Virpobe Luke Paireepinart **Assignment 2** **October 25th, 2009**

Algorithms Design and Implementation - Fall 2009

Texas State University – San Marcos

R4.4 – Show that the running time of the merge-sort algorithm on an n-element sequence is O(nlogn), even when "n" is not a power of 2.

Suppose we take a sequence of 15 elements.

After 1 division step, there will be 2 groups, one of 8 and one of 7 elements.

After 2 division steps, there will be 4 groups of 4, 4, 3, and 4.

After 3 division steps, there will be 8 groups of 2, 2, 2, 2, 2, 1, 2, 2.

After 4 division steps, there will be 16 groups of 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1

Now all groups are either 1 or 0 elements, so they will be merged back together.

After the first merge step, there will be 8 groups again, of 2, 2, 2, 2, 2, 1, 2, 2.

After the second merge step, there will be 4 groups, of 4, 4, 3, and 4.

After the third merge step, there will be groups of 8 and 7 elements.

After the fourth merge step, there will be one sorted sequence of 15 elements.

Now suppose we take a sequence of 10 elements.

After 1 division step, there will be 2 groups, 5 and 5.

After 2 division steps, there will be 4 groups, 2, 3, 2, 3.

After 3 division steps, there will be 8 groups, 1, 1, 1, 2, 1, 1, 1, 2.

We can't stop here even though most of the groups are 1, because they all must be either 1 or 0 elements long, so we have to keep going.

After 4 division steps, there will be 16 groups, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1.

Now we can stop because all groups have 1 or 0 elements.

Then we will merge them back, in 4 steps, which is just the reverse of the above.

If you will notice, the number of subdivision steps required was log(2^4), or 4. This is true for any input "n", so the number of subdivision steps will be O(log(k)) where k is n + c such that n + c = 2^d (that is, k is the next largest power-of-two above n). The number of comparisons that are required to merge back to a sorted list is O(n). Thus the run time of the algorithm is O(n * log(k)) where k is some n + c, but we can ignore the "c" in this case as it is a constant, and thus our run time is just O(n log n).

R4.5 – Suppose we are given two n-element sorted sequences A and B that should not be viewed as sets (that is, A and B may contain duplicate entries). Describe an O(n)-time method for computing a sequence representing the set A U B (with no duplicates).

Suppose A is [0, 1, 3, 3, 3, 5, 7] and B is [2, 4, 5, 7, 7]. The resulting list we would like to have is [0, 1, 2, 3, 4, 5, 7].

The proposed algorithm will do the following:

1. create a new list and an iterator for the list.

result = []

last = -1

2. create index variables to iterate over a and b.

a = 0

b = 0

3. until we run out of items in both A and B,

- if last is not -1:

-- if result[last] is the same as A[a] then advance a, if result[last] is the same as b advance b, GOTO 3.

- if A[a] is less than B[b], insert A[a] in result and advance a and increment last.

- if B[b] is less than A[a], insert B[b] in result and advance b and increment last.

Here is an example python program that does the specified function. Note some deviations from the pseudocode were taken to simplify the program (eg. result[-1] indexes the 1st item from the END of the list, so there is no need to maintain the "last" variable.)

```
def union(A, B):
    result = []
    a = 0
    b = 0
    while a < len(A) and b < len(B):
        if result:
            if a < len(A) and result[-1] == A[a]:
                a += 1
                continue
            if b < len(B) and result[-1] == B[b]:
                b += 1
                continue

        if a >= len(A):
            #A is out of items, just insert B.
            result.append(B[b])
        elif b >= len(B):
            result.append(A[a])

        elif A[a] < B[b]:
            #A[a] is lower.
            result.append(A[a])

        else:
            result.append(B[b])
    return result


A = [0, 1, 3, 3, 3, 5, 7]
B = [2, 4, 5, 7, 7]
print union(A, B)
```

It generates the correct result for all the sample inputs I tried.

The runtime is O(n) because it only traverses the lists A and B once each. It has a small comparison cost for each iteration but this is a small constant value and doesn't increase with 'n', so the runtime is cn + d but in O notation this is still linear growth so our runtime is O(n).

The reason the algorithm is able to be run in linear time is because we are guaranteed the input items are in sorted order, so a lot of assumptions can be made (i.e. the last element of "result" must be the highest value in "result" so we do not need to search through "result" to find duplicate entries, we can just check the end.)

R4.12 – Suppose that algorithm inPlaceQuickSort (Algorithm 4.17) is executed on a sequence with duplicate elements. Show that, in this case, the algorithm correctly sorts the input sequence, but the result of the divide step may differ from the high-level description given in seciton 4.3 and may result in inefficiencies. In particular, what happens in the partition step when there are elements equal to the pivot? Is the sequence E (storing the elements equal to the pivot) actually computed? Does the algorithm recur on the subsequences L and R, or on some other subsequences? What is the running time of the algorithm if all the elements of the input sequence are equal?

The sequence "E" is not computed at all. If there are elements equal to the pivot in the partition step, they will remain in the 'L' side and be skipped over. The algorithm recurs on subsequences L and G, which are the items on either side of "l" and "r" after they meet ("l" and "r" are scanned in from the outside edges and items are swapped if List[l] > pivot and List[r] < pivot.) If all the elements of the input sequence are equal, the algorithm will choose pivot at the end, traverse the whole list with "l" and "r" (but doing no swaps), then recurse on (n-1)/2 elements for L and (n-1)/2 elements for R. Therefore it will exhibit the worst-case runtime of O(n^2) but it will be a lower O(n^2) than if swaps were required because there will be no memory copying overhead.

R4.14 – Which, if any, of the algorithms bubble-sort, heap-sort, merge-sort and quick-sort are stable?

merge-sort is stable if you want it to be. If you are sorting in ascending order, you must take values from the first list if they are <= values in the second list. If you only take values from the left list if they are < , and if they're >= you take values from the right list, then it will not be stable. That is,

when merging [1, 2, 3, 4] and [1, 5, 6, 7] , in the original list the left group's 1 came first, so if we merge in the left items first when we have an equal value, we will keep our stability. If we merge in the right group's 1 then we will break our stability.

bubble-sort must be stable, because it works by looping over the array and propagating values up or down the array depending on equality checks by swapping adjacent values that are not in sorted order. The algorithm only terminates when there are no more items to swap. Obviously if bubble sort swapped equal elements (which would make it unstable) then the algorithm would never terminate on an input with duplicates, it would just keep traversing the list and swapping the equal values.

heap-sort is not stable. The way it constructs the heap, there is a heirarchy such that children of the heap are <= to their parents, so depending on the way terms are added to the heap (which changes as the heap changes size), in some cases the equal nodes won't stay in order on heap rebuilds.

quick-sort is not stable. For example, if the input list were [1, 2, 5, 5, 3] and it chose the first 5, and it was set to move all <= values to the left of the pivot, 5 would be moved to the left hand side (out of order). However if the second 5 were chosen, it would preserve the input order. So it is not stable because it depends how the pivot is selected.

C4.22 – Let A and B be two sequences of 'n' integers each. Given an integer x, describe an O(nlogn)-time algorithm for determining if there is an integer 'a' in A and an integer 'b' in B such that x = a + b.

We can sort B with O(nlog(n)) time complexity (say with merge-sort).

Once B is sorted, we iterate over every element in A.

For each element in A, we calculate b = x – a. We then check B to see if the item 'b' exists by using a binary search. Binary search runs in log(n) time.

Since we have "n" elements in A, we will run binary search "n" times. Therefore our time complexity for the second step (iterating over elements in A, checking B to see if x = a + b) will be nlogn as well.

And nlog(n) + nlog(n) = 2nlogn . For a time complexity analysis we can ignore the constant term 2.

Therefore the runtime of this algorithm is O(nlog(n)).