Obligatorisk uppgift 4, del 2

Er uppgift

Syftet med denna gruppövning/laboration är att ni ska bestämma tidskomplexiteten för de nedanstående sorteringsalgoritmerna. Detta ska göras genom att räkna antalet *primitiva operationer* som varje utförs i varje algoritm. Utifrån detta ska sedan definitionen av *Ordo* användas för att ge uttryck för varje algoritms tidskomplexitet. Konstanterna c och n_0 skall bestämmas.

Fullständig analys för båda *worst-case* och *best-case* skall utföras för alla algoritmer med undantag för In-place quicksort och randomsort där ett resonemang kring komplexiteten räcker (OBS! att bara ange vad komplexiteten är utan motivering räcker inte). Förslag till analys mottages dock gärna!

Inlämning

Arbetet påbörjas på gruppövning 2 den 23 april och den färdiga lösningen inlämnas renskriven senast den 7 maj, kl 12:00.

Algoritmer

Swap

Denna funktion byter plats på värdena som skickas in till parametrarna. Används i alla nedanstående algoritmer utom randomsort.

```
function swap(a, b)
begin
        temp := a
        a := b
        b := temp
end
```

Naiv bubblesort

Den enklaste varianten av bubblesort. Loopar över onödigt stort intervall och bryr sig inte om att undersöka om listan är färdigsorterad i förtid.

```
function bubbleSort(numElements, list[])
begin
        for i := 0 to (numElements – 1)
        begin
                for j := (numElements – 1) downto 0
                begin
                        if list[j] < list[j – 1] then
                        begin
                                swap(list[j], list[j – 1])
                        end
                end
        end
end
```

Optimerad bubblesort

En bättre bubblesort. Ser till att inte loopa över de element som redan ”bubblats” till rätt position och ser till att avsluta om listan blir färdigsorterad i förtid.

```
function bubbleSort(numElements, list[])
begin
    done := false
    i := 0

    while (i < numElements) and (done = false)
    begin
        done := true

        for j := (numElements – 1) downto i
        begin
            if list[j] < list[j – 1] then
            begin
                swap(list[j], list[j – 1])
                done := false
            end
        end

        i := i + 1
    end
end
```

In-place quicksort

Quicksort som använder konstant mängd minne (det vill säga, minne allokeras ej för varje ny dellista utan det befinliga minnet används istället).

```
function quickSort(leftBound, rightBound, list[])
begin
    if leftBound < rightBound then
    begin
        pivot := list[rightBound]

        left := leftBound
        right := rightBound – 1
        while left <= right
        begin
            while (left <= right) and (list[left] <= pivot)
            begin
                left := left + 1
            end
            while (right >= left) and (list[right] >= pivot)
            begin
                right := right - 1
            end

            if left < right then
```

```
                begin
                        swap(list[left], list[right])
                end
        end

        swap(list[left], list[rightBound])

        quickSort(leftBound, left – 1, list)
        quickSort(left + 1, rightBound, list)
    end
end
```

Randomsort

Denna algoritm sorterar en lista genom att prova alla möjliga sätt den kan vara ordnad på tills en sorterad lista hittas.

```
function randomSort(numElements, list[])
begin
    sorted := false
    while sorted = false
    begin
        sorted := true
        tempList := nextPermutation(list)
        i:=0;
        while (i <= numElements – 2 and sorted = true)
        begin
            if tempList[i] > tempList[i + 1] then
            begin
                sorted := false
            end
            i := i +1
        end
    end
end
```

Under beräkning av komplexiteten kan man anta att anropet till funktionen *nextPermutation* alltid levererar en ny unik permutation av listan och att den kräver 20 operationer.