КАТЕДРА: КОМПЮТЪРНИ СИСТЕМИ И ТЕХНОЛОГИИ
ДИСЦИПЛИНА: АЛГОРИТМИ СТРУКТУРИ ОТ ДАННИ
ДИСЦИПЛИНА: СИНТЕЗ И АНАЛИЗ НА АЛГОРИТМИ

ЛАБОРАТОРНО УПРАЖНЕНИЕ № 5

ТЕМА:Алгоритми за търсене. Двоично търсене

ЦЕЛ:

Целта на упражнението е студентите да се запознаят с методите за търсене, като се запознаят по задълбочено върху двоичното търсене. След упражнението студентите би следвало да могат да прилагат подходящия алгоритъм върху различни структури от данни.

I. ТЕОРЕТИЧНА ЧАСТ

Методът "Двоично търсене" се нарича "Binary Search" на английски, като много често може да го срещнете като "Байнъри". То представлява ефективен алгоритъм за намиране на елемент от подреден списък с елементи. Работи, като неколкократно разделя наполовина тази част от списъка, която може да съдържа търсения елемент, докато възможните позиции се сведат до една.

Един от най-често срещаните начини за използване на двоично търсене е намиране на елемент в масив. Съществуват множество линейни алгоритми чрез който може да се извърши такова търсене, но при всички тях в най – лошия случай ще са необходими многократно повече стъпки за да се намери търсения елемент. При двоичното търсене броя на тези стъпки е намален до минимум.

За да се приложи този алгоритъм обаче е необходимо елементите в съответния списък да бъдат предварително сортирани, т.е. предварително трябва да се приложи някой от алгоритмите за сортиране.

Идеята на алгоритъма е след всяка итерация интервалът на търсене да се разделя на половина. Това се осъществява като винаги първо се проверява стойността в средата на интервала, благодарение на което се отстраняват половината елементите още след първата итерация. Алгоритъма проверява средната стойност от списъка и ако тя е по голяма от търсената се продължава с елементите от ляво на търсената стойност, като отново се прилага същия алгоритъм, ако средната стойност е по – малка от търсената то се продължава с елементите от дясно на нея. Описанието на алгоритъма стъпка по стъпка е следното:

1) Задават се минималния и максималния брой елементи в търсения интервал. Примерно min=1 и max = n;
2) Намира се средната стойност на min и max и се закръгля в посока на долу до най близкото цяло число;
3) Ако числото е познато. Алгоритъма приключва.
4) Ако отговорът е че търсеното число е по малко от предположеното, то се задава на max стойност, която е с едно по малка от предположеното число;
5) Ако отговорът е че търсеното число е по голямо от предположеното, то се задава за min стойност, която е с едно по голяма от предположеното число;
6) Връщане към стъпка 2.

Тъй като се търси нещо в някакъв интервал е хубаво, е хубаво да си представим интервала по някакъв начин. В програмните езици, най – често се търси нещо в масив,

като интервалът се представя чрез най - левия и най десния индекс на масива (примерно наречени left и right). При всяко изпълнение на 2-ра стъпка се изчислява една нова стойност (средата на интервала), записана в променлива примерно *mid*. За да се вземе средата на интервала по принцип се взема левия му край и се добавя половината от масива. В код това изглежда по следния начин:

mid=left+(right-left)/2

По – често обаче се използва по краткия вариант *mid=(left+right)/2*. Ако обаче се използва този вариант, то трябва да се внимава от „overflow“ (или препълване). Дори целият интервал да се събира в даден тип, то сумата на лявата и дясната граница може да прехвърли възможностите на типа. Примерно какво би станало, ако интервала е с граници [1, 2 000 000 000] от тип *int* и се използва съкратения вариант:

***int** mid = (left+right)/2;*

Ако търсената стойност е близка до горната граница, то ще има стъпки в които *(left + right),* ще прехвърли възможностите на *int* и програмата ще даде грешен резултат. В това отношение по – дългия запис е по безопасен, тъй като няма този проблем.

Максималният брой опити, необходими на алгоритъма за намирането на елемент в сортиран списък, е приблизително двоичен логаритъм от дължината му закръглен в посока към по ниската стойност увеличена с единица. *($log_2(N) + 1$),* т.е за намиране на елемент в сортиран списък от 1000 елемента, алгоритъмът ще извърши най-много 10 опита (*$log_2(1000)+1 \approx 9+1=10$*). Това лесно може да се докаже като се има предвид факта, че при всяко предположение броя на елементите намалява на половина. Спрямо максималния брой операции може лесно да се предвиди, че сложността на алгоритъма спрямо “*Big O”* нотацията е *$O(log_2(N))$*.

Ако за търсенето на елемент в сортиран списък от 1000 елемента се използва линейния алгоритъм, който обхожда всички елементи на масива последователно, то в най - лошия случай ще са необходими 1000 опита (когато търсения елемент е последен). Ако се използва някой от линейните алгоритми с прескачане на елементи, върху същия списък от 1000 елемента, то броя на предположенията ще намалее, но не толкова че да подобри алгоритъма за двоично търсене. Това показва, че алгоритъма за “Binary Search” е многократно по добър от който и да е линеен алгоритъм за търсене на елемент.

Тъй като, само по себе си, двоичното търсене е сравнително кратко и просто за писане, най-често в задачи, то бива съчетано с друг, по-сложен алгоритъм или структура данни. То би могло да се прилага и в задачи чиито интервал е непрекъснат, тогава за границите на търсения интервал ще зависят от спецификата на задачата. Също така то би могло да се прилага и върху задачи, чиито стойности в търсените интервали са реални числа.

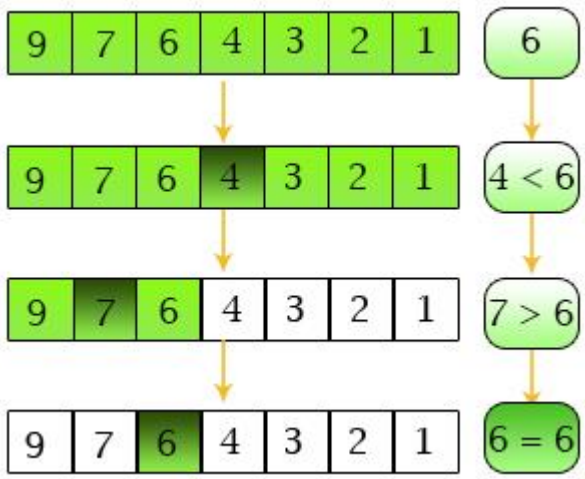
II. ПРАКТИЧЕСКА ЧАСТ

В практическата част са показани примери, които показват действието на разглежданите алгоритми за сортиране.

ЗАДАЧА1: *Да се приложи алгоритъма за двоично търсене като се търси числото 6 в следния сортиран масив от 7 елемента.*

{9, 7, 6, 4, 3, 2, 1}

РЕШЕНИЕ:



Фиг.1. Демонстрира решението на задача 1.

ПОЯСНЕНИЕ

Първоначално се намира средния елемент от масива. Това е елементът със стойност 4. Тъй като 4 не съвпада с търсената от стойност, алгоритъмът продължава да търси в лявата половина, понеже търсеното число е по – голямо от 4 (т.е. между елементи със стойности 9, 7, и 6). Отново се определя средният елемент, който е числото 7. Той е по-голям от търсената стойност. Всички числа вляво се елиминират и остава елементът 6. Тъй като той съвпада с търсената от нас стойност, алгоритъмът приключва.

ЗАДАЧА2: *С кои числа ще бъде сравнено числото 3 при двоично търсене в предния масив*

ОТГОВОР: *4, 2, 3*

ПОЯСНЕНИЕ

Първото число с което ще се сравни е 4 понеже той е средния елемент. Тъй като 4 не съвпада с търсената от стойност, алгоритъмът продължава да търси в дясната половина, понеже търсеното число е по – малко от 4 (т.е. между елементи със стойности 3, 2, и 1). Отново се определя средният елемент, което е числото 2. Следователно това е второто число с което ще се сравни търсената стойност. То е по – малко от търсената стойност. Всички числа в дясно се елиминират и остава елементът 3. Извършва се последно сравняване и се установява че това е търсеното число.

ЗАДАЧА3: *Да се напише фрагмент от програма която реализира търсене чрез стъпка на даден масив от 1000 елемента. Нека да се изчислява и времето за изпълнение на функцията*

ОТГОВОР:

```
#include <chrono>
Using namespace std::chrono;
......
int jmp_search(int arr[], int n, int key, unsigned step)
{
    int i, rt, lf;
    for (i = 0; i < n && arr[i] < key; i += step);
    if (i < step)
        lf = 0;
    else
        lf = i + 1 - step;
    if (n < i)
        rt = n-1;
    else
        rt = i;
    return seq_search(arr, lf, rt, key);  // фунцкия за последователно търсене в
//интервала от lf до rt
}// 5 7 8 9 12 45 65 68 69 70 80 90

……..

int main()

{   int arr[1000];

    int step, k, n = sizeof(arr) / sizeof(arr[0]);
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 5001; // задаване на произволна стойност от 0 до 5000 за
//текущия елемент
    }

……..

    auto start = high_resolution_clock::now();
    int pos = jmp_search(arr, n, k, step);
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<nanoseconds>(stop - start);

    cout << "\nVremeto za izpalnenie na jmp_search e: " << duration.count() << "
nanoseconds" << endl;

….


}
```

ПОЯСНЕНИЕ

Функцията приема 4-ри параметъра. Параметърът arr е масивът, n е броя на елементите в него , key е търсения елемент , step е стъпката. Цикъл for обхожда масива от ляво на дясно, като за следващия е елемент се определя от стъпката. Цикалът се обхожда докато текуция елемент е по малък от търсения и докато позицията на текущия елемент е по малка от броя на елементите в масива. В момента в който i стане по голямо от n или ако текущия елемент съвпада с търсения или ако е по голям от него

обхождането приключва. Следва да се извика линейна функция за последователно търсене която ще обходи елементите в съответния посочен итервал за да провери дали търсения елемент е в тях. Преди да се извика тази функция е необходимо да бъдат изчислени лявата и дясната граница на съответния интервал и да се подадат към тази функция. В горния фрагмент код във main функцията е показано как се генерира масив от 1000 елемента със произволни стойности. Като тези стойности може да са в интервала от 0 до 5000. Накрая е показано как се изчислява времето за изпълнение на дадена функция като се ползва хронометър. Задължително е в началото на програмата да бъде включена библиотеката `chrono`.

III. Въпроси и задача за самостоятелна работа.

1. Даден е масив, който съдържа сортирани простите числа от 2 до 311:

   {2, 3, 5, 7, 11, 13, ..., 307, 311}

   В масива има 64 елемента. Ако се използва двоичното търсене, колко сравнения трябва да се направят, преди да се установи, че числото 52 не е просто?
2. Напишете програма реализираща последователно търсене в несортиран масив. Редактирайте програмата да е по оптимална ако работи за сортиран масив.
3. Напишете програма, използвайки метода на двоичното търсене, която търси позицията на елемент със стойност 80 от следния масив:

   {18, 19, 22, 23, 29, 32, 41, 43, 69, 77, 80, 86, 90, 92, 95}

4. Допишете задача 3 от практическата част като довършите кода. Сравнете времето за изпълнение като търсите последния елемент на последователния алгоритъм от предната задача и този със стъпка, като за самата стъпка зададете 1. Анализирайте резултатите при следните стъпки: 1, 6, 10, 12, 15, 20, 50, 100, 200, 333, 800, 1000
5. Напишете програма, използвайки метода на двоичното търсене, която търси последния елемент от масив с размерност 1000:Анализирайте резултата от този при линейно последователно търсене и търсене чрез стъпка, като използвате тази стъпка която е била с най добро време спрямо предната задача.