IT224 LAB ASSIGNMENT 2

Design and Analysis of Algorithms



**Submitted By**

Name : Deepjyoti Deka

Roll No : 190103014

Semester: 4th Semester , Branch : CSE

**Experiment 1**

Aim : Sort a given set of elements using the Quicksort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted, and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

Objective :To write a program to print the time required by a quicksort algorithm to sort the elements.Repeat the experiment for different values of n - the number of elements in the list to be sorted.Plot a graph of the Size versus Time taken.Use of a time recording library to capture the start and the end time of a process.

Theory

Quicksort is an in-place sorting algorithm. Developed by British computer scientist Tony Hoare in 1959 and published in 1961, it is still a commonly used algorithm for sorting. When implemented well, it can be somewhat faster than merge sort and about two or three times faster than heapsort.

Source Code

#include <iostream>

#include <chrono>

using namespace std;

using namespace std::chrono;

void swap(int \*a, int \*b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

int partition(int arr[], int left, int right)

{

int pivot = arr[right];

int i = (left - 1);

for (int j = left; j < right; j++)

{

if (arr[j] < pivot)

{

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[right]);

return (i + 1);

}

void quickSort(int arr[], int left, int right)

{

if (left < right)

{

int pivot = partition(arr, left, right);

quickSort(arr, left, pivot - 1);

quickSort(arr, pivot + 1, right);

}

}

int graphtime(int arr[], int n)

{

auto start = high\_resolution\_clock::now();

quickSort(arr, 0, n - 1);

auto end = high\_resolution\_clock::now();

auto time\_taken = duration\_cast<microseconds>(end - start);

return (time\_taken.count());

}

void graph()

{

int size[] = {10, 100, 1000, 10000};

int x[4], k = 0;

for (int n = 0; n < 4; n++)

{

int arr[size[n]];

for (int j = 0; j < n; j++)

{

arr[j] = rand() % size[n];

}

x[k] = graphtime(arr, size[n]);

k++;

}

for (int i = 0; i < 4; i++)

{

cout << size[i] << " --> " << x[i] << "ms" << endl;

}

}

int main()

{

int array[1000];

for (int i = 0 ; i<999 ; i++){

array[i] = rand() % 1000;

}

int n = sizeof(array) / sizeof(array[0]);

auto start = high\_resolution\_clock::now();

quickSort(array, 0, n - 1);

auto end = high\_resolution\_clock::now();

auto timetaken = duration\_cast<microseconds>(end-start);

cout << "Sorted array: \n";

for (int i = 0; i < n; i++)

cout << array[i] << " ";

cout << endl;

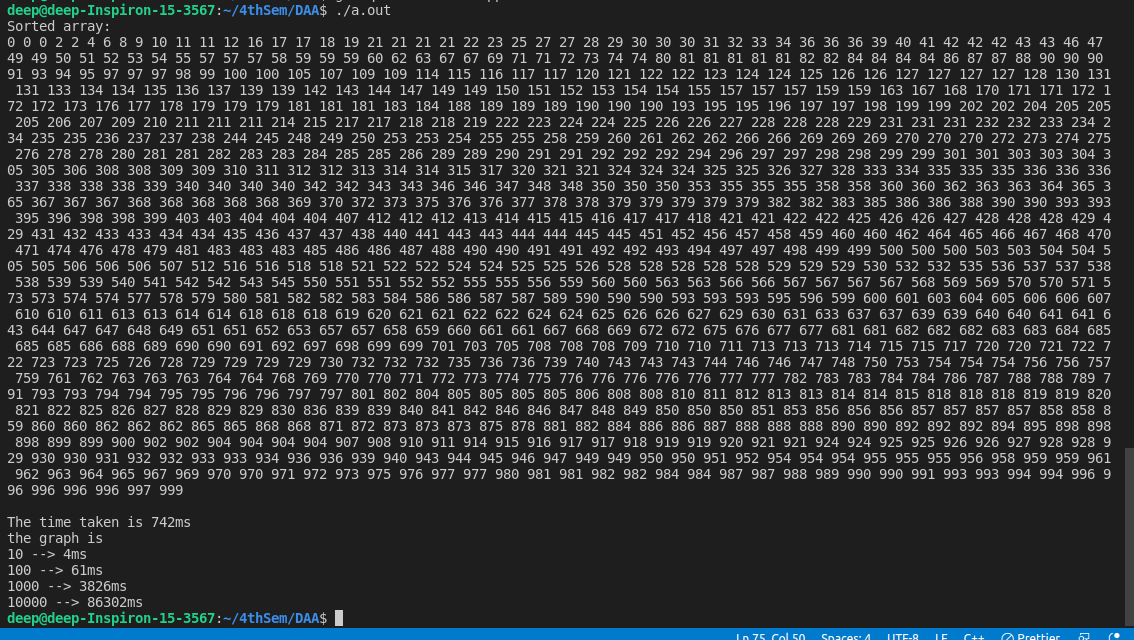
cout << "\nThe time taken is " << timetaken.count() <<"ms"<<endl;

cout<<"the graph is \n";

graph();

}

Output



Result

1000 random elements are sorted and the graph of 10 , 100 , 1000 , 10000 elements is plotted..

**Experiment 2**

Aim : To find size versus time mapping for Bubble sort algorithm.

Objective

The main objective is to write a program to print the time required by a bubble sort algorithm to sort the elements. We repeat the experiment for different values of n - the number of elements in the list to be sorted. Then we plot a graph of the Size versus Time taken. We use a time recording library to capture the start and end time of a process.

Theory

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

Source Code

#include<iostream>

#include<chrono>

#include <string.h>

using namespace std;

using namespace std::chrono;

void swap(int \*a, int \*b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

void printArray(int arr[] , int n){

for (int i = 0; i < n; i++)

cout << arr[i] << " ";

cout << endl;

}

void bubbleSort(int arr[] , int length ) {

int list[length];

memcpy(list, arr, length \* sizeof(int));

int temp;

for(long i = 0; i < length; i++) {

for(long j = 0; j < length-1; j++) {

if(list[j] > list[j+1]) {

temp = list[j];

list[j] = list[j+1];

list[j+1] = temp;

}

}

}

printArray(list , length);

}

int main() {

int n;

cout << "Enter length of array"<<endl;

cin >> n;

int array[n];

for (int i = 0 ; i<n ; i++){

array[i] = rand() % n;

}

printArray(array , n);

cout << "the sorted arrays are below " <<endl;

// timeCapture(&bubbleSort, array , n);

auto start1 = high\_resolution\_clock::now();

bubbleSort(array , n);

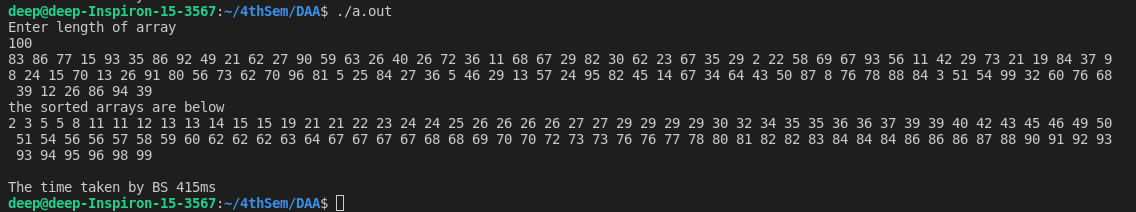
auto start2 = high\_resolution\_clock::now();

auto bubblesorttimetaken = duration\_cast<microseconds>(start2-start1);

cout << "\nThe time taken by BS " << bubblesorttimetaken.count() <<"ms"<<endl;

}

Output



Result

1000 random elements are sorted and the time taken is shown.

**Experiment 3**

Aim : To find size versus time mapping for Selection sort algorithm.

Objective : The main objective is to write a program to print the time required by a selection sort algorithm to sort the elements. We repeat the experiment for different values of n - the number of elements in the list to be sorted. Then we plot a graph of the Size versus Time taken. We use a time recording library to capture start and end time of a process.

Theory

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list. The smallest element is selected from the unsorted array and swapped with the left most element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right. This algorithm is not suitable for large data sets as its average and worst case complexities are of Ο(n2 )

Source Code

#include<iostream>

#include<chrono>

using namespace std;

using namespace std::chrono;

void swap(int\* a, int\* b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

void selectionSort(int arr[], int n) {

int i, j, minIndex;

for(i = 0; i<n-1; i++) {

minIndex = i;

for(j = i+1; j<n; j++){

if (arr[j] < arr[minIndex]){

minIndex = j;

}

swap(arr[i], arr[minIndex]);

}

}

}

void printArray(int arr[], int size)

{

int i;

for (i = 0; i < size; i++)

cout << arr[i] << " ";

cout << endl;

}

int sortTime(int arr[], int n)

{

auto start = high\_resolution\_clock::now();

selectionSort(arr, n);

auto end = high\_resolution\_clock::now();

auto time\_taken = duration\_cast<milliseconds>(end - start);

return (time\_taken.count());

}

void generateRandom(int arr[], int n){

srand(time(0));

for (int i=0;i<n;i++){

arr[i]=rand()%1000;

}

}

int main(){

int arr[999999],n;

cout << "Enter the number of values to be generated: ";

cin >> n;

generateRandom(arr,n);

cout << "Array before sorting" << endl;

cout << "[ ";

for (int i=0;i<n;i++){

cout << arr[i] << " ";

}

cout << " ]";

cout << "\n\n";

auto start = high\_resolution\_clock::now();

selectionSort(arr, n);

auto end = high\_resolution\_clock::now();

auto duration = duration\_cast<milliseconds>(end-start);

cout<<"\nSorted array is: \n" << endl;

cout << "[ ";

for (int i=0;i<n;i++){

cout << arr[i] << " ";

}

cout << " ]";

cout << "\n\nThe time taken for sorting is " << duration.count() << " ms\n\n" << endl;

}

Output



Result

Result

1000 random elements are sorted and the time taken is shown.

**Experiment 4**

Aim : To find size versus time graph mapping for Insertion sort algorithm

Objective

The main objective is to write a program to print the time required by a quick sort algorithm to sort the elements. We repeat the experiment for different values of n - the number of elements in the list to be sorted. Then we plot a graph of the Size versus Time taken. We use a time recording library to capture start and end time of a process.

Theory

The main objective is to write a program to print the time required by a quick sort algorithm to sort the elements. We repeat the experiment for different values of n - the number of elements in the list to be sorted. Then we plot a graph of the Size versus Time taken. We use a time recording library to capture start and end time of a process.

Source Code

#include<iostream>

#include<chrono>

#include <string.h>

using namespace std;

using namespace std::chrono;

void swap(int \*a, int \*b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

void printArray(int arr[] , int n){

for (int i = 0; i < n; i++)

cout << arr[i] << " ";

cout << endl;

}

void insertionSort(int arr[] , int length) {

int list[length];

memcpy(list, arr, length \* sizeof(int));

int temp;

for(long i = 1; i < length; i++) {

temp = list[i];

long j;

for(j = i-1; j >= 0 && list[j] > temp; j--) {

list[j+1] = list[j];

}

list[j+1] = temp;

}

printArray(list , length);

}

int main() {

int n;

cout << "Enter length of array"<<endl;

cin >> n;

int array[n];

for (int i = 0 ; i<n ; i++){

array[i] = rand() % n;

}

printArray(array , n);

cout << "the sorted arrays are below " <<endl;

// timeCapture(&bubbleSort, array , n);

auto start2 = high\_resolution\_clock::now();

insertionSort(array ,n);

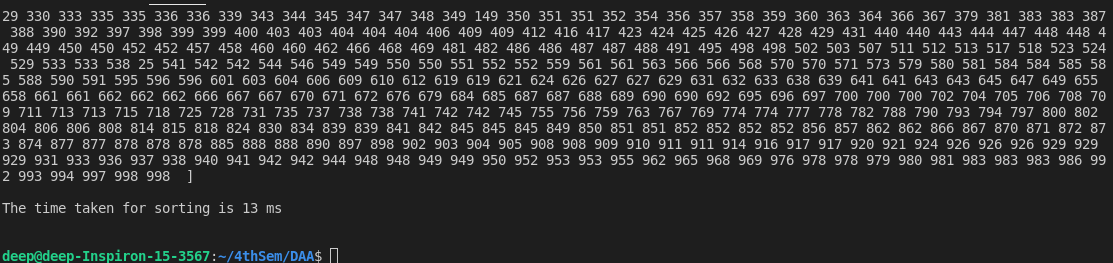
auto start3 = high\_resolution\_clock::now();

auto insertionsorttimetaken = duration\_cast<microseconds>(start3-start2);

cout << "\nThe time taken by IS " << insertionsorttimetaken.count() <<"ms"<<endl;

}

Output



Result

500 random elements are sorted and the time taken while sorting is recorded

**Experiment 5**

Aim : To find size versus time graph mapping for Merge sort algorithm.

Objective

The main objective is to write a program to print the time required by a Merge sort algorithm to sort the elements. We repeat the experiment for different values of n - the number of elements in the list to be sorted. Then we plot a graph of the Size versus Time taken. We use a time recording library to capture start and end time of a process.

Theory

Merge sort is a sorting technique based on divide and conquer technique. With worstcase time complexity being Ο(n log n), it is one of the most respected algorithms. Merge sort first divides the array into equal halves and then combines them in a sorted manner. The mapping of the size versus time is a visualisation of the increasing time with increasing size of the input size. The values of the times taken would differ from pc to pc.

Source Code

#include<iostream>

#include<chrono>

#include <string.h>

using namespace std;

using namespace std::chrono;

void swap(int \*a, int \*b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

void printArray(int arr[] , int n){

for (int i = 0; i < n; i++)

cout << arr[i] << " ";

cout << endl;

}

void merge(int array[], int l, int m, int r) {

int i, j, k, nl, nr;

nl = m-l+1; nr = r-m;

int left\_arr[nl], right\_arr[nr];

for(i = 0; i<nl; i++)

{

left\_arr[i] = array[l+i];

}

for(j = 0; j<nr; j++)

{

right\_arr[j] = array[m+1+j];

}

i = 0; j = 0; k = l;

while(i < nl && j<nr) {

if(left\_arr[i] <= right\_arr[j]) {

array[k] = left\_arr[i];

i++;

}

else{

array[k] = right\_arr[j];

j++;

}

k++;

}

while(i<nl) {

array[k] = left\_arr[i];

i++; k++;

}

while(j<nr) {

array[k] = right\_arr[j];

j++; k++;

}

}

void mergeSort(int arr[], int l, int r) {

if(l < r) {

int mid = (l+r)/2;

mergeSort(arr, l, mid);

mergeSort(arr, mid + 1, r);

merge(arr, l, mid, r);

}

}

void mergesortInitatite(int arr[] , int length){

int list[length];

memcpy(list, arr, length \* sizeof(int));

mergeSort(list , 0 , length-1);

printArray(list , length);

}

int main() {

int n;

cout << "Enter length of array"<<endl;

cin >> n;

int array[n];

for (int i = 0 ; i<n ; i++){

array[i] = rand() % n;

}

printArray(array , n);

cout << "the sorted arrays are below " <<endl;

// timeCapture(&bubbleSort, array , n);

auto start4 = high\_resolution\_clock::now();

mergesortInitatite(array,n);

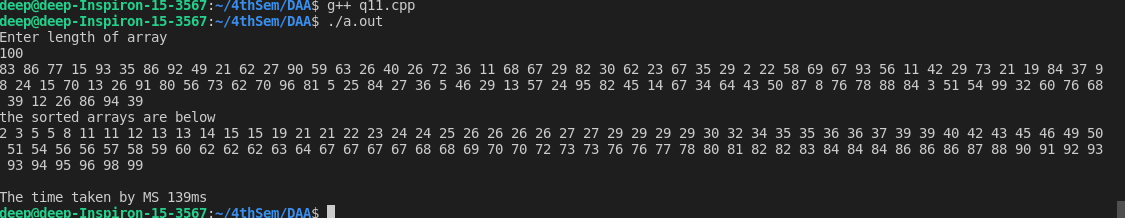
auto start5 = high\_resolution\_clock::now();

auto mergesorttimetaken = duration\_cast<microseconds>(start5-start4);

cout << "\nThe time taken by MS " << mergesorttimetaken.count() <<"ms"<<endl;

}

Output



Result

The first array with 100 elements was sorted and the time taken for sorting was measure in microseconds.

**Experiment 6**

Aim : To find size versus time graph mapping for Heap sort algorithm.

Objective

The main objective is to write a program to print the time required by a Heap sort algorithm to sort the elements. We repeat the experiment for different values of n - the number of elements in the list to be sorted.

Theory

Heapsort is a comparison-based sorting algorithm. Heapsort can be thought of as an improved selection sort: like selection sort, heapsort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region. Unlike selection sort, heapsort does not waste time with a linear-time scan of the unsorted region; rather, heap sort maintains the unsorted region in a heap data structure to more quickly find the largest element in each step. Heapsort is an in-place algorithm, but it is not a stable sort. The mapping of the size versus time is a visualisation of the increasing time with increasing size of the input size. The values of the times taken would differ from pc to pc.

Source Code

#include<iostream>

#include<chrono>

using namespace std;

using namespace std::chrono;

void swap(int\* a, int\* b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

void heapify(int arr[], int n, int i)

{

int largest = i;

int l = 2 \* i + 1;

int r = 2 \* i + 2;

if (l < n && arr[l] > arr[largest])

largest = l;

if (r < n && arr[r] > arr[largest])

largest = r;

if (largest != i) {

swap(arr[i], arr[largest]);

heapify(arr, n, largest);

}

}

void heapSort(int arr[], int n)

{

for (int i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

for (int i = n - 1; i >= 0; i--) {

swap(arr[0], arr[i]);

heapify(arr, i, 0);

}

}

void printArray(int arr[], int size)

{

int i;

for (i = 0; i < size; i++)

cout << arr[i] << " ";

cout << endl;

}

int sortTime(int arr[], int n)

{

auto start = high\_resolution\_clock::now();

heapSort(arr, n);

auto end = high\_resolution\_clock::now();

auto time\_taken = duration\_cast<milliseconds>(end - start);

return (time\_taken.count());

}

void graph()

{

int size[] = {50, 1000, 5000, 10000};

int x[4], k = 0;

for (int n = 0; n < 4; n++)

{

int arr[size[n]];

for (int j = 0; j < n; j++)

{

arr[j] = rand() % size[n];

}

x[k] = sortTime(arr, size[n]);

k++;

}

for (int i = 0; i < 4; i++)

{

cout << size[i] << " --> " << x[i] << " ms" << endl;

}

}

void generateRandom(int arr[], int n){

srand(time(0));

for (int i=0;i<n;i++){

arr[i]=rand()%1000;

}

}

int main(){

int arr[999999],n;

cout << "Enter the number of values : ";

cin >> n;

generateRandom(arr,n);

cout << "Array before sorting\n\n" << endl;

for (int i=0;i<n;i++){

cout << arr[i] << " ";

}

auto start = high\_resolution\_clock::now();

heapSort(arr, n);

auto end = high\_resolution\_clock::now();

auto duration = duration\_cast<milliseconds>(end-start);

cout<<"Sorted array is:" << endl;

cout << "[ ";

for (int i=0;i<n;i++){

cout << arr[i] << " ";

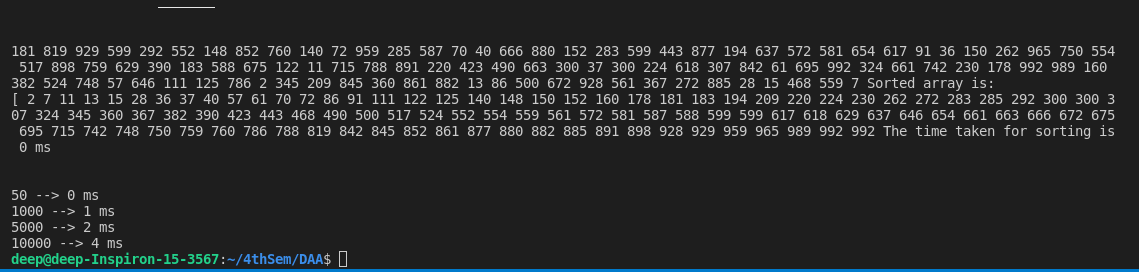
}

cout << "The time taken for sorting is " << duration.count() << " ms\n\n" << endl;

graph();

}

Output



Result

An array with 100 elements was sorted and the time taken for sorting was measure in microseconds. Graph mapping is plotted for different size values.

**Experiment 7**

Aim : To find a solution for the activity selection problem using greedy method.

Objective

The objective is to find a solution set having maximum number of non-conflicting activities that can be executed in a single time frame.

Theory

Activity Selection sort states that “Given a set of n activities with their start and finish times, we need to select maximum number of nonconflicting activities that can be performed by a single Person,given that the person can handle only one activity at a time

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Greedy algorithms are used for optimization problems.

Source Code

#include <stdio.h>

void ActivitySelection(int start[], int finish[], int n)

{

printf("The following activities choosed:\n");

int j = 0;

printf("%d ", j);

int i;

for (i = 1; i < n; i++)

{

if (start[i] >= finish[j])

{

printf("%d ", i);

j = i;

}

}

}

int main()

{

int n;

printf("Enter the no of activities: ");

scanf("%d", &n);

int start[n];

int finish[n];

printf("Enter the start and finish time\n");

for (int i = 0; i < n-1; i++)

{

scanf("%d", &start[i]);

}

for (int i = 0; i < n-1; i++)

{

scanf("%d", &finish[i]);

}

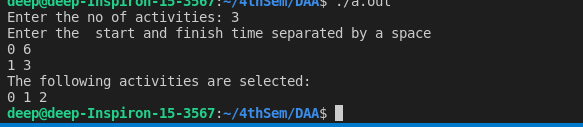
ActivitySelection(start, finish, n);

printf("\n");

return 0;

}

Output



Result

The algorithm choose the best local optimal way to select the activity..

**Experiment 8**

Aim : To find a maximum profit and number of jobs done.

Objective :

To determine how the next job is selected for an optimal solution and to sort all jobs in decreasing order of profit.and then iterate on jobs in decreasing order of profit.

Theory

Sequencing can be defined as the selection of an order for a series of jobs to be done on a number of service facilities (machine). In sequencing, a systematic procedure is adopted in assigning priorities to waiting jobs thereby determining the sequence in which jobs will be processes

Source Code

#include <stdio.h>

#define MAX 100

typedef struct Job {

char id[5];

int deadline;

int profit;

} Job;

void jobSequenanddeadlin(Job jobs[], int n);

int minValue(int x, int y) {

if(x < y) return x;

return y;

}

int main(void) {

int i, j;

Job jobs[5] = {

{"j1", 2, 30},

{"j2", 1, 40},

{"j3", 3, 10},

{"j4", 2, 50},

{"j5", 1, 20},

};

Job temp;

int n = 5;

for(i = 1; i < n; i++) {

for(j = 0; j < n - i; j++) {

if(jobs[j+1].profit > jobs[j].profit) {

temp = jobs[j+1];

jobs[j+1] = jobs[j];

jobs[j] = temp;

}

}

}

printf("%10s %10s %10s\n", "Job", "Deadline", "Profit");

for(i = 0; i < n; i++) {

printf("%10s %10i %10i\n", jobs[i].id, jobs[i].deadline, jobs[i].profit);

}

jobSequenanddeadline(jobs, n);

return 0;

}

void jobSequenanddeadline(Job jobs[], int n) {

int i, j, k, maxprofit;

int timeslot[MAX];

int filledTimeSlot = 0;

int dmax = 0;

for(i = 0; i < n; i++) {

if(jobs[i].deadline > dmax) {

dmax = jobs[i].deadline;

}

}

for(i = 1; i <= dmax; i++) {

timeslot[i] = -1;

}

printf("dmax: %d\n", dmax);

for(i = 1; i <= n; i++) {

k = minValue(dmax, jobs[i - 1].deadline);

while(k >= 1) {

if(timeslot[k] == -1) {

timeslot[k] = i-1;

filledTimeSlot++;

break;

}

k--;

}

if(filledTimeSlot == dmax) {

break;

}

}

printf("Here Required Jobs: ");

for(i = 1; i <= dmax; i++) {

printf("%s", jobs[timeslot[i]].id);

if(i < dmax) {

printf(" --> ");

}

}

maxprofit = 0;

for(i = 1; i <= dmax; i++) {

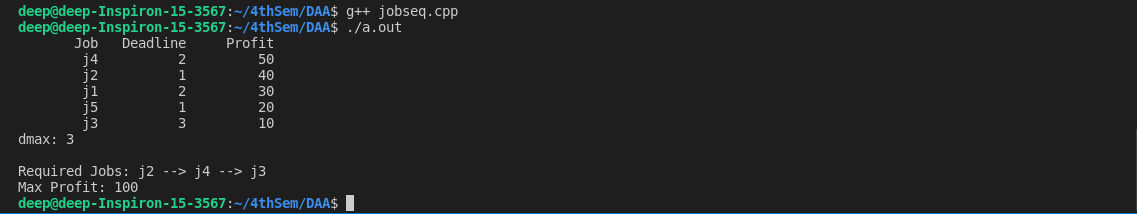
maxprofit += jobs[timeslot[i]].profit;

}

printf("Maximum attained Profit: %d\n", maxprofit);

}

Output



Result

The algorithm choosed the maximum profit and the required jobs for it.

**Experiment 9**

Aim : To find optimum solution for knapsack problem

Objective

The objective is to fill the knapsack of some given volume with different materials such that the value of selected items is maximized.

Theory

The knapsack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Source Code

#include <stdio.h>

int max(int a, int b) { return (a > b) ? a : b; }

int knapSack(int W, int weightt[], int val[], int n)

{

if (n == 0 || W == 0)

return 0;

if (wt[n - 1] > W)

return knapSack(W, weightt, val, n - 1);

else

return max(

val[n - 1]

+ knapSack(W - weightt[n - 1],

weightt, val, n - 1),

knapSack(W, weightt, val, n - 1));

}

int main()

{

int W = 50;

int arr[] = { 40, 200, 80 };

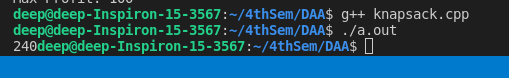
int wt[] = { 10, 20, 40 };

int n = sizeof(val) / sizeof(val[0]);

printf("%d", knapSack(W, weightt, val, n));

}

Output



Result

The greedy algorithm finds the optimal solution for the knapsack problem

**Experiment 10**

Aim : To find size versus time graph mapping for Quicksort algorithm.

Objective

• To write a program to print the time required by a quicksort algorithm to sort the elements.

• Repeat the experiment for different values of n - the number of elements in the list to be sorted.

• Plot a graph of the Size versus Time taken.

• Use of a time recording library to capture start and end time of a process.

Theory

Quicksort is a sorting algorithm based on the divide and conquer approach where an array is divided into sub-arrays by selecting a random pivot element (element selected from the array). While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot. The left and right sub-arrays are also divided using the same approach. This process continues until each sub-array contains a single element. At this point the array will be sorted.

The mapping of the size versus time is a visualisation of the increasing time with increasing size of the input size. The values of the times taken would differ from pc to pc.

Source Code

#include <iostream>

#include <chrono>

using namespace std;

using namespace std::chrono;

void swap(int \*a, int \*b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

int partition(int arr[], int left, int right)

{

int pivot = arr[right];

int i = (left - 1);

for (int j = left; j < right; j++)

{

if (arr[j] < pivot)

{

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[right]);

return (i + 1);

}

void quickSort(int arr[], int left, int right)

{

if (left < right)

{

int pivot = partition(arr, left, right);

quickSort(arr, left, pivot - 1);

quickSort(arr, pivot + 1, right);

}

}

int graphtime(int arr[], int n)

{

auto start = high\_resolution\_clock::now();

quickSort(arr, 0, n - 1);

auto end = high\_resolution\_clock::now();

auto time\_taken = duration\_cast<microseconds>(end - start);

return (time\_taken.count());

}

void graph()

{

int size[] = {10, 100, 1000, 10000};

int x[4], k = 0;

for (int n = 0; n < 4; n++)

{

int arr[size[n]];

for (int j = 0; j < n; j++)

{

arr[j] = rand() % size[n];

}

x[k] = graphtime(arr, size[n]);

k++;

}

for (int i = 0; i < 4; i++)

{

cout << size[i] << " --> " << x[i] << "ms" << endl;

}

}

int main()

{

int array[1000];

for (int i = 0 ; i<999 ; i++){

array[i] = rand() % 1000;

}

int n = sizeof(array) / sizeof(array[0]);

[i] << " ";

c