Fundamentals of Computer Engineering

Homework - 3

Akshata Kumble

1.

The Rand_Select algorithm is used to find the k^{th} smallest element in an unordered list in linear expected time. The algorithm is similar to Quicksort but instead of sorting the array completely, it recursively partitions the array around a pivot and only continues the search on the side that contains the k^{th} smallest element.

- The array is partitioned around a randomly selected pivot such that some elements are smaller than the pivot and others are larger. The pivot is placed in its correct sorted position.
- After partitioning, the position of the pivot is compared with the k^{th} smallest element's position. If the pivot is the k^{th} smallest element, it is returned. Else the search continues either to the left or right part of the array, depending on where the k^{th} smallest element lies.

The expected time complexity is $O(n)$, where n is the number of elements.

main.cpp

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <chrono>

// partitioning array around the pivot and returning the pivot's final position
int partition(std::vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return i + 1;
}

// selecting a random pivot and partitioning around it
int randomized_partition(std::vector<int>& arr, int low, int high) {
    int pivot_idx = low + rand() % (high - low + 1);
    std::swap(arr[pivot_idx], arr[high]);
    return partition(arr, low, high);
}

// rand-select algorithm to find the k-th smallest element
int randomized_select(std::vector<int>& arr, int low, int high, int k) {
    if (low == high) {
        return arr[low];
    }
```

main.cpp

```cpp
    int pivot_idx = randomized_partition(arr, low, high);
    int order = pivot_idx - low + 1;

    if (order == k) {
        return arr[pivot_idx];  // the k-th smallest element
    } else if (k < order) {
        return randomized_select(arr, low, pivot_idx - 1, k);  // recur on the left
    } else {
        return randomized_select(arr, pivot_idx + 1, high, k - order);  // recur on the righ
    }
}

// function to shuffle the array
void shuffle_array(std::vector<int>& arr) {
    std::srand(unsigned(std::time(0)));
    std::random_shuffle(arr.begin(), arr.end());
}

// main function
int main() {
    std::vector<int> A(100);
    for (int i = 0; i < 100; ++i) {
        A[i] = i + 1;
    }

    shuffle_array(A);  // shuffle to create a random permutation

    int k;
    std::cout << "Enter k (1 to 100) to find the k-th smallest element: ";
    std::cin >> k;

    if (k < 1 || k > 100) {
        std::cerr << "Invalid input! k must be between 1 and 100." << std::endl;
        return -1;
    }

    std::cout << "\nOriginal array: ";
    for (int num : A) {
        std::cout << num << " ";
    }
    std::cout << "\n\n";

    // measuring time
    auto start = std::chrono::high_resolution_clock::now();
    int result = randomized_select(A, 0, A.size() - 1, k);
    auto stop = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);

    // results
    std::cout << "Randomized Select result for " << k << "th smallest element: " << result << std::endl;
    std::cout << "Time taken: " << duration.count() << " microseconds" << std::endl;

    return 0;
}
```

input

```
Enter k (1 to 100) to find the k-th smallest element: 45

Original array: 21 65 6 51 26 88 36 40 58 82 66 99 9 79 77 85 53 3 62 100 20 10 31 11 23 17 90 69 84 30 94 70 52 72 63 92 19 50 55 1 12 64 96 81 61 14 8 27 95 86 43 89 4 5 67 83 7 97
 59 57 87 80 34 48 28 15 75 22 38 39 54 71 73 68 47 13 60 93 46 37 29 2 35 74 78 18 33 42 24 25 76 16 44 56 98 32 41 49 45 91

Randomized Select result for 45th smallest element: 45
Time taken: 3 microseconds


...Program finished with exit code 0
Press ENTER to exit console.
```

input

```
Enter k (1 to 100) to find the k-th smallest element: 75

Original array: 49 98 43 55 2 6 78 35 22 57 100 18 56 34 47 80 70 4 99 87 89 96 30 83 26 28 42 90 62 73 5 91 60 86 77 59 32 64 36 33 81 7 51 15 44 58 84 53 24 67 92 94 79 1 12 69 11 23 8 76 19 38 17 45 6
8 74 61 66 27 39 72 25 9 93 63 3 85 95 65 31 21 46 97 52 71 20 16 41 82 88 75 14 37 13 50 29 40 10 48 54

Randomized Select result for 75th smallest element: 75
Time taken: 6 microseconds


...Program finished with exit code 0
Press ENTER to exit console.
```

Select Algorithm.

The select algorithm (Median of Medians) guarantees linear worst case time. It works by finding a good pivot (median of medians) to ensure that the partitions are always balanced enough to achieve linear time. It recursively reduces the problem by dividing the array into manageable subgroups, finding the medians and using the median of medians as the pivot for partitioning.

Key steps:

- Divide input array into groups of 5
- Find median of each group
- Recursively find median of medians
- Use this median as pivot for partitioning.
- Recursively solve smaller subproblems to find the k^{th} smallest element in the appropriate partition

Worse Case : $T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n) = O(n)$

main.cpp

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>

// partitioning the array around a pivot and return its final position
int partition(std::vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; ++j) {
        if (arr[j] <= pivot) {
            i++;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return i + 1;
}

// finding the median of a small group of elements (at most 5)
int find_median(std::vector<int>& arr, int low, int high) {
    std::sort(arr.begin() + low, arr.begin() + high + 1);
    return arr[(low + high) / 2];
}

// function to perform partitioning around the median of medians
int deterministic_partition(std::vector<int>& arr, int low, int high, int pivot) {
    int pivot_idx = std::find(arr.begin() + low, arr.begin() + high + 1, pivot) - arr.begin();
    std::swap(arr[pivot_idx], arr[high]);
    return partition(arr, low, high);
}


int median_of_medians(std::vector<int>& arr, int low, int high, int k) {
    int n = high - low + 1;

```

```cpp
    if (n <= 5) {
        std::sort(arr.begin() + low, arr.begin() + high + 1);
        return arr[low + k - 1];
    }

    // divide arr into groups of 5, and find the median of each group
    std::vector<int> medians;
    for (int i = low; i <= high; i += 5) {
        int group_high = std::min(i + 4, high);
        medians.push_back(find_median(arr, i, group_high));
    }

    // Find the median of the medians
    int median_of_median = median_of_medians(medians, 0, medians.size() - 1, (medians.size() + 1) / 2);

    // partitioning around the median of medians and recur
    int pivot_idx = deterministic_partition(arr, low, high, median_of_median);
    int order = pivot_idx - low + 1;

    if (order == k) {
        return arr[pivot_idx];
    } else if (k < order) {
        return median_of_medians(arr, low, pivot_idx - 1, k);
    } else {
        return median_of_medians(arr, pivot_idx + 1, high, k - order);
    }
}

// main function
int main() {
    std::vector<int> A(100);
    for (int i = 0; i < 100; ++i) {
        A[i] = i + 1;
    }

    // shuffle array to randomize the order
    std::srand(unsigned(std::time(0)));
    std::random_shuffle(A.begin(), A.end());

    // getting user input for k
    int k;
    std::cout << "Enter k (1 to 100) to find the k-th smallest element: ";
    std::cin >> k;

    if (k < 1 || k > 100) {
        std::cerr << "Invalid input! k must be between 1 and 100." << std::endl;
        return -1;
    }


    std::cout << "\nOriginal array: ";
    for (int num : A) {
        std::cout << num << " ";
    }
    std::cout << "\n\n";

    //measuring time
    auto start = std::chrono::high_resolution_clock::now();
    int result = median_of_medians(A, 0, A.size() - 1, k);

    auto stop = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);

    // output the result and the time taken
    std::cout << "Select result for " << k << "th smallest element: " << result << std::endl;
    std::cout << "Time taken: " << duration.count() << " microseconds" << std::endl;

    return 0;
}
```

```
input
Enter k (1 to 100) to find the k-th smallest element: 45

Original array: 38 37 18 76 16 74 87 55 7 97 61 12 72 52 60 79 25 77 32 71 39 42 10 47 8 58 2 80 75 44 1 68 82 13 98 70 84 35 45 19 3 14 86 50 5 27 83 67 33 46 40 22 94 15 41 29 20 9
2 85 51 64 17 28 31 23 36 43 88 57 48 89 11 78 53 63 62 59 24 21 95 66 96 93 69 49 100 9 99 6 65 26 91 4 54 30 81 90 73 34 56

Select result for 45th smallest element: 45
Time taken: 57 microseconds
```

```
input
Enter k (1 to 100) to find the k-th smallest element: 75

Original array: 82 38 55 10 76 19 22 31 74 63 6 96 53 34 32 21 77 56 83 91 65 33 72 47 37 44 3 48 99 93 20 41 69 78 58 52 51 90 75 95 11 7 13 79 57 2 86 67 70 100 59 66 84 50 28 24 81 35 45 62 40 64 29 7
1 5 92 39 60 85 94 36 16 98 17 61 4 89 30 43 46 68 15 12 26 8 73 23 27 97 80 49 9 1 18 14 42 88 54 87 25

Select result for 75th smallest element: 75
Time taken: 43 microseconds


...Program finished with exit code 0
Press ENTER to exit console.
```

Rand-Select algorithm takes comparatively lesser time (3,6 microseconds) than Select (45, 43 microseconds)

3.

The longest common subsequence is a classic dynamic programming problem. The LCS problem finds the longest subsequence present in both sequences in the same order. We solve it by defining a table L[i][j] that holds the length of the LCS up to i^{th} character of the string and j^{th} character of the second string.

Key Steps:

- Initialize L[0][0] as 0 and build the array bottom-up
- For each character comparison:

If characters match L[i][j] = L[i-1][j-1] + 1

Else take the maximum of L[i-1][j] or L[i][j-1]

Time Complexity: O(m x n) where m and n are the lengths of the two seq

main.cpp

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <set>
#include <algorithm>

using namespace std;

// compute the length of the Longest Common Subsequence
int lcs(const string& X, const string& Y, vector<vector<int>>& L) {
    int m = X.length();
    int n = Y.length();

    // initialize the L table in bottom-up fashion
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
            if (X[i - 1] == Y[j - 1]) {
                L[i][j] = L[i - 1][j - 1] + 1; // if characters match
            } else {
                L[i][j] = max(L[i - 1][j], L[i][j - 1]); // if not, take the maximum
            }
        }
    }

    return L[m][n]; // length of LCS
}

// backtracking function to find all LCSs
void findAllLCS(const string& X, const string& Y, vector<vector<int>>& L, int i, int j, string currentLCS, set<string>& allLCS) {

    if (i == 0 || j == 0) {
        reverse(currentLCS.begin(), currentLCS.end());
        allLCS.insert(currentLCS);
        return;
    }
```

```
    // if characters match, include this character in the LCS and move diagonally up left
    if (X[i - 1] == Y[j - 1]) {
        findAllLCS(X, Y, L, i - 1, j - 1, currentLCS + X[i - 1], allLCS);
    } else {
        // if characters don't match, explore both directions ==> up and left
        if (L[i - 1][j] == L[i][j]) {
            findAllLCS(X, Y, L, i - 1, j, currentLCS, allLCS); // move upwards
        }
        if (L[i][j - 1] == L[i][j]) {
            findAllLCS(X, Y, L, i, j - 1, currentLCS, allLCS); // move leftwards
        }
    }
}

int main() {
    string X, Y;

    // user input for the two sequences
    cout << "Enter the first sequence: ";
    cin >> X;
    cout << "Enter the second sequence: ";
    cin >> Y;
    int m = X.length();
    int n = Y.length();

    // create a 2D array to store lengths of longest common subsequence
    vector<vector<int>> L(m + 1, vector<int>(n + 1, 0));

    // compute the length of the LCS
    int length = lcs(X, Y, L);
    cout << "Length of Longest Common Subsequence: " << length << endl;

    // the set to store all unique LCSs
    set<string> allLCS;
    findAllLCS(X, Y, L, m, n, "", allLCS);

    // print all unique LCSs
    cout << "All Longest Common Subsequences:\n";
    for (const string& lcs : allLCS) {
        cout << lcs << endl;
    }
    return 0;
}
```

```
Enter the first sequence: ABAABA
Enter the second sequence: BABBAB
Length of Longest Common Subsequence: 4
All Longest Common Subsequences:
ABAB
ABBA
BAAB
BABA


...Program finished with exit code 0
Press ENTER to exit console.
```

```
Enter the first sequence: ABCBDAB
Enter the second sequence: BDCAB
Length of Longest Common Subsequence: 4
All Longest Common Subsequences:
BCAB
BDAB


...Program finished with exit code 0
Press ENTER to exit console.
```