Алгоритми та складність

I семестр

Лекція 6

- Бульбашкове
- Вставками
- Вибором
- Злиттям
- Пірамідальне
- Швидке

Що є спільного в цих алгоритмах сортування?

Сортування порівнянням

- Бульбашкове
- Вставками
- Вибором
- Злиттям
- Пірамідальне
- Швидке

Для сортування використовуються тільки порівняння вхідних елементів.

Нижні оцінки алгоритмів сортування

- Щоб відсортувати вхідну послідовність використовуються тільки попарні порівняння елементів:
  - для визначення взаємного порядку елементів a_i та a_j виконується одна з перевірок: $<, \leq, =, \geq, >$
  - значення самих елементів чи інша інформація про них не доступні.

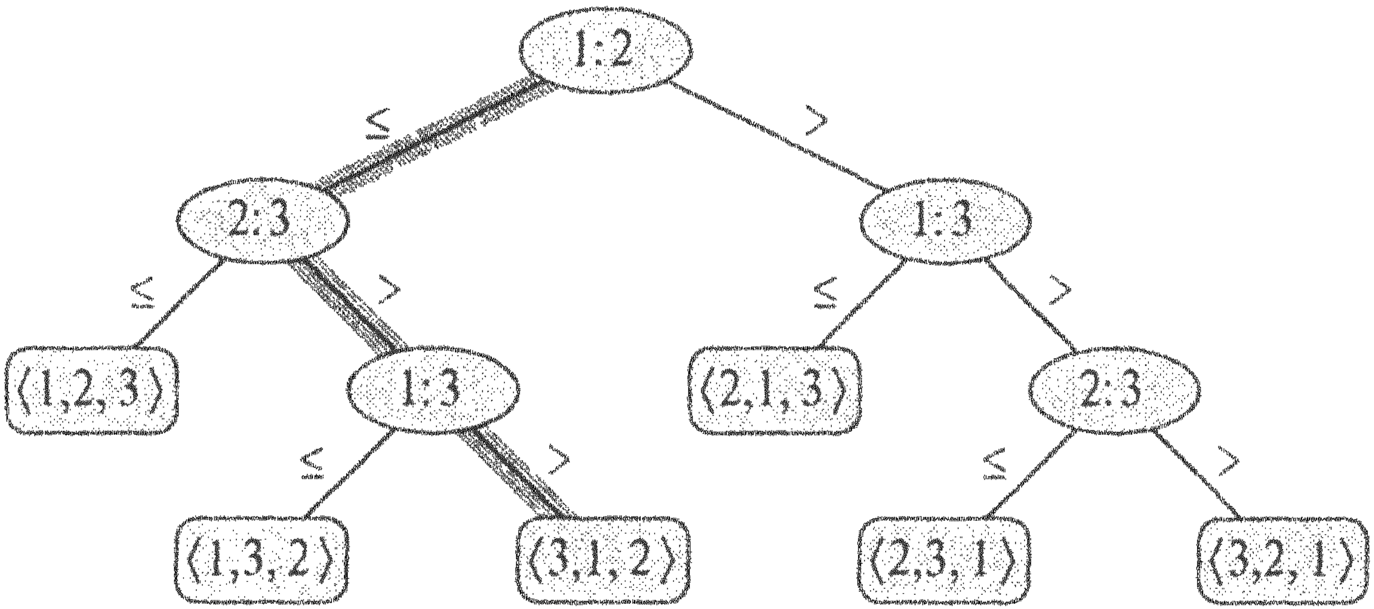
- Вважатимемо, що всі порівняння мають вигляд $\leq$.
- Надалі припустимо, що всі вхідні елементи різні, тоді всі інші операції порівняння дадуть по суті еквівалентну інформацію про взаємне розташування елементів.

Модель дерева розв’язків

- *Дерево розв’язків* (*decision tree*) – спосіб представлення правил в ієрархічній, послідовній структурі, де кожному об'єкту відповідає єдиний вузол, що дає розв'язок.
- В нашому випадку це повне бінарне дерево, де представлені операції порівняння елементів, що виконуються певним алгоритмом сортування над даними визначеного розміру.
- Всі інші аспекти алгоритму ігноруються.

Модель дерева розв’язків

Дерево розв’язків (вставка, 3 елементи)



Приклад сортування входу $a_1 = 6,\ a_2 = 8,\ a_3 = 5$

Модель дерева розв’язків

- Мітка кожного внутрішнього вузла: *i:j*, де $i \le j$, $j \le n$ (*n* – кількість вхідних елементів).
- Мітка *i:j* вказує, що порівнюються a_i та a_j.
- Кожен лист помічений перестановкою $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$, що позначає впорядкування елементів $\left\langle a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)} \right\rangle$.

Наприклад, на малюнку сірим наведена послідовність рішень, які приймає алгоритм при роботі над входом $a_1 = 6$, $a_2 = 8$, $a_3 = 5$.

Перестановка $\langle 3,1,2 \rangle$ означає, що

$$a_3 = 5 \le a_1 = 6 \le a_2 = 8.$$

Модель дерева розв’язків

- В кожному вузлі реалізується розгалуження: перевірка $a_i \le a_j$. Ліве піддерево міститиме всі порівняння за умови $a_i \le a_j$, праве – за умови $a_i > a_j$
- При досягненні листа визначається відповідне впорядкування елементів.
- Коректний алгоритм сортування має вміти здійснювати будь-яку перестановку вхідних елементів, тому листи дерева розв’язків повинні містити всі $n!$ перестановок n елементів, і до кожного листа можна прокласти шлях (досяжність листків).

Нижня оцінка найгіршого випадку

- Величина найдовшого шляху від кореня дерева розв'язків до будь-якого з його досяжних листків відповідає кількості порівнянь, які виконуються в розглянутому алгоритмі сортування в найгіршому випадку.
- Отже, кількість порівнянь, що виконуються в тому чи іншому алгоритмі сортування порівнянням в найгіршому випадку, дорівнює висоті його дерева розв'язків.
- Тому нижня оцінка висот для всіх дерев, в яких усі перестановки представлені досяжними листками, є нижньою оцінкою часу роботи для будь-якого алгоритму сортування порівнянням .

Нижня оцінка найгіршого випадку

Теорема. В найгіршому випадку в ході виконання будь-якого алгоритму сортування порівнянням виконується $\Omega(n \lg n)$ порівнянь.

Доведення. Для доведення теореми досить визначити висоту дерева, в якому кожна перестановка представлена досяжним листом.

Розглянемо дерево розв'язків висотою h з l досяжними листками, яке відповідає сортуванню порівнянням n елементів.

Оскільки кожна з $n!$ перестановок вхідних елементів зіставляється з одним з листків, то $n! \le l$.

Нижня оцінка найгіршого випадку

Оскільки бінарне дерево висоти h має не більше 2^h листів, отримуємо

$$n! \le l \le 2^h.$$

Прологарифмуємо вираз. В силу монотонності логарифму та співвідношення $\lg(n!) = \Theta(n \lg n)$ маємо: $h \geqslant \lg(n!) = \Omega(n \lg n)$.

Наслідок. Пірамідальне сортування та сортування злиттям – асимптотично оптимальні алгоритми сортування.

Доведення. Їх верхні границі часу роботи O(n lg n) збігаються з нижніми Ω(n lg n) для найгіршого випадку.

Сортування за лінійний час

- Вище було показано, що в класі алгоритмів сортування порівнянням не існує таких, що працюватимуть швидше за $\Omega(n \lg n)$.
- Існують алгоритми, засновані на принципах, відмінних від порівняння.
- Для них вже будуть інші оцінки часу роботи.
- Розглянемо алгоритми, швидкість роботи яких лінійно залежить від кількості вхідних елементів.

Сортування підрахунком (counting sort)

- Дано n вхідних цілих чисел з інтервалу від *0* до k, де k – ціла константа.
- Якщо $k = O(n)$, то час роботи сортування $\Theta(n)$.
- Для кожного вхідного елемента x підраховується кількість елементів, менших за нього. Ця інформація дозволяє розмістити елемент на своєму місці.
- Якщо серед елементів допускаються повтори, алгоритм потрібно буде трохи модифікувати.

Сортування підрахунком (counting sort)

Вхід: масив A[1..*n*] (length[A]=*n*)

Вихід: відсортований масив B[1..*n*]

Допоміжний масив C[0..*k*]

```
АЛГОРИТМ COUNTING_SORT (A, B, k)
for  i <= 0  to  k
   do  C[i] <= 0
for  j <= 1  to  length[A]
   do  C[A[j]] <= C[A[j]] + 1
// в C[i] – кількість елементів, що дорівнюють i
for  i <= 1  to  k
   do  C[i] <= C[i] + C[i – 1]
// в C[i] – кількість елементів, що не перевищують i
for  j <= length[A]   downto  1
   do  B[C[A[j]]] <= A[j]
       C[A[j]] <= C[A[j]] – 1
```

Вхід: A[1..8], *k = 5*

(а) рядок 5

(б) рядок 8

(в)-(д) три перші ітерації циклу рядків 9-11

(е) результат

```
АЛГОРИТМ COUNTING_SORT(A, B, k)
1  for i <= 0 to k
2    do C[i] <= 0
3  for j <= 1 to length[A]
4    do C[A[j]] <= C[A[j]] + 1
5  // В C[i] – кількість елементів, що дорівнюють i
6  for i <= 1 to k
7    do C[i] <= C[i] + C[i – 1]
8  // В C[i] – кількість елементів, що не перевищують i
9  for j <= length[A]  downto 1
10   do B[C[A[j]]] <= A[j]
11      C[A[j]] <= C[A[j]] – 1
```

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

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

а)

	0	1	2	3	4	5
C	2	2	4	7	7	8

б)

	1	2	3	4	5	6	7	8
B							3	

	0	1	2	3	4	5
C	2	2	4	6	7	8

в)

	1	2	3	4	5	6	7	8
B		0					3	

	0	1	2	3	4	5
C	1	2	4	6	7	8

г)

	1	2	3	4	5	6	7	8
B		0				3	3	

	0	1	2	3	4	5
C	1	2	4	5	7	8

д)

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

е)

Цикл **for** рядків 9-11

```
9   for j <= length[A]  downto  1
10    do  B[C[A[j]]] <= A[j]
11        C[A[j]] <= C[A[j]] – 1
```

- Кожен елемент A[*j*] поміщається в належну позицію вихідного масиву B.
- Якщо всі *n* елементів різні, то при першому переході до рядку 9 для кожного елемента A[*j*] у змінній C[A[*j*]] зберігається коректний індекс кінцевого положення цього елемента у вихідному масиві, оскільки існує C[A[*j*]] елементів, менших або рівних A[*j*].
- Оскільки різні елементи можуть мати одні й ті ж значення, поміщаючи значення A[*j*] в масив B, ми щоразу зменшуємо C[A[*j*]] на одиницю. Завдяки цьому наступний вхідний елемент, значення якого дорівнює A[*j*] (якщо він існує), у вихідному масиві розміщується безпосередньо перед елементом A[*j*].

Час виконання сортування підрахунком

- Цикл **for** рядки 1-2: Θ(*k*)
- Цикл **for** рядки 3-4: Θ(*n*)
- Цикл **for** рядки 6-7: Θ(*k*)
- Цикл **for** рядки 9-11: Θ(*n*)

```
1   for i <= 0 to k
2      do C[i] <= 0

3   for j <= 1 to length[A]
4      do C[A[j]] <= C[A[j]] + 1

6   for i <= 1 to k
7      do C[i] <= C[i] + C[i – 1]

9   for j <= length[A] downto 1
10     do B[C[A[j]]] <= A[j]
11        C[A[j]] <= C[A[j]] – 1
```

Отже, повний час Θ(*k*+*n*)

На практиці алгоритм застосовують при *k* = O(*n*), тоді час роботи складе Θ(*n*).

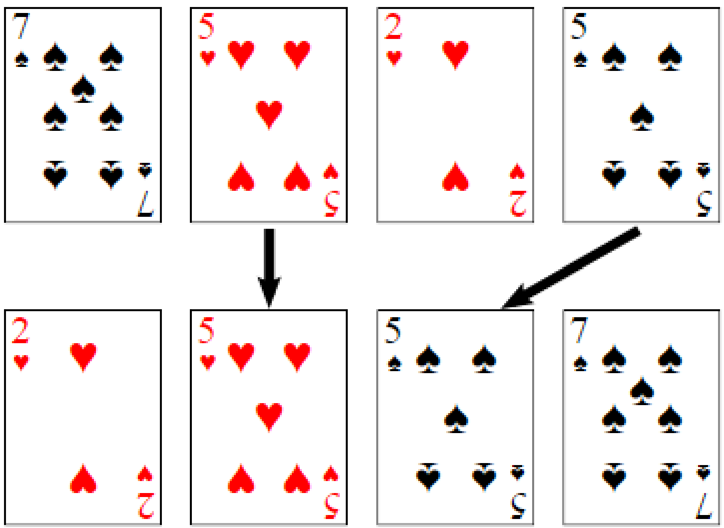
Сортування підрахунком

- Код не містить порівнянь вхідних значень.
- Замість цього за допомогою їх значень (самі цілочисельні невід’ємні числа) елементам співставляються конкретні індекси.
- Алгоритм є *стійким* (stable):

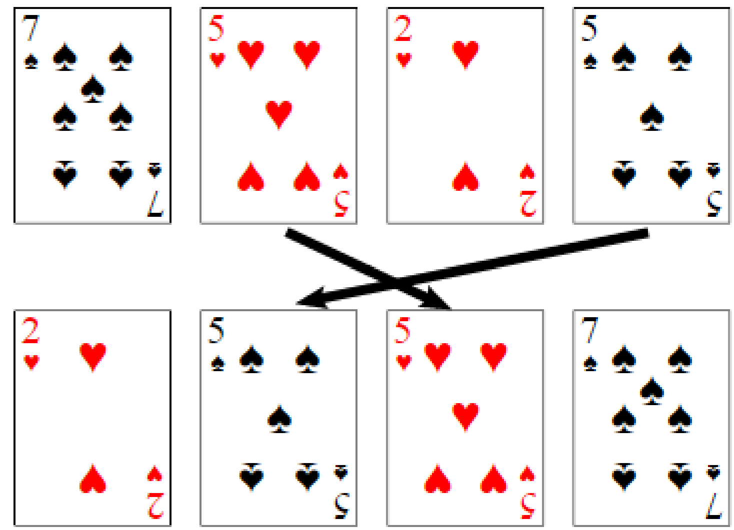
  елементи з однаковими значеннями стоять у вихідному масиві в тому ж порядку, що й у вхідному.

Стійкі / нестійкі алгоритми сортування

Стійкий алгоритм



Нестійкий алгоритм



Стійкі / нестійкі алгоритми сортування

Бульбашкове?

Стійке **Нестійке**

Стійкі / нестійкі алгоритми сортування

Вставками?

Стійке

– Бульбашкове

Нестійке

Стійкі / нестійкі алгоритми сортування

Вибором?

Стійке **Нестійке**

– Бульбашкове

– Вставками

Стійкі / нестійкі алгоритми сортування

Злиттям?

Стійке

– Бульбашкове

– Вставками

Нестійке

– Вибором

Стійкі / нестійкі алгоритми сортування

Швидке?

Стійке

– Бульбашкове

– Вставками

– Злиттям

Нестійке

– Вибором

Стійкі / нестійкі алгоритми сортування

Пірамідальне?

Стійке

– Бульбашкове

– Вставками

– Злиттям

Нестійке

– Вибором

– Швидке

Стійкі / нестійкі алгоритми сортування

Стійке

– Бульбашкове

– Вставками

– Злиттям

Нестійке

– Вибором

– Швидке

– Пірамідальне

Порозрядне сортування (radix sort)

- Алгоритм використовувався в машинах для сортування перфокарт (перше документоване посилання – 1929 рік).
- Сортуються числа з фіксованою кількістю цифр.
- Послідовно виконується сортування по кожному з розрядів, починаючи з наймолодшого.
- Таким чином, кількість проходів відповідає кількості розрядів числа.

Вважаємо, що кожен елемент масиву A – d-цифрове число, причому перша цифра відповідає молодшому розряду

329		720		720		329
457		355		329		355
657		436		436		436
839	→	457	→	839	→	457
436		657		355		657
720		329		457		720
355		839		657		839

АЛГОРИТМ *RADIX_SORT* (A, *d*)
for *i* <= 1 **to** *d*
  do Стійке сортування масиву A по *i*-й цифрі

Порозрядне сортування

<u>**Твердження 1**</u>. Нехай є *n* *d*-значних чисел, в яких кожна цифра приймає одне з *k* можливих значень. Тоді алгоритм Radix_Sort дозволяє виконати коректне сортування цих чисел за час Θ(*d*(n+k)), якщо використовується стійке сортування з часом роботи Θ(n+k).

<u>Доведення.</u> Коректність доводиться індукцією по стовпцям, які сортуються. Час залежить від конкретного стійкого алгоритму сортування. Якщо кожна цифра належить інтервалу 0..*k*, при цьому *k* невелике, то можна брати сортування підрахунком. Для обробки кожної з *d* цифр *n* чисел треба часу Θ(n+k), тоді для всіх цифр час буде Θ(*d*(n+k)).

Порозрядне сортування

Якщо d – константа, а $k = O(n)$, то час роботи алгоритму буде $\Theta(n)$. Інакше маємо

<u>**Твердження 2.**</u> Нехай є n b-бітових чисел та натуральне $r \le b$. Тоді алгоритм Radix_Sort дозволяє виконати коректне сортування цих чисел за час $\Theta((b/r)(n+2^r))$.

<u>Доведення.</u> Для $r \le b$ кожний ключ можна розглядати як число, що складається з $d = \lceil b/r \rceil$ цифр по r бітів кожна. Всі цифри є цілими в інтервалі від 0 до $(2^r - 1)$, тому можна скористатися алгоритмом сортування підрахунком з $k = 2^r - 1$. Кожен його прохід займе $\Theta(n+k) = \Theta(n+2^r)$. Усього проходів d, тоді повний час $\Theta(d(n+2^r)) = \Theta((b/r)(n+2^r))$.

Порозрядне сортування

Наприклад, 32-бітне слово можна розглянути як число, що складається з чотирьох 8-бітових цифр, так що $b = 32$, $r = 8$, $k = 2^r - 1 = 255$, $d = b / r = 4$.

Виберемо для двох заданих значень n та b таку величину $r \le b$, що мінімізувала би вираз $(b/r)(n+2^r)$.

- Якщо $b < \lfloor \lg n \rfloor$, то будь-яке $r \le b$ дає $(n+2^r)=\Theta(n)$, тому можна вибрати $r = b$.
- Якщо $b \geq \lfloor \lg n \rfloor$, то найкращий час отримується при $r = \lfloor \lg n \rfloor$ і дорівнює $\Theta(b \cdot n / \lg n)$.

Порозрядне сортування vs Quicksort

- При b = O(lg n) і виборі $r \approx \lg \mathrm{n}$ час роботи порозрядного сортування Θ(n).
- Середній час роботи швидкого сортування складає Θ(n lg n)
- Однак кожен прохід порозрядного сортування може тривати суттєво довше.
- Якщо в якості проміжного сортування вибране сортування підрахунком, знадобиться додаткова пам’ять, тоді як швидке сортування обробляє елементи на місці.

Сортування черпаками (bucket sort)

- Вважаємо, що всі вхідні елементи генеруються випадковим процесом і рівномірно розподілені в інтервалі [0,1).
- Інтервал розбивається на *n* однакових інтервалів (черпаків, кишень), по яким розподіляються вхідні величини.
- Припускається, що до кожного черпака потрапить небагато елементів.
- Елементи в черпаках сортуються.
- Для отримання відсортованого результату послідовно перелічимо елементи кожного з черпаків.

Сортування черпаками (bucket sort)

Кожен елемент n-елементного масиву $A[i] \in [0,1)$.

Потрібен допоміжний масив для черпаків $B[0..(n-1)]$

АЛГОРИТМ *BUCKET_SORT* (A)

n <= *length*[A]

for i <= 1 **to** n

do Вставити елемент A[i] в список $B[\lfloor nA[i] \rfloor]$

for i <= 0 **to** $n - 1$

do Сортування вставкою списку B[i]

Об’єднання списків B[0], B[1], …, B[$n - 1$]

Час роботи сортування черпаками складає $\Theta(n)$ в середньому.

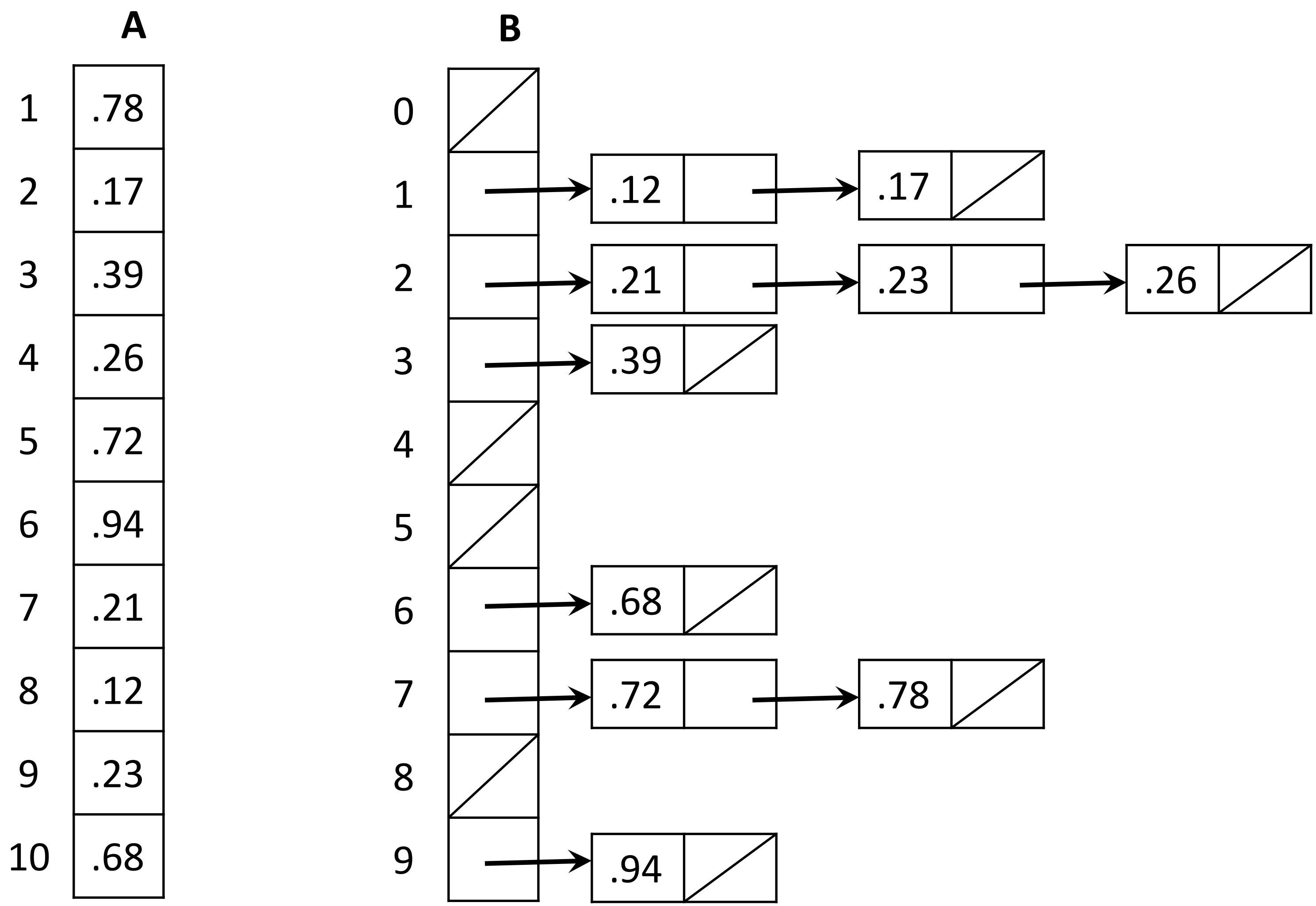
Сортування черпаками (bucket sort)

Покажемо, що сортування працює коректно.

- Розглянемо елементи A[i] та A[j], причому не обмежуючи загальності нехай A[*i*]≤A[*j*].
- Оскільки $\lfloor n\mathrm{A}[i] \rfloor \le \lfloor n\mathrm{A}[j] \rfloor$, то елемент A[*i*] поміщається або в одну кишеню з A[j], або в кишеню з меншим індексом.
- В першому випадку елементи A[i] та A[j] будуть розташовані в правильному порядку внаслідок сортування кишені. В другому – при об’єднанні результатів з кишень.

Час роботи Θ(*n*) в середньому збережеться навіть якщо вхідні елементи не будуть рівномірно розподілені, за умови, якщо сума квадратів розмірів кишень лінійно залежить від кількості вхідних елементів.

Сортування черпаками (bucket sort)



Медіани та порядкові статистики

- *i*-та *порядкова статистика* *n*-елементної множини – її *i*-й елемент в порядку зростання.
- *Мінімум* – перша порядкова статистика (*i = 1*).
- *Максимум* – *n*-та порядкова статистика (*i = n*).
- *Медіана* – середина множини: при $i = (n+1)/2$.
- *Нижня медіана* – при $i = \lfloor (n+1)/2 \rfloor$.
- *Верхня медіана* – при $i = \lceil (n+1)/2 \rceil$.

Надалі для простоти під медіаною розумітимемо нижню медіану.

Задача вибору

- Нехай маємо n-елементну множину, в якої всі елементи різні (можливе узагальнення на наявність повторів) та число i ($1 \le i \le n$).
- Треба знайти i-ту порядкову статистику – елемент множини, більший рівно за (i – 1) її елементів.

Можна розв'язати її за час O(n lg n): відсортувати масив пірамідою чи злиттям та повернути i-й елемент, але існують швидші алгоритми.

Пошук максимуму та мінімуму

- Пошук мінімального елемента

АЛГОРИТМ *MINIMUM* (A)

min <= A[1]

for i <= 2 **to** $length$[A]

do **if** min > A[i]

then min <= A[i]

return min

Для пошуку потрібно рівно (n – 1) порівняння, цей алгоритм оптимальний.

- Аналогічно може шукатися максимум.

Одночасний пошук максимуму і мінімуму

- Незалежно знаходиться мінімум та максимум (в сумі $(2n-2)$ порівняння).
- Можна зменшити кількість порівнянь:
  – беремо вхідні елементи парами,
  – порівнюємо один з одним,
  – менший порівнюємо з поточним мінімумом, більший – з поточним максимумом,
  – при непарній кількості елементів спочатку беремо один елемент як початковий максимум та мінімум.
- При непарній кількості маємо $3\lfloor n/2\rfloor$ порівнянь.
- При парній кількості: початкове порівняння та $3(n-2)/2$ порівнянь – в сумі $3n/2-2$ порівнянь.
- В обох випадках кількість порівнянь не перевищить $3\lfloor n/2\rfloor$.

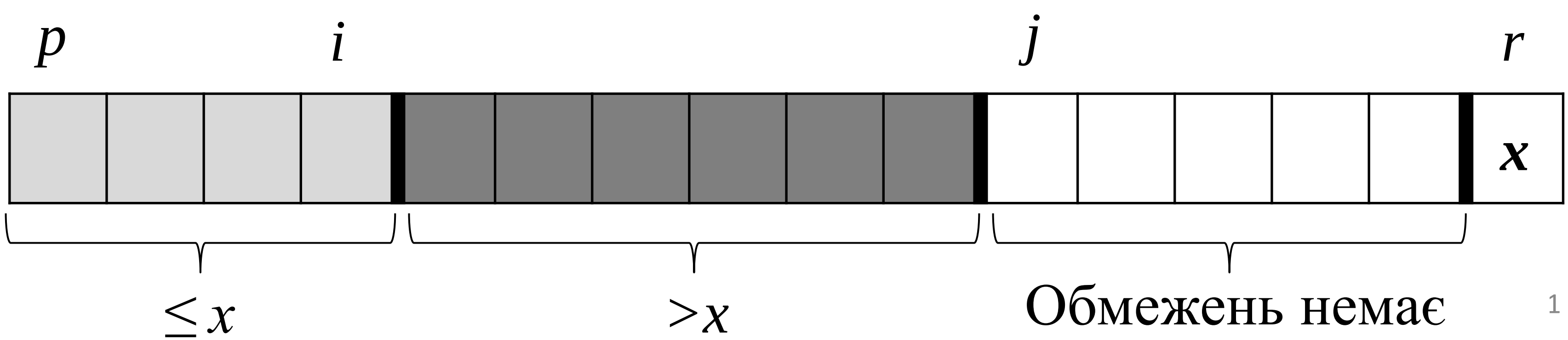
Рандомізований алгоритм пошуку і-ї статистики

- Ідея запозичена зі швидкого пошуку. При цьому рекурсивно обробляється лише одна з частин масиву.
- Середній час виконання $\Theta(n)$.

```
АЛГОРИТМ RANDOMIZED_SELECT (A, p, r, i)
if  p = r
   then  return  A[p]
q <= RANDOMIZED_PARTITION(A, p, r)
k <= q – p + 1
if  i = k              // Опорне значення – це відповідь
   then  return  A[q]
   else if  i < k
      then  return   RANDOMIZED_SELECT(A, p, q–1, i)
      else return  RANDOMIZED_SELECT(A, q+1, r, i–k)
```

```
АЛГОРИТМ RANDOMIZED_PARTITION (A, p, r)
i <= RANDOM(p, r)
Обміняти  A[r] <=> A[i]
return  PARTITION(A, p, r)
```

```
АЛГОРИТМ PARTITION (A, p, r)
x <= A[r]
i <= p – 1
for  j <= p  to  r – 1
    do  if  A[j] <= x
          then  i <= i + 1
                Обміняти  A[i] <=> A[j]
Обміняти  A[i + 1] <=> A[r]
return i + 1
```



Алгоритм вибору з лінійним часом в найгіршому випадку

- Алгоритм SELECT має схожий принцип пошуку потрібного елемента шляхом рекурсивного розбиття вихідного масиву.
- Але в його основі лежить ідея, що потрібно *гарантувати* хороше розбиття масиву.
- По ходу використовується модифікована процедура PARTITION зі швидкого сортування, яка містить додатковий параметр – конкретний елемент, відносно якого відбувається розбиття.
- При $n = 1$ процедура SELECT повертає єдине вхідне значення, інакше відбувається наступне.

Алгоритм вибору з лінійним часом в найгіршому випадку

1. Всі *n* елементів розбиваються на $\lfloor n/5 \rfloor$ груп по 5 елементів в кожній і одну з рештою *n* mod 5 елементів (можливо, порожню).
2. Вставками сортується кожна з груп, потім в кожному списку вибирається медіана.
3. Рекурсивно через дану процедуру SELECT шукається медіана *x* множини медіан, знайдених на кроці 2.
4. Поділ за допомогою модифікованої процедури PARTITION відносно медіани медіан *x*. Нехай число *k* на одиницю перевищує число елементів, що потрапили до нижньої частини розбиття. Тоді *x* – *k*-й в порядку зростання елемент, і до верхньої частини розбиття потрапляє (*n* – *k*) елементів.
5. При *i* = *k* повертається значення *x*. Інакше процедура викликається рекурсивно і шукається *i*-й в порядку зростання елемент в нижній частині при *i* < *k* або у верхній частині при *i* > *k* (як в попередньому алгоритмі).

Алгоритм вибору з лінійним часом в найгіршому випадку

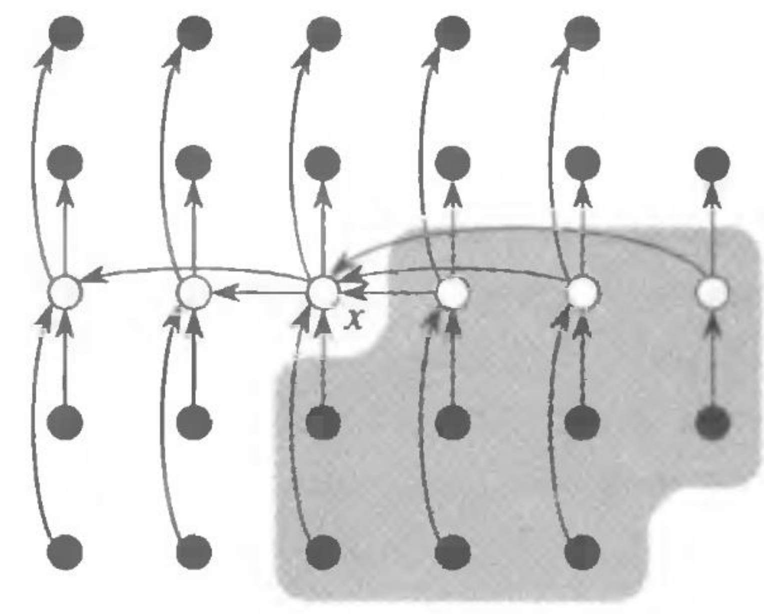
Кожна група – стовпчик.

Медіани груп – білі кружечки.

x – медіана медіан.

Стрілки йдуть від більших елементів до менших.

На сірому фоні елементи, що більші за *x.*



В кожній повній групі з 5 елементів справа від *x* міститься по 3 елементи, що його перевищують, а в кожній повній групі зліва від *x* – по 3 елементи, що менші за нього.

Алгоритм вибору з лінійним часом в найгіршому випадку

- Вважаючи, що всі елементи різні, як мінімум половина медіан, знайдених на кроці 2, більше чи дорівнюють медіані медіан *x*.
- Тоді половина з $\lceil n/5 \rceil$ груп міститиме по 3 елементи, які перевищують *x*, окрім тієї, що сама включає *x* і, можливо, групи-залишку, де менше 5 елементів.
- Отже, кількість елементів, що перевищують *x*, складе, як мінімум
$$3\left(\left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$$
- Аналогічно, знайдеться не менше 3*n*/10 – 6 елементів, що менші за *x*.
- Загалом, на кроці 5 буде виклик SELECT для максимум 7*n*/10 + 6 елементів.

Алгоритм вибору з лінійним часом в найгіршому випадку

- Сформулюємо рекурентне співвідношення T(n) для роботи SELECT в найгіршому випадку.
- Кроки 1, 2 і 4 – час O(n) (крок 2 містить O(n) викликів сортування вставками для множин розміру O(1)).
- Крок 3 – час $\mathrm{T}(\lceil n/5 \rceil)$.
- Крок 5 – час не більший за T(7n/10 + 6).
- Припустимо, масив розміром меншим за 140, обробляється за константний час.

$$\mathrm{T}(n) \le \begin{cases} \mathrm{O}(1), & \text{при } n < 140, \\ \mathrm{T}(\lceil n/5 \rceil) + \mathrm{T}(7n/10 + 6) + \mathrm{O}(n), & \text{при } n \ge 140. \end{cases}$$

Алгоритм вибору з лінійним часом в найгіршому випадку

$$\mathrm{T}(n) \le \begin{cases} \mathrm{O}(1), & \text{при } n < 140, \\ \mathrm{T}(\lceil n/5 \rceil) + \mathrm{T}(7n/10 + 6) + \mathrm{O}(n), & \text{при } n \ge 140. \end{cases}$$

- Покажемо методом підстановки, що час роботи SELECT лінійний: виконується T(*n*) ≤ *cn* для всіх *n* > 0 та достатньо великого *c*.
- Нехай при *n* < 140 все виконується, тоді для *n* ≥ 140:

$$\begin{aligned} T(n) &\le c\lceil n/5 \rceil + c(7n/10 + 6) + an \\ &\le cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) \;. \end{aligned}$$

Алгоритм вибору з лінійним часом в найгіршому випадку

- Отримали $T(n) \le cn + (-cn/10 + 7c + an)$.
- Вираз не перевищить значення cn, якщо виконується

$$-cn/10 + 7c + an \le 0.$$

- При $n > 70$ нерівність еквівалентна $c \ge 10a(n/(n-70))$.
- За припущенням $n \ge 140$, тоді $n/(n-70) \le 2$ і можна вибрати $c \ge 20a$, щоб справджувалась вихідна нерівність.
- В рекурентному співвідношенні в якості початкової умови замість 140 можна взяти довільне натуральне число більше 70.

Порівняння з алгоритмами сортування

- Розглянуті алгоритми, на відміну від алгоритмів сортування, які працюють за лінійний час, не вимагають ніяких додаткових припущень щодо вигляду вхідних даних.
- Час роботи виявився лінійним тому, що не застосовуються сортування.
- Тому початкова пропозиція з використанням сортування буде неефективною.

Запитання і завдання

- Сортування на місці за лінійний час

 Нехай маємо масив, що містить *n* записів з даними для сортування, і що ключ кожного запису приймає значення 0 або 1. Алгоритм для сортування такого набору записів повинен мати деякі з трьох наступних характеристик:

1) час роботи алгоритму O(*n*);

2) алгоритм має бути стійким;

3) сортування проводиться на місці, тобто крім вихідного масиву використовується додаткова пам’ять, що не перевищує деякої постійної величини.

A. Розробіть алгоритм, що задовольняє критеріям 1 і 2.

B. Розробіть алгоритм, що задовольняє критеріям 1 і 3.

C. Розробіть алгоритм, що задовольняє критеріям 2 і 3.

Запитання і завдання

- Нехай n записів мають ключі, значення яких знаходяться в інтервалі від 1 до k. Покажіть, як можна модифікувати алгоритм сортування підрахунком, щоб забезпечити сортування цих записів на місці за час $O(n+k)$. Додатково можна використовувати пам'ять об'ємом $O(k)$. Чи стійкий цей алгоритм? (*Вказівка*: подумайте, як можна розв‘язати задачу для $k = 3$)
- Сортування елементів змінної довжини

  Дано масив цілих чисел, причому різні його елементи можуть мати різну кількість цифр; але загальна кількість цифр в усіх числах дорівнює n. Потрібно відсортувати цей масив за час $O(n)$.

Запитання і завдання

- За визначенням, k-ми квантилями (quantiles) n-елементної множини називають (k – 1) порядкових статистик, що розбивають цю відсортовану множину на k однакових підмножин (з точністю до одного елемента). Сформулюйте алгоритм, який би виводив список k-х квантилів множини за час O(n lg k).
- Опишіть алгоритм, який для заданої множини S, що складається з n різних чисел, і додатної цілої константи $k \le n$ визначав би k найближчих сусідів медіани множини S, що належать до цієї множини. Час роботи має складати O(n).
- Нехай X[1..n] та Y[1..n] – два масиви, кожен з яких містить по n вже відсортованих елементів. Розробіть алгоритм, в якому пошук медіани всіх 2n елементів, що містяться в масивах X та Y, виконується за час O(lg n).

Запитання і завдання

- Найбільші *n* елементів в порядку сортування

  Нехай є *n*-елементна множина, в якій за допомогою алгоритму, заснованого на порівняннях, потрібно знайти *i* найбільших елементів, розташованих в порядку сортування. Сформулюйте алгоритми, які реалізують кожен з указаних нижче методів з найкращим можливим асимптотичним часом роботи в найгіршому випадку. Проаналізуйте залежність часу роботи цих алгоритмів від *n* та *i*.

A. Всі числа сортуються та виводяться *i* найбільших.

B. Створіть із чисел незростаючу пріоритетну чергу та *i* раз викличте процедуру ExtractMax.

C. Знайдіть за допомогою алгоритму порядкової статистики *i*-й по порядку найбільший елемент, зробіть розбиття відносно нього та виконайте сортування *i* найбільших чисел.