Алгоритми та складність

І семестр

Лекція 3

Аналіз нерекурсивних алгоритмів

Приклад 1. Пошук найбільшого елемента в списку з *n* чисел (список у вигляді масиву).

АЛГОРИТМ *MaxElement* (A[0 .. $n-1$])
// **Вхідні дані:** масив дійсних чисел A[0 .. $n-1$]
// **Вихідні дані:** повертається значення найбільшого елемента
// в масиві A
maxval <= A[0]
for i <= 1 **to** $n-1$ **do**
  if A[i] > *maxval*
    maxval <= A[i]
return *maxval*

- від чого залежить розмір вхідних даних?
- основні операції в циклі? яку з них обрати базовою?
- чи потрібно окремо розглядати складність в найгіршому, найкращому випадках та в середньому?

Приклад 1 (закінчення).

```
АЛГОРИТМ  MaxElement ( A[0 .. n − 1] )
// Вхідні дані:      масив дійсних чисел A[0 .. n − 1]
// Вихідні дані:     повертається значення найбільшого елемента
//                   в масиві A
maxval <= A[0]
for  i <= 1  to  n − 1 do
   if  A[i] > maxval
         maxval <= A[i]
return maxval
```

Нехай *C(n)* – кількість операцій порівняння в алгоритмі при розмірі входу *n*.

Одне виконання циклу – одне порівняння, і так для кожного значення змінної циклу *i*, що змінюється від 1 до (*n*–1). Тому

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

Деякі правила сумування

$$\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i \qquad \text{(R1)}$$

$$\sum_{i=l}^{u}(a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i \qquad \text{(R2)}$$

$$\sum_{i=l}^{u} 1 = u - l + 1, \text{ де } l \leq u \ - \text{цілі числа,}$$

що представляють нижню та верхню межі суми (S1)

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2) \ \text{(S2)}$$

Загальний план аналізу ефективності нерекурсивних алгоритмів

1. Виберіть параметр (або параметри), за яким буде оцінюватися розмір вхідних даних алгоритму.

Загальний план аналізу ефективності нерекурсивних алгоритмів

1. Виберіть параметр (або параметри), за яким буде оцінюватися розмір вхідних даних алгоритму.

2. Визначте основну операцію алгоритму. (Як правило, вона знаходиться в найглибше вкладеному внутрішньому циклі алгоритму.)

Загальний план аналізу ефективності нерекурсивних алгоритмів

1. Виберіть параметр (або параметри), за яким буде оцінюватися розмір вхідних даних алгоритму.

2. Визначте основну операцію алгоритму. (Як правило, вона знаходиться в найглибше вкладеному внутрішньому циклі алгоритму.)

3. Перевірте, чи залежить число виконуваних основних операцій лише від розміру вхідних даних. Якщо воно залежить ще й від інших факторів, розгляньте за необхідності, як змінюється ефективність алгоритму для найгіршого, середнього і найкращого випадків.

Загальний план аналізу ефективності нерекурсивних алгоритмів

1. Виберіть параметр (або параметри), за яким буде оцінюватися розмір вхідних даних алгоритму.

2. Визначте основну операцію алгоритму. (Як правило, вона знаходиться в найглибше вкладеному внутрішньому циклі алгоритму.)

3. Перевірте, чи залежить число виконуваних основних операцій лише від розміру вхідних даних. Якщо воно залежить ще й від інших факторів, розгляньте за необхідності, як змінюється ефективність алгоритму для найгіршого, середнього і найкращого випадків.

4. Запишіть суму, що виражає кількість виконуваних основних операцій алгоритму.

Загальний план аналізу ефективності нерекурсивних алгоритмів

1. Виберіть параметр (або параметри), за яким буде оцінюватися розмір вхідних даних алгоритму.

2. Визначте основну операцію алгоритму. (Як правило, вона знаходиться в найглибше вкладеному внутрішньому циклі алгоритму.)

3. Перевірте, чи залежить число виконуваних основних операцій лише від розміру вхідних даних. Якщо воно залежить ще й від інших факторів, розгляньте за необхідності, як змінюється ефективність алгоритму для найгіршого, середнього і найкращого випадків.

4. Запишіть суму, що виражає кількість виконуваних основних операцій алгоритму.

5. Використовуючи стандартні формули і правила підсумовування, спростіть отриману формулу для основних операцій алгоритму. Якщо це неможливо, визначте хоча б їх порядок зростання.

Аналіз нерекурсивних алгоритмів

Приклад 2. Перевірка єдиності елементів масиву.

АЛГОРИТМ *UniqueElements (* A[0 .. $n - 1$])
// **Вхідні дані:** масив дійсних чисел A[0 .. $n - 1$]
// **Вихідні дані:** повертається значення "**true**", якщо всі
// елементи масиву A різні, та "**false**" інакше.
for i <= 0 **to** $n - 2$ **do**
  for j <= $i + 1$ **to** $n - 1$ **do**
    if A[i] = A[j]
      return false
return true

- від чого залежить розмір вхідних даних?
- яка базова операція?
- чи потрібно окремо розглядати складність в найгіршому, найкращому випадках та в середньому?

Приклад 2 (продовження).

```
АЛГОРИТМ  UniqueElements ( A[0 .. n − 1] )
// Вхідні дані:    масив дійсних чисел A[0 .. n − 1]
// Вихідні дані:   повертається значення “true”, якщо всі
//                 елементи масиву A різні, та “false” інакше.
for  i <= 0  to  n − 2  do
   for  j <= i + 1  to  n − 1  do
      if  A[i] = A[j]
         return false
return true
```

Обмежимося розглядом найгіршого випадку.

Які найгірші випадки вхідних даних?

Приклад 2 (продовження).

АЛГОРИТМ *UniqueElements* (A[0 .. $n-1$])
// **Вхідні дані:** масив дійсних чисел A[0 .. $n-1$]
// **Вихідні дані:** повертається значення "**true**", якщо всі
// елементи масиву A різні, та "**false**" інакше.
for i <= 0 **to** $n-2$ **do**
  for j <= $i+1$ **to** $n-1$ **do**
    if A[i] = A[j]
      return false
return true

Обмежимося розглядом найгіршого випадку.

Які найгірші випадки вхідних даних?

- єдині два однакові елементи розташовані в самому кінці масиву;
- всі елементи масиву різні.

Приклад 2 (закінчення).

В найгіршому випадку і зовнішній, і внутрішній цикли мають виконатися до кінця. Тому

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) =$$

$$= (n-1) + (n-2) + \cdots + 1 \overset{(S2)}{=} \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

Даний результат можна було отримати і з міркувань, що у найгіршому випадку для масиву з n елементів знадобиться порівняти між собою всі $(n-1)n/2$ різних пар елементів.

Аналіз нерекурсивних алгоритмів

Приклад 3. Множення квадратних матриць.

```
АЛГОРИТМ  MatrixMultiplication ( A[0 .. n − 1, 0 .. n − 1], B[0 .. n − 1, 0 .. n − 1] )
// Виконується множення двох квадратних матриць розміром n x n.
// В основі алгоритму лежить визначення цієї операції
// Вхідні дані:     дві квадратні матриці A та B розміром n x n
// Вихідні дані:    матриця C = AB
for  i <= 0  to  n − 1  do
   for  j <= 0  to  n − 1  do
      C[i, j] <= 0.0
      for  k <= 0  to  n − 1  do
         C[i, j] <=  C[i, j] + A[i, k] * B[k, j]
return C
```

- від чого залежить розмір вхідних даних?
- основні операції в циклі? яку з них обрати базовою?
- чи потрібно окремо розглядати складність в найгіршому, найкращому випадках та в середньому?

Приклад 3 (продовження).

for i <= 0 **to** $n-1$ **do**
  for j <= 0 **to** $n-1$ **do**
    C[i, j] <= 0.0
    for k <= 0 **to** $n-1$ **do**
      C[i, j] <= C[i, j] + A[i, k] * B[k, j]
return C

Можна підраховувати як кількість множень, так і кількість додавань, результат вийде однаковий. Розпишемо суму для множення. При кожній внутрішній ітерації виконується одна операція множення. Тому загалом отримуємо:

$$M(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1.$$

Користуючись формулами (R1) та (S1), маємо

$$M(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

Приклад 3 (закінчення).

Якщо вважати, що c_m – час виконання операції множення на деякому комп’ютері, то загальний час виконання алгоритму буде

$$T(n) \approx c_m M(n) = c_m n^3$$

Точнішою буде оцінка з урахуванням часу виконання команд додавання c_a:

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3$$

Однак різниця буде лише в коефіцієнтах, а не в порядках зростання.

Проаналізуйте алгоритм множення неквадратних матриць

Аналіз нерекурсивних алгоритмів

Приклад 4. Кількість розрядів у двійковому представленні натурального числа.

```
АЛГОРИТМ  Binary (n)
// Вхідні дані:    ціле додатне число n
// Вихідні дані:  кількість розрядів
//                          в двійковому представленні n
count <= 1
while  n > 1  do
   count <= count + 1
   n <= ⌊n/2⌋
return  count
```

- від чого залежить розмір вхідних даних?
- основні операції в циклі? яку з них обрати базовою?
- чи потрібно окремо розглядати складність в найгіршому, найкращому випадках та в середньому?

Приклад 4 (продовження).

```
АЛГОРИТМ  Binary (n)
// Вхідні дані:     ціле додатне число n
// Вихідні дані:   кількість розрядів
//                         в двійковому представленні n
count <= 1
while  n > 1  do
   count <= count + 1
   n <= ⌊n/2⌋
return  count
```

Скільки разів виконається цикл?

Приклад 4 (закінчення).

```
АЛГОРИТМ  Binary (n)
// Вхідні дані:     ціле додатне число n
// Вихідні дані:   кількість розрядів
//                          в двійковому представленні n
count <= 1
while  n > 1  do
   count <= count + 1
   n <= ⌊n/2⌋
return  count
```

Скільки разів виконається цикл?

При кожній ітерації значення *n* зменшується приблизно вдвічі, тому кількість циклів буде мати порядок $\log_2 n$ (якщо точно, то $\lfloor \log_2 n \rfloor + 1$).

Метод зменшення розміру задачі

- Використовується співвідношення між розв’язком даного екземпляру задачі і розв’язком меншого екземпляру тої самої задачі.
- Таке співвідношення може використовуватися або згори донизу (як правило, рекурсивно), або знизу вгору (без рекурсії).
- Основні різновиди методу:
  - зменшення на постійну величину (найчастіше на одиницю): сортування вставкою;
  - зменшення на постійний множник (найчастіше вдвічі): бінарний пошук;
  - змінне зменшення розміру: алгоритм Евкліда для обчислення НСД.

Інваріанти циклу і коректність алгоритму

Розглянемо найпопулярніший різновид алгоритму сортування вставками:

<u>Приклад 5.</u> Сортування вставками масиву.

АЛГОРИТМ *Insertion_Sort* (A)
for *j* <= 2 **to** *length*[*A*]
  do *key* <= A[*j*]
// Вставка елемента A[*j*] у відсортовану послідовність A[1.. *j*–1]
  i <= *j* – 1
  while *i* > 1 та A[*i*] > *key*
    do A[*i* + 1] <= A[*i*]
      i <= *i* – 1
  A[*i* + 1] <= *key*

На виході отримуємо відсортований масив *A*.

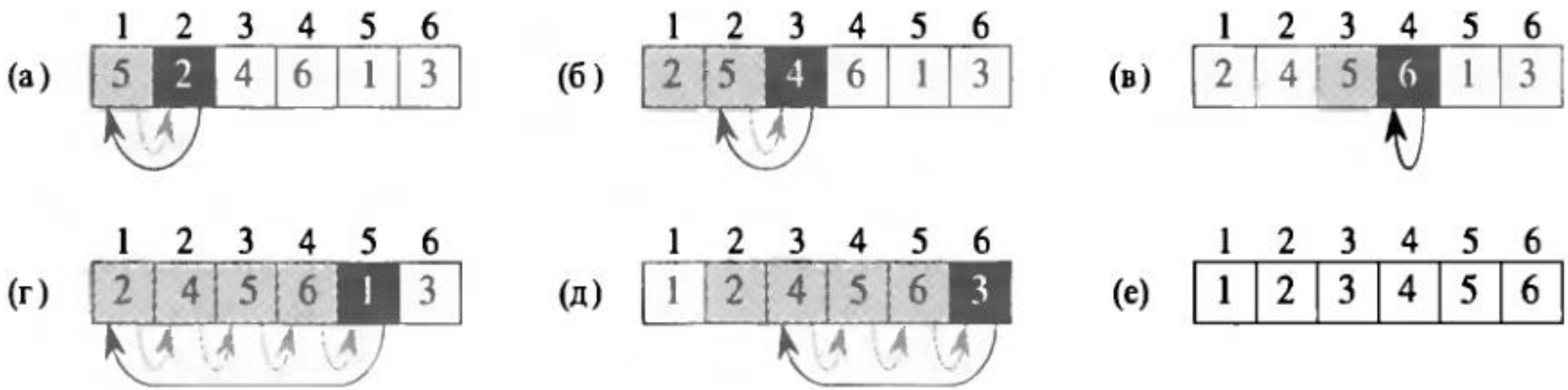
Інваріанти циклу і коректність алгоритму

Підходи до сортування вставками:

- сканування підмасиву зліва направо;
- сканування підмасиву справа наліво (краще працює з відсортованими і майже відсортованими масивами) – використовують на практиці;
- бінарне сортування вставкою: позиція вставки елемента визначається бінарним пошуком.

Інваріанти циклу і коректність алгоритму

<u>Приклад 5 (далі).</u> Ілюстрація роботи сортування вставками для *A=<5,2,4,6,1,3>*



На початку виконання *j*-го кроку (ітерації) елементи на позиціях з 1 по (*j*–1) виявляються відсортованими. Під час виконання тіла циклу елемент, що був на позиції *j*, «знаходить» собі місце.

Інваріанти циклу і коректність алгоритму

Приклад 5 (завершення).

АЛГОРИТМ *Insertion_Sort* (A)
for *j* <= 2 **to** *length*[*A*]
  do *key* <= A[*j*]
// Вставка елемента A[*j*] у відсортовану послідовність A[1.. *j*–1]
  i <= *j* – 1
  while *i* > 1 та A[*i*] > *key*
    do A[*i* + 1] <= A[*i*]
      i <= *i* – 1
   A[*i* + 1] <= *key*

- від чого залежить розмір вхідних даних?
- як виглядає найкращий випадок і який при цьому час роботи алгоритму?
- вигляд найгіршого випадку?

Інваріанти циклу і коректність алгоритму

- Кількість порівнянь для сортування вставками при роботі над довільним масивом (середній випадок) буде десь вдвічі меншою, ніж при роботі над "розвернутим" масивом (найгірший випадок).
- Це краще, ніж в алгоритмах бульбашкового сортування і сортування вибором, але останній може виявитися бажанішим при високій ціні запису в пам’ять (флеш-пам’ять).
- Сортування вставками використовують для оптимальніших реалізацій швидкого сортування і сортування злиттям (підмасиви до 10-20 елементів).
- Модифікований алгоритм – сортування Шелла – порівнює елементи з кроком, що зменшується з кожним проходом. Може працювати швидше за $O(n^2)$ в найгіршому випадку.

Інваріанти циклу і коректність алгоритму

Інваріант циклу – логічний вираз, істинний після кожного проходу тіла циклу та перед початком його виконання, що залежить від змінних, які змінюються в тілі циклу.

Вважають, що перед першою ітерацією локальні змінні вже проініціалізовані.

Інваріанти циклу дозволяють зрозуміти, чи коректно працює алгоритм. Необхідно показати, що інваріанти циклів мають наступні три властивості.

- *Ініціалізація*. Вони справедливі перед першою ітерацією циклу.
- *Збереження*. Якщо вони істинні перед черговою ітерацією циклу, то залишаються істинні й після неї.
- *Завершення*. По завершенні циклу інваріанти дозволяють переконатися в правильності алгоритму.

Інваріанти циклу і коректність алгоритму

Властивості інваріантів циклу мають схожість з методом математичної індукції.

Але чи є різниця?

Інваріанти циклу і коректність алгоритму

Властивості інваріантів циклу мають схожість з методом математичної індукції.

Але чи є різниця?

Математична індукція дозволяє робити узагальнення для всіх значень, тоді як інваріанти циклу мають зберігатися лише перед і під час виконання циклу, тобто завідомо скінченну кількість кроків.

Обов'язкова умова для циклу – це його гарантоване завершення з бажаним результатом.

Інваріанти циклу і коректність алгоритму

«Тонкі місця» циклів:

- *перший крок*

він не має попереднього кроку, тому всі умови інваріанту мають бути коректно враховані – за необхідності або явно задані, або окремо прописані дії першого кроку;

- *момент останнього кроку*

слід переконатися, що цикл виконується потрібну кількість разів (часті помилки – зайва ітерація або втрата останньої ітерації).

Інваріанти циклу і коректність алгоритму

Покажемо коректність наведеного алгоритму сортування вставками через його інваріанти циклу.

<u>Інваріант циклу</u>: на початку кожної ітерації циклу **for** підмасив *A[1..j–1]* містить ті ж елементи, які були в ньому з самого початку, але у відсортованому порядку.

<u>Ініціалізація</u>. Покажемо справедливість інваріанта циклу перед першою ітерацією, тобто при *j=2*. Підмножина елементів *A[1..j–1]* складається з єдиного елемента *A[1]*, що зберігає початкове значення, і в цій підмножині елементи відсортовані (тривіальне твердження). Тобто інваріант циклу виконується перед першою ітерацією.

Інваріанти циклу і коректність алгоритму

Збереження. Покажемо, що інваріант циклу зберігається після кожної ітерації. Неформально кажучи, в тілі зовнішнього циклу **for** відбувається зміщення елементів *A[j–1], A[j–2], A[j–3], ...* на одну позицію вправо доти, поки не звільниться відповідне місце для елемента *A[j]*, куди він і переміщується. При формальному розгляді слід було б також сформулювати й обґрунтувати інваріант для внутрішнього циклу **while**.

*Сформулюйте інваріант для внутрішнього циклу **while**.*

Інваріанти циклу і коректність алгоритму

Завершення. Подивимося, що відбувається по завершенні роботи циклу. Під час сортування зовнішній цикл **for** завершується, коли *j* перевищує *n*, тобто при *j=n+1*. Підставивши у формулювання інваріанта циклу значення (*n+1*), отримаємо: в підмножині елементів *A[1..n]* знаходяться ті ж елементи, які були в ньому до початку роботи алгоритму, але розташовані у відсортованому порядку. Але *A[1..n]* і є сам масив *A*. Таким чином, весь масив відсортований, що й підтверджує коректність алгоритму.

Метод декомпозиції

Складне завдання розбивається на декілька простіших, подібних до вихідної задачі, але меншого об'єму; ці допоміжні задачі вирішуються рекурсивним методом, після чого отримані рішення комбінуються і отримується розв’язок початкової задачі.

Принцип «розділяй та владарюй»

- *Поділ* задачі на декілька підзадач.
- *Підкорення* – рекурсивне розв'язання цих підзадач. Коли об'єм підзадачі досить малий, вона вирішується безпосередньо.
- *Комбінування* розв'язку початкової задачі з розв'язків допоміжних задач.

Сортування злиттям

- *Поділ:* послідовність, що сортується і складається з *n* елементів, розбивається на дві менші по *n/2* елементів кожна. При цьому рекурсія закінчується, коли послідовність містить єдиний (і очевидно відсортований) елемент.
- *Підкорення:* сортування обох допоміжних підпослідовностей шляхом злиття.
- *Комбінування:* злиття двох відсортованих підпослідовностей для отримання кінцевого результату.

Сортування злиттям

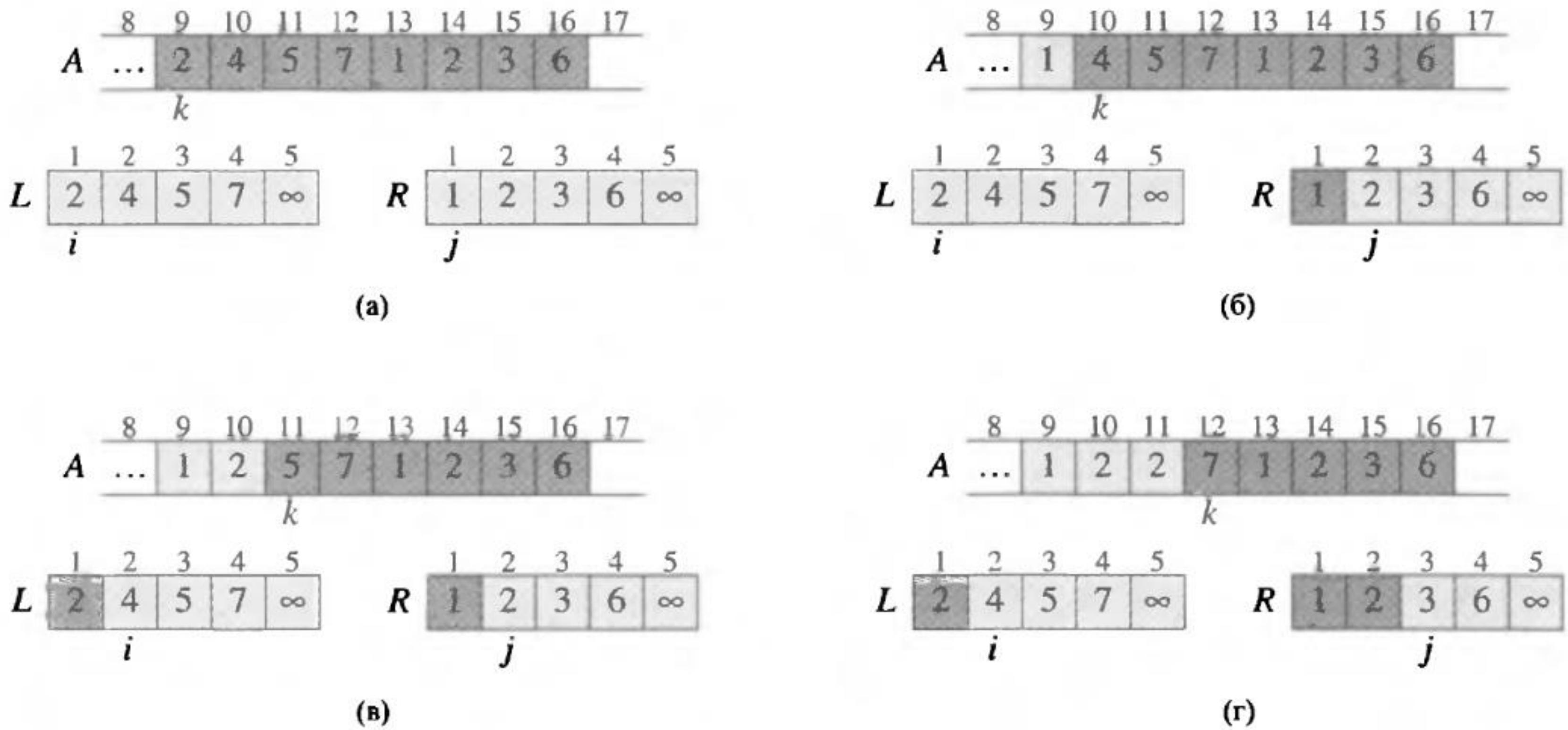
- Основна операція – об'єднання двох відсортованих послідовностей в ході комбінування (останній етап).
- Вводиться допоміжна процедура **Merge***(A,p,q,r)*, де А – масив; *p, q, r* – індекси елементів масиву ($p \le q < r$). Вважається, що елементи підмасивів *A[p..q]* та *A[*(*q+1*)*..r]* впорядковані. Процедура зливає їх в один відсортований, елементи якого замінюють поточні елементи підмасиву *A[p..r]*.
- Додаткова ідея: в кінець допоміжних масивів додається сигнальний елемент, який слугуватиме ознакою вичерпання підмасиву, його значення перевищує значення будь-якого іншого елемента. Тоді злиття виконається рівно за (*r–p+1*) крок.

Сортування злиттям

```
Merge ( A, p, q, r )
n1 <= q – p + 1
n2 <= r – q
Створюємо масиви  L[1.. n1+1]  та  R[1.. n2+1]
for  i <= 1  to  n1
   do L[i] <= A[p + i – 1]
for  j <= 1  to  n2
   do R[i] <= A[q + j]
L[n1+1] <= ∞
R[n2+1] <= ∞
i <= 1
j <= 1
for  k <= p  to  r
   do  if  L[i] ≤ R[j]
          then A[k] <= L[i]
               i <= i + 1
          else A[k] <= R[j]
               j <= j + 1
```

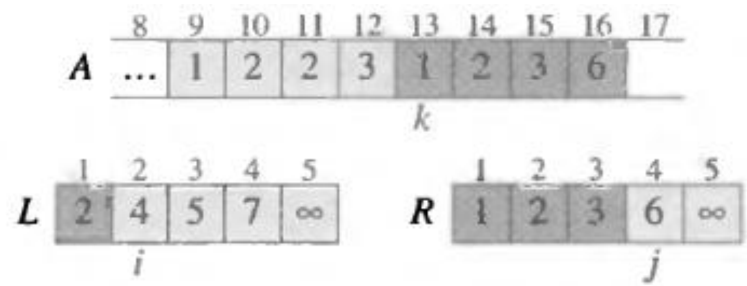
Покажіть, що процедура виконується за час Θ(*n*).

Сортування злиттям

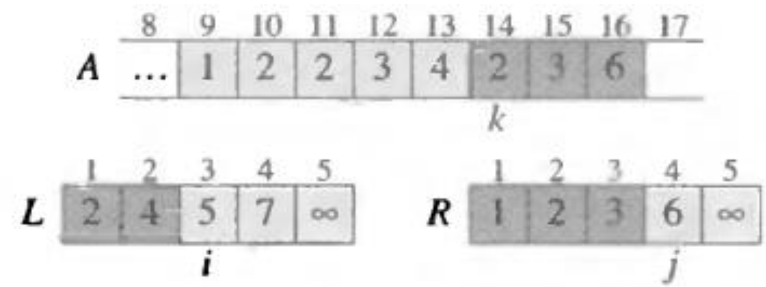


Ілюстрація Merge(A,9,12,16)
при вході <2,4,5,7,1,2,3,6>, A[9..16]]

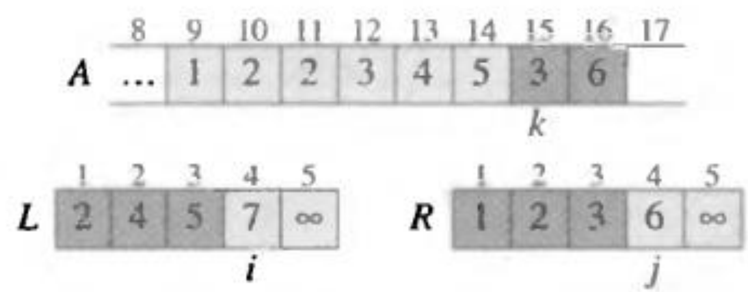
Сортування злиттям



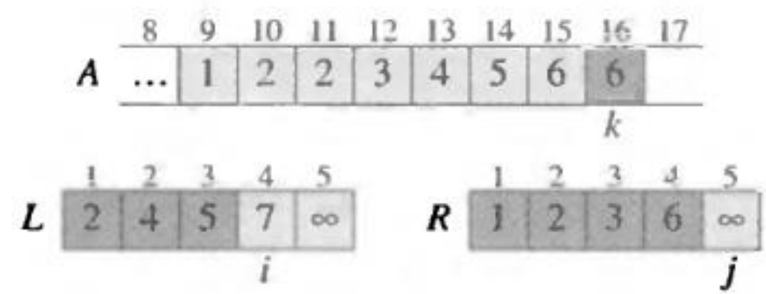
(д)



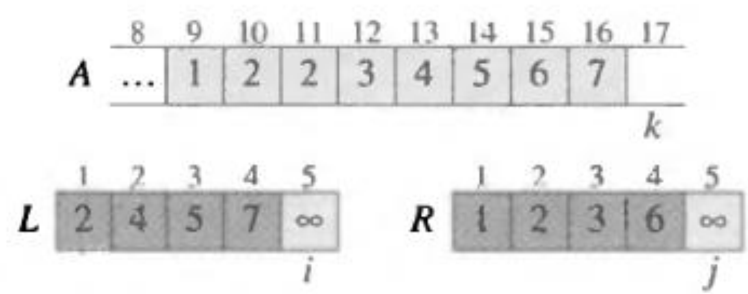
(е)



(ж)



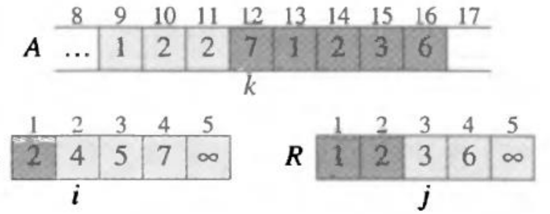
(з)



Ілюстрація Merge(A,9,12,16)

при вході <2,4,5,7,1,2,3,6>, A[9..16]]

Сортування злиттям

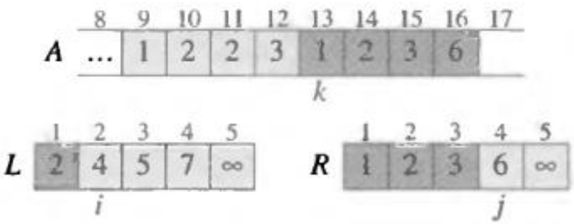


Перевіримо коректність алгоритму.

<u>Інваріант циклу</u>: на початку кожної ітерації циклу **for** (рядки 12-17) підмасив *A[p..k–1]* містить (*k–p*) найменших елементів масивів $L[1..n_1+1]$та $R[1..n_2+1]$ у відсортованому порядку. При цьому елементи *L[i]* і *R[j]* є найменшими нескопійованими елементами масивів *L* та *R*.

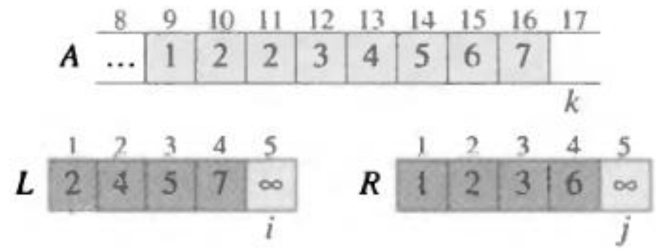
<u>Ініціалізація</u>. Перед першою ітерацією *k=p,* тому підмасив *A[p..k–1]* порожній. Він містить *k–p=0* найменших елементів масивів *L* та *R*. Оскільки *i=j=1*, то елементи *L[i]* і *R[j]* є найменшими нескопійованими елементами масивів *L* та *R*.

Сортування злиттям



<u>**Збереження**</u>. Припустимо *L[i]≤R[j].* Тоді *L[i]* – найменший нескопійований до *A* елемент. Оскільки підмасив *A[p..k–1]* містить (*k–p*) найменших елементів, після присвоєння *A[k]* значення *L[i]* (рядок 14), підмасив *A[p..k]* міститиме (*k–p+1*) найменший елемент. В результаті збільшення значення змінної *i* (рядок 15) та параметра циклу *k* інваріант циклу відновиться перед наступною ітерацією. Якщо ж *L[i]>R[j]*, то, за аналогією, виконання рядків 16-17 також відновить інваріант циклу.

Сортування злиттям



Завершення. Алгоритм завершується при $k=r+1$. Відповідно до інваріанту циклу, підмасив $A[p..k–1]$ (тобто $A[p..r]$) містить $k–p=r–p+1$ найменших елементів масивів $L[1..n_1+1]$ та $R[1..n_2+1]$ у відсортованому порядку. Всього в масивах L і R міститься елементів $n_1+n_2+2=r–p+3$. Усі вони, крім двох найбільших (а вони є сигнальними), скопійовані в масив A.

Сортування злиттям

Повний алгоритм сортування:

Merge_Sort (A, p, r)

if $p < r$

  then $q \Leftarrow \lfloor (p + r)/2 \rfloor$

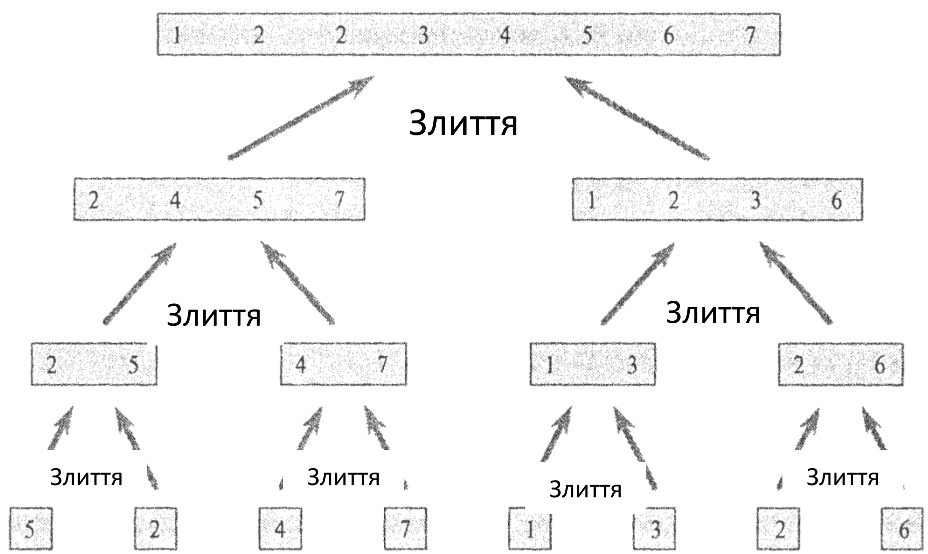
    Merge_sort (A, p, q)

    Merge_sort (A, $q + 1$, r)

    Merge (A, p, q, r)

перший виклик:
Merge_Sort($A,1,n$),
де $n = length[A]$

Відсортована послідовність



Вихідна послідовність

Ілюстрація сортування
A=<5,2,4,7,1,3,2,6>

Сортування злиттям

- Алгоритм має складність $\Theta(n \log n)$.
- Сортування допускає природне розпаралелення.
- Не має "поганих" вхідних даних, але й відсутній виграш у швидкості роботи для майже відсортованих даних порівняно з середнім випадком.
- Єдиний недолік – використання додаткової пам’яті (пропорційно до кількості вхідних елементів). Можлива реалізація без задіювання додаткової пам’яті, однак вона додає великий множник до часу роботи.

Запитання і завдання

- Емпірична дисперсія вибірки з n елементів $x_1,\ldots,x_n$ обчислюється за однією з двох формул:

$$s^2 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}, \text{ де } \bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

або

$$s^2 = \frac{\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2/n}{n-1}.$$

Знайдіть і порівняйте кількість операцій ділення, множення, додавання і віднімання (останні дві операції зазвичай об'єднуються і вважаються однією), які необхідно виконати для обчислення емпіричної дисперсії по кожній з наведених формул.

- Визначте інваріанти циклів в прикладах 1–4.

Запитання і завдання

- Визначте порядки зростання наведених сум. Використовуйте позначення Θ(*g(n)*) з найпростішою *g(n)*.

  a) $\sum_{i=0}^{n-1}(i^2+1)^2$ b) $\sum_{i=2}^{n-1}\lg i^2$

  c) $\sum_{i=1}^{n}(i+1)2^{i-1}$ d) $\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}(i+j)$

- Розглянемо задачу додавання двох двійкових цілих чисел довжиною *n* бітів кожне, що зберігаються в масивах A і B з *n* елементів. Суму цих двох чисел необхідно занести в двійковій формі до масиву C, що складається з (*n*+1) елемента. Дайте строге формулювання задачі. Наведіть псевдокод алгоритму. Доведіть його коректність за допомогою інваріанта циклу.

Запитання і завдання

- За допомогою інваріанта циклу доведіть коректність алгоритму лінійного пошуку:

```
АЛГОРИТМ  SequentialSearch ( A[0 .. n − 1], K )
// Вхідні дані:     масив чисел A[0 .. n − 1] та ключ пошуку K
// Вихідні дані:    повертається значення найбільшого елемента
//                  масиву A, що дорівнює K, або –1,
//                  якщо шуканий елемент не знайдено
i <= 0
while  i <= n  and  A[i] ≠ K  do
  i <= i + 1
if  i < n
  return i
else
  return –1
```

- Наведіть псевдокод сортування вибором. Який інваріант циклу зберігається для цього алгоритму? Чому його досить виконати для перших (*n*–*1*) елементів, а не для всіх *n* елементів? Визначте час роботи алгоритму в найкращому і найгіршому випадках і запишіть його в Θ-позначеннях.