CS 3510 — Algorithms, Spring 2022

Homework 1B Solutions

Divide and Conquer
Given an array $A[n]$ whose elements are odd numbers (positive and negative). Suppose that n is a power of 2, the elements of $A[n]$ are sorted and distinct. Design an algorithm to check whether there is at least one element in the array satisfies $A(i) = 2i + 5$ where $1 \leq i \leq n$ is the index of the i^{th} element in the array.

If such an element exists, your algorithm should return `yes` and otherwise return `no`. Ensure the running time is $O(\log n)$

Description: Our algorithm works with the following steps:

1. Calculate the middle index by dividing array size by 2
2. If the middle element ($A[i]$) is greater than $2i + 5$ where i is the index from the prior step, then reduce your search space to the left half (excluding middle element) of the array.
3. If the middle element is less than $2i + 5$ (computed above), reduce your search space to the right half (excluding middle element) of the array.
4. If the middle element is exactly equal to $2i + 5$, then return `yes`. This step also runs during a base case when only a single element is in the list.
5. If there are no more elements in your search space, return `no`

Justification: This algorithms works by using a few facts from the question and a modified binary search. If the current element is greater than $2i + 5$ (where i is the index of that element), each element to the right of $2i + 5$ will also be greater than $2k + 5$ where k is their relevant index.

If we treat $2i + 5$ as a linear map, then each change in 1 for the domain results in a change of 2 for the codomain, and each adjacent value is distinct and odd meaning the next element is 2 greater than the current element.

This works without loss of generality for the left half of the array if the element is less than $2i + 5$.

Runtime: During this algorithm, each iteration requires $O(1)$ non-recursive work as retrieving the middle index, computing $2i + 5$, and comparing the middle element to that computed number takes constant time with respect to the size of the array.

This $O(1)$ non-recursive work enables the algorithm to reduce the search space by $1/2$, hence giving a recurrence relation of:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

By using Master's Theorem, $a = 1, b = 2, d = 0$ giving a runtime of $O(\log n)$.

Function Analysis
Assume n is a power of 3. Assume we are given an algorithmic routinr $f(n)$ as follows:

```
function $f(n)$:
if $n > 1$:
  for $i$ in range($n$):
    for $j$ in range($n$):
      print("dividing")
  $f(n/3)$
  $f(n/3)$
  $f(n/3)$
else:
  print("conquered")
```

Part A: What is the running time of $T(n)$ for this function $f(n)$?

Solution: This function non recursively does $O(n^2)$ work and calls itself 3 times with input size $n/3$ giving a runtime of:

$$T(n) = 3T\left(\frac{n}{3}\right) + O(n^2)$$

By Master's Theorem, we can conclude $T(n) = O(n^2)$.

Part B:
How many dividings will this function print? How many conquereds will this function print? What is their relation to the running time $T(n)$?

Solution: The recurrence relation from above can be expanded as so:

$$\begin{aligned}
T(n) &= 3T\left(\frac{n}{3}\right) + cn^2 \\
&= 3\left(3T\left(\frac{n}{9}\right) + c\left(\frac{n^2}{9}\right)\right) + cn^2 \\
&= 9T\left(\frac{n}{9}\right) + cn^2 + c\frac{n^2}{3} \\
&= 9T\left(\frac{n}{9}\right) + cn^2\left(1 + \frac{1}{3}\right) \\
&\vdots \\
&= 3^iT\left(\frac{n}{3^i}\right) + cn^2\left(1 + \frac{1}{3} + \cdots + \frac{1}{3^{i-1}}\right)
\end{aligned}$$

We are curious when $T(n/3^i) = T(1) \implies i = \log_3 n$, hence

$$\begin{aligned}
&= 3^{\log_3 n}T(1) + cn^2\left(1 + \frac{1}{3} + \cdots + \frac{1}{3^{\log_3 n-1}}\right) \\
&= nT(1) + cn^2\left(1 + \frac{1}{3} + \cdots + \frac{1}{3^{\log_3 n-1}}\right)
\end{aligned}$$

Interpreting this in context, we know that conquered is only printed during the base case (when the input to the function is 1). In other words, whenever we see $T(1)$, we will print conquered. In the above expanded recurrence relation, there are n conquered printed since $T(1)$ occurs n times. For dividing, the second term can be additionally simplified using a partial geometric sequence. The number of dividing is equal to:

$$n^2 \sum_{i=0}^{\log_3 n} \frac{1}{3^i} = n^2\left(\frac{1-(1/3)^{\log_3 n-1}}{1-1/3}\right) = \frac{3}{2}(n^2 - n)$$

The relationship between the number of dividing and conquered is given by the expansion of the recurrence relation. The sum of the terms that relate to conquered ($nT(1)$) and dividing is equal exactly to the running time.

Generalized Merge Operation
Given k sorted lists, each of size n, we want to design an algorithm that will do a k-way merge and produce one final sorted list with the elements of all of the individual lists.

Design a D&C algorithm for this problem. What is its running time in Big O?

Description: The algorithm begins by determining if the number of lists passed in is 1. In the case only 1 list is passed in, we simply return this list back.

Otherwise, the algorithm recursively calls itself on the left and right halves of the list of lists such that the input size to the function call is $k/2$ lists (each list still having n elements).

The recursive call returns a $k/2$-way merge which we merge together using the blackbox merge operation in $O(n)$ time. Finally, we then return this merged array.

Justification: Our claim is that a list of lists of size k can be merged using our algorithm. We prove this by using a strong induction. First, the base case can be proved by noticing that the k- way merge works for both 1 and 2 lists. When 2 lists are passed in, the recursive calls of size 1 return the arrays and are merged (correctly) using the Merge blackbox routine.

Now, we can assume the strong inductive hypothesis from $j = 1 \to k/2$ where j is also a power of 2. The hypothesis states any function call for this method with j sized input will return a j-way merge of the input that is sorted of length jn.

In the k sized input, two recursive calls are made with $k/2$ input size and by the inductive hypothesis, we can assume these two recursive calls return a properly sorted array with size $nk/2$.

Finally, since the Merge blackbox function correctly merges two arrays, then it merges these two sorted arrays to give a single k-way merged sorted array of size nk. Therefore, for any power of 2, we have proved through strong induction that this algorithm is correct. $\square$

Runtime: The runtime of this algorithm is given by the following recurrence relation:

$$T(n, k) = T\left(n, \frac{k}{2}\right) + O(nk)$$

This is because each recursive call the method with $k/2$ lists where each list has the same size (always n). Each merge operation takes $O(nk)$ work.

Solving this recurrence (or using inspection) shows there are $\log k$ levels of recursion where each level does roughly $O(nk)$ work, hence the final runtime is:

$$O(nk \log k)$$