

Search on Vectors

Linear search, binary search, sorting and a first look at algorithm cost

Professors de Pro1

Department of Computer Science

Contents

- 1 The Search Problem
- 2 Linear Search
- 3 A First Look at Algorithm Cost
- 4 Binary Search
- 5 Sorting with `sort()`
- 6 When Is It Worth Sorting?
- 7 Proposed Exercises

The search problem

Problem

Given a vector v and a value x , find a position i such that $v[i] == x$, or report that x is **not** in v .

Conventions for this chapter

- v is an **integer vector**: `vector<int>`.
- n is `v.size()`.
- When x is not in v , we return `-1` (it is never a valid index).

Key question of this chapter

How **fast** can we search? And how do we **measure** it?

Linear (sequential) search

Idea: look at the items one by one, from left to right. It is the same **search scheme** we saw in IP06 for sequences.

```
1 //pre: ---
2 //post: returns index i such that v[i] is x
3 //      when x is in v
4 //      returns -1 when x is not in v
5 int search(const vector<int>& v, int x) {
6     int n = v.size();
7     int i = 0;
8     bool found = false;
9     // Inv: x is not in v[0..i-1]
10    //      found => v[i] is x
11    while (not found and i < n) {
12        if (v[i] == x) found = true;
13        else ++i;
14    }
15    if (found) return i;
16    else return -1;
17 }
```

Example: $x = 29$

35	11	52	29	17	44
0	1	2	3	4	5

Tests: 35, 11, 52, 29 ✓

⇒ returns 3 after 4 tests

Linear search: a compact version

We can put the test `v[i] != x` directly in the loop condition:

```
1 int search(const vector<int>& v, int x) {  
2     int n = v.size();  
3     int i = 0;  
4     // Inv: x is not in v[0..i-1]  
5     while (i < n and v[i] != x) ++i;  
6     if (i < n) return i;  
7     else return -1;  
8 }
```

Short-circuit evaluation

In `A and B`, if `A` is false then `B` is **not evaluated**. When `i == n`, `v[i]` is never accessed.

The order of the conditions matters!

`while (v[i] != x and i < n)` is **wrong**: when `x` is not in `v`, it accesses `v[n]`, which is out of range.

How many tests does linear search do?

The running time is proportional to the number of tests $v[i] == x$ that are done.

Case	Situation	Tests
Best case	x is at position 0	1
Worst case	x is at the end, or not in v	n

Conclusion

In the worst case, the number of tests is $\approx n$.

We say that linear search is an $O(n)$ **algorithm**.

What does $O(n)$ mean exactly? Let us take a short detour...

Measuring the cost of an algorithm

Time complexity

How the **running time** grows as the **input size** n increases.

Space complexity

How much **memory** the algorithm uses.

Why is it important?

- With 10 numbers, **any** algorithm is quick.
- With 10 million numbers, some algorithms take **much longer** than others.
- Complexity analysis helps us **predict** how a program scales with the size of the input and **choose** the most efficient algorithm.

In this course we focus on **time complexity**.

Big O notation

Idea

Big O describes the **rate of growth** of the running time.

- We count **steps** (e.g., comparisons), not seconds.
- Keep the **fastest-growing term** and ignore constant factors and lower-order terms.
- It is **machine independent**: it ignores hardware details.

Number of steps	Big O	Name
5	$O(1)$	constant
$3 \log n + 10$	$O(\log n)$	logarithmic
$3n + 7$	$O(n)$	linear
$n \log n + 5n$	$O(n \log n)$	linearithmic
$4n^2 + 3n + 10$	$O(n^2)$	quadratic

Common complexity classes

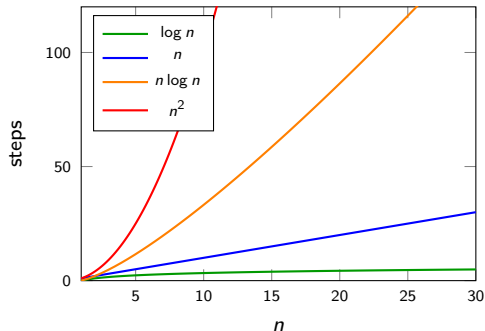
Notation	Growth	Example
$O(1)$	same time, whatever n is	accessing $v[i]$
$O(\log n)$	grows very slowly	binary search (<i>today!</i>)
$O(n)$	grows in direct proportion to n	linear search
$O(n \log n)$	slightly faster than linear	<code>sort()</code> (<i>today!</i>)
$O(n^2)$	grows quickly	two nested loops over v
$O(2^n)$	explodes	trying all subsets

Note

Here $\log n$ means $\log_2 n$: how many times we can halve n until we reach 1.

Big O: a numeric view

n	$\log n$	n	$n \log n$	n^2
10	3.3	10	33	100
100	6.6	100	664	10^4
1,000	10	1,000	9,970	10^6
10,000	13.3	10,000	132,900	10^8
100,000	16.6	100,000	$1.7 \cdot 10^6$	10^{10}
10^6	19.9	10^6	$2 \cdot 10^7$	10^{12}



Observations

For large n , $O(n^2)$ is practically unusable, while $O(n \log n)$ is still manageable. $O(\log n)$ is extremely efficient: a million inputs take **fewer than 20 steps**.

Binary search: the idea

Precondition

The input vector is **sorted** (from the smallest to the greatest).

Main idea

When we look for a word in a dictionary, we **do not proceed sequentially!** We open it in the middle and discard one half.

If v is sorted:

- If $x > v[i]$, then x is **not** in $v[0..i]$.
- If $x < v[i]$, then x is **not** in $v[i..n-1]$.

The running time can be **drastically reduced** if the vector is sorted.

Binary search: code

```
1 //pre: v is sorted (from least to greatest)
2 //post: returns index i such that v[i] is x when x is in v
3 //      returns -1 when x is not in v
4 int binary_search(const vector<int>& v, int x) {
5     int left = 0;
6     int right = v.size() - 1;
7     // Inv: if x is in v, it is in v[left..right] (the "active" area)
8     while (left < right) {
9         int mid = (left + right)/2;
10        // mid is in the active area: left <= mid < right
11        if (x > v[mid]) left = mid + 1;    // rule out v[left..mid]
12        else right = mid;                 // rule out v[mid+1..right]
13    }
14    // left > right: v is empty; left == right: v[left] is the only candidate
15    if (left == right and v[left] == x) return left;
16    else return -1;
17 }
```

left: left limit, right: right limit, mid: middle position of the active area.

Binary search: execution trace for $x = 90$

1 left=0, right=18, mid=9: $90 > 52$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Binary search: execution trace for $x = 90$

1 left=0, right=18, mid=9: $90 > 52$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

2 left=10, right=18, mid=14: $90 > 85$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Binary search: execution trace for $x = 90$

1 left=0, right=18, mid=9: $90 > 52$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

2 left=10, right=18, mid=14: $90 > 85$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

3 left=15, right=18, mid=16: $90 \leq 91$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Binary search: execution trace for $x = 90$

1 left=0, right=18, mid=9: $90 > 52$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

2 left=10, right=18, mid=14: $90 > 85$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

3 left=15, right=18, mid=16: $90 \leq 91$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

4 left=15, right=16, mid=15: $90 > 89$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Binary search: execution trace for $x = 90$

1 left=0, right=18, mid=9: $90 > 52$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

2 left=10, right=18, mid=14: $90 > 85$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

3 left=15, right=18, mid=16: $90 \leq 91$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

4 left=15, right=16, mid=15: $90 > 89$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

5 left=16, right=16: $90 \neq 91$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Binary search: execution trace for $x = 90$

1 left=0, right=18, mid=9: $90 > 52$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

2 left=10, right=18, mid=14: $90 > 85$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

3 left=15, right=18, mid=16: $90 \leq 91$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

4 left=15, right=16, mid=15: $90 > 89$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

5 left=16, right=16: $90 \neq 91$

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Result

left < right no longer holds: only $v[16]$ is left, it is not 90, and the function returns -1 after only **5 tests** (linear search would need 19).

Exercise: your turn

Exercise

Write the execution trace of `binary_search` with the same vector and `x = 17`.

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Exercise: your turn

Exercise

Write the execution trace of `binary_search` with the same vector and `x = 17`.

11	17	24	29	30	35	44	48	51	52	57	63	67	71	85	89	91	94	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Solution

```
1 left=0, right=18, mid=9  17 ≤ 52  right = 9
2 left=0, right=9, mid=4   17 ≤ 30  right = 4
3 left=0, right=4, mid=2   17 ≤ 24  right = 2
4 left=0, right=2, mid=1   17 ≤ 17  right = 1
5 left=0, right=1, mid=0   17 > 11  left = 1
6 left=1, right=1          17 = 17  returns 1
```

How many tests does binary search do?

Each iteration **halves** the size of the active area:

$$n \longrightarrow \frac{n}{2} \longrightarrow \frac{n}{4} \longrightarrow \dots \longrightarrow 1$$

The number of halvings until we reach 1 is $\approx \log n$.

Conclusion

Running time $\approx \log n$ tests: binary search is an $O(\log n)$ **algorithm**.

n	$\log n$
100	7
1,000	10
10,000	14
100,000	17
1,000,000	20
10,000,000	24

Linear vs. binary

Searching in 10 million sorted items:

linear: up to 10,000,000 tests

binary: at most 24 tests

Recursive version of binary search

The local variables `left` and `right` (limits of the active area) become **parameters**, so the recursive calls can change them.

```
1 //pre:  v is sorted (from least to greatest) and left, right are valid indexes of v
2 //post: returns i such that left <= i <= right and v[i] is x
3 //      when x is in v[left..right]
4 //      returns -1 when x is not in v[left..right]
5 int rec_binary_search(const vector<int>& v, int x, int left, int right) {
6     if (left > right) return -1;           // direct case: empty area
7     else {
8         int mid = (left + right)/2;
9         // left <= mid <= right
10        if (x > v[mid]) return rec_binary_search(v, x, mid + 1, right);
11        else if (x < v[mid]) return rec_binary_search(v, x, left, mid - 1);
12        else return mid;                  // direct case: found
13    }
14 }
```

Initial call

```
rec_binary_search(v, x, 0, v.size() - 1)
```

Sorting a vector with `sort()`

Sorting may be useful or even **unavoidable**: for instance, binary search needs a sorted vector.

Step 1: include the library

```
#include <algorithm>
```

Step 2: sort `v` from smallest to greatest

```
sort(v.begin(), v.end());
```

After the call, `v` contains the **very same values**, now ordered from smallest to greatest.

`sort()` is an **efficient** sorting function implemented in C++ and available through the library `algorithm`.

sort(): examples

Integer vector

Before:

85	23	34	32	11	21	83	67
0	1	2	3	4	5	6	7

After:

11	21	23	32	34	67	83	85
0	1	2	3	4	5	6	7

String vector

```
w = "joan" "anna" "marta" "pere" "pau" "isabel" "tom"
```

```
After sort(w.begin(), w.end());
```

```
w = "anna" "isabel" "joan" "marta" "pau" "pere" "tom"
```

Strings are sorted **alphabetically** (lexicographically): it is their default order.

A complete program (1/2): reading and writing procedures

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 using namespace std;
5
6 //pre:  there are at least u.size() integers on the input channel
7 //post: u stores the first u.size() integers from cin
8 void read_vector(vector<int>& u) {
9     int n = u.size();
10    for (int i = 0; i < n; ++i) cin >> u[i];
11 }
12
13 //pre:  u.size() > 0
14 //post: writes the values of u on the output channel, separated by spaces
15 void write_vector(const vector<int>& u) {
16     int n = u.size();
17     // first element treated specially because of the spaces
18     // separating output integers
19     cout << u[0];
20     for (int i = 1; i < n; ++i) cout << ' ' << u[i];
21     cout << endl;
22 }
```

A complete program (2/2): main

```
1 // ... includes, read_vector and write_vector ...
2
3 //in:  an integer n followed by n integers on the input channel
4 //out: writes the n integers sorted from smallest to greatest
5 int main() {
6     int n;
7     cin >> n;
8     vector<int> v(n);
9     read_vector(v);
10    sort(v.begin(), v.end());
11    write_vector(v);
12 }
```

Example

Input: 5 42 7 19 3 25 **Output:** 3 7 19 25 42

Replace the sort line by `sort(v.begin(), v.end(), cmp);` to use a custom sorting criterion (next slides).

Custom sorting criteria

What if we need an order **different** from the usual one? We add a **third argument**:

```
sort(v.begin(), v.end(), cmp);
```

cmp is the name of a function (**we write it**) that says how two items are compared in the new order. If v is a vector of type Type:

```
1 //pre:  ---
2 //post: returns true if x is before y in the new order
3 //      returns false otherwise
4 bool cmp(Type x, Type y)
```

The name of the function is arbitrary; we use cmp as an example.

Careful

We pass the **name** cmp, without parentheses: sort will call it as many times as it needs.

Custom criteria: from greatest to smallest

```
1 //pre:  ---
2 //post: returns true when x is before y (that is, when x > y)
3 //      returns false otherwise
4 bool cmp(int x, int y) {
5     return x > y;
6 }
```

```
sort(v.begin(), v.end(), cmp);    // only change in the main program
```

Example

Before:

85	23	34	32	11	21	83	67
0	1	2	3	4	5	6	7

After:

85	83	67	34	32	23	21	11
0	1	2	3	4	5	6	7

Custom criteria: shortest strings first

Shortest strings first; strings of the **same length** in alphabetical order:

```
1 //pre:  ---
2 //post: returns true when either s1 is shorter than s2, or s1 and s2
3 //      have the same length but s1 is before s2 in alphabetical order
4 //      returns false otherwise
5 bool cmp(const string& s1, const string& s2) {
6     int n = s1.length();
7     int m = s2.length();
8     if (n == m) return s1 < s2;
9     else return n < m;
10 }
```

Note: we pass the two strings by constant reference, to avoid multiple copies on local variables.

Example

```
w = "joan" "anna" "marta" "pere" "pau" "isabel" "tom"
```

```
After sort(w.begin(), w.end(), cmp);
```

```
w = "pau" "tom" "anna" "joan" "pere" "marta" "isabel"
```

How much does `sort()` cost?

Let n be the size of the vector we want to sort.

Cost of `sort()`

Its running time is proportional to $\approx n \log n$. It is an $O(n \log n)$ **algorithm**: more than a linear search, much less than $O(n^2)$.

`sort()` combines a variety of sorting methods (e.g., quicksort, heapsort and insertion sort) and, since C++11, the standard guarantees $O(n \log n)$ comparisons even in the worst case.

Algorithm	Cost	Needs sorted vector?
Linear search	$O(n)$	no
Binary search	$O(\log n)$	yes
<code>sort()</code>	$O(n \log n)$	not applicable

So... is it worth sorting the vector **before** searching?

Is it worth sorting?

Nonsense 1: sorting to find the minimum

Sorting ($O(n \log n)$) **wastes resources**: we can compute the minimum in linear time $O(n)$.

Nonsense 2: sorting to do only a few searches

Sorting just to do **one or a few** binary searches instead of linear searches does **not** pay off.

Good idea: sorting to do many searches

Sorting to do **many** binary searches makes a lot of sense. When n is big, it is nonsense **not** to sort!

Some numbers

We have to do k searches in a vector of size n :

Option A: k linear searches

$$\approx k \cdot n$$

Option B: sort, then k binary searches

$$\approx n \log n + k \log n$$

$n = 10^6$, $\log n \approx 20$, worst case			
k	Option A	Option B	Better
1	$\approx 10^6$	$\approx 2 \cdot 10^7$	A
10	$\approx 10^7$	$\approx 2 \cdot 10^7$	A
1,000	$\approx 10^9$	$\approx 2 \cdot 10^7$	B (50 times faster!)

When k is small, option A is better. When k is big enough (for instance, $k = \sqrt{n}$), option B is better.

Proposed exercises

- 1 `last_position(v, x)`: like `search`, but returns the **last** position of `x` in `v`. What is its cost?
- 2 `is_sorted(v)`: checks whether `v` is sorted from smallest to greatest. Can you do it in $O(n)$?
- 3 `first_not_smaller(v, x)`: for a sorted `v`, returns the first position `i` such that `v[i] >= x` (or `n` if there is none), using the idea of binary search.
- 4 Sort a vector of strings from **longest to shortest**, with ties in alphabetical order.
- 5 Give the Big O of: $7n + 3 \log n + 100$ and $2n^2 + 50n$.
- 6 What is the cost of the “number of different letters” program from the previous chapter?
Hint: two nested loops.

Search on vectors on one slide

- **Linear search:** any vector, $O(n)$.
- **Binary search:** **sorted** vectors only, $O(\log n)$. It halves the active area `v[left..right]` at each step.
- `sort(v.begin(), v.end())` sorts in $O(n \log n)$.
- `sort(v.begin(), v.end(), cmp)`: add an auxiliary function `cmp` for a custom order.
- **Big O** measures how the running time **grows** with n : keep the dominant term, ignore constants.

Sort once, search many times!

Questions?

Search on Vectors: linear search, binary search and sorting in C++