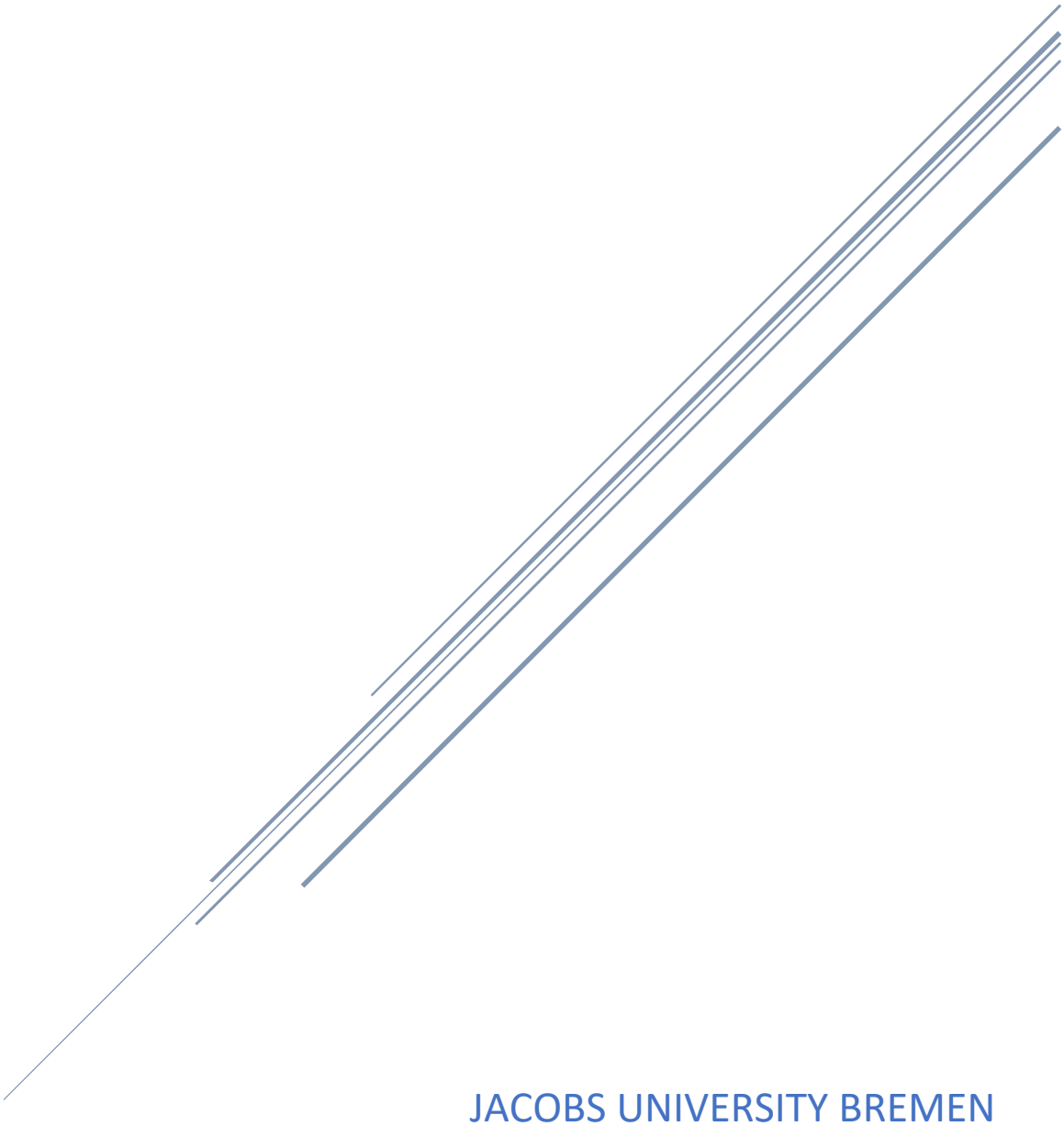
HOMEWORK 4

ALGORITHMS AND DATA STRUCTURES



JACOBS UNIVERSITY BREMEN
Mario Alberto Hernandez Salamanca

Problem 4.1 Bubble Sort & Stable and Adaptive Sorting

a)

```
1. Void bubbleSort(int arr[], int n)
2. {
3.     int i, j;
4.     for (i = 0; i < n-1; i++)
5.
6.         // Last i elements are already in place
7.         for (j = 0; j < n-i-1; j++)
8.             if (arr[j] > arr[j+1]) //compare adjacent numbers
9.                 swap(&arr[j], &arr[j+1]); //swap if arr[j] is bigger
10.  }
11.
```

b)

In Bubble Sort, there will be n-1 comparisons in 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on.

So the total number of comparisons will be,

$$(n-1) + (n-2) + (n-3) + \ldots\ldots + 3 + 2 + 1$$

$$\text{Sum} = \frac{n(n-1)}{2}$$

$$\text{i.e } O(n2)$$

Hence the **time complexity** of Bubble Sort is **$O(n^2)$**.

In best case time complexity is **O(n)** ,the time complexity can occur if the array is already sorted, and that means that no swap occurred and only 1 iteration of n elements

The worst case is if the array is already sorted but in descending order. This means that in the first iteration it would have to look at n elements, then after that it would look n - 1 elements (since the biggest integer is at the end) and so on and so forth till 1 comparison occurs. Therefore the **time complexity in worst case** of Bubble Sort is **$O(n^2)$**.

In average case the time complexity is **$O(n^2)$** because is the same time complexity but still a fraction of a quadratic complexity.

c)

Algorithms that are stable by definition or nature would be Insertion Sort , Merge Sort and Bubble Sort because at the time we make the comparisons the relative position of equivalent elements remain same before and after the sorting.

Heap sort on the contrary is an unstable sorting algorithm at the time of doing the heap the relative position might change depending on the position so you are not sure if it will be the same position.

d)

If the order of the elements to be sorted from an input array matters (or) affects the time complexity of a sorting algorithm, that algorithm is called the "Adaptive" sorting algorithm.

Therefore using this definition we can say that **Insertion sort and Bubble Sort** if we used an almost sorted array they would take a time complexity of **O(n)** so they are adaptive

But **Merge Sort and Heap Sort** are not adaptive because they have the same time complexity, it doesn’t matter if the elements are already sorted, they will almost the same time.

Problem 4.2 Heap Sort

a)

```
1. void max_heapify(int HEAPTIME[], int n, int index)
2. {
3.     int raiz = index; // Initialize raiz as root
4.     int l = 2*index + 1; // left = 2*index + 1
5.     int r = 2*index + 2; // right = 2*index + 2
6.
7.     // If left child is larger than root
8.     if (l < n && HEAPTIME[l] > HEAPTIME[raiz])
9.         raiz = l;
10.
11.      // If right child is larger than raiz so far
12.      if (r < n && HEAPTIME[r] > HEAPTIME[raiz])
13.          raiz = r;
14.
15.      // If raiz is not root
16.      if (raiz != index)
17.      {
18.          swap(HEAPTIME[index], HEAPTIME[raiz]);
19.
20.          // Recursively max_heapify the affected sub-tree
21.          max_heapify(HEAPTIME, n, raiz);
22.      }
23.  }
```

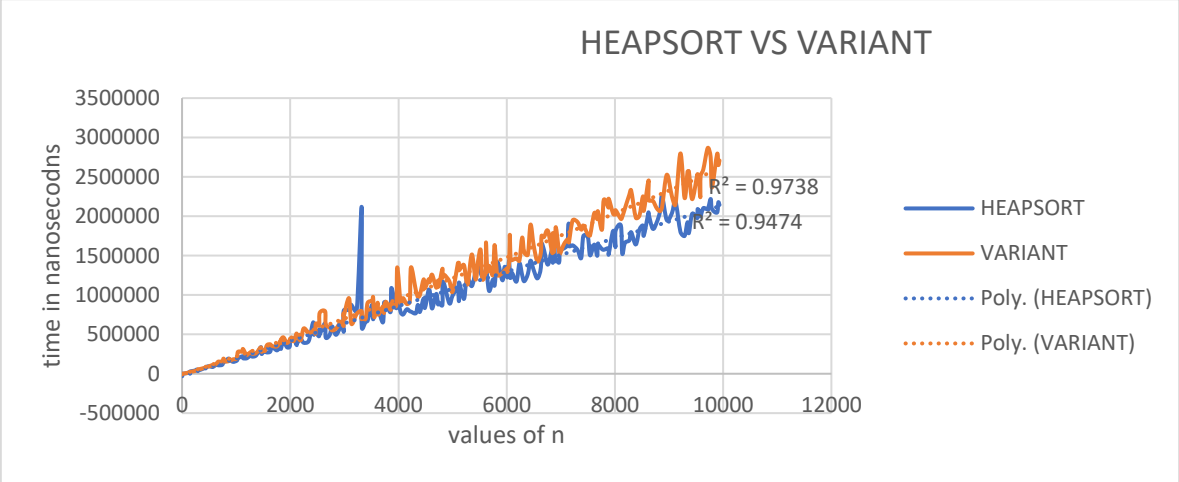
```
1.  void HeapSort(int HEAPTIME[], int n)
2. {
3.     // Build heap (reHEAPTIMEange HEAPTIMEay)
4.     for (int i = n / 2 - 1; i >= 0; i--)
5.         max_heapify(HEAPTIME, n, i);
6.
7.     // One by one extract an element from heap
8.     for (int i=n-1; i>=0; i--)
9.     {
10.          // Move current root to end
11.          swap(HEAPTIME[0], HEAPTIME[i]);
12.
13.          // call max max_heapify on the reduced heap
14.          max_heapify(HEAPTIME, i, 0);
15.      }
16.  }
```

b) Using the same max_heapify function the variant is in the heapsort itself using a new function that I called VARIANT.

```
1. void VARIANT(int HEAPTIME[], int n, int min_indx) {
2.   int raiz = min_indx; // Initialize raiz as root
3.      int l = 2*min_indx + 1; // left = 2*min_indx + 1
4.      int r = 2*min_indx + 2; // right = 2*min_indx + 2
5.      // If we are at the leaf levels, when the new root doesnt have
   children
6.      // means that we cant continue so we have to do fixing step
7.      // if this is not true at least the root have one child
8.      if(l >= n) {
9.           float root = min_indx/2;
10.          int root1 =  floor(root);
11.          if(HEAPTIME[min_indx] > HEAPTIME[root1]){
12.                  swap(HEAPTIME[min_indx],HEAPTIME[root1]);
13.          }
14.          return;
15.      }
16.      // we have three cases when both children exist or only the
   left exist
17.      // because we know that is a max_heap
18.      // so we fill it starting from the left child we took care of
   the base case before
19.      if (l < n && r < n && HEAPTIME[l] > HEAPTIME[r]) {
20.          raiz = l;
21.      }
22.      else if(l < n && r < n && HEAPTIME[l] < HEAPTIME[r]){
23.          raiz = r;
24.      }
25.      else {
26.          raiz = l;
27.      }
28.      //we check the bigger value between the left and right
   children, the root in this case is usually always smaller  than
29.      // both left and right child, but the max-heap is mantain
30.      swap(HEAPTIME[min_indx], HEAPTIME[raiz]);
31.      VARIANT(HEAPTIME, n, raiz);
32.    }
33.
34.
```

```
35.  // main function to do heap sort
36.  void HeapSort(int HEAPTIME[], int n)
37.  {
38.      // Build max heap (rearrange array)
39.      for (int i = n / 2 - 1; i >= 0; i--)
40.          max_heapify(HEAPTIME, n, i);
41.
42.      // One by one extract an element from heap
43.      for (int i=n-1; i>0; i--)
44.      {
45.
46.          // Move current root to end
47.          swap(HEAPTIME[0], HEAPTIME[i]);
48.          // call max max_heapify on the reduced heap
49.         VARIANT(HEAPTIME, i, 0);
50.      }
51.  }
```

c)



the algorithms have the same tendency they look really similar as n growth but the HEAPSORT is still faster than the VARIANT this can be true because maybe I did a not optimal solution, I think that if I have the capacity of doing a more optimal algorithm using the same logic the VARIANT would be faster because you can skip lots of swaps.