|  |  |
| --- | --- |
| **Total Marks:** | **04** |
| **Obtained Marks:** |  |

**Design & Analysis of Algorithm**

**Assignment # 03**

**Submitted to: Dr. Shahzad Latif**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Submitted by: Ubaib Bin Waris**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Reg Number: 2212416**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Problem 1: Sorting Student Test Scores**

A university needs to sort student test scores to rank them. The data includes:

* A completely sorted list from one department.
* A reverse-sorted list from another department.
* A randomly ordered list from a third department.

**Tasks:**

1. Implement Merge Sort and Quick Sort to sort the scores.

**Code**

#include <iostream>

using namespace std;

void merge(int arr[], int left, int mid, int right)

{

    int n1 = mid - left + 1;

    int n2 = right - mid;

    int \*leftPart = new int[n1];

    int \*rightPart = new int[n2];

    for (int i = 0; i < n1; i++)

        leftPart[i] = arr[left + i];

    for (int j = 0; j < n2; j++)

        rightPart[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2)

    {

        if (leftPart[i] <= rightPart[j])

        {

            arr[k++] = leftPart[i++];

        }

        else

        {

            arr[k++] = rightPart[j++];

        }

    }

    while (i < n1)

        arr[k++] = leftPart[i++];

    while (j < n2)

        arr[k++] = rightPart[j++];

    delete[] leftPart;

    delete[] rightPart;

}

void mergeSort(int arr[], int left, int right)

{

    if (left < right)

    {

        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);

        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);

    }

}

int partition(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++;

            swap(arr[i], arr[j]);

        }

    }

    swap(arr[i + 1], arr[high]);

    return i + 1;

}

void quickSort(int arr[], int low, int high)

{

    if (low < high)

    {

        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

void printArray(int arr[], int size)

{

    for (int i = 0; i < size; i++)

    {

        cout << arr[i] << " ";

    }

    cout << endl;

}

// Main Function for Testing

int main()

{

    srand(time(0));

    const int SIZE = 20;

    int scores[SIZE];

    for (int i = 0; i < SIZE; i++)

    {

        scores[i] = rand() % 100; // Random numbers between 0 and 99

    }

    cout << "Original Scores: ";

    printArray(scores, SIZE);

    // Testing Merge Sort

    int mergeSortedScores[SIZE];

    copy(scores, scores + SIZE, mergeSortedScores);

    mergeSort(mergeSortedScores, 0, SIZE - 1);

    cout << "Merge Sorted Scores: ";

    printArray(mergeSortedScores, SIZE);

    // Testing Quick Sort

    int quickSortedScores[SIZE];

    copy(scores, scores + SIZE, quickSortedScores);

    quickSort(quickSortedScores, 0, SIZE - 1);

    cout << "Quick Sorted Scores: ";

    printArray(quickSortedScores, SIZE);

    return 0;

}

1. Compare the runtime for sorted, reverse-sorted, and random inputs for datasets of sizes 1,000, 5,000, and 10,000.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Number of**  **input size** | **Merg Sort** | | | **Quick Sort** | | |
| Best  O(n) | Worst  O(n log n) | Average  O(n log n) | Best  O(n log n) | Worst  O(n2) | Average  O(n log n) |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1,000 | 1,000 | 3,000 | 3,000 | 3,000 | 1,000,000 | 3,000 |
| 5,000 | 5,000 | 61,438 | 61,438 | 61,438 | 25,000,000 | 61,438 |
| 10,000 | 10,000 | 132,877 | 132,877 | 132,877 | 100,000,000 | 132,877 |

1. Explain why the performance of Merge Sort and Quick Sort differs for these input types.

**Merge Sort**

* **Performance**: Always O(nlog⁡n)O(n \log n)O(nlogn), regardless of input type.
* **Reason**: Divides the array into halves and merges them in sorted order. The input order doesn't affect this process.

**Quick Sort**

* **Performance**:
  + Best/Average: O(nlog⁡n)O(n \log n)O(nlogn), with good pivot choices (balanced partitions).
  + Worst: O(n2)O(n^2)O(n2), with poor pivot choices (e.g., sorted or reverse-sorted inputs with last/first element as the pivot).
* **Reason**: Efficiency depends on the pivot's ability to split the array evenly. Random inputs typically result in better pivots.

**Problem 2: Merging Sales Data**

An online retailer receives sales data from two warehouses. The data from each warehouse is sorted, and the company needs a single sorted list.

**Tasks:**

1. Use Merge Sort to design an algorithm for merging the two sorted lists.

**Code**

#include <iostream>

#include <vector>

using namespace std;

vector<int> mergeSortedLists(const vector<int>& list1, const vector<int>& list2) {

// Initialize pointers for both lists

int i = 0, j = 0;

vector<int> mergedList;

// Traverse through both lists and merge them

while (i < list1.size() && j < list2.size()) {

if (list1[i] <= list2[j]) {

mergedList.push\_back(list1[i]);

i++;

} else {

mergedList.push\_back(list2[j]);

j++;

}

}

// If any elements are left in list1, add them to the merged list

while (i < list1.size()) {

mergedList.push\_back(list1[i]);

i++;

}

// If any elements are left in list2, add them to the merged list

while (j < list2.size()) {

mergedList.push\_back(list2[j]);

j++;

}

return mergedList;

}

int main() {

// Example sorted lists

vector<int> list1 = {1, 3, 5, 7};

vector<int> list2 = {2, 4, 6, 8};

// Merge the lists

vector<int> mergedList = mergeSortedLists(list1, list2);

// Print the merged sorted list

cout << "Merged sorted list: ";

for (int num : mergedList) {

cout << num << " ";

}

cout << endl;

return 0;

}

1. Compare the performance of Merge Sort with Quick Sort for merging pre-sorted lists of sizes 10,000 and 50,000.

|  |  |  |  |
| --- | --- | --- | --- |
| **Number of**  **input size** | **Merg Sort** | **Quick Sort** | |
| Best  O(n+m) | Best  O(n log n) | Worst  O(n2) |
| 1 | 1 | 1 | 1 |
| 10,000 | 10,000 | 132,877 | 100,000,000 |
| 50,000 | 50,000 | 234,948 | 2,500,000,000 |

1. Analyze the time complexity of both approaches and explain which is better suited for merging sorted data.

**Time Complexity Analysis for Merging Pre-Sorted Lists**

**1. Merge Sort:**

* **Time Complexity**: **O(n + m)**, where n and m are the sizes of the two sorted lists. Since the lists are already sorted, the merging process is linear.
* **Space Complexity**: **O(n + m)** for the merged list.

**2. Quick Sort:**

* **Time Complexity**:
  + Worst case: **O(n^2)** (if pivot is chosen poorly).
  + Average case: **O(n log n)** (with good pivot selection).
* **Space Complexity**: **O(log n)** (average case), **O(n)** (worst case).

**Problem 3: Quick Sort Pivot Optimization**

An analytics team processes datasets of different sizes. They noticed that Quick Sort performs poorly for nearly sorted data due to unbalanced partitions.

**Tasks:**

1. Implement a basic Quick Sort algorithm and analyze its performance for:

a) Nearly sorted data.

b) Randomly ordered data.

c) Reverse-sorted data.

**Code**

#include <iostream>

#include <vector>

#include <ctime>

using namespace std;

// Partition function: selects a pivot and partitions the array around it

int partition(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++;

swap(arr[i], arr[j]);

}

}

swap(arr[i + 1], arr[high]);

return i + 1;

}

// Quick Sort function: recursively sorts the array

void quickSort(vector<int>& arr, int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

// Function to print an array

void printArray(const vector<int>& arr) {

for (int num : arr) {

cout << num << " ";

}

cout << endl;

}

int main() {

// Example for testing with different types of data

// a) Nearly Sorted Data

vector<int> nearlySorted = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

cout << "Nearly Sorted Data:\n";

clock\_t start = clock();

quickSort(nearlySorted, 0, nearlySorted.size() - 1);

clock\_t end = clock();

cout << "Sorted Result: ";

printArray(nearlySorted);

cout << "Time Taken: " << double(end - start) / CLOCKS\_PER\_SEC << " seconds\n\n";

// b) Randomly Ordered Data

vector<int> randomData = {9, 4, 7, 2, 8, 1, 5, 3, 10, 6};

cout << "Randomly Ordered Data:\n";

start = clock();

quickSort(randomData, 0, randomData.size() - 1);

end = clock();

cout << "Sorted Result: ";

printArray(randomData);

cout << "Time Taken: " << double(end - start) / CLOCKS\_PER\_SEC << " seconds\n\n";

// c) Reverse-Sorted Data

vector<int> reverseSorted = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

cout << "Reverse-Sorted Data:\n";

start = clock();

quickSort(reverseSorted, 0, reverseSorted.size() - 1);

end = clock();

cout << "Sorted Result: ";

printArray(reverseSorted);

cout << "Time Taken: " << double(end - start) / CLOCKS\_PER\_SEC << " seconds\n\n";

return 0;

}

2. Modify the algorithm to use the median-of-three pivot selection strategy.

**Code**

#include <iostream>

#include <vector>

#include <ctime>

#include <algorithm> // For std::swap and std::min/max

using namespace std;

// Function to find the median of three elements: first, middle, and last

int medianOfThree(vector<int>& arr, int low, int high) {

int mid = low + (high - low) / 2;

// Sort first, middle, and last elements

if (arr[low] > arr[mid]) swap(arr[low], arr[mid]);

if (arr[low] > arr[high]) swap(arr[low], arr[high]);

if (arr[mid] > arr[high]) swap(arr[mid], arr[high]);

// Place the median at the high position (pivot position)

swap(arr[mid], arr[high]);

return arr[high];

}

// Partition function: selects a pivot and partitions the array around it

int partition(vector<int>& arr, int low, int high) {

int pivot = arr[high]; // pivot is the median of three

int i = low - 1;

for (int j = low; j < high; j++) {

if (arr[j] <= pivot) {

i++;

swap(arr[i], arr[j]);

}

}

swap(arr[i + 1], arr[high]);

return i + 1;

}

// Quick Sort function: recursively sorts the array

void quickSort(vector<int>& arr, int low, int high) {

if (low < high) {

// Choose pivot using median-of-three strategy

medianOfThree(arr, low, high);

// Partition the array and recursively sort the left and right subarrays

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

// Function to print an array

void printArray(const vector<int>& arr) {

for (int num : arr) {

cout << num << " ";

}

cout << endl;

}

int main() {

// Example for testing with different types of data

// a) Nearly Sorted Data

vector<int> nearlySorted = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

cout << "Nearly Sorted Data:\n";

clock\_t start = clock();

quickSort(nearlySorted, 0, nearlySorted.size() - 1);

clock\_t end = clock();

cout << "Sorted Result: ";

printArray(nearlySorted);

cout << "Time Taken: " << double(end - start) / CLOCKS\_PER\_SEC << " seconds\n\n";

// b) Randomly Ordered Data

vector<int> randomData = {9, 4, 7, 2, 8, 1, 5, 3, 10, 6};

cout << "Randomly Ordered Data:\n";

start = clock();

quickSort(randomData, 0, randomData.size() - 1);

end = clock();

cout << "Sorted Result: ";

printArray(randomData);

cout << "Time Taken: " << double(end - start) / CLOCKS\_PER\_SEC << " seconds\n\n";

// c) Reverse-Sorted Data

vector<int> reverseSorted = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

cout << "Reverse-Sorted Data:\n";

start = clock();

quickSort(reverseSorted, 0, reverseSorted.size() - 1);

end = clock();

cout << "Sorted Result: ";

printArray(reverseSorted);

cout << "Time Taken: " << double(end - start) / CLOCKS\_PER\_SEC << " seconds\n\n";

return 0;

}

1. Compare the performance of the basic and optimized Quick Sort for datasets of sizes 1,000, 5,000, and 10,000.

|  |  |  |
| --- | --- | --- |
| **Number of**  **input size** | **Quick Sort** | |
| Best  O(n log n) | Worst  O(n2) |
| 1 | 1 | 1 |
| 1,000 | 3,000 | 1,000,000 |
| 5,000 | 61,438 | 25,000,000 |
| 10,000 | 132,877 | 100,000,000 |

**Problem 4: Sorting Large Datasets**

A logistics company needs to sort delivery orders based on distance from their hub. The dataset includes over 1 million entries.

**Tasks:**

* 1. Compare the performance of Merge Sort and Quick Sort on the dataset.

**Time Complexity Analysis:**

* **Merge Sort**:
  + **Best, Average, Worst Case**: O(nlog⁡n)O(n \log n)O(nlogn), where nnn is the number of elements.
  + Merge Sort always divides the array into two halves and sorts them recursively, making it more predictable in terms of performance. However, it requires additional space to store the merged result during each recursive step.
* **Quick Sort**:
  + **Best and Average Case**: O(nlog⁡n)O(n \log n)O(nlogn), where nnn is the number of elements.
  + **Worst Case**: O(n2)O(n^2)O(n2) if the pivot selection is poor (e.g., choosing the first or last element in a sorted or nearly sorted array). In practice, though, with a good pivot selection strategy (like randomized or median-of-three), the worst case rarely occurs.
  1. Analyze the memory usage of both algorithms and recommend which is more suitable for this problem.

**Merge Sort:**

* **Space Complexity**: O(n)O(n)O(n) due to the extra space used for merging the two halves of the dataset. It requires additional memory for creating temporary sub-arrays during each recursive step.

**Quick Sort:**

* **Space Complexity**: O(log⁡n)O(\log n)O(logn) on average due to the space used by the recursion stack (because the partitioning happens in place).
  + However, if the recursion depth increases due to poor pivot selection, the space complexity can go up to O(n)O(n)O(n) in the worst case.

**Memory Usage in Practice:**

* **Merge Sort**: Consumes more memory since it requires O(n)O(n)O(n) extra space for sorting.
* **Quick Sort**: Generally more memory-efficient with O(log⁡n)O(\log n)O(logn) space usage for the average case. However, the worst-case space complexity can be higher if poor pivoting leads to deep recursion.

**Problem 5: Hybrid Sorting Strategy**

For small datasets, Quick Sort performs well, but for large datasets or nearly sorted data, Merge Sort is often more reliable.

**Tasks:**

* + 1. Design a hybrid sorting algorithm that uses Quick Sort for smaller partitions (below 10,000 elements) and switches to Merge Sort for larger partitions.

**Code**

#include <iostream>

#include <vector>

#include <algorithm>

// Merge Sort function

void merge(std::vector<int>& arr, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

std::vector<int> L(n1), R(n2);

for (int i = 0; i < n1; i++) L[i] = arr[left + i];

for (int i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];

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

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

if (L[i] <= R[j]) arr[k++] = L[i++];

else arr[k++] = R[j++];

}

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

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

}

void mergeSort(std::vector<int>& arr, int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

// Quick Sort function

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) std::swap(arr[++i], arr[j]);

}

std::swap(arr[i + 1], arr[high]);

return i + 1;

}

void quickSort(std::vector<int>& arr, int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

// Hybrid Sort function

void hybridSort(std::vector<int>& arr, int low, int high, int threshold = 10000) {

if (high - low <= threshold) {

quickSort(arr, low, high); // Use Quick Sort for small partitions

} else {

mergeSort(arr, low, high); // Use Merge Sort for large partitions

}

}

int main() {

std::vector<int> arr = {9, 7, 5, 11, 12, 2, 14, 3, 10, 6, 13, 1, 8}; // Example dataset

int n = arr.size();

hybridSort(arr, 0, n - 1);

// Output sorted array

for (int num : arr) {

std::cout << num << " ";

}

return 0;

}

* + 1. Compare its performance with standalone Merge Sort and Quick Sort.

When comparing the performance of the **Hybrid Sort** with **Standalone Merge Sort** and **Quick Sort**, several key factors come into play:

1. **Execution Time**: **Quick Sort** tends to perform faster on smaller datasets or when partitions are balanced, due to its O(nlog⁡n)O(n \log n)O(nlogn) average time complexity. However, it can degrade to O(n2)O(n^2)O(n2) in the worst case. **Merge Sort** provides consistent O(nlog⁡n)O(n \log n)O(nlogn) performance but requires additional time for merging. The **Hybrid Sort** combines the strengths of both: it uses **Quick Sort** for smaller partitions (which are faster) and **Merge Sort** for larger ones, leading to improved overall performance, especially on large or nearly sorted datasets.
2. **Memory Usage**: **Quick Sort** is memory-efficient with its in-place sorting, requiring only O(log⁡n)O(\log n)O(logn) memory for recursion. **Merge Sort** requires O(n)O(n)O(n) extra memory for its temporary arrays used during the merge process. **Hybrid Sort** has similar memory usage to **Merge Sort** when dealing with large datasets but benefits from the lower memory requirement of **Quick Sort** for smaller partitions.

Overall, **Hybrid Sort** typically offers better performance by adapting to the dataset's characteristics, providing the speed of **Quick Sort** for smaller partitions and the stability of **Merge Sort** for larger ones.