CS325 Assignment 1

Brock Smedley

January 14, 2018

1

Assume we have a merge sort function called $mergesort(a)$ which takes an array a and returns the sorted version of a.

First we sort S using mergesort (which is O(nlogn)). The algorithm then adds the first and last elements of the array. If their sum equals x, then we return true.

If the sum is less than x, then we remove the first element (we need to add a marginally larger number).

If the sum is greater than x, then we remove the last element and add back the first element if it had been previously removed (we may need the previous first element which we previously eliminated to make a valid sum with the new end number).

We recursively call a helper function with the updated S (either with the first element removed; or the last element removed and the first element re-added). This way, we're guaranteed to hone in on the best candidates for making the sum.

I believe the worst case running time for this algorithm is $O(3/2n)$ where X is a constant less than 3/2. The only way one could incur this many operations is for the algorithm to make n comparisons, half of which result in the first element being re-appended while the last element is deleted, effectively keeping S the same size (unless the first element hasn't been previously deleted). Because S is sorted, the algorithm is only able to do this once for every other element (worst-case) because if we have to re-add the first element, we can be guaranteed that our sum was too high. If we only ever get sums that are too low, we'll finish the operation in n iterations. If we only ever get sums that are too high, we'll do the same. But if we go back and forth between low and high at every iteration, then every other iteration leaves the size the same as the last iteration. As a result, in this case the size of S decreases by 1 every other iteration. So in this special case, we add n/2 operations to our base count of n iterations, which we know has to happen to cover all possible pairs with the first element.
Regardless of how much computing time this algorithm takes, it is still linear, and less than nlogn, which is the running time of a merge sort. Therefore, we simply take O(nlogn) to be our final running time, which we can call Θ(nlogn) because we know that our sub-algorithm is linear and the runtime depends most on the merge sort, the running time of which is easy to predict. Our bounds c1 and c2 would be defined by the minimum and maximum runtime of our sub-algorithm: $\Omega(1)$*(logn) and O(3n/2)*(logn).

The following code implements the algorithm in Python:

```
# python 2.7 code
def hasSum(S,x):
    S = mergesort(S)
    return checkSum(S,x,-1,-1)


def checkSum(S,x,h):
    first = S[0]
    last = S[-1]
    _sum = first + last

    # base case 1: out of elements
    if (len(S) == 1):
        return False

    # base case 2: found a pair
    if (_sum == x):
        return True
    elif (_sum < x):
        h = S.pop(0)
    elif (_sum > x):
        if (h != -1):
            S.insert(0,h)
        h = -1
    return checkSum(S,x,h)
```

2

(a) f is O(g).

Because the functions are polynomials of different degrees, we can guarantee that the one with the highest degree will always grow faster (the highest degree belonging to g(n)). g is an upper limit on f so f is O(g).

(b) f is Ω(g).

After n=1 (where we will set n_0), f(n) grows faster than g(n). Therefore g is a lower bound on f and f is Ω(g).

(c) f is O(g).

f(n) never exceeds g(n); g is an upper limit on f, so f must be O(g).

(d) f is Ω(g).

Because these are exponential functions with the same power, we know that the function with the higher base will always grow faster. Therefore, g is a lower limit on f and f is Ω(g).

(e) f is O(g).

Because these are exponential functions with the same base, we know that the function with the highest power will grow faster, which is g. g is an upper limit on f, so f is O(g).

(f) f is O(g).

We know that the factorial function eventually grows faster than any exponential function. Therefore we know that g is an upper bound on f, so f is O(g).

3

(a) $f_1(n) = \text{O(g)}$
$f_2(n) = \text{O(g)}$

$$\left.\begin{array}{l} f_1(n) \leq c_1 * g(n) \\ f_2(n) \leq c_2 * g(n) \end{array}\right\} \text{ By definition of big-O}$$

$f_1(n) + f_2(n) \leq c_1 * g(n) + c_2 * g(n)$
$f_1(n) + f_2(n) \leq (c_1 + c_2) * g(n)$

Because the value of constants can be ignored, we can combine the constants to get the following:

$f_1(n) + f_2(n) \leq (c) * g(n) = O(g(n))$ (by definition of big-O) $\square$

(b) Counter-example: $f(n) = log(n)$
$g_1(n) = log(n^2)$
$g_2(n) = n^2$
For $g_1(n)$ to be $\Theta(g_2(n))$ it must be $\Omega(g_2(n))$ and $O(g_2(n))$. g_1 cannot be $\Omega(g(n))$ so therefore it cannot be $\Theta(g(n))$.

4