CS F211: DATA STRUCTURES & ALGORITHMS (2ND SEMESTER 2024-25)

Divide & Conquer, Dynamic Programming

Chittaranjan Hota, PhD
Senior Professor, Computer Sc.
BITS-Pilani Hyderabad Campus
hota[AT]hyderabad.bits-pilani.ac.in

ALGORITHM TECHNIQUES: DIVIDE-AND-CONQUER

Divide-and conquer is a general algorithm design paradigm:

- Divide: divide the input data S in two disjoint subsets S_1 and S_2
- Recur: solve the sub-problems associated with S_1 and S_2
- Conquer: combine the solutions for S_1 and S_2 into a solution for S

The base case for the recursion are subproblems of size 0 or 1

Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm

Like heap-sort

- It uses a comparator
- It has $O(n \log n)$ running time

Unlike heap-sort

- It does not use an auxiliary priority queue
- It accesses data in a sequential manner (suitable to sort data on a disk)

DIVIDE AND CONQUER

-Distinguish between small and large instances.

-Small instances solved differently than larger instances.

-How did you solve your BITS F232 course assignments?

-Solving a small: <u>Min&max. of n<=2 elements</u>

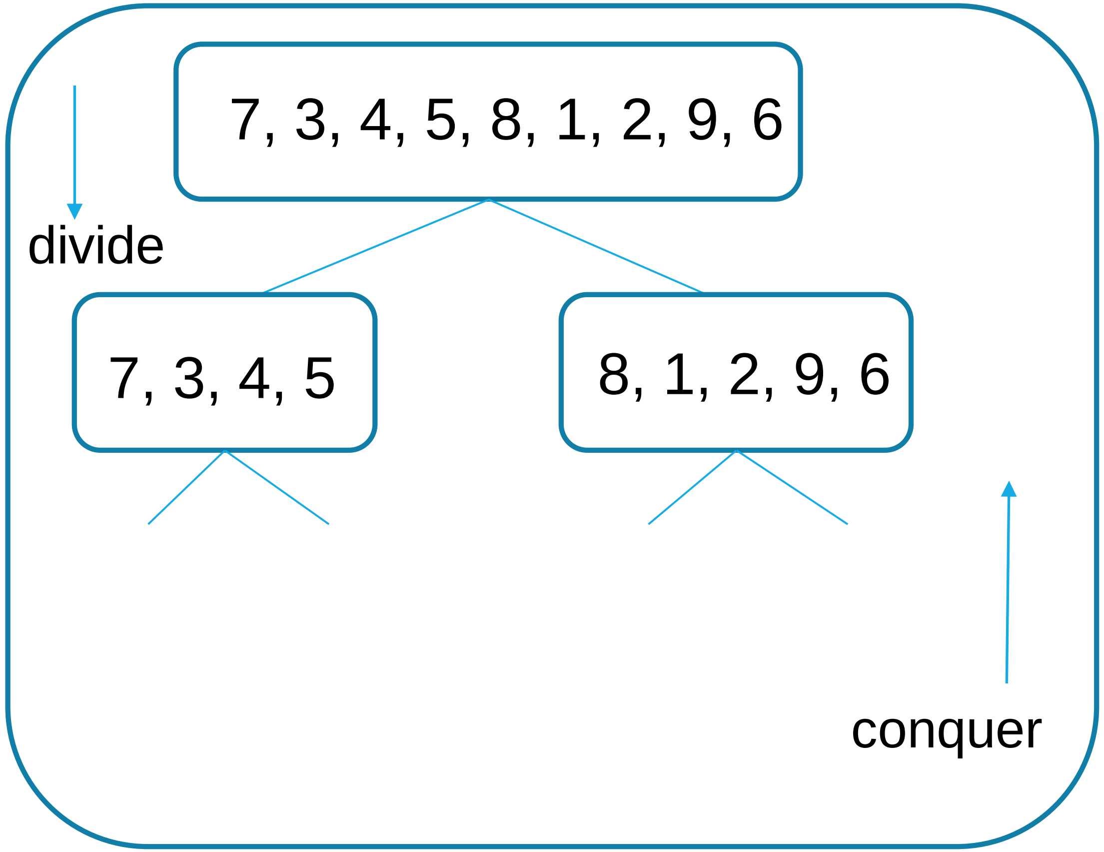
-n=0, no min and max

-n=1, min. and max is the single element

-n=2, if a<b, min = a & max=b, else min=b &max = a.

Direct/ Simple Strategy

Let us see the same for a larger instance, say 7, 3, 4, 5, 8, 1, 2, 9, 6.



MERGE-SORT: DIVIDE AND CONQUER

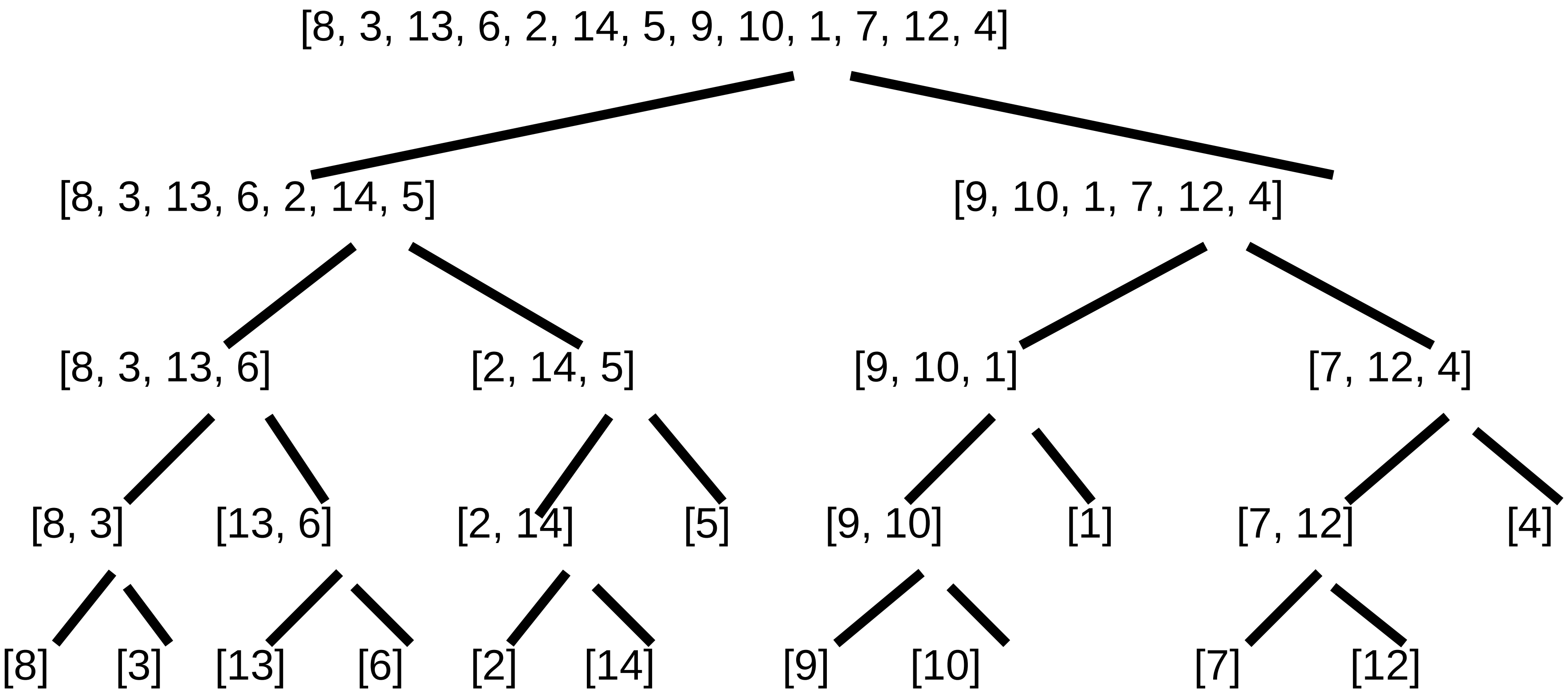
```
Algorithm mergeSort (S, C)
Input: seq S with n elements, comparator C
Output: sequence S sorted according to C
if S.size() > 1{
        (S1, S2) ← partition (S, n/2);
        mergeSort(S1, C);
        mergeSort(S2, C);
        S ← merge(S1, S2);
    }
```

Merging two sorted sequences, each with **n**/2 elements and implemented by means of a doubly linked list, takes **O**(**?**) time.

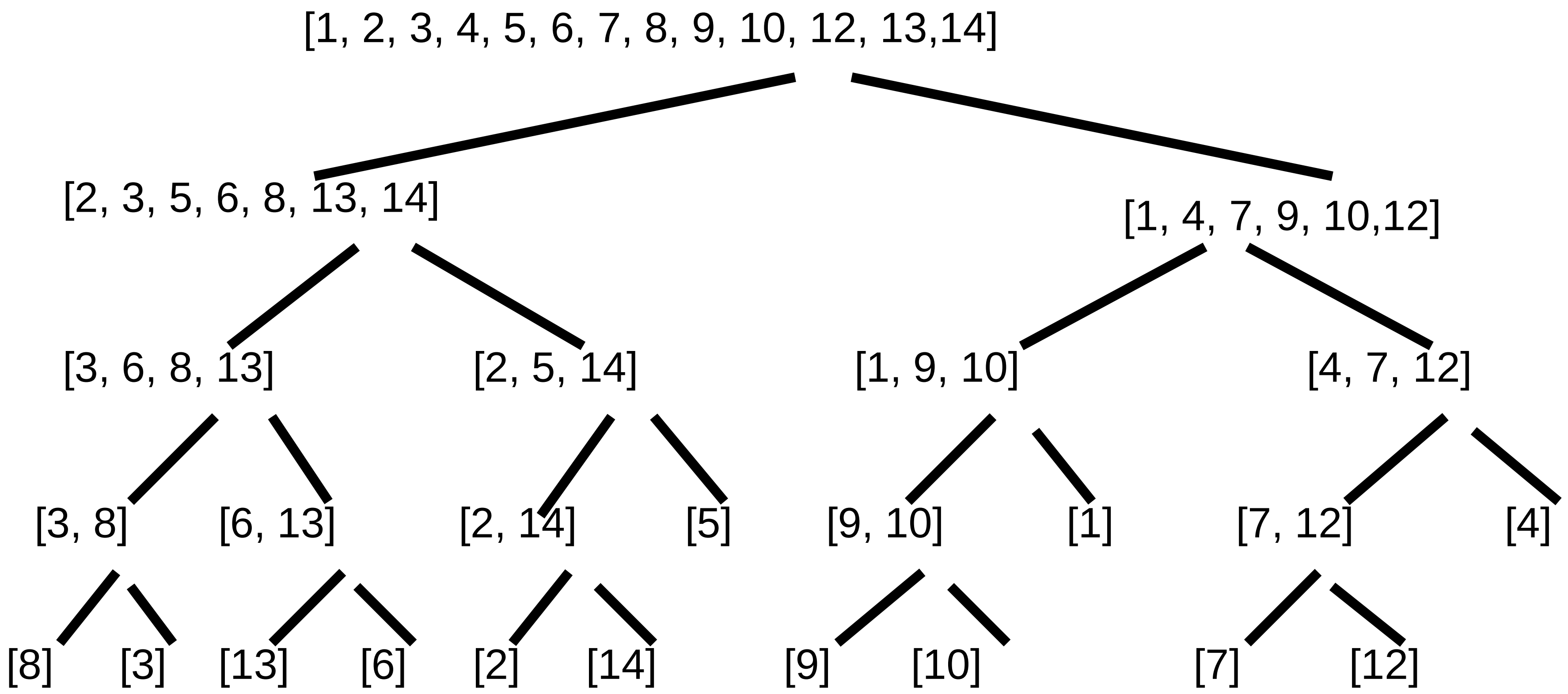
```
Algorithm merge(A, B)
Input: sequences A and B with n/2 elements each
Output: sorted sequence of A ∪ B

S ← empty sequence
while ¬A.empty()  ∧ ¬B.empty()
    if A.front() < B.front()
        S.addBack(A.front()); A.eraseFront();
    else
        S.addBack(B.front()); B.eraseFront();
while ¬A.empty()
    S.addBack(A.front()); A.eraseFront();
while ¬B.empty()
    S.addBack(B.front()); B.eraseFront();
return S
```

EXECUTION EXAMPLE



CONTINUED....


[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13,14]
[2, 3, 5, 6, 8, 13, 14]
[1, 4, 7, 9, 10,12]
[3, 6, 8, 13]
[2, 5, 14]
[1, 9, 10]
[4, 7, 12]
[3, 8]
[6, 13]
[2, 14]
[5]
[9, 10]
[1]
[7, 12]
[4]
[8]
[3]
[13]
[6]
[2]
[14]
[9]
[10]
[7]
[12]

ANALYSIS OF MERGE-SORT

The height h is $\mathbf{O}(\log n)$

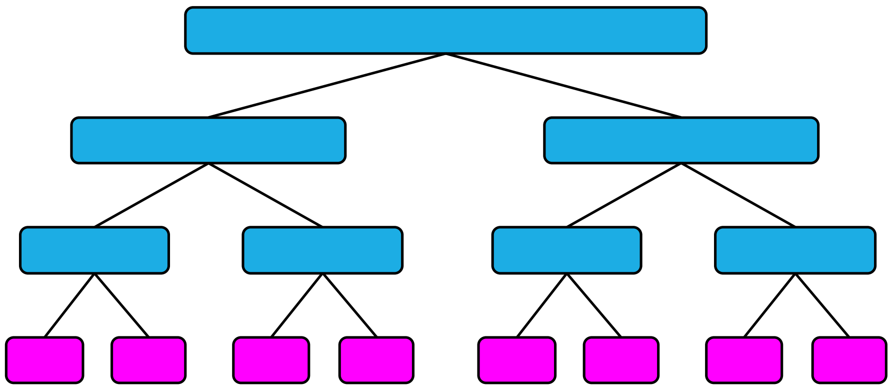
- at each recursive call we divide into half the sequence

The overall amount of work done at the nodes of depth i is $\mathbf{O}(n)$

- we partition and merge 2^i sequences of size $n/2^i$
- we make 2^{i+1} recursive calls

Thus, the total running time of merge-sort is $\mathbf{O}(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



RECURRENCE EQUATION ANALYSIS

The conquer step of merge-sort consists of merging two sorted sequences, each with $\boldsymbol{n}/2$ elements and implemented by means of a doubly linked list, takes at most $\boldsymbol{bn}$ steps, for some constant $\boldsymbol{b}$.

Likewise, the base case ($\boldsymbol{n} < 2$) will take at most $\boldsymbol{b}$ steps.

Therefore, if we let $\boldsymbol{T(n)}$ denote the running time of merge-sort:

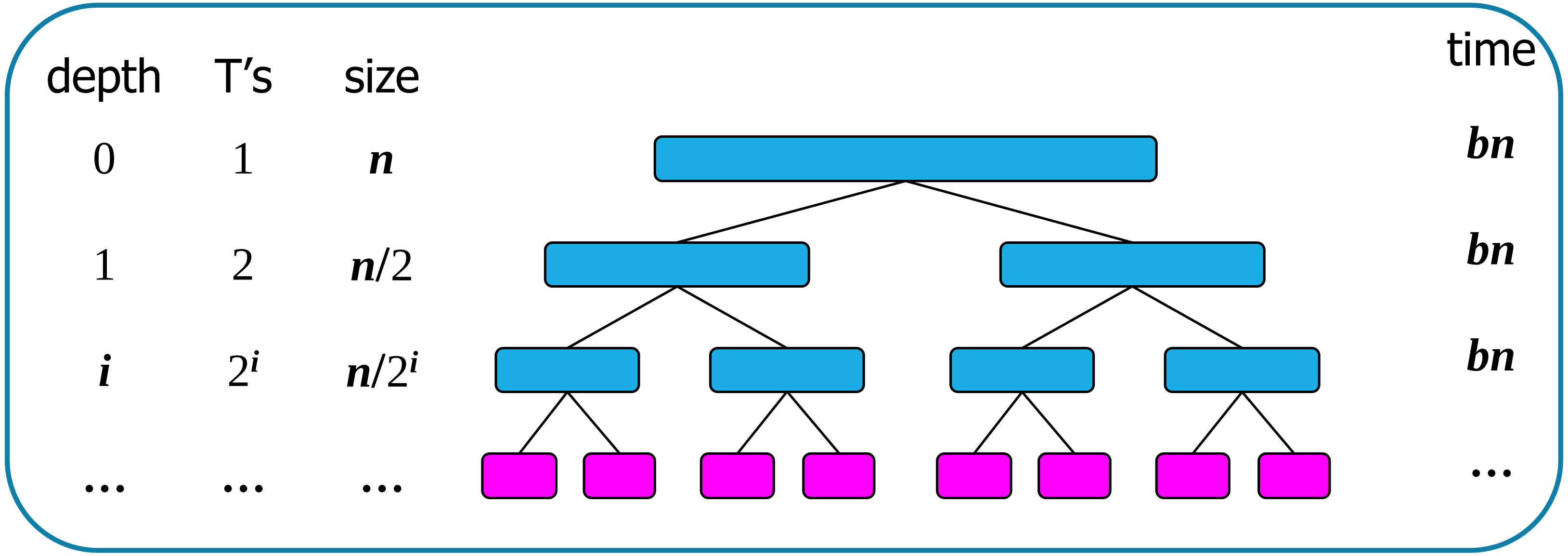
$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

We can therefore analyze the running time of merge-sort by finding a closed form solution to the above equation.

- That is, a solution that has $\boldsymbol{T(n)}$ only on the left-hand side.

SOLUTION TO RECURRENCE RELATION

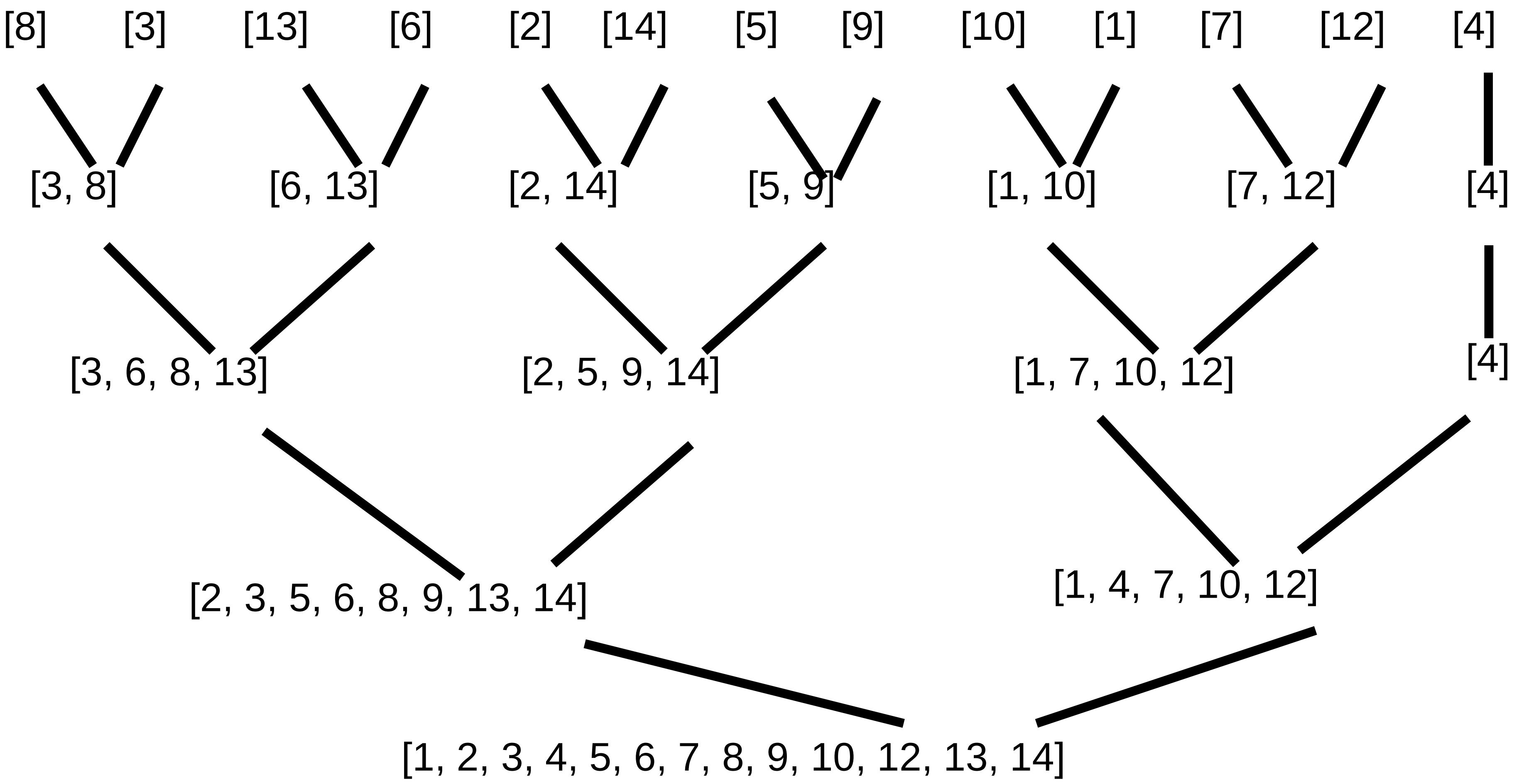
(Recursion Tree)



Total time = ***bn*** + ***bn*** log ***n***

(last level plus all previous levels)

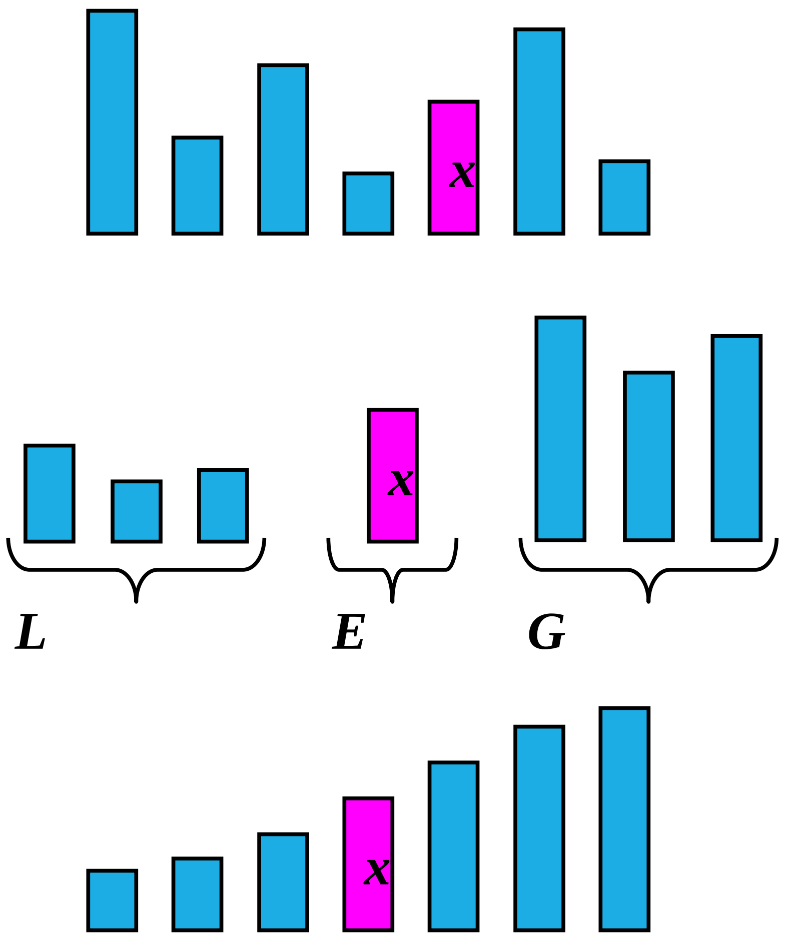
NONRECURSIVE MERGE SORT



QUICK-SORT: DIVIDE AND CONQUER

Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:

- Divide: pick a random element x (called pivot) and partition S into
  - L elements less than x
  - E elements equal x
  - G elements greater than x
- Recur: sort L and G
- Conquer: join L, E and G



PARTITION STEP

We partition an input sequence as follows:

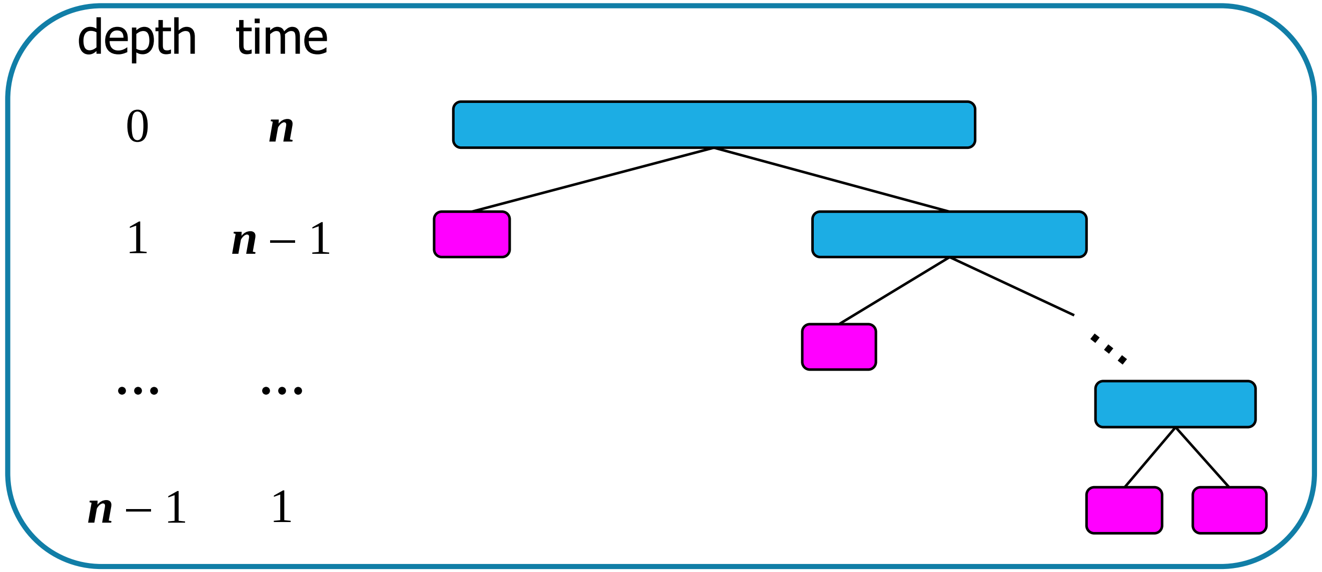
- We remove, in turn, each element y from S and
- We insert y into L, E or G, depending on the result of the comparison with the pivot x

Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time

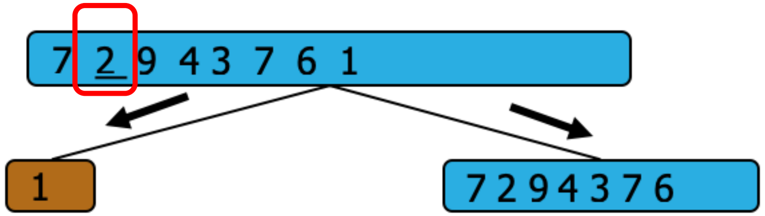
Thus, the partition step of quick-sort takes $O(n)$ time

Algorithm *partition*(*S*, *p*)
  Input sequence ***S***, position ***p*** of pivot
  Output subsequences ***L, E, G*** of the
    elements of ***S*** less than, equal to,
    or greater than the pivot, resp.
  L, E, G ← empty sequences
  x ← ***S.erase***(***p***)
  while ¬***S.empty***()
    y ← ***S.eraseFront***()
    if ***y*** < ***x***
      L.insertBack(***y***)
    else if ***y*** = ***x***
      E.insertBack(***y***)
    else { ***y*** > ***x*** }
      G.insertBack(***y***)
  return ***L, E, G***

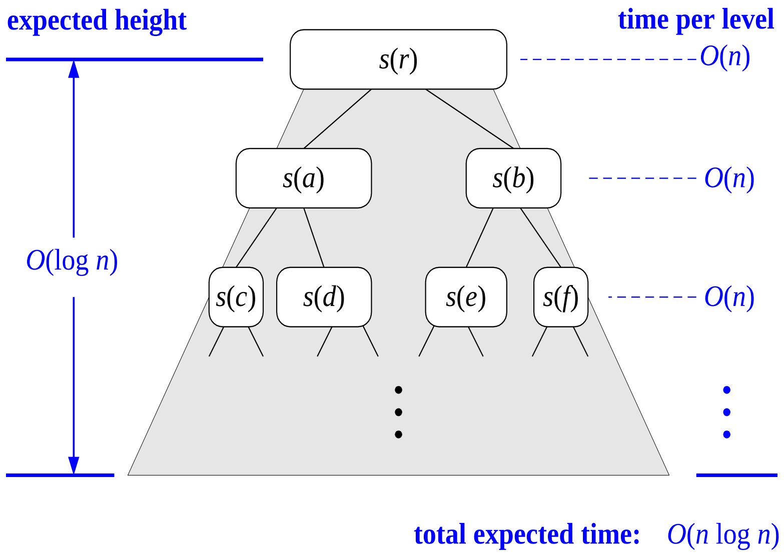
RUNNING TIME(WORST & BEST)



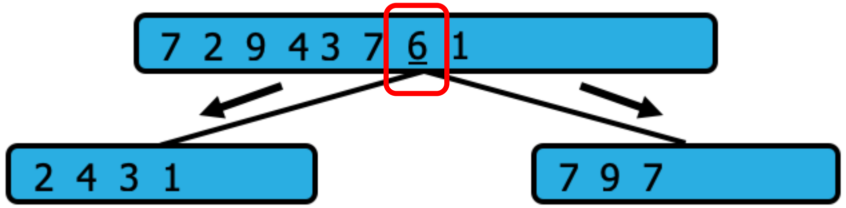
(Worst case: Pivot with unique minimum or maximum value that makes either L or G with 'n-1' size and the ot



Good or bad Pivot?



(Best/ expected case)



Good or bad Pivot?

QUICK SORT: RECURRENCE RELATION

Divide step: The time complexity of this step is equal to the time complexity of the partition algorithm
= O(n).

Conquer step time complexity = Time complexity to sort left subarray recursively + Time complexity to sort right subarray recursively
= T(i) + T(n - i - 1)

Combine step: This is a trivial step and there is no operation in the combine part of quick sort. So time complexity of combine step
= O(1).

Recurrence relation of the quick sort:

- $T(n) = c$, if $n = 1$
- $T(n) = T(i) + T(n - i - 1) + cn$, if $n > 1$

Worst case: $i = n-1$ => $O(n*n)$

Best case: $i = n/2$ => $O(n\log n)$

BUCKET-SORT: LINEAR TIME SORTING

Let S be a sequence of n (key, element) entries with keys in the range $[0, N-1]$

Bucket-sort uses the keys as indices into an auxiliary array B of sequences (buckets)

Phase 1: Empty sequence S by moving each entry (k, o) into its bucket $B[k]$

Phase 2: For $i = 0, \ldots, N-1$, move the entries of bucket $B[i]$ to the end of sequence S

Analysis:

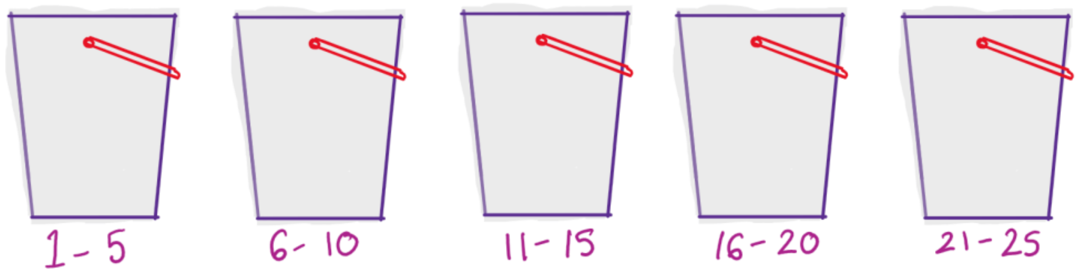
- Phase 1 takes $O(n)$ time
- Phase 2 takes $O(n+N)$ time

Bucket-sort takes $O(n+N)$ time → Best case (keys are distributed evenly in each bucket)

```
Algorithm bucketSort(S, N)
Input sequence S of (key, element)
      items with keys in the range [0, N - 1]
Output sequence S sorted by increasing
keys
B ← array of N empty sequences
while (¬S.empty()) {
    (k, o) ← S.front();
    S.eraseFront();
    B[k].insertBack((k, o));
    }
    for i ← 0 to N – 1{
        while (¬B[i].empty())
            (k, o) ← B[i].front();
            B[i].eraseFront();
            S.insertBack((k, o));
    }
```

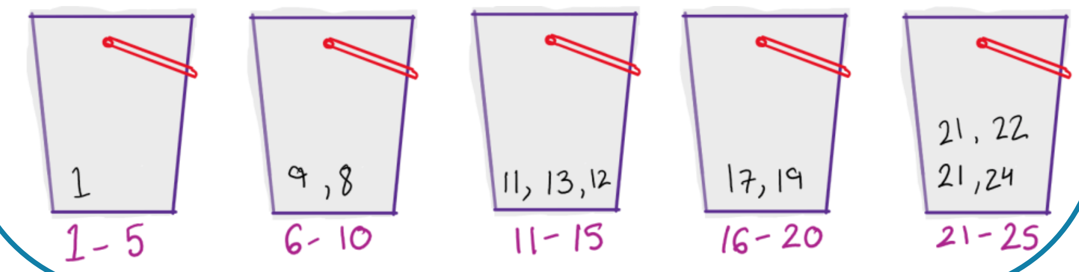
EXAMPLE OF BUCKET SORT

The values range from 0-25.

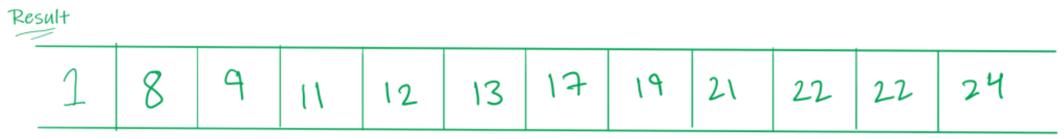


Divide the elements into smaller problems.

→ A little bit of Divide and Conquer…



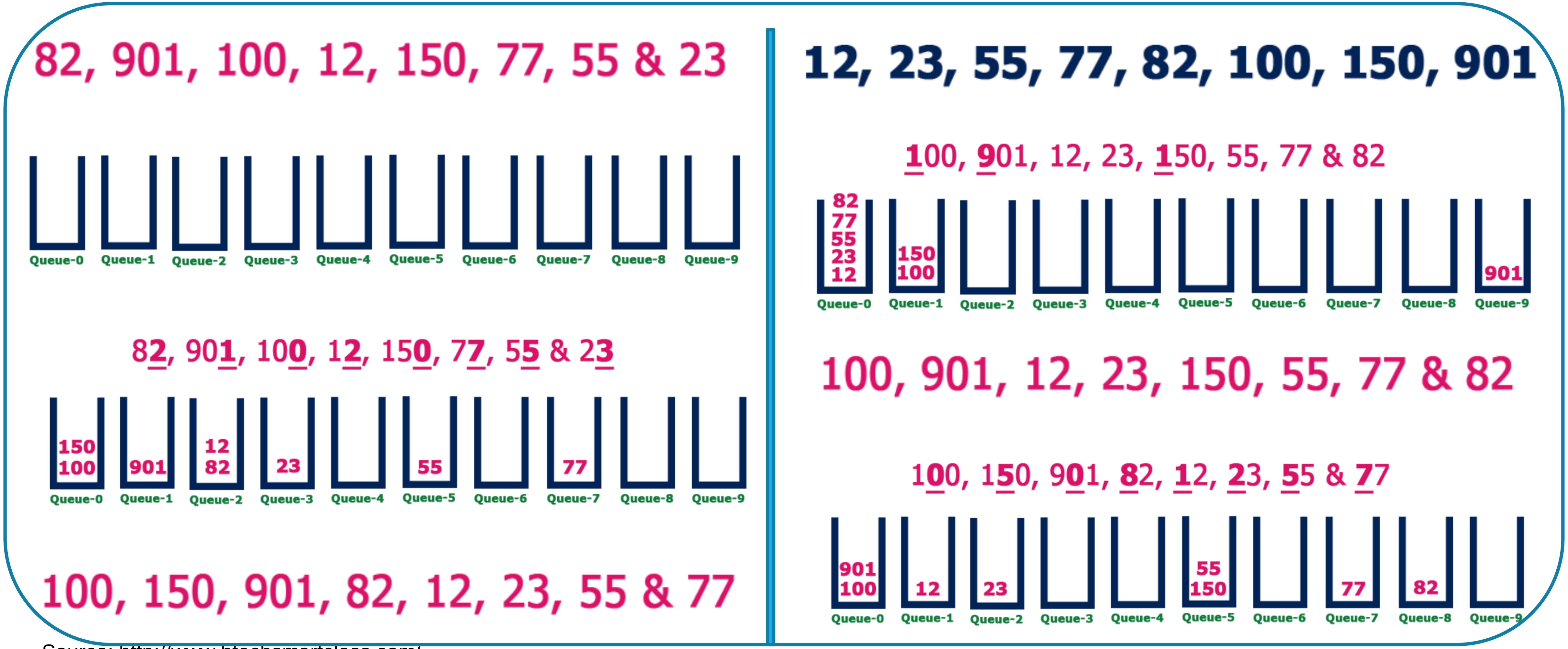
The relative order of any two items with the same key is preserved after the execution of the algorithm→ Stable sort



Iterate over each and populate the final array.

Sort these buckets individually.

RADIX SORT: LINEAR TIME (LEXICOGRAPHIC SORT)



Source: http://www.btechsmartclass.com/

DYNAMIC PROGRAMMING (DP)

- Both Divide & Conquer (DC), Dynamic Programming (DP) break a large problem into small sub-problems.
- What will you choose for these four: Binary Search, Dijkstra's Shortest Path, Towers of Hanoi, Closest Pair of Points?
- Fibonacci sequence?
- Can you solve all problems that are solvable by recursion using Dynamic Programming like Merge and Quick sort?
- Complexity improvement from Exponential to Polynomial.
- 0/1 Knapsack?

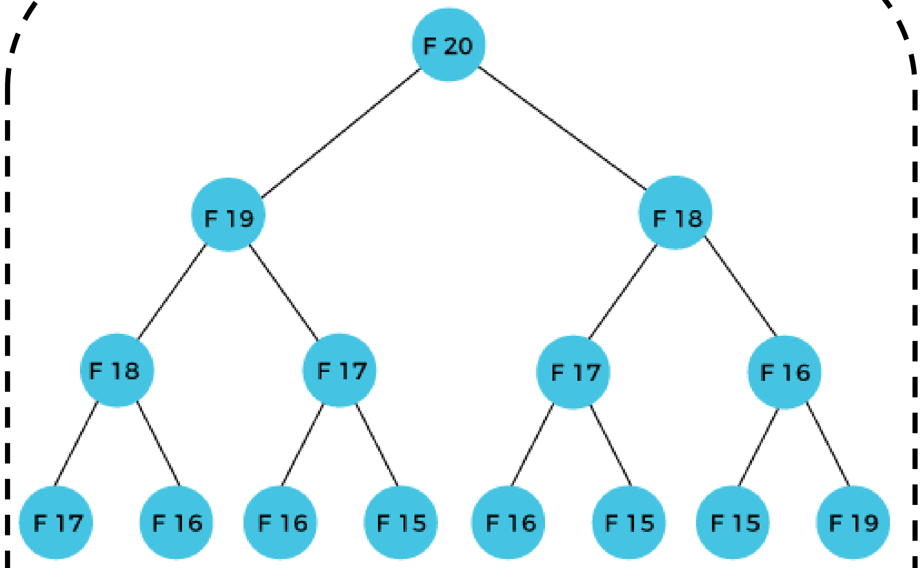
Input: N = 3, W = 4, profit[] = {1, 2, 3}, weight[] = {4, 5, 1}
Output: ?

EXAMPLE

```cpp
#include <iostream>
#include <vector>
using namespace std;
int fibonacci(vector <int> &dp, int n)
{
        if(n == 0)
                return 0;
        if(dp[n] != 0)
                return dp[n];
        dp[n] = fibonacci(dp, n - 1) + fibonacci(dp, n - 2);
        return dp[n];
}
int main() {
        int n;
        cin >> n;
        vector <int> dp(n, 0);
        dp[1] = 1;
        cout << fibonacci(dp, n - 1) << endl;
}
```

RECAP: DYNAMIC PROGRAMMING

- A technique for solving a complex problem by first breaking into a collection of simpler sub-problems, solving each sub-problem just once, and then storing their solutions to avoid repetitive computations.
- The sub-problems are optimized to optimize the overall solution is known as optimal substructure property.



- break the complex one down into simpler subproblems.
- find the optimal solution to these sub-problems.
- store the results of subproblems (memoization).
- so that, reuse them when required.
- finally, calculate the result of the complex problem.

DIVIDE & CONQUER VS. DYNAMIC PROGRAMMING

Feature	Divide and Conquer	Dynamic Programming
Approach	Recursively breaks problems into independent subproblems.	Solves overlapping subproblems optimally, storing solutions to avoid recomputation.
Subproblem Overlap	Subproblems are independent (no overlap).	Subproblems are overlapping (reused multiple times).
Storage	Does not store solutions to subproblems.	Stores solutions (memoization or tabulation) for reuse.
Base Case	Relies on recursion until reaching a base case.	Builds solutions iteratively or recursively with stored states.

- When to use what? Use D&C when subproblems are independent (e.g., sorting, searching). Use DP when subproblems overlap and optimal substructure exists (e.g., optimization problems).

FIBONACCI SERIES USING DYNAMIC PROGRAMMING

F(0) → F(1) → F(2) → F(3) → F(4) → F(5)

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

(Recursive approach: Complexity 2^n for large 'n')

```
static int count = 0;
int fib(int n) {
  If (memo[n]!= NULL)
    return memo[n];
  count++;
  if( n<0 )
   error;
 if(n == 0)
   return 0;
 if(n == 1)
   return 1;
  sum = fib(n-1) + fib(n-2);
  memo[n] = sum;
}
```

(Dynamic programming: Top down approach with complexity O(n))

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

(Dynamic programming: Bottom-up approach)

MATRIX CHAIN PRODUCT OR MATRIX CHAIN MULTIPLICATION

Given matrices: A: 10×30, B: 30×5, and C: 5×60, We want to fully parenthesize A × B × C to minimize scalar multiplications.

Matrix dimensions array: P=[10, 30, 5, 60]

This means:

- A = P[0] × P[1] = 10 × 30
- B = P[1] × P[2] = 30 × 5
- C = P[2] × P[3] = 5 × 60

Goal:

Minimize the number of scalar multiplications for:

A×B×C

There are two ways to parenthesize:

1.?

2.?

Let us compute both:

A×(B×C) → Total: 9000 + 18000 = 27000

B= 30X5, C=5X60 → BXC = 30X60, → Scalar Multiplications: 30X5X60 = 9000

AX(BXC) → A=10X30, BXC=30X60 → AX(BXC)=10X60 → Scalar Multiplications: 10X30X60=18000

(AXB)XC →

What is the best way?

Total: 1500 + 3000 = 4500

A=10X30, B=30X5 → AXB = 10X5, Scalar multiplications: 10X30X5 =1500

(AXB)XC → AXB = 10X5, C=5X60 → (AXB)XC = 10X60 → Scalar Multiplications: 10X5X60 = 3000

MATRIX CHAIN PRODUCT USING DP

p[] = {5, 4, 6, 2, 7}

Step 1: Initialize the DP table

Define m[i][j] as the minimum number of multiplications needed to compute matrices from i to j.

For any i == j, m[i][i] = 0 since a single matrix doesn't need multiplication.

Fill the table for increasing chain lengths:

Step 3: Chain length L = 3 (three matrices)

m[1][3] → A * B * C

Try all splits:

k = 1: (A)*(BC) → 5×4×2 + m[2][3] = 40 + 48 = 88

k = 2: (AB)*C → 5×6×2 + m[1][2] = 60 + 120 = 180

min = 88, so m[1][3] = 88

m[2][4] → B * C * D

Try all splits:

k = 2: (B)*(CD) → 4×6×7 + m[3][4] = 168 + 84 = 252

k = 3: (BC)*D → 4×2×7 + m[2][3] = 56 + 48 = 104

min = 104, so m[2][4] = 104

A (5×4), B (4×6), C (6×2), D (2×7)

Step 2: Chain length L = 2 (two matrices)

m[1][2] → A * B

Cost = 5×4×6 = 120

m[2][3] → B * C

Cost = 4×6×2 = 48

m[3][4] → C * D Cost = 6×2×7 = 84

Step 4: Chain length L = 4 (all four matrices)

m[1][4] → A * B * C * D

Try all splits:

k = 1: (A)*(BCD) → 5×4×7 + m[2][4] = 140 + 104 = 244

k = 2: (AB)*(CD) → 5×6×7 + m[1][2] + m[3][4] = 210 + 120 + 84 = 414

k = 3: (ABC)*D → 5×2×7 + m[1][3] = 70 + 88 = 158

min = 158, so m[1][4] = 158

MCP: DP USING BOTTOM UP (TABULAR METHOD)

$A_{5X4}.B_{4X6}.C_{6X2}.D_{2X7}$:

	0	1	2	3
0	0			
1		0		
2			0	
3				0

$A_{5X4}.B_{4X6}$ =5X4X6=120

$B_{4X6}.C_{6X2}$ = 4X6X2=48

$C_{6X2}.D_{2X7}$ = 6X2X7=84

	0	1	2	3
0	0	120		
1		0	48	
2			0	84
3				0

$A_{5X4}.B_{4X6}.C_{6X2}$ = min((A.B)C, A(B.C))
= min(120+5x6x2, 5x4x2+48)
= min(180, 88) = 88

$B_{4X6}.C_{6X2}.D_{2x7}$ = min((B.C)D, B(C.D))
= min(48+4x2x7, 4x6x7+84)
= min(104, 252) = 104

	0	1	2	3
0	0	120	88	
1		0	48	104
2			0	84
3				0

	0	1	2	3
0	0	120	88	158
1		0	48	104
2			0	84
3				0

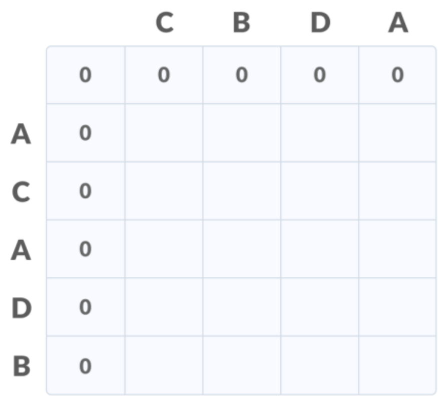
$A_{5X4}.B_{4X6}.C_{6X2}.D_{2x7}$ = min((A.B.C)D, (A.B)(C.D), A.(B.C.D))
= min(88+5x2x7, 120+ 5X6X7 +84, 5X4X7 + 104)
= min(158, 414, 244) = 158

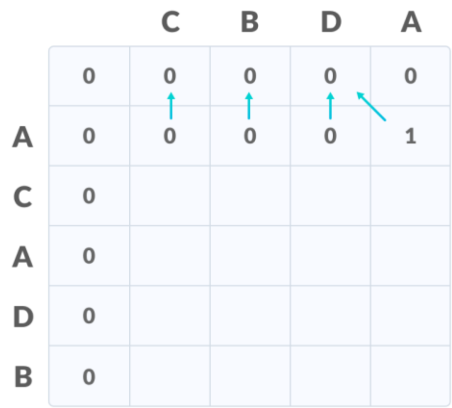
$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

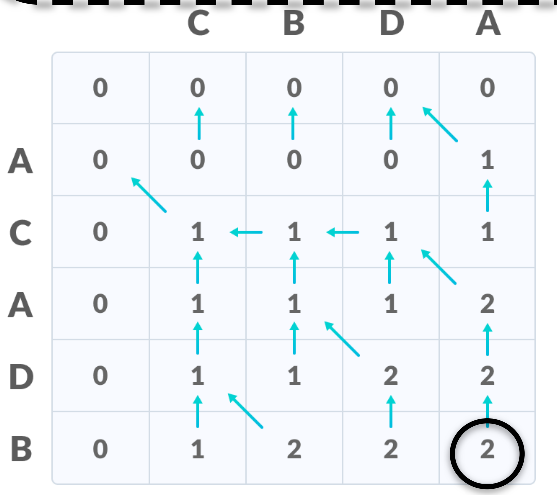
LONGEST COMMON SUBSEQUENCE USING MEMOIZATION

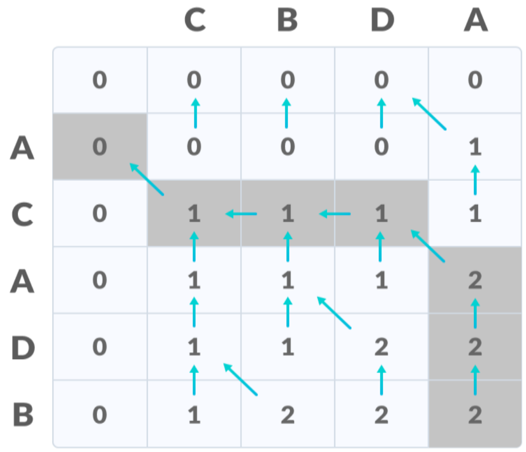
- Finding the longest sequence that appears in the same relative order (but not necessarily contiguously) in two strings. Ex: X = "ACADB", Y = "CBDA". Applications: Version control, Plagiarism check, DNA seq.

```
for each (i, j) {
  If (X[i-1] == Y[j-1])
     memo[i][j] = 1 + memo[i-1][j-1];
  else
     memo[i][j] = max(memo[i-1][j], memo[i][j-1])
}
```

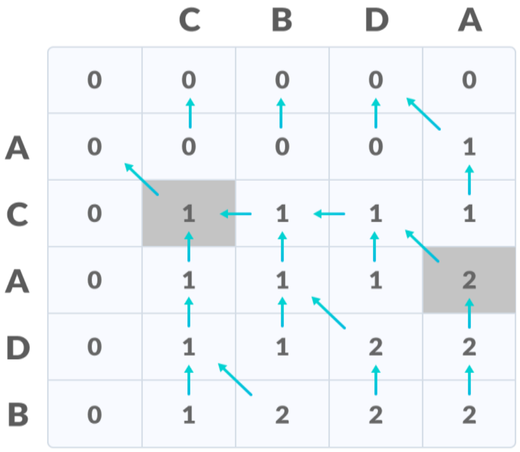








Select the cells with diagonal arrows



LCS: CA

Top down

THANK YOU!

Next class: Pattern Matching Algorithms