

CSE3358 Problem Set 3 Solution

Problem 1: Multiplying large numbers

We often assume that multiplying two integers is a $\Theta(1)$ -time operation. This assumption becomes unrealistic if the numbers grow very large. For instance, to multiply two n -bit integers, u and v , the traditional algorithm (do it by hand to see) requires $\Theta(n^2)$ bit multiplications and additions. In this problem we will investigate a divide-and-conquer algorithm for multiplying two numbers. This will be similar to Strassen's algorithm, but applied to n -bit numbers instead of matrices of size $n \times n$.

We will divide each number into two parts, the high order bits and the low order bits. Thus u can be written as $a2^{n/2} + b$ where a represents the high order bits of u and b represents the low order bits of u . Similarly, v can be written as $c2^{n/2} + d$. Therefore,

$$uv = (a2^{n/2} + b)(c2^{n/2} + d) = ac2^n + (ad + bc)2^{n/2} + bd$$

Multiplying by a power of 2 is a simple shifting operation that can be done in linear time $\Theta(n)$. So the dominating operations are the 4 multiplications ac , ad , bc , and bd .

- (a) (10 points) Write pseudocode for a divide-and-conquer algorithm that recursively computes the products ac , ad , bc , and bd , and combines them to compute the product uv . Write a recurrence describing the algorithm running time and solve it.

ANSWER:

NOTE: $\ll x$ shifts to the left x times by filling zeros i.e. multiplies by 2^x .

```
MULTIPLY( $u, v$ )
   $p \leftarrow$   $2n$  bit number initialized to zero
   $\triangleright$  DIVIDE
   $n \leftarrow \text{length}[u]$ 
   $m \leftarrow \lfloor n/2 \rfloor$ 
  if  $n=1$ 
    then do  $p \leftarrow u.v$ 
    return  $p$ 
   $a \leftarrow u[m + 1..n]$ 
   $b \leftarrow u[1..m]$ 
   $c \leftarrow v[m + 1..n]$ 
   $d \leftarrow v[1..m]$ 
   $\triangleright$  CONQUER
   $p_1 \leftarrow$  MULTIPLY( $a, c$ )
   $p_2 \leftarrow$  MULTIPLY( $a, d$ )
   $p_3 \leftarrow$  MULTIPLY( $b, c$ )
   $p_4 \leftarrow$  MULTIPLY( $b, d$ )
   $\triangleright$  COMBINE
   $p \leftarrow p_1 \ll 2m + (p_3 + p_4) \ll m + p_2$ 
  return  $p$ 
```

Each problem recursively solves four subproblems of half the size. The divide step can be done in linear time by scanning the bits of u and v . The combine step can also be done in linear time because it involves additions of $m = n/2$ bit numbers. Note that the multiplications by 2^n and $2^{n/2}$ are shifting operations which can be done in linear time as well. Therefore, the divide and combine steps take $\Theta(n)$ time. The recurrence will be $T(n) = 4T(n/2) + \Theta(n)$ which has a solution $T(n) = \Theta(n^2)$ using the master method. Therefore, there is no benefit over the usual $\Theta(n^2)$ multiplication process.

- (b) (10 points) Design a different divide-and-conquer algorithm: compute ac and bd , then compute $ad + bc$ using only one multiplication along with addition and subtraction, thus reducing the number of sub-problems in each level of the recursion to 3 instead of 4. Write the recurrence describing the algorithm running time and solve it.

ANSWER: The key to this part is that $(a + b).(c + d) = ac + bd + ad + bc$. Therefore, we compute ac and bd with two multiplications. We can then add $a + b$ and $c + d$ and perform one more multiplication to obtain $(a + b).(c + d)$. Finally subtraction of $ac + bd$ from the last multiplication gives $ad + bc$.

MULTIPLY(u, v)

```

     $p \leftarrow 2n$  bit number
    ▷ divide
     $n \leftarrow \text{length}[u]$ 
     $m \leftarrow \lfloor n/2 \rfloor$ 
    if  $n=1$  then
        then do  $p \leftarrow u.v$ 
        return  $p$ 
     $a \leftarrow u[m + 1..n]$ 
     $b \leftarrow u[1..m]$ 
     $c \leftarrow v[m + 1..n]$ 
     $d \leftarrow v[1..m]$ 
     $e \leftarrow a + b$ 
     $f \leftarrow c + d$ 
    ▷ conquer
     $p_1 \leftarrow \text{MULTIPLY}(a,c)$ 
     $p_2 \leftarrow \text{MULTIPLY}(b,d)$ 
     $p_3 \leftarrow \text{MULTIPLY}(e,f)$ 
    ▷ combine
     $p \leftarrow p_1 \ll 2m + (p_3 - p_1 - p_2) \ll m + p_2$ 
    return  $p$ 

```

Note that we have two extra additions and two extra subtractions. But the time for the divide and combine steps is still $\Theta(n)$. The recurrence for this algorithm is now $T(n) = 3T(n/2) + \Theta(n)$ which has a solution $T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.58})$ using the master method. This is better than the $\Theta(n^2)$ time algorithm.

Problem 2: Solving Recurrences

Use the Master Theorem to solve the following recurrences. State which case of the Master Theorem you are using (and justify) for each of the recurrences.

NOTE: In all of the following problems, the regularity condition for case 3 of the master theorem is satisfied because $f(n) = n^k$.

a. $T(n) = 2T(\frac{n}{2}) + n^3$

ANSWER: $\frac{f(n)}{n^{\log_2 2}} = \frac{n^3}{n} = n^2 = \Omega(n^2)$. This is case 3; therefore, $T(n) = \Theta(n^3)$.

b. $T(n) = T(\frac{9n}{10}) + n$

ANSWER: $\frac{f(n)}{n^{\log_{10/9} 1}} = \frac{n}{n^0} = n = \Omega(n^1)$. This is case 3; therefore, $T(n) = \Theta(n)$.

c. $T(n) = 16T(\frac{n}{4}) + n^2$

ANSWER: $\frac{f(n)}{n^{\log_4 16}} = \frac{n^2}{n^2} = 1 = \Theta(\log^0 n)$. This is case 2; therefore, $T(n) = \Theta(n^2 \log n)$.

d. $T(n) = 7T(\frac{n}{3}) + n^2$

ANSWER: $\frac{f(n)}{n^{\log_3 7}} = \frac{n^2}{n^{\log_3 7}} = n^{2 - \log_3 7}$. Since $2 - \log_3 7 > 0$, this is case 3 again, and $T(n) = \Theta(n^2)$.

e. $T(n) = 7T(\frac{n}{2}) + n^2$

ANSWER: $\frac{f(n)}{n^{\log_2 7}} = \frac{n^2}{n^{\log_2 7}} = n^{2 - \log_2 7}$. Since $2 - \log_2 7 < 0$, this is case 1, and $T(n) = \Theta(n^{\log_2 7})$.

f. $T(n) = 2T(\frac{n}{4}) + \sqrt{n}$

ANSWER: $\frac{f(n)}{n^{\log_2 4}} = \frac{n^{1/2}}{n^{1/2}} = 1 = \Theta(\log^0 n)$. This is case 2, therefore; $T(n) = \Theta(\sqrt{n} \log n)$.

Problem 3: Partitioning in Quicksort

Consider the following code for partitioning an array in Quicksort.

```

PARTITION(A,p,r)
x←A[p]    ▷ this is the pivot
i←p-1    ▷ i starts at the far left
j←r+1    ▷ j starts at the far right

while TRUE
  do repeat j←j-1    ▷ move j down until it finds an element ≤ x
    until A[j]≤ x
  repeat i←i+1    ▷ move i up until it finds an element ≥ x
    until A[i]≥ x
  if i<j          ▷ if they did not cross
    then swap A[i]↔A[j]
  else return j

```

The Quicksort algorithm used with that partitioning code is the following:

```

QUICKSORT(A,p,r)
if p<r
  then q←PARTITION(A,p,r)
      QUICKSORT(A,p,q)
      QUICKSORT(A,q+1,r)

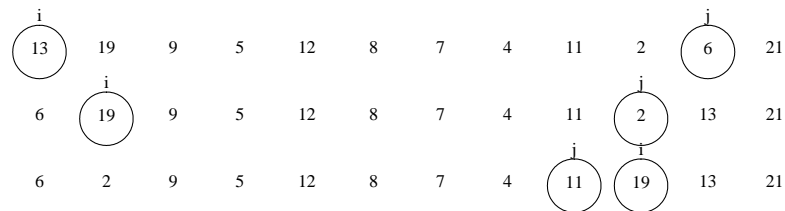
```

When PARTITION terminates, every element in $A[1..j]$ is less than or equal to every element in $A[j+1..r]$, for $p \leq j < r$. Moreover, it always places the pivot value (originally in $A[p]$) into one of these two partitions. Since $p \leq j < r$, this split is always nontrivial i.e. both partitions are non-empty.

(a) (3 points) Show the result of above PARTITION on the array $A = [13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21]$ step by step.

ANSWER:

$A[p..r] = [13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21]$
 Pivot = $A[p] = 13$



PARTITION will produce:

$A[p..q] = [6, 2, 9, 5, 12, 8, 7, 4, 11]$

$A[q+1..r] = [19, 13, 21]$

There are technicalities that make the pseudocode of PARTITION a little tricky. For example, the choice of $A[p]$ as pivot is not arbitrary.

(a) (5 points) Show that if $A[r]$ is used as a pivot element in the PARTITION code, the QUICKSORT algorithm above might never terminate.

ANSWER: If $A[r]$ is taken as pivot and it happens that it is the uniquely largest element in the array, then i will stop at r because $A[r]$ is the first element greater or equal to the pivot that i will encounter. Similarly, j will stop at r because $A[r]$ is the first element less or equal to the pivot that j will encounter. Therefore, i and j will stop at r and from the condition at the end of the PARTITION code (i and j are crossing), the returned value will be $j = r$ which goes into q in the QUICKSORT code. As a result, $\text{QUICKSORT}(A, p, r)$ will recursively call $\text{QUICKSORT}(A, p, q)$ i.e. $\text{QUICKSORT}(A, p, r)$. Since nothing has changed in the array $A[p..r]$ during the partitioning process, this will result in an infinite loop where the same thing happens over and over.

(b) (5 points) The PARTITION code here is faster in practice than the one we saw in class. Construct an array of size n for any n such that the PARTITION code here performs zero swaps while the one we saw in class performs n swaps.

ANSWER: Consider a sorted array $A[p..r] = [p, p + 1, \dots, r]$. For the PARTITION code above, the pivot is $x = A[p] = p$. Therefore, i will stop at $A[p]$ (this is the first element $\geq x$ it will encounter) and j will stop at $A[p]$ (this is the first element $\leq x$ it will encounter). Since $i = j$, i.e. it is not the case that $i < j$, the code will return $j = p$ without any swaps.

For the partition code we saw in class, however, the pivot is $x = A[r] = r$. j will scan the array from p to $r - 1$. At every position of j from p to $r - 1$, the element $A[j]$ will always be less than the pivot $A[r]$, so it will be swapped with $A[i]$ after incrementing i by 1. The first time this happens, we swap $A[p]$ with $A[p]$. The second time, we swap $A[p + 1]$ with $A[p + 1]$, etc... We keep on performing redundant swaps until we swap $A[r - 1]$ with $A[r - 1]$ where i and j are both equal to $r - 1$. Finally, we swap $A[r]$, the pivot, with $A[i + 1] = A[r]$. Therefore, the total number of swaps is equal to the number of element, which is equal to $r - p + 1 = n$.

(c) (5 points) The partitioning code we saw in class does not yield a balanced partitioning when the elements are not distinct. Construct an array of size n for any n such that the PARTITION code here splits the array in two halves while the the one we saw in class produces a worst-case partition.

ANSWER: Consider the array $A[p..r] = [x, x, x, \dots, x]$, i.e. all the elements are equal to x . The pivot will definitely be x in all cases. In the partition code above, i and j will stop at every position on their way towards the middle of the array, and will be swapping equal elements. The partitioning will be optimal because i and j will partition the array A at the middle (this is where i and j will cross). The partition code we saw in class will move j all the way from p up to $r - 1$ swapping every element with itself on the way (because every element is \leq to the pivot). Finally, it will swap the pivot $A[r]$ with itself and return r , yielding to a partition of size $n - 1$, which is the worst case.

NOTE: The example of making $A[r]$ the smallest or largest element works but is not a fair example because the same kind of example can cause the PARTITION code above to produce a bad partition, simply making $A[p]$ the smallest or largest.