Group A : DESIGN AND ANALYSIS ALGORITHM

****

Assignment No: 1

Title Name: Calculate Fibonacci numbers and find its step count.

Name: Abhishek Santosh Bankar

Class : BE **Div**: 1 **Batch**: A

Roll No: 405A011

****

Program:

Fibonacci Series in C++ without Recursion

Code:

```
#include <iostream>
using namespace std;
int main()
{
	int n1=0,n2=1,n3,i,number;
	cout<<"Enter the number of elements: ";
	cin>>number;
	cout<<n1<<" "<<n2<<" "; //printing 0 and 1
	for(i=2;i<number;++i)       //loop starts from 2 because 0 and 1 are already
	printed
	{
		n3=n1+n2;
		cout<<n3<<" ";
		n1=n2;
		n2=n3;
	}
	 return 0;
 }
```

Output:

```
Output

/tmp/zol19eVye4.o
Enter the number of elements: 7
0 1 1 2 3 5 8
```

Fibonacci series using recursion in C++

Code:

```
#include<iostream>
using namespace std;
void printFibonacci(int n)
{
        static int n1=0, n2=1, n3;

        if(n>0)
        {
                n3 = n1 + n2;

                n1 = n2; n2 = n3;
                cout<<n3<<" ";
                 printFibonacci(n-1);
        }
}
int main()
{
         int n;

         cout<<"Enter the number of elements: ";
         cin>>n;
         cout<<"Fibonacci Series: ";
         cout<<"0 "<<"1 ";

         printFibonacci(n-2); //n-2 because 2 numbers are already printed
         return 0;
```

}

Output:

```
Output

/tmp/zol19eVye4.o
Enter the number of elements: 7
Fibonacci Series: 0 1 1 2 3 5 8
```

****

Assignment No: 2

Title Name: Write a program to implement Job sequencing with deadlines using a greedy method.

Name: Abhishek Santosh Bankar

Class : BE **Div**: 1 **Batch**: A

Roll No: 405A011

****

Job sequencing with deadlines using a greedy method.

Code:

```
#include<iostream>
#include<algorithm>
using namespace std;
// A structure to represent a job
struct Job {
    char id;
    int dead;
    int profit;
};
   This function is used for sorting all the jobs according to the profit
       compare(Job a, Job b) {
 return (a.profit > b.profit);
}

void jobschedule (Job arr[], int n) {
 // Sort all jobs according to decreasing order of prfit
 sort(arr, arr+n, compare);

 int result[n]; // To store result

 bool slot[n];
 // Initialize all slots to be free

 for (int i=0; i<n; i++)
    slot[i] = false;

 for (int i=0; i<n; i++) {
     // Find a free slot for this job (Note that we start
```

```
    // from the last possible slot)
    for (int j=min(n, arr[i].dead)-1; j>=0; j--) {
        // Free slot found
        if (slot[j]==false) {
            result[j] = i; // Add this job to result

            slot[j] = true; // Make this slot occupied

            break;

    }
 }
 // Print the result

 for (int i=0; i<n; i++)
 if (slot[i])
 cout << arr[result[i]].id << " ";

}
int main() {
 Job arr[] = { {'a', 2, 20}, {'b', 2, 15}, {'c', 1, 10},{'d', 3, 5}, {'e', 3, 1}};
 int n = 5;

 cout << "maximum profit sequence of jobs is-->";
 jobschedule(arr, n);
}
```

Output:

```
Output

/tmp/zol19eVye4.o
maximum profit sequence of jobs is-->b a d
```

****

Assignment No: 3

Title Name: Huffman Encoding using a greedy strategy

Name: Abhishek Santosh Bankar

Class : BE **Div**: 1 **Batch**: A

Roll No: 405A011

****

Program:

```
// Huffman Coding in C++

#include <iostream>
using namespace std;
#define MAX_TREE_HT 50

struct MinHNode {
  unsigned freq;
  char item;
  struct MinHNode *left, *right;
};

struct MinH {
  unsigned size;
  unsigned capacity;
  struct MinHNode **array;
};

// Creating Huffman tree node
struct MinHNode *newNode(char item, unsigned freq)
{
  struct MinHNode *temp = (struct MinHNode *)malloc(sizeof(struct MinHNode));
  temp->left = temp->right = NULL;
```

```
  temp->item = item;
  temp->freq = freq;
  return temp;
}

// Create min heap using given capacity
struct MinH *createMinH(unsigned capacity)
{
  struct MinH *minHeap = (struct MinH *)malloc(sizeof(struct MinH));
  minHeap->size = 0;
  minHeap->capacity = capacity;
  minHeap->array = (struct MinHNode **)malloc(minHeap->capacity * sizeof(struct MinHNode *));
  return minHeap;
}

// Print the array
void printArray(int arr[], int n)
{
  int i;
  for (i = 0; i < n; ++i)
    cout << arr[i];
  cout << "\n";
}

// Swap function
void swapMinHNode(struct MinHNode **a, struct MinHNode **b)
{
  struct MinHNode *t = *a;
  *a = *b;
  *b = t;
}
```

```
// Heapify
void minHeapify(struct MinH *minHeap, int idx)
{
  int smallest = idx;
  int left = 2 * idx + 1;
  int right = 2 * idx + 2;
  if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq)
    smallest = left;
  if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq)
    smallest = right;
  if (smallest != idx) {
    swapMinHNode(&minHeap->array[smallest],
          &minHeap->array[idx]);
    minHeapify(minHeap, smallest);
  }
}

// Check if size if 1
int checkSizeOne(struct MinH *minHeap)
{
  return (minHeap->size == 1);
}
// Extract the min
struct MinHNode *extractMin(struct MinH *minHeap)
{
  struct MinHNode *temp = minHeap->array[0];
  minHeap->array[0] = minHeap->array[minHeap->size - 1];
  --minHeap->size;
  minHeapify(minHeap, 0);
  return temp;
}
```

```
// Insertion
void insertMinHeap(struct MinH *minHeap, struct MinHNode *minHeapNode)
{
  ++minHeap->size;
  int i = minHeap->size - 1;
  while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {
    minHeap->array[i] = minHeap->array[(i - 1) / 2];
    i = (i - 1) / 2;
  }
  minHeap->array[i] = minHeapNode;
}

// BUild min heap
void buildMinHeap(struct MinH *minHeap)
{
  int n = minHeap->size - 1;
  int i;
  for (i = (n - 1) / 2; i >= 0; --i)
    minHeapify(minHeap, i);
}
int isLeaf(struct MinHNode *root) {
  return !(root->left) && !(root->right);
}


struct MinH *createAndBuildMinHeap(char item[], int freq[], int size)
{
  struct MinH *minHeap = createMinH(size);
  for (int i = 0; i < size; ++i)
    minHeap->array[i] = newNode(item[i], freq[i]);
```

```
  minHeap->size = size;
  buildMinHeap(minHeap);
  return minHeap;
}

struct MinHNode *buildHfTree(char item[], int freq[], int size)
{
  struct MinHNode *left, *right, *top;
  struct MinH *minHeap = createAndBuildMinHeap(item, freq, size);
  while (!checkSizeOne(minHeap)) {
    left = extractMin(minHeap);
    right = extractMin(minHeap);
    top = newNode('$', left->freq + right->freq);
    top->left = left;
    top->right = right;
    insertMinHeap(minHeap, top);
  }
  return extractMin(minHeap);
}

void printHCodes(struct MinHNode *root, int arr[], int top)
{
  if (root->left) {
    arr[top] = 0;
    printHCodes(root->left, arr, top + 1);
  }
  if (root->right) {
    arr[top] = 1;
    printHCodes(root->right, arr, top + 1);
  }
  if (isLeaf(root)) {
```

```
    cout << root->item << "  | ";
    printArray(arr, top);
  }
}
// Wrapper function
void HuffmanCodes(char item[], int freq[], int size)
{
  struct MinHNode *root = buildHfTree(item, freq, size);
  int arr[MAX_TREE_HT], top = 0;
  printHCodes(root, arr, top);
}
int main()
{
  char arr[] = {'A', 'B', 'C', 'D'};
  int freq[] = {5, 1, 6, 3};
  int size = sizeof(arr) / sizeof(arr[0]);
  cout << "Char | Huffman code ";
  cout << "\n----------------------\n";
  HuffmanCodes(arr, freq, size);
}
```

Output:

```
Char | Huffman code
----------------------
C  | 0
B  | 100
D  | 101
A  | 11
```

****

Assignment No: 4

Title Name: Solve a fractional Knapsack problem using a greedy method

Name: Abhishek Santosh Bankar

Class : BE **Div**: 1 **Batch**: A

Roll No: 405A011

****

Program:

Code:

```
// C++ program to solve fractional Knapsack Problem
#include <bits/stdc++.h>
using namespace std;

// Structure for an item which stores weight and corresponding value of Item
struct Item

{
    int value, weight;
    // Constructor
    Item(int value, int weight)
    {
        this->value = value;
        this->weight = weight;

    }
};

// Comparison function to sort Item according to val/weight ratio
bool cmp(struct Item a, struct Item b)
{
    double r1 = (double)a.value / (double)a.weight;
    double r2 = (double)b.value / (double)b.weight;
    return r1 > r2;
}
```

```
double fractionalKnapsack(int W, struct Item arr[], int N)
{
    sort(arr, arr + N, cmp);
    double finalvalue = 0.0; // Result (value in Knapsack)
    for (int i = 0; i < N; i++)
    {
        // If adding Item won't overflow, add it completely
        if (arr[i].weight <= W)
        {
            W -= arr[i].weight;
            finalvalue += arr[i].value;

         }
        else
        {
            finalvalue+= arr[i].value * ((double)W / (double)arr[i].weight);
            break;
        }
    }
    return finalvalue;
}

// Driver's code
int main()
{
    int W = 50; // Weight of knapsack
    Item arr[] = { { 60, 10 }, { 100, 20 }, { 120, 30 } };
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function call
    cout << "Maximum value we can obtain = "
    << fractionalKnapsack(W, arr, N);
    return 0;
}
```

Output:

```
Output

/tmp/2et7aK3pF9.o
Maximum value we can obtain = 240
```

****

Assignment No: 5

Title Name: Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy

Name: Abhishek Santosh Bankar

Class : BE **Div**: 1 **Batch**: A

Roll No: 405A011

****

Program

```
// C++ program to solve knapsack problem using branch and
#include <bits/stdc++.h>
 struct Item

 {
        float weight;
        int value;
 };

 // Node structure to store information of decision tree
 struct Node
 {
        // level --> Level of node in decision tree (or index in arr[]
        // profit --> Profit of nodes on path from root to this node (including this node)
        // bound ---> Upper bound of maximum profit in subtree of this node/
        int level, profit, bound;
        float weight;
 };

 // Comparison function to sort Item according to val/weight ratio
 bool cmp(Item a, Item b)
 {
        double r1 = (double)a.value / a.weight;
        double r2 = (double)b.value / b.weight;
        return r1 > r2;
 }
```

```
// Returns bound of profit in subtree rooted with u. This function mainly uses Greedy
solution to find an upper bound on maximum profit.
int bound(Node u, int n, int W, Item arr[])
{
	// if weight overcomes the knapsack capacity, return 0 as expected bound
	if (u.weight >= W)
		return 0;

	// initialize bound on profit by current profit
	int profit_bound = u.profit;

	// start including items from index 1 more to current item index
	int j = u.level + 1;
	int totweight = u.weight;

	// checking index condition and knapsack capacity condition
	while ((j < n) && (totweight + arr[j].weight <= W))
	{
		totweight += arr[j].weight;
		profit_bound += arr[j].value;
		j++;
	}

	// If k is not n, include last item partially for upper bound on profit
	if (j < n)

		profit_bound += (W - totweight) * arr[j].value /arr[j].weight;

	return profit_bound;
}

// Returns maximum profit we can get with capacity W
int knapsack(int W, Item arr[], int n)
{
	// sorting Item on basis of value per unit weight.
	sort(arr, arr + n, cmp);

	// make a queue for traversing the node
	queue<Node> Q;
	Node u, v;
```

```
        // dummy node at starting
        u.level = -1;
        u.profit = u.weight = 0;
        Q.push(u);

        // One by one extract an item from decision tree compute profit of all children of
extracted item and keep saving maxProfit
        int maxProfit = 0;
        while (!Q.empty())
        {
                // Dequeue a node
                u = Q.front();
                Q.pop();

                // If it is starting node, assign level 0
                if (u.level == -1)
                        v.level = 0;

                // If there is nothing on next level
                if (u.level == n-1)
                        continue;

                // Else if not last node, then increment level, and compute profit of children
nodes.
                v.level = u.level + 1;

                // Taking current level's item add current level's weight and value to node
u's weight and value
                v.weight = u.weight + arr[v.level].weight;
                v.profit = u.profit + arr[v.level].value;

                // If cumulated weight is less than W and profit is greater than previous
profit,
                // update maxprofit
                if (v.weight <= W && v.profit > maxProfit)
                        maxProfit = v.profit;

                // Get the upper bound on profit to decide whether to add v to Q or not.
                v.bound = bound(v, n, W, arr);
```

```
                // If bound value is greater than profit, then only push  into queue for further
consideration
                if (v.bound > maxProfit)
                        Q.push(v);

                // Do the same thing, but Without taking the item in knapsack
                v.weight = u.weight;
                v.profit = u.profit;
                v.bound = bound(v, n, W, arr);
                if (v.bound > maxProfit)
                        Q.push(v);
        }
        return maxProfit;
}

// driver program to test above function
int main()
{
        int W = 10; // Weight of knapsack
        Item arr[] = {{2, 40}, {3.14, 50}, {1.98, 100},
                                {5, 95}, {3, 30}};
        int n = sizeof(arr) / sizeof(arr[0]);
        cout << "Maximum possible profit = "
                << knapsack(W, arr, n);
        return 0;
}
```

Output:

Output

```
/tmp/DGYb11Undn.o
Maximum possible profit = 235
```

****

Assignment No: 6

Title Name: Design 8-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final 8-queen's matrix.

Name: Abhishek Santosh Bankar

Class : BE **Div**: 1 **Batch**: A

Roll No: 405A011

****

Program:

Code:

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#define N 8
using namespace std;

/* print solution */
void printSolution(int board[N][N])
{
 for (int i = 0; i < N; i++)
 {
     for (int j = 0; j < N; j++)
     cout<<board[i][j]<<" ";
     cout<<endl;
 }
}
   check if a queen can be placed on board[row][col]*/
bool isSafe(int board[N][N], int row, int col)
{
 int i, j;
 for (i = 0; i < col;
 {
     if (board[row][i])
     return false;
 }
 for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
 {
     if (board[i][j])
     return false;
```

```
 }
 for (i = row, j = col; j >= 0 && i < N; i++, j--)
 {
     if (board[i][j])
     return false;
 }
 return true;
}
/*solve N Queen problem */
bool solveNQUtil(int board[N][N], int col)
{
 if (col >= N)
 return true;
 for (int i = 0; i < N; i++)
 {
    if ( isSafe(board, i, col) )
    {
        board[i][col] = 1;
        if (solveNQUtil(board, col + 1) == true)
        return true;
        board[i][col] = 0;
    }
 }
 return false;
}
/* solves the N Queen problem using Backtracking.*/
bool solveNQ()
{
 int board[N][N] = {0};
 if (solveNQUtil(board, 0) == false)
 {
    cout<<"Solution does not exist"<<endl;
    return false;
 }
 printSolution(board);
 return true;
}

int main()
{
 solveNQ();
 return 0;
```

```
}
```

Output:

```
Output

/tmp/DGYb11Undn.o
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
```

****

DAA Mini Project

Title Name: Implement merge sort and multithreaded merge sort. Compare time required by both the algorithms. Also analyze the performance of each algorithm for the best case and the worst case.

Name: Abhishek Santosh Bankar

Class : BE **Div**: 1 **Batch**: A

Roll No: 405A011

****

Title: Implement merge sort and multithreaded merge sort. Compare time required by both the algorithms. Also analyze the performance of each algorithm for the best case and the worst case.

Problem Statement: Write a program for merge sort and multithreaded merge sort and analyze the performance of each algorithm for the best case and the worst case.

Prerequisites: Design and analysis algorithm

Objectives: To understand for merge sort and multithreaded merge sort and analyze the performance of each algorithm for the best case and the worst case.

Theory:

Merge Sort:

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the merge() function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	O(n*logn)

Multi-Threading:

In the operating system, **Threads** are the lightweight process which is responsible for executing the part of a task. Threads share common resources to execute the task concurrently.

Multi-threading is an implementation of multitasking where we can run multiple threads on a single processor to execute the tasks concurrently. It subdivides specific operations within a single application into individual threads. Each of the threads can run in parallel.

For Example-:

In −int arr[] = {3, 2, 1, 10, 8, 5, 7, 9, 4}

Out −Sorted array is: 1, 2, 3, 4, 5, 7, 8, 9, 10

Program :

```
#include <iostream>
using namespace std;
// Merges two subarrays of array[]. First subarray  is arr[begin..mid] .Second subarray is
arr[mid+1..end]
void merge(int array[], int const left, int const mid,
{
        auto const subArrayOne = mid - left + 1;
        auto const subArrayTwo = right - mid;
        // Create temp arrays
        auto *leftArray = new int[subArrayOne],
                *rightArray = new int[subArrayTwo];

       // Copy data to temp arrays leftArray[] and rightArray[]
       for (auto i = 0; i < subArrayOne; i++)
               leftArray[i] = array[left + i];
```

```
    for (auto j = 0; j < subArrayTwo; j++)
        rightArray[j] = array[mid + 1 + j];

    auto indexOfSubArrayOne = 0, // Initial index of first sub-array
        indexOfSubArrayTwo = 0; // Initial index of second sub-array
    int indexOfMergedArray = left; // Initial index of merged array

    // Merge the temp arrays back into array[left..right]
    while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo <
subArrayTwo)
    {
        if (leftArray[indexOfSubArrayOne] <= rightArray[indexOfSubArrayTwo])
        {
            array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
            indexOfSubArrayOne++;
        }
        else
        {
            array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
        }
        indexOfMergedArray++;
    }

    // Copy the remaining elements of left[], if there are any
    while (indexOfSubArrayOne < subArrayOne)
    {
      array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
        indexOfMergedArray++;
    }

    // Copy the remaining elements of right[], if there are any
    while (indexOfSubArrayTwo < subArrayTwo)
    {   array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
        indexOfMergedArray++;
    }
    delete[] leftArray;
    delete[] rightArray;
```

```
}

// begin is for left index and end is right index of the sub-array of arr to be sorted */
void mergeSort(int array[], int const begin, int const end)
{
	if (begin >= end)
		return; // Returns recursively

	auto mid = begin + (end - begin) / 2;
	mergeSort(array, begin, mid);
	mergeSort(array, mid + 1, end);
	merge(array, begin, mid, end);
}

// UTILITY FUNCTIONS - Function to print an array
void printArray(int A[], int size)
{
	for (auto i = 0; i < size; i++)

		cout << A[i] << " ";

}

// Driver code
int main()
{
	int arr[] = { 12, 11, 13, 5, 6, 7 };
	auto arr_size = sizeof(arr) / sizeof(arr[0]);
	cout << "Given array is \n";
	printArray(arr, arr_size);
	mergeSort(arr, 0, arr_size - 1);
	cout << "\nSorted array is \n";
	printArray(arr, arr_size);
	return 0;
}
```

Output:

Output

```
/tmp/DGYb11Undn.o
Given array is
12 11 13 5 6 7
Sorted array is
5 6 7 11 12 13
```

Multithreaded Merge sort

Code:

```
#include <iostream>
#include <pthread.h>
#include <time.h>

// number of elements in array
#define MAX 20


// number of threads
#define THREAD_MAX 4
using namespace std;
// array of size MAX
int a[MAX];
int part = 0;

// merge function for merging two parts
void merge(int low, int mid, int high)
{
	int* left = new int[mid - low + 1];
	int* right = new int[high - mid];

	// n1 is size of left part and n2 is size of right part
	int n1 = mid - low + 1, n2 = high - mid, i, j;

	// storing values in left part
	for (i = 0; i < n1; i++)
```

```
            left[i] = a[i + low];


        // storing values in right part
        for (i = 0; i < n2; i++)
            right[i] = a[i + mid + 1];
        int k = low;
        i = j = 0;


        // merge left and right in ascending order
        while (i < n1 && j < n2)
        {
            if (left[i] <= right[j])
                a[k++] = left[i++];
            else
                a[k++] = right[j++];
        }

        // insert remaining values from left
        while (i < n1)
        {
            a[k++] = left[i++];
        }

        // insert remaining values from right
        while (j < n2)
        {
            a[k++] = right[j++];
        }
}

// merge sort function
void merge_sort(int low, int high)
{
        // calculating mid point of array
        int mid = low + (high - low) / 2;
        if (low < high) {

            // calling first half
            merge_sort(low, mid);
```

```
		// calling second half
		merge_sort(mid + 1, high);

		// merging the two halves
		merge(low, mid, high);
	}
}

// thread function for multi-threading
void* merge_sort(void* arg)
{
	// which part out of 4 parts
	int thread_part = part++;

	// calculating low and high
	int low = thread_part * (MAX / 4);
	int high = (thread_part + 1) * (MAX / 4) - 1;

	// evaluating mid point
	int mid = low + (high - low) / 2;
	if (low < high) {
		merge_sort(low, mid);
		merge_sort(mid + 1, high);
		merge(low, mid, high);
	}
}

// Driver Code
int main()
{
	// generating random values in array
	for (int i = 0; i < MAX; i++)
		a[i] = rand() % 100;

	// t1 and t2 for calculating time for merge sort
	clock_t t1, t2;

	t1 = clock();
	pthread_t threads[THREAD_MAX];
```

```
    // creating 4 threads
    for (int i = 0; i < THREAD_MAX; i++)
        pthread_create(&threads[i], NULL, merge_sort, (void*)NULL);

    // joining all 4 threads
    for (int i = 0; i < 4; i++)
        pthread_join(threads[i], NULL);

    // merging the final 4 parts
    merge(0, (MAX / 2 - 1) / 2, MAX / 2 - 1);
    merge(MAX / 2, MAX/2 + (MAX-1-MAX/2)/2, MAX - 1);
    merge(0, (MAX - 1)/2, MAX - 1);
    t2 = clock();

    // displaying sorted array
    cout << "Sorted array: ";
    for (int i = 0; i < MAX; i++)
        cout << a[i] << " ";

    // time taken by merge sort in seconds
    cout << "Time taken: " << (t2 - t1) / (double)CLOCKS_PER_SEC << endl;
    return 0;
}
```

Output:

Output:

```
Sorted array: 15 21 26 26 27 35 36 40 49 59 62 63 72 77 83 86 86 90 92 93
Time taken: 0.001023
```