Robert Barnes

20407 - 2013b

Maman 12

Question 1

A) $T(n) = 8T(n/2) + n + n^3$ We have that $f(n) = n^3 + n$ and that $n^{\log_2 8} = n^3$. Thus $f(n) == \Theta(n^{\log_2 8})$ and by case 2 of the Master Method we have that $T(n) = \Theta(n^3 \lg n)$.

B) $T(n) = kT(n/2) + (k-2)n^3 \; \forall k \geq 2 \in \mathbb{N}$

Case 1: $k = 2$ Here we have $f(n) = 0$ thus $n^{log_2 2} = n^1$ dominates $f(n)$ polynomially $\forall \epsilon : 1 \geq \epsilon > 0$ and by Case 1 of the Master Method $T(n) = \Theta(n)$.

Case 2: $2 < k < 8$ Here we have that $n^{log_2 3} \leq n^{log_2 k} \leq n^{log_2 7} \Rightarrow n^{1.58} \leq n^{log_2 k} \leq n^{2.80}$. We have then that $f(n) = \Omega(n^{log_2 k + 0.2})$ and by case 3 of the Master Method $T(n) = \Theta(n^3)$ if the regularity condition is fulfilled. $f(n) = (k-2)n^3$ implies we are looking for $c < 1$ such that $\frac{k(k-2)}{8} n^3 \leq c(k-2)n^3 \Rightarrow \frac{k}{8} \leq c$. Since $3 \leq k \leq 7$ we have that the regularity condition is fulfilled $\forall c : \frac{7}{8} \leq c < 1$.

Case 3: $k = 8$ Here we have that $n^{log_2 8} = n^3$ and thus $f(n) = \Theta(n^{log_2 8})$ and by case 2 of the Master Method $T(n) = \Theta(n^3 \lg n)$.

Case 4: $k > 8$ Here we have that $n^{log_2 k} \geq n^{log_2 9} \approx n^{3.16}$ Thus $f(n) = O(n^{log_2 k - 0.16})$ and by case 1 of the Master Method $T(n) = \Theta(n^{log_2 k})$.

C) $T(n) = 2T(n/4) + \sqrt{n} \lg n$ We have that $f(n) = n^{1/2} \lg n$ and $n^{log_4 2} = n^{1/2}$. Thus $f(n) = \Theta(n^{1/2} \lg^1 n)$ and by case 2 extended of the Master Method we have that $T(n) = \Theta(n^{1/2} \lg^2 n)$

D) $T(n) = 4T(\sqrt{n}) + \lg^2 n \lg \lg n$ First we use substitution to arrive at a new formula. We observe that:

$$\begin{aligned} \lg n = m \Rightarrow n = 2^m \\ \Rightarrow T(2^m) = 4T(\sqrt{2^m}) + lg^2(2^m) \lg \lg(2^m) \\ = 4T(2^{m/2}) + (m \lg 2)^2 \lg(m \lg 2) \\ = 4T(2^{m/2}) + m^2 \lg^2 2[\lg m + \lg \lg 2] \\ = 4T(2^{m/2}) + \Theta(m^2 \lg m) \end{aligned}$$

We define $S(m) \equiv T(2^m) \Rightarrow S(m/2) = T(2^{m/2})$ and thus

$$S(m) = 4S(m/2) + m^2 \lg m$$

Now we have that $m^{log_2 4} = m^2$ and thus $f(m) = \Theta(m^2 \lg^1 m)$. Thus by case 2 extended of the Master Method $S(m) = \Theta(m^2 \lg^2 m)$. Since $S(m) = T(2^m) = T(n)$ we have by substitution that $m = \lg n$ and thus $T(n) = \Theta(\lg^2 n \cdot [\lg^2 \lg n])$.

Question 2

I've included an implementation in `C` of the algorithm in a separate file, `missingInt.c`. Basically, it takes `ceil( lg n+1 )` bits to hold the values from 0 to n. This number of bits can hold a total of $2^{ceil(\lg n+1)}$ values. If we look at all these values, in any order, there will always be an even number of 0's and 1's in the ith bit of all the numbers. So if we remove any number from the list, either the number of 0's or 1's in the ith bit will be odd, and this tells us whether we're missing a 1 or 0 in that bit. It also tells us that we only have to look at values which having the missing bit in the ith position on the next pass. So we use the auxillary array B for two purposes. The first is to hold any missing values between n and $2^{ceil(\lg n+1)}$. The second purpose is to hold the indexes of the values we want to continue looking at on the next pass. This won't interfere with the first purpose, since any missing values will only be in the second half of the array, and any indexes after the first pass will be in the first half of the array since the number of values of interest is approximately halved on each pass. So we need to initialize B which takes $\Theta(n)$ time, and each time through the loop we go over the values of interest twice, once to count the 1's, and once to record the indexes of which values to look at on the next pass. This gives us the recurrence $T(n) = T(n/2) + 2n$, which by case 3 of the Master Method gives us that $T(n) = \Theta(n)$ since $n/2 \leq cn \; \forall c : 1/2 \leq c < 1$ and the regularity condition is fulfilled. Thus, the algorithm overall is $\Theta(n)$.

Question 3

Give an $O(n \lg k)$ -time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists.

Assume we have k sorted lists, with a total of n elements and that the lists are numbered 1 to k.

1. Extract the smallest element from each list, keeping track of which list each element originated in. Build a k element min-heap from these elements. Allocate an empty array `A` of size n.
2. Remove the smallest element from the min-heap and place it in the next free position in `A`. If the list from which the element was originally removed has any more elements, remove the next smallest element and insert it into the heap.
3. Loop until all the lists and and the heap are empty, resulting in `A` containing a sorted list.

Since the lists are sorted, extracting elements takes $O(1)$ time. It takes $O(k)$ time to build the min-heap. Then for each element we remove from the heap to place in `A`, it takes maximum $\lg k$ time to extract it, and then inserting the next

value into the heap takes maximum $\lg k$ time, resulting in $2 \lg k$. If we implement an extract-replace function which extracts the min and replaces it with the new value before calling heapify, then these two steps can be done together at a max cost of $\lg k$. In any event, we extract n elements from the heap, taking $n \lg k$ time and we have that $O(k) + O(n \lg k) = O(n \lg k)$. When $k = 2$ you can special case it and use the `merge` routine from `mergesort` to get $O(n)$ time.

Question 4

I used the psuedocode conventions from the third edition of Kormen, which is what I'm learning from. I hope that's ok. Also, I'm not completely sure why this works, it's just the pattern that emerged when I ran build-heap by hand on a large diff tree.

```
heapify( A, i )
l = Left(i)
r = Right(i)

if l <= A.heap-size and A[l] < A[i]
    largest = l
else largest = i

if r <= A.heap-size and A[r] < A[largest]
    largest = r

if largest != i
    A[largest] = -A[largest]

    if i != 1
        A[i] = A[i] - A[largest]
    else
        A[i] = A[i] + A[largest]

    if largest = l
        otherChild = r
    else
        otherChild = l

    if otherChild <= A.heap-size
        A[otherChild] = A[otherChild] + A[largest]

    if Left(largest) <= A.heap-size
        A[Left(largest)] = A[Left(largest)] - A[largest]

    if Right(largest) <= A.heap-size
        A[Right(largest)] = A[Right(largest)] - A[largest]
```

```
    heapify(A, largest)
```

Since only a constant number of operations are added to the function, the above changes do not change the time complexity of the function, which remains $O(\lg n)$.

```
build-heap(A)
A.heap-size = A.length
for i = floor( A.length / 2 ) downto 1

    // diffify the children
    l = Left(i)
    if l <= A.length
        A[l] = A[i] - A[l]
    r = Right(i)
    if r <= A.length
        A[r] = A[i] - A[r]

    // diffify the parent
    if i != 1
        A[i] = A[Parent(i)] - A[i]
    heapify(A, i)
```

Again, we only add a constant number of operations to the function, leaving it's asymptotic complexity unchanged at $O(n)$, since `heapify`'s complexity is also the same.

Now for `heapsort`. Here we need a helper function which climbs the tree to determine the "real" value of the parent.

```
getParentVal(A, i)
if i = 1
    error "root has no parent"
diff = 0
i = Parent(i)
while ( i != 1 )
    diff = diff + A[i]
    i = Parent(i)
return A[1] - diff
```

This function runs in $O(\lg n)$ time, since in the worst case it must climb the entire tree to it's root.

```
heapsort(A)
build-heap(A)
```

```
for i = A.length downto 2
    A[i] = getParentVal(i) - A[i]

    // adjust the diff values of the root children so they're relative to
    // A[i]
    l = Left(1)
    if l <= A.heap-size
        A[l] = A[i] - ( A[1] - A[l] )
    r = Right(1)
    if r <= A.heap-size
        A[r] = A[i] - ( A[1] - A[r] )

    swap( A[i], A[1] )
    A.heap-size = A.heap-size - 1
    heapify(A, 1)
```

The original `heapsort` operatates in $O(n \lg n)$ time, as does this one, since `getParentVal` operates in $O(\lg n)$ time. Thus we have $O(n) + 2 \cdot O(n \lg n)$ which is still $O(n \lg n)$.

```
heap-extract-max(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = getParentVal(i) - A[A.heap-size]

// adjust the diff values of the root children so they're relative to
// A[A.heap-size]
l = Left(1)
if l <= A.heap-size
    A[l] = A[1] - ( max - A[l] )
r = Right(1)
if r <= A.heap-size
    A[r] = A[1] - ( max - A[r] )

A.heap-size = A.heap-size - 1
heapify(A, 1)
return max
```

We only add a single call to `getParentVal`, which is $O(\lg n)$, and a constant number of new operations, so we get $2 \cdot \lg n = O(\lg n)$ and the big O running time is not changed.

```
heap-insert(A, key)
A.heap-size = A.heap-size + 1
```

```
i = A.heap-size
A[i] = getParentVal(i) - key
i = Parent(i)
while ( i != 0 )
    heapify(A, i)
    i = Parent(i)
```

This function also stays $O(\lg n)$ since each call to heapify will terminate after the recursive call because we're simply percolating up a large value which will never cause any swap except with it's parent, so heapify will only perform a constant number of operations.