CS161: Design and Analysis of Algorithms **Summer 2004**

Problem Set #4 Solutions

General Notes

- **Regrade Policy:** If you believe an error has been made in the grading of your problem set, you may resubmit it for a regrade. If the error consists of more than an error in addition of points, please include with your problem set a detailed explanation of which problems you think you deserve more points on and why. We reserve the right to regrade your entire problem set, so your final grade may either increase or decrease.
- On problem 1(f), many people assumed that if the $key[y] < key[x]$ then y was in the left subtree of x or that if $priority[y] < priority[x]$ then y was a child of x. These stipulations are not true, since in either case you can also have y not be a descendant of x and the condition may still hold. Also, many people did not prove both the if and the only if parts of the statement.
- For augmented red-black trees in general, if the augmentation does not satisfy Theorem 14.1, you should show that it can be maintained efficiently through insertion, deletion, *and rotations.*
- On problem 5(b), a common mistake was to sort by the x indices, but to not take into account the order for equal x index. For equal x_i's, you need to sort by decreasing index y_j so that you can match at most one y_j to each x_i.

1. [**34 points**] Treaps

   Throughout this problem, please be as formal as possible in your arguments whenever asked to show or conclude some result.

   (a) [**4 points**] Do problem 13-4(a) on page 297 of CLRS.

   Answer: Our proof is inductive and constructive. The induction is on the number of elements in the treap. The base case of 0 nodes is the empty treap. We now suppose that the claim is true for $0 \leq k < n$ nodes, and show it is true for n. Given a set of n nodes with distinct keys and priorities, the root is uniquely determined - it must be the node with the smallest priority in order for the min-heap property to hold. Now suppose this root node has key r. All elements with key $< r$ must be in the left subtree, and all elements with key $> r$ must be in the right subtree. Notice that each of these sets has at most $n-1$ keys, and therefore has a unique treap by the inductive assumption. Thus, for the n-node treap, the root is uniquely determined, and so are its subtrees. Therefore, the treap exists and is unique.

   Notice this suggests an algorithm for finding the unique treap: choose the element with the smallest priority, partition the remaining elements around its key, and recursively build the unique treaps from the elements smaller than the root key

(these will go in the left subtree) and the elements larger than the root key (right subtree).

(b) [**2 points**] Do problem 13-4(b) on page 298 of CLRS.

Answer: Recall from lecture and section 12.4 of CLRS that the expected height of a randomly built binary search tree is $O(\lg n)$. We show that constructing the unique treap is equivalent to inserting the elements into a binary search tree in random order.

Suppose we insert the elements into a BST in order of increasing priority. Normal BST insertion will maintain the binary search tree property, so we only need to show that the min-heap property is maintained on the priorities. Suppose v is a child of u. The insertion procedure from section 12.3 of CLRS always inserts each element at the bottom of the tree, therefore v must have been inserted after u. Thus, the priority of v is greater than the priority of u, and we have built a treap.

The priorites are assigned randomly, which means any permutation of the priorities is equally likely. When considering this as BST insertion, it translates into any ordering being equally likely, and therefore the expected height of a treap is equal to the expected height of a randomly built BST, which is $O(\lg n)$.

(c) [**6 points**] Do problem 13-4(c) on page 298 of CLRS.

Answer: Treap-Insert works by first inserting the node according to its key and the normal BST insertion procedure. This is guaranteed to maintain the first two conditions of the treap, since those correspond to BST properties. However, we may have violated the third condition, the min-heap ordering on the priorities. Note that (initially) this violation is localized to the node we're inserting and its parent, since we always insert at the leaves. Rotations preserve BST properties, so we will use rotations to move our node x up as long as the min-heap ordering is violated, i.e. as long as $priority[x] < priority[parent[x]]$. If x is a left child we will right-rotate it, and if it's a right child we will left-rotate it. Notice that this preserves the heap property elsewhere in the treap, assuming that the children of x had higher priority than $parent[x]$ prior to the rotation (to be completely formal we would have to prove this using a loop invariant).

The pseudocode is as follows:

```
TREAP-INSERT(T, x, priority)
1   TREE-INSERT(T, x)
2   while parent[x] ≠ NIL ∧ priority[x] < priority[parent[x]]
3      if left[parent[x]] == x
4         Right-Rotate(T, parent[x])
5      else
6         Left-Rotate(T, parent[x])
```

(d) [**2 points**] Do problem 13-4(d) on page 298 of CLRS.

Answer: TREAP-INSERT first performs a BST insert procedure which runs in time proportional to the height of the treap. Then, it rotates the node up one level until the min-heap property is satisfied. Thus, the number of rotations we perform is bounded by the height of the treap. Each rotation takes constant

time, so the total running time is proportional to the height of the treap, which we showed in (b) to be expected $\Theta(\lg n)$.

(e) [**4 points**] Do problem 13-4(e) on page 298 of CLRS.

Answer: We will prove this using a loop invariant: after having performed k rotations during the insertion of x, $C_k + D_k = k$.
Initialization: After we insert x using BST-INSERT but before performing any rotations, x is a leaf and its subtrees are empty, so $C_0 = D_0 = 0$.
Maintenance: Assume that we have performed k rotations on x and that $C_k+D_k = k$. Now, if we perform a right-rotation on x, the left child of x remains the same, so $C_{k+1} = C_k$. The right child of x changes from β to y with subtrees β (left) and γ (right) using the notation from page 278. The left spine of the right child used to be the left spine of β and now it is y plus the left spine of β. Therefore, $D_{k+1} = D_k + 1$. Thus, $C_{k+1} + D_{k+1} = C_k + D_k + 1 = k + 1$, which is precisely the number of rotations performed. The same holds for left-rotations, where we can show that $D_{k+1} = D_k$ and $C_{k+1} = C_k + 1$.
Termination: After TREAP-INSERT is finished, the number of rotations we performed is equal to $C + D$, precisely the condition that needed to be shown.

(f) [**5 points**] Do problem 13-4(f) on page 298 of CLRS.

Answer: First, assume $X_{i,k} = 1$. We will prove $priority[y] > priority[x]$, $key[y] < key[x]$, and $\forall z$ such that $key[y] < key[z] < key[x]$, we have $priority[y] < priority[z]$. The first property follows from the min-heap property on priorities, since y is a descendant of x. The second follows from the BST property, since y is in the left subtree of x. Finally, consider any node z satisfying $key[y] < key[z] < key[x]$, and imagine an inorder tree walk on the treap. After y is printed, we will print the right subtree of y, at which point the entire left subtree of x will have been printed since y is in the right spine of that subtree. Thus, the only nodes printed after y but before x are in the right subtree of y; therefore, z must be in the right subtree of y and by the min-heap property $priority[y] < priority[z]$.

Now, we will assume the three properties and prove that $X_{i,k} = 1$. First consider the possibility that y is in the left subtree of x but not in the right spine of that subtree. Then there exists some node z in the spine such that going left from z will lead to y. Note that this z satisfies $key[y] < key[z] < key[x]$, but $priority[z] < priority[y]$ which violates the third property. Clearly y cannot be in the right subtree of x without violating the second property, and x cannot be a descendant of y without violating the first property. Suppose that x and y are not descendants of each other, and let z be their common ancestor. Again we have $key[y] < key[z] < key[x]$ but $priority[z] < priority[y]$, violating the third property. The only remaining possibility is that y is in the right spine of the left subtree of x, i.e. $X_{i,k} = 1$.

(g) [**4 points**] Do problem 13-4(g) on page 300 of CLRS.

Answer: Assume that $k > i$. $Pr\{X_{i,k} = 1\}$ is the probability that all the conditions in part (f) hold. Consider all the elements with keys $\{i, i + 1, \ldots, k\}$.

There are $k-i+1$ such elements. The values of their priorities can be in any order, so there are $(k-i+1)!$ possible (and equally likely) permutations. In order to satisfy our conditions, we need $priority[z] > priority[y] > priority[x]$ for all $z \in \{i+1, i+2, \ldots, k-1\}$. This fixes the priorities of x and y to be the lowest two priorities, allowing $(k-i-1)!$ permutations of the remaining priorities among the elements with keys in $\{i+1, \ldots, k-1\}$. The probability of $X_{i,k} = 1$ is the ratio of these two, which gives $Pr\{X_{i,k} = 1\} = \frac{(k-i-1)!}{(k-i+1)!}$. Most of the terms in the factorials will cancel, except for the first two terms in the denominator. This leaves us with $Pr\{X_{i,k} = 1\} = \frac{1}{(k-i+1)(k-i)}$.

(h) [**3 points**] Do problem 13-4(h) on page 300 of CLRS.

Answer: The expected value for C, which is the number of elements in the right spine of the left subtree of x is simply the expectation of the sum of $X_{i,k}$ over all $i < k$. This is

$$
\begin{aligned}
E[C] &= E[\sum_{i=1}^{k-1} X_{i,k}] \\
&= \sum_{i=1}^{k-1} E[X_{i,k}] \\
&= \sum_{i=1}^{k-1} Pr[X_{i,k} = 1] \\
&= \sum_{i=1}^{k-1} \frac{1}{(k-i+1)(k-i)} \\
&= \frac{1}{(k)(k-1)} + \frac{1}{(k-1)(k-2)} + \cdots + \frac{1}{(2)(1)} \\
&= \sum_{j=1}^{k-1} \frac{1}{(j+1)(j)} \\
&= \sum_{j=1}^{k-1} \left(\frac{1}{j} - \frac{1}{j+1} \right) \\
&= \frac{1}{1} - \frac{1}{k} \\
&= 1 - \frac{1}{k}
\end{aligned}
$$

(i) [**2 points**] Do problem 13-4(i) on page 300 of CLRS.

Answer: Note that the expected length of the right spine of the left subtree of x depends only on the rank k of the element x. The expected length of the left spine of the right subtree will have the same expected value with respect to the rank of x in the reverse ordering, which is $n-k+1$. Thus, $E[D] = 1 - \frac{1}{n-k+1}$.

(j) [**2 points**] Do problem 13-4(j) on page 300 of CLRS.

Answer: Since the number of rotations equals $C + D$, the expected number of rotations equals $E[C + D] = E[C] + E[D] = 1 - \frac{1}{k} + 1 - \frac{1}{n-k+1} \leq 2$. So, the expected number of rotations is less than 2.

2. [**32 points**] Augmenting Data Structures

Throughout this problem, your explanations should be as complete as possible. In particular, if you use an augmented data structure that is not presented in the textbook, you should explain how that data structure can be efficiently maintained, explain how new operations can be implemented, and analyze the running time of those operations. If the problem asks you to give an algorithm, you do not need to give pseudocode or formally prove its correctness, but your explanation should be detailed and complete, and should include a running time analysis.

(a) [**9 points**] Do problem 14.1-8 on page 308 of CLRS.

Answer: To solve this problem, we will first define some kind of coordinate system. This can be done by picking an arbitrary nonendpoint starting point on the circle and setting it to zero and defining clockwise as positive values (for example, number of degrees from the starting point). We can label each chord's endpoints as S_i, E_i where the start endpoint has lower value. We then notice that to count the chords which intersect a particular chord i, we can count the chords which have only one endpoint between S_i and E_i. Since we'll then double-count the number of intersections, we'll just count those intersections that are in the *forward* direction, namely the chord has its start endpoint in S_i, E_i, but not its other endpoint. The chords which have their end endpoints in the interval will count i as intersecting them.

We can then use an order-statistic red-black tree to help us count the number of intersections. We will first sort the endpoints according to our coordinate system. Since there are n chords each with 2 endpoints, this step will take $\Theta(2n \lg 2n) = \Theta(n \lg n)$. We then process the endpoints according to their sorted values, starting with the smallest. If it is a start endpoint, we insert it into the order-statistic tree ($\Theta(\lg n)$). If it is an end endpoint, we want to count the number of start endpoints we have seen without matching end endpoints. If we delete start endpoints as we see their corresponding end endpoints, we will maintain the property that all the nodes in our tree are *open* chords, ones which we can still intersect. So, for each end endpoint, we delete the corresponding start endpoint and then count the number of open endpoints left in the tree - this will be precisely the number of chords that have their starting endpoint between S_i and E_i but have their end endpoint after E_i. Thus, we add this number to a counter and delete the start endpoint of that chord (each end endpoint can have a pointer to its start). This step takes $\Theta(\lg n)$ for the deletion and $\Theta(1)$ to figure out the number of forward intersections for the chord. We proceed like this through all the endpoints. Since each endpoint takes time $\Theta(\lg n)$ to insert or delete from the tree, the total running time of this step is $(\Theta(n \lg n)$.

After processing all endpoints, we will have a count of the number of intersections inside the circle. The total running time is the time to sort plus the time to insert and delete from the OS-tree, both of which are $\Theta(n \lg n)$. Therefore, the total running time is $\Theta(n \lg n)$.

(b) [**4 points**] Do problem 14.2-1 on page 310 of CLRS.

Answer: We can find the SUCCESSOR and PREDECESSOR of any node x in time $\Theta(1)$ by storing pointers to the successor and predecessor of x inside the node structure. Note that these values do not change during rotations, so we only need to show that we can maintain them during insertions and deletions.

As soon as we insert a node x, we can call the regular TREE-SUCCESSOR(x) and TREE-PREDECESSOR(x) functions (that run in $O(\lg n)$) to set its fields. Once the successor and predecessor are determined, we can set the appropriate fields in x, and then update the SUCC field of the predecessor of x and the PRED field of the successor of x to point to x. When we delete a node, we update its successor and predecessor to point to each other rather than to x. To simplify the handling of border cases (e.g. calling SUCCESSOR on the largest element in the tree), it may be useful to keep a dummy node with key $-\infty$ and a dummy node with key $+\infty$.

Finally, MINIMUM and MAXIMUM (which are global properties of the tree) may be maintained directly. When we insert a node, we check if its key is smaller than the current minimum or larger than the current maximum, and update the MINIMUM and/or MAXIMUM pointer if necessary. When we delete a node, we may need to set MINIMUM and/or MAXIMUM to its successor and/or predecessor, respectively, if we are deleting the minimum or maximum element in the tree.

Thus, these operations run in $\Theta(1)$ and may be maintained without changing the asymptotic running time of other tree operations.

(c) [**5 points**] Do problem 14.2-5 on page 311 of CLRS.

Answer: Using our answer from the previous part where we showed how to maintain SUCCESSOR and PREDECESSOR pointers with $O(1)$ running time, we can easily perform RB-ENUMERATE.

```
RB-ENUMERATE(x, a, b)
1   start ← TREE-SEARCH(x, a)
2   if start < a
3      then start ← SUCC[start]
4   while start ≤ b
5      print start
6      start ← SUCC[start]
```

We must modify TREE-SEARCH to return the smallest element greater than or equal to a rather than NIL if a is not in the tree.

This algorithm finds and outputs all the elements between a and b in x. The

TREE-SEARCH operation takes time $\lg n$ where n is the number of nodes in the tree. The while loop runs m times where m is the number of keys which are output. Each iteration takes constant time since finding the SUCCESSOR of a node now takes constant time. Therefore, the total running time of the algorithm is $\Theta(m + \lg n)$.

We can also answer this question without resorting to augmenting the data structure: we perform an inorder tree walk, except we ignore (do not recurse on) the left children of nodes with key $< a$ and the right children of nodes with key $> b$. Thus, if we consider both children of a node we will output at least one of them, therefore, all nodes not counted by m will lie on two paths from the root to a leaf in the tree. This yields a total running time of $\Theta(m + \lg n)$ as before.

(d) **[6 points]** Do problem 14.3-6 on page 317 of CLRS. You may assume that all elements of Q will always be distinct.

Answer: We can keep an augmented red-black tree of the different elements in Q. Each node x in the tree will be augmented with the following three fields:

i. $mingap[x]$ is the minimum different between two nodes in the subtree rooted at x. If x is a leaf, $mingap[x]$ is ∞.
ii. $max[x]$ is the maximum key value in the subtree rooted at x.
iii. $min[x]$ is the minimum key value in the subtree rooted at x.

We can efficiently calculate these fields based simply on the node x and its children.

$$min[x] = \begin{cases} min[left[x]] & \text{if left child exists} \\ key[x] & \text{otherwise} \end{cases}$$

$$max[x] = \begin{cases} max[right[x]] & \text{if right child exists} \\ key[x] & \text{otherwise} \end{cases}$$

$$mingap[x] = min \begin{cases} mingap[left[x]] & \text{if left child exists} \\ mingap[right[x]] & \text{if right child exists} \\ key[x] - max[left[x]] & \text{if left child exists} \\ min[right[x]] - key[x] & \text{if right child exists} \\ \infty & \end{cases}$$

Notice that to calculate the *mingap* of a node x, we compare the *mingap*s of the subtree's rooted at the children as well as the two possible minimum gaps that x can have, namely where you subtract x from the smallest element largest than it and where you subtract the largest element smaller than x from x.

All three fields depend only on the node x and its children so we can efficiently maintain them through insertion, deletion, and rotation according to theorem 14.1 in CLRS. Notice that we keep the *min* and *max* values so that we can calculate *mingap* using only information from the node and its children.

The running time of the insert and delete operations are the same as for the red-black tree and are $\Theta(\lg n)$. The procedure $MinGap(Q)$ can simply look in the root node and return the *mingap* field which holds the minimum gap in the tree. Thus, the running time for $MinGap$ is $O(1)$.

(e) **[8 points]** Do problem 14-1(b) on page 318 of CLRS. You may assume the result of 14-1(a) without proof.

Answer: We know from part (a) that the point of maximum overlap is the endpoint of one of the intervals. So, we wish to find the endpoint which is the point of maximum overlap. To do this, we will follow the hint and keep a red-black tree of the endpoints of the intervals. With each node x, we will associate a field $v[x]$ which equals $+1$ if x is a left endpoint and -1 if x is a left endpoint. We will also augment each node with some additional information to allow us to find the point of maximum overlap efficiently.

First, we will provide some intuition. If $x_1, x_2, \ldots, x_n$ are the sorted sequence of endpoints of the intervals, then if we sum from $i = 1$ to j of $v[x_i]$, we find precisely the number of intervals which overlap endpoint j (this assumes our intervals are half-open of the form $[a, b)$). So, the point of maximum overlap will be the value of j which has the maximum value of $\sum_{i=1}^{j} v[x_i]$.

We will augment our red-black tree with enough information to calculate the point of maximum overlap for the subtrees rooted at each node. The point of maximum overlap depends on the cumulative sum of the values of $v[x]$. So, in addition to each $v[x]$, we will keep a variable $SUM[x]$ at each node which is the sum of the $v[i]$ values of all the nodes in x's subtree. Also, we will keep a variable $MAX[x]$ which is the maximum possible cumulative sum of $v[i]$'s in the subtree rooted at x. Finally, we will keep $POM[x]$ which is the endpoint x_i which maximizes the MAX expression above. Clearly, $POM[root]$ will be the point of maximum overlap on the entire set of intervals.

We will demonstrate how to compute these fields using only information at each node and its children. Therefore, by theorem 14.1, we can maintain this information efficiently through insertions, deletions, and rotations. The sum of the $v[i]$ of the subtree rooted at node x will simply be
$SUM[x] = SUM[left[x]] + v[x] + SUM[right[x]]$.
The maximum cumulative sum can either be in the left subtree, x itself, or in the right subtree. If it is in the left subtree, it is simply $MAX[left[x]]$. If it is x, the cumulative sum till x is $SUM[left[x]] + v[x]$. If it is in the right subtree, we have to add the cumulative sum up till the right subtree to the max value of the right subtree, $SUM[left[x]] + v[x] + MAX[right[x]]$.
$MAX[x] = \max(MAX[left[x]], SUM[left[x]] + v[x], SUM[left[x]] + v[x] + MAX[right[x]])$
The point of maximum overlap is the value of the node which maximized the above expression. ($l[x]$ and $r[x]$ refer to the left and right children of x respectively)

$$PMO[x] = \begin{cases} PMO[l[x]] & \text{if } MAX[x] = MAX[l[x]] \\ x & \text{if } MAX[x] = SUM[l[x]] + x \\ PMO[r[x]] & \text{if } MAX[x] = SUM[l[x]] + x + MAX[r[x]] \end{cases}$$

So, since each just depends on itself and its children, Interval-Insert and Interval-Delete will run in time $\Theta(\lg n)$ since we either insert or delete 2 nodes from the tree. At any time, $PMO(root)$ will hold the point of maximum overlap and

$MAX(root)$ will return how many intervals overlap it. Thus, the operation Find-PMO can be performed in $\Theta(1)$ time.

3. **[15 points]** Dynamic Programming: Optimization

(a) **[7 points]** You are planning a cross-country trip along a straight highway with $n+1$ gas stations. You start at gas station 0, and the distance to gas station i is d_i miles ($0 = d_0 < d_1 < \ldots < d_n$). Your car's gas tank holds $G > 0$ gallons, and your car travels $m > 0$ miles on each gallon of gas. Assume that G, m, and all d_i are integers. You start with an empty tank of gas (at gas station 0), and your destination is gas station n. You may not run out of gas, although you may arrive at a gas station with an empty tank. The price of gas at gas station i is p_i dollars per gallon (not necessarily an integer). You cannot sell gas.

Give an algorithm to calculate the cost of the cheapest possible trip. State the running time and space usage of your algorithm in terms of the relevant parameters.

Answer: We will solve this problem using dynamic programming by noting that if we knew the optimal way to leave gas station $i-1$ with any amount of gas in the tank, then we could easily calculate the optimal way to leave gas station i with any amount of gas in the tank. We could do this by taking the minimum over the cost of leaving gas station $i-1$ plus the cost of however much gas we'd need to put in the tank at gas station i. Notice that the amount of gas in the tank times m must be an integer in the range from 0 to mG since the distances we travel are given by gas/m and are all integers. From now on, we will call these values of $gas \times m$ units of gas. So, we can build a table T which is mG by n.

We initialize the first row of the table by noticing that the only way to leave gas station 0 with k units of gas is to fill up k units of gas at gas station 0. This has a cost of $(p_0 \times k)/m$. So, our base case of the table T is

$$T[0,k] = (p_0 \times k)/m.$$

We can then define each additional row in terms of a minimum over the previous row. For example, let's say we want to get to gas station i with no units of gas in the tank. There is only one way to do that since we can't sell gas. We must leave gas station $i-1$ with exactly enough gas to get to gas station i. This distance is $d_i - d_{i-1}$, and the amount of gas necessary is $(d_i - d_{i-1}) \times m$, so our units of gas are exactly the distance, $d_i - d_{i-1}$. So, the value of $T[i,0]$ is equal to $T[i-1, d_i - d_{i-1}]$. Let's now consider the case where we want to get to gas station i with 1 unit of gas in the tank. There are two ways we can acheive this. Either we can leave gas station $i-1$ with one more unit of gas in the tank or we can leave with exactly enough gas to get to i and buy one unit of gas at gas station i. $T[i,1] = \min(T[i-1, d_i - d_{i-1} + 1], T[i-1, d_i - d_{i-1}] + (p_i \times 1)/m)$. In general, we take the minimum over many cells in the previous row. First, define Δ_i as the distance between gas station i and $i-1$ so $\Delta_i = d_i - d_{i-1}$. We can calculate

$T[i,k] = \min_{\Delta_i \leq k' \leq \Delta_i + k} T[i-1,k'] + (p_i \times (k' - \Delta_i))/m$

Some times we will look in the table for values that don't exists, for example when k equals mG. Assume that $T[i-1, k' > mG]$ will return infinity.

After filling the table, we will search in the row $T[n,k]$ for the lowest value and return that as the cheapest possible trip. Notice that this cheapest cell will always be at $T[n,0]$ since any extra gas we have will have cost us something, and we may as well not have bought it. Notice also that we can return the actual trip by keeping pointers for each cell which point to the cell which caused the minimum.

For each cell in the table, we have to take a minimum over at most mG items. Since there are $n \times mG$ cells in the table, the running time for this algorithm is $\Theta(n(mG)^2)$. The space requirement is the size of the table which is $\Theta(nmG)$. Since calculating each row in the table only depends on the previous row, we can delete rows as we are finished processing them. So, we can reduce the space requirement to $\Theta(mG)$. However, if we wish to calculate the actual trip, we will need to maintain the $\Theta(nmG)$ table of pointers, which we aren't able to delete, since we don't know what our optimal trip is until after calculating the whole table. So, in that case the space requirement remains at $\Theta(nmG)$.

If we are clever in the way we update cells, we can actually reduce the running time of the algorithm to $\Theta(nmG)$ as well. Let's say we break the update to the cells into two phases, before adding gas at station i and after adding gas at station i. For the pre-fillup phase, we only have one way to arrive at a gas station with a given amount of gas k, so we simply set each

$T[i,k] = T[i-1, \Delta_i + k]$

where again we assume that $T[i-1, k' > mG] = \infty$. For the post-fillup stage, we can update each $T[i,k]$ by looking at the cost of adding one unit of gas to $T[i,k-1]$, assuming $T[i,k-1]$ already holds the minimum value to get there.

$T[i,k] = T[i,k-1] + (p_i \times 1)/m$ if that value is less than the current value in the cell.

Since each cell now only looks back at a constant number of cells instead of $O(mG)$ cells, the running time is reduced to $\Theta(nmG)$.

(b) **[8 points]** Do problem 15-7 on page 369 of CLRS.

Answer: Since there are so many variables in this problem, the first thing we need to figure out is what our optimal subproblems are. We do this by considering an optimal schedule over jobs $a_1, \ldots, a_j$ that runs until time t. Assume that we know which subset of jobs we perform to get the highest profit. What order do we need to perform those jobs in? Notice that the job with the latest deadline should be performed last. If we don't perform that job last, then we will waste the time between the second last deadline and the last deadline. The same thing

then applies for the second last deadline job being performed second last. We can argue that the subset of jobs that we perform will be done in increasing order of deadline.

This gives us our optimal subproblems. Namely, we order the jobs by increasing deadline. When we consider job a_i finishing at any time t (we assume these are now in sorted order so it has the ith deadline) we can simply look back at the optimal way to schedule the $i-1$ jobs and whether or not we add a_i to the schedule. We will also make the additional assumption that we leave no time gaps between the jobs. It is easy to argue that if we have a schedule with time gaps between jobs, we can also do the jobs in the same order with no time gaps and receive the same profit and possibly more.

The actual algorithm is as follows. We keep a grid of the n jobs versus the time, which can run up till d_n (since the jobs are now sorted by deadline). Notice that since the processing times are integers between 1 and n, the maximum time taken to complete all the jobs is at most n^2. So, if we have any deadline which exceeds n^2, we can simply replace it by n^2. Thus, our table is at most $n \times n^2$. Each cell in the table i, j will represent the maximum profit possible for scheduling the first i jobs in time *exactly* j, assuming that we have no gaps between our jobs.

We initialize the first row in the table by figuring out our possible profit based on completing the first job at exactly time t. Since we assume no gaps in our schedule, we must start the first job at time 0. Therefore, we will have a profit of p_1 in the cell at t_1 if $t_1 \leq d_1$ and a profit of zero otherwise. We'll initialize all other cells to zero. So, if we call our table T, we have

$$T[1,t] = \begin{cases} 0 & \text{if } t \neq t_1 \\ p_1 & \text{if } t = t_1 \leq d_1 \\ 0 & \text{if } t = t_1 > d_1 \end{cases}$$

Then, we can set the remaining cells in our table based simply on the previous row. At each cell $T[i,t]$, we have the choice of whether or not to perform job i. If we decide not to perform job i, then our profit is simply $T[i-1,t]$. If we decide to perform job i, then we know it takes t_i units of time to complete it. So, we must finish the previous jobs exactly at time $t - t_i$. We will get profit for the job if $t < d_i$. We will pick the maximum profit from these two choices.

$$T[i,t] = \max \begin{cases} T[i-1,t] & \\ T[i-1,t-t_i] + p_i & \text{if } t \leq d_i \\ T[i-1,t-t_i] & \text{if } t > d_i \end{cases}$$

At each cell, we will also keep a pointer to the cell which maximized the above expression. After filling in the whole table, we can search through the $T[n,t]$ row for the highest profit. Then, we can follow the pointers back to find the schedule. If in row i we take the pointer above us, we add i to the end of the schedule. Otherwise, if we take the pointer that is diagonal, we say we complete i at that time.

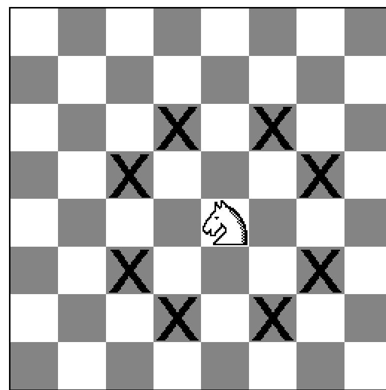
The running time of this algorithm is the number of cells in the table since each cell takes a constant amount of time to evaluate. The search through the last row

and following of the pointers both take time $\Theta(n)$. Therefore, the running time is $\Theta(n^3)$ if $d_n \geq n^2$ and $\Theta(nd_n)$ if $d_n < n^2$.

The space requirement for this algorithm is the size of the table, which is the same as the running time. Notice that since we want the actual schedule, we can't compress the size requirement.

4. [**15 points**] Dynamic Programming: Counting and Games

(a) [**7 points**] Consider a chess knight on an $n \times n$ chessboard. The knight may move from any square (i, j) to (i', j') if and only if $(|i' - i|, |j' - j|) \in \{(1, 2), (2, 1)\}$ and that square is on the board; in other words, the knight may move two squares in one direction and one square in a perpendicular direction (see diagram).



Give an algorithm to calculate the number of ways in which the knight can travel from square (i_s, j_s) to (i_t, j_t) in exactly $k \geq 0$ moves. State the running time and space usage of your algorithm in terms of the relevant parameters.

Answer: We can calculate the solution to this problem by noticing that if we knew the number of ways to get to every square on the board in $k-1$ moves from the starting location, we could easily calculate the number of ways to get to a square in k moves by simply summing over the atmost 8 squares that the knight could move from. Each way to get to the predecessor in $k-1$ moves contributes one way to get to the square in k moves.

Our DP matrix will be $n \times n \times k$. We initialize for all $k = 0$ the number of ways to get to that square in 0 moves. Namely,
$Ways(a, b, 0) = 1$ if $a = i_s$ AND $b = j_s$ and zero otherwise. We can build up our DP matrix for each $0 < i \leq k$ from these values. For each value of $i = 1 \ldots k$, and for each square (a, b) on the $n \times n$ board, we set the value of $Ways(a, b, i) = \sum_{(u,v) \in neighbors(a,b)} Ways(u, v, i-1)$. The neighbors of a cell are simply those that follow the condition given above for the legal knight's moves. At the end, we look at $Ways(i_t, j_t, k)$ to find the number of ways to get to (i_t, j_t) from (i_s, j_s) in exactly k moves.

Notice here a trend common to DP problems. In order to solve how many ways there are to get from a certain cell to another cell in k moves, we actually solve

more than asked for. We figure out how many ways there are to get from the start cell to **every** cell in k moves. While it seems like we do more computation, this actually makes solving the next layer more efficient.

The running time of this algorithm will be proportional to the number of cells in the matrix, since each cell takes a constant amount of time to calculate. Thus, the running time is $\Theta(n^2k)$. The amount of space is the same at $\Theta(n^2k)$. Notice, however, that to calculate the matrix for a given value of i we only use the values at $i-1$. So, at any time, we don't need to store the entire n^2k matrix, we only need two levels of it. If we delete levels (in k) as we finish using them, we only need space $\Theta(n^2)$.

(b) [**8 points**] The cat and mouse game is played as follows: there is an $n \times n$ board with a cat in (initial) location (i_c, j_c), a mouse in (initial) location (i_m, j_m), and a piece of swiss cheese in location (i_s, j_s).

Each cell (i, j) on the board has up to four neighbors: north $(i, j+1)$, south $(i, j-1)$, east $(i+1, j)$, and west $(i-1, j)$. All four cells adjacent in this manner are considered neighbors, except for the edges of the board and where there are **walls blocking some of the directions.** Therefore, assume that for any given cell (i, j) you have a list of zero to four neighboring cells $NBR(i, j)$ specified as you like (linked list, for example). You may assume the neighbor relationship is **symmetric**, i.e. if $(i', j') \in NBR(i, j)$, then $(i, j) \in NBR(i', j')$ and vice versa.

The game alternates between the moves of the mouse, who moves first, and the cat. The cheese does not move. If the mouse reaches the cheese before the cat catches it (i.e. without ever entering the same cell as the cat), then the mouse wins (the cheese endows the mouse with super powers). If the cat catches the mouse before the mouse can reach the cheese, the cat wins. If both conditions are satisfied simultaneously, i.e. the mouse enters a cell with the cat and the cheese in it, the game is considered a draw. You may assume that if none of this occurs within $2n^2$ moves, the game ends in a draw. Also, assume the cat and the mouse **must** move every round.

Assume that the cat and mouse are infinitely intelligent, can see the entire board and location of each other, and always make optimal decisions.

Give a dynamic programming algorithm to predict who will win the game. State its running time (a good upper bound is sufficient) and space usage in terms of n.

Answer: We can solve this problem by noticing that if the mouse knew the outcome for each of his at most four possible moves, he would know which way to go. If there was a move for which the mouse wins, he'll choose that move and win. If not, but there's a move for which the game is a draw, he'll choose that and draw. Otherwise, the cat wins for every move of the mouse, so we know the cat wins. Similar reasoning holds for the cat's strategy.

From this, we can see that our subproblems are who wins for each possible board configuration, whose move it is, and which move we're on. Since after $2n^2$ moves the game ends in a draw, we know the base case of whether the cat wins, mouse

wins, or a draw for the $2n^2$ move. We can define a set of boards for $2n^2+1$ which have the cat winning if the cat and the mouse are on the same square not on the cheese, the mouse winning if the mouse is on the cheese and the cat is elsewhere, and a draw otherwise.

We need to keep track of all possible boards for each move k. A board position consists of a cat position (n^2) and a mouse position (n^2), as well as whose turn it is. Since the mouse goes first, we know that for odd k it is the mouse's turn and for even k it is the cat's turn. We keep $2n^2$ possible sets of boards, each corresponding to which move number we are on. We initialize the $k = 2n^2$ board as above, and work backwards.

For each $0 \leq k < 2n^2$ and each of the n^4 possible board configurations, we wish to calculate who wins or if it is a draw. Our base cases are if the cat and the mouse are on the same square not on the cheese, then the cat wins. If the cat and the mouse are both on the same square as the cheese, then it's a draw. And if the mouse is on the cheese and the cat is elsewhere, then the mouse wins. Otherwise, we look at the at most four possible board configurations in the $k+1$ level corresponding to the possible moves. If k is odd (mouse's turn) and there is a possible board where the mouse wins, then we label this board as mouse, else if there is a possible board where there is a draw, we label this board draw, else we label this board cat. The symmetric case holds for k even (cat's turn).

After filling out our $(2n^2) \cdot (n^4)$ table, we look at the board in the $k = 0$ level corresponding to the starting locations of the cat and mouse and give the label of that board (cat, mouse, draw) as who wins the game.

Each cell looks at a constant number of other cells, so the running time of this algorithm is $\Theta(n^6)$. The memory requirement is also $\Theta(n^6)$. Notice however that we only need the $k+1$ boards to calculate the k boards. So, we need only store two sets of board configurations at a time, leading to a space requirement of $\Theta(n^4)$.

We can actually achieve a running time of $\Theta(n^4)$ if we do our updates in an intelligent manner. Rather than doing update move by move, we will initailiaze our board ($n^2 \times n^2 \times 2$ states) with the known terminal states (cat win, mouse win, draw), and the rest will start out unknown. As some state is updated, it may trigger up to four updates for its neighboring states. If we keep the states that need to be updated in a data structure such as a queue, we can process them one by one, doing a constant amount of work per update. We begin with neighbors of the terminal states in the queue, and continue adding neighbors to the queue for each update. The total number of updates done is at most the number of states on the board, which yields a running time of $O(n^4)$. Any state that fails to be updated when the queue is empty is then labeled as a draw state due to the assumption that any game that lasts at least $2n^2$ moves will end in a draw.

5. [**14 points**] Dynamic Programming: Sequences and Sparse DP

   (a) [**6 points**] Let X be a sequence $x_1, \ldots, x_n$ with associated weights $w_1, \ldots, w_n$. An increasing subsequence of t elements satisfies $x_{j_1} < \ldots < x_{j_t}$ and its weight

is $\sum_{i=1}^{t} w_i$. Describe a $\Theta(n \lg n)$ algorithm for finding the heaviest increasing subsequence.

Answer: This problem is similar to the longest increasing subsequence problem, where in that problem all elements had weight equal to 1. In that problem, we were able to store the lowest value of the last element of a subsequence of length i in an array $L[i]$. We showed in class that the $L[i]$ array remained sorted. For each element j in our sequence, we were able to perform a binary search in the L array for the largest value smaller than j and update our array accordingly. This led to a total running time of $\Theta(n \lg n)$ since each binary search took time $\Theta(\lg n)$.

In this case, we can't keep an array since the weights may not be integral values. However, we can keep a red-black tree which will allow us to find elements in time $\Theta(\lg n)$. Also notice that adding a single element may cause **many** previous subsequences to never be used again. For example, if we have the sequence of $(key, weight)$ of $(3,3), (2,2), (1,1), (0,5)$, we would find for the first three elements a subsequence of weight 1 ending in key 1, one of weight 2 ending in key 2 and one of weight 3 ending in key 3. Notice that all three of these subsequences may still be part of our heaviest subsequence. But, when we add the fourth element which gives a subsequence with weight 5 ending in key 0, none of the previous 3 subsequences can be part of our heaviest subsequence, since any time we could append elements to those, we could append them to the subsequence ending in 0 and achieve a higher weight. Thus, in order to maintain only the useless sequences we may now have to delete more than one element at a time.

Notice that the total number of elements deleted over the course of the algorithm is n, which yields running time of $O(n \lg n)$. However, each particular update may now take $\Theta(n)$, so without this realization (which is a mild form of amortized analysis) we do not have the desired bound with this approach.

The following is an alternative solution that uses augmented red-black trees. If you're comfortable with counting the deletion time separate from insertion time for the solution above, feel free to skip it. If you fear amortization like the plague, read on.

So, for each key, we will wish to compute the heaviest subsequence ending in that key. We will keep an augmented red-black tree of these (key, weight of subsequence) pairs keyed on the key. We will refer to the weight of the subsequence as w_s, not to be confused with the weight of the element. We will also keep pointers for each element of which element is its predecessor in the heaviest subsequence ending in that element. At the end, we will search through the tree for the highest weight and follow the pointers to find our heaviest subsequence.

For each element i, we will need to find in the tree the node x with the highest weight such that $key[x] < key[i]$. x is the node that we should append i onto to get the heaviest increasing subsequence ending in i. To do this, we will augment each node with $W[x]$, the maximum weight of subsequences ending in elements less than or equal to x in the subtree rooted at x. We can efficiently calculate $W[x]$ as $\max(W[left[x]], w_s[x])$. Since this property only depends on the values stored

at node x and its children, we can maintain it during insertion, deletion, and rotation. We will also keep at each node a pointer $P[x]$ which points to the node which attains that maximum weight. $P[x] = P[left[x]]$ if $W[left[x]] > weight[x]$ and $P[x] = x$ if $weight[x] \geq W[left[x]]$.

When we look at a new element i , we will insert it into the red-black tree. We initialize $totalweight = -\infty$ and $pointer = NIL$. We'll search in the tree for where i should go. Everytime we follow a left child, we do nothing. Everytime we follow a right child of some parent x, we set $totalweight = \max(totalweight, W[x])$ and $pointer = P[x] if W[x] > totalweight$. So, we essentially keep track of the maximum weight and the pointer to the node that had it for all keys that are less than i. When we find where to insert x, we set the weight of the subsequence ending in x to be $w_s[x] = totalweight + weight[x]$. We also set $W[x] = w_s[x]$ and $P[x] = x$. In addition, in our original array, we add a pointer from x to *pointer*, telling us how to backtrack to find the heaviest increasing subsequence.

We do this for all elements in the sequence (starting with the dummy element $-\infty$ with weight 0). Each insert into the tree takes time $\lg n$ and we have to perform n of them. At the end, we search through the tree for the highest weight ($\Theta(n)$) and follow the pointers back from that element to find the actual heaviest increasing sequence ($O(n)$). So, the total running time is $\Theta(n \lg n)$.

(b) **[8 points]** Let $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$ be two sequences. Assume that the number of pairs (x_i, y_j) that match (i.e. $x_i = y_j$) is small, i.e. $\emptyset(k)$ for some $k(m, n)$. Assume the set of matching pairs is given to you as part of the input. Give an $O(k \lg k)$ algorithm for finding the longest common subsequence of X and Y under these conditions.

Hint: reduce this to another sparse DP problem we have seen.

Answer: We will try to reduce our problem to the longest increasing subsequence problem. Notice that if we try to read through the entire input arrays, we will take time $\Theta(n + m)$ which could be bigger than k, so we are restricted to simply looking through the pairs of elements given. The longest common subsequence of X and Y must have increasing indexes in both X and Y. Thus, if we sort the pairs according to their index in X and feed the corresponding Y indexes into the longest increasing subsequence algorithm, we will obtain a common subsequence that is nondecreasing in the X index and increasing in the Y index. To avoid the problem of potentially having several pairs (x_i, y_j) and $(x_i, y_{j'})$ chosen by the LIS algorithm, we place the pairs with equal values of x_i in decreasing order of y_j, so that two pairs with the same X index can never be returned as part of a subsequence with increasing Y index. Sorting can be done using mergesort in $\Theta(k \lg k)$ time, and the sparse version of LIS runs in $\Theta(k \lg k)$ time, so the overall running time is $\Theta(k \lg k)$.