Algoritmer och datastrukturer
version 2015-11-13 perjee

Laboration 4:
Exekveringstid för sorteringsalgoritmer

Syfte

Laborationen handlar om att ge studenten kunskap om några välkända sorteringsalgoritmer och om att empiriskt studera exekveringstid och ordonalitet för dessa algoritmer.

Förberedelse

Läs de relevanta avsnitten i kurslitteraturen (se litteraturhandledningen) och föreläsningsanteckningarna till föreläsning 10 och 11.

Uppgifter

Ni ska

1. utföra de beskrivna experimenten och besvara ställda frågor.
2. förbereda för tentamina: Visa att antalet jämförelser utförda av *Insertionsort* eller *Selectionsort* (välj en av algoritmerna) är $O(n^2)$ i sämsta fall.
3. implementera en av algoritmerna *Quicksort*, *Mergesort* eller *Heapsort* i Java. Skriv ett testprogram som visar att implementationen fungerar.

Genomförande

Uppgift 1

Ni ska utföra ett experiment för att avgöra i vilken ordning olika sorteringsalgoritmer av ett givet program utförs.

För att få ett intryck om prestanda-skillnader mellan olika sorteringsalgoritmer kan ni prova följande demonstrationsprogram:

`http://www.cs.wpi.edu/~cs2005/e00/handouts/sorting/sorting.html`

De sorteringsalgoritmer som ni skall undersöka är implementerade i ett färdigt `C++`-program. Programmet finns tillsammans med annat material i `ZIP`-filen `aod_lab4_u1.zip`. En del av koden som implementerar olika sorteringsmetoder återges i en bilaga till detta dokument (på sida 5ff).

Själva programmet finns i versioner för Windows (`AOD_Lab4.exe`) och för Linux (`AOD_Lab4`). Filen `aod_lab4_u1.zip` innehåller i katalogen `data` även ett antal textfiler som ska användas som testdata.

Programmet är en applikation som måste exekveras i kommandotolken. För att kunna använda programmet behöver ni starta kommandotolken (`cmd.exe` under Windows, terminal under Linux). Det är enklast om ni navigerar till katalogen där ni lagrar filerna till uppgift 1: Anta att ni använder katalogen `C:\AOD\lab4`. Ifall ni använder Windows och ni befinner er i en annan enhet än `C`, så måste ni först byta enheten - det görs genom att mata in

```
C:
```

Sedan byter ni till katalogen `\AOD\lab4`: Mata in

```
cd \AOD\lab4
```

En översikt över katalogens innehåll får ni med hjälp av kommandot

```
dir
```

(Windows) eller

```
ls -l
```

(Linux). Utskriften skulle se ut som i exemplet nedan (i Linux):

```
total 748
-rwxr-x--- 1 willie willie 724448 Nov 26 12:53 AOD_Lab4
-rwxr-xr-x 1 willie willie  33280 Nov 13 15:11 AOD_Lab4.exe
drwxr-xr-x 2 willie willie   4096 Nov 26 12:47 data
```

När ni kör programmet, så kan ni ange vilken fil som ni vill exekvera de fyra sorteringsalgoritmerna på:

```
AOD_lab4.exe data\a.txt
```

sorterar innehållet av filen `a.txt` i katalogen `data`. Om ni dessutom anger optionen `-p`, så skrivs sorterade värdena ut:

```
AOD_lab4.exe -p data\a.txt
```

Tips: Man kan leda programmets utskrifter till en textfil, så slipper man att använda papper och penna för att notera resultaten:

```
AOD_lab4.exe -p data\a.txt > logg_a.txt
```

Det data som ni skall sortera finns i sju filer, `a.txt` till `g.txt`. De varierar i storlek och innehåll, så det är viktigt att ni håller reda på det när ni skall lista ut i vilken ordning sorteringsalgoritmen exekveras. Titta på vad det är för sorts data ni sorterar - öppna filerna och undersök innehållet och storlek!

Viktigt: Sortingsalgoritmerna anropas alltid i samma ordning. Notera att ordningen på metoderna i bilagan inte nödvändigtvis har någon relevans för hur de anropas i programmet `AOD_Lab4`.

Besvara följande frågor:

1. Vilken strategi används för att välja pivotelement i denna implementation av *Quicksort*?
2. Vissa tider avviker från den tid som man skulle förvänta sig. Vad är orsaken till detta? Är det slumpmässiga förändringar eller kan de förklaras på något sätt? Hänvisa till kursboken eller föreläsningsanteckningarna för att stödja din förklaring.

3. Vilken sorteringsalgoritm används av sortingsmetod 1, vilken av metod 2, av 3 och av 4?

Tips

Skapa en tabell som visar exekveringstiderna för respektive datafil `a.txt` till `g.txt`, sorterade med en viss algoritm.

Utgående från denna tabell ska ni skapa ett x-y-diagram där x-axeln visar datastorlek och y-axeln exekveringstid. Tänk på att x-axeln bör vara linjär för att ni skall kunna göra vettiga slutsatser. Y-axeln kan vara linjär eller logaritmisk, beroende på vad ni får ut. Det kan även vara en idé att presentera resultaten i två eller fler olika grafer.

Använd INTE stapeldiagram, för de kommer inte att kunna ge er den information som ni behöver. Om ni använder Excel eller OpenOffice, använd ett punkt-diagram.

De tre frågorna ovan är relaterade till varandra. Avgör respektive algoritmens teoretiska sämsta- och medel-tidskomplexitet (ordonalitet) och identifiera algoritmerna från detta och graferna. Avvikelser från förväntat resultat är ledtrådar för att avgöra vilken algoritmus som är vilken. För att kunna ge rätta svar är det kanske bra att även undersöka data sparade i filerna `a.txt` - `g.txt`.

För att vara säker på att exekveringstiderna inte beror på tillfälliga variationer av maskinens belastning, så kör flera gånger och ta ett medelvärde på exekveringstiden för respektive sorteringsalgoritm/datafil.

Uppgift 2

Använd aritmetiska serier för att ange ett matematiskt uttryck för antal jämförelser för respektive algoritm. Det finns ledtrådar hur detta skall göras i både boken och föreläsningsanteckningarna.

Presentera dessa i rapporten. Detta är något som kan komma upp på tentaminan.

Uppgift 3

Skriv ett Java-program som implementerar *Mergesort* eller *Quicksort* (med en bättre strategi för att välja pivot-element än i `sorter.cpp`). Indata är en array eller `List`-objekt som skall skickas som argument till metoden. Metoden skall sedan returnera en sorterad version. Metoden ska vara kapabel att sortera element som implementerar interface:et `Comparable`.

Sorteringsmetoden skall ha en signatur som

```
public static <T extends Comparable<?  super T>> void sort (T[] array)
```

eller

```
public static <T extends Comparable<?  super T>> void sort (List<T>
list)
```

För mer information om generiska klasser resp. metoder se avsnitt 6.4 i Weiss och Java-tutorials om generiska klasser/metoder på
http://docs.oracle.com/javase/tutorial/java/generics/index.html.

Dokumentation till `Comparable`:

http://docs.oracle.com/javase/8/docs/api/java/util/Comparable.html

En bra introduktion till `Collections`-API finns under
http://docs.oracle.com/javase/tutorial/collections/index.html.

Se dessutom:
http://docs.oracle.com/javase/8/docs/technotes/guides/collections/index.html.

Uppgift 4

Om man vill kan man tala om hur sorteringsalgoritmen skall sortera data genom att skicka med ett objekt som implementerar interface:et `Comparator`. Skriv en passande `Comparator`-klass och använd den på din sorteringsalgoritmus.

Dokumentation till `Comparator`:

http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

Rapportering

Ni bör lösa och rapportera denna uppgift i grupper om två. Skriv en rapport på uppgift 1 och uppgift 2 i PDF-format. Källkoden och Java-doc samt ett testprogram skall lämnas in via BlackBoard tillsammans med rapporten som vanligt.

Lycka till!

Bilaga

Kod till `sorter.cpp`:

```
// BUBBLESORT
void Sorter::bubbleSort (std::string * data, int size)
{
    bool changed = true;
    int step = 0;

    do {
        changed = false;
        for (int i = 0; i < size - 1; i++)
        {
            if (data [i].compare (data [i + 1]) > 0)
            {
                swap (data, i, i + 1);
                changed = true;
            }
        }
        step++;
    } while (changed);
}
```

```
// INSERTION_SORT
void Sorter::insertionSort (std::string * data, int size)
{
    insertionSortWorker (data, 0, size);
}


void Sorter::insertionSortWorker (std::string * data, int l, int h)
{
    for (int i = l + 1; i < h; i++)
    {
        for (int k = i; k >= l + 1; k--)
        {
            if (data[k].compare (data[k - 1]) < 0)
                swap (data, k, k - 1);
        }
    }
}
```

```
// MERGESORT
void Sorter::mergeSort (std::string * data, int size)
{
    std::string * tmp = new std::string [size];
    mergeSortWorker (data, tmp, 0, size - 1);
}



void Sorter::mergeSortWorker (std::string * data, std::string * tmp,
                                                        int low, int high)
{
    int middle, indexLow, indexHigh;

    if (high - low >= 1)
    {
        middle = (low + high) / 2;
        mergeSortWorker (data, tmp, low, middle);
        mergeSortWorker (data, tmp, middle + 1, high);

        for (int i= middle; i >= low; i--)
            tmp[i] = data[i];

        for (int i= middle + 1; i <= high; i++)
            tmp[middle + 1 + high - i] = data[i];

        indexLow = low;
        indexHigh = high;

        for (int i = low; i <= high; i++)
        {
            if (tmp[indexLow] > tmp[indexHigh])
            {
                data [i] = tmp[indexHigh];
                indexHigh--;
            } else {
                data [i] = tmp[indexLow];
                indexLow++;
            }
        }
    }
}
```

```
// QUICKSORT
void Sorter::quickSort (std::string * data, int size)
{
    quickSortWorker (data, 0, size - 1);
}



void Sorter::quickSortWorker (std::string * data, int low, int high)
{
    int pivot = 0;
    if (high - low >= 1)
    {
        pivot = partition (data, low, high);
        quickSortWorker (data, low, pivot);
        quickSortWorker (data, pivot + 1, high);
    }
}



int Sorter::partition (std::string * data, int low, int high)
{
    int pivot = low;
    int left = low;
    int right = high;
    bool finished = false;
    std::string pivotValue = data [pivot];

    while (!finished)
    {
        while ((data[right] >= pivotValue) && (right > low))
            right--;
        while ((data[left] < pivotValue) && (left < high))
            left++;

        if (left < right)
            swap (data, left, right);
        else
            finished = true;
    }
    return right;
}



void Sorter::swap (std::string * a, int b, int c)
{
    std::string v = a [b];
    a [b] = a [c];
    a [c] = v;
}
```