

CSC10004: Data Structure and Algorithms

Lecture 4: Search

Lecturer: Bùi Văn Thạch

TA: Ngô Đình Hy/Lê Thị Thu Hiền

{bvthach,ndhy}@fit.hcmus.edu.vn, lththien@hcmus.edu.vn

Course topics

1. Introduction
2. Algorithm complexity analysis
3. Recurrences
4. Search
5. Sorting
6. Stack and Queue
7. Linked list
8. Priority queue
9. Tree
 1. Binary search tree (BST)
 2. AVL tree
11. Graph
 1. Graph representation
 2. Graph search
12. Hashing
13. Algorithm designs
 1. Greedy algorithm
 2. Divide-and-Conquer
 3. Dynamic programming

Goals

1. To help students able to understand the basic of search algorithms.

Outline

1. Overview
2. Linear search
3. Linear search with sentinel
4. Binary search
5. Randomized search

Outline

1. Overview
2. Linear search
3. Linear search with sentinel
4. Binary search
5. Randomized search

Complexity

	Best	Worst	Average	Extra requirements
Linear search				
Linear search with sentinel				
Binary search				
Randomized search				

- Search problem
 - Look for an element inside an array and return its position

- Example:

- Given an array:

0	1	2	3	4	5	6	7
7	2	1	5	3	6	19	9

- Search **6**, output is 5
 - Search 55, output is -1 or 8

- Example:
 - More generally, given an array of students
[{1912340, “Le Thanh Xuan”},
{1813242, “Nguyen Thai Hoang”},
{1923422, “Hoang Xuan Phuc”}, ...]
 - Search student with ID of 1813242
 - Output is {1813242, “Nguyen Thai Hoang”}

- A basic programming technique necessary for many scenarios, for example:
 - Check if an array contains a certain element
 - Search for info of a student with a certain ID

Outline

1. Overview
2. Linear search
3. Linear search with sentinel
4. Binary search
5. Randomized search

- Input: an array A with a size of n and a key k
- Output: position of found element
 - For non-existing element, position could be -1 or n
- Idea:
 - Loop through the array from the beginning to the end
 - Check if the i^{th} element match k
 - Return i if matched
 - Otherwise, return -1 or n

Example

	0	1	2	3	4	5	6	7	
A	7	2	1	5	3	6	19	9	$n = 8, k = 6$
i=0	7	2	1	5	3	6	19	9	$7 == 6, \text{ false}$
i=1	7	2	1	5	3	6	19	9	$2 == 6, \text{ false}$
i=2	7	2	1	5	3	6	19	9	$1 == 6, \text{ false}$
i=3	7	2	1	5	3	6	19	9	$5 == 6, \text{ false}$
i=4	7	2	1	5	3	6	19	9	$3 == 6, \text{ false}$
i=5	7	2	1	5	3	6	19	9	$6 == 6, \text{ TRUE}$

Found!

Source code

```
int linearSearch(int a[], int n, int k)
{
    int position = -1;
    for(int i=0; i<n; i++)
    {
        if(a[i] == k)
        {
            position = i;
            break;
        }
    }
    return position;
}
```

Analysis

- Complexity

7	2	1	5	3	6	19	9
---	---	---	---	---	---	----	---

- Best case: $O(1)$
- Worst case: $O(n)$
- Average: $O(n)$

- Best: $k = 7$
 - $O(1)$
- Worst: $k = 15$ or $k = 200$
 - $O(n)$
- Average:
 - $\Pr(k = a[i]) = \frac{1}{n+1}, i = 0, 1, \dots, n-1$ and $\Pr(k \neq a[i]) = \frac{1}{n+1}$
 - $\mathbb{E}[T] = \sum_{i=1}^{n+1} i \Pr(k = a[i]) = \frac{1}{n+1} \sum_{i=1}^n i + 1 = \frac{n+2}{2}$
 - Complexity: $O(n)$

Complexity

	Best	Worst	Average	Extra requirements
Linear search	$O(1)$	$O(n)$	$O(n)$	NO
Linear search with sentinel				
Binary search				
Randomized search				


Outline

1. Overview
2. Linear search
3. Linear search with sentinel
4. Binary search
5. Randomized search

- In linear search, we perform two checks for each iteration
 - One for loop
 - The other for key matching
- Can we do better?

```
int linearSearch(int a[], int n, int k)
{
    int position = -1;
    for(int i=0; i<n; i++)
    {
        if(a[i] == k)
        {
            position = i;
            break;
        }
    }
    return position;
}
```

- Idea: remove one check in each iteration
 - Add a fake element, namely, sentinel, equal to the key **k** at the end of array
 - Loop through the array until reaching **k**
- When reaching **k**, we need to identify if it is fake
 - If **k** is at the end, it is fake or **k** does not exist in the array
 - Otherwise, it is a real element in the array



7	2	1	5	3	6	19	9	6
---	---	---	---	---	---	----	---	---

Source code

```
int sentinelLinearSearch(int a[], int n, int k)
{
    int i = 0;
    for(; a[i] != k; i++)
        ;
    if (i < n)
        return i;
    return -1;
}
```

```
int linearSearch(int a[], int n, int k)
{
    int position = -1;
    for(int i=0; i<n; i++)
    {
        if(a[i] == k)
        {
            position = i;
            break;
        }
    }
    return position;
}
```

Analysis

- Complexity
 - Best case: $O(1)$
 - Worst case: $O(n)$
 - Average: $O(n)$
- Advantage
 - In practice, it is about **25%** outperforming regular linear search
- Disadvantage
 - We need **an extra slot** at the end of the array for the sentinel

Complexity

	Best	Worst	Average	Extra requirements
Linear search	$O(1)$	$O(n)$	$O(n)$	NO
Linear search with sentinel	$O(1)$	$O(n)$	$O(n)$	an extra slot at the end of the array for the sentinel
Binary search				
Randomized search				

Outline

1. Overview
2. Linear search
3. Linear search with sentinel
4. Binary search
5. Randomized search

- Linear search checks all the elements, hence the search space is large
- In case of sorted arrays, with a given key, we can narrow down the search space

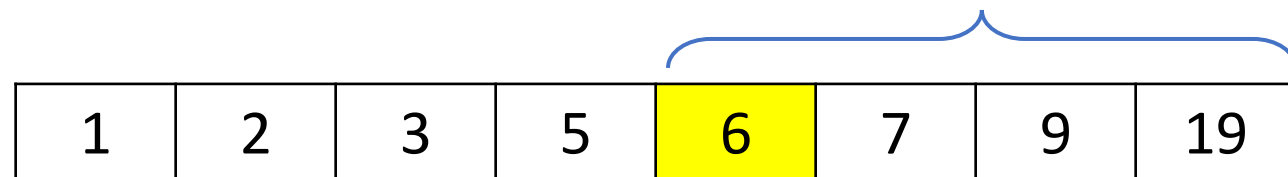
1	2	3	5	6	7	9	19
---	---	---	---	---	---	---	----

k=6

Search in the right!

- **Input:** an array **A** with a size of **n** and a key **k**

- **Output:** position of found element



1	2	3	5	6	7	9	19
---	---	---	---	---	---	---	----

- **Idea:**

$k = 6$

- Split the input array into two parts
- Compare **k** with the middle element
 - If matched, **k** is found
 - Otherwise, if **k** is smaller than the middle element, search in the left
 - Otherwise, search in the right

Example

key $k = 60$

$$i = \frac{7}{2} = 3$$

$$i = \frac{7}{2} + \frac{4}{2} = 5$$

$$i = \frac{7}{2} = 3$$

0	1	2	3	4	5	6	7
7	2	1	5	3	6	19	9

1	2	3	5	6	7	9	19
---	---	---	---	---	---	---	----

1	2	3	5	6	7	9	19
---	---	---	---	---	---	---	----

1	2	3	5	6	7	9	19
---	---	---	---	---	---	---	----

Found!

Source code: **recursive** programming

```
int binarySearch(int a[], int left, int right, int k)
{
    if(left > right)
        return -1;
    int mid = (left + right)/2;
    if(a[mid] == k)
        return mid;
    if(k > a[mid])
        return binarySearch(a, mid+1, right, k);
    return binarySearch(a, left, mid - 1, k);
}
```

Source code: **Non-recursive** programming

```
int binarySearch(int a[], int n, int k)
{
    int result = -1, left = 0, right = n - 1;
    while(left <= right)
    {
        int mid = (left + right)/2;
        if(a[mid] == k)
        {
            result = mid;
            break;
        }
        if(k > a[mid])
            left = mid + 1;
        else
            right = mid - 1;
    }
    return result;
}
```

Analysis

- Complexity
 - Best case: $O(1)$
 - Worst case: $O(\log_2 n)$
 - Average: $O(\log_2 n)$
- Disadvantage
 - Input array needs to be sorted beforehand

Complexity

	Best	Worst	Average	Extra requirements
Linear search	$O(1)$	$O(n)$	$O(n)$	NO
Linear search with sentinel	$O(1)$	$O(n)$	$O(n)$	an extra slot at the end of the array for the sentinel
Binary search	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$	Input array needs to be sorted beforehand
Randomized search				

Outline

1. Overview
2. Linear search
3. Linear search with sentinel
4. Binary search
5. Randomized search

Idea

1. Set a counter to be zero.
2. Randomly generate an index $i \in [0, n - 1]$.
3. Check whether $a[i] == \text{key}$.
 - If YES, return i .
 - If NO, increase the counter by 1 and repeat **Step 2**.
4. Return -1 (not found) after repeating $n^{0.9} + 1$ times, i.e., $\text{count} = n^{0.9}$.

Programming

```
int randomizedSearch(int a[], int n, int k)
{
    int i = rand(1, n);
    int count = 0;

    while (count < pow(n, 0.9)) {
        if (a[i] == k)
            return i;
        else
            i = rand(1, n);
        count++;
    }
    return -1;
}
```

Why do we use rand() even we know there is a possibility that the same value could appear twice?

Analysis: best and worst

- Best: $O(1)$.
- Worst: $O(n^{0.9})$ to return the found index or -1.

Analysis: average case (1/2)

- Suppose that there are m keys in the array.
 - $\rightarrow m$ indices in the array that make $(a[i] == k)$ true.
 - \rightarrow The probability of getting i such that $(a[i] == k)$ by calling $i = \text{rand}(1, n)$ is $p = m/n$.

- The probability that after c times, the key is found is

$$(1 - p)^{c-1} p = \left(1 - \frac{m}{n}\right)^{c-1} \cdot \frac{m}{n}$$

- Since this is the geometric distribution, the expectation to hit the key in the array is:

$$\frac{1}{p} = \frac{n}{m}$$

Analysis: average case (2/2)

- The probability that after count times, the key has not been found is

$$(1 - p)^{\text{count}} = \left(1 - \frac{m}{n}\right)^{\text{count}} \leq \exp\left(-\frac{m}{n} \times \text{count}\right)$$

- Substitute $\text{count} = n^{0.9}$, we get

$$(1 - p)^{\text{count}} \leq \exp\left(-\frac{m}{n} \times n^{0.9}\right) = \exp(-mn^{-0.1})$$

- The probability the key is found after count times is at least

$$1 - \exp(-mn^{-0.1})$$

- $m = 1, n = 100, \text{count} = 63 \Rightarrow \geq 0.4$.

Complexity

	Best	Worst	Average	Extra requirements
Linear search	$O(1)$	$O(n)$	$O(n)$	NO
Linear search with sentinel	$O(1)$	$O(n)$	$O(n)$	an extra slot at the end of the array for the sentinel
Binary search	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$	Input array needs to be sorted beforehand
Randomized search	$O(1)$	$O(n^{0.9})$	$O\left(\frac{n}{m}\right)$	<ol style="list-style-type: none"> 1. m is the number of keys in the array 2. Probabilistic method

Summary

- Linear search loops through an array from the beginning to the end for searching
- Linear search is slow with large arrays
- Binary search takes the advantage of arrays being sorted to narrow down searching space.
- Binary search significantly outperforms linear search with large arrays
 - For example, with one billion elements, binary search tries 30 times at maximum
- Randomized algorithms seems better than linear search.

Q & A