Når skal sortering benyttes?

Samle ting som hører sammen

- Vi kan ordne kategorier skulle ting falle i diverse kategorier. Eks: sport, underholdning, drama osv.
- Sorterer vi etter kategoriene, så samler vi alt som faller i samme kategori
- Dette kalles også partisjonering.

Matche

- Gitt to sekvensielle strukturer, kan vi finne elementer som matcher ved å løpe over kun en gang.

Søk

- Tidligere ble det lært om hvordan å søke i sorterte arrayer ved å benytte binærsøk algoritmen og brukte logaritmisk tid som betyr at søket foregår veldig raskt.

PS!!!
Når det gjelder sorteringsalgoritmene så har jo programmeringspråk en betydning hvor i python så behøves ikke [A] - 1 mens i java så er dette nødvendig.

Bubble sort

Ideen bak "bubble-sort" er å løpe gjennom et array og "rette opp" feil og bare fortsetter å praktisere dette helt til det ikke er noen flere feil å rette opp. Feil kan være at vi har en liste som f.eks: [1,2,3,4,5,10,8,9]. Feilen her ligger i at vi har at tallet 10 forekommer før 8 og 9. Så derfor blir denne algoritmen relevant for dette. Blir da en rekursiv algoritme

En mer presis forklaring:

1. Løpe over hvert par av etterfølgende elementer i arrayet
2. Bytte om rekkefølgen et par dersom det ikke er ordnet
3. Gå til 1. dersom det forekom minst et bytte

Algoritme for Bubble sort:

Input: Et array "A" med "n" elementer

Output: Et sortert array med de samme "n" elementene

```
Procedure BubbleSort(A)
    for (i <- 0) to (n - 2) do
        for (j <- 0) to (n - i - 2) do  # Gjør at vi sjekker neste indeks
            if A[j] > A[j + 1] then
                A[j], A[j + 1] <- A[j + 1], A[j]
        end
    end
```

Denne varianten er ikke optimalisert, fordi si at i den andre for loopen(j), så utfører den aldri et bytte. Det blir unødvendig å fortsette løkken da omsorteringen blir aldri utført.

Optimalisert versjon

Procedure BubbleSort(A)

```
    har_byttet = True
    while (har_byttet != False)
        har_byttet = False
        for (i <- 0) to (n - 2) do
            for (j <- 0) to (n - i - 2) do  # ser vekk fra indekser før A[i]
                if A[j] > A[j + 1] then
                    A[j], A[j + 1] <- A[j + 1], A[j]
                    har_byttet = True
            end
        end
    end
```

Kjøretidsanalyse

Vi iterer fra (0) til (n-2), som svarer til n-1 iterasjoner.

For hver iterasjon løper vi fra (0) til (n-i-2)

- Dette gir “O(n)” steg, men hvorfor?
- For hver iterasjon blir “i” større, så vi iterer over mindre
- I verste tilfelle (når i = 0) får vi (n - 1) iterasjoner
- Men når (i-n-2) får vi ingen iterasjoner.

Hvis vi teller det totale antall iterasjoner får vi

n - 1 + n - 2 + …. + 1 = 1 + 2 + ….. + n -1 = n(n-1)/2 viser til hvor mange iterasjoner som utføres

Gir oss da O(n(n-1)/2) men ved forenkling så får vi O(n^2)

<u>Selection sort</u>

Ideen bak selection sort er å finne det minste i lista og plassere det først. Så vi henter alltid det minste elementet og plassere denne først.

En presis forklaring:

1. La “i” være 0
2. Finn hvor det minste elementet fra “i” og utover ligger
3. Bytt ut elementet på plass “i” med det minste (hvis nødvendig)
4. Øk “i” og gå til 2. frem til “i” når størrelsen av arrayet

Grunnen til at vi øker med “i”, er fordi at når vi først har funnet det minste elementet i hele listen og plasser den på posisjon 0, så må vi sørge for plassere det neste minste elementet i neste indeks.

Algoritme for Selection sort:

Input: Et array “A” med “n” elementer

Output: Et sortert array med de samme “n” elementene

Procedure SelectionSort(A)

```
for (i <- 0) to (n - 1) do
        k <- i
        for (j <- i + 1) to (n - i) do  # øker indeks med 1 for A[i]
                if A[j] < A[k] then
                        k = j # Indeksen som peker på det minste elementet i liste
        end
        if i != k then
                A[i], A[k] <- A[k], A[i]
end
```

Kjøretidskompleksitet gir oss O(n^2) og er raskere enn bubble sort siden i bubble sort så itererer vi gjennom flere ganger for å bytte om plasser til tallene. Selection sort vil derimot sette det minste elementet i en gitt plass og det minste elementet vil forbli i den posisjonen.

Insertion sort

Ideen bak insertion sort er å plassere alle elementene sortert inn i en liste. F.eks, [1,2,3,4,5,6,7] at ved innsetting så blir den sortert slikt. Vi lar alt til venstre for en gitt posisjons [i] være sortert.

Presis forklaring:

1. La “i” være 1
2. Dra det “i-te” elementet mot venstre som ved sortert insetting
3. Øk “i” og gå til 2.frem til “i” når størrelsen av arrayet

Algoritme for Insertion sort:

Input: Et array “A” med “n” elementer

Output: Et sortert array med de samme “n” elementene

```
Procedure InsertionSort(A)
    for (i <- 1) to (n - 1) do
        j <- i
        while j > 0 and A[j-1] > A[j] do
            A[j-1], A[j] <- A[j], A[j-1]
            j <- j - 1
        end
    end
```

Kjøretidskomplekstiteten er $O(n^2)$. Er rask på “nesten sorterte” arrayer og er blant de raskeste algoritmer for små arrayer.

Heapsort

Ideen bak heapsort er å bygge en heap og poppe elementer av heapen. For å bygge en heap så kan vi implementere den gjennom en array og gjøre denne arrayen om til en heap

Presis forklaring:

1. Gjør arrayet om til en max-heap (hver node er større enn begge barna)
2. La “i” være (n-1) der “n” er størrelsen på arrayet
3. Pop fra max-heapen og plasser elementet på plass “i”
4. Senk “i” og gå til steg 3, frem til “i” blir 0

Hvordan bygge max-heap:

- Omgjøre array til en max-heap
- En max-heap er en heap hvor node er større enn begge barna
- En node svarer til en posisjon i arrayet hvor:
    - Roten ligger på plass 0
    - Venstre barn ligger på plass (2i + 1)
    - Høyre barn ligger på plass (2i + 2)
- BuildMaxHeap() er metoden som omgjør arrayet til en maxheap

Hjelpeprosedyre for å bygge en max-heap

Input: En (uferdig) heap “A” med “n” elementer der “i” er roten

Output: En mindre uferdig heap

```
Procedure BubbleDown(A,i,n)
        largest <- i
        left <- 2i + 1
        right <- 2i + 2
        if (left < n and A[largest] < A[left]) then
                largest, left <- left, largest
        if (right < n and A[largest] < A[right]) then
                largest, right <- right, largest
        if (i != largest) then
                A[i], A[largest] <- A[largest], A[i]
                BubbleDown(A,largest,n)
```

Bygge en max heap

Input: Et array “A” med “n” elementer

Output: “A” som en max heap

```
Procedure BuildMaxHeap(A,n)
    for i <- [n/2] down to 0 do
        BubbleDown(A,i,n)
    end
```

Algoritme for Heapsort

Input: Et array A med “n” elementer

Output: Et sortert array med de samme “n” elementene

```
Procedure HeapSort(A)
    BuildMaxHeap(A,n)
    for i <- n - 1 down to 0 do
        A[0], A[i] <- A[i], A[0]
        BubbleDown(A,0,i)
    end
```