Design and Analysis of Algorithms

Unit - 2

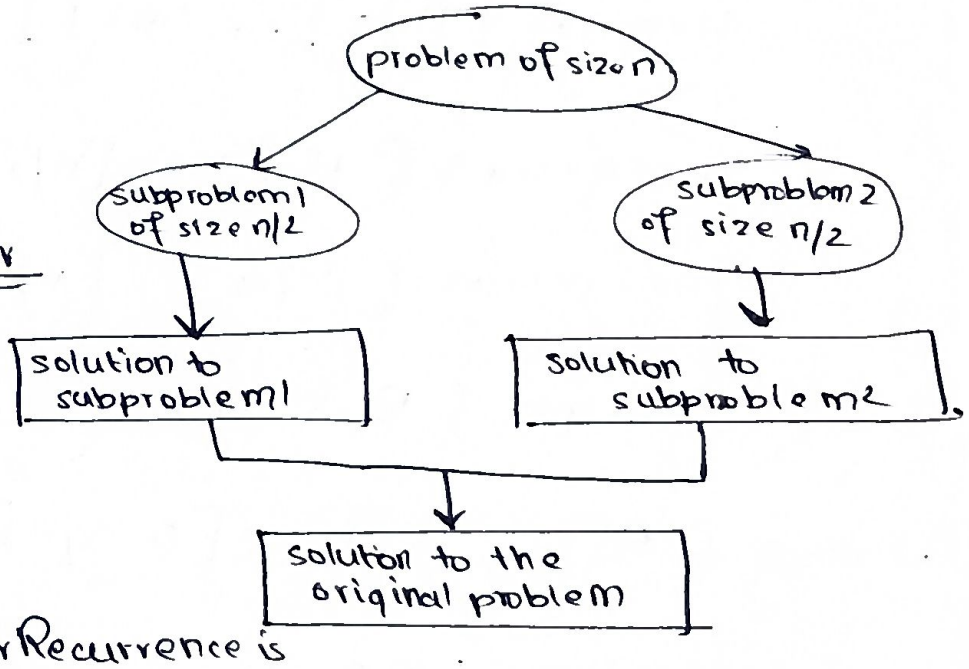
Divide and Conquer - Backtracking

* Divide and Conquer

→ A problem is divided into several subproblems of the same type, ideally of about equal size.

→ The subproblems are solved recursively.

→ The solutions to the subproblem are combined to get a solution to the original problem.



* Recurrence for Divide and Conquer

→ A problem of size n is divided into 'b' instances, 'a' of them have to be solved.

Then, the General Divide & Conquer Recurrence is

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

↳ function that accounts for the time spent on dividing an instance of size n into instances of size n/b

Solving the recurrence

If $f(n) \in \Theta(n^d)$

$$T(n) \in \begin{cases} \Theta(n^d) & a < b^d \\ \Theta(n^d \log n) & a = b^d \\ \Theta(n^{\log_b a}) & a > b^d \end{cases}$$

① Mergesort

→ Sorts a given array $A[0 \ldots n-1]$ by dividing it into 2 halves $A[0 \ldots \lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor \ldots n-1]$, sorting each of them recursively, and then merging the 2 smaller sorted arrays into a single sorted one.

Algorithm

```
Mergesort (A[0 ... n-1])
if n>1
    copy A[0.. ⌊n/2⌋ - 1] to B[0.⌊n/2⌋-1]
    copy A[⌊n/2⌋ ... n-1] to C[0.. ⌊n/2⌋ - 1]
    mergesort ( B[0.. ⌊n/2⌋ - 1])
    mergesort ( C[0.. ⌊n/2⌋ - 1])
    merge (B, C, A)
Merge ( B[0.. p-1], C[0.q-1], A[0.. p+q-1])
  i ← 0, j ← 0, k ← 0
  while i < p and j < q
      if B[i] ≤ C[j]
          A[k] ← B[i]
          i ← i+1
      else
          A[k] ← C[j]
          j ← j+1
      k ← k+1
```

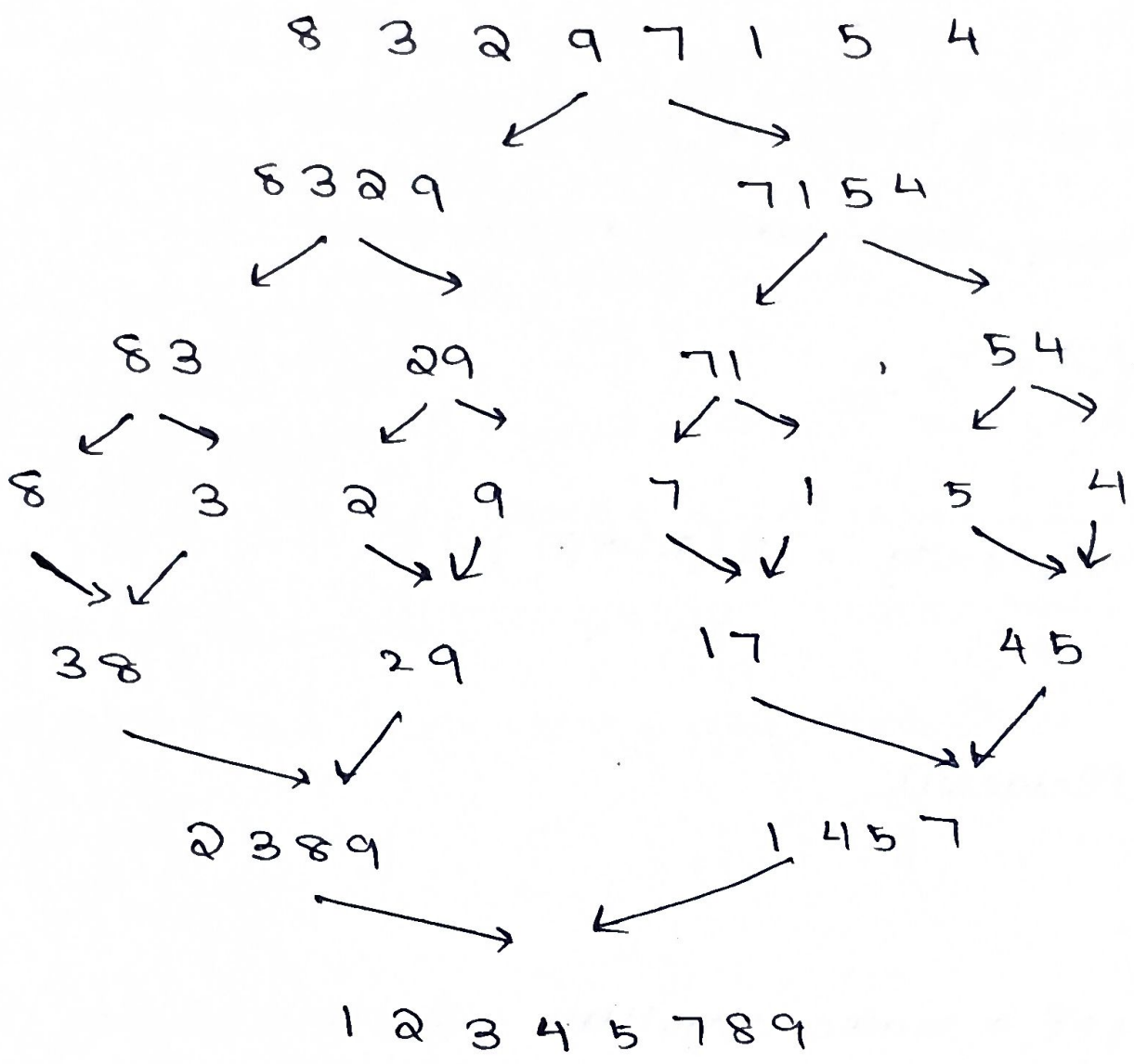
if i == p

    copy c[j .. q-1] to A[k .. p+q-1]

else

    copy B[i .. p-1] to A[k ... p+q-1]

Handtrace



Time Complexity Analysis

nlogn

$C(n) = 2C\left(\frac{n}{2}\right) + C_{merge}(n) \quad C(1)=0$

$C_{merge}(n)$ → no. of comparisons made during merging

a. Worst Case

$C(n) = 2C\left(\frac{n}{2}\right) + n-1$

consider 2 subarrays: [1, 3, 5] & [2, 4, 6], would have to do 5 diff. comparisons

Solution

here: $a = 2$

$b = 2$ $\qquad a = b^d$

$d = 1$

$$\Rightarrow \boxed{C_{worst}(n) = n \log n - n + 1}$$

Best Case

→ when the input array is already sorted or nearly sorted.

→ lesser no. of comparisons during merge step

$$C_{best}(n) = 2C\left(\frac{n}{2}\right) + \frac{n}{2}$$

$$\boxed{\text{Time Complexity} = O(n \log n)}$$

Advantages of Mergesort

⇒ Stability

→ The stability of a sorting algorithm refers to its ability to maintain the relative order of elements with equal keys. That is, if there are multiple elements with the same key in the input array, a stable sorting algorithm will ensure that the order of those elements remains the same in the sorted array.

→ In merge sort, if there are equal elements in the subarrays, the elements in the first subarray are placed before elements in the second subarray. ⇒ Relative order is preserved.

eg. [5A, 2, 5B, 1, 5C]

Resulting o/p => [1, 2, 5A, 5B, 5C]

→ Stability of mergesort is important in scenario where sorting is performed based on multiple criteria.

Disadvantage

→ requires a linear amount of extra storage

Variations of mergesort

(i) bottom up version – merge pairs of array's elements, then merge sorted pairs

(ii) multiway mergesort – divide a list to be sorted in more than 2 parts, sort each recursively, and then merge them together.

(2) Quicksort

→ works on the basis of a partition, where all elements to the left of some element A[s] are lesser than it, and all elements right of some element A[s] are greater than it.

→ After partition is achieved, A[s] would be in the correct position, ~~whereas~~ and the 2 subarrays to the left & right can be sorted again

Difference from mergesort

mergesort → work happens during merging

quicksort → work happens in the division stage, no work during merging

Algorithm Quicksort (A[l..r])

```
if l < r
    s ← Partition (A[l...r])
    Quicksort (A[l.. s-1])
    Quicksort (A[s+1...r])
```

Partition (A[l..r]) // Hoare's partitioning → pivot is the first element

```
p ← A[l]
i ← l  j ← r+1
do
    do i ← i+1 until A[i] ≥ p
    do j ← j+1 until A[j] ≤ p
    swap (A[i], A[j])
until i ≥ j
swap (A[i], A[j])      // undo last swap
swap (A[l], A[j])      // swap pivot & jth element
```

Working of Partition

(i) left to right scan (i) : elements < pivot in left subarray

(ii) right to left scan(j): elements > pivot in right subarray

After both scans stop, 3 cases arise:

A. If i and j have not crossed ⇒ swap A[i] and A[j], and continue scanning

B. If i and j have crossed over ⇒ subarray is partitioned, swap A[j] and A[l] (pivot)

c. If i = j, then they are both pointing to the pivot. → implement case B (swap A[l] and A[j])

Time Complexity Analysis

Best Case

→ If no crossover of i and j happens ⇒ no. of comparisons = n
(meaning pivot pos has been identified)

→ If there is a crossover ⇒ no. of comparisons = n+1

$$C_{best}(n) = 2C_{best}\left(\frac{n}{2}\right) + n$$

$$\Rightarrow C_{best}(n) \in \Theta(n \log_2 n)$$

Worst Case

→ all the splits will be skewed to one extreme
→ one subarray would be empty, another would contain all of the elements
→ happens in the case of a sorted array
→ Initially, there are n+1 comparisons
Here, the empty subarray requires no further processing, but the other subarray would be of size $[1 \ldots n-1]$
→ Next there would be $[2 \ldots n-1]$
→ Eventually, there would be 3 segments.

$$\Rightarrow C_{worst}(n) = (n+1) + (n) + (n-1) + (n-2) + \ldots 3$$

$$\rightarrow \frac{(n+1)(n+2)}{2} - 3 \Rightarrow \Theta(n^2)$$

Average Case Behaviour

→ depends on the position of the pivot

→ Initially, there are n+1 comparisons : a partition can happen at any position $(0 \le s \le n-1)$

→ The left & right subarrays will have s & n-1-s elements

→ Let the partition happen at any position s. ⇒ probability = 1/n

$$Cavg(n) = \frac{1}{n} \sum_{s=0}^{n-1} [n+1 + Cavg(s) + Cavg(n-1-s)]$$

$$\Rightarrow Cavg(n) \approx 2n \ln n$$

Handtrace

4	2	3	1	6	5	7
↑	↑					↑
pivot	l					r

move l to the right until an element greater than pivot is found

move r to the left until an element < than pivot is found

if l < p & r > p ⇒ swap

Comparison 1:

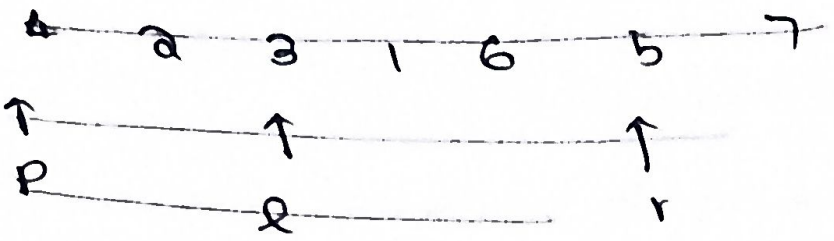
4	2	3	1	6	5	7
↑	↑					↑
p	l					r

no swap

⇒

4	5	3	1	6	2	7
↑	↑				↑	
pivot	l					

Comparison2:



no

Comparison1

i j

(4) 2 3 1 6 5 7

j i

(i) 4 2 3 1 6 5 7

i > j

swap pivot & A[j]

⇒ 1 2 3 4 6 5 7.

already sorted

i j

(6) 5 7

(i) (6) 5 7

i = j

swap pivot and A[j]

(ii) 5 6 7

i j

(10) 80 30 90 40 50 70

j i

(i) 10 80 30 90 40 50 70

~~80 10~~ j > i

swap A[j] and pivot

(ii) 10 | 80 30 90 40 50 70

i j

80 30 90 40 50 70

i < j

swap A[i], A[j]

i j

80 30 70 40 50 90

j i

80 30 70 40 50 90

swap A[j] and pivot

50 30 70 40 80 90

i j

(50) 30 70 40

i j

50 30 70 40

i j

50 30 40 70

j i

50 30 40 70

40 30 50 70

i j

(40) 30

30 40.

Variants of Quicksort

(i) randomized quicksort - uses a random element

(ii) median of three - uses the median of the leftmost, rightmost and middle element of the array

(iii) switch to insertion sort on very small subarrays

(3) Multiplication of Large Integers

= Karatsuba's Algorithm

→ Any no. when split into half can be expressed as:

$$x = a * 10^{n/2} + b$$

→ For the multiplication of 2 numbers x & y

$$x = a * 10^{n/2} + b$$

$$y = c * 10^{n/2} + d$$

$$x * y = (a * 10^{n/2} + b)(c * 10^{n/2} + d)$$

$$= (a * 10^{n/2}) * (c * 10^{n/2}) + ad * 10^{n/2} + bc * 10^{n/2} + bd$$

$$= ac * 10^{2n/2} + (ad + bc) * 10^{n/2} + bd$$

$$= ac * 10^{2n/2}$$ ⇒ reduces the no. of multiplications from n to n-1

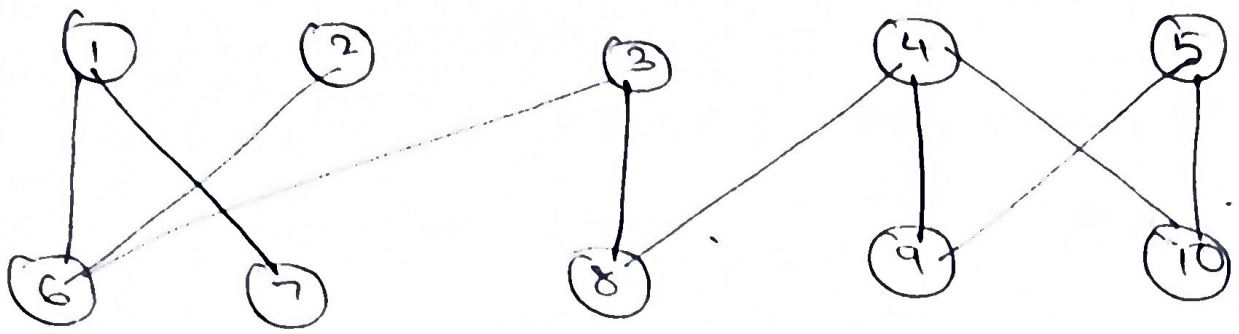
→ ac = karatsuba (a,c)

bd = karatsuba (b,d)

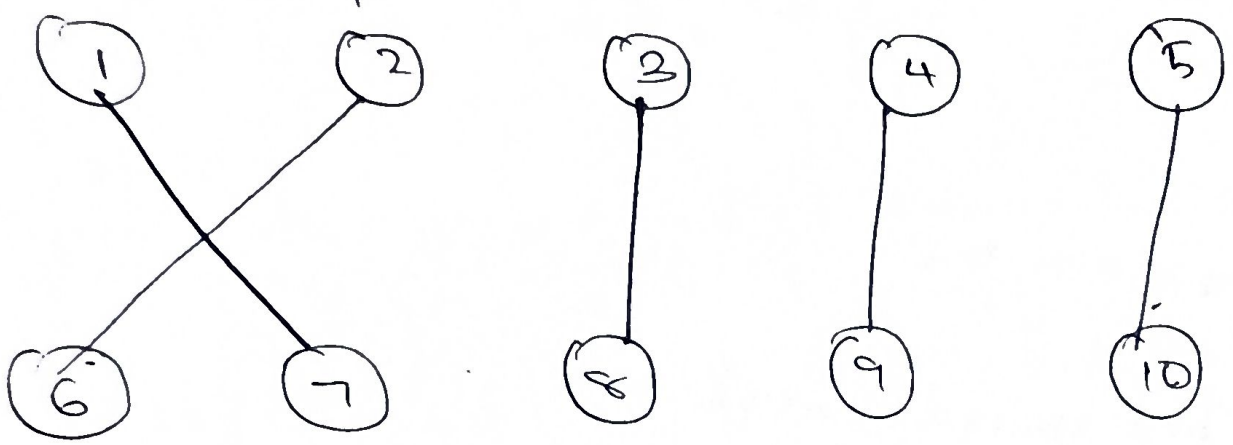
ad_plus_bc = karatsuba (a+b, c+d) - ac - bd

(3) Iterative Improvement : Maximum Matching in Bipartite Graphs.

eg.

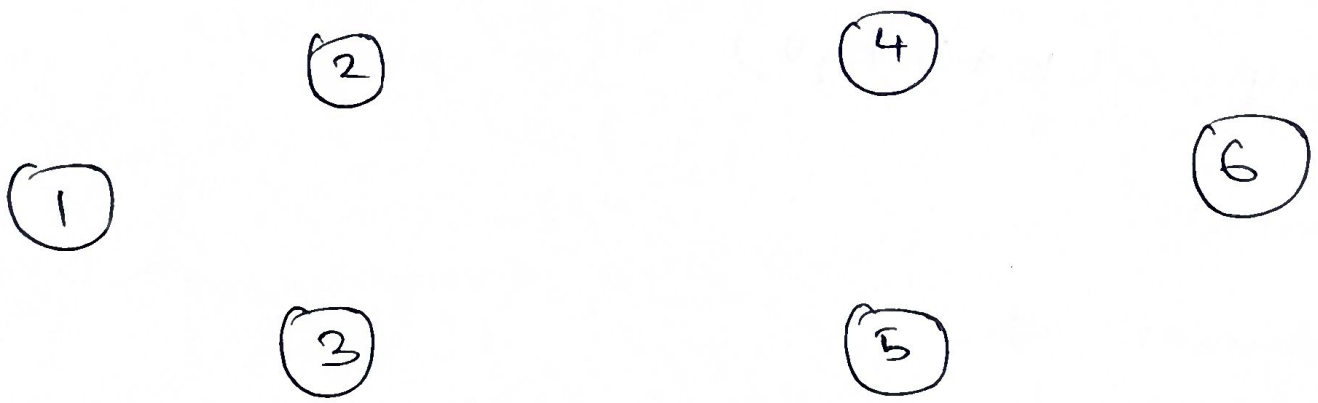


max. matching



(4) Iterative Improvement : Max Flow $O(|E| * f)$

max flow from source to sink



max-flow, min-cut : value of max flow - capacity of min cut

Ford-Fulkerson - uses aug. paths to produce residual networks, which gives rise to new aug. paths.

(3) Iterative Improvement: Stable Matching?

matching in a graph - no 2 edges share a vertex

max matching - matching w/ the largest no. of edges

perfect matching - every vertex of the graph is incident on exactly one edge of the matching

Stable marriage : $O(n^2)$

(6) Greedy - Prim's

find minimum spanning tree

look at node - find shortest edge

proceed, looking at lengths from all selected nodes

should not have cycles

running time w/ heap = $O(E \log V)$

fibonacci heap $O(E + V \log V)$

(7) Greedy - Kruskal's

→ can have disjoint sets

→ sort in inc. order

→ limited by time to sort sets

$O(E \log E)$ (w/ mergesort)

(8) Dijkstra's Algorithm

→use burning flame idea (init all vertices to ∞

while loop : $O(|V|)$

extract min $O(|V|)$ to find min element

$O(|V|^2)$

use a min. priority queue

fibonacci heap.

$O(|E| + |V| \log|V|)$

(9) Dynamic Programming - 0-1 Knapsack Problem

$$F(i,j) = \begin{cases} \max(F(i-1,), vi + F(i-1, j-wi) \\ F(i-1,j) \end{cases}$$

initial condns: $F(0,j) = 0$ $(F(i,0) = 0$

	0	0	0	0
0				
0				
0				

fractional knapsack = greedy

Time complexity = $O(nW)$

$W = O(2^n)$

NP complete

(10) Floyd - Warshall

$$d_{ij}^{(k)} = \min \{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \}$$

for k
  for i
    for j

$$D[i,j] \leftarrow \min \{ D[i,j], D[i,k] + D[k,j] \}$$

$\Theta(n^3)$

(11) DP - Ordering of matrices

→ min. no. of basic multiplications to find order

$$m[i,j] = \begin{cases} 0, & i = j \\ \min_{i \le k < j} & m[i,k] + m[k+1, j] + p_{i-1} p_k p_j \text{ if } i < j \end{cases}$$

$\Theta(n^2)$

(12) Binomial Coefficient

find coeff of x^4 in the expansion of $(1+x)^n$

value $= {}^nC_k$

$C(n,k) = C(n-1, k-1) + C(n-1, k)$

$$TC = O(nk)$$

(13) Fibonacci

$$fib(n-1) + fib(n-2)$$

memoization : top down solve subproblems & store results

bottom up — simply fill table

$$TC : O(2^n) \text{ to } O(n)$$

Master Theorem

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$f(n) = \Theta(n^k \log^p n)$

~~A. $f(n) = \Theta(n^{\log}$~~

A. $a > b^k \Rightarrow T(n) = \Theta(n^{\log_b a})$

B. $a = b^k$

$p > -1 \quad T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$

$p = -1 \quad T(n) = \Theta(n^{\log_b a} \log\log n)$

$p < -1 \quad T(n) = \Theta(n^{\log_b a})$

Unit -2

Divide and Conquer

1. Mergesort

→ recursively divide each input list into 2 nearly equal sublists

→ sort each sublist individually

→ repeatedly merge the sorted sublists

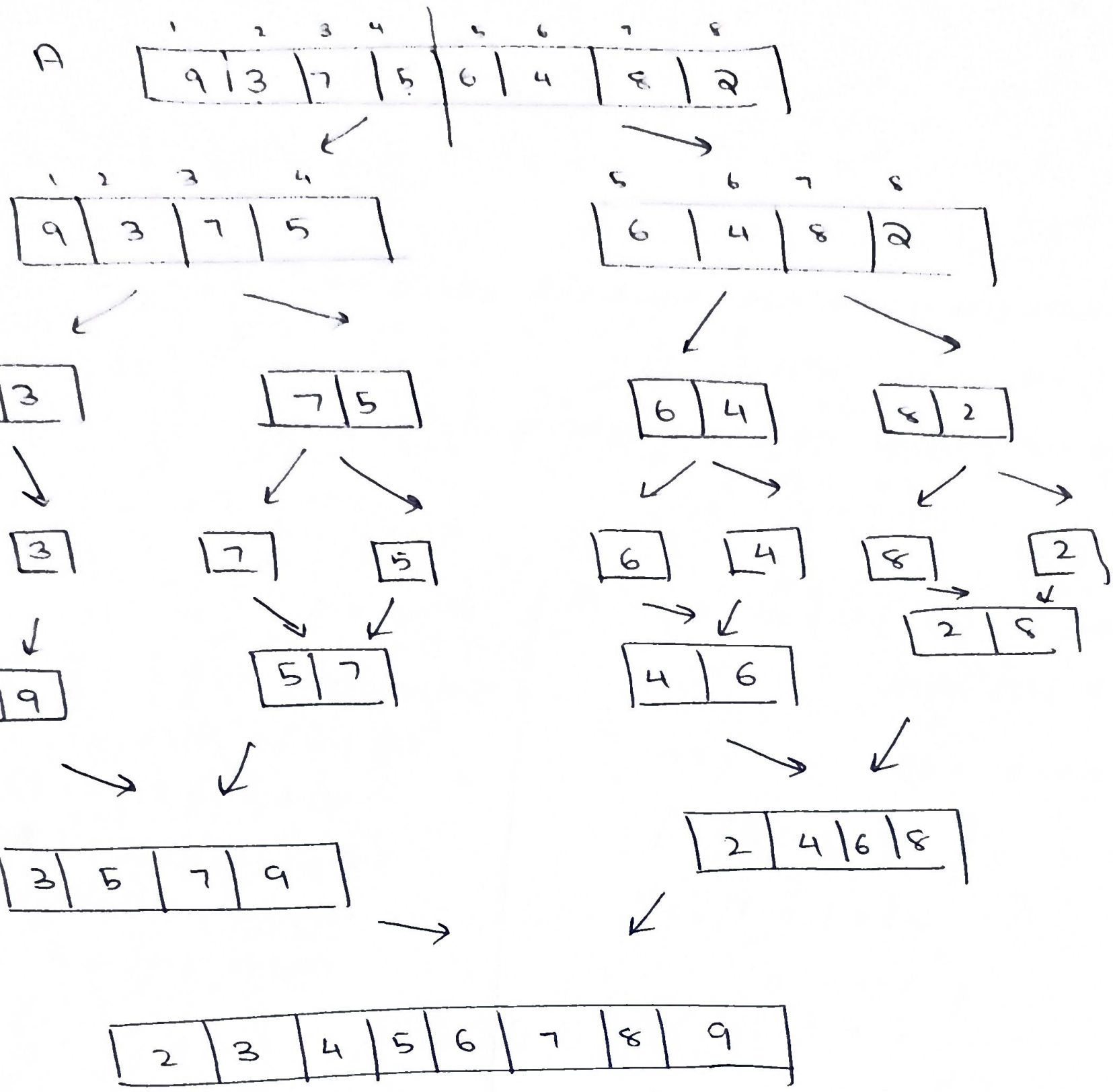
Algorithm

```
merge ( A , B , C , m , n)
    i , j , k ← 0

while i<m and j<n
    if A[i] < B[j]
        C[k] ← A[i]
        k++
        i++
    else
        C[k] ← B[j]
        k++
        j++

while i < m
    C[k] = A[i]
    k++
    i++
while j < n
    C[k] = B[j]
    k++
    j++
```

Meraesort

```
meragesort ( l,h)        l = 0
                         h ← len(l)
if (l < h)
    mid ← (l+h) /2
    mergesort ( l, mid)
    mergesort (mid+1 , h)
    merge ( l, mid, h)
```

HandTrace



Time Complexity Analysis

$T(n) \leftarrow$ mergesort (l, h)

If (l < h)

$T(1) \leftarrow$ mid = (l+h)/2

$T(n/2) \leftarrow$ mergesort (l, mid)

$T(n/2) \leftarrow$ mergesort (mid+1, h)

$n \leftarrow$ merge (l, mid, h)

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

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

$T(n) = 2\{2T(n/4)+1\} + n$

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

$= 4\{2T(n/8)+1\} + n$

$T(n/2) = 2T(n/4) + 1$

$T(n/4) = 2T(n/8) + 1$

$$= 8T(n/8) + k$$

$$\Rightarrow 2^k T(n/2^k) + k$$

$$\frac{n}{2^k} = 1 \qquad n = 2^k$$

$$k = \log n$$

$$\Rightarrow T(n) = T(1) + c$$

$$= n \log n + c$$

$$\boxed{O(n \log n)}$$

2. Quicksort

→ uses the divide and conquer technique

→ partitions the input list into 2 sublists

→ partitioning happens based on a pivot

→ one sublist holds value > pivot and the other holds values < pivot

HandTrace

	l								h
A	10	16	8	12	15	6	3	9	5

i (pivot), j

Step1

10	16	8	12	15	6	3	9	5

i, j

move i, since 16 > 10

move j, since 5 < 10

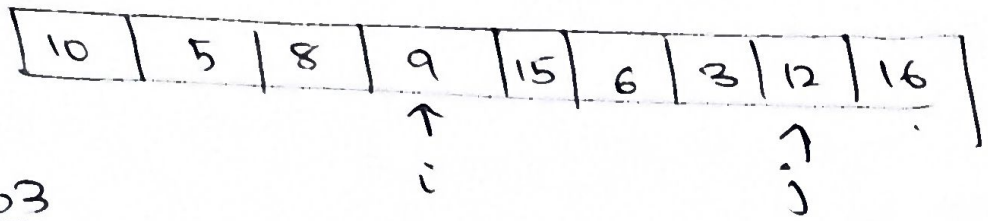
swap

10	5	8	12	15	6	3	9	16

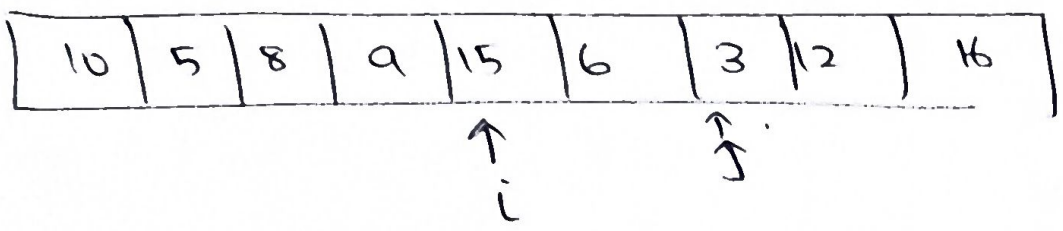
Step2

10	5	8	12	15	6	3	9	16

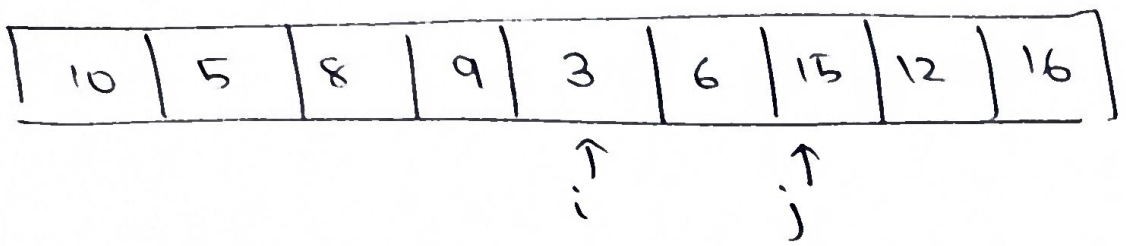
i, j



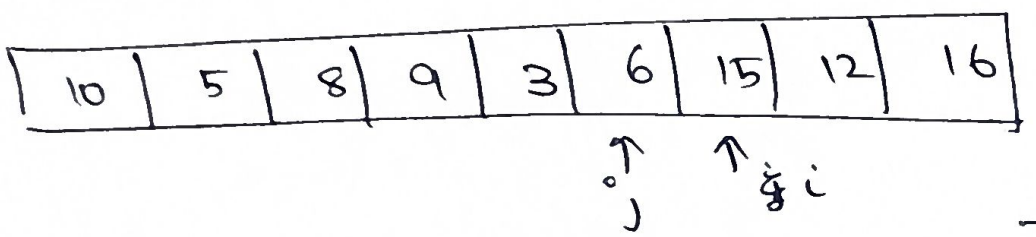
Step3



swap



Step 4

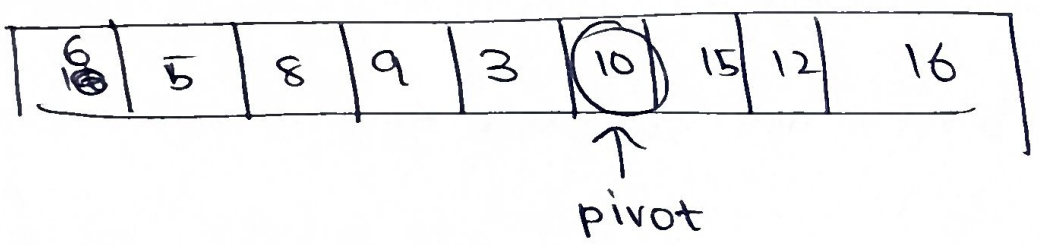


here $j < i$

$\Rightarrow$ don't swap

The pivot should be put at the pos of j

Step5



Algorithm

```
partition(l,h) {
  pivot = A[l]
  i = l    j = h    (len(A)
while(i<j) {
  do {
      i++;
  } while (A[i] ≥ pivot)

  do {j++;
  } while (A[j] ≤ pivot)
```

```
        if (i<j)
            swap(A[i], A[j])
    }
    swap(A[l], A[j])
    return j
}

Quicksort(l, h) {
    if (l<h) {
        j = partition(l,h)
        quicksort(l, j);
        quicksort(j+1, h);
    }
}
```

Time Complexity

→ same as that of merge sort = nlogn

worst case: when the pivot splits the list into sublists of length 0 and n-1

The recurrence would be:

$$T(n) = T(n-1) + T(0) + n$$

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

$$= O(n^2)$$

3. Multiplication of large Numbers - Karatsuba's Algorithm

→ a divide and conquer approach to multiply large nos.

Consider a no : 146123

Split into 2 parts 146 123

$= 146 \times 10^3 + 123$

In general: $\boxed{x = a \times 10^{n/2} + b}$

Now, consider 2 numbers to multiply

$x = a \times 10^{n/2} + b$

$y = c \times 10^{n/2} + d$

$x * y = (a \times 10^{n/2} + b) \cdot (c \times 10^{n/2} + d)$

$= ac \cdot 10^n + ad\ 10^{n/2} + bc\, 10^{n/2} + bd$

$\boxed{x*y = ac \cdot 10^n + (ad+bc)\, 10^{n/2} + bd}$

Now only ac, bd & ad + bc need to be computed, which can be done recursively.

This is done as:

ac = Karatsuba (a,c) → T(n/2)

bd = Karatsuba (b,d) → T(n/2)

ad_plus_bc = Karatsuba (a+b, c+d) - ac - ad → T(n/2)

since $(a+b)\cdot(c+d) - ac - bd$

$= ad + bc$

return

ac * (10**n) + (ad_plus_bc) * (10 ** (n/2)) + bd → T(n)

Time Complexity

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

Master Theorem:

$$T(n) = aT(n/b) + \theta(n^c \log^d n)$$

here $a = 3 \quad c = 1$

$b = 2 \quad d = 0$

$\log_b a = 1.59 \quad c < \log_b a \Rightarrow \theta(n^{1.58}) < \theta(n^2)$

4. Strassen's Multiplication for Matrices

→ regular matrix multiplication has a time complexity of $O(n3)$

Algo using divide & conquer

→ Assume that each matrix is a square matrix of size $n \times n$, where n is some power of 2.

→ Let X and Y be 2 $n \times n$ matrices such that each matrix can be split into 4 $n/2$ blocks as shown.

$$X = \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \quad Y = \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix}$$

Then multiply these smaller matrices

Time Complexity : There are 8 subproducts to be computed

$$T(n) = 8T(n/2) + \theta(n^2) \approx O(n3)$$

Strassen's Multiplication for Matrices

→ defines 7 new matrices

→ $T(n) = 7T(n/2) + \Theta(n^2)$

$$n^{\log_b a} = n^{\log_7 2}$$

$$n^{\log_b a} = n^{\log_2 7} = O(n^{2.8}) \;//$$

→ complexity reduces from $O(n3)$ to $O(n^{2.807})$

* **Backtracking** – A more intelligent version of the exhaustive search

→ construct solutions 1 component at a time

→ If the partially constructed solution can be developed further w/o violating any constraints, then take the first remaining option for the next component.

→ If there are no legitimate options for the next component, backtrack to the last component.

State Space Tree – used to represent backtracking

→ root = initial level

→ nodes in level 1 represent choices made for the first component, that in level 2 for the second component & so on.

→ Complete solutions are arrived at the leaves.

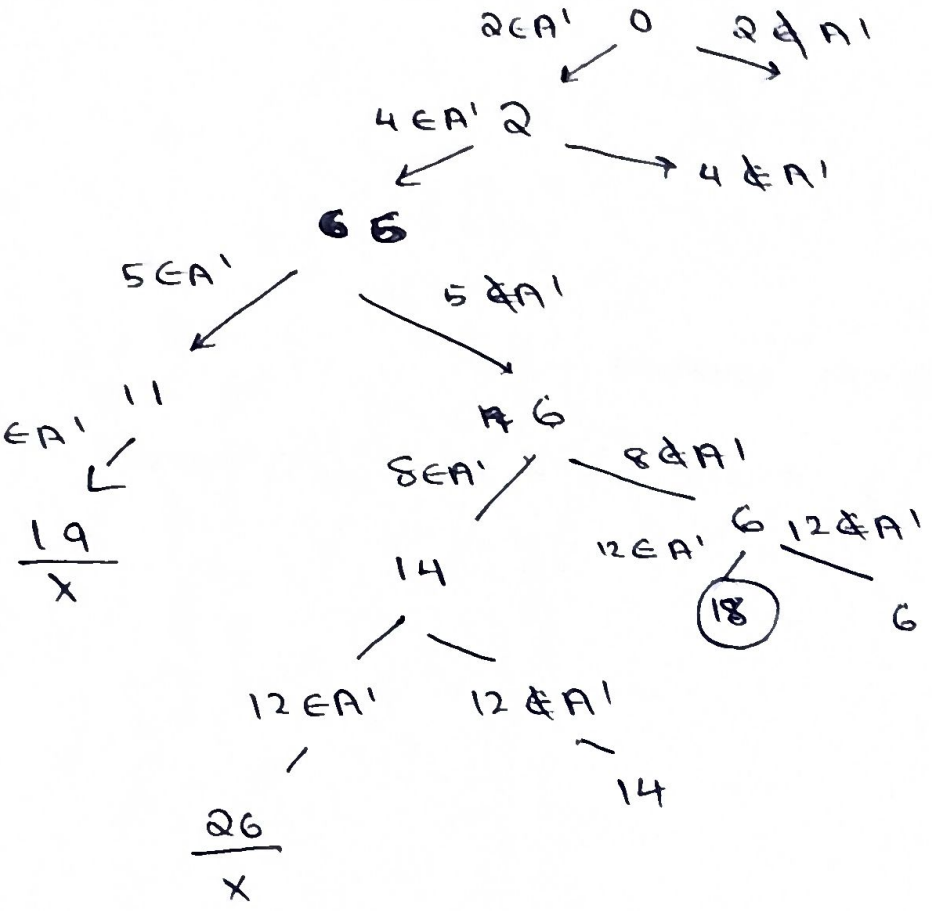
Subset - Sum using Backtracking

Given a list $A = \{a_1, a_2 \ldots a_n\}$ and >0, find a subset $A' \subseteq A$ such that the sum of elements in $A' = d$.

eg. If $A = \{1, 5, 8\}$ $d = 13$

$\Rightarrow A' = \{5, 8\}$

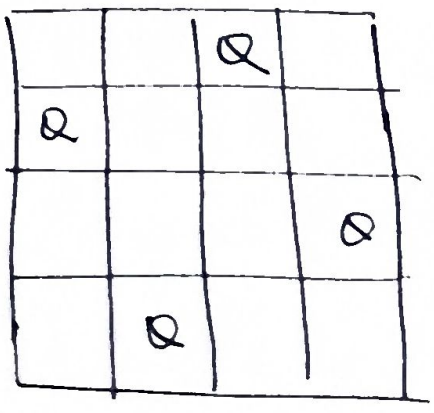
eg. Let $A = \{2, 4, 5, 8, 12\}$ $d = 18$



N-Queens Problem using Backtracking

→ Given an nxn chessboard, place n queens on the board such that no queen is under attack.

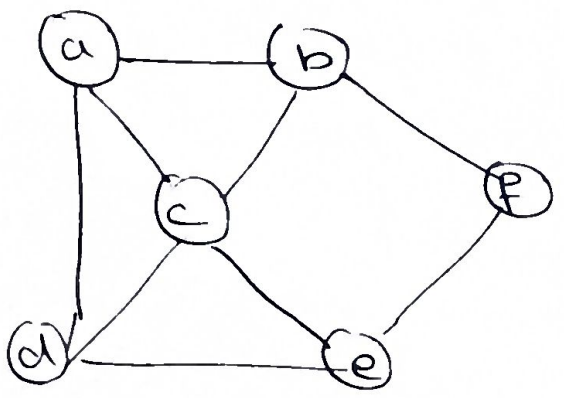
4x4



```
NQueens (k, n)
    For  i = 1 to n  do

        if (place (k, n)) then

            x[k] = i
            if (k == n)
                print (x[1:n])
            else
                Nqueens (k+1, n)

    x = [ ]

Place (k, i)

for  j = 1  to k-1

        if   x[j] = i          // same column
            or (abs (x[i] - i)) = abs (j-k)      // diagonal

                return false

return true
```

* Hamiltonian Circuit

Given an unweighted graph $G = (V, E)$, find a path in G that visits all the vertices of G exactly once before returning to the starting vertex.



Backtracking Approach