

TDDI16 – Föreläsning 8

Sortering

Filip Strömbäck

Planering

Vecka	Fö	Lab
35	Komplexitet, Linjära strukturer	----
36	Träd, AVL-träd	1---
37	Hashning, meet-in-the-middle	12--
38	Grafer och kortaste vägen	-23-
39	Fler grafalgoritmer	--3-
40	Sortering	--34
41	Repetition	---4

Information

Lektion på tisdag

- Ett pass körs utomhus (R26)
- Ett pass körs på plats, men inne (R27)
- Ett pass på distans, via Zoom. Se kurshemsidan för länk.

Laborationer

- Inte längre delgrupper, OK att vara 2 per dator.

- 1 Motivation
- 2 Sorteringsalgoritmer
- 3 Divide and conquer
- 4 Ännu bättre algoritmer?
- 5 Sammanfattning

Varför behöver vi sortera data?

Varför behöver vi sortera data?

- Hitta i stora datamängder
 - Presentera data i ett lätthanterligt format
 - ...
-
- Förenkla vissa typer av problem

Exempelproblem

Du skriver ett system för ett sjukhus, och håller på med delen som håller reda på vilka prover som tagits på alla patienterna på sjukhuset. För att spara minne har du valt att lagra detta som en array där varje element innehåller beteckningen på ett prov som tagits.

För att underlätta för läkarna vill du tillhandahålla möjligheten att jämföra en patient (A) med en "referenspatient" (B) för att snabbt kunna se om de har glömt några prover, eller tagit några extra prover.

Hur gör du detta effektivt utan mer minne?

Lösningssidé 1

1. Iterera genom array A .
 - För varje element, se om elementet finns i array B .
2. Iterera genom array B .
 - För varje element, se om elementet finns i array A .

Tidskomplexitet:

Lösningssidé 1

1. Iterera genom array A .
 - För varje element, se om elementet finns i array B .
2. Iterera genom array B .
 - För varje element, se om elementet finns i array A .

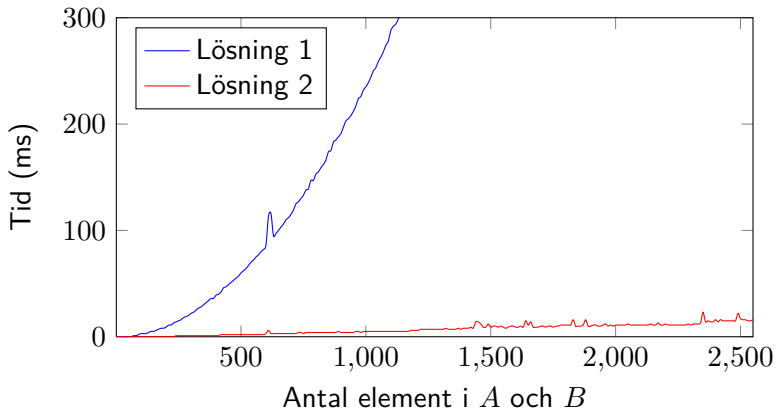
Tidskomplexitet: $\mathcal{O}(n^2)$

Lösningsside 2

1. Sortera A och B , sätt $i, j = 0$
2. Titta på $A[i]$ och $B[j]$:
 - Om de är lika: elementet fanns i båda arrayerna
 - Om $A[i] < B[j]$: element $A[i]$ saknas i B
 - Om $A[i] > B[j]$: element $B[j]$ saknas i A

Tidskomplexitet: $\mathcal{O}(n+?)$

Vi testar!



- 1 Motivation
- 2 **Sorteringsalgoritmer**
- 3 Divide and conquer
- 4 Ännu bättre algoritmer?
- 5 Sammanfattning

Formell problembeskrivning

Givet en sekvens av element och en jämförelseoperator, $<$, ordna elementen så att a kommer före b om $a < b$.

- Hur många jämförelser behöver vi göra?
- Hur många element måste vi flytta på?
- Är sorteringen stabil eller inte?
- Minnesanvändning?
- Tidsanvändning?

Ett första försök – Insertion sort

```
void insertion_sort(vector<int> &v) {  
    // Dela i två delar: en sorterad och en osorterad.  
    for (size_t pos = 1; pos < v.size(); pos++) {  
        // Flytta det första osorterade elementet  
        // tills det hamnar på rätt plats i den  
        // sorterade delen.  
        for (size_t i = pos; i > 0; i--) {  
            if (v[i - 1] <= v[i])  
                break;  
            swap(v[i - 1], v[i]);  
        }  
    }  
}
```

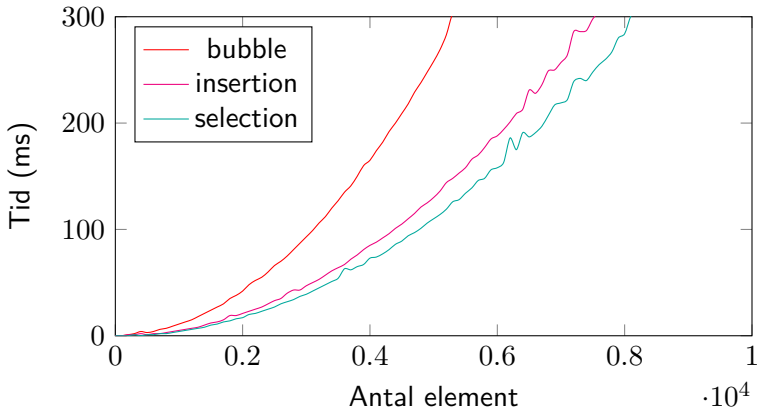
Selection sort

```
void selection_sort(vector<int> &v) {  
    if (v.empty()) return;  
    // Dela i två delar: en sorterad och en osorterad.  
    for (size_t pos = 0; pos < v.size() - 1; pos++) {  
        // Hitta det minsta i den osorterade delen...  
        size_t min = pos;  
        for (size_t i = pos; i < v.size(); i++)  
            if (v[min] > v[i])  
                min = i;  
        // ... och sätt det sist i den sorterade.  
        swap(v[min], v[pos]);  
    }  
}
```

Bubble sort

```
void bubble_sort(vector<int> &v) {  
    for (size_t t = 0; t < v.size(); t++) {  
        // Iterera genom arrayen och flytta element.  
        for (size_t i = 1; i < v.size() - t; i++) {  
            if (v[i - 1] > v[i])  
                swap(v[i - 1], v[i]);  
        }  
    }  
}
```

Vi testar de vi har!



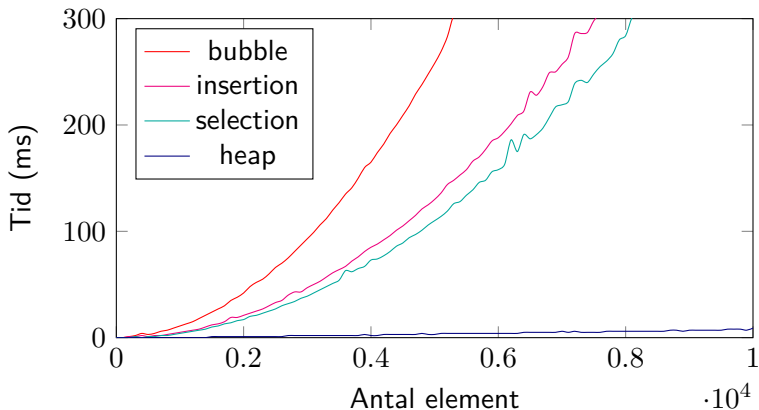
Kan vi hitta en bättre lösning?

Träd har ju bra tidskomplexitet!

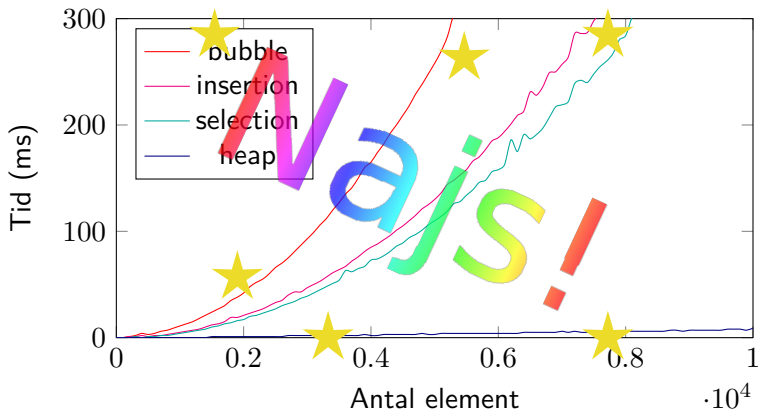
Heapar är bra, de kan vi enkelt lagra i arrayer!

```
void heap_sort(vector<int> &v) {  
    make_heap(v.begin(), v.end());  
  
    for (iter end = v.end(); end != v.begin(); --end)  
        pop_heap(v.begin(), end);  
}
```

Vi testar igen!



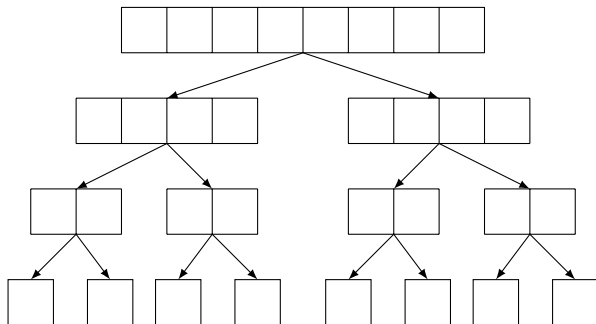
Vi testar igen!



- 1 Motivation
- 2 Sorteringsalgoritmer
- 3 **Divide and conquer**
- 4 Ännu bättre algoritmer?
- 5 Sammanfattning

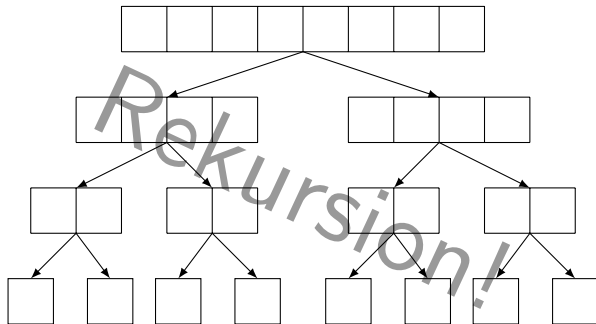
Ny taktik – divide and conquer

Idé: Vi delar upp problemet i mindre bitar som vi kan lösa enklare!



Ny taktik – divide and conquer

Idé: Vi delar upp problemet i mindre bitar som vi kan lösa enklare!



Merge sort – Idé

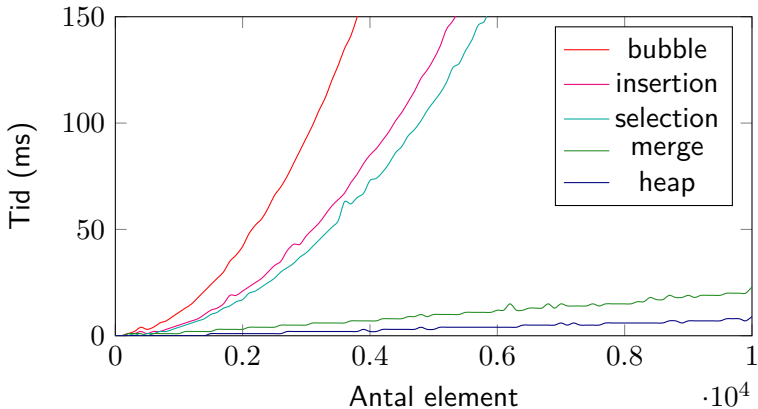
1. Om indatan bara innehåller ett element, returnera originallistan
2. Dela indatat i två lika stora delar
3. Sortera båda delarna
4. Slå samman de två sorterade listorna till en och returnera den

Merge sort – implementation

```
vector<int> merge_sort(iter begin, iter end) {  
    if (end - begin <= 1)  
        return vector<int>(begin, end);  
  
    iter half = begin + (end - begin)/2;  
    vector<int> a = merge_sort(begin, half);  
    vector<int> b = merge_sort(half, end);  
  
    return merge(a, b);  
}
```

Implementationen av merge finns på kurshemsidan

Vi testar!



Quicksort – Idé

Mergesort "baklänges"

1. Välj ett *pivotelement* i arrayen
2. Partitionera arrayen i två delar, element större än och mindre än pivotelementet
3. Kör quicksort på båda delarna av arrayen

Quicksort – implementation

```
void quicksort(iter begin, iter end) {  
    if (end - begin <= 1)  
        return;  
    // Vi väljer alltid första elementet för  
    // enkelhets skull. Det finns bättre alternativ.  
    iter pivot = begin;  
    iter middle = partition(begin, end, pivot);  
    // Sortera båda halvorna med quicksort.  
    quicksort(begin, middle);  
    quicksort(middle + 1, end);  
}
```

Implementationen av partition finns på kurshemsidan

Vilket pivotelement ska vi välja?

Dåligt pivotelement $\Rightarrow$ dålig prestanda!

- Välj ett fördefinierat element (möjligtvis dåligt)
- Medianen av tre
- Ett slumpmässigt element

Bästa möjliga pivot: det element som kommer hamna i mitten, dvs. medianelementet. Varför är det ett dåligt alternativ?

En blandning av olika algoritmer

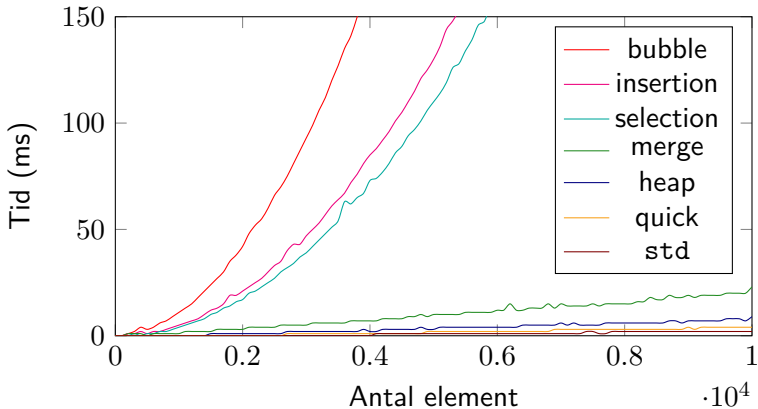
Olika algoritmer är bra i olika fall.

Exempelvis: Quicksort är bra för stora mängder data, men sämre för mindre arrayer.

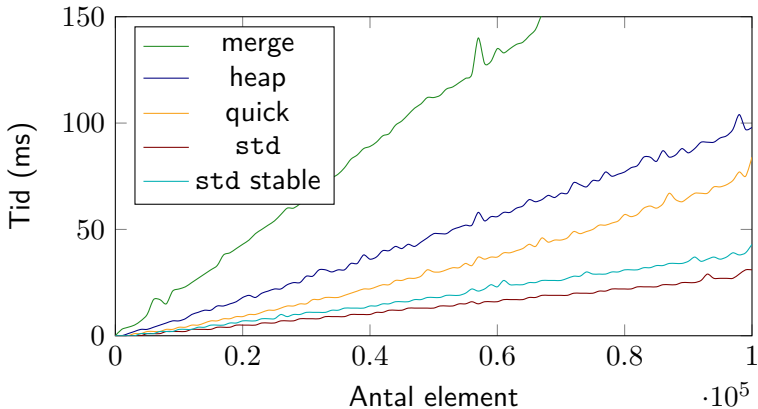
⇒ Använd ex.vis insertion sort för de små fallen i Quicksort

Timsort är ett annat exempel. Den använder mergesort, insertion sort och diverse andra metoder baserat på hur indata ser ut.

Vi testar!



De snabba algoritmerna och stora indata



- 1 Motivation
- 2 Sorteringsalgoritmer
- 3 Divide and conquer
- 4 Ännu bättre algoritmer?
- 5 Sammanfattning

Bättre än $\mathcal{O}(n \log n)$?

Vi vet att sorteringsproblemet avgränsas av $\Omega(n)$ och $\mathcal{O}(n \log n)$ (varför?).

Kan vi hitta en ännu bättre algoritm som sorterar indata med hjälp av en jämförelsefunktion?

Bättre än $\mathcal{O}(n \log n)$?

Vi vet att sorteringsproblemet avgränsas av $\Omega(n)$ och $\mathcal{O}(n \log n)$ (varför?).

Kan vi hitta en ännu bättre algoritm som sorterar indata med hjälp av en jämförelsefunktion?

Sterlings formel: $\ln(n!) \approx n \ln(n) - n$ för stora n .

Alltså: $\Omega(\log(n!)) = \Omega(n \log(n))$

Sortering som inte baseras på jämförelser

Gränsen $\Omega(n \log(n))$ gäller bara för jämförelsebaserad sortering. Det finns andra algoritmer som inte bygger på jämförelser, exempelvis:

- Count sort
- Bin sort
- Bucket sort
- Radix sort
- ...

Mer om detta nästa gång!

- 1 Motivation
- 2 Sorteringsalgoritmer
- 3 Divide and conquer
- 4 Ännu bättre algoritmer?
- 5 **Sammanfattning**

Originalproblemet

1. Sortera A och B , sätt $i, j = 0$ $\mathcal{O}(n \log(n))$
2. Titta på $A[i]$ och $B[j]$: $\mathcal{O}(1)$, n gånger
 - Om de är lika: elementet fanns i båda arrayerna
 - Om $A[i] < B[j]$: element $A[i]$ saknas i B
 - Om $A[i] > B[j]$: element $B[j]$ saknas i A

Tidskomplexitet: $\mathcal{O}(n + n \log(n)) = \mathcal{O}(n \log(n))$

Vilken algoritm ska jag välja?

Jag vill sortera en kortlek. Vilken algoritm är bäst?

Jag vill sortera ett relativt litet antal rockar som hänger i galgar efter storlek. Vilken algoritm är bäst?

I kursen framöver

- Denna vecka
 - Försök bli klar med lab 3 under denna vecka
 - Lektion på tisdag (sortering, 1 pass körs utomhus)
 - Nästa föreläsning: Sortering på $\mathcal{O}(n)$ samt andra intressanta algoritmer
- Sedan
 - Tentaförberedelse
 - För intresserade: Lab 3 på ca 0.05 s

I kursen framöver

- Uppgifter i Kattis
 - cd (enkel)
Hitta dubbletter i två cd-samlingar. Kan du utnyttja någon egenskap i indata?
 - froshweek (svårare)
Räkna antalet swaps som krävs för att sortera en viss indata.

Filip Strömbäck

www.liu.se