Ahsania Mission University of Science & Technology

Lab Report

Course Code: CSE 2202

Course Title: Computer Algorithm Sessional

Experiment No: 05

Experiment Date: 12.03.25

Submitted By:

Mehrin Nusrat Chowdhury

Roll: 1012320005101026

1st Batch, 2nd Year, 1st Semester

Department of Computer Science and Engineering

Ahsania Mission University of Science & Technology

Submitted To:

Md. Fahim Faisal

Lecturer

Department of Computer Science and Engineering

Faculty of Engineering, Ahsania Mission University of Science & Technology

Submission Date: 19.03.25

Task 01: A C++ program that will implement the Binary Search algorithm.

Theory:

Binary search is a widely used algorithm for searching sorted arrays efficiently. It operates by repeatedly dividing the search interval in half, narrowing down the location of the target item. The process begins by comparing the middle element of the array to the search item. Based on the comparison, either the lower or upper half of the array is considered for the next iteration. This eliminates half of the potential search area in every step, making binary search significantly faster than linear search in terms of computational complexity. The method guarantees optimal performance for sorted datasets and is commonly employed in various applications like databases, dictionaries, and search engines. Its efficiency is marked by a time complexity of (*O(log n)*).

Code:

```
#include<iostream>
using namespace std;

int main()
{
   int n;
   cout << "Enter the number of the elements: ";
   cin >> n;

   int a[n];
   cout << "Enter the elements: ";
   for(int i = 0 ; i < n ; i++)
   {
      cin >> a[i];
   }

   int item;
   cout << "Enter the Item: ";
   cin >> item;

   int b = 0;
   int e = n - 1;
   int mid = (b + e) / 2;
   int loc;
```

```
    while(b <= e && a[mid] != item)
    {
        if(item < a[mid])
        {
            e = mid - 1;
        }
        else
        {
            b = mid + 1;
        }

        mid = (b + e) / 2;
    }

    if(a[mid] == item)
    {
        loc = mid;
        cout << "Item found in location " << loc;
    }
    else
    {
        loc = NULL;
        cout << "Item not found in any location.";
    }

    return 0;
}
```

Output(s):

```
Enter the number of the elements: 10
Enter the elements: 12 34 45 66 67 78 79 81 89 95
Enter the Item: 89
Item found in location 8
Process returned 0 (0x0)   execution time : 53.430 s
Press any key to continue.
```

```
Enter the number of the elements: 10
Enter the elements: 12 34 45 66 67 78 79 81 89 95
Enter the Item: 52
Item not found in any location.
Process returned 0 (0x0)   execution time : 17.227 s
Press any key to continue.
```

Conclusion:

Through the implementation of binary search, we observed its systematic approach to locating elements in a sorted array. The program accurately demonstrated how the algorithm partitions the array into smaller intervals and checks the middle element iteratively. This study reinforced the understanding of binary search's effectiveness in optimizing search operations while reducing computational overhead. The results validated the algorithm's utility in situations requiring quick and efficient search techniques.

Task 02: A C++ program that will implement Merge Sort for Counting Array Inversion.

Theory:

Counting inversions in an array is a problem related to determining how out of order an array is with respect to its sorted state. An inversion is defined as a pair of indices (*(i, j)*) in an array such that ($i < j$) and (*arr[i] > arr[j]*). Correcting inversions typically involves swapping the inverted elements to achieve a sorted array. The merge sort algorithm is an efficient way to both sort an array and count the number of inversions. During the merging step of merge sort, the algorithm identifies and counts inversions by comparing elements from two subarrays. This dual-purpose approach leverages the divide-and-conquer paradigm, achieving a time complexity of ($O(n \log n)$), making it highly effective for large arrays.

Code:

```
#include<iostream>
#include<cmath>
using namespace std;

int inversionCount = 0;

void Merge(int arr[] , int p , int q , int r)
{
   int n1 = q - p + 1;
   int n2 = r - q;

   int L[n1] , R[n2];

   for(int i = 0 ; i < n1 ; i++)
   {
      L[i] = arr[p + i];
   }

   for(int j = 0 ; j < n2 ; j++)
   {
      R[j] = arr[q + 1 + j];
   }

   int i = 0 , j = 0 , k = p;

   while(i < n1 && j < n2)
```

```
    {
        if(L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
            inversionCount += (n1 - i);
        }
        k++;
    }

    while(i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    while(j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void Merge_sort(int a[] , int p , int r)
{
    if(p < r)
    {
        int q = (p + r) / 2;
        Merge_sort(a , p , q);
        Merge_sort(a, (q + 1) , r);

        Merge(a, p , q , r);

    }
}

int main()
{
    int d;
    cout << "Enter the number of datasets: ";
    cin >> d;

    while(d--)
    {
```

```
        int n;
        cout << "Enter the number of elements of the array: ";
        cin >> n;

        int A[n];
        cout << "Enter the elements of the array: ";
        for(int i = 0 ; i < n ; i++)
        {
           cin >> A[i];
        }

        inversionCount = 0;
        Merge_sort(A , 0 , n - 1);

        cout << "\nNumber of swaps: " << inversionCount << "\n";
    }

    return 0;
}
```

Output:

```
Enter the number of datasets: 2
Enter the number of elements of the array: 5
Enter the elements of the array: 1 1 1 2 2

Number of swaps: 0
Enter the number of elements of the array: 5
Enter the elements of the array: 2 1 3 1 2

Number of swaps: 4

Process returned 0 (0x0)   execution time : 64.666 s
Press any key to continue.
```

Conclusion:

The implementation of merge sort for counting inversions successfully highlighted the algorithm's ability to sort data while quantifying the level of disorder within the array. By integrating inversion counting into the merging step, the program demonstrated the efficiency of solving two problems simultaneously. This experiment showcased the practical application of the divide-and-conquer approach in tackling complex computational problems and reinforced the importance of algorithmic optimization for performance-critical tasks.

Task 03: A C++ program that will construct a permutation for controlled Merge Sort calls.

Theory:

Merge Sort is a widely used sorting algorithm based on the divide-and-conquer approach. It recursively splits an array into two halves, sorts them, and merges them back together. The depth of the recursion tree in Merge Sort is $\mathbf{\log_2(n)}$, but the number of function calls varies depending on how the array is divided.

In this problem, we aim to construct a permutation of size **n** such that sorting it with Merge Sort results in exactly **k** function calls. The function calls range from **n** (best case) to **2n - 1** (worst case), depending on the arrangement of elements. By strategically modifying the order of elements, we control the number of recursive calls to match the required value of **k**. If **k** falls outside this valid range, it is impossible to construct such a permutation.

Code:

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;

void constructPermutation(vector<int>& arr , int left , int right , int& k)
{
   if(right - left <= 1 || k <= 1)
   {
      return;
   }

   int mid = (left + right) / 2;

   k -= 2;

   constructPermutation(arr , left , mid , k);
   constructPermutation(arr , mid , right , k);

   if(right - left == 3)
   {
      swap(arr[left], arr[left + 1]);
   }
   else
```

```
    {
        reverse(arr.begin() + left, arr.begin() + right);
    }
}

bool isValidPermutation(int n , int k)
{
    return (k % 2 == 1) && (k <= (2 * n - 1));
}

vector<int> findPermutation(int n , int k)
{
    if(!isValidPermutation(n , k))
    {
        return {-1};
    }

    vector<int> arr(n);
    for(int i = 0 ; i < n ; i++)
    {
        arr[i] = i + 1;
    }

    if(k == 1)
    {
        return arr;
    }

    constructPermutation(arr , 0 , n , k);

    return arr;
}

int main()
{
    int n , k;
    cin >> n >> k;

    vector<int> result = findPermutation(n , k);

    if(result[0] == -1)
    {
        cout << "-1" << "\n";
    }
    else
    {
        for(int num : result)
        {
            cout << num << " ";
        }
```

```
        cout << "\n";
    }

    return 0;
}
```

Output(s):

```
3 3
2 1 3

Process returned 0 (0x0)   execution time : 2.692 s
Press any key to continue.
```

```
4 1
1 2 3 4

Process returned 0 (0x0)   execution time : 5.475 s
Press any key to continue.
```

Conclusion:

To construct a permutation of size n that results in exactly k recursive calls during merge sort, we exploit how disorder influences recursion depth. Valid values for k must be odd and lie within the range $[n, 2n-1]$. By selectively disturbing array segments, we can force merge sort to split and recurse accordingly. If k falls outside this range, no such permutation can be formed.

Task 04: A C++ program that will implement binary Search on Unsorted Permutations.

Theory:

Binary search is a fundamental algorithm used to efficiently locate a target value in a sorted array. It works by repeatedly dividing the search interval in half. If the target value is less than the middle element, the search continues in the left half; if greater, it proceeds in the right half.

However, the problem addressed here challenges that precondition: it applies binary search to a **permutation**, which by definition is unsorted but contains all elements from 1 to n exactly once. Since comparisons still determine the search path, it's possible to simulate binary search behavior even without sorting — but it may lead to incorrect results unless the array is adjusted.

To correct the path, we're allowed to perform up to **two swaps** before initiating the search. These swaps aim to influence how binary search traverses the array, ensuring that the final index l reached by the search holds the target value. The solution lies not in enforcing global order but in localized correction via strategic swaps.

This approach highlights the distinction between **sortedness** and **comparative navigation**, and explores how minimal transformations can yield correct algorithmic outcomes in constrained environments.

Code:

```
#include <iostream>
#include <vector>
#include <tuple>
#include <algorithm>
using namespace std;

int simulate(const vector<int>& p, int x) {
    int l = 0;
    int r = p.size();
    while (r - l > 1)
    {
        int m = (l + r) / 2;
        if(p[m] <= x)
        {
```

```
            l = m;
        }
        else
        {
            r = m;
        }
    }
    return l;
}

int main()
{
    int t;
    cin >> t;
    while(t--)
    {
        int n, x;
        cin >> n >> x;
        vector<int> p(n);
        for(int i = 0; i < n; ++i)
        {
            cin >> p[i];
        }

        int originalTarget = simulate(p, x);
        if(p[originalTarget] == x)
        {
            cout << "0\n";
            continue;
        }

        vector<tuple<int, int, int, int>> twoSwap; // (a,b,c,d)
        vector<pair<int, int>> oneSwap;

        for(int a = 0; a < n; ++a)
        {
            for(int b = a + 1; b < n; ++b)
            {
                swap(p[a], p[b]);
                for(int c = 0; c < n; ++c)
                {
                    for(int d = c + 1; d < n; ++d)
                    {
                        if(c == a && d == b)
                        {
                            continue;
                        }
                        swap(p[c], p[d]);
                        int l = simulate(p, x);
                        if(p[l] == x)
```

```
                {
                    twoSwap.emplace_back(a + 1, b + 1, c + 1, d + 1);
                }
                swap(p[c], p[d]);
            }
        }
        swap(p[a], p[b]);
    }
}

if(!twoSwap.empty())
{
    sort(twoSwap.begin(), twoSwap.end());
    cout << "2\n";
    auto [a, b, c, d] = twoSwap[0];
    cout << a << " " << b << "\n" << c << " " << d << "\n";
    continue;
}

for(int a = 0; a < n; ++a)
{
    for(int b = a + 1; b < n; ++b)
    {
        swap(p[a], p[b]);
        int l = simulate(p, x);
        if(p[l] == x)
        {
            oneSwap.emplace_back(a + 1, b + 1);
        }
        swap(p[a], p[b]);
    }
}

if(!oneSwap.empty())
{
    sort(oneSwap.begin(), oneSwap.end());
    cout << "1\n" << oneSwap[0].first << " " << oneSwap[0].second << "\n";
    continue;
}

cout << "-1\n";
  }
  return 0;
}
```

Output:

```
5

6 3
1 2 3 4 5 6
0

6 5
3 1 6 5 2 4
2
1 2
3 4

5 1
3 5 4 2 1
2
1 2
1 5

6 3
4 3 1 5 2 6
2
1 2
2 4

3 2
3 2 1
1
1 2

Process returned 0 (0x0)   execution time : 119.907 s
Press any key to continue.
```

Conclusion:

By simulating binary search on an unsorted permutation and strategically swapping elements, it's possible to guide the search to terminate correctly. The key is to manipulate comparison outcomes — not full array order — to ensure that the final index l contains the target value. The solution involves evaluating all swap combinations and selecting the lexicographically smallest sequence that satisfies the condition. This approach deepens understanding of algorithmic control through minimal data rearrangement.