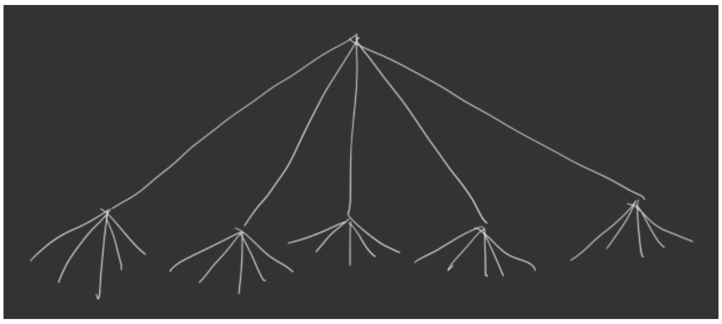
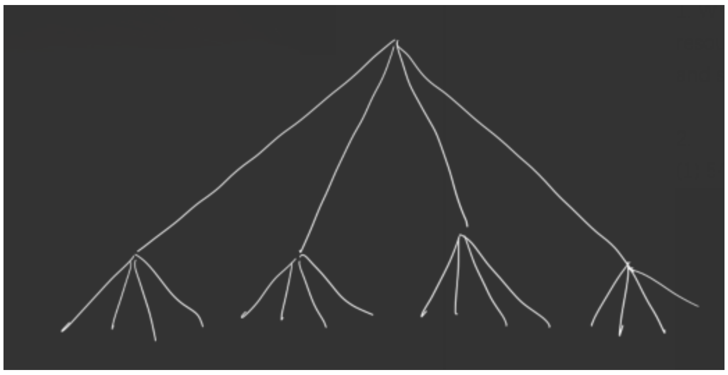
1. There would be several ways to solve a problem. When measuring the advantages or disadvantages of these algorithms, the efficiency and the resources consumed are two important parts. To evaluate these two parts, time-complexity and space-complexity are used. Time-complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Space-complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run a function of the length of the input. So they are both important while doing algorithm analysis.

2.

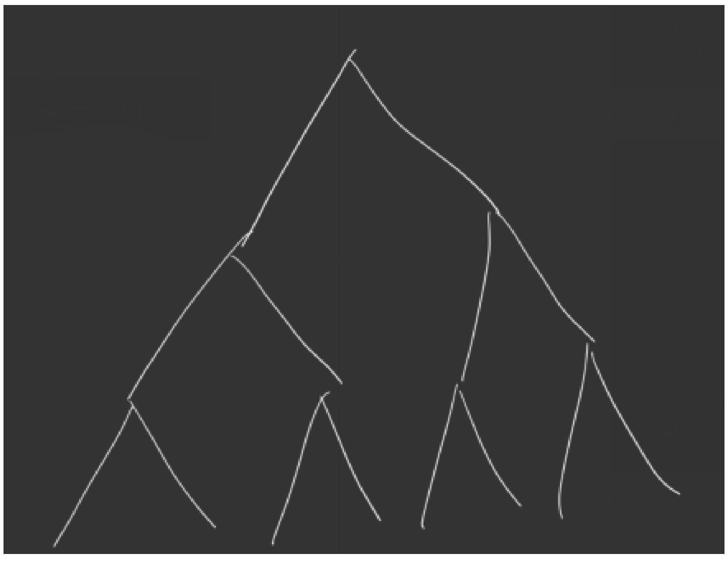
(1) 5^2



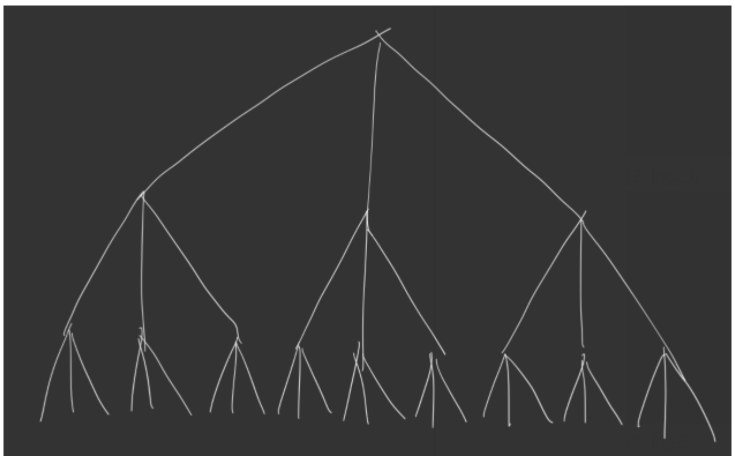
(2) $\log_4 16$



(3) $\log_2 8$



(4) $\log_3 27$



3.

Order of growth is how the execution-time depends on the length of input.

Clearly execution-time is linearly depending on the length of the array.

Order-of-growth will help us to compute the running-time with ease.

We will ignore the lower order terms, since the lower order terms are relatively insignificant for large input.

We use different notation to describe limiting behavior of a function:

Big-O (Upper Bound): f(n)=O(g(n)) means there exists some constant c such that f(n)<=c·g(n), for large enough n (that is, as $n \to \infty$)

Big-Omega (Lower Bound): f(n)=Ω(g(n)) means there exists some constant c such that f(n)>=c·g(n), for large enough n (that is, as $n \to \infty$)

Big-Theta (“Tight” Bound): f(n)= ϴ(g(n)) means there exists some constants c1 and c2 such that f(n)<=c1(g(n)) and f(n)>=c2(g(n)).

4.

A) In the worst case, “count++” is executed $1/2N^2$-1/2N times. So the time-complexity is $O(N^2)$

$0+1+2+\cdots+(N-1)=N(N-1)/2=1/2N^2-1/2N$

B) In the worst case, “count++” is executed 2N times. So the time-complexity is O(N)

$N+N/2+N/4+N/8+\cdots=2N$

5.

The code samples are shown in the .java file.

For constant 1:

count++;

time-complexity: 1=1~O(1)

For logN:

for(int i = N; i > 0; i /= 2) {count++;}

time-complexity: logN=logN~O(logN)

For N:

for(int i = 0; i < N; i++) {count++;}

time-complexity: N=N~O(N)

For NlogN:
```
        for(int i = 0; i < N; i++) {
                for(int j = N; j > 0; j /= 2) {
                        count++;
                }
        }
```
time-complexity: N*logN=NlogN~O(NlogN)

For N^2:
```
        for(int i = 0; i < N; i++) {
                for(int j = 0; j < N; j++) {
                        count++;
                }
        }
```
time-complexity: N*N=N^2~O(N^2)

For N^3:
```
        for(int i = 0; i < N; i++) {
                for(int j = 0; j < N; j++) {
                        for(int k = 0; k < N; k++) {
                                count++;
                        }
                }
        }
```
time-complexity: N*N*N=N^3~O(N^3)

For 2^N:

a.
```
int sum = 1;
        for(int i = 0; i < N; i++) {
                sum *= 2;
        }
        for(int i = 0; i < sum; i++) {
                count++;
        }
```
time-complexity: 2^N=2^N~O(2^N)

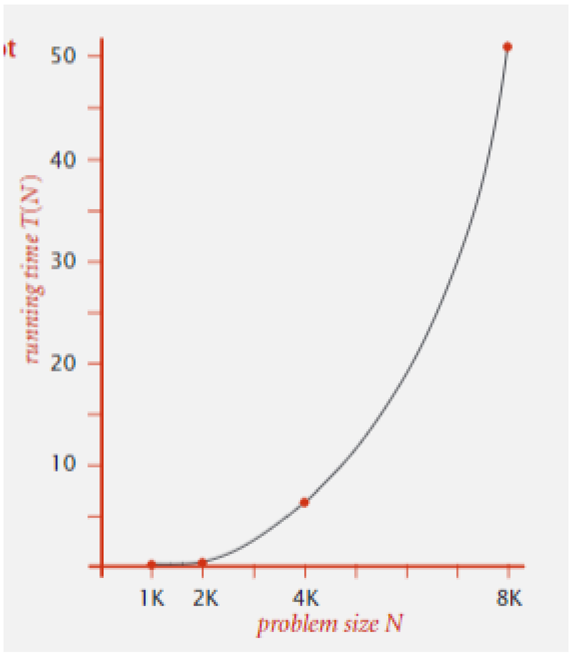
b.
```
public int Fibonacci(int n){
        if(n < 1) return0;
        if(n == 1 || n == 2) return 1;
        return Fibonacci(n-1) + Fibonacci(n – 2);
}
```
time-complexity: O(2^N)

6.

In the first case, which is $1/6N^3+20N+16$, when N gets bigger, $20N+16$ becomes negligible comparing to $1/6N^3$ because they have the lower power. So we can consider $1/6N^3+20N+16$ as $1/6N^3$.

In the second and third case, which is $1/6N^3+100N^{4/3}+56$ and $1/6N^3-1/2N^2+1/3N$, they're similar to the first case, so we can consider them as $1/6N^3$, too.
That's the reason why the results are the same for the three examples.

7.
A)
The y-axis is the time that is used and the x-axis is the size of the data. As we can see in the picture, the quadratic grows fastest, and the linear and linearithmic grows much more slowly. So we should avoid quadratic in algorithms while coding.

B)


As shown in the figure above is the relationship between N and time (seconds).
The input size and the time that is given is in the formula of N=c1*2^x and t=c2*2^(3x). So we can logarithm both of them, then we get x=log(N/c1) and 3x=log(t/c2).
From these two equations we can get 3log(N/c1)=log(t/c2)
Then after leaner regression, we can get t=1.006*10^(-10)*N^2.999.
So when the size is 16,000, the time is 410.8 seconds.

8.
This algorithm compares the sum of three out of N numbers without repetition one by one, so it's Brute-Force algorithm.
The time complexity is the same as that of taking 3 numbers out of N numbers. So it's the same as C(N, 3) which equals (N(N-1)(N-2))/3!.
So the time complexity is O(N^3).