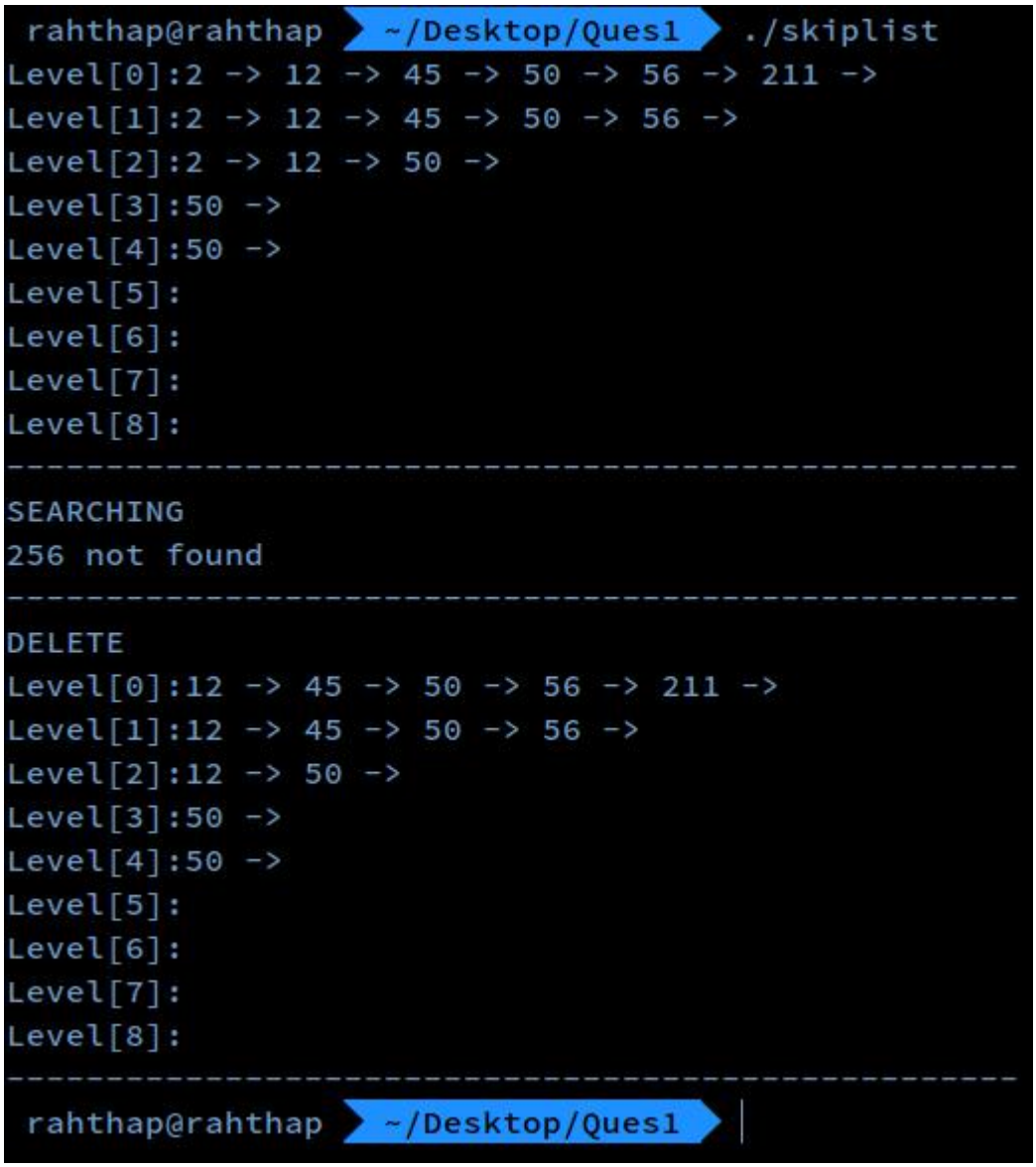
FINAL REPORT : Algorithms Final Lab

NAME : RAHUL THAPAR
ID : 1410110321
DATE : 20-April 2017

LIST

QUES 1

Data Structure with log(n) complexity : SKIP LIST



CODE

```
#include "skiplist.h"
static node *create_node(int level, int key, object *obj){
        int i;
        node *nd = (node *)malloc(sizeof(node) + level * sizeof(node *));
        nd->obj = obj;
        nd->key = key;

        for (i = 0; i < level; i++) {
                nd->forward[i] = NULL;
        }

        return nd;
```

```
}

skiplist *create_skiplist(void){
        skiplist *sl = (skiplist *)malloc(sizeof(skiplist));
        sl->head = create_node(MAX_LEVEL, 0, NULL);
        sl->level = 1;
        return sl;
}

static void free_node(node *nd){
        free(nd);
}

void free_skiplist(skiplist *sl){
        node *nd, *next;

        nd = sl->head->forward[0];
        free_node(sl->head);

        while (nd) {
                next = nd->forward[0];
                free_node(nd);
                nd = next;
        }

        free(sl);
}

static int random_level(){
        int level = 1;
        while ((rand() & 0xFFFF) < (0.5 * 0xFFFF)) {
                level += 1;
        }
        return (level < MAX_LEVEL) ? level : MAX_LEVEL;
}

void insert(skiplist *sl, int key, object *obj){
        node   *update[MAX_LEVEL];
        node   *nd;

        int i, level;

        nd = sl->head;
```

```
	for (i = sl->level - 1; i >= 0; i--) {
		while (nd->forward[i] != NULL && nd->forward[i]->key < key)
			nd = nd->forward[i];
		update[i] = nd;
	}

	level = random_level();
	if (level > sl->level) {
		for (i = sl->level; i < level; i++) {
			update[i] = sl->head;
		}
		sl->level = level;
	}

	nd = create_node(level, key, obj);
	for (i = 0; i < level; i++) {
		nd->forward[i] = update[i]->forward[i];
		update[i]->forward[i] = nd;
	}
}

static void delete_node(skiplist *sl, node *nd, node **update){
	int i;
	for (i = 0; i < sl->level; i++) {
		if (update[i]->forward[i] == nd) {
			update[i]->forward[i] = nd->forward[i];
		}
	}

	for (i = i - 1; i >= 0; i--) {
		if (sl->head->forward[i] == NULL) {
			sl->level--;
		}
	}
}

void delete(skiplist *sl, int key){
	node *update[MAX_LEVEL], *nd;
	int i;

	nd = sl->head;
	for (i = sl->level - 1; i >= 0; i--) {
		while (nd->forward[i] && nd->forward[i]->key < key) {
			nd = nd->forward[i];
```

```
                        }
                        update[i] = nd;
                }

                nd = nd->forward[0];

                if (nd && nd->key == key) {
                        delete_node(sl, nd, update);
                        free_node(nd);
                }
}

node *find(skiplist *sl, int key){
                node *nd;
                int i;

                nd = sl->head;
                for (i = sl->level - 1; i >= 0; i--) {
                        while (nd->forward[i] != NULL) {
                                if (nd->forward[i]->key < key)
                                        nd = nd->forward[i];
                                else if (nd->forward[i]->key == key)
                                        return nd->forward[i];
                                else
                                        break;
                        }
                }

                printf(" %d not found\n", key);
                return NULL;
}

void print(skiplist *sl){
                node *nd;
                int i;

                for (i = 0; i <= MAX_LEVEL; i++) {
                        nd = sl->head->forward[i];
                        printf("Level[%d]:", i);

                        while (nd) {
                                printf("%d -> ", nd->key);
                                nd = nd->forward[i];
                        }
```

```
                    printf("\n");
            }
}
```

Ques 2

Generate a random graph with n > 10 nodes.
Generate random n^2/2 directly edges.
Assign a weight in the range (1, 10).
Apply Dijkstra’ s algorithm.
Find out the single source shortest path.

Generate Random Graph

```
Graph::Graph(int n) : number_of_vertices(n)
{
    connectivity_matrix = new float *[number_of_vertices];
    int i;
    for (i = 0; i < number_of_vertices; i++)
    {
        connectivity_matrix[i] = new float[number_of_vertices];
    }
}

Graph::~Graph()
{
    int i;
    open_set.clear();
    closed_set.clear();
    for (i = 0; i < number_of_vertices; i++)
    {
        delete [] connectivity_matrix[i];
    }
    delete [] connectivity_matrix;
    connectivity_matrix = NULL;
}
##### Initializing Graph
void Graph::initiate_graph(float d, float min, float max)
```

```
{
    if (NULL == connectivity_matrix){
        cout << "Memory Allocate Failed!" <<endl;
        return ;
    }
    if ((min <= 0.0) || (max <= 0.0) || (min >= max)){
        cout << "Invalid Edge Cost Range!" <<endl;
        return ;
    }
    graph_density = d;
    int i, j;
    float random;
    srand((unsigned int)time(NULL));
    for (i = 0; i < number_of_vertices; i++)
    {
        connectivity_matrix[i][i] = 0.0;                    //the cost from i to i is 0
        for (j = i + 1; j < number_of_vertices; j++)
        {
            random = random_generator(0.0, 1.0);            //get a decimal
between 0 and 1
            if (random >= graph_density)                    //there is no path if
random is less than density
            {
                connectivity_matrix[i][j] = 0.0;
                connectivity_matrix[j][i] = connectivity_matrix[i][j];
//undirected graph
            }
            else                                            //else, there is a path
            {
                connectivity_matrix[i][j] = random_generator(min, max);         //get
a value between min to max, default is 1 to 10
                connectivity_matrix[j][i] = connectivity_matrix[i][j];
            }
        }
    }
}
```

Generate random cost edges

```
float random_generator(float lower, float upper)
{
  //Get random Cost
    int range = (int)upper * 10 - (int)lower * 10;
    int temp = rand() % range + (int)lower * 10;
```

```
    return (float)temp / 10;
}
```

Apply Dijkstra’ s Algorithm

```
void Graph::dijkstra_algorithm(int s, int t){
    if (closed_set.empty() == true)     {
        Vertex V;
        V.vertex_no = s;
        V.cost_from_start = 0;
        V.path_from_start.push_back(s);
        closed_set.push_back(V);
    }
    int current = s;
    if ((false == update_open_set(closed_set.back())) && (open_set.empty() == true))
{
        return;   //stop when open set is not updated and it is empty
    }else{
        current = update_closed_set();
    }
    if (current == t){
        return;   //stop when destination is included in closed set
    } else{
        dijkstra_algorithm(current, t);
    }
}
```

Find Shortest Path

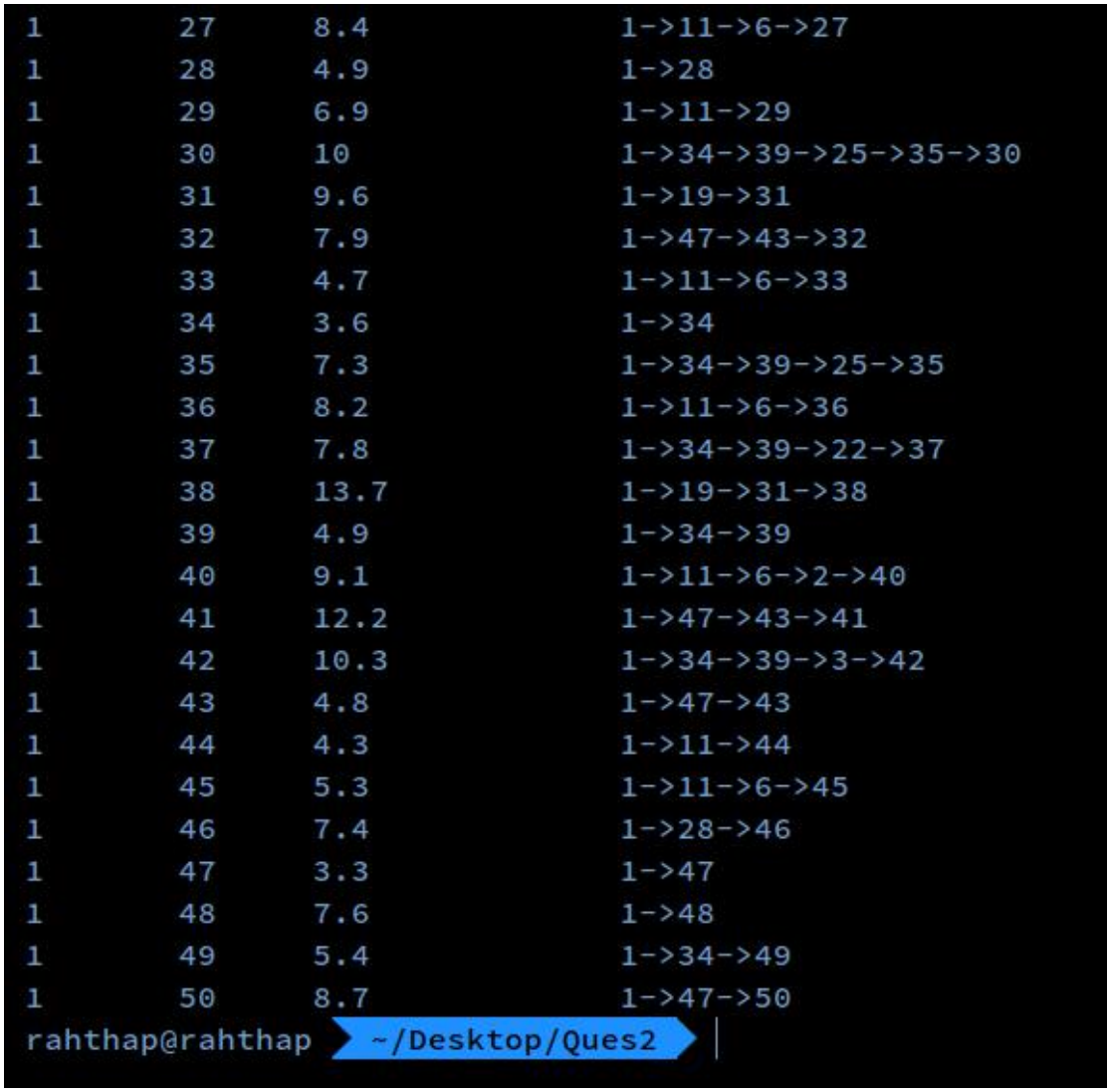
```
float Graph::get_shortest_path(int s, int t){
    if ((s < 1) || (s > number_of_vertices))     {
        cout << "No Such Start Vertex " << s <<endl;
        return 0.0;
    }
    if ((t < 1) || (t > number_of_vertices))     {
        cout << "No Such End Vertex " << t <<endl;
        return 0.0;
    }
    if (s == t)     {
        return 0.0;
    }
    dijkstra_algorithm(s - 1, t - 1);                   //perform dijkstra's algorithm
    Vertex T = closed_set.back();                       //the last member of closed set
    if (T.vertex_no != t - 1)                           //the destination is not in
```

```
    {
        cout << "No Path From " << s << " To " << t << endl;
        open_set.clear();
        closed_set.clear();
        return -1;
    }
    cout <<" "<<s <<"\t" <<" "<<t << "\t";
    cout <<T.cost_from_start <<"\t\t";
    list<int>::iterator iter;

    for (iter = T.path_from_start.begin(); iter != T.path_from_start.end(); iter++){
        if (*iter != T.path_from_start.back()){
            cout << (*iter) + 1 << "->";
        }else{
                    cout << (*iter) + 1 <<endl;
        }
    }
    open_set.clear();
    closed_set.clear();
    return T.cost_from_start;
}
```

SCREENSHOTS

```
rahthap@rahthap ~/Desktop/Ques2 ./a.out
FROM    TO      COST            PATH
1       2       6               1->11->6->2
1       3       7.1             1->34->39->3
1       4       6               1->47->4
1       5       9.4             1->11->29->5
1       6       3.7             1->11->6
1       7       4.9             1->47->7
1       8       8.7             1->34->39->22->8
1       9       4.4             1->9
1       10      6.8             1->9->10
1       11      2.6             1->11
1       12      9.6             1->11->6->12
1       13      11.1            1->34->39->25->13
1       14      10.3            1->11->6->36->14
1       15      6.2             1->47->43->15
1       16      9.7             1->34->16
1       17      7.3             1->47->7->17
1       18      6               1->47->18
1       19      6.7             1->19
1       20      7.9             1->11->6->45->20
1       21      8.5             1->47->43->21
1       22      6.5             1->34->39->22
1       23      9.8             1->47->43->21->23
1       24      8.4             1->9->10->24
1       25      5.9             1->34->39->25
1       26      5.6             1->47->26
1       27      8.4             1->11->6->27
1       28      4.9             1->28
1       29      6.9             1->11->29
1       30      10              1->34->39->25->35->30
1       31      9.6             1->19->31
```



Ques 3

Max Flow Problem :

// COULD NOT IMPLEMENT INTO CODE

Approach taken :

For a given graph (map of the town) every edge/ block connecting the other block has max capacity 1. Once the block is visited it becomes

0/1 -----> 1/1

So, the other boy cannot take this path. At the corner however they can take, this means:

A ------ 1/1 --------> B (Already traversed by boy 1)

If (Boy 2 moves from B -------> A) //There is no arrow from B to A
Then : the flow capacity becomes 1/1 --------> 0/1 i.e. At the corner boy2 crosses boy 1

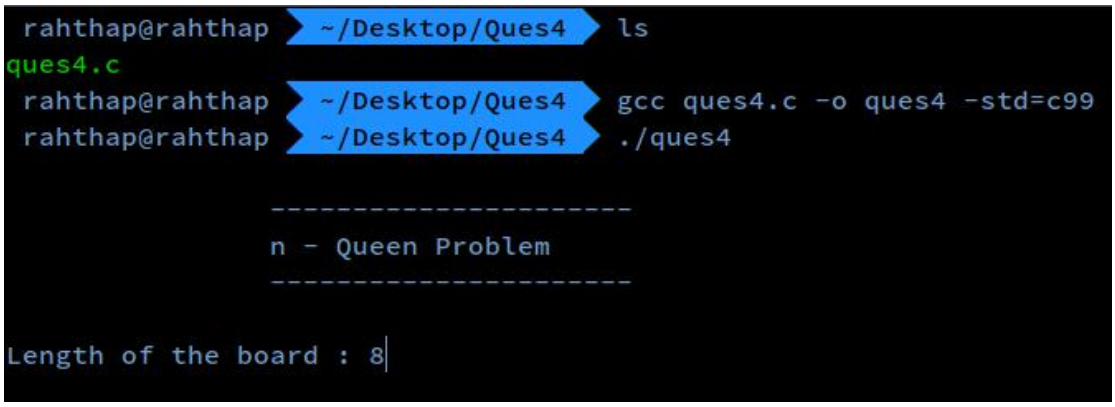
When max flow = 2 : Both the boys can go to the same school.

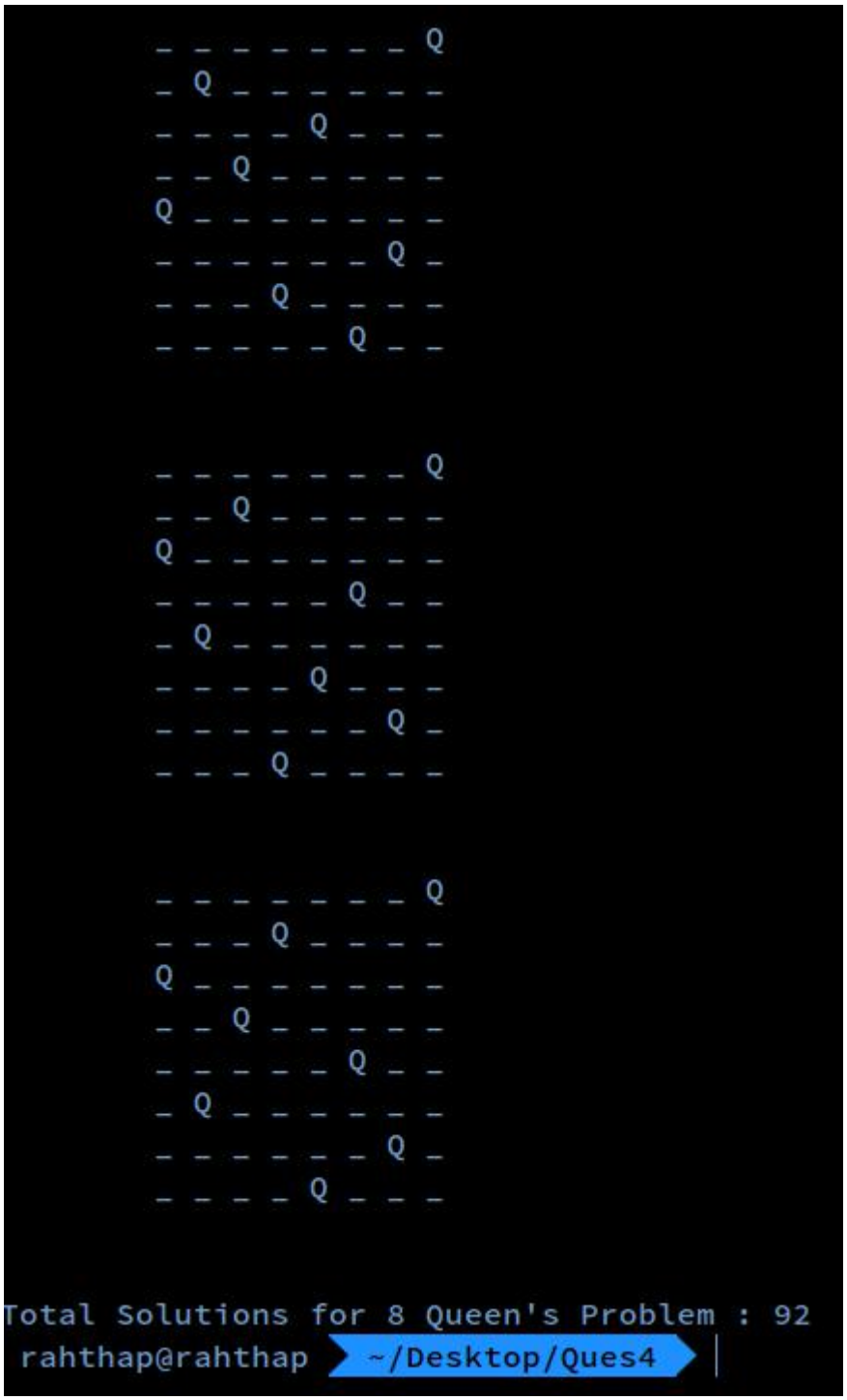
STUCK AT?

Not able to make 1/1 ------> 0/1 after boy 1 has already taken that block at the corner.

Ques 4

N-Queen Problem





CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int count=0;
int position_judge (int row,int column,int *a){
    int judge=1;
    int line_count;
    for (line_count=0; line_count<row; line_count++)    {
```

```
        if (column==a[line_count])
            judge=0;
        else if (abs(column-a[line_count])==abs(row-line_count))
            judge=0;
    }
    return judge;
}
void position_print (int n,int *a){
    int row;
    int column;
    printf("\t");
    for ( row=0; row<n; row++)      {
        for (column=0; column<a[row]; column++)
            printf("_ ");
        printf("%c",'Q');
        printf(" ");
        for (column=a[row]+1; column<n; column++)
            printf("_ ");
        printf("\n");
        printf("\t");
    }
    printf("\n\n");
}
void find_next (int n,int row,int *a){
    row++;
    int column;
    for (column=0; column<n; column++)
        if (position_judge(row,column,a)==1)
        {
            a[row]=column;
            if (row<n-1)
                find_next(n,row,a);
            else if (position_judge(row,column,a)==1)
            {
                position_print(n,a);
                count ++;
            }
        }
}
int main(){
    printf("\n\t\t----------------------\n");
    printf("\t\tn - Queen Problem \n");
    printf("\t\t----------------------\n\n");
    int n=8,row,column;
```

```
    printf("Length of the board : ");
    scanf ("%d",&n);
    int a[n];
    for (int z=0; z<n; z++)
        a[z]=0;

    row=0;
    for (column=0; column<n; column++)      {
        a[row]=column;
        find_next(n,row,a);
    }
    printf("Total Solutions for %d Queen's Problem : %d\n",n,count);
    return 0;
}
```

Ques 5

```
Input Array Size : N
Number of Processors : P
Constraint : N >> P
```

Normal Bubble Sort Algorithm

```
Bubble-sort (A)

for i = 1 to N do
  for j = N to i +1 do
          If A[j] < A[j-1] then
            Exchange A[j]  ↔  A[j-1]
```

Explanation :

The above algorithm simply means that

1. Start from the first element of the array.
2. Compare 2 consecutive elements.
3. If the present element is greater than the element to right : SWAP them.
4. When no swapping is required : elements are sorted.

Parallel Bubble Sort Algorithm

Bubble Sort has various parallel variants such as :

1. Odd-Even Transposition
2. Cocktail sort

```
Bubble Sort (A)
begin
   for i = 1 to N do
      begin
        if i is odd then
            for j = 0 to n/2-1 do
                If A[2i+1] > A[2i+2] then
                    Interchange A[2i+1] ↔ A[2i+2]
        else
          if i is even then
           for j =1 to n/2 -1 do
             If A[2i] > A[2i+1] then
                    Interchange A[2i] ↔ A[2i+1]
   END for
END
```

Explanation:

1. The idea is processors are grouped int odd/even and even/odd pairs.

2. Odd/even Phase : The odd processes P compare and exchange their elements with the even processors P+1.

3. Even/Odd Phase : The even processes compare and exchange their elements with the odd processors P+1.

Analysis of this Parallel Algorithm

Both the phases of the algorithm requires O(N) comparisons.

Taking the worst case where all the elements are sorted **in** Descending order **and** we have to sort them **in** ascending order :

8 7 6 5 4 3 2 1

Pass 1a : 7 8 5 6 3 4 1 2
Pass 1b : 7 5 8 3 6 1 4 2

Pass 2a : 5 7 3 8 1 6 2 4
Pass 2b : 5 3 7 1 8 2 6 4

Pass 3a : 3 5 1 7 2 8 4 6
Pass 3b : 3 1 5 2 7 4 8 6

Pass 4a : 1 3 2 5 4 7 6 8
Pass 4b : 1 2 3 4 5 6 7 8 <- SORTED

This means that **is** we have 8 elements **then** we have to **do** 4 complete passes i.e. 4*2 = 8 passes to sort the elements. So we have to **do** n passes **in** an array **in** the worst case where n **is** the number **of** elements **in** the array.

How is this different from Normal Bubble Sort :

Bubble sort **is** inherently sequential because every step **of** computation **is** dependent on the result **of** the previous step. By "offsetting" the computation slightly however, we get rid **of this** dependency, allowing the sorting **of** adjacent pairs to be carried out **in** parallel - That's the difference between odd-even and bubble sort.