Opgave 1: Rekursiv algoritme

Skriv en rekursiv algoritme som kan tælle antallet af forekomster af et bestemt bogstav i en String.

```
public class bogstavTaeller{
    public static boolean erBogstav(char c){
        return "abcdefghijklmnopqrstuvwxyz".indexOf(c) !=-1;
    }

    public static int antalBogstaver(String str, int i){
        //Base case: Hvis vi har nået slutningen af strengen
        if (i < 0){
            return 0;
        }

        //Rekursiv case: Tæl bogstavet på den aktuelle position
        // kald rekursivt for resten af strengen
        return (erBogstav(str.charAt(i)) ? 1: 0) + antalBogstaver(str, i-1);
    }

    public static void main(String[] args) {
        String str = "banana";
        int antalBogstaver = antalBogstaver(str, str.length()-1);
        System.out.println("Antal bogstaver i \"" + str + "\":" +antalBogstaver);
    }
}
```

NB: Da jeg orienterede mig i opgaven igen, gik det op for mig at der med ”banana” skulle være returneret 3, da der, selvom der optræder 6 bogstaver, kun findes 3 forskellige i ordet. Det havde jeg overset – men dette kunne have været løst ved at lave en if statement der kun talte et bogstav hvis ikke det allerede var blevet talt en gang.

Opgave 2: Tidskompleksitet

Der gives en algoritme hvori der indgår to for-loops. Angiv tidskompleksiteten i store-o notation:

Det ses at x og y initialiseres og tager konstant tid O(1).

Der laves en løkkeanalyse for løkke 1:

Ydre løkke: Kører fra i til N, og det noteres at i+= i ikke bidrager til tidskompleksiteten. Bidrager altså lineært O(N),

Midterste løkke: Kører fra j til logaritmen af N opløftet i 2. Bidrager altså med O(N) * O(log(N)^2.

Inderste: Kører fra k til denne er mindre end, eller lig med, kvadratroden af N, og bidrager derfor med O(√(N)).

Samlet: O(N) * O(log(N)^2) * O(√(N))

Løkkeanalyse for løkke 2:

Itererer N*√(N) gange, k++ og y++ bidrager ikke til den asymptotiske tidskompleksitet da de udføres et konstant antal gange per iteration. Bidrager med O(N*sqrt(N))

Opgave 3: Tilføj funktionalitet til CriticalPath

Tilføj funktionalitet til udleveret kode, funktionaliteten skal finde den aktivitet som har det største slæk (slack), og tillige angive varigheden af slækket.

Her ville jeg have kigget på index, og udplukket det tredje element for hver aktivitet, altså den tilsvarende tidsforbruget. Herefter ville jeg have trukket det højeste fra det laveste tidsforbrug for hinanden, og dermed fået differencen. Så ville jeg have sammenholdt de forksellige aktiviteter for at se hvilken af dem som havde den højeste slack.

Opgave 4: Binære træ

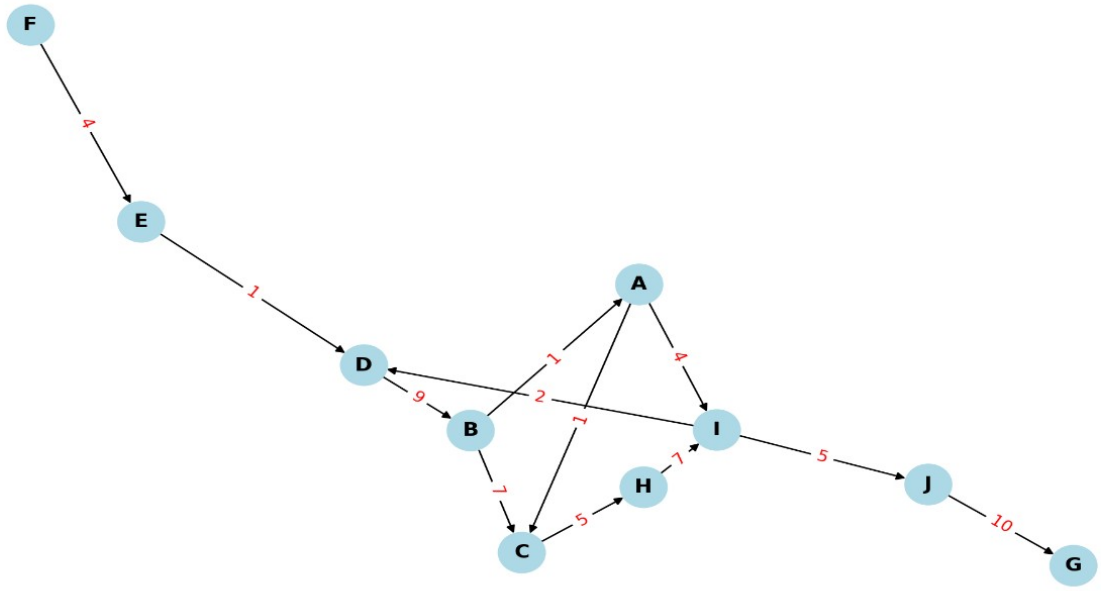
Skriv en metode som kan returnere fra roden til en bestemt node i det binære søgetræ. Kaldet med værdien 10 vil det givne træ returnere ”45 15 10”, og kaldt med 50 vil det returnere ”45 79 55 50”. Hvis ikke værdien findes i træet bør der gives en fejlmelding.

Her ville jeg have implementeret traversering, så jeg havde mulighed for at pointe til en specifik knude, og derefter returnere subtræet derned til.

Opgave 5: Dikjstra

1) **Traverser grafen med Dijkstras algoritme under anvelde af startkonfigurationen**

   Dikjstra er en grådig algoritme, som i hvert stadie fokuserer på det der ser ud til at være den bedste løsning. I hvert stadie vælger algoritmen den node med den korteste distance, dette gentager den til alle mulighederne for en node er undersøgt. Herefter går den videre til næste node.
   Man starter med at sætte afstanden til F på 0, og sætter derefter de andre afstande til uendelig. Nu vælger man knuden med kortest afstand fra F (Her E), og markerer denne knude som besøgt, herfra regner man nu afstand. Dette gøres til man har været alle knuderne igennem.



V	Known	Dv	Pv
F	T	0	0
E	T	4	F
D	T	1	E
B	T	9	D
A	T	1	B
C	T	1	A
H	T	5	C
I	T	7	H
J	T	5	I
G	T	10	J

2) Etabler et MST ved at bruge Kruskal.

Minimum spanning tree er en måde at forbinde alle noder med minimal omkostning, dette er noget som kan gøres med directed graphs. Gøres dette med Kruskal finder man MST ved at sortere alle kanter efter vægt og tilføje dem en ad gangen, startende med den mindste. Kun kanter der forbinder uafhængige komponenter tilføjes, fordi cyklusser ikke er tilladt. Algoritmen

fortsætter indtil alle hjørner er forbundet. Kruskal er ret effektiv og nem at implementere, særligt i grafer der ikke har så mange kanter.

Den give graf bliver her hurtigt cyklisk, men da man med Kruskal ikke er forpligtet til, at kanterne skal være forbundet til det allerede eksisterende træ, men blot skal tage kanterne med lavest værdi ses:

Kant	Vægt	Action
FE	4	Accepted
ED	1	Accepted
DB	9	Rejected
DI	2	Accepted
IH	7	Rejected
IA	4	Accepted
AB	1	Accepted
AC	1	Accepted
CH	5	Accepted
IJ	5	Accepted
JG	10	Accepted

Samlede vægt: 4+1+2+4+1+1+5+5+10 = 33

NB: Dette virker højt, og hvis jeg skal forstå unidirektionelt som først antaget, så bliver den endnu højere, da man så kun har mulighed for at følge pilene den vej de peger, dette ville nemlig give os 43.

Opgave 6: Hashtable

Der gives en tabel, hvor de første mange indsættelser hasher til indeks 3. Skriv et program der simulerer quadratic probing på en tabel med 13 elementer.

```
public class HashTable {
    private static final int TABLE_SIZE = 13;


    //Implementering af quadratic probing
```

```
    private static int quadraticProbe(int index, int i, int[], table){
        while(table[index] != 0){
            index = (index + i * i) % TABLE_SIZE;
            i++;
        }
    return index;
    }
    public static void main(String[] args) {
        int[] table = new int[TABLE_SIZE]; // Squadratic probing

    // Indsæt elemtner i tabellen
    char[] keys = {'H','E','J'};
    for(char key){
        int index = hash(key);
        index = quadraticProbe(index, i, table);
        table[index]=key;
    }
}
}
```

Opgave 7: Prioritetskø

Tabellen repræsenterer kun en prioritetskø hvis alle efterkommere er større end noden, dette læses ikke som tilfældet her, fx ved indeks 15 ses prioritet 18, mens indeks 14 indeholder prioritet 81. Man burde se at dette var i ascenderende rækkefølge.