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 180 points. Do not spend too much time on any one problem. Read them all first, and attack them in the order that allows you to make the most progress.
- **You are allowed a 3-page cheat sheet**. 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. Simply cite them.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is not required unless asked for.
- **Pay close attention to the instructions for each problem**. Depending on the problem, partial credit may be awarded for incomplete answers.

Problem	Points	Grade	Grader
1	45		
2	10		
3	10		
4	10		
5	15		
6	15		
7	15		
8	20		
9	20		
10	20		
Total	180		

Name: ______________________________

Circle your recitation:	R01,2 Alin Tomescu **10,11AM**	R03 Deepak Narayanan **12PM**	R04 Joseph Henke **12PM**	R05 Casey O'Brien **1PM**	R06,7 Ilia Lebedev **1,2PM**	R08 Andreea Bodnari **2PM**	R09 Kevin Zatloukal **3PM**	R10 Deniz Oktay **4PM**

Problem 1. Part A: Multiple Choice

[45 points] (9 parts)

(a) [5 points] Which of the following are CORRECT solutions to the given recurrences? In all cases, you may ignore roundoffs and just assume that n is a power of 2.

☐ (i) $T(1) = O(1), T(n) = 2T(n/2) + O(n)$ for $n > 1$. Solution: $T(n) = O(n^2)$

☐ (ii) $T(1) = O(1), T(n) = 8T(n/2) + O(n^2)$. Solution: $T(n) = O(n^3)$

☐ (iii) $T(1) = O(1), T(n) = 7T(n/2) + O(n^2)$. Solution: $T(n) = O(n^{\log_2 7})$

☐ (iv) $T(1) = O(1), T(n) = 8T(n/2) + O(n^3)$. Solution: $T(n) = O(n^3 \log n)$

☐ (v) $T(1) = O(1), T(n) = T(n/2) + 2T(n/4) + O(n)$. Solution: $T(n) = O(n^2)$

Solution: (ii), (iii), and (iv) are true.

Items (i)-(iv) can be determined by the Master Theorem. However, we also have examples of some: (i) is merge sort, (ii) is brute-force matrix multiplication, (iii) is Strassen's matrix multiplication.

Finally, one can check by substitution that $\frac{2}{3}Cn \log n$ is a solution to recurrence (v), where C is the constant inside the $O(n)$.

(b) [5 points] Which of the following are TRUE statements about sorting algorithms? In all cases, n is the number of elements in the input array.

☐ (i) Merge Sort takes time $\Theta(n \log n)$ in the worst case, and time $\Theta(n)$ if the input array happens to be already sorted.

☐ (ii) Insertion Sort takes time $\Theta(n^2)$ in the worst case, and time $\Theta(n)$ if the input array happens to be already sorted.

☐ (iii) Heap Sort based on a max-heap takes time $\Theta(n \log n)$ in the worst case, and time $\Theta(n)$ if the input array happens to be already sorted in decreasing order.

☐ (iv) Binary Search Tree Sort takes time $\Theta(n \log n)$ in the worst case, and time $\Theta(n)$ if the input array happens to be already sorted.

☐ (v) Radix Sort for base 10 numbers of length at most d takes time $\Theta(dn)$ in the worst case.

Solution: (ii) and (v) are true.

(i) is false since the merge step takes $\Theta(n)$ time no matter how the input is sorted (which means the whole algorithm takes $\Theta(n \log n)$ time.

(iii) is false since `extract-max` in a heap still takes $\Theta(\log n)$ time regardless of how the input is sorted, which means the whole algorithm takes $\Theta(n \log n)$ time. Note that, if the input is sorted, then it may be easier to build the heap. However, we can already do that in $\Theta(n)$ time using the `build-heap` algorithm. The expensive part of the algorithm is not building the heap but rather extracting the maximum element n times.

(iv) is false because inserting n elements in order into a binary search tree actually takes $\Theta(n^2)$ time (not $\Theta(n \log n)$ time): this is the worst case of an imbalanced tree. If we use an AVL tree instead, then it is true that it takes $\Theta(n \log n)$ time to build the tree in the worst case, but this is the correct bound even if the input is sorted: we will have to perform $\Theta(n \log n)$ work to create a balanced tree. (On the other hand, if we used a splay tree, the algorithm would indeed be $\Theta(n)$ for sorted input and $\Theta(n \log n)$ in the worst case.)

(c) [5 points] Consider an arbitrary Binary Search Tree structure augmented with a count at each node of the number of nodes in the subtree below that node. Which of the following operations can be performed in the indicated time bounds? Here, n is the number of nodes in the tree, and h is its height.

- ☐ (i) Finding the median key value, in time $O(\log n)$.
- ☐ (ii) Returning a sorted list of all the keys in the BST, in time $O(n)$.
- ☐ (iii) Finding the mean of all the keys in the tree, in time $O(h)$.
- ☐ (iv) Finding the key that is the $n/4^{th}$ largest, in time $O(h)$.
- ☐ (v) Finding all keys with values between a and b, for some arbitrary a and b, in time $O(n)$.

Solution: (ii), (iv), and (v) are true.

(i) is false because it will take $O(h)$ time to find the median and h is not necessarily $O(\log n)$. More generally, we can find the k-th largest element in $O(h)$ time performing binary search on the size of the subtree to the left, which is why (iv) is true.

(ii) and (v) are both true since we can enumerate the elements of the tree (using an inorder traversal) in $O(n)$ time. Doing this allows us to find those in $[a, b]$.

(iii) is false because computing the *mean* requires looking at every item, which clearly takes $\Omega(n)$ time.

(d) [5 points] Which of the following are guaranteed to be TRUE for an arbitrary AVL tree?

☐ (i) The median key is guaranteed to appear in the top node.

☐ (ii) The key in any node is at least as great as the keys in both of its children nodes.

☐ (iii) A list of the keys in the tree, in sorted order, can be produced in time $O(n)$.

☐ (iv) Rebalancing the tree after a single insertion can be done in time $O(1)$.

☐ (v) Given any key value k, the smallest key in the tree that is strictly larger than k can be found in time $O(\log n)$.

Solution: (iii) and (v) are true.

(i) and (ii) are just plain false. There is no reason to think (i) is true, and (ii) is actually the invariant of a max-heap not a BST.

(iv) is false because, even though we will only perform a single rotation, we may need to move all the way up to the root node to find the imbalance. Hence, this takes $O(\log n)$ time.

(v) is true since searching for k will lead us to either the successor or predecessor of k. If we end up at the predecessor, we can move to the successor in $O(\log n)$ time using the algorithm described in recitation.

(e) [5 points] Consider a hash table with chaining. The hash table has m locations numbered $0, \ldots, m-1$. Consider scenarios in which n Insert operations occur for successive, distinct keys $k_1, k_2, \ldots, k_n$, followed by a single Search operation. Which of the following are TRUE?

☐ (i) Suppose that half the Inserts happen to be for keys that hash to location 0, and the other half to location 1. The Search operation requests a key randomly and uniformly from the key space, and the resulting hash value is equally likely to be any location in the hash table. Then the expected time for the Search operation to complete is $O(n/m + 1)$.

☐ (ii) Suppose that the Inserts are for keys that hash to arbitrary locations. The Search operation is as above, in Part (i). Then the expected time for the Search operation to complete is $O(n/m + 1)$.

☐ (iii) Suppose that half the Inserts are for keys that hash to location 0, and the other half to location 1, as in Part (i) above. Now the Search operation is known to be successful, that is, it requests a key that is in the table. Suppose that it is equally likely to be any key in the table. Then the expected time for the Search operation to complete is $O(n/m + 1)$.

☐ (iv) Now suppose that the Inserts are for keys that hash to arbitrary locations. The Search operation is as in Part (iii). Then the expected time for the Search operation to complete is $O(n/m + 1)$.

☐ (v) Now suppose that the Inserts satisfy the Simple Uniform Hashing Assumption. The Search operation is as in Part (iii). Then the expected time for the Search operation to complete is $O(n/m + 1)$.

Solution: (i), (ii), and (v) are true.

(i) and (ii) are both true because the expected cost for a uniformly random hash location is the average length of the lists, which is n/m no matter how the items are distributed.

(iii) is false because the time to search the chain is now $\Theta(n)$ on average. (iv) includes (iii) as a special case (it says "arbitrary locations"), so this is also false.

(v) is true. This is the bound we proved in class.

(f) [5 points] Which of the following are TRUE about Newton's method?

- ☐ (i) Newton's method for square roots always converges for a positive initial guess.
- ☐ (ii) Newton's method for square roots always converges for a sufficiently close initial guess.
- ☐ (iii) Newton's method for reciprocal always converges.
- ☐ (iv) Newton's method for reciprocal always converges for a sufficiently close initial guess.
- ☐ (v) Newton's method for reciprocal always converges quadratically for a sufficiently close initial guess.

Solution: (i), (ii), (iv), and (v) are true.

(i) is true because, as we saw in recitation, Newton's method for square roots always converges at least fast as binary search for a positive initial guess, so in particular, it always converges for a positive initial guess. Finally, (i) implies (ii), so (ii) must also be true.

(iii) is false. We only proved that Newton's method for reciprocal converges (in fact, quadratically) for a sufficiently close guess. In fact, our error analysis implied that the iterations diverge if the guess is not sufficiently close, so (iii) is false. We proved (v) and (v) implies the (iv), so both of these are true.

(g) [5 points] Which of the following must be TRUE about the structure produced by Depth-First Search for an undirected graph G?

☐ (i) All the edges are tree edges or back edges, and there are no cross edges.

☐ (ii) Two nodes u and v that are reachable from each other in G must be in the same tree.

☐ (iii) If G is a connected graph then the DFS forest consists of a single tree.

☐ (iv) If u is the root of some tree in the DFS forest and u's removal from G disconnects G, then G has more than one child in its DFS tree.

☐ (v) If u is an internal node of some tree in the DFS forest and u's removal from G disconnects G, then no child of u in its DFS tree has a back edge to a proper ancestor of u.

Solution: (i), (ii), (iii), and (iv) are true.

(i) is true, as we showed in recitation. (The fact that there are no forward edges is by definition.)

(ii) is true because, for whichever of u or v is found first, the properties of DFS imply that v will end up as a descendent.

(iii) is implied by (ii).

(iv) was shown in the homework.

(v) is false because, as we saw in the homework, u disconnects G if and only if *every* child has a *descendent* with an edge to a proper ancestor of u.

(h) [5 points] Which of the following are TRUE statements about Topological Sorting?

☐ (i) Topological Sort sorts an array of numbers of length n in time $O(n)$.

☐ (ii) A Topological Sort algorithm sorts the nodes of an arbitrary directed graph G in an order that is consistent with all the paths in G, that is, if there is a path from u to v in G then u precedes v in the resulting sorted list.

☐ (iii) Topological Sort of a DAG can be implemented easily using Depth-First Search, in time $O(V + E)$, based on the times when the nodes in the DAG are first encountered in the DFS.

☐ (iv) Topological Sort can be used as a subroutine to find shortest paths in a weighted DAG in time $O(V + E)$; in particular, the time does not depend on the magnitudes of the weights on the edges, and the weights on the edges may be negative.

☐ (v) When Topological Sort is used to find shortest paths in a weighted DAG, no relaxation steps are necessary.

Solution: (iv) is true.

(i) is false because Topological Sort applies to graphs not arrays of numbers.

(ii) is false because Topological Sort applies to DAGs only (not arbitrary directed graphs).

(iii) is false we need to use the times at which the DFS *finishes* processing each node.

(iv) is true. We saw this algorithm in class. It only uses Topological Sort, though, to order the nodes. It still goes through them and relaxes at their edges, so (v) is false.

(i) [5 points] A *reduction* allows one problem Q to be used as a subroutine to help in solving another problem P. To measure the amount of help that Q provides, we assume that the answers to Q are free, except for any time needed to produce the subroutine's inputs and process the outputs. We analyze the remaining time for the algorithm that uses Q to solve P, and compare that to the best known time for solving P from scratch. That is, we consider the cost of solving P *relative to* Q and compare that to the cost of solving P on its own.

Which of the following represent helpful reductions, that is, they illustrate situations where Q reduces the known time needed to solve P?

☐ (i) Q is the Satisfiability problem for Boolean formulas. P is the Traveling Salesman problem.

☐ (ii) Q is the single-source Shortest Paths problem for weighted directed graphs with nonnegative weights. P is the Traveling Salesman problem.

☐ (iii) Q is the Traveling Salesman problem. P is the Satisfiability problem for Boolean formulas.

☐ (iv) Q is the problem of finding the shortest path from a node s to a node t in a connected weighted directed graph with nonnegative weights. P is the same problem but for connected weighted undirected graphs.

☐ (v) Q is the problem of producing any Binary Search Tree from a given array of integers. P is the problem of sorting the array.

Solution: (i), (iii), (iv), and (v) are true.

(i) and (iii) are true because both of these problems are NP-complete.

(iv) is true because we can create an equivalent directed graph in $O(|V| + |E|)$ time, whereas the fastest known algorithm for this problem takes $O(|V| \log |V| + |E|)$ time.

(v) is true because we can produce a sorted array in $O(n)$ time if we are given a binary search tree, whereas all known sorting algorithms take $\omega(n)$ time.

Part B: Shorter Problems [75 points]

Problem 2. [10 points] A max-priority queue is a data structure for maintaining a set P of elements, each with an associated key. Since P is a set, we assume that it contains no duplicate elements, but we do allow duplicate keys. Here we assume that the structure supports the following operations:

- `insert`(P, x, k), which inserts the element x into the set P, with key k.
- `max`(P), which returns an element x of P with the largest key. If P is empty it returns "empty".
- `extract-max`(P), which returns an element x of P with the largest key and also removes x from P. If P is empty it returns "empty".

(a) [5 points] Describe in pseudocode or English an algorithm that uses a max-priority queue to implement a FIFO queue data structure. A FIFO queue supports operations:

- `enqueue`(Q, x), which adds element x to the end of queue Q.
- `dequeue`(Q), which returns the first element of queue Q and removes that element. If Q is empty, it returns "empty".

Solution: Our queue Q will be a pair (P, k) where P is a max-priority queue, initially empty, and k is an integer, initially zero. To perform `enqueue`(Q, x), we add (x, k) to P and then decrement k. The key of (x, k) will be k. To perform `dequeue`(Q), we invoke `extract-max` on P and return its result (which may be "empty"). Since each enqueued element is given a smaller key than anything enqueued before, these will be dequeued in FIFO order.

Note that the number k is never reused, so even if x is enqueued multiple times, it will enter the max-priority queue as distinct elements (each a pair).

(b) [5 points] Describe in pseudocode or English an algorithm that uses a max-priority queue to implement a stack data structure. A stack supports operations:

- `stack-empty`(S), which returns $true$ if stack S is empty, $false$ otherwise.
- `push`(S, x), which adds element x to the top of stack S.
- `pop`(S), which returns the top element of stack S and removes that element. If S is empty, it returns "empty".

Solution: We can do this exactly as above but now we increment k instead of decrementing it for each `push` operation. Since each new element has a higher number than those already in the queue, these will be popped in LIFO order.

We have one extra operation in this case: `stack-empty`(S) is performed by calling `max` on P and returning true if `max` returns "empty".

Problem 3. [10 points] Suppose you are given a 3-gallon jug and a 5-gallon jug, and you want to end up with exactly 4 gallons in the 5-gallon jug. The jugs start out empty. What you are allowed to do is:

- Empty a jug by pouring its contents on the ground.
- Fill a jug to the rim.
- Pour the contents of one jug into the other jug until either the first jug is empty or the second one is full.

Give a *shortest* sequence of moves that will produce the needed result. Represent a "state" of the system as a pair (a, b) of numbers, where $0 \leq a \leq 3$ and $0 \leq b \leq 5$, representing the amount of water in both jugs. The start state is $(0, 0)$. Show your work.

Solution: Breadth first search discovers the nodes, starting from state $(0, 0)$, in the manner depicted in the following picture.

$$\begin{tikzcd}
 & 3,0 \arrow[r] \arrow[dr] & 0,3 \arrow[r] & 3,3 \arrow[r] & 1,5 \arrow[r] & 1,0 \arrow[r] & 0,1 \\
0,0 \arrow[ur] \arrow[dr] & & 3,5 & & & & \\
 & 0,5 \arrow[r] \arrow[ur] & 3,2 \arrow[r] & 0,2 \arrow[r] & 2,0 \arrow[r] & 2,5 \arrow[r] & 3,4
\end{tikzcd}$$

Nodes lined up vertically are equal distance from $(0, 0)$. Note that this pictures does not show any edges in the graph from later nodes back to closer ones since no shortest path could use such nodes.

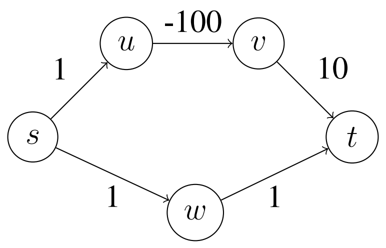
This graph shows that the distance to the solution is 6. (BFS would discover two more nodes after this, but we can stop at this point since we have found the solution, and any nodes discovered later would be further away.)

Problem 4. [10 points] Suppose we have a directed graph G with (possibly negative) integer weights on the edges. We are interested in determining the shortest path from a particular source node s to a particular target node t. If there is no path from s to t then the computed distance should be ∞, whereas if there are paths of arbitrarily large negative lengths, then the computed distance should be $-\infty$.

Here is a suggested algorithm for solving this problem. Let $-w$ be the largest negative weight on any edge of G. Construct a new graph H having the same nodes and edges as G but different weights: the weight of every edge in H is just its weight in G plus w. Thus, the weights of all the edges in H are non-negative. Then run Dijkstra's shortest paths algorithm on H to find the shortest path from s to t in H and return that path.

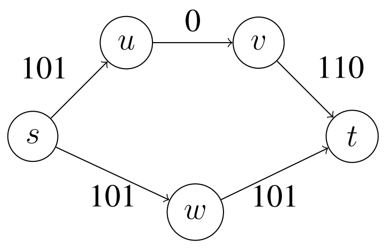
Is this algorithm correct for solving the original problem? Either give an informal correctness argument or else give a small counterexample (at most 5 nodes).

Solution: This algorithm is incorrect. To see this, consider the following example:



The top path s, u, v, t has weight $1 - 100 + 10 = -89$, while the bottom path s, w, t has weight $1 + 1 = 2$, so the top path is shorter in this graph.

However, if we add 100 to each node, then we get the graph



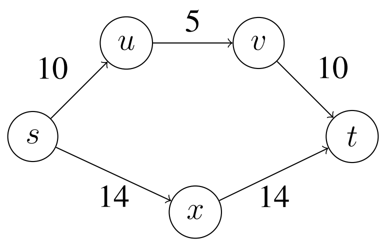
Now the weight of the top path is $101 + 0 + 110 = 211$, while the bottom path has weight $101 + 101 = 202$. Thus, in the transformed graph, the longer path has become shorter.

This reason this transformation fails is that the weight added to each path is proportional to the number of edges in the path. Hence, this can make some longer paths become shorter merely because they have fewer edges (even though those weights have more total weight).

Problem 5. [15 points] In lecture we talked about using Dijkstra to find a shortest path in a directed graph G from some node s to some other node t. The method involved alternating steps of a Dijkstra algorithm starting from s and a Dijkstra algorithm on the "reverse" graph (with edges reversed) starting from t, until the first time when a single node x appears in both sets of processed nodes, S_s and S_t.

(a) [5 points] Draw a small graph showing that we do not necessarily obtain the shortest path by simply joining the already-found shortest path from s to x and the already-found shortest path from x to t.

Solution:



We can see that x is distance 14 from both s and t, and x will be the first node to appear in $S_s \cap S_t$. At that point, we will have $S_s = \{s, u, x\}$ since u is closer to s than x and $S_t = \{t, v, x\}$ since v is closer to t than x.

We can see that the path s, x, t has weight 28. However, the shortest path is actually s, u, v, t, which has weight 25.

(b) [5 points] Explain why there must exist a shortest path from s to t that uses some edge (u, v), where u is in S_s and v is in S_t, that is, u has already been found from s and v has already been found from t. The shortest path consists of the already-found shortest path from s to u, the edge (u, v), and the already-found shortest path from v to t.

Solution: Let $s, v_1, \ldots, v_k, t$ be a shortest path. Let j be such that the weight of $s, v_1, \ldots, v_t$ is less than that of $s \rightsquigarrow x$ while $s, v_1, \ldots, v_t, v_{t+1}$ is at least that of $s \rightsquigarrow x$. (There must be some such j or else $t \in S_s$!) We can see that v_{t+1} must be closer to t (in the reverse graph) that x since otherwise the weight of the path from s to t via x

would be less than the weight of the path via v_{t+1}, which we assumed to be a shortest path. Since v_{t+1} is closer to t than x, we must have $v_{t+1} \in S_t$. Thus, we have $v_t \in S_s$, $v_{t+1} \in S_t$, and an edge (v_t, v_{t+1}) between them.

(c) [5 points] Describe a simple algorithm for determining the shortest path from s to t, given that we have already executed Dijkstra from both ends and found the first common node x. Your algorithm should have a reasonable time complexity no worse than $O(E)$.

Solution: Go through each $v \in V$ and compute $d_s[v] + d_t[v]$, where d_* is Dijkstra's computed distances from s and t, respectively, and return the smallest value computed. As we saw in the previous answer, the node v_{t+1} is adjacent to a node v_t that is in S_s. Hence, Dijkstra will have already relaxed at (v_t, v_{t+1}), which means that $d_s[v_{t+1}]$ will be correct. Also, since we have $v_{t+1} \in S_t$, we know that $d_t[v_{t+1}]$ is correct. Hence, $d_s[v_{t+1}] + d_t[v_{t+1}]$ is the true shortest path distance. And since the d values are always lower bounds, this will be no larger than the same sum for any other node, so the return value will be this sum, which is the true shortest path weight.

Problem 6. [15 points] This problem involves designing a dynamic programming algorithm that takes as inputs:

- A weighted directed graph G, with weights that are non-negative integers.
- A source node s and a target node t.
- A positive integer k, representing the number of edges in a path.

The algorithm is supposed to produce the minimum cost of a path from s to t in G that consists of exactly k edges. Here, the cost of a path means the sum of the weights of the edges along the path.

Assume that you are given the graph G in adjacency-list format, as usual.

(a) [4 points] Define subproblems that you can use to solve this problem using dynamic programming. There should be $O(kV)$ subproblems.

Solution: We will have a subproblem $C(v, \ell)$ for each vertex $v \in V$ and length $\ell \leq k$. This will return the minimum cost of a path from s to v using exactly ℓ edges.

(b) [4 points] Give a system of recursion equations that can form the basis of an efficient dynamic programming algorithm for solving this problem. Include a formula for the final answer to the problem.

Solution: We have $C(v, 0) = 0$ if $v = s$ and ∞ otherwise. For $\ell > 0$, we have

$$C(v, \ell) = \min_{(u,v) \in E} C(u, \ell - 1) + w_{u,v}$$

The answer to the original problem is $C(t, k)$.

(c) [4 points] What is the time complexity for solving a single subproblem, when given solutions to its subproblems for free?

Solution: The time is $\Theta(1)$ times the in-degree of the node. Since we can trivially bound the in-degree by $|V|$, we get a bound of $O(|V|)$.

(d) [3 points] What is the overall time complexity of a dynamic programming algorithm based on your recursion equations?

Solution: By the previous two parts, the total time is $O(k|V|^2)$. However, we can analyze this more carefully by summing the work at each node, which is the in-degree of that node. The sum of all the in-degrees is E, so a more accurate bound is $\Theta(k|E|)$.

Problem 7. [15 points] Suppose you are given a set of n rectangular three-dimensional blocks, where block B_i has length l_i, width w_i, and height h_i, all real numbers. You are supposed to determine the maximum height of a tower of blocks that is as tall as possible, using any subset of the blocks.

There are two constraints:

1. You are not allowed to rotate the blocks: the length always refers to the east-west direction, the width is always north-south, and the height is always up-down.
2. You are only allowed to stack block B_i on top of block B_j if $l_i \leq l_j$ and $w_i \leq w_j$, that is, the two dimensions of the base of block B_i are no greater than those of block B_j.

(a) [5 points] Define subproblems that you can use to solve this problem using dynamic programming. There should be $O(n)$ subproblems.

Solution: The key insight is to notice that, if $l_i \leq l_j$ and $w_i \leq w_j$, then we certainly have $l_i w_i \leq l_j w_j$. In other words, B_i can only be blocked on B_j if B_i has smaller *area* than B_j. (This is necessary but not sufficient, of course.) Hence, we will start by sorting the list of blocks by their area. In the rest of the problem we will assume they are sorted this way.

We define a subproblem $H(i)$ to be the height of the tallest tower, using only blocks $B_1, \ldots, B_i$, that has B_i on top. We can see that there are n subproblems.

(b) [5 points] Give a system of recursion equations that can form the basis of an $O(n^2)$ dynamic programming algorithm for solving this problem. Include a formula for the final answer to the problem.

Solution: We can see that

$$H(i) = \left(\max_{j \leq i : B_j \preceq B_i} H(j) \right) + h_i$$

where $B_j \preceq B_i$ means that B_i can be stacked on B_j (i.e., $l_i \leq l_j$ and $w_i \leq w_j$).

The answer to the problem is $\max_i H(i)$ since some B_i is on top of the tallest stack.

(c) [5 points] What is the overall time complexity of a dynamic programming algorithm based on your recursion equations?

Solution: It is easy to see that the recurrence can be computed in $O(n)$ time, and since there are n subproblems, this gives a total time bound of $O(n^2)$.

Part C: Longer Problems [60 points]

Problem 8. [20 points] This problem involves implementing a hash table T that supports $insert$, $delete$, and $search$ operations, using an open addressing strategy. The hash function is $h : U \times \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}$, mapping the universe U of possible keys and an index to a location in the table. An entry in the table may be either a key in U, the "empty" indicator nil, or the "deleted" indicator del.

To make this combination of operations work correctly, your algorithm should preserve the following invariant:

"For any key k and index i, if k appears in the table at location $h(k, i)$, then there is no $i' < i$ such that location $h(k, i')$ contains nil."

(a) [6 points] Complete the following pseudocode for the $search$ operation by filling in the four blanks, and use the invariant to explain very briefly why this is guaranteed to find the key if it is in the table. (You may fill in the blanks with any usual programming statements, including skip, return, etc.)

```
Search(T,k):
for i = 0,...,m-1 do
  j := h(k,i)
  if T[j] = k then return j
  if T[j] = some other key then __________
  if T[j] = nil then __________
  if T[j] = del then __________
return __________
```

Solution:

```
Search(T,k):
for i = 0,...,m-1 do
  j := h(k,i)
  if T[j] = k then return j
  if T[j] = some other key then skip
  if T[j] = nil then return "not here"
  if T[j] = del then skip
return "not here"
```

We know this will find the key because we keep executing the loop unless we see "nil", and in that case the invariant tells us that k is not in the table.

(b) [7 points] Complete the following pseudocode for the *insert* operation by filling in the blanks, and explain briefly why this operation preserves the invariant. Assume that, when this operation is called for a key k, k is not already in the table.

```
Insert(T,k)
for i = 0,...,m-1 do
  j := h(k,i)
  if T[j] = some key other than k then ___________
  if T[j] = nil then ___________
  if T[j] = del then ___________
return
```

Solution:

```
Insert(T,k)
for i = 0,...,m-1 do
  j := h(k,i)
  if T[j] = some key other than k then skip
  if T[j] = nil then T[j] := k; return;
  if T[j] = del then T[j] := k; return;
return
```

We know this preserves the invariant because:

1. For the key k itself: we have only skipped over slots that are occupied by keys, no nil slots.

2. For any other key k': the invariant was true before this insertion and we don't falsify it, because we don't change any location's contents to nil.

(c) [7 points] Complete the following pseudocode for the *delete* operation by filling in the blanks, and explain briefly why this operation preserves the invariant. Assume that, when this operation is called for a key k, k is in the table.

```
Delete(T,k)
for i = 0,...,m-1 do
  j := h(k,i)
  if T[j] = k then ___________
  if T[j] = some key other than k then ___________
  if T[j] = nil then ___________
  if T[j] = del then ___________
return
```

Solution:

```
Delete(T,k)
for i = 0,...,m-1 do
  j := h(k,i)
  if T[j] = k then T[j] = del; return
  if T[j] = some key other than k then skip
  if T[j] = nil then return
  if T[j] = del then skip
return
```

We know this preserves the invariant because:

1. For k itself: It's not in the table after the operation, so the claim is vacuously true.
2. For another k': As for part (b), we don't change any location's contents to nil.

Problem 9. [20 points]

Consider the following system of difference constraints involving the three variables x_1, x_2, and x_3:

$$\begin{aligned}
x_1 &\le x_2 + 4\\
x_2 &\le x_1 - 3\\
x_1 &\le x_3 + 2\\
x_3 &\le x_1 - 1\\
x_2 &\le x_3 + 2\\
x_3 &\le x_2 + 1
\end{aligned}$$

From this, we can derive a corresponding *constraint graph*. This is a weighted directed graph. Its nodes correspond to the variables, plus a new source node x_0. Node x_0 has an edge with weight 0 to all the other nodes. The other edges are derived directly from the inequalities: an inequality of the form $x_i \le x_j + c$ is represented by an edge from node x_j to node x_i with weight c.

(a) [5 points] Use Bellman-Ford on the constraint graph to derive a solution to the original system of inequalities, based on the distances computed by Bellman-Ford. You need only give the final answers.

x_1 = ________________
x_2 = ________________
x_3 = ________________

Solution:

$x_1 = 0$

$x_2 = -3$

$x_3 = -2$

(b) [5 points] Explain why the following general statement is true: If the constraint graph derived from a system of difference constraints has no negative-weight cycle, then Bellman-Ford yields a correct solution to the original system of inequalities.

Solution: When BF is applied to a graph with no negative-weight cycles, it converges after the first (relaxation) phase. At that point, the final distance estimates $d[]$ satisfy the following stability property: For any edge (i, j) from node i to j, $d[j] \le d[i] + weight(i, j)$. This corresponds to the constraint $x_j \le x_i + c$, associated with the edge.

(c) [5 points] Now consider the same constraints as above, except that the first inequality $x_1 \leq x_2 + 4$ is replaced by $x_1 \leq x_2 + 2$. What happens when you run Bellman-Ford on the resulting constraint graph? Why does this happen?

Solution: After running Bellman Ford we obtain distances $(-\infty, -\infty, -\infty)$ to the three nodes. This is because we have a negative-weight cycle that is accessible from x_0 and from which all other nodes are reachable.

(d) [5 points] Explain why the following general statement is true: If the constraint graph derived from a system of difference constraints has a negative-weight cycle, the system has no solution. To be definite, suppose that the constraint graph has a negative-weight cycle $x_1, x_2, ..., x_n, x_1$ with weights $w_1, ..., w_n$ on the successive edges.

Solution: The corresponding inequalities are:

$x_2 \leq x_1 + w_1$

$x_3 \leq x_2 + w_2$

...
$x_n \leq x_{n-1} + w_{n-1}$
$x_1 \leq x_n + w_n$
If these are all satisfied, then we can add them and get that $0 \leq w_1 + ... + w_n$, which implies that the cycle has nonnegative weight, contradiction.

Problem 10. (20 points)

Scheduling using Dynamic Programming: Suppose that we are given, as input, a set J of *job types* for jobs that can run on a machine. Each job type j has an associated positive integer duration $j.duration$ and an associated positive integer value $j.value$, which is the amount of money that the owners of the machine can charge for running a job of that type. We are also given a positive integer T representing the total amount of time available on the machine.

Assume that there are an unlimited number of jobs of each type. We would like a dynamic programming algorithm to determine the maximum total value for a sequence of jobs that can be run on the machine in a given time.

(a) [4 points] As an example, suppose that the input is $J = \{j_1, j_2, j_3\}$, where

$$\begin{aligned} j_1.duration &= 5 \quad , j_1.value = 19, \\ j_2.duration &= 3 \quad , j_2.value = 9, \\ j_3.duration &= 2 \quad , j_3.value = 5, \end{aligned}$$

and $T = 21$. Then what is the maximum achievable value?

Solution:

Max achievable value is 76, obtained with 4 jobs of type j_1.

(b) [4 points] Define the subproblems for a dynamic programming solution to the general problem (for arbitrary sets J and arbitrary T).

Solution:

For any positive integer t, let $C(t)$ represent the maximum total value achievable by running jobs of the given types within time bound t.

(c) [4 points] Give recursion equations expressing the maximum value that is achievable within a given time T.

Solution:

Base case: $C(0) = 0$

$C(t) = MAX(C(t-1), C(t - j.duration) + j.value)$ over all $j \in J$ such that $j.duration \leq t$

(d) [4 points] Give pseudocode for a dynamic programming solution based on your recursion equations in Part (b).

Solution:

```
def solve(J, T):
  C[0] = 0
  for t = 1 to T:
    C[t] = C[t-1]
    for j in J:
      if j.duration <= t:
        C[j] = max(C[j], C[j - j.duration] + j.value)
  return C[T]
```

(e) [4 points] Analyze the time complexity of your solution.

Solution:

$O(TN)$, where $N = |J|$. That's because we have T different subproblems, and solving each in terms of the previous ones uses time $O(N)$.