CSE 102 Spring 2021
Homework Assignment 3

Jaden Liu
University of California at Santa Cruz
Santa Cruz, CA 95064 USA

April 20, 2021

1 HW3

1. Consider the problem of multiplying two large n-bit integers in a machine where the word size is one bit. The first three sub-problems will be discussed in the lecture.

a) **(0 point) Describe the straightforward algorithm that takes n^2 bit-multiplications.**

b) **(0 point) Find a way to compute the product of the two numbers using three multiplications of $n/2$ bit numbers (you will also have to do some shifts, additions, and subtractions, and ignore the possibility of carries increasing the length of intermediate results). Describe your answer as a divide and conquer algorithm.**

c) **(0 point) Assume that adding/subtracting numbers takes time proportional to the number of bits involved, and shifting takes constant time. Derive a recurrence relation for the running time of your divide and conquer algorithm. Use the master theorem to get an asymptotic solution to the recurrence.**

d) **(2 points) Now assume that we can find the product of two n-bitnumbers using some number of multiplications of $n/3$-bit numbers (plus some additions, subtractions, and shifts). What is the largest number of $n/3$ bit number multiplications that leads to an asymptotically faster algorithms than the $n/2$ divide and conquer algorithm above?**

Solution for a. Multiply each bit of two numbers and add them together, which will take n bits $\times$ n bits = n^2 bit-multiplications. $\square$

Solution for b.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1)2^{n/2} + x_0y_0 \\ &= x_1y_1 \cdot 2^n + ((x_1 + x_0)(y_0 + y_1) - x_1y_1 - x_0y_0) \cdot 2^{n/2} + x_0y_0 \end{aligned}$$

$\square$

Solution for c. Now we only need to compute three multiplication x_1y_1, $(x_1+x_0)(y_0+y_1)$, and x_0y_0. Therefore, the time complexity is $T(n)=3T(n/2)+cn$, whose $\log_b a \approx 1.58$, $n^{\log_b a} \approx n^{1.58}$, $f(n)=cn=O(n^{1.58})$. Using Master Theorem we get $T(n)=O(n^{\log_2 3})$ □

Solution for d.

$$\begin{aligned}xy &= (x_2 2^{2n/3}+x_1 2^{n/3}+x_0)(y_2 2^{2n/3}+y_1 2^{n/3}+y_0)\\ &= x_2y_2 2^{4n/3}+(x_2y_1+x_1y_2)2^n+(x_2y_0+x_0y_2+x_1y_1)2^{2n/3}+(x_1y_0+x_0y_1)2^{n/3}+x_0y_0\\ &= x_2y_2 2^{4n/3}+((x_2+x_1)(y_1+y_2)-x_1y_1-x_2y_2)2^n+\\ &((x_2+x_0)(y_2+y_0)-x_2y_2-x_0y_0+x_1y_1)2^{2n/3}+((x_1+x_0)(y_1+y_0)-x_1y_1-x_0y_0)2^{n/3}+x_0y_0\end{aligned}$$

The recursion will have 5 branch and n/3 depth. Thus, $T(n)=5T(n/3)+cn$, $\log_b a=\log_3 5$,$n^{\log_b a} \approx n^{1.46497}$, $f(n)=cn=O(n^{1.46497})$. Using Master Theorem, we get $T(n)=O(n^{\log_3 5}) \approx O(n^{1.46497})$. □

2. (4 points) You are given two sorted arrays of same size n. Design a divide-and-conquer $O(\log n)$ algorithm to compute the median of the these two sorted array. Provide pseudocode that you understand [do not copy and paste solutions found somewhere on the internet] including the base case of two arrays each of size 1 or size 2. Clearly state the size of subproblems, number of subproblems, and the order of the work required to combine the subproblems. Thus, establish the asymptotic computational complexity of the algorithm.

Solution. First we can consider finding median as dividing the "merge" array into two halfs. Then we will automatically have the median. To divide the merge array, we use binary search to find the median.

Base case 1: size of 1, arrayA=a, arrayB=b
The median is $\frac{a+b}{2}$.

Base case 2: size of 2, arrayA=a_1,a_2, arrayB=b_1,b_2
Assume the a_1 is smaller than b_1, otherwise, exchange the two arry.
Hence, there are only three possible cases: a_1,a_2,b_1,b_2, a_1,b_1,a_2,b_2, a_1,b_1,b_2,a_2. Thus the median is $\frac{max(a_1,b_1)+min(a_2,b_2)}{2}$.

```
Case 1: n is odd =2m+1:
    median of arrayA and B are $m_1$, $m_2$
    Case 1.1 if $m_1 < m_2$:
        recursively find from $m_1 to A[-1]$ and $B[0] to m_2$
    Case 1.2 if $m_1 > m_2$:
        recursively find from $A[0] to m_1$ and $m_2 to B[-1]$
    Case 1.3 if $m_1 = m_2$:
        we already find, which is m1 or m2.
    Case 1.4 if two sub array size falls to base case:
        do what we list in the base.
```

Case 2: n is even =2m:
median of arrayA and B are $m_1 = \frac{A[m]+A[m+1]}{2}$, $m_2 = \frac{B[m]+B[m+1]}{2}$
Case 2.1 if $m_1 < m_2$:
recursively find from $m_1 to A[-1]$ and $B[0] to m_2$
Case 2.2 if $m_1 > m_2$:
recursively find from $A[0] to m_1$ and $m_2 to B[-1]$
Case 2.3 if $m_1 = m_2$:
we already find, which is m1 or m2.
Case 2.4 if two sub array size falls to base case:
do what we list in the base.

The reason why the solution to the subproblem is the same to original problem is that we used same approach and do the operation on the same array, so the same index, same size. Therefore, the size of subproblem is $2n/2$, the number of subproblems is 2. Thus, $T(n) = 2T(n/2) + c$, using Master Theorem, $T(n) = O(\log n)$

□

3. (3 points) Assume you have an array A[1..n] of n elements. A majority element of A is any element occurring in more than n/2 positions(so if n = 6 or n = 7, any majority element will occur in at least 4 positions). Assume that elements cannot be ordered or sorted, but can be compared for equality. (You might think of the elements as chips, and there is a testor that can be used to determine whether or not two chips are identical.) Design a O(n lg n) divide and conquer algorithm to find a majority element in A (or determine that no majority element exists). Establish the computational complexity.

Solution. First we divide the array to two part and keep find the majority in the two halfs until the base case: start_index == end_index.

Base case1: 1 elements, has majority
Base case2: 2 elements, no majority
In the conquer part:
Case1: left,right no majority:
return no majority
Case2: left or right have majority:
determine if the left/right that has majority count is larger than size(left+right)/2
Case3: both majority:
determine which majority is larger and whether it's the new majority.

During each function, it need $O(n)$ time to find the majority in the present half. Therefore $T(n) = 2T(n/2) + cn$, using Master Theorem, we get $T(n) = O(n \log n)$.

```

            # codes from stackoverflow

            # Returns v if v is a majority;
            # otherwise, returns None
            def f(arr, low, high):
```

```
if low == high:
return arr[low]

if low + 1 == high:
return arr[low] if arr[low] == arr[high] else None

n = high - low + 1
mid = (low + high) / 2

l = f(arr, low, mid)
r = f(arr, mid + 1, high)

print 'n: ' + str(n) + '; l: ' + str(l) + '; r: ' + str(r) + ';
 L: ' + str((low, mid)) + '; R: ' + str((mid + 1, high))

if l == r:
return l

counts = [0, 0]

for i in xrange(low, high + 1):
if arr[i] == l:
counts[0] = counts[0] + 1
if arr[i] == r:
counts[1] = counts[1] + 1

if l and counts[0] * 2 > n:
return l

if r and counts[1] * 2 > n:
return r

return None
```

□

4. (3 points) A round-robin tournament is a collection of games in which each team plays each other exactly once. A schedule can be represented as an array of triplets, where the triplet (d, i, j) means that team i will play team j on day d. A schedule is reasonable if no team plays more than one game per day, and optimal if it uses the smallest number of days out of all reasonable schedules.

a) Design an optimal schedule for the tournament using divide and conquer assuming that n is a power of 2, where n is the number of teams. Provide a brief argument that the algorithm is optimal

b) Establish the computational complexity of the algorithm.

Solution for a. According to the divide-and-conquer strategy, all the players can be divided into two halves, and the game schedule of n players can be determined by the game schedule of n/2 players. Recursively divide the players into two with the strategy of dividing into two. When there are only two players left, the schedule of the game becomes very simple. At this time, just let these two players compete.
I will use a table to format these, (d,i,j) in problems will represent team i(row i) will compete with the team j in in day d (column d). Therefore, the value in table[i][d] is team j who should compete with.

1	2
2	1

This the base case

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

Notice that it's generate by recurrence, left top 2x2 is same as right bottom 2x2, so as the other two 2x2 matrix.

□

Solution for b. The problem split to 2 subproblems each times and split half. Thus $a, b = 2$ $T(n) = 2T(n/2) + n^2$, using the Master Theorem, we get $T(n) = \theta(n^2)$ □

5. Quicksort Algorithm: Read the description of Quicksort Algorithm

a) (1 point) Problem 7.1.1, CLRS Page 173: Illustrate the operation of PARTITION on the array A = 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11.

b) (1 point) Problem 7.1.3, CLRS Page 174: Give a brief argument that the running time of PARTITION algorithm on a subarray of size n is $\theta(n)$.

c) (1 point) Worst-case Partitioning: Establish that the worst-case running time algorithm for quicksort is $\theta(n^2)$.

d) (1 point) Best-case Partitioning: establish that the best-case running time algorithm for quicksort is $\theta(n \log n)$

e) (1 point) Proportional Partitioning: Assume that the split at each level of recursion in quicksort is a constant ratio of $r:1$, where r is a constant. Establish that the running time of quicksort is still $\theta(n \log n)$.

Solution for a. First let the A[0]=13 to be the base point, then divide them into two sub-part; one is larger than 13; the other one is smaller than 13. Then keep divide each part to 2 part again until it cannot be divided anymore
11,6,2,5,4,5,8,12, 13, 19,21 (First divide)
6,2,5,4,5,8, 11, 12, 13, 19, 21 (Second divide, right part reaches the end)

2,5,4,5, 6, 8, 11, 12, 13, 19, 23(Third divide)
2, 5,4,5, 6, 8, 11, 12, 13, 19, 23(Fourth)
2, 4, 5, 5, 6, 8, 11, 12, 13, 19, 23 (Finish) □

Solution for b. Basically, during partition part, what we need to do is just compare each element in this array with a given value in the array. Therefore, the compare part need n-1 times. Each move is follow by comparing, at most n-1 times. $T(n) = n-1+O(n-1) = \theta(n)$ □

Solution for c. The worst case happens that the choosen value is the smallest/ largest element each time, so it cannot divide the array to subarray, which will be same as selection sort. $T(n) = T(n-1)+cn = O(n^2)$. □

Solution for d. The best case happens when the choosen value is the median each time, so it will divide the array to two equal sub array each time. $T(n) = 2T(n/2)+cn$, using the Master Theorem, we get $T(n) = O(n\log n)$ □

Solution for e.

$$\begin{aligned}
T(n) &= T(\frac{r}{r+1}n) + T(\frac{1}{r+1}n) + cn \\
&= T((\frac{r}{r+1})^2 n) + T(\frac{r}{r+1}\frac{1}{r+1}n) + \frac{r}{r+1}cn + T((\frac{1}{r+1})^2 n) + T(\frac{1}{r+1}\frac{r}{r+1}n) + \frac{1}{r+1}cn + cn \\
&= T((\frac{r}{r+1})^2 n) + 2T(\frac{r}{r+1}\frac{1}{r+1}n) + T((\frac{1}{r+1})^2 n) + 2cn \\
&\vdots \\
&= \sum_{i=1}^{\log_{r+1} n} T((\frac{r}{r+1})^i \cdot (\frac{1}{r+1})^{\log_{r+1} n - i} n) + \log_{r+1} n \cdot cn \\
&= O(\log n) + \log_{r+1} n \cdot cn
\end{aligned}$$

Thus we can see the time complexity is $O(n\log n)$. □

References

[1] https://stackoverflow.com/questions/48583034/most-element-in-array-divide-and-conquer-on-logn