1 Q1 (30 pts): Short questions

1. (3 pts) Suppose algorithm FAST runs in Θ(n²) time for all inputs of size n, and algorithm SLOW runs in Θ(2^n) time for all inputs of size n. FAST is faster than SLOW:

∃n₀, cFAST, cSLOW such that ∀n ≥ n₀, FAST(n) ≤ cFASTn², SLOW(n) ≥ cSLOW2ⁿ

⇒ limₙ→∞ FAST(n) / SLOW(n) ≤ limₙ→∞ n² / 2ⁿ = 0 [apply L’Hôpital’s Rule]

⇒ ∃n₀ such that ∀n ≥ n₀, FAST(n) < SLOW(n)

However, consider f(n) = n², g(n) = 2ⁿ. f(n) > g(n) ∀n < 5

(d) FAST is faster than SLOW on all but at most a finite number of inputs

2. (3 pts) What is the worst-case time to search for an item in a (not necessarily balanced) binary search tree with n nodes?

The worst-case search time is Θ(height of the tree). In the worst-case the tree will be linear, and height is n

(c) Θ(n)

3. (4 pts) The recursion T(n) = 1 for any n ≤ 10 and T(n) = n³ + n².93 + 8T(n/2) solves to:

T(n) ≥ L(n) = n³ + 8T(n/2). But L(n) = Θ(n³ log n) by Master’s Theorem. So T(n) = Ω(n³ log n)

T(n) ≤ U(n) = 2n³ + 8T(n/2). But U(n) = Θ(n³ log n) by Master’s Theorem. So T(n) = O(n³ log n)

(b) Θ(n³ log n)

4. (7 pts) Suppose you have a counter, represented by n binary bits. The cost of incrementation or decrementation is the number of bits you need to flip in order to represent the new number. What is the worst-cast amortized cost per operation we pay for a series of increment-operations and decrement-operations? Remember, n is the number of binary bits, not the number of operations.

Let some sequence of increments and decrements take the counter value 2ⁿ⁻¹ = 1000... Now consider a sequence of alternate decrements and increments. The counter value hops between 2ⁿ⁻¹ = 1000... and 2ⁿ⁻¹ − 1 = 0111... For each operation, every bit is flipped, and therefore
the amortized cost over this worst-case sequence of operations is (c) \( \Theta(n) \)

5. (7 pts) Suppose you have a data structure that supports two operations: \texttt{Insert}(x) and \texttt{Remove}_\text{min}. Given that the data-structure contains \( n \) elements, \texttt{Insert} adds a new element to the data structure in time \( T_n \), and \texttt{Remove}_\text{min} removes (and returns) the smallest element in time \( O(\sqrt{\log(n)}) \).

What is the best possible value of \( T_n \)?

We assume a comparison-based model. If you assumed a different model, you should have clearly stated what model.

We use the data structure to sort \( n \) elements into a list \( L \). First insert the \( n \) elements into the structure and then sequentially remove the minimum and add it to the end of \( L \). The total number of comparisons required is \( \leq nT_n + cn\sqrt{\log(n)} \). But as we don’t violate the sorting bound \( n\log(n) \), so \( nT_n + cn\sqrt{\log(n)} \geq n\log(n) \). So \( T(n) \geq \log(n) - c\sqrt{\log(n)} \) is (c) \( O(\log(n)) \)

6. (6 pts) Answer TRUE or FALSE with respect to the following directed graph:

(a) FALSE. The following is a possible DFS tour: \((g, c, b, h, a, d, f, e)\). \( d \) should have been visited after \( h \) before \( a \) was.

(b) FALSE. The following is a possible BFS tour: \((e, c, a, d, b, f, g, h)\). \( h \) should have been visited after \( f \) before \( g \) was.

2 Q2 (25 pts): Dynamic Programming

- 1,2,10,3
- See figure below
- Take 1, then 10 (no matter what the opponent did).
First define $T(i, j)$ to be the maximum amount of money that player 1 can guarantee given that it is currently his turn and the game is currently being played on the subarray $[a_i, a_{i+1}, \ldots, a_j]$. We now note that

$$T(i, j) = \max \{ a_i + \min \{ T(i + 1, j - 1), T(i + 2, j) \}, a_j + \min \{ T(i + 1, j - 1), T(i, j - 2) \} \}$$

The intuition behind this is very simple. When it’s player 1’s turn he has two choices: namely pick $a_i$ or pick $a_j$. We are essentially maximizing over this choice. Now given that player 1 has chosen either $a_i$ or $a_j$ player 2 now makes a move. Once player 2 makes a move it becomes player 1’s turn. We take the minimum of the amount of money we can make on our next move because that’s the smallest we can guarantee since player 2 can use any strategy he pleases.

The base cases are now just $T(i, j) = 0$ when $i > j$.

Given all of this we can now build a dynamic programming algorithm for computing the maximum amount of money player 1 can guarantee he wins. This bottom up dynamic programming algorithm would simply fill in the $n \times n$ tables by starting at the largest diagonal (filling in $T[i, i]$ for all $1 \leq i \leq n$) and then filling in the next diagonal. Once the table is filled out we can then simply output the $T[1, n]$ entry of the table as our final solution.

Extra credit: Color elements red and black (alternating). If you go first, you can always decide in advance that you will take all of the reds (or all of the blacks), and force your opponent to cooperate (this works since $n$ is even). All you need to do is calculate in advance which subset has a larger sum, and then take only items of the corresponding color.

3 Q3

1. A universal hash family is a set of functions $U \rightarrow \{0, 1, \ldots, m - 1\}$, s.t. for any $x, y, x \neq y$, $Pr_{h \leftarrow H}[h(x) = h(y)] \leq 1/m$.

   “Universal” means roughly ”one item is good for all elements”.

2. Yes. Look at the following hash-family $H$: 
First, observe that $H$ is a UHF: brute force checking over all 6 pairs gives that $\forall x \neq y, Pr_{h \leftarrow H}[h(x) = h(y)] \leq 1/2$.

The adversary can ask for $x = 2$, and views $h(2)$. If she views 0, it means that $h \equiv h_1$, so she picks $y = 1$ and have a collision: $h_1(x) = h_1(y) = 0$. If she views 1, it means that $h \equiv h_2$, so she picks $y = 4$ and again, have a collision: $h_2(x) = h_2(y) = 1$. So for this $H$, in the new model the adversary has a strategy that causes a collision w.p. 1.

Note that the question required you to show a complete strategy for the adversary, that is – specify what the adversary does under any possible answer for $h(x)$. (In this case, both for 0 and for 1.)

3. Proof: Fix any pair $x \neq y$.

$$Pr_h[h(x) = h(y)] = Pr_h[\exists i, h(x) = h(y) = i] \overset{(a)}{=} \sum_i Pr_h([h(x), h(y)] = [i, i]) \overset{(b)}{=} m \cdot \frac{1}{m^2} = \frac{1}{m}$$

where (a) follows from the fact that these are $m$ disjoint events, and (b) follows from the definition of 2-universal - as all pairs are equally like, the particular pair $[i, i]$ has probability of $1/m^2$.

4. This time the answer is No. We claim the following: For any $h \in H$, any $x \in U$, any $k, l \in \{0, 1, \ldots, m - 1\}$, and any $y \neq x$, we claim the following: $Pr_{h \leftarrow H}[h(y) = l \mid h(x) = k] = 1/m$.

First observe that this claim proves the “No” answer: no matter which $x$ the adversary picked, and which $k$ the adversary saw in reply for the query $h(x)$, any $y$ she picks is equally likely to be mapped by $h$ to any of the possible $m$ values. Therefore, the probability that $h(y) = k = h(x)$ and we have a collision is at most $1/m$.

The claim itself is pretty trivial, and follows from the Bayesian rule:

$$Pr_{h \leftarrow H}[h(y) = l \mid h(x) = k] = \frac{Pr_{h \leftarrow H}[h(y) = l \land h(x) = k]}{Pr_{h \leftarrow H}[h(x) = k]} \overset{(a)}{=} \frac{1/m^2}{Pr_{h \leftarrow H}[h(x) = k]} \overset{(b)}{=} \frac{1/m^2}{1/m} = 1/m$$

where (a) follows from the definition of 2-universal hash family directly, and (b) is derived in a similar fashion to the previous article:

$$Pr_{h \leftarrow H}[h(x) = k] = \sum_{k' = 0}^{m-1} Pr_{h \leftarrow H}([h(x), h(y)] = [k, k']) = m \cdot \frac{1}{m^2} = \frac{1}{m}$$

4. **Q4**

1. Say all edges have weight $w$. We take the DFS tree; this takes $O(m + n)$. We note that for any two trees with edge sets $E_1, E_2$ we have that, since the edge weights have the same size, the trees have the same weight. Formally,

\[
\sum_{e \in E_1} w(e) = \sum_{e \in E_1} w = |E_1|w =
\]
\[ |E_2|w = \sum_{e \in E_2} w = \sum_{e \in E_2} w(e) \]

Thus, the DFS tree (as well as any other tree) is an MST.

Common mistakes:
Many people attempted to modify Kruskal’s algorithm, which resulted in inefficiency.
Many people attempted to sort, by weight, the edges that all have the same weight...

2. We first, via DFS from the root, calculate the weights to every vertex currently in the tree.
Now, we may rebuild the heap and continue the algorithm.

Common Mistakes:
Many people attempted to run Prim’s algorithm, rather than Dijkstra’s, ignoring the distances from the current vertices in the tree to the root.