Introduction to Programming (in C++)

Vectors

Jordi Cortadella, Ricard Gavaldà, Fernando Orejas
Dept. of Computer Science, UPC

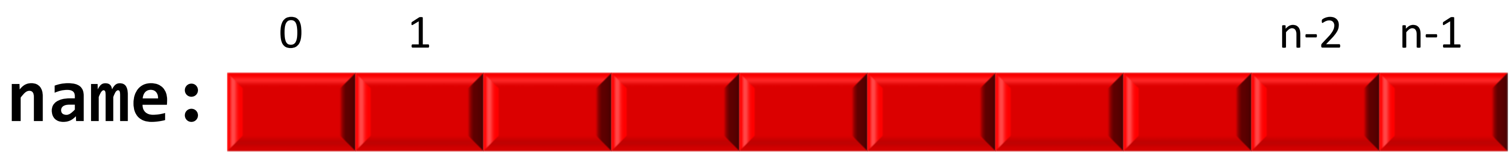
Outline

- Vectors
- Searching in vectors

Vectors

- A vector is a data structure that groups values of the same type under the same name.

- Declaration: `vector<type> name(n);`



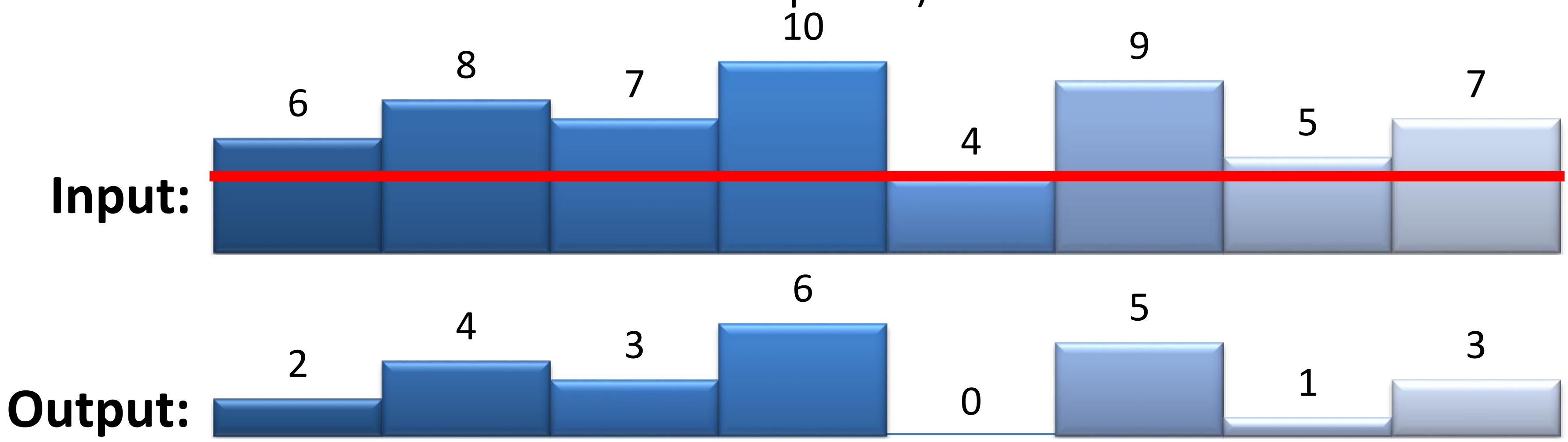
- A vector contains *n* elements of the same type (*n* can be any expression).

- `name[i]` refers to the *i*-th element of the vector (*i* can also be any expression)

- Note: use **`#include<vector>`** in the program

Normalizing a sequence

- Write a program that normalizes a sequence (i.e. subtracts the minimum value from all the elements in the sequence)



- The input and output sequences will be preceded by the number of elements in the sequence.

```
Input:      8   6   8   7  10   4   9   5   7
Output:     8   2   4   3   6   0   5   1   3
```

- The program cannot read the sequence more than once.

Normalizing a sequence

```
// Input:  a sequence of numbers preceded by the length of the
//         sequence (there is at least one element in the sequence)
// Output: the normalized input sequence (subtracting the minimum
//         element from each element in the sequence)
int main() {
    int n;
    cin >> n;
    // Store the sequence in a vector
    vector<int> S(n);
    for (int i = 0; i < n; ++i) cin >> S[i];

    // Calculate the minimum value
    int m = S[0];
    for (int i = 1; i < n; ++i) {
        if (S[i] < m) m = S[i];
    }

    // Write the normalized sequence
    cout << n;
    for (int i = 0; i < n; ++i) cout << " " << S[i] - m;
    cout << endl;
}
```

Can we do this more efficiently?

Min value of a vector

```
// Pre: A is a non-empty vector
// Returns the min value of the vector

int minimum(const vector<int>& A) {
    int n = A.size();
    int m = A[0]; // visits A[0]
    // loop to visit A[1..n-1]
    for (int i = 1; i < n; ++i) {
        if (A[i] < m) m = A[i];
    }
    return m;
}
```

Vectors

- Vectors introduce some issues that must be taken into account:

  - a reference to a vector may not always exist. For example, if i=25 and vector x has 10 elements, then the reference x[i] does not exist.

  - So far, if x and y are two variables with different names, it can be assumed that they are different and independent objects. The only exception is when the *alias effect* is produced in the call to a function or procedure. For example:

```
int main() {
    int n;
    ...
    S(n,n)
    ...
}
```

Vectors

- if S is the procedure, then x and y become aliases of the same object (i.e., they represent the same object):

```
void S(int& x, int& y) {
    x = 4;
    y = 5;
    cout << x; // Writes 5
    ...
}
```

- When using vectors, x[i] and x[j] can be aliases if i and j have the same value. For example:

```
i = 4;
j = 3;
A[i] = 5;
A[j + 1] = 6;
cout << A[i]; // Writes 6
```

Vectors

- When a variable x has a simple type (e.g. int, char, …), the variable represents the same object during the whole execution of the program. However, when a vector x is used, the reference x[i] may represent different objects along the execution of the program. For example:

```
vector<int> x(5);
...
x[x[0]] = 1;
cout << x[x[0]]; // What does this write?
```

Vectors

```
vector<int> x(5);
x[0] = 0;
x[1] = 0;
x[2] = 0;
x[3] = 0;
x[4] = 0;

x[x[0]] = 1;
cout << x[x[0]]; // Writes 0
```

Constant parameters and variables

- A call-by-value parameter requires a copy of the parameter from the caller to the callee. It may be inefficient if the parameter is large (e.g. a large vector).

- Call-by-reference can be more efficient, but the callee may unexpectedly modify the parameter.

- **const** parameters can be passed by reference and be protected from any modification.

- How is the protection guaranteed?
  - **const** parameters cannot be written inside the function or passed to another function as a non-const parameter.

- **const** can also be applied to variables. Their value cannot change after initialization. Use constant global variables only to declare the constants of the program.

Constant parameters and variables

```
const double Pi = 3.14159; // Constant variable

void g(vector<int>& V) {
    ...
    V[i] = V[i - 1] + 1; // Allowed (V is not const)
    ...
}

int f(const vector<int>& A) {
    ...
    A[i] = A[i - 1] + 1; // Illegal (A is const)
    g(A); // Illegal (parameter of g is not const)
    Pi = 3.14;         // Illegal (Pi is const)
    ...
}
```

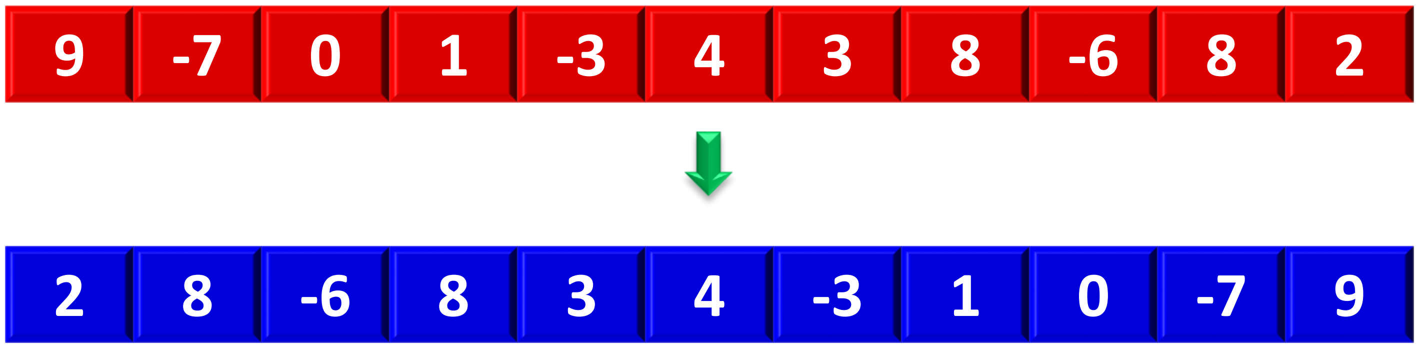
Average value of a vector

- Given a non-empty vector, return the average value of the elements in the vector.

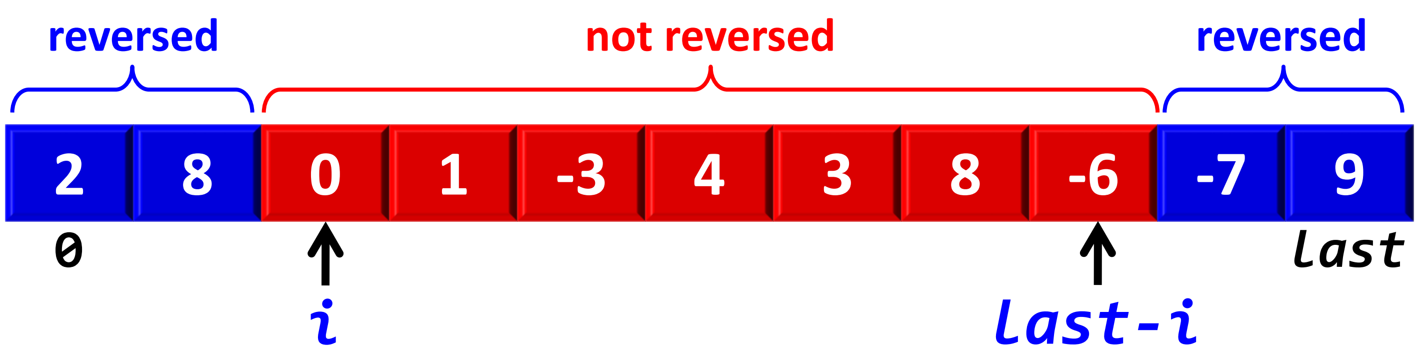
```
// Pre:  a non-empty vector A
// Returns the average value of the elements in A

double average(const vector<int>& A) {
    int n = A.size();
    int sum = 0;
    for (int i = 0; i < n; ++i) {
        sum = sum + A[i];
    }
    // Be careful: enforce a “double” result
    return double(sum)/n;
}
```

Reversing a vector

- Design a procedure that reverses the contents of a vector:



- Invariant:



Reversing a vector

```
// Pre:   -
// Post: A contains the reversed contents
//       of the input vector

void reverse(vector<int>& A) {
    int last = A.size() - 1;
    // Calculate the last location to reverse
    int middle = A.size()/2 - 1;

    // Reverse one half with the other half
    for (int i = 0; i <= middle; ++i) {
        int z = A[i];
        A[i] = A[last - i];
        A[last - i] = z;
    }
}
```

Reversing a vector (another version)

```
// Pre:   -
// Post: A contains the reversed contents
//       of the input vector

void reverse(vector<int>& A) {
    int i = 0;
    int last = A.size() - 1;
    // Inv: The elements in A[0…i-1] have been
    //      reversed with the elements in
    //      A[last+1…A.size()-1]
    while (i < last) {
        int z = A[i];
        A[i] = A[last];
        A[last] = z;
        i = i + 1;
        last = last - 1;
    }
}
```

The largest null segment of a vector

- A null segment is a compact sub-vector in which the sum of all the elements is zero.

- Let us consider vectors sorted in increasing order.

-9	-7	-6	-4	-3	-1	3	5	6	8	9

Null segment

-9	-7	-6	-4	-3	-1	3	5	6	8	9

Largest null segment

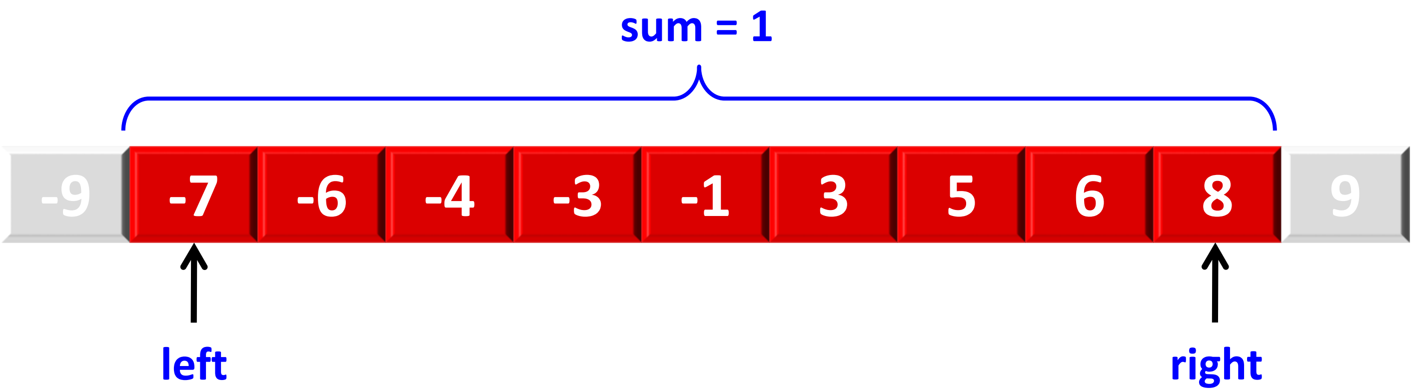
-9	-7	-6	-4	-3	-1	3	5	6	8	9

The largest null segment of a vector

- Observations:
  - If a null segment contains non-zero elements, then it must contain positive and negative elements.
  - Let us consider a segment of a vector. If the sum of the elements is positive, then the largest positive value cannot belong to any null segment included in the segment.
  - The same is true for negative numbers.

The largest null segment of a vector

- Invariant:



- The largest null segment is included in the [left…right] segment
- sum contains the sum of the elements in the [left…right] segment

Observation: the search will finish when sum = 0.
If the segment becomes empty (no elements) the sum will become 0.

The largest null segment of a vector

```
// Pre:  A is sorted in increasing order
// Post: <left,right> contain the indices of the
//       largest null segment. In the case of an empty
//       null segment, left > right.

void largest_null_segment (const vector<int>& A,
                           int& left, int& right)
    left = 0;
    right = A.size()-1;
    int sum = sum_vector(A); // Calculates the sum of A
    while (sum != 0) {
        if (sum > 0) {
            sum = sum - A[right];
            right = right - 1;
        }
        else {
            sum = sum - A[left];
            left = left + 1;
        }
    }
    // sum = 0 and the largest segment is A[left...right]
}
```

typedef

- **Typedef** declarations create *synonyms* for existing types:

```
// Declaration of the type
typedef vector<double> listTemperatures;

// Declaration of a variable
listTemperatures MyTemp;

// The parameter of a function
double maxTemp(const listTemperatures& L) {
    ...
}
```

Polynomial evaluation (Horner's scheme)

- Design a function that evaluates the value of a polynomial.

- A polynomial of degree n can be represented by a vector of n+1 coefficients $(a_0,\ldots,a_n)$. It can be efficiently evaluated using Horner's algorithm:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 =$$

$$(\cdots((a_n x + a_{n-1})x + a_{n-2})x + \cdots)x + a_0$$

- Example:

$$3x^3 - 2x^2 + x - 4 = \big((3x - 2)x + 1\big)x - 4$$

Polynomial evaluation (Horner's scheme)

```
// Definition of a polynomial (the coefficient of degree i
// is stored in location i of the vector).
typedef vector<double> Polynomial;

// Pre:  -
// Returns the evaluation of P(x)
double PolyEval(const Polynomial& P, double x) {

    double eval = 0;
    int degree = P.size() - 1;
    /* Invariant: the polynomial has been evaluated
       up to the coefficient i+1 using Horner's scheme */
    for (int i = degree; i >= 0; --i) {
        eval = eval*x + P[i];
    }
    return eval;
}
```

SEARCHING IN VECTORS

Search in a vector

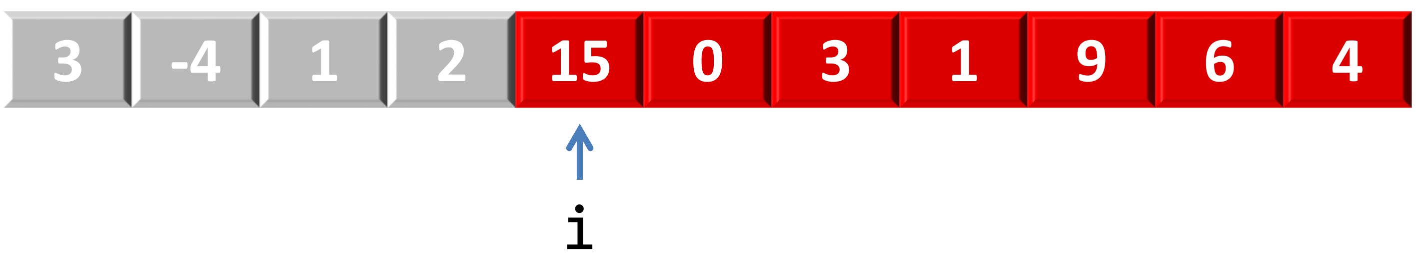
- We want to design a function that searches for a value in a vector. The function must return the index of the location in which the value is found. It must return -1 if not found.

- If several locations contain the search value, it must return the index of one of them.

```
// Pre: A is a non-empty vector
// Returns i, such that A[i] == x, if x is in A.
// Returns -1 if x is not in A.
```

Search in a vector

Invariant: x does not exist in A[0..i-1].



Note: an interval A[p..q] with p > q is assumed to be an empty interval.

Search in a vector

```
// Pre: --
// Returns i, such that A[i] == x, if x is in A.
// Returns -1 if x is not in A.

int search(int x, const vector<int>& A) {

    // Inv: x does not exist in A[0..i-1].
    for (int i = 0; i < A.size(); ++i) {
        if (A[i] == x) return i;
    }

    return -1;
}
```

Search with sentinel

- The previous code has a loop with two conditions:
  - **i < A.size()**: to detect the end of the vector
  - **A[i] == x**: to detect when the value is found

- The search is more efficient if the first condition is avoided (if we ensure that the value is always in the vector).

- To enforce this condition, a *sentinel* may be added in the last (unused) location of the vector. When the sentinel is found, it indicates that the value was not anywhere else in the vector.

Search with sentinel

```
// Returns i, such that A[i] == x, if x is in A.
// Returns -1 if x is not in A.
// Post: the vector is temporarily modified, but the
//       final contents remains unchanged.

int search(int x, vector<int>& A) {
    int n = A.size();
    A.push_back(x);      // Writes the sentinel

    int i = 0;
    // Inv: x does not exist in A[0..i-1]
    while (A[i] != x) ++i;
    A.pop_back();        // Removes the sentinel

    if (i == n) return -1;
    return i;
}
```

Be careful: not a const parameter

How would you search in a dictionary?

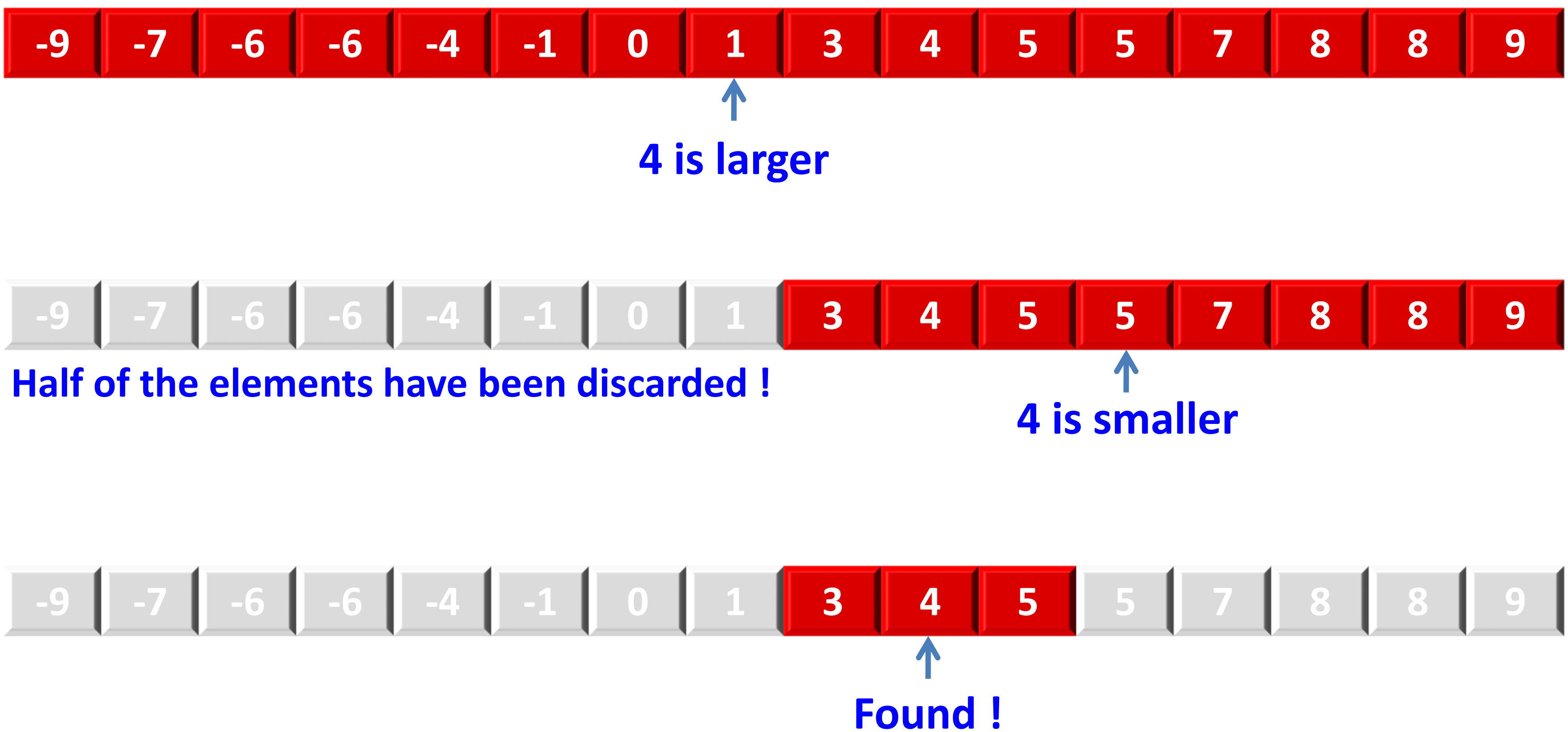
- Dictionaries contain a list of *sorted* words.

- To find a word in a dictionary of 50,000 words, you would never check the first word, then the second, then the third, etc.

- Instead, you would *look somewhere in the middle* and decide if you have to continue forwards or backwards, then you would look again around the middle of the selected part, go forwards/backwards, and so on and so forth …

Binary search

- **Is 4 in the vector?**



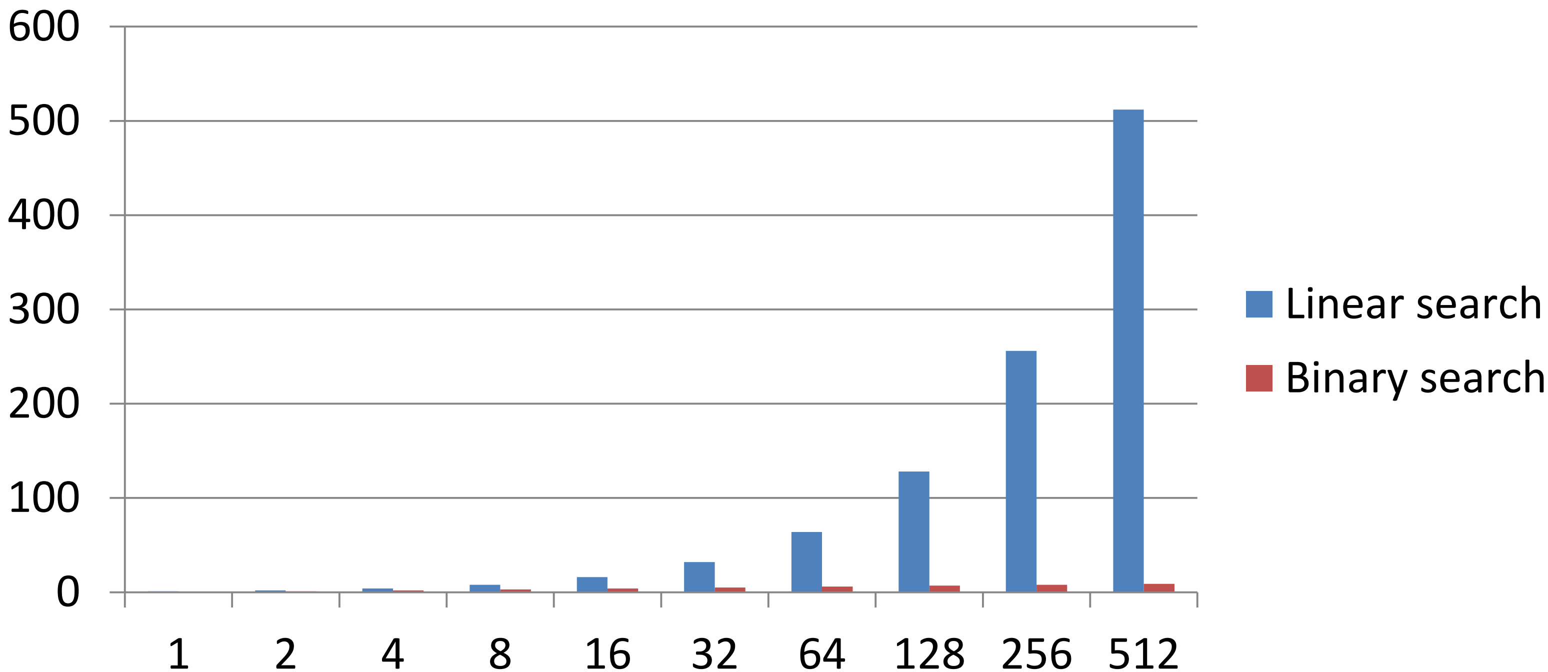
Binary search

- How many iterations do we need in the worst case?

iteration	0	1	2	3	4	5	6	7	i
elements	n	n/2	n/4	n/8	n/16	n/32	n/64	n/128	$n/2^i$

- The search will finish when only one element is left:

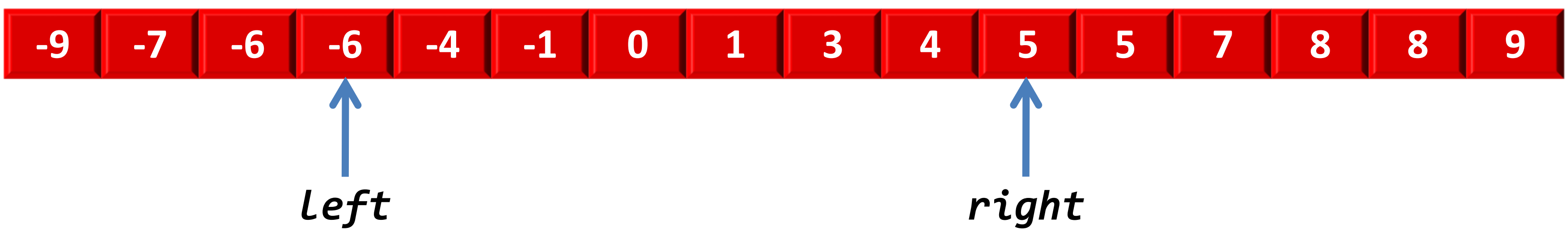
$$n = 2^i \quad \Rightarrow \quad i = \log_2 n$$



Binary search

Invariant:

If x is in vector A, then it will be found in fragment A[left...right]



The search will be completed when the value has been found or the interval is empty (left > right)

Binary search

```
// Pre: A is sorted in ascending order,
//      0 <= left,right < A.size()
// Returns the position of x in A[left...right].
// Returns -1 if x is not in A[left...right].

int bin_search(int x, const vector<int>& A,
               int left, int right) {

    while (left <= right) {
        int i = (left + right)/2;
        if (x < A[i]) right = i - 1;
        else if (x > A[i]) left = i + 1;
        else return i; //Found
    }

    return -1;
}
```

Binary search

```
// The initial call to bin_search should
// request a search in the whole array

   ...

   int i = bin_search(value, A, 0, A.size() - 1);

   ...
```

Binary search (recursive)

```
// Pre: A is sorted in ascending order,
//      0 <= left,right < A.size()
// Returns the position of x in A[left...right].
// Returns -1 if x is not in A[left...right].

int bin_search(int x, const vector<int>& A,
               int left, int right) {

    if (left > right) return -1;
    else {
        int i = (left + right)/2;
        if (x < A[i]) return bin_search(x,A,left,i-1);
        else if (x > A[i]) return bin_search(x,A,i+1,right);
        else return i;  // found
    }
}
```