Joshua Sloan
University of Wyoming
jsloan3@uwyo.edu

Homework 2
COSC 5110 - Analysis of Algorithms

1 Exercise 2.23

An array A[1...n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like integers, and so there can be no comparisons of the form: is $A[i] > A[j]$?. (Think of the array elements are GIF files, say). However you can answer questions of the form: is $A[i] = A[j]$? in constant time.

(a)
Show how to solve this problem in $O(nlogn)$ time. (Hint: Split the array A into two arrays A_1 and A_2 of half the size. Does knowing the majority elements of A_1 and A_2 help you figure out the majority element of A? If so, you can use a divide-and-conquer approach.)

majority(A[1...n]**)** //input is an array of length n
if length(A) == 1 **then**
return A[1]
end if
k = $\lfloor \frac{n}{2} \rfloor$
A_1 = A[1...k]
A_2 = A[k+1...n]
majA_1 = majority(A_1)
majA_2 = majority(A_2)
if majA_1 == majA_2 **then**
return majA_1
else
numA_1 = count(A[1...n],majA_1) //linear time for-loop that counts number of instances in array
numA_2 = count(A[1...n],majA_2) //linear time for-loop that counts number of instances in array
if numA_1 > k+1 **then**
return majA_1
else if numA_2 > k+1 **then**
return majA_2
else
return FALSE
end if
end if
NOTE: I included the count function for the sake of readability and to cut down on large-body redundancy (given that it's a very straight-forward function, as it is simply a single for-loop which iterates through the array and increments a counter, initialized to 0, by 1 every time A[i] == the second parameter, returning the counter value).
The algorithm splits the work into 2 sub-problems which are 1/2 of the size of the original until the base case where the sub-problem size is 1.
In the worst case, 2 calls to the linear run-time count (to decide which the majority is when the two branches disagree) is needed to be called at each comparison.

This makes the recurrence relation: T(n) = 2T(n/2) + 2O(n), which by the Master Theorem is O(nlogn).
□

(b)
Can you give a linear-time algorithm? (Hint: Here's another divide-and-conquer approach:
- Pair up the elements of A arbitrarily, to get n/2 pairs
- Look at each pair: if the two elements are different, discard both of them; if they are the same, keep just one of them

Show that after this procedure there are at most n/2 elements left, and that they have a majority element if A does.)

```
linMaj(A[1...n]) //input is an array of length n
  if length(A) is odd then
    counter = 0
    for i=1 to (n-1) do
      if A[n] == A[i] then
        counter = counter + 1
      end if
    end for
    if counter > n/2 then
      return A[n]
    else
      linMaj(A[1...(n-1)])
    end if
  else if length(A) is even then
    i = 1
    B[]
    while i ≯ length(A) do
      if A[i] == A[i+1] then
        B.append(A[i])
      end if
      i = i + 2
    end while
    if length(B) == 0 then
      return FALSE
    else if length(B) == 1 then
      return B[1]
    else
      linMaj(B[1...length(B)) //we know B to have length n/2 in the worst case (S.T. every pair ∈ B)
    end if
  end if
```

We know this algorithm is linear. Even in the worst case, every time through the algorithm, the length of the array is odd and we need to make a linear check to see if the last element is the majority element (before removing it). The algorithm will then pair the maximum of n/2 elements (the last of which not being the majority element) in linear time and it repeat the process on itself.

Because we are only dividing the problem into one sub-problem of at most half the size at every iteration, we can guarantee that we are never performing more than a constant amount of linear work operations.
This is because our recursive calls are bounded by the harmonic series:

$\sum_{k=0}^{log_2 n} \frac{1}{2^k} = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + ...$

which is bounded to be < 2.

This makes the recurrence relation: T(n) = T(n/2) + 2O(n), which by the Master Theorem is O(n).

We know that at each pairing, there are at most n/2 elements in the new array. If this array has more than n/4 elements (i.e. at least n+1/4) that are the same, then that must also be the majority of n (belonging to A[1...n]) given that there are 2 of each element in A for each element represented in B and 2 * (n/4) = n/2, so if B has a majority $>$ n/4, then A has a majority $>$ n/2. $\square$.

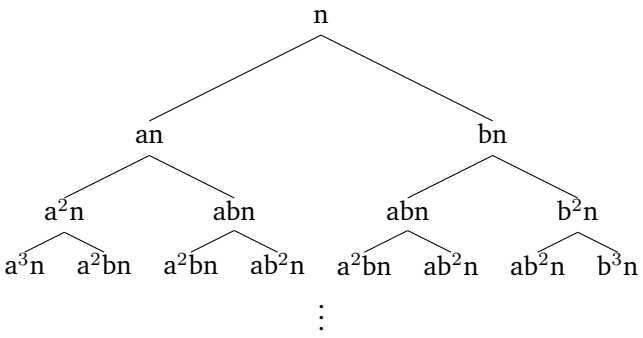
2

Let

$$T(n) = T(an) + T(bn) + cn, \tag{1}$$

where $1 > a \geq b > 0$ and $c > 0$ are constants.

(a) Prove that the total work performed across the k^{th} level of the recursion tree is at most $c(a+b)^k n$, for every $k \geq 0$.



where,
k = 0 $\leq c(a+b)^0 n$ = cn
k = 1 $\leq c(a+b)^1 n$ = c(an+bn)
k = 2 $\leq c(a+b)^2 n$ = c(a^2n + abn + abn + b^2n) = c(a^2 + 2ab + b^2)n
k = 3 $\leq c(a+b)^3 n$ = c(a^3n + a^2bn + a^2bn + ab^2n + a^2bn + ab^2n + ab^2n + b^3n) = c(a^3 + 3a^2b + 3ab^2 + b^3)n

Proof by induction:
Assume the worst case for each level, in which the level is complete for all branches through that respective level (i.e. no branches terminate early until the entire tree reaches its depth). I will show that the work for any level k is = $c(a+b)^k n$. In this manner, any branches that do terminate early will only reduce the amount of work at the kth level S.T. the work at level k is $< c(a+b)^k n$, completing the statement: work at the kth level $\leq c(a+b)^k n$.

Base case: k = 0
If the amount of work performed at each level is c ·n, the root node is trivial give that we can denote the work performed on the level of the 0^{th} level to be c · (a+b)^0 · n = c · 1 · n = c · n = $c(a+b)^k n$, where k = 0.

I will extend this one more level to show that the root node is not a special case:
k = 1
Given that n will divide into two leaves, an and bn, we can denote the work done at the level as c · (an + bn) = c · (a+b) · n = c · (a+b)^1 · n

Inductive step:
Assume k = i holds.
Then the total work at the ith level = $c(a+b)^i n$, where $(a+b)^i = \sum_{j=0}^{i} \binom{i}{j} a^{i-j} b^j$ **(By the binomial theorem)**

Show k = i+1 holds:
The total work for the i+1 level = $c(\sum_{j=0}^{i+1} \binom{i+1}{j} a^{i-j+1} b^j) n$
= $c(\binom{i+1}{0} a^{i+1-0} b^0 + \binom{i+1}{1} a^{i+1-1} b^1 + \binom{i+1}{2} a^{i+1-2} b^2 ...) n$
= $c(a+b)^{i+1} n$

Because we have shown that in the worst case, every level k has total work = $c(a+b)^k n$ (and in better cases the total work only decreases),
we can confidently say that the total work at the kth level $\leq c(a+b)^k n$, k ≥ 1. □

NOTE: Intuitively, this makes sense given that you can see how each complete level k adds up to $c(a+b)^k n$ from the above tree.

(b) For which values of k is the total work performed across the k^{th} level of the recursion tree equal to $c(a+b)^k n$?

The total work performed at the kth level is equal to $c(a+b)^k n$ for all levels for which k $\leq$ the depth of the shortest branch.

In other words, for all values where $k \leq log_x n$, where x = (min[a,b])$^{-1}$.

(c) What is the depth of the recursion tree?

The depth of the recursion tree, $d = log_x n$, where x = (max[a,b])$^{-1}$.

Note that this is very similar to the answer for 2b, except we are trying to find the depth of the longest branch instead of the shortest.

(d) Solve for T(n) in each of the following cases:
i. $a+b<1$.
Because a+b < 1, in the worst case the work done at each level will always be at most $c(a+b)^k n$, S.T. $(a+b)^k$ will always be shrinking at every subsequent level.

We can denote $c(a+b)^k n$ by the geometric sum:
$\sum_{k=1}^{d} c(a+b)^k n$, where d is our depth answer from problem 2c
$= cn \sum_{k=1}^{d} (a+b)^k \Rightarrow cn\frac{(a+b)^{d+1}-1}{(a+b)-1}$
We can see that given (a+b) < 1, the denominator will be a negative number $>$ -1 whereas the numerator will be an even smaller number $>$ -1.
Because both the numerator and denominator are negative, they will cancel, meaning that the sum, s, will just be a small constant number S.T. c $\cdot$ s $\cdot$ n = O(n).

Additionally, if we want to check that T(n) = O(n), then we can show that T(n) = T(an) + T(bn) + cn $\leq$ c(an) + c(bn) + xn for some constant c (where x bounds the O(n) time).
T(n) = T(an) + T(bn) + cn $\leq$ c(an) + c(bn) + xn
$\Rightarrow$ c(an + bn) + xn
$\Rightarrow$ (c(a + b) + x)n, which we need to be $\leq$ cn
so, c(a+b)+x $\leq$ c
$\Rightarrow$ x $\leq$ c - c(a+b)
$\Rightarrow$ x $\leq$ c(1-(a+b))
Because we know that a+b < 1, then we know that there exists some value for x, such that x $\leq$ c(1-(a+b)).
$\therefore$ T(n) = O(n). $\square$

ii. $a+b=1$.
Because a+b = 1, in the worst case the work done at each level will always be at most $c(a+b)^k n$, S.T. $(a+b)^k$ is always 1. Therefore, we are performing c $\cdot$ n work at each level for $log_x n$ (where x = (max[a,b])$^{-1}$) total levels (depth of the tree). The total amount of work is then O(c $\cdot$ n $\cdot$ logn) or simply: O(nlogn).

Even considering the lower bound, we would prune all branches to the length of the shortest branch (i.e. whichever terminates to reach the base case first in the case that a $\neq$ b). The work performed is still c $\cdot$ n work at each level for $log_x n$, where x = (min[a,b])$^{-1}$, total levels (depth of shortest branch). The total amount of work is then O(c $\cdot$ n $\cdot$ logn) or simply: O(nlogn). $\square$

iii. $a+b>1$.
Because a+b > 1, in the worst case the work done at each level will always be at most $c(a+b)^k n$, S.T. $(a+b)^k$ will always be growing at every subsequent level.

We can denote $c(a+b)^k n$ by the geometric sum:
$\sum_{k=1}^{d} c(a+b)^k n$, where d is our depth answer from problem 2c
$= cn \sum_{k=1}^{d} (a+b)^k \Rightarrow cn\frac{(a+b)^{d+1}-1}{(a+b)-1}$
We can see that given (a+b) > 1, the denominator will be a small constant whereas the numerator grows by a factor of logn.

In this manner, the work performed at the final level will be c $\cdot$ $(a+b)^{log_d n}$ $\cdot$ n (where where d = (max[a,b])$^{-1}$

= depth of the tree). To find the total work done over all levels, we must take c · $(a+b)^{logn}$ · n work (where the base of logn is dependent on the level k of the tree being analyzed) for all $log_x n$ levels, such that we perform O(n^{logn}logn) total work. □

3

The median-of-medians algorithm we covered in class begins by dividing the array into blocks of 5. What happens if we modify the algorithm to use a different block size? Bound the number of comparisons in each of the following cases. (You may use your results from the previous problem.)

(a) Block size 3.
We can guarantee the median of medians will be > n/3 elements and < n/3 elements, S.T. in the worst case,
For block size of 3: T(n) = T($\frac{n}{3}$) + T($\frac{2n}{3}$) + cn
Because we know that 1/3 + 2/3 = 1, from the previous problem we know that the number of comparisons is bounded by c · nlog(n) = O(nlogn).

(b) Block size 5.
In the worst case,
For block size of 5: T(n) = T($\frac{n}{5}$) + T($\frac{7n}{10}$) + cn
Because we know that 1/5 + 7/10 = 9/10 < 1, from the previous problem we know that the number of comparisons is bounded by c · n = O(n).

(c) Block size 7.
In the worst case,
For block size of 7: T(n) = T($\frac{n}{7}$) + T($\frac{10n}{14}$) + cn
Because we know that 1/7 + 10/14 = 12/14 < 1, from the previous problem we know that the number of comparisons is bounded by c · n = O(n).

(d) Block size 9.
In the worst case,
For block size of 9: T(n) = T($\frac{n}{9}$) + T($\frac{13n}{18}$) + cn
Because we know that 1/9 + 13/18 = 15/18 < 1, from the previous problem we know that the number of comparisons is bounded by c · n = O(n).

(e) $Extra\ credit$: Block size $2k+1$, for a general $k \geq 1$.
In the worst case,
For block size of $2k+1$: T(n) = T($\frac{n}{2k+1}$) + T($\frac{(3k+1)n}{4k+2}$) + cn
Because we know that n/2k+1 + (3k+1)n/4k+2 = 1 when k=1 and n/2k+1 + (3k+1)n/4k+2 < 1 when k ≥ 2, from the previous problem we know that the number of comparisons is either bounded by c · nlog(n) = O(nlogn), or c · n = O(n), depending on the value of k.

(f) Compare your above results [(a)-(d), or (a)-(e) if you did (e)] and determine which block size minimizes the number of comparisons.

Because (b) through (d) and (e) where k≥2 all belong to the same complexity class (i.e. linear), we are only able to remove (a) and (e) where k=1 from consideration for minimizing the number of comparisons, given that they belong to a higher complexity class.
To decide which block size performs the best, we must derive the number of constant comparisons that need to be made for each block size.

Starting with block size 5: We want to show T(n) ≤ cn for some constant c.
Suppose that this is true, then we will try to find a c that works.
T(n) ≤ T($\frac{n}{5}$) + T($\frac{7n}{10}$) + an (where a is the number of comparisons that bounds the O(n) time).
≤ c · n/5 + c · 7n/10 + an
= (9c/10 + a)n, which we need to be ≤ cn.
9c/10 + a ≤ c ⇒ c ≥ 10a
so, c = 10a ∴ T(n) ≤ 10an, where an is the number of comparisons needed to sort all of the n/5 groups
S.T. an = (n/5) * $\binom{5}{2}$
= 10n/5 ⇒ a=2, meaning T(n) = 20n.

Similarly for block size 7: We want to show T(n) $\leq$ cn for some constant c.
Suppose that this is true, then we will try to find a c that works.
T(n) $\leq$ T($\frac{n}{7}$) + T($\frac{10n}{14}$) + an (where a is the number of comparisons that bounds the O(n) time).
$\leq$ c $\cdot$ n/7 + c $\cdot$ 10n/14 + an
= (12c/14 + a)n, which we need to be $\leq$ cn.
12c/14 + a $\leq$ c $\Rightarrow$ 2c $\geq$ 14a
so, c = 7a $\therefore$ T(n) $\leq$ 7an, where an is the number of comparisons needed to sort all of the n/7 groups
S.T. an = (n/7) * $\binom{7}{2}$
= 21n/7 $\Rightarrow$ a=3, meaning T(n) = 21n.

Similarly for block size 9: We want to show T(n) $\leq$ cn for some constant c.
Suppose that this is true, then we will try to find a c that works.
T(n) $\leq$ T($\frac{n}{9}$) + T($\frac{13n}{18}$) + an (where a is the number of comparisons that bounds the O(n) time).
$\leq$ c $\cdot$ n/9 + c $\cdot$ 13n/18 + an
= (15c/18 + a)n, which we need to be $\leq$ cn.
15c/18 + a $\leq$ c $\Rightarrow$ 3c $\geq$ 18a
so, c = 6a $\therefore$ T(n) $\leq$ 6an, where an is the number of comparisons needed to sort all of the n/9 groups
S.T. an = (n/9) * $\binom{9}{2}$
= 36n/9 $\Rightarrow$ a=4, meaning T(n) = 24n.

For the arbitrary block size of 2k+1:
We are only concerned about showing the relation for values of k $\geq$ 2, given what we know about the time complexity for 2k+1 when k = 1 (i.e. that it is of a higher complexity class than when k takes values $\geq$ 2).
We want to show T(n) $\leq$ cn for some constant c.
Suppose that this is true, then we will try to find a c that works.
T(n) $\leq$ T($\frac{n}{2k+1}$) + T($\frac{3k+1)n}{4k+2}$) + an (where a is the number of comparisons that bounds the O(n) time).
$\leq$ c $\cdot$ n/2k+1 + c $\cdot$ (3k+1)n/4k+2 + an
= ((3k+3)c/4k+2+ a)n, which we need to be $\leq$ cn.
(3k+3)c/4k+2 + a $\leq$ c $\Rightarrow$ (3k+3)c + a(4k+2) $\leq$ c(4k+2) $\Rightarrow$ kc - c $\geq$ 4ak+2a
so, c = (4ak+2a)/(k-1) $\therefore$ T(n) $\leq$ (4ak+2a)an/(k-1), where an is the number of comparisons needed to sort all of the n/2k+1 groups
S.T. an = (n/2k+1) * $\binom{2k+1}{2}$
= ($2k^2$n+2kn)/4k+2 $\Rightarrow$ a=($2k^2$+2k)/4k+2.
This means that T(n) = is minimized when k is the smallest value for k S.T. k $\geq$ 2.

So, in this case the optimal value for k to minimize the number of comparisons is k = 2
plugging in k = 2 gives us (2(2)^2 + 2(2))/(4k+2) = 16+4/10 = 20/10 = 2 and therefore a = 2,
which is equivalent to our a value for block size 5 which has T(n) = 20n.

$\therefore$ the block size that minimizes the number of comparisons is 5. □

I worked with Mason and Danny on this assignment.