Лабораторная работа № 4

“Исследование простых алгоритмов поиска”

Задание:

1. Разработать проект для исследования простых алгоритмов поиска в соответствии с вариантом.
2. Разработать интерфейс проекта, *позволяющий*:
   - задавать размерность и диапазон элементов массива;
   - элемент для поиска (образец);
   - осуществлять выбор алгоритма поиска для исследования;
   - осуществлять вывод информации о результатах исследования алгоритма поиска (исходный массив, пошаговую работу алгоритма поиска (при небольшой размерности массива), показатели качества работы алгоритма поиска).
3. Создать подпрограмму, реализующую алгоритм поиска в соответствии с вариантом. В подпрограмме *предусмотреть*:
   - поиск заданного элемента массива и индекс его первого вхождения;
   - определение числа присваиваний алгоритма поиска - *Pr*;
   - определение числа сравнений алгоритма поиска - *Sr*;
   - определение суммы присваиваний и сравнений алгоритма поиска *Sum*.
4. В качестве исследуемого массива использовать одномерный массив целых чисел, равномерно распределенных в интервале [*A, B*]. Для формирования элементов одномерного массива и образца использовать датчик равномерно распределенных чисел *rand*().
5. Результатом работы алгоритма поиска является номер первого вхождения образца в исследуемый массив.

Письменный отчет по лабораторной работе должен содержать:

1. Титульный лист. (Название лабораторной работы. Фамилия, имя, отчество, номер группы исполнителя, дата сдачи.)
2. Математическую постановку задачи.
3. Выполняемый вариант и его содержание.
4. Диаграмму классов.
5. Распечатку *подпрограмм алгоритмов поиска* и *кода интерфейса* проекта для исследования алгоритмов поиска (*обязательны комментарии* к программе).
6. Исследование программной реализации простых алгоритмов поиска, содержащее следующие материалы, таблицы и графики:
   - *примеры пошаговой работы* исследуемых алгоритмов поиска для небольшой размерности задачи ($n = 7$, диапазон чисел 0..10, искомое число есть в массиве, искомого числа нет в массиве);
   - сведенную в *таблицу* зависимость количества сравнений, присваиваний и их суммы (или времени выполнения) исследуемых

алгоритмов поиска от размерности задачи *n* для случайного равномерного расположения элементов в массиве,

Пример таблицы:

Таблица 1 - Зависимость показателя качества бинарного поиска от размерности массива

n	100	200	300	400	500	600	700	800	900	1000
Sr										
Pr										
Sum										

- *графики* зависимости показателя качества (количества операций или времени выполнения) исследуемых алгоритмов поиска от размерности задачи.

7. При исследовании *m*-блочного поиска определить также зависимость показателя качества алгоритма от числа блоков разбиения *m*.
8. Количество сравнений и присваиваний (или время счета) исследуемого алгоритма поиска определяется как математическое ожидание *для числа испытаний не менее 100*.
9. Привести результаты тестирования проекта.
10. *Выводы* по лабораторной работе (в выводах провести сравнительную характеристику исследованных алгоритмов поиска).

Варианты задач по лабораторной работе:

1. Линейный поиск (или линейный поиск с барьером).
2. Бинарный поиск.
3. Интерполяционный поиск.
4. *m*–блочный поиск.
5. Поиск с помощью встроенных функций C# (или *C++*).

Таблица 1 Варианты исследуемых алгоритмов поиска

Вариант	*Интервал [A, B]*	*Варианты задач*
1.	[10, 5000]	1, 2, 5
2.	[1000, 2000]	1, 3, 5
3.	[100, 6000]	1, 4, 5
4.	[100, 900]	2, 3, 5
5.	[0, 1000]	2, 4, 5
6.	[10, 200]	3, 4, 5
7.	[10, 1000]	1, 2, 5
8.	[100, 900]	1, 3, 5
9.	[10, 800]	1, 4, 5
10.	[50, 500]	2, 3, 5
11.	[0, 999]	2, 4, 5
12.	[100, 1000]	3, 4, 5
13.	[0, 1000]	1, 2, 5
14.	[1000, 5000]	2, 4, 5

15.	[10, 999]	1, 4, 5
16.	[10, 1500]	2, 3, 5
17.	[100, 3500]	2, 4, 5
18.	[0, 1600]	3, 4, 5
19.	[0, 900]	1, 2, 5
20.	[0, 800]	1, 3, 5

Таблица 2 Основные встроенные методы поиска языка C#

№	Метод поиска	Класс
1	IndexOf()	**Array**
2	LastIndexOf()	Array
3	BinarySearch()	Array
4	Find()	Array
5	FindIndex()	Array
6	FindLast()	Array
7	FindLastIndex()	Array
8	FindAll()	Array
9	Exists()	Array
10	IndexOf()	**String**
11	IndexOfAny()	String
12	LastIndexOf()	String
13	LastIndexOfAny()	String
14	StartsWith()	String
15	EndsWith()	String
16	IsMatch()	**Regex**

Литература

1. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ.-М.:МЦНМО, 2000.-960с.(с.236-253, 780-809)
2. Кнут Д. Искусство программирования для ЭВМ. Т.3. Сортировка и поиск.-М.:Мир,1978.-844с.(с.470-492).
3. Седжвик Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск.- К.: Диасофт, 2001. – с. 688 (с. 496)
4. Вирт Н. Алгоритмы и структуры данных.- М.: Мир, 1989. (с. 72, 74-78)
5. Проценко В.С. Техніка програмування мовою Сі: Навч. Посібник,-К.:Либідь, 1993.-224с.(с.111-116).
6. Мейер Б., Бодуэн К. Методы программирования Т.2.-М.:Мир, 1982.-368с.(с.108-115).
7. Макконелл Дж. Анализ алгоритмов. Вводный курс.-М.: Техносфера, 2002. (с. 53)

Контрольные вопросы и задания

1. Каковы основные операции информационного поиска?
2. В чем состоит задача простого поиска?
3. Каковы основные типы условий операции поиска?
4. Перечислите простые алгоритмы поиска.

5. В чем состоит алгоритм линейного поиска в массиве? Каковы его сложность, достоинства и недостатки?
6. В чем состоит алгоритм поиска с индексацией по ключам? Каковы его сложность, достоинства и недостатки?
7. В чем состоит алгоритм последовательного поиска в упорядоченном массиве? Каковы его сложность, достоинства и недостатки?
8. В чем состоит алгоритм бинарного поиска? Каковы его сложность, достоинства и недостатки?
9. В чем состоит алгоритм интерполяционного поиска? Каковы его сложность, достоинства и недостатки?
10. В чем состоит алгоритм *m*–блочного поиска? Каковы его сложность, достоинства и недостатки?

Лабораторная работа № 4

“Исследование простых алгоритмов поиска”

Задание (более сложное):

1. Провести сравнительную оценку всех 5-ти алгоритмов простого поиска (линейный поиск (или линейный поиск с барьером), бинарный поиск, интерполяционный поиск, *m*–блочный поиск, поиск с помощью встроенных функций *C++*/C#.) по методике Лр № 4.
2. Алгоритмы простого поиска должны являться методами класса.

Краткий теоретический материал

Алгоритмы решения простой задачи поиска

1. Линейный поиск в массиве

Элементы множества *S* запоминаются в массиве *X*.

Идея алгоритма – последовательный просмотр элементов массива *X* с увеличением той его части, где искомого элемента *a* не обнаружено. Алгоритм линейного поиска в массиве возвращает индекс искомого элемента *a* в массиве (индекс записи, содержащей нужный ключ). Если искомый элемент *a* не найден, то алгоритм поиска обычно возвращает значение индекса, превышающего верхнюю или нижнюю границу массива.

Пусть массив *X* задан так : *int X[N]*;

Условия окончания алгоритма линейного поиска:

1. элемент найден, то есть *X*[*i*]= *a*,
2. весь массив просмотрен и совпадений не обнаружено, то есть *i>N (*или *i<0).*

Таким образом, алгоритм линейного поиска имеет вид (алгоритм находит совпадающий элемент с наименьшим индексом) -

i=0; While((*i<N*) *&&* (!(*X*[*i*]= =*a*))) *do* ++*i*;

Ясно, что на каждом шаге требуется увеличивать индекс и вычислять логическое выражение. Можно ли эту работу упростить и таким образом ускорить поиск? Единственная возможность – попытаться упростить само логическое выражение, ведь оно состоит из двух частей. Надо сформировать простое логическое условие, эквивалентное используемому выше сложному. Это можно сделать, если гарантировать, что совпадение *X*[*i*]=*a* всегда произойдет. Для этого достаточно в конец массива *X* поместить дополнительный элемент со значением *a*. Такой вспомогательный элемент называется “барьером”, так как он охраняет от перехода за пределы массива. Теперь массив должен быть описан так – *int X[N+1]*;

Алгоритм линейного поиска с барьером выглядит следующим образом:

X[*N*] = *a;*

i=0*;*

While (! (*X*[*i*] = = *a*)) *do* ++*i*;

Ясно, что равенство *i= N* свидетельствует о том, что совпадения (если не считать совпадения с барьером) не было.

Сложность алгоритма составляет *O*(*n*).

Наихудший случай линейного алгоритма поиска:

- искомый элемент стоит в массиве последним;
- искомого элемента в массиве нет.

2. Поиск с использованием индексации по ключам

Данный алгоритм поиска идейно совпадает с алгоритмом сортировки подсчетом.

Ключ
Информационная часть

Пусть значения ключей – отдельные *целые положительные числа*. Данный алгоритм поиска состоит в сохранении элементов в массиве, индексированном по ключам. Исходная последовательность ключей записана в массиве *X*[1..*n*], *C*[1..*k*] –

вспомогательный массив (k – мощность диапазона чисел, используемых в исходном массиве $X[1..n]$).

Особенности алгоритма:

1. все элементы массива C инициализируются значением *null*,
2. информационную часть элемента со значением ключа $X[i]$ сохраняем в массиве по индексу $C[X[i]]$,
3. поиск ключа $a = X[j]$ успешен, если информационная часть элемента $C[a]$ не равна *null*,
4. сложность поиска равна $O(1)$ → главное *достоинство* данного алгоритма.

Пример. Пусть массив ключей и соответствующих информационных частей имеет вид:

Массив ключей	2	7	1	6	4	9
Массив информационных частей	3,15	3,25	3,16	3,78	4,20	5,11

Очевидно, что $k = 9$, а вспомогательный массив $C[1..k]$ имеет вид:

Индекс	*Значение*
1	3,16
2	3,15
3	*null*
4	4,20
5	*null*
6	3,78
7	3,25
8	*null*
9	5,11

Недостатки алгоритма поиска с индексацией по ключам:

1. в качестве ключей нужны специальные числа (или необходимо тратить время на преобразование исходных ключей в “хорошие” ключи),
2. проблема возникает при наличии дублированных ключей → переход к хешированию,
3. проблемы при неплотных значениях ключей - пусть число ключей $n = 2$, но $X[1]=1$, а $X[2]=20$ – это потребует массива записей C размерности 20, причем использоваться будет только две ячейки из 20 (занято $2/20=0{,}10$ → 10% памяти отведенной под массив C),
4. проблемы при плотных значениях ключей в диапазоне $[A, B]$, но если $(A>>1)$ – пусть число ключей $n = 3$, но $X[1]=255$, $X[2]=257$, а $X[3]=256$ – это потребует массива записей C размерности 257, причем использоваться будет только $n = 3$ ячейки из 257 (занято $3/257=0{,}0117$ → 1,17% памяти отведенной под массив C),
5. проблема при малом числе ключей, но большом их диапазоне – пусть число ключей $n = 2$, но $X[1]=1$, а $X[2]=10000$ – это потребует массива записей C размерности 10000.

3. Бинарный поиск

Поиск можно делать значительно более эффективным, если данные в массиве будут *упорядочены*.

Алгоритм бинарного поиска

1. (Начальная установка.) Установить l=0. n=(N-1) (l - нижний индекс упорядоченного массива, n - верхний индекс).

2. (Нахождение середины.) (Если искомый элемент a есть в массиве, то выполняется неравенство $X[l] \le a \le X[n]$.) Если $n < l$, то алгоритм заканчивается неудачно, в противном случае установить i=[(l+n) /2] ([*] – ближайшее меньшее целое), теперь i указывает примерно на середину массива.

3. (Сравнение.) Если $a< X[i]$, то перейти на 4, если $a> X[i]$, то перейти на 5, если $a= X[i]$, то поиск заканчивается удачно.

4. (Коррекция n.) Установить $n = i$ -1 и вернуться к шагу 2.

5. (Коррекция l.) Установить $l = i$ +1 и вернуться к шагу 2.

0	…	i-1	i	i+1	…	n

Пример. Исходный массив - X={15, 61, 87, 88, 90, 93, 95, 98}, размерность исходного массива - n =8, искомый элемент - a=95,

а) l=0, n=7, i=[(l+n)/2]=[3.5]=3
15, 61, 87, **88**, [90, 93, 95, 98]
$a>X[3]$, l=i+1=4, n=7, i=[(4+7)/2]=[5.5]=5
15, 61, 87, 88, [90, **93**, 95, 98]
$a>X[5]$, l=i+1=6, n=7, i=[(6+7)/2]=[6.5]=6, $a=X[6]$ → поиск закончен.
15, 61, 87, 88, 90, 93, [**95**, 98]

Потребовалось 3 сравнения.

б) Поиск числа 40.
[15, 61, 87], **88**, 90, 93, 95, 98
[15, **61**, 87], 88, 90, 93, 95, 98
[15], 61, 87, 88, 90, 93, 95, 98

Недостаток бинарного поиска – сложность создания отсортированного массива и сложность включении новых элементов в отсортированный массив.

Наихудший случай алгоритма бинарного поиска – число проходов (сравнений) при поиске равно $\log_2(N+1)$.

Теорема. При $2^{k-1} \le N < 2^k$ удачный поиск, использующий алгоритм двоичного поиска, требует (min 1, max k) сравнений. Неудачный поиск требует k сравнений при $N=2^k-1$, либо − (k -1) или k сравнений при $2^{k-1} \le N < 2^k$.

Таким образом, алгоритм бинарного поиска требует максимум $\approx k=\log_2 N+1$ сравнений, то есть имеет сложность $O(\log N)$.

Особенности алгоритма бинарного поиска:

1. Индекс разбиения может выбираться произвольно, корректность алгоритма при этом не нарушается. Однако использование балансировки, то есть выбор каждый раз среднего элемента, является оптимальным решением, так как при этом каждый раз исключается половина проверяемого массива.

2. В общем случае дерево решений относительно сбалансировано, поскольку всегда выбирается середина различных частей списка.

3. Максимальное число сравнений равно $[\log_2(N+1)]$, то есть временная сложность алгоритма бинарного поиска $O(\log N)$.

4. Интерполяционный поиск

Одно из возможных усовершенствований бинарного поиска – более точное предположение о размещении ключа поиска *a* в текущем интервале. Эта тактика имитирует способ поиска абонента в телефонном справочнике (или слова вы словаре) – если искомая запись начинается с буквы, находящейся в начале алфавита, то поиск выполняется вблизи начала справочника, но если она начинается с буквы, находящейся в конце алфавита, поиск выполняется в конце справочника.

Выражение для поиска индекса середины интервала ((*l*+*n*)/2) равнозначно выражению (*l*+(*n*-*l*)/2) – вычисляется середина интервала, добавляется к левой граничной точке половина размера интервала. Использование интерполяционного поиска сводится к замене в этой формуле коэффициента *k*=1/2 ожидаемой позицией ключа – а именно *k*=(*a* - *X*[*l*])/(*X*[*n*] - *X*[*l*]). При этом предполагается, что значения ключей (элементов массива *X*) являются числовыми и равномерно распределенными.

В интерполяционном поиске индекс прогнозируемого места искомого элемента *a* определяется по формуле *i*=[(*l*+(*n*-*l*)***k*)] ([*] – ближайшее меньшее целое, *i* указывает на примерное место нахождения ключа *a*).

Алгоритм интерполяционного поиска

1. (Начальная установка.) Установить *l*=0. *n*=(*N*-1) (*l* – нижний индекс упорядоченного массива, *n* – верхний индекс, *N* – число элементов в упорядоченном массиве), *a* – искомый элемент.

2. (Вычисление коэффициента интерполяции.) (Если искомый элемент а есть в массиве, то выполняется неравенство *X*[l] ≤ *a* ≤ *X*[*n*].) *k* = (*a* - X[*l*]) / (*X*[*n*] - X[*l*]) (важна проверка знаменателя на равенство 0 – равные элементы в массиве).

3. (Нахождение прогнозируемого места искомого элемента.) Если *n* ≤ *l*, то алгоритм заканчивается неудачно, в противном случае установить *i*=[(*l*+(*n*-*l*)***k*)] ([*] – ближайшее меньшее целое), теперь *i* указывает на прогнозируемое место искомого элемента в массиве.

4. (Сравнение.) Если *a* < *X*[*i*], то перейти на 5, если *a* > *X*[*i*], то перейти на 6, если *a*= X[*i*], то поиск заканчивается удачно.

5. (Коррекция *n*.) Установить *n* = *i* -1 и вернуться к шагу 2.

6. (Коррекция *l*.) Установить *l* = *i* +1 и вернуться к шагу 2.

Пример 1. Исходный массив - *X*={15, 61, 87, 88, 90, 93, 95, 98}, размерность исходного массива - *N* =8, искомый элемент – *a* =95, *l* =0, *n* =7.

k=(*a* - *X*[*0*])/(*X*[*7*] - *X*[*0*])=(95-15)/(98-15)=0,9638.

i=[(*l*+(*n*-*l*)***k*)]=[0+(7-0)***0,9638]=[6,72]=6 → действительно искомый элемент *a*=95 равен *X*[*6*]=95.

Пример 2. Исходный массив - *X*={15, 61, 87, 88, 90, 93, 95, 98}, искомый элемент – *a* =40, *l* =0, *n* =7.

k=(*a* - *X*[*0*])/(*X*[*7*] - *X*[*0*])=(40-15)/(98-15)=25\83=0,3012.

i=[(*l*+(*n*-*l*)***k*)]=[0+(7-0)***0,3012]=[2,1]=2

a =40 < *X*[2]=87 → коррекция *n* − *n*=i-1=2-1=1

k=(*a* - *X*[*0*])/(*X*[*1*] - *X*[*0*])=(40-15)/(61-15)=25\46=0,5435.

i=[(*l*+(*n*-*l*)***k*)]=[0+(1-0)***0,5435]=[0,5435]=0

a =40 > *X*[0]=15 → коррекция *l* − *l*=i+1=0+1=1.

Так как n=1 ≤ *l*=1, то поиск неуспешен.

Пример 3. Исходный массив - *X*={6, 21, 25, 26, 31, 37}, искомый элемент – *a* =29, *l* =0, *n* =5, *N* =6.

k=(*a* - *X*[*0*])/(*X*[*5*] - *X*[*0*])=(29-6)/(37-6)=0,7419.

i=[(l+(n-l)*k)]=[0+(5-0)*0,7419]=[3,7]=3

a =29 > X[3]=26 → коррекция l − l=i+1=3+1=4

Так как a =29 < X[4]=31, то искомого элемента нет в массиве, то есть поиск не успешен.

При интерполяционном поиске в файлах с произвольными ключами для каждого поиска (попадания или промаха) используется менее *loglogN* сравнений. Эта функция растет очень медленно и на практике ее можно считать постоянной – если N равно 1 миллиарду (10^6), то *loglogN* <5.

Особенности интерполяционного поиска:

1. Ключи в массиве должны быть упорядочены и не равны друг другу.
2. Интерполяционный поиск в значительной степени основывается на предположении, что ключи распределены во всем интервале более-менее равномерно – вид коэффициента k зависит от закона распределения ключей в массиве.
3. Коэффициент k требует дополнительных вычислений.
4. Для небольших значений N сложность бинарного поиска O(*logN*) достаточно близка к сложности интерполяционного поиска O(*loglogN*), и поэтому вряд ли стоит использовать последний метод.
5. Интерполяционный поиск эффективен:
   - при работе с большими файлами,
   - при использовании внешних файлов, что сопряжено с большими затратами на доступ.

5. m-блочный поиск в массиве

Идея алгоритма состоит в следующем:

– исходный упорядоченный массив X длины n делится на m упорядоченных подмассивов $X_1, X_2,\ldots, X_m$ длин $n_1, n_2, \ldots n_m$, $\sum_{i=1}^{m} n_i = n$.

– при поиске ключа a сначала определяется первый подмассив X_i, 1<=i<=m, последний элемент которого ≥ a, а потом к этому подмассиву применяется линейный поиск (или какой-либо другой) (По существу m-блочный поиск является комбинированным алгоритмом поиска.). Подмассивы X_i можно хранить связанно.

Выбор длины подмассивов – l=n/m =[√n] ([x] – целая часть x).

Если длины всех подмассивов примерно одинаковы, их число m=√n, то максимальное число сравнений в m–блочном поиска равняется 2*√n. При одинаковой частоте использования всех элементов среднее число сравнений равняется √n. Следовательно, *сложность алгоритма* $O(n^{1/2})$.

Пример. Пусть исходный массив - X={15, 61, 87, 88, 90, 93, 95, 98}, размерность исходного массива - n =8, искомый элемент - a=95, пусть m=4.

X_1={15, 61}, X_2={87, 88}, X_3={90, 93}, X_4={95, 98}, n_i=2, i=1, 4

1. Определить первый подмассив, последний элемент которого ≥a. Этот подмассив X_4.

2. С помощь линейного поиска найдем образец a=95 в подмассиве X_4.

Число сравнений: – на шаге 1 – 4, на шаге 2 – 1.