1 Short Answers

1. (3 points) Consider two versions of the `Quickselect` algorithm discussed in class:

   - `RQS` uses a random element as pivot.
   - `MQS` uses the element of the array at the "middle" position as pivot.

   Let A and B be two possible inputs to the problem. Select all true statement(s) about the two algorithms.

   ○ `RQS` runs faster than `MQS` on a sorted array.
   ○ `RQS` has a better worst-case running time than `MQS`.
   ● `RQS`'s time complexity on input A might be in $\Theta(n^2)$.
   ○ `MQS`'s average-case running time on A is equal to `MQS`'s average-case running time on B.
   ● `RQS`'s average-case running time on A is equal to `RQS`'s average-case running time on B.

2. (3 points) Decide if the following statement is true or false. If it is true, give a short explanation. If false, give and briefly explain a counterexample.

   Statement: In every instance of the Stable Matching Problem, there is a stable matching containing a pair (e, a) such that e is ranked first on the preference list of a and a is ranked first on the preference list of e.

   Solution: The statement is false. As a counterexample, consider the SMP instance:

   $e_1 : [a_1, a_2]$ $\quad a_1 : [e_2, e_1]$
   $e_2 : [a_2, a_1]$ $\quad a_2 : [e_1, e_2]$.

   In this instance, there is no pair (e, a) such that e is ranked first on the preference list of a and vice versa, so we obviously cannot obtain a stable matching that contains such a pair.

3. (3 points) Select all statements that are true about dynamic programming algorithms.

   ● They use a recurrence relation.
   ○ They are more efficient than a greedy algorithm that solves the same problem.
   ○ They combine a choice with a (possibly non-optimal) solution to a subproblem.
   ● They store information about subproblem solutions in a data structure such as an array.
   ○ None of the first four statements are true.

4. (3 points) Recall the statement of the Master theorem:

   Let $a \geq 1$, $b > 1$ be real constants, let $f(n) : \mathbf{N} \to \mathbf{R}^+$, and let $T(n)$ be defined by

   $$T(n) = \begin{cases} aT(n/b) + f(n) & \text{if } n \geq n_0 \\ \Theta(1) & \text{if } n < n_0 \end{cases}$$

   where n/b might be either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then

1. If $f(n) \in O(n^{(\log_b a)-\varepsilon})$ for some $\varepsilon > 0$ then $T(n) \in \Theta(n^{\log_b a})$.
2. If $f(n) \in \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$, then $T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) \in \Omega(n^{(\log_b a)+\varepsilon})$ for some $\varepsilon > 0$, and $af(n/b) < \delta f(n)$ for some $0 < \delta < 1$ and all n large enough, then $T(n) \in \Theta(f(n))$.

and consider the following recurrence relation:

$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{if } n \geq b \\ \Theta(1) & \text{if } n < b \end{cases} \qquad \text{where} \qquad f(n) = \begin{cases} 1 & \text{if } n \text{ is even} \\ n^7 & \text{if } n \text{ is odd} \end{cases}$$

You are told that (for some specific values of a and b that we are not giving you), it **is** possible to apply the Master theorem to solve this recurrence.

(a) Which case of the Master theorem is applied (there is a single valid answer)?

● Case 1 ○ Case 2 ○ Case 3

(b) Give a Θ bound on $T(n)$.

Solution: $T(n) \in \Theta(n^{\log_b a})$.

2 From weighted lists to weighted trees

You are writing an application that needs to manipulate sorted lists where most elements are repeated multiple times. Instead of including many copies of each value, you decide to store only one copy of each element, along with a count indicating how many times it occurs. For instance, the list

```
[
  1,  1,  1,  4,  4,  4,  4,  4,  6,  6,  9,  13,
  13, 13, 13, 13, 13, 17, 17, 17, 17, 22, 22, 22
]
```

would be represented by

```
[
  (key: 1, count: 3), (key: 4, count: 5), (key: 6, count: 2), (key: 9, count: 1),
  (key: 13, count: 6), (key: 17, count: 4), (key: 22, count: 3)
]
```

For reasons too complicated to explain here, you then need to turn the list containing the (key, count) pairs into a *binary search tree* T that is balanced in the following way:

> For every node N of the tree, the sum of the counts in the left subtree rooted at N differs from the sum of the counts in the right subtree rooted at N by at most the count at the node N.

In the example given above, we can make the pair `(key: 13, count: 6)` the root of the tree, because the sum of the counts in the left subtree is

$$3 \text{ (for key 1)} + 5 \text{ (for key 4)} + 2 \text{ (for key 6)} + 1 \text{ (for key 9)} = 11$$

, the sum of the counts in the right subtree is

$$4 \text{ (for key 17)} + 3 \text{ (for key 22)} = 7$$

and $11 - 7 \leq 6$ (the count for key 13). However we could not make the pair `(key: 9, count: 1)` the root, because then the sum of the counts in the left subtree would be

$$3 \text{ (for key 1)} + 5 \text{ (for key 4)} + 2 \text{ (for key 6)} = 10$$

, the sum of the counts in the right subtree would be

$$6 \text{ (for key 13)} + 4 \text{ (for key 17)} + 3 \text{ (for key 22)} = 13$$

and $13 - 10 > 1$ (the count for key 9).

1. (8 points) Here is a partial algorithm that constructs the tree T given a list L, where the function `Node(x, y, z)` returns a node containing the list element x, with left child y and right child z. Fill in the blanks to complete the algorithm. Use the notations `L[i].key` and `L[i].count` to access the `key` and `count` fields of list element `L[i]`.

```
Algorithm BuildTree(L)
  L[0].prevsum = 0
  for i ← 1 to len(L) - 1 do
    L[i].prevsum ← L[i-1].prevsum + L[i-1].count

  return BuildTreeHelper(L, 0, len(L) - 1)


//
// The helper uses divide and conquer.
//
Algorithm BuildTreeHelper(L, first, last)
  if first > last then
    return null

  if first = last then
    return Node(L[first], null, null)

  mid_sum = ⌊ (L[first].prevsum + (L[last].prevsum + L[last].count)) / 2 ⌋
  use binary search to find mid where
              L[mid].prevsum < mid_sum ≤ L[mid].prevsum + L[mid].count
```

```
return Node(
    L[mid],
    BuildTreeHelper(L, first, mid-1),
    BuildTreeHelper(L, mid+1, last)
)
```

3 Gotta Be 5G

You're installing cell towers in a remote area that doesn't yet have cell service. You're working on a long country road in this area with houses scattered very sparsely along it. You want to make sure that no house on this road is more than 6 kilometres away from any cell tower. At the same time, you want to minimize the number of towers you need to build. Formally, we can picture the road as a line segment, with the positions of houses given by their distance (in kilometres) from the start of the road.

For example, if the house positions are given in the array $H = [1, 10, 13, 17, 20]$, then an optimal solution is to place two cell towers at positions 5 and 19. The optimal solution for this instance is not unique: we could also place the cell towers at positions 7 and 20, or 7 and 21, and so forth. You may assume that the road continues for at least 6 km beyond the position of the last house, and that you can therefore install a cell tower past the last house; however, you may not place any cell tower before the road starts (i.e., at a position less than 0). Additionally, neither the house positions nor the positions of the cell towers are required to be integers.

In the problems below, assume the array H is not empty and uses 1-based indexing.

1. (3 points) Fill in the blanks of the following pseudocode for a greedy algorithm that optimally solves this problem.

   Sort H in increasing order.

   While H is not empty:

   Place the first cell tower at position $H[1] + 6$

   Remove from H all values between $H[1]$ and $H[1] + 12$

2. (9 points) Fill in the blanks in the following **PARTIAL** proof of correctness for our greedy algorithm.

   Let $\mathcal{G} = [g_1, g_2, \ldots, g_k]$ denote the positions of the cell towers in our greedy solution, sorted in increasing order, and let $\mathcal{O} = [o_1, o_2, \ldots, o_m]$ denote the tower positions in an optimal solution, again in increasing order. We will prove by induction the following "greedy-stays-ahead" lemma: $\mathcal{G}[i] \geq \mathcal{O}[i]$ for all $i = 1, \ldots m$ (using 1-based indexing).

   Base case: When $i = 1$, we must have $\mathcal{G}[i] \geq \mathcal{O}[i]$ because

The greedy algorithm puts the first cell tower at position $H[1]+6$, which is as far down the road as we can place it without leaving the first house more than 6km from a cell tower. If the optimal solution has the first cell tower further than that, then the first house will be more than 6 km from a tower, which will mean the optimal solution isn't optimal.

Inductive case: Consider an arbitrary integer i where [2] $\leq i \leq k$. Assume that $\mathcal{G}$[$i-1$] $\geq \mathcal{O}$[$i-1$]. Then we must have $\mathcal{G}[i] \geq \mathcal{O}[i]$ because

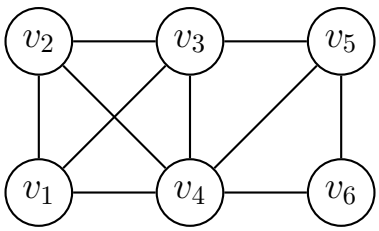
By the inductive hypothesis, $\mathcal{G}[i-1] \geq \mathcal{O}[i-1]$, so the first house not within range of a tower in our greedy solution is also out of range of a tower in the optimal solution (because the last cell tower in $\mathcal{O}$ is at the same position or earlier as the last cell tower in $\mathcal{G}$). The greedy algorithm places a tower 6km beyond the position of this house, which is as far as we can place this tower without leaving this house without coverage. Therefore, if $\mathcal{O}$ has tower i placed later, this house will be without coverage, which will mean the optimal solution isn't optimal.

4 Cliquebait

Recall that a **clique** is a subset W of the vertices V in a graph with the property that every two elements in W are joined by an edge. In this question, we consider the *k-clique* problem:

Given a graph $G = (V, E)$ and integer $k \geq 3$, does G contain a clique of size k? That is, is there a subset W of V with $|W| = k$ such that every two elements in W are joined by an edge in E?

For example, the following graph contains a k-clique for $k = 4$ (given by $[v_1, v_2, v_3, v_4]$) and several for $k = 3$ ($[v_3, v_4, v_5], [v_4, v_5, v_6]$ or any three of v_1, v_2, v_3 and v_4), but not for $k \geq 5$.



Recall moreover the *Boolean satisfiability* (SAT) problem, defined as follows:

The input is a collection of m *clauses* over p boolean variables X_1, X_2, $\ldots X_p$. Each clause is a disjunction of some of the variables or their complements.

The problem consists in answering the question "Is there a way to assign truth values to each variable that makes **every** clause of the instance TRUE?

Here is an incomplete reduction from k-clique to SAT. Given a graph $G = (V, E)$ and a value k, we define the variable $X_{i,j}$ for $i = 1$ to n and for $j = 1$ to k, which represents whether the vertex v_i is the jth element of the k-clique. We define clauses as follows:

(A) For each integer a from 1 to k, add the clause:

$$X_{1,a} \vee X_{2,a} \vee \ldots \vee X_{n,a}.$$

(B) For each pair of vertices v_p and v_q and for every a in 1 to k, add the clause:

$$\overline{X}_{p,a} \vee \overline{X}_{q,a}.$$

(C) For each vertex v_p and for every two distinct integers a, b from 1 to k, add the clause:

$$\overline{X}_{p,a} \vee \overline{X}_{p,b}.$$

(D) *To be completed.*

Finally, G contains a k-clique if and only if the reduced SAT instance is satisfiable.

1. (1 point) For this reduction, briefly explain the purpose of the clauses added in (A).

   Solution: The clauses in (A) ensure that at least one vertex is in any position of the clique.

2. (1 point) Briefly explain the purpose of the clauses added in (B).

   Solution: The clauses in (B) ensure that no more than one vertex is in any particular position of the clique.

3. (1 point) Briefly explain the purpose of the clauses added in (C).

   Solution: The clauses in (C) ensure that no vertex can be in more than one position in the clique.

4. (7 points) Complete the reduction by adding the necessary clauses in (D). You do not need to prove the correctness of the reduction, but you should explain the purpose of the clauses you are adding. You should not introduce any new variables.

   Solution: We still need to ensure that the clique consists only of connected vertices. If two vertices v_p and v_q don't share an edge, we know they can't both belong to the clique. In order to ensure they aren't both in the clique, we need to specify that they can't both be in any position within the clique. Therefore, we add the following clauses:

For all pairs of vertices v_p and v_q that do not share an edge, add the clauses

$$\overline{X}_{p,a} \vee \overline{X}_{q,b},$$

for every a and every b between 1 and k.

A note on the exact values of a and b: it's okay (though not required) to leave out clauses where $a = b$, as these will be duplicates of clauses added in step (B). However, if we are assuming $p < q$, it is NOT okay to only consider pairs with $a < b$ (such as the nested for loop: for $a = 1 : k - 1$ and for $b = a + 1 : k$), because this would prevent v_q from being in any position after v_p in the k-clique but would not prevent v_q from being placed *before* v_p in the k-clique. However, as long as your answer did not **obviously** make this mistake (say, by explicitly defining indices or loops for both the p, q and a, b pairs such that $q > p$ and $b > a$, or saying something like "for every distinct pair" of vertices and integers) we were not picky about this for grading purposes.