9 . АЛГОРИТМЫ ПОИСКА

Поиск - обработка некоторого множества данных с целью выявления подмножества данных, соответствующего критериям поиска.

Все алгоритмы поиска делятся на

- поиск в неупорядоченном множестве данных;
- поиск в упорядоченном множестве данных.

Упорядоченность – наличие отсортированного ключевого поля.

9.1 Поиск в неупорядоченном множестве данных.

9.1.1 Линейный, последовательный поиск

Линейный, последовательный поиск (также известен как поиск методом полного перебора или *брутфорса*) — алгоритм нахождения заданного значения произвольной функции на некотором отрезке. Данный алгоритм является простейшим алгоритмом поиска и, в отличие, например, от двоичного поиска, не накладывает никаких ограничений на функцию и имеет простейшую реализацию. Поиск значения функции осуществляется простым сравнением очередного рассматриваемого значения и, если значения совпадают (с той или иной точностью), то поиск считается завершённым.

Пример 9.1. Поиск первого вхождения заданного значения в массив.

Вариант 1.

```
void main()
{
        setlocale (LC_CTYPE,"rus");
        int a[100];
//формирование массива a
      int key;
      cout<<"\nKey =?";
      cin>> key;
      int i, nom=-1;
      for(i=0; i<n; i++)
      if(a[i]== key){nom=i; break;}
      if(nom !=-1)
      cout<<"\nnom="<<nom+1;
      else
      cout<<"\nthere is no such element! ";
}
```

Вариант 2 (вместо выделенного фрагмента).

```
       int i = 0;
       while ( (i< n) && (a[i] != key) )          i++;
      if (i==n)
      cout<<"\nthere is no such element!";
```

```
else
cout<<"\nnom="<<i+1;
```

Единственная модификация этого метода, которую можно сделать, - избавиться от проверки номера элемента массива в заголовке цикла (i < n) за счет увеличения массива на один элемент в конце, значение которого перед поиском устанавливают равным искомому ключу - key - так называемый "барьер":

```
a[n] = key;
int i = 0;
while (a[i] != key)          i++;
if (i==n)
cout<<"\nthere is no such element!";
else
cout<<"\nnom="<<i+1;    // i< n – возврат номера элемента
```

При необходимости нахождения последнего вхождения заданного значения в массив проверку надо организовать, начиная с последнего элемента массива (for(i= n-1; i>=0; i--)).

Данный код представляет собой самый простой поиск элемента в массиве. Однако в нем не рассматривается случай, когда в массиве не единственный элемент равный *key*. Будем считать все случаи, включая наихудший, в котором все элементы массива равны *key*. Модифицируем программу – введем массив *b*, в который будем сохранять позиции элементов массива, значения которых равны *key*.

Пример 9. 2. Поиск в массиве позиций всех элементов, равных заданному значению.

```
void main()
{
        setlocale (LC_CTYPE,"rus");
        int a[100],b[100];
        int n;
        //формирование массива a
        int key;
        cout<<"\nKey =?";
        cin>> key;
        int i, nom=-1;
        for(i=0; i<n; i++)
                if(a[i]== key) b[++nom]=i;
        if(nom !=-1)
        {
                cout<<"\nnom=";
                for(i=0; i<=nom; i++) cout<<b[i]<<"  ";
        }
```

```
        else
                cout<<"\nthere is no such element! ";
}
```

9.1.2 Метод транспозиции

Улучшением рассмотренного метода является метод транспозиции: каждый запрос к записи сопровождается сменой мест этой и предшествующий записи; в итоге наиболее часто используемые записи постепенно перемещаются в начало таблицы; и при последующем обращении к ней запись будет находиться почти сразу.

Пример 9.3. Поиск в массиве методом транспозиции.

```
void main()
{
        setlocale (LC_CTYPE,"rus");
        int a[100];
//формирование массива a
      int key;
      cout<<"\nKey =?";
      cin>> key;
      int i, nom=-1;
      for(i=0; i<n; i++)
      if(a[i]== key)
      {
        nom=i;
        if(nom == 0)   break;
        temp = a [i];
        a [i] = a [i-1];
        a [i-1] = temp;
        break;
      }
      if(nom !=-1)
      cout<<"\nnom="<<nom;
      else
      cout<<"\nthere is no such element! ";
}
```

9.1.3 Метод перемещения в начало

Каждый запрос к записи сопровождается её перемещением в начало таблицы; в итоге в начале таблицы оказывается запись, используемая в последний раз.

Пример 9.4. Поиск в массиве методом перемещения в начало.

```
int search(int *k, int n, int key) {
  int nom = -1;
```

```
 int temp;
 for(int i=0; i<n; i++)
{
   if(k[i] == key)
  {
     nom = i;
     temp = k[i];
     k[i] = k[0];
     k[0] = temp;
     break;
   }
 }
return(nom);
}
```

Если отрезок имеет длину *n*, то найти решение с точностью до ε можно за время n/ε. Т.о. асимптотическая сложность алгоритма — *O(n)*. В связи с малой эффективностью по сравнению с другими алгоритмами линейный поиск обычно используют, только если отрезок поиска содержит очень мало элементов. Тем не менее, линейный поиск не требует дополнительной памяти или обработки/анализа функции, так что может работать в потоковом режиме при непосредственном получении данных из любого источника. Также линейный поиск часто используется в виде линейных алгоритмов поиска максимума/минимума.

Анализ.

Для списка из *n* элементов лучшим случаем будет тот, при котором искомое значение равно первому элементу списка и требуется только одно сравнение. Худший случай будет тогда, когда значения в списке нет (или оно находится в самом конце списка), в случае чего необходимо *n* сравнений.

Если искомое значение входит в список *k* раз и все вхождения равновероятны, то ожидаемое число сравнений

$$\begin{cases} n, & k = 0 \\ \dfrac{n+1}{k+1}, & 1 \le k \le n \end{cases}$$

Например, если искомое значение встречается в списке не один раз, и все вхождения равновероятны, то среднее число сравнений равно $\frac{n+1}{2}$. Однако, если известно, что оно встречается один раз, то достаточно *n - 1* сравнений, и среднее число сравнений будет равняться $\frac{(n+2)(n-1)}{2n}$.

(для n = 2 это число равняется 1, что соответствует одной конструкции if-then-else).

В любом случае вычислительная сложность алгоритма *O(n)*.

Если известно, что данные в массиве расположены упорядоченно (будем считать, что элементы отсортированы по возрастанию), то можно предложить намного более эффективный способ поиска элементов.

9.2 Поиск в упорядоченном множестве данных.

9.2.1 Двоичный поиск

Двоичный (бинарный) поиск (также известен как метод деления отрезка пополам и дихотомия) — классический алгоритм поиска элемента в отсортированном массиве (векторе), использующий дробление массива на половины. Используется в информатике, вычислительной математике и математическом программировании.

Алгоритм. Массив делится пополам *s=(l+r)/2(* где *l, r* – левая и правая границы массива соответственно*)* и определяется, в какой части массива находится нужный элемент *key*. Т.к. массив упорядочен, то если *a[s] < key*, то искомый элемент находится в правой части массива, иначе – находится в левой части. Выбранную часть массива снова надо разделить пополам и т.д., до тех пор, пока границы отрезка *l* и *r* не станут равны [11].

Для того, чтобы найти нужную запись в таблице, в худшем случае нужно $\log_2(N)$ сравнений (округление производится в большую сторону до ближайшего целого числа). Это значительно лучше, чем при последовательном поиске.

Приведем иллюстрация бинарного поиска на примерах.

Пример 9.5.Реализация алгоритма бинарного поиска.

1) Вариант 1.

```
void main()
{
	setlocale (LC_CTYPE,"rus");
	int a[100];
	int n,I;
//формирование массива a
	cout<<"\nEnter the size of array:";
	cin>>n;
	for( I=0; I<n; I++) cin>>a[I];
	int key;
	cout<<"\nKey =?";
	cin>> key;
	int l=0, r=n-1, s;
	bool Found = false;		// флаг
	while ( (l <=r) && !Found)	// цикл, пока интервал поиска не
// сузиться до 0
	{
		s = ( l + r ) / 2;		// середина интервала
		if ( a[s] == key )
		Found = true;			// ключ найден
		else
			if ( a[s] < key )
				l = s + 1;		// поиск в правом подинтервале
			else
				r = s - 1;		// поиск в левом подинтервале
	}
	 if (Found == true)  cout<< "Элемент найден на позиции "<< s+1<< " \n";
	 else cout<< "Элемент не найден!\n";

}
```

2) Вариант 2.

```
void main()
{
	setlocale (LC_CTYPE,"rus");
	int a[100];
	int n,I;
	//формирование массива a
	cout<<"\nEnter the size of array:";
	 cin>>n;
	for( I=0; I<n; I++) cin>>a[I];
	int key;
	cout<<"\n Key =?";
	cin>> key;
	int l=0, r=n-1, s;
	do
	{
		s=(l+r)/2;		//найти средний элемент
		if (a[s]< key) l=s+1;	//перенести левую границу
		else r=s;		//перенести правую границу
	}
	while(l!=r);
	if(a[l]== key) cout<< "Элемент найден на позиции "<< l+1<< " \n";
	else cout<< "Элемент не найден!\n";
}
```

Алгоритм может быть определен в рекурсивной и нерекурсивной формах.

Бинарный поиск также называют поиском методом деления отрезка пополам.

Количество шагов поиска определится как $\log_2 n\uparrow$, где n-количество элементов, ↑ - округление в большую сторону до ближайшего целого числа (рис.9.1).

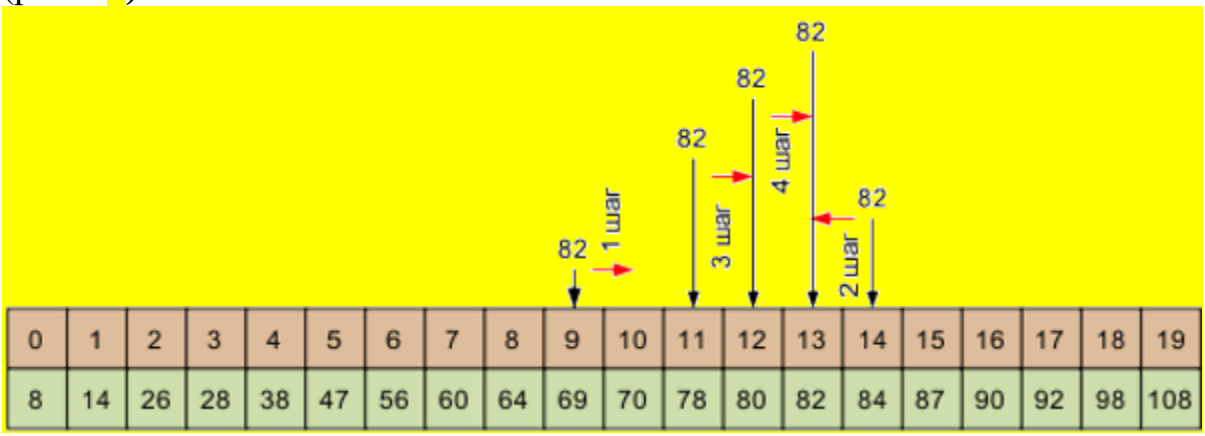


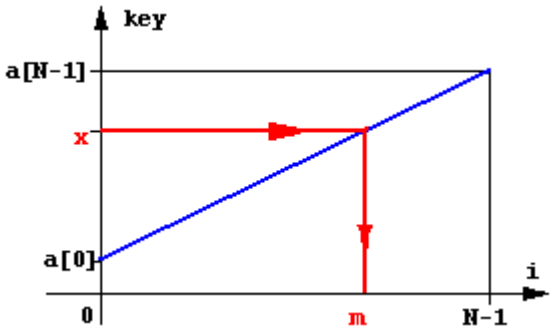
Рис. 9.1. Демонстрация метода бинарного поиска

Частным случаем двоичного поиска является метод бисекции, который применяется для поиска корней заданной непрерывной функции на заданном отрезке.

Практические приложения метода двоичного поиска разнообразны:

- Широкое распространение в информатике применительно к поиску в структурах данных. Например, поиск в массивах данных осуществляется по ключу, присвоенному каждому из элементов массива (в простейшем случае сам элемент является ключом).
- Также его применяют в качестве численного метода для нахождения приближённого решения уравнений.
- Метод используется для нахождения экстремума целевой функции и в этом случае является методом условной одномерной оптимизации. Когда функция имеет вещественный аргумент, найти решение с точностью до ε можно за время $log_2 1/\varepsilon$. Когда аргумент дискретен, и изначально лежит на отрезке длины *N*, поиск решения займёт $1 + log_2 N$ времени. Наконец, для поиска экстремума, скажем, для определённости минимума, на очередном шаге отбрасывается тот из концов рассматриваемого отрезка, значение в котором максимально.

9.2.2 Метод интерполяции



Если нет никакой дополнительной информации о значении ключей, кроме факта их упорядочения, то можно предположить, что значение *key* увеличиваются от *a[0]* до *a[N-1*] более или менее "равномерно". Это означает, что значение среднего элемента *a[N/2]* будет близким к среднему арифметическому между наибольшим и наименьшим значением. Но, если искомое значение *key* отличается от указанного, то есть некоторый смысл для проверки брать не средний элемент, а "средне-пропорциональный", то есть тот, номер которого пропорционален значению *key*:

```
s=l+(key-a[l])*(r-l)/(a[r]-a[l]);      // "средне-пропорциональный"
```

Выражение для текущего значения и получено из пропорциональности отрезков на рисунке: (a[r]-key)/(key-a[l])=(r-s)/(s-l);

Пример 9.5.Реализация метода интерполяции

```
void main()
{
    setlocale (LC_CTYPE,"rus");
    int a[100];
    int n,I;
    //формирование массива a
    cout<<"\nEnter the size of array:";
    cin>>n;
    for( I=0; I<n; I++) cin>>a[I];
    int key;
    cout<<"\nKey =?";
    cin>> key;
    int l=0, r=n-1, s;
    bool Found = false;                 // флаг
    while ( (l <=r) && !Found)      // цикл, пока интервал поиска не
// сузится до 0
    {
            s=l+(key-a[l])*(r-l)/(a[r]-a[l]);  // "средне-пропорциональный"
            if ( a[s] == key )
            Found = true;                       // ключ найден
            else
                    if ( a[s] < key )
                            l = s + 1;              // поиск в правом подинтервале
                    else
                            r = s - 1;              // поиск в левом подинтервале
    }
     if (Found == true)  cout<< "Элемент найден на позиции "<< s+1<< " \n";
     else cout<< "Элемент не найден!\n";
}
```

В среднем этот алгоритм должен работать быстрее бинарного поиска, но в худшем случае будет работать гораздо дольше.

9.2.3 Индексно-последовательный поиск

Для индексно-последовательного поиска в дополнение к отсортированной таблице заводится вспомогательная таблица, называемая ***индексной***. Каждый элемент индексной таблицы состоит из ключа и указателя на запись в основной таблице, соответствующей этому ключу. Элементы в индексной таблице, как элементы в основной таблице, должны быть отсортированы по этому ключу. Если индекс имеет размер, составляющий 1/8 от размера основной таблицы, то каждая восьмая запись основной таблицы будет представлена в индексной таблице (рис.9.2).

Если размер основной таблицы — **n**, то размер индексной таблицы — **ind_size = n/8**.

Достоинство алгоритма – сокращается время поиска, так как последовательный поиск первоначально ведется в индексной таблице, имеющей меньший размер, чем основная таблица. Когда найден правильный индекс, второй последовательный поиск выполняется по небольшой части записей основной таблицы.

Рис. 9.2. Демонстрация индексно-последовательного поиска

Пример 9.6. Реализация алгоритма индексно-последовательного поиска

```
using namespace std;
void main()
{
  int k[20]; // массив ключей
  int i, j, ind_size;
  int key;
  int kindex[3]; // массив ключей индексной таблицы
  int pindex[3]; // массив индексов индексной таблицы
  setlocale (LC_CTYPE,"rus");
  // Инициализация ключевых полей упорядоченными значениями
 cout<<"Инициализация ключевых полей упорядоченными значениями\n";
```

```
  for( i=0; i<20; i++)  cin>>k[i];
// Формирование индексной таблицы
  for(i=0, j=0;i<20;i=i+8)
 {
    kindex[j] = k[i];
    pindex[j] = i;
    j++;
  }
  ind_size = j;
  pindex[j] = 20;
// Поиск
  cout<<"Введите key: ";
  cin>>key;
  for(j=0; j<ind_size; j++)
    if(key < kindex[j])      break;
  if(j==0)  i=0;
  else        i = pindex[j-1];
  for (; i<pindex[j]; i++)
     if(k[i]==key)     cout<<"   k["<<i<<"]="<<k[i];
}
```

9.3 Вопросы для самоконтроля

1. Классификация алгоритмов поиска.
2. Алгоритм линейного поиска.
3. Достоинства и недостатки линейного поиска.
4. Какие алгоритмы поиска в отсортированном массиве вам известны?
5. Алгоритм бинарного поиска.
6. Применение алгоритмов бинарного поиска.
7. Алгоритм индексно-последовательного поиска.