Алгоритмы поиска

Цель работы:

1)Знакомство с алгоритмами поиска.

2)Получение навыков для работы с алгоритмами.

3)Практическое применение алгоритмов.

Теоретические сведения:

1)Поиск – процесс нахождения конкретной информации в ранее созданных последовательностях данных. Обычно данные представляют собой элемент, каждый из которых имеет хотя бы один ключ.

2)Ключ поиска – это поле данных, по значению которого происходит поиск. Ключи используются для отличия одного данного от другого.

Алгоритм линейного поиска

Для примера взят массив данных.

1)Массив просматривается последовательно от первого до последнего элемента. Наихудший случай – искомого элемента нет в массиве, тогда вывод результата можно сделать после просмотра всего массива.

2)Линейный поиск обычно используют, только если отрезок поиска содержит очень мало элементов.

3)Алгоритм не требует дополнительной памяти или обработки/анализа функции, может работать в потоковом режиме при непосредственном получении данных.

4)Достоинства – простая реализация.

5)Недостаток – длительное время работы.

Алгоритм:

Шаг 1) Начальное значение переменной цикла i = 0.

Шаг 2) Если значение элемента массива arr[i] равно значению key, по которому происходит поиск, то возвращается индекс искомого элемента, и алгоритм завершает работу. В противном случае значение переменной цикла увеличивается на единицу i=i+1;

Шаг 3) Если i < size, где size – число элементов массива arr, то выполняется шаг 2. В противном случае – работа алгоритма завершается и возвращается значение -1) При наличии в массиве нескольких элементов со значением key алгоритм находит только первый из них (с наименьшим индексом).

Код алгоритма.

int Search(int \*arr, int size, int key) {//массив, размер массива и искомое значение

for (int i = 0; i < size; i++){

if (arr[i] == key) {

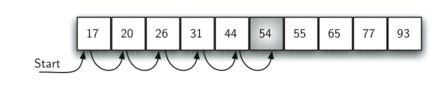
return i; //если нашли, то возвращаем индекс

}

}

return -1; //если не нашли, то возвращаем -1

}



Алгоритм бинарного поиска

Бинарный поиск – это поиск заданного элемента на упорядоченном множестве, который осуществляется путем неоднократного деления этого множества на две части таким образом, что искомый элемент попадет в одну из этих частей. Поиск заканчивается нахождением искомого элемента, который является границей между частями множества или же при отсутствии искомого элемента.

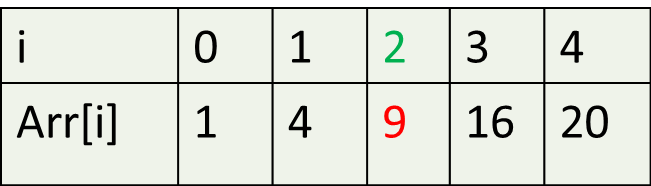
Изначально right и left – индексы границ множества.

Шаг 1) Определить индекс среднего элемента mid = (right+left)/2, где right и left – индексы границы текущей части множества.

Шаг 2) Если значение элемента, находящегося посередине текущей части множества, равно искомому, то элемент найден, его индекс выводится и алгоритм заканчивает работу.

Результат 2

key = 9; left = 0; right = 4; mid = (0+4)/2 = 2



Шаг 3) Если искомое значение больше значения элемента, находящегося посередине текущей части множества, то элемент с индексом mid+1 становится новой левой границей, иначе элемент с индексом mid-1 становится новой правой границей, и далее поиск будет продолжатся на выбранной половине текущей части множества. Затем необходимо вернуться к шагу 1.

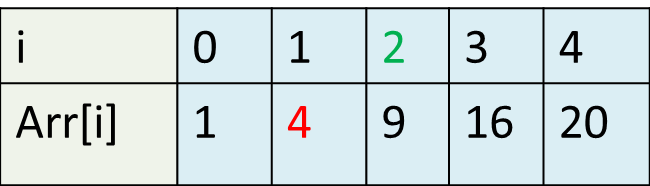
1. key = 4; left = 0; right = 4; mid = (0+4)/2 = 2
2. key = 4; left = 0; right = 2-1 = 1; mid = (0+1)/2 = 0
3. key = 4; left = 1; right = 1; mid = (1+1)/2 = 1

key > arr[mid]

left = mid +1

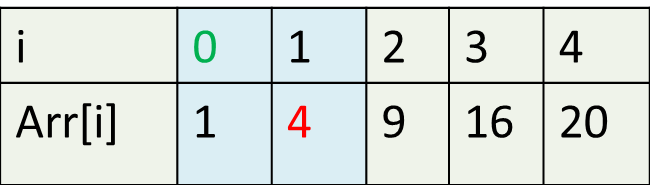
key = arr[mid]

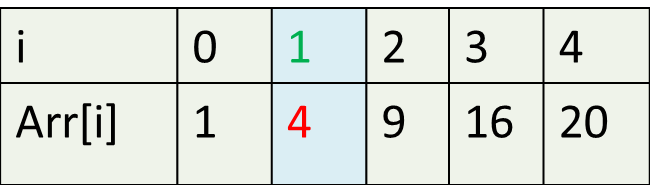
Результат 1



key < arr[mid]

right = mid -1





Один шаг бинарного поиска уменьшает число рассматриваемых элементов в 2 раза.

Если в множестве несколько элементов с искомым значением, то алгоритм найдет первый совпавший элемент и выведет его.

Например, в массиве a = {1, 3, 3, 3, 3, 3, 3, 9, 18} с ключом key = 3 совпадет элемент с индексом 4.

В процессе работы алгоритма бинарного поиска количество элементов, в которых осуществляется поиск, уменьшается в 2 раза. Поэтому достоинством алгоритма является его большая скорость поиска, по сравнению с последовательным поиском. Недостатком является то, что бинарный поиск работает только в отсортированном множестве.

Код алгоритма.

int binSearch(int \*arr, int size, int key){ //массив, размер массива и значение элемента индекс которого надо найти

bool flag = false; //найден ли элемент

int left = 0, right = size - 1, mid; //начальные значения границ

while (!flag && left <= right) { //пока элемент не найден или не просмотрен весь массив

mid = (left + right) / 2; //среднее элемент

if (key == arr[mid]) { //если это искомый элемент

flag = true; //элемент найден

}else if (arr[mid] < key) { //иначе смещение границы

left = mid + 1; //смещение левой границы

}

else{

right = mid - 1; //смещение правой границы

}

}

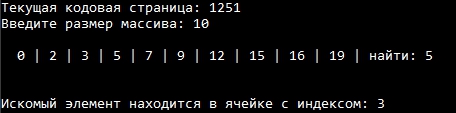
if (flag) //если элемент найден

return mid; //возвращаем индекс

else

return -1; //возвращаем -1 если не нашли

}



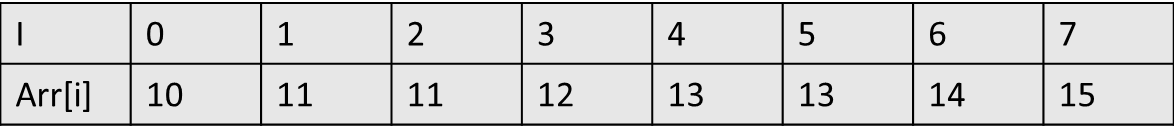
Алгоритм интерполяционного поиска

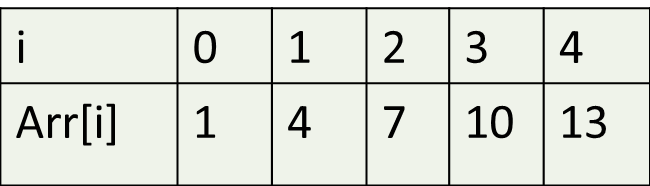
1)Интерполяционный поиск – это поиск заданного элемента на упорядоченном множестве, который осуществляется путем неоднократного деления этого множества на две части таким образом, что искомый элемент попадет в одну из этих частей. Поиск заканчивается нахождением искомого элемента, который является границей между частями множества или же при отсутствии искомого элемента.

2)Если элементы распределены **достаточно *равномерно***, то можно исключать из рассмотрения поиска на каждом шаге еще больше элементов, используя интерполяционный поиск.

3)Интерполяционный поиск основывается на том, что массив можно интерполировать в прямую без потери данных.

Оба массива подходят для интерполяционного поиска.

****

****

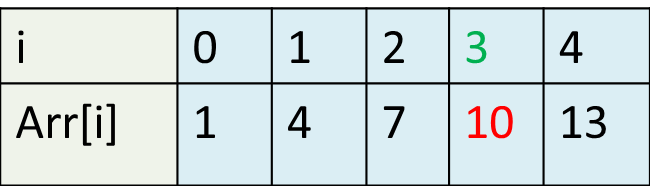
Шаг 1) Найдем mid, индекс элемента, который находится в заданных границах, по формуле.

Шаг 2) Если искомый элемент равен элементу с индексом **mid**, поиск завершается.

key = 10; left = 0; right = 4; mid = 0+(4-0)\*(10-1)/(13-1)= 3

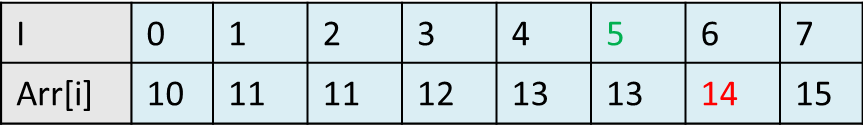
key = arr[mid]

Результат 3



Шаг 3) Если искомое значение больше значения элемента, находящегося по индексу mid, то элемент с индексом mid+1 становится новой левой границей, иначе элемент с индексом mid-1 становится новой правой границей, и далее поиск будет продолжатся на выбранной части. Затем необходимо вернуться к шагу 1.

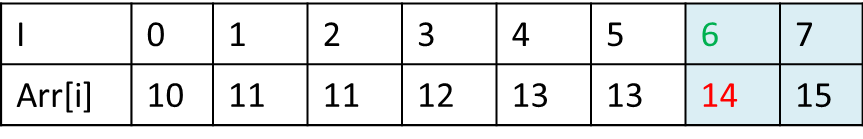
key = 14; left = 0; right = 7; mid = 0+(7-0)\*(14-10)/(15-10)= 5



key > arr[mid]

left = mid +1

key = 14; left = 6; right = 7; mid = 6+(7-6)\*(14-13)/(15-13)= 6



Результат 6.

int interpolationSearch(int \*arr, int size, int key) {//массив, размер массива и значение элемента индекс которого надо найти

bool flag = false; //найден ли элемент

int left = 0, right = size - 1, mid; //начальные значения границ

while (!flag && left <= right && arr[left] <= key && key <= arr[right]) {//пока элемент не найден или не просмотрен весь массив

mid = left + (key - arr[left]) \* (right - left) / (arr[right] - arr[left]);

if (key == arr[mid]) {//если это искомый элемент

flag = true; //элемент найден

}

else if (arr[mid] < key) {

left = mid + 1;

}else if (arr[mid] < key) { //иначе смещение границы

left = mid + 1; //смещение левой границы

}

else{

right = mid - 1; //смещение правой границы

}

}

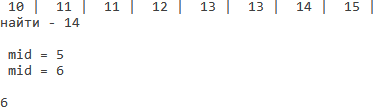
if (flag) //если элемент найден

return mid; //возвращаем индекс

else

return -1; //возвращаем -1 если не нашли

}



Алгоритмы поиска для данных символьного типа.  
Поиск подстроки в строке

1)Заданы строка **str**, которая содержит **strl** – символов, и подстрока **substr** , которая содержит **substrl** – символов.

2)Проверить, входит ли заданная подстрока в данную строку. Если входит, то найти **индекс символа, начиная с которого подстрока входит в строку.**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| str | A | B | C | A | F | D | F | A | B | C | A | B | D |
| substr |  |  |  |  |  |  |  | A | B | C | A | B | D |

Результат 7.

Прямой поиск подстроки в строке.

Основная идея алгоритма в заключается в последовательном сравнении строки с подстрокой. Вначале сравнивается первый символ строки с первым символом подстроки, второго символа строки со вторым символом подстроки и так далее. Если все символы совпали то строка найдена. Иначе происходит сдвиг строки на 1 позицию и сравнение начинается со второго символа строки с первым символом подстроки. Сдвиг происходит до тех пор пока подстрока не будет найдена или не дойдет до конца строки.

Достоинства – простая реализация.

Недостаток – длительное время работы.

Алгоритм.

i - счетчик индекс элемента в строке, изначально равен 0

J - счетчик индекс элемента в подстроке, изначально равен 0

1. Происходит сравнение **( j + i )-ого** символа строки с **j-м** символом подстроки.
2. Пока **j** меньше размера подстроки происходит последовательное сравнение символов.
3. Если какие-то символы не равны, то j = 0 , i += 1 и операции повторяются.

Если произошло совпадение всех символов, то фиксируется факт нахождения подстроки.

Строка ABBCABBC. Подстрока ABBC.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| str | A | B | B | C | A | B | B | C |
| subctr | A | B | B | C |  |  |  |  |

Все символы попарно равны.

Результат – 0.

Строка AGABBC. Подстрока ABBC.

* 1. ‘A’ = ‘A’, да => j += 1; // i = 0; j = 1;
  2. ‘G’ = ‘B’, нет => j = 0; i += 1; // i = 1; j = 0;
  3. ‘G’ = ‘A’, нет => j = 0; i += 1; // i = 2; j = 0;

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| str | A | G | A | B | B | C |
| subctr | A | B | B | C |  |  |
| subctr |  | A | B | B | C |  |
| subctr |  |  | A | B | B | C |

* 1. ‘A’ = ‘A’, да => j += 1; // i = 2; j = 1;
  2. ‘B’ = ‘B’, да => j += 1; // i = 2; j = 2;
  3. ‘B’ = ‘B’, да => j += 1; // i = 0; j = 3;
  4. ‘C’ = ‘C’, да => j += 1; // i = 2; j = 4;

Результат 2.

Код алгоритма.

int search(string str, string substr) { //строка и подстрока

int strl, substrl, res = -1;

strl = str.size(); substrl = substr.size(); //длина строки и подстроки

if (strl != 0 && substrl != 0){

for (int i = 0; i < strl - substrl +1; i++){ //Проход по строке

for (int j = 0; j < substrl; j++){ //Проход по подстроке

if (substr[j] != str[i + j]) {

break;

}else if(j == substrl - 1){ //Если это последний символ

res = i; //в подстроке

break; //то строка найдена

}

}

}

return res;

}

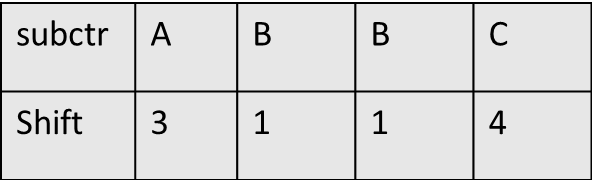
return -1;

}



Алгоритм Бойера и Мура для поиска подстроки в строке

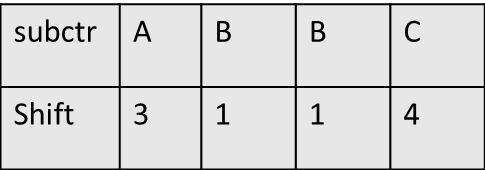
1. Перед поиском на основе подстроки формируется таблица переноса Shift:
2. Для каждого символа подстроки, кроме последнего, в таблицу Shift заносится **величина переноса,** то есть расстояние символа до символа, стоящего на правом конце подстроки
3. Если символ уже участвовал в расчете величины переноса, то в таблицу Shift заносится её ранее вычисленное значение.
4. Величина переноса для последнего символа равна длине строки.
5. Величина переноса для любого символа, не входящего в подстроку, равна длине строки. « \* »



Алгоритм.

Строка AGABBC. Подстрока ABBC

Таблица Shift

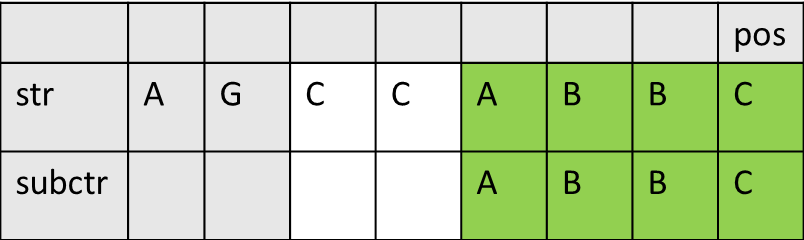


Рассматривается подстрока, начиная с её последнего символа.

1. Pos = (длина подстроки) -1 = 4 - 1 = 3;
2. Сравниваем с конца, ‘C’ и’ C’ равны.
3. Сравниваем ‘C’ и ‘B’, они не равны => сдвиг по таблице Shift для ‘C’



1. Pos += Shift[(int)(‘C’)] = 3+4 = 7;
2. Сравниваем символы, они все попарно равны



Результат = Pos – (длина подстроки) + 1 = 7-4+1 = 4

Код алгоритма.

int boyera\_mura(string str, string substr) {

int strl, substrl, res = -1;

strl = str.size(); substrl = substr.size();

if (strl != 0 && substrl != 0) {

int i, Pos;

int bias[256]; //массив смещения

for (i = 0; i < 256; i++) {

bias[i] = substrl; //присвоить каждому длину подстроки

}

for (i = substrl - 2; i >= 0; i--) { //Корректируем массив

if (bias[int((unsigned char)substr[i])] == substrl) {

bias[int((unsigned char)substr[i])] = substrl - i - 1;

}

}

Pos = substrl - 1; //позиция последнего символа подстроки относительно строки

while (Pos < strl) { //есть подстрока в строке

if (substr[substrl - 1] != str[Pos]) {//последний символ подстроки и символ стороки

Pos += bias[int((unsigned char)str[Pos])];//сдвиг по таблице

}

else {

for (i = substrl - 1; i >= 0; i--) { //проход по символам подстроки начиная с предпоследнего

if (substr[i] != str[Pos - substrl + i + 1]){ //символ подстроки и символ стороки

Pos += bias[int((unsigned char)str[Pos - substrl + i + 1])];

break;

}

else if (i == 0) { //если прошли всю подстроку

return Pos - substrl + 1; //считаем индекс

}

}

}

}

}

return res;

}

**ЗАДАНИЕ НЕ ДЛЯ ВЫПОЛНЕНИЯ!**

Постановка задачи:

Реализовать алгоритмы поиска, для заданного варианта используя теоретические сведения.

Варианты заданий.

1. Алгоритм бинарного поиска.
2. Прямой поиск подстроки в строке.
3. Алгоритм бинарного поиска.
4. Алгоритм линейного поиска.
5. Алгоритм бинарного поиска.
6. Алгоритм Бойера и Мура для поиска подстроки в строке.
7. Алгоритм линейного поиска.
8. Алгоритм интерполяционного поиска.
9. Прямой поиск подстроки в строке.
10. Алгоритм бинарного поиска.
11. Алгоритм бинарного поиска.
12. Алгоритм Бойера и Мура для поиска подстроки в строке.
13. Прямой поиск подстроки в строке.
14. Алгоритм бинарного поиска.
15. Алгоритм Бойера и Мура для поиска подстроки в строке.
16. Прямой поиск подстроки в строке.
17. Алгоритм линейного поиска.
18. Алгоритм интерполяционного поиска.
19. Алгоритм Бойера и Мура для поиска подстроки в строке.
20. Алгоритм интерполяционного поиска.