AJOU UNIV.
1973

HW 1 Report

*Sequential Search*와 *Binary Search*의 비교

자료구조 및 알고리즘 이해 (19-2, C070-1) / 이정원 교수님

201120696 최제헌

문제분석 (요구사항)

- Sequential Search와 Binary Search 알고리즘의 이해
- Binary Search 알고리즘 재귀적으로 구현
- 구현된 소스코드 분석 및 이해
- Sequential Search와 Binary Search 알고리즘 시간 측정 및 결과 비교

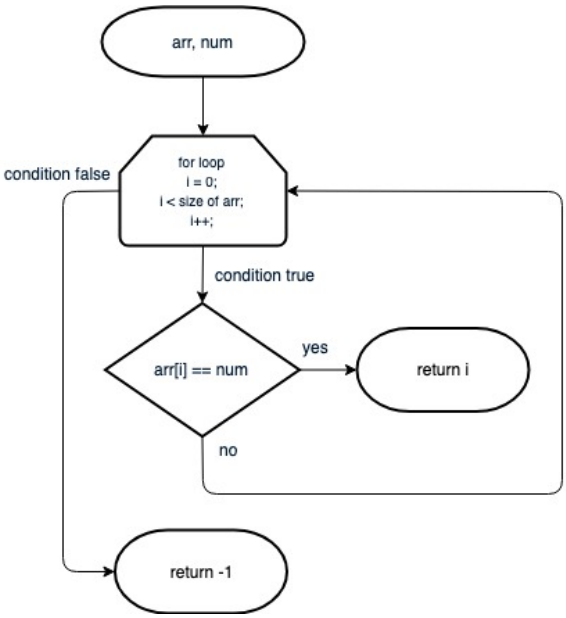
사용한 알고리즘의 설명

1. Sequential search (순차 탐색)

특징

- 배열에서 특정한 값을 찾는 방법 중 하나
- 앞에서부터 순서대로 탐색
- 정렬되지 않은 배열에서도 사용 가능

구현한 함수의 알고리즘 설명



구현된 함수는 int* arr, int num
2개의 입력 매개변수를 받는다.
arr: 탐색 값을 찾을 배열 / num: 탐색 값

for문으로 변수 i의 값을 0부터 arr 크기 - 1까지
1씩 증가시키며 반복문을 수행하도록 한다.
초기값: int i = 0 / 조건식: i < size of arr / 증감연산: i++

i번 인덱스의 arr 값이 num(탐색 값)과 동일하면
해당 i 인덱스 값을 반환하도록 한다.

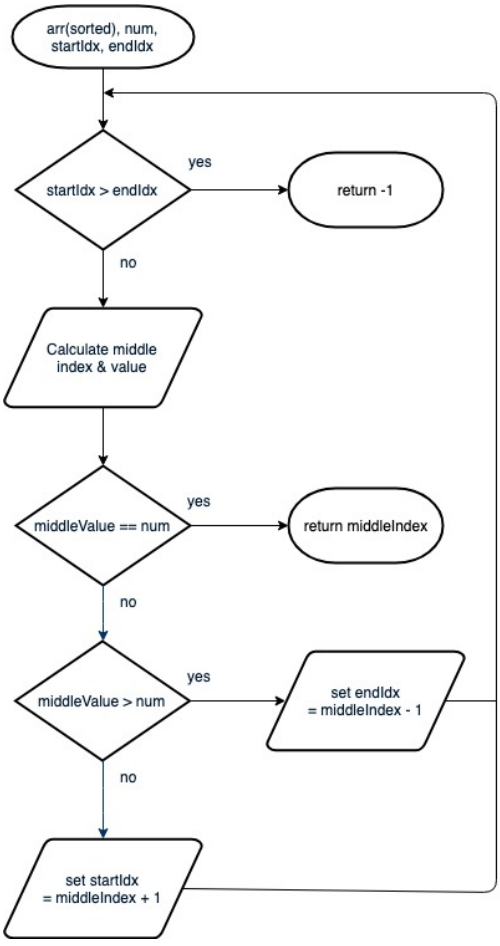
for문이 종료 됐는데도 index 값을 찾지 못하면
검색 값이 없는 것이므로 -1을 반환하도록 한다.

2. Binary search (이진 탐색)

특징

- 배열에서 특정한 값을 찾는 방법 중 하나
- 중간의 값을 임의로 선택하여, 그 값과 크고 작음을 비교
- 값의 비교를 통해 구간을 다시 설정
- 탐색할 때까지 위의 두가지 과정을 반복
- 정렬된 배열에 대해서만 사용 가능

구현한 함수의 알고리즘 설명



구현된 함수는 int* arr, int num, int startIdx, int endIdx 4개의 입력 매개변수를 받는다.
arr: 탐색 값을 찾을 **정렬 된** 배열 / num: 탐색 값
startIdx: 탐색 구간 처음 위치 / endIdx: 탐색 구간 마지막 위치

매개변수 startIdx가 endIdx 보다 더 클 경우 -1을 반환한다.
탐색 값이 주어진 배열 arr 안에 없는 경우다.
(재귀 호출이 진행될수록 startIdx 값은 증가하고, endIdx 값은 감소된다. startIdx가 endIdx 보다 +1 더 커질 때까지 재귀 호출이 되도록 알고리즘을 설계하였다.)

탐색 구간의 중간 위치(index)와 arr 내의 해당 위치 값을 middleIndex, middleValue 각각의 변수에 할당한다.

middleValue가 탐색 값과 같다면,
해당 index 값인 middleIndex를 반환한다.

middleValue가 탐색 값보다 크다면,
endIdx 변수 값을 middleIndex - 1로 할당하고

middleValue가 탐색 값보다 작다면,
startIdx 변수 값을 middleIndex + 1로 할당하여 탐색 범위를 줄인다.

줄여진 탐색 범위를 포함한 입력 매개변수로
구현된 함수를 다시 재귀 호출한다.

소스코드 분석

기존 제공된 코드

include Libraries

```
#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>
```

- stdio.h: printf 등 기본적인 i/o 함수 사용을 위해 포함
- stdlib.h: srand 등 난수 발생을 위해 포함
- Windows.h: usec 단위로 시간 측정을 위한 카운터 함수 포함

함수 프로토타입, 상수, 변수, 초기화

```
#define ARRAY_SIZE 100000000        //배열의 크기
#define MAX_DIFF 10                 //배열의 값 사이의 최대 차이

int sequentialSearch(int* arr, int num);
int binarySearch(int* arr, int num, int startIdx, int endIdx);

int main(void) {
    int itr;
    int arr[ARRAY_SIZE];
    int findNum;
    int idx;
    LARGE_INTEGER frequency, startTime, endTime;
    QueryPerformanceFrequency(&frequency);
```

상수는 주석 참조
순차 탐색, 이진 탐색 함수 프로토 타입 선언 (코드 상단에 있는 main에서, 하단에 있는 탐색 함수 호출을 위한 선언 과정)
변수
- itr: 난수 반복 생성에 사용되는 변수
- arr[ARRAY_SIZE]: 정렬된 임의의 값을 저장할 배열
- findNum: 탐색할 값, arr에 저장된 값들 중 임의의 값 하나
- idx: 탐색 결과 값을 받을 변수

그 외 시간 측정을 위한 frequency, startTime, endTime 변수 및 QueryPerformanceFrequency 함수 호출

정렬된 임의의 값 배열에 저장

```
    srand(time(NULL));
    arr[0] = (rand() % MAX_DIFF) + 1;

    for (itr = 1; itr < ARRAY_SIZE; itr++)
        arr[itr] = arr[itr - 1] + (rand() % MAX_DIFF) + 1;  //현재 값 = 이전 값 + (1 ~ MAX_DIFF) 사이의 값

    findNum = arr[rand() % ARRAY_SIZE];                     //찾고자 하는 값(임의로 설정)

    printf("배열 길이                  : %d\n", ARRAY_SIZE);
    printf("찾는 수                    : %d\n", findNum);
```

19번 라인
rand 함수를 통한 난수 발생을 하는데, 따로 seed 값을 지정하지 않으면 default 1이 할당되어 있어
항상 똑같은 패턴의 난수가 발생하게 된다.
난수 발생의 seed 값을 지정하는 srand 함수를 통해
시간에 따라 매번 달라지는 값인 time(null)을 seed로 할당해주어 매번 다른 난수가 발생하도록 하였다.
(time 함수 인자 값으로 null을 넘기면 1970년 1월 1일 0시 (UTC) 이후부터 현재까지 흐른 초 수를 리턴한다.)

20번 라인
난수 값에 MAX_DIFF(=10) 나머지 연산 후, + 1 을 하여 arr 첫번째 인덱스의 값을 1~MAX_DIFF 사이의 값을 할당하도록 하였다.

22~23번 라인
ARRAY_SIZE - 1 횟수 만큼 반복문을 실행하도록 하였고,
정렬된 임의의 값을 처음부터 할당하기 위해, 발생된 난수에 이전 인덱스의 값을 더하였다.

25~28번 라인
임의의 Index 탐색 값을 지정하고, 배열 길이 및 찾는 수를 콘솔에 print 하였다.

탐색 알고리즘 실행 및 시간 측정

```
29
30      QueryPerformanceCounter(&startTime);
31      idx = sequentialSearch(arr, findNum);                   //순차 탐색
32      QueryPerformanceCounter(&endTime);
33      printf("idx                         : %d\n", idx);
34      printf("Sequential Search 소요 시간 : %.6f\n", (double)(endTime.QuadPart - startTime.QuadPart) /
    (double)frequency.QuadPart);
35
36      QueryPerformanceCounter(&startTime);
37      idx = binarySearch(arr, findNum, 0, ARRAY_SIZE - 1);    //이진 탐색
38      QueryPerformanceCounter(&endTime);
39      printf("idx                         : %d\n", idx);
40      printf("Binary Search     소요 시간 : %.6f\n", (double)(endTime.QuadPart - startTime.QuadPart) /
    (double)frequency.QuadPart);
41
42      return 0;
43  }
44
```

순차 탐색, 이진 탐색 함수를 순차적으로 실행하며, 실행결과 및 측정된 시간을 콘솔에 print 하는 코드

순차 탐색 알고리즘 구현부

```
44
45  int sequentialSearch(int* arr, int num) {
46      int idx;
47
48      for (idx = 0; idx < ARRAY_SIZE; idx++) {
49          if (arr[idx] == num)
50              return idx;
51      }
52
53      return -1;
54  }
55
```

전달 받은 매개변수 arr 배열 크기만큼, 0번째 index부터 반복문을 실행하며, 탐색 값 num과 동일한 arr[index] 값이 있는지 찾는다.
동일한 값을 찾게되면, 그 값의 위치 index를 반환한다.
반복문이 종료되면 arr 배열 내에 num이 없다는 의미이므로 -1 값을 반환한다.

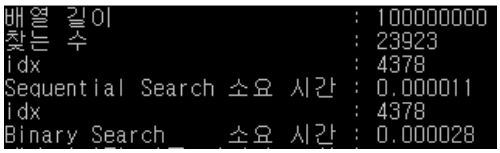
본인 작성 코드

이진 탐색 알고리즘 구현부

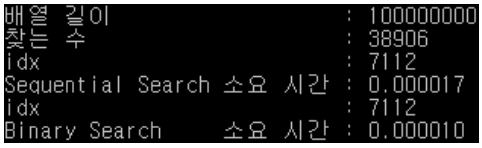
```
55
56  int binarySearch(int* arr, int num, int startIdx, int endIdx) {
57      if (startIdx > endIdx) return -1;
58
59      int idx = (startIdx + endIdx) / 2;
60      int idxValue = arr[idx];
61
62      if (idxValue == num) {
63          return idx;
64      }
65      else if (idxValue > num) {
66          endIdx = idx - 1;
67      }
68      else {
69          startIdx = idx + 1;
70      }
71
72      return binarySearch(arr, num, startIdx, endIdx);
73  }
```

재귀 사용 조건, 기본적으로 선언되어있는 함수 인터페이스에 맞춰 이진 탐색 알고리즘 함수를 구현하였다.
범위 내의 중간 위치의 값을 비교해가며, 범위를 줄여 재귀 호출을 반복하며 탐색을 수행한다.
이 재귀 함수의 경우 바닥조건은 57번 라인 (startIdx > endIdx)로 재귀호출을 반복할 수록 해당 조건으로 수렴하도록 되어있다.
또한, 범위 내의 중간 인덱스 선택은, 소수점 버림 방식으로 구현하였다.

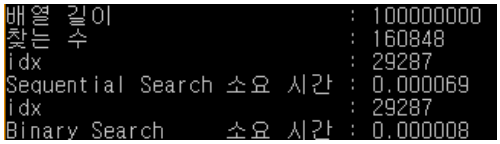
출력된 결과화면 캡쳐 및 결과에 대한 분석



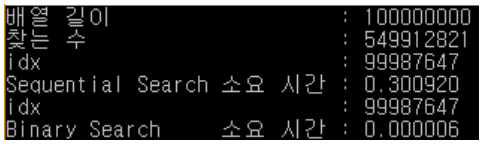
결과 1



결과 2



결과 3



결과 4

Sequential search의 경우 배열의 0번 index부터 n-1번 index까지 순차적으로 탐색을 진행하므로 찾는 index(or 찾는 수)가 커질 수록 오래걸리는 것을 확인할 수 있고,

Binary search의 경우 배열의 중간 index를 찾고 비교하는 것을 반복하는 방식이므로 찾는 index(or 찾는 수)의 크기에 큰 상관 없이 비교적 균등한 시간이 걸린 것을 확인할 수 있다.

이 결과를 통해 결과 1과 같이 배열의 초반 index의 value를 찾을 때 Sequential search가 Binary search 보다 더 빠르게 결과를 찾기도 하지만 전체적으로 시간 성능이 균등하며, 평균적으로 더 빠른 성능을 보이는 건 Binary search인 것을 확인할 수 있다.

시간 복잡도 및 수행 시간 분석

Sequential search (순차 탐색)

```
44
45 int sequentialSearch(int* arr, int num) {
46     int idx;
47
48     for (idx = 0; idx < ARRAY_SIZE; idx++) {
49         if (arr[idx] == num)
50             return idx;
51     }
52
53     return -1;
54 }
55
```

루프 제어 연산은 제외
n번의 비교 연산 (최대)
1번의 반환 연산 (*반환연산 둘 중 택 1)

1번의 반환 연산 (*반환연산 둘 중 택 1)

총 연산수= n + 1 (최대)

Time Complexity

Best: O(1) / Worst: O(n)

최선의 경우 0번 index에서 찾게 되는 것이므로 O(1)
최악의 경우 총 연산수이므로 n + 1 => O(n)

Binary search (이진 탐색)

```
56 int binarySearch(int* arr, int num, int startIdx, int endIdx) {
57     if (startIdx > endIdx) return -1;
58
59     int idx = (startIdx + endIdx) / 2;
60     int idxValue = arr[idx];
61
62     if (idxValue == num) {
63         return idx;
64     }
65     else if (idxValue > num) {
66         endIdx = idx - 1;
67     }
68     else {
69         startIdx = idx + 1;
70     }
71
72     return binarySearch(arr, num, startIdx, endIdx);
73 }
```

처음 입력된 개수 n

1번째 시행 후, 절반이 버려져서 => $\frac{n}{2}$

2번째 시행 후, 또 절반이 버려져서 => $\frac{1}{2} \times \frac{n}{2}$

3번째 시행 후, 또 절반이 버려져서 => $\frac{1}{2} \times \frac{1}{2} \times \frac{n}{2}$

…

k번째 시행 후 => $\frac{1}{2}^{k} \times n$

최악의 경우, 마지막까지 탐색한 경우이므로 남은 자료 1개 => $\frac{1}{2}^{k} \times n = 1$

=> $k = log_2 n$

Time Complexity

Best: O(1) / Worst: O(logn)

최선의 경우 첫번째 시행 n/2번 index에서 찾게 되는 것이므로 O(1)
최악의 경우 n번째 시행 O(logn)

비교 분석

Sequential search의 경우 시간복잡도 O(n)으로 선형
Binary search의 경우 시간복잡도 O(logn)으로 로그형을 나타낸다.

이는 입력의 개수(n)가 많아질 수록

Sequential search는 비례해서 연산 횟수가 늘어나고,
Binary search는 연산 횟수가 수렴하는 것을 의미한다.

Array_Size	1차	2차	3차
100	배열 길이 : 100 찾는 수 : 233 idx : 44 Sequential Search 소요 시간 : 0.000001 idx : 44 Binary Search 소요 시간 : 0.000001	배열 길이 : 100 찾는 수 : 31 idx : 3 Sequential Search 소요 시간 : 0.000000 idx : 3 Binary Search 소요 시간 : 0.000001	배열 길이 : 100 찾는 수 : 393 idx : 76 Sequential Search 소요 시간 : 0.000001 idx : 76 Binary Search 소요 시간 : 0.000002
10,000	배열 길이 : 10000 찾는 수 : 14992 idx : 2718 Sequential Search 소요 시간 : 0.000007 idx : 2718 Binary Search 소요 시간 : 0.000001	배열 길이 : 10000 찾는 수 : 21790 idx : 3929 Sequential Search 소요 시간 : 0.000009 idx : 3929 Binary Search 소요 시간 : 0.000002	배열 길이 : 10000 찾는 수 : 39043 idx : 7132 Sequential Search 소요 시간 : 0.000017 idx : 7132 Binary Search 소요 시간 : 0.000002
32,767	배열 길이 : 32767 찾는 수 : 83508 idx : 15205 Sequential Search 소요 시간 : 0.000051 idx : 15205 Binary Search 소요 시간 : 0.000001	배열 길이 : 32767 찾는 수 : 167961 idx : 30419 Sequential Search 소요 시간 : 0.000067 idx : 30419 Binary Search 소요 시간 : 0.000001	배열 길이 : 32767 찾는 수 : 57694 idx : 10525 Sequential Search 소요 시간 : 0.000024 idx : 10525 Binary Search 소요 시간 : 0.000003

* 소스코드에서 사용한 rand() 함수의 범위는 0~32767 (RAND_MAX: 0x7fff)이므로
Array_Size >= 32767은 동일한 범위의 난수 Index를 발생시키므로 32767 이하로 범위를 설정하였다.

위 Array_Size 변경에 따른 실행한 결과에도 나와있듯이, 찾는 index 크기가 커질수록 Sequential search는 비례해서 소요 시간이 늘어나는 반면, Binary search는 균등한 소요 시간이 걸리는 것을 확인할 수 있다.

‘Array_Size: 100, 2차’ 와 같이 초기 Index를 찾는 경우 Sequential search가 더 적은 소요 시간이 걸릴 수도 있지만, μs 단위로 근소한 차이를 보이는 만큼 Array_Size가 100 이하로 매우 작지 않은 이상 Sequential search를 사용하는 것이 더 좋은 선택지로 보인다.

추가적으로 Binary search가 반드시 정렬 된 배열을 이용해야하는 단점이 있기에 연속적인 정렬 연산까지 수행이 필요한 경우, 좋은 성능의 정렬 알고리즘을 사용했을시 시간복잡도는 O(nlogn)이 되어 조금 느려질 수 있다.

이와 다르게 Sequential search는 정렬이 필요 없고 모든 상황에 대해 O(n)을 나타내므로 모든 상황에 대해서 Binary search가 Sequential search 보다 좋은 선택지라는 것을 보장할 수는 없다. 때문에 각 상황에 맞춰 Sequential search, Binary search를 적절히 선택해 사용하는것이 바람직하다.

전체 소스코드

```c
#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>

#define ARRAY_SIZE 100000000        //배열의 크기
#define MAX_DIFF 10                 //배열의 값 사이의 최대 차이

int sequentialSearch(int* arr, int num);
int binarySearch(int* arr, int num, int startIdx, int endIdx);

int main(void) {
    int itr;
    int arr[ARRAY_SIZE];
    int findNum;
    int idx;
    LARGE_INTEGER frequency, startTime, endTime;
    QueryPerformanceFrequency(&frequency);

    srand(time(NULL));
    arr[0] = (rand() % MAX_DIFF) + 1;

    for (itr = 1; itr < ARRAY_SIZE; itr++)
        arr[itr] = arr[itr - 1] + (rand() % MAX_DIFF) + 1;  //현재 값 = 이전 값 + (1 ~ MAX_DIFF) 사이의 값

    findNum = arr[rand() % ARRAY_SIZE];                     //찾고자 하는 값(임의로 설정)

    printf("배열 길이                   : %d\n", ARRAY_SIZE);
    printf("찾는 수                     : %d\n", findNum);

    QueryPerformanceCounter(&startTime);
    idx = sequentialSearch(arr, findNum);                   //순차 탐색
    QueryPerformanceCounter(&endTime);
    printf("idx                         : %d\n", idx);
    printf("Sequential Search 소요 시간 : %.6f\n", (double)(endTime.QuadPart - startTime.QuadPart) /
(double)frequency.QuadPart);

    QueryPerformanceCounter(&startTime);
    idx = binarySearch(arr, findNum, 0, ARRAY_SIZE - 1);    //이진 탐색
    QueryPerformanceCounter(&endTime);
    printf("idx                         : %d\n", idx);
    printf("Binary Search     소요 시간 : %.6f\n", (double)(endTime.QuadPart - startTime.QuadPart) /
(double)frequency.QuadPart);

    return 0;
}

int sequentialSearch(int* arr, int num) {
    int idx;

    for (idx = 0; idx < ARRAY_SIZE; idx++) {
        if (arr[idx] == num)
            return idx;
    }

    return -1;
}

int binarySearch(int* arr, int num, int startIdx, int endIdx) {
    if (startIdx > endIdx) return -1;

    int idx = (startIdx + endIdx) / 2;
    int idxValue = arr[idx];

    if (idxValue == num) {
        return idx;
    }
    else if (idxValue > num) {
        endIdx = idx - 1;
    }
    else {
        startIdx = idx + 1;
    }

    return binarySearch(arr, num, startIdx, endIdx);
}
```