CSE 102: Spring 2021: HW 3 Solutions

1. **n-bit Multiplications**:

   Consider the problem of multiplying two large n-bit integers in a machine where the word size is one bit.

   (a), (b), and (c) not required for the assignment and was discussed in the class. These solutions are repeated here. Only (d) is required.

   (a) Consider the problem of multiplying two large n-bit integers in a machine where the word size if one bit. Describe the straightforward algorithm that takes n^2 bit-multiplications.

   This algorithm is the algorithm that we all learned in grade school to multiply two numbers.

```
procedure:  multiply(A,B)
  // assume A,B each have n bits
  // the lower order bit of A will be written A[0],
  // the highest will be A[n], same for B
  let result = 0
  let temp = 0
  for j = 1 to n do
    for i = 1 to n do
      temp[i] = B[i] * A[j]
    end for
    shift temp by (j - 1) bits to the left
    set result to result plus temp
    set temp = 0
  end for
  return result
```

The two `for` loops that surround the bit multiplication makes this algorithm do n^2 bit-multiplications.

(b) 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.

The algorithm that follows is motivated by the following example where two 2-digit (base 10) numbers are multiplied:

```
          a  b
        * c  d
        ------
         ad bd
      ca  cb
     ----------
  ca (ad+bc) bd  (assuming no carries)
```

In other words:

$$(a\times 10^1+b\times 10^0)\times(c\times 10^1+d\times 10^0) = ca\times 10^2+(ad+bc)\times 10^1+bd\times 10^0$$

Converting now to bits, if a, b, c, d are all $n/2$-bit numbers then multiplying two n-but numbers (ab) and (cd) takes 4 multiplications of size $n/2$, 3 shifts and 3 addition/subtraction operations. We can reduce the number of multiplications by noticing that:

$$ad + bc \;=\; (a+b)(c+d) - ac - bd$$

and therefore we have all the numbers we need with only three multiplications, and we can write down the divide and conquer algorithm (for binary numbers):

```
procedure:  DC_multiply(X,Y)
  // assume X,Y each have n bits, where n is a power of 2
  if X and Y are both 1 bit long, return X*Y;
  else
    let k = n/2;
    let a = X div 2^k and b= X mod 2^k (so a*2^k + b = X)
```

```
let c = Y div 2^k and d= Y mod 2^k (so c*2^k + d = Y)
// a,b,c,d are all n/2-bit numbers
temp_1 = DC_multiply(c,a)
temp_2 = DC_multiply(a+b,c+d)
temp_3 = DC_multiply(b,d)
return ( temp_1*2^(2k) + \\
(temp_2 - temp_1 - temp_3)*2^k + temp_3 )
```

In this algorithm we have 3 recursive calls on $n/2$-bit numbers, 2 shifts and 6 additions/subtractions on $\Theta(n)$-bit numbers (not counting the shifts/subtractions needed for the div and mod operations).

(c) Assuming that adding/subtracting numbers takes time proportional to the number of bits involved, and shifting takes constant time.

Using the above algorithm we can immediately write down the following recurrence for its running time. $T(1) = 1$ and

$$T(n) = 3T(\lceil n/2 \rceil) + G(n) \quad \text{for } n > 1$$

where $G(n) \in \Theta(n)$ is the time needed for the additions, shifts, and overhead. Applying the master theorem we get **Case 1**: $E = \lg 3 / \lg 2 = \lg 3$, so $T(n) \in \Theta(n^{\lg 3})$ (recall that $\lg 3 \approx 1.585$).

(d) Now assume that we can find the product of two n-bit numbers 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?

The running time of a 5-multiplication method for two n-figure numbers has the recurrence relation

$$T(n) = 5T(n/3) + G(n)$$

where $5T(n/3)$ is the time to do the 5 recursive multiplications, and $G(n) \in \Theta(n)$ is the time needed for the additions, shifts, and overhead. Applying case 1 of the Master Theorem, we get

$T(n) \in \Theta(n^{\log_3 5})$, and $\log_3 5 \approx 1.465$, so this is better than the 2-multiplication method.

A 6-multiplication method would have the recurrence relation

$$T(n) = 6T(n/3) + G(n),$$

where $6T(n/3)$ is the time to do the six recursive multiplications and $G(n) \in \Theta(n)$ is the time needed for the additions, shifts, and overhead. Using the the Master Theorem on this recurrence yields $T(n) \in \Theta(n^{\log_3 6})$, and $\log_3 6 = 1.631$, which is asymptotically worse than the algorithm we developed for our 2-multiplication method.

Therefore 5 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.

2. **Median of two sorted arrays of same size**

   https://www.geeksforgeeks.org/median-of-two-sorted-arrays/

   Outline of proof of base case for arrays of length 2 was presented in the class. Left as a simple exercise by examining three separate cases.

   Outline of computation of exact size of subarrays was presented in class. Also left as a simple exercise for the two cases: odd and even length arrays.

3. **Majority Problem**

   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 tester 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).

   A $\Theta(n \log n)$ running time divide and conquer algorithm:

The algorithm begins by splitting the array in half repeatedly and calling itself on each half. This is similar to what is done in the merge sort algorithm. When we get down to single elements, that single element is returned as the majority of its (1-element) array. At every other level, it will get return values from its two recursive calls.

The key to this algorithm is the fact that if there is a majority element in the combined array, then that element must be the majority element in either the left half of the array, or in the right half of the array. There are 4 scenarios.

(a) Both return "no majority." Then neither half of the array has a majority element, and the combined array cannot have a majority element. Therefore, the call returns "no majority."

(b) The right side is a majority, and the left isn't. The only possible majority for this level is with the value that formed a majority on the right half, therefore, just compare every element in the combined array and count the number of elements that are equal to this value. If it is a majority element then return that element, else return "no majority."

(c) Same as above, but with the left returning a majority, and the right returning "no majority."

(d) Both sub-calls return a majority element. Count the number of elements equal to both of the candidates for majority element. If either is a majority element in the combined array, then return it. Otherwise, return "no majority."

The top level simply returns either a majority element or that no majority element exists in the same way.

To analyze the running time, we can first see that at each level, two calls are made recursively with each call having half the size of the original array. For the non-recursive costs, we can see that at each level, we have to compare each number at most twice (which only happens in the last case described above). Therefore, the non-recursive cost is at most $2n$ comparisons when the procedure is called with an array of size n. This lets us upper bound the number of comparisons done by $T(n)$

defined by the recurrence $T(1) = 0$ and

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

We can then determine that $T(n) \in \Theta(n \log n)$ as desired using Case 2 of the Master Theorem.

4. **Round-Robin**

`https://exams.ubccsss.org/`
`\\`
`cs320/2010/cs320-2010-t1-sample-midterm2-solution.pdf`

Make sure you can unpack (understand) the mathematical expressions here, $(j - i) \bmod \frac{n}{2}$.

5. **Quicksort Algorithm**

Solutions not provided for (a) and (b): easy. Solutions for (c), (d), and (e) are in CLRS.