Homework 2 Solutions

Divide-and-Conquer Algorithms, Sorting Algorithms, Greedy Algorithms

Problem 1 [20 points]

Solving most of these recurrences corresponds to a simple application of the Master Theorem. You can apply either the version from the DPV book with the O notation or the one from the CLRS book (also included in the resources document available on Sakai) with the Θ notation. Below we are following the CLRS book description of the Master Theorem.

A. $T(n) = 2T(\frac{n}{4}) + \sqrt{n} : a = 2, b = 4, f(n) = \sqrt{n}$

Apply case 2 of the Master Theorem, yielding $\boxed{\Theta(\sqrt{n}\ log\ n)}$.

B. $T(n) = 7T(\frac{n}{2}) + n^2 : a = 7, b = 2, f(n) = n^2$

Apply Case 1 of the Master Theorem, yielding $\boxed{\Theta(n^{log_2 7}) \approx \Theta(n^{2.81})}$.

C. $T(n) = T(n-1) + n$

Here, the Master theorem does not apply, as we are not dividing the size of the problem for each step. Instead, we can observe the running time by inspection. There are approximately n recurrences each costing $\Theta(n)$, or more specifically, $T(n) = n + (n-1) + (n-2) + ... + 1 = \boxed{\Theta(n^2)}$.

D. $T(n) = T(\sqrt{n}) + 1$

Again, the Master Theorem does not directly apply; however, we can apply a variable substitution so that we will be later able to apply the Master Theorem. Let $k = log_2 n$, or equivalently, $n = 2^k$. Then $R(k) = T(2^k)$. This yields the recurrence:

$$\begin{aligned} R(k) &= T(2^k) \\ &= T(\sqrt{2^k}) + 1 \\ &= T(2^{k/2}) + 1 \\ R(k) &= R(k/2) + 1 \end{aligned}$$

Now the recurrence $R(k)$ is of a form $R(k) = R(k/2) + 1$, which allows us to apply the Master theorem, i.e., $a = 1, b = 2, f(n) = 1$. Apply Case 2 of the Master Theorem to obtain $R(k) = \Theta(log\ k)$. Note however that $T(n) = R(log_2\ n)$ from above. Thus, $\boxed{T(n) = \Theta(log\ log\ n)}$.

Problem 2 [20 points]

We begin by formalizing some components of the problem. In particular we model testing a nut with a screw as an operator <> as follows:

$$a_i <> b_j = \begin{cases} 1 & : \text{if nut } a_i \text{ is too big for screw } b_j \\ 0 & : \text{if nut } a_i \text{ matches screw } b_j \\ -1 & : \text{if nut } a_i \text{ is too small for screw } b_j \end{cases}$$

where a_i is the i-th nut in the set of n nuts, and b_j is the j-th screw in the set of n screws. From this, we can begin to draw the intuition that this problem is not much different than a comparison-based sorting algorithm.

A. [10 points]

We follow the same approach as discussed in class, and construct a decision tree using the above operator. In traditional comparison-based sorting, there are two outcomes of a comparison, while in this case, there are three. This means the decision tree will be ternary (have a branching factor of three) rather than binary. To parallel the proof for sorting, we again seek to find a *lower bound* on the height of the decision tree.

The first question to ask is, how many leaves are there - or equivalently, how many different ways can the nuts be mapped to the screws. If we arrange the screws in a sequence, then one out of n nuts can fit the first screw, $n-1$ nuts are left to fit the second screw, $n-2$ nuts for the third screw, and so on. The number of possible mappings is thus the same as for sorting, namely $n!$. So we have at least $n!$ reachable leaves. Now a ternary tree of height h has no more than 3^h leaves, therefore we must have $n! \leq 3^h$. Taking logs on both sides, we have $h \geq log_3 n! = \Omega(nlog_3 n)$. Hence the worst case number of comparisons, is $\Omega(nlogn)$.

B. [10 points]

The following solution for this problem mimics Quicksort.

i. Randomly pick a screw b_j.

ii. Compare all nuts to b_j, and group all nuts smaller than b_j (i.e., $a_i <> b_j = -1$) into a set N_1, all nuts larger than b_j (i.e., $a_i <> b_j = 1$) into a set N_2. Let a_i be the nut that matches b_j (i.e. $a_i <> b_j = 0$).

iii. Now compare all screws (except b_j) to a_i and partition them into two sets B_1, B_2 as above.

iv. Recurse on N_1, B_1.

v. Recurse on N_2, B_2.

The total work done in steps (i), (ii) and (iii) is $\Theta(n)$. We thus get the same recurrence as for randomized quicksort:

$$T(n) = \frac{1}{n}\sum_{q=0}^{n-1}(T(q) + T(n-q-1)) + \Theta(n)$$

Thus the solution is the same as shown in the class for randomized quicksort, namely, $O(nlogn)$.

Problem 3 [25 points]

A. [10 points]

Because the n lists are already sorted, we can simply employ the merge operation from `mergesort`. To merge two lists of sizes m and n respectively, the merge operation takes $O(m+n)$. Define the recurrence $T(i)$ to be the time to merge lists 1 to i. There are two parts to this recurrence:

- We need the time it took to merge all of the lists from 1 to $i-1$. This is the recursive part, of cost $T(i-1)$.
- We also need to merge the next list of size k with the existing list of size $k(i-1)$.

Thus the recurrence is:

$$T(i) = T(i-1) + O(ki)$$

Unfortunately, this is not of the form we can use the Master Theorem on, but does yield an intuitive answer:

$$T(n) = \sum_{i=2}^{n} O(ki) = O\left(k \sum_{i=2}^{n} i\right) = O(n^2 k)$$

B. [15 points]

The high level algorithmic idea is to recursively divide the lists into two groups, each of $k/2$ lists. Recursively merge the lists within the 2 groups and finally merge the resulting 2 sorted lists. This final list yields the desired result.

Pseudocode

```
Input: n lists of k elements
Output: The merged, sorted list of kn elements

procedure multiple_merge(arrays)
  if size(arrays) == 1:
    return arrays.pop()
  else
    first_half = first half of the list arrays
    second_half = second half of the list arrays
    return merge(multiple_merge(first_half), multiple_merge(second_half))
```

Consider now the running time of this algorithm. Define the recurrence $T(i)$ for the i-th level of recursion. At the i-th level, there are $(n/2^i)$ arrays, each of length $2^i k$. Therefore, there are $(n/2^{i+1})$ merges to perform. This requires $(n/2^{i+1}) \cdot O(2^i k + 2^i k) = O(kn)$ time for this level.

To get the overall runtime, we sum over all levels. There are *log n* levels; however, no work is required on the last level, because the lists are already sorted. Therefore, we compute the total running time as:

$$T(k, n) = \sum_{i=0}^{log\ n-1} O(nk) = O(nklog\ n)$$

To argue the correctness of the approach, we present two arguments.

- The algorithm terminates. At each step of the loop, we are dequeuing 2 lists, merging them and enqueuing the result. Therefore the size of the queue is decremented by 1 at each step. Eventually, we will have the size of the queue being lower than 1 and the algorithm will terminate.
- The algorithm returns the right result. As we merge all the lists given as input into one single and the order of merging them does not matter, the output is the merging of all the lists.

Problem 4 [20 points]

To do this efficiently, we must use a linear-time sorting algorithm. Due to the fact that we are sorting based on distance, which is a real-valued and generally non-integer, the bucket sort makes the most sense for this solution. A naïve approach would create buckets along a fixed resolution according to d; however, this would not yield linear time sorting in general, as the number of points ending up in the last bucket will be significantly higher, since the outer-most shell according to the fixed resolution subdivision will have much larger volume than the other shells. The goal then is to define the buckets so that each corresponds to a subset of the sphere of equal volume.

The volume of a sphere is $(4/3)\pi r^3$. For our sphere of radius R, we wish for each of n buckets to take points from shells of the sphere with equal volume. It therefore follows that the volume of each bucket/shell should be $[(4/3)\pi R^3]/n$. From this, we must deduce the radii, r_i, which give these volumes. We provide the first two radii, and come to a conclusion about the form of the radius. For the first radius, it must be that:

$$\frac{4}{3}\pi r_1^3 = \frac{4}{3n}\pi R^3$$
$$r_1^3 = \frac{1}{n}R^3$$
$$r_1 = \sqrt[3]{\frac{R^3}{n}}$$

Similarly, we can solve for r_2:

$$\frac{4}{3}\pi(r_2^3 - r_1^3) = \frac{4}{3n}\pi R^3$$
$$\frac{4}{3}\pi r_2^3 = \frac{4}{3n}\pi R^3 + \frac{4}{3n}\pi R^3$$
$$\frac{4}{3}\pi r_2^3 = 2 \cdot \frac{4}{3n}\pi R^3$$
$$r_2^3 = \frac{2}{n}R^3$$
$$r_2 = \sqrt[3]{\frac{2R^3}{n}}$$

This results in a pattern of the form:

$$r_i = \sqrt[3]{\frac{i \cdot R^3}{n}}$$

and it can be verified that in general:

$$\frac{4\pi}{3}(r_{i+1}^3 - r_i^3) = \frac{4}{3n}\pi R^3$$

One critical element still missing though is being able to efficiently place a given sample into its appropriate bin. Calculating the distance of the point to the origin is considered to be constant time in this setting, as it does not depend on the number of samples p. The straightforward way to do this is to compute the distances, d_i, and iteratively compare them to the increasing value of the radii; however, this would result in an overall running time of $\Theta(p \log n)$, where it's expected that $n = O(p)$.

To compute the index, we build off the fact that if $r_i \leq d < r_{i+1}$, then the sample belongs in bucket i. We produce the following derivation:

$$\sqrt[3]{\frac{i \cdot R^3}{n}} \leq \sqrt{x^2 + y^2 + z^2}$$
$$\frac{i \cdot R^3}{n} \leq (x^2 + y^2 + z^2)^{\frac{3}{2}}$$
$$i \leq \frac{n(x^2 + y^2 + z^2)^{\frac{3}{2}}}{R^3}$$
$$i = \left\lfloor \frac{n(x^2 + y^2 + z^2)^{\frac{3}{2}}}{R^3} \right\rfloor$$

Computing i will take some effort, but it will be constant time relative to the number of samples, therefore, we have attained a sorting algorithm which sorts points in a sphere in $\Theta(p)$ time.

Problem 5 [25 points]

A greedy algorithm will efficiently solve this problem. Consider the state of the greedy algorithm after $i-1$ hours to be the number of hours devoted to each of the p problems up to that point: $x(i) = \{x_1(i), x_2(i), x_3(i), \ldots x_p(i)\}$. Initially, we start with assigning no hours to any problem, or $x(1) = \{0, 0, 0, \ldots, 0\}$. The amount of additional points the algorithm can receive for devoting the i-th hour to problem p is then: $G_p(x_p(i)+1)$.

The obvious greedy choice is to choose the problem $p^*(i)$ to work on, such that $p^*(i) = argmax_p G_p(x_p(i)+1)$. Intuitively, this means the algorithm works on whatever problem will give the most points for devoting one more hour on it given the already allotted hours.

Greedy choice property: We can show that there is an optimal solution that contains the greedy choice at hour h. Consider that at hour h we are making a different choice p^{opt} than the greedy, which corresponds to an optimal solution OPT. Then we can show that we can replace p^{opt} with $p^* = argmax_p G_p(x_p(h)+1)$ in OPT so as to define a new solution OPT^*, which is also optimal. The switch of p^{opt} with p^* definitely results in a better payoff at time h, so this switch is of no concern regarding the optimality of OPT^*. If the optimal solution OPT contained future assignments to the p^* problem past time h, we can just replace the first occurrence of p^* in OPT with p^{opt} and the solution OPT^* will be getting the same grade payoff as OPT. If the optimal solution OPT contained additional assignments to problem $p^{opt}(h)$, these future assignments will be actually providing higher or equal grade payoff in OPT^* than in OPT due to the monotonicity of the G_p function. Thus, overall there is an optimal solution that contains the greedy choice at an hour h.

Optimal Substructure: Assume that you have an optimal distribution of time $x^{opt}(h)$ for the first $h-1$ hours. Then by combining $x^{opt}(h)$ with the greedy choice $p^*(h)$ for the h^{th} hour, we can get an optimal distribution of time $x^{opt}(h+1)$ for the first h hours. This is based again on the property that $p^*(h) = argmax_p G_p(x_p(h)+1)$ and can be used to inductively show that the greedy choice always results in the optimum solution for increasing values of h.

An efficient implementation of this algorithm would use a max heap which takes as elements pairs of the form $(p, G_p(x_p+1))$, keyed on the second value. Use the divide-and-conquer `Heapify` algorithm (for example, see CLRS Section 6.3), which runs in $O(n)$ to build the initial heap. Then, we have a total of H hours to allocate, so we will perform H iterations. In each iteration, we remove the maximum element from the heap, let's say it is $(q, G_q(x_q+1))$, then increment x_q, and finally re-add the element $(q, G_q(x_q+1))$, with the updated value of x_q. The process of updating this element is $O(log\ p)$. Thus, our total running time will be $O(p + H\ log\ p)$.