Algorithms Analysis

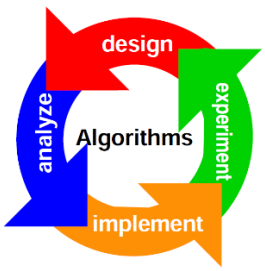
What is the algorithm?

Set of finite steps to solve a certain problem.

Why do we need to use algorithms to solve the problems?

To save resources, time and money.

What is the process of the algorithm?



What is the main strategy to enhance the algorithm performance?

- **Time Analysis**
  - Instructions take time.
  - How fast does the algorithm perform?
  - What affects its runtime?
- **Space Analysis**
  - Data structures take space.
  - What kind of data structures can be used?
  - How does choice of data structure affect the runtime?

What's the (Time & Space) Trade-offs?

Sometimes we increase space to reduce time, **ex.: Fibonacci** 0, 1, 1, 2, 3, 5, 8, 13, 21, …

F(0)=0, F(1)=1	F(n)=F(n-1) + F(n-2) for n>1
F(n) If (n≤1): return (n) Else: return F(n-1)+F(n-2)	F(n) F[0] = 0; F[1] = 1 For(I = 2 to n) do: F[I] = F[i-1] + F[i-2] return F[n]
Time= $O(2^N)$ Space= O(1)	Time= O(N) Space= O(N)

Runing Time VS. Order:

Running Time T(N): it's counting number steps that the algorithm takes as a function in the input size.

Ex.1. Sequence of Operations:

Code	Cost
count = count + 1 sum = sum + count	C1 C2
Total Cost	**C1 + C2**

Ex.2. Simple Loop:

Code	Cost	Time

i = 1;
sum = 0;
while (i<= n)
{
  i= i+ 1;
  sum = sum + i;
}
	C1
	C2
	C3
	
	C4
	C5
		1
		1
		n+1
		
		n
		n

Total Cost	**C1 + C2 + (n+1)*C3 + n*C4 + n*C5**	**T(N) = n**

Ex.3.:

Code	Cost	Time

max = A[1]
for I = 1 to n
  If (A[I] > Max)
  max = A[I]
return Max
	C1
	C2
	C3
	C4
	C5
		1
		1 + n
		n
		n
		1

Total Cost	**C1 + (1+n)*C2 + n*C3 + n*C4 + C5**	**T(N) = n**

Order/Complexity O(…): Is the max dominant factor in the running time without any constants.

Examples:

- $T(N) = c1 + c2 + (n+1)\times c3 + n\times c4 + n\times c5$ => $O(N)$
- $T(N) = 2xN^2 + 10^6xN + 1000$ => $O(N^2)$
- $T(N) = (20xN)^7$ => $O(N^7)$
- $T(N) = 10^8$ => $O(1)$
- $T(N) = \log(N^{100}) + 10^{10}$ => $O(\log(N))$

Example (Constants & small factors do not effect on the large inputs):

- **Constance not important:**

N	$T(N) = 10^6 \log_2(N)$	$T(N) = 20N$	Comparison
8 10^6 10^9	$3*10^6$ $20*10^6$ $30*10^6$	$3*10^6$ $20*10^6$ $20*10^9$	O(N) is better. Equals. $O(\log_2(N))$ is better.

- **Large factors not important:**

N	$T(N) = N^2 + N*\log(N) + 10N$	$O(N^2)$	T(N) / O(N)
10 1000	210 $10^6 + 1000x3 + 10x1000 = 10^6 + 13000$	100 10^6	2.1 1.013

What we should take in consideration Runing Time or Order:

For a small input we take the Running time and for large input we take the Order, in Algorithms we are interested with the large inputs so, order is the most important.

Algorithm Cases:

- **Worst case:** A situation (i.e., input) that leads the algorithm to behave at its worst time.
- **Best case:** A situation (i.e., input) that leads the algorithm to behave at its best time.
- **Expected (Average) case:** The expected time of the algorithm when it runs over random (unexpected) input.

Example: (Linear search for an item in the array):

- **Best case:** item is found at first place => O(1)
- **Worst case:** item is not found => O(N)
- **Expected (Average) case:** item is found at any place:
  - To find it at any location (i):
    - Equally like probability = 1/N
    - Number of comparisons = i
    - Order: $$\sum_{i=1}^{N}\left(\frac{1}{N}\times i\right) = \frac{1}{N}\sum_{i=1}^{N}(i) = \frac{1}{N}\times\frac{N(N+1)}{2} = \boldsymbol{O(N)}$$

How to Calculate the Order O()?

- **Statements:**
  - O(1)
  - Exception for the function calls!
- **Conditions:**
  - The condition statements.
  - Choose the max body order.
- **Loops:**
  - Number of iterations * body order.

Example 1:		
Code	**Type**	**Order**

i = 1;
sum = 0;
while (i <= n)
{
  i = i + 1
  sum = sum + i
}
	Statement
	Statement
	Loop
	
	Statement
	Statement
		1
		1
		#iterations * body
		
		1
		1

#iterations = N & body = 1. So, Final order = O(N)		

Example 2:		
Code	**Type**	**Order**

Switch (choice)
  Case 1:
  Print N
  break
  Case 2:
  MaxItem = GetMax(A, N)
  Print MaxItem
  break
  Case 3:
  Sum = 0
  For I = 1 to N
  If (Sum < GetMax(A, N))
  Sum = Sum + A[I]
  End if
  End for
  Print Sum
  break
End Switch
	Statement
	Statement
	Statement
	Statement
	Statement
	Loop
	Statement
	Statement
	Statement
	Statement
	Loop
	Loop
	Statement
	
	Statement
	Statement
		1
		1
		1
		1
		1
		#iterations 1 * body 1
		1
		1
		1
		1
		#iterations 2 * body 2
		#iterations 3 * body 3
		1
		
		1
		1

GetMax function(#iterations 1 * body 1 & #iterations 2 * body 2): **#it = N & body = 1, So, it takes O(N)** **Case 3 loop: #iteration2 = N & body2 = 1, So, it takes O(N)** **So, Final order = O(N^2)**		

Example 3:

Code	Type	Order

i = 1;
sum = 0;
j = 1;
while (i <= n) {
  while (j <= n) {
  sum = sum + i;
  j = j + 1;
  }
  i = i+1;
}
	Statement
	Statement
	Statement
	Loop
	Loop
	Statement
	Statement
	
	Statement
		1
		1
		1
		#iterations 1 * body 1
		#iterations 2 * body 2
		1
		1
		
		1

Inner loop: #iteration1 = N & body1 = 1
Outer loop: #iteration2 = N & body2 = 1 (the inner loop not working unless the first time only)
So, Final order = O(N)

Example 4: (find the exact complexity (Θ))

Code	Type	Order
Sum = 0 For I = 1 to Min(1000, N) Sum = Sum + A[I] End for	Statement Loop Statement Statement	1 #iterations * body 1 1

This loop will run 1000 times for the worst case and this still constant time.
So, Final order = Θ(1)

Example 5:

Code	Type	Order

while (n > 1)
{
  n = n/10;
}
	Loop
	
	Statement
		#iterations * body
		
		1

The number of iteration values different here:

Iteration number	1	2	3	4	…	K
Iterator value	N	N/10	$N/10^2$	$N/10^3$	…	N/K^{k-1}

Termination: when the Iterator reaches 1
when N / 10k-1 = 1
$N = 10^{k-1}$
$K = \log_{10}(N + 1)$
So, Final order = O(log N)

Example 6:

Code	Type	Order
i=1;	Statement	1
sum = 0;	Statement	1
while (i <= n)	Loop	#iterations1 * body1
{		
j=1;	Statement	1
while (j <= i)	Loop	#iterations2 * body2
{		
sum = sum + i;	Statement	1
j = j + 1;	Statement	1
}		
i = i +1;	Statement	1

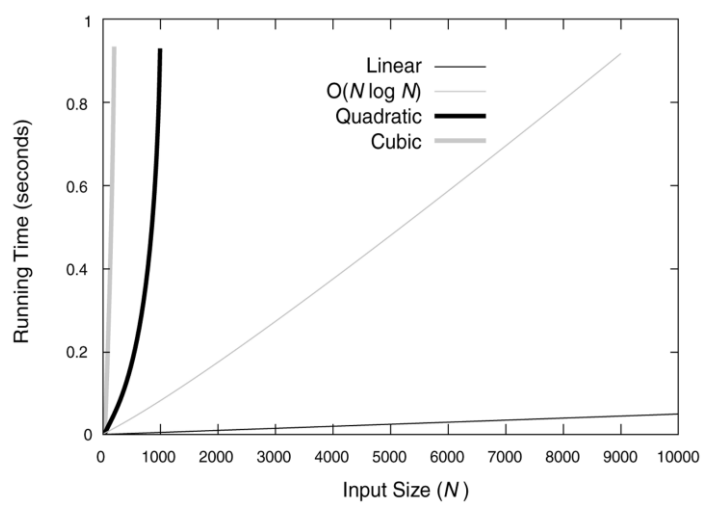
# of iterations of inner loop for each of outer loop

Iteration number	1	2	3	4	...	N
Iterator value	1	2	3	4	...	N

So, the total of iterations = $\mathbf{1 + 2 + 3 + \cdots + N} = \sum_{i=1}^{N} i = \frac{N(N+1)}{2} = \frac{N^2}{2} + \frac{N}{2}$ **=> O(N^2)**

Common Complexity Classes:

Function	Growth Rate Name
c	Constant
log(N)	Logarithmic
$\log_2 N$	Log-squared
N	Linear
N log(N)	
N^2	Quadratic
N^3	Cubic
2^N	Exponential



What is the difference between the Algorithm Behavior and Asymptotic Notations:

Algorithm Behavior	Asymptotic Notations
Worst	Upper bound: O()
Best	Lower bound: Ω()
Expected	Exact bound: Θ()

Asymptotic Notations:

It is studying the time complexity at large input size and we can be expressed by:

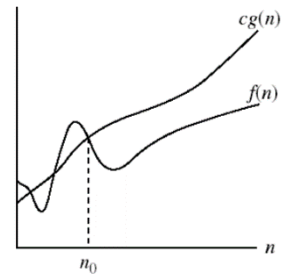
- Upper bound: O()
- Lower bound: Ω()
- Exact bound: Θ()

We always look for Θ, but when it's difficult to obtain Θ look for O ,Ω.

O-Notation:

$\mathrm{f(n)} = \mathrm{O}(\mathrm{g(n)})$ when $\mathrm{cg(n)} \geq \mathrm{f(n)}$ for all $\mathrm{n} \geq n0$

where c is a constant and c, n0 is positive



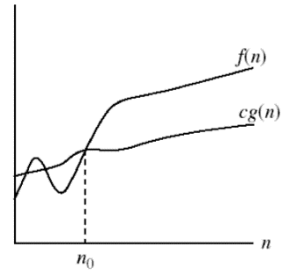
Example1: Dis/Prove the following: (by definition)	
Expression	The prove
$2\mathrm{N}^2 + 10\mathrm{N} = \mathrm{O}(\mathrm{N}^2)$	We need to prove: $\mathrm{C}(N^2) \geq 2N^2 + 10\mathrm{N}$ where $\mathrm{N} \geq \mathrm{n0}$ and $\mathrm{C}, \mathrm{n0} > 0$ $C \geq \frac{2N^2+10N}{N^2}$ So, $C \geq 2(1+\frac{5}{N})$ Let N = 1 (n0 = 1) then, $C \geq 12$ That mean that when n0 = 1 any $C \geq 12$ should satisfy the equation Now we need to prove this when any $N \geq 1$ So, we let the C satisfy the equation when $C \geq 12$ Let C = 14 and ask is $C \geq 2(1+\frac{5}{N})$ valid when C = 14 and $N \geq 1$? That mean, is $14 \geq 2(1+\frac{5}{N})$? (valid) So, $2\mathrm{N}^2 + 10\mathrm{N} = \mathrm{O}(\mathrm{N}^2)$

Example2: Dis/Prove the following: (by definition)	
Expression	The prove
$3^N \neq \mathrm{O}(2^N)$	We need to prove: $\mathrm{C}(2^N) \geq 3^N$ where $\mathrm{N} \geq \mathrm{n0}$ and $\mathrm{C}, \mathrm{n0} > 0$ $C \geq \frac{3^N}{2^N}$ So, $C \geq (\frac{3}{2})^N$ Let N = 1 (n0 = 1) then, $C \geq 1.5$ That mean that when n0 = 1 any $C \geq 1.5$ should satisfy the equation Now we need to prove this when any $N \geq 1$ So, we let the C satisfy the equation when $C \geq 1.5$ Let C = 2 and ask is $C \geq (\frac{3}{2})^N$ valid when C = 2 and $N \geq 1$? That mean, is $2 \geq (\frac{3}{2})^N$? (Not valid) So, $3^N \neq \mathrm{O}(2^N)$

Ω -Notation:

$\mathrm{F(n)} = \Omega(\mathrm{g(n)})$ when $\mathrm{cg(n)} \leq \mathrm{f(n)}$ for all $\mathrm{n} \geq n0$

where c is a constant and $c, n0$ *is positive*



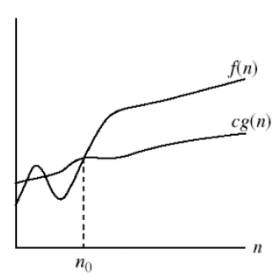
Example1: Dis/Prove the following: (by definition)	
Expression	The prove
$0.5n^3 + 100n = \Omega(n^3)$	We need to prove: $\mathrm{C}(n^3) \leq 0.5n^3 + 100n$ where $\mathrm{N} \geq \mathrm{n0}$ and $\mathrm{C, n0} > 0$ $C \leq \frac{0.5n^3+100n}{n^3}$ So, $C \leq 0.5 + \frac{100}{n^2}$ Let N = 1 (n0 = 1) then, $C \leq 100.5$ That mean that when n0 = 1 any $C \leq 100.5$ should satisfy the equation Now we need to prove this when any $N \geq 1$ So, we let the C satisfy the equation when $C \leq 100.5$ Let C = 90 and ask is $C \leq 0.5 + \frac{100}{n^2}$ valid when C = 90 and $N \geq 1$? That mean, is $90 \leq 0.5 + \frac{100}{n^2}$? (Not valid) So, $0.5n^3 + 100n \neq \Omega(n^3)$

Example2: Dis/Prove the following: (by definition)	
Expression	The prove
$5n^2 = \Omega(n)$	We need to prove: $\mathrm{C}(n) \leq 5n^2$ where $\mathrm{N} \geq \mathrm{n0}$ and $\mathrm{C, n0} > 0$ $C \leq \frac{n^2}{n}$ So, $C \leq n$ Let N = 1 (n0 = 1) then, $C \leq 1$ That mean that when n0 = 1 any $C \leq 1$ should satisfy the equation Now we need to prove this when any $N \geq 1$ So, we let the C satisfy the equation when $C \leq 1$ Let C = 0.5 and ask is $C \leq n$ valid when C = 0.5 and $N \geq 1$? That mean, is $0.5 \leq n$? (valid) So, $5n^2 = \Omega(n)$

Θ -Notation:

$\mathrm{f(n)} = \Theta(\mathrm{g(n)})$ when $\mathrm{c1g(n)} \leq \mathrm{f(n)} \leq \mathrm{c2g(n)}$ for all $\mathrm{n} \geq n0$

where $c1, c2$ *is a constants and* $c1, c2, n0$ *is positive*



Example2: Dis/Prove the following: (by definition)	
Expression	The prove
$0.5n^2 - 3n = \Theta(n^2)$	We need to prove: $\mathrm{C1}(n^2) \leq 0.5n^2 - 3n \leq \mathrm{C1}(n^2)$ where $\mathrm{N} \geq \mathrm{n0}$ and $\mathrm{C1}, \mathrm{C2}, \mathrm{n0} > 0$ $\frac{\mathrm{C1}(n^2)}{n^2} \leq \frac{0.5n^2-3n}{n^2} \leq \frac{\mathrm{C1}(n^2)}{n^2}$ So, $C1 \leq 0.5 - \frac{3}{n} \leq C2$ Let N = 10 (n0 = 10) then, $C1 \leq 0.3 \leq C2$ That mean that when n0 = 10 any $C1 \leq 0.3 \leq C2$ should satisfy the equation Now we need to prove this when any $N \geq 1$ So, we let the C satisfy the equation when $C1 \leq 0.3 \leq C2$ Let C1 = 0.2 & C2 = 1 and ask is $C1 \leq 0.5 - \frac{3}{n} \leq C2$ valid when C1 = 0.2 & C2 = 1 and $N \geq 1$? That mean, is $0.2 \leq 0.5 - \frac{3}{n} \leq 1$? LHS: $0.2 \leq 0.5 - \frac{3}{n}$ (valid) RHS: $0.5 - \frac{3}{n} \leq 1$ (valid) So, $0.5n^2 - 3n = \Theta(n^2)$

Example2: Dis/Prove the following: (by definition)	
Expression	The prove
$100\mathrm{n} + 5 = \Theta(n^2)$	We need to prove: $\mathrm{C1}(n^2) \leq 100n + 5 \leq \mathrm{C1}(n^2)$ where $\mathrm{N} \geq \mathrm{n0}$ and $\mathrm{C1}, \mathrm{C2}, \mathrm{n0} > 0$ $\frac{\mathrm{C1}(n^2)}{n^2} \leq \frac{100n+5}{n^2} \leq \frac{\mathrm{C1}(n^2)}{n^2}$ So, $C1 \leq \frac{100n+5}{n^2} \leq C2$ Let N = 1 (n0 = 1) then, $C1 \leq 105 \leq C2$ That mean that when n0 = 1 any $C1 \leq 105 \leq C2$ should satisfy the equation Now we need to prove this when any $N \geq 1$ So, we let the C satisfy the equation when $C1 \leq 105 \leq C2$ Let C1 = 100 & C2 = 110 and ask is $C1 \leq \frac{100n+5}{n^2} \leq C2$ valid when C1 = 100 & C2 = 110 and $N \geq 1$? That mean, is $100 \leq \frac{100n+5}{n^2} \leq 110$? LHS: $100 \leq \frac{100n+5}{n^2}$ (not valid) So, $100\mathrm{n} + 5 \neq \Theta(n^2)$

Example3: calculate big-Θ	
Code	Analysis

sum = 0;
for(i = 1; i < N; i*= 2)
{
sum += i;
}
	The Big-O for this code is O(N) but, by tracing:
	i = 1, 2, 4, 16, …., 2^L (L is number of iterations)
	So, $2^L = N$ => $\log(2^L) = \log(N)$ => $L = \log_2(N)$
	So, $\Theta(\log_2(N))$

Example4: calculate big-Θ	
Code	Analysis

sum = 0; i= 1;
while (i< N)
{
sum += i;
if (i*i > N) break;
i++;
}
	The Big-O for this code is O(N) but, by tracing:
	i = 1, 2, 3, 4, …., L^2 (L is number of iterations)
	So, $L^2 = N$ => $L = \sqrt{N}$
	So, $\Theta(\sqrt{N})$

Analyzing the Recursive code:

1. **Calculate Running Time T(N) [Recurrence]:**
   - T(Base) = Time of Base Case
   - T(N) Time of Normal Code + T(Recursive Code)
2. **Solve it to Get the Order (Θ or O)**
   - Iteration method
   - Master method
   - Recursive Tree method

Examples: (Calculate Running Time T(N) [Recurrence]):

Example1: (Factorial)		
Code	**Component**	**T(N)**

if (N = 0) then
return 1
else
return N * Fact(N-1)
	Base Case
	
	Recursive Code
		T(0) = Θ(1)
		
		T(N) = Θ(1) + T(N-1)

Example2: (Fibonacci)		
Code	**Component**	**T(N)**

if (N == 0 or N == 1)
return 1
else
return Fib(N-1) + Fib(N-2)
	Base Case
	
	Recursive Code
		T(1) = Θ(1)
		
		T(N) = T(N-1) + T(N-2)

Example3: (Binary Search)		
Code	**Component**	**T(N)**

BinSrch(A, I, S, E)
if (S>E) return false
M = (S + E) / 2
If (I == A[M]) return true
else If (I > A[M]) return BinSrch(A, I, M+1, E)
else If (I < A[M]) return BinSrch(A, I, S, M-1)
	Base Case
	
	Recursive Code
		T(1) = Θ(1)
		
		T(N) = Θ(1)+ T(N/2)

Recursion:

- Function calls itself for one or more times.
- Consists of two parts:
  - Recursive part
  - Base case (stopping condition)
- General form:

```
Fun(…)
    ….          Pre-code (Base case)
    ….          Pre-code (others)
    Fun(…), Fun(…), …etc.     Recursive Code
    ….
    ….          Post-code
    ….
```

Example: (Factorial)

Code	Component	Trace

Fact(N)
if (N = 0) then
return 1
else
F = Fact(N-1)
return N * F
endif
end
	Base Case
	
	Recursion
	Post Code
		algorithm Test
		ans <- Fact(3)
		endalgorithm
		
		if (3 = 0) then return 1
		Else
		F = Fact(2)
		returns 3 * F
		endif
		endfunction
		
		if (3 = 0) then return 1
		Else
		F = Fact(2)
		returns 3 * F
		endif
		endfunction
		
		if (3 = 0) then return 1
		Else
		F = Fact(2)
		returns 3 * F
		endif
		endfunction
		
		if (0 = 0) then return 1
		else
		Fact returns 1
		endif
		endfunction

Recursion analyzing methods:

1- Iteration method.
2- Master method.
3- Recursive Tree method (we usually use it because it solves most of the problems.

2) Master Method:

1. **Direct apply:** We apply this way if the recurrences on following form:

$$\mathbf{T(n) = aT(n/b) + f(n), T(Base) = \Theta(1)} \qquad (\boldsymbol{where}: \boldsymbol{a \geq 1, b > 1})$$

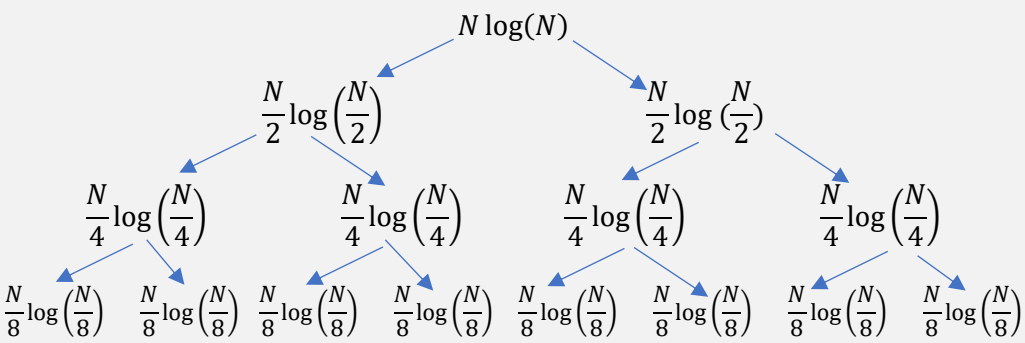
Examples (Direct apply)	
T(N)	**complexity**
T(N) = 2 T(N/2) + CN^2	T
T(N) = T(N/3) + O(1)	T
T(N) = 2 T(3N/2) + √N	F (b = 2/3 < 1)
T(N) = T(N-1) + T(N-2) + C	F (not on the form)

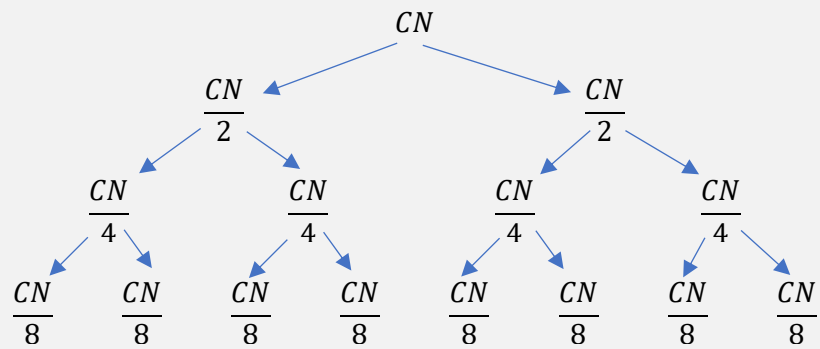
2. **For the relative growth rates of a, b, and f(n):**
   I. **Case 1:** If f(n) = O($n^{\log_b a - \varepsilon}$) and $\varepsilon > 0$ then the time complexity of the algorithm is dominated by the recursive subproblems. The overall time complexity can be expressed as T(n) = Θ($n^{\log_b a}$).
   II. **Case 2:** If f(n) = Θ($n^{\log_b a}$) then the time complexity of the algorithm is evenly balanced between the recursive subproblems, and the work done outside the recursive calls. The overall time complexity can be expressed as T(n) = Θ($n^{\log_b a} * \log n$).
   III. **Case 3:** If f(n) = Ω ($n^{\log_b a + \varepsilon}$) and $\varepsilon > 0$ and $a * f\left(\frac{n}{b}\right) \leq c * f(n), \quad 0 < c < 1$) then the time complexity of the algorithm is dominated by the work done outside the recursive calls. The overall time complexity can be expressed as T(n) = Θ(f(n)).

Example1	
T(N)	**complexity**
T(n) = 9T(n/3) + n	a=9, b=3, f(n) = n $n^{\log_b a} = n^{\log_3 9} = n^2$ Since f(n) = O($n^{\log_3 9 - \varepsilon}$), where ε =1, Case 1 applies: T(n) = Θ($n^{\log_b a}$) when f(n) = O($n^{\log_b a - \varepsilon}$) So, T(n) = Θ($n^2$)
T(n) = 4T(n/2) + n^2	a=4, b=2, f(n) = n^2 $n^{\log_b a} = n^{\log_2 4} = n^2$ Since f(n) = Θ($n^{2-\varepsilon}$), where ε =0, Case 2 applies: T(n) = Θ($n^2 \log n$) when f(n) = Θ($n^{\log_b a - 0}$) So, T(n) = Θ($n^2 \log n$)
T(n) = 3T(n/4) + $n\log n$	a=3, b=4, f(n) = $n\log n$ $n^{\log_b a} = n^{\log_4 3} = n^{0.793}$ Since f(n) = Ω($n^{2+\varepsilon}$), where ε =0.2, Case 3 applies: Check: $a * f\left(\frac{n}{b}\right) \leq c * f(n), \quad 0 < c < 1$) $3 * \left(\frac{n}{4}\right) \log \frac{n}{4} \leq \frac{3}{4} * n \log n, \quad c = \frac{3}{4}$ (Valid) So, T(n) = f(n) = Θ($n\log n$)

3) Recursion Tree Method steps:

1. Draw the tree for at least three levels.
2. Calculate number of levels
3. Calculate complexity of each level
4. Calculate complexity of LAST level
5. Total = sum of all levels

Example1: T(N) = 2T(N/2) + N log(N); T(1) = C	
Step	**Solution**
1- Draw the tree for at least three levels.	
2- Calculate number of levels	$N, \frac{N}{2}, \frac{N}{4}, \frac{N}{8}, \dots, \left(\frac{1}{2}\right)^L N$ where L: # of levels $\left(\frac{1}{2}\right)^L N = 1 \quad So, L = \log_2 N$
3- Calculate complexity of each level	$N\log(N) + 2\left[\frac{N}{2}\log\left(\frac{N}{2}\right)\right] + 4\left[\frac{N}{4}\log\left(\frac{N}{4}\right)\right] + 8\left[\frac{N}{8}\log\left(\frac{N}{8}\right)\right] + \cdots.$ $= N\log(N) + N\log\left(\frac{N}{2}\right) + N\log\left(\frac{N}{4}\right) + N\log\left(\frac{N}{8}\right) = N\log\left(\frac{N}{2^i}\right)$ $= N\log(N) - N\log(2^i) = N\log(N) - N * i \quad where\ i = level\ numbers$
4- Calculate complexity of LAST level	= # leaves * T(base case) $= 2^L * 1 \quad and\ L = \log_2 N \quad So, the\ copmlexity = 2^{\log_2 N} = N$
5- Total	Total = sum of all levels $\text{Total} = \text{N} + \sum_{i=0}^{\log_2(N)-1} N\log(N) - N * i = N + N\log(N) - N\sum_{i=0}^{\log_2(N)-1} i$ So, $\text{Total} = \theta(\text{N}(\log N)^2)$

Example2: T(N) = 2T(N/2) + CN; T(1) = C	
Step	**Solution**
1- Draw the tree for at least three levels.	
2- Calculate number of levels	$N, \frac{N}{2}, \frac{N}{4}, \frac{N}{8}, \dots, \left(\frac{1}{2}\right)^L N$ where L: # of levels $\left(\frac{1}{2}\right)^L N = 1 \quad So, L = \log_2 N$
3- Calculate complexity of each level	$CN + 2\left[\frac{CN}{2}\right] + 4\left[\frac{CN}{4}\right] + 8\left[\frac{CN}{8}\right] + \cdots.$ $= CN + CN + CN + CN = CN$
4- Calculate complexity of LAST level	= # leaves * T(base case) $= 2^L * C \quad and\ L = \log_2 N \quad So, the\ copmlexity = CN$
5- Total	Total = sum of all levels $\text{Total} = \text{CN} + \sum_{i=0}^{\log_2(N)-1} CN = CN + N\log(N)$ So, Total $= \theta(\text{Nlog(N)})$

Example3: Analyze:

```
F2(N)
    M = 10^6
    If (N > 1)
        For I = 1 to N do
            J = M
            While J > 1 Do
                Print "Hello World!!"
                J = J / 2
            End while
        End For
        If ((F2(N/2) + random(1,M)) % 2 = 0)
            return F2(N/4)
        Else
            return F2(N/2)
    End If
End F2
```

Non-Recursive Part

Recursive Part

Non-Recursive Part

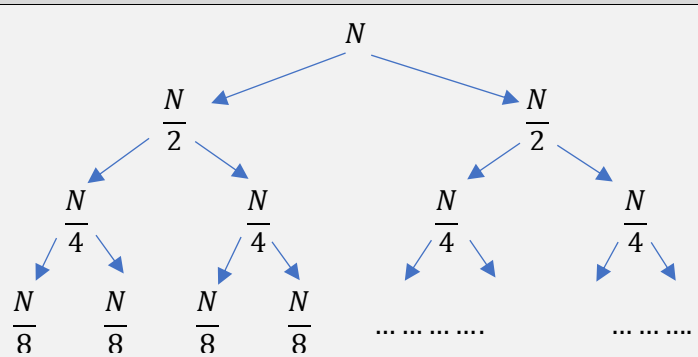
Outer loop = #iteration * Body = N * O(the inner loop)
Inner loop = #iteration * Body = 10^6 * 1 = θ(1)
So, the Non-Recursive Part = θ(N)

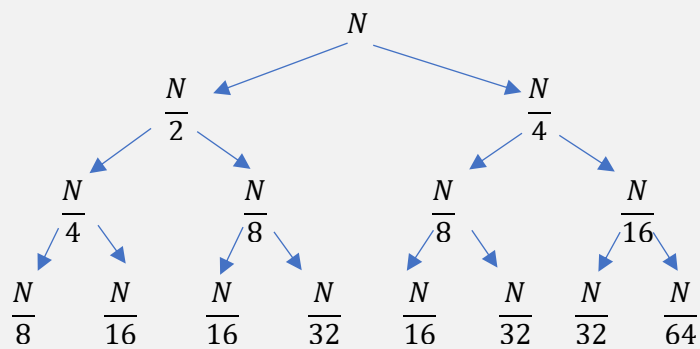
Recursive Part

T(N) = $\theta(N) + T\left(\frac{N}{2}\right) + T\left(\frac{N}{4}\right)$ → [if condition part] (Best Case)

OR

$\theta(N) + 2T\left(\frac{N}{2}\right)$ → [else condition part] (Worst Case)

Worst Case Steps	$\mathbf{T(N) = \theta(N) + 2T\left(\frac{N}{2}\right)}$
1- Draw the tree for at least three levels.	
2- Calculate number of levels	$N, \frac{N}{2}, \frac{N}{4}, \frac{N}{8}, \ldots, \left(\frac{1}{2}\right)^L N \quad So, L = \log_2 N$ where L: # of levels
3- Calculate complexity of each level	$N + 2\left[\frac{N}{2}\right] + 4\left[\frac{N}{4}\right] + 8\left[\frac{N}{8}\right] + \cdots. = N + N + N + N = N$

4- Calculate complexity of LAST level	= # leaves * T(base case) $= 2^L * C \quad and\ L = \log_2 N \quad So, the\ copmlexity = N$
5- Total	Total = sum of all levels $\text{Total} = \text{N} + \sum_{i=0}^{\log_2(N)-1} N = N + N\log(N)$ So, Total $= \theta(\text{Nlog(N)})$
Best Case Steps	$\mathbf{T(N) = \theta(N) + T\left(\frac{N}{2}\right) + T\left(\frac{N}{4}\right)}$
1- Draw the tree for at least three levels.	
2- Calculate number of levels	Because the tree unbalanced, so we will calculate it over & under estimate: Over-estimate(long branch): $N, \frac{N}{2}, \frac{N}{4}, \frac{N}{8}, \dots, \left(\frac{1}{2}\right)^L N \quad So, L = \log_2 N \quad$ where L: # of levels under-estimate(short branch): $N, \frac{N}{4}, \frac{N}{16}, \frac{N}{32}, \dots, \left(\frac{1}{4}\right)^L N \quad So, L = \log_4 N \quad$ where L: # of levels
3- Calculate complexity of each level	$N + \frac{3}{4}N + \frac{9}{16}N + \frac{27}{64}N + \cdots. = \left[\frac{3}{4}\right]^i N$
4- Calculate complexity of LAST level	= # leaves * T(base case) $= 1$ (there is only 1 leave at the end of tree) So, it is $= \theta(1)$
5- Total	Total = sum of all levels Over-estimate: $\text{Total} = 1 + \sum_{i=0}^{\log_2(N)-1} \left[\frac{3}{4}\right]^i N = 1 + N * \frac{1}{1-\frac{3}{4}}$ So, Total = $\theta(\text{N})$ Under-estimate: $\text{Total} = 1 + \sum_{i=0}^{\log_4(N)} \left[\frac{3}{4}\right]^i N = 1 + N$ So, Total = $\theta(\text{N})$

Algorithms Design

What does Algorithms Design mean?

It is the process of creating a step-by-step procedure or set of rules to solve a specific computational problem. It involves formulating a solution strategy that can be implemented in a computer program or any other computational system.

Algorithms Design techniques:

- Brute Force & Exhaustive Search.
- Divide & conquer.
- Transformation.
- Dynamic Programming.
- Greedy.
- Randomization.
- Iterative Improvement.

Brute Force & Exhaustive Search:

Straightforward way to solve a problem, based on the definition of the problem itself; often involves checking all possibilities.

Advantages:

- widely applicable.
- easy.
- good for small input sizes.

Disadvantages:

- often inefficient for large inputs.

Example1: Maximum Sequence Sum

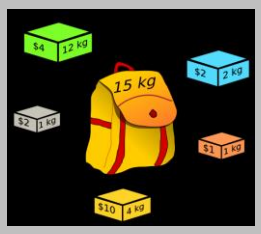
- **Given:** array of N numbers.
- **Required:** find a sequence of contiguous numbers with max sum.

Steps	Solution

1- Find all possible sequences.
2- Choose one with max sum.
3- Select from:
{31,-41,59,26,-53,58,97,-93,-23,84}:
{59,26,53,58,97}
4- It takes Complexity: $\Theta(N^2)$
	Pseudocode:
	publicstaticintMCSS(int[]a) {
	int max=0,sum=0;
	for(i=0; i < a.length; i++){
	for(j=i; j < a.length; j++){
	sum=0;
	for(k=i;k<=j;k++): sum+=a[k];
	if(sum>max): max=sum;
	} }
	returnmax; }

Example2: Knapsack

- **Given:**
  - There are N different items in a store.
  - Each item has only one instance.
  - The item i weighs w_i pounds and is worth $\$v_i$.
  - A thief breaks in.
  - He can carry up to W pounds in his knapsack.
- **Required:**
  - What should he take to maximize his benefit?



Items	Solution
item 1: 7 lbs., $42 item 2: 3 lbs., $12 item 3: 4 lbs., $40 item 4: 5 lbs., $25 bag W = 10	Consider every possible subset of items. Calculate total value and total weight. Discard if weight is more than W. Then choose the remaining subset with maximum total value.

subset	total weight	total value
Ø	0	$0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1,2}	10	$54
{1,3}	11	infeasible
{1,4}	12	infeasible
{2,3}	7	$52
{2,4}	8	$37
etc.		

Here we need to check 16 possibilities = 2^4 possibilities
So, The Running time: $\Omega(2^n)$ time

Example3: Select K_{th} Order Element

- **Given:** array of N numbers.
- **Required:** find value of Kth smallest/largest element

Steps	Solution
1- Sort the array. 2- Then select.	**Pseudocode:**

```
publicstaticintMCSS(int[]a) {
        int max=0,sum=0;
        for(i=0; i < a.length; i++){
                for(j=i; j < a.length; j++){
                        sum=0;
                        for(k=i;k<=j;k++):   sum+=a[k];
                        if(sum>max):   max=sum;
                } }
        returnmax;  }
```

Divide & Conquer:

It is an algorithm technique that involves breaking down a complex problem into smaller, more manageable subproblems, solving them independently, and then combining their solutions to obtain the final result. It follows a recursive strategy of dividing the problem into smaller subproblems until they become simple enough to be solved directly.

Its steps:

1- **Divide:** the problem into small distinct sub-problem(s).
2- **Conquer:** the sub-problems by solving them recursively in the same manner.
3- **Combine:** the solution of sub-problems to get the final solution.

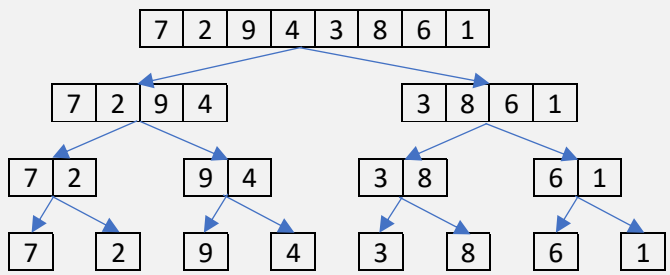
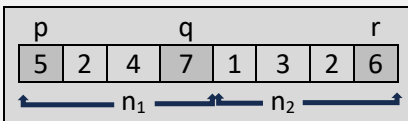
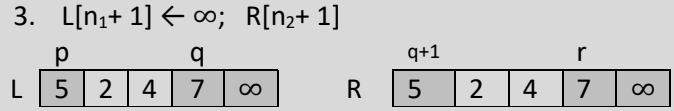
It's a Recursive by nature:

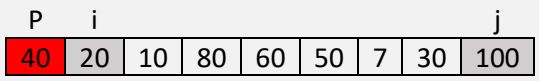
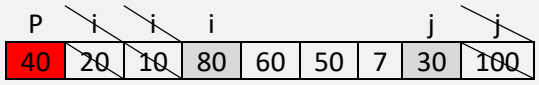
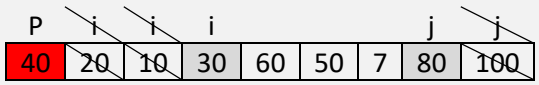
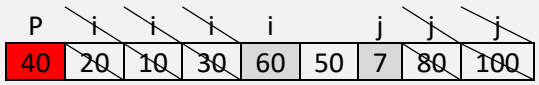
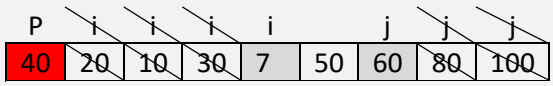
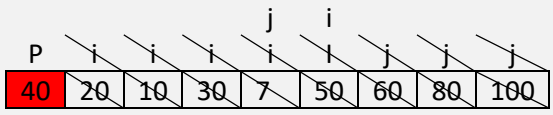
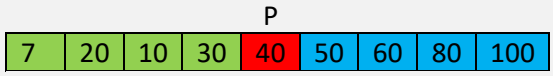
	Fun(…)
	…. **Pre-code (Base case)**
Divide	**….** **Pre-code (others)**
Conquer	**Fun(…), Fun(…), …etc.** **Recursive Code**
Combine	**….** **Post-code**

Time Complexity T(N) = Time of Normal Code + Time of Recursion Code + …etc.

= (Divide & combine) + (Conquer) + …etc.

Example1: Binary Search	
Steps	**Solution**
Divide	check the middle element.
Conquer	Recursively search 1 subarray.
Combine	Trivial (we don't need it, just return the value or false).
Recurrence T(N)	= (Divide & combine) + (Conquer) = O(1) + T(N/2)
Time Complexity	By Recursion Tree: $\theta(\log(N)$
Trace	Find(7) -> {1, 3, 7, 8, 9, 10, 11}: check the center {8} then (7 < 8) -> {1, 3, 7}: check the center {3} then (7 > 3) -> {3, 7}: check the center {7} then return 7
Pseudocode	f C e 1 3 7 8 9 10 11 n_1 n_2 **==> Find(array, value):** 1. Compare the value with the central of array C. 2. If sub array is empty, then return not found; 3. If (value = C), then return found; 4. Else if (value > C), then Find(C+1, e); 5. Else Find(f, C-1);

Example2: Merge Sort	
Steps	**Solution**
Divide	Trivial (split into two halves).
Conquer	Recursively sort 2 subarrays(2 halves).
Combine	Merge two sorted subarrays[linear time]
Recurrence T(N)	= (Divide & combine) + (Conquer) = $\theta(N)$ + 2T(N/2)
Time Complexity	By Recursion Tree: $\theta(N\log(N)$
Trace	 {2, 7} {4, 9} {3, 8} {1, 6} {2, 4, 7, 9} {1, 3, 6, 8} {1, 2, 3, 4, 6, 7, 8, 9} We merge the 2 subsets in a combined step by compare the first element in first subarray by first element in second subarray then move the cursor when we select the smallest one and if one subset is empty, we select the other elements in the second subarray.
Pseudocode	 **==> MERGE(A, p, q, r):** 1. Compute n_1 andn_2 2. Copy the first n_1 elements into: L[1 . . n_1+ 1] and the next n_2 elements into R[1 . . n_2+ 1] 3. L[n_1+ 1] ← ∞; R[n_2+ 1]  4. i←1; j ←1 5. **for** k ←p **to** r **do**: 6. **if** L[i] ≤ R[j] 7. **then** A[k] ←L[i] 8. i← i+ 1 9. **else** A[k] ←R[j] 10. j← j + 1

Example3: Quick Sort	
Steps	**Solution**
Divide	1) select a pivot. 2) place it in its correct position (Move every element smaller to left and every element else to the right).
Conquer	Recursively sort 2 subarrays
Combine	Trivial
Recurrence T(N)	= (Divide & combine) + (Conquer) = $\theta(N)$ + T(L) + T(N-L) Where L is length of one of the 2 subarrays
Trace	 (We put the 2 pointers I and j then Move them after making sure that (j element L)<P & (i element R)>P  We swap the 2 elements.     After i > j, we stop and swap P with j then divide the array to 2 sub arrays the recursive  ……..
Analysis	**The Best Case:** When each call to partition splits the array into two equal parts. It takes: T(N) = 2T(N/2) + Θ(N) => Θ(N log(N)) by Recursion Tree. **The Worst Case:** When each call to Partition splits the array into one part only. It takes: T(N) = T(N-1) + Θ(N) => Θ(N^2) by Recursion Tree. This case happened when the array is already sorted or reverse sorted. **The Case between Worst and Best:** When each call to Partition splits array into 9-to-1 proportional split It takes: T(N) = T(N/10) + T(9N/10) + Θ(N) => Θ(N log(N)) by Recursion Tree.

How to avoid the Worst Case:

1. Randomly shuffle the input array in Θ(N) before sorting
2. Good choice for the pivot, may be:
   - The element in the middle position.
   - Median of first, last, and middle.
   - Median of k sampled elements.
   - A random element.

Pseudocode

```
==> quicksort(list, first, last)
{
        if first<last:
        splitpoint = partition(list, first, last)
        quicksort (list, first, splitpoint-1)
        quicksort (list, splitpoint+1, last)
}

==> partition(list, first, last)
{
        pivotvalue= list[first]
        leftmark= first+1
        rightmark= last
        done = False

        while not done:
                while (leftmark<= rightmarkand list[leftmark] <= pivotvalue):
                        leftmark = leftmark + 1

                while (list[rightmark] >= pivotvalue and rightmark >= leftmark):
                        rightmark = rightmark – 1

                if rightmark< leftmark:
                        done = True
                else:
                        temp = list[leftmark]
                        list[leftmark] = list[rightmark]
                        list[rightmark] = temp

        temp = list[first]
        list[first] = list[rightmark]
        list[rightmark] = temp
        return (rightmark)
}
```

Example4: Select K_{th} Order Element	
Steps	**Solution**
Divide	1) Select a pivot . 2) Place it in its correct position [linear time].
Conquer	Recursively select in 1 subarray: - Select K_{th} in left subarray, OR - Select (K-J)thin right subarray.
Combine	Trivial
Analysis	**The Best Case:** When each call to Partition splits the array into one half It takes: T(N) = T(N/2) + Θ(N) => Θ(N) by Recursion Tree. **The Worst Case:** When each call to Partition splits the array into one part with N-1 It takes: T(N) = T(N-1) + Θ(N) => Θ(N^2) by Recursion Tree. This case happened when the array is already sorted or reverse sorted. **The Case between Worst and Best:** When each call to Partition splits array into factor from array size It takes: T(N) = T(N/P) + Θ(N) => Θ(N) by Recursion Tree. **How to avoid the Worst Case:** Randomly chose the pivot.

Pseudocode	**==> findKthElement(list, first, last, k)**
	{
	  if (left == right)
	  return A[left]
	  position = partition(A, left, right)
	  count = pos - left + 1
	  if (count == K)
	  return A[pos]
	  if (count > K)
	  return findKthElement (A, left, pos-1, K)
	  else
	  return findKthElement (A, pos+1, right, K-i)
	}
	
	==> partition(list, left, right)
	{
	  x = A[r], i = i-1
	  for (j = l to r-1)
	  if (A[j] <= x)
	  i = i + 1
	  swap(A[i], A[j])
	  swap(A[i+1], A[r])
	  return (i+1)
	}

Dynamic Programming:

It is a careful brute-force paradigm and applied all Divide & Conquer steps with overlapped sub-problems. We usually solve the problem in a bottom-up manner unlike D&C it solves the problem in top-down and stores the previous solutions in extra storage to reuse them.

Why use it:

It leads to efficient solutions (usually turns exponential 'A^N' to polynomial 'N^B').

When use it:

Often in optimization problems, in which:

- There can be many possible solutions,
- Need to try them all. (Brute force but carful)
- Wish to find a solution with the optimal (e.g., min or max) value.

How use it:

There are 2 ways:

Top-down	Bottom-up
Save solution values	Save solution values
Recursive	Loop
Called "memoization"	Called "building table"
Saved only the required	Save all possible solution
Used when no need to solve ALL subproblems	Used when we need to solve ALL subproblems

Conditions:

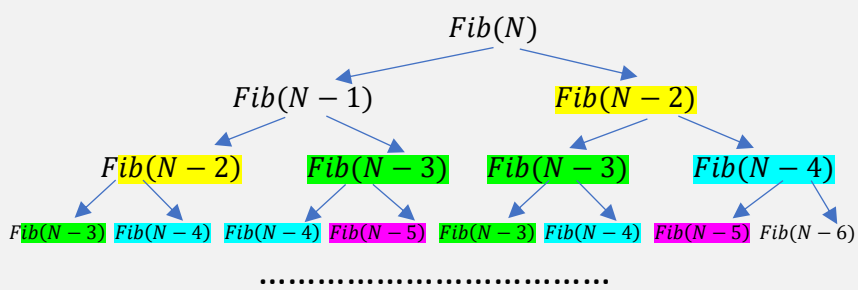
1. The optimal solution to the problem contains within its optimal solutions to two or more sub-problems.
2. Overlapping sub-problems.

Solution Steps:

1. **Define the problem:** Characterize the structure of the optimal solution (params & return)
2. **D&C Solution:** Recursively define the value of an optimal solution:
    a. Guess the possibilities (trials) for the current problem.
    b. Formulate their subproblems.
    c. Define base case.
    d. Relate to them in a recursive way.
3. **Check overlapping**
4. **Switch D&C to DP Solution:**
    a. OPT1: top-down (recurse) + memoization
        i. Define extra storage (hash table OR array) (array dimensions = #varying parameters in the function)
        ii. Initialize it by NIL
        iii. Solve recursively (top-down)
            1. If not solved (NIL): solve & save
            2. If solved: retrieve
        iv. Return solution of main problem

b. OPT2: bottom-up (loop) + building table:
    i. Define extra storage (dictionary OR array) (array dimensions = #varying parameters in the function)
    ii. Equation to fill-in this table
    iii. Solve iteratively (bottom-up)
        1. If base case: store it
        2. else: calculate it
    iv. Return solution of main problem

5. **Extract the Solution:** Construct an optimal solution from computed information
    a. Save info about the selected choice for each subproblem
        i. Define extra array (same dims as solution storage) to store the best choice at each subproblem
    b. Use this info to construct the solution
        i. Backtrack the solution using the saved info starting from the solution of main problem

Example1: Fibonacci	
Steps	**Solution**
Define the problem	Param: index of the F Return: the value of F(index)
D&C solution	Divide: Trivial Conquer: Recursively solve Fibonacci in 2 sub-problems when F(n-1) and F(n-2) Combine: Add the solution of the 2 sub-problems.
Check overlapping	 So, there are overlapping.
Switch D&C to DP Solution [memoization]	1) Define extra storage (we will select dictionary OR array): So, we will define the array of dimension N. 2) Initialize it with NULL (Value we are sure that it can't be a value of F). Here we can choose 0. 3) Solve recursively (top-down): - If not solved (NULL): solve & save. - If solved: retrieve 4) Return solution of main problem.

Pseudocode[memoization]

```
F[0…N] = 0's      //NULL
Fib(n):
    if F[n] ≠ NIL, return F[n]
    if n ≤ 1, return F[n] = 1
return F[n] = Fib(n-1) + Fib(n-2)
```

Storage step

Complexity[memoization]

= # subproblems * time/subproblem

= N * Θ(1) = Θ(N)

Trace[memoization]

Find F(5):

	0	1	2	3	4	5
F	0	0	0	0	0	0

F(5) —Will call→ F(4) —Will call→ F(3) —Will call→ F(2) —Will call→ F(1) (Return F(1) = 1)

F(2) —And will call→ F(0) (Return F(0) = 1)

	0	1	2	3	4	5
F	1	1	0	0	0	0

We will return to F(2) = F(1) + F (2) = 2 and store it:

	0	1	2	3	4	5
F	1	1	2	0	0	0

We will return to F(3) = F(2) + F (1) = 3 and store it:

	0	1	2	3	4	5
F	1	1	2	3	0	0

We will return to F(4) = F(3) + F (2) = 5 and store it:

	0	1	2	3	4	5
F	1	1	2	3	5	0

We will return to F(5) = F(4) + F (3) = 8 and store it:

	0	1	2	3	4	5
F	1	1	2	3	5	8

We call every element recursive only one time and get the other calls from the array.

Switch D&C to DP Solution [building table]

1) Define extra storage (we will select dictionary OR array):
   So, we will define the array of dimension N.
2) Equation to fill-in this table:

$$F[i] = \begin{cases} 1 & \text{if } i \le 1 \\ F(i-1) + F(i-2) & \text{else} \end{cases}$$

3) Solve iteratively (bottom-up)
   - If base case: store it
   - else: calculate it
4) Return solution of main problem.

Pseudocode[building table]

```
Fib(n)
For i=0 to n
   if i ≤ 1, then F[i] = 1
   else, F[i] = F[i-1] + F[i-2]
return F[n]
```

Complexity[building table]

= #iteration * body
= N * Θ(1) = Θ(N)

Trace[building table]

Find F(5):
For i = 0 & i = 1:

	0	1	2	3	4	5
F	1	1	0	0	0	0

i = 2: F(2) = F(1) + F(0) = 2:

	0	1	2	3	4	5
F	1	1	2	0	0	0

i = 3: F(3) = F(2) + F(1) = 3:

	0	1	2	3	4	5
F	1	1	2	3	0	0

i = 4: F(4) = F(3) + F(2) = 5:

	0	1	2	3	4	5
F	1	1	2	3	5	0

i = 5: F(5) = F(4) + F(3) = 8:

	0	1	2	3	4	5
F	1	1	2	3	5	8

Example2: Rod Cutting

- **Given:** A length n and table of prices pi, for i = 1, 2, …, n.
- **Required:** Find the maximum revenue obtainable for rods whose lengths sum to n, computed as the sum of the prices for the individual rods.

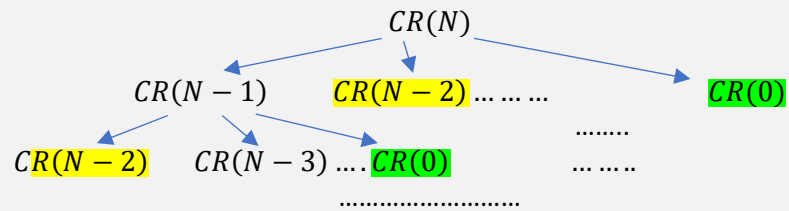
Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

Steps	Solution
Define the problem	Param: length of Rod Return: maximum revenue

D&C solution

1) Guess the possibilities (trials) for the current problem
2) Formulate their subproblems
3) base case: **CR(0) = 0**
4) Relate them in a recursive way

Check overlapping



So, there are overlapping.

Switch D&C to DP Solution

We will select here Button-up [Building Table] because we need to try all the solutions.

1) Define extra storage (we will select dictionary OR array):
   So, we will define the array of dimension N.
2) Equation to fill-in this table:
   CR[n] = 0 if n = 0
   Max(CR[n-1] + p[i]): 1 ≤ i ≤ k else
3) Solve iteratively (bottom-up)
   - If base case: store it
   - else: calculate it
4) Return solution of main problem.

Pseudocode

```
CR(N)
For i=0 to N
    if i=0 then R[i]=0
    else
        m = 0
        for j=1 to k
            m = Max(m, R[i-j] + p[j])
        R[i] = m
return R[N]
```

Complexity

= #iteration * body
= N * Θ(N) = Θ(N^2)

Trace

If the length of Rod is 10:
i: is the length of the piece of Rod, p[i]: is the price of piece
r: is optimal value for rods of length, s: is optimal first cut for rods of length.

i	0	1	2	3	4	5	6	7	8	9	10
p[i]	0	1	5	8	9	10	17	17	20	24	30
r[i]	0	1	5	8	9	10	17	18	22	25	30
s[i]	0	1	2	3	4	2	6	7	7	8	10

Example3: Longest Common Subsequence (LCS)

- **Given:** Two sequences x[1 . . m] and y[1 . . n]
- **Required:** Find a longest subsequence that's common to both of them.
  - Subsequence of a given sequence is the given sequence with zero or more elements left out.
  - Indices are strictly increasing.

Steps	Solution
Define the problem	Param: length of Rod Return: length of longest common subsequence between 2 strings
D&C solution	1) Guess the possibilities (trials) for the current problem 2) Formulate their subproblems 3) base case: **LCS(0, m) = 0, LCS(n, 0) = 0** 4) Relate them in a recursive way **LCS(n-1, m-1) + 1**
Check overlapping	$LCS(n,m)$ $LCS(n,m-1)$ $LCS(n-1,m)$ $LCS(n,m-2)$ $LCS(n-1,m-1)$ $LCS(n-1,m-1)$ $LCS(n-2,m-1)$ ..... So, there are overlapping.
Switch D&C to DP Solution	We will select here Button-up [Building Table] because we don't need to try all the solutions. 1) Define extra storage (we will select dictionary OR array): So, we will define the array of dimension N. 2) Equation to fill-in this table: R[i, j] = { 0 if i = 0 or j = 0; R[i-1,j-1] + 1 if n = 0; Max(R[i,j-1], R[i-1,j]) else } 3) Solve iteratively (bottom-up) - If base case: store it - else: calculate it 4) Return solution of main problem. R[n,m]
Pseudocode	LCS(n,m) for i=0 to n for j=0 to m if i=0 OR j=0, then R[i,j] = 0 else if X[i] = Y[j], then R[i,j] = R[i-1,j-1] + 1 else R[i,j] = Max(R[i,j-1], R[i-1,j]) return R[n,m]
Complexity	= #iteration * body = N * N * Θ(1) = Θ(N^2)

Greedy:

It is an algorithm design approach in which the solution is built iteratively by making locally optimal choices at each step. In other words, at each step, the greedy algorithm makes the best possible choice based on the current information, without considering the overall effect or consequences of that choice.

Characteristics of paradigm:

- It is repeatedly making locally best choice, ignoring effect on future.
- Greedy ≈ D&C paradigm with ONE subproblem: ONE greedy choice + ONE subproblem

Why use it?

It leads to efficient solutions (usually reduce the polynomial of DP)

When to use it?

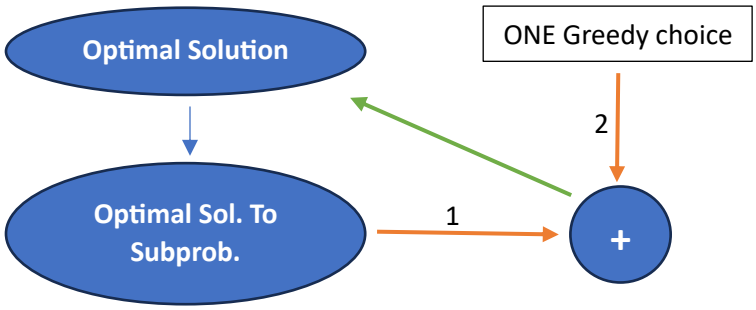
Often in optimization problems, in which there can be many possible solutions and choosing one of them (Greedy) each time leads to final optimal solution.

Conditions:

1. Optimal substructure:
    a. Optimal sol. to problem contains within its optimal solution to ONE sub-problem
    b. i.e., Divide and Conquer
2. Greedy-choice property:
    a. Global optimal can be arrived at by making a local optimal (greedy) choice.
    b. Make the choice that looks best in the current problem, without considering results from sub-problems.

Solution steps:

1. Cast the problem:
    Make a choice ➔ one sub-problem to solve.
2. Prove greedy choice is always SAFE:
    There is always an optimal solution to the original problem that makes the greedy choice
3. Ensure optimal substructure:
    Final optimal = optimal solution to sub-problem + greedy choice.



Compared with D&C:

1. Easy to propose multiple greedy algorithms for many problems.
2. Easy run-time analysis.
3. Hard to establish correctness.

Compared with DP:

	Dynamic	Greedy
Usage	In optimization problem	
Requirements	Optimal substruction (to get the optimal solution we need optimal solution to the sub problems)	
Methodology	Go step by step to reach the final answer.	
Solutions	At each step: • 2 or more solutions. • Every step depends on further step.	At each step: • Only 1 Greedy solution. • Every step may be depending on previous step but don't depend on further step.
Solution Direction	Usually Button up	Usually Top down

Example1: Fractional Knapsack

- **Given:**
  - There are N different items in a store.
  - Item i weighs w_i pounds and is worth $\$v_i$
  - A thief breaks in
  - He can carry up to W pounds in his knapsack
  - Fractions of items can be taken
  - e.g., A thief stealing gold dust
- **Required:**
  - What should he take to maximize his benefit?

Steps	Solution
Cast the problem	We need to make a choice ➔ one sub-problem to solve. 1. Choice: start with ONE from the N items. 2. Sub-problem: Find max benefit to pack the remaining weight from N – 1. So here: 1. Choice: We will choose Max cost per unit from N items. 2. Sub-problem: Max-benefit to pack remain. W from N-1.
Prove greedy choice is always SAFE	By logic: 1. Take as much as possible of the item with the greatest value per unit. 2. This fills each unit of the knapsack with max cost/unit.
Ensure optimal substructure	Final optimal = optimal solution to sub-problem + greedy choice = max benefit to pack remains. weight + item with max c/u

Pseudocode	**FK(size, prices, wights)** Max = **getMaxPerUnit**(prices, wights) **quicksort**(prices, wights) **for**(i = 0 to items) **if**(w[i] ≤ size) Max +=w[i] **else** Max += fraction of w[i] **end FK**
Complexity	O(N log(N))

Example2: Task scheduling

- **Given:**
  - Set of N tasks and each task has processing time P[i].
  - CPU can run 1 task at a time and the tasks run in non-preparation manner.
- **Required:**
  - Schedule these tasks to minimize their average computational time = $\frac{1}{N}\left(\sum_{i=1}^{N} C[i]\right)$

Steps	Solution
Cast the problem	We need to make a choice ➔ one sub-problem to solve. 1. Choice: start with ONE from the N tasks. 2. Sub-problem: Find min processing time to pack the remaining tasks from N – 1 So here: 1. Choice: We will choose the min processing time from N tasks. 2. Sub-problem: Min-time to pack remain. P[i] from N-1
Prove greedy choice is always SAFE	We need to min -> $\frac{1}{N}\left(\sum_{i=1}^{N} C[i]\right)$ C[i] = C[i-1] + P[S[i] : S is the selection choice C[i-1] = C[i-2] + P[S[i-1] …… C[1] = Ø + P[S[1] To min C[1] we need to min P[S[1]] So, we need to choose the min one.
Ensure optimal substructure	Final optimal = optimal solution to sub-problem + greedy choice = Min-time to pack remain + the min P of N task
Pseudocode	**TS(tasks, pTimes)** **quicksort**(pTimes) curTime = totalTime = 0 **for**(i = 0 to tasksSize) curTime **=** curTime + processes[j].processingTime processes[j].completion Time = curTime totalTime += curTime **end Ts**
Complexity	O(N log(N))

Example3: Maximum Sequence Sum - **Given:** array of N numbers. - **Required:** find a sequence of contiguous numbers with max sum.	
Steps	**Solution**
Cast the problem	We need to make a choice ➔ one sub-problem to solve. 1. Choice: start with ONE from the N tasks. 2. Sub-problem: Find min processing time to pack the remaining tasks from N – 1 So here: 1. Choice: a. Start summing the elements from the beginning of the array b. If current sum is positive, then continue (as there's possibility to increase this sum by adding further elements) c. Else if current sum is negative, then reset the sum to 0 and restart from next element (as starting from 0 is better (i.e., give us greater sum) than starting from negative number) 2. Sub-problem: a. Continue the process of calculating current sum on the remaining elements. b. Keep track of the max So Far sum during your process.
Pseudocode	**MSS(list)** maxSoFar = curSum = 0 **for** (i = 1 **to** listSize) curSum += x[i] **if** curSum < 0, **then** curSum = 0 maxSoFar = **max**(maxSoFar, curSum) **end MSS**
Complexity	Θ(N)

Iterative Improvement:

The incremental improvement algorithm design paradigm, also known as the iterative improvement paradigm, involves iteratively refining a solution to a problem by making incremental improvements. The basic idea is to start with an initial solution and repeatedly enhance it until a desired or optimal solution is achieved.

Characteristics:

It's important to note that the specific implementation of the incremental improvement paradigm will vary depending on the problem at hand. The steps and techniques used in one problem may differ from those used in another problem. The key idea is to iteratively improve a solution by evaluating and modifying neighboring solutions until the desired solution quality is achieved.

Solution steps:

1. Start with an initial solution: Begin with an initial solution to the problem. It can be a random solution, a simple heuristic, or any feasible solution.
2. Evaluate the solution: Calculate the objective function or measure the quality of the current solution. This evaluation metric is problem-specific and depends on the problem you are trying to solve.
3. Generate neighboring solutions: Generate a set of neighboring solutions by making small modifications or perturbations to the current solution. The nature of these modifications depends on the problem domain.
4. Evaluate neighboring solutions: Evaluate each neighboring solution using the same objective function or quality measure as in step 2. Compare the neighboring solutions with the current solution to determine if they are better or worse.
5. Select the best neighboring solution: Choose the best neighboring solution that improves upon the current solution according to the evaluation metric. This can involve selecting the solution with the highest objective function value or the lowest cost.
6. Repeat steps 3-5: Repeat steps 3 to 5 until a termination condition is met. The termination condition can be a maximum number of iterations, reaching a specific quality threshold, or any other stopping criterion.
7. Output the final solution: Once the termination condition is met, output the final solution, which is the best solution obtained during the iterative improvement process.

Example1: Travelling Salesman

- **Given:**
  - A salesperson needs to visit a set of cities exactly once and return to the starting city.
  - The cities: {A, B, C, D}
  - The distances: { AB = 10, AC = 15, AD = 20, BC = 25, BD = 30, and CD = 35}
- **Required:**
  - We need to minimize the total distance traveled.

Steps	Solution
Initial Solution	Start with an initial solution, such as a random permutation of the cities.
Evaluate the Solution	Calculate the total distance traveled for the current solution: AB + BC + CD + DA = 10 + 25 + 35 + 20 = 90
Generate Neighboring Solutions	Generate neighboring solutions by swapping the positions of two cities in the current solution: • Swap B and C: A-C-B-D-A. • Swap B and D: A-D-C-B-A. • Swap C and D: A-B-D-C-A.
Evaluate Neighboring Solutions	Calculate the total distance for each neighboring solution: • A-C-B-D-A: AC + CB + BD + DA = 15 + 25 + 30 + 20 = 90. • A-D-C-B-A: AD + DC + CB + BA = 20 + 35 + 25 + 10 = 90. • A-B-D-C-A: AB + BD + DC + CA = 10 + 30 + 35 + 15 = 90.
Select the Best Neighboring Solution	Choose the neighboring solution with the lowest total distance compared to the current solution: All neighboring solutions have the same total distance as the current solution (90).

Repeat	Repeat the steps of generating, evaluating, and selecting until a termination condition is met (reach to a maximum number of iterations or reaching a specific distance threshold).
Output the Final Solution	Return the solution with the lowest total distance obtained during the iterative improvement process: The final solution is A-B-C-D-A with a total distance of 90.
Pseudocode	**TSP(cities)** currentSolution = **initialSolution**(cities) currentDistance = **evaluateSolution**(currentSolution) **while** terminationCondition is not met neighboringSolutions = **generateNeighbors**(currentSolution) **for** i=0 to neighboringSolutions distance = **evaluateSolution**(neighboringSolution) **if** distance < currentDistance, **then** currentSolution = neighboringSolution currentDistance = distance **break** **return** currentSolution **end TSP**
Time Complexity	$O(N^2)$

Example1: Bin Packing

- **Given:**
  - We need to pack a set of items into the minimum number of bins with a fixed capacity.
  - Each item has a specific size.
  - The items are 5 and a bin capacity of 100 units.
  - The sizes of the items: {20, 30, 40, 50, 60}
- **Required:**
  - We need to minimize the number of bins used.

Steps	Solution
Initial Solution	Start with an initial solution, such as packing each item in a separate bin.
Evaluate the Solution	Calculate the number of bins used for the current packing.
Generate Neighboring Solutions	Generate neighboring solutions by moving an item from one bin to another or by swapping items between bins: • Move item 1 from bin 1 to bin 2. • Move item 2 from bin 2 to bin 1. • Swap items 1 and 2 between bins. • Move item 1 from bin 1 to bin 3. • Move item 2 from bin 2 to bin 3

Evaluate Neighboring Solutions	Calculate the number of bins used for each neighboring solution: • Move item 1 to bin 2: 3 bins used. • Move item 2 to bin 1: 4 bins used. • Swap items 1 and 2: 4 bins used. • Move item 1 to bin 3: 4 bins used. • Move item 2 to bin 3: 4 bins used.
Select the Best Neighboring Solution	Choose the neighboring solution with the fewest number of bins compared to the current solution: Moving item 1 to bin 2 results in the fewest number of bins used (3).
Repeat	Repeat the steps of generating, evaluating, and selecting until a termination condition is met (a maximum number of iterations or reaching a specific number of bins).
Output the Final Solution	Return the packing configuration with the minimum number of bins obtained during the iterative improvement process: The final solution is obtained after terminating the algorithm, which has 3 bins used.
Pseudocode	**BinPacking(items, binCapacity)** currentPacking = **initialPacking**(items) currentBinsUsed = **evaluatePacking**(currentPacking) **while** terminationCondition is not met neighboringPackings = generateNeighbors(currentPacking) **for** i=0 **to** neighboringSolutions binsUsed = **evaluatePacking**(neighboringPacking) **if** binsUsed < currentBinsUsed, **then** currentPacking = neighboringPacking currentBinsUsed = binsUsed break return currentPacking **end BinPacking**
Time Complexity	$O(N^2)$