Uke 12,
Forelesning 2

Sortering:

Sammenligning-baserte:
Baserer seg på sammenligning av elemntene i a[ ]

- Instikk, boble, utplukk
- Merge, Heap, Shell, Tree,
- Quicksort

Verdi-baserte :
Direkte plassering basert på verdien av hvert element – ingen sammenligninger med nabo-elementer e.l.

- Bøtte
- Radix
- PSort - idag

TEMA: Sortering

Avslutter sortering…

Sortere ved å lage sorterings-permutasjonen
Psort I

- Hvorfor sortere ?
  - Ikke egentlig interessert i å sortere en **int array**
  - Man sorterer sammenhengende datamengder:
    - Bank : Konto #., navn, adresse, saldo sortert på Konto #.
    - Student: Navn, adresse, institutt #, eksamen, sortert på Inst # og navn
- Antar at data er i **a[ ],b[ ],…,d[ ]**
  og at vi vil presentere **a, b,…d** sortert på **a**
- Tre løsninger:
  1. Flytte data i b,c,...d sammen med og tilsvarende man sorterer **a**
     - meget langsomt !
  2. Legg data(a_i,b_i,...,d_i) i objekt$_i$. Sorter objektene på nøkkelen a_i
     - raskt, men plassforbrukende (og derfor langsomt for store datasett)
  3. Generere sorteringspermutasjonen – **int [ ] p** fra a – slik at:
     - a[p[i]], b[p[i]], ...,d[p[i]] er det sorterte datasettet.

Psort, lager sorterings-permutasjonen

1. (som Radix: Finn max verdi i a).
2. Som Radix: Tell i array count hvor mange det er av hver verdi i a.
3. Som Radix: Adder antallene til logiske pekere i count
   count '[i] = count [i-1] + count [i-2]
4. Lag sorterings- permutasjonen: **p[count[a[i]]++] = i.**

Eks:

	0	1	2	3	4	5
a	3	3	0	5	1	6

1. max = 6

2 . Count

0	1
1	1
2	0
3	2
4	0
5	1
6	1

3 Adder til pekere

0	0
1	1
2	2
3	2
4	4
5	4
6	5

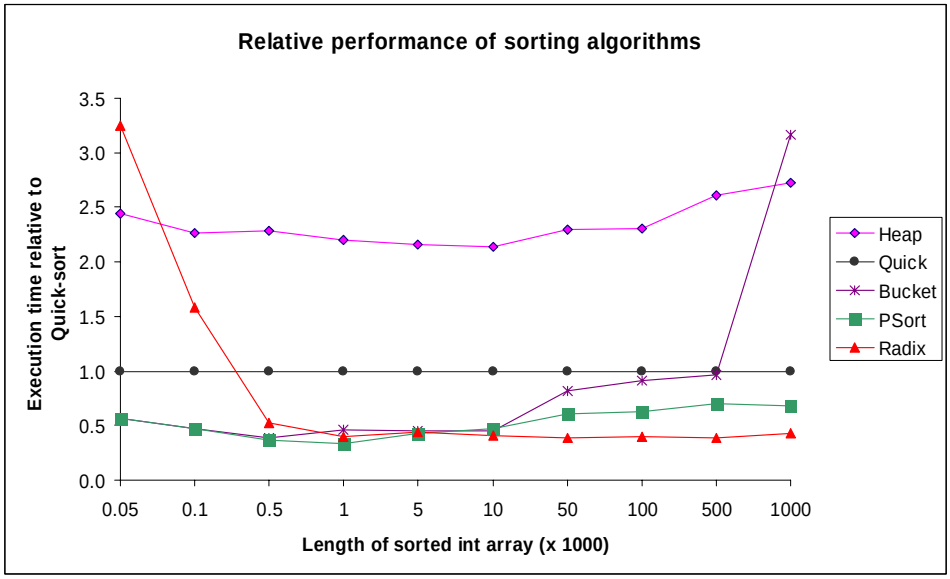
4. Lag p :

0	2
1	4
2	0
3	1
4	3
5	5

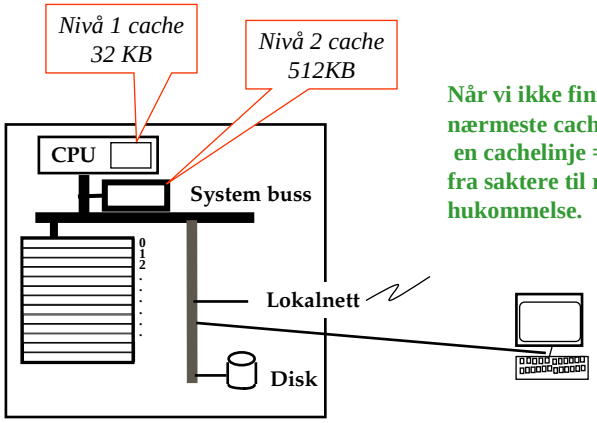

```
int [] psort1 ( int [] a )
{ int [] p = new int [n];
  int [] count ;
  int localMax = 0;
  int accumVal = 0, j, n= a.length;

    for (int i = 0 ; i < n ; i++)
     if( localMax < a[i]) localMax = a[i];

    count = new int[localMax++];

    for (int i = 0; i < n; i++)
       count[a[i]]++;

    for (int i = 0; i < localMax; i++) {
       j = count[i];
       count[i] = accumVal;
       accumVal += j;
    }

    for (int i = 0; i < n; i++)
        p[count[a[i]]++] = i;

    return p;
}
```

Sammenligning av algoritmer

- Heap-sort
- Quick-sort
  - effektivisert med Innstikksortering for subdeler < 10
- Bøtte-sortering
  - Data plassert i objekter. Disse plasseres i lister – en per mulig verdi – generering av objekter utenfor tidtagingen
- P-sort
- Radix
  - Minst signifikant siffer først, 2 pass (hvis nødvendig)
  - 10 bit fast siffer



Hva er caching ?



Når vi ikke finner data i nærmeste cache, overføres en cachelinje = 32 Byte fra saktere til raskere hukommelse.

Effekten av caching – hvor stor ?

- Ser på uttrykk som nyttes i Radix og Psort
- Radix:

```
// flytt  tallene
for (int i = 0; i < n; i++)
   til[ant[((fra[i]>>bit) & rMax)]++] = fra[i];
```

- Psort

```
// make p[]
for (int i = 0; i < n; i++)
   p[count[a[i]]++] = i;
```

- Disse gjennomløpene er ganske cache-uvennlige
  - **De 'hopper' relativt tilfeldig rundt i 'count' og 'p'**

Cache-test

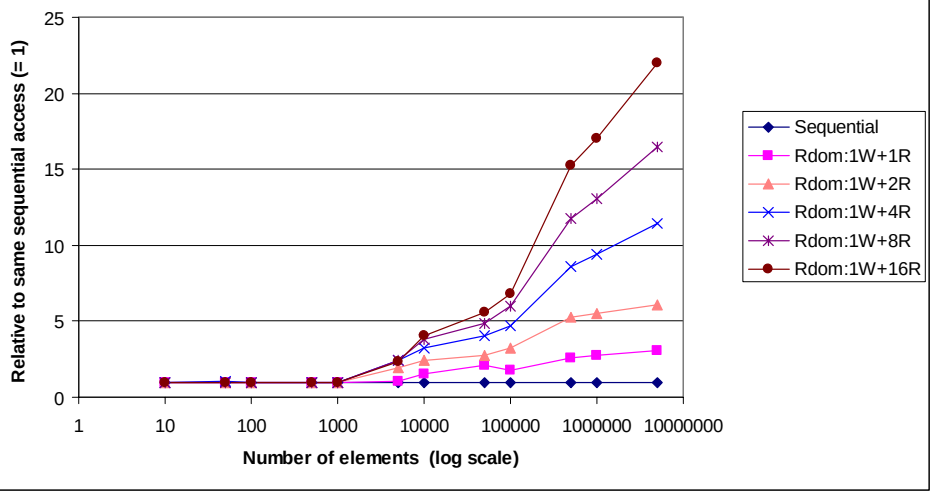
Tester uttrykk av typen med 1 skriv og k les:

```
for(int i= 0; i < n; i++)
        a[b[b[…b[i]…]]] = i ;
```

To innhold i p:

- Sekvensiell: b[i] = i
  - gir neste ingen cache-feil
- Tilfeldig innhold b[i] = tilfeldig int (0..n-1)
  - gir 'nesten' k stk. cache-feil ved økende n, først nivå 1 cache feil, så også nivå 2

Effect of caching



Sammenligning mellom Sekvensiell utførelse = 1 og Random innhold av b[i]

Caching – konklusjoner

- Opp til 22x forsinkelse med 'ekstreme' cache-feil
- Opp til over 5x forsinkelse ved normaltilfellet:
  1 skriv 2-3 les
- Ide:
  - Lag en algoritme som er cache–vennlig
  - Vi kan kanskje ha 2-3x lenger kode (= utførte instruksjoner) enn en cache-uvennlig kode og ***enda være raskere***
- Andre har laget raskere, cache vennlige varianter av tree-, heap og Quick-sort *(sammenlignings-sortering)*
- To forsøk på *verdi-baserte* algoritmer:

Raskere, cache-vennlige algoritmer ?

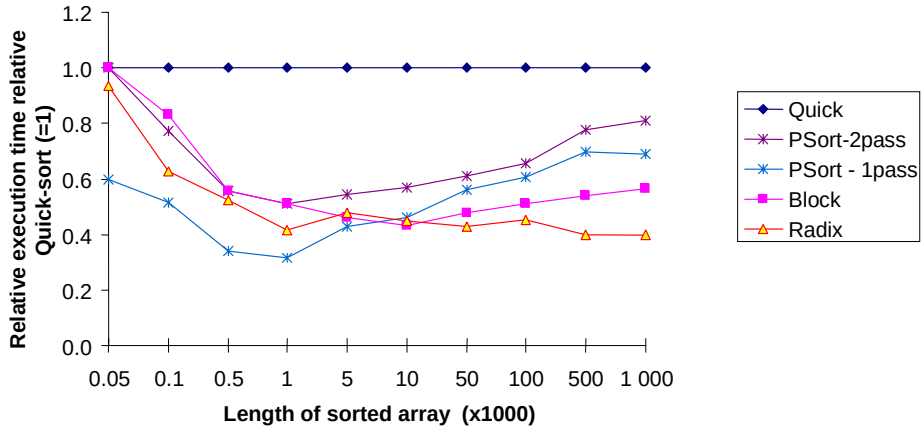
To forsøk på verdi-baserte algoritmer:

1. **2 pass Psort** (sortering max 10 bit ad gangen), kode = ca. 2x
2. **Blokk-sort** – modifisert Radix:
   - Først sorter direkte i 1024 lister med buffre på 10 mest signifikante bit
   - Så vanlig Radix-sortering på de resterende laveste bit (kode = ca. 3x)

Sammenlignet med:

- Quick-sort (= basis)
- 1 pass Psort
- Vanlig Radix-sortering
  - Nå forbedret litt ved å redusere antall bit det sorteres på ved korte arrayer

Cache sorting algorithms compared



Konklusjon

- **Introdusert ny, rask sorteringsalgoritme Psort som har ett klart og stort anvendelsesområde**
- **Caching er en vesentlig effekt ved sortering, og vel også enhver annen algoritme**
- **Ingen ny, raskere algoritme, fordi:**
  - **Ny kode lengere**
  - **Mer komplisert kode**
  - **Gammel algoritme var en blanding av cache vennlig og uvennlig kode. Veiet forbedringsmulighet var da *ikke* 5x men heller ca. 2x**
  - **Nesten like raske algoritmer med 2-3x så lang kode**

Fundamental techniques: DIVIDE AND CONQUER

Divide and conquer is a general metodology for using recurence to design efficient algorithms. It is based on dividing a particular problem into one or more subproblems of the smaller size which are then recursively solved, and then the solutions are "merged" into the solution of the original problem. Exemples:

- binary search
- quick sort
- merge sort

Dynamic programming

Dynamic programming is another technique for designing data structures and algorithms, a bit more difficult to understand than divide-and-conquer. It is a technique to try when it seems that the problem on hand is exponential time, optimization problem. Dynamic programming yields a polynomial time algorithm, usually very easy to code. The problem must have some structure that we can exploit to obtain this simple solution.

Overview: fundamental techniques

- simple subproblems: there has to be a way of breaking the problem into subproblems and defining them with few indices
- subproblem optimization: an optimal solution to global problem must be a composition of optimal subproblem solutions.
- subproblem overlap: optimal solutions to unrelated problems can contain subproblems in common

Overview: fundamental techniques

Eksamples:

- Floyd-Warshals transitive closure algorithm
- matrix chain product
- 0-1 Knapsack problem
- the longest common subsequence problem

Greedy method

Exemples:

- Dijkstra
- Prim
- Kruskal
- Huffman coding

As dynamic programming, the greedy method is applied to optimization problems. In order to solve the problem one proceeds with the sequence of choices, localy optimizing at every step. The method does not always work, buut it works for problems that possess the "greedy property" which is that the global optimum can be found by a series of local optimizations.

INGEN MØTE DEN 10. 11.

Naci Akkøk foreleser den 14. 11. 03!

Gjennomgang av eksamensoppgaver