COMP 3711 Fall 2021

Design & Analysis of Algorithms

Written By: Ljm

Topic 2 **Divide & Conquer**

1 Intro: Binary Search

The main idea of Divide & Conquer is to solve a problem(such as of size n) by breaking it into one or more smaller(size less than n) problems. We use binary search example to illustrate that.

Problem: given an **sorted** array of length n, how to find the position of element x; if x does not exist in the array, output nil.

Since the array is already sorted, it has a good property that: **for each item a_i, those who are larger than a_i must be on its right side, while smaller than a_i must be on its left side.** Hence we come up with an idea that we check the middle item mid first, then we will be able to know which direction to go: left or right, depending on the comparison of mid and x(the item we're looking for). If we go left, then the right half will be directly abandoned. Then we continue this process, check middle item each time, and abandon half items each time.

Algorithm 1: BinarySearch($a[]$, $left$, $right$, x)

```
Data: a[]: the array given, x: the item to find
if left = right then
    if a[left] = x then
        return left
    else
        return nil
    end
else
    mid = ⌊(left + right)/2⌋
    if x ≤ a[mid] then
        BinarySearch(a[], left, mid, x)
    else
        BinarySearch(a[], mid + 1, right, x)
    end
end
```

First call: BinarySearch($a[]$, 1, n, x).

This algorithm is quite efficient, since each time we eliminate half of the array, with one additional comparison, until there is only one item left, when we will end the process.

Then let's analyse its time complexity. Let $T(n)$ be the number of comparisons needed for n elements, then we will have

$$T(n) = T(n/2) + 1,\ T(1) = 1$$

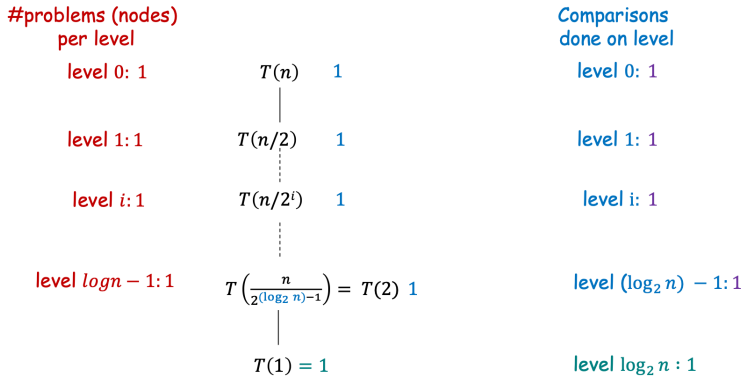
.

Solve this **recurrence**:

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= [T(n/4) + 1] + 1 \\ &= T(n/4) + 2 \\ &= \cdots \\ &= T(n/2^i) + i \end{aligned}$$

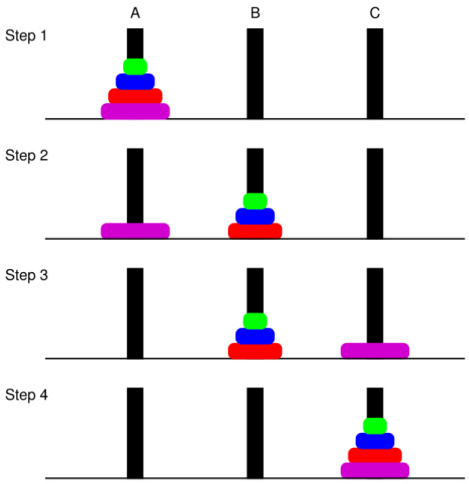
This process ends when reaching $T(1)$, i.e., $i = \log_2 n$, thus, $T(n) = T(1) + \log_2 n = \log_2 n + 1$.

We can also visualize this recurrence with recursion tree: (image from `https://cs61bl.org/su21/labs/lab10/`)



In each recursion step(level), we use 1 comparison(compare mid and x), then call recursion on a half of the original array. From the image above, we can easily notice there are total $1 + 1 + \cdots + 1 = 1 + \log_2 n$ comparisons.

2 Example: Towers of Hanoi



In this example, we want to design an algorithm to move all n discs from peg A(start) to peg C(end), with the constraints: (1) move one disc at a time, and (2) cannot put larger disc on a smaller one. We are given another peg B(helper) where we can temporary storage our discs.

We still use the idea of **Divide & Conquer**, consider how we can turn a problem of n discs into a problem of $n-1$? One possible solution is that, we can call recursion on upper $n-1$ discs, i.e., move upper $n-1$ discs to peg B(helper peg), then move the remaining (the biggest) disc to peg C(end peg), and finally move the $n-1$ discs from peg B(helper) to peg C(end). The following pseudocode shows this idea.

Algorithm 2: MoveTower(n, *start*, *helper*, *end*)

Input: n: num of discs

```
1 if n = 1 then
2 |   move the only disc from start peg to end peg
3 |   return
4 else
  |   // move first n − 1 from start peg to helper peg
  |   // so this time ‘‘helper’’ peg will be the old end peg
5 |   MoveTower(n − 1, start, end, helper)
6 |   move the only disc from start peg to end peg
  |   // finally move first n − 1 from helper peg to end peg
  |   // this time ‘‘helper’’ peg will be the old start peg
7 |   MoveTower(n − 1, helper, start, end)
8 end
```

Now we would like to analyze the time complexity of this algorithm, in other words, how many **steps** are needed. Let $T(n)$ be the num of steps for n discs, each time, we first move $n-1$ disks from *start* to *helper*, costs $T(n-1)$ steps; then we move the biggest disk to *end* peg, costs only 1 step; finally we move $n-1$ disks from *helper* to *end*, again costs $T(n-1)$ steps. To sum up:

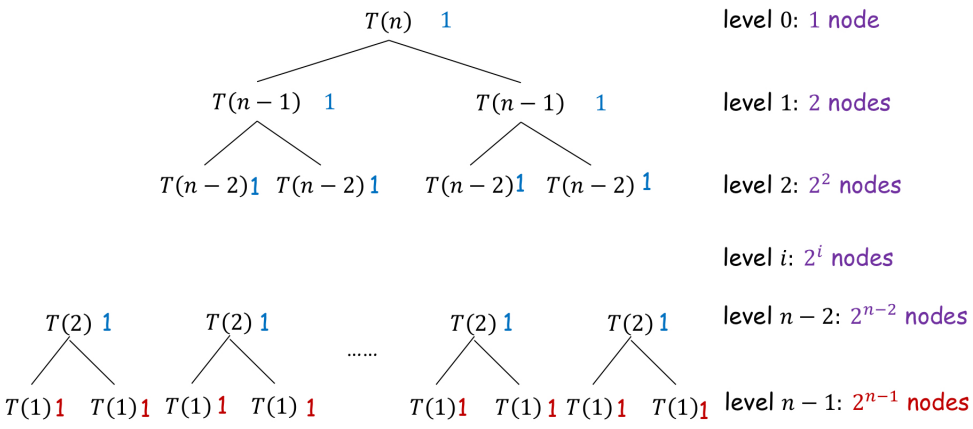
$$T(n) = 2T(n-1) + 1$$

when $n > 1$, and $T(1) = 1$.

Now we solve the recurrence by the **expansion method**:

$$\begin{aligned}
T(n) &= 2T(n-1)+1 \\
&= 2[2T(n-2)+1]+1 \\
&= 4T(n-2)+3 \\
&= 4[2T(n-3)+1]+3 \\
&= 8T(n-3)+7 \\
&= \cdots \\
&= 2^i T(n-i)+(2^i-1) \\
&= 2^{n-1}T(1)+(2^{n-1}-1) \\
&= 2^n-1
\end{aligned}$$

Or, with the recursion tree method:



There are, altogether, $1+2+2^2+2^3+\cdots+2^{n-2}+2^{n-1}=2^n-1$ nodes, and we are doing one work(step) each node, then the time complexity is again, 2^n-1.

3 Merge Sort

Now we again back to sorting, and we would like to introduce a new algorithm or sorting: Merge Sort. This is a typical example of divide & conquer, and its process is like: (1) we first divide array into two halves, (2) then we recursively sort each half, (which means we continuously divide it into halves, and then halves...) (3) finally **merge** two halves to get a whole.

The **merge** operation may confuse you most. Here it means combine two **sorted lists** into a whole sorted list. For example, given two sorted lists: $A=[2,5,7]$ and $B=[3,4,6,10,12]$, then after **merge** operation, we will get

$result = [2, 3, 4, 5, 6, 7, 10, 12]$. Since these two lists are sorted, we can do this process in $O(n)$ time, where n is the length of result list.(how many numbers in total) The basic idea is: we compare the first item of A and B, put the smaller one, say, $A[1]$, in the first position of result list, then we move on to the next item of A, but compare it still with the **first** item of B(since the first item of B has not yet inserted into result list), and again put the smaller one into result list, then continue move on. An example may help you understand the process:

(1) Compare first items: $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $2 < 3$, so $result = [2]$;

(2) then compare 2nd in A and 1st in B, $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $3 < 5$, so $result = [2, 3]$;

(3) continue the process, similarly, $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $4 < 5$, so $result = [2, 3, 4]$;

(4) $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $5 < 6$, so $result = [2, 3, 4, 5]$;

(5) $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $6 < 7$, so $result = [2, 3, 4, 5, 6]$;

(6) $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $7 < 10$, so $result = [2, 3, 4, 5, 6, 7]$;

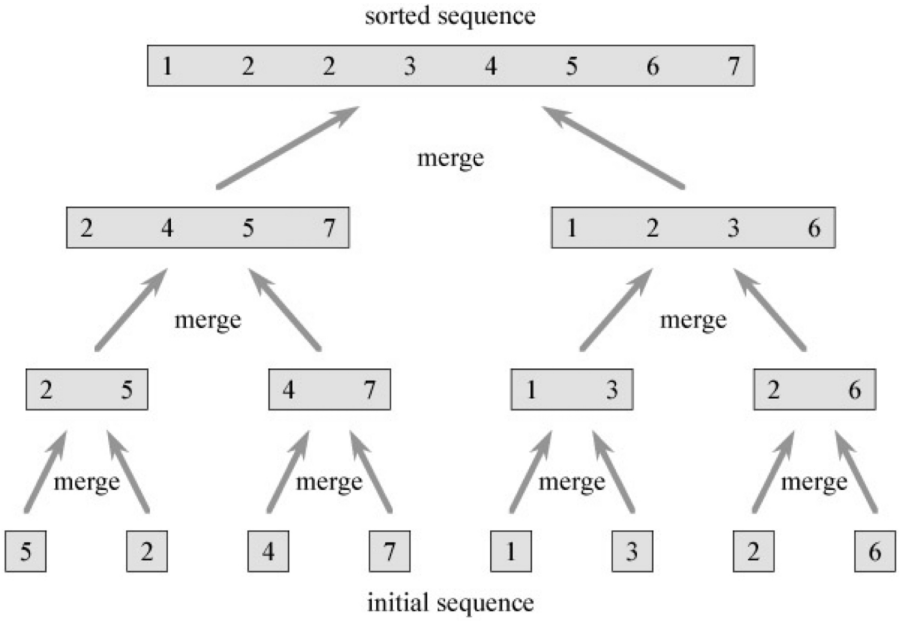
(7) Now, all items in A have already been inserted into result list so that no items can be compared with items in B. Then we simply add remaining items in B to result list, this will, obviously, ensure a sorted result list.(you may think of why) Hence $result = [2, 3, 4, 5, 6, 7, 10, 12]$

The pseudocode below shows the process: (below, append ∞ at the end of two lists can free us from considering the situation that one list is empty, like (7) above. Though different implementation, the idea is entirely the same)

Algorithm 3: Merge(A, $left$, mid, $right$)

```
// merge two sorted list:  A[left ··· mid] and A[mid + 1 ··· right]
L ← A[left ··· mid], R ← A[mid + 1 ··· right]
append ∞ at the end of L and R      // see explanation above
i ← 1, j ← 1     // two pointers point at items in L and R
for k ← left to right do
    // always choose the smaller one to insert, and move on
    if L[i] ≤ R[j] then
        A[k] ← L[i]
        i ← i + 1
    else
        A[k] ← R[j]
        j ← j + 1
    end
end
```

After learning how **Merge** works, you now, hopefully, are able to understand how Merge Sort works, with the image below:



We break down array recursively, until one element left, and then merge from bottom to up. The complete pseudocode for Merge Sort is given below:

Algorithm 4: MergeSort(A, $left$, $right$)

```
1 if left = right then
2 |   return
3 end
4 mid ← ⌊(left + right)/2⌋
  // recursively divide array into two halves
5 MergeSort(A, left, mid)
6 MergeSort(A, mid + 1, right)
  // then merge from bottom to up
7 Merge(A, left, mid, right)
```

Firstly call **MergeSort(A, 1, n)** to sort array A.

As usual, we are interested in the running time of Merge Sort algorithm. Let $T(n)$ be the running time on an array of size n, it's not hard to find $T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$, when $n > 1$ and $T(1) = O(1)$.

Here we are actually able to simplify the equation. Firstly we can replace $\leq$ with $=$, since we are insterested in big-O upper bound of $T(n)$; and with the same reason, we can replace $O(n)$ with n, $O(1)$ with 1; finally, we can assume n is a power of 2 for the sake of simplicity but doesn't change the result at all, as $T(n) \leq T(n') \leq T(2n) = O(T(n))$ where n' is the smallest power of 2 such that $n' \geq n$.

Now we want to solve: $T(n) = 2T(n/2) + n$ for $n > 1$, and $T(1) = 1$.

$$\begin{aligned} T(n) &= 2\left(\frac{n}{2}\right) + n \\ &= 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n = 2^2 \cdot T\left(\frac{n}{2^2}\right) + 2n \\ &= 2^2 \cdot \left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n = 2^3 \cdot T\left(\frac{n}{2^3}\right) + 3n \\ &= \cdots \\ &= 2^k \cdot T\left(\frac{n}{2^k}\right) + kn \end{aligned}$$

We know the process ends with $\frac{n}{2^k} = 1$ i.e. $k = \log_2 n$, thus

$$\begin{aligned} T(n) &= 2^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) + n \cdot \log_2 n \\ &= n \log_2 n + n \end{aligned}$$

In summary, merge sort runs in $O(n \log n)$ time. It is also worth pointing out that merge sort **always** runs in $O(n \log n)$ time, which means best case is the same as worst case, as you may think of it, the complexity of merge sort *does not depend on inputs*, it always break array down and then merge up.

4 Inversion Numbers