# Exam Questions VL01

## What is the time complexity of the following code(big_O_examples.cpp)?

The time Complexity of "foo" in "big_O_examples.cpp" is 2n where n is the size of the vectro "vec".

---

## For each of the following time complexities: $O(1), O(log\, n),\ O(n),\ \ O(n\, log\, n)$ and $O(n^2)$ give an algorithmic example:

### $O(1)$:

```
int add(const int a, const int b){
return a+b;
}
```

Just add two integers and return them.

$O(log\, n)$:

```
struct node* search(struct node* root, int
key){

    // Base Cases: root is null or key is
    // present at root`

    if (root == NULL || root->key == key)

       return root;

    // Key is greater than root's key

    if (root->key < key)
        return search(root->right, key)

    // Key is smaller than root's key

    return search(root->left, key);

}
```

Source: https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/

In a Binary-Tree the height of the tree is $log\, n$ therefore if you search for an Element you just have to follow the right path to get to your Element in $O(log\, n)$ time.

## $O(n)$

```
int retSmallest(const vector<int> &vec){
        int min = INT_MAX;
        for(int i: vec){
                if(min > i) min = i;
        }
        return min;
}
```

This function loops through every Element inside "vec". Therefore it has a runtime of $O(n)$.

## $O(n\ log\ n)$

```
struct Node {
    int data;
    Node* left;
    Node* right;
};

Node* CreateNode(int data)
{
    Node* newNode = new Node();
    if (!newNode) {
        cout << "Memory error\n";
        return NULL;
    }
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}
```

```
Node* InsertNode(Node* root, int data)
{
    // If the tree is empty, assign new node
address to root
    if (root == NULL) {
        root = CreateNode(data);
        return root;
    }

    // Else, do level order traversal until
we
        // find an empty
    // place, i.e. either left child or
right
    // child of some
    // node is pointing to NULL.
    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        Node* temp = q.front();
        q.pop();

        if (temp->left != NULL)
            q.push(temp->left);
        else {
            temp->left = CreateNode(data);
            return root;
        }

        if (temp->right != NULL)
            q.push(temp->right);
        else {
            temp->right = CreateNode(data);
            return root;
        }
    }
}
```

Source:
[https://www.geeksforgeeks.org/insertion-in-a-binary-tree-in-level-order/](https://www.geeksforgeeks.org/insertion-in-a-binary-tree-in-level-order/)

If you create and insert every Element of an Array it has to traverse through the hole Array making it already $O(n)$ and insert has a time complexity of $O(log\, n)$ which makes it $O(n\ log\ n)$
.

$O(n^2)$

```
void printDoubles(const vector<int> &vec){
        for(int a = 0; a < vec.size(); a++){
                for(int b = 0; b <
vec.size(); b++){
                        if(a!=b && vec) cout
<< vec.at(b);
                }
        }
}
```

Checks every Element against each other if there is a double and if there is it prints it out. Checking every Element of "vec" against every Element of "vec" makes it : $n \cdot n = n^2 \Rightarrow O(n^2)$

---

### Describe a systematic approach for solving a problem in a programming interview.

First find an bruteforce example and after this try to improve from there.

---

## Exam Questions VL02

### How can we exploit bit operations for operations with sets?

If we use bit operations we can cast the datatype (whatever it is) to just some bits that we can easily manipulate without caring about the datatype we're looking at.

---

## Explain the difference between arithmetic and logical right shifts for negative signed integers:

If we shift arithemically we divide by 2 regardless of the sign of the value e.g. : -1 shifted by 1 or 2 bits is still -1, but if we do it logically we will fill every "new" bit with a 0, even the MSB(most significant bit), which is used to determin the sign of our integer, so a negative number becomes positve and even worse, because of the two's complement we get a completly unrelated number out of this.

---

## Explain what the following code does (n & (n-1)) == 0:

The Code checks if $n$ is either a 0 or a power of 2.

If it is $n = 0b0000\_0000$ then $n - 1 = 0b1111\_1111$, because of two's complement so $n\&(n-1) = 0b0000\_0000$ because they have no bit in common, it is the same for any $2^n$, because it only contains 1 bit(this is defined in the binary numbers). So if you subtract one from that single bit every bit before is flipped and they have nothing in common so the result will also be 0.

---

**Outline an algorithm to count the number of bits that are set to 1 in an integer. (no bitset or built-in functions):**

```
size_t countBits(size_t n){
        size_t sum      = 0;
        size_t a        = 1;
        size_t compare  = 0;

        while(n != compare){
            if(a & n) sum++;
            n>>=(unsigned)1u;
        }

        return sum;
}
```

We create a sum that we will return to count all bits. We also create a variable initialized as 1 so we can compare if the LSB(least significant bit) is set, if so we add one to the sum, then we shift n one to the right(logically as explained before). The loop ends when the given value is equal to 0, therefore no bits are left.

---

## Outline an algorithm to reverse the bits in an unsigned integer.

The easiest, but not fastest way, is to just put them into a stack and pop back every Element.

You can also do this faster by just using ~ in front of the element. This will flip every bit.

---

**Imagine you have to reorder integers in an array so that the odd entries appear first. How is this problem related to Quicksort's partitioning function?**

Well instead of comparing the numbers for larger or smaller you can just check if the modulo 2 is qual to two or not and then sort that way.

## Exam Questions VL03

**Outline an algorithm to convert a string to an unsigned integer (no negative numbers).**

```
unsigned int StringToInt(string s){
    unsigned int sum = 0;
    unsigned int count = 1;
    for(int i = s.length() - 1; i >= 0; i--)
{
        char c = s[i];
        switch(c){
            case '1': sum += count;
                    break;
            case '2': sum += 2 * count;
                    break;
            case '3': sum += 3 * count;
                    break;
            case '4': sum += 4 * count;
                    break;
            case '5': sum += 5 * count;
                    break;
            case '6': sum += 6 * count;
                    break;
            case '7': sum += 7 * count;
                    break;
            case '8': sum += 8 * count;
                    break;
            case '9': sum += 9 * count;
```

```
                    break;
                default: break;
            }
            count*= 10;
        }
        return sum;
    }
```

We start by looking at the string from behind. We look at the $10^0$ number and add accordingly. We do this until the end of the string and multiply count everytime by 10 so we have the right power of 10 when adding. If the string is 0 at some point we just skip, but still multiply by 10. In the end we return the sum of everything.

---

## Outline an algorithm to convert an unsigned integer to a string.

```
char returnChar(uint32_t c){
    switch(c){
        case 1: return '1';

        case 2: return '2';

        case 3: return '3';

        case 4: return '4';

        case 5: return '5';

        case 6: return '6';

        case 7: return '7';

        case 8: return '8';

        case 9: return '9';

        case 0: return '0';
        default: cerr << "this shouldn't
happen\n";
                 return ' ';
    }
}

string intToString(uint32_t t){
    if(t == 0) return "0";
    string sum = "";
    uint32_t divider = 1000000000;
    while(t / divider == 0){
        t %= divider;
        divider /= 10;
    }

    sum += returnChar(t / divider);

    while(divider > 0){
        t %= divider;
        divider /= 10;
        sum += returnChar(t / divider);
    }

    return sum;
}
```

returnChar could also be inside of intToString, but it is way more easy to look at this way. First the maximum unsigned int (for 32bit) is 4294967295.
If we want to extract the '4' we have to divide by 10^9 which is exactly one tenth of the highest power of ten accepted by 32bit unsigned integer. In the algorithmen shown we extract every 0 before so don't add them to the number. After this we add the first number to the string and continue. We extract by always losing the highest power of ten by just simply doing t %= divider this way everything except the highest power of ten will stay inside ten.
We then divide the divider by 10 so we can

continue until we reach 10^0. After we completed the string we will return it.

---

## Explain the characteristics of the character encodings ASCII, Extended ASCII, UTF-8, UTF-16 and UTF-32.

- ASCII only has about 128 characters and only 95 of them are printable
- Extended ACSII includes a full byte therefore 256 different characters and all of them are printable
- UTF-8 is a one to four byte. How much bytes are used is defined in the first byte. It is also backwards compatible to ASCII and has about 1.112.064 characters(according to Wikipedia)
- UTF-16 can variable-sized by using one or two words(where each word is 16 bit long) to encode

- UTF-32 always uses 4 Bytes and is the only one of those four that has a fixed size

---

## What advice can you give to modify strings as quickly as possible?

I would recommend to use C-Style strings, because manipulation of them is faster and they have way less overhead compared to C++-Style strings.

---

## Why are hardcoded regex expressions often much faster?

Using C++ or C functions always comes with an overhead and has edge cases in it which lengthen the runtime. If you just search for a specific pattern and you want to ignore every edge case you can hardcode them and they will be much faster.

---

### Is run-length encoding a good compression method for strings?

Depending on the use, but nowerdays I would say it doesn't matter, because we have so much Memory and Storage it is unamagineable that using compressesion is needed. The only thing I can imagine is that you will have faster compare times on CPU, as you will have less Cache misses which will result in lower waiting time. (You can put more Data into your CPU-Cache)

## Exam Questions VL04

### How to delete a (non-tail) node from a singly linked list without knowing its predecessor in O(1) time?

We just copy the data from the successor Data into the node that has to be deleted. We then point to the next node of the successor and after this we delete the successor.

---

## Explain the runner technique using a linked list example.

You could use two points $p_1, p_2$ where $p_1$ moves to the next Element while $p_2$ moves two each times $p_1$ moves one. When you reach the end with $p_2$ you know $p_1$ is at the middle. ($p_2$ has to start)

---

## If you were to tune the code for speed, how would you implement a stack?

```
struct Node{
        int data;
        Node node *last;
};

class Stack{
        Node head;
        public:
                Stack(){
                        head = NULL;
                }
                push(int n){
                        if(head == NULL){
                                head.data =
n;
                                head->last=
NULL;
                        }else{
                                Node node;
                                node.data =
head->data;
                                node.last=
head->last;
```

```
                                head.data =
n;
                                head.last =
*node;
                        }
                }
                int pop(){
                        if(head != NULL){
                                Node node =
head->last;
                                int tmp =
head->data;
                                delete head;
                                head = node;
                                return tmp;
                        }else return NULL;
                }
}
```

Everytime I add an Element I will link to the last Element before. When pushing an Element we differentiate between an empty stack or pushing a a new Element. If it is empty we create the head if not we will copy everything from head to a new node and then create a new head with the new data pointing to the old information. When we pop we check if the Stack is empty. If not we will create a pointer to the node before and safe the information in a temporary integer. We then delete the head and reasign it to the node we saved earlier. After this we can finally return the data stored in the old head.

---

## How can we use a std::list in C++ as a queue?

In a Queue utilizes the FIFO(first-in first-out). If we only use pop_front() and push_back(data d) we can use it as a queue.

---

## How does a binary tree and a binary search tree differ?

Binary search tree's are sorted where each element left of a node is smaller and every element on the right of a binary search tree is larger. In a Binary Search Tree Each Element has to only appear once, while in a binary tree you can have multiple of one value. Operations on binary tree's are usally slower and the reason for that is that they are unsorted.

---

## Describe the different types of binary tree traversals.

In pre order you print the node you starded from and go to the lift print this and go again to the left until you reach NULL after this you start from the last visited node and go to the right until you reach null again. You print each node upon visit. In order you go to the left until you reach NULL following this you will print the node and go to the right of the last visited node until you get to NULL. You repeat this until you reach NULL for the last right node. In this order every node printed is the one from the left to the right starting by the last left going to the last right node.

---

## How does a trie and a radix tree differ?

The radix tree is more space efficient as it saves string/word parts as information, but only refers to it's parent and is therefore not as versatile as a trie. The trie saves each character of words used and terminates on NULL nodes. For that reason it is not as space efficient but can save more unique strings/words and is in my opinion better fort autocompletion.

---

## Exam Questions VL05

### Describe briefly the heap data structure.

A heap is a priority queue rigid structure which means it has a shape that is very consistant. This shape is that there will be no node in depth x when there is at least one space in depth x - 1. Every child of every node is also smaller as their parent counter part.

---

### How would you find the k longest words in a data stream? (You cannot back up to read an earlier value.)

I would use a minimum heap to store the strings I'm reading in. When every if find a word longer than the smallest element in the heap I would just exchange them by pop() the last element and then push the ne word onto it.

---

## How would you address the problem of merging multiple sorted files that are too large for RAM?

I would solve it by divide and conquer. I start by using a fraction of both arrays (1 of both would also be enough) and merge them together after that delete the first space of both files and then continue until it is completly merged.

---

## What are sorting networks?

Sorting networks can be interpreted as sorting algorithms that designed like logic gates that have constant sorting times because of the way they compare each element.

---

## Why sorting may speed up set operations?

An easy example would be above where it is asked to find the k longest words in a stream. If you're looking to accomplish a specific goal a specific order of your items/elements may be necessary. The example above wants to store the k elements that are the largest ones appearing. If you just add them to a list you always have to look if every word you already have is bigger or smaller as the new one. For comparison the simpler algorithm explained would have a runtime of $O(n \cdot k)$ which is significantly higher than $O(n \cdot log(k))$.

---

## What is a minimal perfect hash?

A minimal perfect hash is when every produced hash has a place without collision. That means you get your key and hash it put it into another list or table and this new list/table is the exact same size of your original table. No collision means if you hash your key another key won't produce the same hash as all the other ones you already produced.

---

## Exam Questions VL06

### What is backtracking?

Backtracking is kinda like brute force, the difference is that when you're backtracking your solutions and if you find one that won't succeed you reject it. It is basically a trimmed version of brute force.

---

### How would you proceed to compute the n-Queens problem with backtracking as fast as possible?

The fastest way to backtrack this problem is to just test every configuration and reject every configuration that won't lead to a solution.

---

**Describe some ideas how to solve the 15-puzzle with backtracking. (Tell me what you think is important.)**

---

**Why is efficient parallelization not trivial in most divide and conquer algorithms.**

Because other algorithms fail where divide and conquer shines. We can, as stated in the name, divide the problem into subproblems where each of the subproblems can be computed by a different thread, which makes it easy multithread problems this way. Also merge them together can be easy as every thread can get their own space of memory and write to that(depending on the algorithmn).

---

**Why may decrease and conquer be a better name choice for algorithms like quickselect or recursive binary search than calling them divide and conquer ?**

Because you reduce the number of possible solutions every step we make or in other words we narrow it down.

---

**What is dynamic programming?**

Dynamic programming is, when we can use something we already computed in another computation later on. For example if you want to backtrack Fibonacci-Numbers we can cache the already calculated ones in an array and can look up if we already solved one of them, so we don't have to calculate each number again and again.

---

**Compare the top-down approach with the bottom-up approach used in dynamic programming.**

For example we can use the Fibonacci approach again. Bottom-up starts from the bottom of our fibonacci numbers as in F[0]= 1 annd F[1] = 1.
If we now search for n-th number we will start computing from the first two to n and save every solution in a vector or array. Top-down is used by searching the n number from F[n] = F[n - 1] + F[n - 2] until we come to F[0]= 1 annd F[1] = 1. We also save every solution once they are computed.

---

**Present briefly a greedy heuristic of your choice.**

The non overlapping scheduling Problem has an heuristic that is easy to understand. You look up the first starting interval. From there you always pick up the closest next interval to the end of your current interval. Do this until no more intervals remain.

## Exam Questions VL07

## Exam Questions VL08

### If you need to find the shortest path between two nodes in a graph where all edges have length 1, which algorithm would you use and why?

I would use Dijkstra to solve this, because it is an simple task and everything else would be overkill.

---

### Assume that you need to find a shortest path in a relatively small graph with arbitrary edge lengths, and you have to have a working algorithm very soon. Which algorithm would you choose and why?

I would use Floyd-Warshall algorithm, because it is easy to implement and will compute a bit faster than Dijkstra. It will find the fastest path in the fastest time.

---

### Why does Dijkstra's algorithm not work with negative edge lengths?

If we try Dijkstra on a Graph with a negative weight it will compute infinity as a Distance for those distances which is wrong.

---

## Exam Questions VL09

**Argue why topological sortings only exist for acyclic graphs.**

Well if there is a cycle inside the directed Graph and you somehow reach the cycle(which has to happen sometime). You will enter the cycle and look for and end. You will be a dog chasing his own tail, as there is no end to it and you're stuck in the loop without ever leaving it.

---

**Briefy describe how you approached and solved Project Euler Problem 1.**

Well I remember that is is solvable by using inclusion exclusion and if you for:
$x = 1000, N := \{3, 5\}$ and multiples of both
$3, 5 = 3 \cdot 5 = 15$
$\lfloor \frac{x}{N} \rfloor \Leftrightarrow \lfloor \frac{1000}{3} \rfloor = 333, \lfloor \frac{1000}{5} \rfloor = 200, \lfloor \frac{1000}{15} \rfloor = 66$

$\sum_{k=1}^{i} a \cdot k = a \cdot \frac{i(i+1)}{2}$
$\Rightarrow 3 \cdot \frac{333(333+1)}{2} + 5 \cdot \frac{200(200+1)}{2} - 15 \cdot \frac{66(66+1)}{2}$

$= 234168$

---

## Exam Questions VL10

### Describe the general idea of the algorithm by Edmonds & Karp in your own words.

You start with an empty sum of flow. Everytime you find a new flow from starting note to destination(target) node you will add it to the sum. You iterate through every possible flow and adjust the remaining flows accordingly to the flow you already added(subtract on the value of the edges).
You do this until there is no more flow to be found.

---

### Describe your algorithm that solves the Problem $100$ ("$3n + 1$"): what algorithmic

## techniques do you use?

My algorithm uses backtracking and saves every solution to a hashmap so we can lookup every solution we already calculated, so we don't have to calculate them over and over again.

---

## Which algorithm do you use for Problem 459 ("Graph Connectivity"), and why?

First we read in n, therefore we are reading in n cases. Next we read in the first node and build up a array in which we store every character beginning by the first letter, we name it parent. In parallel we construct another array in which we fill with 0, we name it rank. We also initialize a value where we save our disjointed Sets, we name it DJSets.

We loop through every node and the edges. If we find a edge we unite them, but first look if the edge already exists. For every new edge we count down on DJSets. When we finish we will output DJSets as these has to be the number of our disjointed sets.

## Exam Questions VL11

## Exam Questions VL12

### Describe a kernel of the k-VertexCover-Problem: figure out which nodes have to (and which don't have to) be in the VertexCover.

---

### Design an algorithm that constructs a minimal vertex cover for graphs where all nodes have degree at most 2. Your algorithm should run in time $O(n^c)$ where $n$ is the number of nodes in the graph, and $c$ is a

**constant (bonus point if you achieve $c = 1$). Prove the correctness and the runtime of your algorithm.**

Let A be list where you store every node inside the minimum Vertex Cover. First you look at the size of $|V|$. If $|V|$ is odd $\Rightarrow |A| = \frac{|V|-1}{2}$ if $|V|$ is even $\Rightarrow |A| = \frac{|V|}{2}$. A graph where ever node has a degree of 2 at most, has to be either a circle or a line, where there is only one path. In a circle every node has degree 2 and in a line the first and the last node have degree 1 and the rest has degree 2.

Algorithmn:

```
1. if $|V|\%2 = 1$:
   if first node.degree() = 1:
   for i in $|V|(i = \{1, 2, \ldots, |V|\}$
   if $i\%2 = 0$
   $A$ add node.at(i)
   else:
```

```
for i in $|V|(i = \{1, 2, \ldots, |V|\}$
if $i\%2 = 1$
$A$ add node.at(i)
else:
for i in $|V|(i = \{1, 2, \ldots, |V|\}$
if $i\%2 = 1$
$A$ add node.at(i)
```

```
// our index here starts at 1
// i%2 is for modulo
```

The only if where it matters that $i\%2 = 0$ is the first one. On all other cases it is only necessary to add each second node. This has Time Complexity of $O(n)$. This works because, if the size of the vertices determines which Vertex Cover is the minimal one.

Case odd:
The minimal Vertex Cover of an odd line or circle is adding every second vertex, by starting at the second vertex, because if you

start by the first one you have$\lceil \frac{|V|}{2} \rceil + 1$ vertices and the Complement of that has $\lceil \frac{|V|}{2} \rceil - 1$ because the first will always also add the last and the Complement always will have those in between.

Case even:
In an even case it doesn't matter if it is a line or a circle. You can just add every second vertex. You can either start by the first or the second. In a Circle it works, because every vertex has degree 2 and if you add one you will also add two edges so you only need $\frac{|V|}{2}$ vertices. In a line it is basically the same just by starting at the first you will include the start but exclude the ending one and vice versa.

---

**Describe your approach at this weeks programming exercise (Problem 1773-A).**

How does your algorithm and a pair of valid permutations?

While looking at the Problem I discovered for every $n > 3$ has a solution and that it can be found by following a "path" described by my algorithmn. In short my algorithm looks at the first entry($a[0]$) then tries the smallest possible combination $i$.(Exluding: $i \neq 0 \ \wedge \ i \neq a[0]$). Then writing $q[a[i]] = i$. (In the first case hast $i$ to be $i \in \{0, 1, 2\}$), then it writes to $p[i] = 0$. After the first step we remember the value $i$ and look for which $j$ is $i = a[j]$(I use a Hashmap for faster Index Recovery) to find the next entry we will work on. We now continue until we reach the end. If we somehow land on a number we already accessed this way we will skip to the next entry we haven't touched yet and continue from there. I also have special cases for $n < 2$ which is impossible, $n == 3$ where I hardcoded every case and for $a = \{1, 2, \ldots, n\}$.