Assignment 1 – Solutions

COMP3121/9101 22T1

Released February 15, due March 3

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

1. (20 points) You are playing a level of the latest smash-hit kung-fu video game. The level is composed of a sequence of n rooms filled with enemies. The hero must go through all n rooms in order, fighting all the enemies in each room. The level is completed when the hero exits the final room. For each room i, you know the number of times a_i that the hero will die in that room.

   Furthermore, this game has a very interesting death mechanic. The hero starts at age 20, and each time they die, their age is increased by the value of the death counter, which is the number of times that the hero has died. For example, after the first, second and third deaths, the hero's age becomes 21, 23 and 26 respectively. Moreover, before the hero enters any room, you can reset the death counter to 0, though this ability may only be used *once* in the entire level.

   Design an algorithm which runs in $O(n)$ time and calculates the minimum age at which the hero can finish the level.

Solution

Method I

First sum up the number of deaths in the entire level ($S = \sum a_i$). Then, loop through each room, keeping track of how many deaths we have suffered so far (l), and how many deaths we still have to come ($r = S - l$). For each room, we consider using the reset ability in the current room; the final age of the hero would be

$$20 + (1 + 2 + \ldots + l) + (1 + 2 + \ldots + r),$$

which we can calculate using arithmetic series as

$$20 + \frac{l(l+1)}{2} + \frac{r(r+1)}{2}.$$

The smallest of these n values is the minimum age at which the hero can finish the level.

This is an exhaustive search, considering all n possible uses of the reset ability, so it must be correct.

S is calculated in $O(n)$ time. For each of n rooms, we update l and r in constant time, and calculate the final age also in constant time. Therefore the overall time complexity is $O(n)$.

Method II

As in Method I, but in each room instead of calculating the final age directly, we consider modifying the previous value. The final age after resetting in room k was given by

$$20 + \underbrace{\frac{1}{2}l(l+1)}_{L} + \underbrace{\frac{1}{2}r(r+1)}_{R},$$

so resetting in room $k+1$ gives final age

$$20 + \underbrace{\frac{1}{2}l^*(l^*+1)}_{L^*} + \underbrace{\frac{1}{2}r^*(r^*+1)}_{R^*},$$

where $l^* = l + a_k$ and $r^* = r - a_k$. Therefore we can update L and R via

$$\begin{aligned} L^* &= \frac{1}{2}\left[(a_k+l)(a_k+l+1)\right] \\ &= \frac{1}{2}\left[a_k^2 + (2l+1)a_k + l(l+1)\right] \\ &= a_k\left[a_k + l + \frac{1}{2}\right] + L, \end{aligned}$$

and similarly

$$R^* = a_k\left[a_k - r - \frac{1}{2}\right] + R,$$

so the final age changes by

$$a_k\left(a_k + l - r\right).$$

Since $a_k \geq 0$, the sign of this quantity is determined by

$$\begin{aligned} &a_k + l - r \\ &\quad = a_k + (a_1 + \ldots + a_{k-1}) - (a_k + a_{k+1} + \ldots + a_n) \\ &\quad = (a_1 + \ldots + a_{k-1}) - (a_{k+1} + \ldots + a_n), \end{aligned}$$

which is non-decreasing in k. Indeed, we need only find the first room k for which this quantity is non-negative, i.e. the smallest k such that

$$(a_1 + \ldots + a_{k-1}) \geq (a_{k+1} + \ldots + a_n).$$

We can perform a linear search to find this k, at which point the minimum final age can be computed directly.

Method III

As in Method II, but we claim that the optimal room is that which minimises $\left|\frac{S}{2} - l\right|$. This claim must be proven, using the proof below or equivalent.

The final age can be simplified as follows:

$$\begin{aligned}
&20 + \frac{l(l+1)}{2} + \frac{r(r+1)}{2} \\
&= 20 + \frac{l(l+1)}{2} + \frac{(S-l)(S-l+1)}{2} \\
&= 20 + \frac{l^2+l}{2} + \frac{(S-l)^2 + (S-l)}{2} \\
&= 20 + \frac{l^2 + l + S^2 - 2Sl - l^2 + S - l}{2} \\
&= 20 + \frac{S}{2} + \frac{S^2}{2} - Sl + l^2 \\
&= 20 + \frac{S}{2} + \frac{S^2}{4} + \left(\frac{S}{2} - l\right)^2,
\end{aligned}$$

which is minimised when $\left|\frac{S}{2} - l\right|$ is minimised, as required.

Method IV

As in Method IV, but we claim that the optimal room is that which minimises $|r-l|$. This claim is equivalent to that made in Method III, since

$$\left|\frac{S}{2} - l\right| = \left|\frac{l+r}{2} - l\right| = \left|\frac{r-l}{2}\right|.$$

2. You are given an array A consisting of n integers, each between 1 and M inclusive. You are guaranteed that no more than k distinct values appear in the array.

   You are required to identify which of the values between 1 and M appear in the array, and for each such value, how many times it appears. Design algorithms which solve this problem and run in:

   (a) (5 points) worst case $\Theta(n + M)$ time;

   (b) (5 points) worst case $\Theta(n \log n)$ time;

   (c) (5 points) worst case $\Theta(n \log k)$ time;

   (d) (5 points) *expected* $\Theta(n)$ time.

Solution

(a) First, we create a zero-initialised array B of size M, which will serve as a frequency table. Next, we iterate through array A. At index i, we record an instance of the value $A[i]$ by incrementing $B[A[i]]$. Finally, we iterate through B and report the pairs $(j, B[j])$ where $B[j]$ is nonzero, representing $B[j]$ occurrences of j in array A.

Creating array B takes $\Theta(M)$ time. Each of n entries of A is processed in constant time. Reporting the answer takes $\Theta(M)$ time. Therefore, this algorithm takes $\Theta(n + M)$ overall.

(b) First, we sort A using mergesort. Now, we begin iterating through the sorted array A. When we see a new value, we report it, and begin a counter at 1, which is incremented for each subsequent occurrence of that number and reported when a new number is seen.

After sorting the array, any identical elements are consecutive, so the frequencies are correctly counted by this iterative procedure.

Mergesort takes $\Theta(n \log n)$ time. Each of n entries of A is processed in constant time. Therefore, this algorithm takes $\Theta(n \log n)$ overall.

(c) We create a self-balancing binary search tree of key-value pairs, where the key is a number appearing in A and the value is its frequency in A. For each index i of A, we search the BST to determine whether $A[i]$ is an existing key. If so, we increment the corresponding value, and if not, we insert the key-value pair $(A[i], 1)$. Once we have exhausted A, we traverse the BST using DFS (or BFS) and report each key-value pair.

In a self-balancing BST with k keys, searching and insertion each take $\Theta(\log k)$ time. For each of n entries, we perform at most two such tree operations. Finally, reporting the answer takes $\Theta(k)$ time. Therefore, this algorithm takes $\Theta(n \log k)$ overall, since $k \leq n$.

(d) We create a hash table, where each key is a number appearing in A and the value is its frequency in A. For each index i of A, we search the hash table to determine whether $A[i]$ is an existing key. If so, we increment the corresponding value, and if not, we insert it with value 1. Once we have exhausted A, we iterate over the hash table and report each key with its corresponding value.

In a hash table with k keys, searching and insertion each take expected $\Theta(1)$ time. For each of n entries, we perform at most two such hash table operations. Finally, reporting the answer takes $\Theta(k)$ time. Therefore, this algorithm takes expected $\Theta(n)$ overall, since $k \leq n$.

3. (20 points) You are given an n by n matrix of *distinct* integers. A *summit* is an element that is larger than all its neighbours in each of the (up to) 4 directions. Design an algorithm which runs in $O(n \log n)$ time and finds the location of *any* summit.

   Note that integers in the corners and boundaries can be summits, and there will always be at least one summit, the maximum element. However, you do not necessarily have to find this specific summit.

Solution

Firstly, note that we can always find a summit that is at least as big as any particular element. Starting at that element, keep stepping towards any neighbouring element that is larger. This process terminates when we are unable to step in any direction, so by definition we have reached a summit.

We can use this observation to develop a divide and conquer solution. First, pick the middle column, and perform a linear search to find the maximum element out of that column and the (up to) two on either side. If this maximum element is in the central column, then it must be greater than all its neighbours and we are done.

If the maximum element is in the left column, then by our first observation, we know that there must be a summit somewhere on the left of the central column – since that element is larger than everything in the central column, running our stepping strategy from the maximum will never cross from the left side over to the central column or anything to the right of it, and since we're guaranteed to find a summit, there must be a summit on the left side.

This reasoning also applies to the right side, in which case we restrict our search to half of the matrix.

As such, we can use divide and conquer, at each stage testing the middle three columns. We terminate at subproblems with three columns or fewer, where the linear search finds a summit. After each stage, the number of columns is halved, so there are at most $\log_2 n$ stages, each requiring at most $3n$ comparisons. Therefore the overall time complexity is $O(n \log n)$.

Note however that this is not the fastest possible algorithm. If we first test the middle three columns and then analogously the middle three rows, we will have reduced the search space from n-by-n to $\frac{n}{2}$-by-$\frac{n}{2}$ using approximately $3n + \frac{3n}{2}$ comparisons. Therefore, the time complexity is given by the following recurrence:

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n),$$

so by Case 3 of the Master Theorem $T(n) = \Theta(n)$.

4. (20 points) You have n poles of distinct heights spaced in a line, the ith of which is h_i metres tall. You can wire together two poles $i < j$ only if they are *both* taller than all poles in between them, that is, for each $i < k < j$, $h_k < h_i$ and $h_k < h_j$.

   Design an algorithm which runs in $O(n \log n)$ time and calculates the total number of pairs of poles that you can wire together.

Solution

Split the poles into two halves, so that half the poles are on the left and half the poles are on the right. Then the number of pairs you can make is equal to the number of pairs completely in the left half, plus the number of pairs completely in the right half, plus the number of pairs containing one pole on the left and one pole on the right.

We use divide and conquer, counting pairs within each half recursively. We now find the number of wires bridging the midpoint, using a two pointers method.

1. Let $i = \lfloor n/2 \rfloor$ and $j = i + 1$, the indices of the last entry of the left half and the first entry of the right half respectively.
2. Add one to the answer, representing the wire between these consecutive poles.
3. Create two variables l and r representing the largest height seen so far in each half. Initially, these will be the heights of poles i and j respectively, i.e. $l = h_i$ and $r = h_j$.
4. Now we repeat the following until termination:
   - If $l < r$, decrement i until:
     - i reaches zero, in which case we terminate, or
     - $h_i > l$, in which case we update l to the value h_i and add one to the answer, representing the wire between poles i and j.
   - If instead $r < l$, increment j until:
     - j exceeds n, in which case we terminate, or
     - $h_j > r$, in which case we update r to the value h_j and add one to the answer, representing the wire between poles i and j.

To prove the correctness of this algorithm, we must guarantee that it counts all wires crossing the midpoint. Such a wire must join pole i to pole j, where $i \leq \lfloor n/2 \rfloor$ and $j > \lfloor n/2 \rfloor$ and poles $i + 1$ to $j - 1$ have height less than $\min(h_i, h_j)$. We need to confirm that updates to l and r happen at all such pairs (i, j) and only these pairs.

Claim: If l is not updated at index i, then there are no wires (i, j) crossing the midpoint.

Proof: This occurs when pole i is shorter than some pole k where $i < k \leq \lfloor n/2 \rfloor$. This pole obstructs any wires (i, j) crossing the midpoint.

Similarly, if r is not updated at index j, then there are also no wires (i, j) crossing the midpoint.

If we say that l and r are updated at *strong* poles, then any wire crossing the midpoint must join two strong poles. However, not all pairs of strong poles can be connected by a wire.

Claim: If there is a wire (i, k) where $h_i < h_k$, then there are no wires (i, j) where $k < j$.

Proof: Pole k is taller than pole i, so it obstructs any such wires.

Therefore, when we count a wire (i, k) where pole i is taller than pole k, any other wires from pole i have already been counted. There may still be other wires to pole k, which we can find by iterating left of pole i. If we find a strong pole left of pole i, it is guaranteed to have a wire to pole k, at which point we can repeat the procedure, having counted all wires within the current range of poles. If instead there is no strong pole left of pole i, we have counted all wires so we terminate.

In the two pointers procedure, each update to i or j is processed in constant time, and they can be updated only n times in total. Therefore, the loop takes $\Theta(n)$ time in total. The recurrence is therefore

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n),$$

so the algorithm runs in $O(n \log n)$ overall.

Note that there is also a $\Theta(n)$ time solution using sorted stacks.