[03–60–454] Assignment 2

Stephen Nusko
103693282

March 10, 2015

1 Question 1

1.1 English description of the Algorithm

This algorithm will find the most common element in a multiset that has a total order. It accomplishes this by assuming the following operations hold.

1. It is possible to iterate through the list in linear time.
2. It is possible to access any particular element in constant time.
3. It is possible to get the size of a list in constant time.

The algorithm finds the most common element in the list L and returns it. It is best described recursively though it is laid out iteratively in the pseudo-code below. The algorithm starts by checking the whole list for an element that occurs 50% or more in the list, if such an element is found then it returns this element. However in the likely case it doesn't find such an element. The algorithm will proceed to partition the matrix around a pivot that ensures a reasonable split. It will track where the partition points are, in each further call the algorithm will then have the original list and a list of partition points. It starts by checking each sublist between the partition points for a 50% or more majority, if it finds one it ensures that no other sublist has an element that has more occurrences. It then returns that element. Otherwise it proceeds much like the first time, each sub list get a pivot selection and gets partitioned. The algorithm then repeats.

1.2 Pseudo-Code and Examples

We will start with an example of the algorithm, and the proceed to lay out the pseudo-code for all the methods used in our algorithm.

Given the multiset L = $\{a, f, b, b, e, c, b, g, a, i, b\}$. The algorithm will insert into a new list partition_list 1, and 11 marking the beginning and the end.
The algorithm will then check to see if L contains an element with a 50% or more majority between 1 and 11. Since $N = 11$ in this case and the element b has 4 occurrences no majority element is found.
The algorithm will then select an approximate median to use. It does this by using the method select. Select will subdivide L into three pieces $L_1 = \{a, f, b, b, e\}$ and $L_2 = \{c, b, g, a, i\}$, and finally $L_3 = \{b\}$. Since it knows the size of each list we can find the median in constant time. This will find $L_{m_1} = b$ and $L_{m_2} = c$ and $L_{m_3} = b$. Select will then return the median of the found medians, in this case L_{m_1}.
We use this value as a pivot to partition L. Resulting in L looking like $\{a, a, b, b, b, b, f, e, c, g, i\}$. We insert the number 7 into partition_list so that it looks like $\{1, 7, 11\}$.
The algorithm will then check to see if the first partition in the interval $[1, 6]$ contains a majority. In this case b has a $\frac{2}{3}$ majority, the algorithm will then check the interval $[7, 11]$ to ensure that the second partition doesn't have a larger majority.
When it doesn't find one it will return the element b since it has the most frequent occurrences.

Algorithm Select(L, K);
described in the CourseWare, Chapter 3, slide 27.

Algorithm check_for_majority(L);
Input: A list L of N elements.
Output: A pair where the first is an element that has a 50% majority in the list L or $NULL$ otherwise, and the second is the count of that element found in the list.

begin
let counter = 0
let potential_element = $NULL$
for i = 1, . . ., N, do
if L[i] == potential_element
$counter = counter + 1$
else if counter == 0, then
potential_element = L[i]
counter = 1
else
$counter = counter - 1$
end if
end for
if potential_element == $NULL$, then
return ($NULL$, 0)
end if
set counter = 0
for j = 1, . . ., N, do
if L[j] == potential_element, then
counter = counter + 1
end if
end for
if $\frac{counter}{N} \geq 0.5$
return (potential_element, counter)
else
return ($NULL$, 0)
end if
end

Algorithm split(L, lower, upper, var splitpoint)
Described in the CourseWare Chapter 2, slide 13.

Algorithm insert_after(L, node, new_node)
Input: A single linked list L made up of nodes, a node in the list, and the new node to be inputted.
Output: The list L will be modified to have new_node be after node.
Note: that a node has the following structure
node {
element // the element being stored in the node.
next // which points to the next node
}

begin
new_node.next = node.next
node.next = new_node
end

Algorithm find_most_frequent_element(L)
Input: A list L of N elements from a totally ordered set representing a multiset
Output: A element in the list that has the most occurrences, if there are multiple elements the first found is returned.

begin
    let partition_list = be a forwarded linked list with the elements {1, N}
    let has_majority = $false$
    let found_element = $NULL$
    while has_majority == $false$, do
        let max_count = $-\infty$
        for i = 1, ..., $size(partition\_list) - 1$, do
            let part_point$_i$ = partition_list[i]
            let part_point$_{i+1}$ = partition_list[i+1]
            let (element, count) = $check\_for\_marjority$(set[$part\_point_i, \ldots, part\_point_{i+1}$])
            if element ! = $NULL$, then
                if (count > max_count) then
                    found_element = element
                    has_majority = $true$
                    max_count = count
                end if
            end if
        end for
        if (has_majority == $true$) then
            $return$ found_element
        end if
        let i = 1
        while i $\leq size(partition\_list) - 1$, do
            let part_point$_i$ = partition_list[i]
            let part_point$_{i+1}$ = partition_list[i+1]
            let pivot = $select(L, \lfloor \frac{(part\_point_i + part\_point_{i+1})}{2} \rfloor)$
            let partition_point = $split$(L, $part\_point_i, par\_point_{i+1}, pivot$)
            insert_after (partition_list, part_point$_i$, partition_point)
            i = i + 2
        end while
    end while
end

1.3 Proofs of Correctness

We will prove correctness for each function used as they are listed in the pseudo-code section.

1.3.1 Select(L, k)

We will only prove only the special case of Select(L, $\lfloor \frac{n}{2} \rfloor$).
(By Induction)
Base Case: for a list L of size 1. The algorithm will split the list into only one group since $\lceil \frac{n}{5} \rceil = 1$, Since we have only one element we can find the median of the sub list, and since it is the only element we have found the median of L.
Therefore, the base case is proven.

Inductive Step: Assume that select finds the $\lfloor \frac{n}{2} \rfloor$-smallest element on all lists of size $m < n$.

For a list of size n, the list will be divided into $\lceil \frac{n}{5} \rceil$ groups, from which the median of each group is found. The algorithm is then used to find the median of these medians m^*. By the inductive hypothesis Select will find the correct median.
We then have three cases.
Case 1: $|L_\prec| < \lfloor \frac{n}{2} \rfloor \leq |L_\prec| + |L_=|$
In this case the median is contained in the elements that are equal to m^*. So the algorithm returns m^* which is the median, so we have selected the $\lfloor \frac{n}{2} \rfloor$-smallest element.

Case 2: $\lfloor \frac{n}{2} \rfloor \leq |L_\prec|$
In this case the element we are looking for is still in the first $|L_\prec|$ elements, By the inductive hypothesis Select($L_\prec$, $\lfloor \frac{n}{2} \rfloor$) will find the $\lfloor \frac{n}{2} \rfloor$-th smallest element.

Case 3: $\lfloor \frac{n}{2} \rfloor > |L_\prec| + |L_=|$
In this case the $\lfloor \frac{n}{2} \rfloor$-smallest element of L is in the $L_\succ$ list. We adjust Select by calling Select($L_\succ$, $\lfloor \frac{n}{2} \rfloor - |L_\prec| - |L_=|$), this by the inductive hypothesis will find the $\lfloor \frac{n}{2} \rfloor - |L_\prec| - |L_=|$-th element of $L_\succ$. However because we have discarded the first $(|L_\prec| + |L_=|)$-elements of L adding the discarded elements to the $L_\succ$ we will increase the position of the found element, making it the $(\lfloor \frac{n}{2} \rfloor - |L_\prec| - |L_=| + |L_\prec| + |L_=|)$-smallest element = $\lfloor \frac{n}{2} \rfloor$-smallest element.

In all cases we found the $\lfloor \frac{n}{2} \rfloor$-th smallest element. Therefore we have proven the inductive step, and the base case.
Therefore, for all n, select will correctly find the $\lfloor \frac{n}{2} \rfloor$-th smallest element.

1.3.2 check_for_majority(L)

The algorithm precedes by noting that when we have gone through the list the first time potential_element will either be equal to an element of the majority if it exists, and if no majority exists, some random element, which we will then verify by counting the element and ensuring it occurs $\geq \frac{1}{2}$ the time. This is the Boyer-Moore algorithm it has been very well studied and for the original paper please see this site (`http://www.cs.utexas.edu/~moore/best-ideas/mjrty/`) which has the original paper including a formal proof of its method.
We can however provide our own logical proof of the method. When the list is not empty the potential_element will be set to some element, because after it has been set in the first counter == 0, statement, it is never returned to NULL
For an element to have a majority it has to occur $\geq \frac{n}{2}$ times while all other elements are $\leq \frac{n}{2}$, if we note the fact that removing a pair of elements (a, b) where 'a' is a majority element, and b is not. The majority element will not change. By the end of the first loop we will have our potential candidate.
In a non-empty list the if statement will fail, however in an empty list we will correctly return ($NULL$, 0). The final loop will count the occurrences of the loop, and dividing by N will ensure we have found the majority element, otherwise ($NULL$, 0) will be returned.

1.3.3 split(L, lower, upper, var splitpoint)

Proven in the CourseWare, chapter 2 slide 15.

1.3.4 insert_after(L, node, new_node)

base case N = 0
A empty single linked list has a empty node HEAD that next node points to $NULL$, since N = 0 the list consists of just the HEAD, and HEAD.next = $NULL$, in line 1 the new_node pointer is set to $NULL$, and HEAD now points to the new_node, and the list has the following structure $HEAD \rightarrow new\_node$, as desired.
inductive step
Assume that for a list of size $k < n$ insert_after correctly inserts new_node after node. Let L be a list of size n, new_node is to be added in front of node. We have the structure as follows if node_succ is the node after

node.
$HEAD \to \ldots \to node \to node\_succ \to \ldots \to NULL$.
At line 1, new_node.next will be set to node.next resulting in new_node structure as follows
$new\_node \to node\_succ \to \ldots \to NULL$.
On line 2 node.next is set to new_node resulting in the structure
$HEAD \to \ldots \to node \to new\_node \to node\_succ \to \ldots \to NULL$.
We have correctly updated the list to have new_node to be after node.
Therefore for a list sizes insert_after will correctly insert the new_node after node.

1.3.5 find_most_frequent_element(L)

Note that for the maximum occurring element to have a majority we need $\frac{M}{n_d} \geq 0.5$, however each time we partition we partitions by the median element, so we split the list into two parts, so at a particular depth d we have $n_d = \frac{n}{2^d}$. So replacing that we get $\frac{M}{\frac{n}{2^d}} = \frac{M}{2^{-d}n} \geq \frac{1}{2}$, this implies that $\frac{M}{2^{-d}} \geq \frac{n}{2}$ which is what it means for an element to be a majority.
Therefore there will be $\lg(\frac{n}{M})$ partitions since this will give us list lengths of$\frac{n}{2^d}$, this fact will be used in the time complexity as well. However for us this proves that we will find a majority element based on the frequency on M. Next the if statement that checks for $NULL$ if it is true, then we have found a majority element. We save the count, and note that we should end the loop, however first the algorithm checks the rest of the list ensuring accuracy in case some other partition had more elements that were the same.
Next when we have successfully found a majority on that level of the iteration we do no more work, rather returning the found_element that has that majority. If we have not found the majority element, we insert a new partition point between each existing partition point, and by moving forward by two we skip the newly inserted partition.
Therefore the partitioning of the array is correctly handled by the correctness of select, and split, and insert_after. While the majority has been proven correct by previous math, and correctness of check_for_majority. Therefore regardless of the input find_most_frequent_element will return the correct result.

1.4 Time Complexity Analysis

1.4.1 Select(L)

Proven in the CourseWare to be $O(n)$.

1.4.2 check_for_majority(L)

1. lines 1 and two do constant work, say C_1 work.
2. the for loop on line 3 will execute N times,
3. the lines between 3, and 12, all do constant work C_2.
4. therefore the loop starting on line 3 and ending on 12 does $C_2 n$ work.
5. the if statement on line 13 does constant work C_3.
6. The statement on line 16 does constant work C_4.
7. the loop on line 17 will execute N times.
8. on the lines 18 to 20 each line does constant work of C_5
9. therefore the for loop running on line 17 then the work is $C_5 n$
10. the statements on line 22 to the end of the algorithm all take constant time C_6

Therefore the amount of work is $C_1 + C_2 n + C_3 + C_4 + C_5 n + C_6 \leq (|C_1| + |C_2| + |C_3| + |C_4| + |C_5| + |C_6|)n$
Therefore check_for_majority runs in $O(n)$ time.

1.4.3 split(L, lower, upper, var splitpoint)

Proven in the CourseWare to be $O(n)$.

1.4.4 insert_after(L, node, new_node)

Line 1, and line 2 both run in constant time, therefore it runs in C_1.
Therefore insert_after runs in $O(1)$ time.

1.5 find_most_frequent_element(L)

The loop will run until a majority element in one have the partitions is found. Because M is the number of occurrences of the element that is the most frequent. We must divide n in half M times before that element will achieve the majority in one of the partitions. This gives us the recursive depth of $\lg(\frac{n}{M})$. We must now prove that the work inside the loop maintains only $O(n)$ work to get the desired $O(n\lg(\frac{n}{M})+n)$ running time.
On the any given level the number of partitions is 2^d where d is the depth we have gone. Since we are partitioning around the median, each partition has approximately $\frac{n}{2^d}$ elements.
since all statements besides check_for_majority take constant time say C_1, and check_for_majority takes linear time with respect to its input length which in this case is $\frac{n}{2^d}$.
So each loop will go through (2^d-1) partitions doing $\frac{n}{2^d}$ work resulting in the loop taking $\frac{(2^d-1)n}{2^d} = n-\frac{n}{2^d} \leq cn$ for all $c, n \geq 1$
Therefore the first for loop runs in $O(n)$ time.

The if statement takes constant time C_2.
Much like the first for loop this loop will go through at most 2^d-1 partitions each of length $\frac{n}{2^d}$ elements.
As before all statements besides select, and split take constant time.
Like the first for loop select, and split are both linear time operations on their input sizes.
Therefore as before This for loop runs in n time.

Since we have found the loop depth for the majority element to be found is $\lg(\frac{n}{M})$ and we have found that each iteration of the loop does linear work, we conclude that $n\lg(\frac{n}{M}) \in O(n\lg(\frac{n}{M})+n)$.

2 Question 2

2.1 a

Insertion sort takes $O(n^2)$ time so insertion on $2\sqrt{n}+1$ elements takes $(2\sqrt{n}+1)^2 = 4n+4\sqrt{n}+1$
Once sorted the median can be found in $O(1)$ time.
Finally the only other work outside the recursive calls is the partitioning of the list. Split takes $(n-1)$ time (CourseWare) so we can add $n-1$. So all the work done outside the function making $f(n) = 4n+4\sqrt{n}+1+1+n-1 = 5n+4\sqrt{n}+1$
In addition there are two calls on the different partitions this results in the following recurrence.

$$T(n) = T(|L_{\precsim}|) + T(|L_{\succ}|) + 5n + 4\sqrt{n} + 1$$

2.2 b

We know we selected the median of a sorted $2\sqrt{n}+1$ elements so we know that both partitions are bounded below by $\frac{1}{2}(2\sqrt{n}+1) = \sqrt{n}+\frac{1}{2}$, since by dropping the $\frac{1}{2}$ we only make the bound slightly worse we do that for ease of calculations.
Well above there is $n-\frac{1}{2}(2\sqrt{n}+1) = n-\sqrt{n}-\frac{1}{2}$, again we drop the $-\frac{1}{2}$ which still bounds above the size of the partition.

Since we are looking for worst case performance we can without loss of generality assume that the maximum unbalanced partition will cause the worst performance. So we can rewrite the recurrence as

$$T(n) = T(\sqrt{n}) + T(n - \sqrt{n}) + 5n + 4\sqrt{n} + 1$$

2.3 c

2.3.1 Part 1

Provided on a separate piece of paper, to allow easy drawing.
Note that we find in section 3.4 d, we bound the maximum depth by $\sqrt{n}$ by expanding along the $T(n - \sqrt{n})$ recurrence. To help in the drawing I will bound the other side also.

1. n starting point depth 0.
2. $\sqrt{n} = n^{\frac{1}{2}}$ depth 1
3. $\sqrt{\sqrt{n}} = n^{\frac{1}{2^2}}$ depth 2
4. ...
5. $n^{\frac{1}{2^d}}$ depth d

To find the depth of this side of the tree we solve $\lfloor n^{\frac{1}{2^d}} \rfloor = 2$. We can choose 2 because we know how the recurrence expands at 2. $T(2) = T(\lfloor\sqrt{2}\rfloor) + T(\lfloor 2 - \sqrt{2}\rfloor) = T(1) + T(0)$

1. $\lfloor n^{\frac{1}{2^d}} \rfloor = 2$ starting point
2. $2 \leq n^{\frac{1}{2^d}} < 3$ definition of floor
3. $\lg 2 \leq \lg n^{\frac{1}{2^d}} < \lg 3$ lg is monotonic
4. $1 \leq \frac{\lg n}{2^d} < \lg 3$ log rules
5. $2^d \leq \lg n < 2^d \lg 3$ multiple everything by 2^d
6. $d \leq \lg \lg n < 2^d \lg 3$ log is monotonic, and log rules.

Therefore there is at most $1 + \lg \lg n$ levels (because we only went to $T(2)$) on the $T(\sqrt{n})$ side.

2.3.2 Part 2

First we note that on each level we add the respective constant work of $5n_i + 4\sqrt{n_i} + 1$ where each n_i is the size of n for each recurrence call given.
We can see that each recurrence call on a level will add one of these terms, and that it belongs to $O(n)$, which means it is bounded by some cn we can safely ignore it in our proofs of the levels average work, knowing we just need to increase our result by some constant factor.
On that regard we ignore each level's constant factors, and focus on only the expansion of n on each recurrence.

1. $T(\sqrt{n}) + T(n - \sqrt{n}) = (\sqrt{n}) + (n - \sqrt{n}) = n \leq cn$
2. $T(\sqrt{\sqrt{n}}) + T(\sqrt{n} - \sqrt{\sqrt{n}}) + T(\sqrt{n - \sqrt{n}}) + T((n - \sqrt{n}) - \sqrt{n - \sqrt{n}}) = \sqrt{\sqrt{n}} + \sqrt{n} - \sqrt{\sqrt{n}} + \sqrt{n - \sqrt{n}} + n - \sqrt{n} - \sqrt{n - \sqrt{n}} = n \leq cn$

Because we keep adding smaller and smaller terms, we will always find a value of n that will make the inequality true, as well as the fact that one side is subtracting exactly what we are adding from other calls.

2.4 d

We note that on one side the recurrence expands by $T(\sqrt{n})$ well on the other it is $T(n-\sqrt{n})$. $(n-\sqrt{n}) \geq \sqrt{n}, \forall n \geq 4$ therefore, when finding the depth of the tree we only care about the maximum depth. We only expand along the $T(n-\sqrt{n})$ call each time.

1. n first call depth 0.
2. $n - \sqrt{n} = n - a_1$ where $a_1 = \sqrt{n} \leq \sqrt{n}$, second call depth 1.
3. $n - a_1 - a_2$ where $a_2 = \sqrt{n - \sqrt{n}} \leq \sqrt{n}$, third call depth 2.
4. $\ldots$ Each call will ad an addition term that is $\leq \sqrt{n}$
5. $n - a_1 - a_2 - \cdots - a_q$ at depth q.
6. $\leq n - q\sqrt{n}$ $(\forall i) a_i \leq \sqrt{n}$

The last level will occur when $n - q\sqrt{n} = 1$

1. $n - q\sqrt{n} = 1$
2. $\frac{n-q\sqrt{n}}{\sqrt{n}} = \frac{1}{\sqrt{n}}$ Divide both sides by $\sqrt{n}$
3. $\frac{n}{\sqrt{n}} - \frac{q\sqrt{n}}{\sqrt{n}} = \frac{1}{\sqrt{n}}$ separate division on subtraction
4. $\sqrt{n} - q = \frac{1}{\sqrt{n}}$ cancel terms.
5. $\sqrt{n} - \frac{1}{\sqrt{n}} = q$ rearrange
6. $\sqrt{n} \geq q$ drop the subtraction of positive term.

$\therefore$ there the depth of the recurrence is at most $\sqrt{n}$.

2.5 e

We have found in section 2.3.2, that on each row we do cn work, while in section 2.4 we have found the maximum depth of the $T(n-\sqrt{n})$ side of the tree is $\sqrt{n}$.
therefore the amount of work done is $\sqrt{n} * cn = cn^{3/2}$, so $Qsort \in O(n^{3/2})$ this is better then Quicksort's worst case of $O(n^2)$.

3 Question 3

We are to prove that given any algorithm, that searches a string L of even length comprised only of 'a's and 'b's for the sequence 'ab' will have to examine every element in L.

Our adversarial argument will involve two pointers l and r, l will start pointing at 0 outside of the list, while r will start at $n+1$ again outside the list.
l represents the furtherest right 'b' we have seen, as such it will only move when we declare a location to be a 'b' that is farther to the right.
likewise r represents the furtherest left 'a' we have seen, as such it will only move when we declare a location be a 'a' that is farther to the left.

Our adversarial choices are as follows, if the character examined is to the left of l then respond with a 'b', and it is to the right of r respond with an 'a'.
In the instance it is between l and r we have two cases, if the index is even respond with a 'b' and move l

to that position, otherwise if the index is odd move r to it and respond with 'a'.

First we prove some helpful lemmas.

3.1 Lemma *

Lemma * proves that if at any time L contains a 'b' that follows after an 'a', then the sequence 'ab' must exist in L.

Proof by induction on the space between the 'a', and the 'b'

Base case $n = 0$
In this case there is no gap between 'a' and 'b', therefore L looks like '$\ldots ab \ldots$'
which contains the subsequence 'ab'.
$\therefore$ in the base case of no gap if a 'b' follows after an 'a' then 'ab' $\in L$.

Inductive step assume that for all gaps of size $k < n$ if an 'b' follows after an 'a' then 'ab' $\in L$.
let L look like the following '$\ldots a \ldots n$ gaps$\ldots b \ldots$', however since L is comprised of only 'a's or 'b's we have two cases.
Case 1: the character after 'a' is another 'a'
In this case L looks like '$\ldots aa \ldots n-1$ gaps$\ldots b \ldots$'. The distance between the second 'a' and the 'b' is $n-1$ which is less than n, therefore by the inductive hypothesis there exists a sequence 'ab' $\in L$.
Case 2: The character after 'a' is a 'b'
Then in this case the sequence 'ab' $\in L$.
$\therefore$ in all cases we proved that if for a gap of size $k < n$ between a 'a' followed by a 'b' there is a sequence 'ab' $\in L$, that for a gap of size n the same holds.

Therefore, we have proven the inductive step, and the base case.
Therefore, if a 'b' that follows after an 'a' is shown to be in L, then the sequence 'ab' $\in L$ $\square$

3.2 Lemma l

We prove that l is always on a even index, and points to an 'b' or outside the list.
Proof by induction on the number of times l is moved.
Base case: n = 0
l_0 is l after zero moves, since l was initially set to $0 = 2(0)$ and is therefore at an even index, and since $0 < 1$, l_0 is outside the list.
Since l_0 is outside the list, it is vacuously true that l_0 points to a 'b'.
inductive step assume $l_i = 2q, q \in \mathbb{Z}$ and L_{l_i} ='b'
Since the only case we move l is when we examine a character between l and r that is at an even index, we can see that l_{i+1} is also an even index, we declare it equal to a 'b', therefore l_{i+1} ='b'.
Therefore if $l_i = 2q, q \in \mathbb{Z}$ and L_{l_i} ='b' then $l_{i+1} = 2w, w \in \mathbb{Z}$ and $L_{l_{i+1}}$ ='b'
Therefore we have proven the base case and the inductive step, therefore after all moves $l_i = 2q, q \in \mathbb{Z}$ and L_{l_i} ='b'.

3.3 Lemma r

We prove that r is always on a odd index, and points to an 'a' or outside the list.
Proof by induction on the number of times r is moved.
Base case: n = 0
r_0 is r after zero moves, since r was initially set to n + 1, and n was even, therefore r is at an odd index, and since $n < n + 1$, r_0 is outside the list.
Since r_0 is outside the list, it is vacuously true that r_0 points to a 'a'.
inductive step assume $r_i = 2q, q \in \mathbb{Z}$ and L_{r_i} ='a'

Since the only case we move r is when we examine a character between l and r that is at an odd index, we can see that r_{i+1} is also an odd index, we declare it equal to a 'a', therefore l_{i+1} ='a'.
Therefore if $r_i = 2q, q \in \mathbb{Z}$ and L_{r_i} ='a' then $r_{i+1} = 2w, w \in \mathbb{Z}$ and $L_{r_{i+1}}$ ='a'
Therefore we have proven the base case and the inductive step, therefore after all moves $r_i = 2q, q \in \mathbb{Z}$ and L_{r_i} ='a'.

3.4 Proofs of adversarial choices

We now proceed with proving our algorithm for the adversary.
When a character is examined we have three cases,
Case 1: the character is to the left of l
In this case since l is the rightmost 'b', if we as the adversary were to say 'a' then by Lemma *, there would be a subsequence 'ab', and the algorithm could stop without examining all characters. However as stated above we will always say 'b' this means that to ensure that there is no 'a' the algorithm must examine all the spots, otherwise if it does not, and says there is no 'ab' we switch the location that was unexamined to a 'a' and have an input in which all choices would be the same but the answer would be wrong. Likewise if the algorithm were to say there was an 'ab' on the left we would show the case where all the characters to the left are 'b'. Therefore proving all characters to the left must be examined.
Case 2: the character is to the right of r
Similar to case 1, however since r represents the leftmost 'a', so for similar reasons as case 1, if the adversary were to say 'b' then the algorithm could stop, therefore the adversarial approach is to reply with a 'a'.
Again if the algorithm stops without examining all locations to the right of r then by switching the unchecked location to 'b' we will cause the algorithm to end incorrectly for this input, even though in all other ways the list is identical, also as in Case 1, if the algorithm says 'ab' is in right of L then we show the case where all characters to the right of r are 'a's.
Case 3: The charactor is inbetween L_l **and** L_r
We will prove that by choosing 'a' if the character is at an odd index, or 'b' if even index, and moving r, or l respectively, that any algorithm will be forced to exam all locations.
Proof by induction on the length of the even subsequence $|L_l \Leftrightarrow L_r|$ that is the subsequence between L_l and L_r. That each character in the subsequence is examined.
Base case: $|L_l \Leftrightarrow L_r| == 2(1)$
The our sequence is of the form '??', There are two cases in this.

sub case 1: no elements are checked

In this case if the algorithm says that 'ab' does not occur we set our sequence to be 'ab', proving them wrong, if they says it does occur we set the sequence to 'bb', thereby proving checking no elements will not guarantee correctness.

sub Case 2: one element is checked

If the first element is checked our adversarial choice is to say 'a' making our sequence 'a?', if the algorithm says that there is no 'ab' we set the second element to 'b', otherwise we set it to 'a'. So if they check the first element their algorithm is not correct.

If the second element is checked our adversarial choice will be to say 'b' making our sequence '?b', if the algorithm says that there is no 'ab' we set the first element to 'a', otherwise we set it to 'b'.

From the two cases above it is clear that for a subsequence of length 2, every element must be examined for the algorithm to choose correctly for all inputs.

inductive step Assume that for a sequence of size $|L_l \Leftrightarrow L_r| \leq 2q$ an algorithm must check every element.

Let $|L_l \Leftrightarrow L_r| = 2(q+1)|$ We have two cases the character chosen between l and r is at an even or odd index, we consider them separately.

sub Case 1: the charactor is at an odd index

We will move r to this index, and declare it to be an 'a', from case 2, we know that all characters to the right of r must be checked, this leaves us with the characters remaining between l and r.

Now since r on a odd index, and is moved to an odd index, the number of elements removed from the $|L_l \Leftrightarrow L_r|$ is at least 2. Therefore the length is now $|L_l \Leftrightarrow L_r| \leq 2(q+1) - 2 = 2q + 2 - 2 = 2q$. By the inductive hypothesis, every character between l and r must therefore be examined.

Therefore in sub case 1 all charactors must be examined.

sub case 2: the charactor is at an even index

We will move l to this index, and declare it to be an 'b', from case 1, we know that all characters to the left of l must be checked, this leaves us with the characters remaining between l and r.

Now since l was on a even index, and is moved to an even index, the number of elements removed from the $|L_l \Leftrightarrow L_r|$ is at least 2. Therefore the length is now $|L_l \Leftrightarrow L_r| \leq 2(q+1) - 2 = 2q + 2 - 2 = 2q$. By the inductive hypothesis, every character between l and r must therefore be examined.

Therefore in sub case 2 all charactors must be examined.

Therefore, the base case and the inductive step is proven, and we can say that if the character is chosen between L_l and L_r the algorithm will ensure all elements must be examined.

Therefore in all three cases the adversary will ensure that every character is examined.
Therefore for any algorithm to determine if the subsequence 'ab' occurs in a string of even length comprised of only 'a's and 'b's correctly, the algorithm must examine every element. □