CSCD320 Homework2

Ethan Tuning

Problem 1. *Finding the maximum in a sequence of n numbers is a simple task: simply scan the sequence and pick/return the maximum. The time cost of this simple method is clearly θn, which is optimal because one has to take one look at every number in the sequence in order to find the maximum. Now Dr. Nonsense wants to use the divide-conquer strategy to overkill this task of finding the maximum among the sequence of n number.*

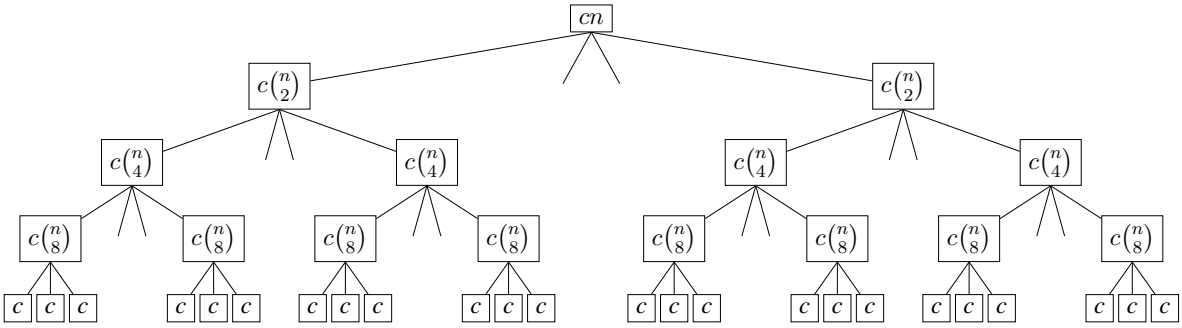
1. *Describe the algorithmic idea and give the pseudocode of Dr. Nonsense's algorithm.*
2. *Give the time cost in the θ-notation of Dr. Nonsense's algorithm by using the recursion tree method for solving the recurrence.*
3. *Is this Dr. Nonsense's d-c based algorithm asymptoticall slower or faster or identical, compared with the simple $\theta(n)$-time method?*
4. *How is the time efficiency in practice of Dr. Nonsense's algorithm, compared to the simple method? Why?*

Answer 1. *Answers are in order by question asked respectively.*

1. *So for Dr. Nonsense's algorithm we would just divide the sequence into two parts over and over, until we came up with groups of only 2 numbers each. Then we would pick the biggest of the 2 and return that value. We would just call this function recursively twice and return the larger of the two. Essentially merging the results and only grabbing the larger.*

```
int maximum(A[], left, right){
  if(right - left equals 1)
    return A[left];
  mid = (left + right) / 2;
  number1 = maximum(A, left, mid);
  number2 = maximum(A, mid, right);
  return number1 > number2;
}
```

2. *So if we look at the recursion tree down below we can see that the problem is being cut in half each time. We can see that there is am recurrence of $T(n) = 2T(n/2) + C * n$ if we work this out then we can see that the recurrence can be solved and the time cost of $\theta(n \log(n))$.*



3. *Dr. Nonsense should not be a Dr., because if we take his algorithm, the time cost is a lot more then the trivial solution to this easy problem.*

4. *In practice you probably would not ever notice a difference until the data size was very massive. So is this really all that bad? No, not really, but you always want to use the fastest of algorithms, and this solution is just not it.*

Problem 2. *Suppose you are given an array $Prod[1..n]$ of n positive integers, where each number $Prod[i], 1 \leq n$, represents the sequential id of the ith product that is manufactured in your company. All the sequential ids are distinct and the array $Prod[1..n]$ has been sorted in ascending order. You want to design a divide-and-conquer strategy based efficient algorithm to determine whether there exists a product $Prod[i]$ such that its sequential id is equal to i, i.e., $Prod[i] = i$. If there exist such a product $Prod[i] = i$, you return the value of i. In the case where you have multiple choices, any choice is fine Otherwise, you return -1.*

1. *Describe your algorithm idea and its pseudocode.*

2. *Give and explain the time cost of your algorithm using the big-O notation and make your bound as tight as possible.*

Answer 2. *Answers are in order by question asked respectively.*

1. *So with this algorithm, it is pretty straight forward. We will just search for the value. We can do this many ways. Let us just return -1 if the low is greater than the high. Else, we will just cut the array into 2 parts, and recursively call the method over incrementing one half and decrementing the other.*

```
int findi(Prod[], i, low, high){
  if(low > high)
    return -1;
  mid = (low + high) / 2;
  if(Prod[mid] equals i)
    return mid;
  else if(Prod[mid] < i)
    return findi(Prod, i, mid+1, high);
  else
    return findi(Prod, i, low, mid-1);
}
```

2. *The time of cost of this algorithm will obviously be $\log n$ because it is yet another divide and conquer, but we are not adding any extra work with actually evaluating anything, like we would add with merge sort, do to its merge method. This is just searching an array recursively. Also, there is a faster way to do this that does not involve divide and conquer. The other method can maintain a linear time cost.*

Problem 3. *Search and learn three existing algorithms that use divide-conquer strategy. For each algorithm, in your own language, concisely and clearly describe:*

1. *The problem statement.*

2. *The algorithmic idea in the solution.*

3. *The time complexity.*

4. *The condition, on which the worst-case running time appears.*

5. *The source of your finding.*

Answer 3. *Each answer is given with each algorithm numbered 1, 2, 3 and then within those numbers we have answers to the three above questions about each algorithm in the same order asked.*

1. *The Strassen Matrix Algorithm*

   - *This algorithm finds the product of two matrices.*
   - *The idea is to, instead of iterating through each matrix and multiplying, we can use many addition operations. You can take the two matrices and divide them into 4 sectors each and then add the corresponding sectors with one another. Strassen defined each sector to only do 7 products instead of 8, so you do an extra addition instead. Merge your results and BOOM. This will result in a little bit faster time then just doing the traditional way of multiplying two matrices. There is a noticeable difference when doing complex, large, matrices.*
   - *The time complexity of this algorithm is* $O(n^{\log 7+o(1)})$.
   - *Your worst case will be the same as above.*
   - *Links: https://martin-thoma.com/strassen-algorithm-in-python-java-cpp/*

2. *Maximum difference between two ordered elements.*

   - *If given an array of numbers, we want to find the difference between two elements, such that the bigger element comes after the smaller element i.e. the largest difference. What we can do is recursively divide the array until we only have sub-arrays of size 2. We then take the difference of the elements in each sub-array. Then merge our results back, comparing each result for the largest. Keep this larger value and continue, over and over. Then once we get to end we will have, indeed, the largest difference.*
   - *The time complexity will be linear. We only are doing one operation after all the recursive calls. There is not much work being don at all. Very easy algorithm.*
   - *Even the worst case will be linear.*
   - *Links: http://stackoverflow.com/questions/24055608/divide-and-conquer-algo-to-find-maximum-difference-between-two-ordered-elements*

3. *Karatsuba's Algorithm*

   - *We are given 2 integers, multiply them together. This seems silly! This is not he case though, what if you are to mulitply 2 HUGE numbers? Well this algorithm will basically reduce the large problem into 3 smaller problems and compute those, then add them together, so we get* $A * B = x * r^n + y * r^{n/2} + z$ *where r is the base and n is the number of digits.*
   - *The time complexity will be* $O(n^1.584...)$ *as apose to the normal* n^2 *solution. So saves you a little bit of time.*

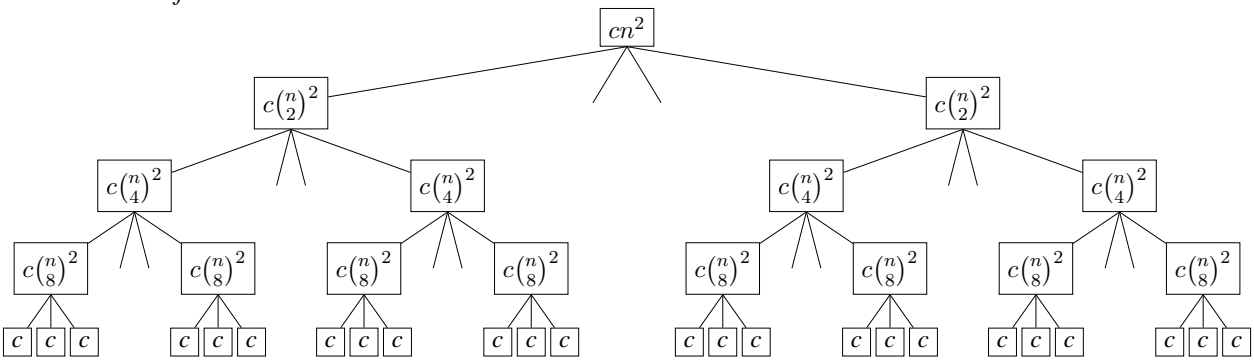
- *The worst case will also be the above.*
- *Links: https://courses.csail.mit.edu/6.006/spring11/exams/notes3-karatsuba*

Problem 4. *Prove:* $T(n) = 2T(n/3) + n^2 = O(n^2)$

Answer 4. *For this we just have to find some constant* n_0 *and* C*, such that* $T(n) \leq Cn^2$ *for all* $n \geq n_0$*. Let us have a base be* $T(3) = 2T(3/3) + 3^2 = 9$*. Which makes this statement true. So let us choose two values that make this work. So let us choose* $C = 5$ *and* $n_0 = 3$*. This will give us* $9 \leq 225$*. So this is true.*

Problem 5. *Suppose someone proposed a slow merge procedure for the merge sort, where the slow merge procedure will take* n^2 *time instead of the* $\Theta(n)$ *time. Using such a slow merge procedure, the time complexity of the new merge sort can be expressed by the following recurrence:* $T(n) = 2T(n/2) + n^2$*. Using the recursion tree method, show the time cost of the new merge sort is* $\Theta(n^2)$*, i.e.,* $T(n) = \Theta(n^2)$*.*

Answer 5. *So for this let us look at the recursion tree below.*



Now if add the levels up, we will find that the result will be dominated by the n^2 *term. So we can say that this will have a time cost of* $\Theta(n^2)$*.*