MALABAR INSTITUTE OF TECHNOLOGY, ANJARAKANDY

ANJARAKANDY
INTEGRATED CAMPUS
KANNUR - 670 612, KERALA

LAB RECORD

MCS207(P) SOFTWARE SYSTEMS LAB

By

YOUR NAME

YOUR REG NUMBER

DEPARTMENT OF CSE

MALABAR INSTITUTE OF TECHNOLOGY, ANJARAKANDY - 670612
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ANJARAKANDY
INTEGRATED CAMPUS
KANNUR - 670 612, KERALA

CERTIFICATE

Certified that this is a bonafide record of the laboratory work done in the
MCS207(P) SOFTWARE SYSTEMS LAB
by
YOUR NAME
Your Reg No
of second semester M. Tech in partial fulfillment of the requirements for the award of the degree of Master of Technology in Computer Science and Engineering of the Kannur University during the academic year 2014-2015

Lab in Charge
Name
Designation

Head of Department
Mr. Rijin I K
Assistant Professor

Internal Examiner

External Examiner

Contents

1 FINDING k-th LARGEST ELEMENT

Experiment Number : 1 Date : 05-11-2014

1.1 AIM

Write a program to find the k-th largest element in an array.

1.2 OBJECTIVE

The aim can be achieved by modifying a selection sort program that sorts n elements in descending order to stop iteration after selecting k elements.

1.3 THEORETICAL BACKGROUND

Selection sort is a comparison based sort that works with a complexity of $O(n^2)$. For each position starting from the first, it selects the appropriate element for that position in each iteration. This is why it is called *Selection Sort*. The advantage of this method is that the array will be sorted starting from the initial position. After each iteration, the size of the sorted sub-array increases. Thus we can access the k-th largest element after k iterations of the position selector loop of the selection sort.

1.4 PROCEDURE

The procedure adopted for implementing the aim is as follows.

1. Read the value of n, the array of n numbers and k
2. Run the selection sort to sort first k elements in decreasing order
3. Return the k-th element in the array as the answer

1.5 ALGORITHMS

1.5.1 Modified Selection Sort

Algorithm 1 : kSelectionSort(A, size, k)

```
1: for i=0 to k do
2:     for j=i+1 to n-1 do
3:         if A[i] < A[j] then
4:             temp = A[i]
5:             A[i] = A[j]
6:             A[j] = temp
7:         end if
8:     end for
9: end for
10: return A[k-1]
```

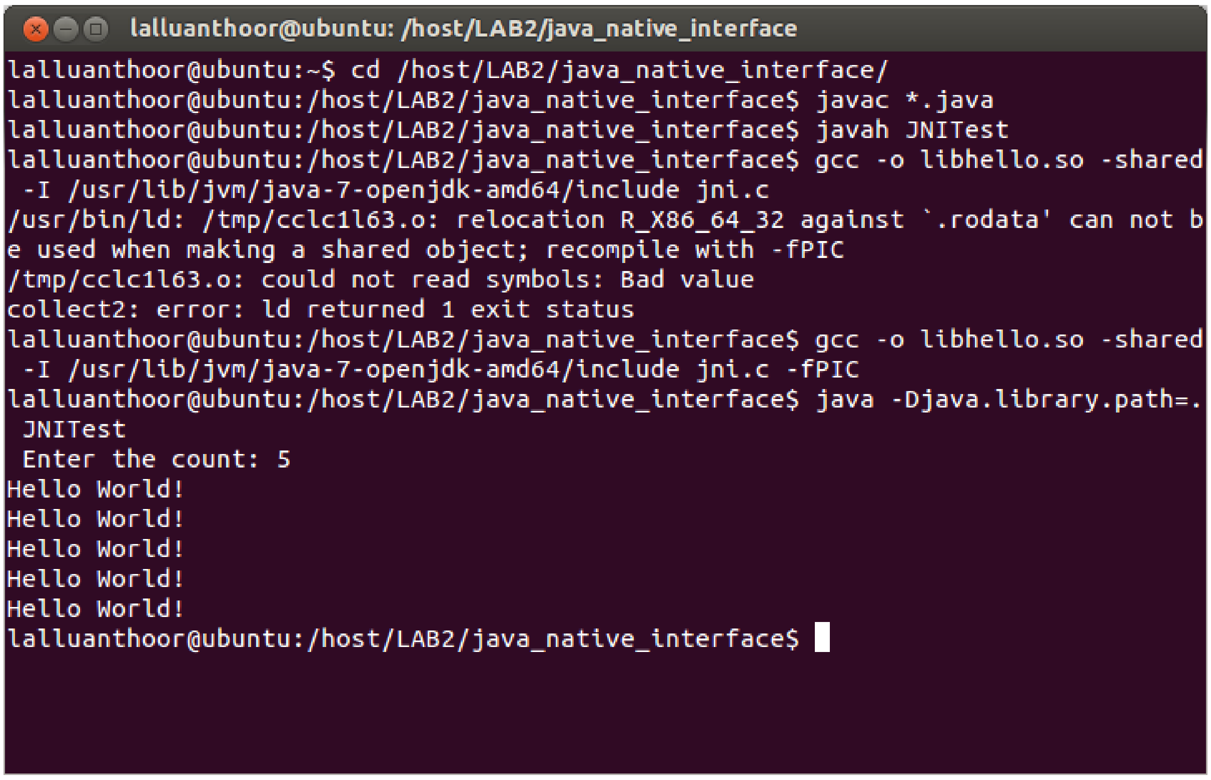
The input parameters to the algorithm are the array (denoted as A), the number of elements in that array (denoted as n) and the value of k (denoted as k). The output of the algorithm will be the k-th largest element of that array.

1.6 PROGRAM

1.6.1 kth_largest.c

```c
#include<stdio.h>

#define MAX_SIZE 10

int k_selection_sort( int[], int, int );

int main( int argc, char ** argv ) {
	int n, k;
	int A[MAX_SIZE];

	printf( "Enter the number of elements (n) : " );
	scanf( "%d", &n );

	if( n > MAX_SIZE ) {
		printf( "Cannot store more than %d elements.", MAX_SIZE );
		return 1;
	} else {
		int i, kth;
		printf( "Enter the %d elements separated by spaces : ", n );
		for( i=0; i<n; i++ ) {
			scanf( "%d", &A[i] );
		}
		printf( "Enter the value for k : " );
		scanf( "%d", &k );
		if( k > n ) {
			printf( "k cannot be greater than n." );
			return 2;
		}

		kth = k_selection_sort( A, n, k );

		printf( "The %dth largest element is : %d\n", k, kth );
	}

	return 0;
}

int k_selection_sort( int A[], int n, int k ) {
	int i, j;
	for( i=0; i<k; i++ ) {
		for( j=i+1; j<n; j++ ) {
			if( A[i] < A[j] ) {
				int temp = A[i];
				A[i] = A[j];
				A[j] = temp;
			}
		}
	}

	return A[k-1];
}
```

1.7 SAMPLE OUTPUT



```
lalluanthoor@ubuntu:~$ cd /host/LAB2/java_native_interface/
lalluanthoor@ubuntu:/host/LAB2/java_native_interface$ javac *.java
lalluanthoor@ubuntu:/host/LAB2/java_native_interface$ javah JNITest
lalluanthoor@ubuntu:/host/LAB2/java_native_interface$ gcc -o libhello.so -shared
 -I /usr/lib/jvm/java-7-openjdk-amd64/include jni.c
/usr/bin/ld: /tmp/cclc1l63.o: relocation R_X86_64_32 against `.rodata' can not b
e used when making a shared object; recompile with -fPIC
/tmp/cclc1l63.o: could not read symbols: Bad value
collect2: error: ld returned 1 exit status
lalluanthoor@ubuntu:/host/LAB2/java_native_interface$ gcc -o libhello.so -shared
 -I /usr/lib/jvm/java-7-openjdk-amd64/include jni.c -fPIC
lalluanthoor@ubuntu:/host/LAB2/java_native_interface$ java -Djava.library.path=.
 JNITest
 Enter the count: 5
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
lalluanthoor@ubuntu:/host/LAB2/java_native_interface$ 
```

Figure 1: Screen-shot of Output

1.8 RESULT

The k-th largest element of an array was found out and displayed. The scalability and the accuracy of the program were verified.

1.9 INFERENCES

The selection sort algorithm for sorting elements in descending order can be used to find the k-th largest element. The time complexity of the modified selection sort was found to be $O(n.k)$.

2 ALICE IN WONDERLAND

Experiment Number : 2 Date : 06-11-2014

2.1 AIM

Alice and her family are residents of *Wonderland*. One day Alice decided to pay a visit to her uncle who is living in a far away place. She looked up the possible routes using *Google Maps*. She was astounded by the number of paths returned by *Google Maps*. Your task is to help Alice find the least possible amount of money with which she can make the travel to her uncle's home.

The transportation system of *Wonderland* was created in a special way to avoid accidents. All the roads in *Wonderland* are one-way roads. But the roads are designed in such a way that one can reach from a place to any other place.

2.2 OBJECTIVE

The problem at hand can be solved using the Dijkstra's Algorithm for finding the shortest path between two nodes in a graph. The cities in Wonderland can be seen as the nodes and the roads can be seen as the edges.

From the description of the transportation system, it is evident that the edges are directional. Thus, we can use the matrix representation of graph for solving the problem.

2.3 THEORETICAL BACKGROUND

Dijkstra's algorithm, conceived by computer scientist Edsger Dijkstra in 1956 and published in 1959, is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree. This algorithm is often used in routing and as a subroutine in other graph algorithms.

For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path algorithm is widely used in network routing protocols, most notably IS-IS and OSPF (Open Shortest Path First).

2.4 PROCEDURE

The procedure adopted for implementing the aim is as follows.

1. Read the number of cities, the details of the roads in between them
2. Read the source and destination cities
3. Run the Dijkstra's algorithm for these two cities
4. Return the cost

2.5 ALGORITHMS

2.5.1 Dijkstra's Algorithm

Algorithm 2 : modified_Dijkstras(*Graph*, *source*, *destination*)

```
dist[source] := 0
for all vertex v ∈ Graph do
    if v ≠ source then
        dist[v] := infinity
        prev[v] := undefined
    end if
    Q.add_with_priority(v, dist[v])
end for
while Q is not empty do
    u := Q.extract_min()
    mark u as scanned
    for all neighbor v of u do
        if v is not yet scanned then
            alt = dist[u] + length( u, v )
            if alt < dist[v] then
                dist[v] := alt
                prev[v] := u
                Q.decrease_priority( v, alt )
            end if
        end if
    end for
end while
return  dist[destination]
```

The input parameters to the algorithm are the graph, source and destination. The array *dist*[*i*] stores the distance between the source and the corresponding city *i*. The array *prev*[*i*] stores the city visited just before the city *i*.

The data structure *Q* is a priority queue with operations such as *add_with_priority*, *decrease_priority*, etc. The function $length(u, v)$ will give the cost of travelling from city *u* to *v*.

2.6 PROGRAM

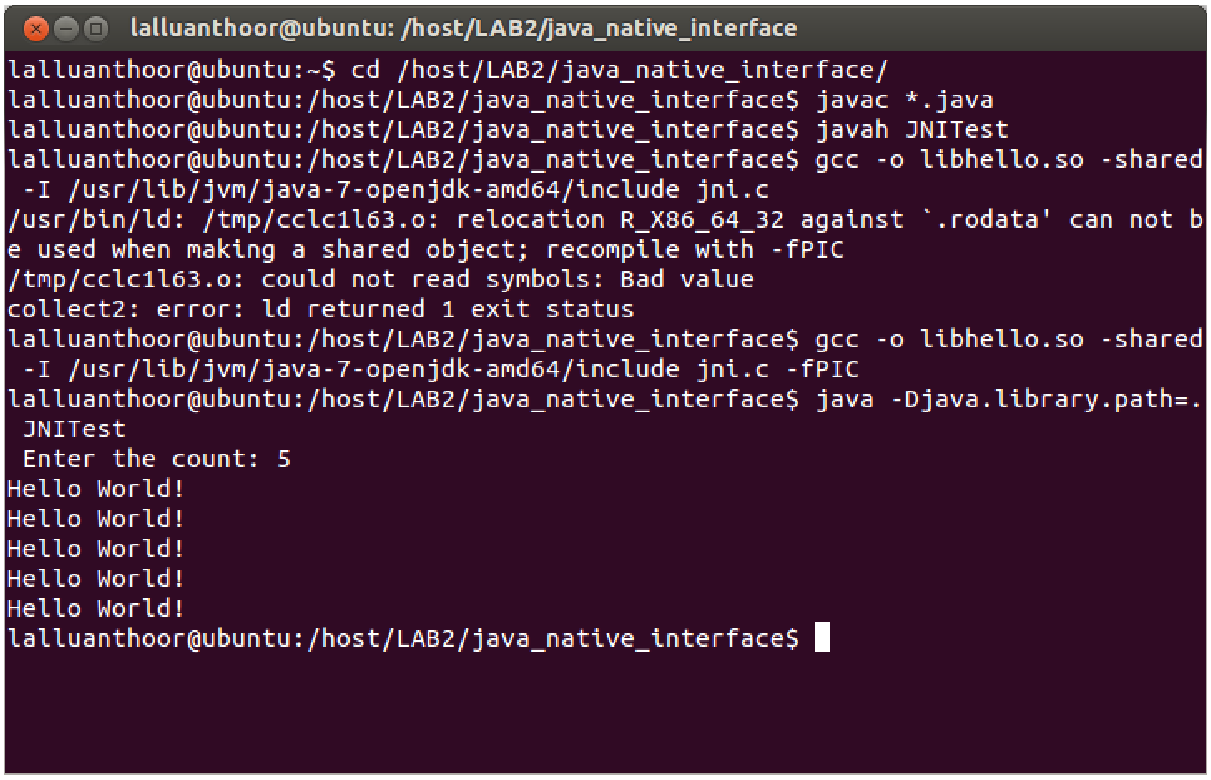
2.6.1 Alice.c

```c
#include<stdio.h>
#include<stdlib.h>

#define MAX 1001
#define INF 9999
#define INITIAL 0
#define WAITING 1
#define FINISHED 2

struct node {
	int data;
	int priority;
	struct node *link;
};

int ADJACENT[MAX][MAX];
int DISTANCE[MAX];
int STATUS[MAX];

void init( int );
void dijkstra( int, int, int );
void enqueue( struct node **, struct node **, int , int );
int dequeue( struct node **, struct node ** );
void decrease_priority( struct node **, struct node **, int , int );
int is_empty( struct node * );

int main( int argc, char **argv )
{
	int v1, v2, w, N, s, i, d;
	scanf( "%d", &N );

	init( N );

	while( 1 )
	{
		scanf( "%d %d %d", &v1, &v2, &w );
		if( v1 == -1 && v2 == -1 && w == -1 )
			break;
		if( ADJACENT[v1][v2] > w )
			ADJACENT[v1][v2] = w;
	}

	scanf( "%d %d", &s, &d );

	dijkstra( s, N, d );
	if( DISTANCE[d] < INF )
		printf( "%d\n", DISTANCE[d] );
	else
		printf( "IMPOSSIBLE\n" );

	return 0;
}

void init( int count )
{
	int i, j;
	for( i=1; i<=count; i++ )
	{
		for( j=1; j<=count; j++ )
			if( i != j )
				ADJACENT[i][j] = INF;
		DISTANCE[i] = INF;
		STATUS[i] = INITIAL;
	}
}

void dijkstra( int source, int N, int destination )
{
	struct node *front=NULL, *rear=NULL;
	int i, v, alt;

	DISTANCE[source] = 0;
	for( i=1; i<=N; i++ )
	{
		if( STATUS[i] == INITIAL )
		{
			enqueue( &front, &rear, i, DISTANCE[i] );
			STATUS[i] = WAITING;
		}
	}

	while( !is_empty( front ) )
	{
		v = dequeue( &front, &rear );
		STATUS[v] = FINISHED;

		if( v == destination )
			return;

		for( i=1; i<=N; i++ )
		{
			if( ADJACENT[v][i] != INF && STATUS[i] == WAITING )
			{
				alt = DISTANCE[v] + ADJACENT[v][i];
				if( alt < DISTANCE[i] )
```

```
                        {
                              DISTANCE[i] = alt;
                              decrease_priority( &front, &rear, i, alt );
                        }
                  }
            }
      }
}

void enqueue( struct node **frontp, struct node **rearp, int data, int
   priority )
{
      struct node *new_node = (struct node *)malloc(sizeof(struct node));
      new_node->data = data;
      new_node->priority = priority;
      new_node->link = NULL;

      if( *rearp == NULL )
            *frontp = *rearp = new_node;
      else if( priority <= (*frontp)->priority )
      {
            new_node->link = *frontp;
            *frontp = new_node;
      }
      else if( priority >= (*rearp)->priority )
      {
            (*rearp)->link = new_node;
            *rearp = new_node;
      }
      else
      {
            struct node *parent = *frontp, *temp = (*frontp)->link;

            while( temp != NULL && temp->link != NULL && temp->priority <
               priority )
            {
                  parent = temp;
                  temp = temp->link;
            }
            parent->link = new_node;
            new_node->link = temp;
      }
}

int dequeue( struct node **frontp, struct node **rearp )
{
      int data = (*frontp)->data;
      *frontp = (*frontp)->link;
      if( *frontp == NULL )
            *rearp = NULL;
```

```
        return data;
}

void decrease_priority( struct node **frontp, struct node **rearp, int data,
    int priority )
{
        struct node *p = *frontp, *t=(*frontp)->link;
        if( t == NULL || p->data == data )
                p->priority = priority;
        else
        {
                while( t->data != data )
                {
                        p = t;
                        t = t->link;
                }
                p->link = t->link;
                enqueue( frontp, rearp, data, priority );
        }
}

int is_empty( struct node *front )
{
        if( front == NULL )
                return 1;
        return 0;
}
```

2.7 SAMPLE OUTPUT



```
lalluanthoor@ubuntu:~$ cd /host/LAB2/java_native_interface/
lalluanthoor@ubuntu:/host/LAB2/java_native_interface$ javac *.java
lalluanthoor@ubuntu:/host/LAB2/java_native_interface$ javah JNITest
lalluanthoor@ubuntu:/host/LAB2/java_native_interface$ gcc -o libhello.so -shared
 -I /usr/lib/jvm/java-7-openjdk-amd64/include jni.c
/usr/bin/ld: /tmp/cclc1l63.o: relocation R_X86_64_32 against `.rodata' can not b
e used when making a shared object; recompile with -fPIC
/tmp/cclc1l63.o: could not read symbols: Bad value
collect2: error: ld returned 1 exit status
lalluanthoor@ubuntu:/host/LAB2/java_native_interface$ gcc -o libhello.so -shared
 -I /usr/lib/jvm/java-7-openjdk-amd64/include jni.c -fPIC
lalluanthoor@ubuntu:/host/LAB2/java_native_interface$ java -Djava.library.path=.
 JNITest
 Enter the count: 5
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
lalluanthoor@ubuntu:/host/LAB2/java_native_interface$ 
```

Figure 2: Screen-shot of Output

2.8 RESULT

The least possible ticket cost for Alice to reach her uncle's home is found out and displayed.

2.9 INFERENCES

The Single Source Shortest Path algorithm deviced by Dijkstra was implemented using C. The graph was represented using adjacency matrix and the Dijkstra's algorithm was implemented using a *Priority Queue*.