Komplexitetsanalys

Niklas Baerveldt, Lucas Ruud

January 2018

1 2

1.1 a)

Om vi gör en grov komplexitetsanalys så ser vi att, för att räkna fram värdemåttet av varje punkt så måste traversera, i det här fallet en array, att travesrera en array tillhör $O(n)$. Dock så är denna array utanför en while-loop, vilket gör att den inte kommer att multipliceras med någonting. Sedan har vi while-loopen vilken körs tills vi har mindre än k punkter kvar. Vi kan anta att denna också tillhör $O(n)$ om vi gör en grov analys. I while-loopen så tar vi bort den minst betydelsefulla punkten ur arrayen. Detta tillhör också $O(n)$ eftersom om man tar bort ett element ur en array så måste alla element som kommer efter att skjutas upp för att fylla den lediga platsen. Till sist så räknar vi om värdet av två punkter i arrayen, detta kan vi antar $O(1)$ tid eftersom vi har direkt-access till alla element i arrayen och vi tar inte bort något eller lägger till något.
Så totalt sett så med en grov komplexitetsanalys komma fram till att detta algoritm tillhör $O(n^2)$.
Orsaken till att "tag bort" i en array tar lång tid är, som tidigare nämnt att man måste flytta alla element som ligger under det borttagna upp ett steg. Detta innebär att man måste, i värsta fall traversera hela arrayen och flytta upp varje element ett i taget till en plats ovanför. Detta är väldigt tidskrävande och särskilt om man vill göra det många gånger, vilket man vill göra eftersom man vill ta bort punkter tills man har k stycken kvar.
Orsaken till att använda en linked list blir ungefär lika dåligt är för att trots att det är lätt att lägga till och ta bort saker i en linked list så tar det väldigt lång tid att hitta rätt plats. Alltså varje gång man vill leta upp ett speciellt index i listan så måste man börja med att söka från börja av listan igen. Detta går ganska fort om det man letar efter redan finns långt fram i listan, men om det man letar efter finns långt bak i listan och listan är väldigt lång så kan detta ta väldigt lång tid. Särskilt om man vill gå igenom listan många gånger vilket man vill i denna uppgiften.

1.2 b)

Koden till addLast metoden

```
203     public void addLast(Point p) throws NullPointerException{
204         if(p==null)
205         {
206             throw new NullPointerException();
207         }
208         if(head == null)
209         {
210             head = new Node(p,size++);
211             tail = head;
212         }
213         else {
214             tail.next = new Node(p, size++);
215             tail.next.prev = tail;
216             tail = tail.next;
217         }
218         // TODO
219     } // end addLast
```

Koden till reduceListToKElemnts metoden

```
226
227     public void reduceListToKElements(int k) {
228         // TODO
229         // Calculates the initial important measure for all nodes.
230         Node n = head.next;
231         q.add(head);
232         for(int i = 1; i < size-1 ; i++)
233         {
234             n.imp = importanceOfP(n.prev.p,n.p,n.next.p);
235             q.add(n);
236             n=n.next;
237         }
238         q.add(tail);
239         // Assume there are at least 3 nodes otherwise it's all meaningless.
240         // now reduce the list to the k most important nodes
241         while(q.size() > k)
242         {
243             Node node = q.poll();
244             Node nextNode = node.next;
245             Node prevNode = node.prev;
246             nextNode.prev = prevNode;
247             prevNode.next = nextNode;
248             // recalculate importance for rem.next, neighbour to the right
249             if(nextNode != tail) {
250                 nextNode.imp = importanceOfP(nextNode.prev.p, nextNode.p, nextNode.next.p);
251             }
252             // and rem.prev, neighbour to the left
253             if(prevNode != head) {
254                 prevNode.imp = importanceOfP(prevNode.prev.p,prevNode.p,prevNode.next.p);
255             }
256             q.remove(nextNode);
257             q.remove(prevNode);
258             q.offer(prevNode);
259             q.offer(nextNode);
260         }
261     }
262
263 }
```

1.3 c)

Som tidigare så har vi har vi en while-loop som vi kan anta har en tidsåtgång som tillhör $O(n)$. Innuti denna while-loop så utförs flera operationer, men de som är mest intressanta är "remove()", "offer()" och "poll()". Om man kollar i javas API för en priority queue så får vi reda på att metoderna "offer()" och "poll()" har en tidsåtgång som tillhör $O(log(n))$ medans "remove()" har en tidsåtgång som tillhör $O(n)$. Detta leder till att vi har en metod som tar $O(n)$, "remove()", som utförs innuti en while-loop, vilken också har en tidsåtgång som tillhör $O(n)$. Båda dessa gör att vi kan med en grov komplexitetsanalys uppskatta att vår nya algoritm har en tidsåtgång som tillhör $O(n^2)$, vilket är lika med komplexiteten av vår gamla algoritm. Alltså är vår nya algoritm inte heller mycket snabbare än vår gamla.