Лекция 6

Алгоритми за търсене и сортиране

Нели Василева

Съдържание

- Въведение
- Алгоритми за търсене
  - Последователно (линейно) търсене
  - Бинарно търсене
- Алгоритми за сортиране
  - Bubble sort
  - Insertion sort
  - Selection sort
  - Counting sort
- Рекурсивни методи за сортиране - Merge sort

Въведение

- Търсене
  - Зададена е стойност (или обект) . Проверява се за съвпадение с елемент на масив от същия тип както зададената стойност (или обект)
  - Търсене на едно съвпадение- извежда се позицията на съвпадението в масива
  - Търсене на всички съвпадения- извежда се масив с позициита на съвпадения в масива
- Сортиране
  - Подреждане на елементите на масив по даден критерий за наредба в съответствие на зададени ключ(ове) за сортиране

Алгоритми за търсене

- Примери:
  - Линейно търсене
  - Бинарно търсене
  - Рекурсивно линейно търсене
  - Рекурсивно бинарно търсене
  - Линейно търсене в списък
  - Търсене в двоично дърво
  - Метод binarySearch в клас Collections

Linear Search

- Последователно (линейно) търсене
  - Сравняват се последователно всички елементи със зададения шаблон (search key) за търсене
- Алгоритъм
  - Всеки елемент на масива се сравнява със зададения шаблон, докато се намери съвпадение- извежда се позицията на съвпадение
  - Ако се стигне до края на масива без да е намерено съвпадение - извежда се сигнал за липса на съвпадение

Линейно търсене в int масив

```
public int linearSearch( int searchKey, int[] data ) {
   for ( int index = 0; index < data.length; index++ )
            if ( data[ index ] == searchKey )
                     return index; // return index of integer

   return -1; // integer was not found
}
```

Двоично (бинарно) търсене

- По- бързо в сравнение с последователното търсене
- Изисква елементите на масива да са предварително сортирани
- Започва със сравнение на елемента от масива, който е в средата
  - Ако средния елемент на масива съвпада с шаблона- то извеждаме позицията в масива, където е настъпило съвпадението
  - В протевен случай определяме дали шаблона е по- малък или по- голям от средния елемент и продължаваме да търсим по същия начин съответно в половината с по- малките или по-големите елементи от средния
- На всяка итерация се елеминира половина от оставащите елементи за сравнение

Двоично търсене в int масив

```
public int binarySearch( int searchElement, int[] data ) {
    int low = 0; // low end of the search area
    int high = data.length - 1; // high end of the search area
    int middle = ( low + high + 1 ) / 2; // middle element
    int location = -1; // return value; -1 if not found

     do {
            // if the element is found at the middle
            if ( searchElement == data[ middle ] )
                    location = middle; // location is the current middle
            // middle element is too high
            else if ( searchElement < data[ middle ] )
                    high = middle - 1; // eliminate the higher half
            else // middle element is too low
                    low = middle + 1; // eliminate the lower half

            middle = ( low + high + 1 ) / 2; // recalculate the middle

    } while ( ( low <= high ) && ( location == -1 ) );

    return location; // return location of search key
}
```

Алгоритми за сортиране

- Примери:
  - Selection sort
  - Insertion sort
  - Merge sort
  - Bubble sort
  - Quicksort
  - Методът sort в клас Collections

Алгоритми за сортиране

- Сортиране на данни - подреждане на данните в определен ред по отношение на избран ключ за сортиране (прост или съставен)
  - Банкови сметки се сортират по номера на сметката (прост ключ за сортиране)
  - Телефонните номера в указателя се сортират по име и фамилията на потребителя (съставен ключ за сортиране)
- Краен резултат (независимо от алгоритъма)- масив, чиито елементи са подредени в зададен ред (възходящ/низходящ)
- Изборът на алгоритъма има отношение към времето за изпълнение- ефективност на алгоритъма

Bubble sort

- **Bubble sort "*метод на мехурчето*"**
  - По- малките стойности "изплуват" в началото на масива
  - По- големите стойности "потъват" в дъното на масива
  - Използват се вложени цикли за изпълнение на няколко паса по елементите на масива. Всеки пас сравнява последователно всички двойки от елементи
    - Двойките елементи не се разменят ако са в изискваната наредба (нарастваща или намаляваща големина) или равни
    - Двойките елементи се разменят ако не са в изискваната наредба (нарастваща или намаляваща големина)

Bubble sort в int масив

```
public void bubbleSort( int array[] ) {

    // loop to control number of passes
    for ( int pass = 1; pass < array.length; pass++ ) {

        // loop to control number of comparisons
        for ( int element = 0; element < array.length – 1; element++ ) {
            // compare side-by-side elements and swap them if
            // first element is greater than second element
            if ( array[ element ] > array[ element + 1 ] ) {
                int hold; // temporary holding area for swap

                hold = array[ element];
                array[ element] = array[element + 1 ];
                array[element + 1 ] = hold;
            }
        }
    }
}
```

Selection sort

- Прост, лесен за реализация, но не ефективен (бавен) алгоритъм. Има итеративна и рекурсивна версия.
- Алгоритъм:
  1. При първата итерация се избира най- малкия елемент в масива и се разменя с първия елемент на масива
  2. Всяка следваща итерация избира най- малкия от останалите елементи на масива и го разменя със следващия елемент от началото на масива
  3. След *i итерации, най-малките i елемента са* подредени във възходящ ред в първите *i елемента на* масива

Selection sort в int масив

```
public void selectionSort(int[] array) {
    int smallest; // index of smallest element

    for ( int i = 0; i < array.length - 1; i++ ) {
            smallest = i; // first index of remaining array
            // loop to find index of smallest element
            for ( int index = i + 1; index < data.length; index++ ) {
                    if ( data[ index ] < data[ smallest ] )
                            smallest = index;
            }
            // swap smallest element into position
            int hold; // temporary holding area for swap

            hold = array[ i];
            array[ i] = array[smallest];
            array[smallest] = hold;
    }
}
```

Ефективност

- Измерване на ефективността- бързодействие на алгоритъм
- Означение за порядъка (горната граница) от операции O (n), когато размерността на входните данни n клони към безкрайност
  - Изразява най-лошото (дълго) възможно време за изпълнение
  - Това време е в зависимост от броя елементи за сортиране
  - Пример за постоянно време за изпълнение. Означава се
    - *O(1) – времето за изпълнение е ограничено от константа*
    - Не е зависимо от броя елементи за сортиране
  - Пример за линейна зависимост на броя операции от броя на елементите за сортиране
    - *O(n) – времето за изпълнение е ограничено линейна функция*
    - Нараства пропорционално на броя на елементите в масива

Ефективност

- Означението O изразява зависимостта на времето за изпълнение от броя на извършваните операции (за по-просто, считаме всяка операция с единици време за изпълнение)
- Да разгледаме алгоритъм с n^2 сравнения.
  - Така при 4 елемента се изискват 16 сравнения, при 16 елемента - 64 сравнения и пр.
- Да разгледаме алгоритъм с *n^2 / 2 сравнения.*
  - При 4 елемента се изискват 8 сравнения, при 16 елемента – 32 сравнения и пр.
- Проверяваме, че нарастването на сравненията (операциите) нараства като квадрата на нарастването на броя на елементите.
- Така, означението *“голямо” O константата в броя на сравненията не* е съществена и в двата алгоритъма имаме

Ефективност на Selection sort

- *Най- лошият случай при сортиране във* възходящ ред е, когато масивът е нареден в низходящ ред. В този случай:
  - Външният for цикъл ще изпълни *n – 1* итерации
  - Вътрешният for цикъл ще изпълни върху оставащите елементи
  - Общият брой операции е
  - Свежда се до *O(n2)*

Insertion sort

- Сортиране чрез вмъкване - също така прост и лесен за изпълнение, но неефективен алгоритъм за сортиране (огледален образ на Selection sort)
- Алгоритъм за сортиране във възходящ ред
  1. На първата итерация (*pass) се сравнява втория елемент и се "вмъква" пред първия, ако е по- малък от него. Така първите* два елемента са наредени по големина
  2. На всяка следваща итерация се избира следващия елемент от масива и той се "*вмъква" между първите наредени* елементи така че да се запази наредбата на тези елементи
  3. След *i итерации, първите i елемента на масива са* сортирани по големина
- "*Използваната" част от масива е по размер същата,* както "*неизползваната"*
- Следователно, алгоритъмът може да се прилага към един и същ масив, без да се създава копие на масива, в което да се прилага алгоритъма

Insertion sort в int масив

```
public void sort(int [] data) {
    int insert; // temporary variable to hold element to insert

    // loop over data.length - 1 elements
    for ( int next = 1; next < data.length; next++ ) {
            // store value in current element
            insert = data[ next ];

            // initialize location to place element
            int moveItem = next;
            // search for place to put current element
            while ( moveItem > 0 && data[ moveItem - 1 ] > insert ) {
                    // shift element right one slot
                    data[ moveItem ] = data[ moveItem - 1 ];
                    moveItem--;
            }
            data[ moveItem ] = insert; // place inserted element
    }
}
```

Ефективност на Insertion sort

- Същите критерии за най-лошо време като при Selection sort (*масивът е подреден в обратен ред на изисквания).*
  - Външният цкъл се изпълнява за *n – 1 елемента*
  - В най- лошия случай вътрешният цикъл се изпълнява 1,2 ,3, ..., n – 1 пъти
  - Води до порядък *O(n2)*

Counting sort

- Също така прост и лесен за изпълнение, но също и ефективен алгоритъм за сортиране (ефективност ***O(n)***)
- Не използва сравнения (>,< , >= , <= , = =) за определяне на наредбата на елементите.
- За сравнение, алгоритмите използващи сравнение постигат като ефективност най-много ***n log(n)***.
- Забележка : Бързодействието при този алгоритъм се постига за сметка на използване на допълнителна памет

Counting sort

- Сортиране с преброяване (Counting sort) сортира *n положителни числа от интервала [0, k]*
- Основна идея

  Да се определи за всеки елемент *x броя елементи по- малък* от *x. Тази информация се използва за поставяне на x в* неговото точно положение на сортирания масив. Например, ако имаме *17 елемента по- малки от x , то x трябва да се* намира на *18-то място в изходния сортиран масив. При* наличие на повтарящи се елементи в изходния списък с елементи е необходима малка модификация на тази схема.

Алгоритъм

- Алгоритъм за сортиране във възходящ ред

```
CountingSort(A, B, k)
1. for i = 0 to k // инициализираме масива C
2.      do C[i] = 0
3. for j = 0 to length[A]
4.      do C[A[j]] = C[A[j]] + 1
5. // C[i] е броят елементи равен на i
6. for i = 1 to k
7.      do C[i] = C[i] + C[i - 1]
8. // C[i] е броят елементи по- малки или равни на i
9. for j = length[A] downTo 1
10.     do B[C[A[j]] ] = A[j]
11.             C[A[j]] = C[A[j]] - 1
```

Counting sort

- Важно свойство

  Този алгоритъм е представител на алгоритми за сортиране, наричани *устойчиви*.

- Дефиниция

  Един алгоритъм за сортиране се нарича устойчив, ако повтарящите се стойности се извеждат в сортирания масив в същата последователност, в която са били в изходния масив за сортиране

Рекурсия

- Основни елементи на рекурсията

  Базов(и) (граничен/ни) случай/и

  - Рекурсивен метод, който дава решение само за най-простия случай—the base case
  - Ако рекурсивният метод се извика с базовия случай, методът връща резултат (не се извиква рекурсивно)

  Пример:

  - При обхождане на дърво, граничният случай е коренът на дървото за обхождане да е null

Рекурсивна стъпка

- При извикване на рекурсивния метод за решаване на случай, различен от базовия, задачата се разделя на две части— част, която методът дефинира как да изпълни и част, която методът не дефинира как да изпълни (наричана рекурсивна стъпка, извикване)
- Рекурсивна стъпка, извикване - изисквания
  - Трябва да решава зададения първоначален проблем, но в по- прост вид или умален размер
  - Методът извиква себе си за решаване на същия проблем, но с по- малка размерност
  - Обикновено, включва return statement
- Пример за рекурсивна стъпка
  - Обхождането на дърво се свежда до обхождане на лявото и дясното под- дървета

Прост пример за рекурсия

- *N факториел е произведението*

  n! = n (n – 1) (n – 2) … 1

  като дефинираме

  1! = 1

  0! = 1.
- Може да се реализира рекурсивно или итеративно
- За рекурсивното решение забелязваме, че:

  n! = n (n – 1)!
- Тук
  - *граничният случай е n <=1*
  - *Рекурсивната стъпка е (n – 1)!*

Рекурсия и изпълним стек

- stack структурирана памет съхранява поредните извиквания на методи и локалните данни, зададени като аргументи на метода
- Също както и нерекурсивните методи, адресите на рекурсивните методи се записват отгоре на стека с извиквания на методи
- Когато рекурсивен метод изпълни return, техните записи за действие (activation record) и тези на предхождащите ги рекурсивни извиквания се “изхвърлят” от стека
- Текущо изпълняваният метод е този метод, чиито запис за действие е на върха на стека

Merge sort в масиви

- По- ефективен метод за сортиране, но и по сложен
- Разделя дадения масив на два по- малки масива с приблизително равен брой елементи, сортира всеки от тези по- малки масива, накрая смесва двата сортирани подмасива в един масив
- Рекурсивен модел на Merge sort
  - Граничният случай е едно елементен масив, който е сортиран
  - Рекурсивната стъпка включва (1)разделяне на масива на две части, (2)сортиране на всяка част и (3) смесването на двете части

Merge sort в int масив

```
private void sortArray( int low, int high ) {
   // test base case; size of array equals 1
   if ( ( high - low ) >= 1 )  {  // if not base case
         int middle1 = ( low + high ) / 2; // calculate middle of array
         int middle2 = middle1 + 1; // calculate next element over

         // split array in half; sort each half (recursive calls)
         sortArray( low, middle1 ); // first half of array
         sortArray( middle2, high ); // second half of array

         // merge two sorted arrays after split calls return
         merge ( low, middle1, middle2, high );
   }
}
```

```
private void merge(( int[] data, int left, int middle1, int middle2, int right ) {
    int leftIndex = left; // index into left subarray
    int rightIndex = middle2; // index into right subarray
    int combinedIndex = left; // index into temporary working array
    int[] combined = new int[ data.length ]; // working array

    // merge arrays until reaching end of either
    while ( leftIndex <= middle1 && rightIndex <= right ) {
            // place smaller of two current elements into result
            // and move to next space in arrays
            if ( data[ leftIndex ] <= data[ rightIndex ] )
                    combined[ combinedIndex++ ] = data[ leftIndex++ ];
            else
                    combined[ combinedIndex++ ] = data[ rightIndex++ ];
    }
    // if left array is empty
    if ( leftIndex == middle2 )
            // copy in rest of right array
            while ( rightIndex <= right )
                    combined[ combinedIndex++ ] = data[ rightIndex++ ];
    else // right array is empty
            // copy in rest of left array
            while ( leftIndex <= middle1 )
                    combined[ combinedIndex++ ] = data[ leftIndex++ ];
    // copy values back into original array
    for ( int i = left; i <= right; i++ )
            data[ i ] = combined[ i ];
}
```

Ефективност на Merge sort

- Много по-ефективен от сортиране с избор и вмъкване
- Последният merge изисква *n – 1 сравнения за* смесване на сортираните части от масива
- На всяко по- долно ниво има два пъти повече извиквания на merge, и всяко извикване работи с половината от елементите на предишното нива - ***O(n)*** общо сравнения
- Общо нивата на разделяне на половинки е ***O(log n)***
- Общата ефективност е ***O(n log (n))***

Ефективност на алгоритмите

Алгоритъм	Ефективност
За търсене:	
Linear search	**O(n)**
Binary search	**O(log(n))**
Recursive linear search	**O(n)**
Recursive binary search	**O(log(n))**
За сортиране:	
Selection sort	**$O(n^2)$**
Insertion sort	**$O(n^2)$**
Merge sort	**O(n log(n))**
Bubble sort	**$O(n^2)$**
Counting sort	**O(n)**

Въпроси?