Problem 4-1. Xelda

You and your friend are obsessed with an awesome new video game, Xelda. The game works as follows: You walk down an infinite corridor with numerous rooms off of it. At each room, you can choose either to enter or to skip the room. Once inside a room, you fight an evil boss and are scored based on how well you fight. Then you return to the hallway and continue walking in the same direction.

The very first room you enter is a freebie, but after that, if you ever receive a fight score that is strictly less than your best fight score so far, you lose immediately. To win, simply reach the end of the corridor without losing!

Obviously, there are many ways to win this game. For instance, you’re guaranteed to win if you visit zero rooms or any single room. You’ve decided that you’re bored of Xelda, so to make it more interesting, you want to win in every single way possible. Let the term “winning strategy” refer to a subset of rooms to visit that will result in your winning the game; you would like to find all winning strategies.

(a) After playing Xelda a few times, your friend notices that your fight score is solely a function of the room. No matter how well (or poorly) you fight, you always receive the same score in a given room! In other words, if there are \( n \) rooms numbered \( 1, 2, \ldots, n \), then picking room \( i \) always results in a fight score of \( s_i \).

Given \( s_1, s_2, \ldots, s_n \), calculate the number of winning strategies in \( \Theta(n^2) \) time. Please give an algorithm, including pseudocode.

Solution: First, note that this problem is equivalent to: How many increasing (non-decreasing) subsequences of \( s_1, s_2, \ldots, s_n \) are there?

Let \( f_i \) be the number of increasing subsequences starting at room \( i \). However, we want to be able to start at any room, or (for the empty subsequence) to start at no room. Thus, let’s define a dummy room \( 0 \) with \( s_0 = -\infty \). The answer to the problem will be \( f_0 \).

Next, we show that we can compute all \( f_i \) in \( \Theta(n^2) \) time using dynamic programming. Assume we’ve already computed \( f_{i+1}, f_{i+2}, \ldots, f_n \). To compute \( f_i \), note that every increasing subsequence starting at room \( i \) consists of visiting room \( i \) and then appending an increasing subsequence starting at some room \( j \) such that \( s_j \geq s_i \).

In pseudocode:
Problem Set 4 Solutions

XELDA-DP(n, s)
1 for i = n to 0
2 \( f_i = 1 \)
3 for j = i + 1 to n
4 \[ \text{if } s_j \geq s_i \]
5 \[ f_i = f_i + f_j \]
6 return \( f_0 \)

(b) Augment your solution from Part (a) to also output the winning strategies. That is, instead of just counting how many winning strategies there are, you should also determine what they are. You can take \( O(n^2 + kn) \) time if there are \( k \) winning strategies, assuming that it takes \( O(n) \) time to print a single winning strategy. You need not provide pseudocode for this part.

Solution: Imagine drawing the following directed acyclic graph (DAG) as you execute XELDA-DP from Part (a): Start by creating vertices \( v_0, v_1, \ldots, v_n \). Then, for every \( i \) and \( j \) which make the if statement in line 4 evaluate to true, draw a directed edge from \( v_i \) to \( v_j \). We can draw this DAG while maintaining an \( O(n^2) \) runtime.

Clearly, there’s a bijection between winning strategies and paths in the DAG that start at \( v_0 \). The winning strategy corresponding to a path is given by the labels at the vertices along the path. We can therefore enumerate all winning strategies by traversing all paths in a depth-first fashion and storing the labels along the path in a stack.

The traversal has \( O(k) \) steps, each of which spends \( O(n) \) time outputting a strategy. Adding in the time to build the tree, the overall running time is \( O(n^2 + kn) \).

(c) Your friend nerd-snipes\(^1\) you with a really interesting data structures problem, so you take a quick break from playing Xelda to work on it. Given \( n \) bags, where the \( i \)th bag has label \( a_i \) and has \( c_i \) coins in it, you need to be able to support the following queries in \( O(\log n) \) time each:

- \text{COUNT-COINS}(i, j): Count the number of coins in bags whose label is between \( a_i \) and \( a_j \). Formally, compute \( \sum_{k:a_k \in [a_i, a_j]} c_k \).
- \text{ADD-COINS}(i, m): Add \( m \) coins to bag \( i \). Formally, set \( c_i = c_i + m \).

\(^1\)http://xkcd.com/356/
a static range tree because the values (and therefore the tree structure) don’t change, even though the augmentations are dynamic.

To implement ADD-COINS\((i, m)\), search for \(a_i\), and then add \(m\) to the sum at every vertex from the root to \(i\). This takes \(O(\log n)\) time because a 1D range tree is a complete BST.

To implement COUNT-COINS\((i, j)\), search for \(a_i\) and \(a_j\) and find the \(O(\log n)\) nodes and subtrees in between. Simply sum up the augmented values from each of these \(O(\log n)\) subtrees; this too takes \(O(\log n)\) time.

(d) By the time you finish Part (c), the creators of Xelda have released a sequel, Xelda II: To Adventure and Think. True to its name, this sequel requires more thought to play because there are now more than one billion rooms! You’re way too impatient to wait for your \(\Theta(n^2)\) algorithm in Part (a) to finish running.

Your friend snickers, because he read about the sequel on an online forum before it was released. Furthermore, he claims that he anticipated your crisis, and that his purportedly unrelated nerd snipe was actually intended to help you improve your algorithm for Xelda II!

With this advice in mind, design a fast algorithm to calculate the number of winning strategies in only \(O(n \log n)\) time. Please give an algorithm, including pseudocode; you can use ADD-COINS and COUNT-COINS as subroutines in your pseudocode.

**Solution:** We can rewrite our solution from Part (a) as:

```
XELDA-DP-ALTERNATE(n, s)
1  for i = n to 0
2     f_i = 0
3  for i = n to 0
4     sum = 0
5     for j = 0 to n
6       if s_i ≤ s_j ≤ ∞
7           sum = sum + f_j
8     f_i = f_i + 1 + sum
9  return f_0
```

Lines 4–7 look like COUNT-COINS, and line 8 looks like ADD-COINS, so we can use our data structure from Part (c)!

Bag \(i\) will have label \(s_i\) and coin count \(f_i\). As in Part (a), we need a dummy bag with \(s_0 = -∞\) to report the answer at the end of the problem. Additionally, because one of the endpoints of the query in line 6 is \(∞\), we introduce another dummy bag with \(s_{n+1} = ∞\).

We now modify XELDA-DP-ALTERNATE to use the data structure from Part (c). First, create a range tree on \(s_0, s_1, \ldots, s_{n+1}\), where each vertex is initially augmented with 0. Then run:
Problem 4.2. Intergalactic Trade

Intergalactic trade has become much easier since the construction of the Intergalactic Hyper-Freeway. Even so, it’s still time consuming to work out your sales routes and sometimes you wonder whether you’re planning them as efficiently as possible. To solve this problem, you decide to purchase a new Super Calculotron\textsuperscript{TM} 9009 and write a program to help out.

(a) Given $k$ indistinguishable goods to sell and $n$ possible planets to sell them on, with each planet $i$ offering to buy exactly one of the goods for $s_i$ \(\star\) Bucks, and the Intergalactic Hyper-Freeway tolls costing 1 \(\star\) Buck per Parsec traveled, you want to optimize your travel to make the most profit. Conveniently, all the planets on the Intergalactic Hyper-Freeway lie in a line along (positive or negative) integer coordinates in Parsecs. You start in the Galactic Center, corresponding to \((0)\) on the Intergalactic Hyper-Freeway. Give an efficient algorithm\textsuperscript{2} to calculate the maximum profit that can be made.

Solution: Since all the planets are restricted to a line, we can always reorder the way we visit planets by simply sorting them and visiting them in order. Thus we are really asking what interval contains the highest profit margins. We can thus consider including each planet one at a time in order of distance away. First, we sort the planets by their position. In particular, we want two lists, one for planets to the left of the Galactic Center and one for planets to the right of the Galactic Center sorted by their distance away from the Galactic Center. We will use the following strategy to solve this problem with dynamic programming:

1. Guess whether we will travel only to the right, only to the left, to the right and then to the left, or to the left and then to the right.
2. For each case, guess the planet which is farthest out before we stop or turn around.
3. Guess how many items we will sell on each leg of the trip.

\textsuperscript{2}Obtain the best running time you can. You will receive a higher grade for a more efficient algorithm, assuming equal qualities of exposition.
If we imagine traveling out to the $i^{th}$ farthest planet and selling $j$ goods, the distance to planet $i$ is $d_i$, and the value it is sold for at planet $i$ is $v_i$, then the profit for a single side is:

$$p[i, j] = \max(p[i - 1, j], p[i, j - 1] + v_i + d_{i-1} - d_i)$$

If this is the side we are doubling back on, we can simply use the same method with all of the distances doubled. Updating this table takes $O(kn)$ time for iterating though planet choices and how many sales to consider. Checking the combinations of which direction to start in simply adds a constant factor. Since we also had to sort the planets, this algorithm runs in $O(kn + n \log n)$ time.

(b) Now augment your algorithm in Part (a) to output the path of planets to take to maximize this profit.

**Solution:** All we really need is the interval containing the maximum profit planets from the first part. If we store this maximum interval, we can then quickly scan through all the planets in that interval and find the $k$ with the largest profit. Next, we need to split those into two groups, one to the left of the start location and one to the right of the start location. This takes $O(k)$ time. We then determine which distance (left or right) goes farther out and start by going away from that side, since we will have to double back on this leg of the trip. Finally, we perform an in-order walk on our data-structure, $O(n)$ time, storing the sorted order of the planets to determine which order to visit each side in. This does not increase the running time of the previous algorithm.

(c) Despite the convenience of the Intergalactic Hyper-Freeway, some planets you want to sell to haven’t been connected yet. With the success of your last algorithm, you decide to write another one for planning routes using the old-fashioned HyperDrive. Here, every planet has four integer Hyper-Space coordinates. In addition, you only have a total of $f$ units of fuel that you can use. The fuel used to travel between two points is equal to the Manhattan distance between the points (meaning the sum of the distances on each axis). Once again you begin at the Galactic-Center, located at $(0, 0, 0, 0)$. However, your cargo is unstable when undergoing HyperDrive travel and thus you can only carry one piece of cargo at a time. This means you must return to the Galactic-Center after making each delivery. Now give an efficient algorithm for determining the maximum profit attainable while navigating with your HyperDrive.

**Solution:** For this problem, we may note that what we really care about is the total distance away, not the exact coordinates. In addition, planets outside our fuel range need not be considered. Thus we can map every planet to be considered into an integer distance between 0 and $f$. Since we must return on all but the last trip, we will generally be dealing with twice the distance. If we ignore the fact that the last trip takes half
as much fuel, then we have subproblem optimality in terms of smaller amounts of fuel used. In particular, if the profit using a given amount of fuel $i$ and number of planets visited $j$ is $p[i, j]$ and a planet $m$ has distance $d_m$ and sells the good for $v_m$. Then

$$p[i, j] = \max_{2d_m < i, j < k} (p[i - 1, j], v_m + p[i - 2d_m, j - 1])$$

Thus, for each choice of fuel usage and number of planets to visit we must check all possible planets to update. This gives a running time of $O(nfk)$. However, we still need to deal with the fact that we do not need to return after the last shipment. To do this, we will guess which planet we will visit last, adding it’s sell value to our profit and subtracting the fuel cost from our total fuel available. This adds another factor of $n$ to our running time, yielding a final solution which runs in $O(n^2fk)$ time. Note, if $f$ happens to be significantly smaller than $n$ we could also have augmented our algorithm in a similar way to run in $O(nf^2k)$ time by guessing the remaining fuel rather than the last planet to visit.
Problem 4-3. More Hashing Guarantees

As discussed in class, the notion of universal hashing gives us guarantees that hold for arbitrary (i.e., worst-case) key inserts, in expectation over our choice of hash function. Let $H$ be a universal family of hash functions from some universe $U$ into a table of size $m$. Let $S \subset U$ be a set of $n$ elements we wish to hash. We will prove two of these guarantees.

(a) Show that, if $n$ is small enough with respect to $m$, the probability of a collision in the hash table is at most $\frac{1}{2}$.

**Solution:** Let $C$ be the total number of key pairs in $S$ that collide in the hash table. Let’s denote the $n$ distinct keys as $x_1, x_2, \ldots, x_n$. For every pair $(i, j)$, where $i, j \in \{1, 2, \ldots, n\}$ and $i < j$, let the indicator random variable $I_{i,j}$ be 1 if $h(x_i) = h(x_j)$ and 0 otherwise. So $C = \sum_{i<j} I_{i,j}$. By linearity of expectation, we have

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{i<j} I_{i,j}\right] = \sum_{i<j} \mathbb{E}[I_{i,j}] = \sum_{i<j} \Pr[h(x_i) = h(x_j)].$$

By the definition of universal hash functions, we have

$$\mathbb{E}[C] = \sum_{i<j} \Pr[h(x_i) = h(x_j)] \leq \sum_{i<j} \frac{1}{m} = \frac{(n)(n-1)}{2m} \leq \frac{n^2}{2m}.$$

Since $C$ is non-negative and $\mathbb{E}[C]$ is finite, we can apply Markov inequality to $C$,

$$\Pr[C \geq \frac{n^2}{2me}] \leq \Pr[C \geq \frac{\mathbb{E}[C]}{\epsilon}] \leq \epsilon.$$

When $\epsilon = \frac{1}{2}$ and $\frac{n^2}{2me} = 1$, or $n = \sqrt{m}$, we get,

$$\Pr[C \geq 1] \leq \frac{1}{2},$$

which is what we want to show. Therefore, we conclude that $n \leq \sqrt{m}$.

(b) Show that, with probability at least $\frac{3}{4}$, no bucket gets more than $1 + 2\sqrt{m}$ keys, when $n = m$.

**Hint:** Relate the size of each bucket to the total number of colliding key pairs in the hash table.

**Solution:** When $n = m$, from part (a), we get

$$\mathbb{E}[C] \leq \frac{m-1}{2}.$$
By Markov inequality again,
\[
\Pr[C \geq 2(m - 1)] \leq \frac{1}{4}.
\]
Notice that if some bucket had more than \(1 + 2\sqrt{m}\) keys in it, the total number of colliding key pairs in the bucket would be \(\frac{(1+2\sqrt{m})(2\sqrt{m})}{2} > 2m\). So \(C\) would be at least \(2m\) in this case. By the bound we have above, this event happens with probability less than \(\frac{1}{4}\). Therefore, the probability that no bucket has more than \(1 + 2\sqrt{m}\) keys is at least \(\frac{3}{4}\).

Universal hashing is nice, but sometimes, in order to simplify the analysis or to obtain a better performance guarantee, we need a stronger notion of independence from the hash functions. Call a family \(\mathcal{H}\) of hash functions \(k\)-universal if, for any \(k\) distinct keys \(x_1, x_2, \ldots, x_k \in U\) and for any values \(y_1, y_2, \ldots, y_k \in \{0, 1, \ldots, m - 1\}\), we have
\[
\Pr_{h \in \mathcal{H}} [h(x_1) = y_1 \text{ and } h(x_2) = y_2 \text{ and } \cdots \text{ and } h(x_k) = y_k] = \frac{1}{m^k}.
\]

(c) Show that, if \(\mathcal{H}\) is 2-universal, then it is universal. Next show that, if \(\mathcal{H}\) is \(k\)-universal, then it is \((k - 1)\)-universal.

Solution:
1. For two distinct keys \(\{x_i, x_j\}\), by the law of total probability,
\[
\Pr_{h \in \mathcal{H}} [h(x_i) = h(x_j)] = \sum_{y \in [m]} \Pr_{h \in \mathcal{H}} [h(x_i) = y] \cdot \Pr_{h \in \mathcal{H}} [h(x_j) = y].
\]
Since \(\mathcal{H}\) is 2-universal,
\[
\Pr_{h \in \mathcal{H}} [h(x_i) = h(x_j) = y] = \frac{1}{m^2}.
\]
Therefore,
\[
\Pr_{h \in \mathcal{H}} [h(x_i) = h(x_j)] = \sum_{y \in [m]} \frac{1}{m^2} = \frac{1}{m};
\]
\(\mathcal{H}\) is universal.

2. For any \(k - 1\) distinct keys, \(x_1, x_2, \ldots, x_{k-1}\) and for any values \(y_1, y_2, \ldots, y_{k-1}\), let \(x_k\) be another key that is different from \(x_i, i \in \{1, 2, \ldots, k - 1\}\). Let \(E\) be the event that \(h(x_1) = y_1\) and \(h(x_2) = y_2\) and \(\cdots\) and \(h(x_{k-1}) = y_{k-1}\). By the law of total probability again,
\[
\Pr_{h \in \mathcal{H}} [E] = \sum_{y \in [m]} \Pr_{h \in \mathcal{H}} [E \cap h(x_k) = y] = \sum_{y \in [m]} \frac{1}{m^k} = \frac{1}{m^{k-1}},
\]
where we use the definition of \(k\)-universal in the second to the last step.
(d) Assume that \( m \) is prime and \( |U| = m^r \), so that every key \( c \) can be represented as a unique number in base \( m \): \( c = (c_0, c_1, \ldots, c_{r-1}) \). For a key \( a = (a_0, a_1, \ldots, a_{r-1}) \), define \( h_a(c) \) by

\[
h_a(c) = (a \cdot c) \mod m = \sum_{i=0}^{r-1} (a_i \cdot c_i) \mod m.
\]

Let \( \mathcal{H} = \{ h_a \mid a \in U \} \). Recall that this is the dot-product hash family from lecture. Is \( \mathcal{H} \) 2-universal?

**Solution:** Consider the key, \( c \), whose representation is the 0-vector. Then \( h_a(c) = 0 \) for all \( a \in U \). Therefore, given non-zero indices \( y, y' \in [m] \), and another key \( c' \neq c \),

\[
\Pr_{a \in U} [h_a(c) = y \cap h_a(c') = y'] = 0.
\]

\( \mathcal{H} \) is not 2-universal.

(e) Suppose that we modify \( \mathcal{H} \) slightly from Part (d) as follows. For any \( a \in U \) and \( b \in \mathbb{Z}_m \) (integers modulo \( m \)), define

\[
h'_{a,b}(c) = \left( \sum_{i=0}^{r-1} a_i \cdot c_i \right) + b \mod m,
\]

and \( \mathcal{H}' = \{ h'_{a,b} \mid a \in U, b \in \mathbb{Z}_m \} \). Show that \( \mathcal{H}' \) is 2-universal.

**Solution:** Let \( c \) and \( d \) be two distinct keys from \( U \); and let \( \alpha = h'_{a,b}(c) \) and \( \beta = h'_{a,b}(d) \). Since \( c \neq d \), they must differ in at least one position of their base-\( m \) representation. W.L.O.G., assume that \( c_0 \neq d_0 \). Finally, let \( C = \sum_{i=1}^{r-1} a_i \cdot c_i \) and \( D = \sum_{i=1}^{r-1} a_i \cdot d_i \). To show that \( \mathcal{H}' \) is 2-universal, we must show that when \( a \) and \( b \) are randomly chosen, the following happens with probability \( \frac{1}{m^2} \):

\[
\begin{align*}
\alpha &= a_0 \cdot c_0 + C + b \mod m, \\
\beta &= a_0 \cdot d_0 + D + b \mod m;
\end{align*}
\]

or

\[
\begin{align*}
c_0 \cdot a_0 + b &= \alpha - C \mod m, \\
d_0 \cdot a_0 + b &= \beta - D \mod m.
\end{align*}
\]

We can view this as a system of equations with two unknowns, \( a_0 \) and \( b \). Since \( c_0 \neq d_0 \) and \( m \) is prime, the determinant of the matrix \( \begin{pmatrix} c_0 & 1 \\ d_0 & 1 \end{pmatrix} \) is non-zero in \( \mathbb{Z}_m \). This means the system of equations has a unique solution in \( \mathbb{Z}_m \). Note that this statement is true no matter what \( a_1, a_2, \ldots, a_{r-1} \) are. So there are \( m^{r-1} \) possible pairs, \((a, b)\), whose hash functions can make the system of equations hold. Therefore, when \( a \) and \( b \) are randomly chosen, the probability that the system of equations holds is \( \frac{m^{r-1}}{m^{r+1}} = \frac{1}{m^2} \).