

**CS 102: SOLUTIONS TO DIVIDE AND CONQUER  
ALGORITHMS (ASSGN 4)**

- Problem 1.** a. Consider the modified binary search algorithm so that it splits the input not into two sets of almost-equal sizes, but into three sets of sizes approximately one-third. Write down the recurrence for this ternary search algorithm and find the asymptotic complexity of this algorithm.
- b. Consider another variation of the binary search algorithm so that it splits the input not only into two sets of almost equal sizes, but into two sets of sizes approximately one-third and two-thirds. Write down the recurrence for this search algorithm and find the asymptotic complexity of this algorithm.

*Solution.* a. The recurrence for normal binary search is  $T_2(n) = T_2(n/2) + 1$ . This accounts for one comparison (on an element which partitions the  $n$ -element list of sorted keys into two  $\frac{n}{2}$ -element sets) and then a recursive call on the appropriate partition. For ternary search we make two comparisons on elements which partition the list into three sections with roughly  $n/3$  elements and recurse on the appropriate partition. Thus analogously, the recurrence for the number of comparisons made by this ternary search is:

$$T_3(n) = T_3(n/3) + 2.$$

However, just as for binary search the second case of the Master Theorem applies. We therefore conclude that  $T_3(n) \in \Theta(\log(n))$ .

- b. We now consider a slightly modified take on ternary search in which only one comparison is made which creates two partitions, one of roughly  $n/3$  elements and the other of  $2n/3$ . Here the worst case arises when the recursive call is on the larger  $2n/3$ -element partition. Thus the recurrence corresponding to this worst case number of comparisons is

$$T_{3w}(n) = T_{3w}(2n/3) + 1$$

but again the second case of the Master Theorem applies placing  $T_{3w} \in \Theta(\log n)$ . The best case is exactly the same (and thus so must also be the average).

It is interesting to note that we will get the same results for general  $k$ -ary search (as long as  $k$  is a fixed constant which does not depend on  $n$ ) as  $n$  approaches infinity. The recurrence becomes one of

$$\begin{aligned} T_k(n) &= T_k(n/k) + k - 1 \\ T_{kw}(n) &= T_{kw}\left(\frac{(k-1)n}{k}\right) + 1 \end{aligned}$$

depending on whether we are talking about part (a) or part (b). For each recurrence case two of the Master Theorem applies. ■

**Problem 2.** Consider the following variation on *Mergesort* for large values of  $n$ . Instead of recursing until  $n$  is sufficiently small, recur at most a constant  $r$  times,

and then use insertion sort to solve the  $2^r$  resulting subproblems. What is the (asymptotic) running time of this variation as a function of  $n$ ?

*Solution.* Each call to Mergesort on  $n$  elements makes 2 recursive calls of size  $n/2$ , if we recurse at most  $r$  times and then revert to the basic (ad-hoc) method, the modified Mergesort creates  $2^r$  subproblems of size  $\frac{n}{2^r}$ . If the ad-hoc algorithm runs in time  $\Theta(n^2)$  (like selection or insertion sort), this modified mergesort will take at least  $2^r(\frac{n}{2^r})^2 = n^2/2^r$  time for large instances ( $n > 2^r n_0$ ).

Since  $r$  is constant, so is  $2^r$ , and the running time is then  $\Omega(n^2)$ , which is asymptotically worse than the standard mergesort. In essence, this modification requires that we solve  $\Theta(n)$  size problems with the ad-hoc method. If the ad-hoc method is polynomial, then the modification to divide-and-conquer method cannot be more than a constant factor better than simply applying the ad-hoc method to the original problem. ■

**Problem 3.** Design and analyze a divide and conquer MAXMIN algorithm that uses  $\lceil \frac{3n}{2} \rceil - 2$  comparisons for any  $n$ .

*Solution.* The divide and conquer algorithm we develop for this problem is motivated by the following observation. Suppose we knew the maximum and minimum element in both of the roughly  $\frac{n}{2}$  sized partitions of an  $n$ -element ( $n \geq 2$ ) list. Then in order to find the maximum and minimum element of the entire list we simply need to see which of the two maximum elements is the larger, and which of the two minimums is the smaller. We assume that in a 1-element list the sole element is both the maximum and the minimum element. With this in mind we present the following psuedocode for the max/min problem.

```

procedure maxmin(A[1..n] of numbers) -> (min, max)
begin
  if (n == 1)
    return (A[1], A[1])
  else if (n == 2)
    if ( A[1] < A[2] )
      return (A[1], A[2])
    else
      return (A[2], A[1])
  else
    (max_left, min_left) = maxmin(A[1...(n/2)])
    (max_right, min_right) = maxmin(A[(n/2 +1)..n])
    if (max_left < max_right)
      max = max_right
    else
      max = max_left
    if (min_left < min_right)
      min = min_left
    else
      min = min_right
    return (min, max)
end

```

Let  $T(n)$  be the number of comparisons performed by the `maxmin` procedure. When  $n = 1$  clearly there are no comparisons. Thus we have  $T(1) = 0$ . Similarly,  $T(2) = 1$ . Otherwise when  $n > 2$  clearly

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

since `maxmin` performs two recursive calls on partitions of roughly half of the total size of the list and then makes two further comparisons to sort out the max/min for the entire list. (Of course, to be pedantic there should be floors and ceilings in the recursive function, and something should be said about the fact that the following proof is only for  $n$  which are powers of two and how this implies the general result. This is omitted.)

We next show that  $T(n) = \lceil \frac{3n}{2} \rceil - 2$  for all  $n$  which are powers of 2. The proof is by induction on  $n$ .

**Base Case:** ( $n=2$ ): from the recursive definition we have that  $T(2) = 1$ . Similarly, we have that  $\lceil \frac{3 \cdot 2}{2} \rceil - 2 = 3 - 2 = 1$ , thus verifying the base case.

**Inductive Step:** Let  $n > 2$  and  $n = 2^j$  for some integer  $j \geq 2$ . Assume that  $T(k) = \lceil \frac{3k}{2} \rceil - 2$  for all  $k = 2^j$  for some integer  $j \geq 2$  and  $k < n$ . We want to show that this assumption implies that  $T(n) = \lceil \frac{3n}{2} \rceil - 2$  for all positive  $n$  which are powers of 2.

We start with the given recurrence

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2\left(\left\lceil \frac{3(n/2)}{2} \right\rceil - 2\right) + 2 \end{aligned}$$

by the inductive hypothesis with  $k = n/2$ . This gives

$$\begin{aligned} T(n) &= 2\left\lceil \frac{3(n/2)}{2} \right\rceil - 4 + 2 \\ &= \left\lceil \frac{3n}{2} \right\rceil - 2 \end{aligned}$$

showing the desired result. By the principle of mathematical induction we are done. ■

**Problem 4.** 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 an efficient divide and conquer algorithm to find a majority element in  $A$  (or determine that no majority element exists).

Aim for an algorithm that does  $O(n \lg n)$  equality comparisons between the elements. A more difficult  $O(n)$  algorithm is possible, but may be difficult to find.

*Solution.* 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. ■

**Problem 5.** Consider the problem of multiplying two large  $n$ -bit integers in a machine where the word size is one bit.

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

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

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. ■