

Atanas Dimitrov
Homework #1
CSCI 6470
Dr. Cai

Solution to problem 1:

I. First we prove $T(n)=O(n \cdot \log n)$

Inductive hypothesis:

Assume that the bound holds for $n/2$, then

$$T(n/2) \leq c \cdot (n/2) \cdot \log(n/2)$$

Inductive step:

Substituting we get:

$$\begin{aligned} T(n) &\leq 2 \cdot (c \cdot (n/2) \cdot \log(n/2)) + n + 1 \\ &= c \cdot n \cdot (\log n - \log 2) + n + 1 \\ &= c \cdot n \cdot \log n - c \cdot n + n + 1 \\ &\leq c \cdot n \cdot \log n \text{ for } c=2 \text{ } n_0=1 \end{aligned}$$

Base Case:

$$T(1)=1$$

$$T(2)=2 \cdot (T(1)) + 2 + 1 = 5 \quad c \cdot 2 \cdot \log 2 = c \cdot 2 \cdot 1$$

$$T(3)=2 \cdot (T(2)) + 3 + 1 = 6 \quad c \cdot 3 \cdot \log 3 = c \cdot 3 \cdot 1.6$$

Let $n_0=2$ then if we choose $c > 4$ The property holds for the base case.

Thus we have shown that $T(n)=O(n \cdot \log n)$.

II. Second we must prove that $T(n)=\Omega(n \cdot \log n)$

Inductive hypothesis:

Assume that the bound holds for $n/2$, then

$$T(n/2) \geq c \cdot (n/2) \cdot \log(n/2)$$

Inductive step:

Substituting we get:

$$\begin{aligned} T(n) &\geq 2 \cdot (c \cdot (n/2) \cdot \log(n/2)) + n + 1 \\ &= c \cdot n \cdot (\log n - \log 2) + n + 1 \\ &= c \cdot n \cdot \log n - c \cdot n + n + 1 \\ &\geq c \cdot n \cdot \log n \text{ for } c=1 \text{ } n_0=1 \end{aligned}$$

Base Case:

$$T(1)=1$$

$$T(2)=2 \cdot (T(1)) + 2 + 1 = 5 \quad c \cdot 2 \cdot \log 2 = c \cdot 2 \cdot 1$$

$$T(3)=2 \cdot (T(1)) + 3 + 1 = 6 \quad c \cdot 3 \cdot \log 3 = c \cdot 3 \cdot 1.6$$

Let $n_0=2$ then if we choose $c=1$ The property holds for the base case.

Thus we have shown that $T(n)=\Omega(n \cdot \log n)$.

From **I.** and **II.** it follows that $T(n)=O(n \cdot \log n) \cap \Omega(n \cdot \log n)$, thus $T(n)=\Theta(n \cdot \log n)$

Solution to problem 2:

a)

The algorithm takes as inputs the beginning and ending point of the input array (or sub-array) of

numbers. First it makes 1 comparison and switch of two values thus this will be constant time. The next comparison is also constant time. Then it calculates a third of the size of the array (roughly) and takes under consideration a sub-array which is three times the size of the original array as follows:

1. first 2/3
2. last 2/3
3. first 2/3

Thus we can deduce that:

$$T(n) = 3 * (2 * n/3) + C \text{ where } C \text{ is a constant.}$$

b)

The worst case running time is larger than $n * \log n$. Roughly we can approximate using the recurrence's recursion tree. Each time it will branch-off into 3 instances each 2/3 the size of the parent. Thus, the total number of sub-problems will be:

Root Level..... 3^0
 Level 1..... 3^1
 Level 2..... 3^2
 .
 .
 Level (approximately $\log_{3/2}(n)$)..... $3^{\log_{3/2}(n)}$

Sum of all sub problems using the the formula is approximately:

$$\sum_{i \in (1.. \log_{3/2}(n))} 3^i = (3^{\log_{3/2}(n)} - 1) / (3 - 1)$$

This quantity is roughly $O(3^{\log_{3/2}(n)})$ (consider $c=1/2, n_0=1$)

But $3^{\log_{3/2}(n)}$ is by properties of log:

$$n^{\log_{3/2}(3)}, \text{ which is } n^{(2.709511292)} \text{ much larger than}$$

$O(n * \log n)$.

Solution to problem 4:

In the case of finding the maximum (same applies in finding the minimum) in a list of n integer elements we can apply the decision tree technique. The lower bound we are looking for is the height of the tree with at least the minimum number of leaves present. In this case the minimum number of leaves will be n since in the general case (when we don't know which element is the largest) we must have at least n leaves - one for each element. Thus, we have:

- l - # reachable leaves
- h - height of the tree
- n - # elements in the list

Since the decision tree is a binary tree and we need at least n leaves, then

$$n \leq l \leq 2^h$$

thus, $n \leq 2^h$, and since log base 2 function is monotonically increasing we could say that

$$\log(n) \leq \log(2^h)$$

$$\log(n) \leq h$$

So we have $T(n) = \Omega(\log(n))$

Solution to problem 5:

We know (as the textbook has proven) that "Given n d-digit numbers in which each digit can take up to k possible values, RADIX-SORT correctly sorts these numbers in $\Theta(d * (n+k))$ time."

(Lemma 8.3 p. 172) . The number of digits for n numbers between 0 and n^2-1 will take approximately $\log_{10}(n^2)$. Obviously sorting the numbers as decimal numbers will not give us $\Omega(n)$ since as we can see $\log_{10}(n^2) \cdot (n^2+10)$ grows exponentially rather than linearly. We can however start increasing the number of digits available and thus come up with a system large enough to reduce the time complexity when RADIX-SORT is used to linear time. We can notice that if we use system base n then for n^2 numbers we will have $d = \log_n(n^2) = 2$ digits. Using Lemma 8.3(p.172) we have:

$\Theta(d \cdot (n+k))$, substituting yields - $\Theta(2 \cdot (n+n)) = \Theta(2 \cdot n)$ which is $\Theta(n)$ for $c=2, n_0=1$.

Thus the algorithm will proceed as follows:

Input: Non-sorted list of elements A , maximum number of digits in each element on the list d .

Output: A is sorted.

Algorithm:

```

RADIX-SORT (A, d)
  n ← size of A
  for i ← 1 to d ❖ In n base counting system
    do use a stable sort to sort A on digit A

```

Solution to problem 6:

Input: list of elements A , left bound of A l , right bound of A r , i - the place of the element we are looking for in the list when list is sorted.

Output: the value of the i -th element in the list when the list is sorted.

Algorithm:

```

Select (A, l, r, i)
  if l == r
    return A[l]
  pivot = Black-box (A, l, r)
  k ← pivot - l + 1
  if k == i
    return A[k]
  else
    if i < k
      Select (A, l, k - l, i)
    else
      Select (A, k + l, r, i - k)

```

Since black-box finds the median in linear time - $O(n)$ - and since k is the median then after each iteration we will only consider $1/2$ of the original list. Thus the recursion for the algorithm is:

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

We will guess that the solution to the recurrence is $O(n)$ and prove it by induction.

Inductive Hypothesis:

Suppose that $T(n/2) = O(n)$, in other words: $T(n/2) \leq cn/2$ for some $c > 0$ and $n_0 > 0$.

Inductive Step:

Show that $T(n) \leq cn$

We substitute in $T(n)$ as follows:

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

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

This equality always holds when $c/2 + d \leq c$; $d \leq c/2$

Base Case:

$$T(1) = 1 \quad c \cdot n = c \cdot 1$$

$$T(2) = 1 + 2 = 3 \quad c \cdot n = c \cdot 2$$

$$T(3) = 1 + 3 = 4 \quad c \cdot n = c \cdot 3$$

We can clearly see that these inequalities hold for $c=2$ (and $c>2$)

This completes the inductive proof.

Solution to problem 3:

If we take under consideration the properties of the columns and rows, we can clearly see that if we split the original matrix in 4 parts we can eliminate at least one if we compare the values of the top left and bottom right corners of each sub-matrix to the key we are searching for. Thus adopting this approach $T(N)=3T(N/4-5) + C$, where C is constant and expresses the time needed for comparison with the diagonal elements described above.

Inputs: Key that we are searching for, 2 dimensional array A, and its endpoints as Cartesian couples (initially 1,1,n,m where n=column #, m=row #) within the table.

Outputs: True if the key is in the array and false if it isn't.

Algorithm:

```
matrix_search(key, A, x1, y1, x2, y2)
{
    if x1=x2 and y1=y2 and A[x1][y1] = key
        return true
    else
        if A[x1][y1]=key or A[x1][y2]=key or A[x2][y1]=key or A[x2][y2]=key
            or A[(x1+x2)/2][(y1+y2)/2] = key
                return true
        else
            if key > A[(x1+x2)/2][(y1+y2)/2]
                if matrix_search(A, (x1+x2)/2, (y1+y2)/2, x2, y2) is true
                    return true
            if key < A[(x1+x2)/2][(y1+y2)/2]
                if matrix_search(A, x1, y1, (x1+x2)/2, (y1+y2)/2) is true
                    return true
            if matrix_search((x1+x2)/2, y1, x2, (y1+y2)/2) is true
                return true
            if matrix_search(x1, (y1+y2)/2, (x1+x2)/2, y2) is true
                return true
    return false
}
```