Jämförelser av några algoritmers effektivitet

Algoritmer och datastrukturer

Obligatorisk Laboration nr 4

Syfte

Att ge en konkret erfarenhet av hur beräkningstiden hos algoritmer med olika tidskomplexitet beror av storleken på indata.

Programkod

Programkoden finns på kursens hemsida.

Litteratur och föreberedelser

Weiss kap. 5.1-3, 10.1. Uppgifterna 1-4 i laborationen bygger på Weiss kap. 5.1-3. Du måste sätta dig in i algoritmerna i fig. 5.4, 5.5, samt 5.8 för att kunna lösa dessa uppgifter.
Uppgift 5 som bygger på ordpusselproblemet i kap. 10.1 kan nog lösas utan detaljförståelse av alla delar i programmet, den kan du inhämta efter hand, men en grundförståelse av de centrala delarna är nödvändig för att lösa uppgiften.

Färdig programkod

Given programkod som behövs i uppgifterna finns på kurshemsidan under fliken **Laborationer**.

Genomförande och redovisning

Uppgifterna är obligatoriska.

- Redovisning sker i grupper om två personer – inte färre, inte fler.
- Varje grupp skall självständigt utarbeta sin egen lösning.
  Samarbete mellan grupperna om övergripande principer är tillåtet, men inte plagiering.

Gå till kurshemsidan och vidare till **Laborationer->Hur man använder Fire-systemet**.

- Om du inte registrerat labbgrupp i Fire än: Gör laboration 0.0 innan du fortsätter!
- Redovisa filerna `MaxSumTwoDimensions.java`, `lab4_resultattabell.xls`, samt `README.txt` med allmänna kommentarer om lösningen.
- Andra halvan av **Redovisningsproceduren** beskriver hur du postar materialet i Fire.
- Senaste redovisningsdag, se Fire.

Tidmätning

Vid tidmätning i uppgifterna 1 och 5 bör alla övriga program som belastar datorn, och som kan undvaras, stängas av. Mätning av beräkningstid med systemklockan som sker här ger information om förlupen realtid men ingen information om hur mycket processorn arbetat med själva algoritmen. Annan systemtid, I/O m.m. kommer därför att räknas in. Mätningen blir alltså inte helt exakt men bör ändå ge en ganska god uppfattning om skillnaderna mellan de olika algoritmerna.

Uppgift 1 Maximala delsegmentsumman i heltalsvektor `[MaxSumOneDimension.java]`

Notera hur lång tid det tar att få svaret med de olika algoritmerna. Programmet testar algoritmerna i ordningen $O(N)$ (nr 3), $O(N \log N)$ (nr 4), $O(N^2)$ (nr 2), samt $O(N^3)$ (nr 1) för fältlängder 10, 100, 1000, osv.[1] Stämmer det att en tiofaldig ökning av fältlängden ger en hundrafaldig ökning av beräkningstiden för en algoritm med kvadratisk tidskomplexitet, och en tusenfaldig ökning för en med kubisk komplexitet? Kommentera ev. avvikelser.

Generaliserad segmentsumma i två dimensioner

Algoritmerna i uppgift 1 beräknar den största konsekutiva delsegmentsumman i en heltalsvektor och vi har sett att problemet kan lösas i tid $O(n)$. Man kan formulera det analoga problemet för en rektangulär heltalsmatris, att finna den största *delmatrissumman*. Exempel: I matrisen nedan har den inringade delmatrisen högst elementsumma.

-1	-2	1	3
0	2	0	8
-5	10	-2	1
4	5	-7	1

Vi ser att i denna version av problemet kan en ytterkolumn eller ytterrad mycket väl innehålla negativa tal så länge kolumnens eller radens elementsumma är större än noll.
Om tidskomplexiteten anges i termer av sidlängden för kvadratiska matriser så finns en brute-force-algoritm med komplexitet $O(n^6)$. En optimering liknande den som gjordes för det endimensionella fallet från $O(n^3)$ till $O(n^2)$ sänker komplexiteten till $O(n^5)$. Dessutom kan man generalisera den linjära algoritmen för det endimensionella fallet till att lösa det tvådimensionella fallet och sänka komplexiteten ytterligare till $O(n^4)$. Om du tror att det finns en ännu effektivare algoritm – upp till bevis!

För att underlätta bedömningen av algoritmerna skall alla variabler för radindex benämnas r1, r2, … och för kolumnindex c1, c2, … . Använd inte x, y, i, j eller blandningar av dessa.

Uppgift 2 En tvådimensionell brute-force-algoritm `[MaxSumTwoDimensions.java]`

Implementera metoden

```
public static int maxSubMatrixSumBad(int[][] a)
```

som löser problemet genom att beräkna elementsumman hos alla möjliga delrektanglar och returnera den största summan. Om alla element är negativa sätts summan till 0. Metoden `main` innehåller kod för att testa algoritmerna på matriser av ökande storlek. Hur stor matris klarar algoritmen på en minut?

Uppgift 3 En bättre tvådimensionell algoritm

Algoritmen ovan behandlar troligen många element upprepade gånger på ungefär samma sätt som $O(n^3)$-algoritmen i det endimensionella fallet. Implementera $O(n^5)$-metoden

```
public static int maxSubMatrixSumBetter(int[][] a)
```

genom att tillämpa samma optimeringsidé som i $O(n^2)$-algoritmen för det endimensionella fallet.

[1] Observera att algoritmernas numrering från 1 till 4 överensstämmer med ordningen som de presenteras i kursboken, inte i vilken ordning de exekveras i testprogrammet.

Uppgift 4 En ännu bättre tvådimensionell algoritm

Implementera $O(n^4)$-metoden

```
public static int maxSubMatrixSumEvenBetter(int[][] a)
```

genom att generalisera metoden i $O(n)$-algoritmen för det endimensionella fallet:

```
int maxSum = 0;
int thisSum = 0;
for( int j = 0; j < a.length; j++ ) {
    thisSum = Math.max( 0, thisSum + a[ j ] );
    maxSum  = Math.max( maxSum, thisSum );
}
// maxSum innehåller nu resultatet
```

En ledtråd lämnas på begäran, men försök själv först!

Uppgift 5 Ordpussel (Weiss 10.1) `[UHWordSearch.java]`

I den här övningen skall vi analysera och empiriskt undersöka ordpusselprogrammet. Analysen måste förstås ske genom att läsa koden för att se vilka faktorer som påverkar beräkningstiden. Den empiriska delen består i att göra praktiska tidmätningar på pussel av olika storlekar.

På hemsidan finns två arkivfiler med pusselfiler av olika storlekar. Pusselfilerna `2.txt,4.txt,...` innehåller kvadratiska ordpussel. Filen `dictionary.txt` innehåller en engelsk ordlista omfattande ca 110000 ord och förkortningar. Programmet är modifierat så att det vid leveransen söker linjärt, ord för ord, i ordlistan, men detta kan enkelt ändras till den betydligt effektivare metoden *binärsökning* (Weiss 6.6) genom att ändra en boolesk konstant i klassen `UHWordSearch`. Man kan även välja om prefixtestning skall användas vid binärsökning. Notera följande uppgifter i kalkylarket **`lab4_resultattabell.xls`**: *Antal ordförekomster*, *antal sökningar* i ordlistan, *beräkningstid*, samt *tidskvot*. Tidskvoten är kvoten mellan beräkningstiden för en pusselstorlek och beräkningstiden för närmast mindre storlek. Tabellens utseende framgår av bilden på nästa sida. Vid tidmätning bör programmet köras utan utskrifter av de funna orden eftersom tidsåtgången för utskrifterna kan maskera den intressanta beräkningstiden. Utskrifterna kan stängas av/sättas på med en boolesk konstant i klassdefinitionen. Om någon tabellruta blir svår/meningslös att fylla i så gå vidare! Hur stora pussel som kan lösas inom rimlig tid beror på den aktuella datorns prestanda. Tag åtminstone med alla fall som ger beräkningstider mellan 0.1 sek och ca fem minuter. Gör samtliga mätningar på samma dator och avsluta onödiga processer innan mätningarna påbörjas.

1. Prova först med de **små** pusselfilerna som indata.
2. Ändra sedan i programmet så att binärsökning används istället för linjärsökning (det räcker att ändra en boolesk konstant i klassdefinitionen), och jämför med föregående resultat.
3. Ändra till sist så att prefixtestning används. Prefixtestning är bara meningsfull i samband med binärsökning. Om prefixtestning väljs kommer binärsökning att användas automatiskt.

Att besvara i redovisningen

1. Vilka faktorer påverkar beräkningstiden? Finns det sinsemellan oberoende faktorer? Motivera med konkreta hänvisningar till koden.
2. Hur mycket längre tid bör det i princip ta att lösa ett pussel när storleken fördubblas?
3. Hur mycket längre tid tar det i praktiken?
4. Kommentera ev. avvikelser!

Du måste analysera koden noga, särskilt metoderna `solvePuzzle` och `solveDirection`. Dåligt genomtänkta svar ger retur!
Skicka in ifyllt excelark och **`README.txt`** med svar på frågorna via Fire.

Pusselstorlek	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Samtliga metoder													
Antal ordförekomster													
Linjärsökning													
Antal sökningar													
Tid (sek.)													
Tidskvot													
Binärsökning													
Antal sökningar													
Tid (sek.)													
Tidskvot													
Binärsökning													
och prefixtestning													
Antal sökningar													
Tid (sek.)													
Tidskvot													