Lösningsförslag

Algoritmers effektivitet, tidskomplexitet

U 1. Med tidskomplexitet menar vi ett mått på hur tidsåtgången växer som en funktion av problemets storlek. Vi är inte intresserade av tid mätt i millisekunder e.d. eftersom detta ändå blir olika på olika datorer. I stället mäter vi komplexiteten genom att reda ut hur många basala operationer en algoritm utför. På detta sätt får vi möjlighet att jämföra algoritmers effektivitet med varandra. Vi får också möjlighet att avgöra om algoritmen kan användas på stora probleminstanser eller om det kommer att ta orimligt lång tid (t.ex. om tidskomplexiteten är exponentiell).

U 2.
a) Den yttre loopen genomlöps n gånger. För varje gång exekveras den inre loopen två gånger. `SimpleStatement` kommer att exekveras $2n$ gånger. Komplexiteten är $O(n)$.
b) Den yttre loopen genpmlöps n gånger. För varje gång exekveras den inre loopen $n/2$ gånger. Det blir totalt $n * n/2$ gånger som `SimpleStatement` exekveras. Tidskomplexiteten är $O(n^2)$.
c) Den yttre loopen exekveras för $i = 1, 2, ..., n$. När $i = k$ i den yttre loopen exekveras den inre loopen för $j = n, n-1, ...k$ d.v.s. $(n - k + 1)$ gånger. Satsen i den inre loopen kommer alltså att exekveras $n + (n-1) + (n-2) + ... + 2 + 1 = n(n+1)/2$. Algoritmen har tidskomplexitet $O(n^2)$.
d) Den innersta if-satsen kommer att exekveras n^2 gånger. Satsen `SimpleStatement` kommer bara att exekveras när i i den yttre loopen är lika med j i den inre loopen d.v.s. när $i = j = 1, 2, ..., n$. Den exekveras alltså totalt n gånger. Men algoritmens tidskomplexitet är $O(n^2)$ ty if-satsen utförs n^2 gånger.

U 3. Vi antar att det finns n element i kön.
a) `addFirst` och `removeFirst` blir $O(1)$. `addLast` blir $O(n)$, man måste ju leta upp sista elementet. `removeLast` blir $O(n)$ av samma skäl.
b) Nu blir `addFirst`, `removeFirst` och `addLast` $O(1)$. `removeLast` blir $O(n)$ ty vi måste leta upp näst sista elementet i listan, dels för att ta bort det sista elementet, dels för att uppdatera referensen till sista elementet.
c) Om en `dequeue` representeras av en referens till första noden och en referens till sista noden i en dubbellänkad lista blir alla operationerna O(1).

U 4. Därför att tiden för samma algoritm kommer att bli olika på olika datorer. Om den teoretiska tidskomplexiteten är $T(n)$ så är den verkliga tidsåtgången $t(n) \approx c * T(n)$ där c är tidsåtgången för de operationer som utförs $T(n)$ gånger.

U 5. Eftersom tidskomplexiteten är $O(n)$ så gäller det att $t(n) = c * n$ för någon konstant c. Vi kan bestämma värdet på c för den dator där algoritmen exekverats ur sambandet $c * 100 = 1$ vilket ger $c = 1/100$. Eftersom $t(n) = c * n$ kommer ett problem av storleken 1000 att ta $c * 1000$ ms. Om vi nu sätter in vårt värde på c blir detta 10 ms.

Alternativt kan vi i detta fall resonera så här: Vi har linjär tidskomplexitet. Problemstorleken 1000 är 10 gånger större än problemstorleken 100. Alltså krävs det 10 gånger så lång tid, d.v.s. 10 ms.

U 6. På samma sätt som i föregående uppgift kan vi bestämma konstanten c för exekveringstiden $t(n) = c * 2^n$ genom att vi vet att det tog 1 ms för $n = 10$. Detta ger ju att $c * 2^{10} = 1$ eller $c = 2^{-10}$.

En sekund är 1000 ms. Storleken (n) för det problem som kan lösas på denna tid får vi ur sambandet $c * 2^n = 1000$. Om vi sätter in värdet på c ger detta $2^n = 1000/2^{-10}$ eller $2^n = 1000 * 2^{10}$, vilket ger $2^{n-10} = 1000$ eller $n - 10 =^2 log1000 = 9.97$. Det största heltalsvärde för n för vilket exekveringstiden inte överstiger 1 sekund blir därför $10 + 9 = 19$.

U 7.

a) För `ArrayList` är denna operation $O(1)$. För `LinkedList` är den $O(n)$ eftersom man måste leta sig fram till rätt position.

b) För båda blir det $O(n)$. Man måste ju leta upp elementet.

c) Den blir $O(n)$ för båda. I en `ArrayList` kan man visserligen få access till plats `index` på konstant tid. Men före insättningen måste man först flytta alla element till höger om denna plats ett steg högerut. Detts kostar $O(n)$ i värsta fall.

d) Den blir $O(n)$ för båda. Motviering: samma som i föregående deluppgift.

e) Om listan är av typ `ArrayList` kostar varje `get`-operation $O(1)$. Den utförs n gånger vilket ger en total kostand på $O(n)$.
För en lista av typ `LinkedList` kostar `get(i)` i, man måste ju flytta sig i steg framåt i listan. Operationen utförs för $i = 0, 1, 2, ...n - 1$, vilket ger kostnad $1 + 2 + ... + n - 1 = O(n^2)$.

f) Genom att använda en iterator blir kostnaden $O(n)$ i båda fallen:

```
Iterator<Integer> itr = list.iterator();
int sum = 0;
while (itr.hasNext()) {
    sum = sum + itr.next();
}
```

Alternativt (och ekvivalent) kan man använda en foreach-sats:

```
int sum = 0;
for (int i: list)  {
    sum = sum + i;
}
```

U 8.

1. Om det finns $n + k$ studenter i salen måste läraren leta på sin lista $n + k$ gånger. Eftersom den är osorterad letar läraren då antagligen genom att läsa namn successivt tills ett namn hittas eller tills listan är slut. I värsta fall genomletas hela listan utan att namnet hittas. Då måste läraren läsa totalt $(n + k) * n$ namn på listan vilket kostar $O(n^2)$. I k av fallen kommer läraren att också skriva till ett namn på listan. Detta kostar $O(k)$. Totalt blir det $O(n^2)$.
   Om alla de n personerna som finns på listan också finns i lokalen behöver läraren bara läsa i medeltal halva listan med namn för att hitta studenten. Kostnaden sjunker då till hälften, men är fortfarande $O(n^2)$.
2. Läraren läser listan rakt igenom vilket kostar $O(n)$. Därefter skall k studenter skrivas till vilket är $O(k)$. Eftersom k är mycket mindre än n blir det hela $O(n)$.

U 9. Eftersom man kommit fram till att tidskomplexiteten är kvadratiskt växande med problemets storlek n så skall exekveringstiden bli ungefär fyra gånger så lång vid dubblering av problemstorleken. Man kan därför köra algoritmen för en serie fördubblade värden t.ex. $n = 10, 20, 40, 80, 160, ...$och mäta tiderna $t_1, t_2, ...$ för dessa. Då bör det visa sig att $t_i \approx 4 * t_{i-1}$ för alla i.

Alternativt kan vi genom en serie körningar för olika värden på n försöka visa att exekveringstiden blir $c * n^2$. För varje värde på n mäter man då exekveringstiden $t(n)$ och skriver ut kvoten mellan denna och n^2. Om denna kvot visar sig vara ungefär densamma (= konstanten c) för olika värden på n är det troligt att analysen var korrekt.