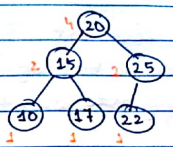
```
Record = node
{
  node *lchild;
  node *rchild;
  type  data;
  type  leftsize;
}
Algorithm Search(k)
{
  found := false;
  t := tree;
  while (t ≠ 0 and not found) do
  {
    if (k = (t→ leftsize)) then
      found := true;
    else if (k<(t→ leftsize)) then
      t := t→ lchild;
    else
    {
      k := k - (t→ leftsize);
      t := t→ rchild;
    }
  }
  if (not found) then return 0;
  else return t;
}
```



→ Given BST

Eg. k = 5

5 = 4 × 1 = 2 × 1 = 1 ✓

5 < 4 × 1 < 2 ✓

5 > 4 ✓ k = 5 - 4 = 1

∴ kth element = 22

Eg. k = 3

3 = 4 × 3 = 2 × 1 = 1 ✓

3 < 4 ✓ 3 < 2 ×

t = 15 3 > 2 ✓ k = 3 - 2 = 1 t : 17

∴ kth element = 17

NOTE :- Complexity wise, this algorithm is better tha
the Inorder + Linear Search Algorithm
- 'k' must always be less than or equal to no
of nodes in the tree

→

a) Algorithm to insert an element into BST

→ First search if element exists in BST. If yes, do not insert, else, create a temporary node with lc, rc = 0.

```
Record = node
{
  node *lchild;
  node *rchild;
  type data;
};
Algorithm Insert (T, x)
{
  if ((Search (T, x)) ≠ 0) the
  { write ("Element already exists");
    return 0; }
  else
  {
    node *temp := new node;
    temp→lc := NULL;
    temp→rc := NULL;
    temp→data = x;
    while (T != 0)
    {
      if (x < T→lchild)
        T := T→lchild;
      else
        T := T→rchild;
    }
```

```
Algorithm Search (x)
{
  found := false;
  p := tree;
  while (p ≠ 0 and not found)
  {
    q := p;
    if (x = p→data) then found := true;
    else if (x < p→data) then p := p→lchild;
    else p := p→rchild;
  }
  if (not found) then
  {
    temp := new Tree node;
    temp→lchild := 0;
    temp→rchild := 0;
    temp→data := x;
    if (tree ≠ 0) then
    {
      if (x < p→data) then
        p→lchild := temp;
      else
        q→rchild := p;
    }
  }
}
```

Deletion:

(a) No children → delete directly
(b) One child → delete and replace with child
(c) Two children → delete and replace by maximum element in left sub tree or minimum element in right sub tree

```
Algorithm Delete (T,x)
{
  if (T= NULL) then
    Write ("Element not found");
  else
  {
    if (x < T→ element data) then
      T→ left := Delete (x, T→ left)
    else if (x > T→ element) then
      T→ right := Delete (x, T→ right)
  }
  if
  {
    Temp: = Find min (T→ right);
    T→ element := Temp→ data;
    T→ right := Delete (T→ element, T→ right)
  }
else
{
  Temp := T;
  if (T→ left = NULL) then
    T := T→ right;
  else if (T→ right = NULL) then
    T := T→ left;
  free (Temp);
  }
}
```

Priority Queue (Heap)

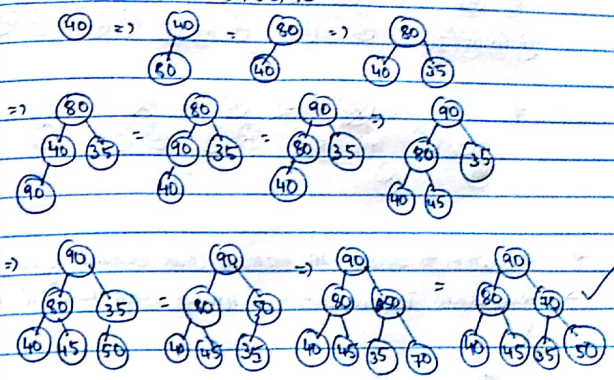
- Supports search, insert, delete (min & max)
- It is a complete binary tree.
- Root node is minimum than its children → min heap or min priority queue
  Root node is maximum than its children → max priority queue (max heap)
- Insert elements from left to right order
  - Structure Property
- Root node is max or min compared to children
  - Order Property

(2 properties of Heaps)

- Complete Binary Tree:
  For any node i: parent = ⌊i/2⌋, leftchild = 2i, rightchild = 2i+1
- Insertion:
  → Insert new element at nth position and verify the properties.
- Deletion:
  → In max heap, delete root node and replace it with the last node and check the properties.
- We must perform percolate up/down operations as required, while adjusting our heap.

Eg. Construct a max heap for the given set of elements:
40, 80, 35, 90, 45, 50, 70.

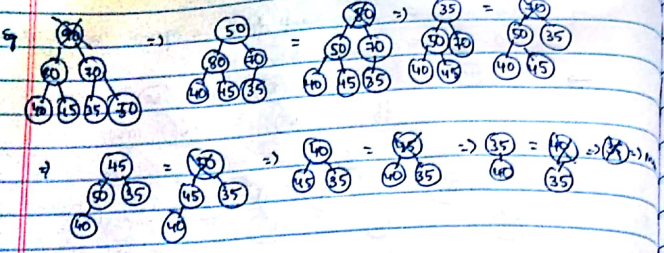
→



* Algorithm to insert an element into heap (max heap)

→ Algorithm Insert (a, n) // a[1 ..... n-1] parent heap, inserting at nth pos

```
{
    i := n;
    item := a[n];
    while (i > 1 and a[⌊i/2⌋] < item) do
    {
        a[i] = a[⌊i/2⌋];
        i := ⌊i/2⌋;
    }
    a[i] := item;
}
```



* Algorithm to delete an element (max element) from heap (max heap)
→ Algorithm Delmax(a,n,x) //Heap with max 'n' elements
{
  if (n=0) then
  {
    write ("Heap is empty");
    return false;
  }
  x := a[1];
  a[1] := a[n];
  Adjust (a,1,n-1);
  return true;
}

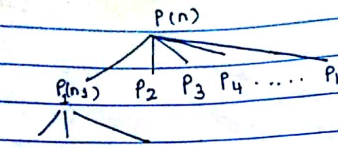
Algorithm Adjust(a,i,n) //Heap with max 'n' elements
{
  j := 2i;
  item := a[i];
  while (j ≤ n) do
  {
    if (j<n and a[j] < a[j+1]) then
      j := j+1;
    if (item ≥ a[j]) then break;
    a[⌊j/2⌋] := a[j];
    j := 2j;
  }
  a[⌊j/2⌋] := item;
}

* Algorithm to perform Heap Sort.
→ Construct heap, apply delete max operation, place in nth location, apply delete max operation, place in (n-1)th location in array, continue till all elements are exhausted, i.e., 1st location is full / heap is empty.
→ Algorithm Sort(a,n)
{
  for i := 1 to n do
    Insert(a,i);
  for i := n to step -1 do
  { Delmax(a,i,x);
    a[i] := x; }
}

03-01-18

UNIT-II: DIVIDE AND CONQUER AND GREEDY METHOD

DIVIDE AND CONQUER



→ P is problem with input size 'n', if 'n' is large divide it into 'k' sub problems of same nature $P_1, P_2, \ldots P_k$ and continue it until problem size becomes small. Small → solved provided without splitting problem defined in our own way depends on nature of problem

* Control Abstraction of Divide And Conquer:

```
Algorithm DandC(P)
{
  if Small(P) then return S(P) // if problem size is small, solves
  else
  {
    Divide P into sub problems P1, P2, ...., Pk
      then apply DandC Procedure
    return Combine(DandC(P1), DandC(P2), ...., DandC(Pk))
  }
}
```

→ We get recurrence relations using D&C
→ Merge Sort and Quick Sort use D&C

⇒ Divide and Conquer
→ Finding Minimum and Maximum
→ Merge Sort
→ Quick Sort
→ Selection
→ Strassen's Matrix Multiplication

Finding maximum and minimum

```
* Algorithm StraightMaximum(a,n)
{
  max := min := a[i];
  for i := 2 to n do
  {
    if (a[i] > max) then max := a[i];
    if (a[i] < min) then min := a[i];
  }
}
```

→ No. of comparisons to decide maximum element = n-1
No. of comparisons to decide minimum element = n-1
∴ Total no. of comparisons required = 2(n-1)
→ Takes maximum time, i.e., maximum cost to compare, thus very costly.
→ n = 1 comparisons = 0
n = 2 comparisons = 1
n > 2 sub problem1 (i, (i+j)/2, max, min) sub problem2 ((i+j)/2, j, max, min) | (i, j, max, min) = (low, high, max, min)
divide into 2 sub problems

Using divide and conquer

$T(n) = T\lfloor n/2 \rfloor + T\lfloor n/2 \rfloor + 2$ (2 to divide problem and 2 to combine the solution)

$= 2T\lfloor n/2 \rfloor + 2$

$= \vdots$

$= 3n/2 - 2$

$\therefore \boxed{T(n) = 3n/2 - 2}$ (by divide & conquere) $< \boxed{T(n) = 2(n-1)}$ (by straight method) $\Rightarrow \Theta(n)$ Asymptotically

→ Algorithm MaxMin (i, j, max, min)

```
{
  if (i = j) then max := min := a[i]; //if problem size is 1
  else if (i = j-1) then //if problem size is 2
  {
    if (a[i] < a[j]) then //single comparison
    {
      max := a[j];
      min := a[i];
    }
    else
    {
      max := a[i];
      min := a[j];
    }
  }
```

```
  else //if problem size is greater than 2
  {
    mid := ⌊(i+j)/2⌋; //divide problem into two halves
    MaxMin(i, mid, max, min);
    MaxMin(mid+1, j, max1, min1);
    //combine the solution
    if (max < max1) then max := max1;
    if (min > min1) then min := min1;
  }
}
```

Eg. $n = 2^k$

$T(n) = T\lfloor n/2 \rfloor + T\lfloor n/2 \rfloor + 2$

$= 2T\lfloor n/2 \rfloor + 2$

$= 2[T\lfloor n/4 \rfloor + T\lfloor n/4 \rfloor + 2] + 2$

$= 2T\lfloor n/4 \rfloor + 6$ $(6 = 2^3 - 2)$

$= \vdots$

$= 2^{k-1}$

$= T[2] + \sum_{i \le j \le k-1} 2^i$

$= 2^{k-1} + 2^k - 2$

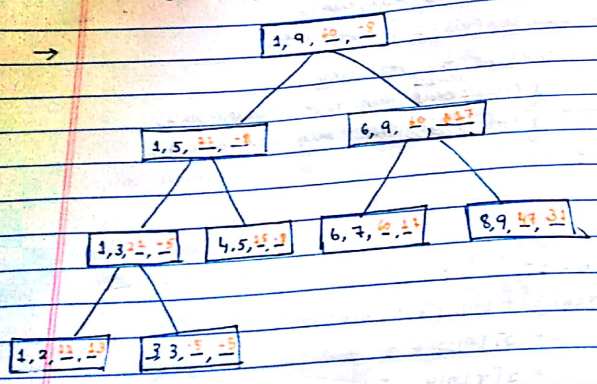
$= 3n/2 - 2$

→ In some cases, straight max min is better than D&C max min because space wise, it occupies 4 stack spaces for each recursive call. Comparison wise, D&C max min is better.

Eg. Trace D & C maxmin for given set of elements and construct a recursive tree for it.

a = {22, 13, -5, -8, 15, 60, 17, 31, 47}
1 2 3 4 5 6 7 8 9

→



→ Space complexity wise, straight algorithm is better.

→ No. of comparisons wise, D and C algorithm is better.

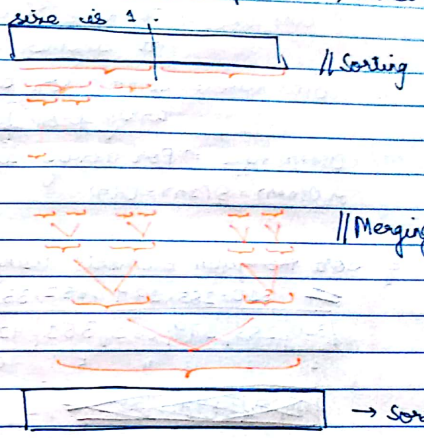
P.T.O. ⟶

08-01-18

* Merge Sort

→ Divide problem into two sub problems and continue doing so with the problems, until the problem size is 1.

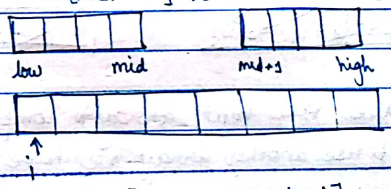
→ Eg.



→ Merge Sort takes 3 parameters

Merge (low, mid, high)

↳ a[low] to a[mid] -> Sorted
a[mid + 1] to a[mid] -> Sorted
} (Merging / Combining them)



Compare: a[low] with a[mid + 1]

If a[low] < a[mid + 1], b[i] = a[low], i++, low++

If a[low] > a[mid + 1], b[i] = a[mid + 1], i++, ...

... continue till last

Recurrence Relation For Merge Sort:

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

(For moving and comparisons $O(n)$)

$O(n)$ time => To move 'n' elements from one array to another array.

$O(n)$ time => Comparison before moving into desired array from first array.

$O(n+n)$ time => For moving and comparison

$\Rightarrow O(n+n) = O(2n) = O(n)$

Eg. Sort the given elements using Merge Sort.

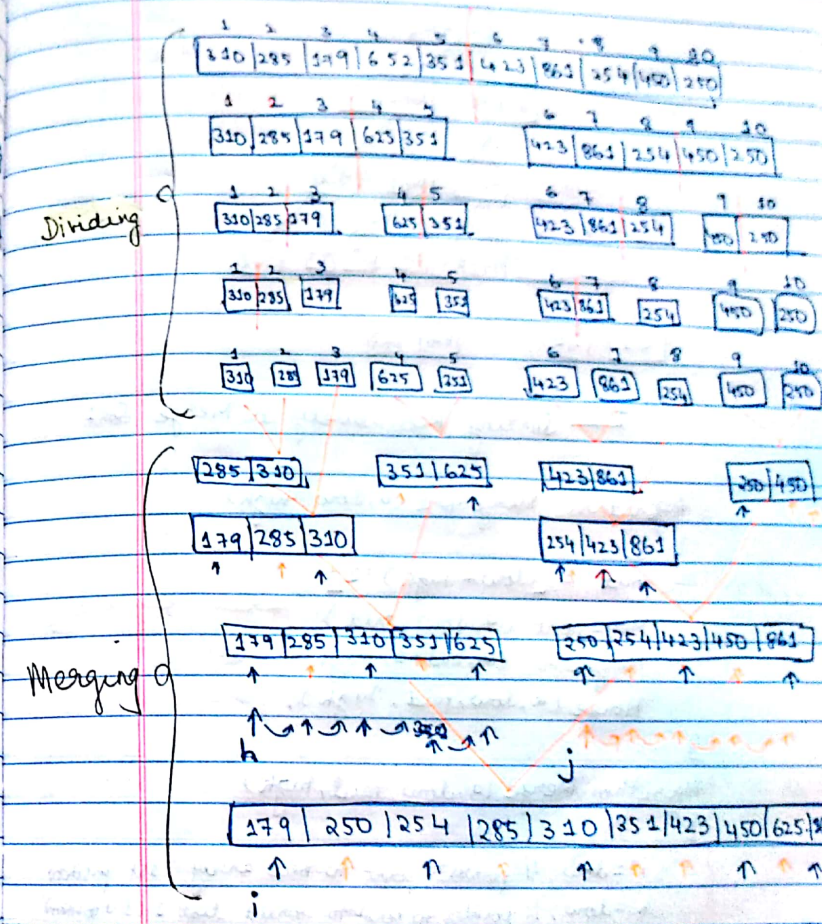
310, 285, 179, 652, 351, 423, 861, 254, 450, 250

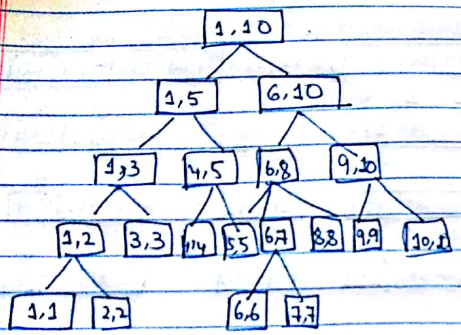
1	2	3	4	5	6	7	8	9	10
310	285	179	652	351	423	861	254	450	250

low = 1, high = 10

mid = (low + high)/2 = (10+1)/2 = 11/2 = 5.5 = 5

NOTE: We take new array to compare and sort. If we don't use another array, we have to use another technique for sorting, and merge sort complexity may increase then.





← Sorting Recursively in Merge Sort

```
Algorithm MergeSort (a, low, high)
{
    mid := ⌊(low + high)/2⌋;
    MergeSort (a, low, mid);      // recursively dividing until problem size is 1
    MergeSort (a, mid+1, high);
    Merge (a, low, mid, high);
}
Algorithm Merge (a, low, mid, high)
{
    i := low; // pointer to point to new array (1st position)
    h := low; // pointer to point to sorted list 1 (1st position)
    j := mid + 1; // pointer to point to sorted list 2 (1st position)
    while (h ≤ mid and j ≤ high) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i] := a[h];
            h = h+1;
        }
        else
        {
            b[i] := a[j];
            j = j+1;
        }
        i := i+1;
    }
    if (h > mid)                  // to check if any elements are present in array 2
        for k := j to high do
        {
            b[i] = a[k];
            i := i+1;
        }
    else // to check if any elements are present in array 1
        for k := h to mid do
        {
            b[i] := a[k];
        }   i := i+1;

        for k := low to high do
        {
            a[k] := b[k];
        } // to copy elements from array b to a
}
```

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + n & \text{if } n > 1 \end{cases}$$

Worst case

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

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

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

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

$= T(n-3) + 2n - 2 - 1$

$\vdots$

$= T(n-k) + kn - (k+1)$

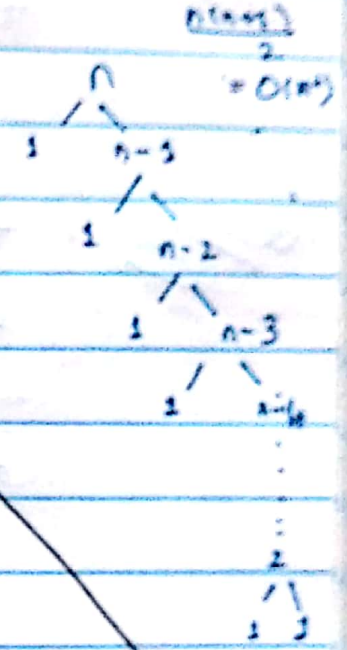
Let $k = (n-1)$

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

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

$= 1 + n^2 - 2n$

$= O(n^2)$



$= T(n-3) + n - 2 + 2n - 1$	$T(n-2) = T((n-2)-1) + n-2$
$= T(n-3) + 3n - 3$	$T(n-2) = T(n-3) + n-2$
$= T(n-4) + n - 3 + 3n - 3$	$T(n-3) = T((n-3)-1) + n-3$
$= T(n-4) + 4n - 6$	$T(n-3) = T(n-4) + n-3$

$\vdots$

$= T(n-k) + kn$

$\vdots$

$T((n-k)+1) = T(n-k) + (k-1)n + \frac{k(k-1)}{2}$

$T(n-k) = T((n-k)-1) + kn + \frac{k(k+1)}{2}$

Best case

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

↳ Pivot element taken divides into two two equal halves, n time for dividing.

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

$= 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n$ Sub. n by n/2

$= 4T\left(\frac{n}{4}\right) + \frac{2n+n}{2}$

$= 4T\left(\frac{n}{4}\right) + 2n$

$= 2^2 T\left(\frac{n}{2^2}\right) + 2n$ Sub. n by n/2

$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$

$\vdots$

$= 2^k T\left(\frac{n}{2^k}\right) + kn$

Let $n = 2^k \Rightarrow \log n = k \log_2 2 \Rightarrow \log n = k$

$= 2^k T\left(\frac{2^k}{2^k}\right) + k \cdot 2^k$

$= 2^k T(1) + k2^k$

$= 2^k \cdot 1 + kn$

$= 2^k + kn$

applying log on both sides

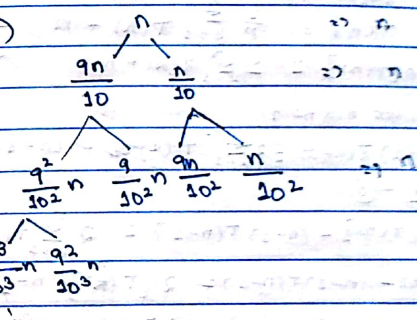
$= n \cdot 1 + n \log n$

$= O(n \log n)$

eg

$T(n) = T(9n/10) + T(n/10) + n$

(iterative method)



max term $= \frac{q^k}{10^k} n \quad \frac{q^{k-1}}{10^k} n$

$n = \left(\frac{10}{9}\right)^k$ max term = 1

Applying log.

$\log_{10/9} n = \log_{10/9} (10/9)^k$

$k = \log_{10/9} n$

Average case

$T(n) = \frac{1}{n}\sum_{k=1}^{n}[T(n-k) + T(k)] + n$

↳ k partitions, 'n' time for partion, 1/n because each element has same probability to pick as pivot element.

$T(n) = \frac{1}{n}\sum_{k=1}^{n}[T(n-k) + T(k)] + n$

on average when $T(n-k) = T(k)$

$T(n) = \frac{2}{n}\sum_{k=1}^{n} T(k) + n$ — (1)

$n[T(n)] = 2\sum_{k=1}^{n} T(k) + n^2$ — (2)

Let $n = n-1$

$(n-1)T(n-1) = 2\sum_{k=1}^{n-1} T(k) + (n-1)^2$ — (3)

(2) − (3)

$\Rightarrow n[T(n)] - (n-1)T(n-1) = 2\sum_{k=1}^{n} T(k) + n^2 - 2\sum_{k=1}^{n-1}[T(k-1)] + (k-1)^2$

$\Rightarrow nT(n) - (n-1)T(n-1) = 2[T(n)] + n^2 - (n-1)^2$

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

$\Rightarrow nT(n) = (n+1)T(n+1) + 2n$ — (4)

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

$\Rightarrow \frac{T(n)}{(n+1)} = \frac{T(n-1)}{n} + \frac{2n}{(n+1)}$

$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2}{3}c$

$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c\sum_{i=3}^{n}\frac{1}{i}$

$O(n \log n)$

Selection (Selection Problem)

→ (Selecting) Finding kth minimum element from given set of elements; using divide and conquer approach.

→ We use partition algorithm. It returns 'j' value, i.e. position of pivot element in sorted array.

→ Compare 'j' value with 'k' value

~~If k<j~~ Selection (a, n, k)

↳ array, n no. of ele, Kth min ele

Initially, low := 1, up := n

If k < j — up := j − 1;

If k = j — Required element found

If k > j — low := j + 1;

→ Algorithm Selection(a, n, k)

```
{
  low := 1; up := n; a[n+1] := ∞;
  j := Partition(a, low, n+1);
  if (k < j)
    up := j - 1;
    Selection (a, low, j);
  else if (k > j)
    low := j + 1;
    Selection (a, low, n+1);
  else
    return j;
}
```

Eg: 43, 53, 65, 35, 23, 18, 1, 85, 99

↳ Find 3rd minimum element

Partition => j = 4

v := 43, i = 1 j = 9

i = 2 j = 8, 7, 6, 5, 4

25	1	18	35	43	65	53	88	99
↑ low								

Partition => j = 3 => Required element = 25

Time Complexity:

For each Partition → (n-1) time

To Find kth min ele → At most n partitions

Worst case: $O(n^2)$

Average Case: $O(n)$

Strassen's Matrix Multiplication

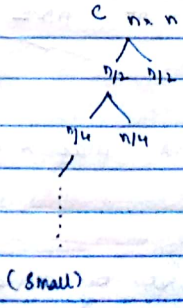
$C_{n \times n} = A_{n \times n} \times B_{n \times n}$

$C(i,j) = \sum_{1 \le k \le n} A_{ik} * B_{kj}$

↳ Requires 'n' multiplications, n^2 elements

=> $O(n^3)$ => Conventional Method

By divide and conquer



Q. $A_{4\times4} \times B_{4\times4}$

a_{11}	a_{12}	a_{13}	a_{14}	b_{11}	b_{12}	b_{13}	b_{14}
a_{21}	a_{22}	a_{23}	a_{24}	b_{21}	b_{22}	b_{23}	b_{24}
a_{31}	a_{32}	a_{33}	a_{34}	b_{31}	b_{32}	b_{33}	b_{34}
a_{41}	a_{42}	a_{43}	a_{44}	b_{41}	b_{42}	b_{43}	b_{44}

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$c_{11} = a_{11} * b_{11} + a_{12} * b_{21}$

$c_{12} = a_{11} * b_{12} + a_{12} * b_{22}$

$c_{21} = a_{21} * b_{11} + a_{21} * b_{21}$

$c_{22} = a_{21} * b_{12} + a_{22} * b_{22}$

} 8 multiplications, 4 additions (n^2)

$$T(n) = \begin{cases} b & n \le 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

$a T(n/b) + f(n)$

$n^{\log_2 8} = n^3$:) Master Method

$n^3 > n^2$

$\Rightarrow O(n^3)$

↳ Time complexity by divide & conquer

Strassen's Multiplication

→ Method in which time complexity is $O(n^{2.81})$

→ It requires 7 multiplications, 18 additions or subtractions

→ $$T(n) = \begin{cases} b & n \le 2 \\ 7T(n/2) + cn^2 & n > 2 \end{cases}$$

$n^{\log_2 7} = n^{2.81}$

$n^{2.81} > n^2$:) Master Method

$\Rightarrow O(n^{2.81})$

↳ By Strassen's Method

$P = (a_{11} + a_{22}) * (b_{11} + b_{22})$

$Q = (a_{21} + a_{22}) * b_{11}$

$R = a_{11} * (b_{12} - b_{22})$

$S = a_{22} * (b_{21} - b_{11})$

$T = (a_{11} + a_{12}) * b_{22}$

$U = (a_{21} - a_{11}) * (b_{11} + b_{12})$

$V = (a_{12} - a_{22}) * (b_{21} + b_{22})$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

→ 7 multiplications

18 subtractions / additions

→ Multiplications are more costly than addition / subtraction

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U$

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ 1 & 2 \\ 3 & 4 \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ 5 & 6 \\ 7 & 8 \end{bmatrix}$$

$$C = A \times B = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

$P = 5 \times 13 = 65 \quad Q = 7 \times 7 = 49 \quad R = 1 \times -2 = -2 \quad S = 4 \times 2 = 6$

$T = 3 \times 8 = 24 \quad U = 2 \times 12 = 24 \quad V = 2 \times 15 = 30$

$c_{11} = P + S - T + V = 65 + 6 - 24 + 30 =$

$c_{12} = R + T = -2 + 24 = 22$ ✓

$c_{21} = Q + S = 49 + 6 = 55$

$c_{22} = P + R - Q + U = 65 + (-2) - 49 + 24$

22-01-19

GREEDY METHOD

If a problem 'P' takes 'n' number of inputs we select a subset of inputs from 'P' that follow problem constraints. This subset of input set satisfying some constraints is called feasible solution. Optimal solution is feasible