Design and Analysis of Algorithms

Dr. Swaminathan J
Department of Computer Science
Amrita Vishwa Vidyapeetham

Objectives

- To provide a warm-up on algorithms.
- To appreciate the need to design good algorithms.
- Algorithm vs. Program – To know how they differ?
- To measure the running time of iterative algorithm and recursive algorithms by counting operations.

Place your
Webcam Video here
Size 100%

Hello World

Place your
Webcam Video here
Size 38%

- Consider the two programs given below and answer the following questions.

1

```
print("Hello world")
```

2

```
print("Hello ")
print("world")
```

Note down
your answers

1. What is the time taken by the first program to run? — Can't say
2. Will the program take same time on every machine it is run? — No
3. Will the program take same time on every run on same machine? — No
4. Will the first program run faster than the second one? — Not always
5. Do the differences in running times matter at all? — No

Sum of two Integers

Place your
Webcam Video here
Size 38%

- Consider the programs given below.

1
```
sum(a,b) :
    return a+b
```

2
```
sum(a,b) :
    c = a+b
    return c
```

3
```
a = input()
b = input()
print(a+b)
```

Again, note down your answers

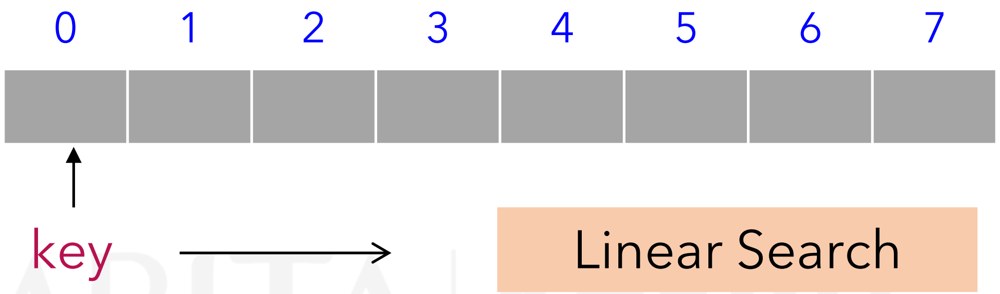
1. Will the first program run faster than the second one? — Not always
2. How about the third program? It takes input from user. — Yes
3. Should user input delays be added to the running time? — Not fair
4. Which program uses more memory among the three? — Program 2?
5. Do the differences in memory usage matter at all? — No

Searching an array

Place your Webcam Video here Size 38%

Given an array A of size n, check if a key is in A.

```
for (i=0; i<n; i++)
    if ( key == A[i] )
        return true
return false
```



- Running time depends on the number of comparisons made.

  (the number of comparisons depends on array size)

Memory used
n+1

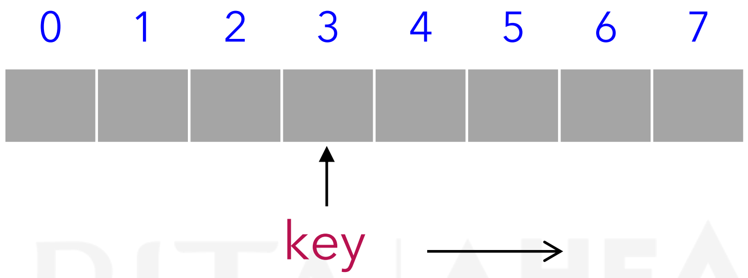
- Best case — 1 comparison — (key = A[0])
- Worst case — n comparisons — (key = A[n] or key not in A)
- Average case — $\frac{n+1}{2}$ comparisons — (key appears in every position with equal probability)

$$\frac{1+2+\ldots+n}{n} = \frac{n(n+1)/2}{n} = \frac{n+1}{2}$$

Searching a sorted array

Given a sorted array A of size n, check if a key is in A.

```
if ( key == A[n/2] )
    return true
if ( key < A[mid] )
    Search the left half
else
    Search the right half
```



Binary Search

Algorithm design matters!

- After each step, search space reduces by half.
  - $n \rightarrow \frac{n}{2} \rightarrow \frac{n}{4} \rightarrow \dots \rightarrow 1$ ($\log_2 n$ comparisons)
- What difference does it make? Say $n = 10^9$.
  - Linear search: 10^9 comparisons (worst case)
  - Binary search: $\log_2 10^9$

$= \log_2 (10^3)^3$

$= 3 \log_2 10^3$

$\approx 3 \log_2 2^{10}$ since $10^3 \approx 2^{10}$

$= 30 \log_2 2$

$= 30$ comparisons (worst case)

Algorithm vs. Program

Place your
Webcam Video here
Size 38%

- An algorithm is a finite sequence of computational steps that transforms an input into an output.

Algorithm	Program
Written in pseudo code	Written in a programming language
Targeted for analysis and reasoning by humans	Targeted for execution on a real machine
Well structured	Need not be well-structured
Includes only what is necessary for analysis	Must be complete and syntactically correct.
Input output instructions are not included	Input output instructions are included

```
Algorithm: LinearSearch
Input:  A[1..n], key
Output: true/false
for i ← 1 to n do
      if key = A[i] then
            return true
return false
```

Running time of Iterative algorithm

- Lets measure the running time of linear search by counting the number of operations.

Algorithm: Search

Input: A[1..n], key

Output: true/false

Algorithm: Search	Running Time: Worst case	Running Time: Best case	
i ← 1	1	1	op (assignment)
while i ≤ n **do**	n+1	1	ops (comparison)
if key = A[i] **then**	2n	2	ops (comparison, indexing)
return *true*	0 / 1	1	op (return)
i ← i + 1	2n	0	op (increment, assignment)
return *false*	1 / 0	0	op (return)
	5n+3	5	ops

Closed form equation

$$T(n) = 5n + 3$$

n is the input size

It is the worst case that is used to characterize any algorithm.

- Best case never.
- Average case occasionally.

All types of operations are assumed to take the same time. In reality, it is not so.

Running time of Recursive algorithm

Place your
Webcam Video here
Size 38%

- Lets measure the running time of linear search by counting the number of operations.

	Algorithm: Search	Running Time: Worst case	Running Time: Best case	
	Input: A[1..n], key			
	Output: true/false			
1.	**if** n = 0 **then**	1	1	op (comparison)
2.	**return** *false*	1	0	op (return)
3.	**if** key = A[n] **then**	2	2	ops (comparison, indexing)
4.	**return** true	1	1	op (return)
5.	else			
6.	**return** Search(A[1..n-1], key)	T(n-1) + 1	0	op (recursion + return)
		T(n-1) + 5	4	ops

Recurrence equation

$$T(n) = \begin{cases} 4 & if\ n = 0 \\ T(n-1) + 5 & otherwise \end{cases}$$

How to solve this recurrence equation?

Solving Recurrences

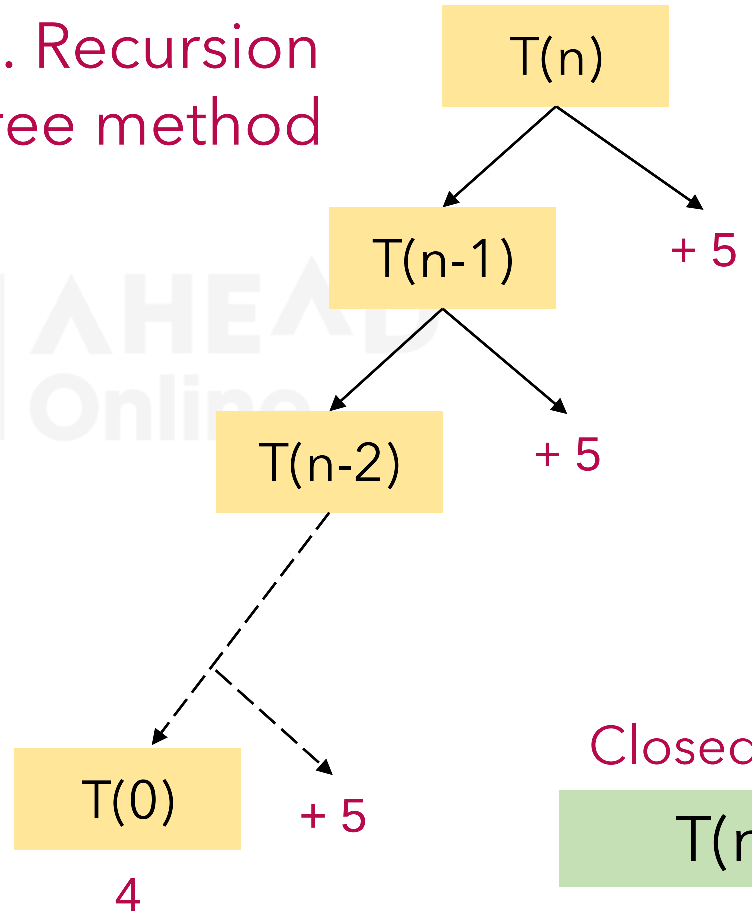
Two methods to solve recurrence equations are given.

$$T(n) = \begin{cases} 4 & if\ n = 0 \\ T(n-1) + 5 & otherwise \end{cases}$$

1. Substitution method

T(n)	= T(n-1) + 5	= T(n-1) + 1*5
	= [T(n-2) + 5] + 5	= T(n-2) + 2*5
	= [T(n-3) + 5] + 2*5	= T(n-3) + 3*5
	= …….	
	= [T(0) + 5] + (n-1)*5	= T(0) + n*5
	= 4 + 5n	= 5n + 4

2. Recursion tree method



Tree height
n

At every level
5 operations
5n

At the last level
4 operations

Closed form equation

T(n) = 5n + 4

Summary

- We showed why design of good algorithms are important.
- We demonstrated how to measure the running time of iterative algorithm and recursive algorithms
by counting operations.

Place your
Webcam Video here
Size 100%

AMRITA VISHWA VIDYAPEETHAM
श्रद्धावान् लभते ज्ञानम्
AHEAD Online

Design and Analysis of Algorithms

Dr. Swaminathan J
Department of Computer Science
Amrita Vishwa Vidyapeetham

Objectives

- To introduce time and space complexity.
- To understand orders of growth in functions.
- To group functions based on their orders of growth.
- To introduce the idea of bounds.
- To introduce asymptotic notations.

Place your
Webcam Video here
Size 100%

Time & Space Complexity

Place your
Webcam Video here
Size 38%

Time complexity refers to the running time of an algorithm.

- It is measured by the number of primitive operations performed by the algorithm as a function of input size.
  - If n is the input size, f(n) denotes the number of operations, Time complexity T(n) = f(n).
- What is its role in algorithm analysis?
  - It allows us to compare two algorithms and choose which is more time efficient.
  - It allows us to classify real-world problems in terms of complexity classes.

Space complexity refers to the number of variables required by the algorithm as a function of input size.

For linear search we noted that T(n) = 5n + 2

Similar to Easy, Medium, Hard.

Space implies memory

Orders of Growth

Place your
Webcam Video here
Size 38%

Given below are the pairs of functions. Determine which among the two grows at a faster rate?

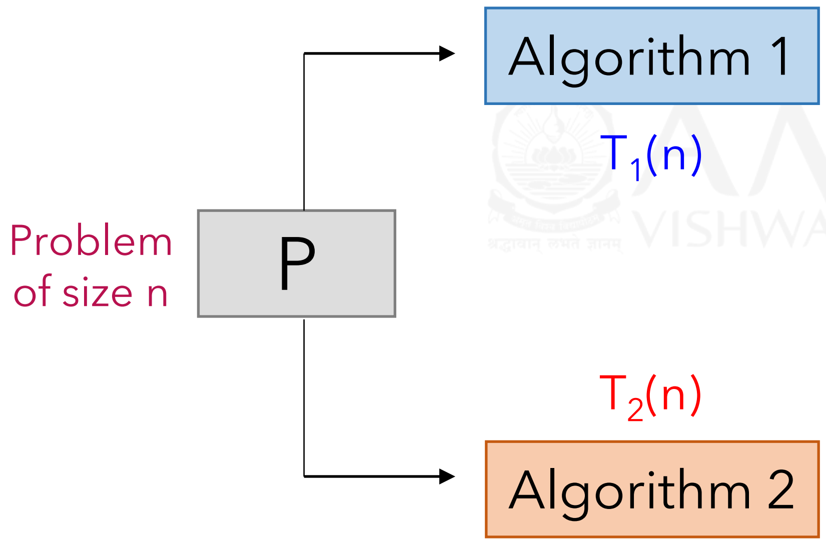
$T_1(n)$	$T_2(n)$	
n	n^2	n^2 grows faster
$1000n$	n^2	n^2 overtakes $1000n$ as n increases
$n(n+1)$	$10n^2$	$n(n+1) = n^2 + n$ … Think about it
$100n^2$	$0.1n^3$	$0.1n^3$ overtakes $100n^2$ as n increases
$\log n$	$10^{-6}n$	$10^{-6}n$ overtakes $\log n$ as n increases
$\log^2 n$	$\log n^2$	since $\log^2 n = (\log n)^2$ and $\log n^2 = 2\log n$
$\log n$	$\sqrt{n}$	$\sqrt{n}$ grows faster than $\log n$
2^n	n^2	2^n grows faster
10^{18}	n	10^{18} is a big constant….. Think about it

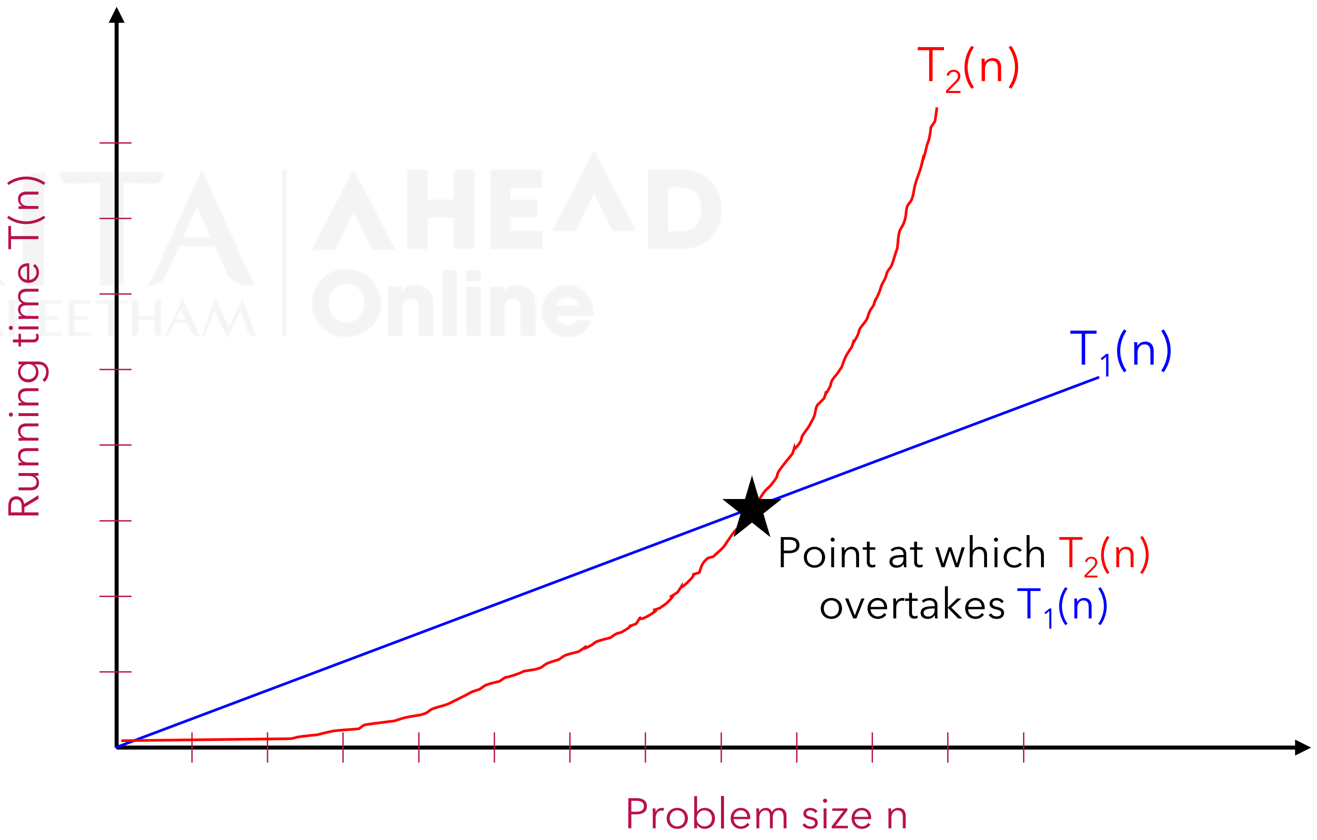
Note down your answers

Orders of growth in Algorithms

Place your Webcam Video here Size 38%

- To be able to compare the running times of different algorithms and choose the better.





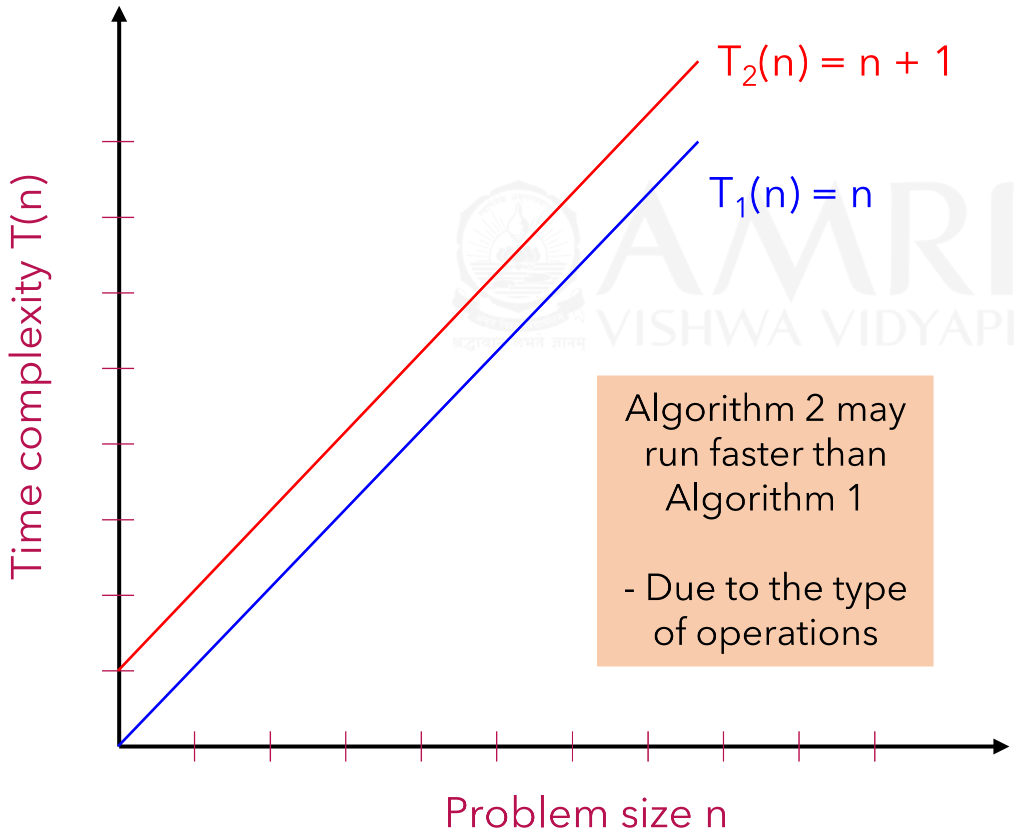
Running time = Number of primitive operations

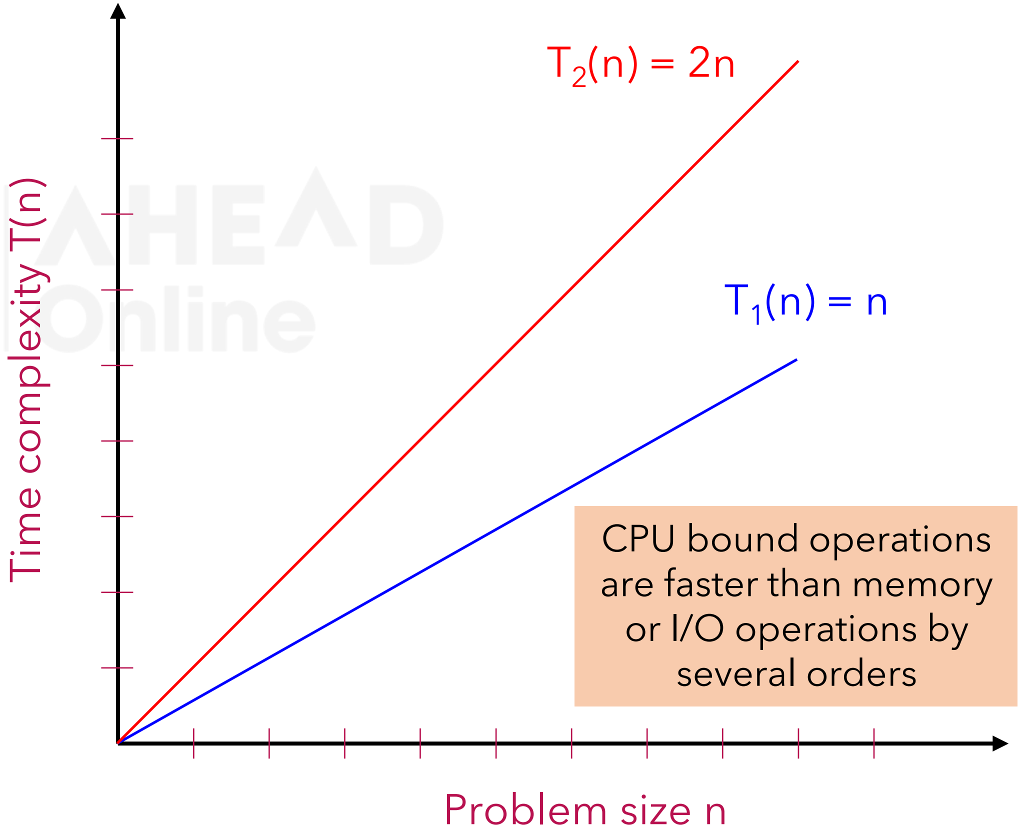
Functions with similar growth

Place your
Webcam Video here
Size 38%

Which among the two algorithms is efficient?

- Discussed using linear functions





Grouping functions based on growth

Place your
Webcam Video here
Size 38%

- Constants don't matter
- Lower order terms don't matter

$\cancel{2}n + \cancel{3}$ vs. $\cancel{5}n - \cancel{7}$

Both are linear

$\cancel{4}n^2 - \cancel{n} + \cancel{3}$ vs. $\cancel{2}n^2 - \cancel{6}$

Both are quadratic

But certainly

2^n grows faster than n^2

How about these pairs?

$3^n > 2^n$ √

$2^{n+1} > 2^n$

$\cancel{2}.2^n$ X

$(n+1)! > n!$

$(n+1)n!$ √

$n^n > \dots > n! \dots > 3^n > 2^n$	$>$	$n^k > \dots > n^3 > n^2 > n > \sqrt{n}$	$>$	$(\log n)^2 > \log n > \log \log n$	$> k$
Exponential		Polynomial		Logarithmic	

The idea of bounds

Place your
Webcam Video here
Size 38%

Lets say there is a medical shop nearby to your house.

- And it is roughly 200 metres away from your home.

If someone knocks at your door and asks you,

- How far is the nearest medical shop from here?

Now, your answer can be one of the following.

Which ones are correct?

- At least 200 m
- At least 180 m
- At least 150 m
- At least 10 m
- At least 1 m

Lower bounds

- About 200 m
- About 180 m
- About 220 m

Tight bounds

- At most 200 m
- At most 220 m
- At most 250 m
- At most 1 km
- At most 5 km

Upper bounds

Walkable

Let's connect it to Algorithms

Place your Webcam Video here Size 38%

We are interested in determining how long would it take for an algorithm to run to compute a solution?

In the example, we would classify distances to several destinations based on whether they are reachable

- ✓ By walk
- ✓ By car
- ✓ By flight
- ✓ By rocket
- Not reachable

For example, reaching a grocery shop, theatre or a medical store belong to walkable class of distances since both are walkable distance away.

Given algorithms to solve several problems, we want to classify them based on their time complexities.

- ✓ Constant
- ✓ Logarithmic
- ✓ Polynomial
  - ✓ Linear, quadratic, cubic, …
- Exponential

Finding max element of an array or searching a key in an array belong to linear class of algorithms since their running times are of form $an + b$.

Asymptotic notations

Place your
Webcam Video here
Size 38%

Let $f(n)$ be the running time (number of operations) of an algorithm where n is the input size.

- We want to characterize the algorithm based on its order of growth.
- Three notations, known as asymptotic notations, are introduced towards this.

Big Omega / Ω(.)

$f(n) \in \Omega(g(n))$ if there exists
- a real constant $c > 0$
- an integer $n_0 \geq 0$

such that
- $f(n) \geq c.g(n)$ for $n \geq n_0$

Big Theta / θ(.)

$f(n) \in \theta(g(n))$ if there exists
- real constants $c_1, c_2 > 0$
- an integer $n_0 \geq 0$

such that
- $c_1.g(n) \leq f(n) \leq c_2.g(n)$ for $n \geq n_0$

Big O / O(.)

$f(n) \in O(g(n))$ if there exists
- a real constant $c > 0$
- an integer $n_0 \geq 0$

such that
- $f(n) \leq c.g(n)$ for $n \geq n_0$

Is $f(n) \in O(g(n))$?

Place your
Webcam Video here
Size 38%

Let's work out some examples with $f(n) = 100n^2$.

1. Is $f(n) \in O(n^2)$?
   - $f(n) = 100n^2$
   - $g(n) = n^2$

Choose c = 101.
$100n^2 \leq 101n^2$ is true for all $n \geq 0$.
i.e. $c = 101$, $n_0 = 0$.

In fact any real value >100 will serve as c.

2. Is $f(n) \in O(n^3)$?
   - $f(n) = 100n^2$
   - $g(n) = n^3$

$c = 101$, $n_0 = 0$ will serve our purpose.

We can also choose c to be 99. $100n^2 \leq 101n^3$ will be true for all $n \geq 2$.
i.e. $c = 99$, $n_0 = 2$.

$f(n) \in O(g(n))$ if there exists
- a real constant $c > 0$
- an integer $n_0 \geq 0$

such that
- $f(n) \leq c.g(n)$ for $n \geq n_0$

3. Is $f(n) \in O(n)$?
   - $f(n) = 100n^2$
   - $g(n) = n$

For any c, however high, $100n^2 \nleq c.n$.
Hence, $f(n) \notin O(n)$.

Understanding $f(n) \in O(g(n))$

Place your
Webcam Video here
Size 38%

- $f(n)$ belongs to the class of functions upper bounded by $g(n)$.
- Since, $g(n)$ grows at least as fast as $f(n)$, $c.g(n)$ overtakes $f(n)$.

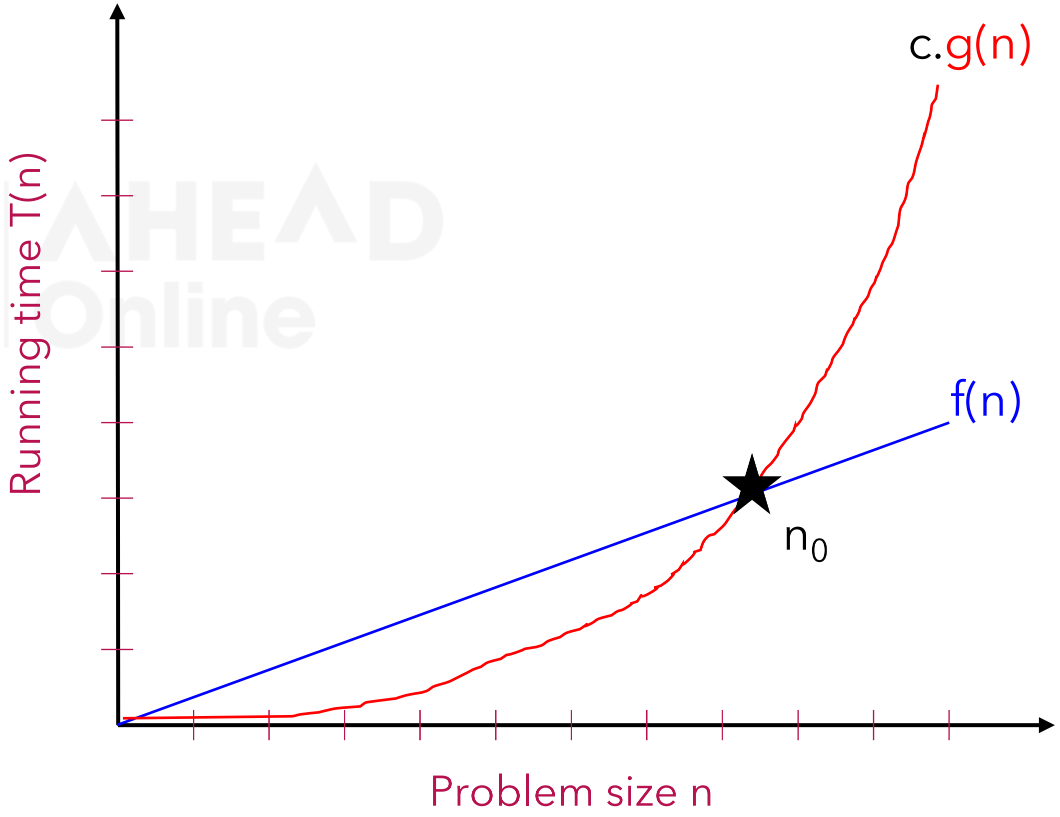
$f(n) = 100n$ and $g(n) = n^2$. Is $f(n) \in O(n^2)$?

To find c & n_0 such that $f(n) \le c.g(n)$ for $n \ge n_0$

Let's take c =15. At n = 7, $15n^2$ overtakes 100n.

n	1	2	3	4	5	6	n_0=7
100n	100	200	300	400	500	600	700
15n^2	15	60	135	240	375	540	735

Therefore, n^2 upper bounds 100n



Note g(n) denotes a class and has no constants or lower order terms.

Understanding $f(n) \in \Omega(g(n))$

Place your
Webcam Video here
Size 38%

- f(n) belongs to the class of functions lower bounded by g(n).
- Since, g(n) grows slower than f(n), c.g(n) cannot keep pace with f(n). i.e. f(n) will overtake c.g(n) whatever be c.

$f(n) = 100n$ and $g(n) = \log n$
Is $f(n) \in \Omega(n^2)$?

To find c and n_0 such that

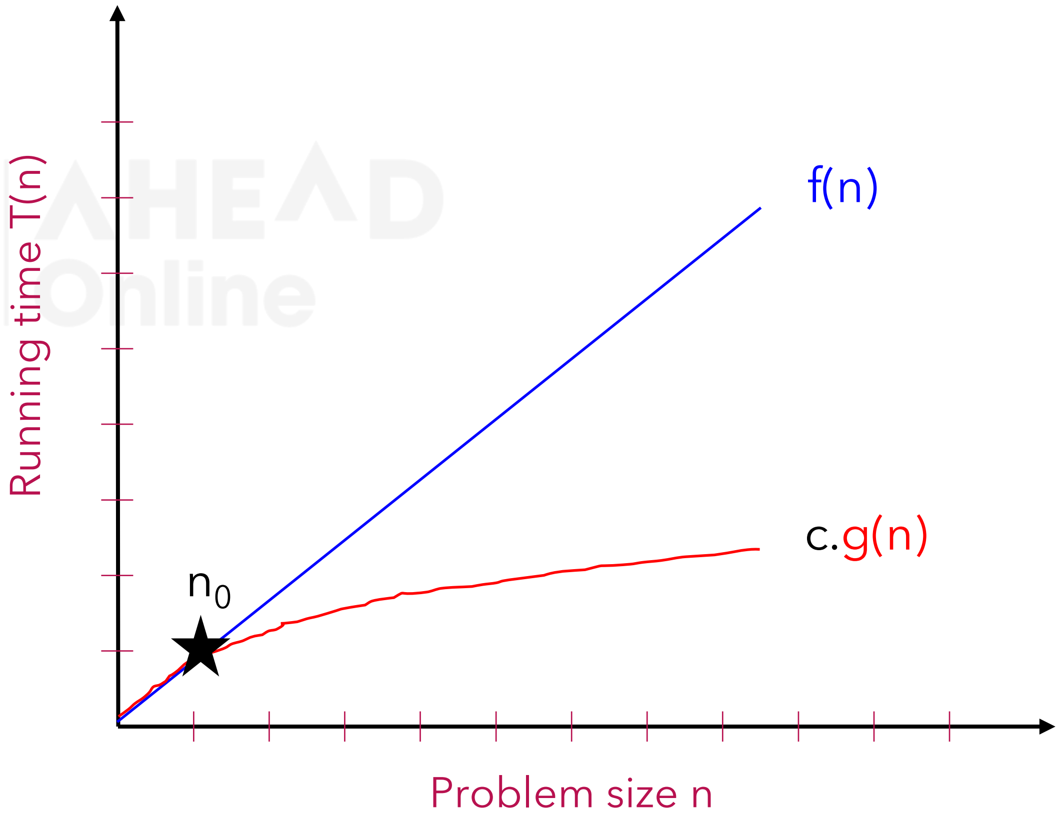
- $f(n) \geq c.g(n)$ for $n \geq n_0$

$$\begin{aligned} f(n) &= 100n \\ &\geq 100 \log n \quad \text{for } n \geq 1 \end{aligned}$$

So, $c = 100$, $n_0 = 1$

$c = 200$, $n_0 = 2$ is a correct too

Hence, log n lower bounds 100n



Understanding $f(n) \in \Theta(g(n))$

Place your Webcam Video here Size 38%

- $f(n)$ belongs to the class of functions that is both upper and lower bounded by $g(n)$.
- Just by changing constants, $g(n)$ can serve as both bounds.

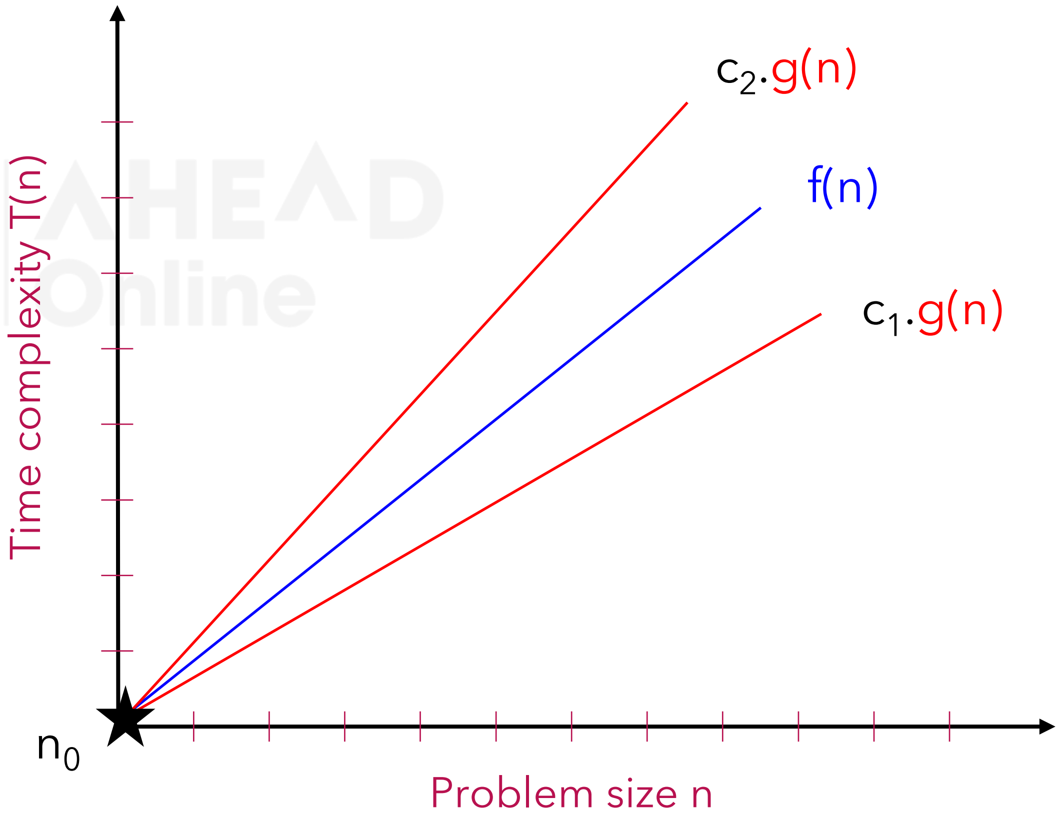
$f(n) = 100n$ and $g(n) = n$
Is $f(n) \in \Theta(n)$?

To find c_1, c_2 and n_0 such that

- $c_1.g(n) \leq f(n) \leq c_2.g(n)$ for $n \geq n_0$

$f(n)$	$= 100n \geq 99n$	for $n \geq 0$
	$= 100n \leq 101n$	for $n \geq 0$
So,	$c_1 = 99$, $c_2 = 101$,	$n_0 = 0$

In fact, any value < 100 fits c_1, >100 fits c_2.

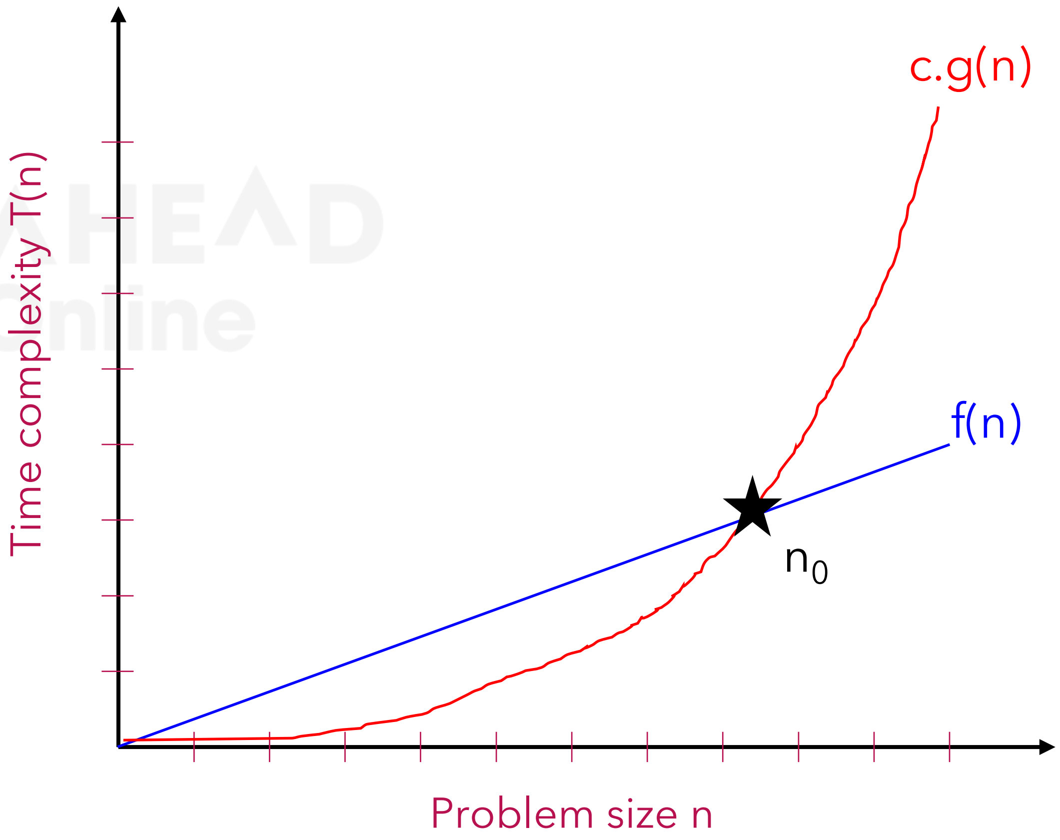


Understanding asymptotic notations

Place your Webcam Video here Size 38%

Let f(n) refer to running time of an algorithm. g(n) refers to a function without constants or lower order terms.

f(n) ∈ O(g(n))	• g(n) upper bounds f(n) for all $n \geq n_0$ • g(n) grows at least as fast as f(n) • c.g(n) overtakes f(n) for some c
f(n) ∈ Ω(g(n))	• g(n) lower bounds f(n) for all $n \geq n_0$ • g(n) grows at most as fast as f(n) • f(n) overtakes c.g(n) for some c
f(n) ∈ θ(g(n))	• g(n) tight bounds f(n) for all $n \geq n_0$ • g(n) grows about as fast as f(n) • By choosing appropriate constants, f(n) can be shown to overtake or overtaken by g(n)



Time complexity & Asymptotic notations

For a given algorithm, we know

- n is the input size.
- f(n) is the time complexity of the algorithm.

$\Omega(g(n))$ refers to the class of all algorithms whose running time f(n) is lower bounded by g(n).

$\Theta(g(n))$ refers to the class of all algorithms whose running time f(n) is tightly bounded by g(n).

$O(g(n))$ refers to the class of all algorithms whose running time f(n) is upper bounded by g(n).

Examples

Place your
Webcam Video here
Size 38%

Let's take $g(n) = n^2$. Consider the three classes below.

$\Omega(n^2)$

- $\cancel{10\sqrt{n}}$
- $\cancel{100n}$
- $\cancel{10^6 n}$
- $0.1n^2$
- $6n^3$
- 3^n
- $\cancel{2n+\log n+7}$
- $\cancel{8\log n+5}$
- $n!$
- $4n^2+7$
- $n(n+1)$
- $\cancel{n^{0.001}}$
- $\cancel{500}$

$\Theta(n^2)$

- $\cancel{10\sqrt{n}}$
- $\cancel{100n}$
- $\cancel{10^6 n}$
- $0.1n^2$
- $\cancel{6n^3}$
- $\cancel{3^n}$
- $\cancel{2n+\log n+7}$
- $\cancel{8\log n+5}$
- $\cancel{n!}$
- $4n^2+7$
- $n(n+1)$
- $\cancel{n^{0.001}}$
- $\cancel{500}$

$O(n^2)$

- $10\sqrt{n}$
- $100n$
- $10^6 n$
- $0.1n^2$
- $\cancel{6n^3}$
- $\cancel{3^n}$
- $2n+\log n+7$
- $8\log n+5$
- $\cancel{n!}$
- $4n^2+7$
- $n(n+1)$
- $n^{0.001}$
- 500

Properties of Asymptotic functions

Place your
Webcam Video here
Size 38%

1. If $f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots.. + a_1 n + a_0$, then
   - $f(n) \in O(n^k)$
   - $f(n) \in \Omega(n^k)$
   - $f(n) \in \Theta(n^k)$

2. If $f(n) \in O(n^k)$, then $f(n) \in O(n^{k+1})$
3. If $f(n) \in \Omega(n^k)$, then $f(n) \in \Omega(n^{k-1})$
4. If $f(n) \in O(n^k)$ and $f(n) \in \Omega(n^k)$, then $f(n) \in \Theta(n^k)$

5. If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$

6. If $f_1(n) \in \Theta(g_1(n))$ and $f_2(n) \in \Theta(g_2(n))$, then
   - $f_1(n) + f_2(n) \in \Theta(\max\{g_1(n), g_2(n)\})$
   - $f_1(n) * f_2(n) \in \Theta\left(g1(n) * g2(n)\right)$

Summary

- We discussed time complexity and asymptotic notations.

Place your
Webcam Video here
Size 100%

Design and Analysis of Algorithms

Dr. Swaminathan J
Department of Computer Science
Amrita Vishwa Vidyapeetham

Objectives

- To introduce the divide and conquer strategy.
- To demonstrate how the strategy can be applied for practical problems such as sorting, searching and selection.

Place your
Webcam Video here
Size 100%

Problems on algorithmic strategies

Place your
Webcam Video here
Size 38%

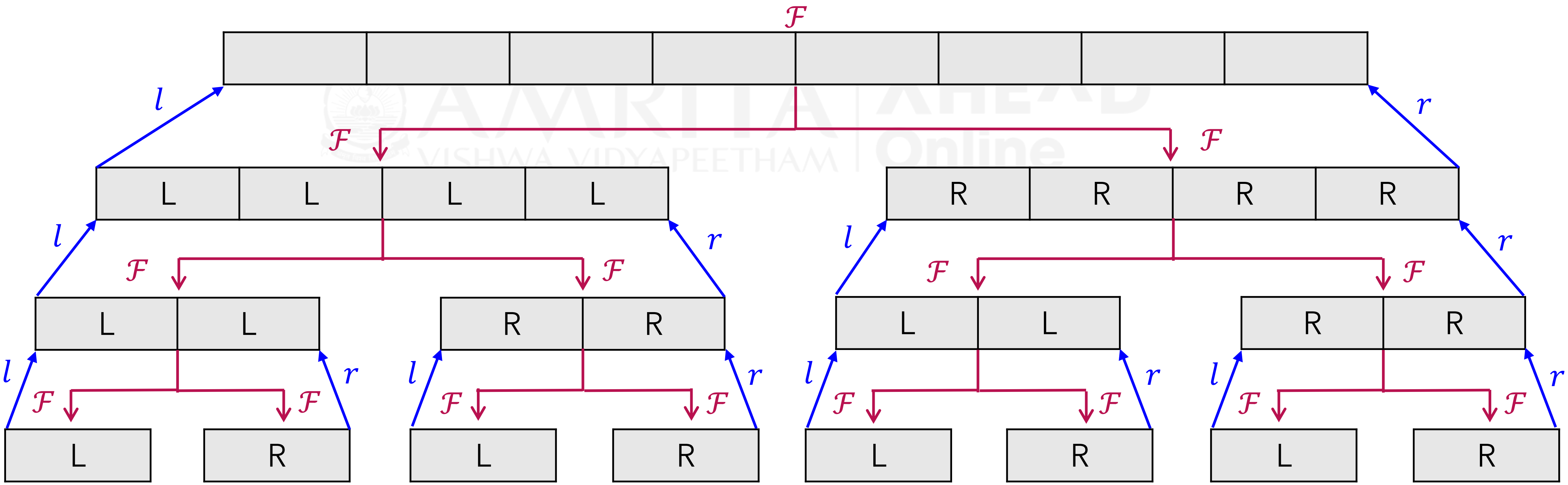
- Divide and Conquer : Binary Search – Merge sort – Quick sort – Multiplication of Large Integers.
- Dynamic programming: Principle of optimality – Coin changing problem, Computing a Binomial Coefficient – Floyd's algorithm – Multi stage graph – Optimal Binary Search Trees – Knapsack Problem and Memory functions.
- Greedy Technique: Container loading – Huffman Trees – Task scheduling – Fractional Knapsack.
- Iterative methods: The Simplex Method – The Maximum-Flow Problem – Maximum Matching in Bipartite Graphs, Stable marriage Problem.
- Backtracking: N-Queens problem – Hamiltonian Circuit Problem – Subset Sum Problem.
- Branch and Bound: – LIFO and FIFO search – Assignment problem – Knapsack Problem – Travelling Salesman Problem
- Approximation Algorithms for NP-Hard Problems – Travelling Salesman problem – Knapsack problem revisited.

Divide-and-Conquer (Data view)

Place your
Webcam Video here
Size 38%

A recursive approach to solving a problem

- Divide the problem into subproblems until small enough
- Solve the subproblems (recurse) and return the solution
- Combine the solutions of the subproblems

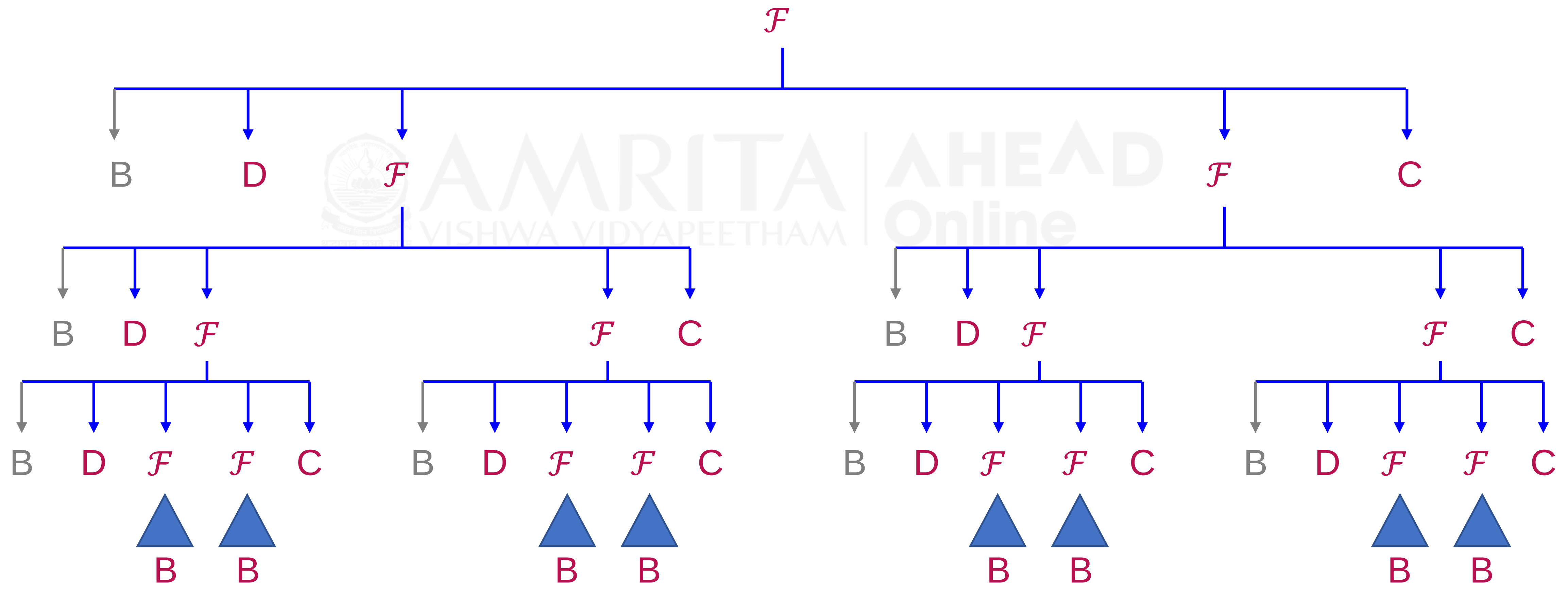


Divide-and-Conquer (Control view)

Place your
Webcam Video here
Size 38%

B ← Base case
D ← Divide step

F ← Recursive step
C ← Combine step

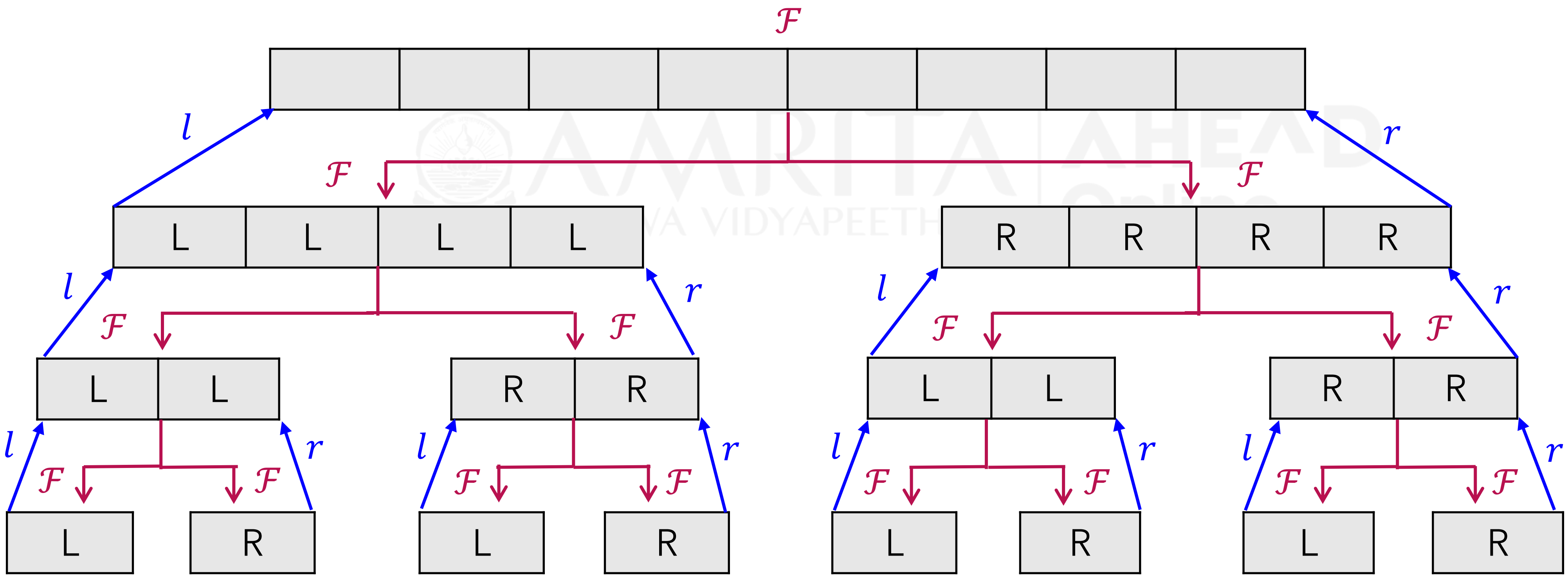


Divide-and-Conquer (Algorithm)

Place your
Webcam Video here
Size 38%

Provides data and control views together.

- Data view in the form of diagram
- Control view in the form of the code



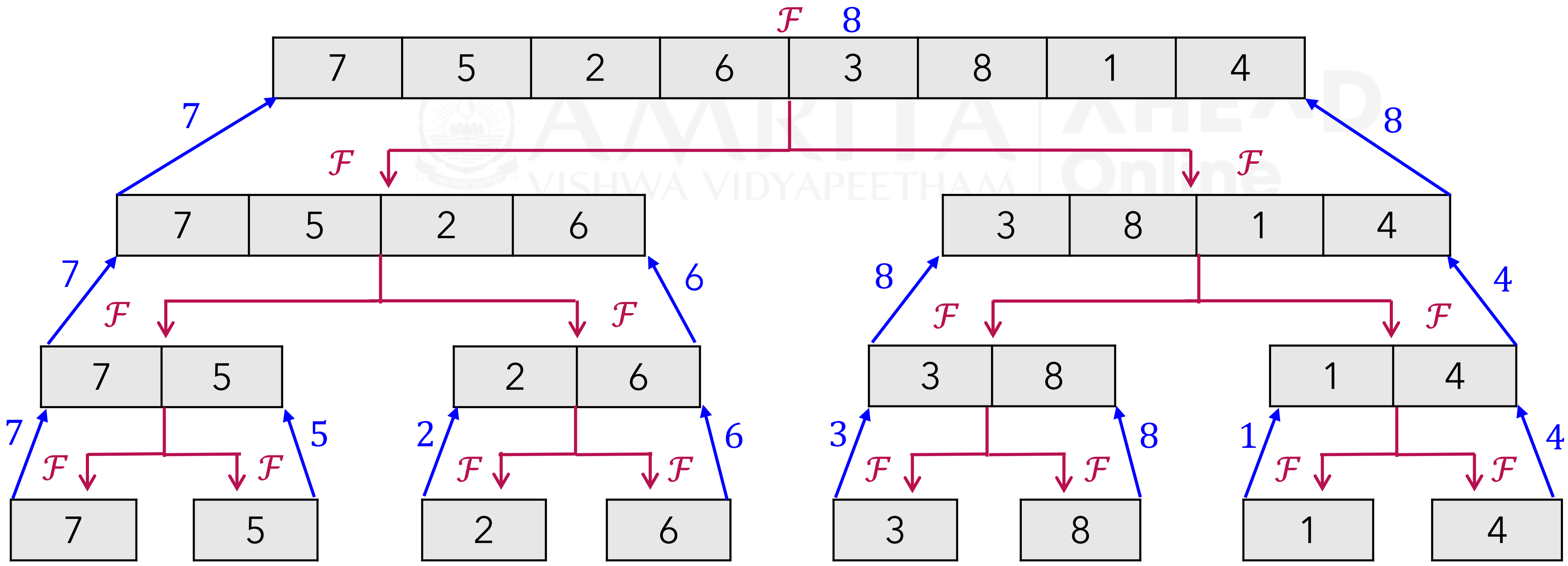

```
Array a[1..n]

F(a, s, e) {
  if (s == e)
    Base case
    return result

  Divide step
  m = (s + e)/2

  l = F(a, s, m)
  r = F(a, m+1, e)

  Combine step
  return result
}
```

1. Finding max of an array

Place your
Webcam Video here
Size 38%

F = FindMax(s, e)

- Here, n = 8. Initially, s = 1 and e = n or (s,e) = (0,n-1).
- Divide: Split the array into two halves.
- Combine: Compare left & right results, return bigger.



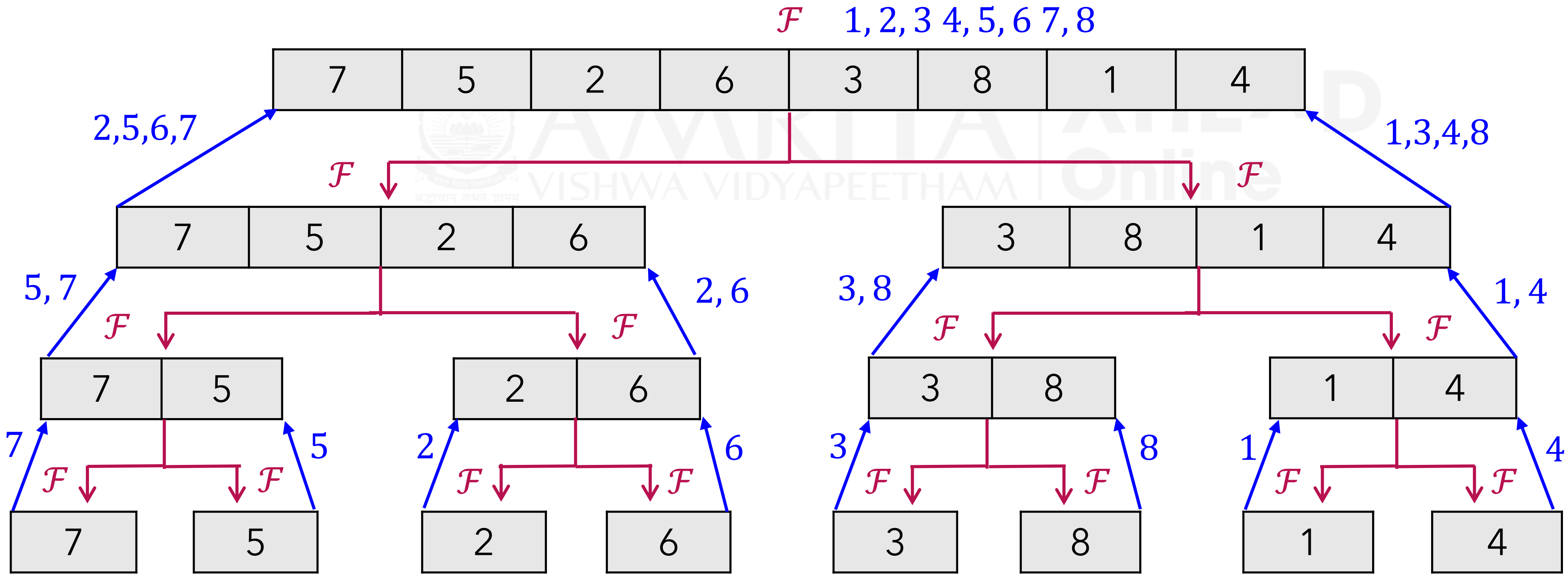

```
Array a[1..n]

F(a, s, e) {
  if (s == e)
    Base case
    return a[s]

  Divide step
  m = (s + e)/2

  l = F(a, s, m)
  r = F(a, m+1, e)

  Combine step
  return l > r ? l : r
}
```

2. Merge sort

Place your
Webcam Video here
Size 38%

F = MergeSort(s, e)

- Here, n = 8. Initially, s = 1 and e = n or (s,e) = (0,n-1).
- Divide: Split the array into two halves.
- Combine: Merge 2 sorted subarrays.




```
Array a[1..n]

F(a, s, e) {
  if (s == e)
    Base case
    return a[s]

  Divide step
  m = (s + e)/2

  l = F(a, s, m)
  r = F(a, m+1, e)

  Combine step
  return merge(l,r)
}
```

The merge step in Merge sort

Place your
Webcam Video here
Size 38%

Given two sorted lists, how many comparisons does it take to merge them into a single sorted list?

Ignoring constants, at most n comparisons at each level

2 5 6 7 1 3 4 8 → 1 2 3 4 5 6 7 8

7 comparisons. In general, $n-1$ comparisons

2 5 6 7 1 3 4 8

5 7 2 6 3 8 1 4

6 comparisons. In general, $(\frac{n}{2}-1)*2$ comparisons

5 7 2 6 3 8 1 4

7 5 2 6 3 8 1 4

4 comparisons $(\frac{n}{4}-1)*4$ comparisons

Solving Recurrence for Merge sort

Place your Webcam Video here Size 38%

Let's use substitution method to solve the recurrence.

$$T(n) = \begin{cases} 1 & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + n & otherwise \end{cases}$$

$T(n) = 2T(\frac{n}{2}) + n \qquad = 2T(\frac{n}{2}) + 1.n$

$= 2\,[2T(\frac{n}{4}) + n/2] + n \qquad = 2^2T(\frac{n}{2^2}) + 2.n$

$= 2\,[2^2T(n/8) + n/4] + 2n \qquad = 2^3T(\frac{n}{2^3}) + 3.n$

$= \ldots\ldots.$

$= 2\,[2^{\log n-1}T(n/2^{\log n}) + \log n.n \qquad = 2^{\log n}T(1) + \log n.n$

$= n + n\log n$

$= n\log n$

Properties of merge sort

Stable, but not in-place

2	5	6	7

1	3	5	8

1	2	3	5	5	6	7	8

Complexity

$$T(n) = \Theta(n\ logn)$$
$$S(n) = \Theta(n)$$

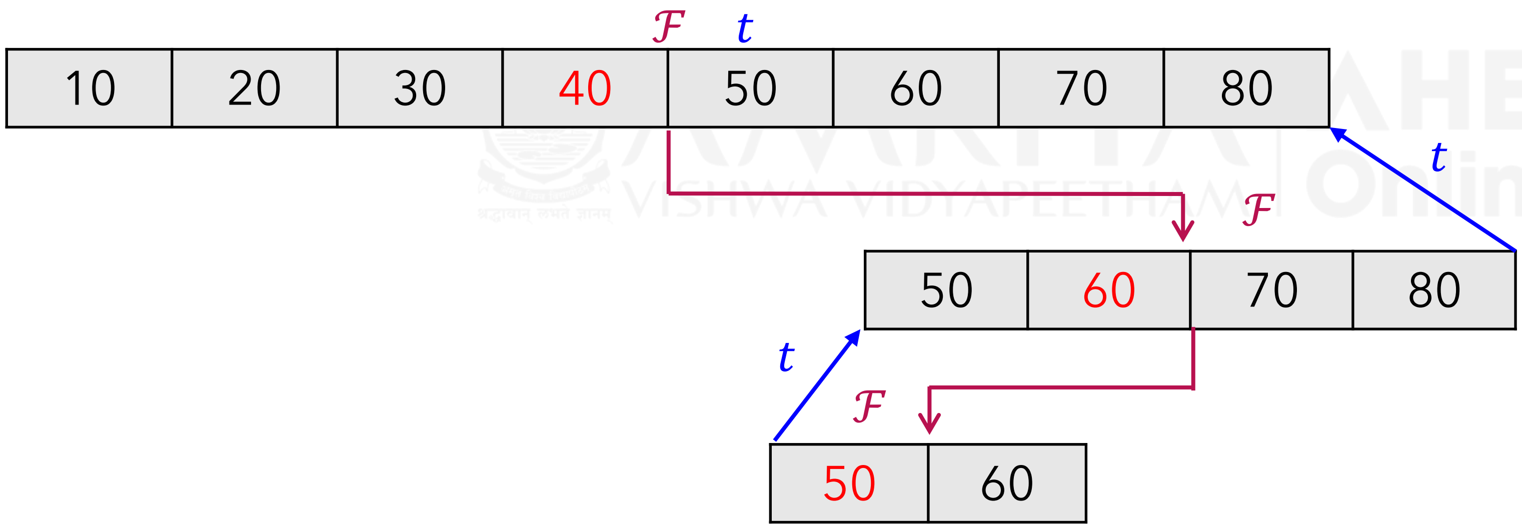
3. Binary Search on Sorted Array

Place your
Webcam Video here
Size 38%

Divide-and-Conquer way to search a sorted array.

F = BinarySearch(a, s, e, k)

- Consider the below example with k = 50.



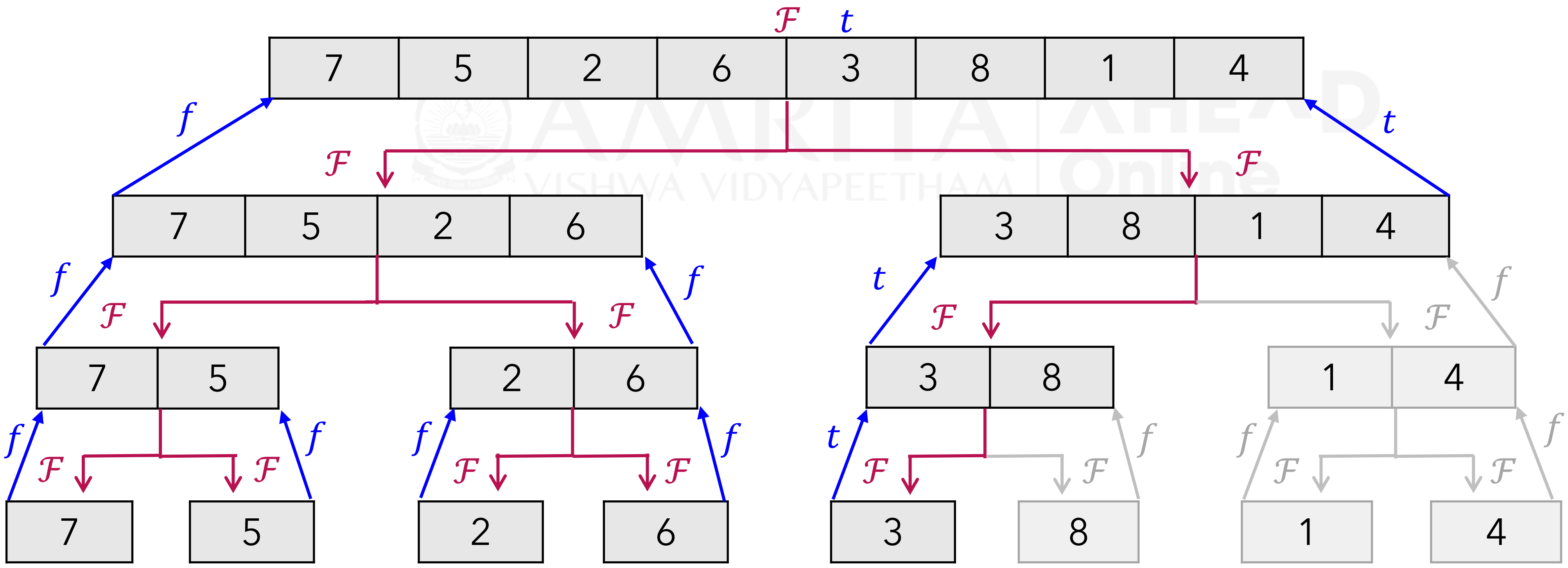

```
Array a[1..n]
F(a, s, e, k) {
  m = (s + e)/2
  if (k == a[m])  // Extra
    return true   // check

  if (s == e)
    return false // Base case

  Divide and recurse
  if (k < a[m])
    return F(a, s, m, k)
  else
    return F(a, m+1, e, k)
}
```

4. Binary Search on Unsorted Array

Place your
Webcam Video here
Size 38%

F = DnCSearch(s, e, k). In the example, let key = 3.

- If found, rest of the search can be abandoned.
- If not, we must compare with every element (worst case).
- So, no advantage due to divide & conquer strategy.



Array a[1..n]

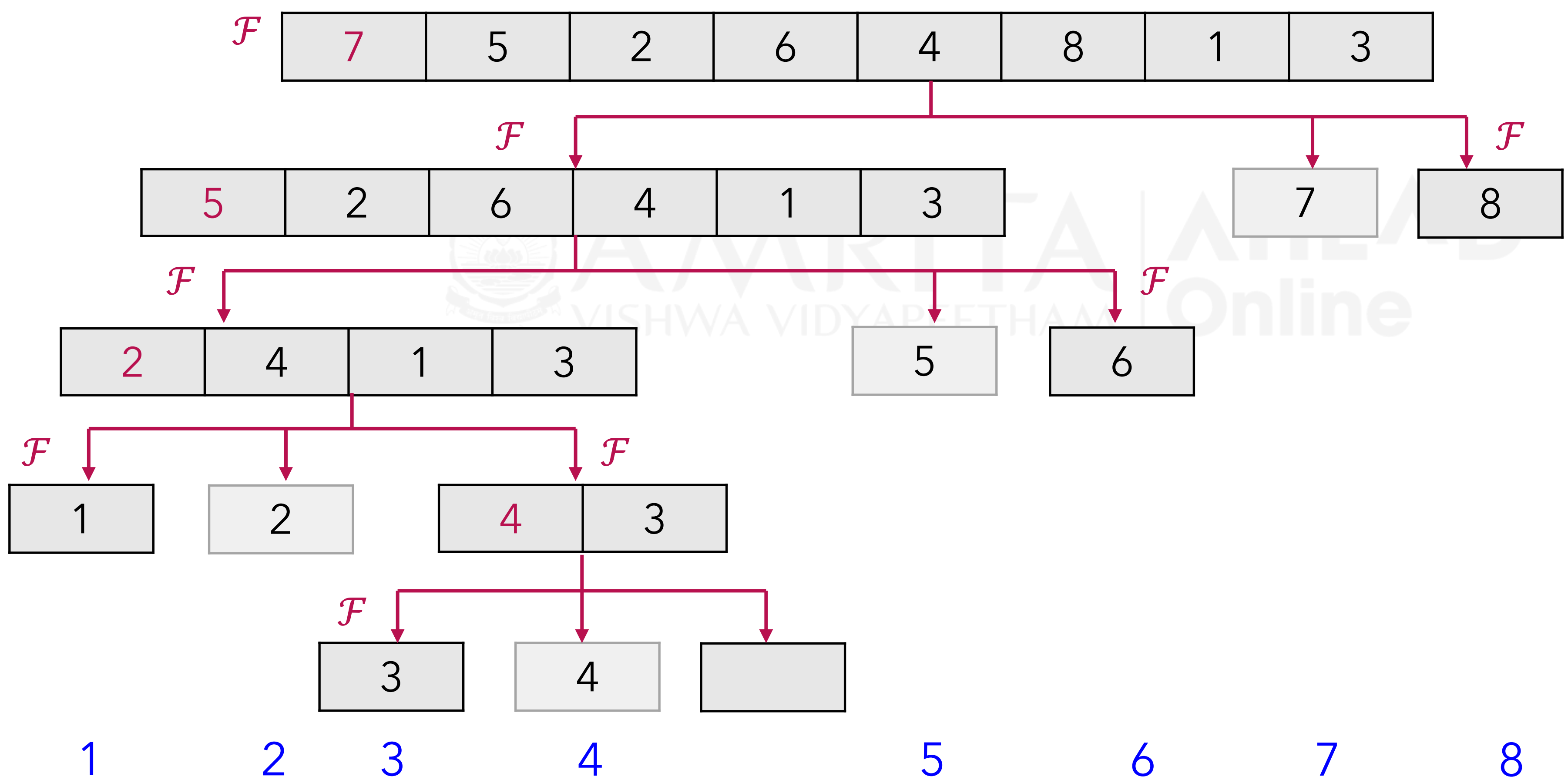
```
F(a, s, e, k) {
  if (s == e ∧ a[s] == k)
    Base case
    return false

  Divide step
  m = (s + e)/2

  l = F(a, s, m, k)
  r = F(a,m+1,e,k)

  Combine step
  return l ∨ r
}
```

5. Quick Sort

Place your
Webcam Video here
Size 38%

F = QuickSort(a, s, e).

- Pick a pivot p. Divide a into { <p }, p, { >= p }



Array a[1..n]
F(a, s, e) {
if (size(a) == 1)
Base case
return a[s]

Divide step
for i ← s to e do
p = a[s]
l = {a[i] ∋ a[i] < p}
r = {a[i] ∋ a[i] ≥ p}

Combine step
return F(l) o p o F(r)
}

Properties of Quick Sort

Place your
Webcam Video here
Size 38%

1. The divide step can leave the left and right arrays to be imbalanced.
   - Worst case: size(l) = n-1, size(r) = 0 or vice-versa.
     - $T(n) = T(n-1) + n \quad \Rightarrow \quad \Theta(n^2)$
   - Best case: size(l) = $\frac{n-1}{2}$, size(r) = $\frac{n-1}{2}$
     - $T(n) \leq 2\,T(\frac{n}{2}) + n \quad \Rightarrow \quad \Theta(n\log n)$
2. Seems like time and space inefficient algorithm.
3. Fastest sorting algorithm in practice.
   - Average case: $\Theta(n\log n)$. Worst case rarely happens.
   - In-place sorting by optimal implementation of inner loop.
4. Space complexity $S(n) = \Theta(n)$.
5. Not a stable sort.

Pivot is max or min at each step.

Pivot is the median at each step.

But not really so!

Rigorous math behind.

We will see next.

Summary

- We looked at Divide-and-conquer approach and how it applies to sorting and searching.

Place your
Webcam Video here
Size 100%

AMRITA VISHWA VIDYAPEETHAM

AHEAD Online

Design and Analysis of Algorithms

Dr. Swaminathan J
Department of Computer Science
Amrita Vishwa Vidyapeetham

Objectives

- To introduce the quick sort algorithm and discuss its properties.
- To take a deeper look at in-place implementation of quick sort.
- To discuss divide-and-conquer approach for quick select and maximum sum subarray problems.
- To show Master's method to determine complexity.

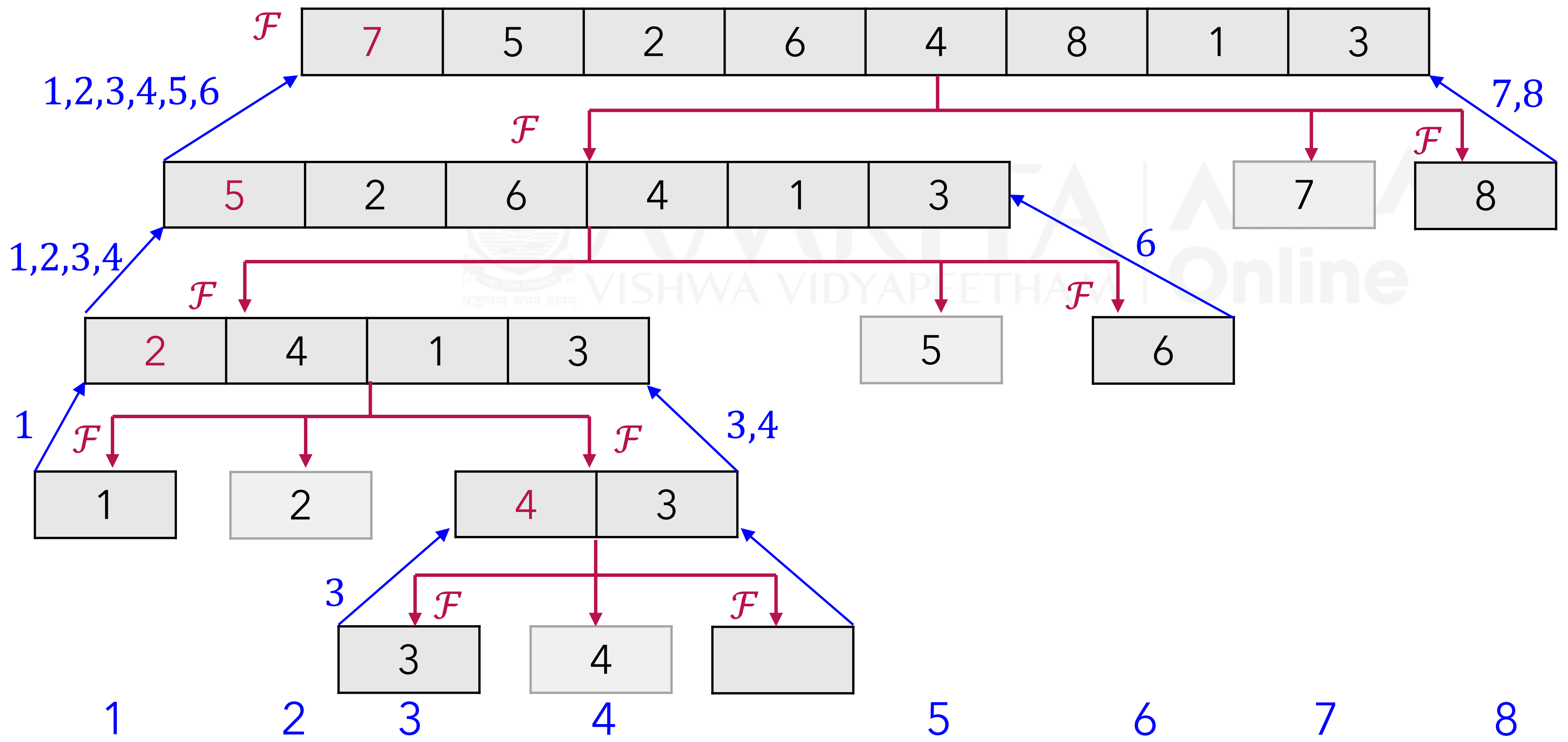
Place your
Webcam Video here
Size 100%

5. Quick Sort

Place your
Webcam Video here
Size 38%

F = QuickSort(a, s, e).

- Pick a pivot p. Divide a into { <p }, p, { >= p }



Array a[1..n]
F(a, s, e) {
if (size(a) == 1)
  Base case
  return a[s]

Divide step
for i ← s to e do
  p = a[s]
  l = {a[i] ∋ a[i] < p}
  r = {a[i] ∋ a[i] ≥ p}

Recurse & combine
return F(l) *o* p *o* F(r)
}

Properties of Quick Sort

Place your
Webcam Video here
Size 38%

1. The divide step can leave the left and right arrays to be imbalanced.
   - Worst case: size(l) = n-1, size(r) = 0 or vice-versa.
     - $T(n) = T(n-1) + n \Rightarrow \Theta(n^2)$

   Pivot is max or min at each step.

   - Best case: size(l) = $\frac{n-1}{2}$, size(r) = $\frac{n-1}{2}$
     - $T(n) \leq 2\,T(\frac{n}{2}) + n \Rightarrow \Theta(n\log n)$

   Pivot is the median at each step.

2. Seems like time and space inefficient algorithm.

   But not really so!

3. Fastest sorting algorithm in practice.
   - Average case: $\Theta(n\log n)$. Worst case rarely happens.
   - In-place sorting by optimal implementation of inner loop.

   Rigorous math behind.

   The divide step.

4. Space complexity $S(n) = \Theta(n)$.
5. Not a stable sort.

   Which repeat element is the pivot?

Time complexity

Place your
Webcam Video here
Size 38%

- Let's examine the worst and best cases.

{1, 2, 3, 4, 5, 6, 7}

{ } 1 {2, 3, 4, 5, 6, 7}

{ } 2 {3, 4, 5, 6, 7}

{ } 3 {4, 5, 6, 7}

{ } 4 {5, 6, 7}

{ } 5 {6, 7}

{ } 6 {7}

Worst case

$$T(n) = T(n-1) + n = \Theta(n^2)$$

Worst case rarely happens. So, average case is considered. O(nlogn).

{1, 2, 3, 4, 5, 6, 7}

{1, 2, 3} 4 {5, 6, 7}

{1} 2 {3} {5} 6 {7}

Best case

$$T(n) = 2T(\tfrac{n-1}{2}) + n \le 2T(\tfrac{n}{2}) + n = O(n\log n)$$

In-place implementation

Place your
Webcam Video here
Size 38%

- As the pointers left and right move towards each other, smaller element at right is swapped with larger one at left.

- left = 2
- left++ → left = 3
- right = 8
- right remains
- swap_pos(3, 8)
- left++, right--
- ➢ left = 4, right = 7
- left++ → left = 5
- left++ → left = 6
- right remains
- swap_pos(6,7)
- left++, right--
- ➢ left = 7, right = 6

pivot

60	30	80	20	50	70	10	40
1	2	3	4	5	6	7	8

60	30	**40**	20	50	70	10	**80**
1	2	3	4	5	6	7	8

60	30	**40**	20	50	**10**	**70**	**40**
1	2	3	4	5	6	7	8

60	30	**40**	20	50	**10**	**70**	**80**
1	2	3	4	5	6	7	8

In-place implementation

Place your
Webcam Video here
Size 38%

- As the pointers left and right move towards each other, smaller element at right is swapped with larger one at left.

```
pivot ← a[1]
left ← 2
right ← n
while left < right  do
   while a[left] < pivot do
      left++
   while a[right] ≥ pivot do
      right--
   swap_pos(left, right)
   left ++ ;    right--
```

$T(n) = S(n) = \Theta(n)$

60	30	80	20	50	70	10	40
1	2	3	4	5	6	7	8

60	30	**40**	20	50	70	10	**80**
1	2	3	4	5	6	7	8

60	30	**40**	20	50	**10**	**70**	**40**
1	2	3	4	5	6	7	8

60	30	**40**	20	50	**10**	**70**	**80**
1	2	3	4	5	6	7	8

Pivot selection and Stability

Place your
Webcam Video here
Size 38%

Better pivot => faster algorithm. Randomized pivot selection is statistically proved to work well.

```
pivot ← a[1]
left ← 2
right ← n
while left < right  do
    while a[left] < pivot do
        left++
    while a[right] ≥ pivot do
        right--
    swap_pos(left, right)
    left ++ ;    right--
```

Not stable

$T(n) = S(n) = \Theta(n)$

60	30	80	20	50	70	10	40
1	2	3	4	5	6	7	8

Median of left, middle and right is a popular pivot choice strategy too. Here, median(60,20,40) = 40.

Question: How will you implement in-place strategy if pivot is not in first position?

Is quicksort stable? No. Because pivot can either be the 1st or 2nd occurrence of a duplicate element.

60	**30**	80	20	50	70	**30**	40

Selection problem – Quick select

Place your
Webcam Video here
Size 38%

Finding the k^{th} ranked element of an array.

- Pivot based strategy works well here.
- Let's say we are interested in finding 6^{th} ranked element.

1. Best case (pivot)

{6, 3, 2, 4, 8, 1, 7, 5}

{3, 2, ~~4~~, 1, 5} 6 {~~8, 7~~}

size(left) = 5
(k-1)

Rank
k

2. In the left

{6, 3, 2, 4, 8, 1, 7, 5}

{3, 2, 4, 1, 5, 6} ~~7~~ ~~{8}~~

size(left) ≥ 6
(≥ k)

select(left, 6)

3. In the right

{6, 3, 2, 4, 8, 1, 7, 5}

~~{2, 1}~~ ~~3~~ {6, 4, 8, 7, 5}

size(left) < 6
(< k)

select(right, 3)
6 – 2 – 1 = 3
rank–size(left)-1=3

A special case of selection problem is finding the median. i.e. k = n/2.

- Randomized quick select takes O(n) time to find the median statistically.

6. Maximum sum subarray

Place your
Webcam Video here
Size 38%

Given an array a[1..n], you need to determine the subarray whose sum is the maximum.

{ -3, 5, -1, 9, -6, 2, -8, 7 }

13 7

$MSS(a[1..n]) = MSS(a[1..\frac{n}{2}])$

Result comes from left

{ -3, 5, -1, 9, -6, -2, 8, 7 }

13 15

$MSS(a[1..n]) = MSS(a[\frac{n}{2}+1..n])$

Result comes from right

{ -3, 5, -1, 9, 6, 2, -8, 7 }

13 8

21

$MSS(a[1..n]) = MSS(a[1..\frac{n}{2}]) + MSS(a[\frac{n}{2}+1..n])$

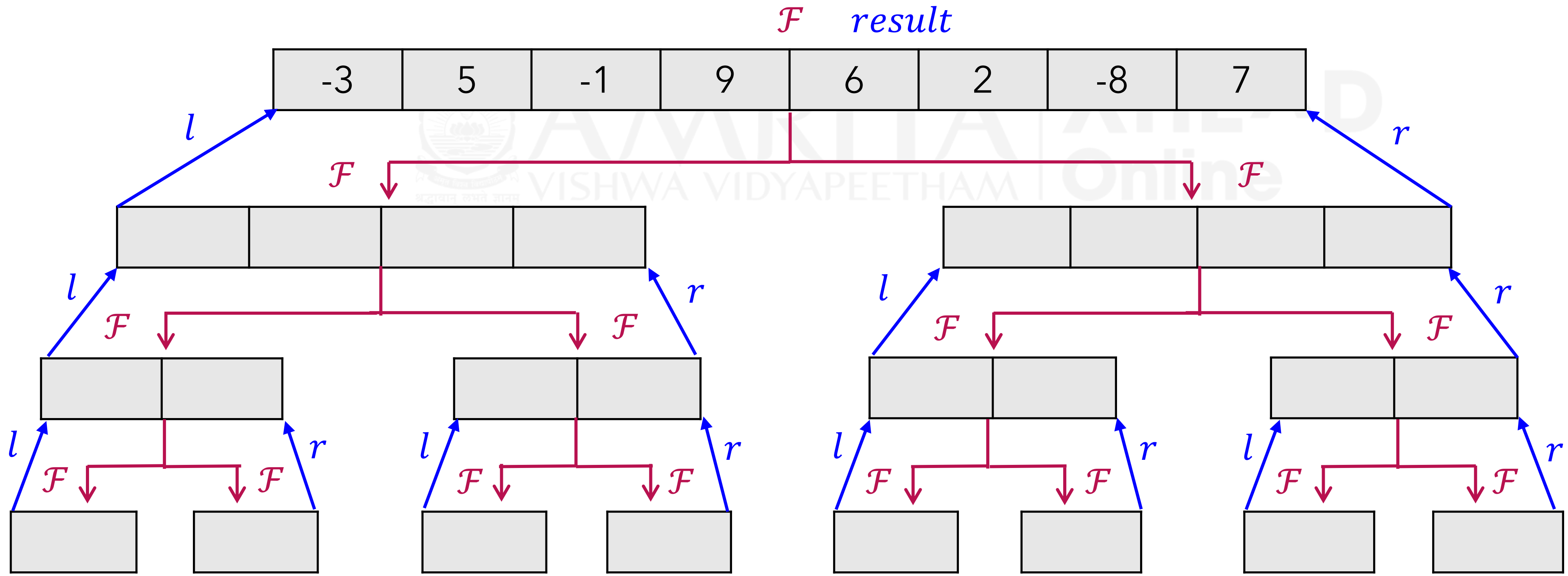
Result is the sum of both left and right

6. Maximum sum subarray

Place your
Webcam Video here
Size 38%

F = MaxSumSubarray(s, e)

- Here, n = 8. Initially, s = 1 and e = n or (s,e) = (0,n-1).
- Divide: Split the array into two halves.
- Combine: Merge 2 sorted subarrays.



Array a[1..n]

```
F(a, s, e) {
  if (s == e)
    Base case
    return a[s]

  Divide step
  m = (s + e)/2

  l = F(a, s, m)
  r = F(a, m+1, e)
  m = ..........

  Combine step
  return max(l,r,m)
}
```

Master's method

Place your Webcam Video here Size 38%

A quick method to determine the time complexity for divide-and-conquer solution.

Consider a generalized recurrence equation: $T(n) = a\,T(\frac{n}{b}) + n^k$

- a denotes the number of subproblems
- b denotes the size of the subproblem
- n^k denotes the effort for divide + combine steps.

1. $n^{\log_b a} > n^k$ — $T(n) = \Theta(n^{\log_b a})$
2. $n^{\log_b a} < n^k$ — $T(n) = \Theta(n^k)$
3. $n^{\log_b a} = n^k$ — $T(n) = \Theta(n^k \log_b a)$

Master's method examples

Place your
Webcam Video here
Size 38%

Three examples are demonstrated below.

$n^{\log_b a} > n^k$	$n^{\log_b a} < n^k$	$n^{\log_b a} = n^k$
$T(n) = \Theta(n^{\log_b a})$	$T(n) = \Theta(n^k)$	$T(n) = \Theta(n^k \log_b a)$

1. $T(n) = 9\,T(\frac{n}{3}) + n$
   - $a = 9, b = 3, k = 1$
   - $\log_b a = \log_3 9 = 2$
   - Here, $\log_b a > k$
   - Hence, $\Theta(n^2)$

2. $T(n) = 3\,T(\frac{n}{4}) + n\sqrt{n}$
   - $a = 3, b = 4, k = 1.5$
   - $\log_b a = \log_4 3 < 1$
   - Here, $\log_b a < k$
   - Hence, $\Theta(n^{1.5})$

3. $T(n) = T(\frac{2n}{3}) + 1$
   - $a = 1, b = 3/2, k = 0$
   - $\log_{3/2} 1 = 0$
   - Here, $\log_b a = k$
   - Hence, $\Theta(\log_{3/2} n)$

Apply this for the algorithms discussed so far.

Summary

- With this we complete the discussion on divide-and-conquer strategy.

Place your
Webcam Video here
Size 100%

AMRITA VISHWA VIDYAPEETHAM

श्रद्धावान् लभते ज्ञानम्

AHEAD Online

Design and Analysis of Algorithms

Dr. Swaminathan J, Ms. Deepa Sreedhar
Department of Computer Science
Amrita Vishwa Vidyapeetham

Objectives

- To look at few optimization problems and recursive formulation of their solutions.

Place your
Webcam Video here
Size 100%

Optimization Problems

Place your
Webcam Video here
Size 38%

In an optimization problem, the goal is to maximize or minimize an objective subject to given constraints.

1. Make an amount with fewest currencies possible.
   - Denominations: ₹1, ₹2, ₹5, ₹10, ₹20, ₹50, ₹100. Make ₹492.

   4 x ₹100 + 1 x ₹50 + 2 x ₹20 + 1 x ₹2 [8]
2. Schedule tasks to minimize average finish time.
   - Task duration T[1..5]: {8, 3, 5, 11, 6}

   T[2], T[3], T[5], T[1], T[4]
3. Finding the longest increasing subsequence.
   - Sequence S[1..12]: 2, 5, -1, 8, 0, 4, 6, 11, 7, 9,12, 10, 14

   -1, 0, 4, 6, 7, 9, 12, 14
4. Finding a subarray that makes the maximum sum.
   - Array A[1..8]: {4, -7, 5, 1, -3, -2, 8, -6}

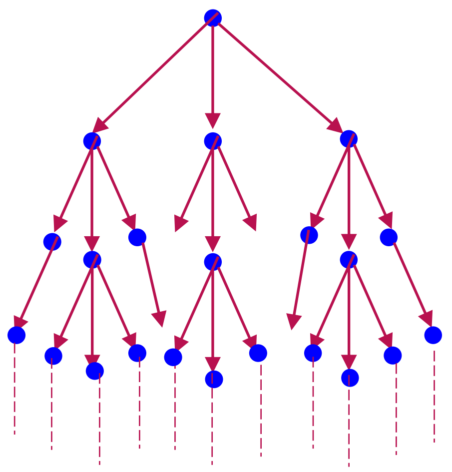
   A[3..7] yields the sum 9
5. Determine the longest common subsequence of 2 strings.
   - Strings: AGGCCTCCAGAATTGA, CCAAACACATATCGAG

   AAAATTGA

Recursive formulation

Place your
Webcam Video here
Size 38%

- An indispensable method to recursively breakdown an optimization problem into subproblem.
  - Let F be the function that solves a problem of size n. The problem is broken down into subproblem of size n-k.
    - F(n) = F(n-k) + extra ops (Usually k = 1)
    - Recursion continues: F(n-k) → F(n-2k) → …… → F(1)
- There may be multiple ways to breakdown.
  - One of them would lead to the optimal solution.
  - You may or may not know which one leads to solution.
- Recursion leads to a tree of choices/possibilities.
  - The tree reveals certain properties which is what we are interested in. They help choose the best strategy to solve.

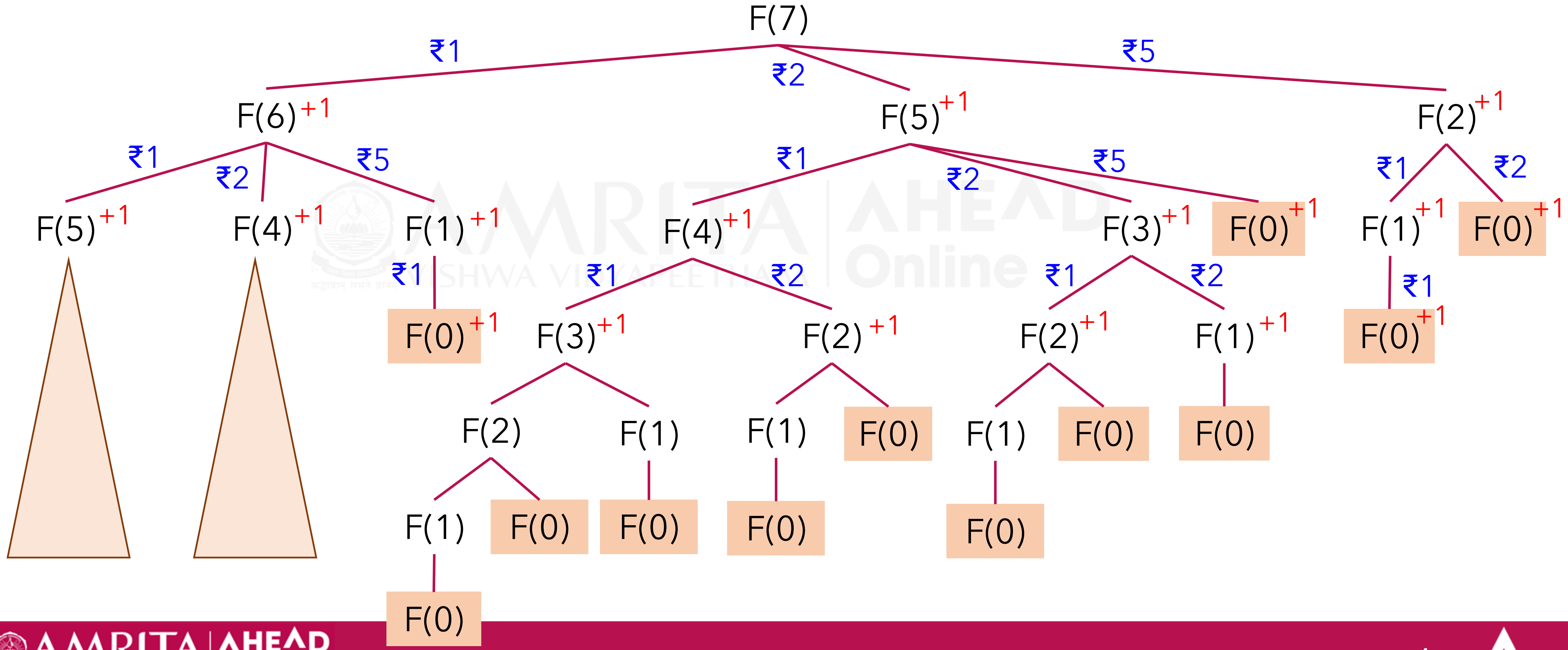


1. Optimal substructure
2. Greedy choice
3. Overlapping subproblems

1. Currency exchange - Example

Place your
Webcam Video here
Size 38%

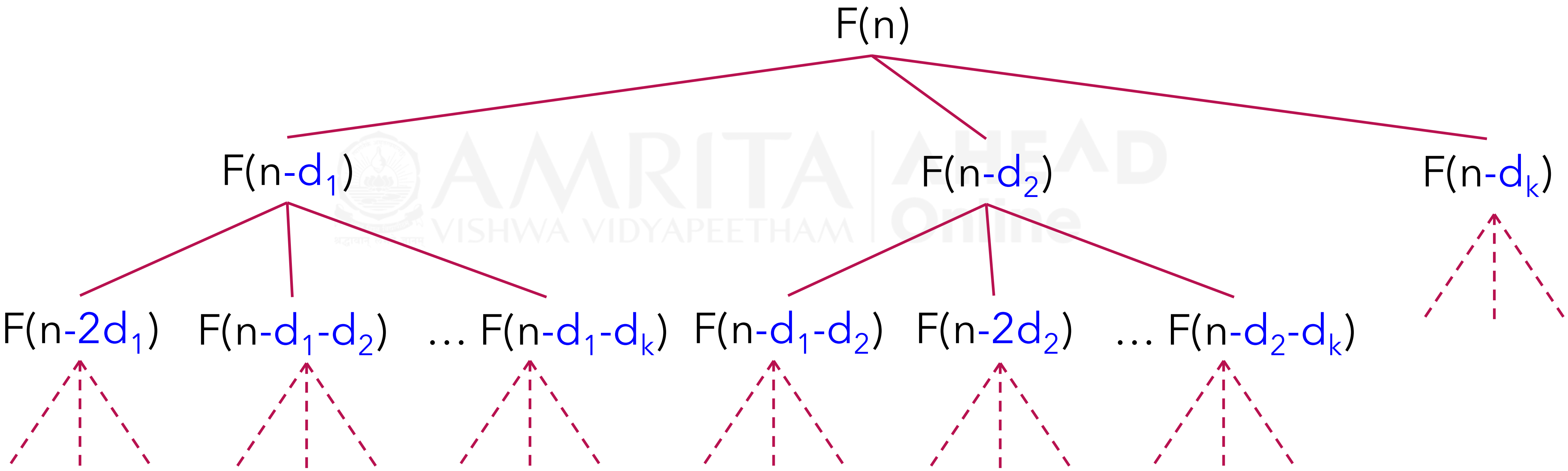
- Denominations: ₹1, ₹2, ₹5. F = MakeChange(7).



1. Currency exchange problem

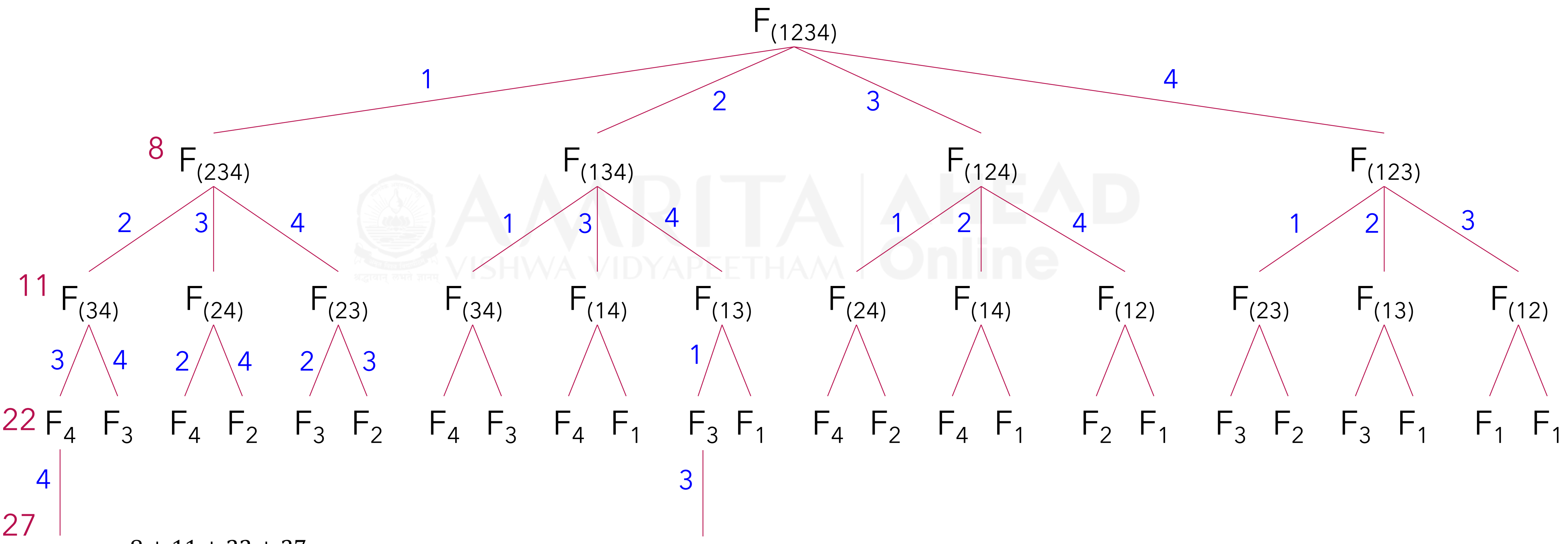
- Denominations: $d_1, d_2, \ldots, d_k$.

F = MakeChange(n)



2. Task scheduling - Example

Task duration: {8, 3, 11, 5}. F = TaskSchedule(1..4).



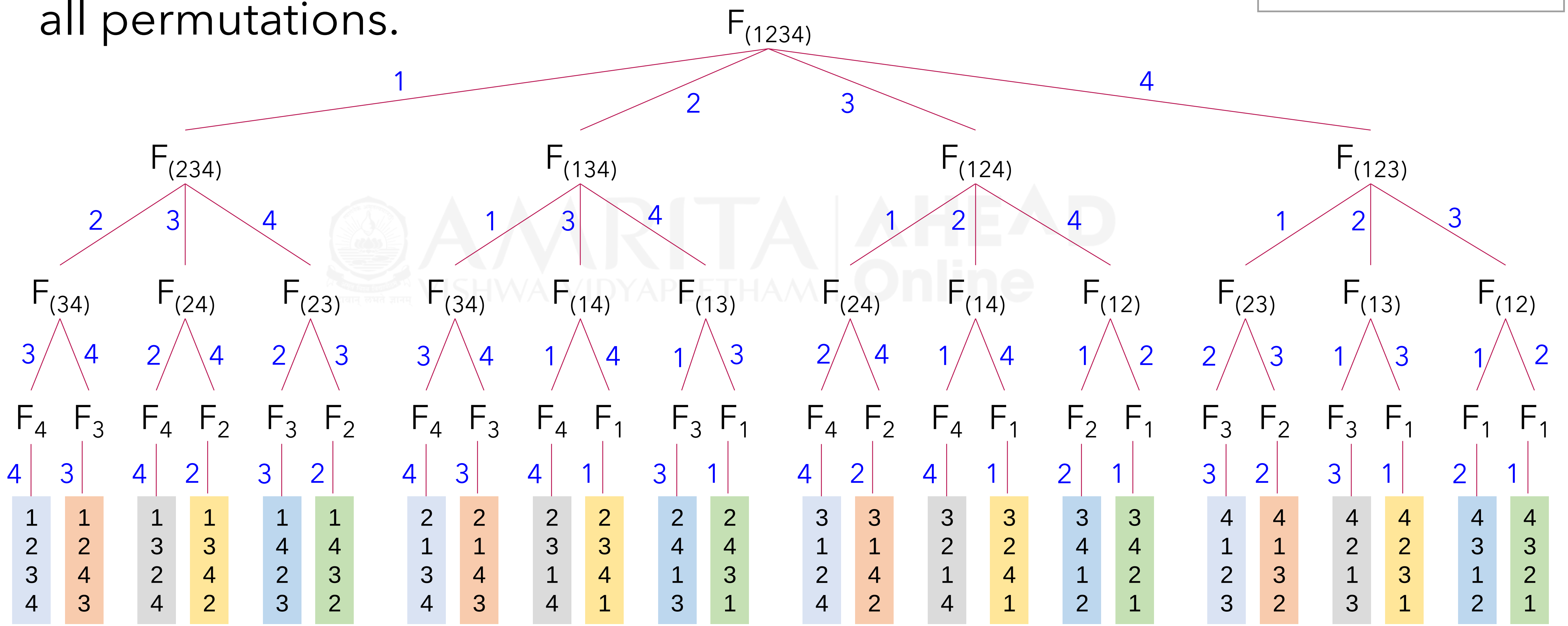
19.5 $\dfrac{8 + 11 + 22 + 27}{4}$

$\dfrac{3 + 8 + 16 + 27}{4} = 13.5$

2. Generating all permutations

Place your
Webcam Video here
Size 38%

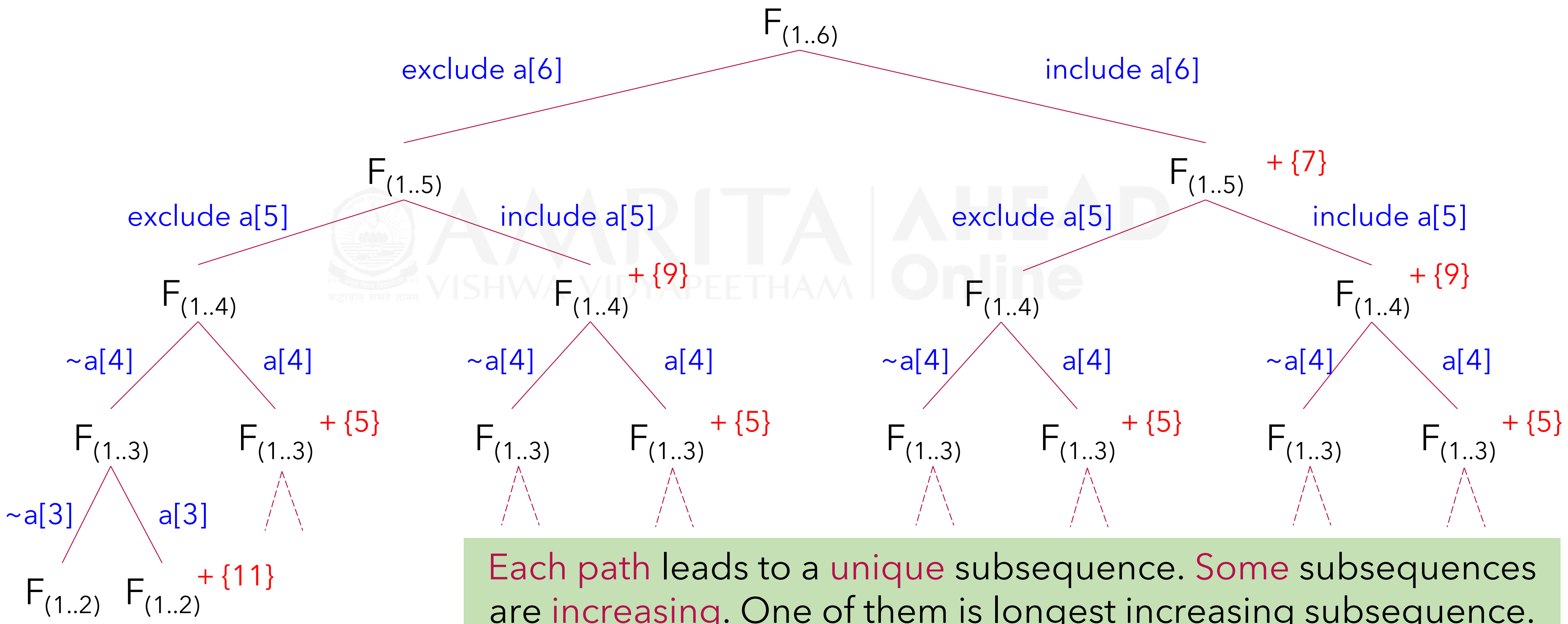
Recursive formulation provides a method to generate all permutations.



3. Longest Increasing Subsequence

Place your
Webcam Video here
Size 38%

Array a[1..6] = {8, 3, 11, 5, 9, 7}. F = LIS(1..6).

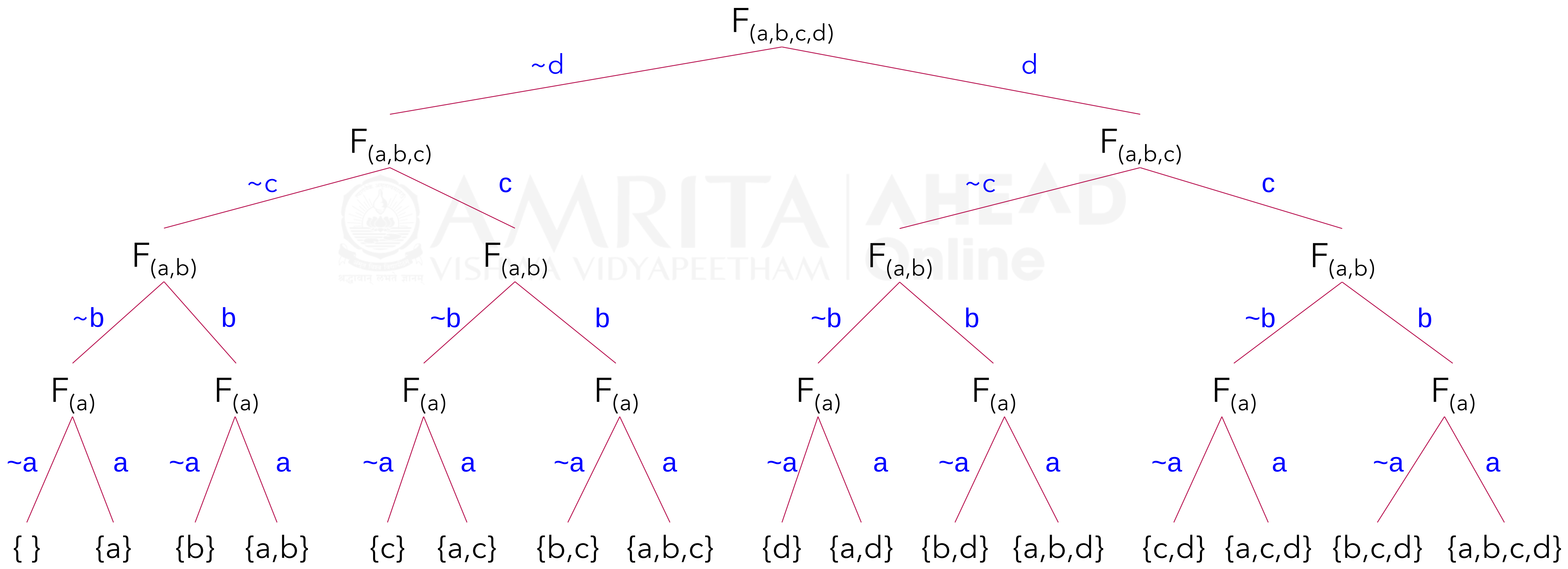


Each path leads to a unique subsequence. Some subsequences are increasing. One of them is longest increasing subsequence.

Generating all sub sequences

Place your
Webcam Video here
Size 38%

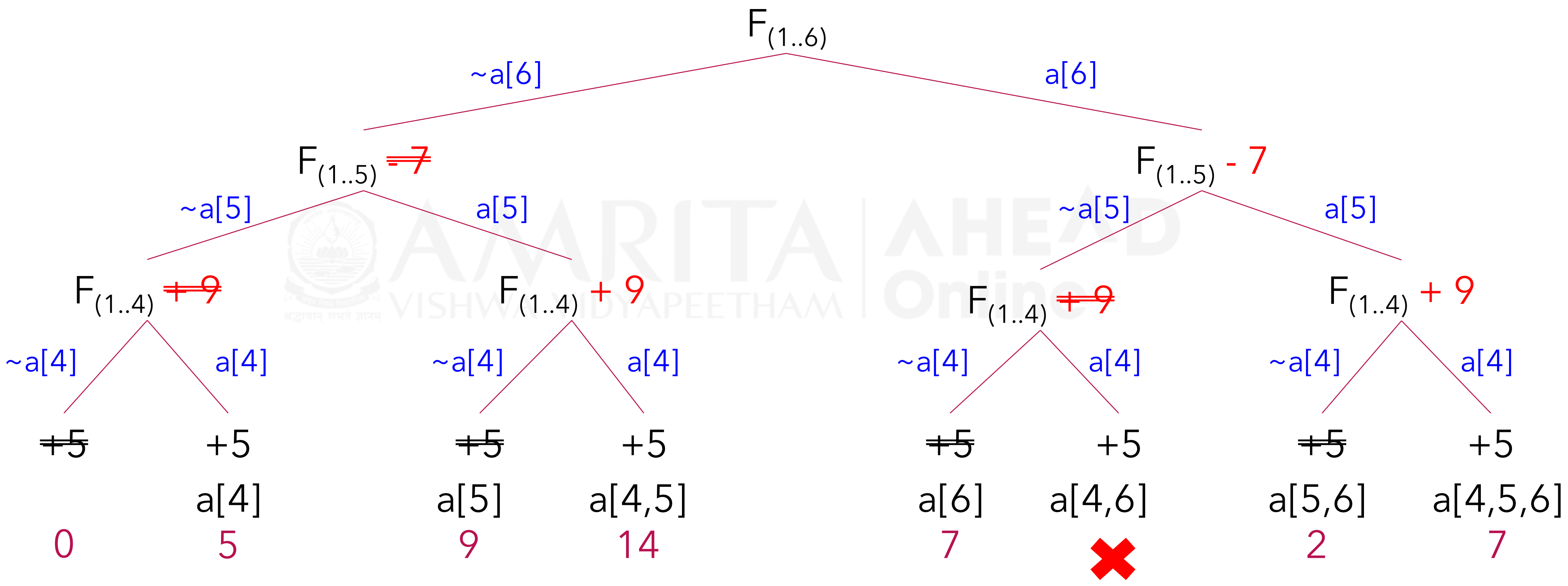
Array a[1..n] = {a, b, c, d}.   F = LIS(1..4).



4. Maximum Sum Subarray

Place your
Webcam Video here
Size 38%

Array a[1..6] = {-8, 3, -11, 5, 9, -7}.   F = MSS(1..6).



5. Longest Common Subsequence

Place your
Webcam Video here
Size 38%

Given two strings S[1..n] and T[1..m], find LCS(S,T). Let's work out few examples.

1. LCS("algae", "league") = "age" or "lge" (length 3)
   - S[n] == T[m] ➔ LCS ("alga", "leagu") + 'e'
2. LCS("algaex", "league") = "age" or "lge" (length 3)
   - S[n] != T[m], but S[n-1] == T[m] ➔ LCS ("algae", "league")
3. LCS("algae", "leaguey") = "age" or "lge" (length 3)
   - S[n] != T[m], but S[n] == T[m-1] ➔ LCS ("algae", "league")

Both S[n] and T[m] are useful

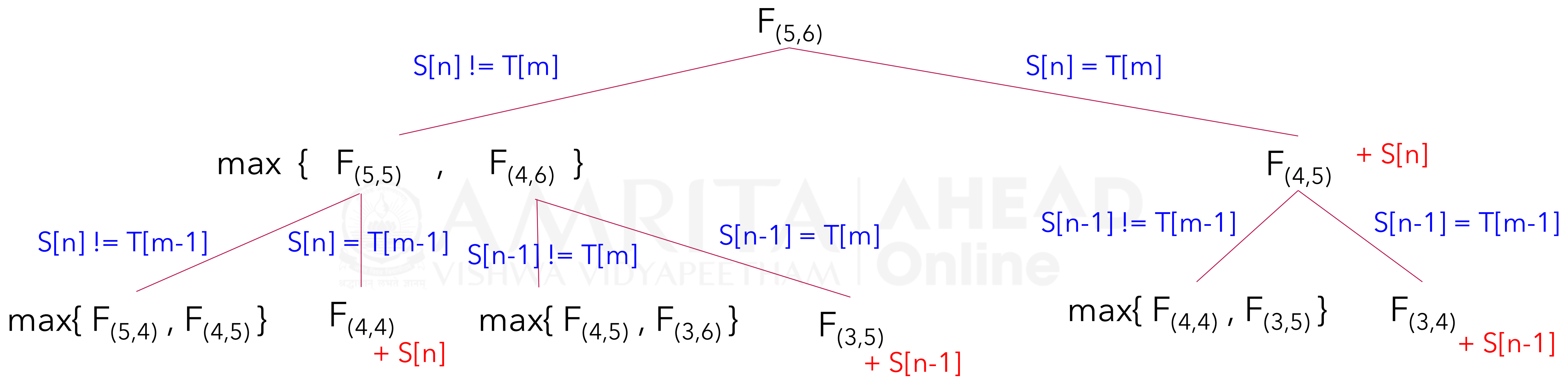
T[m] can be useful

S[n] can be useful

LCS generalized

Place your
Webcam Video here
Size 38%

S[1..n] = “algae”, T[1..m] = “league”.   F = LCS(n,m).



Summary

- We introduced few optimization problems and discussed the recursive formulation of solutions to the problems.

Place your
Webcam Video here
Size 100%

`

AMRITA VISHWA VIDYAPEETHAM

AHEAD Online

Design and Analysis of Algorithms

Dr. Swaminathan J
Department of Computer Science
Amrita Vishwa Vidyapeetham

Objectives

- To critically analyze the recursive formulation.
- To ascertain the properties necessary for applying greedy strategy.
- To discuss problems for which greedy strategy can be applied.

Place your
Webcam Video here
Size 100%

Optimization Problems Recap

Place your
Webcam Video here
Size 38%

In an optimization problem, the goal is to maximize or minimize an objective subject to given constraints.

1. Make an amount with fewest currencies possible.
   - Denominations: ₹1, ₹2, ₹5, ₹10, ₹20, ₹50, ₹100. Make ₹492.

   4 x ₹100 + 1 x ₹50 + 2 x ₹20 + 1 x ₹2 [8]
2. Schedule tasks to minimize average finish time.
   - Task duration T[1..5]: {8, 3, 5, 11, 6}

   T[2], T[3], T[5], T[1], T[4]
3. Finding the longest increasing subsequence.
   - Sequence S[1..12]: 2, 5, -1, 8, 0, 4, 6, 11, 7, 9,12, 10, 14

   -1, 0, 4, 6, 7, 9, 12, 14
4. Finding a subarray that makes the maximum sum.
   - Array A[1..8]: {4, -7, 5, 1, -3, -2, 8, -6}

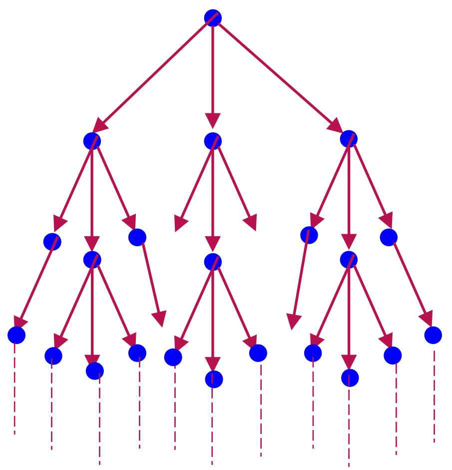
   A[3..7] yields the sum 9
5. Determine the longest common subsequence of 2 strings.
   - Strings: AGGCCTCCAGAATTGA, CCAAACACATATCGAG

   AAAATTGA

Recursive formulation Recap

Place your
Webcam Video here
Size 38%

- An indispensable method to recursively breakdown an optimization problem into subproblem.
  - Let F be the function that solves a problem of size n. The problem is broken down into subproblem of size n-k.
    - F(n) = F(n-k) + extra ops (Usually k = 1)
    - Recursion continues: F(n-k) → F(n-2k) → ...... → F(1)
- There may be multiple ways to breakdown.
  - One of them would lead to the optimal solution.
  - You may or may not know which one leads to solution.
- Recursion leads to a tree of choices/possibilities.
  - The tree reveals certain properties which is what we are interested in. They help choose the best strategy to solve.



1. Optimal substructure
2. Greedy choice
3. Overlapping subproblems

The 3 properties

- You have a recursive formulation of a problem which contains layers of subproblems.
- You look for the following properties.
  1. Optimal Substructure: The optimal solution to a problem contains within itself the optimal solution to its subproblems.
  2. Greedy choice: Picking locally optimal choice all the way leads to globally optimal solution.
  3. Overlapping subproblems: Breaking doing a problem leads to repeated subproblems.

1 & 2 holds → Apply greedy technique

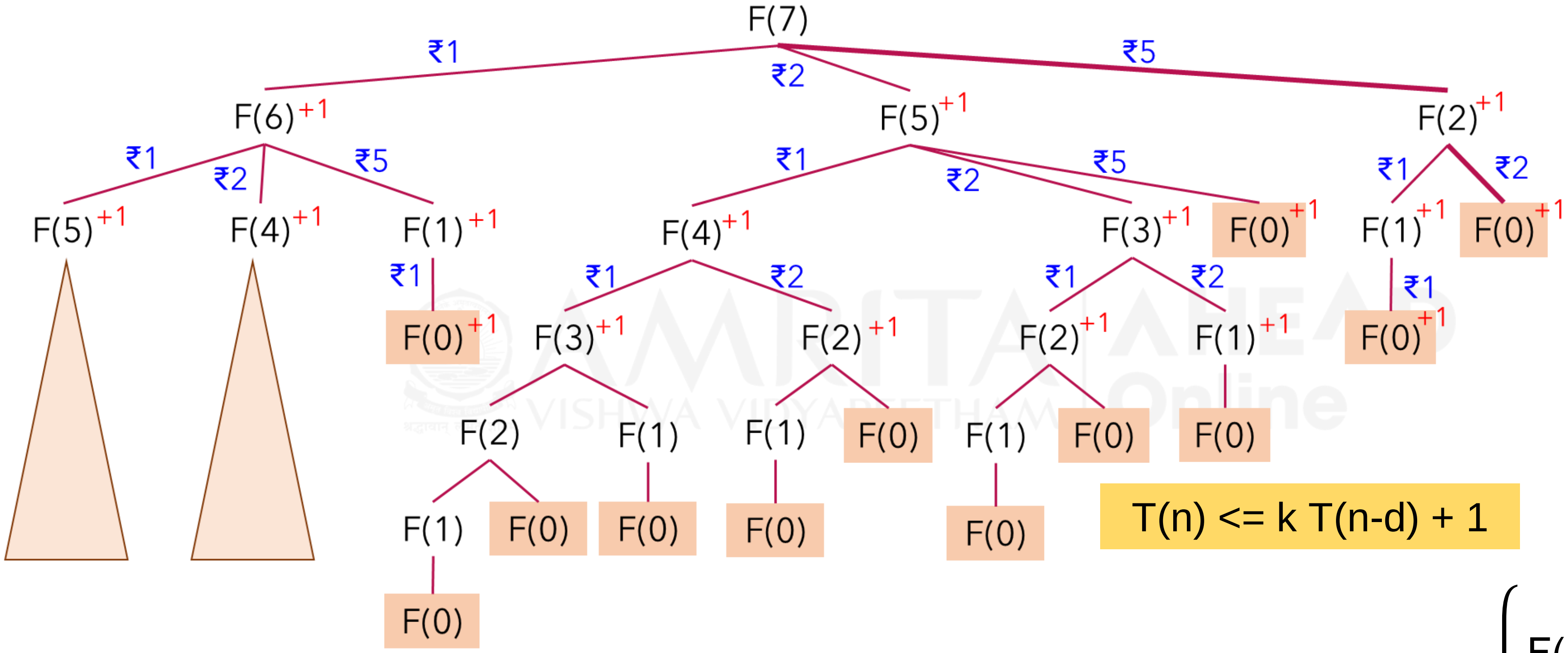
1 & 3 holds → Apply dynamic programming

Let's now look for these on the problems we introduced.

1. Currency exchange

Place your
Webcam Video here
Size 38%

Let's revisit the example.



- ✓ Optimal substructure
- ✓ Greedy choice
- ✓ Overlapping subproblems

T(n) <= k T(n-d) + 1

$$F(n) = \min \begin{cases} 0 & n = 0 \\ F(n - d_1) + 1 & n > d_1 \\ F(n - d_2) + 1 & n > d_2 \\ \dots & \\ F(n - dk) + 1 & n > d_k \end{cases}$$

Denominations: $d_1, \dots, d_k$
F(n) = MakeChange(n)

Minimize $\sum_{i=1}^{k} c_i$ such that $\sum_{i=1}^{k} c_i * di = n$

Greedy Algorithm

Place your
Webcam Video here
Size 38%

Implementation of greedy algorithm.

Denominations: $d_1, \ldots, d_k$
F(n) = MakeChange(n)

Minimize $\sum_{i=1}^{k} c_i$ such that $\sum_{i=1}^{k} c_i * di$ = n

Algorithm: MakeChange
Input: n, d[1..k]
Output: c[1..k]
sort(d[1..k])
c[1] ← c[2] ← … ← c[k] ← 0
denom ← k
while denom > 0 do
    c[denom] ← n / d[denom]
    n ← n mod d[denom]
    denom ← denom – 1
return c[1..k]

T(n) = klogk + k = O(klogk)

The main effort is to sort the denominations.

Maxheap can also be used.

Does greedy choice property hold?

1   15   20

MakeChange(30)

No!

Greedy can result in a suboptimal solution.

Variants of the problem
1. Given d[1..k] with limited number of d[i]'s, determine the minimum number of currency notes required to make change for amount n.
2. Given d[1..k], can you make a change for n?

Currency exchange is a particular case of Knapsack problem. Similar problems include subset sum, set partition problems.

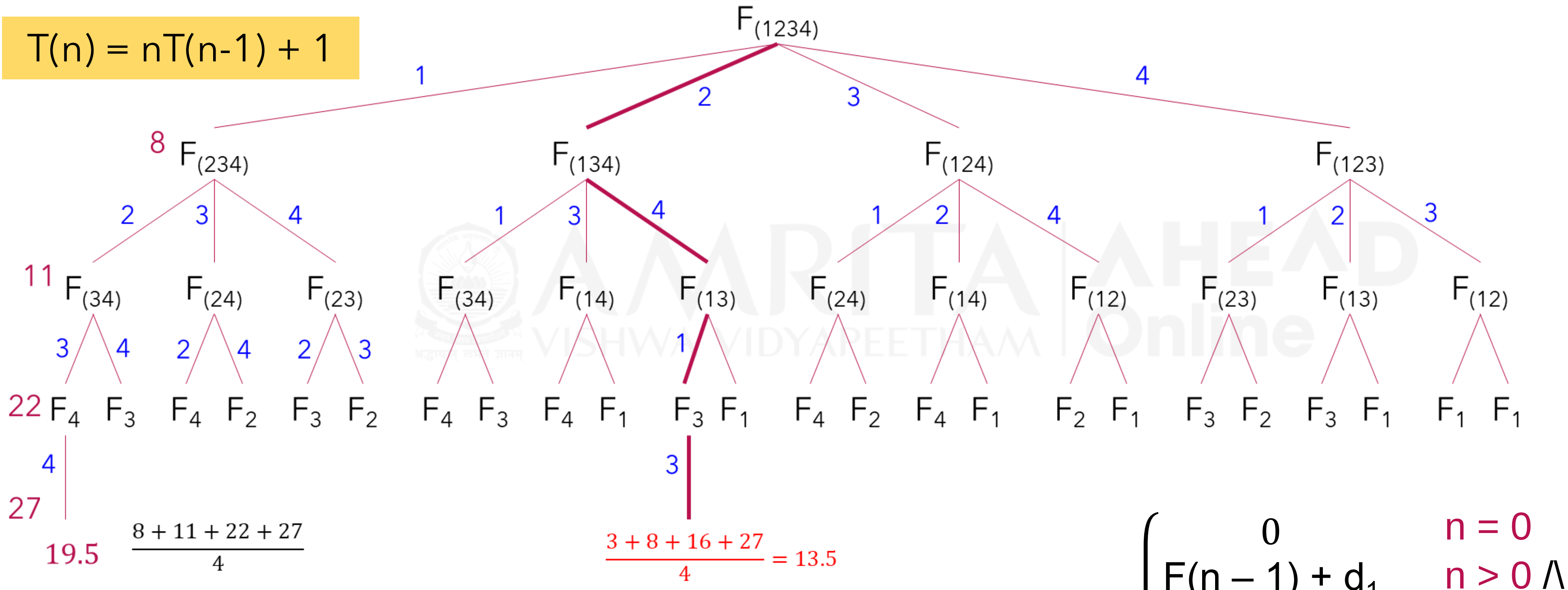
2. Task scheduling

Place your
Webcam Video here
Size 38%

Task duration: {8, 3, 11, 5}.

Task durations: $d_1, \ldots, d_n$
$F(n) = \text{TaskSchedule}(n)$

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



- ✓ Optimal substructure
- ✓ Greedy choice
- ✓ Overlapping subproblems

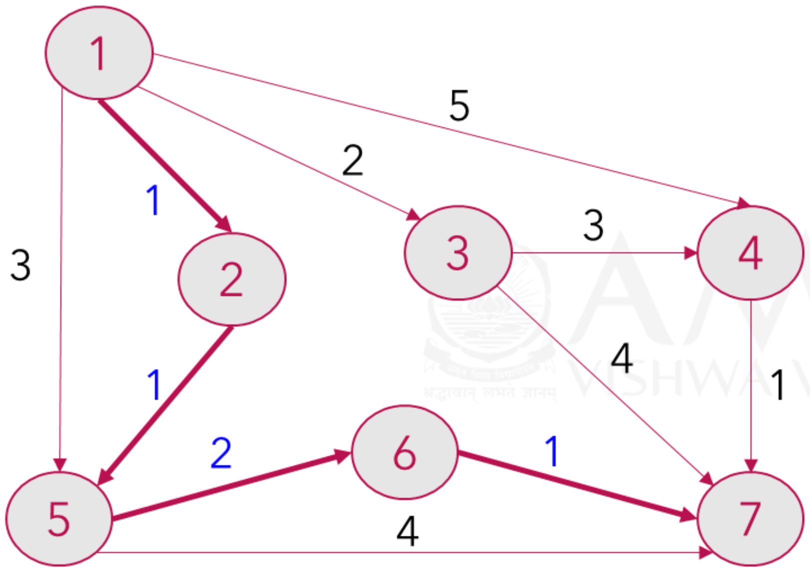
$$F(n) = \min \begin{cases} 0 & n = 0 \\ F(n-1) + d_1 & n > 0 \wedge 1^{st} \text{ not scheduled} \\ F(n-1) + d_2 & n > 0 \wedge 2^{nd} \text{ not scheduled} \\ \ldots & \\ F(n-1) + dn & n > 0 \wedge n^{th} \text{ not scheduled} \end{cases}$$

Greedy: Sort by duration, pull out tasks
$T(n) = O(n \log n)$

3. Single source shortest path

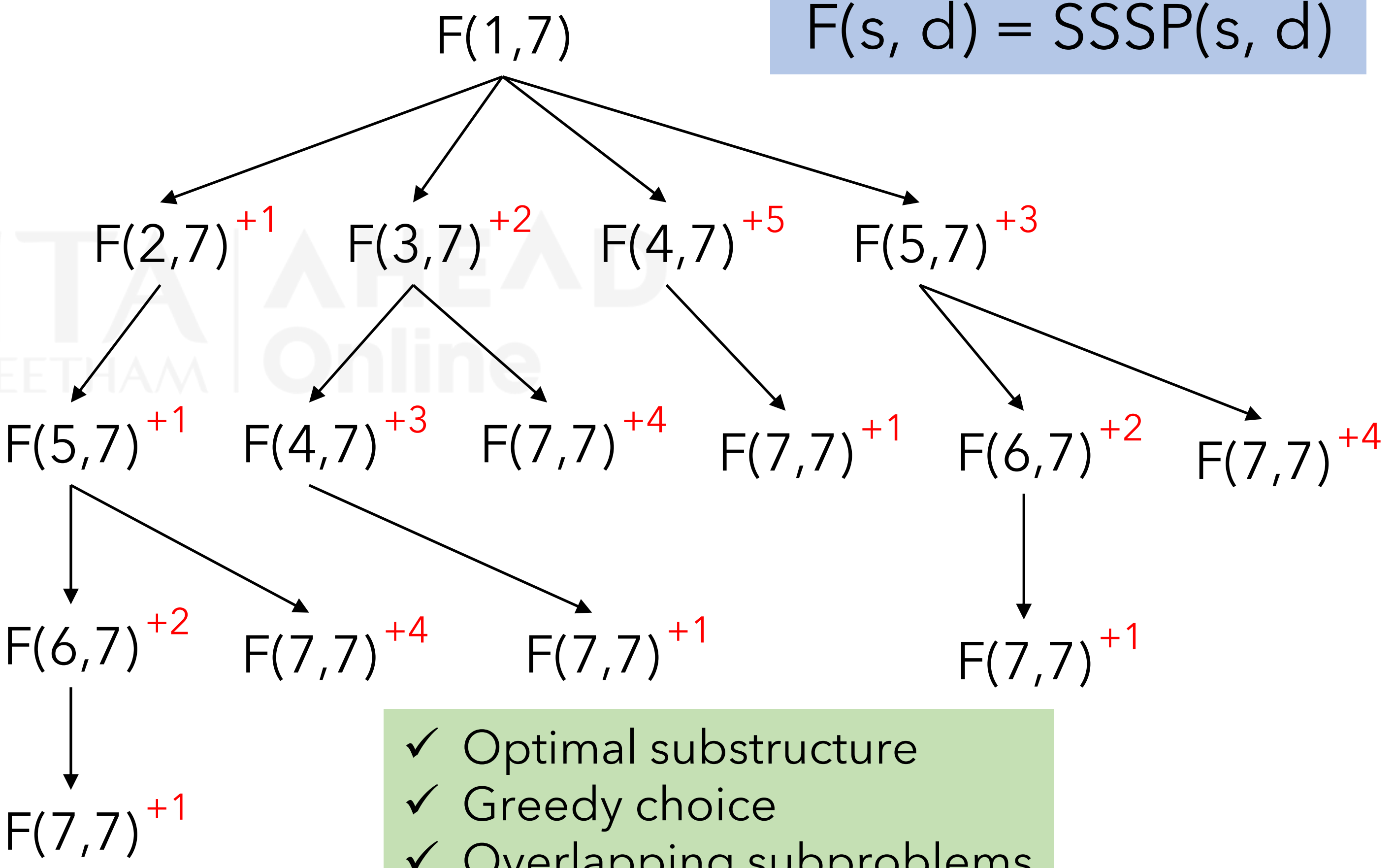
Place your
Webcam Video here
Size 38%

This is a recap from DSA course. We will only look at the recursive formulation and the properties.



$$F(s, d) = \min \begin{cases} 0 & if\ s = d \\ adj[s,v_1] + F(v_1, d) \\ adj[s,v_2] + F(v_2, d) \\ \dots \end{cases}$$

F(s, d) = SSSP(s, d)

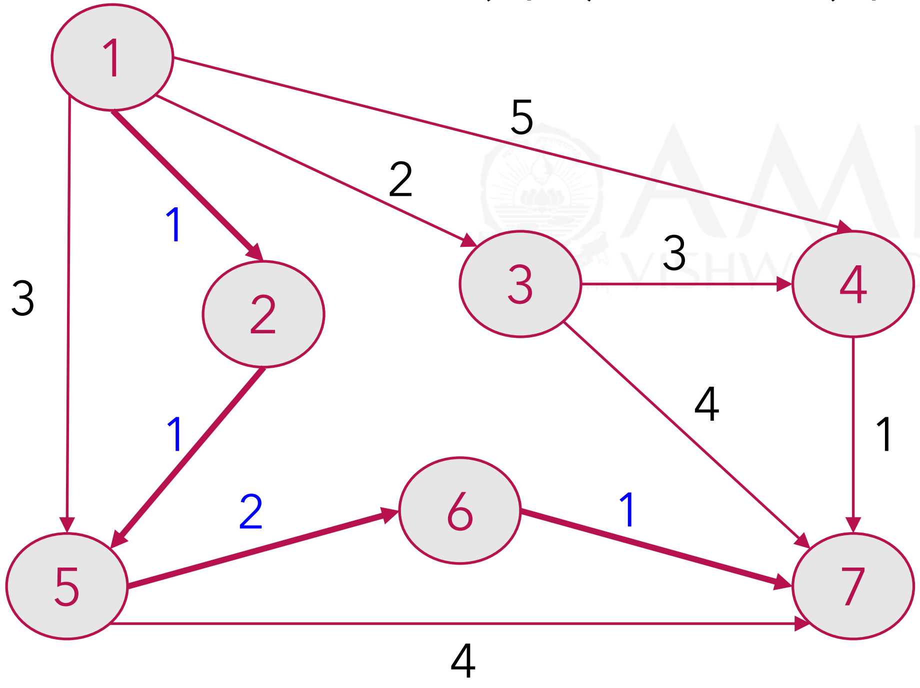


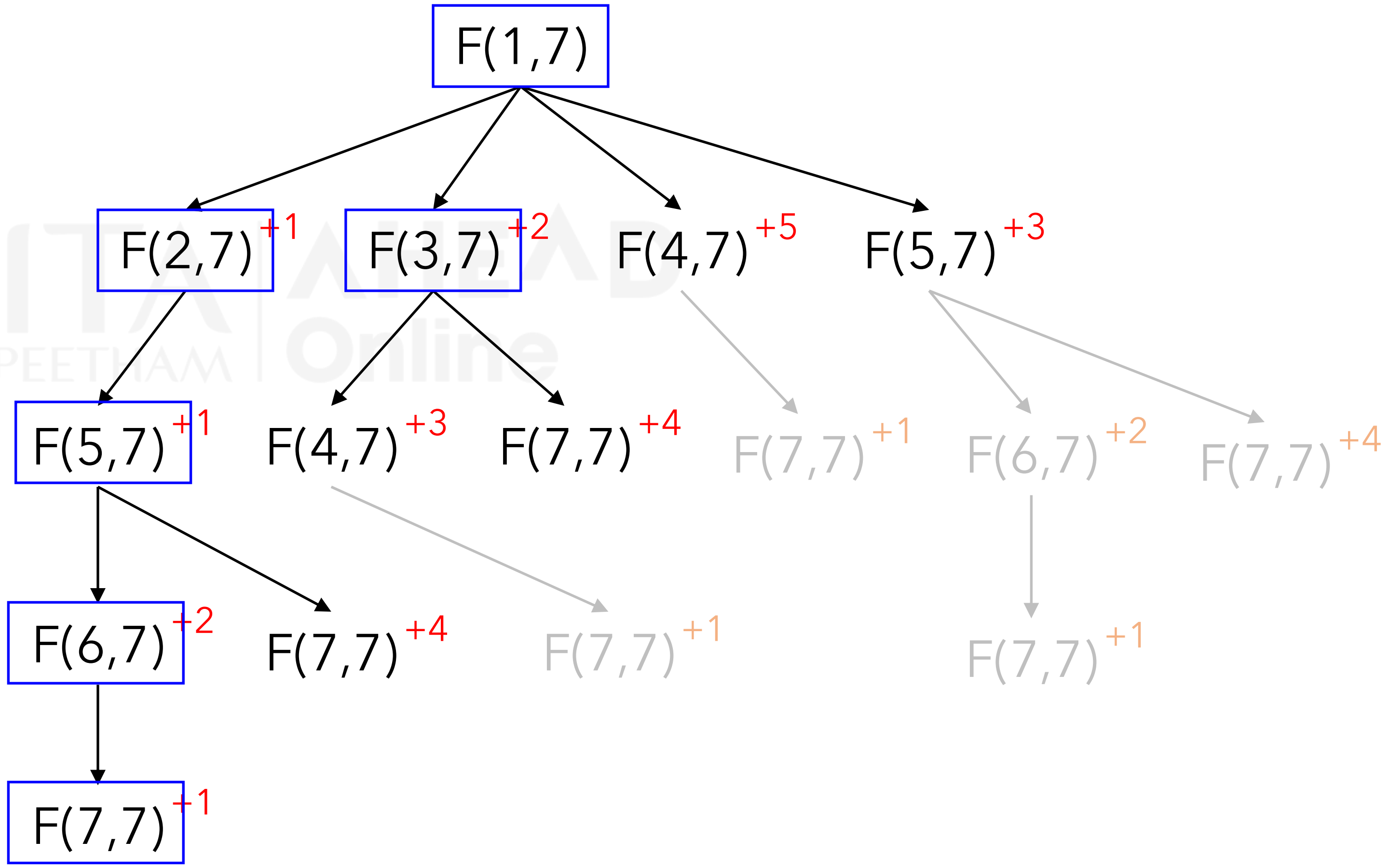
- ✓ Optimal substructure
- ✓ Greedy choice
- ✓ Overlapping subproblems

Dijkstra's Greedy Algorithm

Here is the demonstration of Dijkstra's algorithm runs.

F(s,d) = SSSP(s,d)



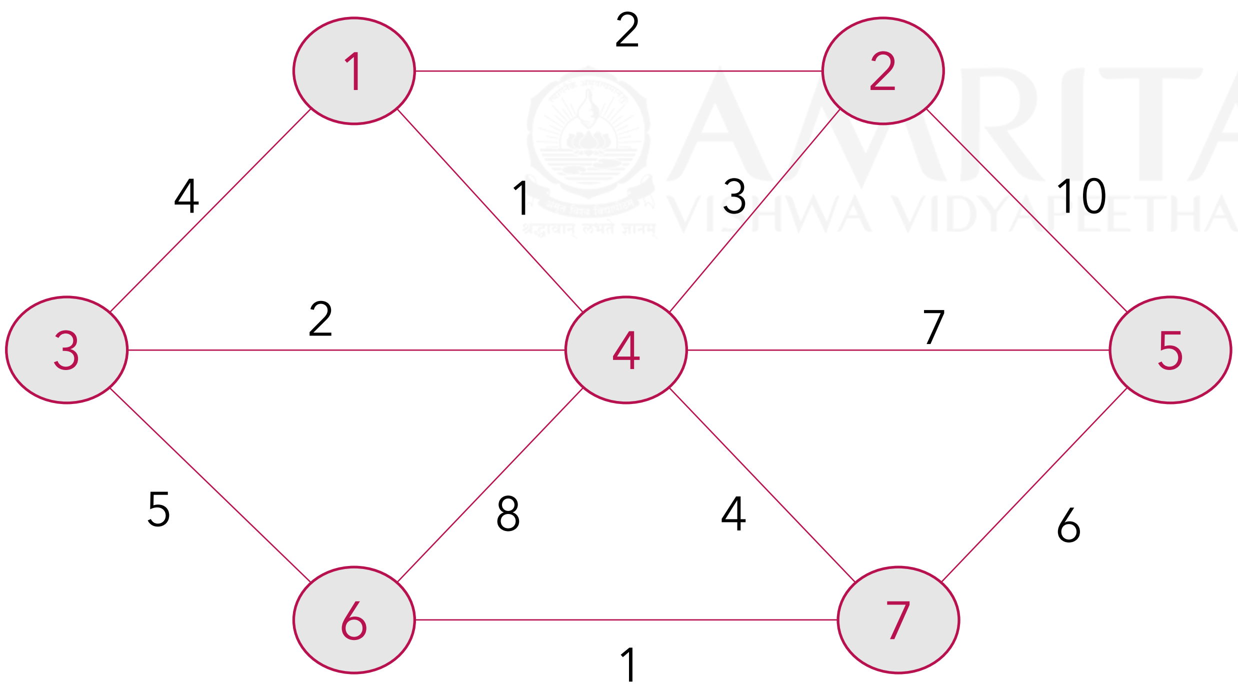


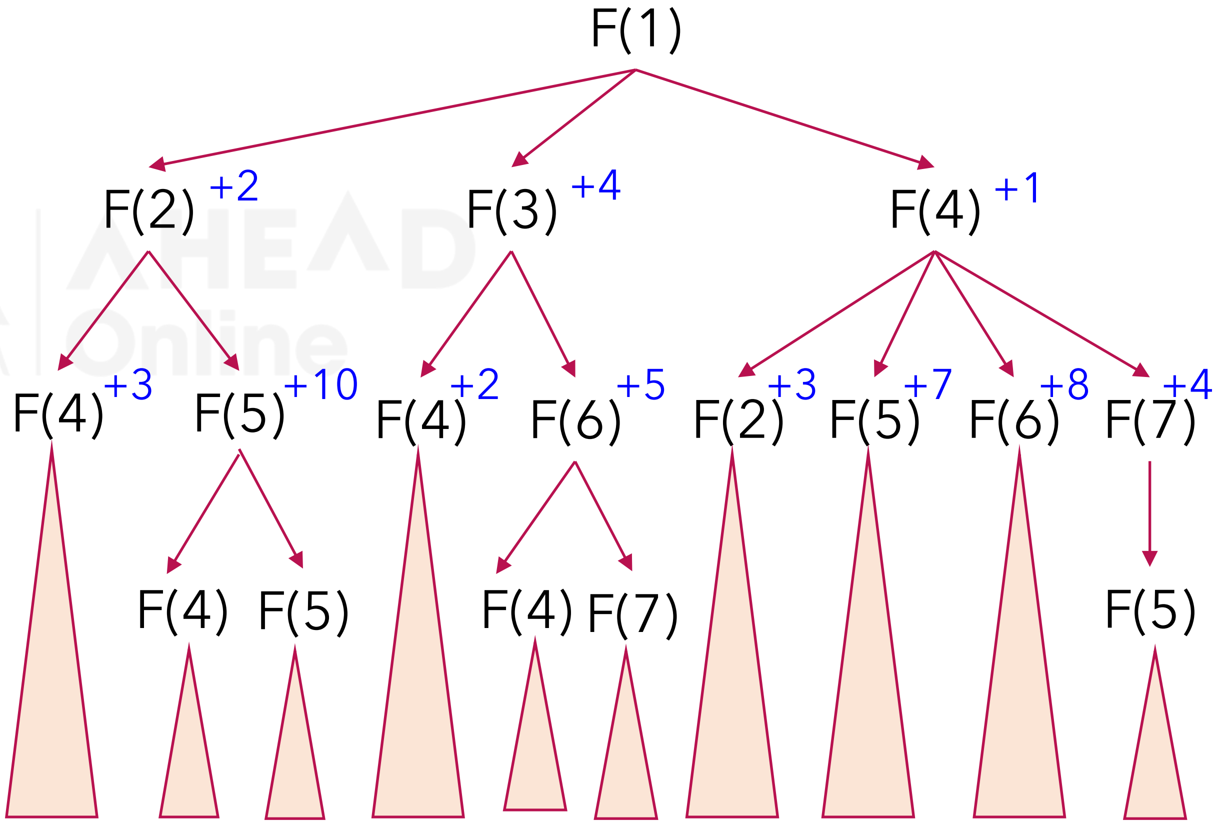
4. Minimum Spanning Tree

Place your
Webcam Video here
Size 38%

This is a recap from DSA course. Formulating the recursive solution and ascertaining properties left as exercise.

$F(G, G^{mst}) = MST(G, G^{mst})$



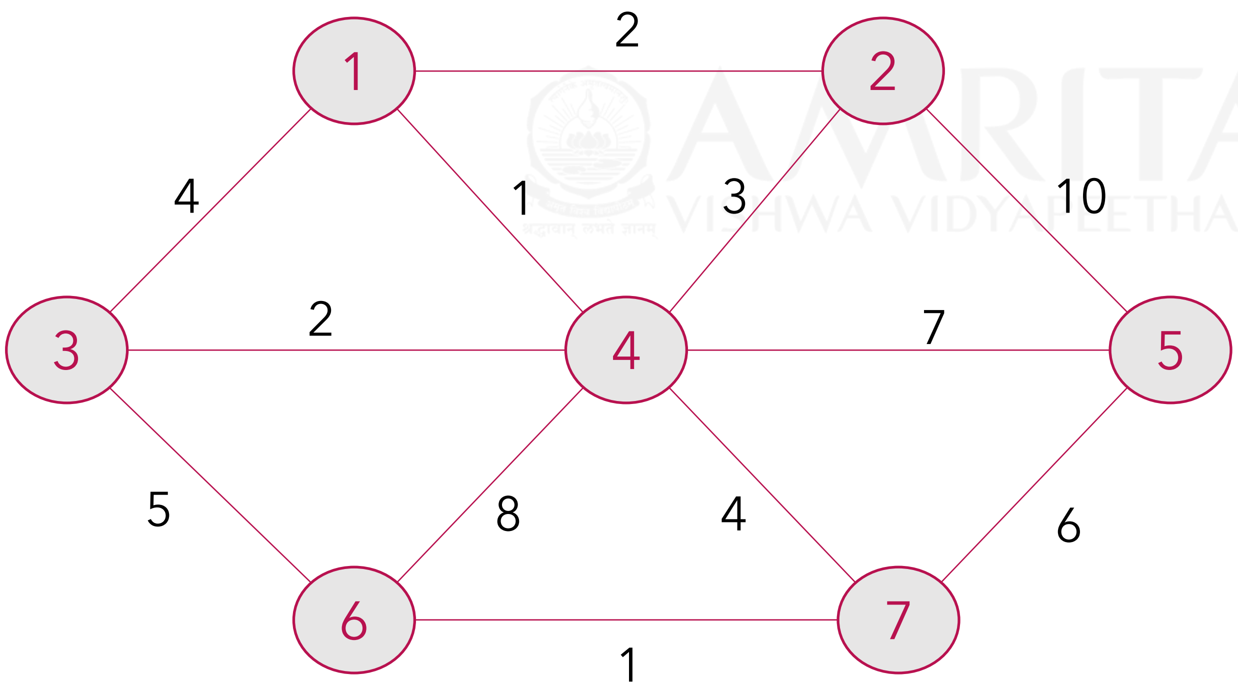


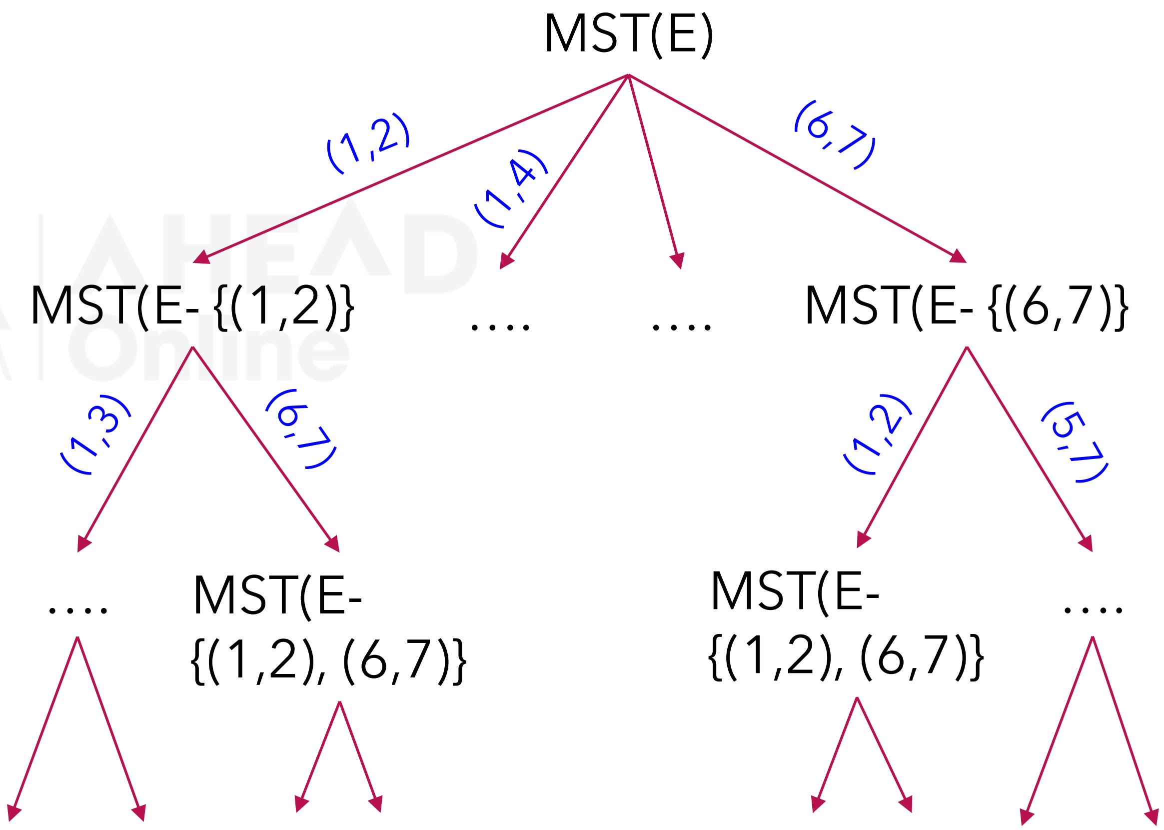
4. Minimum Spanning Tree

Place your
Webcam Video here
Size 38%

This is a recap from DSA course. Formulating the recursive solution and ascertaining properties left as exercise.

$F(E, E^{mst}) = MST(E, E^{mst})$





Huffman's Encoding - Introduction

Place your
Webcam Video here
Size 38%

A greedy algorithm to compress text made up of chars.

Let's say alphabet contains 4 chars. {a, b, c, d}.

- 2 bits are sufficient to represent them.
- a → 00 | b → 01 | c → 10 | d → 11

1) Consider a text file with 100 chars with 25 a's, 25 b's, 25 c's and 25 d's.
The total size of text = 2*100 = 200 bits.

2) Consider another text file with 100 chars with 60 a's, 25 b's, 10 c's and 5 d's. The total size of text is still 2*100 = 200 bits.

Can we do better? YES. But how?

Use fewer bits for more frequent chars.

Consider the following representation.

- a → 1 | b → 01 | c → 001 | d → 000

Text size = 60*1 + 25*2 + 10*3 + 5*3
= 60 + 50 + 30 + 15 = 155 bits.

Is this representation valid? Why or why not?

- a → 0 | b → 1 | c → 01 | d → 10

Try decoding the sequence: 0110000111101.

A very important question is: Can we correctly decode the text back?

- It depends on how encoding was achieved.

Consider the bit sequence: 0110000111101.

- This can be separated as: 0110000111101.
- And equivalent decoding is: b a d b a a a b

Huffman's Encoding – Algorithm

Place your Webcam Video here Size 38%

Let's work with the examples to compute the encoding.

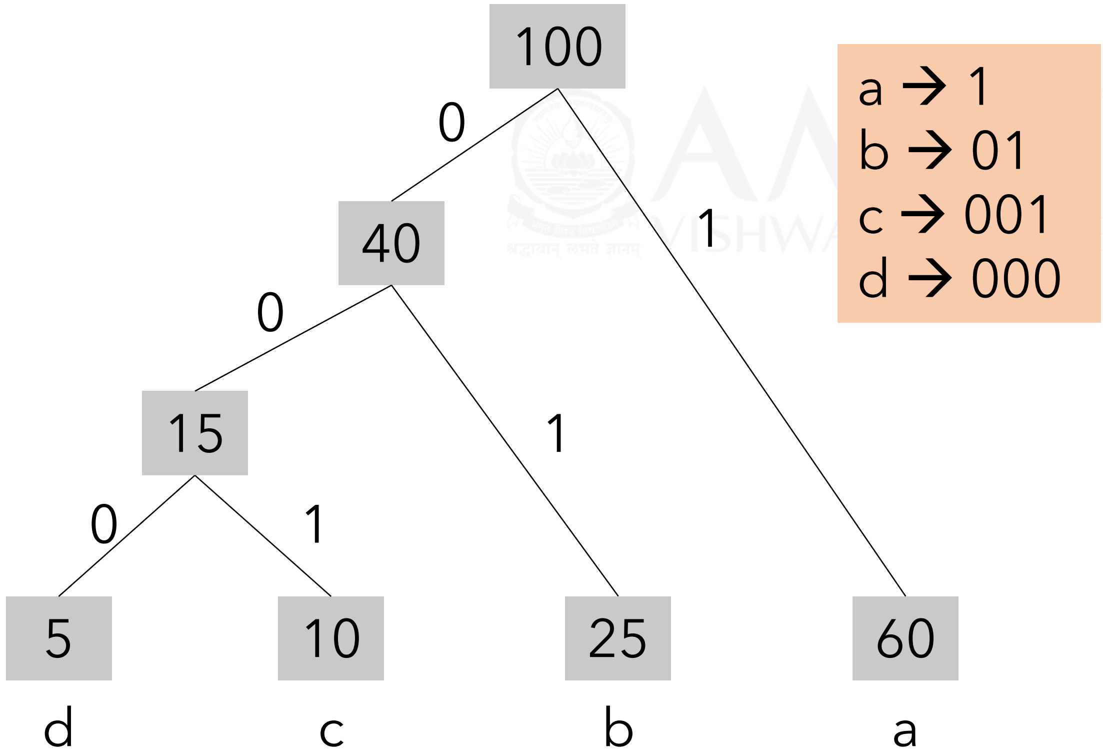
100 chars with 60 a's, 25 b's, 10 c's, 5 d's

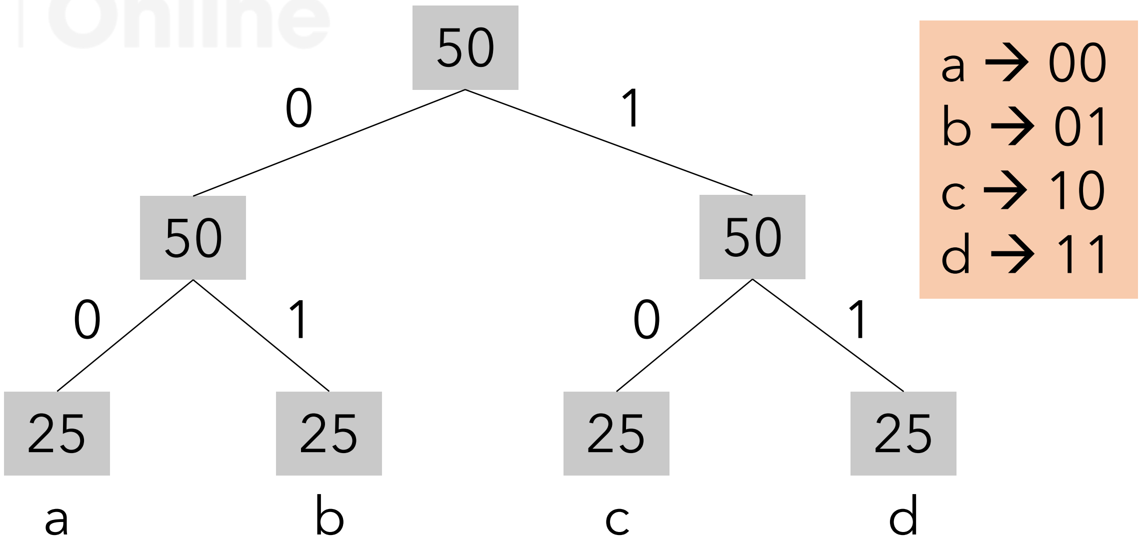
- a → 1 | b → 01 | c → 001 | d → 000

Algorithm: Huffman(occurrence count/%age of symbols)

- Sort the symbols based on occurrence count.
- Make a forest of single node trees - one per symbol.
- Merge two smallest weighted trees until there one tree.

$T(n) = n\log n + n + n-1 = O(n\log n)$





Summary

- We discussed greedy strategy for few optimization problems by analyzing the recursive formulation worked out.

Place your
Webcam Video here
Size 100%