Assignment 2

Benjamin Sorenson

October 10, 2015

1 2.3-7

We could use `MERGE-SORT` to sort S in $\Theta(n \lg n)$ time, and then for each $s \in S$ use `BINARY-SEARCH` to search for $s - x$. `BINARY-SEARCH` is defined below, and on each call it divides the problem by two and makes a recursive call until, in the worst case, the length of the array is ≤ 2—so it's worst-case runtime is $O(\lg n)$. The function `FIND-EXACT-SUM` implements the the described algorithm (I haven't quite got the hang of LaTeXformatting so the algorithms appear on the next page).

2 6.3-3

The number of leaves in a tree of size n is $\lceil \frac{n}{2} \rceil$. Without loss of generality, we will assume that $n = 2^k$. The number of nodes one level up from the bottom ($h = 1$), can be found by subtracting the number of leaves from the total number of elements and applying the same formula. This sets up the recurrence relation

$$\left\lceil \frac{x_n}{2} \right\rceil = \left\lceil \frac{x_{n-1} - \lceil \frac{x_{n-1}}{2} \rceil}{2} \right\rceil$$

with the initial condition $x_0 = 2^k$, this becomes

$$x_n = x_{n-1} - \frac{x_{n-1}}{2}$$

Assume that for some x_r,

$$x_r \leq \frac{2^k}{2^r}$$

Then

$$x_{r+1} = x_r - \frac{x_r}{2} \leq \frac{2^k}{2^r} - \frac{\frac{2^k}{2^r}}{2}$$

$$x_{r+1} \leq \frac{2^k}{2^{(r+1)}}$$

3 6.5-8

The idea begin `HEAP-DELETE` is that we can use two $\Theta(\lg n)$ procedures, `HEAP-INCREASE-KEY` and `HEAP-EXTRACT-MAX` to delete an element of the heap A. First, increase the key to $\max + 1$, and then use `HEAP-EXTRACT-MAX` to remove it.

```
function BINARY-SEARCH(A, l, r, k)
    mid ← ⌊(l+r)/2⌋
    if A[mid] = k then
        return mid
    end if
    if mid = l then
        if A[mid] > k then
            return mid
        else
            return mid + 1
        end if
    end if
    if A[mid] < k then
        return BINARY-SEARCH(A, l, mid, k)
    end if
    if A[mid] > k then
        return BINARY-SEARCH(A, mid + 1, l, k)
    end if
end function
function FIND-EXACT-SUM(S, x)
    MERGE-SORT(S)
    for i ← 1 to length[S] do
        d ← S[i] − x
        j ←BINARY-SEARCH(S, 1, length[S], d)
        if S[j] = x then
            return True
        end if
    end for
    return False
end function
```

```
procedure HEAP-DELETE(A, i)
    max ← HEAP-MAX(A)
    HEAP-INCREASE-KEY(A, i, max + 1)
    HEAP-EXTRACT-MAX(A)
end procedure
```

4 6.5-9

I couldn't quite figure this one out. I know that we could perform `HEAP-EXTRACT-MIN` in $\Theta(\lg n)$ time on each of the k sub-lists, but this wouldn't guarantee that we're getting the lowest values from all k sub-lists. The best I could come up with was to find the min of the mins from each of the k sub-lists, extract it from its sub-list, and place it in the merged array, but that would be $\Theta(nk \lg k)$

5 7.2-1

Upper Bound Show $T(n) \leq cn^2$

> **Assume** $T(n-1) \leq c(n-1)^2$
>
> **Then** $T(n) \leq cn^2 - 2cn + c + c_1 n \leq cn^2$ for $c_1 < 2c$ and $n \geq 1$

Lower Bound show $T(n) \geq cn^2$

> **Assume** $T(n-1) \geq c(n-1)^2$
>
> **Then** $T(n) \geq cn^2 - 2cn + c + c_1 n \geq cn^2$ for $c_1 \geq 2c$ and $n \geq 1$

Base Case $n = 1$, $T(1) = c = c(1^2)$

6 7.3-1

We analyze the expected running time of a randomized algorithm because as n gets large the the probability of the worst-case running time approaches 0 so the expected running gives a more realistic asymptotic behavior.