Algorithms and Recurrence Relations: Examples

Chuck Cusack

1. **What is the worst-case running time of Mergesort?**
The algorithm for `Mergesort` is below. Let $T(n)$ be the worst-case running time of `Mergesort` on an array of size n=right-left. Recall that `Merge` takes two sorted arrays and merges them into one sorted array in time $\Theta(n)$, where n is the number of elements in both arrays. Since the two recursive calls to `Mergsort` are on arrays of half the size, they each require time $T(n/2)$ in the worst-case. The other operations take constant time, as indicated below.

Analysis of Mergesort

Algorithm	Time required
`Mergesort(int[] A,int left,int right) {`	$T(n)$
`    if (left < right) {`	C_1
`        int mid = (left + right)/2;`	C_2
`        Mergesort(A, left, mid);`	$T(n/2)$
`        Mergesort(A, mid + 1, right);`	$T(n/2)$
`        Merge(A, left, mid, right);`	$\Theta(n)$
`    }`	
`}`	

Given this, we can see that

$$\begin{aligned} T(n) &= C_1 + C_2 + T(n/2) + T(n/2) + \Theta(n) \\ &= 2T(n/2) + \Theta(n). \end{aligned}$$

For simplicity, we will write this as $T(n) = 2T(n/2) + cn$ for some constant c.

Now we have a formula for $T(n)$, but it is not straight-forward to use. For instance, if $n = 1000$, what is $T(n)$? We need a formula for $T(n)$ that is not recursive. Finding such a formula is called *solving* a recurrence relation, and we call the formula the *closed-form* for $T(n)$.

It turns out that $T(n) = \Theta(n \log n)$. Although this is not an exact formula, a tight-bound is often all we are interested in when analyzing algorithms. We will prove that $T(n) = O(n \log n)$, and leave the Ω-bound to the reader.

By definition, $T(n) = O(n \log n)$ if and only if there exists constants k and n_0 such that $T(n) \leq kn \log n$ for all $n \geq n_0$. We will use induction to prove this.

For the base case, notice that $T(2) = a$ for some constant a, and $a \leq k2 \log 2 = 2k$ as long as we pick $k \geq a/2$. Now, assume that $T(n/2) \leq k(n/2) \log(n/2)$. Then

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &\leq 2(k(n/2) \log(n/2) + cn \\ &= kn \log(n/2) + cn \\ &= kn \log n - kn \log 2 + cn \\ &= kn \log n + (c - k)n \\ &\leq kn \log n \quad \text{if } k \geq c \end{aligned}$$

As long as we pick $k = \max\{a/2, c\}$, we have $T(n) \leq kn \log n$, so $T(n) = O(n \log n)$ as desired.

2. **How many moves does it take to solve the *Towers of Hanoi* problem?**
The usual (and best) algorithm to solve the *Towers of Hanoi* is as follows:

   - Move the top $n-1$ disk to from peg 1 to peg 2.
   - Move the last disk from peg 1 to peg 3.
   - Move the top $n-1$ disks from peg 2 to peg 3.

   The only question is how to move the top $n-1$ disks. The answer is simple: using the same algorithm (with the peg numbers switched). Don't worry if you don't see why this works. Our main concern here is analyzing the algorithm.

   Let $H(n)$ be the time required to solve the *Towers of Hanoi* problem with n disks. Assuming moving a single disk takes 1 operations, the above algorithm requires

$$H(n) = H(n-1) + 1 + H(n-1) = 2H(n-1) + 1$$

   operations. As with the first example, we want a closed form for $H(n)$. Notice that

$$\begin{aligned} H(1) &= 1 \\ H(2) &= 2H(1) + 1 = 3 \\ H(3) &= 2H(2) + 1 = 7 \\ H(4) &= 2H(3) + 1 = 15 \end{aligned}$$

   From these examples, it appears that $H(n) = 2^n - 1$. We will prove this by induction. Clearly, we already have proven the base case. Assume $H(n-1) = 2^{n-1} - 1$. Then

$$\begin{aligned} H(n) &= 2H(n-1) + 1 \\ &= 2(2^{n-1} - 1) + 1 \\ &= 2^n - 2 + 1 \\ &= 2^n - 1. \end{aligned}$$

   Thus, by the principle of induction, $H(n) = 2^n - 1$ for all $n \geq 1$.

3. **Give a recurrence relation for the following algorithm:**

```
int Nothing(int n) {
     if(n>5) {
        return Nothing(n-1)+Nothing(n-1)+Nothing(n-5)+Nothing(sqrt(n));
     }
     else {
        return n;
     }
}
```

   It is not hard to see that if $T(n)$ is the running time for `Nothing(n)`, then

$$T(n) = 2T(n-1) + T(n-5) + T(\sqrt{n}) + O(1).$$

4. **Exercise:** The `BinarySearch` algorithm is given below. Give a recurrence relation and a tight bound for the worst-case running time of `BinarySearch`.

```
boolean BinarySearch(int[] A,int First,int Last,int Value) {
     if(Last>=First) {
         int mid=(Last+First)/2;
         if(Value==A[mid])       return true;
         else if(Value<A[mid])   return BinarySearch(A,First,mid-1,Value);
         else                    return BinarySearch(A,mid+1,Last,Value);
     }
     else return false;
}
```