Assignment 1 Solutions

COMP3121/9101 21T3

Released September 15, due September 29

This document gives model solutions to the assignment problems. Note that alternative solutions may exist.

1. You are given an array A of n positive integers. A pair of indices (i, j) where $i < j$ is said to be *consistent* if $A[j] - A[i] = j - i$.

   (a) (10 points) Design an algorithm which runs in *expected* $O(n)$ time and counts the number of pairs of consistent indices.

   (b) (10 points) Design an algorithm which runs in *worst case* $O(n \log n)$ time and counts the number of pairs of consistent indices.

   (a) We first rearrange the given equation in the form

   $$A[j] - j = A[i] - i.$$

   Define a new array B of length n, where $B[i] := A[i] - i$, so that (i, j) is consistent if and only if $B[i] = B[j]$. Creating array B takes $O(n)$ time.

   Next, we create a hash table to count the number of occurrences of each element in array B. For each of n elements of B, we perform one lookup and one insertion or update, each in $O(1)$ expected time.

   Each key of the hash table corresponding to a value $k \geq 2$ corresponds to an element of B which is repeated at k different indices. Every pair of these indices is consistent, so we add $k(k-1)/2$ to the total. For each key of the hash table, of which there are at most n, we update the answer in $O(1)$ time.

   This algorithm counts the number of consistent pairs in expected $O(n)$ time.

   (b) Rearrange the equation and construct array B as in part (a). Sort array B using merge sort in $O(n \log n)$ time.

   Now, we are guaranteed that any equal values in B are contiguous, and as in part (a) a run of k equal values contributes $k(k-1)/2$ consistent pairs. We make one pass through the array, maintaining a counter which tracks the number of occurrences of the current value being considered. This counter starts at 1 for $A[1]$, is incremented when consecutive values are equal and is reset to 1 when consecutive values are not equal. Before resetting from a value $k \geq 2$ to 1, we add $k(k-1)/2$ to the total. For each of n elements of B, we update the counter and if necessary the answer in $O(1)$ time.

   This algorithm counts the number of consistent pairs in $O(n \log n)$ time, dominated by sorting.

2. (20 points) You are given an array A of n positive integers, each no larger than m. You may assume that $m \leq n$. The *beauty* of an array is the fewest number of times that any integer from 1 to m inclusive appears in the array. For example, if $m = 3$ then the beauty of $\langle 1, 3, 1, 2, 3, 3, 2 \rangle$ is 2. However, if $m = 4$ the same array would have beauty 0.

   An index i is said to be *fulfilling* if $A[1..i]$ has strictly greater beauty than $A[1..i-1]$. Design an algorithm which runs in $O(n)$ and finds all fulfilling indices.

   We maintain a frequency table f to keep track of the number of occurrences of each $1 \leq j \leq m$. This array is constructed in $O(m)$ time. We also maintain a counter z of the number of zero values in the frequency table (initially m).

   Iterate through array A, starting from index 1. At step i, add one to the frequency $f[A[i]]$, and if the old value was zero, subtract one from z. If z becomes zero as a result, report a fulfilling index, subtract one from all frequencies and recalculate z.

   It is clear that the beauty of a prefix $A[1..i]$, i.e. the smallest frequency of any $1 \leq j \leq m$, is exactly the number of times the frequency table has been decremented. This guarantees that we report the correct indices as fulfilling.

   Each non-fulfilling index, of which there are up to n, is handled in $O(1)$ time. Each fulfilling index is handled in $O(m)$ time, as we need to update every entry of the frequency table and recalculate z. However, we claim that there are at most n/m fulfilling indices, so the algorithm runs in $O(n + m) = O(n)$ time (since $m \leq n$).

   Proof: Suppose there are $k > n/m$ fulfilling indices. Then the beauty of $A[1..n]$ is k, so $f[j] \geq k$ for each $1 \leq j \leq m$. However, the sum of all frequencies is then at least km, which is greater than n. This is a contradiction, so in fact there are at most n/m fulfilling indices.

3. You are given a string s of length n, constructed from an alphabet with k characters. You may assume that all k different characters appear at least once in s.

   A *substring* of s is a contiguous sequence of characters within s.

   Design an algorithm which runs in $O(n)$ time and finds the length of the shortest substring of s which contains all k different characters.

   A solution for the case $k = 3$ will earn up to 10 points.

   For the $k = 3$ case, we first claim that an optimal solution must be of the form $abb \ldots bc$.

   Proof: Let $t = s[i..j]$ be an optimal solution, where clearly $j - i \geq 2$.

   - $s[i]$ and $s[i+1]$ must be different, as otherwise $s[i+1..j]$ would be a shorter substring which still contains all characters. Denote $s[i]$ by a and $s[i+1]$ by b.
   - $s[j]$ must be the third character, say c, as otherwise $s[i..j-1]$ would be a shortest substring which still contains all characters.
   - all characters of $s[i+1..j-1]$ must be b. For $i+1 \leq x \leq j-1$, if $s[x] = a$ then $s[x..j]$ would be a shorter substring which still contains all characters, and conversely if $s[x] = c$ then this would be true of $s[i..x]$.

   We now break the string s into substrings consisting of one repeated character, and identify any such substrings which are preceded by and followed by two different characters. The answer is the length of the shortest such substring plus two.

   We can split up the string s in one pass, also recording the length of each substring of interest and checking the characters before and after it. We can thus calculate the answer in $O(n)$ time.

   For the general case, we first preprocess the string s into k linked lists, each containing the indices (in ascending order) where a particular character occurs in s. This requires one pass through the string, with $O(1)$ work per character.

   Now we maintain pointers into all k lists, representing the next occurrence of each character. As we traverse the string from left to right, we plan to update these pointers so that when we are up to index i, each pointer indicates the smallest index greater than or equal to i where the corresponding character occurs. In this way, the shortest substring starting at position i which contains all characters will end at the maximum of all values pointed to. The length is found by substracting $i - 1$ from this maximum, and the smallest such length is the answer.

   Initially, each pointer refers to the first item of the corresponding linked list, and the maximum of the pointed-to values is calculated in $O(k)$ time. As we move from index i to $i+1$ in s, the next instance of each character is the same for all except $s[i]$. We advance the pointer for that character in $O(1)$ time, and if necessary update the maximum of pointed-to values also in $O(1)$ time. Therefore the algorithm runs in $O(n+k) = O(n)$ time (since $k \leq n$).

4. (20 points) You have $2n+1$ ants along a line. Their positions are described by the real numbers $x_{-n} < x_{-n+1} < \ldots < x_{n-1} < x_n$. You will place food at a point x on the line, and all ants will walk along the line to reach the food. Find the value of x which minimises the total distance walked by all ants.

Define

$$f^{(n)}(x) = \sum_{i=-n}^{n} |x - x_i|,$$

the total distance walked by all ants if the food is placed at x.

Claim 1: The optimal position x is no less than x_{-n} and no greater than x_n.

Proof: Let $\epsilon > 0$. Then each term of $f_n\left(x_n + \epsilon\right)$ is positive, and hence greater than the corresponding term of $f^{(n)}\left(x_n\right)$. Therefore

$$f^{(n)}\left(x_n + \epsilon\right) > f^{(n)}\left(x_n\right),$$

so the minimum cannot occur at $x_n + \epsilon$. Similarly,

$$f^{(n)}\left(x_{-n} - \epsilon\right) > f^{(n)}\left(x_{-n}\right).$$

This concludes the proof of Claim 1.

We can therefore search for the optimal value x in the interval $[x_{-n}, x_n]$ only.

Claim 2: For $x_{-n} \leq x \leq x_n$, the distance walked by the first ant and the last ant add to a constant.

Proof: We can see that

$$|x - x_n| + |x - x_{-n}| = (x_n - x) + (x - x_{-n}) = x_n - x_{-n},$$

which is independent of x. This concludes the proof of Claim 2.

Therefore within the interval of interest, namely $[x_{-n}, x_n]$, minimising $f^{(n)}$ is equivalent to minimising $f^{(n-1)}$. This is equivalent to discarding the first and last ants.

We can continue in this way, each time first restricting the search interval to the first and last remaining ants and then discarding those two ants. This continues until we reach $f^{(0)}(x) = |x - x_0|$, which is minimised at $x = x_0$.

5. Determine if:

(I) $f(n) = O(g(n))$,

(II) $f(n) = \Omega(g(n))$, i.e. $g(n) = O(f(n))$,

(III) both (I) and (II), i.e. $f(n) = \Theta(g(n))$, or

(IV) neither (I) nor (II)

for the following pairs of functions, and justify your answer. Note that log denotes the natural logarithm, with base e.

(a) (6 points) $f(n) = n^{1+\log n}$; $g(n) = n \log n$;

(b) (8 points) $f(n) = n^{1+\frac{1}{2}\cos(\pi n)}$; $g(n) = n$;

(c) (6 points) $f(n) = \log_2 n^{\log(n \log n)}$; $g(n) = (\log n)^2$.

(a) Simplifying gives

$$\begin{aligned} f(n) &= n^{1+\log n} \\ &= n \cdot n^{\log n}. \end{aligned}$$

For $n \geq 3$, we have $\log n \geq 1$, so $f(n) \geq n^2$. Therefore $f(n) = \Omega(n^2)$. It follows that $f(n) = \Omega(g(n))$, but $f(n) \neq O(g(n))$, so only (II) applies.

(b) Simplifying gives

$$\begin{aligned} f(n) &= \begin{cases} n^{1+\frac{1}{2}\cos(2k\pi)} & \text{if } n = 2k \\ n^{1+\frac{1}{2}\cos((2k+1)\pi)} & \text{if } n = 2k+1 \end{cases} \\ &= \begin{cases} n\sqrt{n} & \text{if } n \text{ is even} \\ \sqrt{n} & \text{if } n \text{ is odd.} \end{cases} \end{aligned}$$

For $f(n) = O(g(n))$ to hold, we require $C, N > 0$ such that $f(n) \leq Cg(n)$ for all $n \geq N$. However, for any fixed C, the required inequality fails for all even integers greater than C^2, so no such N can exist.

For $f(n) = \Omega(g(n))$ to hold, we require $c, N > 0$ such that $f(n) \geq cg(n)$ for all $n \geq N$. However, for any fixed c, the required inequality fails for all odd integers greater than c^{-2}, so no such N can exist.

Therefore (IV) applies.

(c) Simplifying gives

$$\begin{aligned} f(n) &= \log_2 n^{\log(n \log n)} \\ &= \log(n \log n) \log_2 n \\ &= (\log n + \log \log n) \frac{\log n}{\log 2}. \end{aligned}$$

Now $\log 2$ is constant, and $\log n \leq \log n + \log \log n \leq 2(\log n)$ for large n, so $\log n + \log \log n = \Theta(\log n)$. Therefore $f(n) = \Theta(g(n))$, so (III) applies.