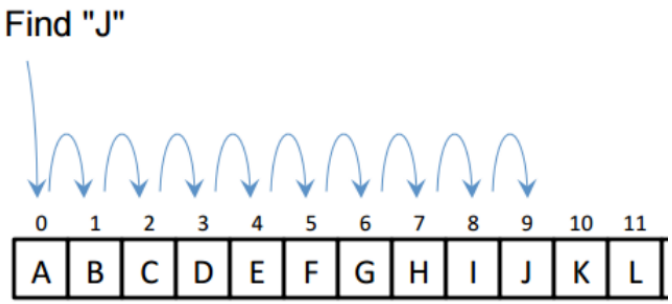
Searching

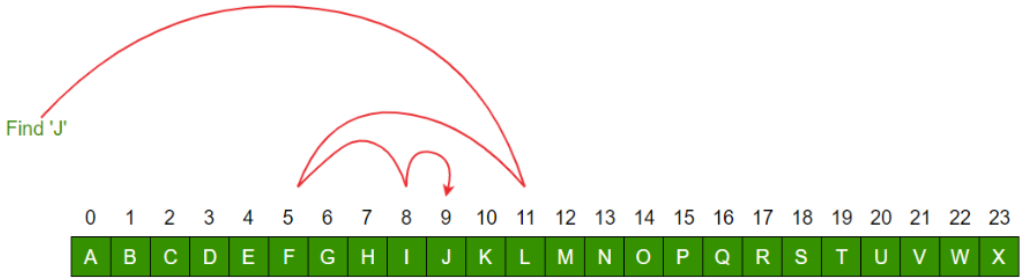
Sequential Search, Binary Search and Variants

L.EIC

Algoritmos e Estruturas de Dados

2024/2025



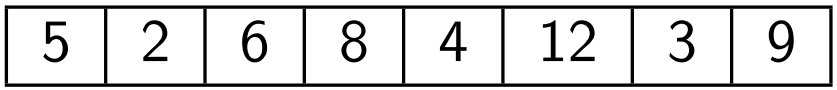


AP Rocha, P Ribeiro, AP Tomás

Storing Data

- One of the most essential programming tasks is to be able **store data** in memory

- We will discuss other **low level data structures** during the course (e.g. linked lists, trees) and **abstract data types** that can be implemented using these (e.g. stacks, queues, deque, maps, sets).
- For today, we will concentrate on the most simple (but powerful) one:



Arrays

The Search Problem

The search problem

Input:

- an array ***v*** storing ***n*** elements
- a target element ***key*** to search for

Output:

- **Index** *i* of ***key*** where ***v[i]=key***
- ***-1*** (if ***key*** is not found)

Example:

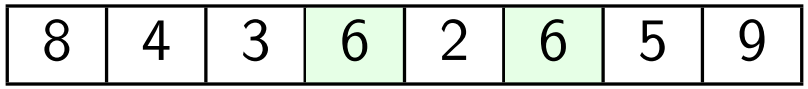
v =

5	2	6	8	4	12	3	9

search(v, 2) = 1
search(v, 7) = -1
search(v, 3) = 6
search(v, 14) = -1

The Search Problem

- variants for the case of arrays with **repeated values**:
  - indicate the position of the first occurrence
  - indicate the position of the last occurrence
  - indicate the position of any occurrence
  - indicate all the occurrences



Sequential Search

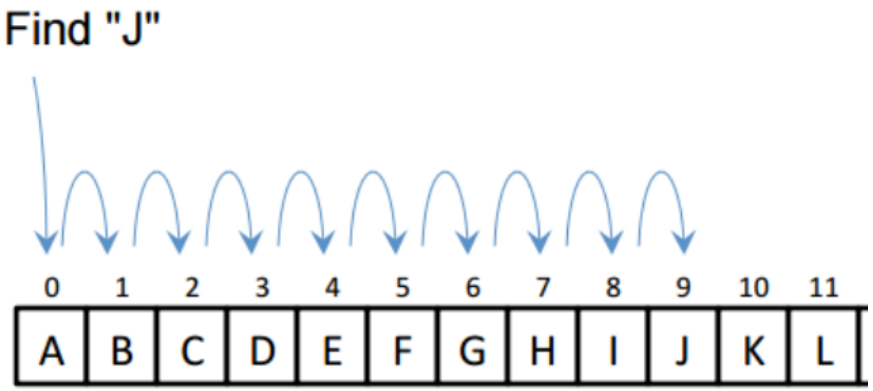
Algorithm

Sequential Search Algorithm

Sequentially checks each element of the array, from the first to the last[a] or from the last to the first[b] , until a match is found or the end of the array is reached

---

[a] if you want to know the position of the first occurrence
[b] if you want to know the position of the last occurrence



Sequential search: suitable for small or unordered arrays

Sequential search is called **linear search** also.

Sequential Search

An implementation

Search for an element *key* in a vector *v* of **comparable** elements. Returns the index of the first occurrence of *key*, if found, or -1, otherwise.

```
template <typename T>
int sequentialSearch(const vector<T> & v, const T & key) {
    for (unsigned i=0; i<v.size(); i++)
        if (v[i] == key)
            return i; // found key
    return -1; // not found
}
```

```
vector<int> v = {8,4,3,6,2,6,5,9};
cout << sequentialSearch(v, 8) << endl;
cout << sequentialSearch(v, 7) << endl;
cout << sequentialSearch(v, 6) << endl;
```

```
0
-1
3
```

Remembering C++ templates

- Templates are a way of creating generic code
- They are expanded in compile time
  - "almost" like macros, but with type checking
  - compiled code can have multiple copies of the same class
  - compiler needs to know code (common to see it implemented on .h)
- typename *vs* class (essentially, no semantic difference)

```
vector<char> v1 = {'A', 'C', 'B','Z','W'};
cout << sequentialSearch(v1, 'W') << endl;
```

```
4
```

```
vector<string> v2 = {"algorithms","data","structures"};
cout << sequentialSearch(v2, string("data")) << endl;
```

```
1
```

Sequential Search

Complexity

- Sequential Search **time complexity**
  - the test `if (v[i] == key)` is performed at most n times (e.g. if it doesn't find the target element)
  - if each test costs $\mathcal{O}(1)$, **time is $\mathcal{O}(n)$, being $\Theta(n)$ in the worst case**.

    (e.g., comparison of strings is not $\mathcal{O}(1)$ unless their length is bounded by a constant)
  - if the target element exists in the vector, we do approximately $n/2$ comparisons on average (assuming random input).

    Expected time is $\Theta(n)$ for "random input" if each test takes $\mathcal{O}(1)$.

    (Expected time complexity means *on average* in the statistical sense; "random" means equiprobable.)

- Sequential Search **space complexity**
  - space on local variables (including arguments)
  - since vectors are passed "by reference", the space taken up by the local variables is constant and independent of the vector size.
  - **Space is $\mathcal{O}(1)$**

sequentialSearch

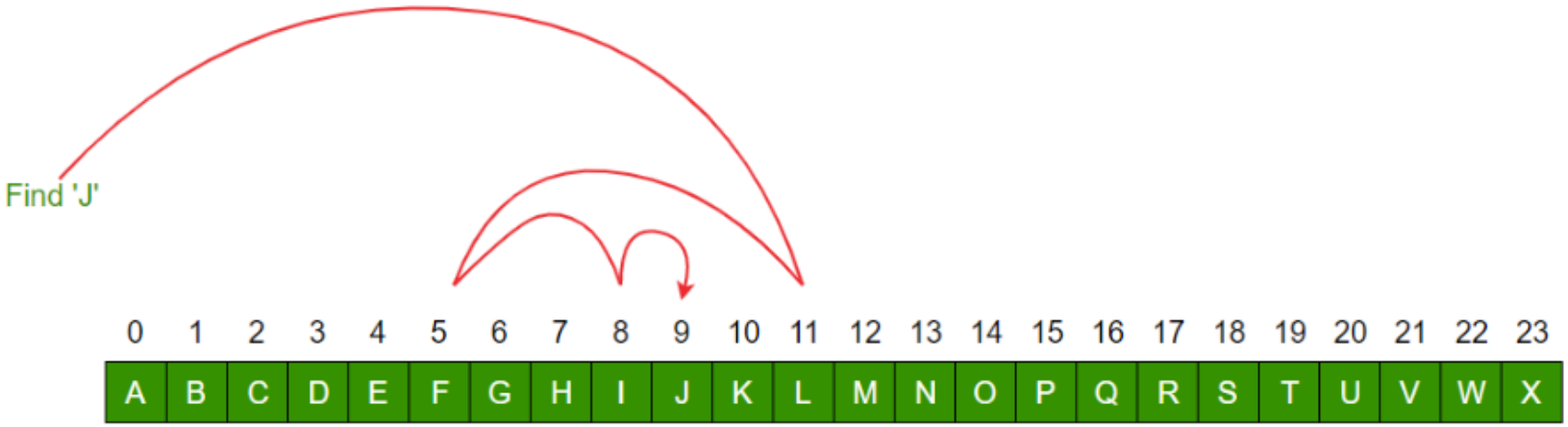
Time complexity analysis in more detail

	Best case	Worst case	Cost per operation
defs. `v` and `key` (passed **by reference**)	1	1	$\Theta(1)$
`unsigned i=0`	1	1	$\Theta(1)$
test `i` $<$ `v.size()`	1	$n+1$	$\Theta(1)$
test `v[i]` $==$ `key`	1	n	????
`return i`	1	0	$\Theta(1)$
`i++`	0	n	$\Theta(1)$
`return` -1	0	1	$\Theta(1)$

- ???? which is the complexity of == for that type of elements?
  $\Theta(1)$ for **int**, $\mathcal{O}(s)$ for two **strings** of size s (that is $\mathcal{O}(1)$ if $s \leq C$, for some constant C), ...
- **Overall time complexity** in the worst case:
  - for `vector<int>`: $\Theta(3n+4) = \Theta(n)$
  - for `vector<string>` and strings of size s: $\Theta(2n+4+ns) = \Theta(ns)$
- The instruction `if v[i] == key` is performed at most n times (exactly n in the worst case) and dominates the time complexity of **sequentialSearch**.

Searching in sorted arrays

- Suppose the array is **ordered**
  (arranged in increasing or non-decreasing order)
  - Sequential search on a sorted array still takes $\mathcal{O}(n)$ time
  - Can exploit sorted structure by performing **binary search**
  - Strategy: inspect middle of the structure so that half of the structure is discarded at every step



Binary Search

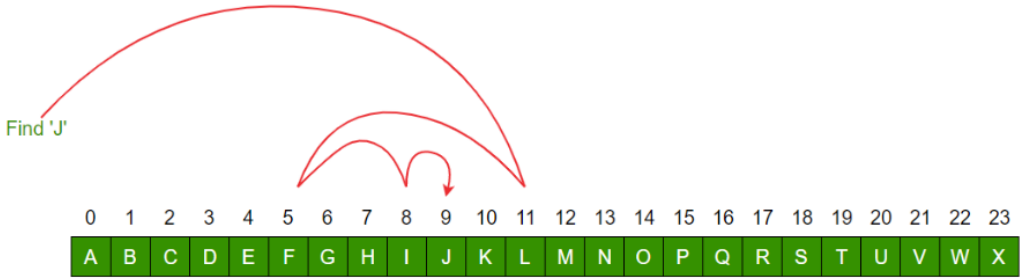
Algorithm

Binary Search Algorithm

compares the element in the middle of the array with the target element:

- is equal to the target element → found
- is greater than the target element → continue searching (in the same way) in the sub-array to the left of the inspected position
- is less than the target element → continue searching (in the same way) in the sub-array to the right of the inspected position

if the sub-array to be inspected reduces to an empty vector, we can conclude that the target element does not exist



Binary Search

Implementation

```
template <typename T>
int binarySearch(const vector<T> & v, const T & key) {
    int low = 0, high = v.size() - 1;
    while (low <= high) {
        int middle = low + (high - low) / 2;
        if (key < v[middle])        high = middle - 1;
        else if (key > v[middle])   low = middle + 1;
        else return middle; // found key
    }
    return -1; // not found
}
```

Loop invariant: if `key` occurs in the initial interval, then it will be in the interval defined by `[a,b]`, in every iteration.

Progress: at least `v[middle]` will be removed in each iteration.

(high-low) decreases

Bugs in binary search

- Why `low+(high-low)/2` instead of `(low+high)/2`?
- Mathematically, even for integer division, it is true that

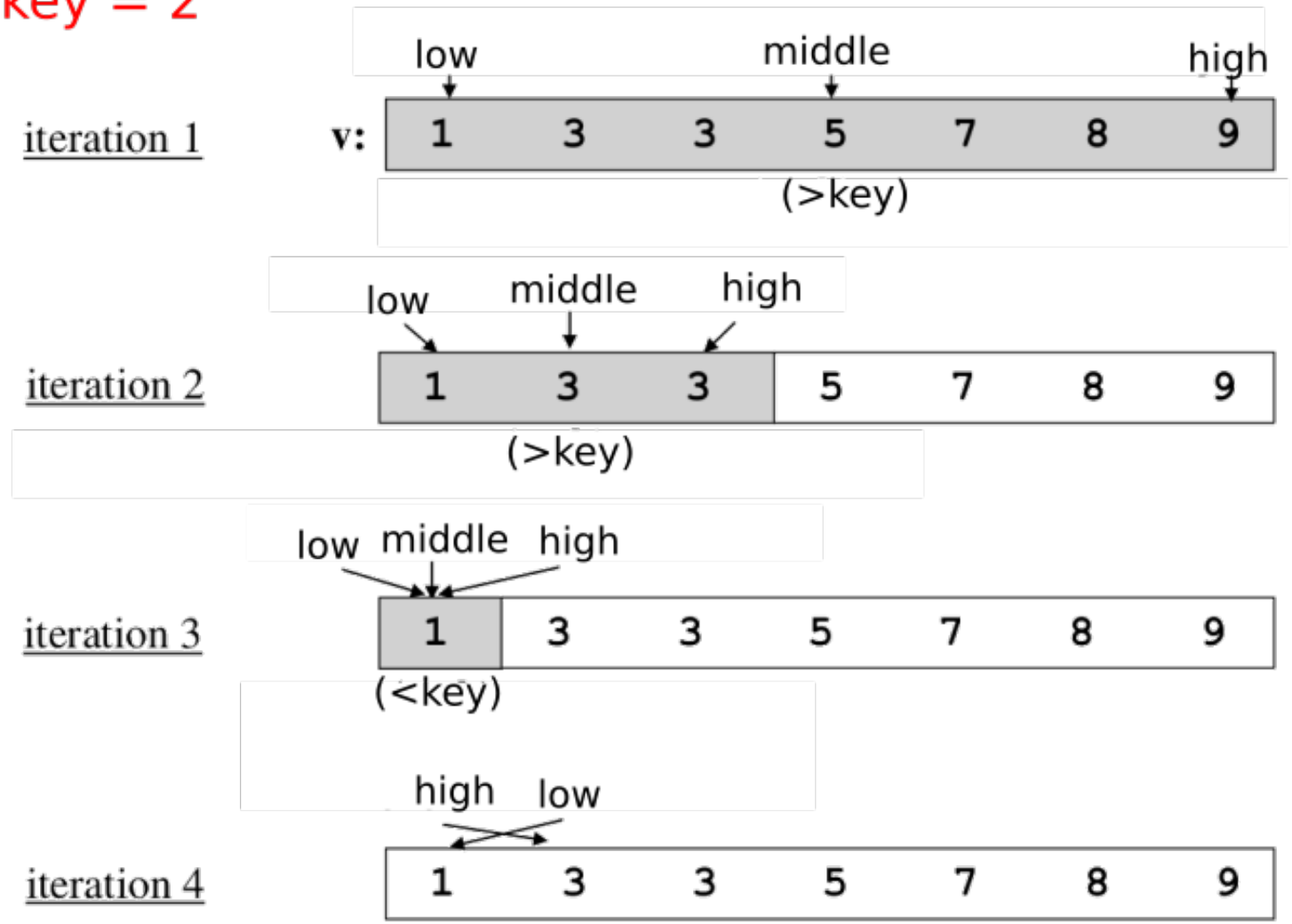
  ```
  low + (high-low)/2 = (low+high)/2
  ```

- But, `high+low` can cause **overflow** when `low` and `high` are large values of type `int`, whereas `low+(high-low)/2` cannot.
- So, in programming, `low + (high-low)/2` and `(low + high)/2` are not equivalent.

Binary Search

Visualization

key = 2



sub-array is empty → element 2 does not exist!

Binary Search

Complexity

- Binary Search **time complexity**
  - In each iteration, the size of the sub-array is divided by two
  - If each test costs $\Theta(1)$, the runtime satisfies this recurrence:

$$T(n) \approx T(n/2) + \Theta(1)$$

$\Theta(1)$ in that expression stands for a function $f(n)$ bounded by a constant

  - With $\mathcal{O}(1)$ per test, **time is $\mathcal{O}(\log n)$, being $\Theta(\log n)$ in worst case**.
  - It is crucial also that the parameter v be passed by reference `vector<T> & v`, which costs $\mathcal{O}(1)$ (with `vector<T> v`, it is $\Omega(n)$).

- Binary Search **space complexity**
  - space on local variables (including arguments)
  - since vectors are passed "by reference", the space taken up by the local variables is constant and independent of the vector size.
  - **Space is $\mathcal{O}(1)$**

Binary Search

Why $\log(n)$?

For simplicity, let us write $T(n) = T(n/2) + 1$, define $T(0) = 1$ and assume that $n = 2^k$, for some k, that is $k = \log_2(n)$.

$$
\begin{aligned}
T(n) &= T(n/2) + 1 = \\
&= (T(n/2/2) + 1) + 1 = T(n/4) + (1 + 1) \\
&= T(n/8) + (1 + 1 + 1) = \\
&\dots \\
&= T(n/2^k) + \sum_{i=1}^{k} 1 = \\
&= T(0) + \sum_{i=1}^{k+1} 1 = \\
&= k + 2 = \log_2(n) + 2
\end{aligned}
$$

Bound on the number of iterations

Let t be the number of divisions by 2.

$n/2^t < 1$ iff $t > \log_2(n)$, for all n.

So, $t \leq 1 + \lfloor \log_2 n \rfloor$.

Basis of logarithm often omitted because $\Theta(\log_a n) = \Theta(\log_b n)$ since $\log_a n = log_a b \times \log_b n$, for all $a, b > 1$.

Example: Saint John Festival (SWERC 2015)

Full description: https://icpcarchive.github.io/Southwestern_Europe_Regional_Contest_(SWERC).html (SWERC 2025)

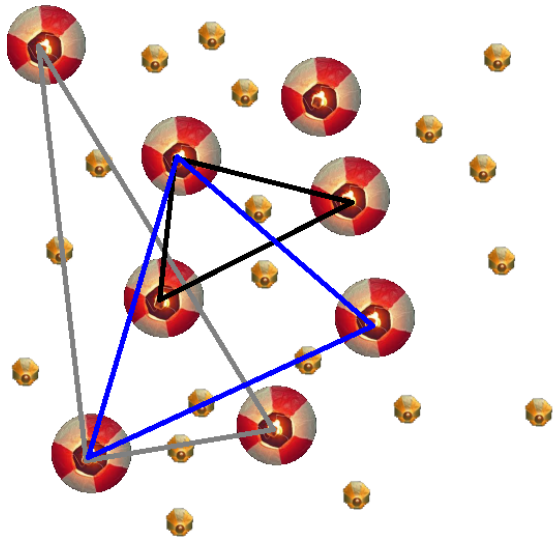
Given two sets of points $\mathcal{A}$ and $\mathcal{B}$ in the **plane**, such that $|\mathcal{B}| \gg |\mathcal{A}|$, how many points in $\mathcal{B}$ are in the interior or on the boundary of triangles defined by any 3 points in $\mathcal{A}$?

Charatheodory's Theorem:

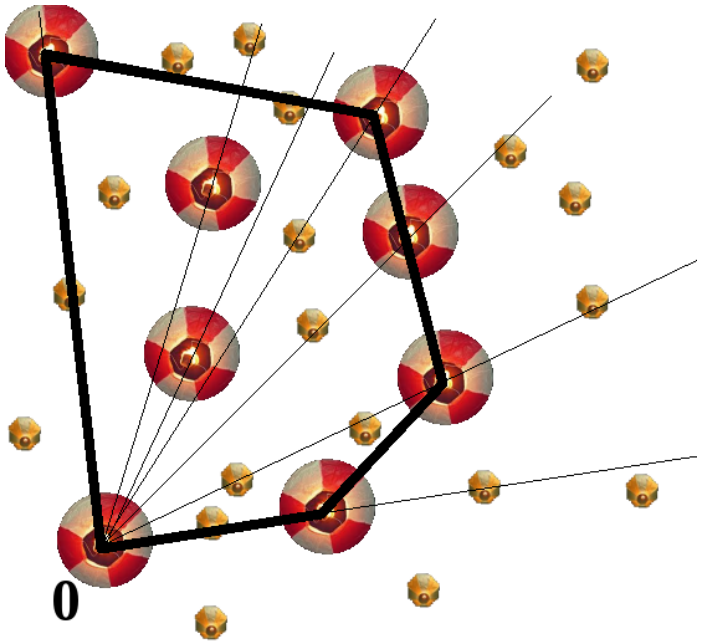
The union of all triangles with vertices in $\mathcal{A}$ is the **convex hull** $\mathcal{CH}(\mathcal{A})$ of $\mathcal{A}$.

Geometry

- **Convex hull**: $\mathcal{CH}(\mathcal{A})$ of $\mathcal{A}$?
- Check $p \in \mathcal{CH}(\mathcal{A})$?
  Point p in **convex polygon**?



Saint John Festival (SWERC 2015)



$\mathcal{CH}(\mathcal{A})$ is the smallest convex polygon that contains all points of $\mathcal{A}$.

Idea:

1. First, find $\mathcal{CH}(\mathcal{A})$ in $\mathcal{O}(L \log L)$, with $L = |\mathcal{A}|$, e.g., by Graham scan.

   (Graham scan will be described later in AED classes)

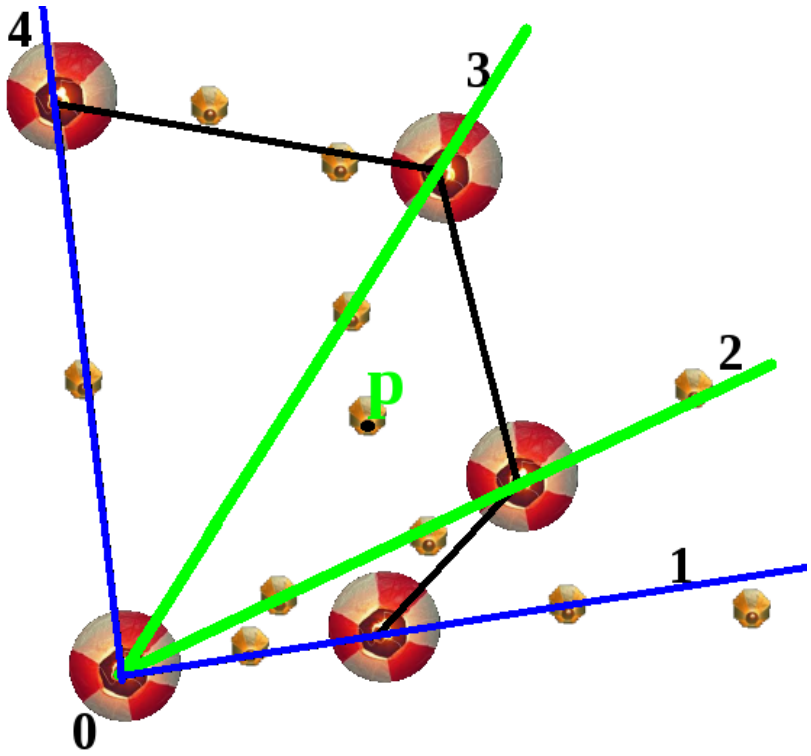
2. Then, for each $p \in \mathcal{B}$, check if $p \in \mathcal{CH}(\mathcal{A})$ efficiently.
   For **convex polygons**, we can use **Binary Search** to check.

   There are algorithms for deciding "$p \in \mathcal{P}$?" in $\mathcal{O}(k)$ time, for any k-vertex simple polygon $\mathcal{P}$. We will see now that, when $\mathcal{P}$ is convex, that can be done in $O(\log_2 k)$ time.

Saint John Festival (SWERC 2015)

Deciding if point p is in convex polygon

For each $p \in \mathcal{B}$, we can check if $p \in \mathcal{CH}(\mathcal{A})$ in $\mathcal{O}(\log h)$, by binary search, being h is the number of vertices of $\mathcal{CH}(\mathcal{A})$.



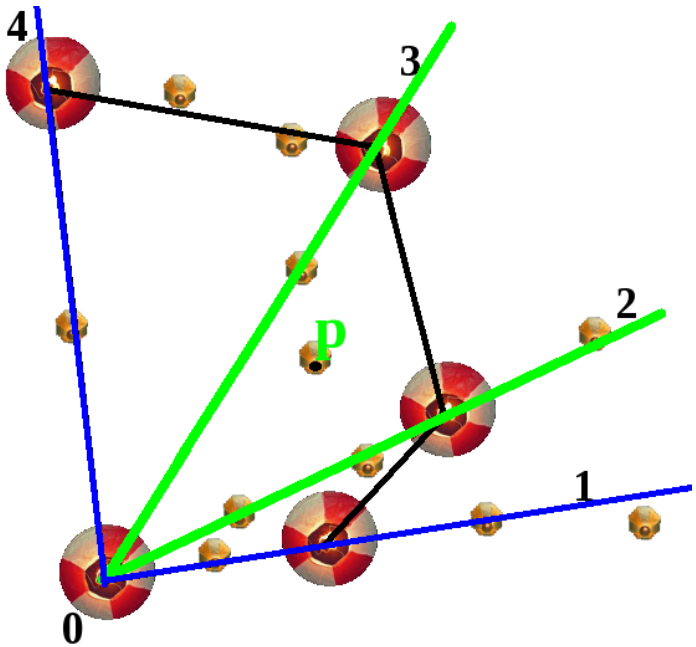
- $\mathcal{CH}(\mathcal{A})$ is defined by the sequence of its vertices $v_0, v_1, v_2, \ldots v_{h-1}, v_0$, in CCW order, given by they coordinates.
- $\mathcal{CH}(\mathcal{A})$ can be divided into wedges (cones with apex v_0) **already sorted** around v_0. Which is the middle one?

  If p is to the right of $v_0 v_k$ or to the left of $v_0 v_{k+1}$, what can be discarded?
- We can check $p \in wedge(v_0, v_k, v_{k+1})$ in $\mathcal{O}(1)$. So, $\mathcal{O}(S \log h)$ for $S = |\mathcal{B}|$ points. If $h \approx L$, this means $\mathcal{O}(S \log L)$.

Therefore, Saint John Festival can be solved in $\mathcal{O}(L \log L + S \log L)$ time.

Saint John Festival (SWERC 2015)

Checking whether p is in a wedge (v_0, v_i, v_{i+1}) in $\mathcal{O}(1)$



- E.g., $p \in \text{wedge}(v_0, v_2, v_3)$ if
  - (v_0, v_2, p) is Left-turn
  - (v_0, v_3, p) is Right-turn
  - (v_2, v_3, p) is Left-turn

- (p, q, r) is a left-turn (right-turn) if the non-null component of the **cross product** $\vec{pq} \times \vec{pr}$ is positive (negative). That component is given by $(x_q - x_p)(y_r - y_p) - (y_q - y_p)(x_r - x_p)$.
- We have to handle collinearities also!
- This problem requires robust tests for left-turn; right-turn; collinear; point in line segment. But, still $\mathcal{O}(1)$.

Another Application: Zeros of continuous functions

Example

How to find a solution of equation $x^3 - 5x + 2 = 0$? That is equivalent to find a zero of the function $f(x) = x^3 - 5x + 2$ that is, a value $x^\star$ such that $f(x^\star) = 0$.

To solve $x^3 - 5x + 2 = 0$, we can try to factorize the polynomial and then apply the formula for solving the quadratic equation. In this case, it is not difficult:

$$x^3 - 5x + 2 = (x^2 + 2x - 1)(x - 2)$$

But, that cannot be generalized.

Zeros of continuous functions

Finding approximate roots?

- For solving quadratic equations $ax^2 + bx + c = 0$, we have the quadratic formula: the equation has a solution in $\mathbb{R}$ iff $b^2 - 4ac \geq 0$ and the solution is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- But for some equations $f(x) = 0$, there is no formula. Even for polynomial equations! **Abel-Ruffini Theorem** says that there is no general solution through radicals for polynomial equations of **degree five or higher**.

  (**Abel-Ruffini Theorem** is a result you were not supposed to know)

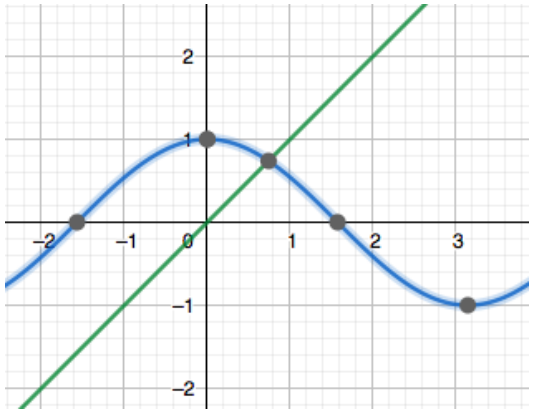
- For **non-polynomial functions**, we run into difficulties also.

  For example, can we find a zero of $g(x) = x - \cos(x)$?

Zeros of continuous functions

Bisection Method

Solve $x - \cos(x) = 0$, with $x \in [0, \pi/2]$?



The root corresponds to the intersection point of the graphs:

$$\begin{cases} y = x \\ y = \cos(x) \end{cases}$$

- We can obtain a sequence of numerical approximations

$$x_1, x_2, \ldots, x_n \ \ldots$$

  that converge to the root (i.e., the solution).
- **In theory**, the more iterations we do, the better the approximation will be.
- Let's look at the **bisection method** (a binary search approach).

Zeros of continuous functions

Bisection Method

We are given:

a function f;

an interval $[a, b]$;

a tolerance $\epsilon > 0$ (used as used as *stopping criterion*)

such that:

1. f is continuous in $[a, b]$;
2. $f(a) \times f(b) < 0$ (that is, f changes sign in $[a, b]$); (this invariant will be maintained)
3. f has a (single) zero in $[a, b]$.

Recall Bolzano's Theorem : if a continuous function f has values of opposite sign inside an interval, then it has a root in that interval. Therefore, if $f(a)f(b) < 0$, then f has at least one zero in $[a, b]$.

(This is a result you might know from the Analysis course unit)

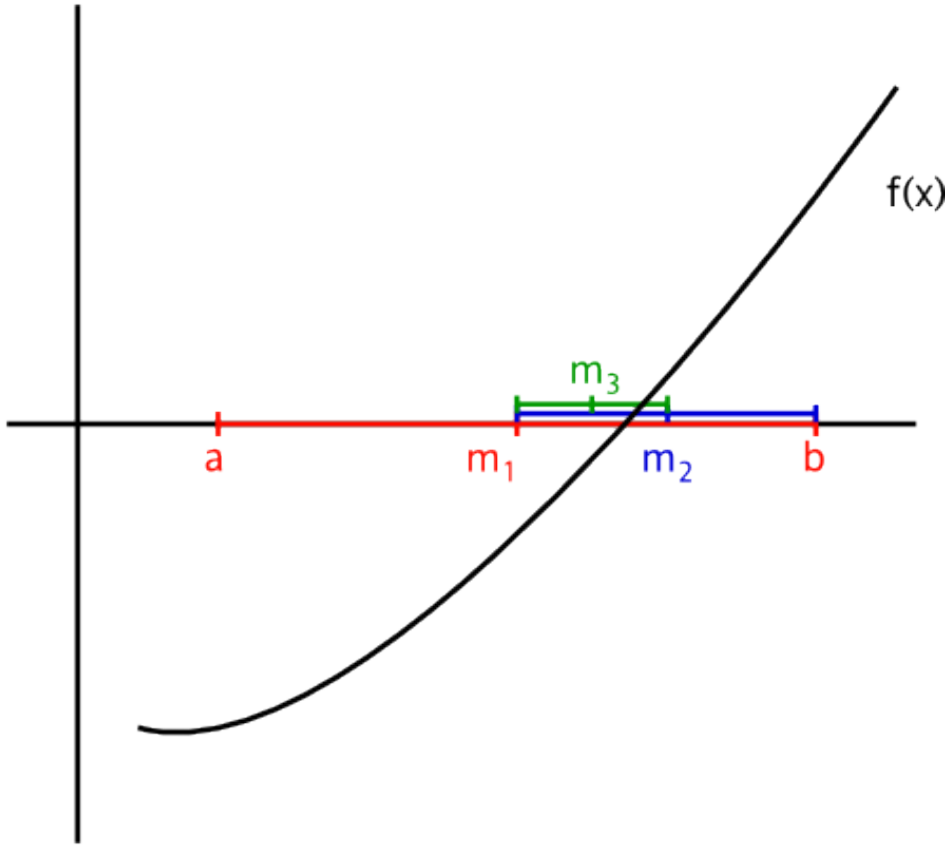
Zeros of continuous functions

Bisection Method – Algorithm

while $b - a > \epsilon$ do:

1. compute the midpoint
   $m \leftarrow (a + b)/2$
2. if $f(a) \times f(m) < 0$:
   the root is in $[a, m]$
   $b \leftarrow m$
3. if $f(a) \times f(m) > 0$:
   the root is in $[m, b]$
   $a \leftarrow m$
4. if $f(m) = 0$:
   the root is m

(usually, it does not obtain an exact root)



Zeros of continuous functions

Bisection Method in Python

```
def bisect(a,b,tol):
    while b-a > tol:
        m = (a+b)/2
        fm = f(m)     # f must be defined somewhere
        if fm == 0:
            return m
        if fm*f(a) > 0:
            a = m
        else:
            b = m
    return (a+b)/2       # an approximate solution


def f(x):    # definition of f
    return x**3-5*x+2
```

Zeros of continuous functions

Bisection Method Application

The root of $P(x) = x^3 - 5x + 2$ in the interval $[0, 1]$ with an error $< 10^{-4}$.

The exact value is $\sqrt{2} - 1 \approx 0.414213562$

```
>>>  bisect(0,1,0.0001)
f(0.000000) = 2.000000 f(1.000000) = -2.000000
a = 0.000000 b = 1.000000 fm = -0.375000    b-a = 1.000000
a = 0.000000 b = 0.500000 fm = 0.765625     b-a = 0.500000
a = 0.250000 b = 0.500000 fm = 0.177734     b-a = 0.250000
a = 0.375000 b = 0.500000 fm = -0.103760    b-a = 0.125000
a = 0.375000 b = 0.437500 fm = 0.035797     b-a = 0.062500
a = 0.406250 b = 0.437500 fm = -0.034290    b-a = 0.031250
a = 0.406250 b = 0.421875 fm = 0.000678     b-a = 0.015625
a = 0.414062 b = 0.421875 fm = -0.016825    b-a = 0.007812
a = 0.414062 b = 0.417969 fm = -0.008079    b-a = 0.003906
a = 0.414062 b = 0.416016 fm = -0.003702    b-a = 0.001953
a = 0.414062 b = 0.415039 fm = -0.001512    b-a = 0.000977
a = 0.414062 b = 0.414551 fm = -0.000417    b-a = 0.000488
a = 0.414062 b = 0.414307 fm = 0.000130     b-a = 0.000244
a = 0.414185 b = 0.414307 fm = -0.000144    b-a = 0.000122
0.414215087890625
```

Zeros of continuous functions

Improving the implementation

```
def bisect(a,b,tol):
    fa = f(a)
    while b-a > tol:
        m = (a+b)/2
        fm = f(m)
        if fm == 0:
            return m
        if fm*fa > 0:
            a = m
            fa = fm
        else:
            b = m
    return (a+b)/2
```

In each iteration, it **computes** f **just once**, instead of twice. That can lead to a significant gain if f is computationally complex.

Zeros of continuous functions

Bisection Method with function f passed as an argument

```
def bisect(f,a,b,tol):    # the first parameter is a function
    fa = f(a)
    while b-a > tol:
        m = (a+b)/2
        fm = f(m)
        if fm == 0:
            return m
        if fm*fa > 0:
            a = m
            fa = fm
        else:
            b = m
    return (a+b)/2
```

Zeros of continuous functions

Bisection Method with function f passed as an argument

We'd rather **pass the function to be evaluated as an argument** or use lambda expressions, for that:

```
>>> bisect(f, 0, 1, 1e-4)
0.41424560546875
>>> bisect(lambda x : x**3-5*x+2, 0, 1, 1e-4)
0.41424560546875
```

In Python, the expressions

```
lambda x : x**3-5*x+2
lambda x : x-math.cos(x)
```

represent the functions $x \longmapsto x^3 - 5x + 2$ e $x \longmapsto x - \cos(x)$.

Lambda expressions allow us to define **anonymous** (unnamed) functions.
They are available in C++ also, but the syntax is different. (see example in these slides)

Other Applications of Binary Search

Revisiting an example of last class

To find the position of the first occurrence of the maximum value of v, assuming that $v[0] \leq v[1] \leq v[2] \leq \ldots \leq v[n-1]$.

```
int posMaxSorted(int v[],int n) {   // sequential search
    int i = n-2;
    while (i >= 0 && v[i] == v[i+1])
        i--;
    return i+1;
```

Best case (sequential search): The maximum occurs only once. Time $\mathbf{\Theta(1)}$
Worst case (sequential search): All elements are equal. Time: $\mathbf{\Theta(n)}$
Conclusion: the time complexity of `posMaxSorted(v,n)` is $\mathcal{O}(\mathbf{n})$.

How to improve the running time to $\mathcal{O}(\log \mathbf{n})$? (We will see next)

Binary Search

A generalization

We can generalize **binary search** for cases where we have something like:

no	no	no	no	no	yes	yes	yes	yes	yes	yes

We want to find the **first yes** (or in some cases the **last no**)

Example:

- Search the smallest element larger or equal to *key*
  (**lower_bound** do C++)

2	5	6	8	9	12
no	no	no	yes	yes	yes

lower_bound(7) $\rightarrow$ condition: $v[i] >= 7$

[the smallest number ≥ 7 on this array is 8]

Binary Search

A generalization

Binary search for *condition* (in "pseudo-code")

```
bsearch(low, high, condition)
    While (low < high ) do
        middle = low + (high − low)/2
        If (condition(middle) == yes) high = middle
        Else                          low  = middle + 1
    If (condition(low) == no) return(-1)
    return(low)
```

v =

2	5	6	8	9	12
no	no	no	yes	yes	yes

bsearch(0, 5, $\geq$ 7)

$low = 0, high = 5, middle = 2$

Since $v[2] \geq 7$ is no: $low = 3, high = 5, middle = 4$

Since $v[4] \geq 7$ is yes: $low = 3, high = 4, middle = 3$

Since $v[3] \geq 7$ is yes: $low = 3, high = 3$ (leaves while)

Since $v[3] \geq 7$ is yes: **return(3)**

Binary Search

Correctness of Binary search for *condition*

Binary search for *condition* (in "pseudo-code")

```
bsearch(low, high, condition)
    While (low < high ) do
        middle = low + (high − low)/2
        If (condition(middle) == yes) high = middle
        Else                          low  = middle + 1
    If (condition(low) == no) return(-1)
    return(low)
```

Loop invariant: if `low` and `high` were valid initially and `low` $\leq$ `high`, then, in every iteration, `low` $\leq$ `high`, both are valid, and the **first** yes is in [`low`, `high`], if there is any. Otherwise, the last no is at `high`.

Progress: in each iteration, `high` − `low` decreases.

(since `low` < `high`, then `low` $\leq$ `middle` < `high` immediately after finding `middle`)

Termination: the loop ends when `low` = `high`. From the invariant, `low` is valid and gives the **first** yes iff `condition(low)` holds. So, the result is correct.

Binary Search the Answer

Balanced Partition Problem

Let's look at a more advanced example of an application of this generalized binary search

Balanced Partition Problem

Input: a sequence $\langle a_1, \ldots, a_n \rangle$ of n integers and an integer k
Output: a way of partitioning the sequence into k **contiguous** subsequences, minimizing the maximum sum of all parts.

Example:
7 9 3 8 2 2 9 4 3 4 7 9 9 $\quad\quad$ $k = 4$ (4 partitions)

7 9 3|8 2 2|9 4 3|4 7 9 9 $\rightarrow 19 + 12 + 16 + 29$
7 9 3 8|2 2 9|4 3 4 7|9 9 $\rightarrow 27 + 13 + 18 + 18$
7 9|3 8 2 2|9 4 3 4|7 9 9 $\rightarrow 16 + 15 + 20 + 25$
...

What is the best partition? (with the smallest possible maximum)

Binary Search the Answer

Balanced Partition Problem

- Exhaustive search would need to look for all the possible partitions! (can you estimate how many?)
- On another course you may revisit this problem to solve it with **dynamic programming**
- In this class we will solve with... **binary search!**

Binary Search the Answer

Balanced Partition Problem

Let's first think on a simpler but "similar" problem:

Can we divide such that the maximum sum of a partition is $\leq X$?

***"Greedy" idea*:** extend the current partition while its *sum* $< X$!

Examples:

Let $X = 21$ and $k = 4$

7 9 3|8 2 2 9 4 3 4 7 9 9

7 9 3|8 2 2 9|4 3 4 7 9 9

7 9 3|8 2 2 9|4 3 4 7|9 9

7 9 3|8 2 2 9|4 3 4 7|9 9 - OK!

Let $X = 20$ and $k = 4$

7 9 3|8 2 2 9 4 3 4 7 9 9

7 9 3|8 2 2|9 4 3 4 7 9 9

7 9 3|8 2 2|9 4 3 4|7 9 9

7 9 3|8 2 2|9 4 3 4|7 9|9 - Failed! We would need more than 4 partitions

Binary Search the Answer

Balanced Partition Problem

Can we divide such that the maximum sum of a partition is $\leq X$?

If we think about the X's for which the answer is yes, we have a search space where the following happens:

no	no	...	no	no	yes	yes	yes	...	yes	yes

We can apply **binary search on X**! *(solution is the first yes)*

- Let s be the sum of all numbers, that is $s = \sum_{i=1}^{n} a_i$.
- At least, X will be 1 (or in alternative the largest a_i). At most, X will be s. That defines an interval, e.g., $[1, s]$ or $[\max_i a_i, s]$.
- Verify the answer for a certain X: $\mathcal{O}(n)$ *(using the greedy method above)*
- Binary Search on X: $\mathcal{O}(\log s)$
- Global time: $\mathcal{O}(n \log s)$ *Note that, this bound depends on the values in the instance! Not only on n.*

Binary Search the Answer

Balanced Partition Problem

Example: 7 9 3 8 2 2 9 4 3 4 7 9 9 $\quad\quad$ $k = 4$ (4 parts)

low = 1, high = 76, middle = 38 → isPossible(38)? Yes
low = 1, high = 38, middle = 19 → isPossible(19)? No
low = 20, high = 38, middle = 29 → isPossible? Yes
low = 20, high = 29, middle = 24 → isPossible? Yes
low = 20, high = 24, middle = 22 → isPossible? Yes
low = 20, high = 22, middle = 21 → isPossible? Yes
low = 20, high = 21, middle = 20 → isPossible? No
low = 21, high = 21

Leaves the cycle and verifies that **isPossible(21)**, with that being the answer!

7 9 3|8 2 2 9|4 3 4 7|9 9 → 19 + 21 + 18 + 18

Binary Search the Answer

Balanced Partition Problem

2nd Example: 7 9 3 8 2 2 9 4 3 4 7 9 9 $\quad\quad$ $k = 3$ (3 parts)

low = 1, high = 76, middle = 38 → isPossible(38)? Yes
low = 1, high = 38, middle = 19 → isPossiblel(19)? No
low = 20, high = 38, middle = 29 → isPossible(29)? Yes
low = 20, high = 29, middle = 24 → isPossible(24)? No
low = 25, high = 29, middle = 27 → isPossible(27)? Yes
low = 25, high = 27, middle = 26 → isPossible(26)? No
low = 27, high = 27

Leaves the cycle and verifies that **isPossible(27)**, with that being the answer!

7 9 3 8|2 2 9 4 3 4|7 9 9 → 27 + 24 + 25

(this methodology is commonly known as **"binary search the answer"**)

Binary Search

- Binary Search is very **useful** and **flexible**
- It can be used on a **large range of applications**
- There are many other **variations**, besides the ones we already discussed.
  - Interpolation search
    (instead of looking at the middle, estimate position)
  - Exponential search
    (start by trying to fix interval in $low = 2^a$ and $high = 2^{a+1}$)
  - Ternary search
    (max or min in unimodal function)
  - ...

Searching in C++

STL Algorithms

- <algorithm> includes functions for searching
- **Sequential search** (some example functions)
  - **find** - find the first element equal to a key in a range
  - **find_if** - find the first element satisfying a criteria in a range
- **Binary search** (some example functions)
  - **binary_search** - binary search for an element in sorted range
  - **lower_bound** - binary search for the first element not less than the given value

C++ Iterators

Before starting, let's just refresh your knowledge of iterators.

```
vector<int> v = {1,2,3,4};

// auto "automatically" discovers type
auto it = v.begin();
cout << *it << endl; // print first element

it++; // move forward (forward iterator)
cout << *it << endl; // print second element

it--; // go back (bidirectional iterator)
cout << *it << endl; // print first element

it+=2; // advance 2 positions (random acess iterator)
cout << *it << endl; // print third element
```

```
1
2
1
3
```

C++ Iterators

Documentation:

<table>
<tr><th colspan="5">Iterator category</th><th>Defined operations</th></tr>
<tr><td rowspan="5">LegacyContiguousIterator</td><td rowspan="4">LegacyRandomAccessIterator</td><td rowspan="3">LegacyBidirectionalIterator</td><td rowspan="2">LegacyForwardIterator</td><td>LegacyInputIterator</td><td>• read
• increment (without multiple passes)</td></tr>
<tr><td></td><td>• increment (with multiple passes)</td></tr>
<tr><td colspan="2"></td><td>• decrement</td></tr>
<tr><td colspan="3"></td><td>• random access</td></tr>
<tr><td colspan="4"></td><td>• contiguous storage</td></tr>
<tr><td colspan="6">Iterators that fall into one of the above categories and also meet the requirements of LegacyOutputIterator are called mutable iterators.</td></tr>
<tr><td>LegacyOutputIterator</td><td colspan="4"></td><td>• write
• increment (without multiple passes)</td></tr>
</table>

C++ Iterators

Iterators can be used to traverse a range

```
vector<int> v = {2,4,6,8};

// v.end() is "after" last position
for (auto it = v.begin(); it!= v.end(); it++)
    cout << *it << " ";
cout << endl;

// foreach" style loops can also be used
for (auto i : v)
    cout << i << " ";
cout << endl;

// we also have reverse iterator
for (auto it = v.rbegin(); it!= v.rend(); it++)
    cout << *it << " ";
cout << endl;
```

```
2 4 6 8
2 4 6 8
8 6 4 2
```

Searching in C++

find

(returns an iterator to the 1st element in the range equal to *value*)

Documentation:

```
template<class InputIt, class T>
InputIt find(InputIt first, InputIt last, const T& value);
```

Example usage:

```
vector<int> v = {2,4,6,8};

auto result1 = find(v.begin(), v.end(), 4);
if (result1 != v.end()) cout << "found 4" << endl;
else cout << "4 not found" << endl;

auto result2 = find(v.begin(), v.end(), 5);
if (result2 != v.end()) cout << "found 5" << endl;
else cout << "5 not found" << endl;
```

```
found 4
5 not found
```

Searching in C++

find

Note how it searches within a range.
For instance, we could use it for all occurrences like this:

```
vector<int> v = {1,2,4,2,2,6};

auto it = find(v.begin(), v.end(), 2);
while (it != v.end()) {
    cout << "found 2 at index " << (it - v.begin()) << endl;
    it++;
    it = find(it, v.end(), 2);
}
```

```
found 2 at index 1
found 2 at index 3
found 2 at index 4
```

Searching in C++

find_if

(returns an iterator to the 1st element in the range for which p returns *true*)

Documentation:

```
template<class InputIt, class UnaryPredicate>
InputIt find_if(InputIt first, InputIt last,
                UnaryPredicate p);
```

Example usage:

```
bool isEven(int i) {
   return i%2 == 0;
}
```

```
vector<int> v = {2,4,6,8};

auto result = find_if(v.begin(), v.end(), isEven);
if (result != v.end()) cout << "found even number" << endl;
else cout << "even number not found" << endl;
```

```
found even number
```

Searching in C++

find_if

We could use "lambda" expressions to have more compact declarations (or even "anonymous" functions)

```
vector<int> v = {1,2,3,4,5,6,7,8};

// lambda expression
auto isEven = [](int i){return i%2 == 0;};

auto result1 = find_if(v.begin(), v.end(), isEven);
if (result1 != v.end())
   cout << "first even number is " << *result1 << endl;

// using anonymous function
auto result2 = find_if(v.begin(), v.end(),
                       [](int i) {return i>5;});
if (result2 != v.end())
   cout << "first number >5 is " << *result2 << endl;
```

```
first even number is 2
first number >5 is 6
```

Searching in C++

binary_search

(determines if an element exists in a partially-ordered range)

Documentation:

```
template< class ForwardIt, class T>
bool binary_search(ForwardIt first, ForwardIt last,
                                    const T& value);
```

Example usage:

```
vector<int> v = {2,4,5,7,9};

if (binary_search(v.begin(), v.end(), 5))
   cout << "found 5" << endl;
else cout << "5 not found" << endl;

if (binary_search(v.begin(), v.end(), 6))
   cout << "found 6" << endl;
else cout << "6 not found" << endl;
```

```
found 5
6 not found
```

Searching in C++

lower_bound

(returns an iterator to the first element not less than the given value, in a partially ordered range, using binary search)

Documentation:

```
template< class ForwardIt, class T >
ForwardIt lower_bound(ForwardIt first, ForwardIt last,
                      const T& value);
```

Example usage:

```
vector<int> v = {2,4,5,7,9};

auto result = lower_bound(v.begin(), v.end(), 6);
if (result != v.end())
cout << "first element >= 6 is " << *result << endl;
```

```
first element >= 6 is 7
```

Remembering classes

Refreshing your knowledge:

```
class Person {
    string id;
    string name;
    int age;
public:
    Person(string id, string n="UNDEFINED", int a=0);
    string getId() const;
    string getName() const;
    int getAge() const;
};

Person::Person(string i, string n, int a) :
         id(i), name(n), age(a) {}
string Person::getId() const {return id;}
string Person::getName() const {return name;}
int Person::getAge() const {return age;}
```

Comparable elements

- Custom classes are not inherently comparable

```
Person p1("12345", "John", 42);
Person p2("42424", "Mary", 37);
Person p3("55555", "Chariloe", 61);

vector<Person> v = {p1, p2, p3};
auto result = find(v.begin(), v.end(),p1);
```

```
error: no match for 'operator=='
```

(C++ errors can be hard to read: look at the beginning of error message!)

- We can however add == operator (and others such as <, etc)

```
bool Person::operator==(const Person & p1) {
   return id == p1.id;
}
```

or, alternatively

```
bool operator==(const Person &p1, const Person &p2) {
        return p1.getId() == p2.getId();
}
```