

Sortări avansate

CAPITOLUL VI

Cuprins

Introducere

Sortări avansate

Shellsort

Quicksort

Heapsort

Binsort și counting sort

Radix sort

Mergesort

Sortări externe

Concluzii

Exerciții

Introducere

Limita $O(n^2)$ pentru sortări este mult prea mare și trebuie depășită pentru a reduce timpii de rulare

O metodă generică de a îmbunătății performanța algoritmilor (în acest caz a algoritmilor de sortare) ar fi apelarea algoritmului pe subdiviziuni ale setului de date de intrare (metoda Divide and Conquer)

De la această idee de bază pornesc o serie de algoritmi avansați

Shellsort

Pentru algoritmi de sortare **metoda generală**, de îmbunătățire a performanțelor bazată pe Divide and Conquer ar putea fi:

împarte tabloul a în h subtablouri

pentru i de la 1 la h

sortează subtabloul a_i

sortează întregul tablou

Dacă h este prea mic, atunci subtablourile a_i pot fi prea mari => ineficiență

Dacă h este prea mare, atunci se creează prea multe subtablouri => ineficiență

Shellsort

O soluție la problema anterioară ar fi alegerea unui set de valori, în locul unei valori unice

Acesta fiind principiul sortării **Shellsort**, propusă de **Donald L. Shell** în 1959

Metoda reprezintă o perfecționare a metodei de **sortare prin inserție** directă

În această metodă de sortare, se alege un set de valori pentru h , acesta devenind un tablou, numit și **tablou de incrementi**

Shellsort

Algoritmul devine:

se determină valorile $h_{t-1} \dots h_0$ pentru divizarea tabloului a în subtablouri

pentru fiecare element h_j de la h_{t-1} la h_0

 imparte tabloul a în h_j subtablouri

 pentru i de la 1 la h_j

 sortează subtabloul a_i

sortează tabloul a

Shellsort

Pentru sortarea subtablourilor algoritmul original folosește sortarea prin inserție, aceasta fiind și cea mai întâlnită variantă pentru Shellsort

Mai rămâne o problemă de rezolvat. Cum se aleg valorile pentru tabloul de incrementi (h)?

Ideea algoritmului este să se compare inițial elementele aflate la o distanță mai mare, apoi elementele din ce în ce mai apropiate, până la o comparare a tuturor elementelor adiacente într-o sortare finală

Astfel elementele tabloului h se vor alege în **ordine descrescătoare**, elementul final fiind obligatoriu **1**

Shellsort

Care sunt valorile cele mai potrivite pentru h ?

Problema este încă deschisă cercetării. Nu există o soluție care să fie demonstrată ca fiind optimă

Donald Knuth a demonstrat totuși că și cu doar doi incrementi: $\lceil \frac{16n}{\pi} \rceil^{\frac{1}{3}}$ și 1, Shellsort este mai eficientă decât sortarea prin inserție deoarece este $O(n^{\frac{5}{3}})$ în loc de $O(n^2)$

Eficiența poate fi îmbunătățită folosind un număr mai mare de incrementi

Studii empirice arată o eficiență crescută pentru, următorul șir de valori pentru h , unde $t = \lfloor \log_2 n \rfloor - 1$:

$$h_{t-1} = 1$$

$$h_i = 3h_{i+1} + 1$$

Ex : $h = \{13, 4, 1\}$

Shellsort

Dacă Shellsort folosește sortarea prin inserție, de ce nu avem $O(n^2)$?

Ne bazăm pe două principii

- Cazul cel mai favorabil pentru sortarea prin inserție este $O(n)$, pentru șiruri gata sortate
- Printr-o alegere corectă a valorilor incrementelor se evită compararea în repetate rânduri a acelorași perechi de elemente

Obținem tablouri parțial sortate, după fiecare pas

Evităm valorile incrementelor care se divid una pe cealaltă

De ex. 8,4,2,1 – **nu** este o alegere eficientă, deoarece se compară în repetate rânduri aceleași perechi de elemente și elementele pare se compară cu cele impare doar în ultimul pas

Shellsort

h:	5	3	1
----	---	---	---

a	10	8	6	20	4	3	22	1	0	15	16
Sub-tablourile inainte de sortarea cu h[0]=5	10	-	-	-	-	3	-	-	-	-	16
		8	-	-	-	-	22				
			6	-	-	-	-	1			
				20	-	-	-	-	0		
					4	-	-	-	-	15	
Sub-tablourile dupa sortarea cu h[0]=5	3	-	-	-	-	10	-	-	-	-	16
		8	-	-	-	-	22				
			1	-	-	-	-	6			
				0	-	-	-	-	20		
					4	-	-	-	-	15	

Shellsort

h: 5 3 1

Dupa sortarea anterioara	3	8	1	0	4	10	22	6	20	15	16
Sub-tablourile inainte de sortarea cu $h[1]=3$	3	-	-	0	-	-	22	-	-	15	
		8	-	-	4	-	-	6	-	-	16
			1	-	-	10	-	-	20		
Sub-tablourile dupa sortarea cu $h[1]=3$	0	-	-	3	-	-	15	-	-	22	
		4	-	-	6	-	-	8	-	-	16
			1	-	-	10	-	-	20		
Dupa sortarea anterioara	0	4	1	3	6	10	15	8	20	22	16
Sub-tablourile dupa sortarea cu $h[2]=1$	0	1	3	4	6	8	10	15	16	20	22

Shellsort

Propunere implementare în C:

```
void shell_sort.tip_element a[], int n)
{int i, j, k, hCnt, H; tip_element tmp;
generare_incrementi(h, T);
//pentru fiecare increment
for (i = 0; i < T; i++)
{ H = h[i]; //incrementul curent
//pentru fiecare subtablou
for (hCnt = 0; hCnt < H; hCnt++)
{ //insetion_sort pentru subtablou
for (j = H+hCnt; j < n; j+=H)
//se selecteaza elementele cu pasul H
{tmp = a[j];
for (k = j; (k-H>=0) && tmp.cheie <
a[k - H].cheie; k-=H)
a[k] = a[k - H];
a[k] = tmp; }
}
}
}
```

```
// T este dimensiunea
//tabloului de incrementi
#define T 3
int h[T];
void generare_incrementi(int
h[], int t)
{
int i;
//generare tablou de
incrementi
h[T - 1] = 1;
for (i = T - 2; i >=0; i--)
{
h[i] = 3 * h[i + 1] + 1;
}
}
```

Shellsort

Avantaje

- O sortare mai eficientă decât cele din categoria sortărilor directe (simple)

Dezavantaje

- Performanță mai slabă față de sortările avansate studiate în continuare

Quicksort

Metoda de sortare **Quicksort** (sortare rapidă) este numită așa deoarece, dacă este corect implementată este cea mai rapidă metodă “in situ” de sortare, pentru caz general, considerând **cazul mediu**

Este cea mai răspândită metodă de sortare, implementată în general în bibliotecile standard (qsort)

Este numită și sortare **prin partiționare** pentru că se bazează pe tehnica **Divide and Conquer**

Quicksort

Shellsort împărțea tabloul în subtablouri pe care le sorta separat, ca mai apoi să reconsidere tabloul întreg, care era din nou împărțit în alte subtablouri ce urmau să fie sortate individual, ș.a.m.d.

Quicksort împarte într-un mod mult mai eficient tabloul.

Tabloul este împărțit în două subtablouri, în funcție de valoarea elementelor raportată la un element ales, numit **pivot**.

Quicksort

Principiul de funcționare:

În tablou se formează două subtablouri în funcție de valoarea elementelor raportată la pivot în felul următor: primul subtablou conținând elemente mai mici sau egale cu valoare pivotului și al doilea subtablou conținând elemente egale sau mai mari decât valoarea pivotului. Acest proces poartă numele de **partiționare**

Se apelează algoritmul pe fiecare subtablou în parte. Acestea la rândul lor se partiționează în două subtablouri fiecare, față de un pivot nou ales, algoritmul continuând într-un mod recursiv până se ajunge la subtablouri de un singur element

Quicksort

Algoritmul procedurii de partiționare:

Fie x un **element oarecare** al tabloului de sortat a_1, \dots, a_n .

Se parcurge tabloul de la **stânga spre dreapta** până se găsește primul element $a_i > x$.

În continuare se parcurge tabloul de la **dreapta spre stânga** până se găsește primul element $a_j < x$.

Se **interschimbă** între ele elementele a_i și a_j .

Se continuă parcurgerea tabloului de la **stânga** respectiv de la **dreapta**, din punctele în care s-a ajuns anterior, până se găsesc alte două elemente care se interschimbă, ș.a.m.d.

Procesul se termină când cele două parcurgeri se **"întâlnesc"** undeva în interiorul tabloului.

Efectul final este acela că șirul inițial este **partiționat** într-o **partiție stânga** cu chei mai mici sau egale cu x și o **partiție dreapta** cu chei mai mari sau egale cu x .

Quicksort

În continuare, cu ajutorul partiționării, **sortarea** se realizează simplu:

După o primă partiționare a secvenței de elemente se aplică același procedeu celor două partiții rezultate.

Apoi celor patru partiții ale acestora, ș.a.m.d.

Procesul se termină când fiecare partiție se reduce la un singur element.

Quicksort

Pseudocod:

quicksort (vector[])

 daca lungime(vector) > 1

 alege pivot;

 cat timp mai sunt elemente in vector

 include elemental fie in subtabloul1 = { el: el <= pivot};

 sau in subtabloul2={el: el>=pivot};

 quicksort (subtabloul1);

 quicksort (subtabloul2);

Quicksort

Problema alegerii pivotului nu este simplă

În primul rând se dorește o împărțire cât mai egală a celor două subtablouri din punct de vedere al numărului de elemente

Dacă se alege unul din capetele intervalului, împărțirea este dezechilibrată și duce la o degradare a performanțelor din punct de vedere al timpului de execuție

În funcție de metoda de alegere a pivotului avem mai multe variante de implementare pentru Quicksort

Quicksort

O variantă propusă de implementare în C:

```
void quicksort(tip_element a[], int
prim, int ultim)
{ int stanga = prim + 1;
  int dreapta = ultim;
  //alegere pivot
  swap(&a[prim], &a[(prim + ultim) /
2]);
  //mutare pivot pe prima pozitie
  tip_element pivot = a[prim];
  while (stanga <= dreapta)
//partitionare
  {
    while (a[stanga].cheie <
pivot.cheie)      stanga++;
    while (pivot.cheie <
a[dreapta].cheie) dreapta--;
    //codul se continua ...
```

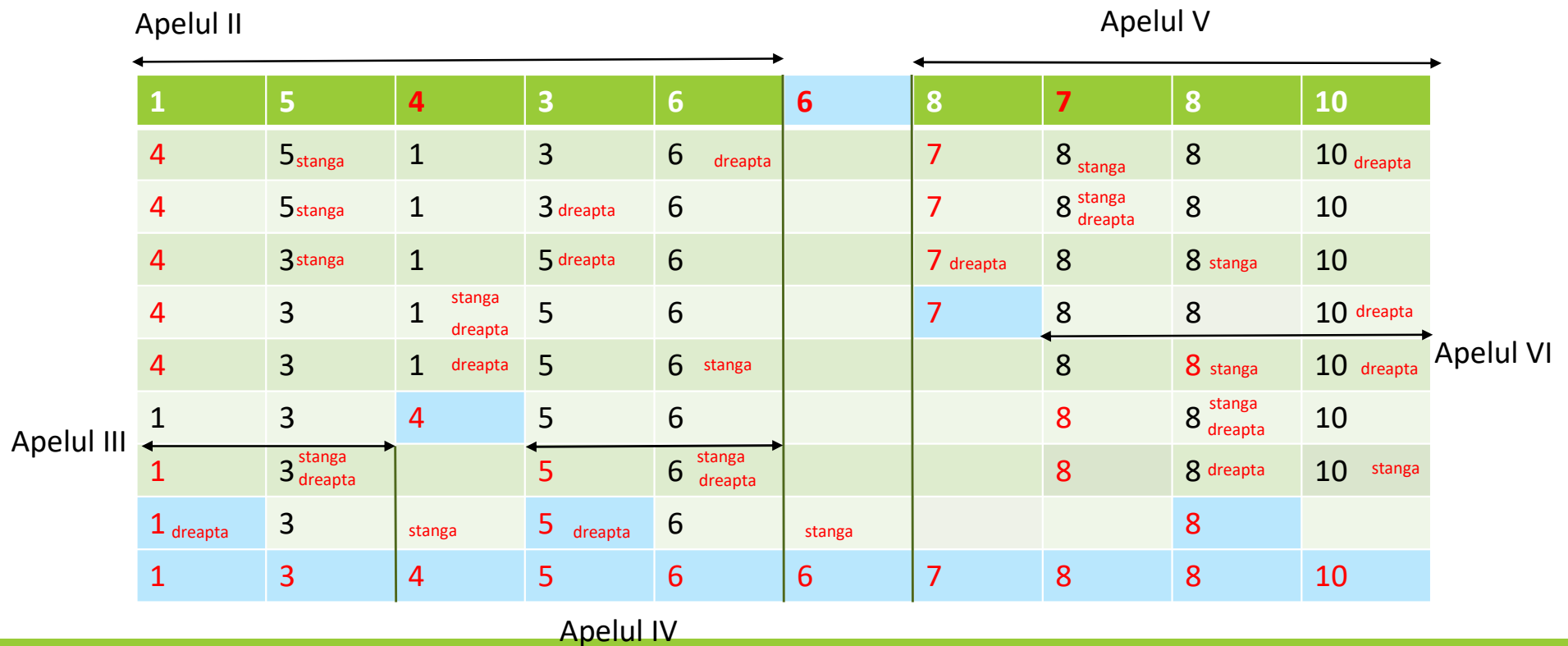
```
    //continua cod
    if (stanga < dreapta)
      swap(&a[stanga++], &a[dreapta--]);
    else
      stanga++;
  }
  //mutare pivot la locul sau final
  swap(&a[dreapta], &a[prim]);
  //apelurile recursive
  if (prim < dreapta - 1)
    quicksort(a, prim, dreapta - 1);
  if (dreapta + 1 < ultim)
    quicksort(a, dreapta + 1, ultim);
}
```

Quicksort

Apelul I

8	5	4	7	6	1	6	3	8	10
6	5 _{stanga}	4	7	8	1	6	3	8	10 _{dreapta}
6	5	4	7 _{stanga}	8	1	6	3 _{dreapta}	8	10
6	5	4	3	8 _{stanga}	1	6 _{dreapta}	7	8	10
6	5	4	3	6 _{stanga}	1	8 _{dreapta}	7	8	10
6	5	4	3	6 _{stanga}	1 _{stanga dreapta}	8	7	8	10
1	5	4	3	6	6 _{dreapta}	8	7 _{stanga}	8	10

Quicksort



Quicksort

Ca orice program implementat într-o variantă recursivă, și funcția propusă anterior poate fi implementat și într-o variantă nerecursivă

Mai multe despre tehnicile de implementare a funcțiilor recursive în variante nerecursive, veți găsi în capitolul de recursivitate

Putem avea mai multe variante de implementare a algoritmului de sortare Quicksort în funcție de modul în care alegem pivotul și în funcție de modul în care facem partiționarea

Quicksort

Analiza algoritmului Quicksort

Cel mai **favorabil** caz îl avem când pivotul împarte tabloul în subtablouri de aproximativ $n/2$ elemente

$$C_{min} = n + 2\frac{n}{2} + 4\frac{n}{4} + \dots + n\frac{n}{n} = n(\log n + 1) = O(n \log n)$$
$$M_{min} = O(n \log n)$$

Cel mai **defavorabil** caz îl avem când pivotul are valoarea maximă sau minimă, astfel fiecare pas va partaja secvența formată din n elemente, într-o partiție cu $n - 1$ elemente și o partiție cu un singur element.

$$C_{max} = n + C(n - 1) = n + n - 1 + C(n - 2) = \dots = n + n - 1 + n - 2 + \dots + 1 = \frac{n(n + 1)}{2} = O(n^2)$$
$$M_{max} = O(n^2)$$

Quicksort

Avantaje

- O sortare eficientă pentru caz general $O(n \log n)$
- O sortare "in situ"

Dezavantaje

- Un algoritm recursiv, se folosește de stivă
- Performanță slabă pentru cazul cel mai defavorabil $O(n^2)$

Heapsort

Face parte tot din categoria sortărilor avansate, dar față de Quicksort, acest algoritm nu se bazează pe recursivitate

Algoritmul se bazează pe un principiu asemănător cu algoritmul de sortare prin selecție (selection sort) și anume acela de a determina elementul cu cheia minimă, respectiv maximă

Acest algoritm selectează elementul cu cheia maximă și îl împinge spre finalul tabloului, spre deosebire de sortarea prin selecție care caută elementul cu cheia minimă pentru a-l aduce spre începutul tabloului

O altă diferență semnificativă este că Heapsort aduce o îmbunătățire față de algoritmul de sortare prin selecție, prin faptul că la fiecare trecere se vor reține mai multe informații, nu doar elementul cu cheia minimă, ca în cazul sortării prin selecție

Heapsort

Heapsort se folosește de conceptul de ansamblu (heap), după care este și denumit

Un ansamblu este un arbore binar cu următoarele proprietăți (max-heap):

- Valoarea cheii fiecărui nod este mai mare sau egală cu valorile cheilor fiilor săi
- Arborele este perfect echilibrat, nodurile frunză de pe ultimul nivel fiind situate în cele mai din stânga poziții

Dacă vorbim despre o relație inversă: valoarea cheii fiecărui nod este mai mică sau egală cu valorile cheilor fiilor săi, acesta se numește min-heap

Heapsort

Altfel spus, un ansamblu (max-heap) este definit ca o secvență de chei h_0, h_1, \dots, h_{n-1} care se bucură de proprietățile:

$$\begin{aligned}h_i &\geq h_{2i+1} \\h_i &\geq h_{2i+2}\end{aligned}$$

pentru toți $i = 0, \dots, (n-1)/2-1$

Pentru min-heap, inegalitățile vor fi invers ($h_i \leq h_{2i+1}$ și $h_i \leq h_{2i+2}$)

Un **ansamblu** poate fi asimilat cu un **arbore binar parțial ordonat** și reprezentat printr-un **tablou**

Heapsort

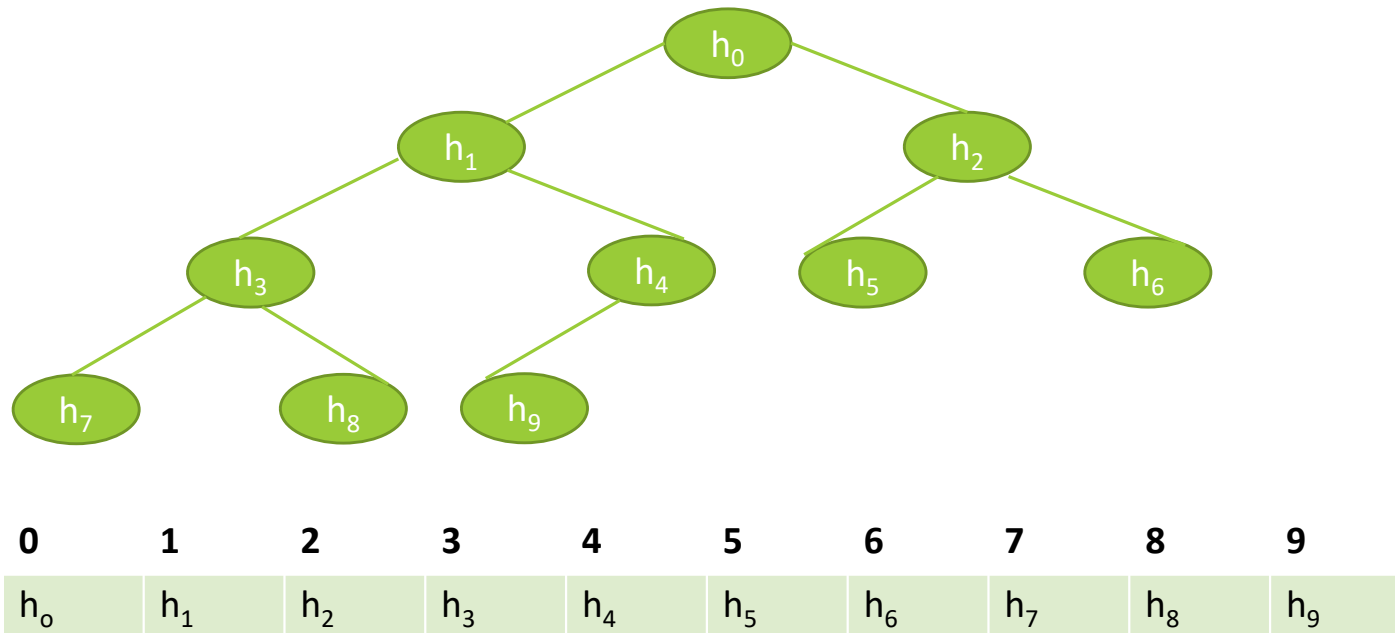
Pentru ca implementarea algoritmului să fie eficientă și din punct de vedere al spațiului, nu doar al timpului, scopul este să nu folosim alte structuri suplimentare, ci să alegem o variantă de implementare "in situ"

Atfel, nu vor fi folosite nici structuri suplimentare de tip arbore, nici alte tablouri

Cum ne vom folosi de structura arborescentă în acest caz?

Heapsort

Tabloul de sortat va fi privit ca un arbore, pe care trebuie să îl ducem la o structură de tip ansamblu



Heapsort

Elementele dintr-un ansamblu nu sunt toate ordonate

Știm doar că elementul cu cheia maximă este rădăcina arborelui (pentru max-heap) și că pentru fiecare nod, cheile fiilor săi sunt mai mici sau egale cu cea a părintelui

Heapsort pornește astfel de la un ansamblu, pune elementul cu cheia cea mai mare la sfârșit, îl ignoră și reface ansamblul care acum are cu un element mai puțin

Atfel în fiecare tură un element ajunge în poziția sa finală și ansamblul devine mai mic cu un element

Heapsort

Pseudocod:

heapsort (a[], n)

 transforma datele intr-un ansamblu

 pentru $i = n-1$; $i > 1$; $i--$

 schimba elemental radacina cu elemental de pe pozitia i

 refa ansamblul pentru a[0],...a[i-1]

Heapsort

Pentru formarea ansamblului se va folosi o metodă de tip buttom-up, "in situ", propusă de R.W.Floyd:

- Se consideră un tablou h_0, \dots, h_{n-1} care conține cele n elemente din care se va construi ansamblul.
- În mod evident, elementele $h_{n/2}, \dots, h_{n-1}$ formează deja un **ansamblu** deoarece **nu** există nici o pereche de indici i și j care să satisfacă relația $j=2*i+1$ (sau $j=2*i+2$).
- Aceste elemente formează cea ce poate fi considerat drept **șirul de bază** al **ansamblului** asociat.
- În continuare, ansamblul $h_{n/2}, \dots, h_{n-1}$ este **extins spre stânga**, la fiecare pas cu câte un element, introdus în vârful ansamblului și deplasat până la locul său.

Heapsort

Prin urmare, considerând că tabloul inițial este memorat în h , procesul de generare "*in situ*" al unui **ansamblu** poate fi descris prin secvența următoare:

Pseudocod:

Floyd($a[]$)

$stanga = n/2 - 1$

 pentru $i = stanga; i \geq 0; i--$

$stanga = stanga - 1$

deplasare ($a, i, n - 1$)

Heapsort

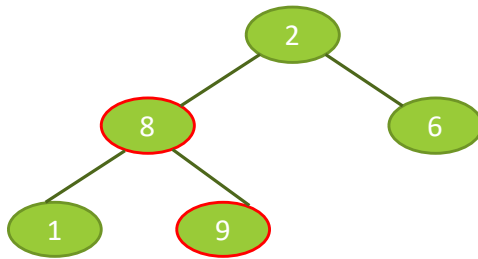
Variantă de **implementare în C** propusă:

```
void heapsort(tip_element a[],int n)
{
    int i;
    //algoritmul lui Floyd
    for (i = n / 2 - 1 ; i >= 0; i--) //se creaza ansamblul
        deplasare(a, i, n - 1);
    //extragerea maximului si refacerea ansamblului
    for (i = n-1; i >= 1; i--)
    {
        swap(&a[0], &a[i]);
        //mutare element maxim pe pozitia a[i]
        deplasare(a, 0, i - 1);
        //se reface proprietatea de ansamblu
    }
}
```

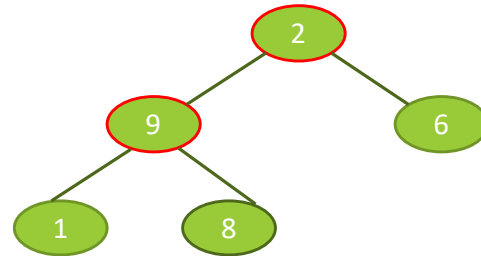
Heapsort

```
void deplasare(tip_element a[], int stanga, int dreapta)
{
    int fiu = 2 * stanga + 1;
    while (fiu <= dreapta)
    {
        //daca al doilea fiu are cheie cea mai mare
        if (fiu < dreapta && a[fiu].cheie < a[fiu + 1].cheie)
            fiu++;
        //retinem al doilea fiu
        if (a[stanga].cheie < a[fiu].cheie) {
            //schimba parinte cu fiu
            swap(&a[stanga], &a[fiu]); //si deplaseaza in jos
            stanga = fiu;
            fiu = 2 * stanga + 1;
        }
        else fiu = dreapta + 1;
    }
}
```

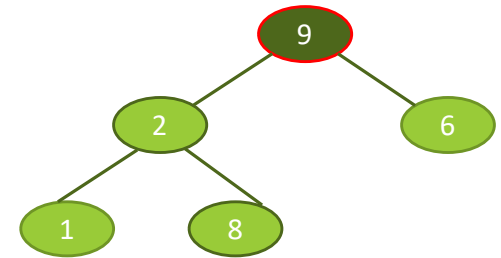
Heapsort



2	8	6	1	9
---	---	---	---	---

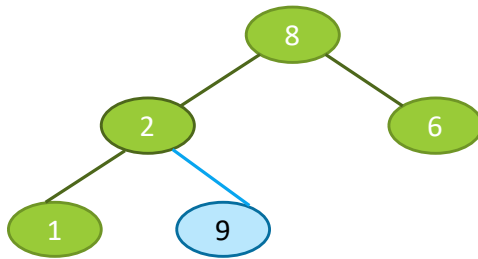


2	9	6	1	8
---	---	---	---	---

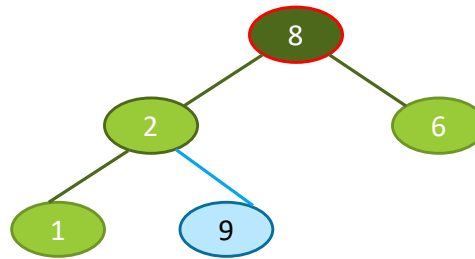


9	2	6	1	8
---	---	---	---	---

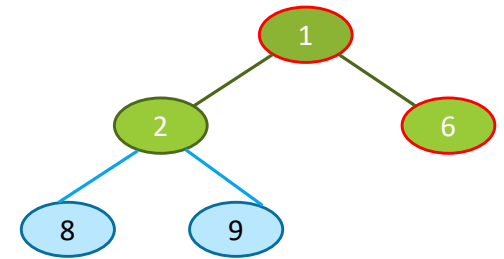
Heapsort



8	2	6	1	9
---	---	---	---	---

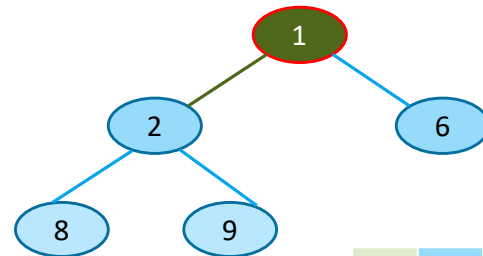
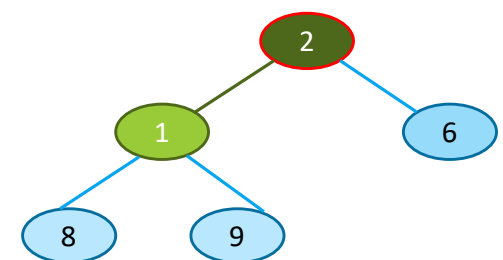
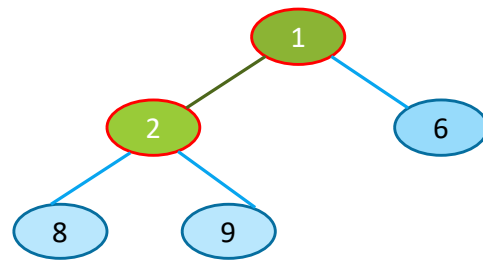
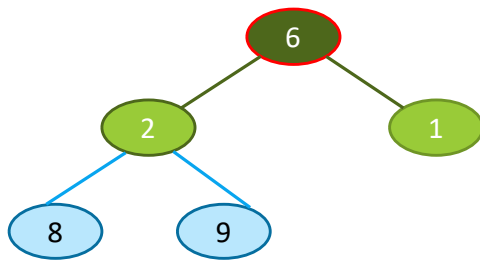


8	2	6	1	9
---	---	---	---	---



1	2	6	8	9
---	---	---	---	---

Heapsort



Heapsort

Analiza algoritmului Heapsort:

La prima vedere nu rezultă în mod evident faptul că această metodă conduce la rezultate bune

Analiza detaliată a performanței metodei heapsort contrazice însă această părere

La **faza de construcție a ansamblului** sunt necesari $n/2$ pași de deplasare

În **fiecare pas** se mută elemente de-a lungul a respectiv $\log(n/2), \log(n/2 + 1), \dots, \log(n-1)$ poziții, (în cel mai defavorabil caz), unde logaritmul se ia în baza 2 și se trunchiază la prima valoare întreagă

Heapsort

În continuare, **faza de sortare** necesită $n-1$ deplasări fiecare cu cel mult respectiv $\log(n-2), \log(n-1), \dots, 1$ mișcări.

În plus mai sunt necesare $3 \cdot (n-1)$ mișcări pentru a **așeza** elementele sortate în ordine.

Toate acestea dovedesc că în cel mai defavorabil caz, tehnica **heapsort** are nevoie de un număr de pași de ordinul $O(n \cdot \log n)$

$$M_{max} = O(n) + O(n \log n) + 3(n - 1) = O(n \log n)$$
$$C_{max} = O(n \log n)$$

Heapsort

În ceea ce privește cazul cel mai **favorabil**, acesta este reprezentat de un tablou cu elemente identice

În acest caz deplasare are $n/2$ pași și nu avem mutări de elemente

În ceea ce privește a doua etapă, heapsort efectuează $n-1$ mutări ale rădăcinii la locul potrivit

În ceea ce privește comparațiile avem n în prima fază și $2(n-1)$ în a doua, în cazul unui tablou cu elemente **identice**

$$C_{min} = M_{min} = O(n)$$

Pentru tablouri cu elemente diferite avem:

$$C_{min} = M_{min} = O(n \log n)$$

Binsort și counting sort

În general algoritmi de sortare bazați pe metode avansate au nevoie de $O(n \cdot \log n)$ pași pentru a sorta n elemente

Trebuie precizat însă faptul că acest lucru este valabil în situația în care:

Nu există nici o altă informație suplimentară referitoare la chei, decât faptul că pe mulțimea acestora este definită o **relație de ordonare**, prin intermediul căreia se poate preciza dacă valoarea unei chei este mai **mică** respectiv mai **mare** decât o alta

După cum se va vedea în continuare, sortarea se poate face și **mai rapid** decât în limitele performanței $O(n \cdot \log n)$, **dacă**:

- Există și **alte informații** referitoare la **cheile** care urmează a fi sortate
- Se **renunță** la constrângerea de sortare "*in situ*"

Binsort și counting sort

Spre **exemplu**:

Se cere să se sorteze un set de n chei de tip întreg, ale căror **valori sunt unice și aparțin intervalului de la 0 la $n-1$** .

Dacă a și b sunt **tablouri** cu câte n elemente, a conținând cheile care urmează a fi sortate, atunci sortarea se poate realiza direct în tabloul b , într-o singură trecere (o sortare liniară), conform secvenței următoare:

```
void sortare_linara(tip_element a[], tip_element b[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        b[a[i].cheie] = a[i];
}
```

Binsort și counting sort

Principiul metodei:

- Se determină locul elementului $a[i]$ și se plasează elementul exact la locul potrivit în tabloul b
- Întregul ciclu necesită $O(n)$ pași.

Rezultatul este însă corect numai în cazul în care există **un singur element** cu cheia x , pentru fiecare valoare cuprinsă între $[0, n-1]$.

Un al doilea element cu aceeași cheie va fi introdus tot în $b[x]$ distrugând elementul anterior.

Binsort și counting sort

Acest tip de sortare poate fi realizat și "*in situ*"

Astfel, fiind dat tabloul a de dimensiune n , ale cărui elemente au respectiv cheile $0, \dots, n-1$, se baleează pe rând elementele sale (bucla **for** exterioară)

Dacă elementul $a[i]$ are cheia j , atunci se realizează interschimbarea lui $a[i]$ cu $a[j]$.

Fiecare interschimbare plasează elementul aflat în locația i exact la locul său în tabloul ordonat, fiind necesare în cel mai rău caz $3 \cdot n$ mișcări pentru întreg procesul de sortare

Binsort și counting sort

index	0	1	2	3	4	5
	2	0	<u>1</u>	5	3	4
1		<u>0</u>	2	5	3	4
0		1	2	5	3	4
0		1	2	5	3	4
0		1	2	5	3	<u>4</u>
0		1	2	4	<u>3</u>	5
0		1	2	3	4	5
0		1	2	3	4	5
0		1	2	3	4	5

Binsort și counting sort

Propunere de implementare în C:

```
void binsort(tip_element a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        while (a[i].cheie != i)
            //daca elementul a[i] nu are
            //cheia i
            //interchimba elementele
            swap(&a[i], &a[a[i].cheie]);
}
```

```
void swap(tip_element *el1,
tip_element *el2)
{
    tip_element tmp;
    tmp = *el1;
    *el1 = *el2;
    *el2 = tmp;
}
```

Binsort și counting sort

Tehnica sortării este simplă:

- Se examinează fiecare element de sortat
- Se introduce în **bin**-ul corespunzător valorii cheii
- În cazul exemplului sortării liniare, bin-urile sunt chiar elementele tabloului b , unde $b[i]$ este binul cheii având valoarea i
- În cazul exemplului anterior de binsort, bin-urile sunt chiar elementele tabloului a după reșezare

Binsort și counting sort

Analiza performanță:

Avem în vedere că în fiecare pas, un element este pus la locul lui în tabloul ordonat

Comparații

$$C_{min} = C_{max} = C_{med} = O(n)$$

Mutări de elemente

$$M_{min} = O(1)$$

$$M_{max} = O(n)$$

Binsort și counting sort

Tehnica aceasta simplă și performantă se bazează pe următoarele **cerințe apriorice**:

- **Domeniul limitat** al cheilor ($0, n-1$)
- **Unicitatea** fiecărei chei

Binsort și counting sort

Dacă cea de-a doua cerință **nu** este respectată, și de fapt acesta este cazul obișnuit, este necesar ca într-un **bin** să fie memorate **mai multe elemente** având aceeași cheie

Acest lucru se realizează fie prin **înșiruire**, fie prin **concatenare**, fiind utilizate în acest scop **structuri listă**

Această situație **nu** deteriorează prea mult performanțele acestei tehnici, efortul de sortare ajungând egal cu $O(n+m)$, unde n este numărul de elemente iar m numărul de chei.

Din acest motiv, această metodă reprezintă punctul de plecare al mai multor tehnici de sortare a structurilor listă

Binsort și counting sort

Spre exemplu, o metodă de rezolvare a unei astfel de situații este cea bazată pe **determinarea distribuției cheilor** (“**distribution counting sort**” sau “**counting sort**”)

Problema se formulează astfel:

- Se cere să se sorteze un tablou cu n articole ale căror chei sunt cuprinse în intervalul $[0, m-1]$.
- Dacă m **nu** este prea mare pentru rezolvarea problemei poate fi utilizat algoritmul de “**determinare a distribuției cheilor**”.

Binsort și counting sort

Ideea algoritmului este următoarea:

1. Se **contorizează** într-o primă trecere **numărul de chei** pentru fiecare valoare de cheie care apare în tabloul a.
2. Se **ajustează** valorile **contoarelor**.
3. Într-o a doua trecere, utilizând aceste contoare, se **mută** direct articolele în poziția lor ordonată în tabloul b.

Binsort și counting sort

O variantă propusă de implementare în C:

```
tip_cheie numar[m];
void countingsort(typ_element a[], typ_element b[], int n)
{
    int i, j;
    for (j = 1; j <= m - 1; j++) numar[j] = 0;
    //initializare tablou contorizare distributie elemente
    for (i = 1; i <= n; i++) numar[a[i - 1].cheie] = numar[a[i - 1].cheie] + 1; //numararea valorilor
    for (j = 1; j <= m - 1; j++) numar[j] = numar[j - 1] + numar[j];
    //ajustare valori contoare
    for (i = n; i >= 1; i--)
    //mutare elemente pe pozitia ordonata in tabloul temporar b
    {
        b[numar[a[i - 1].cheie] - 1] = a[i - 1];
        numar[a[i - 1].cheie] = numar[a[i - 1].cheie] - 1;
    }
    //copiere elemente in tabloul initial
    for (i = 1; i <= n; i++) a[i - 1] = b[i - 1];
}
```


Binsort și counting sort

Funcționarea algoritmului:

- Contoarele asociate cheilor sunt memorate în tabloul numar de dimensiune m .
- Inițial locațiile tabloului numar sunt inițializate pe zero (prima buclă **for**).
- Se contorizează cheile în tabloul numar (a doua buclă **for**).
- În continuare sunt ajustate valorile contoarelor tabloului numar (a treia buclă **for**).

Binsort și counting sort

Funcționarea algoritmului (continuare):

- Se parcurge tabloul a de la sfârșit spre început, iar cheile sunt introduse exact la locul lor în tabloul b cu ajutorul contoarelor memorate în tabloul numar (a patra buclă **for**).
- Concomitent cu introducerea cheilor are loc și **decrementarea** contoarelor specifice astfel încât în final, cheile identice apar în binul specific în **ordinea relativă** în care apar în secvența inițială.
- Ultima buclă **for** realizează **mutarea** integrală a elementelor tabloului b în tabloul a, (dacă acest lucru este necesar).

Binsort și counting sort

Tabloul a

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	1	0	2	0	3	0	1	1	0	3	3	0

index

Tabloul numar

0	1	2	3
0	0	0	0

Inițializare

Tabloul numar

0	1	2	3
6	4	1	3

Contorizare valori

Tabloul numar

0	1	2	3
6	10	11	14

Ajustare valori

Binsort și counting sort

Tabloul numar

0	1	2	3
5	10	11	14



Tabloul a

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	1	1	0	2	0	3	0	1	1	0	3	3	0

Tabloul b

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
						0								

Binsort și counting sort

Tabloul numar

0	1	2	3
5	10	11	13

Tabloul a

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	1	1	0	2	0	3	0	1	1	0	3	3	0

Tabloul b

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
						0								3

Binsort și counting sort

Tabloul numar

0	1	2	3
5	10	11	12

Tabloul a

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	1	1	0	2	0	3	0	1	1	0	3	3	0

Tabloul b

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
						0							3	3

Binsort și counting sort

Tabloul numar

0	1	2	3
4	10	11	12

Tabloul a

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	1	1	0	2	0	3	0	1	1	0	3	3	0

Tabloul b

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
					0	0							3	3

Binsort și counting sort

Tabloul numar

0	1	2	3
4	9	11	12

Tabloul a

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	1	1	0	2	0	3	0	1	1	0	3	3	0

Tabloul b

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
					0	0				1			3	3

Binsort și counting sort

Tabloul numar

0	1	2	3
4	8	11	12

Tabloul a

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	1	1	0	2	0	3	0	1	1	0	3	3	0

Tabloul b

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
					0	0			1	1			3	3

Binsort și counting sort

Tabloul numar

0	1	2	3
3	8	11	12

Tabloul a

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	1	1	0	2	0	3	0	1	1	0	3	3	0

Tabloul b

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
				0	0	0			1	1			3	3

Binsort și counting sort

Tabloul numar

0	1	2	3
3	8	11	11

Tabloul a

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	1	1	0	2	0	3	0	1	1	0	3	3	0

Tabloul b

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
				0	0	0			1	1		3	3	3

Binsort și counting sort

Tabloul numar

0	1	2	3
2	8	11	11

Tabloul a

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	1	1	0	2	0	3	0	1	1	0	3	3	0

Tabloul b

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
			0	0	0	0			1	1		3	3	3

Binsort și counting sort

Tabloul numar

0	1	2	3
2	8	10	11

Tabloul a

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	1	1	0	2	0	3	0	1	1	0	3	3	0

Tabloul b

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
			0	0	0	0			1	1	2	3	3	3

Binsort și counting sort

Tabloul numar

0	1	2	3
1	8	10	11

Tabloul a

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	1	1	0	2	0	3	0	1	1	0	3	3	0

Tabloul b

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
		0	0	0	0	0			1	1	2	3	3	3

Binsort și counting sort

Tabloul numar

0	1	2	3
1	7	10	11

Tabloul a

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	1	1	0	2	0	3	0	1	1	0	3	3	0

Tabloul b

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
		0	0	0	0	0		1	1	1	2	3	3	3

Binsort și counting sort

Tabloul numar

0	1	2	3
1	6	10	11

Tabloul a

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	1	1	0	2	0	3	0	1	1	0	3	3	0

Tabloul b

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
		0	0	0	0	0	1	1	1	1	2	3	3	3

Binsort și counting sort

Tabloul numar

0	1	2	3
0	6	10	11

Tabloul a

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	1	1	0	2	0	3	0	1	1	0	3	3	0

Tabloul b

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	0	0	0	0	0	1	1	1	1	2	3	3	3

Binsort și counting sort

Analiza algoritmului:

Deși se realizează mai multe treceri prin elementele tabloului totuși în ansamblu, **performanța algoritmului de sortare bazat pe determinarea distribuției cheilor** este $O(n)$.

Aceasta metodă de sortare pe lângă faptul că este rapidă are avantajul de a fi **stabilă**, motiv pentru care ea stă la baza mai multor metode de sortare de tip **radix**.

Radix sort

O metodă pe care o putem folosi pentru sortarea unor facturi, ar fi să sortăm facturile pe luni, știind că putem avea doar 12 luni și mai apoi fiecare teanc corespunzând fiecărei luni calendaristice să îl sortăm după ani. În acest mod nu ar trebui să ne preocupăm de faptul că numărul anilor poate fi variabil, deoarece numărul maxim de luni este fix.

Într-un mod asemănător, putem sorta numere sortând pe rând după câte o cifră, indiferent de numărul de cifre pe care îl pot avea aceste numere.

Un **exemplu** sugestiv îl reprezintă sortarea unui teanc de cartele care au inscripționate pe ele **numere formate din trei cifre**.

- Se grupează cartelele în 10 grupe distincte, prima cuprinzând cheile mai mici decât 100, a doua cheile cuprinse între 100 și 199, etc., adică se realizează o sortare după **cifra sutelor**.
- În continuare se sortează pe rând grupele formate aplicând aceeași metodă, după **cifra zecilor**.
- Apoi fiecare grupă nou formată, se sortează după **cifra unităților**.
- Acesta este un exemplu simplu de **sortare radix** cu baza de numerație $m = 10$.

Radix sort

- Metodele de sortare prezentate până în prezent, concep cheile de sortat ca entități pe care le prelucrează integral prin comparare și interschimbare.
- În unele situații însă se poate profita de faptul că aceste chei sunt de fapt **numere** exprimate prin **cifre** aparținând unui **domeniu mărginit**.
- Metodele de sortare care iau în considerare **proprietățile digitale** ale numerelor sunt **metodele de sortare bazate pe baze de numerație** ("radix sort").
- Algoritmii de tip **bază de numerație**:
 - (1) Consideră cheile ca și numere reprezentate într-o **bază de numerație** m , unde m poate lua diferite valori ("radix").
 - (2) Procesează **cifrele individuale** ale numărului.

Radix sort

Pentru sistemele de calcul, unde prelucrările se fac exclusiv în baza 2, se pretează cel mai bine metodele de **sortare radix** care operează cu numere **binare**.

- În general, în **sortarea radix** a unui set de numere, **operația fundamentală** este extragerea unui **set contiguu de biți** din numărul binar care reprezintă **cheia**.
- Spre **exemplu**:
 - Pentru a extrage primii 2 biți ai unui număr binar format din 10 cifre binare:
 - (1) Se realizează o **deplasare la dreapta** cu 8 poziții a reprezentării binare a numărului.
 - (2) Se operează configurația obținută, printr-o **operație "și"** cu masca 0000000011.

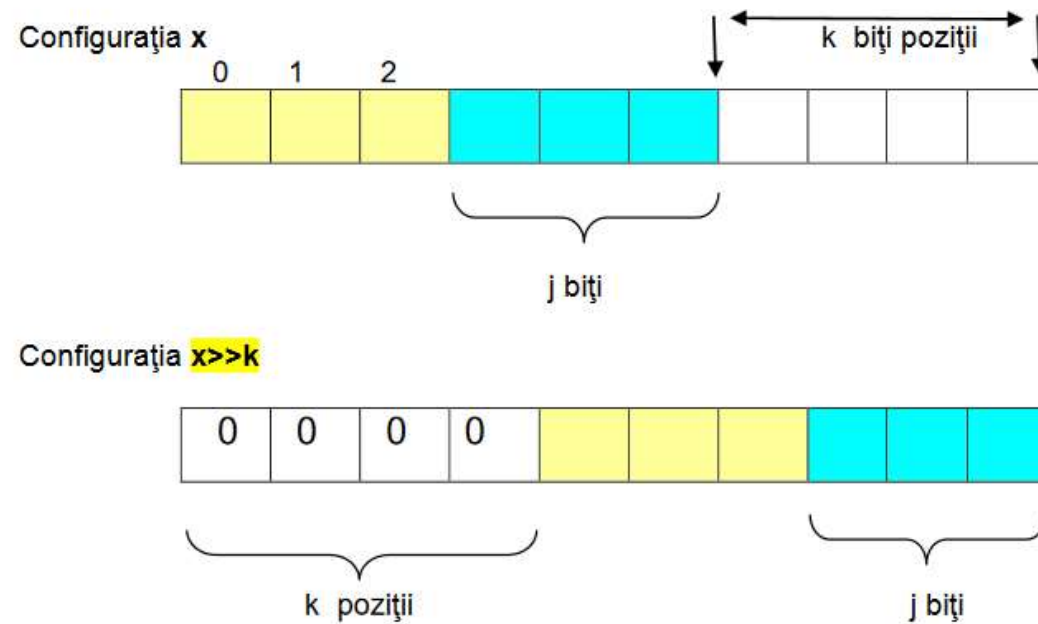
Radix sort

În continuare, pentru implementarea sortărilor radix, se consideră definit un operator **biti(x,k,j:integer):integer** care combină cele două operații returnând valorile a j biți care apar la k poziții de la marginea dreaptă a lui x.

- O posibilă implementare în limbajul C a acestui operator

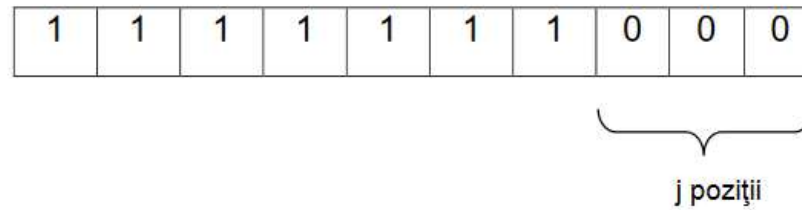
```
-----  
/*operator care returnează j biți care apar la k poziții de  
marginea dreaptă a lui x */  
unsigned biti(unsigned x, int k, int j)  
{  
    return (x>>k) &~ (~0<<j) ;  
}  
-----
```

Radix sort

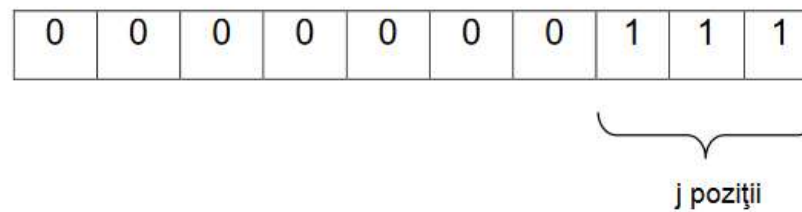


Radix sort

Masca $\sim 0 < j$



Masca $\sim (\sim 0 < j)$



Radix sort

Configurația $(x \gg k) \& \sim (\sim 0 \ll j)$

0	0	0	0	0	0	0			
---	---	---	---	---	---	---	--	--	--

Radix sort

Există **două metode de bază** pentru implementarea **sortării radix**.

- (1) Prima metodă examinează biții cheilor de la **stânga la dreapta** și se numește **sortare radix prin interschimbare ("radix exchange sort")**.
- (2) A doua metodă se numește **sortare radix directă ("straight radix sort")**.

Radix sort

Sortarea radix prin interschimbare ("radix exchange sort")

- **Ideea sortării radix prin interschimbare** este următoarea:
- Se sortează elementelor tabloului a astfel încât toate elementele ale căror chei încep cu un bit zero să fie trecute în fața celor ale căror chei încep cu 1.
- Aceast proces va avea drept consecință formarea a două **partiții** ale tabloului inițial.
- Cele două partiții la rândul lor se sortează independent, conform aceleași metode **după cel de-al doilea bit** al cheilor elementelor, rezultând 4 partiții.
- Cele 4 partiții rezultate se sortează similar după al 3-lea bit, ș.a.m.d.
- Acest mod de lucru sugerează abordarea recursivă a implementării metodei de sortare.

Radix sort

Procesul de sortare radix prin interschimbare se desfășoară exact ca și la **partiționare**:

- Se balează tabloul de la **stânga spre dreapta** până se găsește un element a cărui cheie începe cu 1.
- Se balează tabloul de la **dreapta spre stânga** până se găsește un element a cărui cheie începe cu 0.
- Se **interschimbă** cele două elemente.
- Procesul **continuă** până când indicatorii de parcurgere **se întâlnesc** formând două partiții.
- Se reia aceeași procedură pentru **cel de-al doilea bit** al cheilor elementelor în cadrul **fiecăreia dintre cele două partiții rezultate** ș.a.m.d.

Radix sort

Pseudocod:

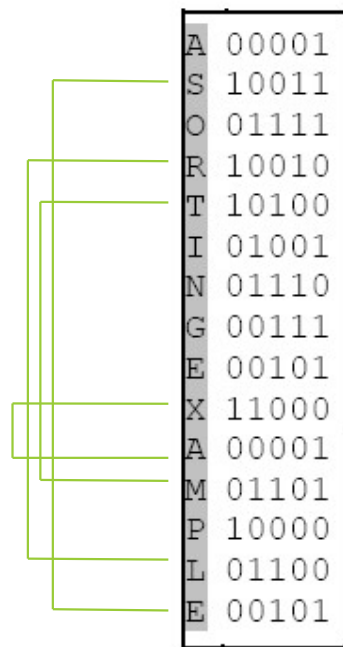
```
RadixInterschimb (stanga,dreapta: TipIndice, b: INTEGER)
//stanga,dreapta - limitele curente ale tabloului de sortat, b - lungimea în biți a cheii de sortat
dacă (dreapta>stanga) și (b>=0) atunci
    i:= stanga; j:= dreapta; b:= b-1;
    executa
        cât timp (biti(a[i].cheie,b,1)=0) și (i<j)
            i:= i+1;
        cât timp(bit(a[j].cheie,b,1)=1) și (i<j)
            j:= j-1;
        interschimba a[i] și a[j]
    cât timp j!=i;
    dacă bit(a[dreapta].cheie,b,1)= 0
        j:= j+1; {dacă ultimul bit testat este 0 se reface lungimea partiției}
    RadixInterschimb(stanga,j-1,b-1);
    RadixInterschimb(j,dreapta,b-1);
```

Radix sort

Propunere implementare C:

```
void radinters(int stanga, int dreapta, int b, int a[])
/*stanga, dreapta - limitele curente ale tabloului de sortat b -
lungimea în biți a cheii de sortat -1*/
{ int i, j;
  if ((dreapta > stanga) && (b >= 0))
  { i = stanga; j = dreapta;
    do { while ((biti(a[i], b, 1) == 0) && (i < j)) i = i + 1;
          while ((biti(a[j], b, 1) == 1) && (i < j)) j = j - 1;
          swap(&a[i], &a[j]);
        } while (!(j == i));
    if (biti(a[dreapta], b, 1) == 0)
      j = j + 1;
    /* dacă ultimul bit testat este 0 se reface lungimea partiției */
    radinters(stanga, j - 1, b - 1, a);
    radinters(j, dreapta, b - 1, a);
  }
}
```


Radix sort



A	00001
S	10011
O	01111
R	10010
T	10100
I	01001
N	01110
G	00111
E	00101
X	11000
A	00001
M	01101
P	10000
L	01100
E	00101

Radix sort

A	00001	A	0	0001
S	10011	E	0	0101
O	01111	O	0	1111
R	10010	L	0	1100
T	10100	M	0	1101
I	01001	I	0	1001
N	01110	N	0	1110
G	00111	G	0	0111
E	00101	E	0	0101
X	11000	A	0	0001
A	00001	X	1	1000
M	01101	T	1	0100
P	10000	P	1	0000
L	01100	R	1	0010
E	00101	S	1	0011



Radix sort

A	00001	A	0	0001	A	0	0	001	A	00	0	01	A	000	0	1	A	0000	1	A
S	10011	E	0	0101	E	0	0	101	A	00	0	01	A	000	0	1	A	0000	1	A
O	01111	O	0	1111	A	0	0	001	E	00	1	01	E	001	0	1	E	0010	1	E
R	10010	L	0	1100	E	0	0	101	E	00	1	01	E	001	0	1	E	0010	1	E
T	10100	M	0	1101	G	0	0	111	G	00	1	11	G	001	1	1	G	0011	1	G
I	01001	I	0	1001	I	0	1	001	I	01	0	01	I	010	0	1	I	0100	1	I
N	01110	N	0	1110	N	0	1	110	M	01	1	10	L	011	0	0	L	0110	0	L
G	00111	G	0	0111	M	0	1	101	M	01	1	01	M	011	0	1	M	0110	1	M
E	00101	E	0	0101	L	0	1	100	L	01	1	00	N	011	1	0	N	0111	0	N
X	11000	A	0	0001	O	0	1	111	O	01	1	11	O	011	1	1	O	0111	1	O
A	00001	X	1	1000	S	1	0	011	S	10	0	11	P	100	0	0	P	1000	0	P
M	01101	T	1	0100	T	1	0	100	R	10	0	10	R	100	1	0	R	1001	0	R
P	10000	P	1	0000	P	1	0	000	P	10	0	00	S	100	1	1	S	1001	1	S
L	01100	R	1	0010	R	1	0	010	T	10	1	00	T	101	0	0	T	1010	0	T
E	00101	S	1	0011	X	1	1	000	X	11	0	00	X	110	0	0	X	1100	0	X

Radix sort

Din punctul de vedere al **performanței**, **metoda de sortare radix prin interschimbare** sortează n chei de b biți utilizând un număr de comparații de biți egal cu $n \cdot b$.

- Cu alte cuvinte, **sortarea radix prin interschimbare** este **liniară** cu **numărul de biți ai unei chei**.
- Pentru o **distribuție normală** a biților cheilor, **sortarea radix prin interschimbare** este ceva mai rapidă decât metoda **quicksort**.

Radix sort

O altă variantă de implementare a **sortării radix** este aceea de a examina biții cheilor elementelor de la **dreapta la stânga**.

Această metodă se numește **sortare radix directă**.

- **Ideea** metodei de **sortare radix directă**:
- Se sortează cheile după un bit examinând biții lor **de la dreapta spre stânga**.
- Sortarea după bitul i constă în extragerea tuturor elementelor ale căror chei au zero pe poziția i și plasarea lor în fața celor care au 1 pe aceeași poziție.
- Când se ajunge la bitul i venind dinspre dreapta, cheile sunt gata sortate pe ultimii $i-1$ biți ai lor.

Nu este ușor de demonstrat că metoda este corectă: de fapt ea este corectă **numai** în situația în care **sortarea după 1 bit** este o **sortare stabilă**.

Datorită acestei cerințe, **interschimbarea normală nu** poate fi utilizată deoarece **nu** este o **metodă de sortare stabilă**.

Radix sort

Am observat la radix prin interschimbare că eficiența în termeni de $O(f(n))$ este $b \cdot n$, unde b este numărul de biți care ne dă și numărul de parcurgeri

Pentru a îmbunătăți această eficiență ne dorim să reducem numărul de parcurgeri, astfel putem sorta cheile nu doar după un bit, ci după un număr de m biți folosind o sortare stabilă

Ne amintim faptul că sortarea prin determinarea distribuției cheilor (counting sort) este o sortare stabilă.

Aceasta poate fi utilizată cu succes în sortarea radix directă

În continuare, urmează un exemplu folosind counting sort în care $m = 2$ și baza de numerație este tot 2

Radix sort

index	0	1	2	3	4	5	6	7
Baza 10	10	8	3	2	1	7	6	9
Baza 2	1010	1000	0011	0010	0001	0111	0110	1001

Sortare dupa cei mai puțin semnificativi 2 biți

m = 2

	Tabloul numar				
Baza 10	0	1	2	3	
Baza 2	00	01	10	11	Inițializare
	0	0	0	0	
	Tabloul numar				
	0	1	2	3	
	1	2	3	2	Contorizare valori

		Tabloul numar						
		0	1	2	3			
		1	3	6	8	Ajustare valori		
index	0	1	2	3	4	5	6	7
Baza 10	8	1	9	10	2	6	3	7
Baza 2	1000	0001	1001	1010	0010	0110	0011	0111

Radix sort

index	0	1	2	3	4	5	6	7	Sortare dupa cei mai semnificativi 2 biți
Baza 10	8	1	9	10	2	6	3	7	
Baza 2	1000	0001	1001	1010	0010	0110	0011	0111	

m = 2

Obs: După biții cei mai puțini semnificativi tabloul este deja sortat din pasul anterior

	Tabloul numar				
Baza 10	0	1	2	3	
Baza 2	00	01	10	11	Inițializare
	0	0	0	0	
	Tabloul numar				
	0	1	2	3	
	3	2	3	0	Contorizare valori

		Tabloul numar						
		0	1	2	3			
		3	5	8	8	Ajustare valori		
index	0	1	2	3	4	5	6	7
Baza 10	1	2	3	6	7	8	9	10
Baza 2	0001	0010	0011	0110	0111	1000	1001	1010

Radix sort

Pentru a crește performanța procesului de sortare, **nu** este indicat să se lucreze cu $m = 2$, ci este convenabil ca m să fie cât mai mare.

- În acest fel se reduce numărul de treceri.
- Dacă se prelucrează m biți **odată**:
 - Timpul de sortare **scade** prin reducerea numărului de treceri.
 - Tabela de distribuții însă **crește** ca dimensiune ea trebuind să conțină $m1 = 2^m$ locații, deoarece cu m biți se pot alcătui 2^m configurații binare.
- În acest mod, **sortarea radix-directă** devine o **generalizare a sortării bazate pe determinarea distribuțiilor**.

Radix sort

Propunere implementare C:

```
void radixdirect(int a[], int n, int b) // b-numarul de biti pe care
este reprezentata cheia -1
{ int i, j, trecere; int numar[m1]; int aux; /*m1:=2^m*/
  for (trecere = 0; trecere <= (b / m) - 1; trecere++)
  {
    for (j = 0; j <= m1 - 1; j++) numar[j] = 0;
    for (i = 0; i <= n - 1; i++)
    { aux = biti(a[i], trecere * m, m); numar[aux] = numar[aux] + 1;
    }
    for (j = 1; j <= m1 - 1; j++) numar[j] = numar[j - 1] + numar[j];
    for (i = n - 1; i >= 0; i--)
    { aux = biti(a[i], trecere * m, m); t[numar[aux] - 1] = a[i];
      numar[aux] = numar[aux] - 1;
    }
    for (i = 0; i < n; i++) a[i] = t[i];
  }
}
```


Radix sort

Performanța sortării radix directe:

- Sortează n elemente cu chei de b biți în b/m treceri.

Dezavantajele metodei de **sortare radix directă**:

- Utilizează un **spațiu suplimentar de memorie** pentru 2^m contoare.
- Utilizează un **buffer** pentru rearanjarea tabloului cu dimensiunea egală cu cea a tabloului original.

Timpul de execuție al celor două metode de sortare radix fundamentale, pentru n elemente având chei de b biți este în esență proporțional cu $n \cdot b$.

Pe de altă parte, timpul de execuție poate fi aproximat ca fiind $n \cdot \log(n)$, deoarece dacă toate cheile sunt diferite, b trebuie să fie cel puțin $\log(n)$.

Mergesort

Mergesort sau sortarea prin interclasare, se bazează, la fel ca și quicksort și shellsort pe principiul "divide et impera" (divide and conquer).

Principiul de sortare este simplu:

- Se împarte secvența de elemente de ordonat în două subsecvențe egale (sau care diferă prin cel mult un element, în cazul unui număr impar de elemente)
- Fiecare subsecvență se împarte la rândul său în alte două subsecvențe până se ajunge la subsecvențe de un element, care se consideră ordonate
- Fiecare două subsecvențe ordonate se combină într-o subsecvență ordonată printr-un proces care se numește interclasare ("merging").
- Procesul se reia pe subsecvențele deja ordonate până când întreaga secvență este ordonată

Interclasarea presupune combinarea a două sau mai multe secvențe ordonate într-o singură secvență ordonată, prin selecții repetate ale componentelor curent accesibile.

Metoda de sortare presupune folosirea de structuri de memorie suplimentare

Mergesort

Interclasarea a două secvențe ordonate într-o altă secvență (tot ordonată)

1	3	6	7
---	---	---	---

s1

2	4	5	8	11
---	---	---	---	----

s2

1<2 , se avansează în secvența s1

1								
---	--	--	--	--	--	--	--	--

Mergesort

Interclasarea a două secvențe ordonate într-o altă secvență (tot ordonată)

1	3	6	7
---	---	---	---

s1

2	4	5	8	11
---	---	---	---	----

s2

2 < 3 , se avansează în secvența s2

1	2							
---	---	--	--	--	--	--	--	--

Mergesort

Interclasarea a două secvențe ordonate într-o altă secvență (tot ordonată)

1	3	6	7
---	---	---	---

s1

2	4	5	8	11
---	---	---	---	----

s2

$3 < 4$, se avansează în secvența s1

1	2	3						
---	---	---	--	--	--	--	--	--

Mergesort

Interclasarea a două secvențe ordonate într-o altă secvență (tot ordonată)

1	3	6	7
---	---	---	---

s1

2	4	5	8	11
---	---	---	---	----

s2

4 < 6 , se avansează în secvența s2

1	2	3	4					
---	---	---	---	--	--	--	--	--

Mergesort

Interclasarea a două secvențe ordonate într-o altă secvență (tot ordonată)

1	3	6	7
---	---	---	---

s1

2	4	5	8	11
---	---	---	---	----

s2

5 < 6 , se avansează în secvența s2

1	2	3	4	5				
---	---	---	---	---	--	--	--	--

Mergesort

Interclasarea a două secvențe ordonate într-o altă secvență (tot ordonată)

1	3	6	7
---	---	---	---

s1

2	4	5	8	11
---	---	---	---	----

s2

$6 < 8$, se avansează în secvența s1

1	2	3	4	5	6			
---	---	---	---	---	---	--	--	--

Mergesort

Interclasarea a două secvențe ordonate într-o altă secvență (tot ordonată)

1	3	6	7
---	---	---	---

s1

2	4	5	8	11
---	---	---	---	----

s2

$7 < 8$, se avansează în secvența s1 care se termină

1	2	3	4	5	6	7		
---	---	---	---	---	---	---	--	--

Mergesort

Interclasarea a două secvențe ordonate într-o altă secvență (tot ordonată)

1	3	6	7
---	---	---	---

s1

2	4	5	8	11
---	---	---	---	----

s2

restul elementelor din s2 se copiaza in secvența rezultat

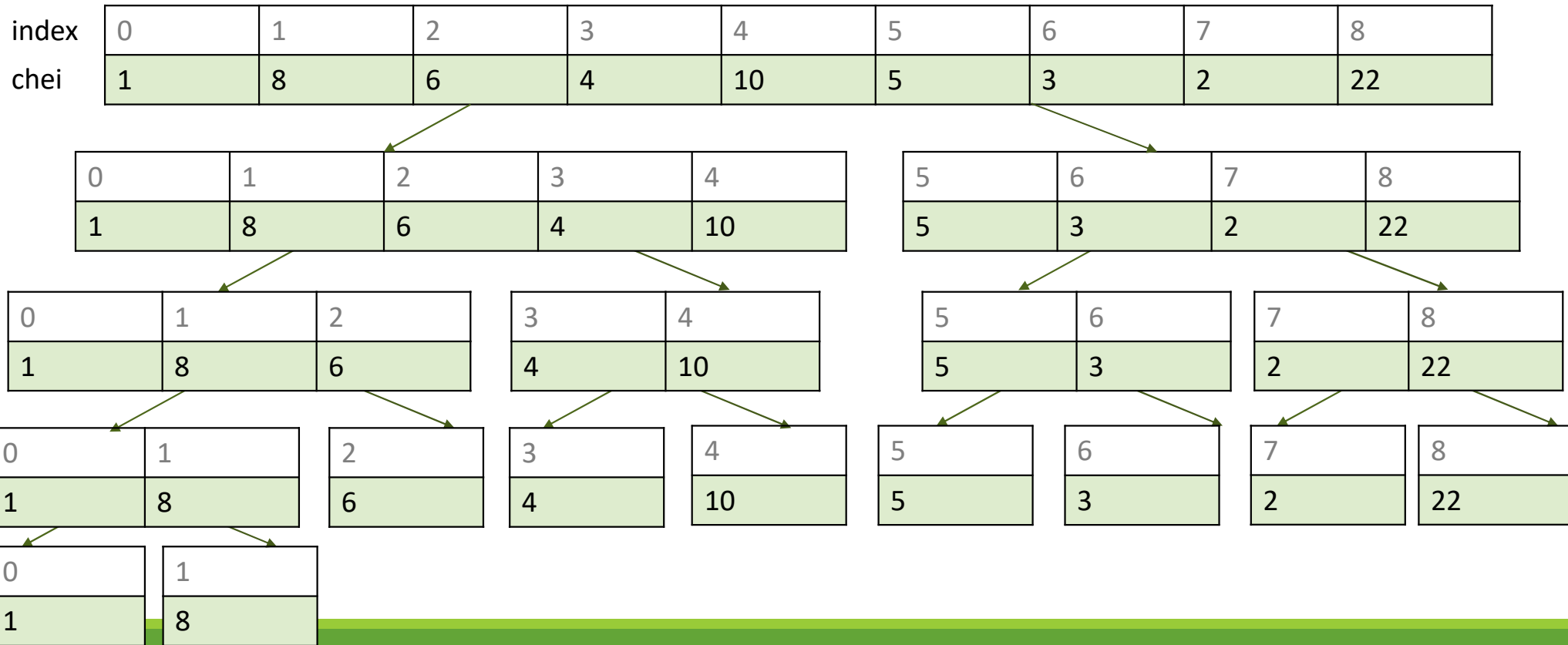
1	2	3	4	5	6	7	8	11
---	---	---	---	---	---	---	---	----

Mergesort

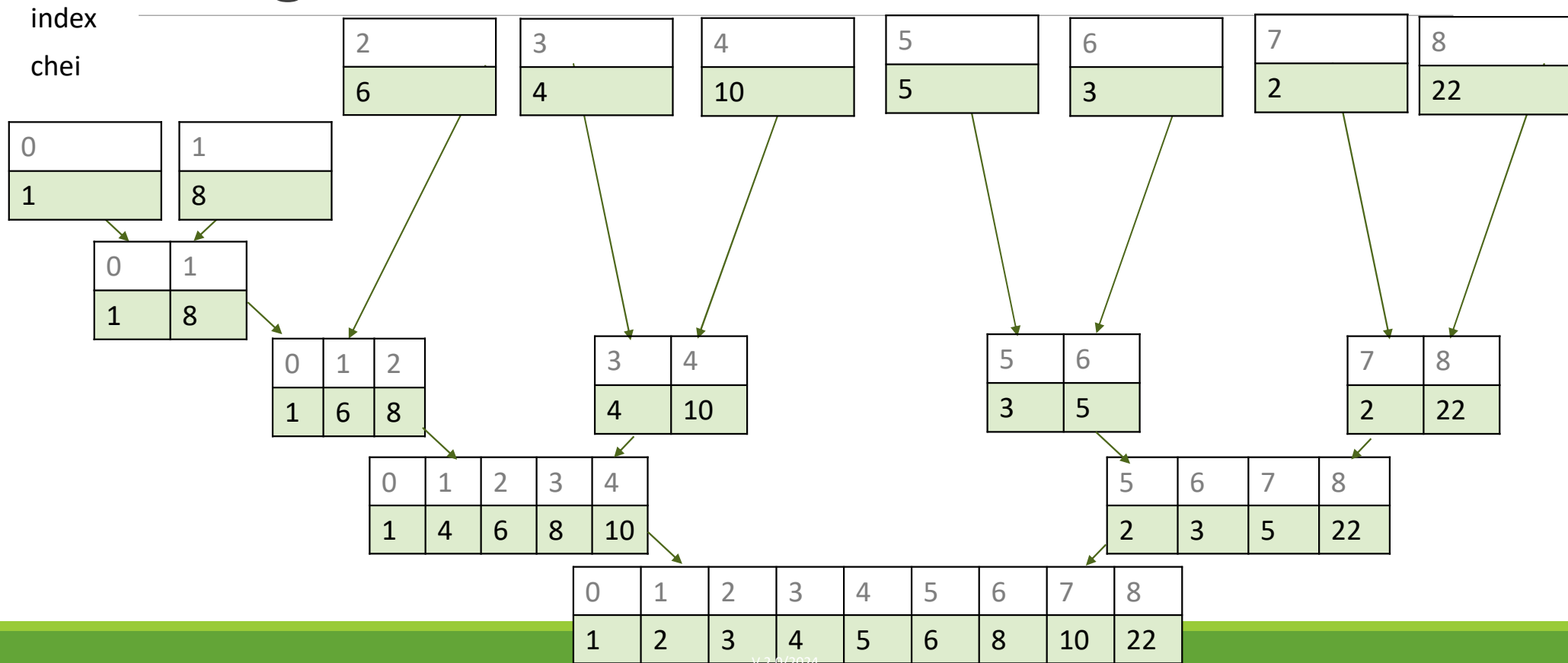
Pseudocod:

```
tablou mergesort(tablou inlist) {  
    daca (inlist.length() <= 1) returneaza inlist;  
    tablou T1 = jumatate din elementele din inlist;  
    tablou T2 = cealalta jumatate din elementele din inlist;  
    returneaza interclasare(mergesort(T1), mergesort(T2));  
}
```

Mergesort



Mergesort



Mergesort

Propunere implementare C:

```
void mergesort(typ_element A[], typ_element temp[], int left, int
right) {
    if (left == right) return; // secventa de un element
    int mid = (left + right) / 2;
    mergesort(A, temp, left, mid);
    mergesort(A, temp, mid + 1, right);
    for (int i = left; i <= right; i++) temp[i] = A[i];
    // se copiaza secventa in temp
    int i1 = left; int i2 = mid + 1; // interclasare inapoi in A
    for (int index = left; index <= right; index++) {
        if (i1 == mid + 1) A[index] = temp[i2++];
        // s-a epuizat secventa din stanga
        else if (i2 > right) A[index] = temp[i1++]; // din dreapta
        else if (temp[i1].cheie < temp[i2].cheie) A[index] = temp[i1++];
        else A[index] = temp[i2++];
    }
}
```

Mergesort

Analiza algoritmului este directă, în ciuda faptului că avem un algoritm recursiv

Interclasarea durează $2k$ timpi, deci are eficiență $\Theta(k)$, unde k este lungimea secvențelor care sunt interclasate

Algoritmul împarte secvența de sortat în jumătate până când avem subsecvențe de un element, astfel numărul de apeluri recursive este $\log n$, pentru n elemente (indiferent de valoarea lor).

Astfel numărul de mutări devine:

$$M(1) = 0$$

$$M(n) = 2M\left(\frac{n}{2}\right) + 2n$$

$$M(n) = 2\left(2M\left(\frac{n}{4}\right) + 2\left(\frac{n}{2}\right)\right) + 2n = 4M\left(\frac{n}{4}\right) + 4n$$

$$= 4\left(2M\left(\frac{n}{8}\right) + 2\left(\frac{n}{4}\right)\right) + 4n = 8M\left(\frac{n}{8}\right) + 6n = \dots = 2^i M\left(\frac{n}{2^i}\right) + 2 \cdot i \cdot n$$

Mergesort

Dacă alegem $i = \log n$, atunci

$$M(n) = 2^i M\left(\frac{n}{2^i}\right) + 2 \cdot i \cdot n = nM(1) + 2n \cdot \log n = 2n \cdot \log n = O(n \cdot \log n)$$

Formula este valabilă pentru toate cazurile (cel mai favorabil, cel mai defavorabil și cazul mediu)

Numărul comparațiilor este dat de o relație similară

$$C(1) = 0$$

$$C(n) = 2C\left(\frac{n}{2}\right) + n - 1$$

Numărul comparațiilor fiind tot $O(n \cdot \log n)$

Sortări externe

Metodele de sortare prezentate anterior se aplicau pe structuri de date de tip tablou, folosindu-se de proprietățile acestuia prin care elementele se pot accesa în mod direct

În continuare vom adresa problema sortării unor secvențe de înregistrări care nu pot fi stocate în memoria operativă din cauza dimensiunilor acestora și nu pot fi accesate în mod direct

În acest caz, înregistrările sunt conținute în fișiere stocate în memoria externă (ex. hard disk)

Atunci când numărul înregistrărilor este prea mare pentru a fi reținute în memoria operativă, sortarea se face citind o serie de înregistrări, ordonându-le într-un anumit mod și scriindu-le înapoi în memoria externă. Acest proces se repetă până când întreg fișierul este sortat

Astfel, tehnicile de sortare prezentate anterior și aplicate tablourilor nu pot fi aplicate și în cazul sortărilor externe

Din punct de vedere al structurilor de date abstracte vom considera în continuare secvența

Sortări externe

Ne amintim:

Particularități ale tablourilor:

- au cardinalitate finită
- pot fi accesate direct

Particularități ale secvențelor:

- au cardinalitate infinită
- pot fi accesate doar în mod secvențial

Ținând cont de proprietățile secvențelor și de costul de citire/scriere din/în memoria externă scopul sortărilor externe este de a minimiza numărul de citiri/scrieri (de) pe disc

Sortări externe

Modelul de acces pe disc împarte fișierele în blocuri de dimensiune fixă

Un bloc poate conține una sau mai multe înregistrări, în funcție de dimensiunea acestora

În continuare vom considera înregistrările de dimensiuni egale, astfel fiecare bloc conține același număr de înregistrări

În general citirea din fișier în ordine secvențială este mai eficientă decât citirea unor blocuri de memorie în ordine aleatoare

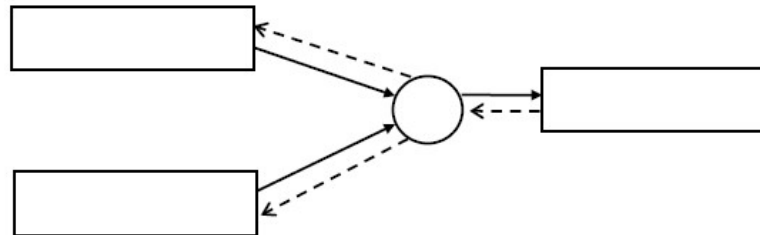
Cu toate că majoritatea algoritmilor de sortare prezentați anterior se bazează pe accesul direct la elemente, Mergesort (sortarea prin interclasare) este un algoritm care poate fi adaptat ușor la o sortare externă

Sortări externe

Mergesort extern (sortare prin interclasare secvențială)

Principiul de funcționare:

1. Se **împarte** secvența A (fișierul original) în două alte secvențe (fișiere) B și C de dimensiune (aproximativ) egală
2. Se **interclasează** subsecvențe de câte o înregistrare din fiecare secvență B și C, obținând subsecvențe ordonate de câte două elemente, care se vor stoca în noua secvență A (rescriind fișierul original)
3. Se **repetă** cu noua secvență A, pașii anteriori, de această dată interclasând subsecvențele ordonate de două elemente în subsecvențe ordonate de patru elemente.
4. Se repetă pașii inițiali, **dublând mereu dimensiunea subsecvențelor ordonate** până la interclasarea întregii secvențe



Sortări externe

Exemplu – sortare cu **3 secvențe** (benzi):

Secvența

A:

34	65	12	22	83	18	4	67	9	11
----	----	----	----	----	----	---	----	---	----

Pasul 1 – distribuire subsecvențe ordonate

B:

34	12	83	4	9
----	----	----	---	---

C:

65	22	18	67	11
----	----	----	----	----

Pasul 2 – interclasare

A:

34	65	12	22	18	83	4	67	9	11
----	----	----	----	----	----	---	----	---	----

Pasul 1 – disribuire

B:

34	65	18	83	9	11
----	----	----	----	---	----

C:

12	22	4	67
----	----	---	----

Sortări externe

Pasul 2 – interclasare

A:	12	22	34	65	4	18	67	83	9	11
----	----	----	----	----	---	----	----	----	---	----

Pasul 1 – distribuire

B:	12	22	34	65	9	11
----	----	----	----	----	---	----

C:	4	18	67	83
----	---	----	----	----

Pasul 2 – interclasare

A:	4	12	18	22	34	65	67	83	9	11
----	---	----	----	----	----	----	----	----	---	----

Pasul 1 – distribuire

B:	4	12	18	22	34	65	67	83
----	---	----	----	----	----	----	----	----

C:	9	11
----	---	----

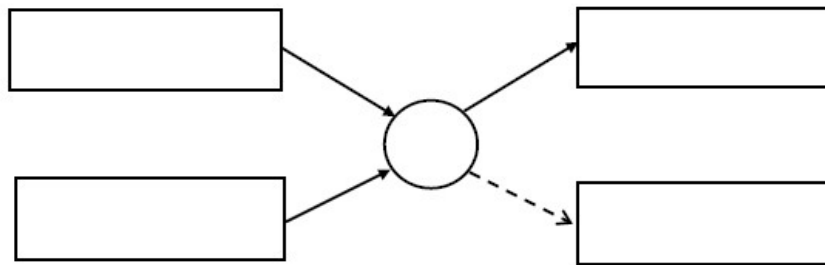
Pasul 2 – interclasare

A:	4	9	11	12	18	22	34	65	67	83
----	---	---	----	----	----	----	----	----	----	----

Sortări externe

Faza de înjumătățire care de fapt nu contribuie direct la sortare (în sensul că ea nu permută nici un element), consumă jumătate din operațiile de copiere

- Acest neajuns poate fi remediat prin combinarea fazei de înjumătățire cu cea de interclasare
- Astfel simultan cu interclasarea se realizează și redistribuirea n-uplelor interclasate pe două secvențe care vor constitui sursa trecerii următoare.
- Acest proces se numește interclasare cu o singură fază sau interclasare echilibrată cu **4 secvențe**



Sortări externe

Exemplu, sortare echilibrată cu **4 secvențe** :

	34	65	12	22	83	18	4	67	9	11
A:	34	12	83	4	9					
B:	65	22	18	67	11					
C:	34	65	18	83	9	11				
D:	12	22	4	67						
A:	12	22	34	65	9	11				
B:	4	18	67	83						
C:	4	12	18	22	34	65	67	83		
D:	9	11								
A:	4	9	11	12	18	22	34	65	67	83

Sortări externe

Tehnica de sortare prin interclasare nu ia în considerare faptul că datele inițiale pot fi parțial sortate, subsecvențele interclasate având o lungime fixă predeterminată adică $2k$ în trecerea k .

- De fapt, oricare două subsecvențe ordonate de lungimi m și n pot fi interclasate într-o singură subsecvență ordonată de lungime $m+n$.
- Tehnica de interclasare care în fiecare moment combină cele mai lungi secvențe ordonate posibile se numește sortare prin **interclasare naturală**.
- În cadrul acestei tehnici un rol central îl joacă noțiunea de monotonie

Formal, se înțelege prin monotonie orice secvență parțială a_i, \dots, a_j care satisface următoarele condiții:

- 1) $1 \leq i \leq j \leq n$;
- 2) $a_k \leq a_{k+1}$ pentru orice $i \leq k < j$;
- 3) $a_{i-1} > a_i$ sau $i = 1$;
- 4) $a_j > a_{j+1}$ sau $j = n$;

• Această definiție include și monotoniile cu un singur element, deoarece în acest caz $i=j$ și proprietatea 2) este îndeplinită, neexistând nici un k cuprins între i și $j-1$.

Sortări externe

Exemplu, sortare **prin interclasare naturală** :

34	65	12	22	83	18	4	67	9	11
----	----	----	----	----	----	---	----	---	----

A:

34	65	18	9	11
----	----	----	---	----

B:

12	22	83	4	67
----	----	----	---	----

C:

12	22	34	65	83	9	11
----	----	----	----	----	---	----

D:

4	18	67
---	----	----

A:

4	12	18	22	34	65	67	83
---	----	----	----	----	----	----	----

B:

9	11
---	----

C:

4	9	11	12	18	22	34	65	67	83
---	---	----	----	----	----	----	----	----	----

Sortări externe

- Complexitatea metodelor de sortare externă prezentate nu permite formularea unor concluzii generalizatoare, cu atât mai mult cu cât evidențierea performanțelor acestora este dificilă.
- Se pot formula însă următoarele **observații**:
 - Există o **legătură indisolubilă** între un anumit **algorithm** care rezolvă o anumită problemă și **structurile de date** pe care acesta le utilizează, influența celor din urmă fiind uneori decisivă pentru algorithm, acest lucru este evidențiat cu preponderență în cazul sortărilor externe care sunt diferite ca mod de abordare în raport cu metodele de sortare internă
 - De multe ori trebuie să facem un compromis între performanța de timp și cea de memorie

Concluzii

- O comparație a algoritmilor de sortare pe caz general studiați

Nume	Caz mediu	Cazul cel mai defavorabil	Memorie auxiliară	Algoritm stabil
Shellsort	Depinde de alegerea lui H	$O(n \log^2 n)$	$O(1)$	Nu
Quicksort	$O(n \log n)$	$O(n^2)$	Depinde de implementare	Depinde de implementare
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(1)$	Nu
Mergesort	$O(n \log n)$	$O(n \log n)$	Depinde de implementare	Da

Concluzii

- Algoritmii de sortare se pot clasifica în funcție de următorii parametri:
 - Numărul de comparații – pentru sortările generale putem obține $O(n \log n)$ pentru cazul cel mai favorabil și $O(n^2)$ pentru cazul cel mai defavorabil.
 - Numărul de interschimbări/ mutări în memorie
 - Memoria utilizată – algoritmi ”in situ” necesită $O(1)$ sau cel mult $O(\log n)$ spații auxiliare de memorie
 - Recursivitate – algoritmi recursivi (ex. variante de Quicksort, Mergesort) sau nerecursivi (ex. Selection Sort, Insertion Sort).
 - Stabilitate – pentru algoritmi stabili, elementele egale își păstrează ordinea relativă pe care au avut-o în șirul inițial.
 - Tipul de memorie folosit pentru stocarea datelor - algoritmi de sortare interni și externi.

Concluzii

- Sortarea elementelor este una dintre cele mai vechi și mai studiate probleme în domeniul informaticii
- Sortările pe bază de comparații au o limitare inferioară (lejeră) $\omega(f(n)) = n \log n$, ceea ce înseamnă că în cazul acestor sortări nu putem obține o performanță mai bună de $O(n \log n)$
- Mergesort și Heapsort au $O(n \log n)$ și pentru cazul cel mai defavorabil, în timp ce Quicksort doar pentru cazul mediu
- Avem și sortări în timp liniar (binsort și radix), dar fac parte dintr-o altă categorie de sortări față de cele bazate pe comparații și nu pot fi utilizate în orice caz
- Decizia cu privire la un algoritm de sortare anume se ia în funcție de mai mulți factori (nu doar în funcție de performanța de timp)

Exerciții

Ex1: Implementați diferite variante de Shellsort pe baza sortărilor simple studiate în capitolul anterior, folosind diferite secvențe de incremenți:

- Numere Fibonacci
- $h_{t-1}=1$, $h_i=3*h_{i+1}+1$, t se alege astfel încât să avem valori ale incremenților mai mici sau egale cu n ($h_i \leq n$)

Alcătuiți grafice cu valorile timpilor de execuție și comparați performanța diferitelor variante de Shellsort implementate la punctele anterioare

Exerciții

Ex2: Pornind de la implementarea algoritmului Quicksort din acest curs, dezvoltați o serie de implementări care să fie cât mai eficiente din punct de vedere al timpului de rulare și a memoriei utilizate considerând:

- Diferite variante de alegere a pivotului
- Ineficiența pentru un număr mic de elemente: în apelul recursiv, nu apălați tot quicksort() pe un tablou cu mai puțin de L elemente (unde valoarea lui L o alegeți dvs.), ci apălați sortarea prin inserție sub această limită (în rest se va apăla quicksort în mod obișnuit)
- Ineficiența pentru cazul în care numărul de elemente egale este semnificativ: implementați o variantă care împarte tabloul în trei partiții în loc de două (prima conține elementele mai mici decât pivotul, a doua elementele egale cu pivotul și a treia, elementele mai mari).

Bibliografie selectivă

- Drozdek, A. (2012). *Data Structures and algorithms in C++*. Cengage Learning.
- Shaffer, C. A. (2012). Data structures and algorithm analysis.
- Crețu, V. Structuri de date și algoritmi, Editura Orizonturi Universitare Timișoara, 2011