CS 161: Midterm — Solutions

Question 1

1. If $f(n) = O(g(n))$ and $g(n) = O(n^2)$, then $f(n) = O(n^2)$.
   True.

2. If the running time of algorithm A is given by $T_A(n) = T_A(n-2) + 5n$, and algorithm B is given by $T_B(n) = 8T_B(n/3) + n$, then algorithm B is asymptotically faster.
   True: $T_B(n) = \Theta(n^{\log_3 8})$ by the master theorem, while $T_A(n) = \Theta(n^2)$ by simply unrolling the recurrence. Since $\log_3 8 < 2$, B is asymptotically faster.

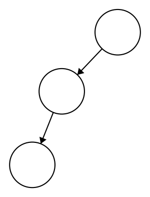
3. If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$.
   False: Consider $f(n) = 2n$ and $g(n) = n$.

4. No sorting algorithm to sort n numbers in $\{1, \ldots, n\}$ can run in $O(n)$ time.
   False

5. If $f(n) = O(n^2)$ then there is a constant c such that $f(n) \leq cn^2$ for all $n \geq 1$.
   True: The definition given in class required this inequality only to hold for $n \geq n_0$, but we can always pick a large enough constant c so that $f(n) \leq cn^2$ for all positive n.

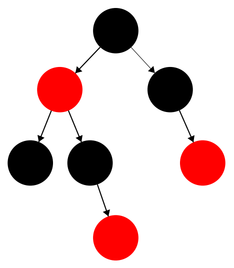
Question 2

(a)



It is impossible to color this BST as a valid red-black tree. (You must take NILs into account.)

(b)



is the only valid coloring.

Question 3

(a) Iterate through the array and check if there is a BLIP at each index i. If index $i = 1$, we only need to check $A[1] > A[2]$. If index $i = n$, we check $A[n] > A[n-1]$. Return the index of the first BLIP. For each index, we perform a constant amount of work. In the worst case, we find a BLIP at index n, so the run time is $O(n)$.

(b) Pick the center element $A[\frac{n}{2}]$ and check whether it is a BLIP. If it is, output it and stop. If it is not, there must exists a larger neighbor (since there are no duplicates). if $A[\frac{n}{2}+1] > A[\frac{n}{2}]$, we recursively search for a BLIP in $A[\frac{n}{2}+1 : n]$. Otherwise, search for a BLIP in $A[1 : \frac{n}{2} - 1]$. The algorithm begins by calling FIND-BLIP$(A, 1, n)$.

```
function FIND-BLIP($A, i, j$)
    if $j - i \leq 1$ then
        if $j - i == 0$ or $A[i] > A[j]$ then
            return $i$
        else
            return $j$
        end if
    end if
    mid $\leftarrow \frac{i+j}{2}$
    if $A[\text{mid}+1] < A[\text{mid}]$ and $A[\text{mid}] > A[\text{mid}-1]$ then
        return mid
    else if $A[\text{mid}+1] > A[\text{mid}]$ then
        return FIND-BLIP($A, \text{mid}+1, j$)
    else
        return FIND-BLIP($A, i, \text{mid}-1$)
    end if
end function
```

We maintain the invariant that the subarray we recursively search on contains a BLIP. Since all elements are distinct, the original array must contain a BLIP (for example, the global maximum in the array is a BLIP).

In addition, every time we call FIND-BLIP(A, i, j), if $i > 1$, $A[i] > A[i-1]$ and if $j < n$, $A[j] > A[j-1]$. Note that at the beginning, $i = 1$ and $j = n$ so this holds. We update i only when we call FIND-BLIP$(A, \text{mid}+1, j)$, and that happens when $A[\text{mid}+1] > A[\text{mid}]$. We update j only when we call FIND-BLIP$(A, i, \text{mid}-1)$, and that happens when $A[\text{mid}] < A[\text{mid}-1]$.

We now claim that when we call FIND-BLIP(A, i, j), the global maximum in that subarray is a BLIP. The global maximum is greater than all the other elements in the subarray. If the index of the global maximum is not i or j, it is a BLIP (by definition). If it is i, we still know that $A[i] > A[i+1]$. Then, either $i = 1$ or $A[i] > A[i-1]$, and we found a blip. The case where the index of the global maximum is j is analogous. This proves that any time we call FIND-BLIP(A, i, j), the subarray $A[i : j]$ contains a BLIP.

Note: A BLIP may occur in the opposite subarray we recursively search in. However, this algorithm always guarantees a BLIP will be found (if one exists).

This problem is analogous to finding a local maximum by hill climbing.

(c) $T(n) = T(n/2) + O(1)$

Question 4

(a) Pseudocode for MultiSearch algorithm:

Algorithm 1 MultiSearch(A,S)

```
if size(S) == 0 then
    return []
end if
if size(S) == 1 then
    return [Select(A,S[1])]
end if
pivot ← Select(A,S$\left[\lfloor \frac{size(S)}{2} \rfloor\right]$)
$A_< \leftarrow \{A[i] | A[i] < \text{pivot}, 1 \leq i \leq n\}$
$A_> \leftarrow \{A[i] | A[i] > \text{pivot}, 1 \leq i \leq n\}$
$S_< \leftarrow S[1 : \lfloor \frac{\text{size(S)}}{2} \rfloor - 1]$
$S_> \leftarrow [S[\lfloor \frac{\text{size(S)}}{2} \rfloor + 1] - \text{pivot}, \ldots, S[\text{size(S)}] - \text{pivot}]$
return MultiSearch($A_<, S_<$) + [pivot] + MultiSearch($A_>, S_>$)
```

Explanation: The basic idea behind the MultiSearch algorithm is to partition the input set S into two halves at each level of recursion, while using the median element of S at each step to find one element in the solution while also partitioning A into two parts. By using this strategy, we are able to achieve $O(\log k)$ levels in the recursion tree with $O(n)$ total work at each level, which leads to the desired running time of $O(n \log k)$. It is important to note that at each level in the recursion, all subproblems are working with distinct subarrays of A. By not doing any overlapping work, the algorithm can ensure that the work does not increase as we go down the recursion tree. The work done outside of the recursive calls consists of linear time Select calls (as described in lecture) and linear time partitioning of A and S. The algorithm finds one of the elements in the solution at each recursive step, either via the Select call when finding the pivot or through the base case when S has size 1.

An alternative solution that we saw divided A into two halves instead of S by picking the median of A as a pivot (using Select). Then, S was split accordingly into two parts that are not necessarily of equal size. The algorithm was called on both matching parts of A and S, and the output was the returned elements from the recursive calls and the median of A if needed. For this solution, it is especially important to specify the base case where S is the empty set. Otherwise, if the algorithm only stopped when the size of A is at most 1, there would be many unnecessary recursive calls that may result in a runtime of $\Theta(n \log n)$. Some solutions used a random element from A as the pivot, however, the question asked for worst-case run time of $O(n \log k)$ and not expected worst-case run time.

(b) For the first solution, there are $O(\log k)$ levels in the recursion tree since the algorithm will partition S into two halves at each step until it has size 1. Each recursive subproblem in the recursion tree may work with a subarray of A of arbitrary size (depending on the value of the pivot used to partition A), but notice that all the subarrays in each level are non-overlapping. Hence, the total size of the subproblems at each level is at most $O(n)$. This, along with the fact that only linear work is done outside of recursive calls, ensures that the total work done at each level is $O(n)$. Thus, we get that the running time of the algorithm is $O(n \log k)$.

For the alternative solution, the analysis is a bit more involved. Note that the total size of the subproblems at each level is still at most n, hence, the total work done at each level is $O(n)$. However, since S is not necessarily divided into two halves at each level, there may be more than $\log k$ levels. At the first $\log k + 1$ levels together, the total work done is $O(n \log k)$. We claim that the work done at all subsequent levels together is $O(n)$.

Since the median is chosen at each level as the pivot, the size of each subproblem at level j is $\frac{n}{2^j}$. Therefore, at level $\log k$, the size of each subproblem is $\frac{n}{k}$. However, note that for each subproblem, the size of the corresponding part of S should be at least 1. S can be divided into at most k disjoint parts, hence, at each level, we can have at most k subproblems. Thus, at level $\log k$, we have at most k problems of size $\frac{n}{k}$, leading to total work of $O(n)$. At the next level, we have at most k problems of size $\frac{n}{2k}$, and so on (the total size of the subproblems decreases by a factor of 2 at each level). The total amount of work done at all these levels is $O(n)$.

(c) Let $\mathcal{A}$ be any comparison-based algorithm that solves the MultiSearch problem. We can represent $\mathcal{A}$ as a decision tree where internal nodes correspond to comparison operations with binary outcomes and leaves correspond to possible outputs. Let h be the height of the decision tree. Each outcome is a mapping $\pi : [k] \to [n]$ such that $\pi(i)$ is the index of the p_i-th smallest element in A. Since $\mathcal{A}$ solves the MultiSearch problem, it must be able to return all possible outputs. There are $\binom{n}{k} \cdot k!$ possible outputs (choose a set of k locations in which the request elements are located, and consider all their permutations), and so, there has to be at least this many leaves. On the other hand, the decision tree is a binary tree and it can have at most 2^h leaves. Then,

$$2^h \geq \# \text{ of leaves} \geq \binom{n}{k} \cdot k! = \frac{n!}{(n-k)!}\,.$$

Therefore, the height of the tree is at least $\log \frac{n!}{(n-k)!}$. We can rewrite $\frac{n!}{n-k!}$ as $n(n-1)\cdots(n-k+1)$. Now we see that at least $k/2$ of these terms must be greater than or equal to $n/2$. To understand why this is true, it may be helpful to consider the following cases: If $k \leq n/2$, it is clear that all k terms are at least $n/2$. If $n/2 < k < n$, then over half of the k terms will be greater than or equal to $n/2$. If $k = n$, then exactly half of the terms will be greater than or equal to $n/2$.

Then for any $n \geq 4$,

$$\begin{aligned}
\log\left(\frac{n!}{(n-k)!}\right) &= \log\left(n(n-1)\cdots(n-k+1)\right) \\
&\geq \log\left(\frac{n}{2}\right)^{\frac{k}{2}} \\
&\geq \frac{k}{2}\log\left(\frac{n}{2}\right) \\
&= \frac{k}{2}(\log n - 1) \\
&\geq \frac{k}{4}\log n
\end{aligned}$$

Therefore $\forall n \geq 4$ and $c = \frac{1}{4}$, $h \geq \log\left(\frac{n!}{(n-k)!}\right) \geq ck\log n$, or $h = \Omega(k \log n)$. As the height of the decision tree corresponds to the number of comparisons needed in the worst case, the lower bound follows.

Alternatively, using Stirling's approximation:

$$\begin{aligned}
\frac{n!}{(n-k)!} &\geq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{1/(12n+1)} \Big/ \sqrt{2\pi(n-k)}\left(\frac{n-k}{e}\right)^{n-k} e^{1/12(n-k)} \\
&\geq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{1/(12n+1)} \Big/ \sqrt{2\pi n}\left(\frac{n}{e}\right)^{n-k} e^{1/12(n-k)} \\
&= n^k \cdot e^{-k} \cdot e^{1/(12n+1) - 1/12(n-k)}\,.
\end{aligned}$$

It follows that

$$2^h \geq n^k \cdot e^{-k} \cdot e^{1/(12n+1) - 1/12(n-k)}\,.$$

Taking ln on both sides, we obtain

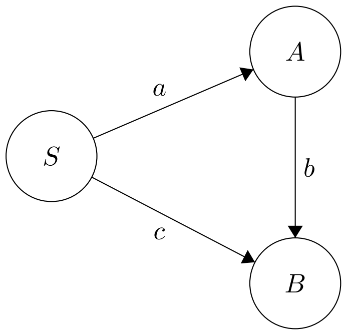
$$h \cdot \ln 2 \geq k \ln n - k + \frac{1}{12n+1} - \frac{1}{12(n-k)} \,,$$

and we see that $h = \Omega(k \log n)$.

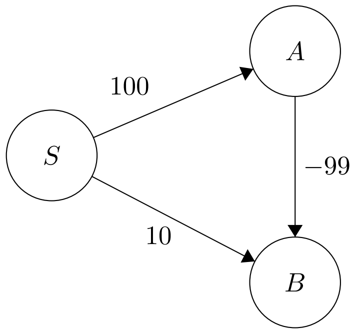
Note that the MultiSearch problem becomes the sorting problem when $k = n$ and the lower bound becomes $\Omega(n \log n)$, which is exactly the lower bound we proved for comparison-based sorting algorithms.

Question 5

(a) Dijkstra's algorithm will fail on any graph of the following form, where $c < a$ and $a + b < c$.



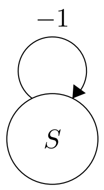
As a concrete example:



In the above example, Dijkstra's algorithm will first set the distance to S to 0. Then, it will consider B (the vertex with the lowest distance estimate) and say that the *final* distance from S to B is 10. However, the actual shortest path has length 1.

Note that this depends on the implementation of the algorithm. The above example works when considering the algorithm as presented in lecture. However, in the efficient implementation that appears in the lecture notes, after vertex A is considered, the distance to B will be updated even though it has already been declared final. When there are no negative weights, the distance to a vertex is never updated after it is final. An example on which that implementation will fail requires four vertices.

Also, although this was not the intent of the question, Dijkstra's algorithm will fail on any graph with negative cycles. For instance:



(b) The provided family is not universal.

To show that a family of hash functions $h_b(x)$ is not universal, it is sufficient to show that for a hash function h randomly chosen from h_b, $P(h(x_i) = h(x_j)) > \frac{1}{n} = \frac{1}{11}$ for some x_i and x_j, $x_i \neq x_j$.

If we take x_i to be 0 and x_j to be 11, then for any hash function h (any value of b) that we select from the family, $h(0) = h(11)$ because $5 \cdot 0 + b \equiv 5 \cdot 11 + b \pmod{11}$. Therefore, $P(h(0) = h(11)) = 1 > \frac{1}{11}$.