UIV – DAA

Page No.
Date

Backtracking Branch & Bound

Backtracking - It is a general algo technique for solving computational problems incrementally by trying out various possibility and abandoning those that fail to meet the problem's constraint. The idea is to explore the solution space systematically backtrack when necessary, and try alternative choices.

> **Principles of Backtracking** -

1) The solution is built incrementally by making a series of choices at each step decision is made.
2) If a chosen path leads to a state that can't possibly be part of valid soln the algo backtracks and abandons the path (Prune).
3) The Soln Space is explored systematically, often using recursion. Algo explores one branch of decesion tree at a time.

> **Control Abstraction** -

1) In backtracking, the soln is defined as an tuple $x = (x_1, x_2, \ldots x_n)$ where $x_i = 0$ or 1 x_i is chosen from set of Finite components S_i. If component $x_i = 1$ else set it to 0. The backtracking approach tries to Find vector x such that it maximizes or minimizes certain criterian Fonc^n

B is Bounding Fonc^n.

Recursive approach - Algorithmn.

```
1. Algorithm Backtrack(k)
{ For each x[k] ∈ T(x[1], x[2] ... x[k-1]) do
{ if (Bk(x[1] ... x[k] ≠ 0) then
{ if (x[1], x[2] ... x[k] is a path to answer node)
        then write (x[1:k]);
  if (k < n) then Backtrack(k+1);
}
}
}
```

> Complexity Analysis -

1) Time to compute next candidate sotation x[i]
2) Number of candidate Soln x[i] Fulfiling the explicit constraints.
3) Time taken by bounding Func^n B_i to check promising candidate Soln
4) No. of candidate Soln x[i] ~~fulfiling~~ fulfiling satisfying the bounding Func^n B_i.

> For a problem instance of size n, if the backtracking algo produces a Soln space of 2^n or n! nodes, then it has worst case running times $O(p(n)2^n)$ or $O(q(n)n!)$ respectively where p(n) & q(n) are polynomials in n.

>

* Basic Backtracking terminologies -

> Solution vector - The desired Soln X to a problem instance P of input size n is expressed as vector of candidate Soln x_i that are selected from finite set of possible Soln S. Thus a Soln can be represented as an n-tuple $(x_1, x_2, \ldots x_n)$ & partial Soln $(x_1, x_2, \ldots x_i)$

> Constraints - This are the rules to confine the Soln vector $(x_1, x_2, \ldots x_n)$. They determine the values of candidate soln and their relationship with each other.

2 types
1) Implicit & 2) Explicit.

> Implicit Constraints -

1) These are the rules that identify the tuples in the solution space S that satisfy the specified criterian fcm^n of problem instance P.

2) example N queen.

> Explicit Constraints - This are the rules by which all candidate Solu^n x_i's are restricted to take on values from a specified set in a problem instance P.

Iterative backtracking Approach.

```
Algorithm Backtrack()
{   k := 1
    while k ≠ 0 do
    if (untried x[k] ∈ T[x[1], x[2] ... x[k-1]]) AND
     (Bk(x[1], x[2] ..., x[k]) is path to answer Node) then
       Print x[1] x[2] ... x[k]  // Soln
        k ← k+1  //Consider next candidate
      else
         k ← k-1  // Backtrack to recent Node
      end
    end.
```

Page No.
Date

N-queen – The N queen is a classical combinatorial problem that asks for all arrangements of N queen on NxN chessboard, such that no two queens threaten each other.

Consider a nxn chessboard ~~on which~~.

1) Initially chess board is empty

2) (Reasons why backtracking is needed explain in own words first without Backtrack then with backtrack).

Pseudocode –

```
Algorithm Queen (n)
  For column ← 1 to n do
    { if (Place (row, column)) then
      { board[row][column] //no conflict, place queen
        if(row = n) then //dead end
        print board (n) //Printing board
        else // try next queen with next posi?
          Queen (row+1, n)
      }
    }
```

↑ Row by Row each queen is placed by satisfying constraints.

```
Algorithm place (row, column)
  for i ← 1 to row-1 do
    { if (board[i] = column) then
      return 0. // Check in same column.
    else if (abs(board[i] - column) = abs (i - row)) then
      return 0 // check diagonally
  // no conflicts then queen can be placed.
  return 1
```

Graph Colouring Problem - Consider a Graph G(V,E) V is set of vertices & E is set of Edges. Then the graph coloring problem asks to assign m colors to the vertices of G such that no two no neighbouring vertices have a similar color.

Pseudocode -

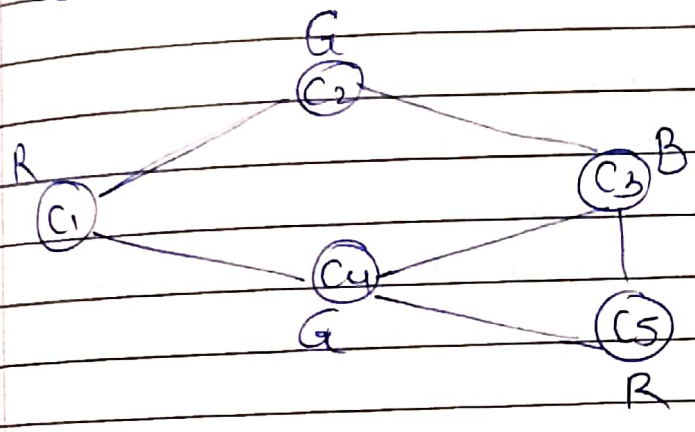
```
Algorithm ColorGraph(G,k):
    colors = [0] * |V|
    colorVertex(G,0,k,colors)

Algorithm colorVertex(G,vertex,k,colors):
    if vertex is to |V| then
        return True. //All vertices are successfully colored
    For color From 1 to k do
        if isSafe(G,vertex,color,colors) is true then
            //Assign color to vertex
            colors[vertex] = color
            if colorvertex(G,vertex+1,k,colors) is true then
                return True. // Coloring successful.
            // if coloring fails
            colors[vertex] = 0
    // if no color is suitable return False.
    return False.

Algorithm isSafe(G, vertex, color, colors):
    For adjacent vertex in neighbors(G,vertex) do
        if colors[adjacentVertex] is equal to color then
            return False //color conflict
    // If no conflict are found, color is safe.
    return True
```

Page No.
Date

Q-

	C_1	C_2	C_3	C_4	C_5
C_1	0	1	0	1	0
C_2	1	0	1	0	0
C_3	0	1	0	1	1
C_4	1	0	1	0	1
C_5	0	0	1	1	0

Color = R G B.



Page No.
Date

Branch & Bound - It systematically explores the Solu^n Space of a problem dividing it into smaller subproblems, solving them & bounding the Solu^n Space to eleminate subproblems that can't lead to an optimal Sol^n

> Characteristics -

1) Divide & Conquer -
2) Explora^n of Solu^n space
3) Bounding Techniques - It involves establishing upper ~~bou~~ & lower bounds. Subproblems that cannot lead to optimal Solu^n then they are pruned.
4) Priority Queue - Algo uses priority Queue
5) Branching & Pruning
6) Termina^n criteria.

> General Algo -

```
Algorithm Branch & Bound ()
    E ← CreateNode
    while True do
        if E is the leaf node then
            print "Display path from E"
            return
        end
        Expand (E)
        if Size (Heap) = = 0 then
            Print "Soln not found"
            return
        end
        E ← deque (H)
    end
```

```
Algorithm Expand (E)
Generate all
possible child
nodes of E
Compute the cost
of each child C.
Insert each child
C in Heap.
```

Search Techniques -

1) LC search (Least Cost Search) - It uses heuristic cost fun^{n} to compute the bound values at each node. Nodes are added to the list of live nodes as soon as they get generated. The node with least value of a cost fun^{n} is selected as a next E-node.

2) BFS FIFO - It maintains the list of live nodes in first in first out ~~order~~ order in a queue.

3) DFS LIFO - Uses Stack

> Control Abstraction for FIFO / BFS

```
Algorithm BB-FIFO()
{ if T is the answer node Then
  { ~~u := min(cost[T], ub[T] + E)~~
    write T
    return;
  }
E := T;      // E-node
Initialize empty Queue of live nodes.
repeat
{ For (each child K OF E)
  { if (K is an answer state)
    { write path from K to T;
      return;
    }
  Append Q(K);
(K → Parent) := E;
}
if (there are no more live nodes)
{ write ("No answer state");
    return; }
E := Delete_Q() // Delete live node at the front & return it
} until(FALSE);
}
```

> Control Abstraction for LIFO / DFS

```
Typedef Struct.
{ listNode-T* Next.
  listNode-T* parent.
  Float cost
} listNode T   T is the State Space tree
Algorithm BB-LIFO()
  if T is the answer node Then
     u ← min [cost(T), ub(T)+E]
       Print(T)
       return
  end end
  E → E &
  E ← T
  While true do
   For each child x of T do
       if x is the answer node then
           Print "Display path from x to root"
           return
   end
   PUSH (x) // Push Neco node on Top of Stack
   X → Parent := E
   if x is answer node and cost(x) u< ub then
       u ← min [cost(T), ub(T)+E]
   end
   if no more live nodes then
       Print "No live nodes"
       Print "Minimum cost : ub
       return
   end
  E' ← POP()
  end.
```

> Control abstraction for LC Search - LC search uses heuristic function to assign the ranks to live nodes. It inestimates the extra computational work (cost) to reach an answer state from current live node. The minimum number of nodes required to be generated in subtree of k to reach an answer state.

$c(k) = f(h(k)) + \hat{g}(k)$ where $f(\cdot)$ is an increasing func.

```
Algorithm BB-LC_Search()
 if T is the answer node then
    Print (T)
    return
 end
 E ← T
 While (True) do
    For each child x of E do
       if x is the answer node then
          Print "Display path from x to root"
          return
       end
       INSERT (x)  // Insert in to list.
       x -> Parent := E
    end
    if no more live nodes then
       Print "soln Found"
       return
    end.
    E ← least_cost() // Find next least node from list.
end
```

UT DAA

Amortized Analysis

> Amortized analysis is a technique used in CS to analyze the average case time complexity of algorithms that perform a sequence of operations, where some operation may be expensive than others. The idea is to spread the cost of these expensive operations over multiple operation, so that the average cost of each insertion becomes constant or less.

> Amortized analysis is useful in designing algorithms for data structures such as dynamic arrays, priority queues. Let us consider ex. of simple hash table. How do we decide on table size? The idea is to increase the size of table whenever it is full and copy content in new table & then forget older one.

> Worst case time ~~is of~~ of insert is $O(n)$ worst case of n inserts is $n * O(n)$ which is $O(n^2)$.

~~X~~ Techniques of Amortized Analysis -

1) Aggregate Method 2) Accounting 3) Potential method

> **Aggregate Method** - This technique involves analyzing a sequence of operations as a whole rather than focusing on individual operation. The total cost of a sequence of operation is divided by the number of operations to obtain the avg cost.

Example 1 : Stack operation.

Consider a stack data structure it performs 2 operation

1> PUSH (x, p) 2> POP (x)

∴ Total cost of a sequence that contains n PUSH & POP operation = $\theta(n)$

Let us consider MultiPop (x, m) which pops the topmost m objects of stack x.

Let us consider an empty stack x and n PUSH, POP & MULTIPOP operation are executed on stack

As there are n PUSH operation executed on the Stack & Stack size will be n so the worst case cost of a single MULTIPOP operation will be O(n)

operation	worst case cost.
PUSH (x, P)	O(1)
POP(x)	O(1)
MULTIPOP(x, m)	O(n)

Worst case of any stack operation is O(n) then n operation on stack will be $O(n^2)$. We can pop at most n object PUSH onto the stack
So the total cost of any sequence of n PUSH, POP & MULTI POP operation on Stack will be O(n) for $\forall$ n
This we get Avg cost of each stack operation in sequence of n PUSH, POP, MULTIPOP = $\frac{O(n)}{n} = O(1)$

This is Amortized cost.

> Accounting Method - When the amortized cost of operation is assigned to the particular object and some are assigned lesser amortized cost than actual cost.
>When Amortized cost is longer that its actual cost, the excess amount is assigned to the particular object in data structure as its credit. This credit is consumed for other operation whose amortized cost is lesser than its actual cost.

> Stack Example.

Stack operations with PUSH(x,p) POP(x) MULTIPOP(x,m)

Opran	Actual cost	Amortized cost
push(x,p)	1	2
POP(x)	1	0
MULTIPOP(x,m)	min(Stack size, m)	0

> For n PUSH operation the amortized cost will be 2n whereas the actual cost of n PUSH operan will be n so total credit = 2n - n = n will be assigned to stack object.

> Actual cost of single POP(x) is 1 and credit of stack object after n PUSH operan is n. Therefore POP(x) seperatedy or MULTIPOP(x) can consume the credit (n). Thus for any sequence of n PUSH POP & MULTIPOP operan As in this example total amortized cost = O(n) the total actual cost of n stack operan is O(n).

> Potential Funcn Method - This is similar to Accounting method but rather then credit, it uses potential or energy to pay the cost of operation. Consider that n operations are carried out on a data structure with initial state S_0, let S_i be the state of whole data structure after performing on i^{th} operan

> The fumction ϕ is selected such that it maps each state of data structure S_i to a real no. $\phi(S_i)$ ϕ is known as "potential function" and $\phi(S_i)$ is called as "Potential Energy"

The amortized cost of i^{th} operation is $\hat{a}_i$

$$\hat{a}_i = a_i + (\phi(S_i) - \phi(S_{i-1}))$$

Total amortized cost $= \sum_{i=1}^{n} \hat{a}_i$

$$\sum_{i=1}^{n} \hat{a}_i = \sum_{i=1}^{n} (a_i + \phi(S_i) - \phi(S_{i-1}))$$

> Stack example - PUSH POP MULTIPOP operation consider potential function ϕ is defined on the stack size. Initially for empty stack $\phi(S_0) = 0$. S_i resulting after the i^{th} operation will have potential $\phi(S_i) \geq 0$

1) PUSH(X, D) - As it pushes a single element onto stack of size k the stack size will increase to k+1. Actual cost of 1 PUSH operation is 1

$$\hat{a}_i = a_i + [\phi(S_i) - \phi(S_{i-1})]$$
$$= 1 + [(k+1) - k] = 2$$

2) POP(X) - As it pops stack size decreases to k-1

$$\hat{a}_i = a_i + [\phi(S_i) - \phi(S_{i-1})]$$
$$= 1 + [(k-1) - k] = 0$$

3) MULTIPOP(X, m) - $z = \min(k, m)$ The actual cost of operation is z

$$\hat{a}_i = a_i + [\phi(S_i) - \phi(S_{i-1})]$$
$$= z + [(k-z) - k] = 0.$$

Since Amortized cost of each PUSH, POP, MULTIPOP operation is $O(1)$ the total Amortized cost of sequence of n PUSH POP MULTIPOP operation is $O(n)$

Page No.
Date

★ Tractable Problems - Feasibility of an algorithm to complete its execun in reasonable time. If they get solved in polynomial time using deterministic algorithms Time complexities $O(n^2)$ (or) $O(n^3)$, $O(1)$

example - Given an array of n elements, the problem is to arrange elements is ascending or descending order Algos like QuickSort, mergesort & HeapSort

linear Search - $O(n)$ binary Search $O(\log n)$

★ Non tractable problems - The infeasibility of an algorithm to complete its execun in reasonable time

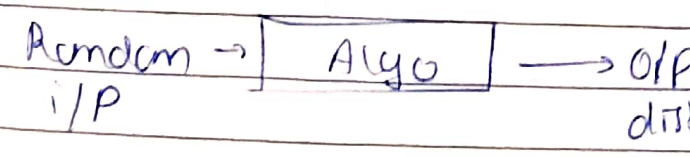
Pro if problems cannot be solved in polynomial time using deterministic algorithm Time complexities like $O(2^n)$, $O(n^n)$, $O(n!)$

ex 0-1 Knapsack $O(2^{n/2})$, TSP $O(n^2 2^n)$

★ Randomized Algorithms - An Algorithm that makes some random choices during its execun depending on random numbers for its operations

> Randomize algo use randomness in their decision making process. This randomness can come from various sources.

> It is also known as "probablistic algorithm" as input is supposed to be from a probability distribun



> The randomized algo avoid worst case behaviour of classic deterministic algo

> These algo are non-deterministic. These algo can make random but most of time correct decision.
> These algo may operate at different efficiencies in different runs for same instance of problem.
> classes of randomized algo
1) Las Vegas algo - This algo always computed the same result for same input. The running time of these algo vary with the o/p randomized The ~~rum~~ running time may varey
2) Monte Carlo - Always computes diff. results in different runs for same input. Produces output with some probability for Fixed input, They do not shoco much variation.

Approximation Algo — it is an algo which efficiently finds a solution to an optimizan problem that is guaranteed to be close to the optimal solution. Finding exact optimal Sol may be computationaley infeasible
> Finding an optimal solution for NP-complete is not polynomial bounded. Approximan algo apply some heuristic to cut out the further computan in each step.
> The quality of Soln produced by an approximan is measured by its approximan ratio

Approx Ratio ρ = $\frac{\text{cost of Approximate Soln}}{\text{optimal cost}}$

> The approx ratio provides a performance guarantee, ensuring that the soln produced by the algo is within a certain factor of optimal sol.

Page No.
Date

Ex -> Greed Algo 2) Heuristic Methods

* Embedded Algo - A computer hardware with software embedded in it is called an Embedded Sys. It's a reliable, micro controller or microprocessor based software driven real time control system.

Characteristics -

1) Single Functioned -
2) Tightly constrained - Size, cost, memory, Power
3) Real time
4) Microprocessor based.
5) Memory -

> The algo implemented on microcontroller or microprocessor based system is embedded Algo They solve specialized oprn in real time embedded algo face challenges like. Limited memory, Less power consumpn, deadline based Scheduling, Less cost

UVI - DAA

Multithreaded & Distributed Algo

A thread is a dispatchable portion of a tasks within a process. Each thread executes its code sequentially
>The algorithm that permit the concurrent execun of multiple threads of a program on multiple core in system with shared memory are known as multithreaded Algo.

→ Static multithreaded Algo - The no. of threads and their execun behaviour is determined before the program start running. Programmer explicitly specifies the no. of threads and the structure of parallelism is fixed at compile time

→ Dynamic multithreaded Algo - The no. of threads and their execun behaviour can change during execun threads are created managed and terminated dynamically based on program's runtime behaviour & workload.

→ Different keywords -
1) Parallel - used with the loop construct like for loops to mention that all iteran can concurrently executed.
2) Spawn - Used to provide the nested parallelism
In a general Fanc^n call, the parent doesn't resume until its child returns, the parent & child executed concurrently
3) Sync - A procedure can safely return values of its spawned children. The procedure is suspended until all its children have executed completely. Then the procedure resumes the instruc^n following Sync Step.

Page No.
Date

Performance measure —

1) Work - Total time required to execute the entire computation on one process, So the work represents the sum of the time taken by each thread. An ideal parallel computer with P processors can perform up to P units of work in one time step It can do $P \times T_p$ work in T_p time Since work is - Total work is $T_1 \cdot P \times T_p \geq T_1$, Hence work law is $T_p \geq \frac{T_1}{P}$

2) Span - It is the longest time it takes to execute threads along any path of the computational DAG (Directed Acyclic Graph) An ideal parallel computer with P processors can't run faster than a machine with an infinite no. of processor. On other hand, can imitate a P processor machine by employing only P of its processor. $\therefore T_p \geq T$ which is Span law

3) Speedup & Parallelism - Speedup of computation on P processors is $\frac{T_1}{T_P}$ Using work law, we can conclude that an ideal parallel computer can not have any more Speedup than no. of processors The parallelism of multithreaded computation is $\frac{T_1}{T_\infty}$, So this ratio means the average amount of work that can be performed for each step of parallel execution time.

4) Slackness - it is parallelism by P ratio $\frac{\left(\frac{T_1}{T_\infty}\right)}{P}$

Analyzing multithreaded Algos.

> Analyzing the work - The work done by a multithreaded algorithm is analyzed by ignoring the parallel constructs and concentrating on the analysis of sequential algo

$T_1(n) = T(n) = T(n-1) + T(n-2) + \Theta(1)$

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

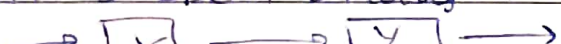
where F_n grows exponentially as $F_n = \phi^n$ where ϕ = a golden ratio = $\left(\frac{1+\sqrt{5}}{2}\right)$

> Analyzing Span -

1) For serial composition of subcomputation - If a set of subcomputation are in series then the span of their composition is obtained by the sum of the spans of those sub-computations

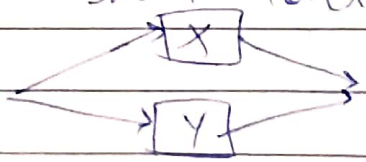
2) For parallel composition - If a set of parallel sub composition computation are in parallel, then the span of their composition is the maximum of the spans of those sub computations

The work & span analysis.



work $T_1(X \cup Y) = T_1(X) + T_1(Y)$ Series.

Span $= T_\infty(X \cup Y) = T_\infty(X) + T_\infty(Y)$



work $= T_1(X \cup Y) = T_1(X) + T_1(Y)$

span $= T_\infty(X \cup Y) = \max(T_\infty(X), T_\infty(Y))$

> Analyzing parallel loops - Any multithreaded Algos uses keyword parallel with the loop constructs to define parallel execution of iteration. To design parallel Algo to multiply nxn matrix $X = x_i$ with vector y_i of size nx1

$$z_i = \sum_{j=1}^{n} x_{ij} y_j$$

Algorithm Parallel_Multi(X,Y):
    parallel For i ← 1 to n do
        $z_i \leftarrow 0$
    end.
    parallel For i ← 1 to n do
        For j ← 1 to n do
            $z_i \leftarrow z_i + (x_{ij} * y_i)$
        end.
    end.

Keyword parallel for indicates that iteration of the loop may ran concurrently.

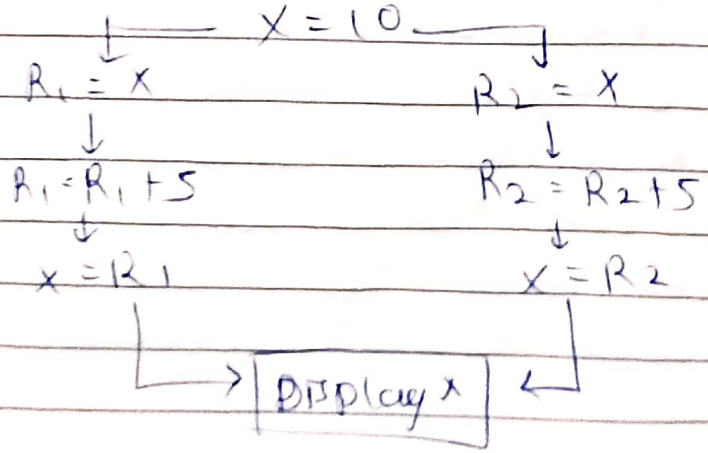
Algorithm matvec MUL_Loop(X, y, z, n, i, i')
    if i ← i' then
    For j ← 1 to n do
        $z_i \leftarrow z_i + (x_{ij} * y_i)$
    end
    else mid ← (i+j)/2
        Spawn matvec_mul_loop(X, y z, n, i, mid)
        MATVEC_MUL_LOOP (X, Y, z, n, mid+1, i')
        Sync
    end

The algo recursively spawn first half of the loop which runs parallel with second half
work done by parallel. matvec is simple case of matrix
$T_1(n) = O(n^2)$
Span is $T_\infty(n) = \Theta(\log n)$.

> Race condition - Algos are said to be deterministic if it always does the same works for the same input, irrespective of how the instruc^n are ~~set~~ scheduled on a multicore machine. When 2 or more threads access shared data concurrently, & atleast one of them modifies the data.

```
Race Condin^n ()
    x ← 10
    parallel for i ← 1 to 2 do
        x ← x + 5
    end.
```

After initializing X to 10 the parallel loop creates two strands each performing addi^n to x.
The expected outcome of code is 20 but algo might return 15. Two strands run in parallel on 2 processors. Both processors loads x, update x, and write back to the memory.



Multithreaded Matrix Multiplicaⁿ -

```
P-matrix-Multiply(A,B)
n = A.rows
// C is a new nxn matrix.
Parallel For i=1 to n do
    Parallel For j=1 to n do
        Cij = 0
        For k=1 to n do
            Cij = Cij + aik x bkj
        end
    end
    return C
end.
```

In this algo the longest path is ~~when spawning~~ the outer and inner parallel loop execuⁿ and then the n execuⁿ for innermost For loop.

So the Spanⁿ of this algo is $T_\infty(n) = \Theta(n)$

$$\frac{T_1(n)}{T_\infty(n)} = \frac{\Theta(n^3)}{\Theta(n)} = \Theta(n^2).$$

Multithreaded Merge Sort -

~~Merge-Sort (a,p,r) At~~
~~if p < r then~~
~~q = (p+r)~~

```
Algorithm merge-sort (x, first, last)
{ if (first < last)
    { mid := (first + last)/2
    Spawn mergsort (x, first, mid);
    Mergesort (x, mid+1, last);
    Sync; }
Merge-Results (x, first, mid, last);
}.
```

Since an algo Merg-Result() remains a sequential algo. its work and Span are $\Theta(n)$ where n is input size.

Work of merge-sort() is ~~$T(n)$~~ :-

$T_1(n) = 2(= 2T_1(n/2) + \Theta(n) = \Theta(n \log_2 n)$

and the Span $T_\infty(n)$ of merge-sort()

$T_\infty(n) = T_\infty(n/2) + \Theta(n) = \Theta(n)$.

Parallelism $= \frac{T_\infty(n)}{T_\infty(n)} \cdot \frac{T_1(n)}{T_\infty(n)} = \frac{\Theta(n \log_2 n)}{\Theta(n)} = \Theta(\log_2 n)$.

Distributed BFS -

A parallel Algo that facilitates concurrent execution in distributed computing environment.

A BFS algo is applied to traverse or search a graph or tree data structure. distributed BFS performs ~~over~~ BFS over ~~distrit~~ distributed network

> In 1-D partitioning a graph is partitioned such that each node & edges originating from it are possessed by one processor

> P is the no. of processors. Edges emanating from vertex V is a list of vertex index in row V of adjacency matrix. Process -

> 1) create a set of Frontier vertices (F) owned by processor

2) Edge lists of F are merged to form a set neighbor vertices N

3) Send msg to processes that own vertices in set N and add these vertices in their next F

4) Receive set N from other processors

5) Merge to form final set N

6) Update level.

Page No.
Date

* Rabin Karp algo - It is a String searching algo

- It computes hash value of the pattern and hash value of overlapping substrings of text.
- Compare hash values. if the hash value matches the text is detected in pattern
- Rolling Hash - To efficiently update the hash value when moving from one substring to the next.

Pseudocode -

```
Function rabin_karp (text, pattern):
    n = length of text
    m = length of pattern
    P-hash = hash (pattern)
    text_hash = hash (text[0:m])

    for i from 0 to n-m:
        if text_hash == P-hash:
            ~~if text[i:~~
            if text[i:i+m] == pattern
                print ("found")
        if i<n-m:
            text_hash = rehash(text_hash, text[i],
                                text[i+m])

Function hash (s):
    result = 0
    for char in s
        result = (result * base + ord(char)) % Prime
    return result.
```