Algoritmer och datastrukturer

F10 – Sorterings-
algoritmer

Varför sortering?

Mycket vanlig och viktig operation!

Grundfrågor...
- Implementation?
  - Datastrukturer?
- Komplexitet?

Några sorteringsalgoritmer

- Insticksssortering (insertion sort)
- Urvalssortering (selection sort)
- Bubbelsortering (bubble sort)
  - Enkelriktad
  - Dubbelriktad
  - Combosort
- Shell-sortering (shell sort)

Insticksssortering

Kan användas på olika datastrukturer
- Länkade listor
- Arrayer

Instickssortering av länkad lista

Kan implementeras med **två listor:**

- **uList** (den osorterade delen)
  - *Hela listan* från början
- **sList** (den sorterade delen)
  - *Tom* från början

Instickssortering av länkad lista (forts)

Så länge som **uList** inte är tom

- *aktuellNod* := RemoveFirst(**uList**)
- Traversera **sList** tills slutet har nåtts ***eller*** en nod med en större nyckel har hittats
- Länka in *aktuellNod* på den position i **sList** som hittades i föregående steg

Instickssortering av länkad lista (forts)

Resultat (postcondition):

- **uList** är tom
- **sList** innehåller den sorterade listan

Instickssortering av array

Låt oss försöka använda samma grundidé!

- Håll reda på två delar av arrayen
  - Sorterad sublista (första delen)
    - Från början är den sorterade delen en sublista som består av ett enda element
  - Osorterad sublista (resten)

Instickssortering av array (forts)

sorterad				k	osorterad		
Före							
sorterad <= k		k	sorterad >k		osorterad		
Efter							

Ordnad insättning i en array

- Leta reda på rätt position för elementet
  - Gå ett steg bakåt så länge som det element som ska sättas in är mindre än nästa element i arrayen
- Skapa utrymme för elementet genom att flytta alla element från och med denna position ett steg ”bakåt” i arrayen
- Lagra elementet på den nu lediga positionen

Algoritm för instickssortering av en array

Repetera tills alla element finns i den sorterade sublistan:

- Ta bort första elementet i den osorterade sublistan
- Minska den osorterade sublistan och utöka den sorterade sublistan med ett element
- Gör en ordnad insättning av elementet i den sorterade sublistan
  - Detta innebär i allmänhet att ett antal element i den sorterade sublistan flyttas ett steg ”bakåt”

Exempel: Instickssortering av array

Sortering av [G,A,Z,R,E,D]

Original	G	A	Z	R	E	D		
Efter p = 1	A	G	Z	R	E	D		1
Efter p = 2	A	G	Z	R	E	D		0
Efter p = 3	A	G	R	Z	E	D		1
Efter p = 4	A	E	G	R	Z	D		3
Efter p = 5	A	D	E	G	R	Z		4

Analys av instickssortering

n element

Enkel analys: 2 nästade loopar

- Alltså $O(n*n)$?
  - Den osorterade sublistan blir mindre och mindre
  - Den sorterade sublistan (som instickningen görs i) är tom från början men växer.
  - ➔ Blir det verkligen $O(n^2)$?

Lite matematik…

Aritmetisk summa

$$1+2+\ldots+(n-1)+n=\sum_{i=1}^{n} i$$

$$\sum_{i=1}^{n} i=\frac{n(n+1)}{2}=\frac{1}{2}(n^2+n)$$

Analys av instickssortering (forts)

Yttre loopen: Index *p* går från 2 till *n*

- För varje värde på *p* görs i värsta fall *p-1* jämförelser
  - Totalt antal jämförelser:
    $J(n) = (2-1) + (3-1) + \ldots + (n-1)$
    $= 1 + 2 + \ldots + (n-1) =$
    $= \frac{1}{2}(n-1)((n-1)+1) =$
    $= \frac{1}{2} n(n-1) =$
    $= \mathbf{O}(n^2)$

Alltså: Totalt $\mathbf{O}(n^2)$ jämförelser i värsta fallet

Urvalssortering

Fungerar bra för sortering av både arrayer och länkade listor

Undviker problemet att dataelement måste flyttas många gånger

- Hur då?

Urvalssortering (forts)

Dela arrayen i **två delar**:

- Osorterad del (första delen)
  - **Hela arrayen** från början
- Sorterad del (sista delen)
  - **Tom** från början

Algoritm för urvalssortering

- Repetera tills den osorterade delen är tom:
  - Sök det *största elementet* (= största nyckeln) i den osorterade delen
  - Låt detta element *byta plats* med det element som ligger sist i den osorterade delen
    - … om elementet inte redan ligger där
  - Minska den osorterade delen med ett

Urvalssortering (forts)

osorterad		max (>k)		k	sorterad		
Före							

osorterad				sorterad			
Efter							

Analys av urvalssortering

Antal jämförelser?

$(n\text{-}1)+(n\text{-}2)+\ldots+1 = ½*n(n\text{-}1) = ½*(n^2\text{-}n)$

➔ **O**(n^2)

Antal platsbyten?

Totalt $(n\text{-}1)$ i värsta fall

Alltså: Totalt *O(n²) jämförelser och O(n) flyttningar*

Instickssortering och urvalssortering: Genomsnittsfallet

Antal operationer i genomsnittsfallet

	Instickssortering	Urvalssortering
Jämförelser	$n^2/4+O(n)$	$n^2/2+O(n)$
Tilldelningar	$n^2/4+O(n)$	$3n+O(1)$

Shell-sortering

Grundidé?

- Urvalssortering
  - Optimerar antal förflyttningar (max *n-1* platsbyten)
  - …men i gengäld många jämförelser
- Instickssortering
  - Optimerar antal jämförelser
  - … men i gengäld många förflyttningar

➔ ”Shell sort”

- Kombinerar det bästa hos båda metoderna

Shell-sortering

Diminishing increment sort

- Instickssortering på flera mindre listor
  - Försöker flytta element närmare slutmålet med få förflyttningar
    - Först ”grovsortering”
    - … därefter ”finjustering” av elementens positioner
  - Instickssorteringen i sista steget behöver inte flytta så många element eftersom de *nästan* finns på rätt plats

Shell-sortering

Hur?

- Välj en sekvens av successivt minskande *inkrement* (exempel: 34, 12, 5, 2, 1)
- För varje inkrement *i*...
  - Betrakta hela listan som bestående av *i* ”sublistor”
    - Sublista 0 = element 0, i, 2i, ...
    - Sublista 1 = element 1, i+1, 2i+1, …
    - …
    - Sublista (i-1) = element (i-1), 2i-1, 3i-1, …
  - Instickssortera varje sublista för sig

Shell-sortering

Exempel *(i = 3)*

G	A	Z	R	E	D	O	P
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Sublista 0							
Sublista 1							
Sublista 2							

Shell-sortering

Efter ”grovsortering”

G	A	D	O	E	Z	R	P
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Sublista 0							
Sublista 1							
Sublista 2							

Shell-sortering

Hur väljs inkrementen?

- Svårt att ge generella svar
  - Välj inkrement som *inte är heltalsmultipler* av varandra

Analys av ”Shell sort”

Svårt att analysera teoretiskt

- För stora *n* är antal platsbyten ungefär $\boldsymbol{O}(n^{5/4})$
- Sedgewick: Vissa inkrementsekvenser ger $\boldsymbol{O}(n^{4/3})$ exekveringstid i värsta fallet
  - Bästa sekvensen: 1, 5, 19, 41, 109, ...

Bubbelsortering

- Sortering av array
  - Börja längst till vänster
    - Jämför de två intilliggande elementen:
      - **Om** sorteringsordningen är rätt, låt elementen vara
      - **Annars** byt plats
    - Gå ett steg till höger
  - **Repetera** ovanstående tills vi har kommit längst till höger
    - ... nu har det största elementet ”bubblat upp” till rätt position

Bubbelsortering (forts)

- Men om något tal blivit flyttat, så vet vi inte om sorterat klart eller ej
  - Upprepa processen på listan, utom sista elementet eftersom största från föregående varv är längst till höger. Avbryt när ingen förflyttning skett eller längden på osorterade delen är 0, då är listan färdigsorterad

➔ ***O(n²)***

Tidsåtgång: Undre gräns

- Finns det någon gräns för hur snabb en sorteringsalgoritm kan bli?
  - Hur snabb är den snabbast möjliga (jämförelsebaserade) sorteringsalgoritmen?

  - Följdfråga: Hur mäter vi snabbhet?
    - Antal **jämförelser**?
    - Antal **tilldelningar**?

Komplexitet för en jämförelsebaserad algoritm

- En lista med *n* element kan ordnas (permuteras) på *n!* olika sätt
- Man kan placera varje permutation i löven på ett beslutsträd, där varje nod är en jämförelse
- Alltså går det alltid att hitta en indatasekvens som kräver *log(n!)* jämförelser (höjden på trädet)

Komplexitet för en ideal jämförelsebaserad algoritm

Alltså: En jämförelsebaserad sorteringsalgoritm kräver $log_2(n!)$ jämförelser för någon indatasekvens

$log(n!) \sim n*log(n) + 1{,}44n$

Alltså: $\Omega(n*log(n))$

Nedre gräns för tidskomplexiteten för alla jämförelsebaserade sorteringsalgoritmer

Bättre algoritmer?

- Divide and conquer (söndra och härska)
  - Merge sort
  - Quicksort
- Heap sort