TDDI16 – Föreläsning 8

Sortering

Filip Strömbäck

LINKÖPING UNIVERSITY

Planering

Vecka	Fö	Lab
36	Komplexitet, Linjära strukturer	----
37	Träd, AVL-träd	1---
38	Hashning	12--
39	Grafer och kortaste vägen	12--
40	Fler grafalgoritmer	-23-
41	Sortering	--3-
42	Mer sortering, beräkningsbarhet	--34
43	Tentaförberedelse	---4

Lektion om sortering

- Se mail
- Kom ihåg att svara!

LINKÖPING UNIVERSITY

Varför behöver vi sortera data?

Varför behöver vi sortera data?

- Hitta i stora datamängder (både program och användare)
- 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?

Lösningsidé 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:

Lösningsidé 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.

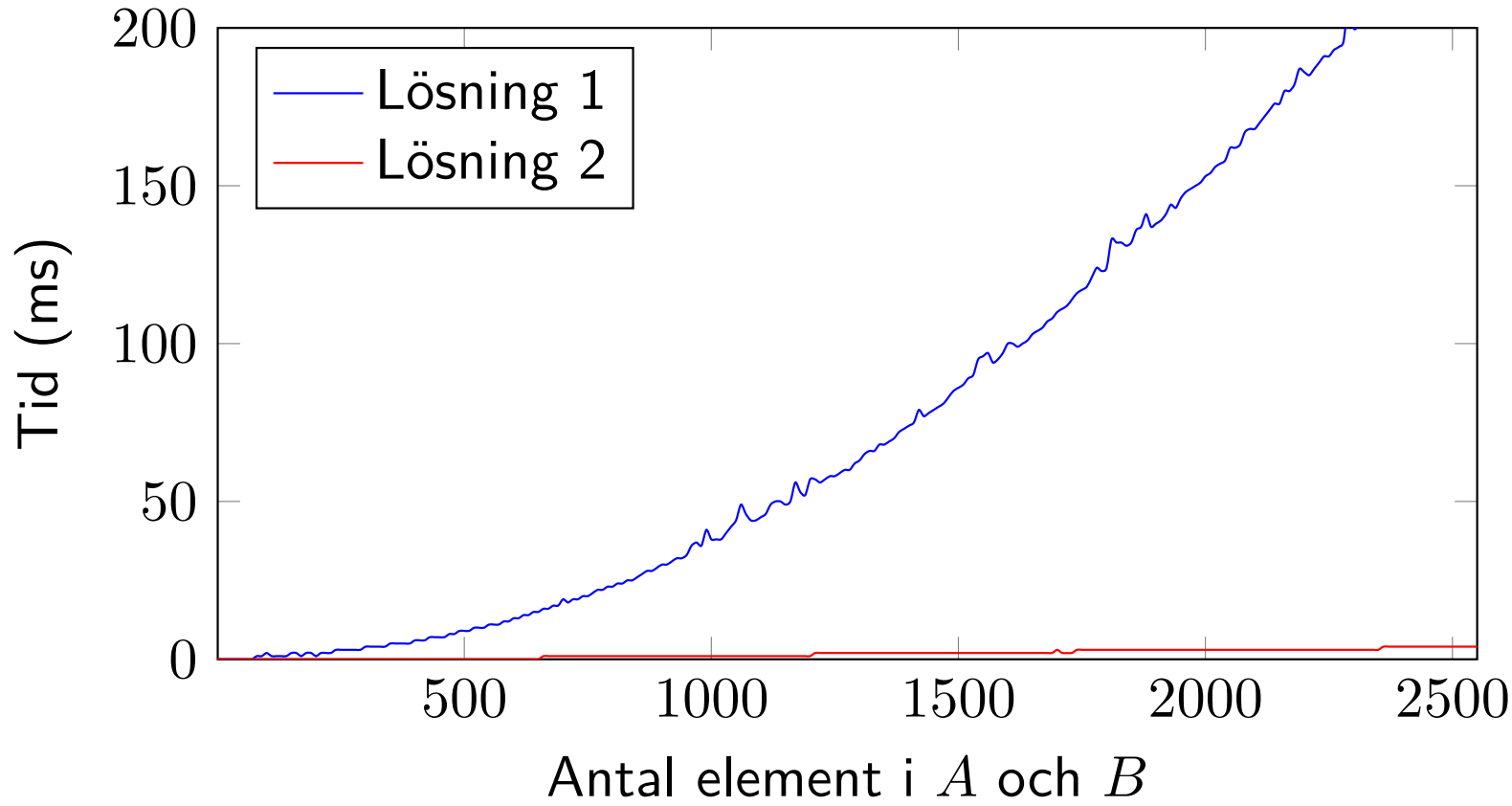
Tidskomplexitet: $\mathcal{O}(n^2)$

Lösningside 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!


Lösning 1
Lösning 2
Tid (ms)
200
150
100
50
0
500
1000
1500
2000
2500
Antal element i A och B

LINKÖPING UNIVERSITY

Formell problembeskrivning

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

Formell problembeskrivning

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

(Likt topologisk sortering, men för en *total ordning*)

Formell problembeskrivning

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

(Likt topologisk sortering, men för en *total ordning*)

- Hur många jämförelser behöver vi göra?
- Hur många element måste vi flytta på?
- Är sorteringen stabil eller inte?
- Sker sorteringen *in-place*?
- Minnesanvändning (utöver indata)?
- 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]);
    }
  }
}
```

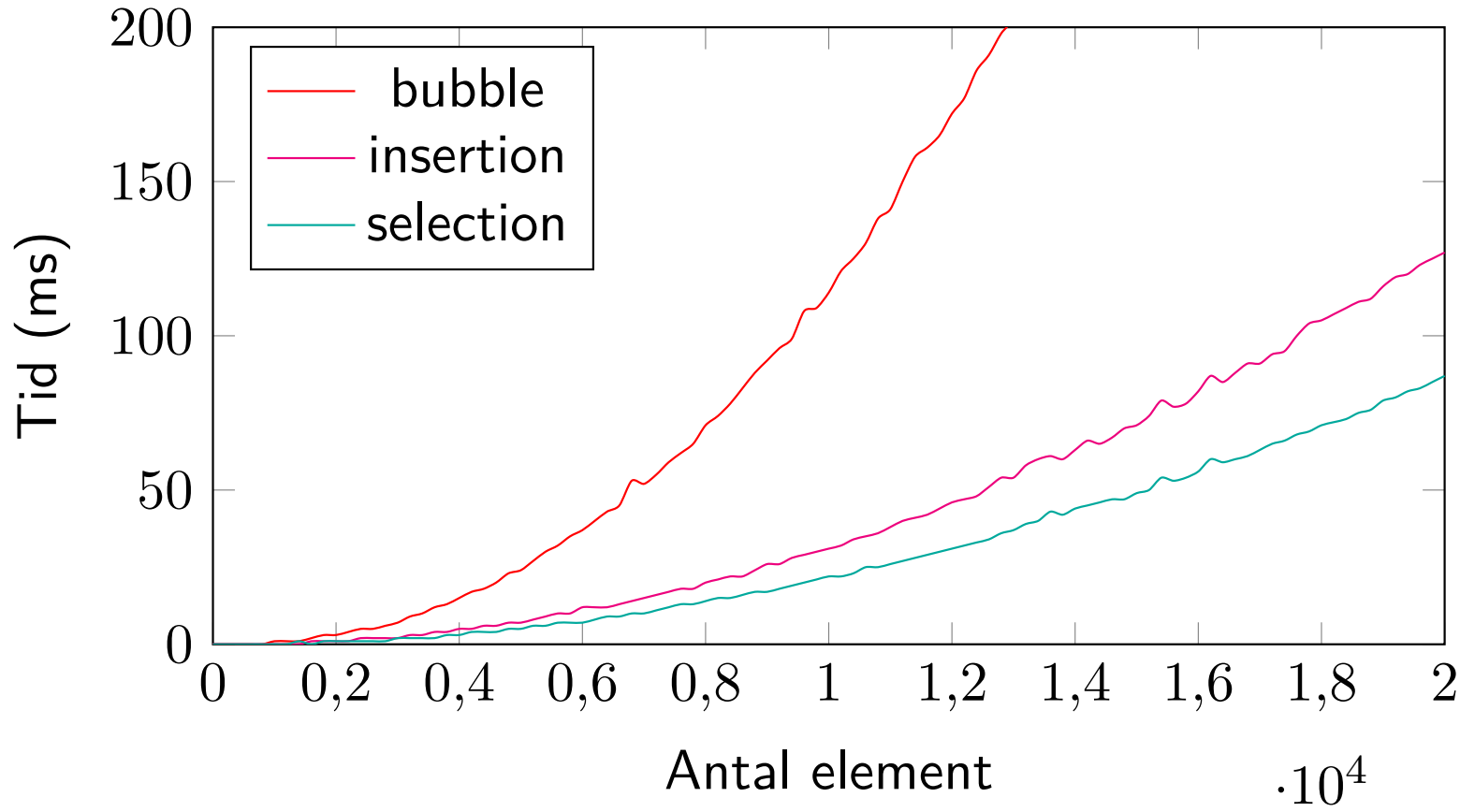
Selection sort

```
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?

Kan vi hitta en bättre lösning?

Balanserade sökträd har ju bra tidskomplexitet!

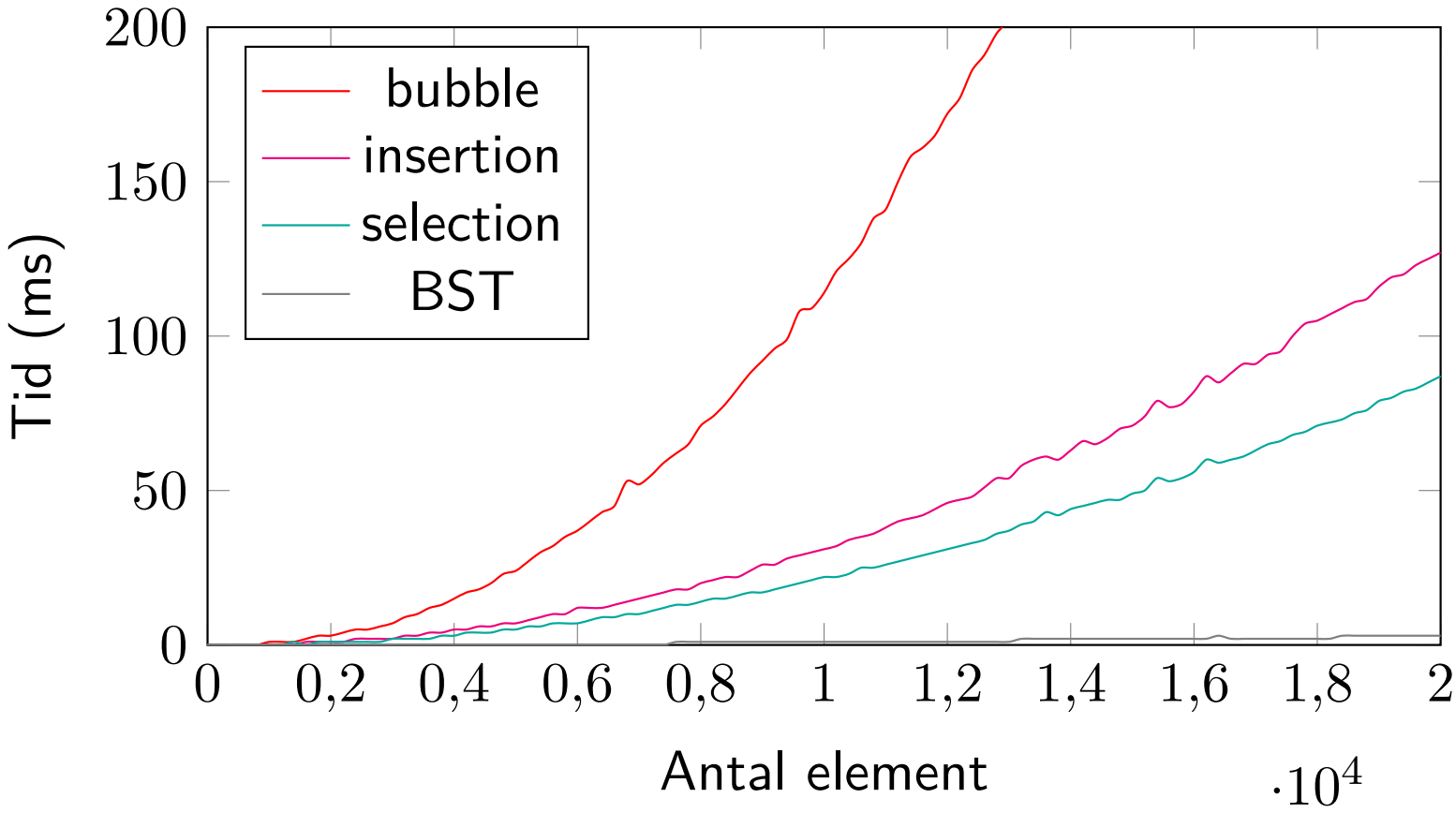
Kan vi hitta en bättre lösning?

Balanserade sökträd har ju bra tidskomplexitet!

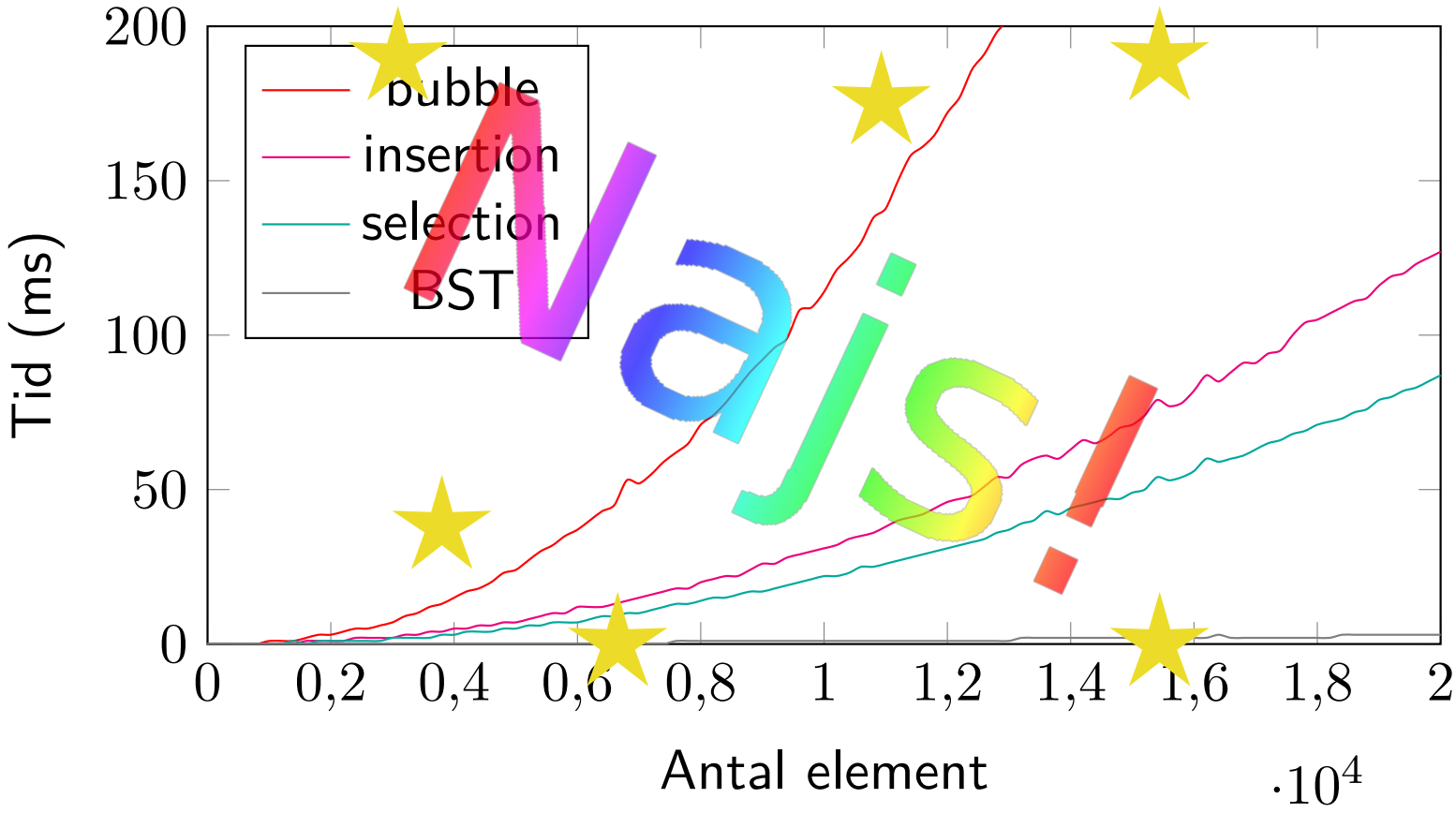
Kan vi använda ett balanserat sökträd?

```
void bst_sort(vector<int> &v) {
  multiset<int> data(v.begin(), v.end());
  size_t pos = 0;
  for (auto &&x : data)
    v[pos++] = x;
}
```

Vi testar igen!



Vi testar igen!



Kan vi hitta en bättre lösning?

Kan vi undvika länkade strukturer och extra minne?

Kan vi hitta en bättre lösning?

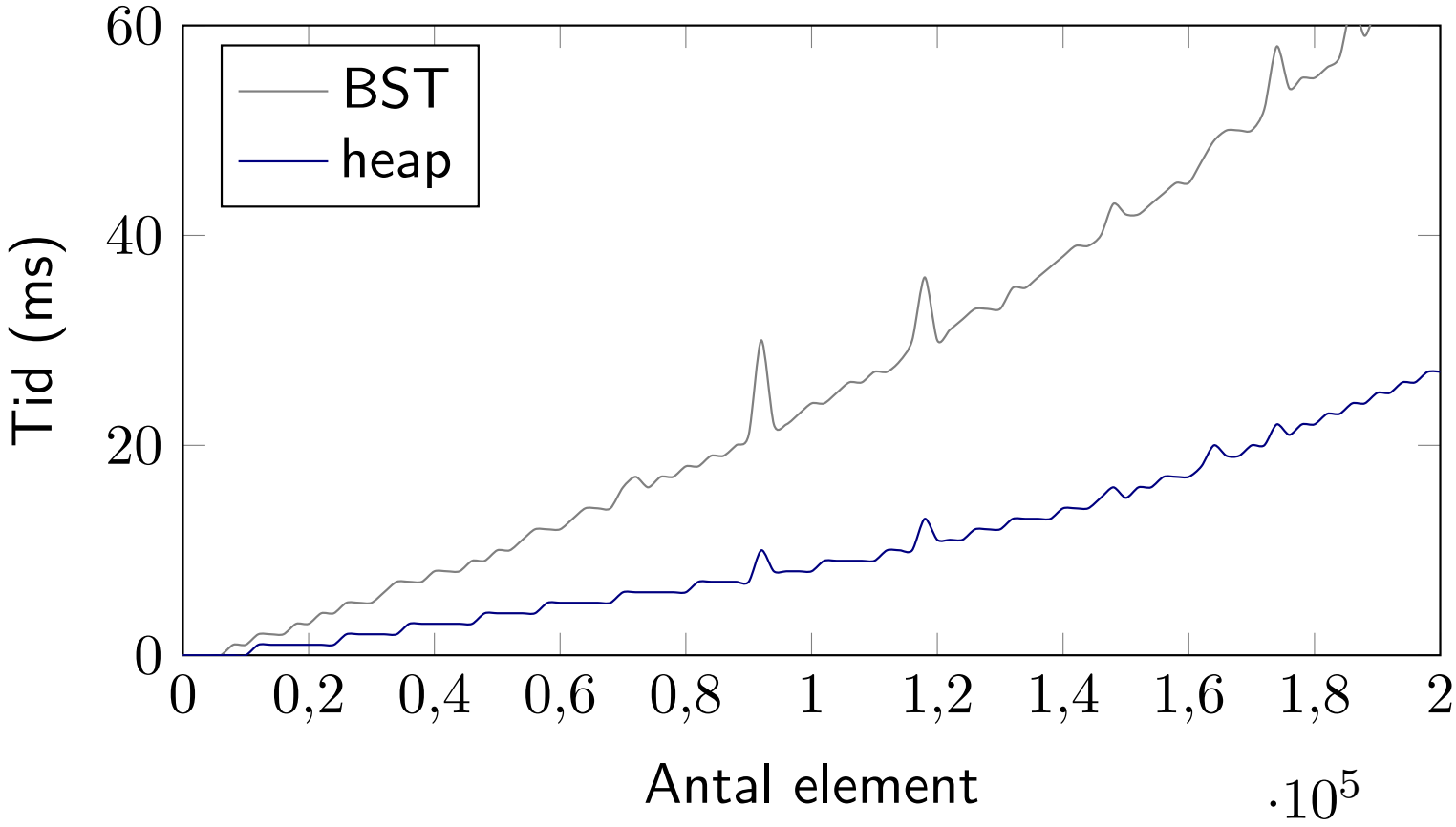
Kan vi undvika länkade strukturer och extra minne?

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);
}
```

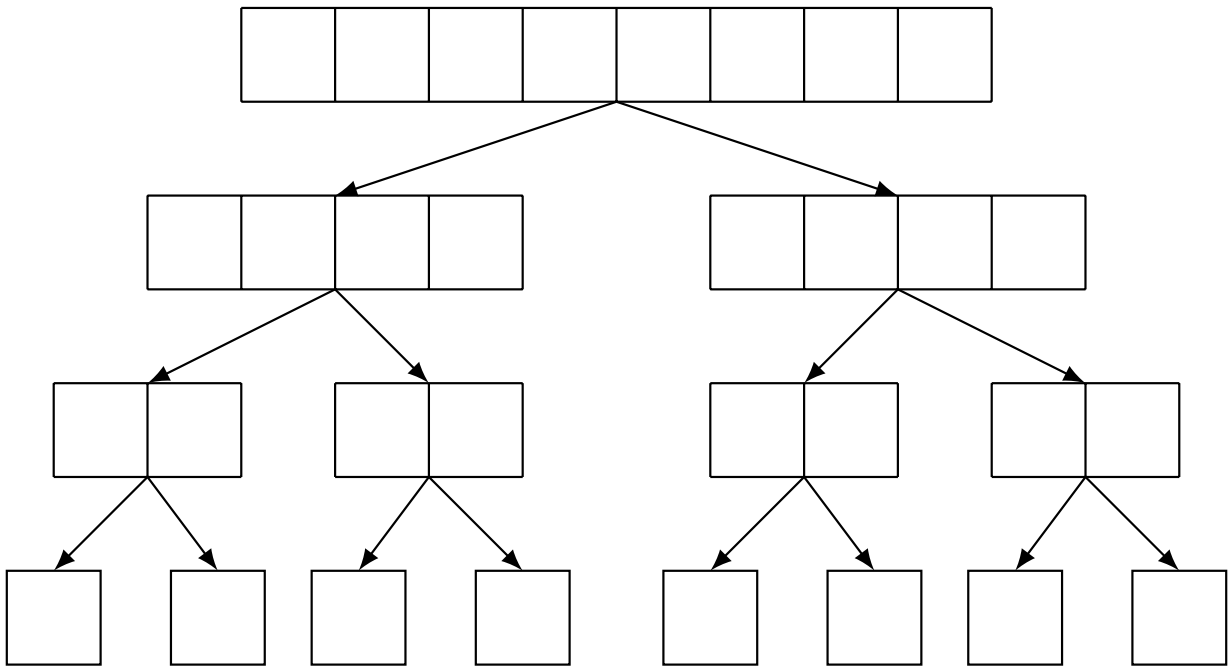
Vi testar igen!



LINKÖPING UNIVERSITY

Ny taktik – divide and conquer

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



Merge sort – Idé

1. Om indatat 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

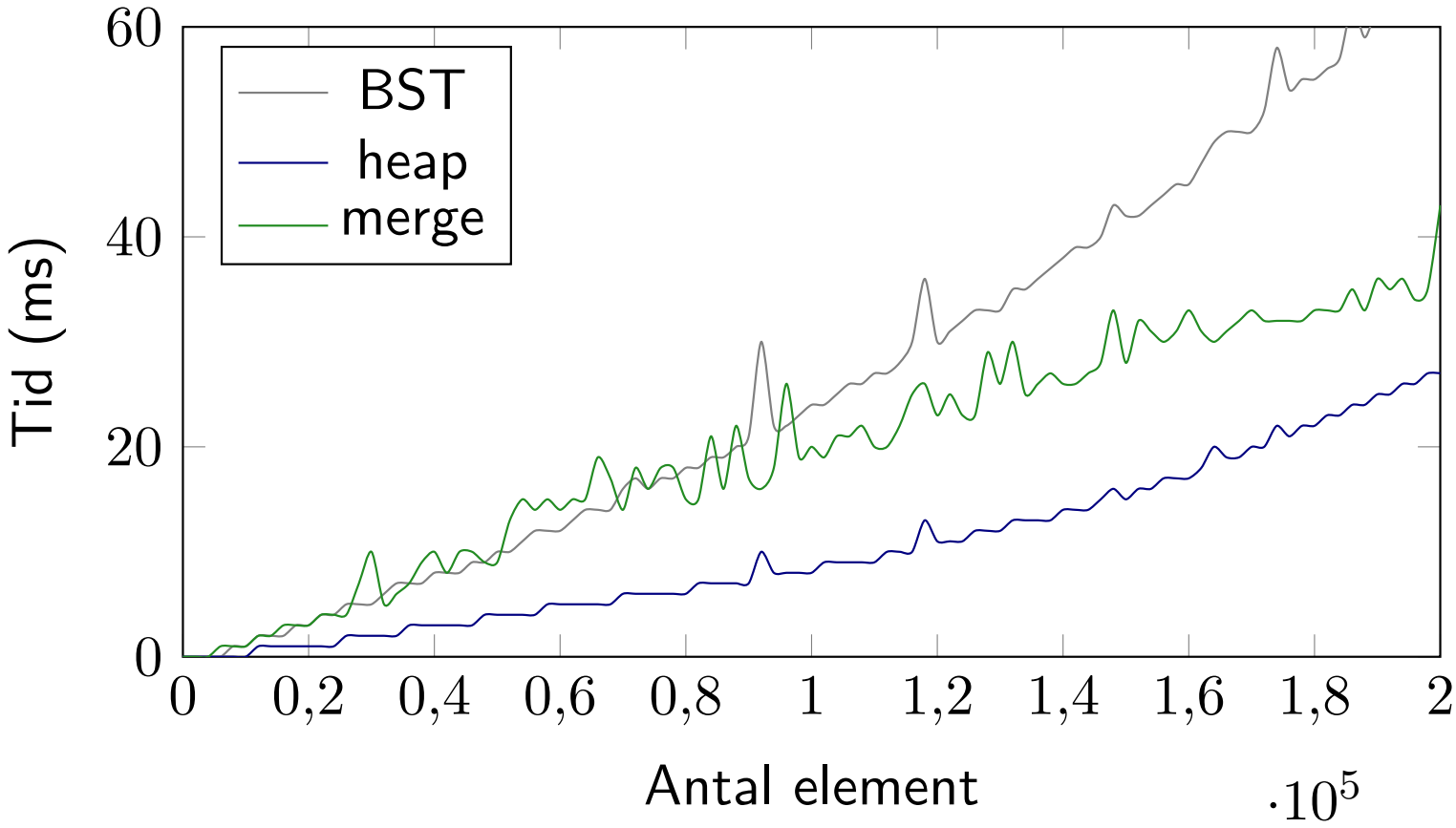
```
vector<int> merge_sort(iter begin, iter end) {
  if (end - begin <= 1)
    return vector<int>(begin, end);

  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!


BST
heap
merge
Tid (ms)
0
20
40
60
0
0,2
0,4
0,6
0,8
1
1,2
1,4
1,6
1,8
2
Antal element
$\cdot 10^5$

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 sista elementet för
  // enkelhets skull. Det finns bättre alternativ.
  iter pivot = end - 1;
  iter middle = partition(begin, end, pivot);
  // Sortera båda halvorna med quicksort.
  quicksort(begin, middle);
  quicksort(middle + 1, end);
}
```

Implementationen av `partition` finns på kurshemsidan

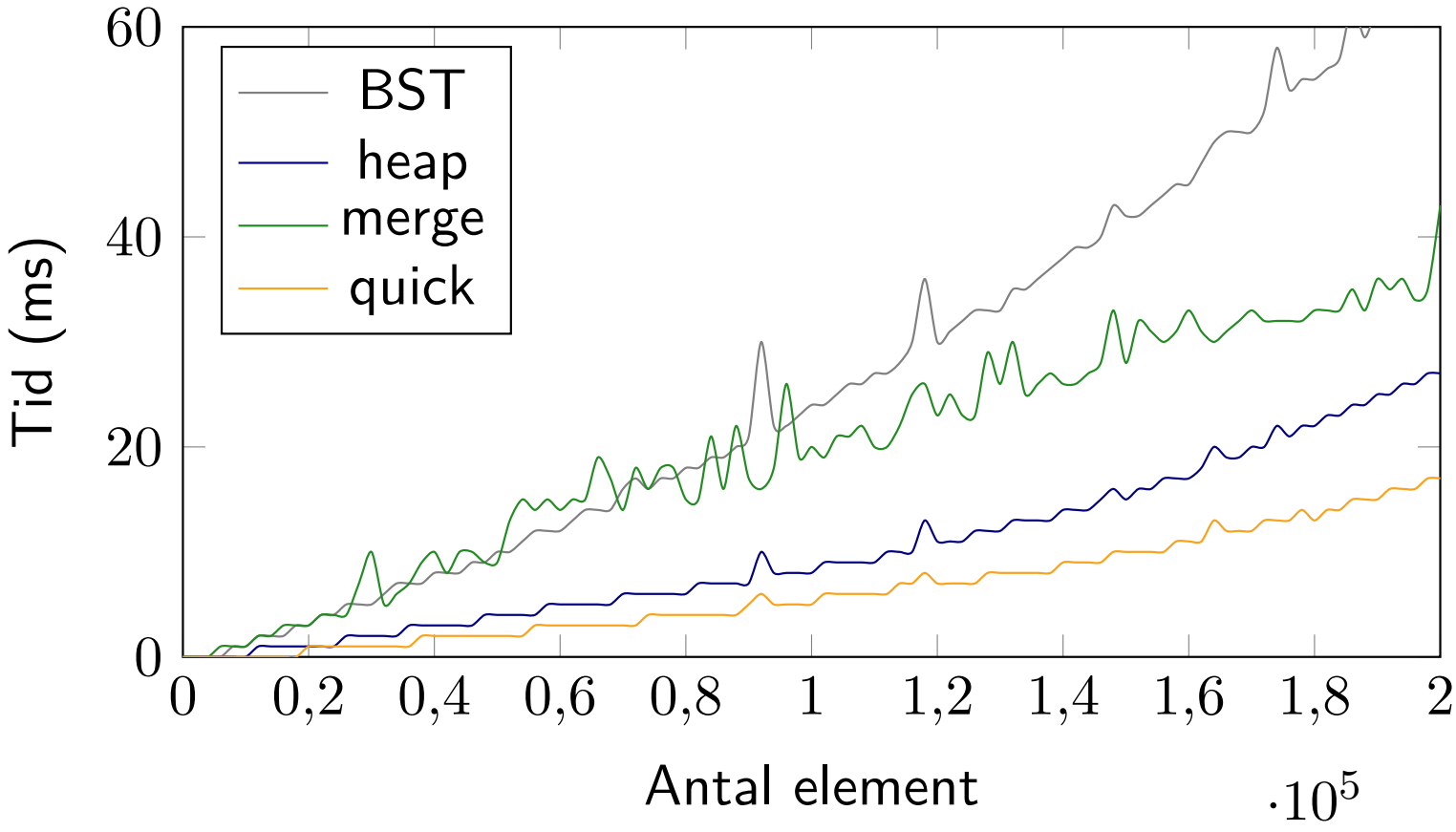
Vilket pivotelement ska vi välja?

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?

Vi testar!


60
40
20
0
Tid (ms)
BST
heap
merge
quick
0
0,2
0,4
0,6
0,8
1
1,2
1,4
1,6
1,8
2
Antal element
$\cdot 10^5$

En blandning av olika algoritmer

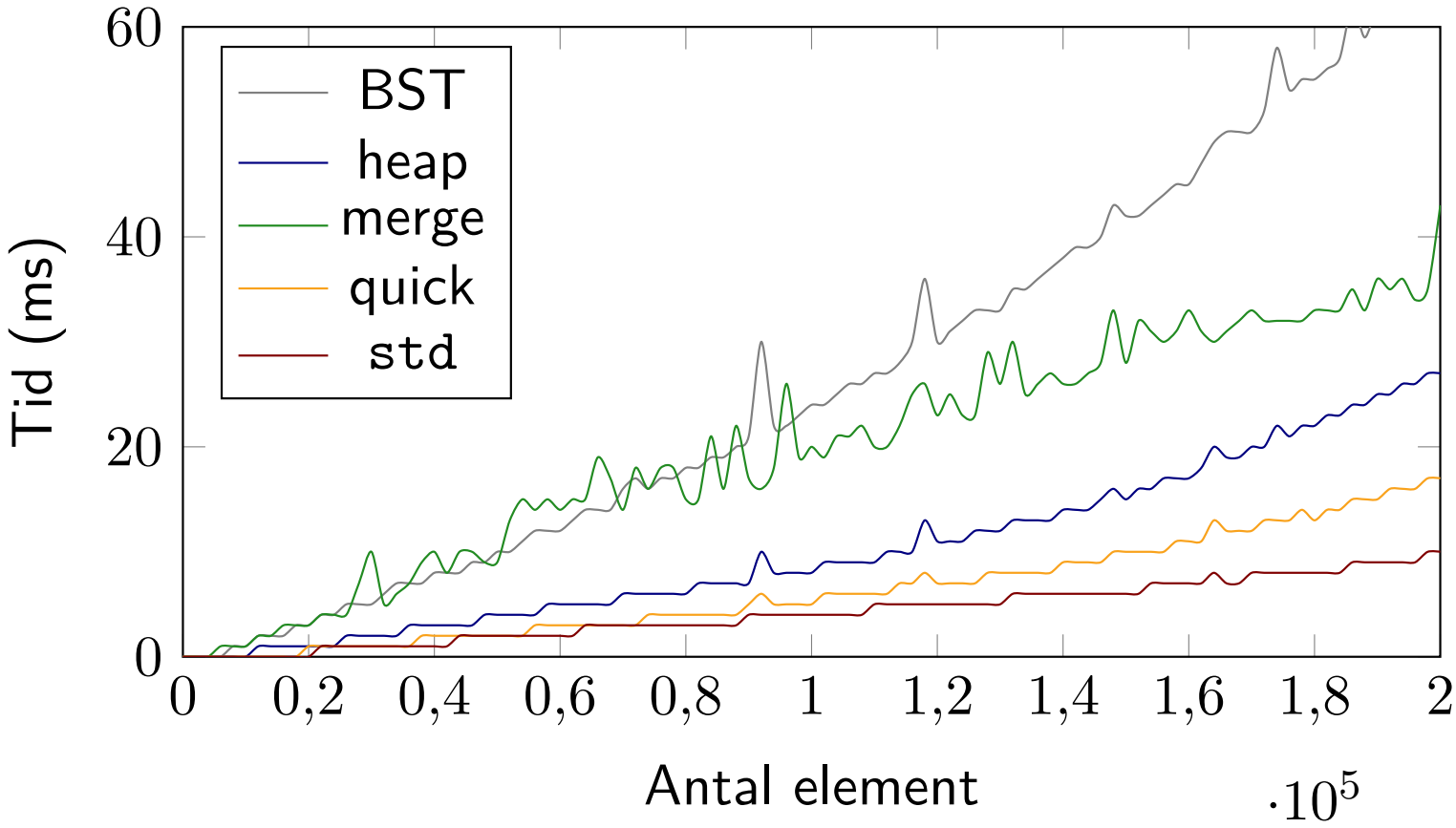
Olika algoritmer är bra i olika fall.

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

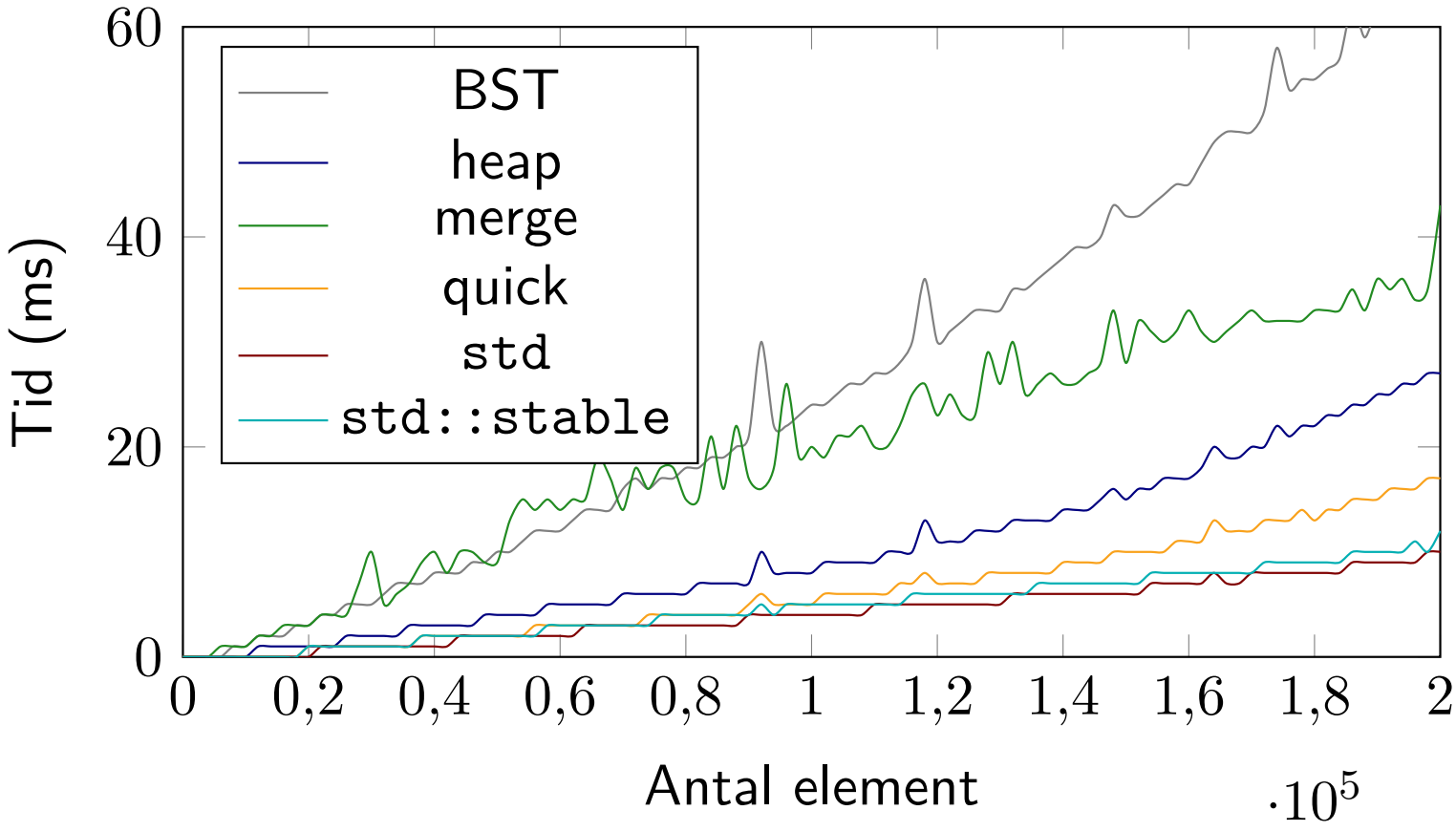
$\Rightarrow$ Använd ex.vis insertion sort/selection sort för de små fallen i Quicksort

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

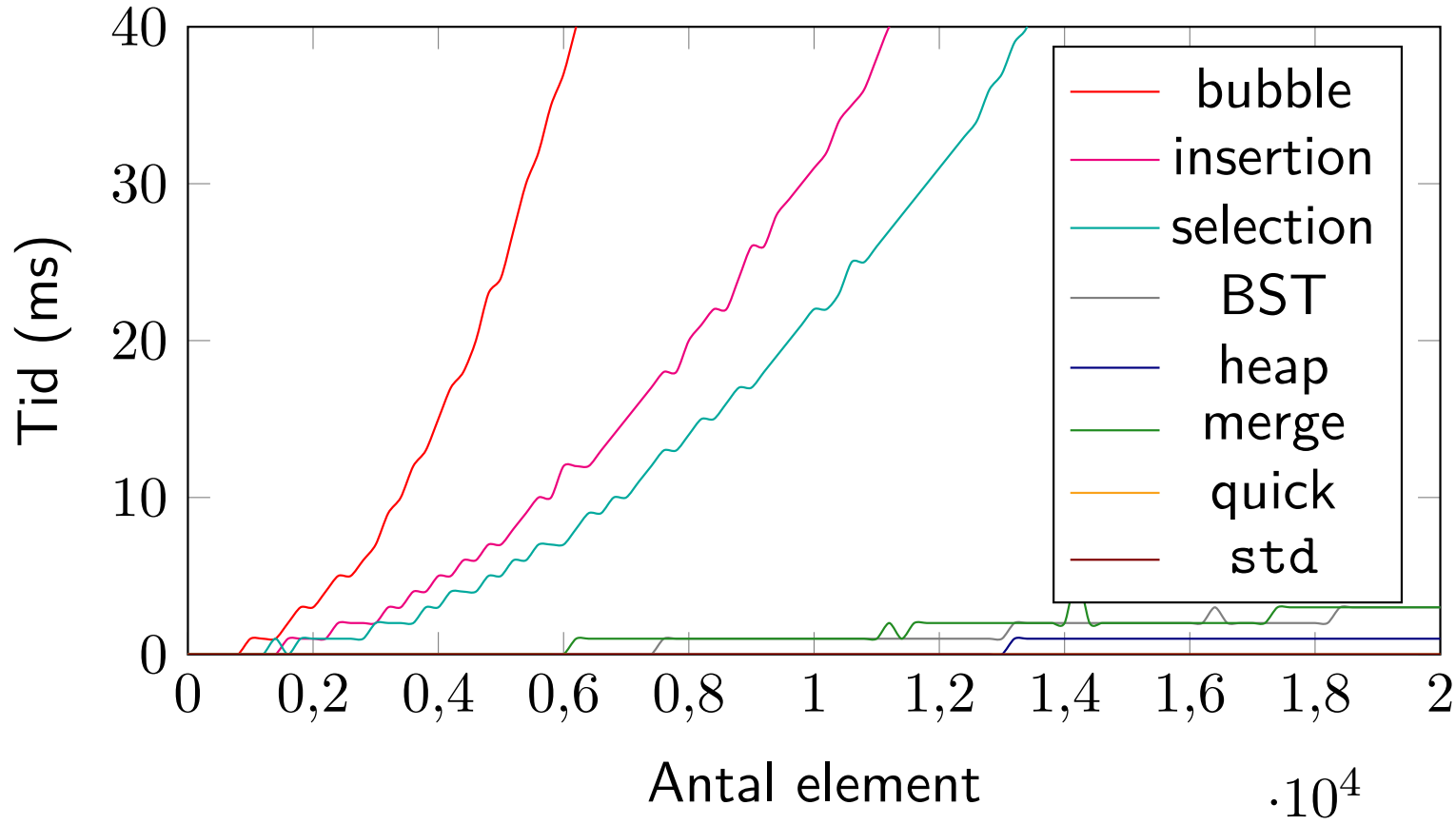
Vi testar!


60
40
20
0
Tid (ms)
BST
heap
merge
quick
std
0
0,2
0,4
0,6
0,8
1
1,2
1,4
1,6
1,8
2
Antal element
$\cdot 10^5$

Vi testar!


60
40
20
0
Tid (ms)
BST
heap
merge
quick
std
std::stable
0
0,2
0,4
0,6
0,8
1
1,2
1,4
1,6
1,8
2
Antal element
$\cdot 10^5$

Alla algoritmerna igen



LINKÖPING UNIVERSITY

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!

LINKÖPING UNIVERSITY

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

I kursen framöver

- Denna vecka
  - Lektion på måndag (sortering, möjlighet till genomgång utomhus)
  - Nästa föreläsning: Sortering på $\mathcal{O}(n)$, samt beräkningsbarhet
  - Sedan: Tentaförberedelse
- Extrauppgifter
  - 624 (enkel)
    Hitta dubbletter i två cd-samlingar. Kan du utnyttja någon egenskap i indatat?
  - 11858 (svårare)
    Räkna antalet swaps som krävs för att sortera en viss indata.

Filip Strömbäck

www.liu.se