|  |
| --- |
| **Department of Computer and Software Engineering – ITU** |
| **SE200T: Data Structures & Algorithms** |

|  |  |
| --- | --- |
| **Course Instructor: Usama Bin Shakeel** | **Dated: 11th Sep 2024** |
| **Teaching Assistant: Zainab, Sadia & Ryan** | **Semester: Fall 2024** |
| **Session: 2023-2027** | **Batch: BSSE2023B** |

# **Assignment 4. Analyzing Time and Space Complexities**

|  |  |  |
| --- | --- | --- |
| **Name** | **Roll number** | **Obtained Marks/35** |
| **Mukarram Raza** | **BSSE-23029-B** |  |

Checked on: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Summary Table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Time Complexity (Best) | Time Complexity (Average) | Time Complexity (Worst) | Heap Space Complexity |
| Bubble Sort | O(n) | O(n²) | O(n²) | O(1) |
| Quick Sort | O(n log n) | O(n log n) | O(n²) | O(log n) (avg), O(n) (worst) |
| Selection Sort | O(n²) | O(n²) | O(n²) | O(1) |
| Merge Sort | O(n log n) | O(n log n) | O(n log n) | O(n) |
| Radix Sort | O(nk) | O(nk) | O(nk) | O(n + k) |
| Count Sort | O(n + k) | O(n + k) | O(n + k) | O(n + k) |
| Bogo Sort | O((n!)n) | O((n!)n) | O((n!)n) | O(1) |
| Bozo Sort | O((n!)n) | O((n!)n) | O((n!)n) | O(1) |

1. Bubble Sort

For 1D array:

|  |  |  |  |
| --- | --- | --- | --- |
| Lines | Cost | Repetition  (Best/Average/Worst) | Total  (Best/Average/Worst) |
| Time Complexity: |  |  |  |
| for (int i = 0; i < n-1; ++i) | 1 | O(1) / O(n) / O(n) | O(1) / O(n) / O(n) |
| for (int j = 0; j < n-i-1; ++j) | 1 | O(n) / O(n) / O(n) | O(n) / O(n²) / O(n²) |
| if (array[j] > array[j+1]) | 1 | O(n) / O(n) / O(n) | O(n) / O(n²) / O(n²) |
| swap(array[j], array[j+1]) | 1 | O(1) / O(n) / O(n) | O(1) / O(n²) / O(n²) |
| Total: |  |  | O(n) / O(n²) / O(n²) |
| Space Complexity: |  |  |  |
| No additional dynamic memory | 1 | O(1) / O(1) / O(1) | O(1) |
| Total: |  |  | O(1) |

Code:

template <typename T> void MyArray<T>::bubbleSort(int row\_num) {

// Check if the given row number is within the valid range

if (row\_num >= 0 && row\_num < row\_size) {

// Get the number of elements in the current row

int n = col\_size[row\_num];

// Perform the bubble sort algorithm on the current row

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

// Compare each pair of adjacent elements in the current row

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

// If the current element is greater than the next element

if (array[row\_num][j] > array[row\_num][j + 1]) {

// Swap the elements

// Create a temporary variable to hold the value of the current element

T temp = array[row\_num][j];

// Assign the value of the next element to the current element

array[row\_num][j] = array[row\_num][j + 1];

// Assign the value of the temporary variable to the next element

array[row\_num][j + 1] = temp;

}

}

}

}

else {

cout << "Row number is not valid" << endl;

}

}

2. Quick Sort

For 1D array:

|  |  |  |  |
| --- | --- | --- | --- |
| Lines | Cost | Repetition  (Best/Average/Worst) | Total  (Best/Average/Worst) |
| Time Complexity: |  |  |  |
| if (left < right) | 1 | O(log n) / O(log n) / O(n) | O(log n) / O(log n) / O(n) |
| int pivotIndex = partition() | 1 | O(log n) / O(log n) / O(n) | O(log n) / O(log n) / O(n²) |
| quick\_Sort(array, left, pivotIndex - 1) | 1 | O(log n) / O(log n) / O(n) | O(log n) / O(log n) / O(n²) |
| quick\_Sort(array, pivotIndex + 1, right) | 1 | O(log n) / O(log n) / O(n) | O(log n) / O(log n) / O(n²) |
| Total: |  |  | O(n log n) / O(n log n) / O(n²) |
| Space Complexity: |  |  |  |
| partition()  recursion stack space | 1 | O(log n) / O(log n) / O(n) | O(log n) / O(log n) / O(n) |
| Total: |  |  | O(log n) / O(log n) / O(n) |

Code:

// Partition function used by quicksort

template <typename T> int partition(T \*array, int left, int right) {

// Take the last element as the pivot

T pivot = array[right];

// Initialize i as the index of the smaller element

int i = left - 1;

// Iterate through the array from left to right

for (int j = left; j < right; ++j) {

// If the current element is smaller than or equal to the pivot

if (array[j] <= pivot) {

// Increment i (index of smaller element)

i++;

// Swap the smaller element with the current element

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

}

}

// Place the pivot element at its correct position

swap(array[i + 1], array[right]);

// Return the index of the pivot element

return i + 1;

}

// Quick sort algorithm

template <typename T> void quick\_Sort(T \*array, int left, int right) {

// Base case: if the subarray has one or zero elements, it's already sorted

if (left < right) {

// Find the pivot index

int pivotIndex = partition(array, left, right);

// Recursively sort the subarrays on the left and right of the pivot

quick\_Sort(array, left, pivotIndex - 1);

quick\_Sort(array, pivotIndex + 1, right);

}

}

3. Selection Sort

For 1D array:

|  |  |  |  |
| --- | --- | --- | --- |
| Lines | Cost | Repetition  (Best/Average/Worst) | Total  (Best/Average/Worst) |
| Time Complexity: |  |  |  |
| for (int i = 0; i < n - 1; ++i) | 1 | O(n) / O(n) / O(n) | O(n) |
| for (int j = i + 1; j < n; ++j) | 1 | O(n) / O(n) / O(n) | O(n²) |
| if (array[j] < array[min]) | 1 | O(n) / O(n) / O(n) | O(n²) |
| swap(array[i], array[min]) | 1 | O(n) / O(n) / O(n) | O(n) |
| Total: |  |  | O(n²) |
| Space Complexity: |  |  |  |
| No additional dynamic memory | 1 | O(1) | O(1) / O(1) / O(1) |
| Total: |  |  | O(1) / O(1) / O(1) |

Code:

template <typename T> void MyArray<T>::selectionSort1D(T\*& array) {

int n = d0;

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

int min = i;

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

if (array[j] < array[min])

min = j;

}

T temp = array[i];

array[i] = array[min];

array[min] = temp;

}

}

4. Merge Sort

For 1D array:

|  |  |  |  |
| --- | --- | --- | --- |
| Lines | Cost | Repetition  (Best/Average/Worst) | Total  (Best/Average/Worst) |
| Time Complexity: |  |  |  |
| if (left < right) | 1 | O(log n) | O(n) |
| int mid = (left + right) / 2 | 1 | O(log n) | O(n²) |
| mergeSortRecursion(array, left, mid) | 1 | O(log n) | O(n²) |
| mergeSortRecursion(array, mid+1, right) | 1 | O(log n) | O(n) |
| merge(array, left, mid, right) | 1 | O(n) | O(n log n) |
| Total: |  |  | O(n log n) |
| Space Complexity: |  |  |  |
| Temporary arrays L and R | 1 | O(n) / O(n) / O(n) | O(n) |
| Recursion stack space | 1 | O(log n) / O(log n) / O(log n) | O(log n) |
| Total: |  | \*the max at once | O(n) |

Code:

template <typename T> void MyArray<T>::merge(T\*& array, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

T\* L = new T[n1];

T\* R = new T[n2];

for (int i = 0; i < n1; i++)

L[i] = array[left + i];

for (int i = 0; i < n2; i++)

R[i] = array[mid + 1 + i];

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

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

if (L[i] <= R[j]) {

array[k] = L[i];

i++;

} else {

array[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

array[k] = L[i];

i++;

k++;

}

while (j < n2) {

array[k] = R[j];

j++;

k++;

}

delete[] L;

delete[] R;

}

template <typename T> void MyArray<T>::mergeSortRecursion(T\*& array, int left, int right) {

if (left < right) {

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

mergeSortRecursion(array, left, mid);

mergeSortRecursion(array, mid + 1, right);

merge(array, left, mid, right);

}

}

5. Radix Sort

For 1D array:

|  |  |  |  |
| --- | --- | --- | --- |
| Lines | Cost | Repetition  (Best/Average/Worst) | Total  (Best/Average/Worst) |
| Time Complexity: |  |  |  |
| for (T bit = 1; bit; bit <<= 1) | 1 | O(k) | O(k) |
| for (size\_t k = 0; k < n; ++k) | 1 | O(n) | O(nk) |
| if (array[k] & bit) | 1 | O(n) | O(nk) |
| queue0[count0++] = array[k] | 1 | O(n) | O(nk) |
| Total: |  |  | O(nk) |
| Space Complexity: |  |  |  |
| Temporary queues queue0 and queue1 | 1 | O(n) | O(n) |
| Count arrays | 1 | O(k) | O(k) |
| Total: |  |  | O(n + k) |

Code:

template <typename T> void MyArray<T>::radix\_sort(T \*array, size\_t n) {

T \*queue0 = new T[n]; // separate queue for 0-bits

T \*queue1 = new T[n]; // separate queue for 1-bits

size\_t count0 = 0; // count of elements in queue0

size\_t count1 = 0; // count of elements in queue1

for (T bit = 1; bit; bit <<= 1) {

count0 = 0;

count1 = 0;

for (size\_t k = 0; k < n; ++k) {

if (array[k] & bit) {

queue1[count1++] = array[k];

} else {

queue0[count0++] = array[k];

}

}

size\_t k = 0;

for (; k < count0; ++k) {

array[k] = queue0[k];

}

for (size\_t j = count0; j < n; ++j, ++k) {

array[j] = queue1[k - count0];

}

}

delete[] queue0;

delete[] queue1;

}

6. Count Sort

For 1D array:

|  |  |  |  |
| --- | --- | --- | --- |
| Lines | Cost | Repetition  (Best/Average/Worst) | Total  (Best/Average/Worst) |
| Time Complexity: |  |  |  |
| for (int i = 0; i < n; ++i) | 1 | O(n) | O(n) |
| count[(arr[i] / exp) % 10]++ | 1 | O(n) | O(n) |
| for (int i = 1; i < k; ++i) | 1 | O(k) | O(k) |
| output[count[(arr[i] / exp) % 10] - 1] = arr[i] | 1 | O(n) | O(n) |
| Total: |  |  | O(n + k) |
| Space Complexity: |  |  |  |
| Output array | 1 | O(n) | O(n) |
| Count array | 1 | O(k) | O(k) |
| Total: |  |  | O(n + k) |

Code:

template <typename T> void MyArray<T>::countSort(T \*arr, int n, T exp) {

T output[n]; // Output array

int count[10] = {0}; // Count array

// Store count of occurrences in count[]

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

count[(arr[i] / exp) % 10]++;

// Change count[i] so that count[i] now contains actual position of this

// digit in output array

for (int i = 1; i < 10; i++)

count[i] += count[i - 1];

// Build the output array

for (int i = n - 1; i >= 0; i--) {

output[count[(arr[i] / exp) % 10] - 1] = arr[i];

count[(arr[i] / exp) % 10]--;

}

// Copy the output array to arr[], so that arr[] now contains sorted numbers

// according to current digit

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

arr[i] = output[i];

}

7. Bogo Sort

For 1D array:

|  |  |  |  |
| --- | --- | --- | --- |
| Lines | Cost | Repetition  (Best/Average/Worst) | Total  (Best/Average/Worst) |
| Time Complexity: |  |  |  |
| while (!isSorted(array)) | 1 | O((n!)n) | O((n!)n) |
| random\_shuffle(array) | 1 | O((n!)n) | O((n!)n) |
| isSorted(array) | 1 | O((n!)n) | O((n!)n) |
| Total: |  |  | O((n!)n) |
| Space Complexity: |  |  |  |
| No additional dynamic memory | 1 | O(1) / O(1) / O(1) | O(1) |
| Total: |  |  | O(1) |

Code(not self):

template <typename T>bool isSorted(const vector<T>& array) { for (size\_t i = 1; i < array.size(); i++) { if (array[i] < array[i - 1]) { return false; } } return true;}template <typename T>void bogoSort(vector<T>& array) { srand(unsigned(time(0))); while (!isSorted(array)) { random\_shuffle(array.begin(), array.end()); }}

8. Bozo Sort

For 1D array:

|  |  |  |  |
| --- | --- | --- | --- |
| Lines | Cost | Repetition  (Best/Average/Worst) | Total  (Best/Average/Worst) |
| Time Complexity: |  |  |  |
| while (!isSorted(array)) | 1 | O((n!)n) | O((n!)n) |
| random\_swap(array) | 1 | O((n!)n) | O((n!)n) |
| isSorted(array) | 1 | O((n!)n) | O((n!)n) |
| Total: |  |  | O((n!)n) |
| Space Complexity: |  |  |  |
| No additional dynamic memory | 1 | O(1) / O(1) / O(1) | O(1) |
| Total: |  |  | O(1) |

Code(not self):

template <typename T>bool isSorted(const vector<T>& array) { for (size\_t i = 1; i < array.size(); i++) { if (array[i] < array[i - 1]) { return false; } } return true;}

template <typename T>void randomSwap(vector<T>& array) { int i =rand() % array.size(); int j = rand() % array.size(); swap(array[i], array[j]);}

template <typename T>void bozoSort(vector<T>& array) {

srand(unsigned(time(0))); while (!isSorted(array)) {

randomSwap(array); }}