path exists.

If such a path exists, we recurse on the lower part of the range we are searching, to see if a tighter fatness bound also applies. If such a path does not exist, we recurse on the upper part of the range we are searching, to relax that bound. When we find two neighboring values, one of which works and one of which doesn't, we have our answer. This takes $O((V + E) \lg E)$ time.

Another good solution is to modify Dijkstra's algorithm. We use "fatness" instead of the sum of edge weights to score paths, and the only change to Dijkstra itself that is necessary is to change the relaxation operation so that it compares the destination node's existing min-fatness with the max of the weight of the incoming edge (i, j) and the min-fatness of any path to i (the source of the incoming edge).

The correctness argument is almost precisely the same as that for Dijkstra's algorithm. A correct solution also had to note that negative-weight edges, which could be present here and normally break Dijkstra, don't do that here; adding negative numbers produces ever-more-negative path weights, but taking their max doesn't. This solution has the same time complexity as Dijkstra's algorithm.

Two less-efficient solutions were to use Bellman-Ford instead of Dijkstra (with the same modified relaxation step and an analagous correctnes argument) and to perform the iterative search linearly instead of in a binary fashion. These recieved partial credit.

Problem 7. Indiana Jones and the Temple of Algorithms [10 points]

While exploring an ancient temple, Prof. Jones comes across a locked door. In front of this door are two pedestals, and n blocks each labeled with its positive integer weight. The sum of the weights of the blocks is W. In order to open the door, Prof. Jones needs to put every block on one of the two pedestals. However, if the difference in the sum of the weights of the blocks on each pedestal is too large, the door will not open.

Devise an algorithm that Prof. Jones can use to divide the blocks into two piles whose total weights are as close as possible. To avoid a boulder rolling quickly toward him, Prof. Jones needs your algorithm to run in pseudopolynomial time.

Solution: Just consider the smaller of the two piles. The goal is to make the weight of this pile as close to $W/2$ as possible, while not exceeding that weight. Our guess for each block is whether the block is included in this smaller pile or not.

Our subproblems are $P(i, w)$, which has the value "true" if it is possible to make a pile of weight $w \leq W/2$ with some subset of blocks 1 through i, and the value "false" otherwise. Initially, let $P(0, 0)$ be true, and $P(0, w)$ be false for all values of $w > 0$.

Let the weight of block i be w_i. Use the following recurrence:

$$P(i, w) = P(i-1, w) \vee P(i-1, w-w_i)$$

(where $P(i-1, w-w_i)$ is considered to be false if $w_i > w$). If $P(i, w)$ is true, record in a separate table $B(i, w)$ which of $P(i-1, w)$ or $P(i-1, w-w_i)$ was true (choose arbitrarily if they are both true).

Find the largest value of w for which $P(n, w)$ is true. Then backtrack using the table B to determine which blocks were included in this pile to achieve this weight. This algorithm runs in $O(nW)$ time, because there are nW subproblems, each of which takes constant time.

Problem 8. Claustrophobic chickens [10 points]

Prof. Tyson has noticed that the chickens in his farm frequently get claustrophobic. He wants to build a monitoring system that will alert him when this happens, allowing him to manually rearrange the chickens. After thorough research, Prof. Tyson determines that a chicken becomes claustrophobic if it is within 2 feet of at least 8 other chickens.

Prof. Tyson has installed a tracker on each chicken, which reports the (x, y) coordinates (measured in feet) of each of his chickens. Suppose that there are n chickens whose locations are represented as a sequence $(x_1, y_1), \ldots, (x_n, y_n)$. Prof. Tyson needs an efficient algorithm to determine whether there are any claustrophobic chickens (causing Prof. Tyson to go out and manually rearrange the chickens). Devise such an algorithm and save the chickens!

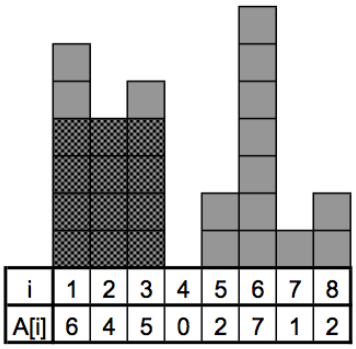
Solution: We shall solve this problem in a way similar to the close pair problem done in lecture, namely, partitioning the plane into a square grid and hashing points to corresponding buckets.

1. Impose a square grid onto the plane where each cell is a $\sqrt{2} \times \sqrt{2}$ square.
2. Hash each point into a bucket corresponding to the cell it belongs to.
3. If there is a bucket with ≥ 9 points in it, return YES.
4. Otherwise, for each point p, calculate distance to all points in the cell containing p as well as the neighboring cells. Return YES if the number of points within distance 2 is ≥ 8. Clearly the algorithm will find a clumped chicken if one exists, either in step 3 or in step 4. By using a hash table of size $O(n)$, we can make the above algorithm run in expected linear time.

Problem 9. Architects 'R Us [15 points]

You are assisting Prof. Gehry with designing the shape of a new room in the Stata Center. The professor has given you n columns, each of the same unit thickness, but with different heights: $A[1], A[2], \ldots, A[n]$. He asks you to permute the columns in a line to define the shape of the room. To make matters difficult, MIT wants to be able to hang a large rectangular picture on the columns. If j consecutive columns in your order all have a height of at least k, then we can hang a rectangle of size $j \cdot k$.

The example below contains 3 consecutive columns with heights of at least 4, so we can hang a rectangle of area 12 on the first three columns.



i	1	2	3	4	5	6	7	8
A[i]	6	4	5	0	2	7	1	2

(a) Give an efficient algorithm to find the largest area of a hangable rectangle for the *initial* order $A[1], A[2], \ldots, A[n]$ of columns.

Solution: The best algorithms we know run in $O(n^2)$ time.

The simplest $O(n^2)$ algorithm is the following. The biggest rectangle is bounded on top by some column. Guess that column i. So the height of the rectangle is $A[i]$. Now walk left until reaching a column of height $< A[i]$, and similarly walk to the right. Count the number k of traversed columns, and multiply by $A[i]$.

Another $O(n^2)$ algorithm is the following. Define $m[i, j]$ to be the minimum of the interval $A[i], \ldots, A[j]$. Then $m[i, j] = \min\{m[i, j-1], A[j]\}$, so by memoization, we can compute all $m[i, j]$'s in $O(n^2)$ time. Now the solution is $\max\{m[i, j] \cdot (j - i + 1) : i \leq j\}$, which takes $O(n^2)$ time to compute given the $m[i, j]$'s.

The easy brute-force algorithm already runs in $O(n^3)$ time (and was worth a base value of 4–5 out of 8 points for this part). Just use the computation above, but without memoizing the $m[i, j]$'s, so each takes $O(n)$ to compute, and we use $O(n^2)$ of them.

An $O(nh)$ algorithm, where $h = \max_i A[i]$, was worth a base value of 6 out of 8 points. We define one subproblem per (x, y) coordinate: $R(x, y)$ is the maximum possible area of a rectangle whose upper-right corner is at (x, y). There are $O(nh)$

such subproblems. To solve the subproblem, we can use the $O(1)$-time recurrence

$$R(x,y) = \begin{cases} R(x-1,y)+y & \text{if } A[x] \geq y \\ 0 & \text{otherwise.} \end{cases}$$

(b) Devise an efficient algorithm to permute the columns into an order that maximizes the area of a hangable rectangle.

Solution: The intended algorithm is to sort the columns in decreasing order, e.g., using merge sort in $O(n \lg n)$ time. This works because, if the correct height of a rectangle is k, then at best it can involve all columns with height $\geq k$, and these are consecutive in the sorted order. In fact, increasing order works just as well, as does a strange order (suggested by several students) of putting the maximum in the middle, then repeatedly placing the next smaller column alternately between the left and right sides of the construction so far.

We can compute the hangable rectangle in $O(n)$ additional time, though this was not necessary to receive full credit. For each prefix $B[1 \cdots i]$ of the sorted array, we'd like to compute $(\min B[1 \cdot i]) \cdot i$, and take the maximum over all i. But $\min B[1 \cdot i] = B[i]$, so this actually takes constant time per choice of i, for a total cost of $O(n)$ time.

2 out of 7 points were removed for lack of justification of sorting. 1 out of 7 points was removed for using counting (or radix) sort, which is not necessarily efficient given the setup of the problem.

Problem 10. Guess Who? [10 points]

Woody the woodcutter will cut a given log of wood, at any place you choose, for a price equal to the length of the given log. Suppose you have a log of length L, marked to be cut in n different locations labeled $1, 2, \ldots, n$. For simplicity, let indices 0 and $n + 1$ denote the left and right endpoints of the original log of length L. Let d_i denote the distance of mark i from the left end of the log, and assume that $0 = d_0 < d_1 < d_2 < \ldots < d_n < d_{n+1} = L$. The ***wood-cutting problem*** is the problem of determining the sequence of cuts to the log that will cut the log at all the marked places and minimize your total payment. Give an efficient algorithm to solve this problem.

Solution: Dynamic programming. $c(i,j) = \min_{i<k<j}\{c(i,k) + c(k,j) + (d_j - d_i)\}$ where $c(i,j)$ is the min cost of cutting a log with left endpoint i and right endpoint j at all its marked locations. Start with $c(i, i+1)$ (consecutive cuts) and move outwards to $c(i, i+2)$, $c(i, i+3)$ until the maximal distance $c(1, n)$, which gives the optimal score for the whole wood. Remember pointers to k that gave max score at each step, and trace back pointers to construct optimal solution. Each iteration takes $O(n)$ (linear search between i and j) and there are $O(n^2)$ entries to fill (a triangle really, not a square). Greedy solutions that pick the maximum cut each time do not work. Similarly, heuristics like picking the point closest to the center do not work.

Problem 11. Varying variance [20 points]

For a set S of numbers, define its ***average*** to be $\bar{S} = \frac{1}{|S|}\sum_{s\in S} s$, and its ***variance*** to be $V(S) = \frac{1}{|S|}\sum_{s\in S}(s-\bar{S})^2$.

A ***segment variance data structure*** supports the following operations on an array $A[1\ldots n]$:

- **Assignment:** given an index i and a value a, set $A[i] = a$.
- **Segment variance:** given a pair of indices i and j, compute $V(\{A[i], A[i+1], \ldots, A[j]\})$.

Initially all entries in A are set to 0.

Design a data structure that supports both operations in $O(\log n)$ time.

Solution: We have $1/|S|\sum_{s\in S}(s-\bar{S})^2 = 1/|S|\sum_{s\in S} s^2 - \bar{S}^2$. Both terms can be maintained using an augmented data structure that, in each internal node, keeps (i) the sum of all leaves and (ii) the sum of squares of all leaves.