DUOMENŲ STRUKTŪROS IR ALGORITMAI

Greitieji rūšiavimo algoritmai

Šelo (Shellsort), Suliejimo (Mergesort), Spartusis (Quicksort)

Praeitos paskaitos santrauka

- Kas yra rūšiavimas ir kam jis reikalingas ?
  - Pavyzdžiai iš kasdieninio gyvenimo
- Kokios yra rūšiavimo bazinės procedūros?
- Paprasčiausi rūšiavimo algoritmai
  - Išrinkimo
  - Įterpimo
  - Burbulo
- Koks rūšiavimo algoritmas vadinamas **stabiliu**?

Paprasčiausių rūšiavimo algoritmų sudėtingumas

	Insertion	Bubble	Selection
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps:			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

Atliekamų operacijų skaičius yra proporcingas N^2 .

Spartieji rūšiavimo algoritmai

Ankščiau nagrinėti paprasti (lėti) rūšiavimo algoritmai, kurių atliekamų operacijų skaičius yra proporcingas N^2.

Šie algoritmai nėra efektyvus, nes rūšiavimo algoritmų apatinis operacijų įvertis yra $N \log N$.

Toliau susipažinsime su sparčiaisiais rūšiavimo algoritmais, kurių sudėtingumas yra artimas optimaliam.

Spartieji rūšiavimo algoritmai:

- Šelo rūšiavimo algoritmas
- Suliejimo rūšiavimo algoritmas
- Spartus rūšiavimo algoritmas
- Piramidinis rūšiavimo algoritmas
- Radix rūšiavimo algoritmas

Skaldyk ir valdyk

Sparčiųjų rūšiavimo algoritmai sudaryti remiantis **skaldyk ir valdyk** metodu.

Metodo žingsniai:

1. Uždavinį skaidome į kelis mažesnius uždavinius.
2. Randame šių uždavinių sprendinius.
3. Iš gautų sprendinių sudarome viso uždavinio sprendinį.

Dalinius uždavinius galime spręsti tokiu pačiu metodu t.y. taip skaidome tol, kol gautieji uždaviniai yra lengvai išsprendžiami.

Toks rekursyvus algoritmas vadinamas *skaldyk ir valdyk* metodu (angl. *divide and conquer*).

Šelo rūšiavimo algoritmas

Šio rūšiavimo algoritmo išradėjas **D.L. Shell**.

Šelo rūšiavimo algoritmas naudoja **Įterpimo rūšiavimo** algoritmo geriausią variantą.

Šelo rūšiavimo algoritmo strategija – padaryti sąrašą beveik surūšiuotą ir galutinį rūšiavimą atlikti naudojant **Įterpimo rūšiavimo** algoritmą.

Gera Šelo algoritmo realizacija blogiausiu atveju surūšiuoja duomenis žymiai greičiau nei $O(n^2)$.

Šelo algoritmo principas

Šelo rūšiavimo algoritmas naudoja didžiąją dalį veiksmų, kurie naudojami visuose greito rūšiavimo algoritmuose:

- Suskaldo duomenų aibę į poaibius,
- Nepriklausomai surūšiuoja poaibius,
- Sudaro naujus poaibius,
- Kartoja rūšiavimą su naujais poaibiais.

Šelo algoritmas suskaldo aibę į mažesnius poaibius ir juos surūšiuoja naudodamas **įterpimo rūšiavimo algoritmą**. Tada suformuojami nauji, didesnį elementų skaičių turinys, poaibiai ir vėl atliekamas rūšiavimas su naujais poaibiais. Poaibiai didinami tol, kol gaunama visa aibė.

Algoritmas nestabilus.

Pavyzdys

Sakykime turime aibę, kurią sudaro **N** elementų, kur $\mathbf{N = 2^k}$.

Suskaidykime aibę į ***N/2*** poaibių, kuriuose būtų po 2 elementus kiekviename, o poaibio elementų indeksai skirtųsi per ***N/2***.

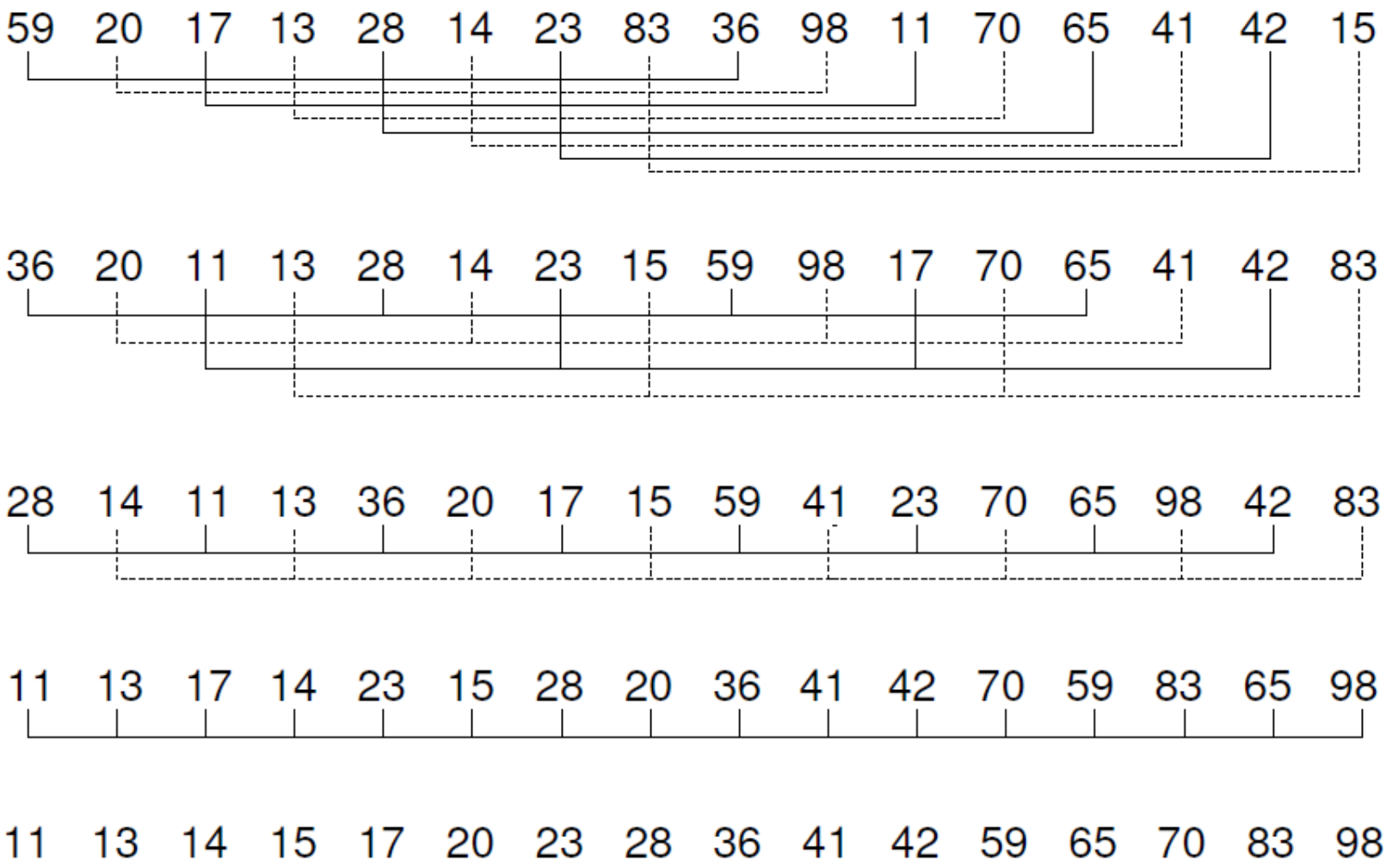
$$\mathbf{n_i = n_{i-1} / 2}, \text{ kur } \mathbf{n_0 = N}$$

Pavyzdys:
Turime 16 elementų, kurių indeksai sunumeruoti nuo 0 iki 15. Sudaliname aibę į 16/2 = 8 poaibių. Elementų indeksai poaibiuose būtų tokie:

{ 0; 8 } {1; 9 } { 2; 10 } { 3; 11 } ..... { 7; 15 }

Kiekvienas poaibis rūšiuojamas taikant Įterpimo rūšiavimo algoritmą.

Pavyzdžio tęsinys



Šelo algoritmo realizacija

```
void ShellSort ( int array [ ], int length)
{
    int i, temp, flag = 1;
    int d = length;
    while( flag || (d > 1))
    {
        flag = 0;                    // kintamasis flag tikrina ar įvyko sukeitimas
        d = (d+1) / 2;               // d  mažinamas per pusę kiekviename žingsnyje
        for (i = 0; i < (length - d); i++)
      {
            if ( array [i + d] < array [i] )
           {
                temp = array [i + d];
                array [i + d] = array [i];
                array [i] = temp;
                flag = 1;
           }
       }   }
      return; }
```

Šelo algoritmo savybės

Algoritmo sudėtingumas priklauso nuo aibės padalinimo į poaibius. Pavyzdžiui dalinant į poaibius, kai elementų, priskiriamų vienam poaibiui, indeksai skaičiuojami pagal formulę:

$$\mathbf{n_j = 3n_{j-1} + 1,} \quad \text{kur } \mathbf{n_0 = 0}$$

sudėtingumas yra $O(n^{1.5})$. Elementų indeksų pavyzdžiai *1; 4; 13; 40; 121; ...*,

Kitiems indeksų skaičiavimo atvejams, algoritmo sudėtingumas nustatomas kaip $O(n^{4/3})$ arba $O(n \cdot \log_2(n))$.

Šelo algoritmas parodo, kaip panaudojant papildomus žingsnius, galima patobulinti (pagreitinti) lėtą algoritmą.

Suliejimo rūšiavimo algoritmas

Suliejimo algoritmas pagristas pastebėjimu, kad nesunku efektyviai sujungti du surūšiuotus poaibius į vieną surūšiuotą aibę. Todėl metodas ir vadinamas *suliejimo* algoritmu (angl. *merge sort*).

Suliejimo rūšiavimo algoritmas yra sukonstruotas remiantis *skaldyk ir valdyk* metodu.

Suliejimo algoritmo skaičiavimų apimtis net ir blogiausiu atveju yra $O(N \log N)$.

Suliejimo algoritmo buvo išrastas 1945. Jo autorius **John von Neumann**.

Suliejimo rūšiavimo algoritmas

Skaldyk ir valdyk metodo taikymas suliejimo algoritme:

- Suskaldome aibę į poaibius, kurie toliau dalinami į dar mažesnius poaibius. Ciklas tęsiamas, kol gaunami galimai mažiausi poaibiai (*skaldyk* žingsnis)
- Surūšiuojame mažus poaibius (*valdyk* žingsnis)
- Sujungiame surūšiuotus poaibius (*sujungimo* žingsnis)

Suliejimo rūšiavimo algoritmas

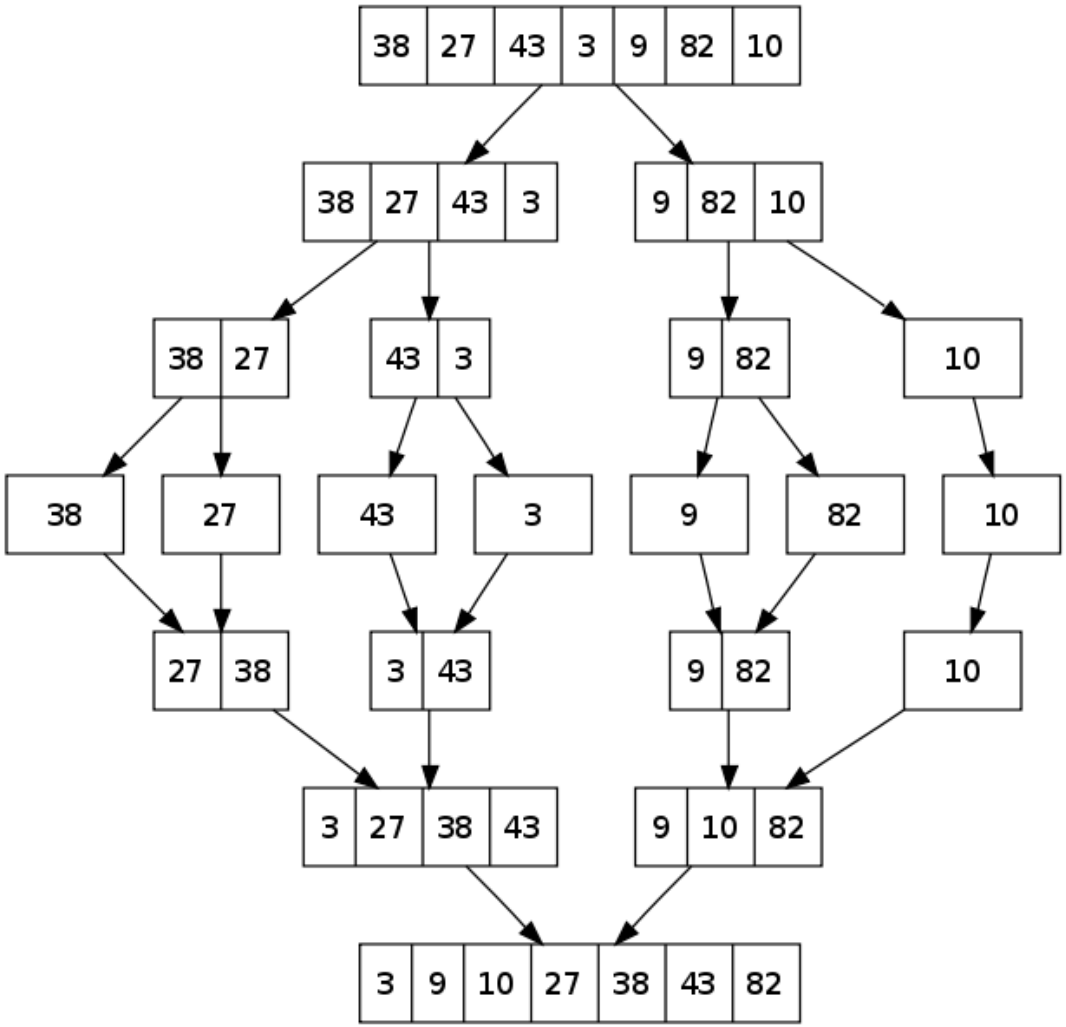
Sakykime turime masyvą ***A*[*p*]**, kurį reikia surūšiuoti**.**
Tegul kinta $p = 1 \ldots n$.
Atliksime sekančius žingsnius, kad surūšiuotume masyvą:

1. **Skaldyk žingsnis**
Jei masyvas *A* neturi elementų arba jų skaičius lygus 1, tuomet neatliekame rūšiavimo. Priešingu atveju daliname masyvą $A[p]$ į du mažesnius masyvus $A[p \,..\, q]$ ir $A[q + 1 \,..\, n]$, kur abu turi po pusę elementų. Čia q yra masyvo $A[p]$ vidurinio elemento indeksas.

2. **Valdyk žingsnis**
Surūšiuojame nepriklausomai du masyvus $A[p \,...\, q]$ ir $A[q + 1 \,..\, n]$.

3. **Sujungimo žingsnis**
Sujungiame dviejų masyvų elementus atgal į masyvą $A[p]$ apjungdami elementus $A[p \,..\, q]$ ir $A[q + 1 \,..\, n]$ į vieną seką.

Suliejimo algoritmo pavyzdys



Suliejimo algoritmo sudėtingumas

Sakykime turime N elementų, kur N yra 2^k skaičius. To reikia, kad dalindami masyvą visuomet gautume submasyvų elementų skaičių lygų $n/2$.

Jei N = 0 arba N = 1, tuomet neatliekame jokių veiksmų.

Jei N ≥ 2, tuomet:

Skaldyk: Randame *q* kaip *p* ir N vidurinę reikšmę. Veiksmo sudėtingumas O(1).

Valdyk: Rekursyviai sprendžiame 2 problemas, kurių dydis $n/2$, kurias atliksime per $2*T(n/2)$.

Sujungimas: Dviejų masyvų su n/2 elementų sujungimas apima $O(n)$.

Suliejimo algoritmo sudėtingumas

Susumavus visus laikus, gauname sudėtingumą, proporcingą N t.y. O(*n*).

T(n) = O(n), jei n = 1

T(n) = 2*T(n/2) + O(n), jei n > 1

T(n) = O(n $\log_2$n), jei naudojama rekursija

Suliejimo algoritmo realizacija

```
#include <iostream.h>

void mergeSort (int numbers[ ], int temp[ ], int array_size);
void m_sort      (int numbers[ ], int temp[ ], int left, int right);
void merge       (int numbers[], int temp[], int left, int mid, int right);

int main()
{
        int array1 [5] = {65, 72, 105, 55, 2};
        int temp_array [5];

        mergeSort (array1, temp_array, 5);
        cout << "Surūšiuotas masyvas \n\n";
            for (int i = 0; i < 5; i++)
            {
                cout << array1[i] << "   ";
             }
        return 0; }
```

Suliejimo algoritmo realizacija

```
void mergeSort  ( int numbers[ ], int temp[ ], int array_size )
{
  m_sort ( numbers, temp, 0, array_size - 1);
}

void m_sort ( int numbers[], int temp[], int left, int right )
{
  int mid;

  if (right > left)
  {
    mid = (right + left) / 2;
    m_sort ( numbers, temp, left, mid );
    m_sort ( numbers, temp, (mid+1), right );

    merge ( numbers, temp, left, (mid+1), right );
  }
}
```

Suliejimo algoritmo realizacija

```
void merge ( int numbers[ ], int temp[ ], int left, int mid, int right )
{
  int i, left_end, num_elements, tmp_pos;
  left_end = (mid - 1);
  tmp_pos = left;
  num_elements = (right - left + 1);

  while ((left <= left_end) && (mid <= right))
  {
    if (numbers [left] <= numbers [mid])
    {
      temp [tmp_pos] = numbers [left];
      tmp_pos += 1;
      left += 1;
    }
    else
    {   temp [tmp_pos] = numbers [mid];
        tmp_pos += 1;
         mid += 1;
    } }
```

Suliejimo algoritmo realizacija

```
while (left <= left_end)
  {
    temp [tmp_pos] = numbers[left];
    left += 1;
    tmp_pos += 1;
  }
  while (mid <= right)
  {
    temp [tmp_pos] = numbers[mid];
    mid += 1;
    tmp_pos += 1;
  }
  // modified
  for (i=0; i < num_elements; i++)
  {
    numbers [right] = temp [right];
    right -= 1;
  } }
```

Spartusis rūšiavimo algoritmas

Šis algoritmas turi tokį pavadinimą, nes tai greičiausias bendros paskirties rūšiavimo algoritmas, kurio vidutinis sudėtingumas **O(*N* log*N*).**

Šis algoritmas nereikalauja papildomos atminties, kaip kad Suliejimo algoritmas, todėl jis racionaliai naudoja atmintį.

Spartusis rūšiavimo algoritmas yra sukonstruotas remiantis *skaldyk ir valdyk* metodu.

Spartusis rūšiavimo algoritmas buvo išrastas TSRS, 1960m. Jo autorius Tony Hoare.

Spartusis rūšiavimo algoritmas

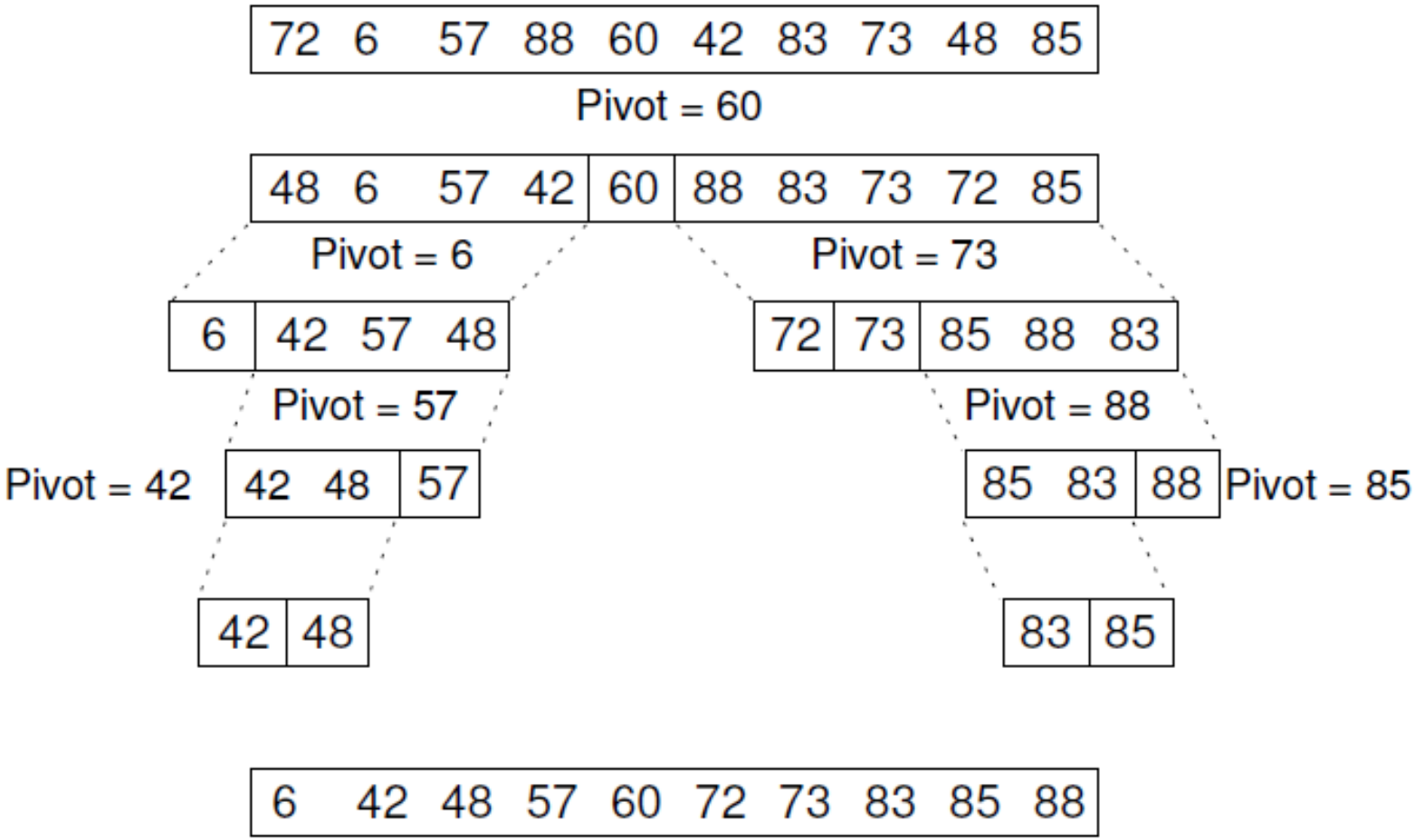
Pagrindinai sparčiojo algoritmo žingsniai:

Skaidymo žingsnis. Visą aibę dalijame į du poaibius. Tuo tikslu parenkame *pagrindinį* elementą a_j (angl. *pivot*), tada pirmajam poaibiui priskiriame elementus mažesnius už a_j , o antrajam poaibiui – nemažesnius už pagrindinį elementą.

Dalinio uždavinio sprendimas. Surūšiuojame abu poaibius kuriuo nors rūšiavimo algoritmu.

Viso uždavinio sprendinio radimas. Kadangi visi pirmojo poaibio elementai mažesni už antrojo poaibio elementus, tai išsprendę dalinius uždavinius mes jau turime surūšiuotą ir visą aibę. Jokių veiksmų nebereikia atlikti.

Pavyzdys


72 6 57 88 60 42 83 73 48 85
Pivot = 60
48 6 57 42 60 88 83 73 72 85
Pivot = 6
Pivot = 73
6 42 57 48
72 73 85 88 83
Pivot = 57
Pivot = 88
Pivot = 42
42 48 57
85 83 88
Pivot = 85
42 48
83 85
6 42 48 57 60 72 73 83 85 88

Spartusis rūšiavimo algoritmas

Sakykime turime masyvą **array *A*[*p*], p=0, .. r**. Skaidome jį į du netuščius masyvus $A[p \ldots q]$ ir $A[q+1 \ldots r]$ taip, kad visi masyvo $A[p \ldots q]$ elementai yra mažesni už bet kurį masyvo $A[q+1 \ldots r]$ elementą.

Tiksli q reikšmė apskaičiuojama dalinimo metu.

QuickSort algoritmas

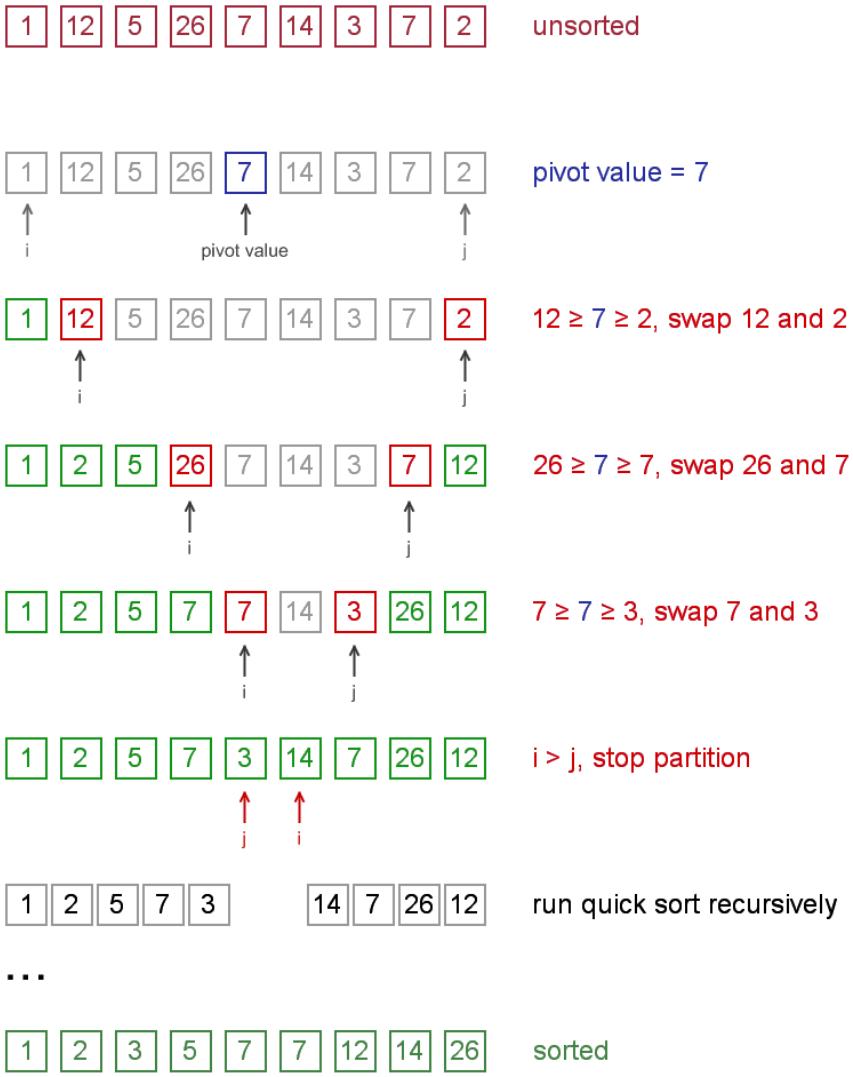
If $p < r$ then

q = Partition (A, p, r)

Recursive call to Quick Sort (A, p, q)

Recursive call to Quick Sort (A, $q + r$, r)

Dalinimas


1 12 5 26 7 14 3 7 2
unsorted
1 12 5 26 7 14 3 7 2
pivot value = 7
i
pivot value
j
1 12 5 26 7 14 3 7 2
12 ≥ 7 ≥ 2, swap 12 and 2
i
j
1 2 5 26 7 14 3 7 12
26 ≥ 7 ≥ 7, swap 26 and 7
i
j
1 2 5 7 7 14 3 26 12
7 ≥ 7 ≥ 3, swap 7 and 3
i
j
1 2 5 7 3 14 7 26 12
i > j, stop partition
j
i
1 2 5 7 3
14 7 26 12
run quick sort recursively
...
1 2 3 5 7 7 12 14 26
sorted

Pagrindinio elemento pasirinkimas

Galimi pagrindinio elemento parinkimo variantai:

- kairiausias elementas (blogas pasirinkimas, jei masyvas jau surūšiuotas)
- dešiniausias elementas (blogas pasirinkimas, jei masyvas jau surūšiuotas)
- atsitiktinis indeksas masyve
- vidurinysis indeksas masyve
- mediana pirmojo, viduriniojo ir paskutiniojo elemento.

P.S. Mediana –tai vidurinis skaičius (surūšiuotose masyvuose).

Sudėtingumas

- Blogiausiu atveju

  $T(n) = O(n^2)$

- Geriausiu atveju

  $T(n) = O(n \log n)$

- Vidutinis atvejis

  $T(n) = O(n \log n)$

Spartusis rūšiavimo algoritmas

```
void quickSort (int arr[ ], int left, int right) {
      int i = left, j = right;
      int tmp;
      int pivot = arr[(left + right) / 2];

            while (i <= j) {                    // dalinimas
            while (arr[i] < pivot)
                  i++;
            while (arr[j] > pivot)
                  j--;
            if (i <= j) {
                  tmp = arr[i];
                  arr[i] = arr[j];
                  arr[j] = tmp;
                  i++;
                  j--;
             }
          }
```

Spartusis rūšiavimo algoritmas

```
/* rekursija */

      if (left < j)
            quickSort(arr, left, j);

      if (i < right)
            quickSort(arr, i, right);

}
```

Užduotys

Nr 1.

- Suskaičiuoti sparčiojo rūšiavimo algoritmo sudėtingumą, kai naudojami skirtingi pagrindiniai elementai.

No 2.

- Modifikuoti sparčiojo rūšiavimo algoritmą, kad būtų surūšiuota mažėjančia tvarka.