**Task 2: Algorithmic problem solving 2**

**DEFINITION** A function *t (n)* is said to be in *O(g(n))*, denoted *t (n)* € *O(g(n)),*if *t (n)* is bounded above by some constant multiple of *g(n)* for all large *n,* i.e., if there exist some positive constant *c* and some nonnegative integer *n*0 such that

***t (n)* ≤ *cg(n)* for all *n* ≥ *n*0*.***

Big oh = F(n)<=C\*g(n)

**Orders of Growth**

**1 log2n n nlog2n n2 n3 2n n!**

**Orders of Growth table**

|  |  |
| --- | --- |
| **Class** | **name** |
| **O(1)** | **constant** |
| **O(log log n)** | **bi-logarithmic or log log n** |
| **O(log n)** | **logarithmic or log n** |
| **O((log n)k) or O(logkn(** | **poly-logarithmic** |
| **O(n)** | **linear** |
| **O(n log n)** | **linear logarithmic or n log n** |
| **O(n2)** | **quadratic** |
| **O(n2 log n)** | **quadratic logarithmic** |
| **O(n3)** | **cubic** |
| **O(2n)** | **base-2 exponential** |
| **O(en)** | **natural exponential** |
| **O(3n)** | **base-3 exponential** |
| **O(n!)** | **factorial** |
| **O(nn)** | **hyper-exponential** |

|  |  |
| --- | --- |
| **Big-O Notation** | **Examples of Algorithms** |
| **O(1)** | **Accessing an array element, Constant loops, Push, Pop, Enqueue (if there is a tail reference), Dequeue** |
| **O(log(n))** | **Binary search** |
| **O(n)** | **Linear search, Summing a 1D-array** |
| **O(n log(n))** | **Heap sort, Quick sort (average-case), Merge sort** |
| **O(n2)** | **Selection sort, Insertion sort, Bubble sort, Summing a 2D-array of size n\*n** |
| **O(n3)** | **Matrix multiplication** |
| **O(2n)** | **Towers of Hanoi, Recursive Fibonacci, Finding the (exact) solution to the traveling salesman problem (TSP) using dynamic programming** |
| **O(n!)** | **Solving the traveling salesman problem via brute-force search** |
| **O(nn)** | **Ackermann function** |

**Ex1**

loglinear complexity

O(nlogn), also known as loglinear complexity, implies that logn operations will occur n times. It’s commonly used in recursive sorting algorithms and binary tree sorting algorithms.

**Coding**

#include <stdio.h>

// lets take a[5] = {32, 45, 67, 2, 7} as the array to be sorted.

**// merge sort function**

void mergeSort(int a[], int p, int r)

{

int q;

if(p < r)

{

q = (p + r) / 2;

mergeSort(a, p, q);

mergeSort(a, q+1, r);

merge(a, p, q, r);

}

}

// function to merge the subarrays

void merge(int a[], int p, int q, int r)

{

int b[5]; //same size of a[]

int i, j, k;

k = 0;

i = p;

j = q + 1;

while(i <= q && j <= r)

{

if(a[i] < a[j])

{

b[k++] = a[i++]; // same as b[k]=a[i]; k++; i++;

}

else

{

b[k++] = a[j++];

}

}

while(i <= q)

{

b[k++] = a[i++];

}

while(j <= r)

{

b[k++] = a[j++];

}

for(i=r; i >= p; i--)

{

a[i] = b[--k]; // copying back the sorted list to a[]

}

}

// function to print the array

void printArray(int a[], int size)

{

int i;

for (i=0; i < size; i++)

{

printf("%d ", a[i]);

}

printf("\n");

}

int main()

{

int arr[] = {32, 45, 67, 2, 7};

int len = sizeof(arr)/sizeof(arr[0]);

printf("Given array: \n");

printArray(arr, len);

// calling merge sort

mergeSort(arr, 0, len - 1);

printf("\nSorted array: \n");

printArray(arr, len);

return 0;

}

### **Complexity Analysis of Merge Sort**

Merge Sort is quite fast, and has a time complexity of O(n\*log n). It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

**Ex 2.1** Sort all the functions below in increasing order of asymptotic (Big O) growth. If some

have the same asymptotic growth, then be sure to indicate that. Note: log means base 2.

5n, 4 logn,4log log n,n4,n1/2log4n, n1/2 log4n,(log n)5logn, nlogn, 5n, 4n4, 44n, 55n, 55n, nn1/5, nn/4,(n/4)n/4

**Solution:** 4 log log n < 4 log n < n1/2 log4n < 5n < n4 < (log n)5logn < nlogn < nn1/5< 5n < 55n < (n/4)n/4 < nn/4 < 4n4 < 44n < 55n

**Ex 2.2** Order the functions according to their growth from slowest to fastest growing

6n3, n log6n, 4n, 8n2, log2n, nlog2n, 64, 82n

**Solution:** 64, log2n, 4n, n log6n, nlog2n, 8n2, 6n3, 82n

**Result:**

Thus, Order the functions according to their growth from slowest to fastest growing using Big O has learnt successfully.