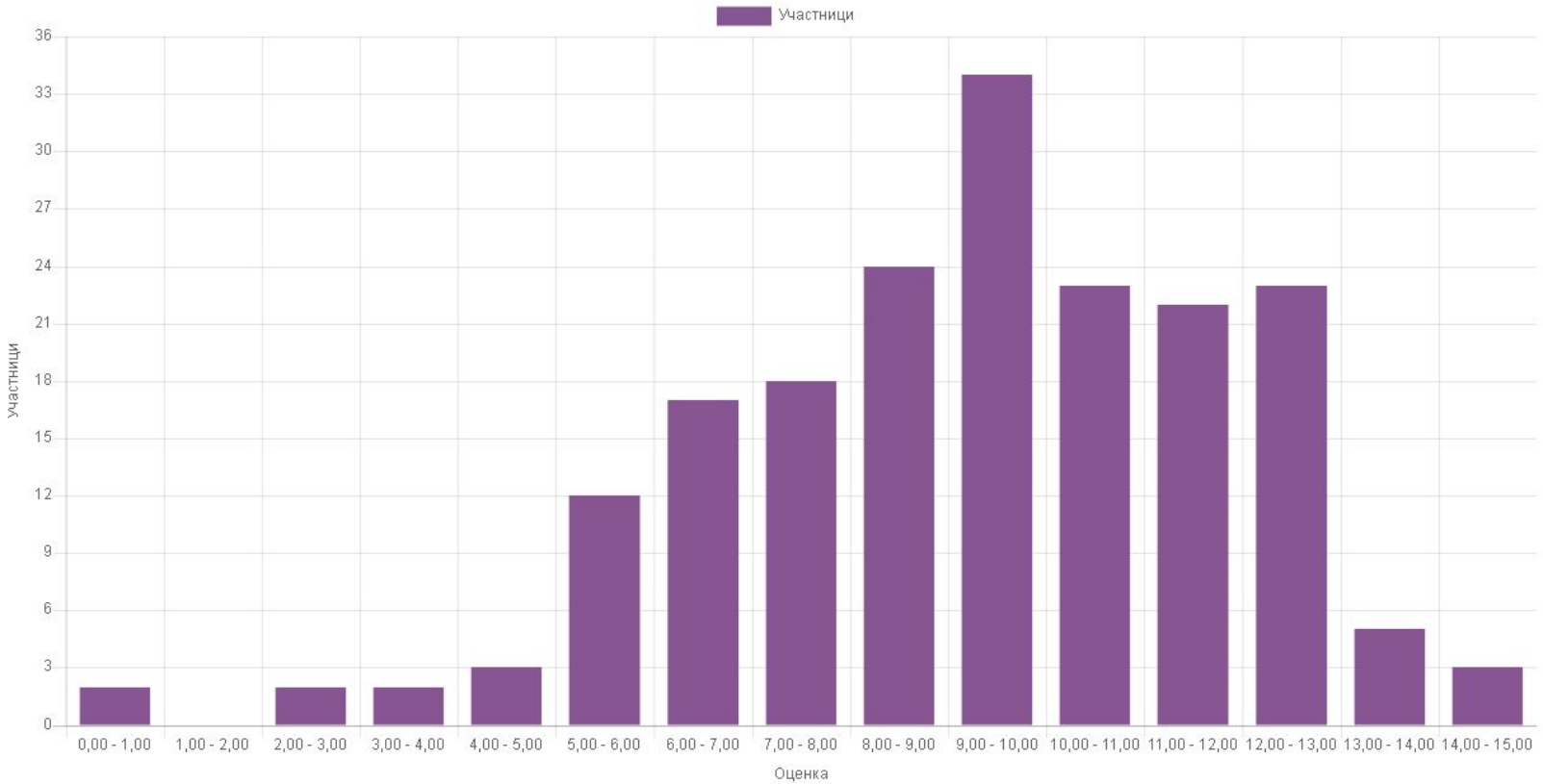
Алгоритми за търсене

Лекция 3 по СДА, Софтуерно Инженерство
Зимен семестър 2018-2019г
Милен Чечев

План за днешната лекция

- Резултати от контролно 1
- Алгоритми за търсене
- Повторение на бързите алгоритми за сортиране с допълнение

Резултати от контролно 1



Решение на задачата от контролно 1

Проблем: Да определим дали в масив имаме 3 числа (a,b,c) такива че a+b=c

Стандартно решение:

```
for(int i = 0 ; i < arr.length;i++)
        for(int j = 0; j < arr.length; j++)
                for(int k = 0 ; k < arr.length; k++)
                        if(arr[i]+arr[j] = arr[k] && i!=j && j!=k && k!=i)
                                return “true”
return “false”
```

Бързо решение

```
// броим и проверяваме
Int[] count = int[1000000]
for(int i = 0 ; i < arr.length ; i++){
	for( int j = i+1; j < arr.length; j++){
		count[arr[i]+arr[j]] = count[arr[i]+arr[j]] + 1;
	}
}
for(int i = 0 ; i < arr.length; i++){
	if(count[arr[i]] >0){
		return “true”
	}
}
return false;
```

Обратна връзка за седмица 2

42 попълнени форми:

- Не се чува. Да се пробва с микрофон.
- Да може да се ползва лист по време на контролното
- Доста шумно по-време на контролното. Проблеми с интернета.
- Твърде много време се отделя по-организационни въпроси.
- Не се отдели достатъчно време за по-сложните алгоритми за сортиране
- Твърде много хора на упражненията на 1 и 2 група.

Алгоритми за търсене

Задачи за търсене

Задача 1: Да се намери дали числото X се среща в масив arr

Задача 2 : Да се намери колко пъти число X се среща в масив.

Задача 3 : Да се намери на кои позиции, се среща числото X в масив.

Линейно търсене (Linear search)

```
boolean linear_search(int[]arr, int x){
    for(int i = 0 ; i < arr.length; i++){
        if(arr[i]==x){
            return true;
        }
    }
    return false;
}
```

Двоично търсене(Binary search)

При несортиран масив няма как да търсим със сложност по-малка от O(n), в случай, че ще извършваме много търсения върху един масив то за да ускорим търсенето можем първоначално да го сортираме (за O(nlog(n))) и после да търсим със по-ниска сложност O(log(n))

Двоично търсене(Binary search)

```
Boolean binary_search(int[] sorted, int x, int start, int end){

        if(start>end) return false;

        if(arr[(end+start)/2] == x) return true;
        if(arr[(end+start)/2] > x) return binary_search(sorted,x,start,middle-1);
        if(arr[(end+start)/2] < x) return binary_search(sorted,x,middle+1,end);
}
```

Тристранно търсене (Ternary Search)

```
boolean ternarySearch(arr, x, left, right){

        if(right < left) return false;

        mid1 = (2*left + right)/3;
        mid2 = (left + 2*right)/3;
        if(arr[mid1] == x || arr[mid2]==x) return true;

        if(arr[mid1]) > x) return ternary_search(arr, x, left,mid1-1);
        if(arr[mid2]) > x) return ternary_search(arr, x, mid1+1,mid2-1);

        return ternary_search(arr,x, mid2+1,right)
}
```

Задача за междучасието

Имаме 2 пластмасови топки и 100 етажна сграда. Искаме да разберем каква е устойчивостта на материала на топките като знаем, че материала при изпускане или се счупва или не понася никакви поражения. С колко най-малко опита може със сигурност да се каже до кой етаж е издръжливостта на такава пластмасова топка?

Търсене със скоци(Jump Search)

```
boolean jumpSearch(int[] arr, int x) {

        int step = (int)Math.floor(Math.sqrt(arr.length));

        int prev = 0; int next = prev + step;
        while (arr[Math.min(next, n)-1] < x) {
                prev = next;
                next += step;
                if (prev >= n)
                return false;
        }
 // continue on next slide
```

Jump Search (продължение)

```
        while (arr[prev] < x) {
                prev++;
                if (prev == Math.min(step, n))
                return false;
        }

        if (arr[prev] == x)
                return true;

        return false;
}
```

Минимум, Максимум, Средна точка

Задача: Намерете най-големият(най-малкият) елемент на масив

Задача: Намерете к-тия най-голям елемент на масив

Задача: Намерете средният елемент на масив.

К-тия най-голям елемент на масив

Подходи:

O(nlog(n)) - сортираме масива и взимаме съответният елемент

Как да се справим по-бързо?

1. Не е необходимо да сортираме масива.
2. Трябва само да знаем кои са елементите по-големи и по-малки от к-тия елемент
3. Може да ползваме подход подобен на quicksort

К-тия най-голям елемент на масив (реализация)

```
randomized_select(arr, left, right, k){
	if(left==right) return arr[left];
	q = randomized_partition(arr,left,right)
	i = q-left+1

	if(i==k) return arr[q]

	if(i < k) return randomized_select(arr,left, q-1,k)

	return randomized_select(arr,q+1,right,k)
}
```

Алгоритми за сортиране

(преговор с допълнение)

Сортиране чрез сливане(merge sort)

Основна идея: Ако имаме два сортирани масива то със линейна сложност може да ги влеем в един масив. Тогава ако разделим масива който искаме да сортираме на по-малки масиви и на всяка стъпка сливаме два по-малки масива в един голям то за log(N) стъпки ще слеем всички масиви до един масив, като всяка от стъпките е била линейна.

https://visualgo.net/en/sorting

Сортиране със сливане реализация

```
void mergesort(int arr[], int l, int r)
    {
        if (l < r) //гранично условие на рекурсията
        {
            mergesort(arr, l, (l+r)/2);
            mergesort(arr , (l+r)/2+1, r);
            merge(arr, l, m, r); //функция която слива два масива
        }
    }
```

Сортиране със сливане реализация(2)

```
void merge(int arr[], int start, int middle, int end){
    // Създаваме масивите arr1 и arr2, които съдържат частите, които ще копитаме. По този начин си освобождаваме основният масив за презаписване.

    i = 0, j = 0, k = start;
    while( i < arr1.length; j < arr2.length){
        // по малкото число от arr[i] и arr2[j] го записваме в  arr[k]
        // увеличаваме брояча на масива от който копирахме
        K++; // увеличаваме брояча за основният масив
    }
    // допълваме с всички останали необходени елементи от arr1 и arr2
}
```

Merge sort complexity

O(nlog(n)) - изчислителна сложност в най-лошият случай

O(n) - сложност по памет

Бързо сортиране (quick sort)

Основна идея: Ако вземем едно произволно число от масива, то с линейна сложност можем да прехвърлим всички по-малки числа от масива да са в ляво на числото, а всички по-големи в дясно.

При бързото сортиране избираме число от масива прехвърляме по-малите отляво, по-големите от дясно и след това изпълняваме същата процедура за лявата и дясната половина.

https://visualgo.net/en/sorting

Бързо сортиране реализация

```
void sort(int arr[], int low, int high)
    {
        if (low < high)
        {
            int pi = partition(arr, low, high);

            sort(arr, low, pi-1);
            sort(arr, pi+1, high);
        }
    }
```

Бързо сортиране реализация(2)

```
int partition(int arr[], int low, int high){
        int pivot = arr[high];
        int border = low; // index of smaller element
        for (int j=low; j<high; j++){
            if (arr[j] <= pivot){
                swap(arr, border, j)
                border++;
            }
        }

        swap(arr, i, high)
        return i;
    }
```

Сложност

Сложност в средният случай O(n.log(n)) ,но….

Сложност в най-лошият случай O(N^2) - когато масива е сортиран наобратно!

Рандомизирано Бързо Сортиране

Справя се с проблема, че точно определена редица прави сложността O (n^2), като използва произволно избиране на елемент за разделяне.

```
int random_partition(int arr[], int low, int high){
	swap(arr, high, random(arr.length));
	partition(arr,low, high);
}
```

Сортиране с броене (Counting sort)

Можем ли да сортираме със сложност по-малка от O(n*log(n))?

Отговор: Да, но с добавяне на допълнителни ограничения.

При сортирането с броене сложността се определя от броя на различните елементи, които може да има в масива.

Merge sort or quick sort

Merge sort изисква O(n) допълнителна памет, докато quicksort не изисква допълнителна памет.

Merge sort е със сложност в най-лошият случай O(nlog(n)), Quicksort -O(n^2)

Quicksort - по-бърз от mergesort за малки масиви, но mergesort е по-добър за големи масиви.

Сортиране с броене

Основна идея: Понеже имаме ограничен брой различни стойности в масива то може да преброим по колко пъти се среща всяка една от тези стойности(с едно обхождане на масива) и след това със второ обхождане да наредим стойностите по техният ред.

Стабилност на сортирането - ако имаме два елемента които са равни в първоначалният масив, то във финалният те се срещат във същият ред като първоначалният масив.

https://visualgo.net/en/sorting

```
void conting_sort(char arr[]) {
        char arr_copy[] = new char[arr.length];
        for (int i = 0; i<arr.length; ++i) {
            arr_copy[i] = arr[i];
        }
        int count[] = new int[256];
        for (int i=0; i<n; ++i) {
            count[arr[i]] = count[arr[i]]+1;
        }
        for (int i=1; i<=255; ++i) {
            count[i] += count[i-1];
        }
        // To make it stable we are operating in reverse order.
        for (int i = n-1; i>=0; i--) {
            arr[count[arr_copy[i]]-1] = [arr_copy[i];
            count[arr_copy[i]] = count[arr_copy[i]] - 1;
        }
}
```

Сложност на сортиране с броене

O(n+k)

Radix Sort (Допълнителен материал)

Основна идея - да използваме подход подобен на сортиране с броене, но да може да го прехвърлим и за големи числа.

При radix sort вместо да броим цели числа ще броим само цифри, като ще сортираме масива подред за всички позиции на цифри(единици,десетици, стотици, хиляди, десетохиляди, и т.н). Понеже сортирането с броене запазва подредбата веднъж сортирани числата по последна цифра, те си остават сортирани и при последващо сортиране по десетици и т.н. до последното сортиране.

https://visualgo.net/en/sorting

Обобщение и следващи стъпки

- Разгледахме основните алгоритми за търсене и сортиране
-
- От другият път започваизучаването на структури от данни
- Другият път ще се проведе контролно 2 върху темите сортиране и търсене
-