Divide And Conquer

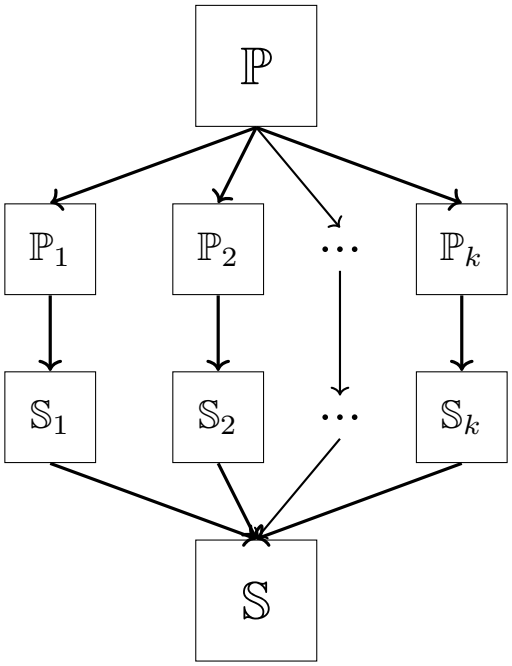
Mohammed Rizin
Unemployed

March 23, 2025

Abstract

Suppose we got a bigger problem, call it P of size n. Sometimes bigger problem are really daunting and we end up doing nothing. Instead, why don't we split the problem into multiple pieces. and solve them and combine the result. Its like we split the work into several pieces and assign to workers independently. So, each worker doesn't feel so hard about solving the problem. So we employ *DivideAndConquer* strategy.

1 Introduction



As discussed earlier, we are dividing a task into several pieces and solve them individually and independently.

1.1 Conditions for Divide and Conquer

1. All the sub-problems should be the same task as the main problem.
2. There should be an known method to combine the solutions at least

Since all the sub-problems are the same task as the parent problem, This is a recursive algorithm.

1.2 Applications of Divide and Conquer

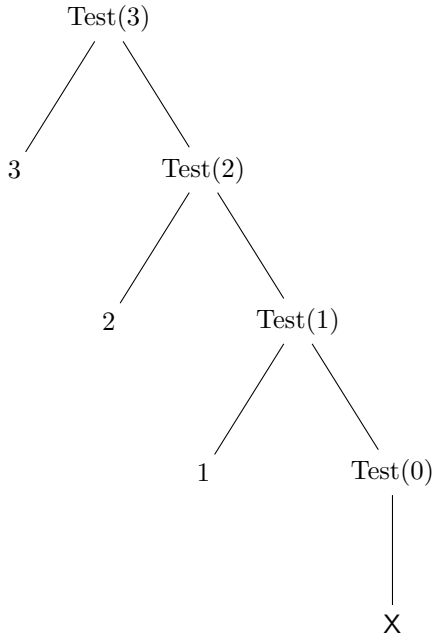
1. Binary search
2. Finding Minimum and Maximum
3. Merge Sort
4. Quick Sort
5. Stressen's Matrix Multiplication

2 Recurrence Relation

To analyze the time complexity of divide and conquer algorithms, we often use recurrence relations. For example:

```
void Test(int n){
    if (n > 0){
        printf("%d", n);
        Test(n-1);
    }
}

Test(3)
```



Since the only statement in the function is printf, which take $O(1)$ time, at each recursive step $O(1)$ is done. If $Test(n)$ is executed, the number of recursive calls are $n + 1$, the number of printf calls are n. So the time complexity of this algorithm is $On(n)$

This won't be possible in every algorithm to open the function and count them and make assumption about n cases. Mathematically, That is not sufficient. Thus, Recurrence Relation Come into play.

Lets take the same code we used before:

Algorithm 1 Simple Printing with recursion

```
procedure TEST(int n)                          ▷ Assume it takes T(n)
    if n > 0 then
        print(n)                                                ▷ O(1)
        Test(n − 1)                                         ▷ T(n − 1)
    end if
end procedure
```

So,

$$T(n) = \begin{cases} T(n-1)+1 & n > 0, \\ 1 & n = 0 \end{cases}$$

$\because \quad T(n) = T(n-1) + 1,$

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

Substituting(3) in (1),

$T(n) = [T(n-2)+1] + 1 = T(n-2) + 2,$

$T(n) = [T(n-3)+1] + 2 = T(n-3) + 3,$

$T(n) = \quad \vdots \quad + \quad \vdots,$

$T(n) = T(n-k) + k,$

$\because \quad T(0) = 1,$

if (k = n),

$T(n) = T(0) + n,$

$T(n) = 1 + n.$

This algorithm is $O(n)$

Lets look Another example

Algorithm 2 Recursion with simple For Loop

```
procedure TEST(int n)                          ▷ Assume it takes T(n)
    if n > 0 then
        for i ← 0 to n − 1 do                  ▷ n − 1
            print(n)                           ▷ Everything inside for will be T(n) = n
        end for
        Test(n − 1)                            ▷ T(n − 1)
    end if
end procedure
```

So,

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

We need to take Asymptotic Notation

$$T(n) \simeq T(n-1) + n$$

$$T(n) = \begin{cases} T(n-1)+n & n > 0, \\ 1 & n = 0 \end{cases}$$

$\because \quad T(n) = T(n-1) + n,$

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

Substituting(3) in (1),

$T(n) = [T(n-2) + n - 1] + n = T(n-2) + (n-1) + n,$

$T(n) = [T(n-3) + n - 2] + (n-1) + n = T(n-3) + (n-2) + (n-1) + n,$

$T(n) = \quad \vdots \quad + \quad \vdots,$

$T(n) = T(n-k) + (n-(k-1)) + \cdots + (n-1) + n,$

$\because \quad T(0) = 1,$

if (k = n),

$$T(n) = T(0) + \frac{n(n+1)}{2} = 1 + \frac{n(n+1)}{2},$$

$$T(n) \approx \frac{n(n+1)}{2} \approx \frac{n^2+n}{2}$$

This algorithm is $O(n^2)$

Lets look Another example

Algorithm 3 Recursion with for loop **Step** $= 2 \cdot i$

```
procedure TEST(int n)                          ▷ Assume it takes T(n)
    if n > 0 then
        for i ← 0 to n − 1 step 2 · i do       ▷ This is log(n) + 1 Not useful.
            print(n)                           ▷ This takes log n
        end for
        Test(n − 1)                            ▷ T(n − 1)
    end if
end procedure
```

Although I know we excluded many of the time complexities simply because we wont using it in asymptotic equation

After taking Asymptotic Notation

$$T(n) = T(n-1) + \log n \tag{1}$$

$$T(n) = \begin{cases} T(n-1) + \log n & n > 0, \\ 1 & n = 0 \end{cases} \tag{2}$$

$$\because \quad T(n) = T(n-1) + \log n$$

$$T(n-1) = T(n-2) + \log(n-1) \tag{3}$$

Substituting eqn (3) in (2),

$$\begin{aligned} T(n) &= [T(n-2) + \log(n-1)] + \log(n) \\ &= T(n-2) + \log(n-1) + \log n \\ &= T(n-2) + \log(n^2 - n), \end{aligned}$$

$$\begin{aligned} T(n) &= [T(n-3) + n - 2] + (n-1) + n \\ &= T(n-3) + \log(n-2) + \log(n-1) + \log n \\ &= T(n-3) + \log((n-2) \cdot (n-1) \cdot (n)) \end{aligned}$$

$$T(n) = T(n-k) + \log\left(\prod_{i=0}^{k-1}(n-i)\right) \tag{4}$$

$$T(n) = T(n-k) + \log\left(\frac{n!}{n-(k-2)!}\right)$$

$$\because \quad T(0) = 1,$$

if (k = n),

$$T(n) = T(0) + \log\left(\frac{n!}{2}\right) = 1 + \cdot \left(\frac{n!}{2}\right)$$

$$T(n) \approx \log n! \approx \log n!$$

Upper bound of $\log n!$ is $O(n \cdot \log n)$

This algorithm is $O(n \cdot \log n)$

$$OR$$

From eqn (4) :

$$T(n) = T(n-k) + \log\left(\prod_{i=0}^{k-1}(n-i)\right)$$

By Binomial Theorem,

$$T(n) = T(n-k) + \log(n^k + \cdots)$$

$$T(n) \approx T(n-k) + k \cdot \log n$$

$$\because \quad T(0) = 1,$$

if (k = n),

$$T(n) = T(0) + n \cdot logn$$

This algorithm is $O(n \cdot \log n)$

By observing above examples, we can observe that for any decreasing recurrence relation function i.e. $(T(n) =$

$T(\underbrace{(n-1)}_{Decreasing} + anything)$ is $O(n \cdot anything)$

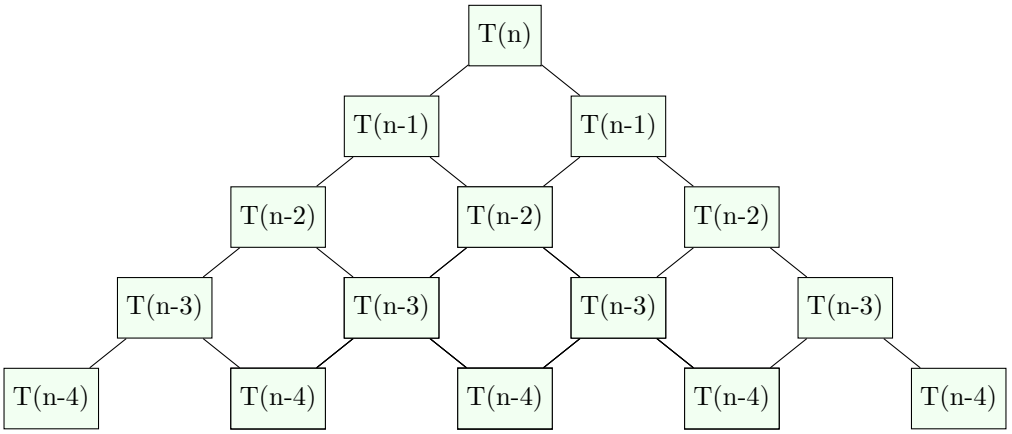
$T(n) = T(n-2) + 1 = 1 + \frac{n}{2} \longrightarrow O(n)$

$T(n) = T(n-100) + n = 1 + \frac{n(n+1)}{200} \longrightarrow O(n^2)$

But what if $T(n) = 2 \cdot T(n-1) + 1$:

Algorithm 4 Multiple Recursion

```
procedure TEST(int n)                          ▷ Assume it takes T(n)
    if n > 0 then
        print(n)                                                ▷ O(1)
        Test(n − 1)                                         ▷ T(n − 1)
        Test(n − 1)                                         ▷ T(n − 1)
    end if
end procedure
```



The algorithm time depends on the number of level. At a certain level k the time taken is 2^k. Then $T(n) = 2^0 + 2^1 + 2^2 + \cdots + 2^k = 2^{k+1} - 1$

Assume $n = k$, Then it is $2^{n+1} - 1$. That is $O(2^n)$.

$$T(n) = \begin{cases} 2 \cdot T(n-1) + 1 & n > 0, \\ 1 & n = 0 \end{cases}$$

$\because \quad T(n) = T(n-1) + 1,$

$T(n-1) = 2 \cdot T(n-2) + 1,$

Substituting(3) in (1),

$T(n) = 2 \cdot [2 \cdot T(n-2) + 1] + 1 = 4 \cdot T(n-2) + 3,$

$T(n) = 4 \cdot [2 \cdot T(n-3) + 1] + 3 = 8 \cdot T(n-3) + 7,$

$T(n) = \quad \vdots \quad + \quad \vdots,$

$T(n) = 2^k \cdot T(n-k) + 2^k - 1,$

$\because \quad T(0) = 1,$

if (k = n),

$T(n) = 2^n \cdot T(0) + 2^n - 1,$

$T(n) = 2^{n+1} - 1.$

This algorithm is $O(2^n)$

3 Master Theorem for Decreasing

Theorem 1. *If* $T(n) = a \cdot T(n-1) + f(n)$ *where* $a > 0$, $b > 0$, *and* $f(n) = O(n^k)$ *where* $k \geq 0$, *then:*

$$\begin{cases} O(n^k) \text{ or } O(f(n)) & \text{if } 0 < a < 1, \\ O(n^{k+1}) \text{ or } O(n \cdot f(n)) & \text{if } a = 1, \\ O(n^k \cdot a^{\frac{n}{b}}) \text{ or } O(a^{\frac{n}{b}} \cdot f(n)) & \text{if } a > 1. \end{cases}$$

Example 1.1. $T(n) = T(n-1) + 1 \to O(n)$

Example 1.2. $T(n) = T(n-1) + n \to O(n^2)$

Example 1.3. $T(n) = T(n-1) + \log n \to O(n \cdot \log n)$

Example 1.4. $T(n) = 2 \cdot T(n-1) + 1 \to O(2^n)$

Example 1.5. $T(n) = 2 \cdot T(n-1) + 1 \to O(3^n)$

Example 1.6. $T(n) = 3 \cdot T(n-1) + n \to O(n \cdot 3^n)$