SCHOOL OF TECHNOLOGY

PANDIT DEENDAYAL ENERGY UNIVERSITY GANDHINAGAR, GUJARAT, INDIA

Computer Science & Engineering LAB File

(2023-24)

Design and Analysis

of

 Algorithm  Lab

  (20CP209P)

**Student Name:** Heet Dobariya

**Enrollment No.:** 22BCP177

**Semester:** IVth

**Division:** 3

**Group:**  G5

**Instructor:** Hargeet Kaur

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **Exp. No.** | **Experiment Title** | **Date** | **Sign** |
| 1 | Implement the following sorting in C programming language.   1. Insertion sort 2. Selection sort   Now, measure the execution time and the number of steps required to execute each algorithm in best case, worst case, and average case. | 16/01/2024 |  |
| 2 | Implement the following sorting in C programming language.   1. Merge sort 2. Quick sort   Now, measure the execution time and the number of steps required to execute each algorithm in best case, worst case, and average case. | 23/01/2024 |  |
| 3 | Use singly linked lists to implement integers of unlimited size. Each node of the list should store one digit of the integer. You should implement addition, subtraction, multiplication, and exponentiation operations. Limit exponents to be positive integers.    What is the asymptotic running time for each of your operations, expressed in terms of the number of digits for the two operands of each function? | 06/02/2024 |  |
| 4 | Implement a city database using unordered lists. Each database record contains the name of the city (a string of arbitrary length) and the coordinates of the city expressed as integer x and y coordinates. Your program should allow following functionalities:   1. Insert a record, 2. Delete a record by name or coordinate, 3. Search a record by name or coordinate. 4. Print all records a given distance of a specified point.   Implement the database using an array-based list implementation, and then a linked list implementation. Perform following analysis:   1. Collect running time statistics for each operation in both implementations. 2. What are your conclusions about the relative advantages and disadvantages of the two within implementations? 3. Would storing records on the list in alphabetical order by city name speed any of the operations? 4. Would keeping the list in alphabetical order slow any of the operations? | 20/02/2024 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Exp. No.** | **Experiment Title** | **Date** | **Signature** |
| 5 (1) | Implement interval scheduling algorithm. Given n𝒏 events with their starting and ending times, find a schedule that includes as many events as possible. It is not possible to select an event partially | 05/03/2024 |  |
| 5 (2) | Implement the Kruskal’s Algorithm to find the minimum spanning tree of a given graph using Union Find Method. | 05/03/2024 |  |
| 6 | Implement both a standard O(n3) matrix multiplication algorithm and Strassen’s matrix multiplication algorithm. Using empirical testing, try and estimate the constant factors for the runtime equations of the two algorithms. How big must 𝒏 be before Strassen’s algorithm becomes more efficient than the standard algorithm? | 19/03/2024 |  |
| 7 | Implement the Floyd Warshall Algorithm for All Pair Shortest Path Problem. You are given a weighted diagraph G = (V, E), with arbitrary edge weights or costs 𝒄𝒗𝒘 between any node 𝒗 and node 𝒘. Find the cheapest path from every node to every other node. | 26/03/2024 |  |
| 8 | Solve the n queens’ problem using backtracking. Here, the task is to place n chess queens on an n x n board so that no two queens attack each other. | 02/04/2024 |  |
| 9 | Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point. Solve this problem using branch and bound technique. | 09/04/2024 |  |
| 10 | To design and solve given problems using different algorithmic approaches and analyze their complexity. | 16/04/2024 |  |

**Practical No. 1:**

Implement the following sorting in any programming language.

1. Insertion sort
2. Selection sort

Now, measure the execution time and the number of steps required to execute each algorithm in best case, worst case, and average case.

1. Insertion sort:

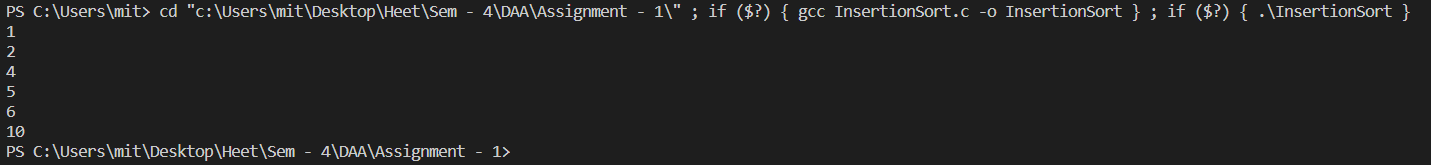
|  |  |  |  |
| --- | --- | --- | --- |
| Code | Frequency  (for worst case) | Frequency  (for best case) | Cost per execution |
| #include <stdio.h>  void main()  { |  |  |  |
| int arr[6] = {5, 4, 10, 1, 6, 2};  int n = 6;  int pivot; | 1  1  1 | 1  1  1 | C1  C2  C3 |
| for (int i = 1; i < n; i++) { | n | n | C4 |
| pivot = arr[i];  int j = i - 1; | n-1  n-1 | n-1  n-1 | C5  C6 |
| while (j >= 0 && arr[j] > pivot) { | C:/Users/mit/AppData/Local/Temp/wps.ooaLZWwps | 1 | C7 |
| arr[j + 1] = arr[j];  j = j - 1;  } | C:/Users/mit/AppData/Local/Temp/wps.Iyvwqwwps | 0  0 | C8  C9 |
| arr[j + 1] = pivot;  } | n-1 | n-1 | C10 |
| for (int i = 0; i < n; i++) | n | n | C11 |
| printf("%d \n", arr[i]); | n-1 | n-1 | C12 |
| } |  |  |  |

Best Case Time Complexity = Ώ(n)

Worst Case Time Complexity = O(n^2)

Average Case Time Complexity = Ɵ(n^2)

* Output:



1. Selection sort:

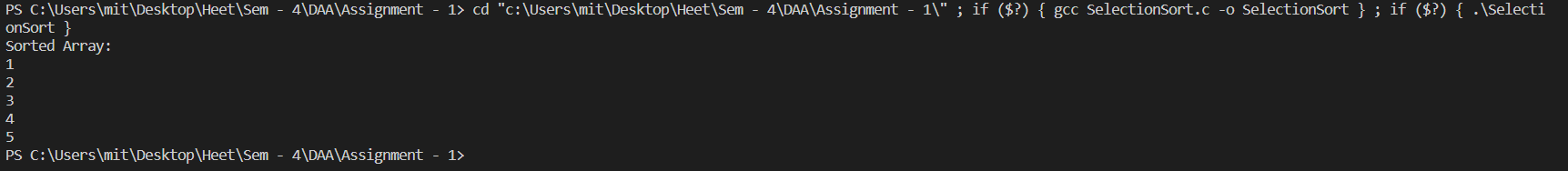
|  |  |  |
| --- | --- | --- |
| Code | Frequency  (for worst and best case both) | Constant |
| #include <stdio.h>  void main()  { |  |  |
| int arr[5]={5,3,4,2,1};  int n=5;  int min; | 1  1  1 | C1  C2  C3 |
| for (int i = 0; i < (n - 1); i++) { | n | C4 |
| min = i; | n-1 | C5 |
| for (int j = i + 1; j < n; j++) { | n\*(n-1) | C6 |
| if (arr[min] > arr[j])  min = j;  } | (n-1)\*(n-1)  (n-1)\*(n-1) | C7  C8 |
| if (min != i)  {  int swap = arr[i];  arr[i] = arr[min];  arr[min] = swap;  }  } | n-1  n-1  n-1  n-1 | C9  C10  C11  C12 |
| printf("Sorted Array: \n"); | 1 | C13 |
| for (int i = 0; i < n; i++) | n | C14 |
| printf("%d\n", arr[i]); | n-1 | C15 |
| } |  |  |

Best Case Time Complexity = Ώ(n2)

Worst Case Time Complexity = O(n2)

Average Case Time Complexity = Ɵ(n2)

* Output:



**Practical No. 2:**

Implement the following sorting in any programming language.

1. Merge sort
2. Quick sort

Now, measure the execution time and the number of steps required to execute each algorithm in best case, worst case, and average case.

1. Merge Sort:

|  |  |  |
| --- | --- | --- |
| Code | Frequency | Cost |
| # include <stdio.h>  void merge(int arr[], int lb, int mid, int ub){  int i = lb;  int j = mid + 1;  int k = lb;  int A[10];  while (i<=mid && j<=ub){  if (arr[i] <= arr[j]){  A[k] = arr[i];  i++;  }  else{  A[k] = arr[j];  j++;  }  k++;  }  if (i>mid){  while (j<=ub){  A[k] = arr[j];  k++;  j++;  }  }  else{  while (i<=mid){  A[k] = arr[i];  k++;  i++;  }  }  for (k = lb; k <= ub; k++){  arr[k] = A[k];  }  }  void MergeSort(int arr[], int lb, int ub)  {  if (lb < ub){  int mid = (lb+ub)/2;  MergeSort(arr, lb, mid);  MergeSort(arr, mid+1, ub);  merge(arr, lb, mid, ub);  }  }  void main()  {  int array[5] = {1,5,4,3,2};  MergeSort(array, 0, 4);  for (int i=0; i<=4; i++){  printf("%d \n", array[i]);  }  } | 1  1  1  1  n+1  n  n  n  1  1  n  1  n+1  n  n  n  n+1  n  n  n  n+1  n  1  1  n/2  n/2  n | C1  C2  C3  C4  C5  C6  C7  C8  C9  C10  C11  C12  C13  C14  C15  C16  C16  C17  C18  C19  C20  C21  C22  C23  C24  C25  C26 |

* Complexity:

T(n) = T(n/2) + T(n/2) + n

T(n) = 2 \* T(n/2) + n

T(n) = 2 \*[2 T(n/4) + n/2 ] + n

T(n) = 4 \* T(n/4) + 2n

T(n) = 4 \*[2 T(n/8) + n/8 ] + 2n

T(n) = 8 \* T(n/4) + 3n

T(n) = 2^k \* T(n/k) + kn

[As T(1) = 1, n/2^k = 1 --> n = 2^k --> k = logn]

T(n) = n \* T(1) + n logn

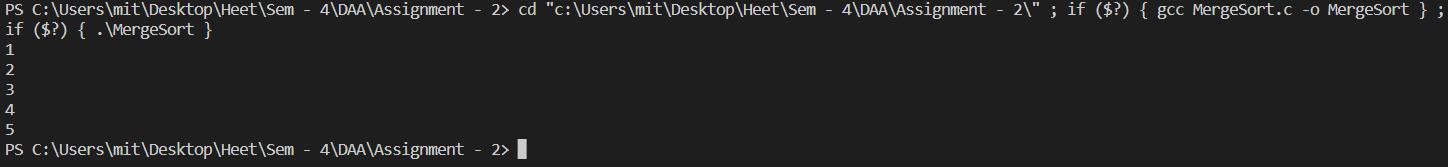
T(n) = n + n logn

T(n) = O(n logn)

Time Complexity of Merge Sort - O(n logn).

Time Compleity of MergeSort Function - O(n).

* Output:



1. Quick Sort:

|  |  |  |
| --- | --- | --- |
| Code | Frequency  (for best case) | Cost |
| #include<stdio.h>  void swap(int \*first, int \*second)  {  int temp = \*first;  \*first = \*second;  \*second = temp;  }  int Partition(int arr[], int lb, int ub)  {  int pivot = arr[ub];  int i = lb-1;  for(int j = lb; j<=ub; j++)  {  if(arr[j]<pivot)  {  i++;  swap(&arr[i], &arr[j]);  }  }  swap(&arr[i+1], &arr[ub]);  return i+1;  }  void QuickSort(int arr[], int lb, int ub)  {  if(lb<ub)  {  int p = Partition(arr, lb, ub);  QuickSort(arr, lb, p-1);  QuickSort(arr, p+1, ub);  }  }  void main()  {  int arr[] = {15, 5, 24, 8, 1, 3, 16, 10, 20};  int n = sizeof(arr)/sizeof(arr[0]);  printf("Initial Array: \n");  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  QuickSort(arr, 0, n-1);  printf("\nSorted Array: \n");  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  } | 1  1  1  1  1  n+1  n  n  n  n  n  n  n/2  n/2 | C1  C2  C3  C4  C5  C6  C7  C8  C9  C10  C11  C12  C13  C14 |

**Time Complexity:**

1. Best Case and Average Case:

T(n) = T(n/2) + T(n/2) + n

T(n) = 2 \* T(n/2) + n

T(n) = 2 \*[2 T(n/4) + n/2 ] + n

T(n) = 4 \* T(n/4) + 2n

T(n) = 4 \*[2 T(n/8) + n/4 ] + 2n

T(n) = 8 \* T(n/8) + 3n

T(n) = 2^k \* T(n/k) + kn

T(n) = n \* T(1) + n logn

T(n) = n + n logn

T(n) = O(n logn)

1. Worst Case:

T(n) = T(0) + T(n-1) + n

T(n) = T(n-1) + n

T(n) = T(n-2) + T(n-1) + n

T(n) = T(n-3) + T(n-2)+ T(n-1) + n

T(n) = T(n-k) + T(n-(k-1))+….+ T(n-1) + n

T(n) = T(1) + T(2) + T(3) + …n

T(n) = 1 + 2 + 3 + 4…n

T(n) = n(n+1)/2

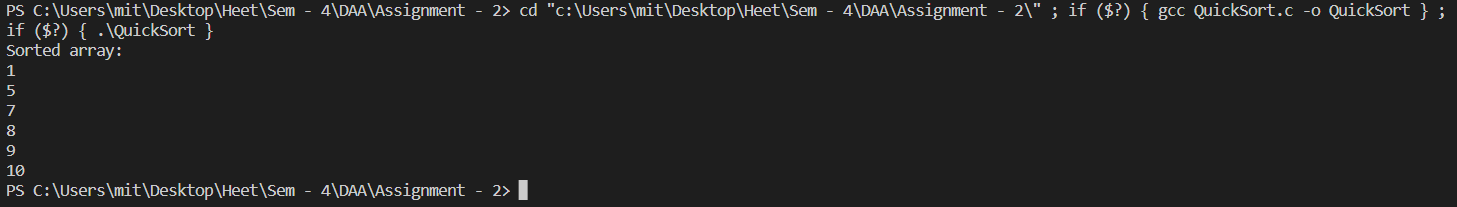
T(n) = O(n2)

Time Complexity of Quick Sort (for best and average case) = O(n logn).

Time Complexity of Quick Sort (for worst case) = O(n2).

Time Complexity of Partition Function = O(n).

* Output:



**Practical No. 3**

Use singly linked lists to implement integers of unlimited size. Each node of the list should store one digit of the integer. You should implement addition, subtraction, multiplication, and exponentiation operations. Limit exponents to be positive integers.

What is the asymptotic running time for each of your operations, expressed in terms of the number of digits for the two operands of each function?

* **Addition and Substraction -**
* Code:

#include<stdio.h>

#include<stdlib.h>

struct Node

{

int data;

struct Node\* next;

};

struct Node\* Create(int number)

{

struct Node\* head = NULL;

struct Node\* tail = NULL;

int quotient=1, remainder;

while(quotient != 0)

{

remainder = number % 10;

quotient = number / 10;

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = remainder;

newNode->next = NULL;

number = quotient;

if (head == NULL)

{

head = newNode;

tail = newNode;

}

else

{

tail->next = newNode;

tail = newNode;

}

}

return head;

}

void PrintReverse(struct Node\* head)

{

if (head == NULL)

return;

PrintReverse(head->next);

printf("%d", head->data);

}

void Print(struct Node\* head)

{

struct Node\* current = head;

while(current!=NULL)

{

printf("%d -> ", current->data);

current = current ->next;

}

printf("NULL\n");

}

void Subtract(struct Node\* head1, struct Node\* head2)

{

struct Node\* current1 = head1;

struct Node\* current2 = head2;

struct Node\* head = NULL;

struct Node\* tail = NULL;

int borrow = 0;

while (current1 != NULL || current2 != NULL)

{

int value1 = (current1 != NULL) ? current1->data : 0;

int value2 = (current2 != NULL) ? current2->data : 0;

value1 = value1 - borrow; // Subtract the borrow from the first number

borrow = 0; // Reset borrow

if (value1 < value2)

{

value1 = value1 + 10; // Borrow from the next higher digit

borrow = 1; // Set borrow flag

}

int digit = value1 - value2;

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = digit;

newNode->next = NULL;

if (head == NULL)

{

head = newNode;

tail = newNode;

}

else

{

tail->next = newNode;

tail = newNode;

}

if (current1 != NULL) current1 = current1->next;

if (current2 != NULL) current2 = current2->next;

}

Print(head);

PrintReverse(head);

printf("\n");

}

void Sum(struct Node\* head1, struct Node\* head2)

{

struct Node\* current1 = head1;

struct Node\* current2 = head2;

struct Node\* head = NULL;

struct Node\* tail = NULL;

int carry = 0;

while (current1 != NULL || current2 != NULL || carry != 0)

{

int value1 = (current1 != NULL) ? current1->data : 0;

int value2 = (current2 != NULL) ? current2->data : 0;

int total = value1 + value2 + carry;

carry = total / 10;

int digit = total % 10;

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = digit;

newNode->next = NULL;

if (head == NULL)

{

head = newNode;

tail = newNode;

}

else

{

tail->next = newNode;

tail = newNode;

}

if (current1 != NULL)

current1 = current1->next;

if (current2 != NULL)

current2 = current2->next;

}

Print(head);

PrintReverse(head);

printf("\n");

}

void main()

{

int num1, num2;

printf("Enter the first number: ");

scanf("%d", &num1);

struct Node\* head1 = Create(num1);

Print(head1);

printf("Enter the second number: ");

scanf("%d", &num2);

struct Node\* head2 = Create(num2);

Print(head2);

printf("Sum of above two linked list is: \n");

Sum(head1, head2);

if(num1 > num2)

{

printf("Substraction of above two linked list is: \n");

Subtract(head1, head2);

}

else

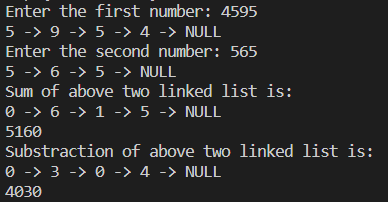
{

printf("Substraction of greater number from smaller one is not possible! \n");

}

}

* Output:



* Time Complexity:

For Create Funtion - O(logn) as it is divided by 10 everytime (i.e. logarithmic).

For Print and PrintReverse Funtion - O(n) as it is traversed to the end of linked list.

For Sum and Subtract Function - O(max(num1, num2)) as it will be iterated till the length of bigger number.

Therefore, the overall time complexity is **O(max(num1, num2))**.

* **Multiplication -**
* Code:

#include<stdio.h>

#include<stdlib.h>

struct Node

{

int data;

struct Node\* next;

};

struct Node\* Create(int number)

{

struct Node\* head = NULL;

struct Node\* tail = NULL;

int dividend = 10, quotient=1, remainder;

while(quotient != 0)

{

remainder = number % 10;

quotient = number / 10;

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = remainder;

newNode->next = NULL;

number = quotient;

if (head == NULL)

{

head = newNode;

tail = newNode;

}

else

{

tail->next = newNode;

tail = newNode;

}

}

return head;

}

void Print(struct Node\* head)

{

struct Node\* current = head;

while(current!=NULL)

{

printf("%d -> ", current->data);

current = current ->next;

}

printf("NULL\n");

}

struct Node\* Reverse(struct Node\* head)

{

struct Node\* prev = NULL;

struct Node\* current = head;

struct Node\* nextNode = NULL;

while (current != NULL) {

nextNode = current->next;

current->next = prev;

prev = current;

current = nextNode;

}

return prev;

}

struct Node\* Multiply(struct Node\* num1, struct Node\* num2)

{

struct Node\* result = Create(0);

struct Node\* result\_head = result;

struct Node\* list1 = num1;

int i = 0;

while (list1 != NULL)

{

struct Node\* temp = result;

int carry = 0;

int j = 0;

struct Node\* list2 = num2;

while (j < i)

{

if (temp->next == NULL)

{

temp->next = Create(0);

}

temp = temp->next;

j++;

}

while (list2 != NULL)

{

int product = (temp->data) + carry + (list1->data) \* (list2->data);

carry = product / 10;

temp->data = product % 10;

if (temp->next == NULL && (list2->next != NULL || carry != 0))

{

temp->next = Create(0);

}

temp = temp->next;

list2 = list2->next;

}

// If there's still a carry after multiplying with all digits of list2

if (carry != 0)

{

temp->data = carry;

}

list1 = list1->next;

i++;

}

return Reverse(result\_head);

}

int main()

{

int num1, num2;

printf("Enter first number: ");

scanf("%d", &num1);

printf("Enter second number: ");

scanf("%d", &num2);

printf("Number1: \n");

struct Node\* list1 = Create(num1);

Print(list1);

printf("Number2: \n");

struct Node\* list2 = Create(num2);

Print(list2);

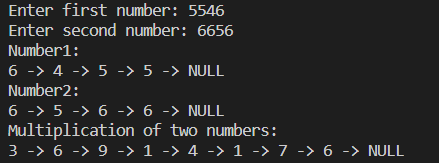
printf("Multiplication of two numbers: \n");

Print(Multiply(list1, list2));

return 0;

}

* Output:



* Time Complexity:

For Create Funtion - O(logn) as it is divided by 10 everytime (i.e. logarithmic).

For Print and PrintReverse Funtion - O(n) as it is traversed to the end of linked list.

For Multiply Function - O(m\*n) where m is no. of digits in num1 and n is no. of digits in num2.

**Practical No. 4**

Implement a city database using unordered lists. Each database record contains the name of the city (a string of arbitrary length) and the coordinates of the city expressed as integer x and y coordinates. Your program should allow following functionalities:

1. Insert a record,
2. Delete a record by name or coordinate,
3. Search a record by name or coordinate.
4. Print all records a given distance of a specified point.

Implement the database using an array-based list implementation, and then a linked list implementation. Perform following analysis:

1. Collect running time statistics for each operation in both implementations.
2. What are your conclusions about the relative advantages and disadvantages of the

two within implementations?

1. Would storing records on the list in alphabetical order by city name speed any of the operations?
2. Would keeping the list in alphabetical order slow any of the operations?

* **For Linked List Implementation:**
* Code:

# include <stdio.h>

# include <stdlib.h>

# include <string.h>

# include <math.h>

struct Node

{

char city[40];

int x;

int y;

struct Node\* next;

};

void Insert(struct Node\*\* head, char city[40], int x, int y)

{

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

strcpy(newNode->city, city);

newNode->x = x;

newNode->y = y;

newNode->next = NULL;

if (\*head == NULL)

{

\*head = newNode;

return;

}

struct Node \*current = \*head;

while (current->next != NULL)

{

current = current->next;

}

current->next = newNode;

}

void SearchCity(struct Node\* head, char city[40])

{

struct Node\* current = head;

while(current != NULL)

{

if (strcmp(current->city,city) == 0)

{

printf("%s city is present in Database!\n", current->city);

break;

}

current = current->next;

}

if (current == NULL)

{

printf("City not found in the Database!\n");

}

}

void SearchCoordinate(struct Node\* head, int x, int y)

{

struct Node\* current = head;

while(current != NULL)

{

if (current->x == x && current->y == y)

{

printf("%s city is present in Database!\n", current->city);

break;

}

current = current->next;

}

if (current == NULL)

{

printf("City not found in the Database!\n");

}

}

void DeleteCity(struct Node\* head, char city[40])

{

struct Node\* current = head;

while (current != NULL)

{

if(strcmp((current->next)->city,city) == 0)

{

current->next = (current->next)->next;

break;

}

current = current->next;

}

if (current == NULL)

{

printf("City not found in the Database!\n");

}

}

void DeleteCoordinate(struct Node\* head, int x, int y)

{

struct Node\* current = head;

while (current != NULL)

{

if((current->next)->x == x && (current->next)->y == y)

{

current->next = (current->next)->next;

break;

}

current = current->next;

}

if (current == NULL)

{

printf("City not found in the Database!\n");

}

}

void Range(struct Node\* head, int x, int y, float distance)

{

struct Node\* current = head;

while (current != NULL)

{

float dist = sqrt(pow((current->x - x),2)+pow((current->y - y),2));

if (dist <= distance)

{

printf("%s is present in this range.\n", current->city);

}

current = current->next;

}

printf("Execution Completed!/n");

}

void Print(struct Node\* head)

{

struct Node\* current = head;

while(current!=NULL)

{

printf("[%s|%d|%d] -> ", current->city, current->x, current->y);

current = current ->next;

}

printf("NULL\n");

}

void main()

{

int n;

struct Node\* head = NULL;

printf("Enter the number of entries you want to enter: ");

scanf("%d", &n);

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

{

char city[40];

int x,y;

printf("Enter the city: ");

scanf("%s",city);

printf("Enter the x-coordinate: ");

scanf("%d",&x);

printf("Enter the y-coordinate: ");

scanf("%d",&y);

Insert(&head, city, x, y);

}

printf("The Database is: \n");

Print(head);

char name1[40];

int X1,Y1;

int choice;

printf("Enter: \n1 if you want to search city with name\n2 is you want to search city with coordinates\n");

scanf("%d",&choice);

switch (choice)

{

case 1:

printf("Enter the city name you want to search: ");

scanf("%s", &name1);

SearchCity(head, name1);

break;

case 2:

printf("Enter x coordinate of the city you want to search: ");

scanf("%d", &X1);

printf("Enter y coordinate of the city you want to search: ");

scanf("%d", &Y1);

SearchCoordinate(head, X1, Y1);

break;

default:

printf("Not valid!!");

break;

}

char name2[40];

int X2,Y2;

int choice2;

printf("Enter: \n1 if you want to delete city using name\n2 is you want to delete city using coordinates\n");

scanf("%d",&choice2);

switch (choice2)

{

case 1:

printf("Enter the city name you want to delete: ");

scanf("%s", &name2);

DeleteCity(head, name2);

Print(head);

break;

case 2:

printf("Enter x coordinate of the city you want to delete: ");

scanf("%d", &X2);

printf("Enter y coordinate of the city you want to delete: ");

scanf("%d", &Y2);

DeleteCoordinate(head, X2, Y2);

Print(head);

break;

default:

printf("Not valid!!");

break;

}

float distance;

int X3,Y3;

printf("For searching cities in the particular range: \n");

printf("Enter x coordinate of the point: ");

scanf("%d", &X3);

printf("Enter y coordinate of the point: ");

scanf("%d", &Y3);

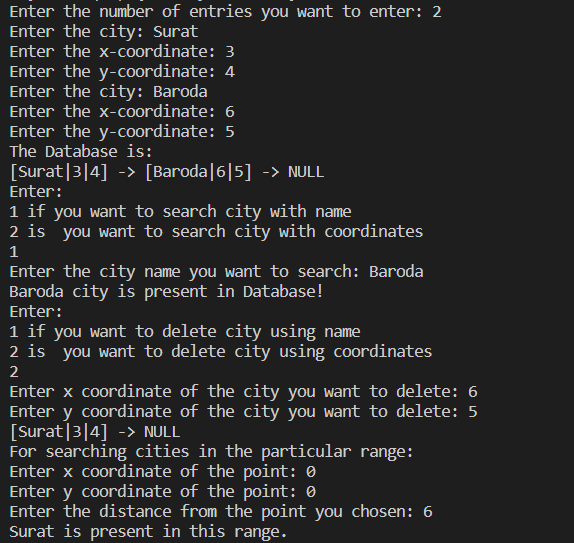
printf("Enter the distance from the point you chosen: ");

scanf("%f", &distance);

Range(head, X3, Y3, distance);

}

* Output:



* **For Array with Structure Implementation:**
* Code:

# include <stdio.h>

# include <stdlib.h>

# include <string.h>

# include <math.h>

struct Data

{

char city[40];

int x;

int y;

};

void Print(struct Data data[], int n)

{

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

{

printf("City = %s, x = %d, y = %d\n", data[i].city, data[i].x, data[i].y);

}

}

void SearchName(struct Data data[], int num, char name[])

{

int i=0;

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

{

if(strcmp(data[i].city, name) == 0)

{

printf("%s city is present in Database!\n", data[i].city);

break;

}

}

if (i == num)

{

printf("City not found in the Database!\n");

}

}

void SearchCoordinate(struct Data data[], int num, int x, int y)

{

int i=0;

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

{

if(data[i].x == x && data[i].y == y)

{

printf("%s city is present in Database!\n", data[i].city);

break;

}

}

if (i == num)

{

printf("City not found in the Database!\n");

}

}

void DeleteName(struct Data data[], int \*num, char name[])

{

for(int i=0; i<\*num; i++)

{

if(strcmp(data[i].city, name) == 0)

{

for(int j=i; j<\*num-1 ; j++)

{

data[j] = data[j+1];

}

(\*num)--;

printf("Entry Deletion Successful!\n");

return;

}

}

printf("Entry not found");

}

void DeleteCoordinate(struct Data data[], int \*num, int x, int y)

{

for(int i=0; i<\*num; i++)

{

if(data[i].x == x && data[i].y == y)

{

for(int j=i; j<\*num ; j++)

{

data[j] = data[j+1];

}

(\*num)--;

printf("Entry Deletion Successful!\n");

return;

}

}

printf("Entry not found");

}

void Range(struct Data data[], int num, int x, int y, int distance)

{

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

{

int dist = sqrt(pow((data[i].x - x), 2) + pow((data[i].y - y), 2));

if(dist <= distance)

{

printf("%s is present in this range.\n", data[i].city);

}

}

}

void main()

{

struct Data data[100];

int num;

char searchcity[100];

int searchx, searchy;

printf("Enter the total number of entries you want to enter: ");

scanf("%d", &num);

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

{

printf("Enter the name of the city: ");

scanf("%s", data[i].city);

printf("Enter the x coordinate of the city: ");

scanf("%d", &data[i].x);

printf("Enter the y coordinate of the city: ");

scanf("%d", &data[i].y);

}

Print(data, num);

int choice;

printf("Enter: \n1 if you want to search city with name\n2 is you want to search city with coordinates\n");

scanf("%d",&choice);

switch (choice)

{

case 1:

printf("Enter the name of the city to be searched: ");

scanf("%s", searchcity);

SearchName(data, num, searchcity);

break;

case 2:

printf("Enter x coordinate of the city to be searched: ");

scanf("%d", &searchx);

printf("Enter y oordinate of the city to be searched: ");

scanf("%d", &searchy);

SearchCoordinate(data, num, searchx, searchy);

break;

default:

printf("Not valid!!");

break;

}

char name[100];

int X, Y;

int choice2;

printf("Enter: \n1 if you want to delete city using name\n2 is you want to delete city using coordinates\n");

scanf("%d",&choice2);

switch (choice2)

{

case 1:

printf("Enter the name of the city to be deleted: ");

scanf("%s", name);

DeleteName(data, &num, name);

Print(data, num);

break;

case 2:

printf("Enter the x coordinate of the city to be deleted: ");

scanf("%d", &X);

printf("Enter the y coordinate of the city to be deleted: ");

scanf("%d", &Y);

DeleteCoordinate(data, &num, X, Y);

Print(data, num);

break;

default:

printf("Not valid!!");

break;

}

int X2, Y2, distance;

printf("Enter the x coordinate of the point: ");

scanf("%d", &X2);

printf("Enter the y coordinate of the point: ");

scanf("%d", &Y2);

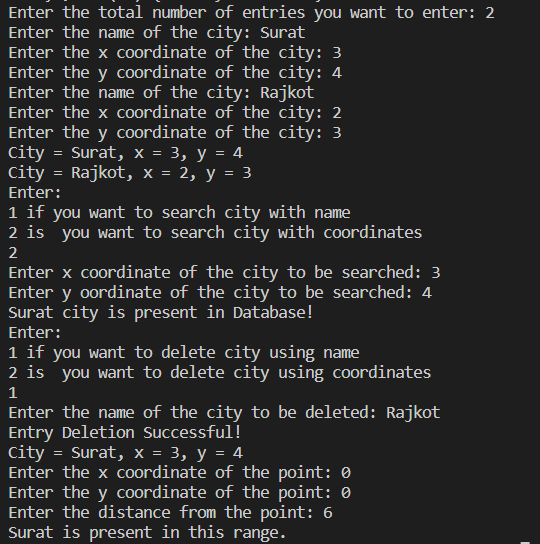
printf("Enter the distance from the point: ");

scanf("%d", &distance);

Range(data, num, X2, Y2, distance);

}

* Output:



Q1) Collect running time statistics for each operation in both implementations.

* Array-Based List:

All the function i.e. Insert, SearchCity, SearchCoordinate, DeleteCity, DeleteCoordinate, Range, Print has the worst case time complexity of O(n) where n is the no. of elements in the array of structure.

* Linked List:

All the function i.e. Insert, SearchCity, SearchCoordinate, DeleteCity, DeleteCoordinate, Range, Print has the worst case time complexity of O(n) where n is the no. of elements in the linked list.

Q2) What are your conclusions about the relative advantages and disadvantages of the two within implementations?

* Array-Based List:
* Advantages:

1. Constant-time access to elements by index.
2. Compact memory representation.

* Disadvantages:

1. Costly resizing operations when the array is full.
2. Expensive deletion operations due to shifting elements.

* Linked List:
* Advantages:

1. Efficient insertion and deletion operations, especially when working with large datasets.
2. Dynamic memory allocation.

* Disadvantages:

1. Poor cache performance due to scattered memory allocation.
2. Inefficient random access; elements can only be accessed sequentially.

Q3) Would storing records on the list in alphabetical order by city name speed any of the operations?

Ans.

Speeding up Operations: Storing records in alphabetical order by city name could speed up search operations for a given city name, as it could potentially reduce the search space by using techniques like binary search.

Q4) Would keeping the list in alphabetical order slow any of the operations

Ans.

Insertion and deletion operations may become slower as they would require maintaining the sorted order, resulting in additional overhead.

**Practical No. 5 (1)**

Implement interval scheduling algorithm. Given events with their starting and ending times, find a schedule that includes as many events as possible. It is not possible to select an event partially. For example, consider the following example:

|  |  |  |
| --- | --- | --- |
| **Event** | **Starting time** | **Ending time** |
| A | 1 | 3 |
| B | 2 | 5 |
| C | 3 | 9 |
| D | 6 | 8 |

Here, maximum number of events that can be scheduled is 2. We can schedule B and D together.

* Code:

# include <stdio.h>

# include <stdlib.h>

# include <string.h>

struct Event

{

char name[40];

int stime;

int ftime;

};

void Print(struct Event event[], int n)

{

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

{

printf("Process = %s | Start Time = %d | End Time = %d\n", event[i].name, event[i].stime, event[i].ftime);

}

}

void Sort(struct Event event[], int len)

{

struct Event temp;

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

{

for (int j=i+1; j<len; j++)

{

if (event[i].ftime > event[j].ftime)

{

temp = event[i];

event[i] = event[j];

event[j] = temp;

}

}

}

Print(event, len);

}

void Selection(struct Event event[], int len)

{

    int lastSelectedFinishTime = -1;

    for (int i = 0; i < len; i++)

    {

        if (event[i].stime >= lastSelectedFinishTime)

        {

printf("Process = %s | Start Time = %d | End Time = %d\n", event[i].name, event[i].stime, event[i].ftime);

            lastSelectedFinishTime = event[i].ftime;

        }

    }

}

void main()

{

struct Event event[100];

struct Event sortedEvent[100];

int num;

printf("Enter the total number of process you want to enter: ");

scanf("%d", &num);

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

{

printf("Enter the name of the process: ");

scanf("%s", event[i].name);

printf("Enter the start time of process: ");

scanf("%d", &event[i].stime);

printf("Enter the end time of process: ");

scanf("%d", &event[i].ftime);

}

printf("List of all events: \n");

Print(event, num);

printf("Sorted events according to end time: \n");

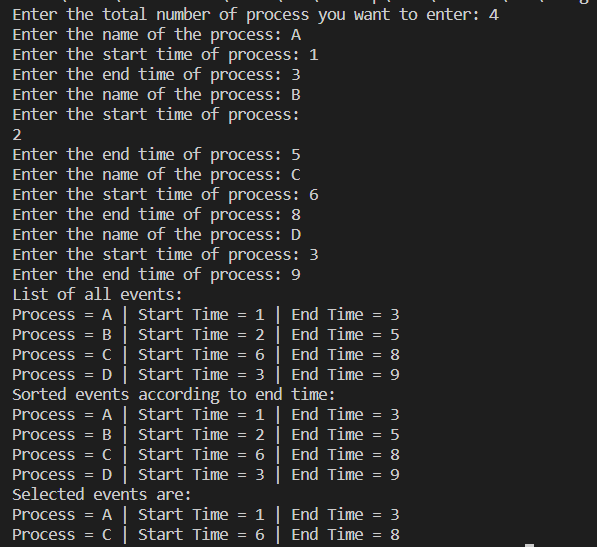
Sort(event, num);

printf("Selected events are: \n");

Selection(event, num);

}

* Output:



* Time Complexity:

For Insertion of events, Print function and Selection function - O(n) where n is number of events.

For Sort Function - O(n^2) as nested for loops are used in it.

Overall Time Complexity - O(n^2).

**Practical No. 5 (2)**

Write a code for finding Minimum Spanning Tree using Union-Find method.

* Code:

#include <stdio.h>

#include <stdlib.h>

// Structure to represent an edge

struct Edge

{

int src, dest, weight;

};

// Structure to represent a subset for Union-Find

struct Subset

{

int parent;

};

// Function to find the parent of a vertex using Union-Find

int Find(struct Subset subsets[], int i)

{

if (subsets[i].parent != i)

subsets[i].parent = Find(subsets, subsets[i].parent);

return subsets[i].parent;

}

// Function to perform union of two subsets

void Union(struct Subset subsets[], int x, int y) {

int xroot = Find(subsets, x);

int yroot = Find(subsets, y);

subsets[xroot].parent = yroot;

}

// Kruskal's algorithm to find MST

void Kruskal(struct Edge edges[], int V, int E) {

struct Edge result[V]; // Stores the MST

struct Subset subsets[V];

for (int v = 0; v < V; ++v)

{subsets[v].parent = v;}

// Sort edges in ascending order of weight

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

for (int j = 0; j < E - i - 1; ++j) {

if (edges[j].weight > edges[j + 1].weight) {

struct Edge temp = edges[j];

edges[j] = edges[j + 1];

edges[j + 1] = temp;

}

}

}

int e = 0;

int i = 0;

while (e < V - 1 && i < E) {

struct Edge nextEdge = edges[i++];

int x = Find(subsets, nextEdge.src);

int y = Find(subsets, nextEdge.dest);

if (x != y) {

result[e++] = nextEdge;

Union(subsets, x, y);

}

}

// Print the MST

printf("Edges in the Minimum Spanning Tree:\n");

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

printf("%d - %d : %d\n", result[j].src, result[j].dest, result[j].weight);

}

void main()

{

int V = 4;

int E = 5;

struct Edge edges[] =

{

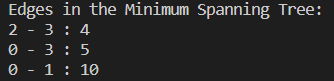
{0, 1, 10}, {0, 2, 6}, {0, 3, 5}, {1, 3, 15}, {2, 3, 4}

};

Kruskal(edges, V, E);

}

* Output:



* Time Complexity:

Time Complexity for sorting the edges is O(E^2).

Time Complexity for both Union and Find is O(log V) for one edge and as it is done for the E no. of edges so the complexity will be O(E logV).

Overall Complexity - O(E^2).

**Practical No. 6 [Divide and Conquer]**

Implement both a standard matrix multiplication algorithm and Strassen’s matrix multiplication algorithm. Using empirical testing, try and estimate the constant factors for the runtime equations of the two algorithms. How big must be before Strassen’s algorithm becomes more efficient than the standard algorithm?

* **Standard Matrix Multiplication -**
* Code:

#include <stdio.h>

void multiply(int n, int A[][n], int B[][n], int C[][n])

{

// Base case: If matrices are 1x1, perform single multiplication

if (n == 1)

{

C[0][0] = A[0][0] \* B[0][0];

return;

}

// Divide matrices into sub-matrices

int half = n / 2;

int a11[half][half], a12[half][half], a21[half][half], a22[half][half];

int b11[half][half], b12[half][half], b21[half][half], b22[half][half];

int c1111[half][half], c1221[half][half], c1112[half][half], c1222[half][half], c2111[half][half], c2221[half][half], c2112[half][half], c2222[half][half];

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

{

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

{

a11[i][j] = A[i][j];

a12[i][j] = A[i][j + half];

a21[i][j] = A[i + half][j];

a22[i][j] = A[i + half][j + half];

b11[i][j] = B[i][j];

b12[i][j] = B[i][j + half];

b21[i][j] = B[i + half][j];

b22[i][j] = B[i + half][j + half];

}

}

// Recursive calls for sub-matrices

multiply(half, a11, b11, c1111);

multiply(half, a12, b21, c1221);

multiply(half, a11, b12, c1112);

multiply(half, a12, b22, c1222);

multiply(half, a21, b11, c2111);

multiply(half, a22, b21, c2221);

multiply(half, a21, b12, c2112);

multiply(half, a22, b22, c2222);

// Combine results back into the original matrix

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

for (int j = 0; j < half; j++) {

C[i][j] = c1111[i][j] + c1221[i][j];

C[i][j + half] = c1112[i][j] + c1222[i][j];

C[i + half][j] = c2111[i][j] + c2221[i][j];

C[i + half][j + half] = c2112[i][j] + c2222[i][j];

}

}

}

void printMatrix(int n, int matrix[][n])

{

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

{

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

{

printf("%d ", matrix[i][j]);

}

printf("\n");

}

}

int main()

{

int n;

printf("Enter the size of square matrices (nxn): ");

scanf("%d", &n);

int A[n][n], B[n][n], C[n][n];

printf("Enter elements for matrix A:\n");

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

{

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

{

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

}

}

printf("Enter elements for matrix B:\n");

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

{

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

{

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

}

}

multiply(n, A, B, C);

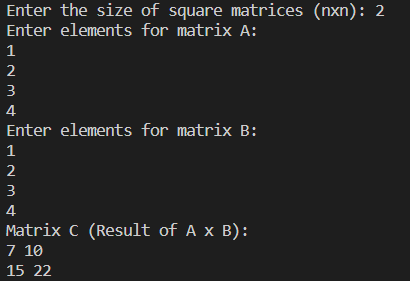
printf("Matrix C (Result of A x B):\n");

printMatrix(n, C);

return 0;

}

* Output:



* **Recursive Equation:**
* **Time Complexity:** by using master’s theorem (Case-1)
* **Strassen’s Matrix Multiplication -**
* Code:

#include <stdio.h>

// Function to add two matrices

void add(int n, int A[][n], int B[][n], int C[][n])

{

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

{

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

{

C[i][j] = A[i][j] + B[i][j];

}

}

}

// Function to subtract two matrices

void subtract(int n, int A[][n], int B[][n], int C[][n])

{

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

{

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

{

C[i][j] = A[i][j] - B[i][j];

}

}

}

// Function to multiply two matrices using Strassen algorithm

void strassen(int n, int A[][n], int B[][n], int C[][n])

{

if (n == 1)

{

C[0][0] = A[0][0] \* B[0][0];

return;

}

int newSize = n / 2;

int A11[newSize][newSize], A12[newSize][newSize], A21[newSize][newSize], A22[newSize][newSize];

int B11[newSize][newSize], B12[newSize][newSize], B21[newSize][newSize], B22[newSize][newSize];

int C11[newSize][newSize], C12[newSize][newSize], C21[newSize][newSize], C22[newSize][newSize];

int P[newSize][newSize], Q[newSize][newSize], R[newSize][newSize], S[newSize][newSize], T[newSize][newSize], U[newSize][newSize], V[newSize][newSize];

int temp1[newSize][newSize], temp2[newSize][newSize];

// Divide matrices A and B into submatrices

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

{

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

{

A11[i][j] = A[i][j];

A12[i][j] = A[i][j + newSize];

A21[i][j] = A[i + newSize][j];

A22[i][j] = A[i + newSize][j + newSize];

B11[i][j] = B[i][j];

B12[i][j] = B[i][j + newSize];

B21[i][j] = B[i + newSize][j];

B22[i][j] = B[i + newSize][j + newSize];

}

}

// Compute P = (A11 + A22) \* (B11 + B22)

add(newSize, A11, A22, temp1);

add(newSize, B11, B22, temp2);

strassen(newSize, temp1, temp2, P);

// Compute Q = (A21 + A22) \* B11

add(newSize, A21, A22, temp1);

strassen(newSize, temp1, B11, Q);

// Compute R = A11 \* (B12 - B22)

subtract(newSize, B12, B22, temp1);

strassen(newSize, A11, temp1, R);

// Compute S = A22 \* (B21 - B11)

subtract(newSize, B21, B11, temp1);

strassen(newSize, A22, temp1, S);

// Compute T = (A11 + A12) \* B22

add(newSize, A11, A12, temp1);

strassen(newSize, temp1, B22, T);

// Compute U = (A21 - A11) \* (B11 + B12)

subtract(newSize, A21, A11, temp1);

add(newSize, B11, B12, temp2);

strassen(newSize, temp1, temp2, U);

// Compute V = (A12 - A22) \* (B21 + B22)

subtract(newSize, A12, A22, temp1);

add(newSize, B21, B22, temp2);

strassen(newSize, temp1, temp2, V);

// Compute C11 = P + S - T + V

add(newSize, P, S, temp1);

subtract(newSize, temp1, T, temp2);

add(newSize, temp2, V, C11);

// Compute C12 = R + T

add(newSize, R, T, C12);

// Compute C21 = Q + S

add(newSize, Q, S, C21);

// Compute C22 = P - Q + R + U

subtract(newSize, P, Q, temp1);

add(newSize, temp1, R, temp2);

add(newSize, temp2, U, C22);

// Combine submatrices to form matrix C

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

{

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

{

C[i][j] = C11[i][j];

C[i][j + newSize] = C12[i][j];

C[i + newSize][j] = C21[i][j];

C[i + newSize][j + newSize] = C22[i][j];

}

}

}

// Function to print a matrix

void printMatrix(int n, int matrix[][n])

{

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

{

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

{

printf("%d ", matrix[i][j]);

}

printf("\n");

}

}

int main()

{

int n;

printf("Enter the size of square matrices (nxn): ");

scanf("%d", &n);

int A[n][n], B[n][n], C[n][n];

printf("Enter elements for matrix A:\n");

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

{

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

{

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

}

}

printf("Enter elements for matrix B:\n");

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

{

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

{

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

}

}

strassen(n, A, B, C);

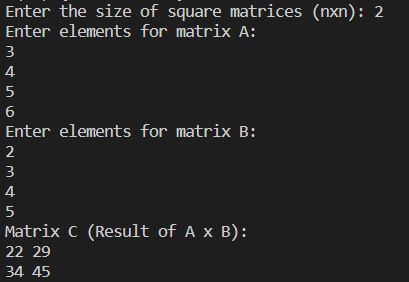
printf("Matrix C (Result of A x B):\n");

printMatrix(n, C);

return 0;

}

* Output:



* **Recursive Equation:**
* **Time Complexity:** by using master’s theorem (Case-1)
* **Empirical Testing:**

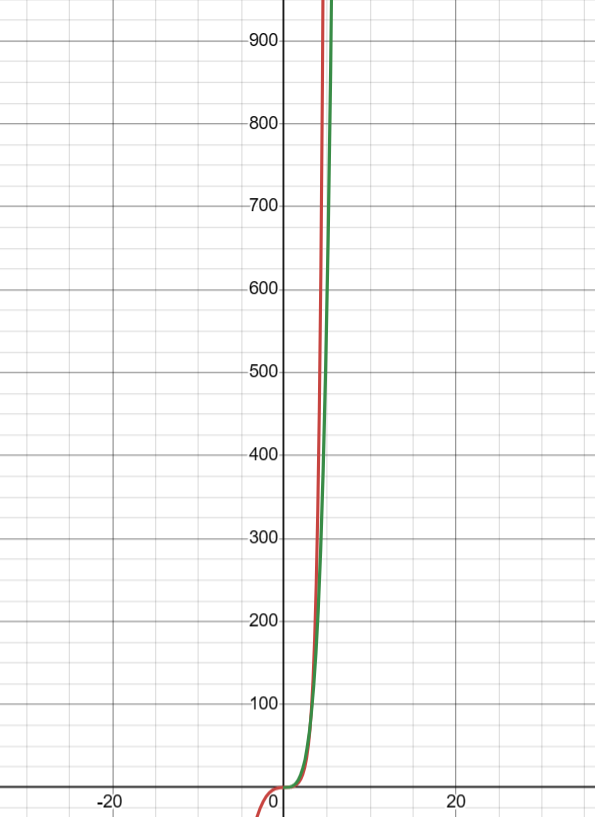
This is the equation that is obtained from the recurrence relation of Recursive Method:

Screenshot (380)

This is the equation of Strassen Method:

Screenshot (381)

By plotting the graph of g(n) vs n of these two equations we get a graph like this:



Here green line represents the curve for strassen multiplication and the red line represents curve for recursive multiplication.

The slope of time execution curve for each algorithm represents the constant factors associated with their runtime equations.

Strassen’s Algorithm may have a lower slope initially due to its lower complexity, but its slope then increases at larger matrix sizes due to its higher overhead.

Recursive Method’s curve is increasing faster because of its higher complexity.

Thus, we can see that at values of n greater than 100 i.e. matrix size greater than 100 the Strassen’s Algorithm is more efficient than standard algorithm.

**Experiment No. 7 [Dynamic Programming]**

Implement the Floyd Warshall Algorithm for All Pair Shortest Path Problem. You are given a weighted diagraph , with arbitrary edge weights or costs between any node and node . Find the cheapest path from every node to every other node. Edges may have negative weights. Consider the following test case to check your algorithm:

|  |  |  |
| --- | --- | --- |
| ***v*** | ***w*** |  |
| 0 | 1 | -1 |
| 0 | 2 | 4 |
| 1 | 2 | 3 |
| 1 | 3 | 2 |
| 1 | 4 | 2 |
| 3 | 2 | 5 |
| 3 | 1 | 1 |
| 4 | 3 | -3 |

* Code:

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define INF INT\_MAX

int n;

void fillDistanceMatrix(int A[n][n], int D[n][n])

{

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

{

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

{

if (i == j)

D[i][j] = 0;

else if (A[i][j] == 0)

D[i][j] = INF;

else

D[i][j] = A[i][j];

}

}

}

void FloydWarshall(int A[n][n], int D[n][n])

{

fillDistanceMatrix(A, D);

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

{

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

{

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

{

if (D[i][k] < INF && D[k][j] < INF)

{

if((D[i][k] + D[k][j]) < D[i][j])

{

D[i][j] = D[i][k] + D[k][j];

}

}

}

}

}

}

int main()

{

printf("Enter the size of array (nxn): ");

scanf("%d", &n);

int A[n][n], D[n][n];

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

{

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

{

printf("Enter element A[%d][%d] : ", i, j);

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

}

}

FloydWarshall(A, D);

printf("Adjacency Matrix is: \n");

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

{

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

{

if (D[i][j] == INF)

printf("%7s", "INF");

else

printf("%7d", D[i][j]);

}

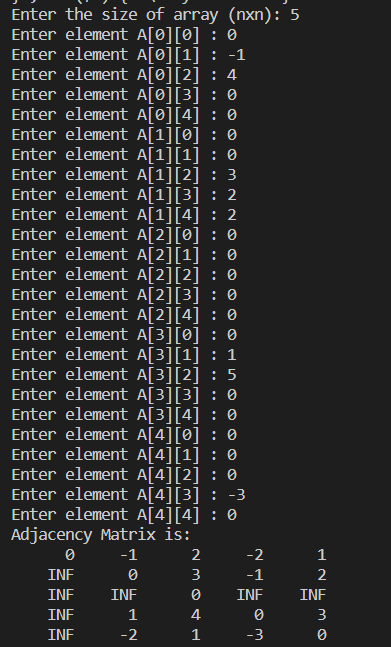
printf("\n");

}

return 0;

}

* Output:



* Time Complexity:

For the fillDistanceMatrix function, time complexity is O(n2) due to two nested for loops.

For the FloydWarshall function, time complexity is O(n3) due to three nested for loops.

Overall Complexity - O(n3)

**Practical No. 8 [Backtracking]**

Solve the queens’ problem using backtracking. Here, the task is to place chess queens on an x board so that no two queens attack each other. For example, following is a solution for the 4 Queen’ problem.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Q |  |  |
|  |  |  | Q |
| Q |  |  |  |
|  |  | Q |  |

* Code:

#include <stdbool.h>

#include <stdio.h>

int N;

// A utility function to print solution

void printSolution(int board[N][N])

{

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

for (int j = 0; j < N; j++) {

if(board[i][j] == 1)

printf("Q ");

else

printf(". ");

}

printf("\n");

}

}

// A utility function to check if a queen can

// be placed on board[row][col]. Note that this

// function is called when "col" queens are

// already placed in columns from 0 to col -1.

// So we need to check only left side for

// attacking queens

bool isSafe(int board[N][N], int row, int col)

{

int i, j;

// Check this row on left side

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

if (board[row][i] == 1)

return false;

// Check upper diagonal on left side

for (i = row, j = col; i >= 0 && j >= 0; i--, j--)

if (board[i][j] == 1)

return false;

// Check lower diagonal on left side

for (i = row, j = col; j >= 0 && i < N; i++, j--)

if (board[i][j] == 1)

return false;

return true;

}

// A recursive utility function to solve N

// Queen problem

bool solveNQUtil(int board[N][N], int col)

{

// Base case: If all queens are placed

// then return true

if (col >= N)

return true;

// Consider this column and try placing

// this queen in all rows one by one

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

// Check if the queen can be placed on

// board[i][col]

if (isSafe(board, i, col)) {

// Place this queen in board[i][col]

board[i][col] = 1;

// Recur to place rest of the queens

if (solveNQUtil(board, col + 1))

return true;

// If placing queen in board[i][col]

// doesn't lead to a solution, then

// remove queen from board[i][col]

board[i][col] = 0; // BACKTRACK

}

}

// If the queen cannot be placed in any row in

// this column col then return false

return false;

}

int main()

{

printf("Enter the size of matrix (nxn): ");

scanf("%d", &N);

int board[N][N];

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

{

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

{

board[i][j] = 0;

}

}

if (solveNQUtil(board, 0) == false) {

printf("Solution does not exist \n");

return false;

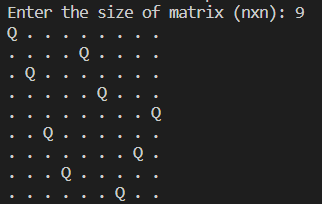
}

printSolution(board);

return 0;

}

* Output:



* Time Complexity:

1. solveNQUtil function: The time complexity of this function is exponential, O(N!), where N is the size of the board.
2. isSafe function: The time complexity of this function is O(N), as it checks each row, upper diagonal, and lower diagonal.
3. The printSolution function: This function simply prints the solution, which takes O(N2) time as it iterates over all cells of the board.

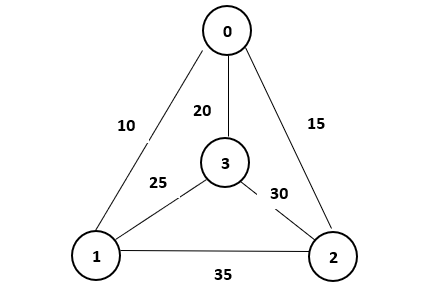
Overall Complexity - O(N!) or O(NN).

**Practical No. 9 [Branch and Bound]**

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point.

Solve this problem using branch and bound technique.

For example, consider the following graph:



A Travelling Salesman Problem (TSP) tour in the graph is . The cost of the tour is .

* Code:

#include <stdio.h>

#include <limits.h>

#define N 10 // Maximum number of cities

int n; // Number of cities

int graph[N][N]; // Graph representing distances between cities

int minCost = INT\_MAX; // Variable to store the minimum cost of the tour

int finalPath[N]; // Array to store the final path

// Function to calculate the lower bound cost of a partial tour using a greedy approach

int calculateLB(int path[], int level, int visited[]) {

// Greedy approach: return the sum of minimum remaining edges from each unvisited city

int lb = 0;

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

if (!visited[i]) {

int minDist = INT\_MAX;

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

if (!visited[j] && graph[i][j] < minDist) {

minDist = graph[i][j];

}

}

lb += minDist;

}

}

return lb;

}

// Function to find the minimum cost using branch and bound

void tsp(int level, int cost, int path[], int visited[]) {

// Base case: if all cities are visited

if (level == n) {

// Check if there's a path back to the starting city

if (graph[path[level - 1]][0] != 0) {

int totalCost = cost + graph[path[level - 1]][0]; // Total cost of the tour

// Update minimum cost if this tour is better

if (totalCost < minCost) {

minCost = totalCost;

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

finalPath[i] = path[i];

}

}

}

return;

}

// Explore child nodes (unvisited cities)

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

if (!visited[i] && graph[path[level - 1]][i] != 0) {

visited[i] = 1; // Mark city i as visited

path[level] = i; // Add city i to the current path

int lowerBound = calculateLB(path, level, visited);

if (cost + graph[path[level - 1]][i] + lowerBound < minCost) {

tsp(level + 1, cost + graph[path[level - 1]][i], path, visited); // Recur for city i

}

visited[i] = 0; // Mark city i as unvisited after exploring all paths from it

}

}

}

int main() {

printf("Enter the number of cities: ");

scanf("%d", &n);

// Prompt user to input the distances between cities

printf("Enter the distances between cities (0 for same city, -1 if not connected):\n");

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

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

scanf("%d", &graph[i][j]); // Input the distances into the adjacency matrix

}

}

// Start with the first city (0)

int path[N], visited[N] = {0};

path[0] = 0;

visited[0] = 1;

// Solve TSP using branch and bound

tsp(1, 0, path, visited);

// Output the minimum cost and the corresponding path

printf("The minimum cost of the tour is: %d\n", minCost);

printf("The tour path is: ");

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

printf("%d ", finalPath[i]);

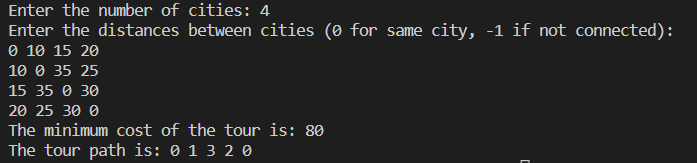
}

printf("%d\n", finalPath[0]); // Return to the starting city

return 0;

}

* Output:



* Time Complexity:

Reading the distances between cities takes O(n2) time.

Calculating the lower bound using calculateLB function takes O(n2) time in the worst case.

The worst-case scenario is when we explore all possible permutations of cities, resulting in a time complexity of O(n!).

Printing the minimum cost and the corresponding path takes O(n) time.

Overall Complexity - **O(n!)**

**Practical No. 10**

To design and solve given problems using different algorithmic approaches and analyze their complexity.

1. Your friends are starting a security company that needs to obtain licenses for different pieces of cryptographic software. Due to regulations, they can only obtain these licenses at the rate of at most one per month. Each license is currently selling for a price of $100. However, they are all becoming more expensive according to exponential growth curves: in particular, the cost of license increases by a factor of each month, where is a given parameter. This means that if license is purchased months from now, it will cost . We will assume that all the price growth rates are distinct; that is, for licenses (even though they start at the sameprice of $100).

The question is: Given that the company can only buy at most one license a month, in which order should it buy the licenses so that the total amount of money it spends is as small as possible?

Give an algorithm that takes the rates of price growth , and computes an order in which to buy the licenses so that the total amount of money spent is minimized. The running time of your algorithm should be polynomial in .

* **Code:**

# include <stdio.h>

int main()

{

int n;

printf("Enter the number of licence required: ");

scanf("%d", &n);

int rates[n];

int sortedrates[n];

int cost = 0;

// Taking rates as input

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

{

printf("Enter rate %d: ", i+1);

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

}

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

{

sortedrates[i] = rates[i];

}

// Sorting rates

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

{

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

{

if (sortedrates[i] < sortedrates[j])

{

int temp = sortedrates[i];

sortedrates[i] = sortedrates[j];

sortedrates[j] = temp;

}

}

}

// Printing the order of rates to be taken for minimum cost.

printf("Order of rates is (rate no., rate): ");

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

{

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

{

if (sortedrates[i] == rates[j])

printf("(%d,%d) ", j+1, sortedrates[i]);

}

}

printf("\n");

// Printing the cost required for n licences.

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

{

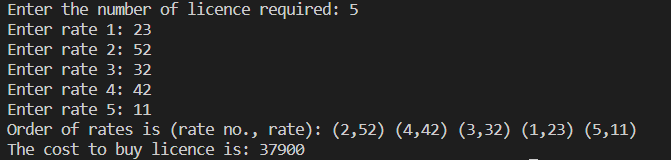
cost += sortedrates[i]\*100\*(i+1);

}

printf("The cost to buy licence is: %d \n", cost);

}

* **Output:**



* **Time Complexity:**

The time complexity of this code is **O(n2)** as bubble sort is used here. Also for printing the order of rates, two nested for loops are used.

1. Suppose you are given an array with entries, with each entry holding a distinct number. You are told that the sequence of values is unimodal. That is, for some index between and , the values in the array entries increase up to position in and then decrease the remainder of the way until position . (So if you were to draw a plot with the array position on the -axis and the value of the entry on the -axis, the plotted points would rise until -value , where they’d achieve their maximum value, and then fall from there on). You’d like to find the “peak entry” without having to read the entire array - in fact, by reading as few entries of as possible. Show how to find the entry by reading at most entries of .

* **Code:**

# include <stdio.h>

int main()

{

int n;

printf("Enter the length of array: ");

scanf("%d", &n);

int arr[n];

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

{

printf("Enter element %d: ", i+1);

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

}

int lb = 0;

int ub = n-1;

while(lb < ub)

{

int mid = (lb + ub)/2;

if(arr[mid] < arr[mid+1])

lb = mid+1;

else if(arr[mid] > arr[mid+1])

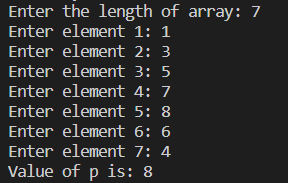
ub = mid;

}

printf("Value of p is: %d \n", arr[lb]);

}

* **Output:**



* **Time Complexity:**

Time complexity of this code is **O(logn)** as the logic similar to binary search is used here and the complexity of binary search is O(logn).