Getting Ready for Canadian Computing Competition

Class 3
December 14, 2019

Outline

- CCC 2015 Senior Q1
- Vectors
  - LIFO
  - Vector operations, erase(), iterators
- Functions
  - Function declarations
  - Functions calls
  - Passing by value
  - Scopes
  - Function overloading
  - Default arguments
- Bubble Sort
- Recursion

2015 Senior Q1

Output for Sample Input 1

```
0
```

Sample Input 2

```
10
1
3
5
4
0
0
7
0
0
6
```

Output for Sample Input 2

```
7
```

Problem S1: Zero That Out

Problem Description

Your boss has asked you to add up a sequence of positive numbers to determine how much money your company made last year.

Unfortunately, your boss reads out numbers incorrectly from time to time.

Fortunately, your boss realizes when an incorrect number is read and says "zero", meaning "ignore the current last number."

Unfortunately, your boss can make repeated mistakes, and says "zero" for each mistake.

For example, your boss may say "One, three, five, four, zero, zero, seven, zero, zero, six", which means the total is 7 as explained in the following chart:

Boss statement(s)	Current numbers	Explanation
"One, three, five, four"	1, 3, 5, 4	Record the first four numbers.
"zero, zero"	1, 3	Ignore the last two numbers.
"seven"	1, 3, 7	Record the number 7 at the end of our list.
"zero, zero"	1	Ignore the last two numbers.
"six"	1, 6	We have read all numbers, and the total is 7.

At any point, your boss will have said at least as many positive numbers as "zero" statements. If all positive numbers have been ignored, the sum is zero.

Write a program that reads the sequence of boss statements and computes the correct sum.

Input Specification

The first line of input contains the integer K ($1 \leq K \leq 100\,000$) which is the number of integers (including "zero") your boss will say. On each of the next K lines, there will either be one integer between 1 and 100 (inclusive), or the integer 0.

Output Specification

The output is one line, containing the integer which is the correct sum of the integers read, taking the "zero" statements into consideration. You can assume that the output will be an integer in the range 0 and 1 000 000 (inclusive).

Sample Input 1

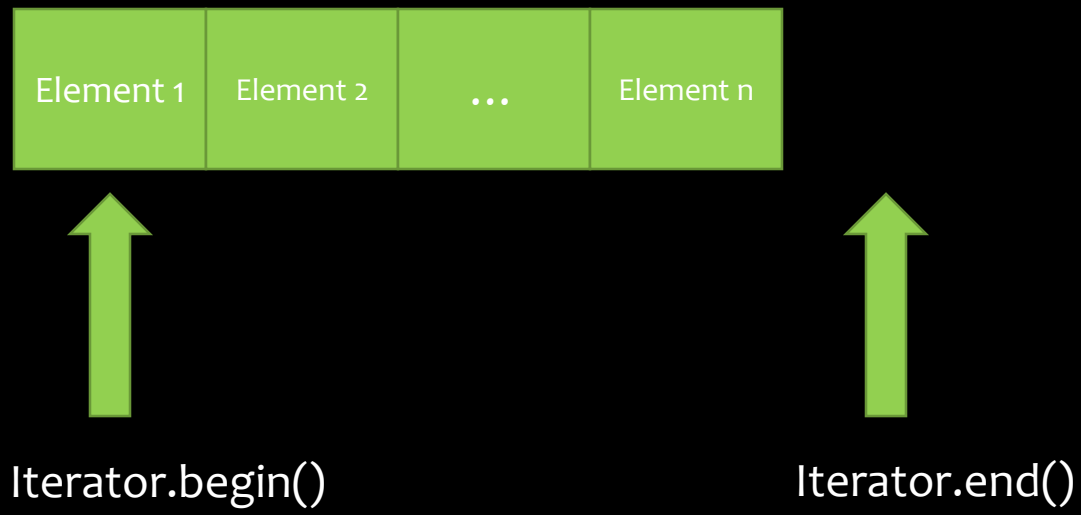
```
4
3
0
4
0
```

More Vector Operations

- vec.size() - returns number of elements in vec
- vec.clear() - removes all elements from vector and leave the vector with a size of 0
- vec.erase(startPos, endPos) - removes element from startPos (inclusive) to endPos (exclusive)
  - must use iterator

Vector Iterators

- vec.begin() – iterator for beginning of vector (like a pointer)
- vec.end() – iterator for end of vector (like a pointer)
  - only use iterator for vec.erase(start, end) for now



Example 1

This will remove every element in myIntVect from index 1 to the end

```
vector <int> myIntVect = {1, 2, 3, 4, 5};

myIntVect.erase(myIntVect.begin() + 1, myIntVect.end());


// new myIntVect is {1}
```

Example 2

This will remove the second element

```
vector <int> myIntVect = {1, 2, 3, 4, 5};

myIntVect.erase(myIntVect.begin() + 1, myIntVect.begin() + 2);


// new myIntVect is {1, 3, 4, 5}
```

Exercise

Given a vector of an unknown size, remove all elements from the beginning to the middle. If there are an odd number of elements, keep the middle.

For example, if there were 5 elements, remove elements at index 0 and 1. If there were 6, remove elements at index 0, 1, 2.

Practice – 2D String Vectors

Create a 2d vector of size nxn storing a chess board. Each cell is black. Use ‘b’ to denote black. Then print the board.

Sample Input:

3

Sample output:

bbb

bbb

bbb

Practice – 2D String Vectors (hw)

Correct the board to contain alternating ‘b’ and ‘w’

Sample Input:

5

Sample Output:

bwbwb

wbwbw

bwbwb

wbwbw

bwbwb

Functions

Functions

- A block of code that runs when it is called
- Can be reused
- You can pass parameters into a function
- Functions can return values
- main() is a pre-defined function that is executed without being manually called
- Main returns 0 if execution is successful (they are omitted in the slides to save space, but it is highly recommended that you had them in your own programs)
- All functions must be declared before the main

Function Declaration

returnType *functionName*(arg1Type arg1Name, … , argnType argnName) {
// function definition

return … (type must match returnType)
}

Ex). This function gets an int and returns the int

```
int foo (int n) {

    return n;

}
```

Function Declaration

- If a function return nothing, we call it a void function:

```
void functionName(arg1Type arg1Name, …, argnType argnName ) {
          // function definition, no return statement
}
```

- A function can also have no arguments:

```
void functionName( ) {
          // function definition, no return statement
}
```

Function Calling

- Syntax

```
functionName (arguments);
```

- The number of **arguments** supplied must equal the number of **parameters** of the function

```
void myFunction(int n) {
  cout << "you passed me " << n << endl;
}

int main() {
  myFunction(5);

  return 0;
}
```

prints: you passed me 5

Function Calling

- Syntax

```
functionName (arguments);
```

- The number of **arguments** supplied must equal the number of **parameters** of the function

```
void myFunction(int n) {
  cout << "you passed me " << n << endl;
}

int main() {
  myFunction(5); // calling myFunction with arugment 5 for the parameter n

  return 0;
}
```

prints: you passed me 5

Examples

Example of a function that returns a value:

```
int myFunction(int n) {
  cout << "I am going to return " << n << endl;

  return n;
}

int main() {
  cout << myFunction(5) << endl;
  return 0;
}
```

What is printed?

A.
I am going to return 5

B.
I am going to return 5
5

C.
5
0

D.
I am going to return 5 5

Examples

Example of a function with no parameters

```
int myFunction() {

  int n = 5;

  return n;
}

int main() {
  cout << myFunction(); // call the function with no arguments
  return 0;
}
```

What is printed?

A. N = 5 B. nothing c. 0 D. 5

Exercise

```
void myFunction() {
}

int main() {
  cout << myFunction();
  return 0;
}
```

What happens if you run this code?

Exercise

```
void myFunction() {
}

int main() {
  cout << myFunction();
  return 0;
}
```

What happens if you run this code?

ERROR, cout cannot handle the value of myFunction()

Repeated Function Calls

A function can be called multiple times:

```
void myFunction() {
  cout << "I just got called" << endl;
}

int main() {
  for (int i = 0; i < 3; ++i) {
    myFunction();
  }
  return 0;
}
```

Repeated Function Calls

A function can be called multiple times:

```
void myFunction() {
  cout << "I just got called" << endl;
}

int main() {
  for (int i = 0; i < 3; ++i) {
    myFunction();
  }
  return 0;
}
```

Prints:
I just got called
I just got called
I just got called

Practice (hw)

Recall the vector chess board problem:

Create a 2d vector of size n by n storing a chess board.

Part 1: Each cell is black. Use “b” to denote black.

Part 2: print the board

Part 3: Now correct the board to contain alternating “b” and “w”.

Place parts 1 and 3 in a function called initialize_board, with one parameter, n.

Place part 2 in a function print_board, with a 2d vector argument that represents the board.

Write the main function reads in n. Then it will call initialize_board and print_board.

Passing by Value and Scopes

What does the following print?

```
int add1(int x) {
    return x + 1;
}
int main() {
    int x = 10;
    cout << add1 (x) << endl;
    cout << x << endl;
}
```

Passing by Value and Scopes

What does the following print?

```
int add1(int x) {
    return x + 1;
}
int main() {
    int x = 10;
    cout << add1 (x) << endl;
    cout << x << endl;
}
```

Print:

10

11

Passing by Value and Scopes

What does the following print?

```
int add1(int x) {
    return x + 1;
}
int main() {
    int x = 10;
    cout << add1 (x) << endl; // we are passing the value 10 to add1
    cout << x << endl; // the value of x remains constant
}
```

Print:

10

11

Passing by Value and Scopes

- Add1 got the value 10 when main called it
- The parameter x is a variable name within the **scope** of the add1 function, we say x **binds** to the function add1

Add1:
Input: x = 10
x = x + 1 => 11
return x (return the value 11)

Main:
x = 10
add1(x) -> add1 (10)

Passing by Value and Scopes

- Add1 got the value 10 when main called it
- The parameter x is a variable name within the **scope** of the add1 function, we say x **binds** to the function add1

- Calling add1(x) passes the value of x to add1, same as add1(10)
- The variable x is a name that binds to the main function

Add1:
Input: x = 10
x = x + 1 => 11
return x (return the value 11)

Main:
x = 10
add1(x) -> add1 (10)

Passing by Value and Scopes

- Add1 got the value 10 when main called it
- The parameter x is a variable name within the **scope** of the add1 function, we say x **binds** to the function add1

Add1:
Input: x = 10
x = x + 1 => 11
return x (return the value 11)

- Calling add1(x) passes the value of x to add1, same as add1(10)
- The variable x is a name that binds to the main function

Main:
x = 10
add1(x) -> add1 (10)

- We say that we passed the value 10 to add1, so this is a function call with arguments passed **by value**
- We will learn how to **pass by reference** later

Passing by Value and Scopes

- Add1 got the value 10 when main called it
- The parameter x is a variable name within the **scope** of the add1 function, we say x **binds** to the function add1

Add1:
Input: x = 10
x = x + 1 => 11
return x (return the value 11)

* The same applies for all values, including strings, chars, bools, arrays, and vectors

- Calling add1(x) passes the value of x to add1, same as add1(10)
- The variable x is a name that binds to the main function

Main:
x = 10
add1(x) -> add1 (10)

- We say that we passed the value 10 to add1, so this is a function call with arguments passed **by value**
- We will learn how to **pass by reference** later

Exercise

```
void foo(int x){
    cout << "x = " << x << endl;
    x = 6;
    cout << "x = " << x << endl;
}

int main() {
    int x = 5;
    cout << "x = " << x << endl;
    foo(x);
    cout << "x = " << x << endl;
    return 0;
}
```

What prints?

A.
x = 5
x = 5
x = 6
x = 5

B.
x = 5
x = 5
x = 6
x = 6

C.
x = 5
x = 5
x = 5
x = 5

D.
x = 5
x = 6
x = 6
x = 6

Function Overloading

- You are overloading a function if the new function share the same name as an excising function
- These two functions must have different function signatures
  - Function signature – number , order, and type of parameters
- The compiler is smart enough to figure out which function you are calling as long as you have different function signatures
- Otherwise it will not let you compile

Example

```
void foo() {
    cout << "You called foo()" << endl;
}
void foo (int n) {
    cout << "You called foo(int n)" << endl;
}
int main() {
    foo ();
    foo (1);
}
```

Output:
You called foo
You called foo(int n)

Exercise

- Write a function weird_math that consumes 1,2, or 3 integers.
  - If there is only one integer, it will return the integer
  - If there are two, it will return their products
  - If there are three, it will return the weird_math of the first and the third integers plus the second integer

Given the main function:

```
int main() {
    cout << weird_math (1, 2, 3) << endl;
    cout <<weird_math (1) << endl;
    cout <<weird_math (1, 2) << endl;
}
```

Produce:

```
5
1
2
```

Exercise

Which of the following are valid function to declare given that

int foo (int a) {…}

has already been defined.

1. int foo (int a) {…}
2. bool foo (int a) {…}
3. int foo (int b) {…}
4. int foo (int a, int b) {…}

A. Yes
B. No

Exercise

Which of the following are valid function to declare given that

int foo (int a) {…} and

int foo (int a, int b) {…}

has already been defined.

1. int foo (int a, bool b) {…}
2. int foo (int b, int a) {…}

A. Yes
B. No

Exercise

Which of the following are valid function to declare given that

int foo (int a) {…} and

int foo (int a, int b) {…}

int foo (int a, bool b) {…}

has already been defined.

1. int foo (bool a, int b) {…}

A. Yes
B. No

Default Arguments

- When calling, you can only use default values at the end of the function
- You cannot selectively use some default values while skipping over others
- Ex). For the function int foo (int a = 0, int b = 0, int c = 0);

  foo() // valid

  foo (1) // valid, a = 1, b = default, c = default

  foo (1,2) // valid, a = 1, b = 2, c = default

  foo(1,2,3) // valid, a = 1, b = 2, c = 3

  foo (1,,3) // invalid, you are trying to call with a = 1, b = default, c = 3

  foo(,,1) // invalid, you are trying to call with a = 0, b = default, c = 1

Exercise

Using function overloading, write functions so that the output is

1 3 6

```
int main() {
    cout << add(1)<<“ “;
    cout << add(1,2)<<“ “;
    cout << add(1,2,3);
}
```

Exercise

Simplify your program using default values

```
int add (int a) {
    return a;
}

int add (int a, int b) {
    return a + b;
}

int add (int a, int b, int c) {
    return a + b + c;
}
```

Exercise

What if we had this?

```
int add (int a) {
    return a;
}

int add (int a, int b) {
    return a + b;
}

int add (int a, int b = 0, int c = 0) {
    return a + b + c;
}

int main() {
    cout << add(1)<< endl;
    cout << add(1,2)<< endl;
    cout << add(1,2,3)<< endl;
}
```

Exercise

What if we had this?

```
int add (int a) {
    return a;
}

int add (int a, int b) {
    return a + b;
}

int add (int a, int b = 0, int c = 0) {
    return a + b + c;
}

int main() {
    cout << add(1)<< endl; // error, ambiguous call
    cout << add(1,2)<< endl;
    cout << add(1,2,3)<< endl;
}
```

Exercise

What will the following produce?

```
void bar (int n, int m = 2) {
    cout << "you called with m = " << m << endl;
}

int main() {
    bar (1, 3);
    bar (1);
}
```

Exercise

What will the following produce?

```
void bar (int n, int m = 2) {
    cout << "you called with m = " << m << endl;
}

void bar (int k) {
    cout << "bar " << endl;
}

int main() {
    bar (1, 3);
    bar (1);
}
```

Sorting Algorithm – Bubble Sort

Bubble Sort – Swap

C++ std::swap

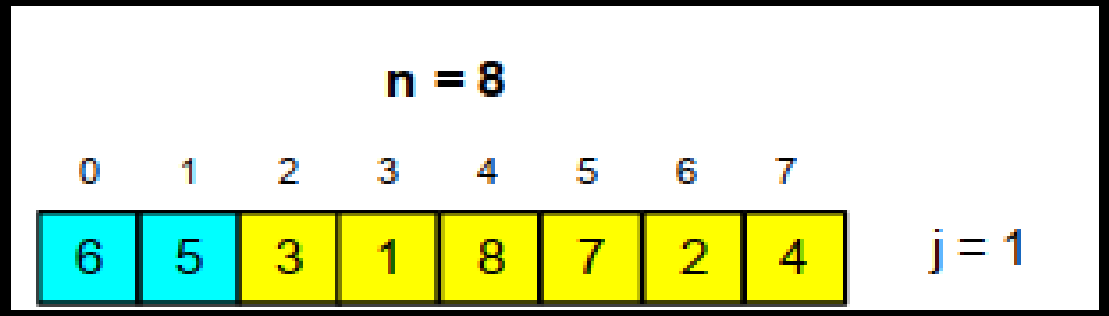
```
int main() {
    vector <int> Test = {1,2,3};
    swap(Test[1], Test[2]);

    int len = Test.size();
    for (int i = 0; i < len; ++i) { // note that I used len, not Test.size()
        cout << Test[i] << endl;
    }
}
```

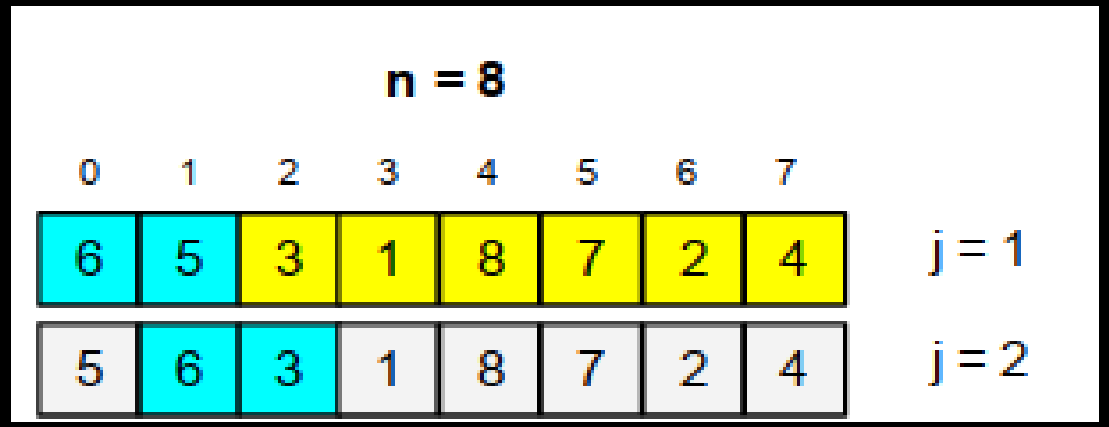
Bubble Sort

- Bubble Sort repeatedly swaps adjacent elements
- In each round, we “move the largest to the last”, the outer for loop always grabs the largest (the newly swapped largest or the existing largest)
- Each loop, we place 1 of n elements in the right place, so we will be sorting n – 1 elements in the next round
- Example: Given an array {6,5,3,1,8,7,2,4}

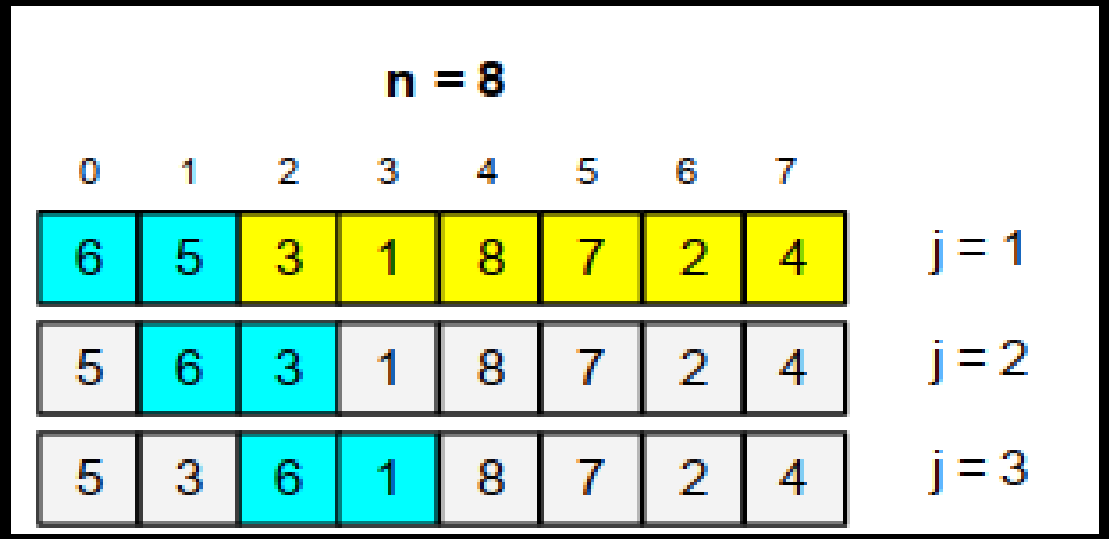
{6,5,3,1,8,7,2,4}



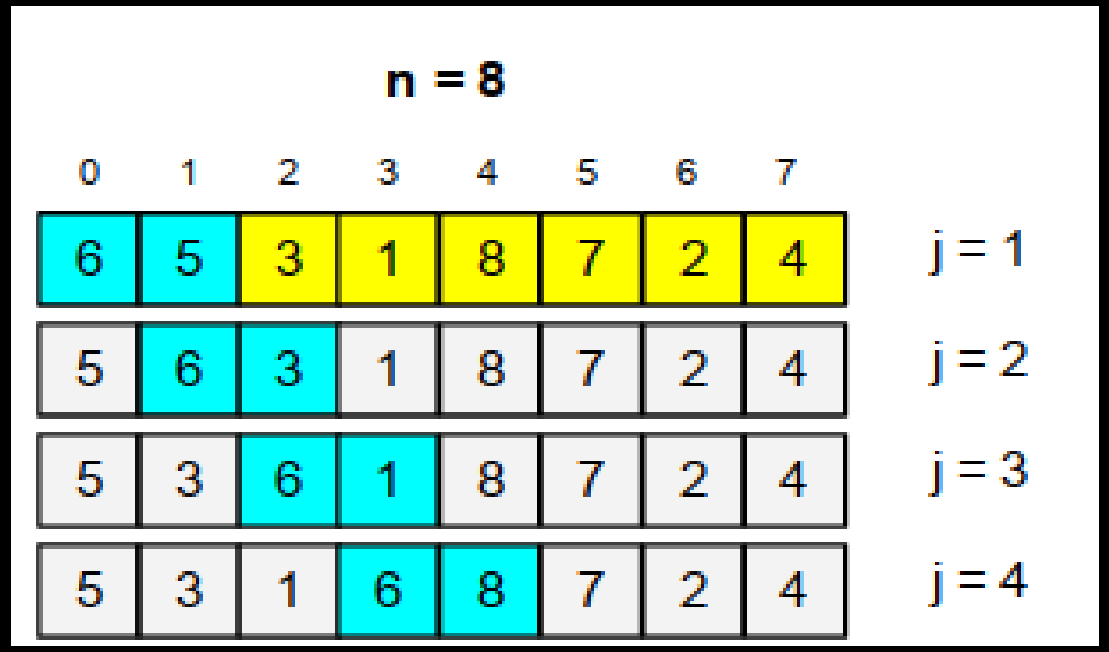
{6,5,3,1,8,7,2,4}



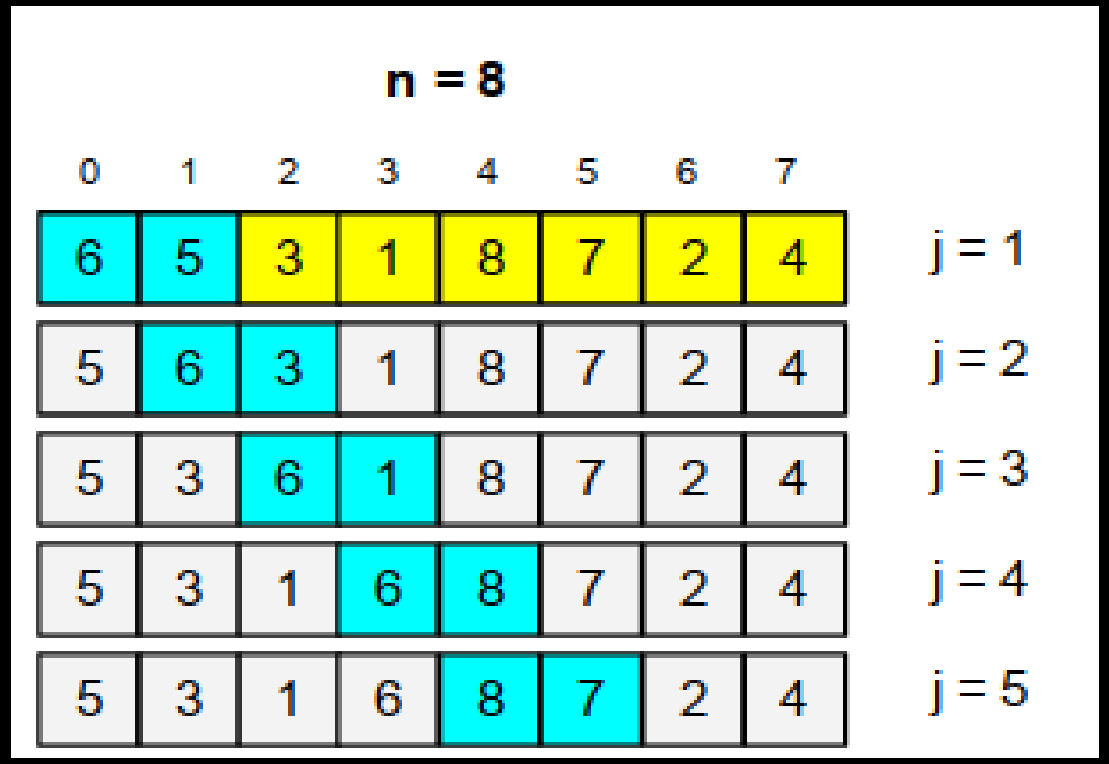
{6,5,3,1,8,7,2,4}



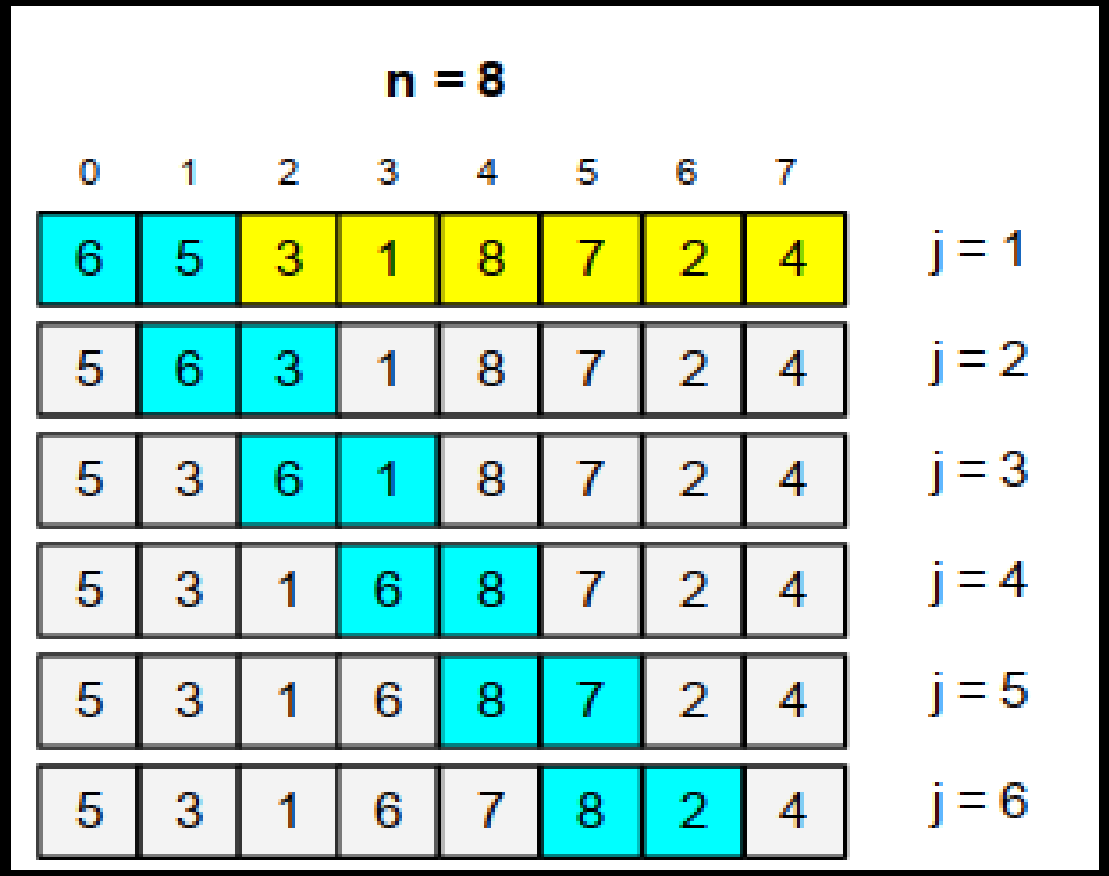
{6,5,3,1,8,7,2,4}


n = 8
0 1 2 3 4 5 6 7
6 5 3 1 8 7 2 4
j = 1
5 6 3 1 8 7 2 4
j = 2
5 3 6 1 8 7 2 4
j = 3
5 3 1 6 8 7 2 4
j = 4

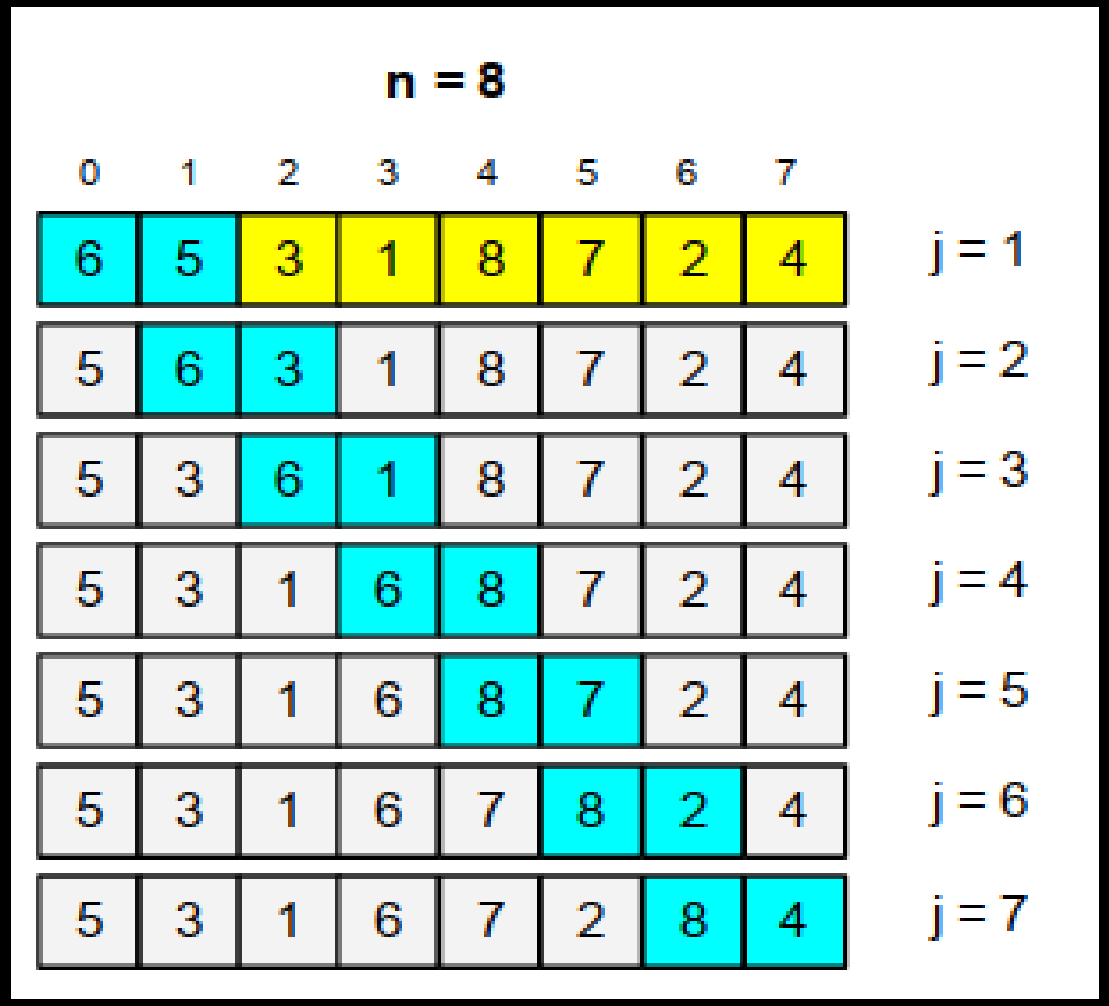
{6,5,3,1,8,7,2,4}


n = 8
0 1 2 3 4 5 6 7
6 5 3 1 8 7 2 4
j = 1
5 6 3 1 8 7 2 4
j = 2
5 3 6 1 8 7 2 4
j = 3
5 3 1 6 8 7 2 4
j = 4
5 3 1 6 8 7 2 4
j = 5

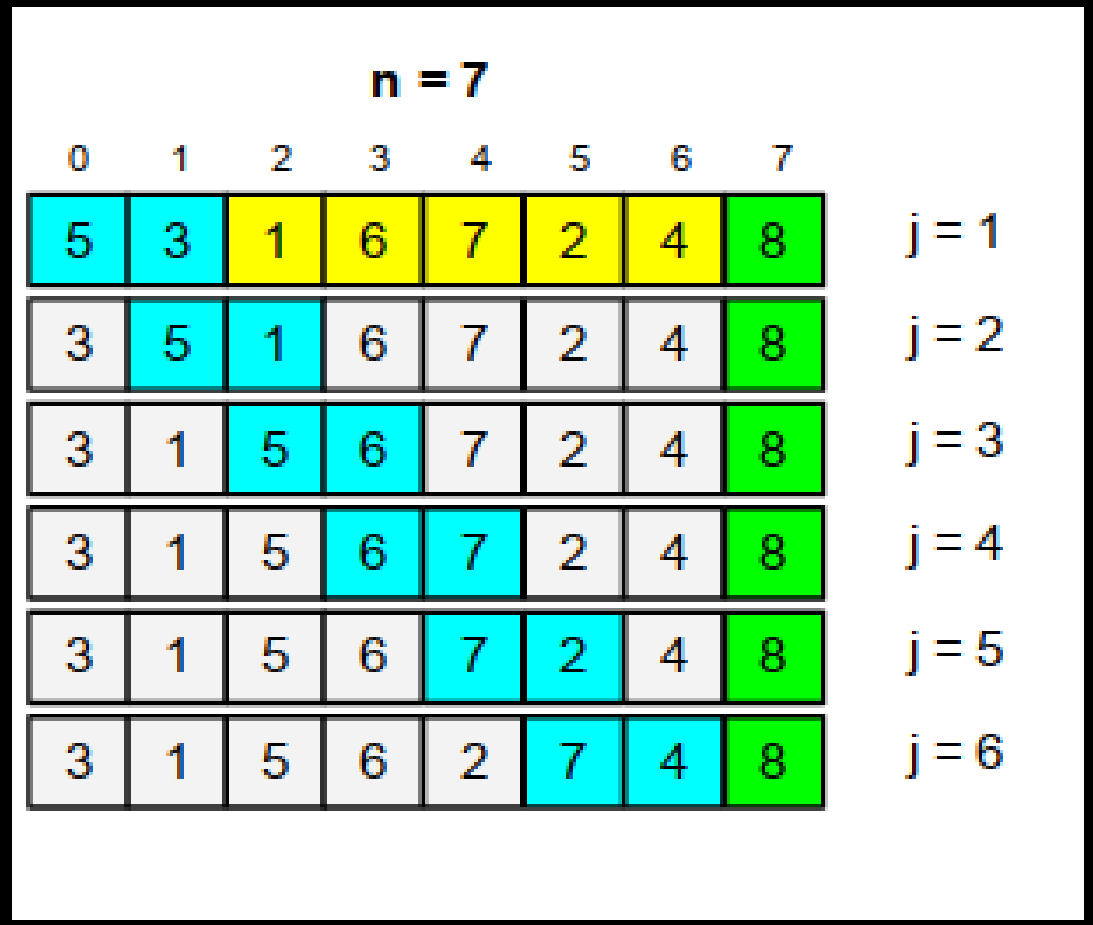
{6,5,3,1,8,7,2,4}


n = 8
0 1 2 3 4 5 6 7
6 5 3 1 8 7 2 4
j = 1
5 6 3 1 8 7 2 4
j = 2
5 3 6 1 8 7 2 4
j = 3
5 3 1 6 8 7 2 4
j = 4
5 3 1 6 8 7 2 4
j = 5
5 3 1 6 7 8 2 4
j = 6

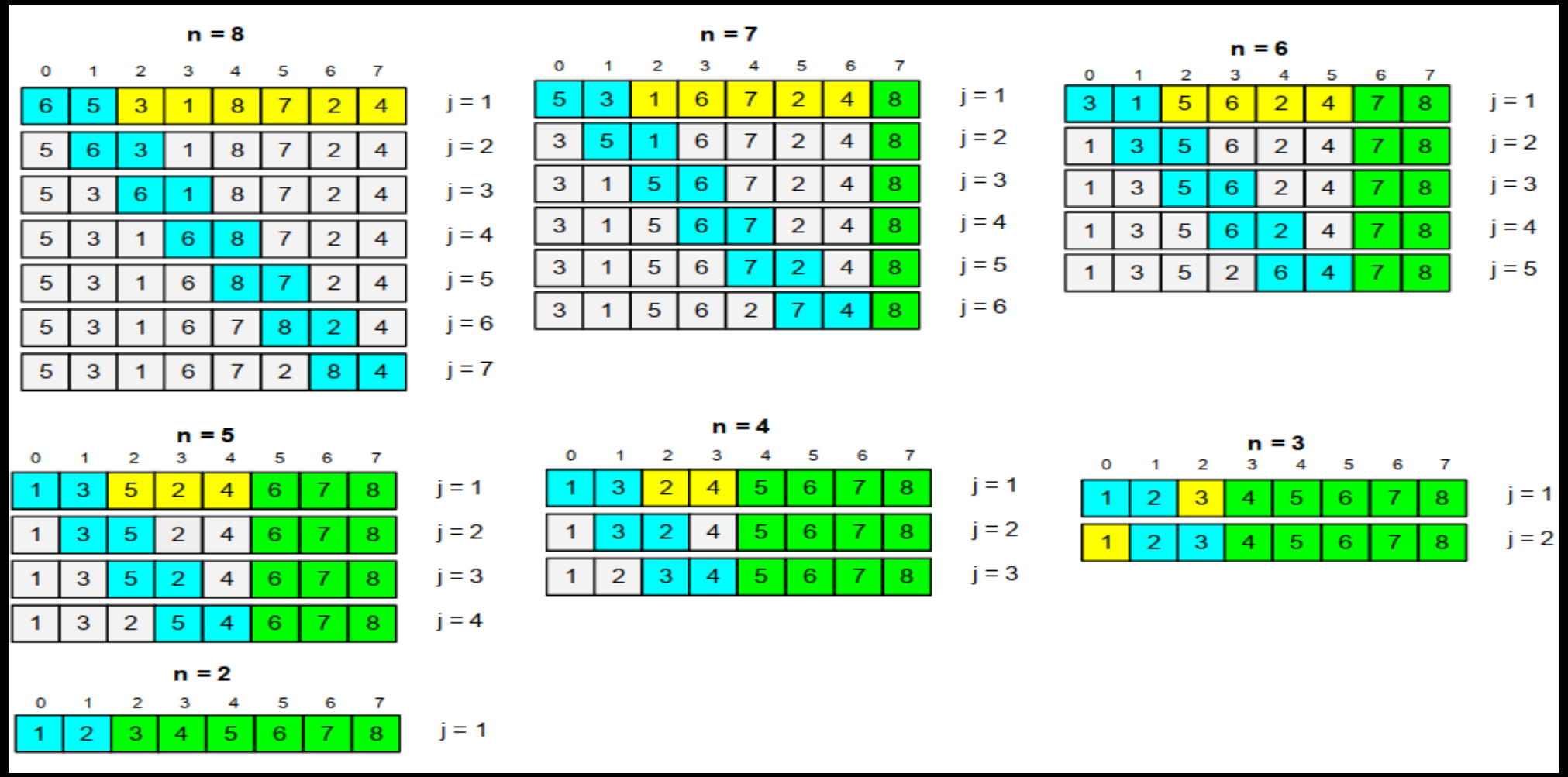
{6,5,3,1,8,7,2,4}


n = 8
0 1 2 3 4 5 6 7
6 5 3 1 8 7 2 4 j = 1
5 6 3 1 8 7 2 4 j = 2
5 3 6 1 8 7 2 4 j = 3
5 3 1 6 8 7 2 4 j = 4
5 3 1 6 8 7 2 4 j = 5
5 3 1 6 7 8 2 4 j = 6
5 3 1 6 7 2 8 4 j = 7

{5,3,1,6,7,2,4,8}


n = 7
0 1 2 3 4 5 6 7
5 3 1 6 7 2 4 8
j = 1
3 5 1 6 7 2 4 8
j = 2
3 1 5 6 7 2 4 8
j = 3
3 1 5 6 7 2 4 8
j = 4
3 1 5 6 7 2 4 8
j = 5
3 1 5 6 2 7 4 8
j = 6

Sorting {6,5,3,1,8,7,2,4}



Exercise

What prints?

```
vector <int> Test = {3, 4, 7, 2, 1};
int len = Test.size();
for (int i = 0; i < len - 1; ++i) { // we don't need to swap the last one
    for (int j = 0; j < len - 1 - i; ++j) { // last i are already sorted
        if (Test[j] > Test[j+1]) {
            cout << "swap: " << Test[j] << " with " << Test[j+1] << endl;
            swap (Test[j], Test[j+1]);
        }
    }
}
```

Exercise

What prints?

```
vector <int> Test = {3, 4, 7, 2, 1};
int len = Test.size();
for (int i = 0; i < len - 1; ++i) { // we don't need to swap the last one
    for (int j = 0; j < len - 1 - i; ++j) { // last i are already sorted
        if (Test[j] > Test[j+1]) {
            cout << "swap: " << Test[j] << " with " << Test[j+1] << endl;
            swap (Test[j], Test[j+1]);
        }
    }
}
```

swap: 7 with 2
swap: 7 with 1
swap: 4 with 2
swap: 4 with 1
swap: 3 with 2
swap: 3 with 1
swap: 2 with 1

Exercise

vector <int> numbers = {3, 4, 1};

```
    int len = numbers.size();
    for (int i = 0; i < len - 1; ++i) { // we don't need to swap the last one
        for (int j = 0; j < len - 1 - i; ++j) { // last i are sorted
            cout << "checking: " << numbers[j] << " and " << numbers[j+1] << endl;
            if (numbers[j] > numbers[j+1]) {
                swap (numbers[j], numbers[j+1]);
            }
        }
    }
```

checking: 3 and 4

checking: 4 and 1

checking: 3 and 1

Exercise

vector <int> numbers = {3, 4, 1};

```
    int len = numbers.size();
    for (int i = 0; i < len - 1; ++i) { // we don't need to swap the last one
        for (int j = 0; j < len - 1 - i; ++j) { // last i are sorted
            cout << "checking: " << numbers[j] << " and " << numbers[j+1] << endl;
            if (numbers[j] > numbers[j+1]) {
                swap (numbers[j], numbers[j+1]);
            }
        }
    }
```

Improving Efficiency (hw)

But this is inefficient, consider the extreme case:

```
vector<int> numbers = {9, 1, 2, 3, 4, 5, 6, 7, 8};
int len = numbers.size();
for (int i = 0; i < len - 1; ++i) {
    for (int j = 0; j < len - 1 - i; ++j) {
     // this would check every ordered pair!
        if (numbers[j] > numbers[j+1]) {
            swap (numbers[j], numbers[j+1]);
        }
    }
}
```

Improving Efficiency (hw)

But this is inefficient, consider the extreme case:

```
vector<int> numbers = {9, 1, 2, 3, 4, 5, 6, 7, 8};
int len = numbers.size();
for (int i = 0; i < len - 1; ++i) {
    for (int j = 0; j < len - 1 - i; ++j) {
     // this would check every ordered pair!
        if (numbers[j] > numbers[j+1]) {
            swap (numbers[j], numbers[j+1]);
        }
    }
}
```

Hint: We want something like this:

checking: 9 and 1
checking: 9 and 2
checking: 9 and 3
checking: 9 and 4
checking: 9 and 5
checking: 9 and 6
checking: 9 and 7
checking: 9 and 8
checking: 1 and 2 // where it checks that everything is in order
checking: 2 and 3
checking: 3 and 4
checking: 4 and 5
checking: 5 and 6
checking: 6 and 7
checking: 7 and 8