CMPUT204
Department of Computing Science
University of Alberta

Assignment #3 Solutions

Question 1. (15 marks)

a. (10 marks) Consider sorting an array of n floating-point numbers. The array, though, contains at most k unique numbers, where k is much smaller than n (i.e., $k < \log n$). Describe an efficient algorithm for sorting this array. The running time of your algorithm should be in $o(n \log n)$ (i.e., faster than MergeSort).

```
MySort(A)
    k ← 0
    for i ← 1 to length[A] do
        found ← false
        for j ← 1 to k do
            if v[j] = A[i] then
                c[j] ← c[j] + 1
                found ← true
        if (not found) then
            k ← k + 1
            v[k] ← A[i]
            c[k] ← 1
    Sort v and c arrays based on the values of v.
    i ← 1
    for j ← 1 to k do
        A[i ... i + c[j]] ← v[j]
        i ← i + c[j]
```

The algorithm loops over every element checking if it's unique. If not, it increments the count of the number of that many elements. If it is unique, it adds it to the list of unique elements. This takes $O(nk)$ time. It then sorts the unique values taking $O(k \log k)$ time. Finally it writes the correct number of copies of each element in sorted order back into the array $O(n)$.

b. (5 marks) In terms of n and k, what is the asymptotic running time of your algorithm?

Overall running time is $O(nk) \in o(n \log n)$.

Question 2. (15 marks)

a. (12 marks) The heap property is one example of a *tree sort*. A rooted tree (not necessarily binary, d-ary, complete, or balanced), is sorted if and only if, for all non-root nodes n, key$[n] \leq$ key[parent$[n]$]. Write an efficient procedure TreeSort(T), that takes an arbitary rooted tree T and sorts the keys in the tree (i.e., the structure of the tree remains unchanged but the tree is sorted.) In your pseudocode feel free to use any convenient access functions you would like, (e.g., root$[T]$, children$[n]$, parent$[n]$).

```
TreeSortify(n)
    if (isleaf[n]) return
    l ← argmax_{c∈children[n]} key[c]
    if key[n] < key[l] then
        key[n] ↔ key[l]
        TreeSortify(l)

TreeSort(T)
    for d ← height[T] down to 0 do
        for n ∈ nodes-at-depth[T, d] do
            TreeSortify(n)
```

b. (3 marks) What is the worst-case running time of your procedure for a tree with n nodes?

$O(n^2)$ when the tree is essentially a linked list.

Question 3. (10 marks)

a. (5 marks) Write the array containing elements $\{1, \ldots, 10\}$ for which quicksort will require the most key comparisons.

$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

b. (5 marks) Write the array containing elements $\{1, \ldots, 10\}$ for which quicksort will require the fewest key comparisons.

Assuming the pivot is chosen as the end element (as in the testbook and class notes), then $\{1, 3, 4, 2, 8, 7, 9, 10, 5\}$ will split as evenly as possible on each recursion.

Question 4. (10 marks) Exercise 8.1-4 from the textbook.

We can proceed in an identical fashion to the lower-bound analysis of standard sorting. First, count the number of unique permutations that could be returned by sorting. Notice that an element in the range $1 \ldots k$ must in the final sort remain in that range. So, all possible sorts just permute the elements in each subsequence. The total number of permutations of this form are $(k!)^{\frac{n}{k}}$. The number of leaves in any deterministic, comparison-based sorting algorithm must be at least the number of possible permutations. Therefore, the height of the tree (i.e., worst-case running time) must be at least,

$$\begin{aligned} \text{Height} &= \log(\#\text{ of Leaves}) = \log\left((k!)^{\frac{n}{k}}\right) \\ &= \frac{n}{k}\log(k!) \\ &\in \Theta\left(\frac{n}{k} k \log k\right) = \Theta(n \log k). \end{aligned}$$

Question 5. (20 marks)

a. (15 marks) Consider a robot moving in an $n \times n$ grid. It starts at the coordinate $(1, 1)$ and it moves either one square up or one square right, i.e., it increases either one but not both of its coordinates by exactly one (if not already equal to n). Eventually the robot will arrive at the coordinates (n, n). You also know, from past observations, the robot's probability of various moves. In particular, if the robot is at position (x, y), its next move will move it up with probability $u(x, y)$ and right with probability $1 - u(x, y)$. You want to prevent this robot from reaching its destination by placing a wall along its path (you cannot place a wall in $(1, 1)$ or (n, n)). Write a procedure to find the optimal place for a wall that will maximize the probability that the robot will enter the chosen location during its trip. Your algorithm should run in $O(n^2)$ time.

ROBOT-DP(n, u)
  $P[1,1] \leftarrow 1$
  for $i \leftarrow 1$ to n do
    if $(i \neq 1)$ then $P[i,1] \leftarrow P[i-1,1]$
    for $j \leftarrow 2$ to n do
      $P[i,j] \leftarrow P[i,j-1]u(i,j-1) + P[i-1,j](1-u(i,j-1))$
  return $\underset{(i,j)\notin\{(1,1),(n,n)\}}{\text{argmax}} P[i,j]$

b. (5 marks) Describe how you could adapt the above algorithm to answer the question, "where is the robot most likely to be after k moves?"

We instead return $\underset{(i,j):i+j=2+k}{\text{argmax}} P[i,j]$.

Question 6. (15 marks)

a. (12 marks) There are n trading posts on the Saskatchewan River. At any of these posts you can rent a canoe to be returned at any other post down-stream (it is next to impossible to paddle against the current). For each possible departure point i and each possible arrival point j the cost of a rental between i and j is $cost[i,j]$. However, it can happen that the cost of renting from i to j is higher than the total cost of a series of shorter rentals, in which case you can return the first canoe at some post k between i and j and continue the journey in a second canoe. There is no extra charge if you change canoes in this way. Give an efficient procedure to determine, given n and costs $cost[i,j]$, the minimum cost of a trip by canoe from post 1 to post n.

CANOE-DP$(n, cost)$
  $C[1] \leftarrow 0$
  $C[2] \leftarrow cost[1,2]$
  for $i \leftarrow 3$ to n do
    $C[i] \leftarrow \max\{cost[1,i], \max_{2\leq j<i} C[j] + cost[j,i]\}$
  return $C[n]$.

b. (3 marks) In terms of n, what is the running time of your algorithm?

The running time is $O(n^2)$.