* Analysis of common loops

→ Example 1 :

n : User Input
c : constant

```
for(int i=0; i<n ; i= i+c)
{
  // some θ(1) work
}
```

loop runs $\theta\left(\left\lceil \frac{n}{c} \right\rceil\right)$

TC : $\theta(n)$

n=10	i=0
c=2	i=2
	i=4
	i=6
	i=8

n=20	i=0
c=6	i=6
	i=12
	i=18

→ Example 2 :

```
for(int i=n; i>0; i=i-c)
{
  // some θ(1) work
}
```

loop runs $\left\lceil \frac{n}{c} \right\rceil$

TC : $\theta(n)$

n=10	i=10
c=2	i=8
	i=6
	i=4
	i=2

n=20	i=20
c=6	i=14
	i=8
	i=2

→ Example 3 :

```
for(int i=1; i<n; i= i*c)
{
  // some θ(1) work
}
```

loop runs $\lceil \log_c n \rceil$

TC : $\theta(\log n)$

n=83	i=1
c=2	i=2
	i=4
	i=8
	i=16
	i=32

n=81	i=1
c=3	i=3
	i=9
	i=27
	.

$c^0, c^1, c^2, \cdots, c^{k-1}$

$c^{k-1} < n$

$(k-1)\log_c c < \log_c n$

$k < \log_c n + 1$

TC : $\theta(\log_c n)$

→ Example 4:

```
for(int i=n; i>1; i=i/c)
{
    //some O(1) work
}
```

n=33	i=33
c=2	i=16
	i=8
	i=4
	i=2

n=81	i=81
c=3	i=27
	i=9
	i=3

$\frac{n}{c^0}, \frac{n}{c^1}, \frac{n}{c^2}, \ldots, \frac{n}{c^{k-1}}$

$\frac{n}{c^{k-1}} > 1$

$c^{k-1} < n$

$k-1 < \log_c n$

$k < \log_c n + 1$

TC: $\boxed{O(\log_c n)}$

→ Example 5:

```
for(int i=2; i<n; i=pow(i,c))
{
    //some O(1) work
}
```

n=33	i=2
c=2	i=4
	i=16

n=514	i=2
c=3	i=8
	i=512

$2, 2^c, (2^c)^c, \ldots, \left(\left(2^c\right)^c\right)^{c\ldots}$

$2^{c^0}, 2^{c^1}, 2^{c^2}, \ldots, 2^{c^{k-1}}$

$2^{c^{k-1}} < n$

$c^{k-1} < \log_2 n$

$k-1 < \log_c \log_2 n$

$k < \log_2 \log_c n + 1$

TC: $O(\log_2 \log_c n)$

* Analysis of Multiple Loops

* Analysis of Multiple Loops

→ Example 1:

Subsequent loops

```
void fun(int n)
{
    for (int i=0; i<n; i++)
    {  //some O(1) work  }          O(n)
                                     +
    for (int i=1; i<n; i=i*2)
    {  //some O(1) work  }          O(log n)
                                     +
    for (int i=1; i<100; i++)
    {  //some O(1) work  }          O(1)

}
```

$= O(n)$

→ Example 2 :

Nested loops

```
void fun(int n)
{
    for(int i=0; i<n; i++)  → O(n)
    {                          *
        for(int j=1; j<n; j=j*2)  → O(logn)
        {                          ⇓
            //some O(1) work       TC: O(nlogn)
        }
    }
}
```

→ Example3 :

Mixed loops

```
void fun(int n)
{
    for(int i=0; i<n; i++)
    {
        for(int j=1; j<n; j=j*2)      } O(nlogn)  *
        {
            //some O(1) work
        }
    }                                              +
    for(int i=0; i<n; i++)
    {                                 } O(n^2)
        for(int j=1; j<n; j++)
        {
            //some O(1) work          ⇓
        }
    }                                 O(n^2)
}
```

→ Example 4 :

Different Inputs

```
void fun(int n, int m)
{
    for(int i=0; i<n; i++)
    {
        for(int j=1; j<n; j=j*2)      } O(nlogn)
        {
            //some O(1) work
        }
    }                                              +
    for(int i=0; i<m; i++)
    {                                 } O(m^2)
        for(int j=1; j<m; j++)
        {
            //some O(1) work          ⇓
        }
    }                                 O(nlogn + m^2)
}
```

* Analysis of Recursion

→ Example1:

```
void fun(int n)
{
    if(n<=0)
      return;
    print("abc");
    fun(n/2);
    fun(n/2);
}
```

$n>0$

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

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

$T(0) = \theta(1)$

→ Example2:

```
void fun(int n)
{
    if(n<=0)
      return;
    for(int i=0; i<n; i++)
        print("abc");
    fun(n/2);
    fun(n/3);
}
```

$n>0$

$T(n) = T(n/2) + T(n/3) + \theta(n)$

$T(0) = \theta(1)$

→ Example3:

```
void fun(int n)
{
    if(n<=1)
      return;
    print("GFG");
    fun(n-1);
}
```

$T(n) = T(n-1) + \theta(1)$

$T(1) = \theta(1)$

* Recursion Tree Method for solving Recurrences

→ We consider the recursion tree and compute total work done

→ We wrote non-recursive part as root of the tree and write the recursive part as children.

→ we keep expanding until we see a pattern

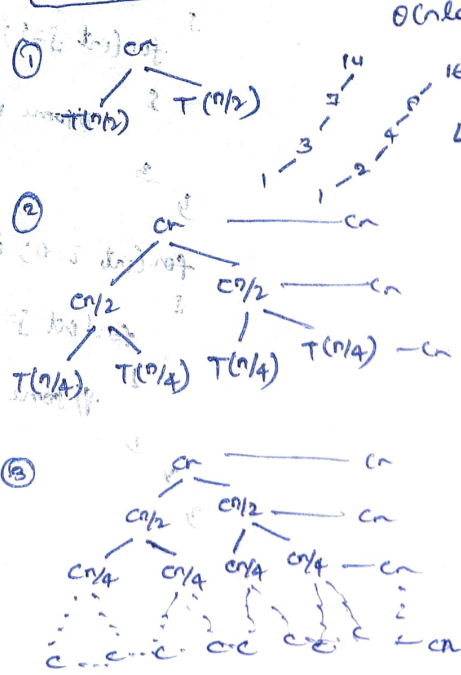
$T(n) = 2T(n/2) + cn$ (Rec + NR)

$T(1) = c$

$cn + cn + cn + \cdots + cn$

$\theta(\log n)$

$\theta(n \log n)$

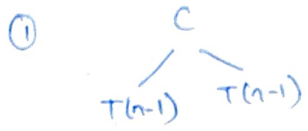


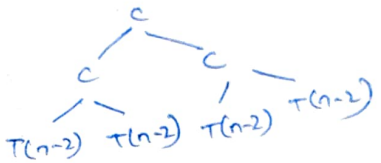
More Example Recurrences

→ Example 1:

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

$T(1) = C$

① 

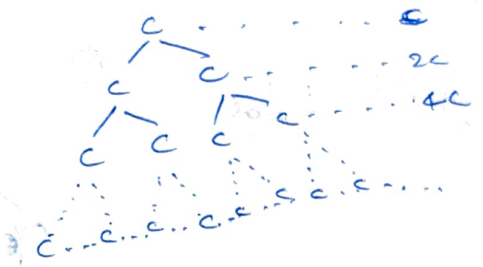
② 

$C + 2C + 4C + \cdots$ (GP)

$\Rightarrow C(1 + 2 + 4 + \cdots)$

$\Rightarrow C \frac{1 \times (2^n - 1)}{1}$

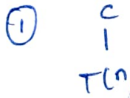
$\theta(2^n)$

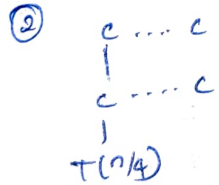
③ 

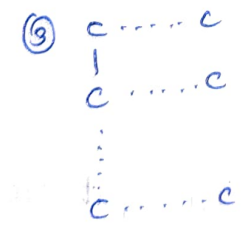
→ Example 2:

$T(n) = T(n/2) + C$

$T(1) = C$

① 

② 

③ 

$C + C + \cdots C$

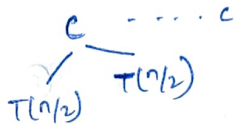
$(\log_2 n) + 1$

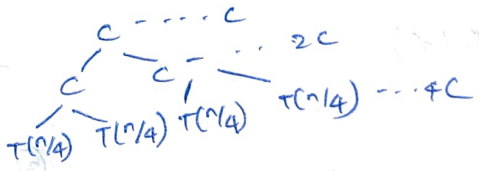
TC: $\theta(\log_2 n)$

→ Example 3:

$T(n) = 2T(n/2) + C$

$T(1) = C$

① 

② 

$\lfloor \log_2 n \rfloor + 1$

$C + 2C + 4C + \cdots$ ($\theta(\log_2 n)$)

$\theta\left(\frac{2^{\log_2 n} - 1}{2 - 1}\right)$

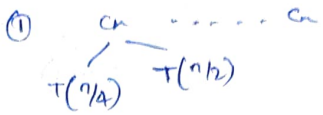
$\theta(n)$

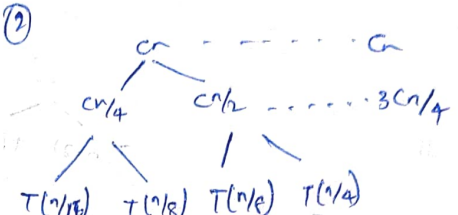
* Upper Bounds Using Recursion Tree Method

→ Example 1:

$T(n) = T(n/4) + T(n/2) + cn$

$T(1) = c$



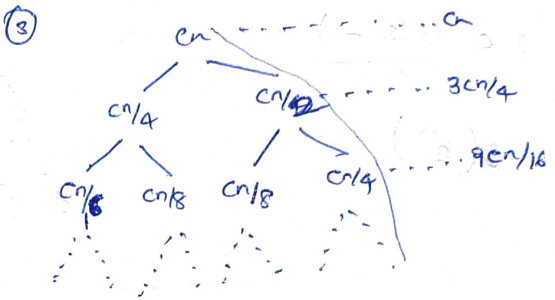


$cn + 3cn/4 + 9cn/16 + \dots$

$\theta(\log n)$

$\theta\left(\frac{cn}{1-3/4}\right)$ → for infinite range

$\theta(n)$

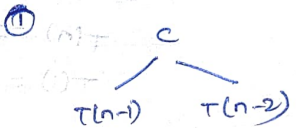


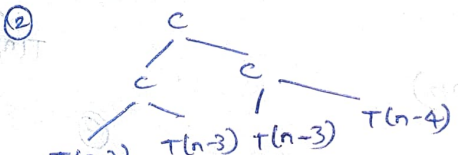
→ Example 2:

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

$T(1) = c$

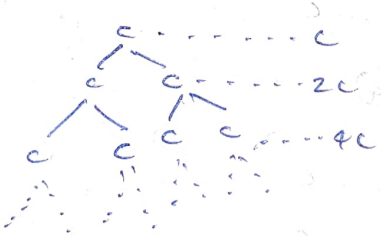
$T(0) = c$





$\underbrace{O(c + 2c + 4c + \dots)}_{n \text{ terms}}$

$O(2^n)$



* Space Complexity

Order of growth of memory (or RAM) space in terms of input size

```
int getSum1(int n)
{
    return n*(n+1)/2;
}
```

SC: O(1) or O(1)

```
int getSum2(int n)
{
    int sum=0;
    for(int i=1; i<=n; i++)
        sum=sum+i;
    return sum;
}
```

SC: O(1) or O(1)

```
int arrSum(int arr[], int n)
{
    int sum = 0;
    for(int i=0; i<n; i++)
        sum = sum + arr[i];
    return sum;
}
```

SC: $\theta(n)$

* Auxilary Space

order of growth of extra space or temporary space in terms of input size

(or)

The space created for memory other than i/p & o/p

→ Example 1:

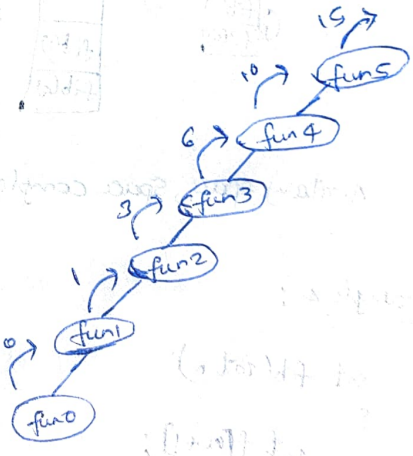
```
int arrSum(int arr[], int n)
{
    int sum=0;
    for(int i=0; i<n; i++)
        sum = sum + arr[i];
    return sum;
}
```

Auxilary Space : $\theta(1)$

space complexity : $\theta(n)$

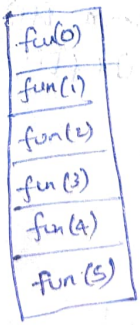
→ Example 2 :

```
int fun(int n)
{
    if(n<=0)
        return 0;
    return n + fun(n-1);
}
```



for recursion we require function call stack

→



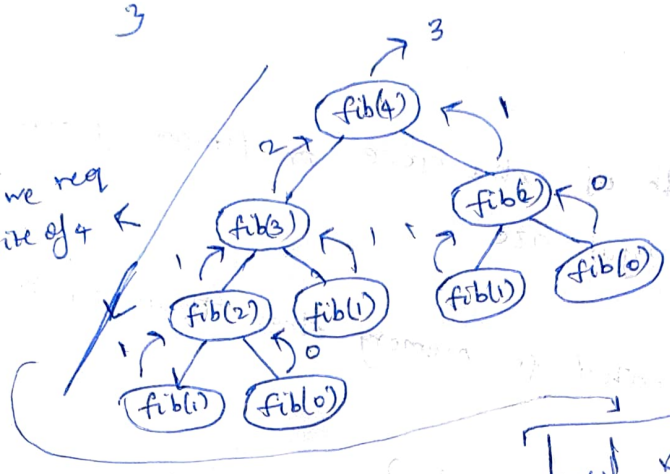
n+1

$\theta(n)$

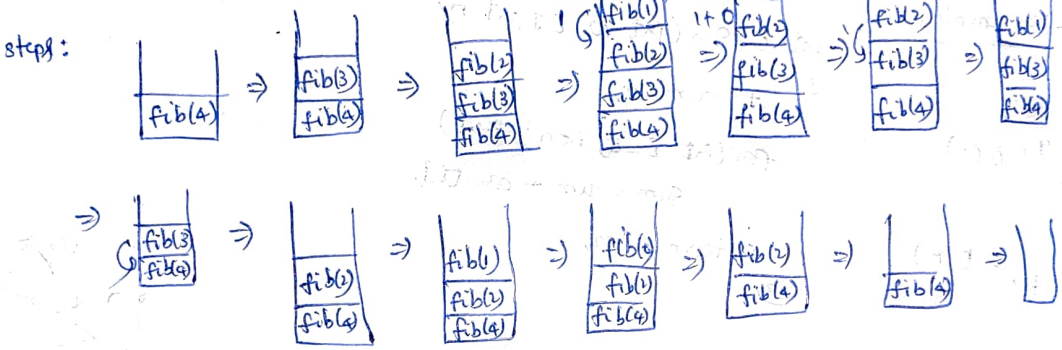
→ Example 3: the space required for recursion is the maximum height of tree

```
int fib(int n)
{
    if (n==0 || n==1)
        return n;
    return fib(n-1) + fib(n-2)
}
```

0, 1, 1, 2, 3, 5, ...
n=0 1 2 3 4



steps:

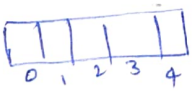


Auxilary space, Space complexity = $\theta(n)$

→ Example 4:

```
int fib(int n)
{
    int f[n+1];
    f[0] = 0;
    f[1] = 1;
    for(int i=2; i<=n; i++)
        f[i] = f[i-1] + f[i-2]
    return f[n];
}
```

n = 4



Auxilary space, Space Complexity = $\theta(n)$

→ Example 5:

n = 4
a = 0, b = 1
i = 2: c = 1, a = 1, b = 1
i = 3: c = 2, a = 1, b = 2
i = 4: c = 3, a = 2, b = 3

```
int fib(int n)
{
    if (n==0 || n==1)
        return n;
    int a=0; b=1;
    for(int i=2; i<=n; i++)
    {
        c = a+b;
        a = b;
        b = c;
    }
    return c;
}
```

Auxilary space, Space Complexity = $\theta(1)$