Datastrukturer och algoritmer - laboration 2

Grupp 2 - Mats Högberg, Maxim Goretskyy

Februari 2016

1 Komplexitetsanalys

1.1 Uppgift a

När vi räknar ut komplexiteten för metoden så bryr vi oss inte om hur lång tid de enskilda operationerna tar, utan bara hur algoritmen skalar beroende på antalet punkter. Vi kan förenkla koden till följande:

```
while (poly.length > k * 2) {
    // Something O(1)

    for (int i = 2; i < poly.length - 2; i += 2) {
        // Something O(1)
    }

    // Something O(1)

    for (int i = 0; i < poly.length; i += 2) {
        // Something O(1)
    }

    // Something O(1)
}
```

Vi låter n beteckna antalet punkter i indatan. Den yttersta loopen kommer köras $n - k$ gånger, den första inre loopen kommer köras $n - 2$ gånger och den andra kommer köras n gånger. Vi räknar med att k är minimal, eftersom det utgör det värsta fallet.

De inre looparna har komplexiteten $O(n - 2) = O(n)$ och $O(n)$, så en iteration av den yttersta loopen har komplexiteten $O(n+n) = O(n)$. Det här kommer upprepas $n - k$ gånger, så den totala tidskomplexiteten blir $O((n - k) * n) = O(n^2)$. Här kan vi bortse från k eftersom vi antar att den är minimal och inte spelar någon roll när n blir stort.

1.2 Uppgift b

Här gör vi på samma sätt som i uppgift a och får följande kod:

```
// Something O(1)

for (int i = 4; i < poly.length; i += 2 ) {
    // Something O(1)
    queue.add(currentNode);  // O(log(n))
    // Something O(1)
}

// Something O(1)

while (queue.size() + 2 > k) {
    // Something O(1)
    DLList.Node<Point> removed = queue.remove(); // O(log(n))
    // Something O(1)
    queue.remove(prev);      // O(n)
    queue.add(prev);         // O(log(n))
    // Something O(1)
    queue.remove(next);      // O(n)
    queue.add(next);         // O(log(n))
    // Something O(1)
}

// Something O(1)

// Copy all points from the list to an array
while (points.getFirst() != null) {
    // Something O(1)
}

// Something O(1)
```

Första loopen körs $n-2$ gånger, och varje iteration har komplexiteten $O(log(n)) = O(log(n))$. Komplexiteten för hela loopen blir alltså $O((n-2) * log(n) = O(nlog(n))$

Tidskomplexiteten för metoderna i PriorityQueue är givna i dokumentationen för klassen. add(Object) har $O(log(n))$ komplexitet, remove(Object) har $O(n)$ och remove() har $O(log(n))$. Komplexiteten för en iteration är $O(log(n) + n + log(n) + n + log(n) = O(n)$. Loopen körs $n-k$ gånger och komplexiteten för samtliga iterationer blir $O((n-k) * n) = O(n^2)$. Även här räknar vi med att k är minimal och alltså inte spelar någon roll när n blir stort.

Sista whileloopen körs n gånger för att vi kör genom hela listan och tar bort element ur den. Komplexiteten här blir $O(n)$.

För att få fram komplexiteten för hela algoritmen summerar vi komplexiteten för alla tre loopar. Vi får då $O(nlog(n) + n^2 + n) = O(n^2)$.

1.3 Diskussion

Vi ser att komplexiteten för den första och den andra algoritmen blir densamma i slutändan. Det här beror på att det inte går att uppdatera ett kö-element i PriorityQueue i Javas implementering. För att komma runt det tar vi först bort elementet för att sedan lägga till det igen. Borttagningen tar $O(n)$ tid, och det är det här som gör att komplexiteten för hela algoritmen blir $O(n^2)$. För att få ner komplexiteten behöver vi göra det snabbare att uppdatera ett element i kön. Det här går att göra på (minst) två sätt:

1. Istället för att faktiskt ta bort elementen ur kön så kan man bara *markera* dem som borttagna. När man stöter på ett element som är markerat så kan man välja att helt enkelt strunta i det. Då krävs det bara en add-operation för att uppdatera ett element - en operation som endast har komplexiteten $O(log(n))$.

2. Skapa en egen implementering av PriorityQueue som har förbättrad funktionalitet för remove(Object). Man kan till exempel använda sig av en *binary heap* som har $O(log(n))$ komplexitet för remove(Object) [1].

References

[1] Binary heap, Wikipedia, 2016-02-02
`https://en.wikipedia.org/wiki/Binary_heap`