# THE "BIG-OH" NOTATION

**Example:** Prove that 8n−2 is O(n).

**Justification:**

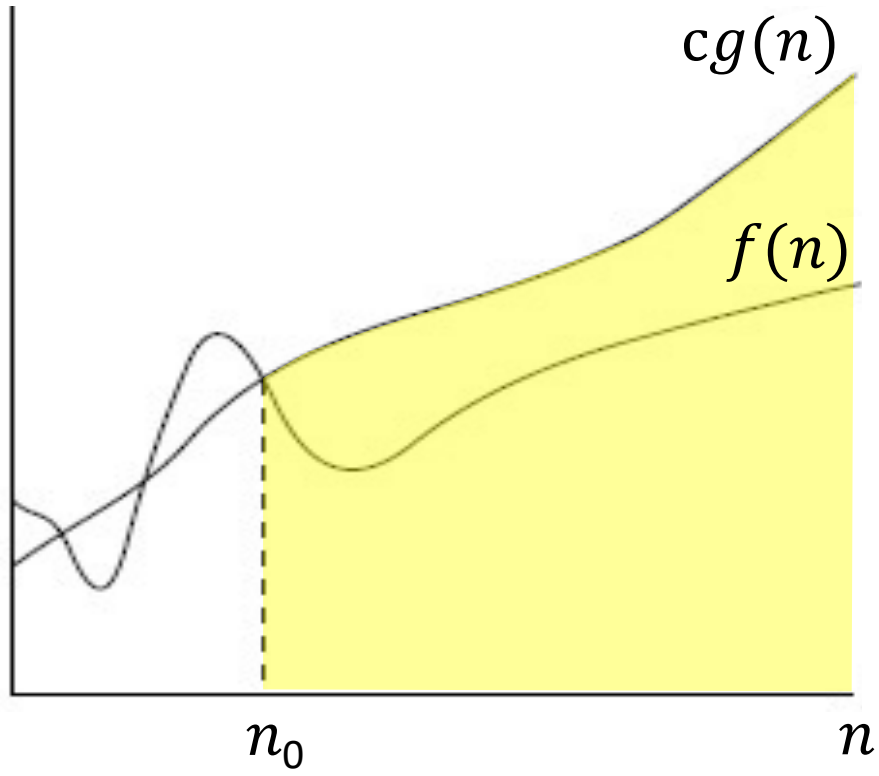
- $f$(n) = 8n-2, g(n) = n, and we need to find c > 0 and $n_0 \geq 1$ such that:

  $f(n) \leq cg(n)$, for $n \geq n_0$

  → 8n−2 ≤ cn, for $n \geq n_0$

- A possible choice is c = 8 and $n_0 = 1$, because:

  8n − 2 ≤ 8n, for n ≥ 1

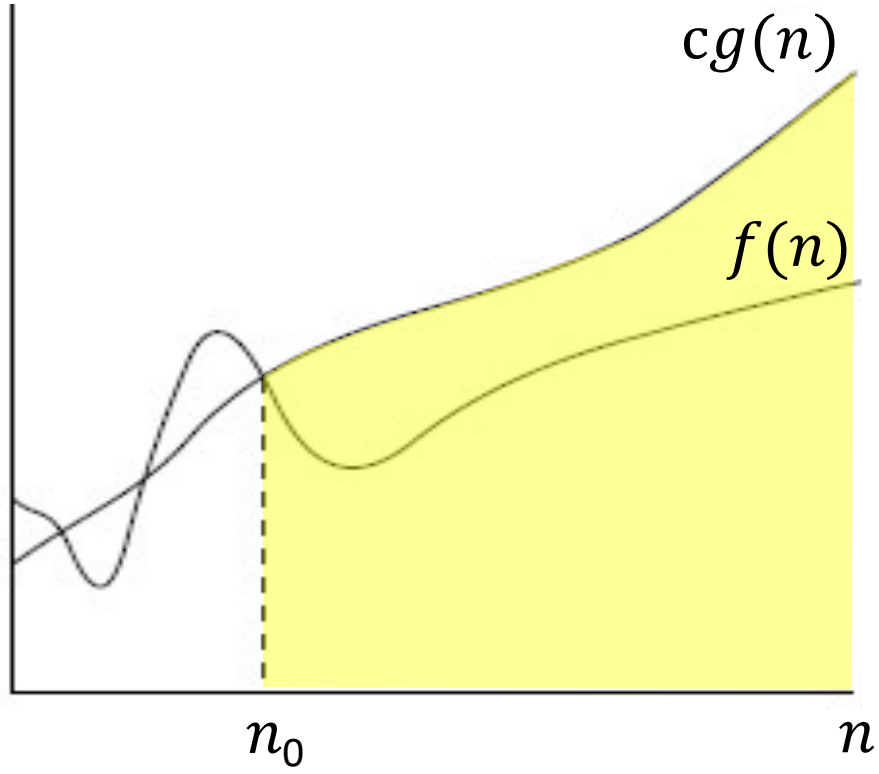


# THE "BIG-OH" NOTATION

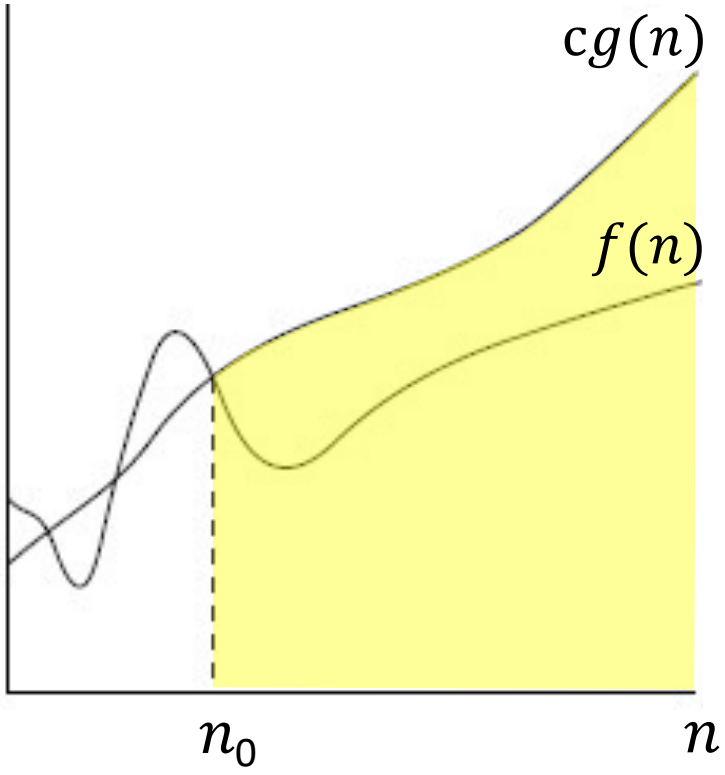
**Rule:** If $f(n)$ is a polynomial of degree $d$, *i.e.*,

$$f(n) = a_0 + a_1 n + \cdots + a_d n^d$$

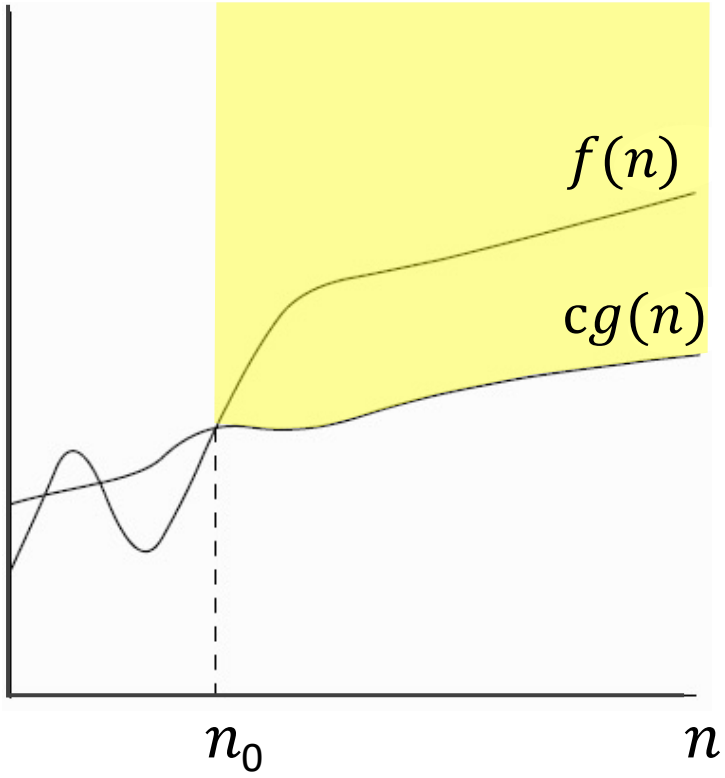
and $a_d > 0$, then: $f(n)$ is $O(n^d)$



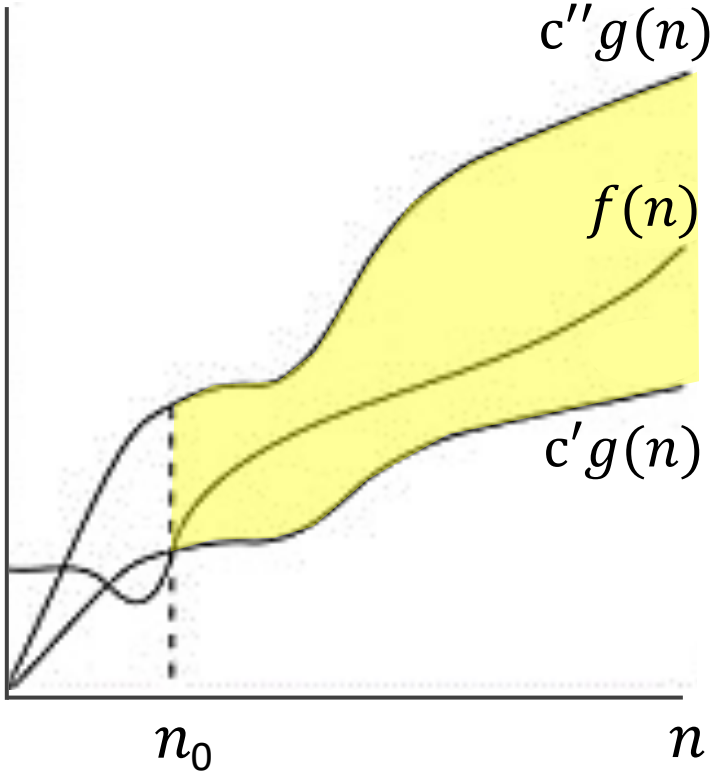
# “BIG-OH” VS. “BIG-OMEGA” VS. “BIG-THETA”




$f(n)$ is $O(g(n))$




$f(n)$ is $\Omega(g(n))$




$f(n)$ is $\Theta(g(n))$

# WHY DO WE USE THE "BIG-OH"

- The big-Oh notation is used widely to characterize **running time** and **space bound** in terms of some parameter, which may vary from an algorithm to another, but is always defined as a chosen measure of the "size" of the algorithm.

- It allows us to **ignore** constant coefficients and lower order terms and focus on the main components of a function that *affect its growth*, i.e. the "**order**" of the function.

# THE "BIG-OH" - CONVENTION

- The **seven functions that we've seen** are the most commonly used, e.g.:
  - An algorithm that runs in worst-case time $4n^2 + n \log n$ is said to be a quadratic-time algorithm, since it runs in $O(n^2)$ time.
  - An algorithm running in time at most $5n+20 \log n+4$ would be called a linear-time algorithm, since it runs in $O(n)$ time.

# THE "BIG-OH" - EXAMPLES

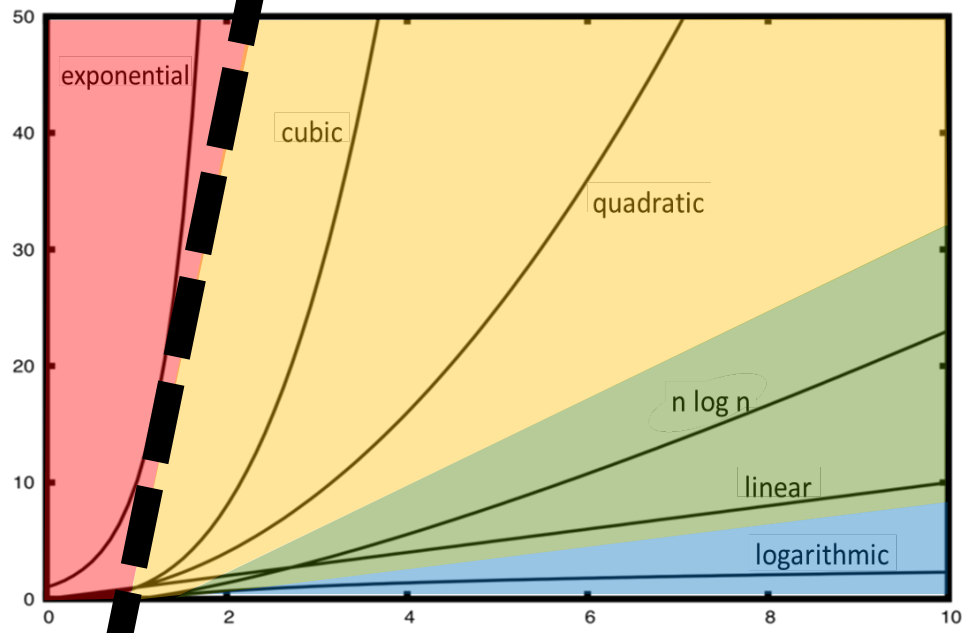
What is the growth rate in big-oh notation for each of these functions?

- $5n^2 + 3n \log n + 2n + 5$ $O(n^2)$
- $20n^3 + 10n \log n + 5$ $O(n^3)$
- $3 \log n + 2$ $O(\log n)$
- $2^{n+2}$ $O(2^n)$
- $2n + 100 \log n$ $O(n)$

# WHAT IS AN "EFFICIENT" ALGORITHM?

**What is an efficient algorithm?**

- If we must draw a line between **efficient** and **inefficient** algorithms, then this line is often considered to fall between algorithms running in **polynomial** time and those running in **exponential** time.




# ANALYSIS EXAMPLE 1 – PREFIX AVERAGE

- Given an array, X, of n elements, design an algorithm that returns an array, A, such that:

  A[i] equals the **average** of the elements X[0], X[1], …, X[i]

- Here is a potential algorithm. What is the complexity?

```
Algorithm prefixAverages1(X):
  for i ← 0 to n − 1 do
    a ← 0                          ← No. of executions: n   O(n)
    for j ← 0 to i do
      a ← a + X[j]                 ← No. of executions: (1+2+3+…+n) = n(n+1)/2   O(n²)
    A[i] ← a/(i + 1)               ← No. of executions: n   O(n)
  return array A
```

- Thus, runtime is a polynomial of degree 2, which is $O(n^2)$

# ANALYSIS EXAMPLE 1 – PREFIX AVERAGE

- Given an array, X, of n elements, design an algorithm that returns an array, A, such that:

  A[i] equals the **average** of the elements X[0], X[1], …, X[i]

- However, one can develop an **alternative** algorithm by exploiting the similarity between A[i-1] and A[i]:

$$A[i-1] = (X[0] + X[1] + \ldots + X[i-1]) / i$$
$$A[i] = (X[0] + X[1] + \ldots + X[i-1] + X[i]) / (i+1)$$

- Thus, if we use a temporary variable, s, that is computed for every i as follows:

$$s = (X[0] + X[1] + \ldots + X[i-1])$$

we can compute A[i] as follows:

$$A[i] = s + X[i] / (i+1)$$

# ANALYSIS EXAMPLE 1 – PREFIX AVERAGE

- Given an array, X, of n elements, design an algorithm that returns an array, A, such that:

  A[i] equals the **average** of the elements X[0], X[1], …, X[i]

- Here is the new algorithm. What is the complexity?

```
Algorithm prefixAverages2(X):
    s ← 0                          ← No. of executions: 1   O(1)
    for i ← 0 to n − 1 do
        s ← s + X[i]               ← No. of executions: n   O(n)
        A[i] ← s/(i + 1)           ← No. of executions: n   O(n)
    return array A
```

- Thus, it runs in O(n) time, which is **much faster than the previous algorithm!**

# ANALYSIS EXAMPLE 2 – POWER FUNCTION

- Design an algorithm that computes power function: p(x,n) = $x^n$
- One potential algorithm uses recursion, based on this definition:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,n-1) & \text{otherwise} \end{cases}$$

- Here is the algorithm. What is the complexity?

```
int p( int x, int n) {
    if (n==0) return 1;
    else return x * p(x,n-1);
}
```

← No. of recursive calls: n O(n)

- But did you know there is another recursive definition of p(x,n)?

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } n > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

# ANALYSIS EXAMPLE 2 – POWER FUNCTION

- Let us explain this definition using an **example** where x = 2 and n = 40:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } n > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

This means if we design a recursive algorithm that uses this definition, it can compute $2^{40}$ using just 5 recursive calls instead of 40.

**Step 1:** $2^{40} = \left(2^{40/2}\right)^2 = (2^{20})^2$

**Step 2:** $2^{20} = \left(2^{20/2}\right)^2 = (2^{10})^2$

**Step 3:** $2^{10} = \left(2^{10/2}\right)^2 = (2^{5})^2$

**Step 4:** $2^{5} = 2\left(2^{4/2}\right)^2 = 2(2^{2})^2$ (because 5 is an odd number)

**Step 5:** $2^{2} = \left(2^{2/2}\right)^2 = (2^{1})^2$

**Step 6:** $2^{1} = 2\left(2^{0}\right)^2 = 2\times 1 = 2$ (because 1 is an odd number)

# ANALYSIS EXAMPLE 2 – POWER FUNCTION

- Here is the new algorithm:

**Algorithm** Power$(x, n)$:
  **if** $n = 0$ **then**
    **return** 1
  **if** $n$ is odd **then**
    $y \leftarrow$ Power$(x, (n-1)/2)$
    **return** $x \cdot y \cdot y$
  **else**
    $y \leftarrow$ Power$(x, n/2)$
    **return** $y \cdot y$

What is the runtime of this algorithm?

- Each recursive call of Power(x,n) divides the exponent, n, by two.
- Thus, there are O(log n) recursive calls
- This is much faster than the previous algorithm that made O(n) recursive calls!

# CHAPTER 5:
# STACKS, QUEUES, AND DEQUES

# ABSTRACT DATA TYPES

- An **abstract data type** (ADT) is a mathematical model of a **data structure** that specifies:
  - The **type of the data** stored (a.k.a. attributes)
  - The **operations** supported on the data (a.k.a. methods)
  - The types of the **parameters of those operations** (a.k.a. arguments)

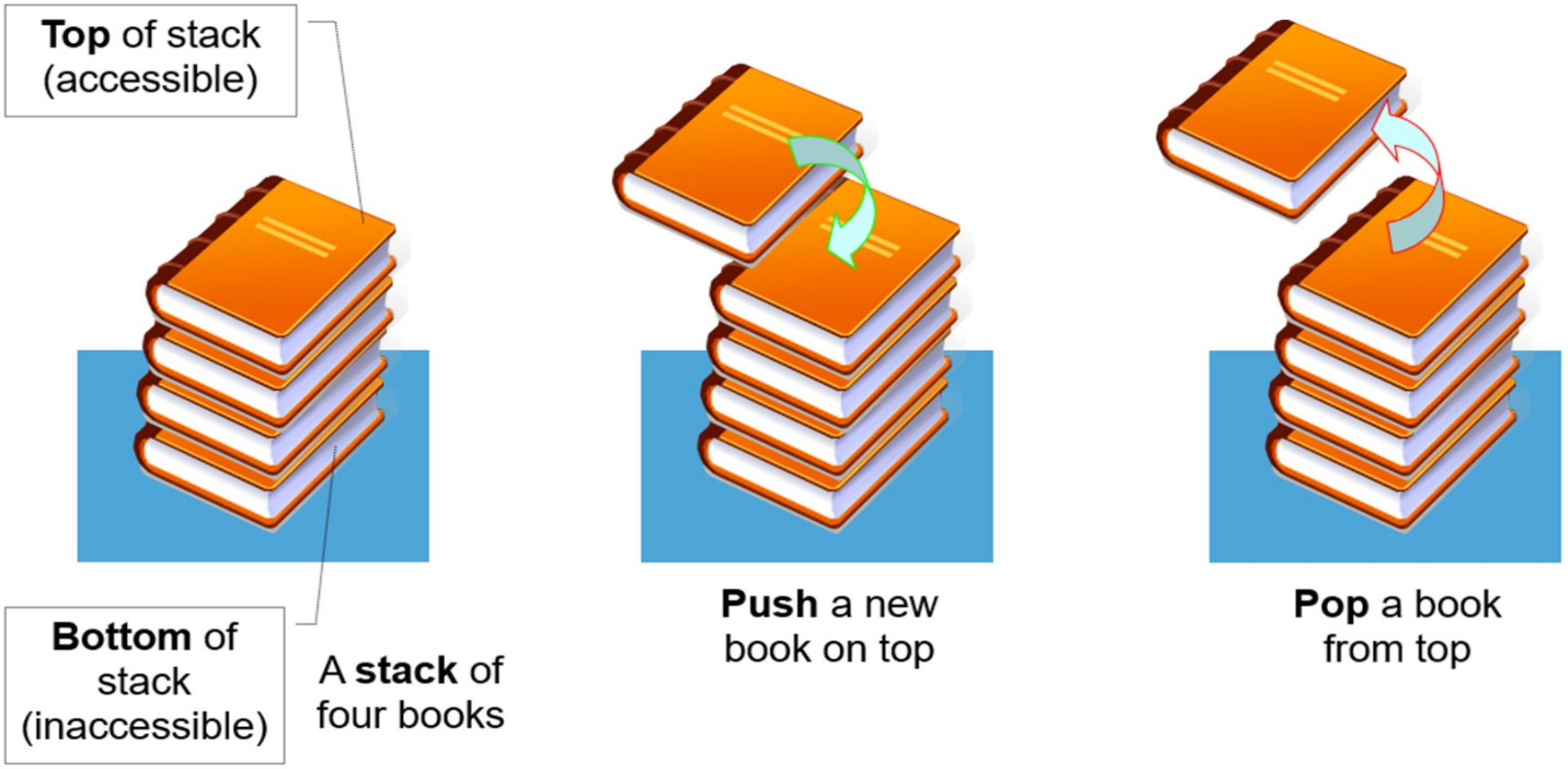
**As developers, we try to implement ADTs via programming languages in a more complete and detailed representation.**

- Here, we will discuss three **examples of ADTs**:
  - Stacks
  - Queues
  - Deques


# STACKS

# STACKS

- A **stack** is a data structure that stores objects, which can be added or removed according to the "***Last-in First-out (LIFO)***" principle.



# STACKS

- Formally, a stack is an abstract data type (ADT) that supports the following operations:
  - `push(e)`: **Insert element** `e` at the top of the stack.
  - `pop()`: **Remove the top element** from the stack (error if empty).
  - `top()`: Return a reference to the **value of the top element** on the stack without removing that element (error if empty).
  - `size()`: Return the **number of element**s in the stack.
  - `empty()`: Return **`true` if the stack is empty** and `false` otherwise

# EXAMPLE

- In this example, if we write a stack as follows:

  (x,y,z)

  It means that:
  - z is the **top** of the stack
  - x is at the **bottom**

Now, let's fill this table!

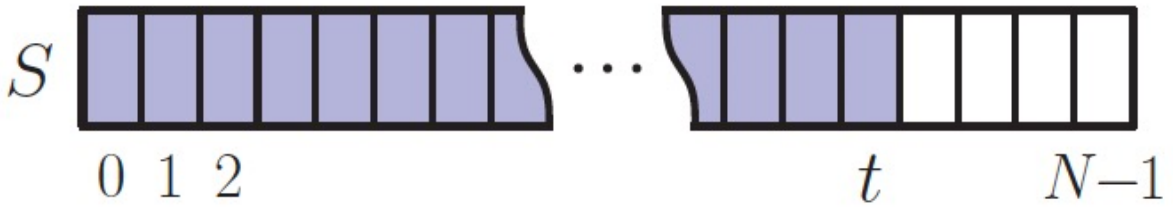
Under "**Output**" write:

- "error" if there is an error
- The value returned (if the function returns a value)
- "–" if there is **no error** and the function returns **no value**

| *Operation* | *Output* | *Stack Contents* |
|---|---|---|
| push(5) | – | (5) |
| push(3) | – | (5,3) |
| pop() | – | (5) |
| push(7) | – | (5,7) |
| pop() | – | (5) |
| top() | *5* | (5) |
| pop() | – | () |
| pop() | *"error"* | () |
| top() | *"error"* | () |
| empty() | *true* | () |
| push(9) | – | (9) |
| push(7) | – | (9,7) |
| push(3) | – | (9,7,3) |
| push(5) | – | (9,7,3,5) |
| size() | *4* | (9,7,3,5) |
| pop() | – | (9,7,3) |
| push(8) | – | (9,7,3,8) |
| pop() | – | (9,7,3) |
| top() | *3* | (9,7,3) |

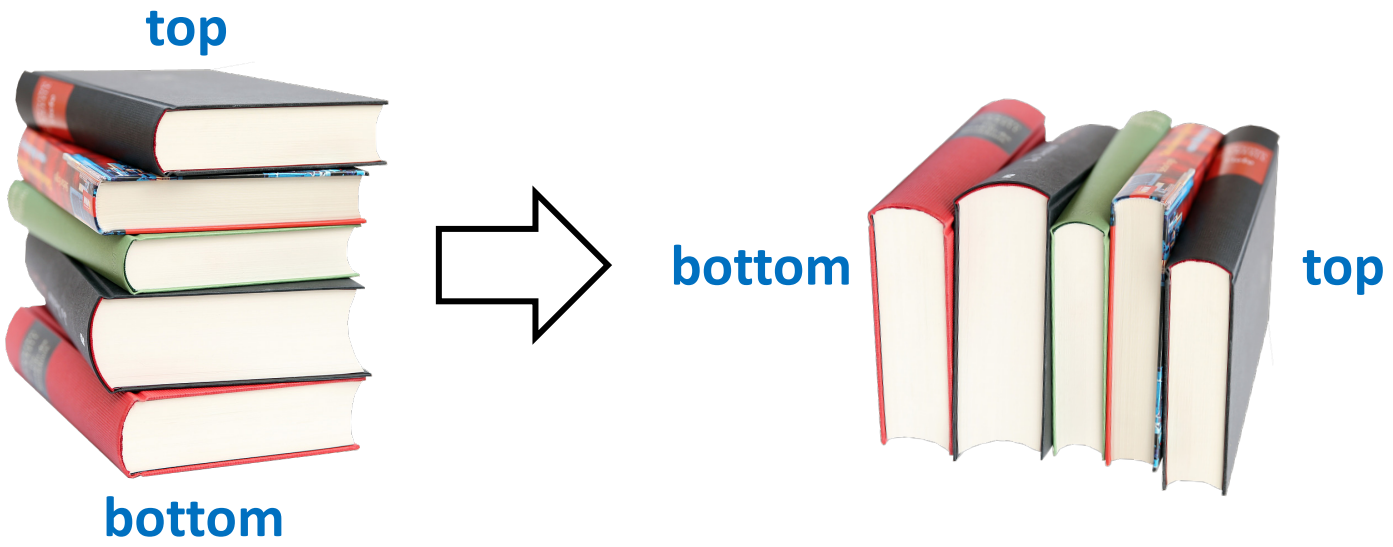
# IMPLEMENTATION VIA AN ARRAY

- We can implement a stack by storing its elements in an array $S$ of a fixed-size $N$, where the bottom element is $S[0]$ and the top one is $S[t]$




Note that $N$ specifies the maximum capacity

- It is as if we rotated the stack clockwise:

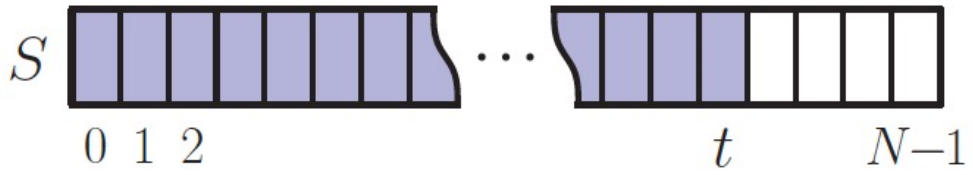


# THE STACK CLASS STRUCTURE

- Main Attributes:
  - The predefined capacity of the stack
  - An array to store the elements of the stack
  - The index of the top element in the stack

- Main Methods:
  - top(), push(e), pop(), size() , empty()

# CLASS DEFINITION



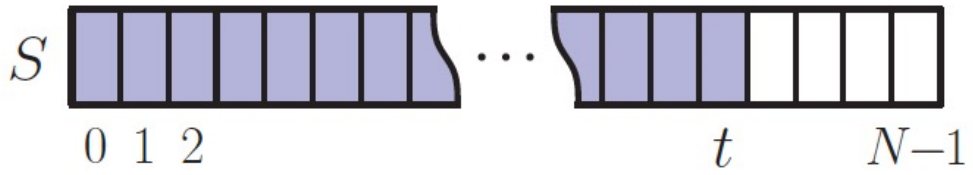

```
template <typename E>
class ArrayStack {
    enum { DEF_CAPACITY = 100 };
public:

    // public members go here…


private:
    E* S; // We will define S as an array that store the elements (see the figure above!)
    int capacity; // stack's maximum capacity (i.e., this is "N" in the figure above!)
    int t; // index of the top of the stack (see the figure above!)
};
```

This line is equivalent to:

```
const int DEF_CAPACITY = 100;
```

Which represents the default stack capacity

# CLASS DEFINITION




```
template <typename E>
class ArrayStack {
    enum { DEF_CAPACITY = 100 };
public:
    ArrayStack(int cap = DEF_CAPACITY);  /* constructor with a parameter "cap"
                                            whose default value is DEF_CAPACITY */
private:
    E* S;
    int capacity;
    int t;
};
```

In general, you can specify the default value of a parameter, e.g.,

```
void printMessage( string m = "Hello!" ){
    cout << m;
}
```

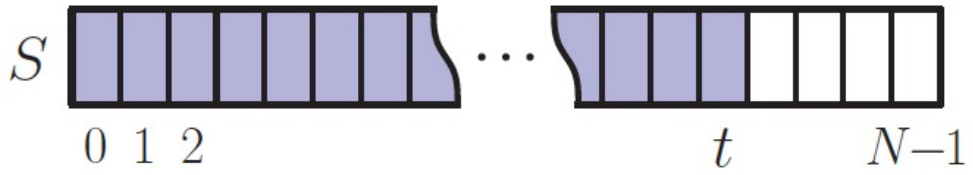
Now if you don't specify the value of "m", it will be set to "Hello!"

```
printMessage( );    // this line prints "Hello!"
```

But you can still specify any value you want for "m", e.g.,

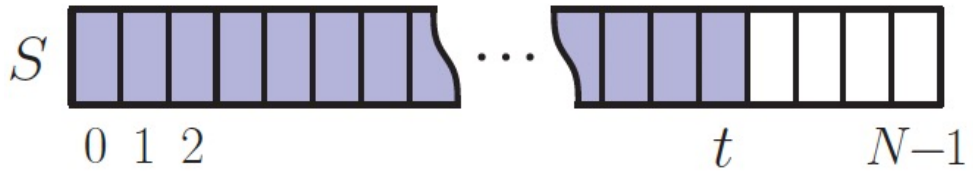
```
printMessage( "Goodbye" );   // this line prints "Goodbye!"
```

# CLASS DEFINITION




```
template <typename E>
class ArrayStack {
    enum { DEF_CAPACITY = 100 };
public:
    ArrayStack(int cap = DEF_CAPACITY);  /* constructor with a parameter "cap"
                                            whose default value is DEF_CAPACITY */
    int size() const;  /* returns the number of elements currently in the stack
                          (we will see its implementation in the coming slides) */
    bool empty() const; // returns "true" if the stack is empty
    const E& top() const throw(StackEmpty); // returns the top element
    void push(const E& e) throw(StackFull); // push element "e" onto stack
    void pop() throw(StackEmpty); // pop the stack (i.e., remove top element)
private:
    E* S; // We will define S as an array that store the elements (see the figure above!)
    int capacity; // stack's maximum capacity (i.e., this is "N" in the figure above!)
    int t; // index of the top of the stack (see the figure above!)
};
```

# METHOD DEFINITION

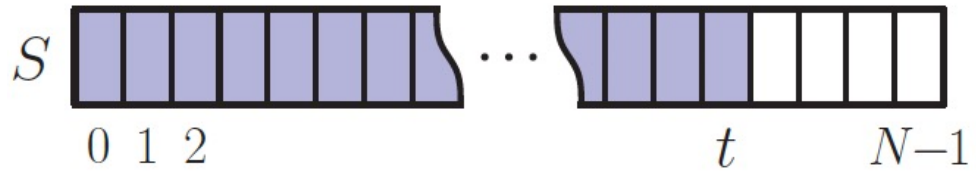



```
// constructor
template <typename E> ArrayStack<E>::ArrayStack(int cap)
: S(new E[cap]), capacity(cap), t(-1) { }
```

This is an initializer list! Remember, this is the same as writing:

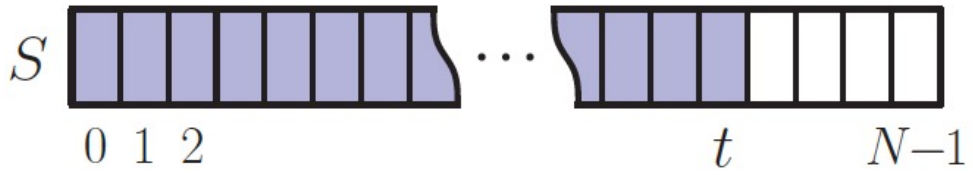
```
S = new E[ cap ];
capacity = cap;
t = -1;
```

# METHOD DEFINITION



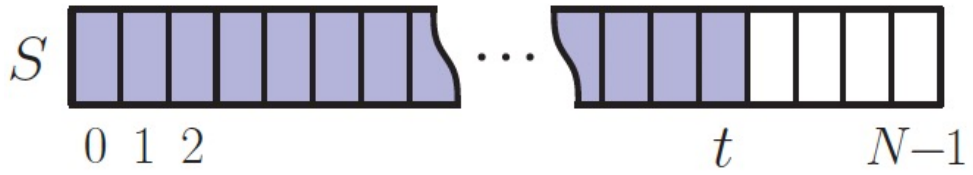

```
// constructor
template <typename E> ArrayStack<E>::ArrayStack(int cap)
: S(new E[cap]), capacity(cap), t(-1) { }

// returns the number of items in the stack
template <typename E> int ArrayStack<E>::size() const {
        return (t + 1);
}
// is the stack empty?
template <typename E> bool ArrayStack<E>::empty() const {
        return (t < 0);
}
// return top of stack
template <typename E> const E& ArrayStack<E>::top() const throw(StackEmpty) {
        if (empty()) throw StackEmpty("Top of empty stack");
        return S[t];
}
```

# METHOD DEFINITION



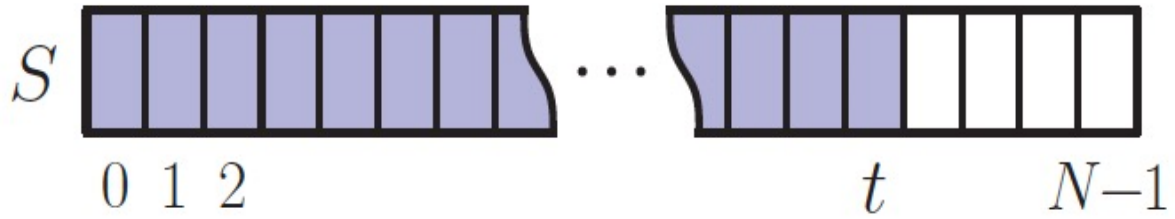

```
// push element onto the stack
template <typename E> void ArrayStack<E>::push(const E& e) throw(StackFull)
{
	if (size() == capacity) throw StackFull("Push to full stack");
	S[++t] = e;
}

// pop the stack
template <typename E> void ArrayStack<E>::pop() throw(StackEmpty)
{
	if (empty()) throw StackEmpty("Pop from empty stack");
	--t;
}
```

# EXAMPLE




```
ArrayStack<int> A;                          // A = [], size = 0
A.push(7);                                  // A = [7*], size = 1
A.push(13);                                 // A = [7, 13*], size = 2
cout << A.top() << endl; A.pop();           // A = [7*], outputs: 13
A.push(9);                                  // A = [7, 9*], size = 2
cout << A.top() << endl;                    // A = [7, 9*], outputs: 9
cout << A.top() << endl; A.pop();           // A = [7*], outputs: 9
ArrayStack<string> B(10);                   // B = [], size = 0
B.push("Bob");                              // B = [Bob*], size = 1
B.push("Alice");                            // B = [Bob, Alice*], size = 2
cout << B.top() << endl; B.pop();           // B = [Bob*], outputs: Alice
B.push("Eve");                              // B = [Bob, Eve*], size = 2
```

# COMPLEXITY




- What is the complexity of the different methods?

```
// returns the number of items in the stack
template <typename E> int ArrayStack<E>::size() const {
        return (t + 1);
}
// is the stack empty?
template <typename E> bool ArrayStack<E>::empty() const {
        return (t < 0);
}
// return top of stack
template <typename E> const E& ArrayStack<E>::top() const throw(StackEmpty) {
        if (empty()) throw StackEmpty("Top of empty stack");
        return S[t];
}
// push element onto the stack
template <typename E> void ArrayStack<E>::push(const E& e) throw(StackFull)
{
       if (size() == capacity) throw StackFull("Push to full stack");
       S[++t] = e;
}

// pop the stack
template <typename E> void ArrayStack<E>::pop() throw(StackEmpty)
{
       if (empty()) throw StackEmpty("Pop from empty stack");
       --t;
}
```

| *Operation* | *Time* |
|---|---|
| size | $O(1)$ |
| empty | $O(1)$ |
| top | $O(1)$ |
| push | $O(1)$ |
| pop | $O(1)$ |