Daa lab

Name : Kabir Mehta
Sap : 500120237
Batch : 57

Q1. Implement the insertion inside iterative and recursive Binary search tree and compare their Performance.

Ans :

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

struct Node
{
  int data;    struct Node
*left;    struct Node
*right;
};

struct Node *createNode(int value)
{
  struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));    if (newNode ==
NULL)
  {
     printf("Memory allocation failed\n");       return
NULL;
  }
  newNode->data = value;    newNode->left = NULL;
newNode->right = NULL;
  return newNode;
}

void inorder(struct Node *root)
{
  if (root != NULL)
  {
     inorder(root->left);       printf("%d ",
root->data);       inorder(root->right);
  }
}

struct Node *insert(struct Node *root, int value)
{
  if (root == NULL)
```

```
  {
    return createNode(value);
  }
  if (value < root->data)
  {
    root->left = insert(root->left, value);
  }
  else if (value > root->data)
  {
    root->right = insert(root->right, value);
  }
  return root;
}

int main() {
  struct Node *root = NULL;    int
value, n, option;

  printf("Enter the number of values you want to insert initially: ");    scanf("%d", &n);

  for (int i = 0; i < n; i++)
  {
    printf("Enter value%d: ", i + 1);
scanf("%d", &value);       root = insert(root,
value);
  }

  printf("Inorder Traversal of the initial BST: ");
inorder(root);    printf("\n");

  while (1)    {
    printf("\nDo you want to insert more elements? (1 for Yes, 0 for No): ");       scanf("%d",
&option);
    if (option == 0)
    {
      break;
    }
    printf("Enter value to insert: ");
scanf("%d", &value);       root = insert(root, value);
printf("Updated Inorder Traversal: ");
inorder(root);       printf("\n");
  }

  return 0;
}
```

Q2 . Implement divide and conquer based merge sort and quick sort algorithms and compare their performance for the same set of elements.

Ans : Merge
sort

```
#include <stdio.h> #include
<stdlib.h>
int main()
{
  int n1, n2;    printf("Enter length of
arr1: ");    scanf("%d", &n1);
printf("Enter length of arr2: ");
scanf("%d", &n2);
  int *arr1 = (int *)malloc(n1 * sizeof(int));    int *arr2 =
(int *)malloc(n2 * sizeof(int));
  if (arr1 == NULL || arr2 == NULL)
  {
     printf("Memory allocation failed\n");
return -1;    }

  printf("Enter the elements of arr1: \n");    for (int i =
0; i < n1; i++)
  {
     scanf("%d", &arr1[i]);
  }

  printf("Enter the elements of arr2: \n");    for (int j =
0; j < n2; j++)
  {
     scanf("%d", &arr2[j]);
  }

  printf("arr1:\n");    for (int i = 0;
i < n1; i++)
  {
     printf("%d ", arr1[i]);
  }

  printf("\narr2:\n");    for (int i =
0; i < n2; i++)
  {
     printf("%d ", arr2[i]);
  }

  int n = n1 + n2;
  int *mergedarr = (int *)malloc(n * sizeof(int));    if
(mergedarr == NULL)
```

```
  {
     printf("Memory allocation failed\n");
free(arr1);       free(arr2);       return -1;    }    int i
= 0, j = 0, k = 0;
  while (i < n1 && j < n2)
  {
     if (arr1[i] < arr2[j])
     {          mergedarr[k] = arr1[i];
i++;       }       else
     {          mergedarr[k] = arr2[j];
j++;
     }       k++;
}

  while (i < n1)    {
mergedarr[k] = arr1[i];       i++;
k++;    }

  while (j < n2)    {
mergedarr[k] = arr2[j];       j++;
k++;    }

  printf("\nMerged array:\n");    for (int
j = 0; j < n; j++)
  {
     printf("%d ", mergedarr[j]);
  }

  free(arr1);    free(arr2);
free(mergedarr);

  return 0;
}
```

Quick sort

```
#include <stdio.h>
void quickSort(int arr[], int l, int h) {    if (l <
h) {       int pivot = arr[h];       int i = l - 1;
for (int j = l; j < h; j++) {          if (arr[j] <=
pivot) {              i++;              int temp =
arr[i];              arr[i] = arr[j];
            arr[j] = temp;
         }
     }
     int temp = arr[i + 1];       arr[i +
1] = arr[h];       arr[h] = temp;

     int pivotIndex = i + 1;       quickSort(arr, l,
pivotIndex - 1);       quickSort(arr, pivotIndex + 1,
h);
```

```
  }
}

int main() {    int
n;
  printf("Enter length of array: ");
scanf("%d", &n);    int arr[n];

  printf("Enter the elements of array: \n");    for(int
i = 0; i < n; i++) {       scanf("%d", &arr[i]);
  }    quickSort(arr, 0, n - 1);

  printf("Sorted array:\n");    for (int
i = 0; i < n; i++) {       printf("%d ",
arr[i]);
  }    printf("\n");

  return 0;
}
```

For most general purposes, **Quick Sort** is preferred due to its speed and space efficiency, but **Merge Sort** is chosen for stability and consistent performance, particularly with larger datasets.

Q3. Compare the performance of Strassen method of matrix multiplication with traditional way of matrix multiplication.

Ans :

```
#include <stdio.h>

void split(int n, int matrix[n][n], int a[n/2][n/2], int b[n/2][n/2], int c[n/2][n/2], int d[n/2] [n/2]) {
  int mid = n / 2;    for (int i = 0; i < mid;
i++) {       for (int j = 0; j < mid; j++) {
a[i][j] = matrix[i][j];                    b[i][j] =
matrix[i][j + mid];                c[i][j] =
matrix[i + mid][j];
        d[i][j] = matrix[i + mid][j + mid];
     }
  }
}

void add(int n, int A[n][n], int B[n][n], int C[n][n]) {    for (int i =
0; i < n; i++)       for (int j = 0; j < n; j++)          C[i][j] = A[i][j] +
B[i][j];
```

```
}

void subtract(int n, int A[n][n], int B[n][n], int C[n][n]) {    for (int i =
0; i < n; i++)       for (int j = 0; j < n; j++)
        C[i][j] = A[i][j] - B[i][j];
}

void strassen(int n, int A[n][n], int B[n][n], int C[n][n]) {    if (n == 1) {
     C[0][0] = A[0][0] * B[0][0];
     return;    }    int

mid = n / 2;

  int a[mid][mid], b[mid][mid], c[mid][mid], d[mid][mid];    int
e[mid][mid], f[mid][mid], g[mid][mid], h[mid][mid];
  int m1[mid][mid], m2[mid][mid], m3[mid][mid], m4[mid][mid], m5[mid][mid], m6[mid][mid],
m7[mid][mid];    int temp1[mid][mid], temp2[mid][mid];

  split(n, A, a, b, c, d);    split(n, B, e,
f, g, h);

  add(mid, a, d, temp1);    add(mid, e, h,
temp2);    strassen(mid, temp1, temp2, m1);

  add(mid, c, d, temp1);    strassen(mid,
temp1, e, m2);    subtract(mid, f, h, temp2);
strassen(mid, a, temp2, m3);

  subtract(mid, g, e, temp2);    strassen(mid, d,
temp2, m4);

  add(mid, a, b, temp1);    strassen(mid,
temp1, h, m5);

  subtract(mid, c, a, temp1);    add(mid, e, f,
temp2);    strassen(mid, temp1, temp2, m6);

  subtract(mid, b, d, temp1);    add(mid, g, h,
temp2);    strassen(mid, temp1, temp2, m7);

  for (int i = 0; i < mid; i++) {       for (int j
= 0; j < mid; j++) {
        C[i][j] = m1[i][j] + m4[i][j] - m5[i][j] + m7[i][j];
        C[i][j + mid] = m3[i][j] + m5[i][j];
        C[i + mid][j] = m2[i][j] + m4[i][j];
        C[i + mid][j + mid] = m1[i][j] - m2[i][j] + m3[i][j] + m6[i][j];
     }
```

```
  }
}

int main() {    int n = 4;
int A[4][4] = {       {17,
34, 7, 2},
      {22, 10, 11, 0},
      {13, 2, 3, 4},
      {0, 1, 4, 5}
  };    int B[4][4] = {
{3, 4, 2, 17},
      {2, 1, 11, 0},
      {0, 11, 7, 1},
      {1, 0, 22, 3}
  };
  int C[4][4] = {0};    strassen(n, A, B,

C);    printf("Product achieved using

Strassen's algorithm:\n");    for (int i =

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

j++) {          printf("%d\t", C[i][j]);

      }
      printf("\n");
  }

  return 0;
}
```

Traditional Method:

Based on the definition of matrix multiplication.
Requires O(n^3) scalar multiplications for two n × n matrices.
Straightforward to implement.

Strassen's Method:

A divide-and-conquer approach.
Reduces the number of scalar multiplications by splitting matrices into smaller submatrices.

Uses 7 scalar multiplications instead of 8 for 2 × 2 submatrices.

Q4. Implement the activity selection problem to get a clear understanding of greedy approach.

Ans :

```
#include <stdio.h>
void printMaxActivities(int s[], int f[], int n) {    int i, j;
  printf("Following activities are selected:\n");    i = 0;
printf("%d ", i);    for (j = 1; j < n; j++) {       if (s[j] >= f[i]) {
printf("%d ", j);
         i = j;
     }
  }
  printf("\n");
}
int main() {
  int s[] = { 1, 3, 0, 5, 8, 5 };    int f[] = {
2, 4, 6, 7, 9, 9 };    int n = sizeof(s) /
sizeof(s[0]);    printMaxActivities(s, f,
n);    return 0;
}
```

Q5. Get a detailed insight of dynamic programming approach by the implementation of Matrix Chain Multiplication problem and see the impact of parenthesis positioning on time requirements for matrix multiplication.

Ans :

```
#include <stdio.h> #include
<limits.h>

int matrixChainOrder(int p[], int n) {
  int m[n][n];
  int i, j, k, L, q;

  for (i = 1; i < n; i++)       m[i][i] = 0;

  for (L = 2; L < n; L++) {       for (i = 1; i < n
- L + 1; i++) {
        j = i + L - 1;          m[i][j] = INT_MAX;
for (k = i; k <= j - 1; k++) {
```

```
            q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];             if (q <
m[i][j])                m[i][j] = q;
        }
     }   }

  return m[1][n - 1];
}

int main() {
  int arr[] = {1, 2, 3, 4};
  int size = sizeof(arr) / sizeof(arr[0]);
  int minCost = matrixChainOrder(arr, size);

  printf("Minimum number of multiplications is %d\n", minCost);    return 0;
}
```

Q6. Compare the performance of Dijkstra and Bellman ford algorithm for the single source shortest path problem.

Ans :

```
#include <stdio.h>
#include <stdlib.h> #include
<limits.h>
#include <stdbool.h> #define INF

INT_MAX

void dijkstra(int graph[][5], int src, int V) {    int
dist[V], visited[V];    for (int i = 0; i < V; i++) {
     dist[i] = INF;       visited[i] =
0;
  }    dist[src] = 0;

  for (int count = 0; count < V - 1; count++) {
     int u = -1, min = INF;       for (int v = 0; v < V;
v++) {          if (!visited[v] && dist[v] <= min) {
            min = dist[v];
            u = v;
        }
}
     visited[u] = 1;

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

```
        if (!visited[v] && graph[u][v] && dist[u] != INF && dist[u] + graph[u][v] < dist[v]) {
            dist[v] = dist[u] + graph[u][v];
        }
      }
  }
  printf("Dijkstra distances from source %d:\n", src);
  for (int i = 0; i < V; i++)
      printf("%d -> %d: %d\n", src, i, dist[i]);
}

void bellmanFord(int graph[][3], int V, int E, int src) {    int dist[V];
for (int i = 0; i < V; i++)       dist[i] = INF;    dist[src] = 0;

  for (int i = 1; i <= V - 1; i++) {       for
(int j = 0; j < E; j++) {          int u =
graph[j][0];          int v = graph[j][1];
int weight = graph[j][2];
        if (dist[u] != INF && dist[u] + weight < dist[v])              dist[v] =
dist[u] + weight;
      }   }

  for (int j = 0; j < E; j++) {       int u = graph[j][0];       int v =
graph[j][1];       int weight = graph[j][2];       if (dist[u] != INF &&
dist[u] + weight < dist[v]) {
        printf("Graph contains negative weight cycle\n");          return;
      }
  }

  printf("Bellman-Ford distances from source %d:\n", src);
  for (int i = 0; i < V; i++)
      printf("%d -> %d: %d\n", src, i, dist[i]);
}

int main() {    int
graph1[5][5] = {
      {0, 4, 0, 0, 0},
      {0, 0, 8, 0, 0},
      {0, 0, 0, 7, 0},
      {0, 0, 0, 0, 9},
      {0, 0, 0, 0, 0}
  };
  int graph2[8][3] = {
      {0, 1, 4}, {0, 2, 4}, {1, 2, -5}, {2, 3, 6},
      {3, 1, -3}, {1, 4, 8}, {4, 3, 1}, {4, 0, 2}
  };    int V1 = 5, V2 = 5, E2 = 8;
```

```
  dijkstra(graph1, 0, V1);    bellmanFord(graph2, V2, E2,
0);

  return 0;
}
```

Q7. Through 0/1 Knapsack problem, analyze the greedy and dynamic programming approach for the same dataset.

Ans :

```
#include <stdio.h> int max(int a, int b) { return (a > b) ?

a : b; }

int knapSack(int W, int wt[], int val[], int n)
{
  if (n == 0 || W == 0)       return 0;

  if (wt[n - 1] > W)
     return knapSack(W, wt, val, n - 1);

  else       return
max(
        val[n - 1] + knapSack(W - wt[n - 1], wt, val, n - 1),          knapSack(W, wt, val, n -
1));
}

int main()
{
  int profit[] = { 60, 100, 120,150 };    int
weight[] = { 10, 20, 30, 35  };    int W = 50;
  int n = sizeof(profit) / sizeof(profit[0]);
  printf("%d", knapSack(W, weight, profit, n));    return 0;
}
```

Q8. Implement the sum of subset.

Ans :

```
#include <stdio.h>

int isSubsetSumRec(int arr[], int n, int sum) {    if (sum
== 0) {        return 1;
  }    if (n == 0) {
return 0;
  }
  if (arr[n - 1] > sum) {
     return isSubsetSumRec(arr, n - 1, sum);
  }
  return isSubsetSumRec(arr, n - 1, sum) || isSubsetSumRec(arr, n - 1, sum - arr[n - 1]);
}

int isSubsetSum(int arr[], int n, int sum) {
  return isSubsetSumRec(arr, n, sum);
}

int main() {
  int arr[] = {3, 34, 4, 12, 5, 2};    int sum =
9;
  int n = sizeof(arr) / sizeof(arr[0]);

  if (isSubsetSum(arr, n, sum)) {
     printf("True\n");
  } else {
     printf("False\n");
  }

  return 0;
}
```

Q9. Compare the Backtracking and Branch & Bound Approach by the implementation

of 0/1 Knapsack problem. Also compare the performance with dynamic programming approach.

Ans :

Comparison:

Backtracking:

Time Complexity: O(2n), where n is the number of items. This is because it explores all possible combinations of items. Space Complexity: O(n) for the recursion stack.

Performance: This approach is inefficient for larger problems, as it explores every subset of items, making it unsuitable for large datasets.

Branch and Bound:
Time Complexity: O(2n) in the worst case. However, it can be much better than backtracking due to pruning, making it more efficient in practice.
Space Complexity: O(n) for the recursion stack.
Performance: Branch and Bound reduces the search space by pruning, which improves the performance compared to pure backtracking, but it is still exponential in the worst case.

Dynamic Programming:
Time Complexity: O(nW), where n is the number of items and W is the knapsack capacity.
Space Complexity: O(nW) for the DP table.
Performance: This is the most efficient approach among the three, particularly when the problem size grows large.
The DP approach guarantees optimal solutions in polynomial time, making it the best for solving knapsack problems with large inputs.

Q10. Compare the performance of Rabin-Karp, Knuth-MorrisPratt and naive stringmatching algorithms.

Ans :

1. Naive String-Matching Algorithm
The naive string-matching algorithm checks for a match at every possible starting position in the text.
It compares the substring of the text with the pattern character by character.
Algorithm:
Start from the first character of the text.
Compare the pattern with the substring starting at that position.
If all characters match, return the starting index of the match.
If not, move to the next position and repeat the comparison.
Time Complexity:
Best Case: O(n), when the pattern matches immediately or no characters match.
Worst Case: O(n×m), where n is the length of the text and m is the length of the pattern. This happens when there are many false matches.

2. Karp Algorithm

The Rabin-Karp algorithm uses a hash function to calculate a hash value for the pattern and for each substring of the text.
If the hash values match, it checks the substring character by character for a match.
Algorithm:
Compute the hash of the pattern and the hash of the initial window of the text.
Slide the window one character at a time, updating the hash and comparing it with the pattern's hash.
If hashes match, compare the actual substrings to verify the match.
Return all starting indices where the hash matches.
Time Complexity:
Best Case: O(n+m) when all hash values are unique or there are no matches.
Worst Case: O(n×m), when there are many hash collisions.
Average Case: O(n+m), assuming a good hash function and low collision rate.

3. Knuth-Morris-Pratt (KMP) Algorithm
The KMP algorithm preprocesses the pattern to create a partial match table (also called "prefix function") that helps skip unnecessary character comparisons.
This table contains the length of the longest prefix that is also a suffix for every position in the pattern.
Algorithm:
Preprocess the pattern to create the partial match table.
Use the table to skip characters in the text when mismatches occur, thereby avoiding redundant comparisons.
Time Complexity:
Best Case: O(n), when the pattern and text are mostly matching. Worst Case: O(n+m), where n is the length of the text and m is the length of the pattern.
Average Case: O(n+m), due to preprocessing the pattern and skipping unnecessary comparisons.