Лекція 8. Алгоритми пошуку. Послідовний пошук елемента. Бінарний пошук елемента. Послідовний пошук рядка.

Пошук - знаходження якої-небудь конкретної інформації у великому обсязі раніше зібраних даних.

Дані діляться на записи, і кожний запис має хоча б один ключ. Ключ використовується для того, щоб відрізнити один запис від іншого.

Метою пошуку є знаходження всіх записів підходящих до заданого ключа пошуку.

Пошук елементу в масиві

Для знаходження інформації в неупорядкованому масиві потрібен послідовний пошук, що починається з першого елемента й закінчується при виявленні підходящих даних або при досягненні кінця масиву. Цей метод підходить для пошуку неупорядкованої інформації, але також можна використовувати його й на відсортованих даних. Однак якщо дані вже відсортовані, можна застосувати двійковий пошук, що знаходить дані швидше.

Послідовний пошук

Послідовний пошук дуже легко запрограмувати. Наведена нижче функція здійснює пошук у масиві символів відомої довжини, поки не буде знайдений елемент із заданим ключем:

```
// Послідовний пошук
int function LinearSearch (Array A, int L, int R, int Key);
begin
  for X = L to R do
    if A[X] = Key then
      return X
  return -1; // елемент не знайдено
end;
```

Функція повертає індекс підходящого елемента, якщо такий існує, або -1 у противному випадку.

Зрозуміло, що послідовний пошук у середньому переглядає *n/2* елементів. У найкращому разі він перевіряє тільки один елемент, а в найгіршому – *n*. Якщо інформація зберігається на диску, пошук може забирати тривалий час. Але якщо дані не впорядковані, послідовний пошук - єдино можливий метод.

Двійковий пошук

Якщо дані, у яких здійснюється пошук, відсортовані, для знаходження елемента можна застосовувати метод, що набагато перевершує попередній – *двійковий пошук*. У двійкового пошуку є й інші назви: *дихотомічний пошук*, *логарифмічний пошук*, *пошук розподілом навпіл*. У ньому застосовується метод половинного розподілу. Спочатку перевіримо середній елемент. Якщо він більше, ніж шуканий ключ, перевіримо середній елемент першої половини, у противному випадку - середній елемент другої половини. Будемо повторювати цю процедуру доти, поки шуканий елемент не буде

знайдений або поки не залишиться чергового елемента.

Наприклад, щоб знайти число 4 у масиві:

1 2 3 4 5 6 7 8 9

При двійковому пошуку спочатку перевіряється середній елемент - число 5. Оскільки воно більше, ніж 4, пошук триває в першій половині:

1 2 3 4 5

Середній елемент тепер дорівнює 3. Це менше, ніж 4, тому перша половина відкидається. Пошук триває в другій частині:

4 5

Цього разу шуканий елемент знайдений.

У двійковому пошуку кількість порівнянь у найгіршому разі дорівнює $\log_2 n$

У середньому випадку кількість порівнянь значно нижче, а в кращому - дорівнює 1. Двійковий пошук суттєво швидший за лінійний, відносно простий в реалізації і загальновживаний. Проте, в реальних програмах трапляються випадки помилкового використання лінійного пошуку в упорядкованих даних, що призводять до значного зменшення швидкодії.

Нижче наведена функція двійкового пошуку.

```
//Ітеративна версія
BinarySearch(A[0..N-1], value) {
  low = 0
  high = N - 1
  while (low <= high) {
    mid = (low + high) / 2
    if (A[mid] > value)
      high = mid - 1
    else
      if (A[mid] < value)
        low = mid + 1
      else
        return mid // знайдено
  }
  return -1 // не знайдено
}
```

Одним із варіантів реалізації алгоритму є рекурсивна функція, що отримує масив, шукане значення та початковий і кінцевий індекси елементів в масиві.

```
//Рекурсивна версія
BinarySearch(A[0..N-1], value, low, high) {
     if (high < low)
         return -1 // не знайдено
     mid = (low + high) / 2
     if (A[mid] > value)
         return BinarySearch(A, value, low, mid-1)
     else if (A[mid] < value)
         return BinarySearch(A, value, mid+1, high)
     else
         return mid // знайдено
   }
```

//Ітеративна версія з доповненнями

```
    size_t first = 0; /* Перший елемент у масиві */
    size_t last = n;    /* Елемент у масиві, НАСТУПНИЙ ЗА останнім */
                                /* Якщо переглядається непорожній фрагмент
first<last */
    size_t mid;

    if (n == 0)
    {
         /* масив порожній */
    }
    else if (a[0] > x)
    {
     /* не знайдено; якщо вам треба вставити його зі зсувом - то в позицію 0
*/
    }
    else if (a[n - 1] < x)
    {
    /* не знайдено; якщо вам треба вставити його зі зсувом - то в позицію n
*/
    }

    while (first < last)
    {
   /* УВАГА! На відміну від більш простого (first+last)/2, цей код стійкий до
переповнень. */
        mid = first + (last - first) / 2;

        if (x <= a[mid])
        {
            last = mid;
        }
        else
        {
            first = mid + 1;
        }
    }

 /* Якщо перевірка n==0 на початку відсутня - значить, тут розкоментувати!*/
    if (/*n!=0 &&*/ a[last] == x)
    {
        /* Шуканий елемент знайдено. last - шуканий індекс */
    } else
    {
        /* Шуканий елемент не знайдено. Але якщо вам раптом треба його
вставити зі зсувом, то його місце - last.    */
    }
```

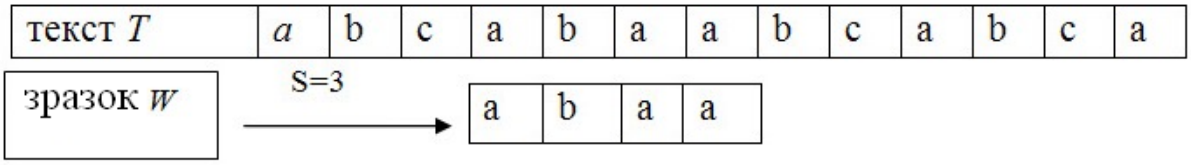
Пошук послідовності елементів в масиві

Одне з найпростіших завдань пошуку інформації – **пошук точно заданого підрядка в рядку**. Проте, це завдання надзвичайно важливе – воно застосовується в текстових редакторах, СУБД, пошукових машинах тощо.

Пошук рядка формально визначається в такий спосіб. Нехай заданий масив T з N елементів і масив W з M елементів, причому $0<M\leq N$. Пошук рядка виявляє перше входження W у T, результатом будемо вважати індекс i, що вказує на перший з початку рядка (з початку масиву T) збіг зі зразком

(словом).

Приклад. Потрібно знайти всі входження зразка W = abaa у текст T=abcabaabcabca.



Зразок входить у текст тільки один раз, зі зсув S=3, індекс i=4.

Алгоритм прямого (послідовного) пошуку

Ідея алгоритму:

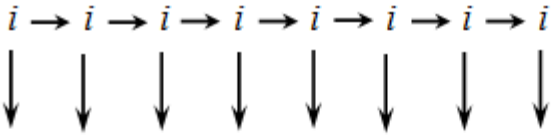
1) I=1,
2) порівняти I-й символ масиву T з першим символом масиву W,
3) збіг → зрівняти другі символи й так далі,
4) розбіжність → I:=I+1 і перехід на пункт 2,

Умова закінчення алгоритму:

1) підряд M порівнянь вдалі,
2) I+M>N, тобто слово не знайдене.

```
Function Search (S: String; X: String; var Place: Byte): Boolean;
{ Функція повертає результат пошуку у рядку S }
{ підрядка X. Place – місце першого входження }
var Res: Boolean; i : Integer;
  Begin
   Res:=FALSE;
   i:=1;
    While (i<=Length(S)-Length(X)+1) And Not(Res) do
     If Copy(S,i,Length(X))=X then
       begin
  Res:=TRUE;
  Place:=i
      end
    else i:=i+1;
       Search:=Res
  End;
```

Приклад. Потрібно знайти підрядок W = abcabd у тексті T=abcabcaabcabd.



рядок	A	B	C	A	B	C	A	A	B	C	A	B	D
підрядок	**A**	**B**	**C**	**A**	**B**	**D**							
		A	B	C	A	B	D						
			A	B	C	A	B	D					
				A	**B**	**C**	**A**	**B**	D				
					A	B	C	A	B	D			
						A	B	C	A	B	D		
							A	**B**	C	A	B	D	
								A	**B**	**C**	**A**	**B**	**D**

Зразок входить у текст тільки один раз, зі зсувом S=7, індекс i=8.

Складність алгоритму:

Гірший випадок. Нехай масив T→{AAA....AAAB}, довжина |T|=N, зразок W→{A….AB}, довжина |W|=M. Очевидно, що для виявлення збігу наприкінці рядка буде потрібно зробити порядку N*M порівнянь, тобто O(N*M).

Недоліки алгоритму:

1) висока складність – O(N*M), у найгіршому випадку – Θ((N-M+1)*M);

2) після розбіжності перегляд завжди починається з першого символу зразка й тому може включати символи T, які раніше вже проглядалися (якщо рядок читається із вторинної пам'яті, то такі повернення займають багато часу);

3) інформація про текст T, одержувана при перевірці даного зсуву S, ніяк не використовується при перевірці наступних зсувів.