inlämningsuppgift i algoritmanalys

Gustav Florén
AJ8349

1a)
I en långsammare algoritm kan man iterera över arrayen två gånger för att hitta de par av tal som är lika. Tidskomplexiteten för bifogad algoritm blir O(n^2), det vill säga *polynomiell.* Den inre loopen itererar genom arrayen N gånger, därigenom får den tidskomplexiteten o(n^2).

```
int equalNumbers(byte[] array){
    int count = 0;
    byte temp;
    for (int i = 0; i < array.length ; i++) {
        temp = array[i];
        for (int j = i+1; j < array.length ; j++) {
            if(temp == array[j]){
                count++;
            }
        }
    }
    return count;
}
```

1b)
Funktion för att hitta par: pairs = m-1 ((m-1) + 1) / 2
M = 2, par = 1 2 - 1 * ((2-1) +1 / 2 = 1
M = 3, par = 3 3 - 1 * ((3-1) + 1) / 2 = 3
M = 4, par = 6 4 - 1 * ((4-1) + 1) /2 = 6

1c)
Istället för att iterera över listan för varje index (nästad for-loop) ger en sorterad lista möjligheten att enbart loopa en gång för att nå uppnå förväntat resultat. Algoritmen får tidskomplexiteten - bortsett från konstanter - O(n). I lösningen nedan har jag tre if-satser:
Den första if satsen gör den sista beräkningen som kommer ske,
Den andra if-satsen ökar värdet på variabeln M med ett varje gång värdet i array[i] och array[i+1] har samma värde. Variabeln används i den första och sista if satsen för beräkning av antal par.
Den tredje if-satsen räknar par efter att en sekvens av par i listan är slut.

```java
public class Uppg1c {
  public static int countpairs(int[] sortedarr){
        int N = sortedarr.length;
        int pairs = 0;
        int m = 1;
        for (int i = 0; i < N; i++) {
            if (N <= (i+1)){
                m = ((m - 1) * ((m -1 ) + 1));
                m = m /2;
                pairs += m;
            }
            else if(sortedarr[i] == sortedarr[i + 1]){
                m++;
            }
            else if (sortedarr[i] != sortedarr[i + 1]){
                m = ((m - 1) * ((m -1 ) + 1));
                m = m /2;
                pairs += m;
                m = 1;
            }
        }
        return pairs;
    }
}
```

1d) Den nya lösningen får tidskomplexiteten O(N log N) då när N går mot oändligheten väljer man det större. Detta eftersom algoritmen som implementerades i 1c) gav tidskomplexiteten O(n).

2 Antag att vi har ett hus med N våningar, och anta att ett ägg går sönder om man släpper det från våning F (F ≤ N) eller högre, men att det klarar sig om man släpper det från lägre våningar. Håller ägget när vi släpper det kan vi använda det igen, men går det sönder kan vi inte prova fler gånger med just det ägget. Vi vet vad N är, och ska ta reda på F. (Ungefär bokens 1.4.24–1.4.25.)

a) Antag vi att vi bara har ett ägg. Tänk ut en strategi för att ta reda på F genom att släppa detta ägg ~ F gånger.

Eftersom ägget går sönder när man släpper det från F (F <= N) och vi bara har ett ägg blir det enklast att släppa det en gång för varje våning med första våningen som utgångspunkt, när ägget går sönder subtraherar man ett från F och därigenom får man F.

b) Antag att vi har ett obegränsat antal ägg att prova med. Tänk ut en strategi för att ta reda på F genom att släppa ägg ~ lg N gånger

Eftersom vi var en obegränsad mängd ägg kan man ta reda på F genom att släppa flera ägg även om de går sönder, och vidare eftersom våningarna är sorterade i storleksordning kan binärsökning tillämpas. Genom att tillämpa binärsökning som börjar från toppen och halverar för varje gång (binary search) för att hitta F når vi i snitt en tidskomplexitet på o(log n).

c) Hitta en strategi som är bättre än den i 2b när F är litet i förhållande till N genom att den klarar sig med att släppa ägg ~ 2 lg F gånger

Förklaring: När F är litet i förhållande till N är det rimligt att börja från 1 som på delfråga-A, men eftersom vi →
Ett: Inte bara har 1 ägg som i delfråga-A och..
Två: Vi söker en snabbare algoritm kräver mindre släppta ägg →
kan man använda sig sig av exponentiell sökning där man dubblar värdet varje gång man inte hittar F. När man väl har hittat F har man en “range/intervall” där man kan använda binary search för att klara sig med ~2 lg F släppta ägg
Exempel:
Tänk en byggnad med 500 våningar, där F = 10.
1, ägg går inte sönder
2, ägg går inte sönder
4, ägg går inte sönder
8, ägg går inte sönder
16, ägg går sönder
binär sökning mellan 8 - 16
dela i mitten, 12 går sönder →
halvera igen, 10 går inte sönder.
Svar: Exponentiell + binärsökning