Moldova State University

Faculty of Mathematics and Informatics

Department of Informatics

Fast sortings

Laboratory Report III

Author Ciobanu Stanislav

Group I2302

Date 08.12.23

Table of contents

Task 1

Quick sort description

For array ar = { ar_1, ar_2, …, ar_3} we choose a pivot (basic) element (This element can be randomly chosen or by a strategy). Then we partition the array. All elements, that are smaller, than pivot are placed on the left side, other on the right. Then we recursively repeat this process for these two small partition arrays.

This process repeats until array won't be finally sorted. The complexity of this sort differs according to the chosen pivot element. It can be O(n log n) in best cases and O(n^2) in the worst.

Task 2

Merge sort pseudocode

```
function megeSort(arr)
{
	if (length(arr) <= 1)
	{
		return 1;
	}
	mid = length(arr) / 2;
	left = mergeSort(arr[0 to mid-1]);
	right = mergeSort(arr[mid to end]);
	return merge(left, right);
}

function merge(left, right)
{
	Result = [];
	leftIndex = 0;
	rightIndex = 0;
	while ((leftIndex < length(left)) and (rightIndex < length(right)))
	{
		if (left[leftIndex] < right[rightIndex])
		{
			Result.append(left[leftIndex]);
			leftIndex++;
		}
		else
		{
```

```
                Result.append(right[rightIndex]);
                rightIndex++;
            }

        }
        result.extend(left[leftIndex]);
        result.extend(right[rightIndex]);
        Return result;
}
```

Heap sort pseudocode

```
function heapSort(arr)
{
        buildMaxHeap(arr); //Convert arr into max heap
        for i from length(arr) - 1 to 1
        {
                swap(arr[0], arr[i]);
                maxHeapify(arr, 0, i);
        }
}

function buildMaxHeap(arr)
{
        for I from length(arr) / 2 - 1 down to 0
        {
                maxHeapify(arr, i, length(arr));
        }
}


function maxHeapify(arr, i, heapSize)
{
        largest = i;
        left = 2 * i + 1;
        right = 2 * i + 2;

        if ((left < heapSize) and (arr[left] > arr[largest]))
        {
```

```
		largest = left;
	}


	if ((left < heapSize) and (arr[right] > arr[largest]))
	{
		largest = right;
	}


	if (largest != i)
	{
		swap(arr[i], arr[largest]);
		maxHepify(arr, largest, heapSize);
	}
}
```

Task 3

Quick sort code

```
int partition(int** arr, int start, int end)
{
	int pivot = *arr[start];

	int count = 0;
	for (int i = start + 1; i <= end; i++) {
		if (*arr[i] <= pivot)
			count++;
	}

	int pivotIndex = start + count;
	swap(*arr[pivotIndex], *arr[start]);

	int i = start, j = end;

	while (i < pivotIndex && j > pivotIndex) {

		while (*arr[i] <= pivot) {
			i++;
		}

		while (*arr[j] > pivot) {
			j--;
		}

		if (i < pivotIndex && j > pivotIndex) {
			swap(*arr[i++], *arr[j--]);
```

```
			}
		}

		return pivotIndex;
}

void QuickSort(int** arr, int start, int end)

{
		if (start >= end)
			return;

		int p = partition(arr, start, end);

		QuickSort(arr, start, p - 1);


		QuickSort(arr, p + 1, end);
}

double StartQuickSort(int** ar, int size)

{
		clock_t c;
		c = clock();

		QuickSort(ar, 0, size - 1);

		return (float)(clock() - c) / 1000;
}
```

Heap sort code

```
void heapify(int** arr, int n, int i)

{
		int largest = i;
		int l = 2 * i + 1;
		int r = 2 * i + 2;

		if (l < n && *arr[l] > *arr[largest])
			largest = l;

		if (r < n && *arr[r] > *arr[largest])
			largest = r;

		if (largest != i)
		{
			Swap(arr[i], arr[largest]);

			heapify(arr, n, largest);
		}
}


double HeapSort(int** arr, int n)

{
		clock_t c;
		c = clock();
```

```
	for (int i = n / 2 - 1; i >= 0; i--)
		heapify(arr, n, i);

	for (int i = n - 1; i > 0; i--)
	{
		swap(arr[0], arr[i]);
		heapify(arr, i, 0);
	}

	return (float)(clock() - c) / 1000;
}
```

Merge sort code

```
void merge(int arr[], int low, int mid, int high)

{
    int n1 = mid - low + 1;
    int n2 = high - mid;

    int* left = (int*)calloc(n1, sizeof(int));
    int* right = (int*)calloc(n2, sizeof(int));

    for (int i = 0; i < n1; ++i) {
        left[i] = arr[low + i];
    }
    for (int i = 0; i < n2; ++i) {
        right[i] = arr[mid + 1 + i];
    }

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

    while (i < n1 && j < n2)

      {
        if (left[i] <= right[j]) {
            arr[k++] = left[i++];
        }
        else {
            arr[k++] = right[j++];
        }
    }

    while (i < n1) {
        arr[k++] = left[i++];
    }
    while (j < n2) {
        arr[k++] = right[j++];
    }
}

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

{
    if (low < high) {
        int mid = low + (high - low) / 2;
        mergesort(arr, low, mid);
        mergesort(arr, mid + 1, high);
        merge(arr, low, mid, high);
    }
}
```

Task 4

Count time

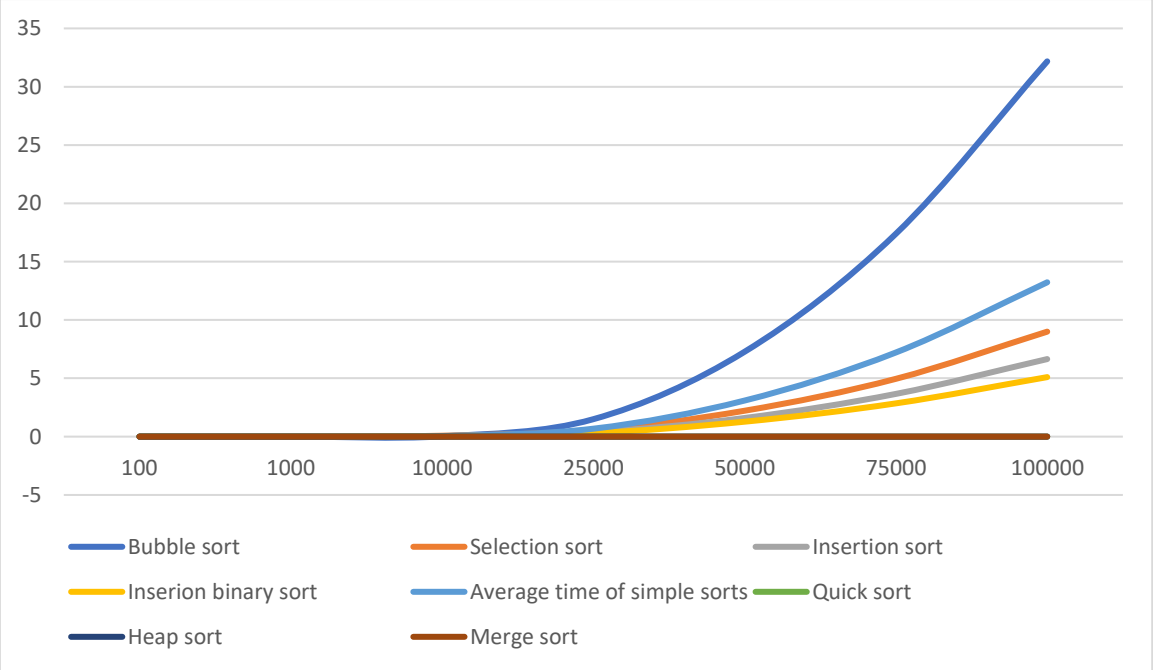
To count time of processing a sort we use this code

```
clock_t c;
c = clock();

// Sort
return (float)(clock() - c) / 1000;
```

Task 5

Analyze and results



	Quick sort	Heap sort	Merge sort
100	0	0	0
1000	0	0	0
10000	0.003	0.004	0.004
25000	0.008	0.01	0.01
50000	0.016	0.021	0.018
75000	0.023	0.033	0.026
100000	0.031	0.041	0.033

1000000	0.404	0.862	0.345
10000000	6.555	14.46	3.55

As we can see, Merge sort is faster than others. Quick sort and Heap sort are also fast, and simple sorting algorithms can not compete with these advanced algorithms.