Week 2: Wednesday

EECS 281

MAXIM ALEKSA

maximal.io

Agenda

Templates

Standard Template Library

Containers

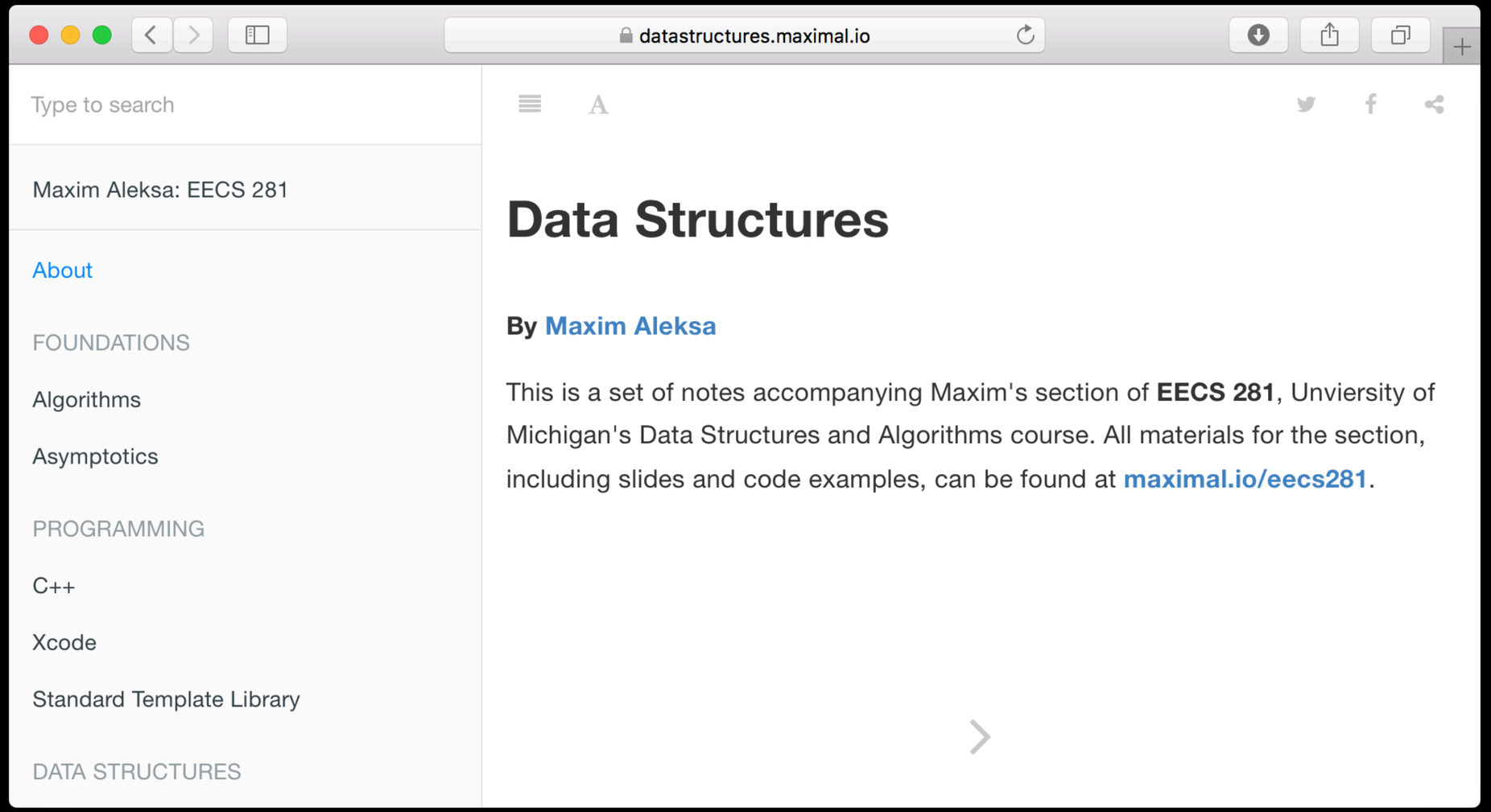
Array

Vector

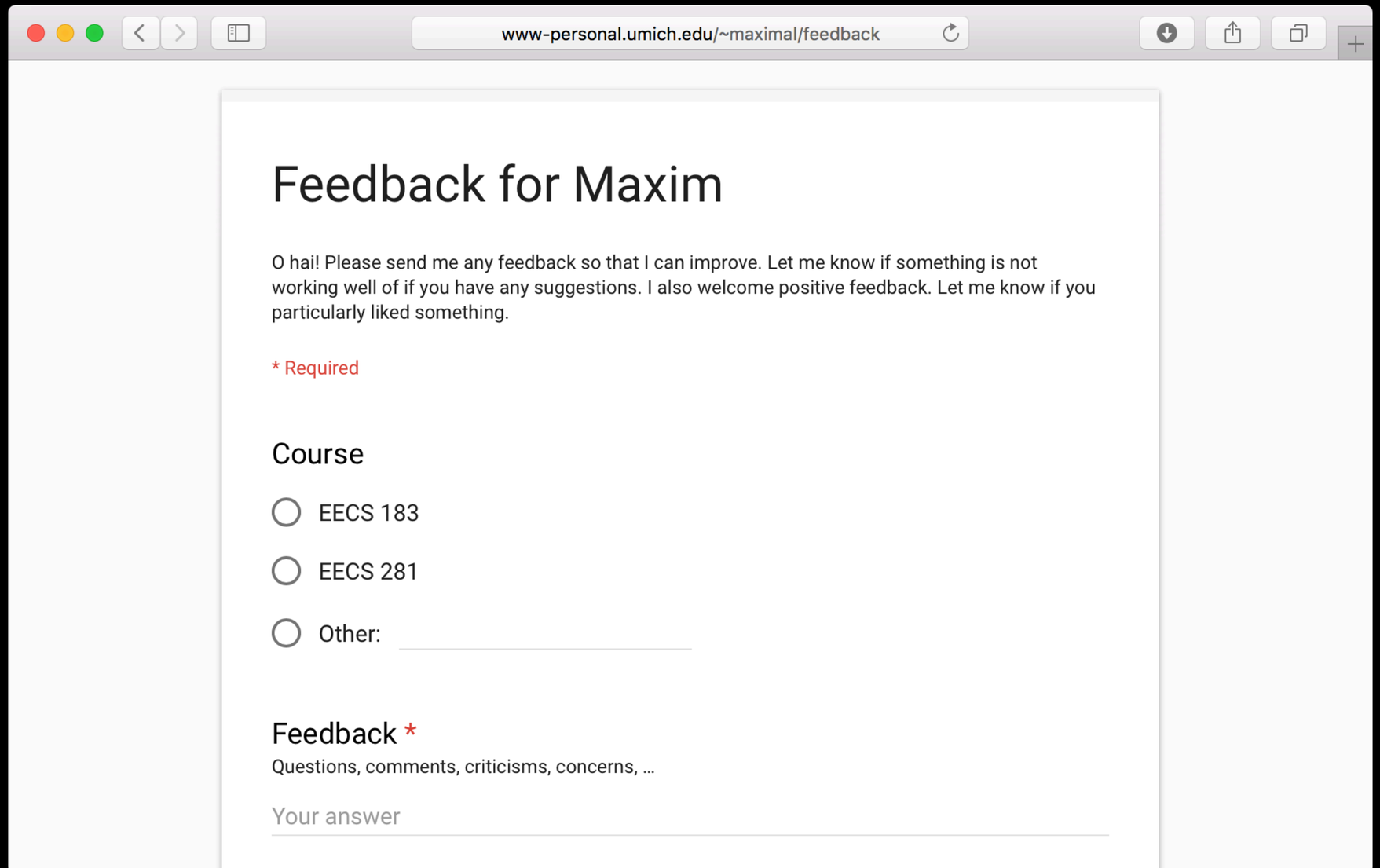
Deque

(Linked) list

datastructures.maximal.io



maximal.io/feedback



Sorting

Improving complexity

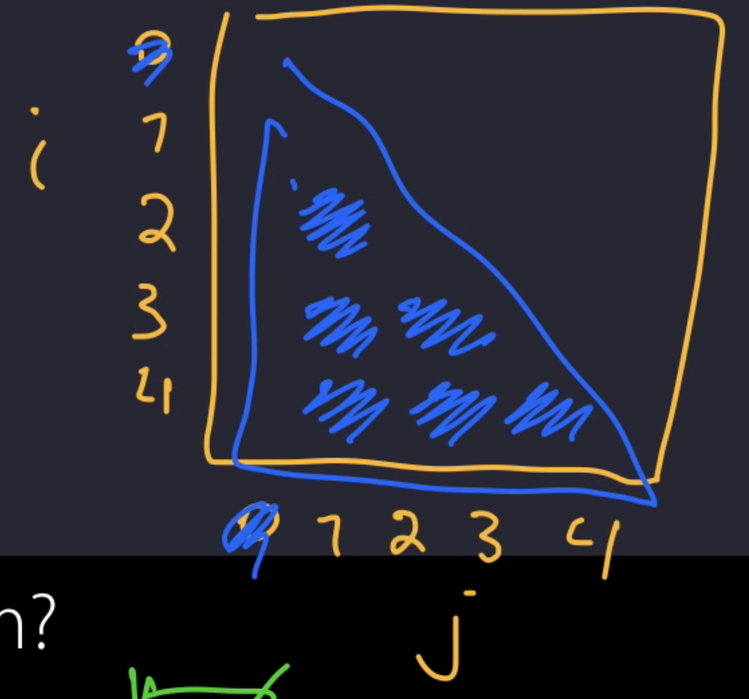
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```

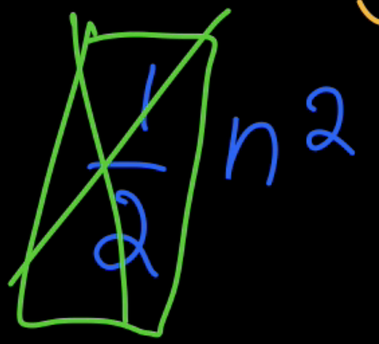
What is the growth complexity of this algorithm?

Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j − 1] > a[j])
        swap a[j] and a[j − 1]
        −−j
```

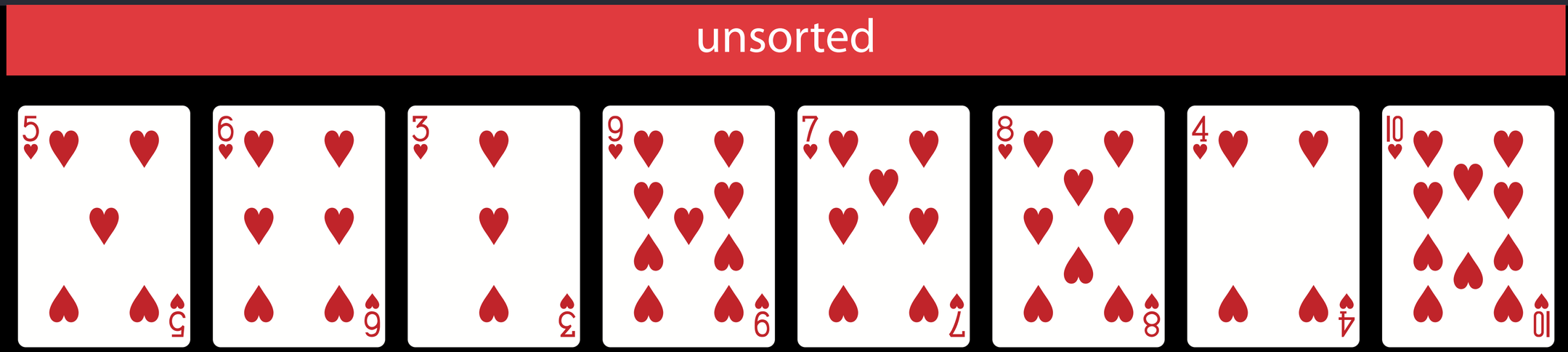


What is the growth complexity of this algorithm?



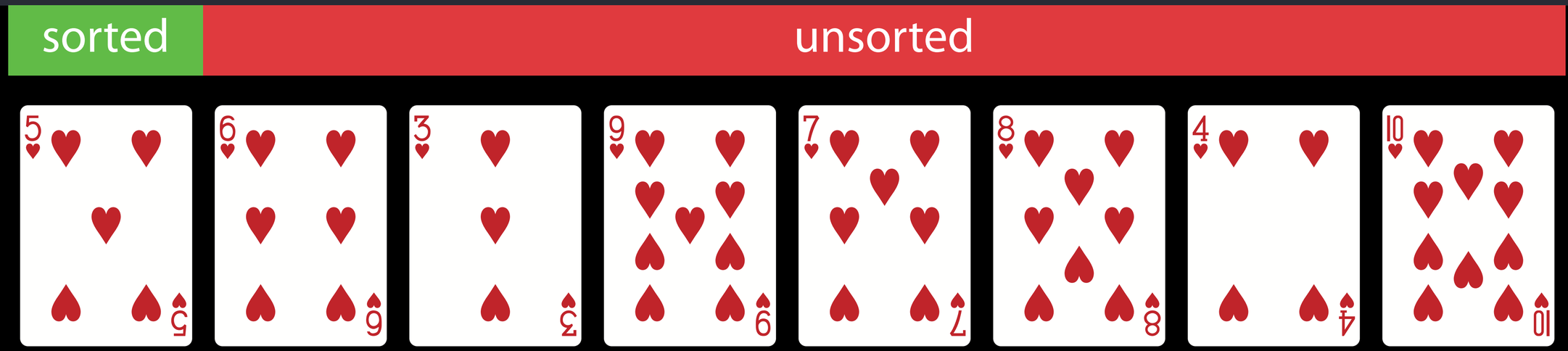
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



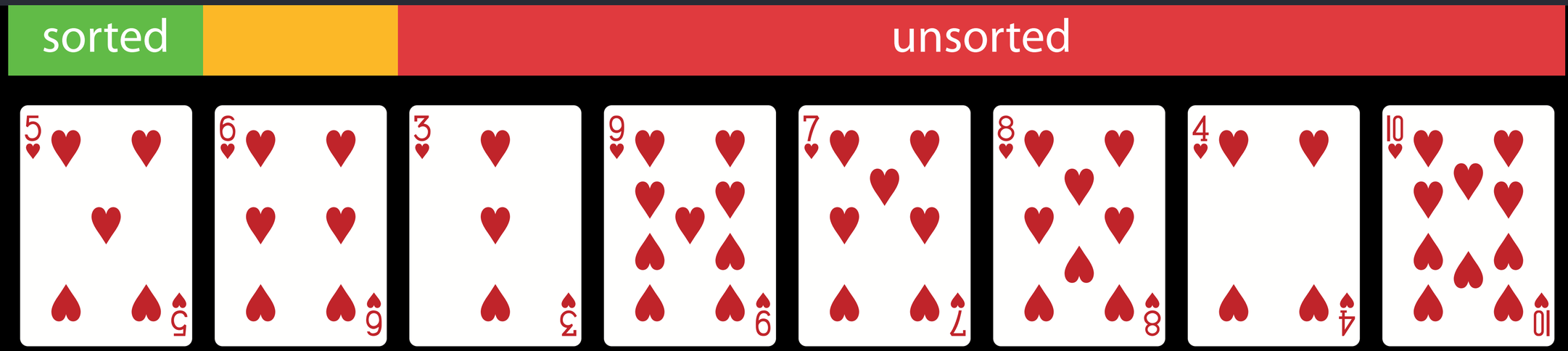
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



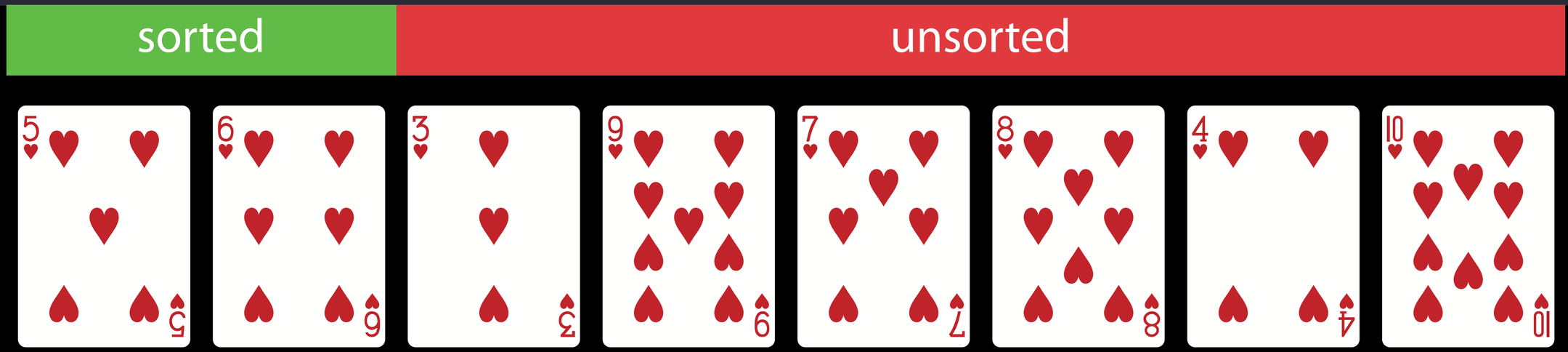
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



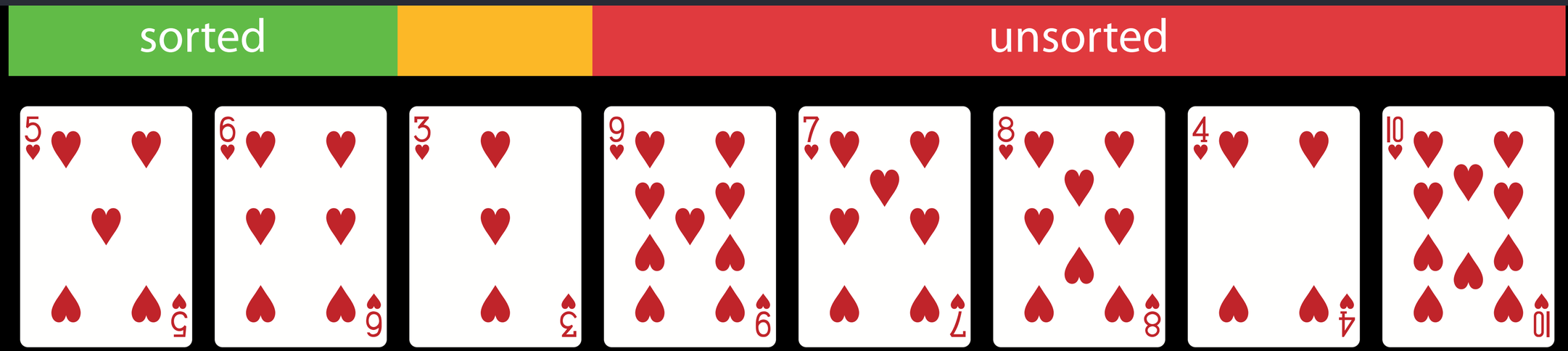
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



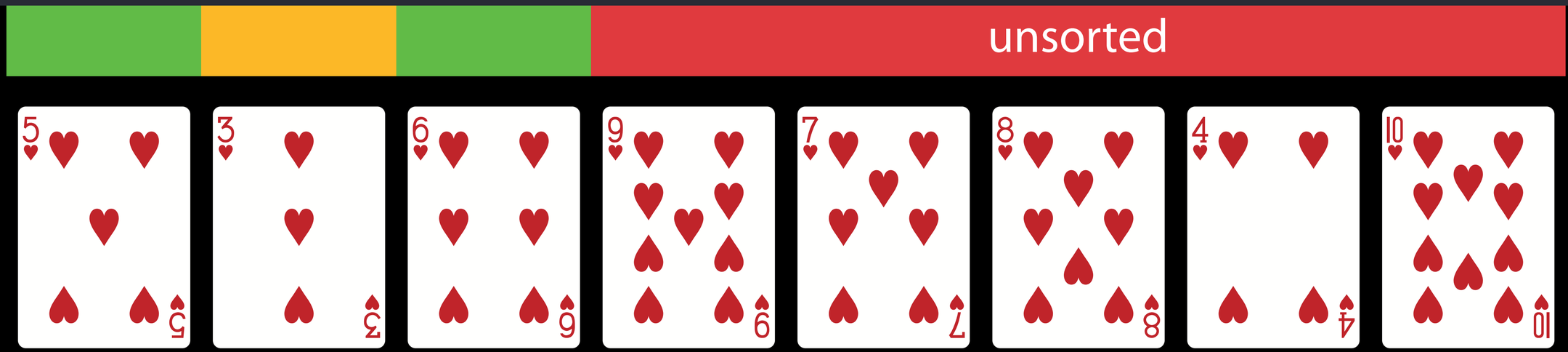
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



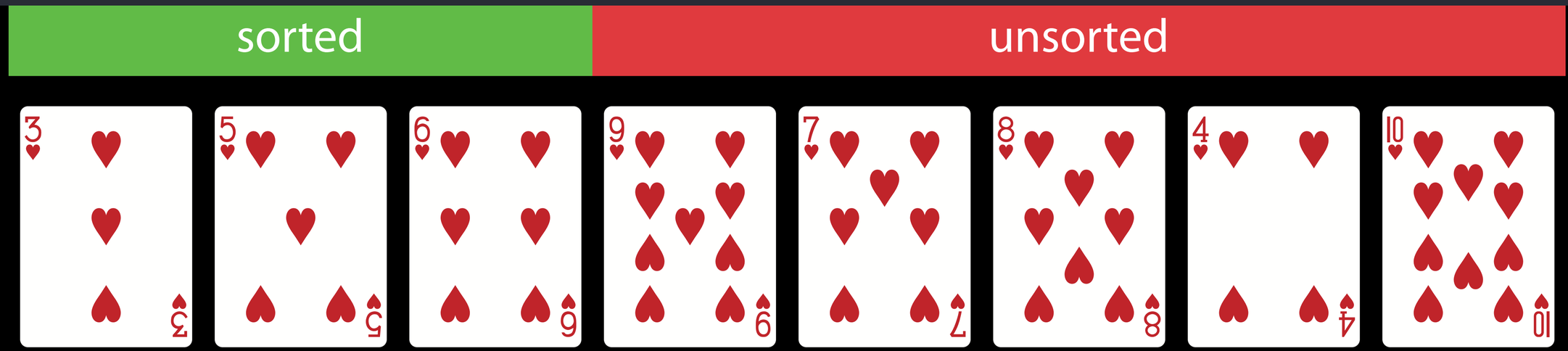
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



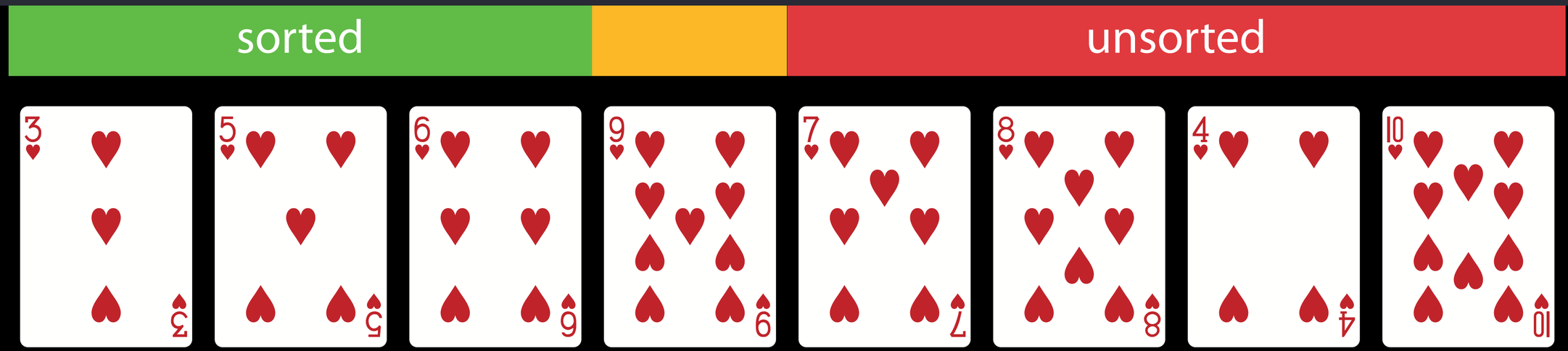
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



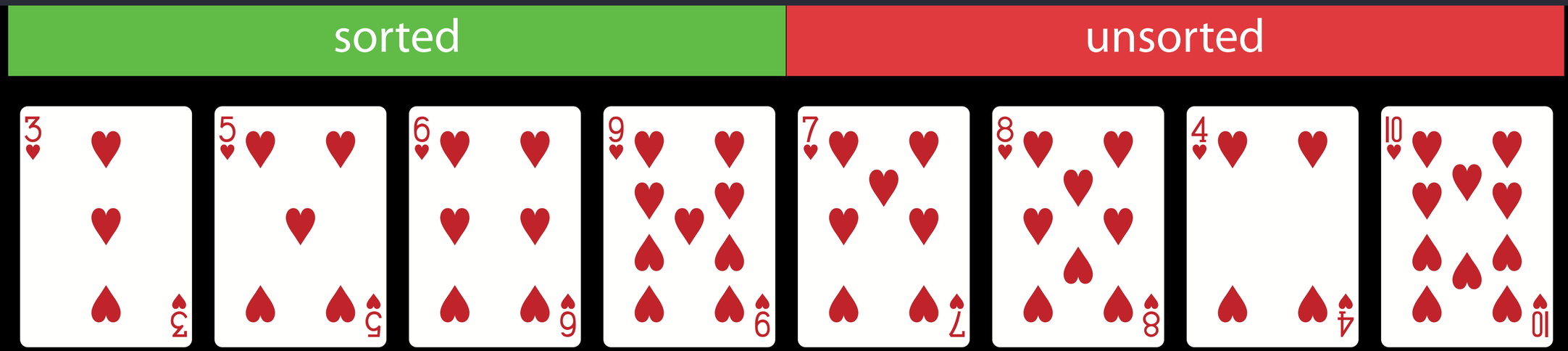
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



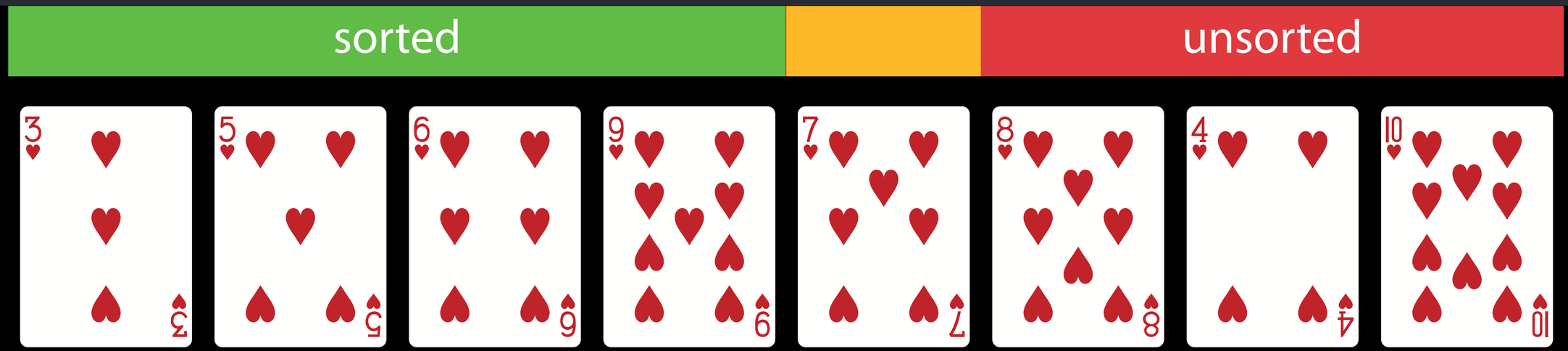
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



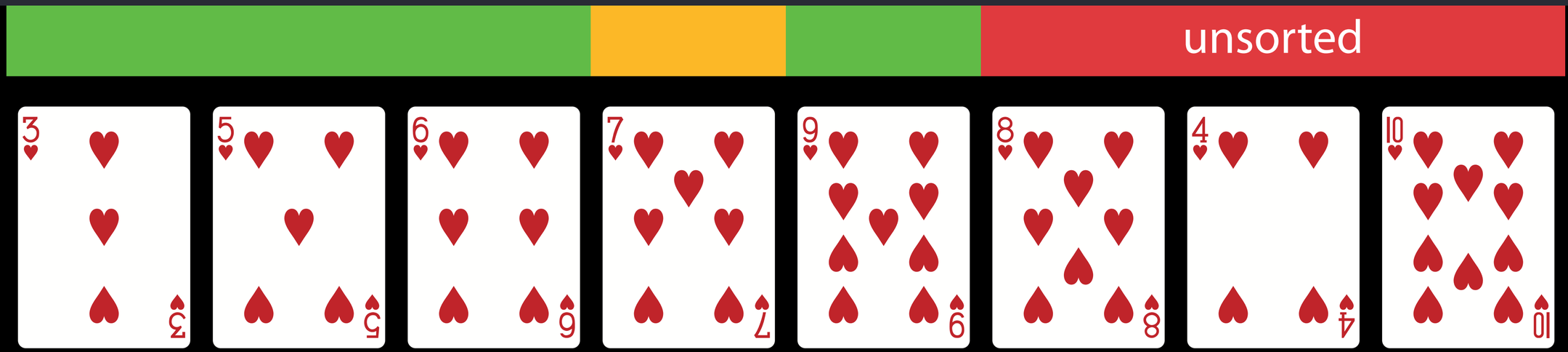
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



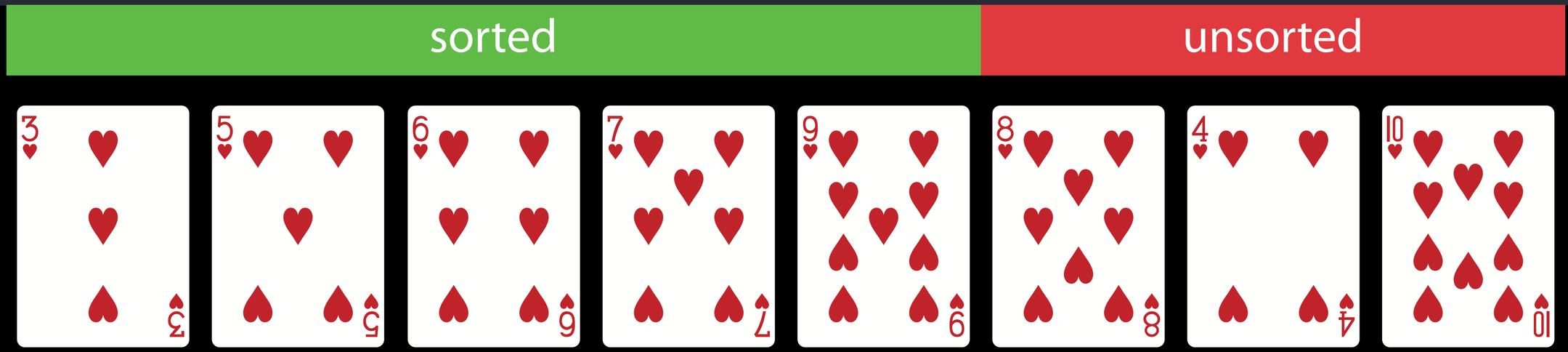
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



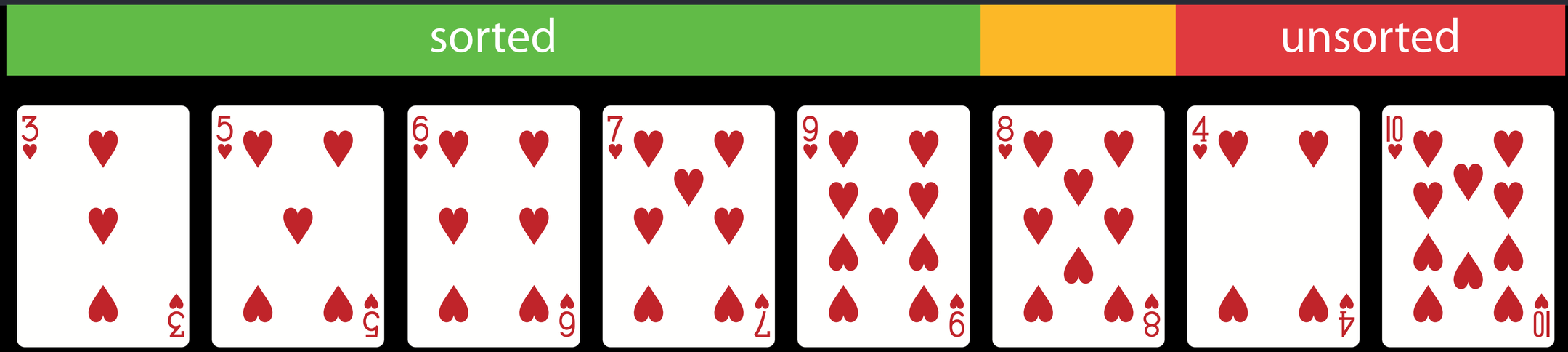
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



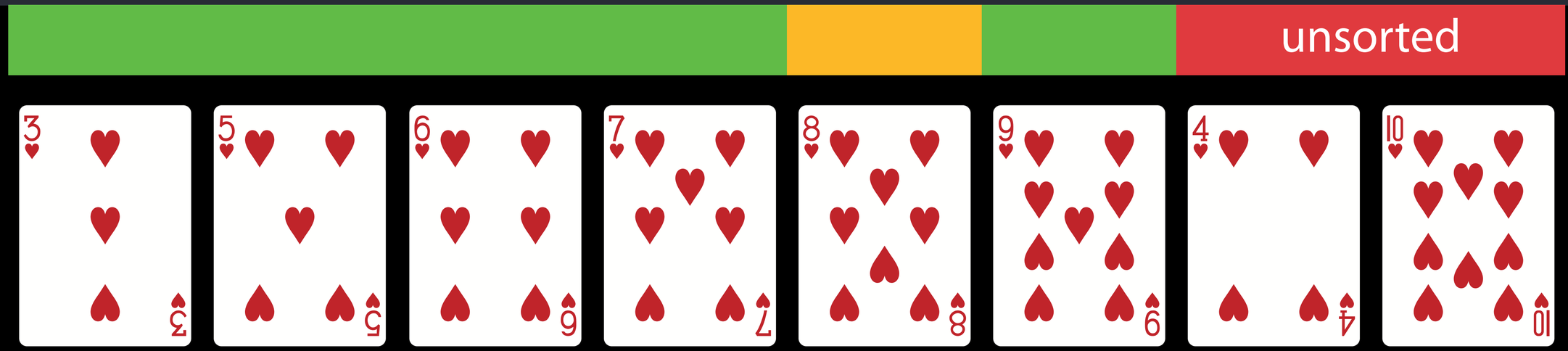
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



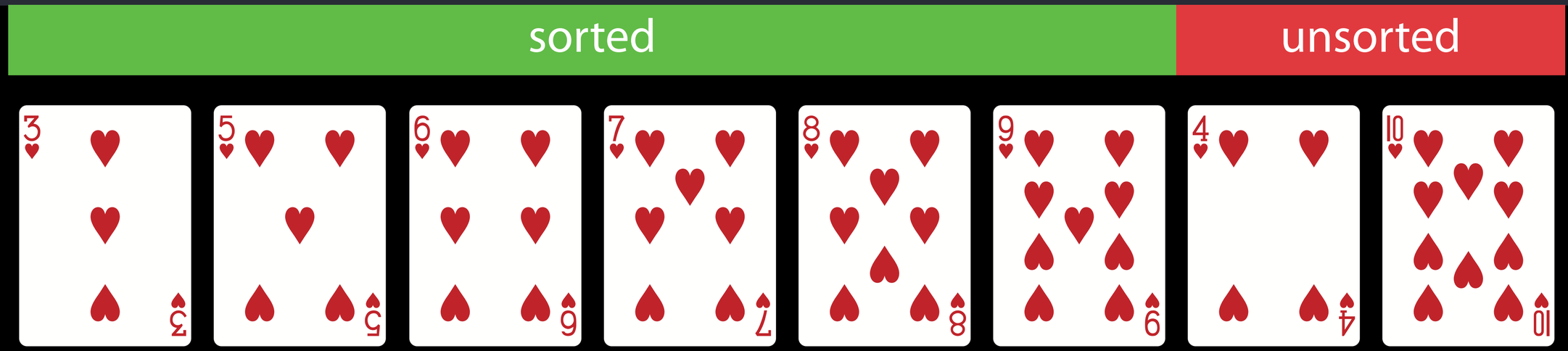
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



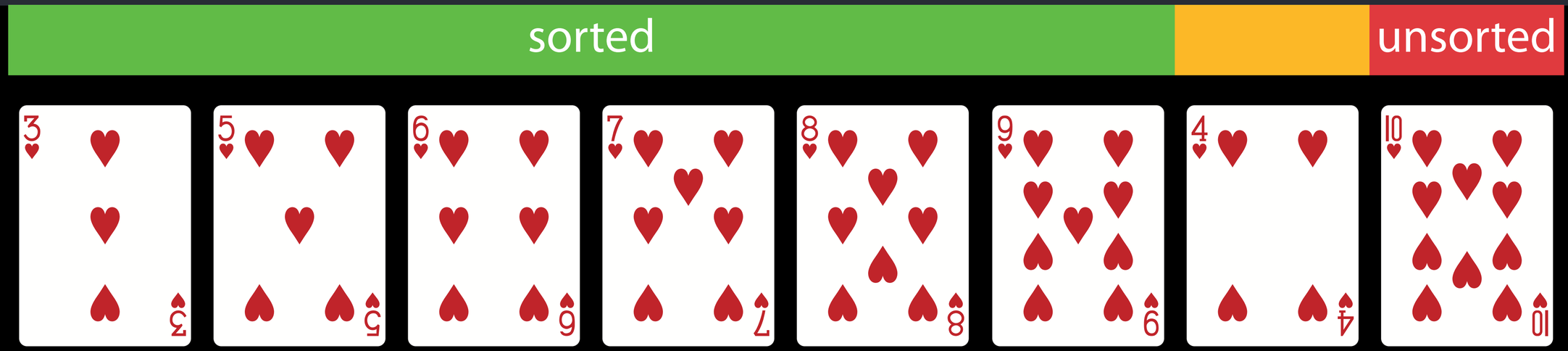
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



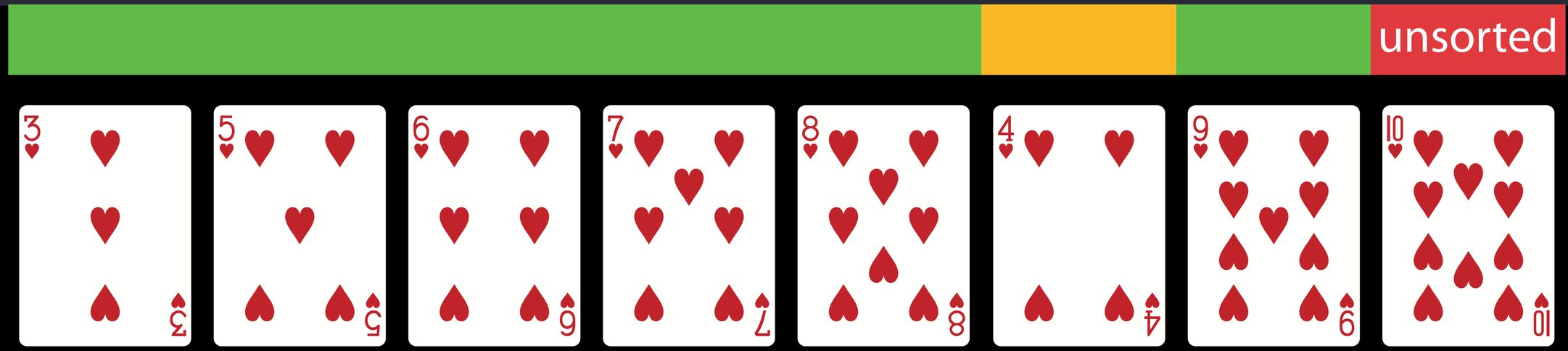
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



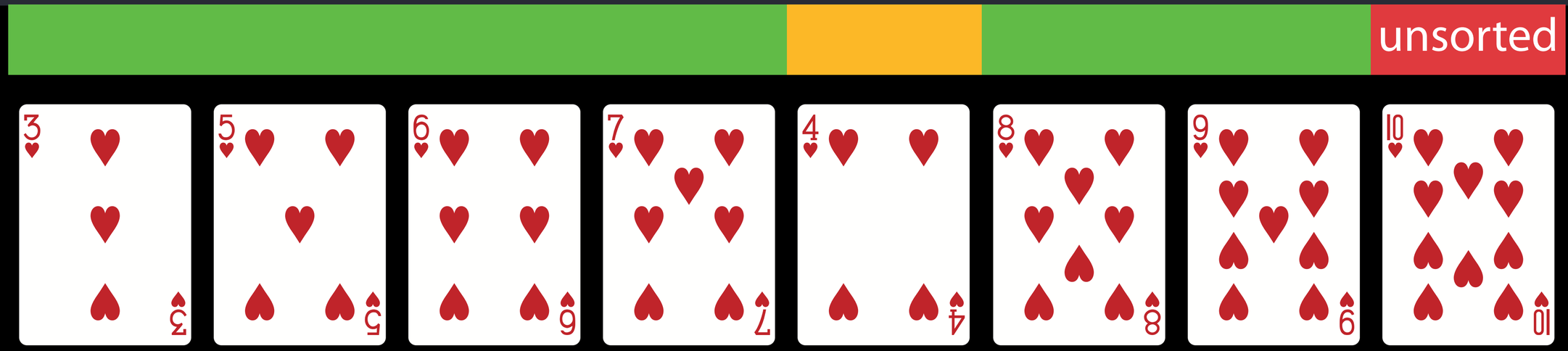
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



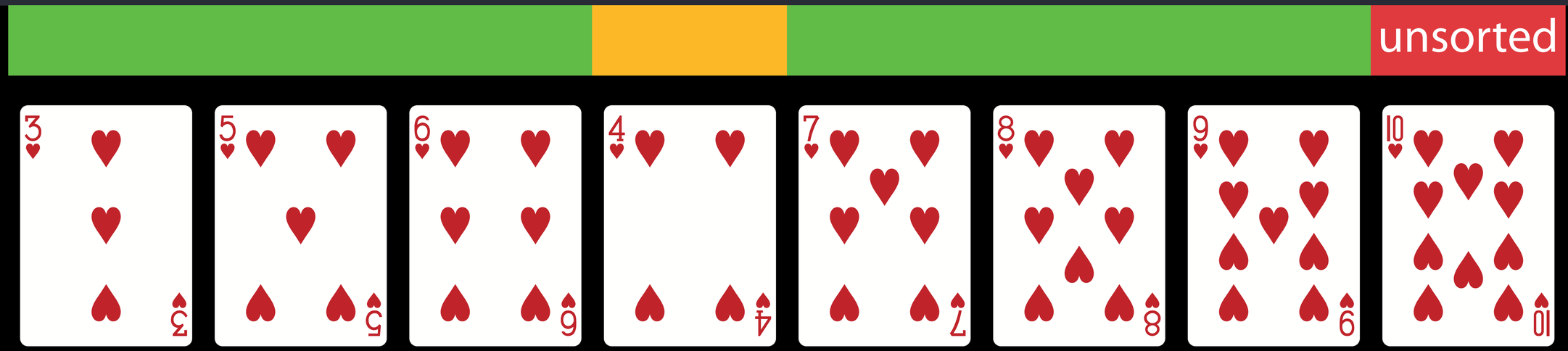
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



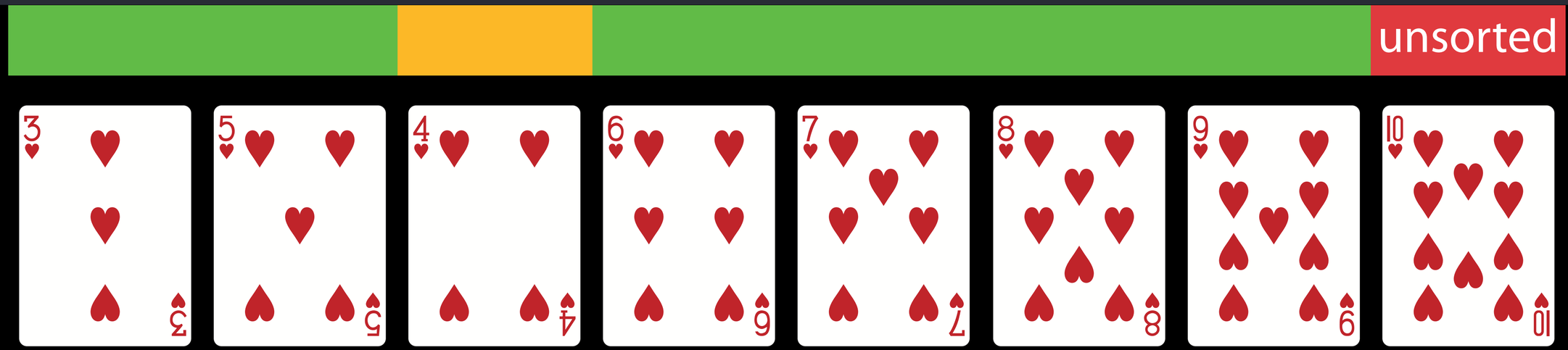
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



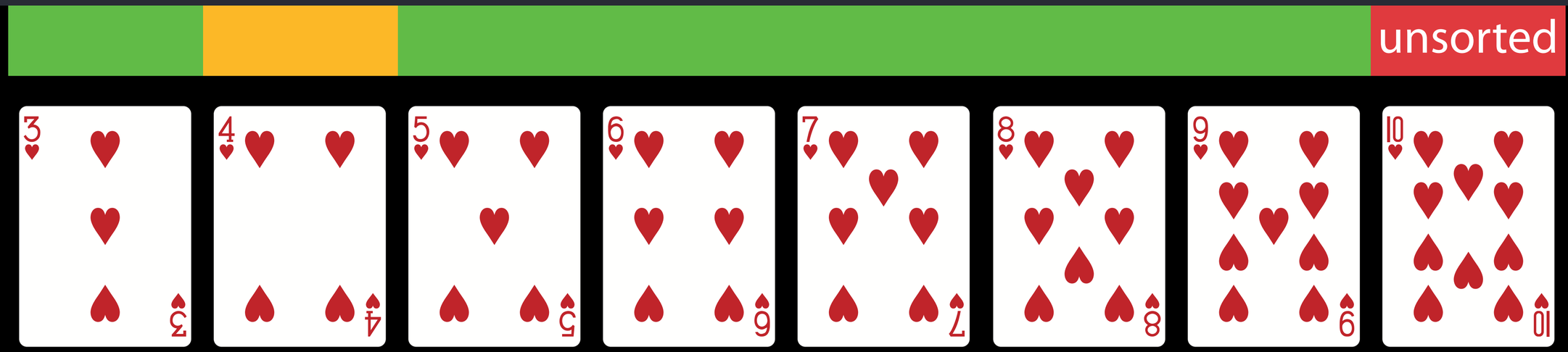
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



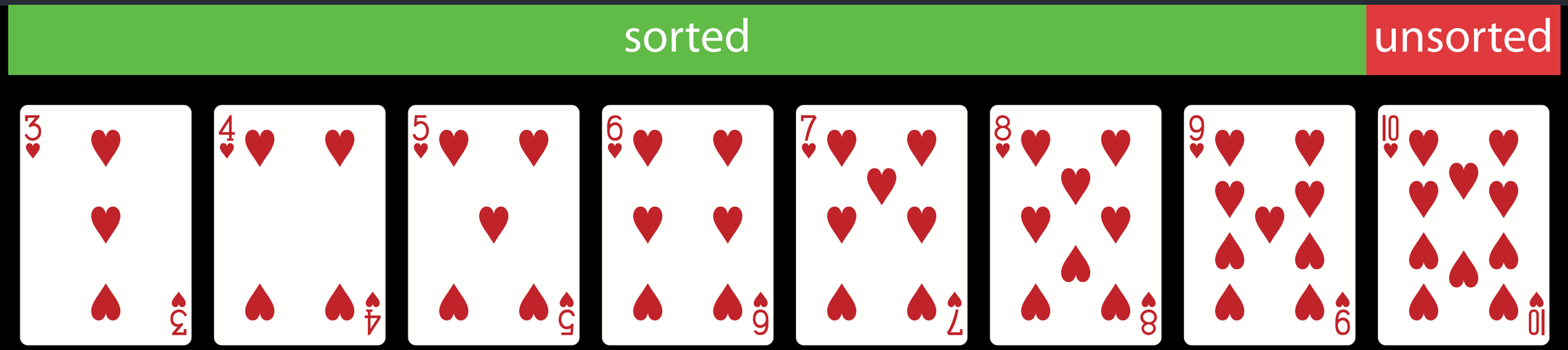
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



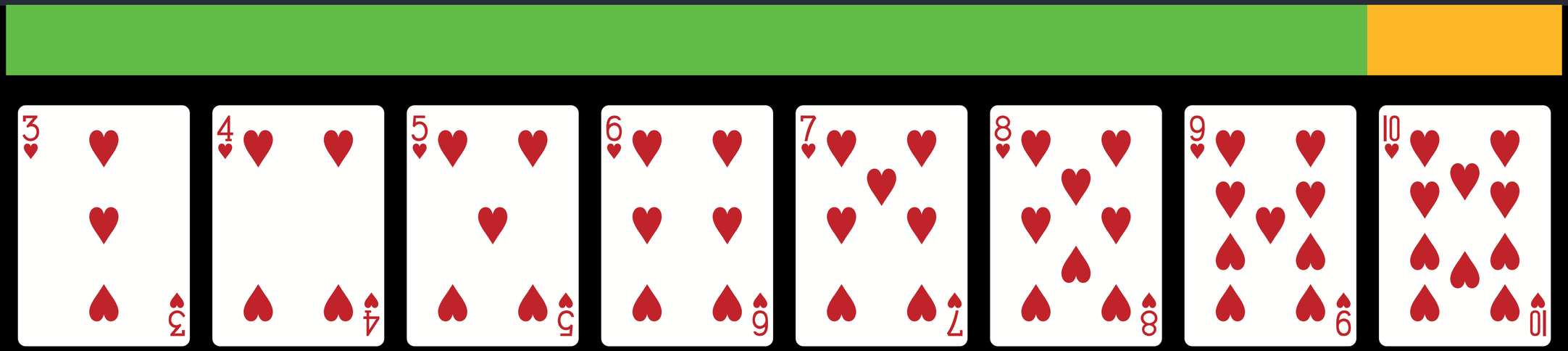
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



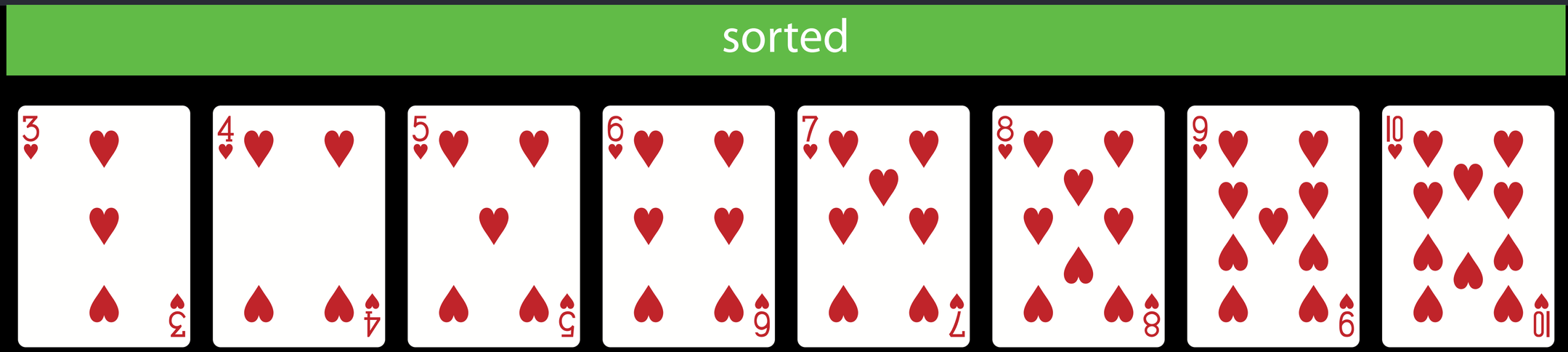
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```



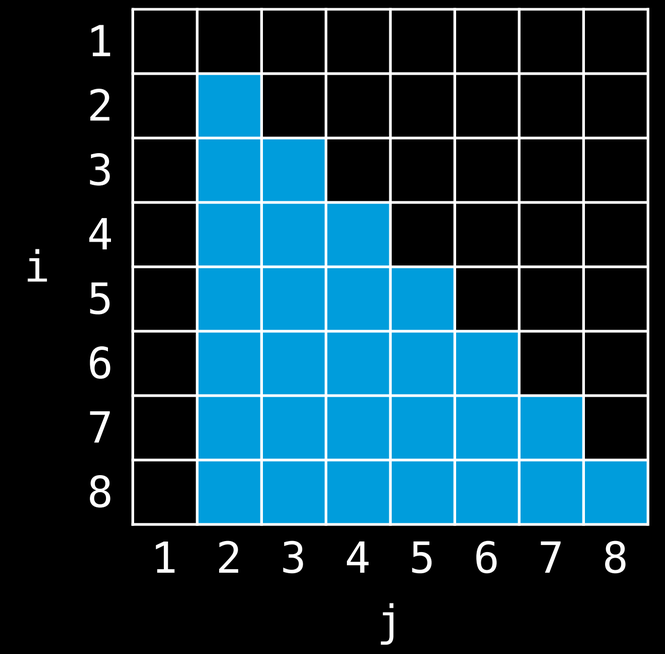
Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j – 1] > a[j])
        swap a[j] and a[j – 1]
        --j
```

What is the growth complexity of this algorithm?

Sorting

```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j − 1] > a[j])
        swap a[j] and a[j − 1]
        −−j
```



Sorting


```
Algorithm sort0(a[], N):
for i=2 to N
    j=i
    while (j > 1) and (a[j - 1] > a[j])
        swap a[j] and a[j - 1]
        --j
```

$T(n) = 2\,T\left(\frac{n}{2}\right) + n$

$a = 2$

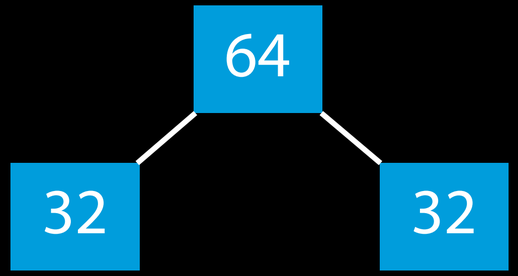
$b = 2$

$c = 1$

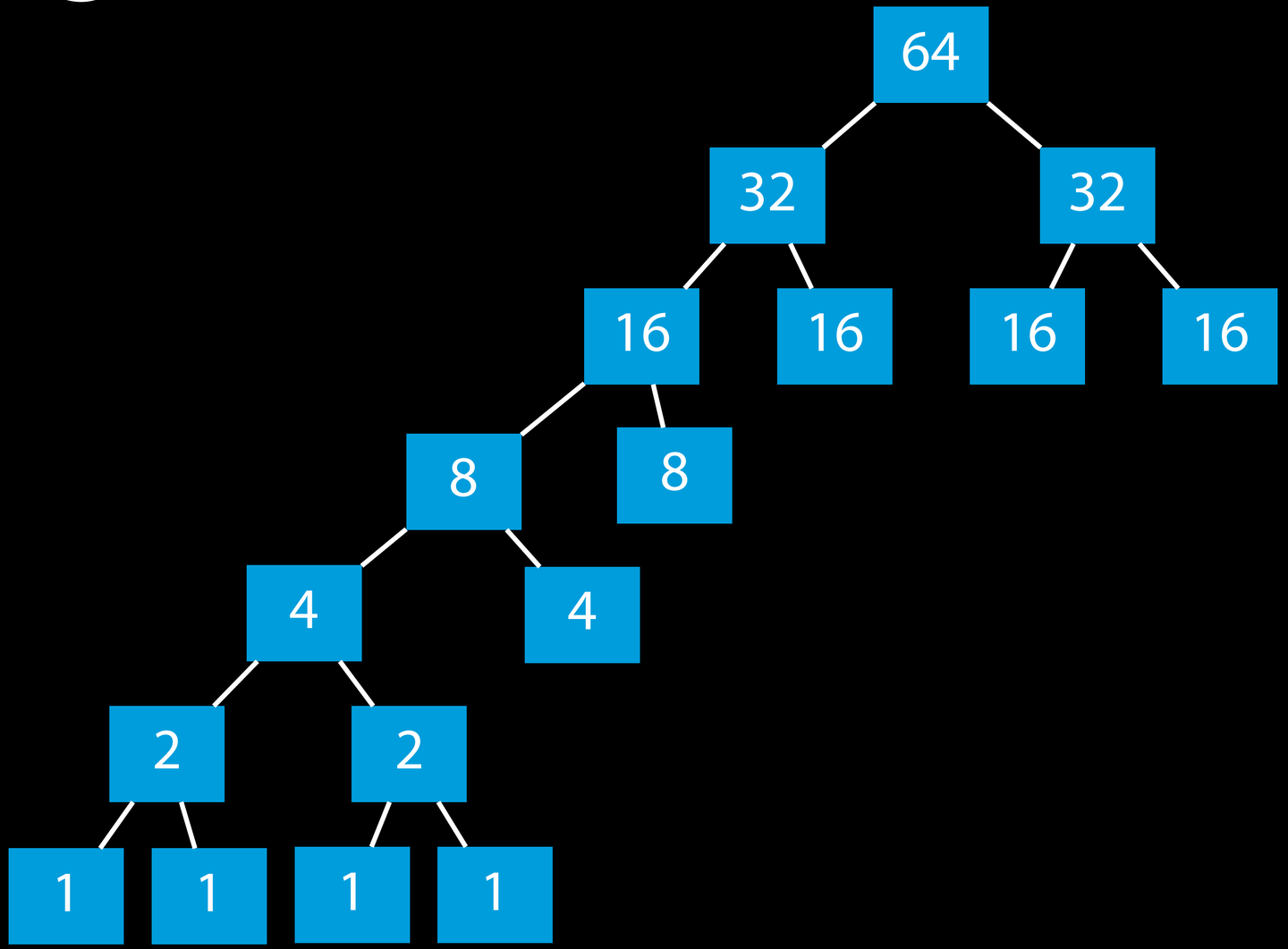
$2 = 2^1 \rightarrow$ case 2

Merge sort: $\Theta(n \log n)$

Sorting



Sorting



Sort/merge

merge

Implement **merge**, a function that combines the elements of two sorted vectors into a single vector and returns the result.

```
vector<int> merge(const vector<int> &a, const vector<int> &b);
```

For example, if a contains {1, 2, 4} and b contains {2, 3, 5}, **merge** should return a vector containing {1, 2, 2, 3, 4, 5}.

Templates

Function templates

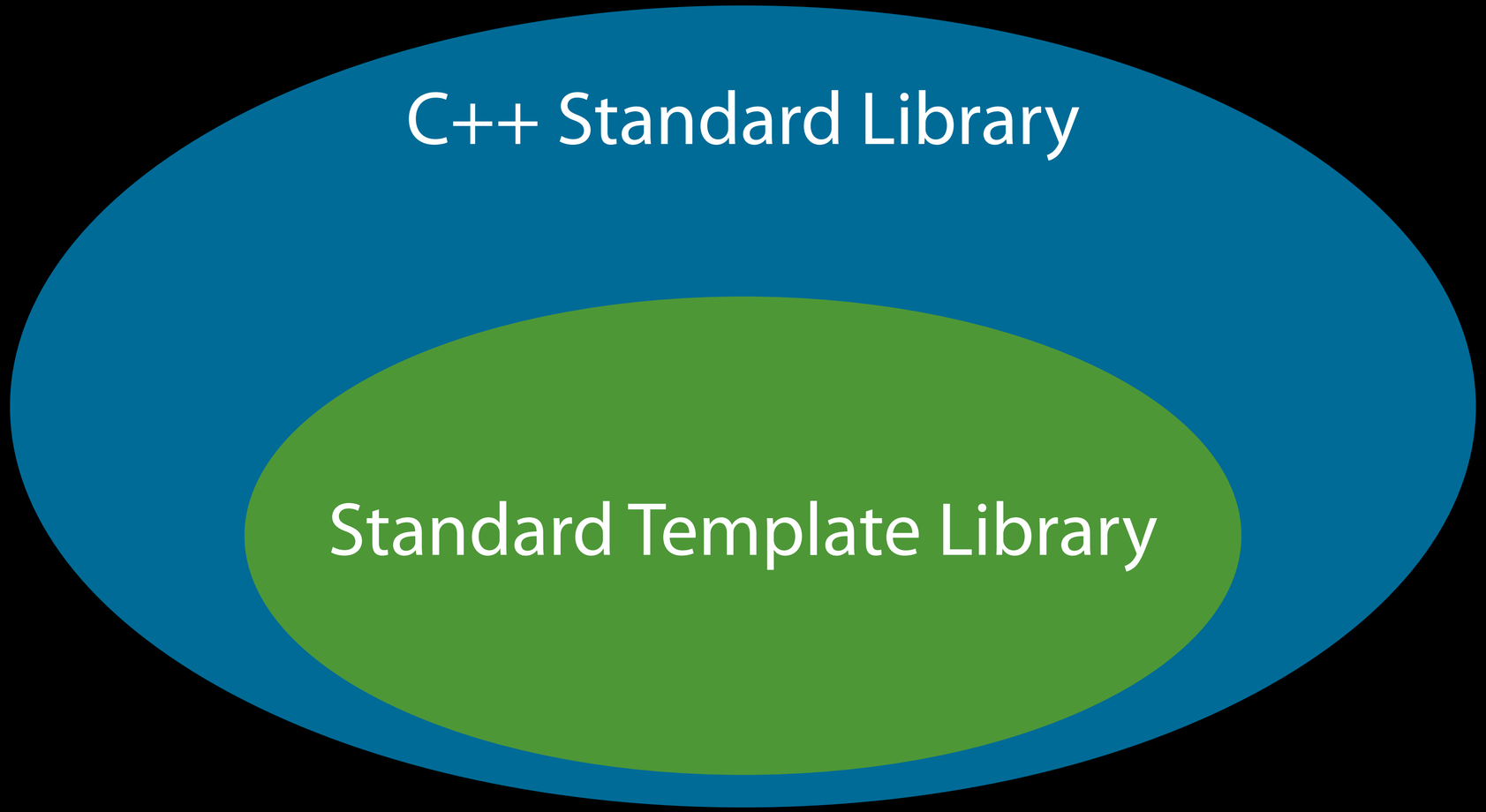
```
template <typename T>
void swapValues(T& a, T& b) {
    T temporaryA = a;
    a = b;
    b = temporaryA;
}
```

Class templates

```
template <typename T>
class NamedStack {
    vector<T> items;
    string name;
public:
    NamedStack(string newName): name(newName) {};
    void push(T newItem) {
        items.push_back(newItem);
    }
    // ...
};
```

Standard Template Library

Standard Template Library



Standard Template Library

Algorithms

Containers

Standard Template Library

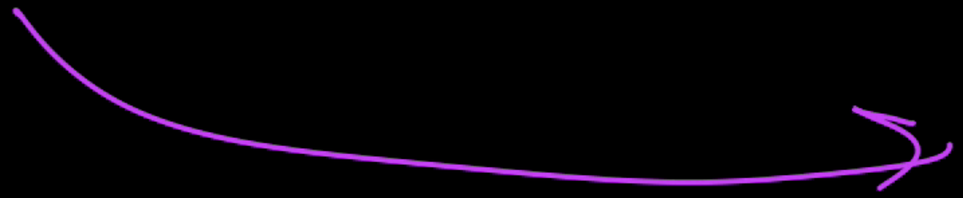
#: A

Algorithms

#: C

Containers

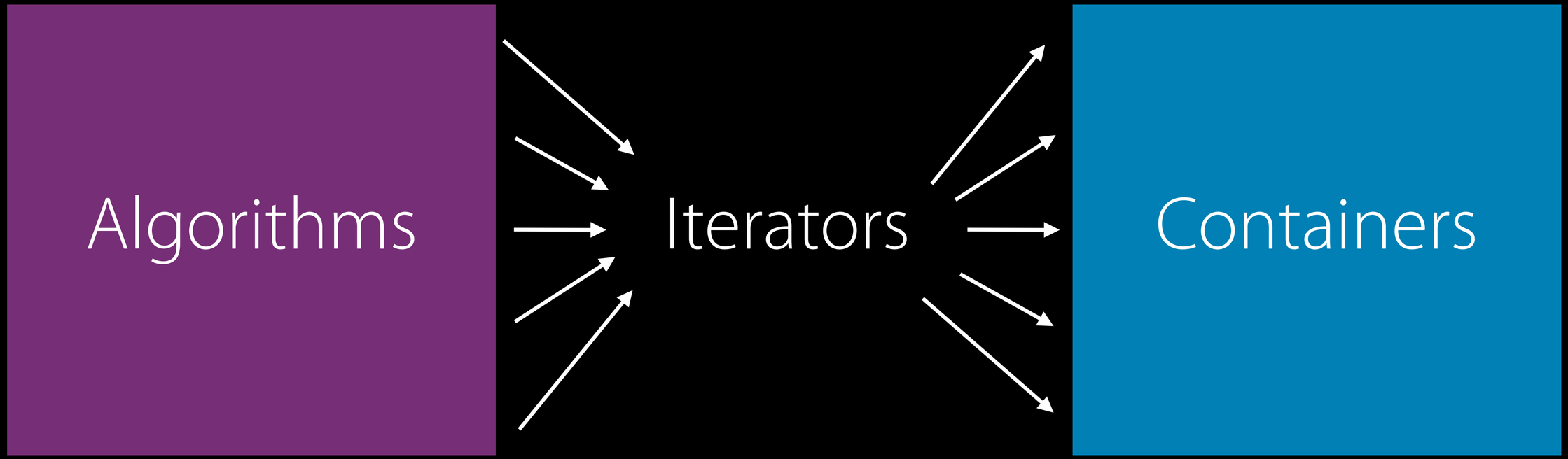
classes



member function

~~Total impl.: AxC~~

Standard Template Library



A + *C* implementations for *A* algorithms and *C* containers.

Reasons to use STL

Code reuse

Efficiency

Less buggy

Standardized, guaranteed availability

Easier to understand others' code

Good knowledge of data structures and algorithms

Iterators

Iterators

Random access **iterators:** random access

Bidirectional **iterators:** forward and backward moving

Forward **iterators:** forward moving

Input **iterators:** read only, forward moving

Output **iterators:** write only, forward moving

Iterators

STL containers

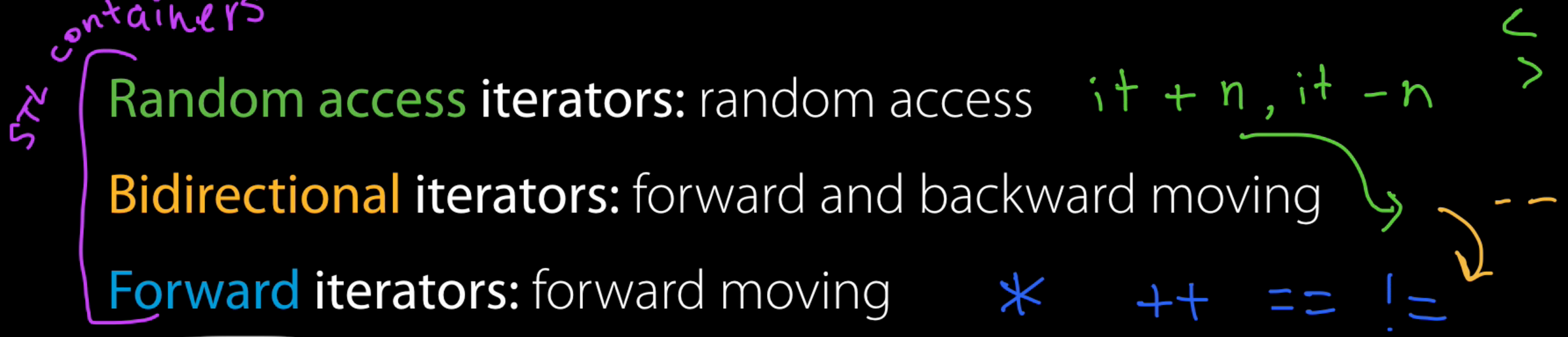
Random access **iterators:** random access

Bidirectional **iterators:** forward and backward moving

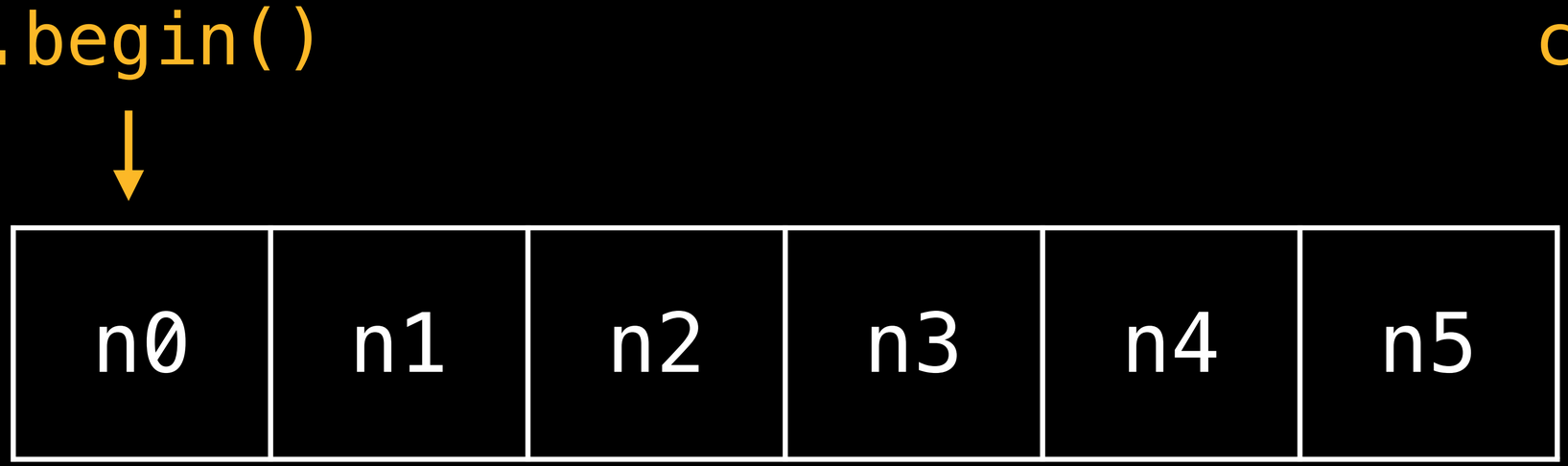
Forward **iterators:** forward moving

Input **iterators:** read only, forward moving

Output **iterators:** write only, forward moving



Iterators




```
// c is some container
for (auto it = c.begin(); it != c.end(); ++it) {
    cout << *it << " ";
}
```

Iterators

[begin, end) ← half-open range


```
// c is some container
for (auto it = c.begin(); it != c.end(); ++it) {
    cout << *it << " ";
}
```

Iterators

Pure abstraction!

Every container has an `iterator` and a `const_iterator`

Iterators

Pure abstraction!

Every container has an `iterator` and a `const_iterator`

```
vector<int>::iterator it = v.begin()
vector<int>::const_iterator cit = v.cbegin()
```

Iterator functions

```
advance(it, 5);
distance(it1, it2);
```

Iterator adapters

Insert iterators

Stream iterators

Reverse iterators

Move iterators

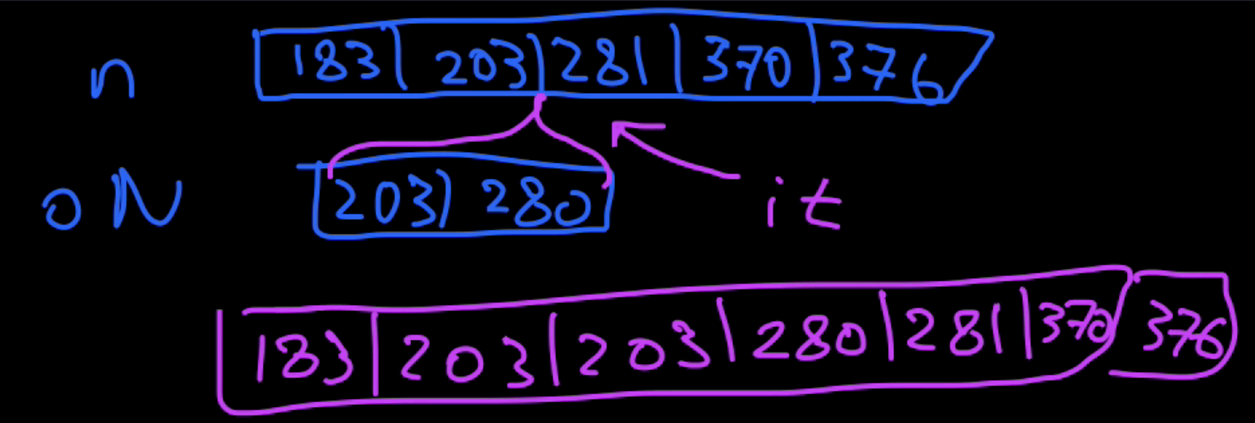
Insert iterators

```
vector<int> numbers = {183, 203, 281, 370, 376};
vector<int> otherNumbers = {203, 280};
vector<int>::iterator it = find(numbers.begin(),
                                numbers.begin(), 281);
insert_iterator<vector<int>> insertIt(numbers, it);
copy(numbers.begin(), numbers.end(), insertIt);
```

Insert iterators

#include <algorithm>

```
vector<int> numbers = {183, 203, 281, 370, 376};
vector<int> otherNumbers = {203, 280};
vector<int>::iterator it = find(numbers.begin(),
                                numbers.begin(), 281);
insert_iterator<vector<int>> insertIt(numbers, it);
copy(numbers.begin(), numbers.end(), insertIt);
```



Stream iterators

```
vector<int> inputNumbers;
copy(istream_iterator<int>(cin),
     istream_iterator<int>(),
     back_inserter(inputNumbers));

copy(istream_iterator<int>(cin),
     istream_iterator<int>(),
     ostream_iterator<int>(cout, "\n"));
```

Reverse iterators

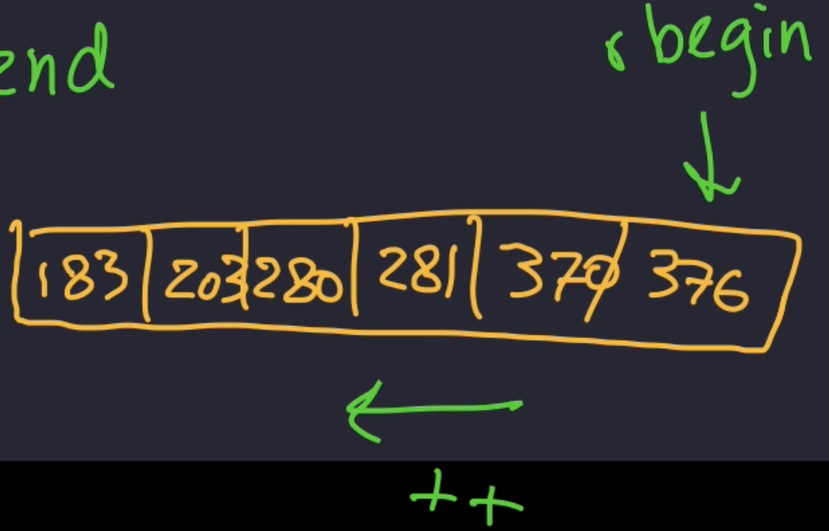
```
vector<int> numbers = {183, 203, 280, 281, 370, 376};
reverse_iterator<vector<int>::iterator> reverseIt;

for (reverseIt = numbers.rbegin();
     reverseIt != numbers.rend();
     ++reverseIt) {
    cout << *reverseIt << endl;
}
```

Reverse iterators

```
vector<int> numbers = {183, 203, 280, 281, 370, 376};
reverse_iterator<vector<int>::iterator> reverseIt;

for (reverseIt = numbers.rbegin();
     reverseIt != numbers.rend();
     ++reverseIt) {
    cout << *reverseIt << endl;
}
```



Containers

Containers

Sequence containers:

`vector, deque, list, forward list, array`

Associative containers:

`(multi)set, (multi)map`

Unordered associative containers:

`unordered_(multi)set, unordered_(multi)map`

Container adapters:

`queue, stack, priority queue`

Containers

random access it

Sequence containers:

`vector, deque, list, forward list, array, string`

bidir it

forward it

Associative containers:

`(multi)set, (multi)map`

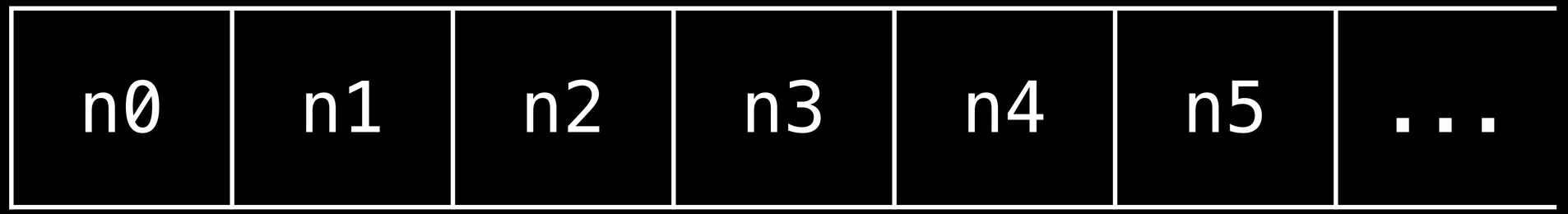
Unordered associative containers:

`unordered_(multi)set, unordered_(multi)map`

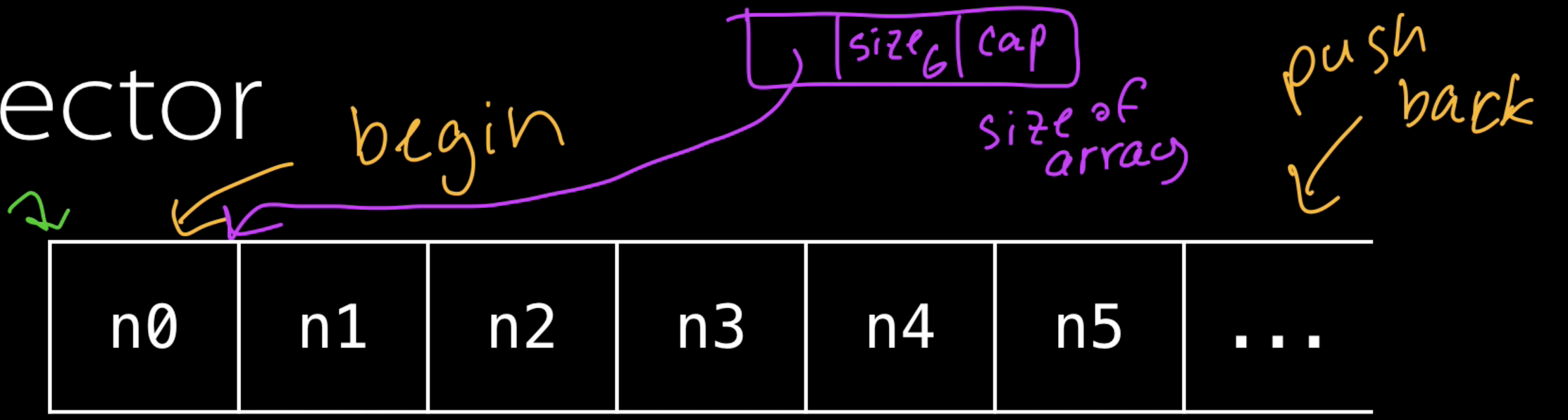
Container adapters:

`queue, stack, priority queue`

Vector



Vector



contiguous block of memory

resize
reserve

size

capacity

insert / remove

beginning O(n)

middle O(n)

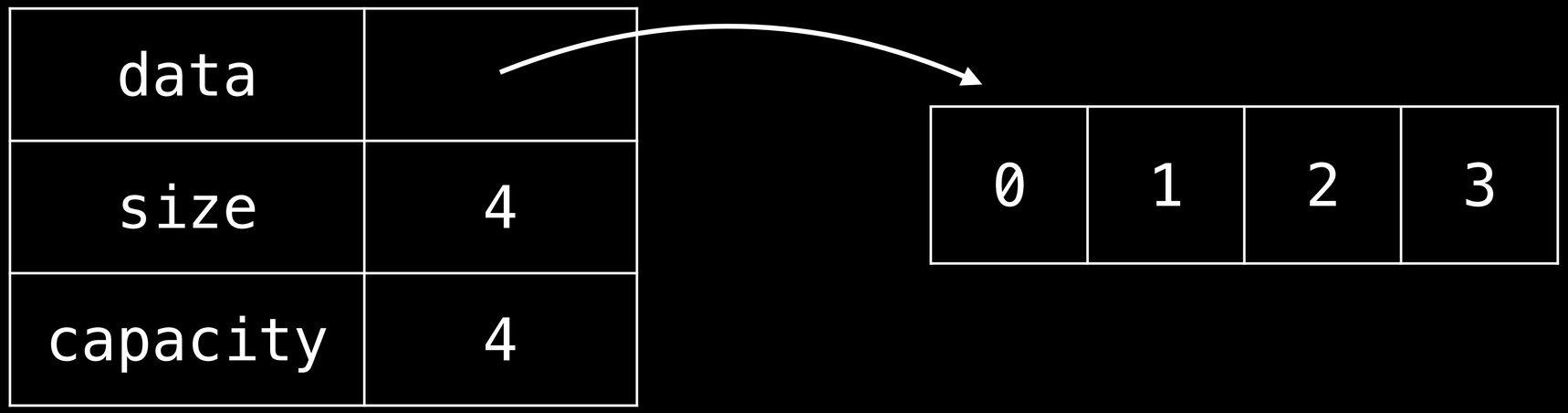
end O(1)

```
int * arr = &v[0];
* (arr + 1);
```

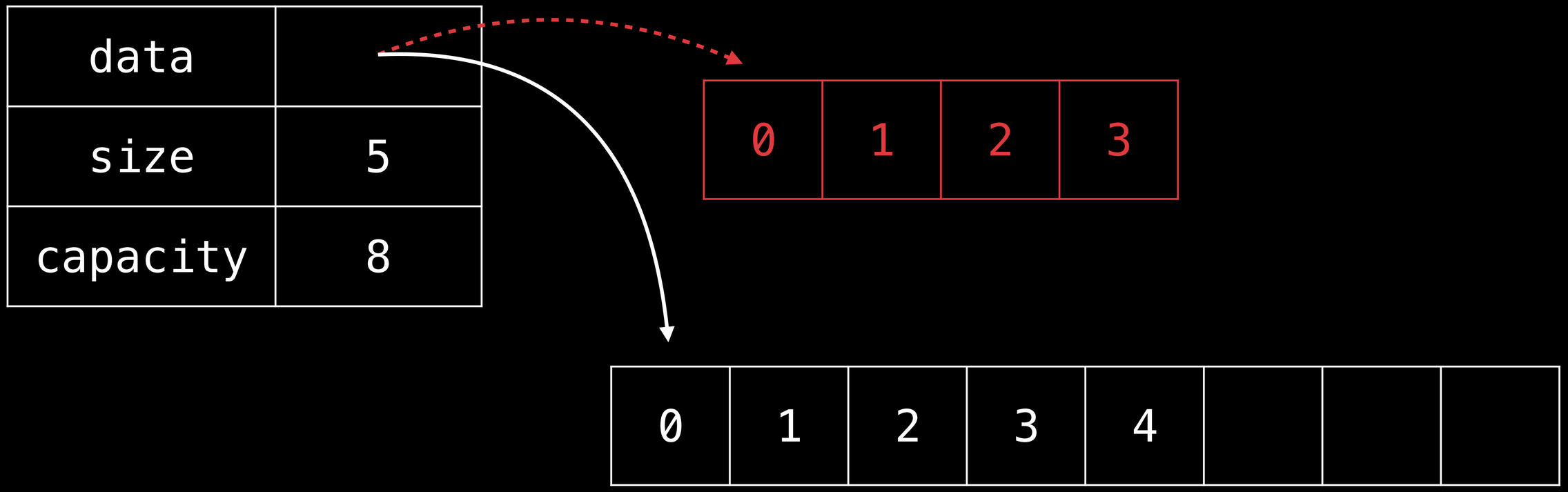
search: O(n)

vector::push_back()

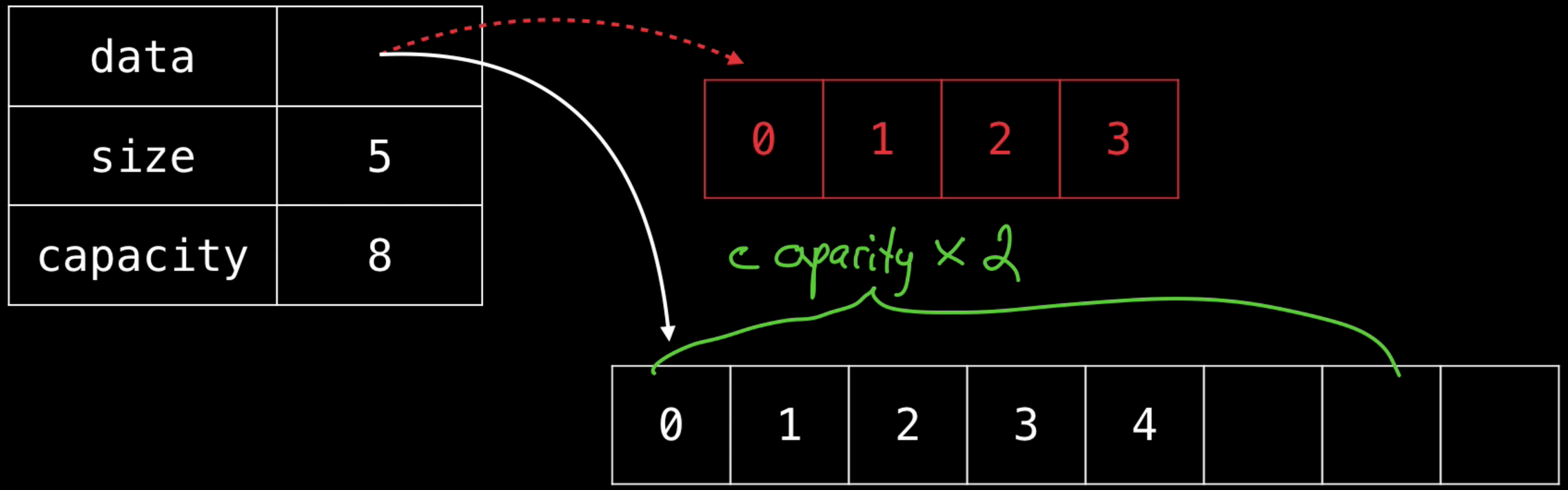
vector::push_back()



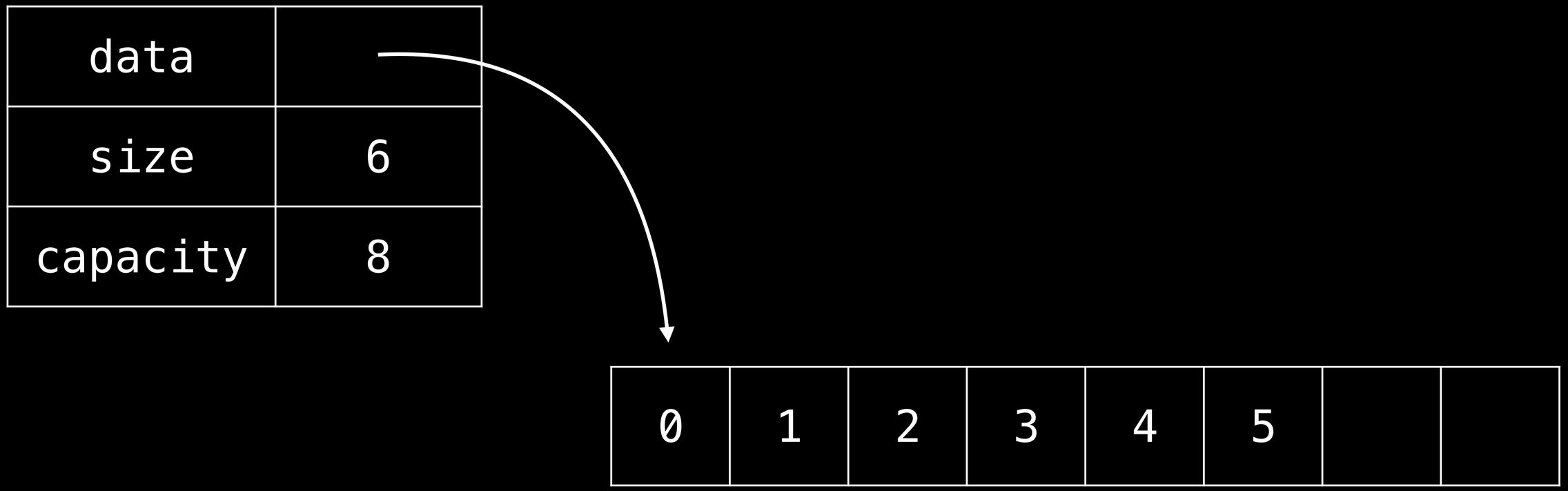
vector::push_back()



vector::push_back()



vector::push_back()



vector::push_back()

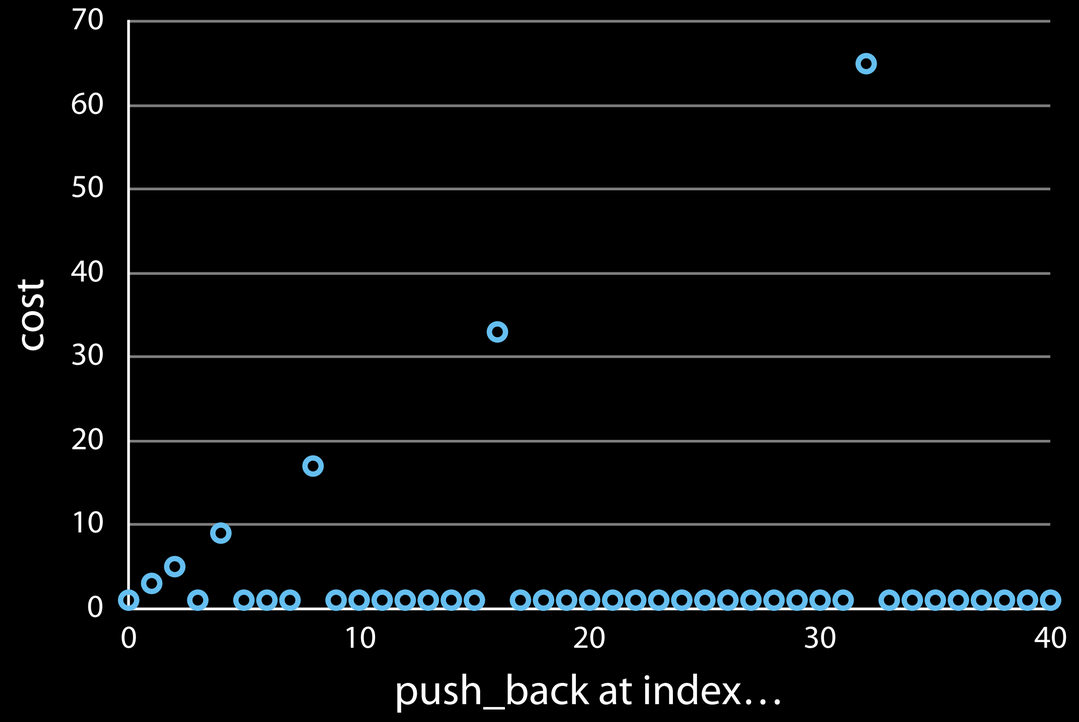
push_back at i…	0	1	2	3	4	5	6	7	8
size									
capacity									
write cost									
resize cost									
total cost									
cumulative cost									
average cost									

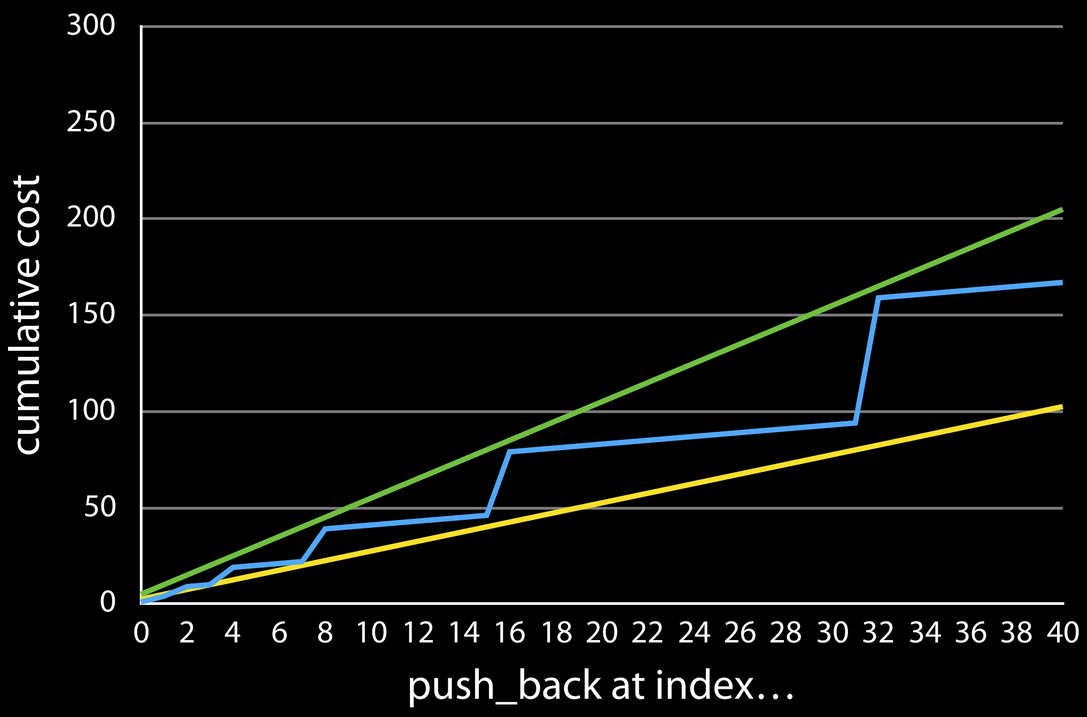
vector::push_back()

push_back at i…	0	1	2	3	4	5	6	7	8
size	1	2	3	4	5	6	7	8	9
capacity	1	2	4	4	8	8	8	8	16
write cost	1	1	1	1	1	1	1	1	1
resize cost	0	2	4	0	8	0	0	0	16
total cost	1	3	5	1	9	1	1	1	17
cumulative cost	1	4	9	10	19	20	21	22	39
average cost	1.00	2.00	3.00	2.50	3.80	3.33	3.00	2.75	4.33

Amortized complexity

Average the time required to perform a sequence of operations over all the operations performed.





Iterator invalidation

`vector`
iterators, pointers and references are invalidated after insertion if new container size is greater than the previous capacity.

`deque`
iterators, pointers and references are invalidated after insertion (unless inserting at the front/end, then references are unaffected).

`list`
iterators, pointers and references are unaffected.

Algorithms

```
#include <algorithm>
```

Algorithms

(Mostly loops)

```
min_element
sort
reverse
copy
find_if
for_each
```

…

Anagrams

Implement **`areAnagrams`**, a function that checks if s1 and s2 are anagrams of each other.

```
bool areAnagrams(const string& s1, const string& s2);
```