Data Structures and Algorithms
Running time, Divide and Conquer
January 25, 2018

Consider the problem of computing 2^n for any non-negative integer n. Below are four similar looking algorithms to solve this problem.

```
powerof2(n)
  if n = 0
    return 1
  else
    return 2 * powerof2(n-1)

powerof2(n)
  if n = 0
    return 1
  else
    return powerof2(n-1)+ powerof2(n-1)

powerof2(n)
  if n = 0
    return 1
  else
    tmp = powerof2(n-1)
    return tmp + tmp

powerof2(n)
  if n = 0
    return 1
  else
    tmp = powerof2(floor(n/2))
    if (n is even) then
      return tmp * tmp
    else
      return 2 * tmp * tmp
```

The recurrence for th first and the third method is $T(n) = T(n-1) + O(1)$. The recurrence for the second method is $T(n) = 2T(n-1) + O(1)$, and the recurrence for the last method is $T(n) = T(n/2) + c$ (assuming that n is a power of 2). In all cases the base case is $T(0) = 1$.

We will solve these recurrences . The recurrence for the first and the third method can be solved as follows.

$$
\begin{aligned}
T(n) &= T(n-1) + c \\
&= T(n-2) + 2c \\
&= T(n-3) + 3c \\
&\ldots \\
&\ldots \\
&= T(n-k) + kc
\end{aligned}
$$

The recursion bottoms out when $n - k = 0$, i.e., $k = n$. Thus, we get

$$
\begin{aligned}
T(n) &= T(0) + kc \\
&= 1 + nc \\
&= \Theta(n)
\end{aligned}
$$

The recurrence for the second method can be solved as follows.

$$
\begin{aligned}
T(n) &= 2T(n-1) + c \\
&= 2^2T(n-2) + (2^0 + 2^1)c \\
&= 2^3T(n-3) + (2^0 + 2^1 + 2^2)c \\
&\ldots \\
&\ldots \\
&= 2^kT(n-k) + c\sum_{i=0}^{k-1} 2^i
\end{aligned}
$$

The recursion bottoms out when $n - k = 0$, i.e., $k = n$. Thus, we get

$$
\begin{aligned}
T(n) &= 2^nT(0) + c\sum_{i=0}^{n-1} 2^i \\
&= 2^n + c(2^n - 1) \\
&= \Theta(2^n)
\end{aligned}
$$

The recurrence for the fourth method can be solved as follows.

$$
\begin{aligned}
T(n) &= T(n/2) + c \\
&= T(n/2^2) + 2c \\
&= T(n/2^3) + 3c \\
&\ldots \\
&\ldots \\
&= T(n/2^k) + kc
\end{aligned}
$$

The recursion bottoms out when $n/2^k < 1$, i.e., when $k > \lg n$. Thus, we get

$$\begin{aligned} T(n) &= T(0) + c(\lg n + 1) \\ &= 1 + \Theta(\lg n) \\ &= \Theta(\lg n) \end{aligned}$$

Linear Search and Binary Search

The input is an array A of elements in any arbitrary order and a key k and the objective is to output true, if k is in A, false, otherwise. Below is a recursive function to solve this problem.

```
LinearSearch (A[lo .. hi], k)
  if lo > hi then
    return False
  else
    return (A[hi] == k) or LinearSearch(A[lo..hi-1], k)
```

The recurrence relation to express the running time of `LinearSearch` is given by $T(n) = T(n-1) + c$, with the base case being $T(0) = 1$. We have already solved this recurrence and it yields a running time of $T(n) = \Theta(n)$.

If the input array A is already sorted, we can do significantly better using *binary search* as follows.

```
BinarySearch (A[lo .. hi], k)
  if lo > hi then
     return False
  else
     mid = floor(lo+hi/2)
     if A[mid] = k then
        return True
     else if A[mid] < k then
        return BinarySearch(A[mid+1 .. hi], k)
     else
        return BinarySearch(A[lo .. mid-1], k)
```

The running time of this method is given the recurrence $T(n) = T(n/2) + c$, with the base case being $T(0) = 1$. As we have seen before, this recurrence yields a running time of $T(n) = \Theta(\log n)$.

Sorting

Below is a recursive version of insertion sort that we studied a couple of lectures ago.

```
InsertionSort(A[lo..hi])
  if lo = hi then
```

```
    return A
  else
    A' = InsertionSort(A[lo..hi-1])
    Insert(A', A[hi])  // insert element A[hi] into the sorted array A'
```

Note that the `Insert` function takes $\Theta(n)$ time for an input array of size n. Thus the running time of Insertion sort is given by the following recurrence.

$$T(n) = \begin{cases} 1, & n = 1 \\ T(n-1) + n, & n \geq 2 \end{cases}$$

It is easy to see that this recurrence yields a running time of $T(n) = \Theta(n^2)$.

To motivate the idea behind the next sorting algorithm (Merge Sort), let's rewrite `InsertionSort` function as follows.

```
InsertionSort(A[lo..hi])
  if lo = hi then
    return A
  else
    //Merge combines two sorted arrays into one sorted array
    Merge(InsertionSort(A[lo..hi-1]), InsertionSort(A[hi..hi]))
```

The function `Merge` is as follows.

```
Merge(A[1..p], B[1..q])
  if p = 0 then
    return B
  if q = 0 then
    return A
  if A[1] <= B[1] then
    return prepend(A[1], Merge(A[2..p], B[1..q]))
  else
    return prepend(B[1], Merge(A[1..p], B[2..q]))
```

Note that the running time of `Merge` is $O(p+q)$. The second recursive call to `InsertionSort` takes $O(1)$ time and hence the running time of `InsertionSort` still is $\Theta(n^2)$.

Observe that in `InsertionSort` the input array A is partitioned into two arrays, one of size $|A| - 1$ and another of size 1. In Merge Sort, we partition the input array of size n in two equal halves (assuming n is a power of 2). Below is the function.

```
MergeSort(A[1..n])
if n = 1 then
  return A
else
  return Merge(MergeSort(A[1..n/2], MergeSort(A[n/2+1..n]))
```

The running time of `MergeSort` is given by the following recurrence.

$$T(n) = \begin{cases} 1, & n = 1 \\ 2T(n/2) + cn, & n \geq 2 \end{cases}$$

Below are some facts on logarithms that you may find useful.

i. $\log_a b = \frac{1}{\log_b a}$

ii $\log_a b = \frac{\log_c b}{\log_c a}$

iii $a^{\log_a b} = b$

iv $b^{\log_a x} = x^{\log_a b}$

We can also solve recurrences by guessing the overall form of the solution and then figure out the constants as we proceed with the proof. Below are some examples.

Example. Consider the following recurrence for the `MergeSort` algorithm.

$$T(n) = \begin{cases} 1, & n = 1 \\ 2T(n/2) + n, & n \geq 2 \end{cases}$$

Prove that $T(n) = O(n \lg n)$.

Solution. We will first prove the claim by expanding the recurrence as follows.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2^2 T(n/2^2) + 2n \\ &= 2^3 T(n/2^3) + 3n \\ &\dots \\ &\dots \\ &= 2^k T(n/2^k) + kn \end{aligned}$$

The recursion bottoms out when $n/2^k = 1$, i.e., $k = \lg n$. Thus, we get

$$\begin{aligned} T(n) &= 2^{\lg n} T(1) + n \lg n \\ &= \Theta(n \log n) \end{aligned}$$

We will now prove that $T(n) = O(n \lg n)$ by using strong induction on n. We will show that for some constant c, whose value we will determine later, $T(n) \leq cn \lg n$, for all $n \geq 2$.
Induction Hypothesis: Assume that the claim is true when $n = j$, for all j such that $2 \leq j \leq k$. In other words, $T(j) \leq cj \lg j$.
Base Case: $n = 2$. The left hand side is given by $T(2) = 2T(1) + 2 = 4$ and the right hand

side is $2c$. Thus the claim is true for the base case when $c \geq 2$.
Induction Step: We want to show that for $k \geq 2$, $T(k+1) \leq c(k+1)\lg(k+1)$. We have

$$\begin{aligned}
T(k+1) &= 2T\left(\frac{k+1}{2}\right) + (k+1) \\
&\leq 2c\left(\left(\frac{k+1}{2}\right)\lg\left(\frac{k+1}{2}\right)\right) + (k+1) \\
&= c(k+1)(\lg(k+1) - \lg 2) + (k+1) \\
&= c(k+1)\lg(k+1) - (c-1)(k+1) \\
&\leq c(k+1)\lg(k+1) \qquad \text{(since } c \geq 2)
\end{aligned}$$

Example. Consider the following recurrence that you may want to try to solve on your own before reading the solution.

$$T(n) = \begin{cases} 1, & n = 1 \\ 2T(n/2) + n^2, & n \geq 2 \end{cases}$$

Prove that $T(n) = \Theta(n^2)$.

Solution. Clearly, $T(n) = \Omega(n^2)$ (because of the n^2 term in the recurrence). To prove that $T(n) = O(n^2)$, we will show using strong induction that for some constant c, whose value we will determine later, $T(n) \leq cn^2$, for all $n \geq 1$.
Induction Hypothesis: Assume that the claim is true when $n = j$, for all j such that $1 \leq j \leq k$. In other words, $T(j) \leq cj^2$.
Base Case: $n = 1$. The claim is clearly true as the left hand side and the right hand side, both equal 1.
Induction Step: We want to show that $T(k+1) \leq c(k+1)^2$. We have

$$\begin{aligned}
T(k+1) &= 2T\left(\frac{k+1}{2}\right) + (k+1)^2 \\
&\leq 2c\left(\frac{k+1}{2}\right)^2 + (k+1)^2 \\
&= \left(\frac{c}{2} + 1\right)(k+1)^2
\end{aligned}$$

We want the right hand side to be at most cn^2. This means that we want $c/2 + 1 \leq c$, which holds when $c \geq 2$. Thus we have shown that $T(n) \leq 2n^2$, for all $n \geq 1$, and hence $T(n) = O(n^2)$.

Quicksort

In quicksort, we first decide on the pivot. This could be the element at any location in the input array. The function `Partition` is then invoked. `Partition` accomplishes the following: it places the pivot in the location that it should be in the output and places all elements that are at most the pivot to the left of the pivot and all elements greater than the pivot to its right. Then we recurse on both parts. The pseudocode for Quicksort is as follows.

```
QSort(A[lo..hi])
  if hi <= lo then
    return
  else
    pivotIndex = floor((lo+hi)/2)  (this could have been any location)
    loc = Partition(A, lo, hi, pIndex)
    QSort(A[lo..loc-1])
    QSort(A[loc+1..hi])
```

One possible implementation of the function `Partition` is as follows.

```
Partition(A, lo, hi, pIndex)
  pivot = A[pIndex]
  swap(A, pivotIndex, hi)
  left = lo
  right = hi-1
  while left <= right do
     if (A[left] <= pivot) then
        left = left + 1
     else
        swap(A, left, right)
        right = right - 1
  swap(A, left, hi)
  return left
```

The worst case running time of the algorithm is given by

$$T(n) = \begin{cases} 1, & n = 1 \\ T(n-1) + cn, & n \geq 2 \end{cases}$$

Hence the worst case running time of `QSort` is $\Theta(n^2)$.

An instance where the quicksort algorithm in which the pivot is always the first element in the input array performs poorly is an array in descending order of its elements.