Homework 4: Divide & Conquer

Instructions: **Due 02/07/25 11:59pm**

- Please type your solutions using LaTeX or any other software. Handwritten solutions will not be accepted.
- Please try to write concise responses.
- You should not use pseudocode to describe your algorithms.
- Unless otherwise stated, saying log means base 2.
- Comparisons and basic arithmetic (addition, subtraction, bit-shifting) operations take $\mathcal{O}(1)$ time.

Q1 We are given an algorithm $f(n)$ as follows. Assume the input n to f is a power of 6.

```
f(n):
    if n > 1 then
        for i := 1 to 34 (inclusive) do
            f(n/6)
        for i := 1 to n^2 do
            print("meow")
        f(n/6)
    else
        print("woof")
```

We are also given an algorithm $g(n)$ as follows. Assume the input n to g is a power of 4.

```
g(n):
    if n > 1 then
        for i := 1 to 5 (inclusive) do
            g(n/4)
        for i := 1 to n do
            print("Hello")
    else
        print("World")
```

(a) What is the runtime of $f(n)$? Write and solve a recurrence.

We see that for a given n we first run $f(n/6)$, 34 times. After than we have a loop running for n^2 times and then we call $f(n/6)$ again. So we can write the recurrence as follows,

$$T(n) = 35 \cdot T(n/6) + O(n^2)$$

Now to solve the recurrence first we see that the total leaves in our recurrence tree is $35^{\log_6(n)}$ or $n^{\log_6(35)}$. We see that,

$$\log_6(35) < \log_6(36) = n^2$$

So using master theorem as $d > \log_b(a)$ the runtime of our algorithm is,

$$O(n^2)$$

(b) What is the runtime of $g(n)$? Write and solve a recurrence.

We see that for a given n we run $g(n/4)$, 5 times. After than we have a for loop printing a statement n times. So we can write the recurrence as follows,

$$T(n) = 5 \cdot T(n/4) + O(n)$$

Now we solve this recurrence. We know the total leaves of our tree is $5^{\log_4(n)} = n^{\log_4(5)}$. Now we notice that,

$$\log_4(5) > \log_4(4) = 1 \implies n < n^{\log_4(5)}$$

Using the master theorem we see that the run time of our algorithm is,

$$O(n^{\log_4(5)})$$

Q2 For any number n, such that n is a power of 2, the Star matrix, S_n is defined recursively as follows:

if $n = 1 \rightarrow S_n = (1)$

if $n > 1 \rightarrow S_n = \begin{pmatrix} 3S_{\frac{n}{2}} & I_{\frac{n}{2}} \\ S_{\frac{n}{2}} & -2S_{\frac{n}{2}} \end{pmatrix}$

Where I_k denotes the $k \times k$ identity matrix.

You will design an efficient *divide-and-conquer* algorithm that performs the following task:

Input: A number n that is a power of 2, and a vector v of length n
Output: The matrix-vector product $S_n \cdot v$

(a) Describe your algorithm.

Note: Feel free to merge (a) and (b) if it's easier to explain that way.

Let $f(S_n, v)$ be our algorithm that will calculate the required result for a given n. First we check the base case when $n = 1$ for which we directly return v. Now let v_1 be the vector with the first $n/2$ elements of v and v_2 the vector with the last $n/2$ elements of v. Now for a given n we recursively call our function as follows,

$$u_1 = f(S_{n/2}, v_1)$$
$$u_2 = f(S_{n/2}, v_2)$$

Now we compute the top block $3u_1 + v_2$ and the bottom block $u_1 - 2u_2$ and we return the concatenated vector,

$$\begin{bmatrix} 3u_1 + v_2 \\ u_1 - 2u_2 \end{bmatrix}$$

(b) Show the correctness of your algorithm.

Our algorithm works because it exploits the fact that we can decompose matrix multiplication by dividing the matrix into different parts and computing the matrix multiplication of the smaller parts. So we have,

$$S_n = \begin{pmatrix} 3S_{\frac{n}{2}} & I_{\frac{n}{2}} \\ S_{\frac{n}{2}} & -2S_{\frac{n}{2}} \end{pmatrix}$$

And we write,

$$v = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$$

Where v_1 and v_2 are the top and bottom half respectively. We see that when we do the multiplication it decomposes into the following,

$$S_n v = \begin{pmatrix} 3S_{\frac{n}{2}} v_1 + I_{\frac{n}{2}} v_2 \\ S_{\frac{n}{2}} v_1 - 2S_{\frac{n}{2}} v_2 \end{pmatrix}$$

So we see that our problem is subdivided into two parts where we need to compute the top half of our

solution and the bottom half of our solution and we just need to concatenate it together. Each half involves two matrix multiplications. But we notice that out of the 4 matrix multiplications there exists only 3 unique ones, one of which is just the identity multiplication. Hence we only have to compute two matrix multiplications,

$$S_{n/2}v_1 \text{ and } S_{n/2}v_2$$

.To get the solutions for the top half and bottom half. We also explicitly return v for the base case when $n = 1$. Hence by recursively applying this algorithm we can compute $S_n v$ optimally.

(c) Provide a recurrence for your algorithm and use it to analyze the runtime.

We see that for a given n we recursively call our function twice with $S_{n/2}$ as inputs. We also have a constant term for adding and multiplying the vectors. So our recurrence will look like,

$$T(n) = 2T(n/2) + O(n)$$

We first see that the number of leaves in our branching tree is around $2^{\log_2 n} = n$. Which is also equal to our constant work of n at each function call. Hence using master theorem the runtime of the algorithm is,

$$O(n \log(n))$$

Q3 You are given two integer lists $A \in \mathbb{Z}^m$ and $B \in \mathbb{Z}^n$ of sizes m and n respectively. Both are sorted in ascending order. We want to find the k^{th} smallest element in the sorted $A \cup B$.

You will design an algorithm that performs the following task in $\mathcal{O}(\log(\min(m, n)))$ time:

Input: two sorted lists $A \in \mathbb{Z}^m$ and $B \in \mathbb{Z}^n$, some $k \in [1, n + m]$
Output: The k^{th} smallest element in the sorted $A \cup B$

(a) Describe your algorithm.

Note: Feel free to merge (a) and (b) if it's easier to explain that way.

Note: The arrays are 0-indexed. $A[0]$ = first element in the array.

Hint: Assume without loss of generality that A is smaller. Perform binary search in A

First we assume that A is smaller that B. Now our base case is if A is empty then we return $B[k-1]$ and if $k = 1$ we return $\min(A[0], B[0])$. Now first let,

$$i = min(m, \lfloor k/2 \rfloor)$$
$$j = k - i$$

Now check the following conditions, $A[i] <= B[j-1]$ if this is the case then we recursively call our algorithm with $f(A[i : m], B, k - i)$ and return this. Now if $B[j] <= A[i-1]$ then we call $f(A, B[j : n], k - j)$ and return this.

(b) Show the correctness of your algorithm.

Our algorithm works because we notice that the kth elements is such that if there are i elements in A and j elements in B such that $k = i + j$ and that each of those i elements are smaller than every elements in B outside of the first j and similarly each of the j elements are smaller than every element in A outside of the first i. If this is the case then we can conclude that our solution would be the max(A[i - 1], B[j - 1]). So our algorithm is constructed to finding this pair of indexs i, j such that it satisfies this condition. So if $A[i]$ (i + 1)'th element in A is smaller than $B[j-1]$ (j'th element in B) this means that our solution exists the right side of A (right of i) as there are smaller elements than the k elements we considerd (i from A and j from B). Similarly we discard the left side of B when we see that the $B[j]$ element is smaller than $A[i-1]$.

(c) Analyze the runtime.

We see that in each function call for an arbitrary n we discard at lest $\lfloor k/2 \rfloor$ elements which we know is smaller than m. Hence we have,

$$T(m) = T(m/2) + O(1)$$

Now using master theorem we can easily see that this is a $\log(m)$ runtime algorithm. Hence our

> algorithm would have a runtime of $\log(min(m, n))$

Q4 We're going to be doing calculus on arrays! Let's say we have an array $A \in \mathbb{Z}^n$ containing distinct integers. A *local maximum* of the array is an entry that is greater than or equal to both its neighbors. That is, $A[i] \geq A[i-1]$ and $A[i] \geq A[i+1]$. For the two boundary elements of the array, they just need to be compared to their respective single neighbors.

You will design an algorithm that performs the following task in $\mathcal{O}(\log(n))$ time:

Input: some array $A \in \mathbb{Z}^n$ containing distinct integers
Output: a local maximum of A

(a) Describe your algorithm.

Note: Feel free to merge (a) and (b) if it's easier to explain that way.

Let our algorithm be $f(A_n)$ that finds the local maxima of our array A. Our base cae is that if the array is of length 1 then we return that value. Now we start and compute the middle element.

$$m = \lfloor n/2 \rfloor$$

And then we check if it is a local maxima. So if the mth element is greater than or equal to the $m-1$ or $m+1$ element then we return it. Now if it is not then at least one of the two elements are greater than the middle element. If this is the case we call our algorithm for that side of the array so if $A[m+1] > A[m-1]$ then we recurs to the right and return $f(A[m+1 : n-1])$ inclusive, else we return f(A[0: m - 1]).

(b) Show the correctness of your algorithm.

The idea is that whatever element is larger than the middle element there will exists a local maxim in that side of the array. Because towards that side we have two cases,

1. The elements towards that side are increasing. If all the elements are increasing then the boundary element will be larger than the element next to it hence will be a maxima.
2. If all the elements are not increasing then there exists some smallest index k for which the element $k-1$ is larger than it (we consider the case where we iterate right without loss of generality)and because k is the smallest by choice then it means that all elements from m to $k-2$ will also be smaller than $k-1$ (as its increasing towards the right from m). But this means that the element at$k-1$ index will be bigger than either of its neighbors and hence will be a local maxima.

(c) Analyze the runtime.

We have our recurrence as follows as we divide the problem by 2 at each function call and it takes constant time to compute whether values are greater than or equal to each other,

$$T(n) = T(n/2) + O(1)$$

Using master theorem we can see that our runtime is $O(\log(n))$

Q5 Let's move up a dimension! Let's say we have a matrix $M \in \mathbb{Z}^{n \times n}$, containing distinct integers. A local maximum of the matrix is an entry that is greater than or equal to all four of its neighbors (above, below, left, and right). Boundary elements are only compared to their existing neighbors.

You will design a *divide-and-conquer* algorithm that performs the following task in $\mathcal{O}(n \log(n))$ time:

Input: some matrix $M \in \mathbb{Z}^{n \times n}$ containing distinct integers
Output: a local maximum of M

(a) Describe your algorithm.

Note: Feel free to merge (a) and (b) if it's easier to explain that way.

Hint: Use your algorithm from (4) as a subroutine.

Let $f(M)$ be our recursive function. First let our base case be when $n = 1$ where we return the value at that index. Now let us choose the middle column and find the local maxima along that column using the subroutine defined in problem (4). Now we check if this is a local maxima, if not we move in the direction of the largest element and repeat the process for that half of the matrix until we get a local maxima.

(b) Show the correctness of your algorithm.

The algorithm will work because similar to the previous question we know that if we move along the side of the matrix that has a larger element to the current element we'll either come across increasing or decreasing values, if it is only decreasing then the boundary would be a local maxima but if it is decreasing then that would mean that we found a local maxima (note that because we're finding the local maxima of the column first that means that the element up and down will be smaller) .

(c) Provide a recurrence for your algorithm and use it to analyze the runtime.

Our recurrence will be,

$$T(n) = T(n/2) + O(\log(n))$$

We see that in each function call we have $O(\log(n))$ work done and we also divide our problem by half. So essentially we would end up with,

$$O(\log(n)^2)$$

total runtime.