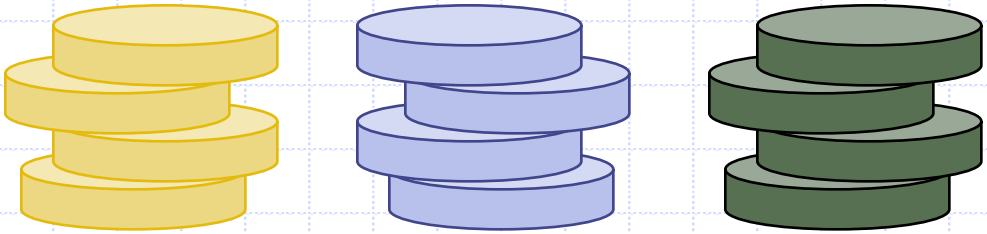
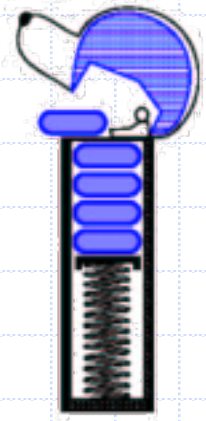
# Stacks

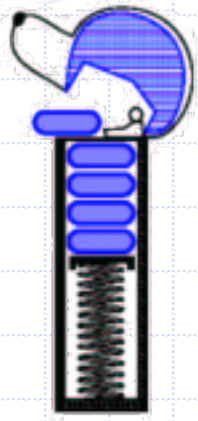


# The Stack

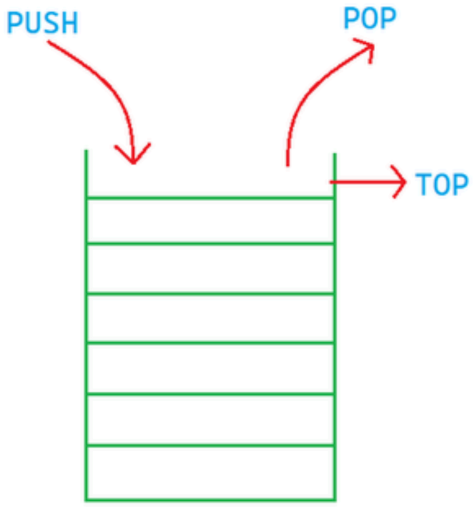


- The Stack stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).
- Example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time

# The Stack



- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element




- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer len(): returns the number of elements stored
  - boolean is_empty(): indicates whether no elements are stored

# Example of Stack operation

| Operation | Return Value | Stack Contents |
|---|---|---|
| S.push(5) | – | [5] |
| S.push(3) | – | [5, 3] |
| len(S) | 2 | [5, 3] |
| S.pop( ) | 3 | [5] |
| S.is_empty( ) | False | [5] |
| S.pop( ) | 5 | [ ] |
| S.is_empty( ) | True | [ ] |
| S.pop( ) | "error" | [ ] |
| S.push(7) | – | [7] |
| S.push(9) | – | [7, 9] |
| S.top( ) | 9 | [7, 9] |
| S.push(4) | – | [7, 9, 4] |
| len(S) | 3 | [7, 9, 4] |
| S.pop( ) | 4 | [7, 9] |
| S.push(6) | – | [7, 9, 6] |
| S.push(8) | – | [7, 9, 6, 8] |
| S.pop( ) | 8 | [7, 9, 6] |

# Applications of Stacks

- Stack is used for evaluating expression with operands and operations.
- Matching tags in HTML and XML
- Undo function in any text editor.
- Infix to Postfix conversion.
- Stacks are used for backtracking and parenthesis matching.
- Stacks are used for conversion of one arithmetic notation to another arithmetic notation.
- Stacks are useful for function calls, storing the activation records and deleting them after returning from the function. It is very useful in processing the function calls.
- Stacks help in reversing any set of data or strings.

# Applications of Stacks real time

- CD/DVD stand.
- Stack of books in a book shop.
- Undo and Redo mechanism in text editors.
- The history of a web browser is stored in the form of a stack.
- Call logs, E-mails, and Google photos in any gallery are also stored in form of a stack.
- YouTube downloads and Notifications are also shown in LIFO format(the latest appears first ).

# Applications of Stacks- Advantages

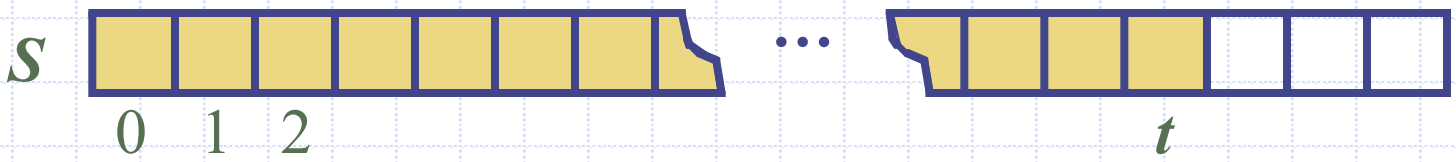
- Stack helps in managing data that follows the LIFO technique.
- Stacks are be used for systematic Memory Management.
- It is used in many virtual machines like JVM.
- When a function is called, the local variables and other function parameters are stored in the stack and automatically destroyed once returned from the function. Hence, efficient function management.
- Stacks are more secure and reliable as they do not get corrupted easily.
- Stack allows control over memory allocation and deallocation.
- Stack cleans up the objects automatically.

# Applications of Stacks-
## Disadvantages

- Stack memory is of limited size.
- The total of size of the stack must be defined before.
- If too many objects are created then it can lead to stack overflow.
- Random accessing is not possible in stack.
- If the stack falls outside the memory it can lead to abnormal termination.

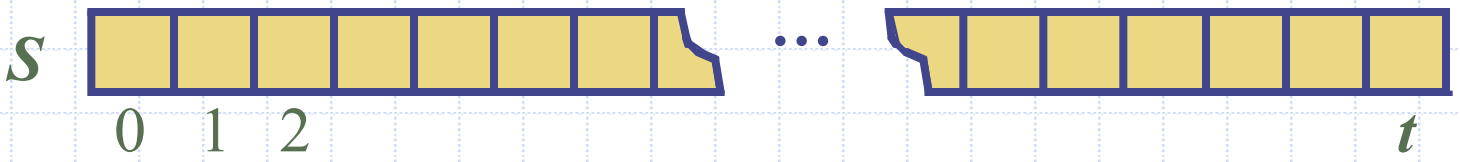
# Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element



# Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then need to grow the array and copy all the elements over.



# Performance and Limitations

- Performance
  - Let $n$ be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$ (amortized in the case of a push)

# Array-based Stack in Python

```
1  class ArrayStack:
2    """LIFO Stack implementation using a Python list as underlying storage."""
3
4    def __init__(self):
5      """Create an empty stack."""
6      self._data = [ ]                       # nonpublic list instance
7
8    def __len__(self):
9      """Return the number of elements in the stack."""
10     return len(self._data)
11
12   def is_empty(self):
13     """Return True if the stack is empty."""
14     return len(self._data) == 0
15
16   def push(self, e):
17     """Add element e to the top of the stack."""
18     self._data.append(e)                   # new item stored at end of list
19
```

```
20   def top(self):
21     """Return (but do not remove) the element at the top of the stack.
22
23     Raise Empty exception if the stack is empty.
24     """
25     if self.is_empty( ):
26       raise Empty('Stack is empty')
27     return self._data[-1]                  # the last item in the list
28
29   def pop(self):
30     """Remove and return the element from the top of the stack (i.e., LIFO).
31
32     Raise Empty exception if the stack is empty.
33     """
34     if self.is_empty( ):
35       raise Empty('Stack is empty')
36     return self._data.pop( )               # remove last item from list
```

# Linked list- Stack in Python

Do your self

# Linked list- Stack in Python

```
class Node:

    # Class to create nodes of linked list
    # constructor initializes node automatically
    def __init__(self,data):
        self.data = data
        self.next = None

class Stack:

    # head is default NULL
    def __init__(self):
        self.head = None

    # Checks if stack is empty
    def isempty(self):
        if self.head == None:
            return True
        else:
            return False
```

# Linked list- Stack in Python

```
def push(self,data):

    if self.head == None:
        self.head=Node(data)

    else:
        newnode = Node(data)
        newnode.next = self.head
        self.head = newnode

# Remove element that is the current head (start of the stack)
def pop(self):

    if self.isempty():
        return None

    else:
        # Removes the head node and makes
        #the preceding one the new head
        poppednode = self.head
        self.head = self.head.next
        poppednode.next = None
        return poppednode.data
```

# Linked list- Stack in Python

```
def peek(self):

    if self.isempty():
        return None

    else:
        return self.head.data

# Prints out the stack
def display(self):

    iternode = self.head
    if self.isempty():
        print("Stack Underflow")

    else:

        while(iternode != None):

            print(iternode.data,"->",end = " ")
            iternode = iternode.next
        return
```

# Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect: )(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

# Parentheses Matching Algorithm

**Algorithm** ParenMatch(*X*,*n*):
***Input:*** An array *X* of *n* tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number
***Output:*** **true** if and only if all the grouping symbols in *X* match
Let *S* be an empty stack
**for** *i*=0 to *n*-1 **do**
    **if** *X*[*i*] is an opening grouping symbol **then**
        *S*.push(*X*[*i*])
    **else if** *X*[*i*] is a closing grouping symbol **then**
        **if** *S*.is_empty() **then**
            **return false** {nothing to match with}
        **if** *S*.pop() does not match the type of *X*[*i*] **then**
            **return false** {wrong type}
**if** *S*.isEmpty() **then**
    **return true** {every symbol matched}
**else return false** {some symbols were never matched}

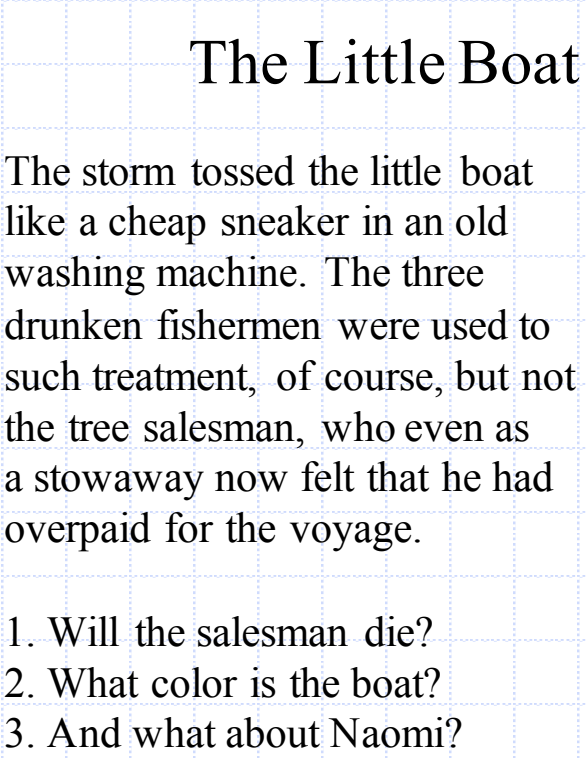
# Parentheses Matching in Python

```
1  def is_matched(expr):
2    """Return True if all delimiters are properly match; False otherwise."""
3    lefty = '({['                                  # opening delimiters
4    righty = ')}]'                                 # respective closing delims
5    S = ArrayStack( )
6    for c in expr:
7      if c in lefty:
8        S.push(c)                                  # push left delimiter on stack
9      elif c in righty:
10       if S.is_empty( ):
11         return False                             # nothing to match with
12       if righty.index(c) != lefty.index(S.pop( )):
13         return False                             # mismatched
14   return S.is_empty( )                           # were all symbols matched?
```

# HTML Tag Matching

- For fully-correct HTML, each <name> should pair with a matching </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```



# Tag Matching Algorithm in Python

```
1  def is_matched_html(raw):
2    """Return True if all HTML tags are properly match; False otherwise."""
3    S = ArrayStack( )
4    j = raw.find('<')                    # find first '<' character (if any)
5    while j != -1:
6      k = raw.find('>', j+1)             # find next '>' character
7      if k == -1:
8        return False                     # invalid tag
9      tag = raw[j+1:k]                   # strip away < >
10     if not tag.startswith('/'):        # this is opening tag
11       S.push(tag)
12     else:                              # this is closing tag
13       if S.is_empty( ):
14         return False                   # nothing to match with
15       if tag[1:] != S.pop( ):
16         return False                   # mismatched delimiter
17     j = raw.find('<', k+1)             # find next '<' character (if any)
18   return S.is_empty( )                 # were all opening tags matched?
```

# Evaluating Arithmetic Expressions

Slide by Matt Stallmann included with permission.

$14 - 3 * 2 + 7 = (14 - (3 * 2)) + 7$

Operator precedence

* has precedence over +/–

Associativity

operators of the same precedence group evaluated from left to right

Example: $(x - y) + z$ rather than $x - (y + z)$

Idea: push each operator on the stack, but first pop and perform higher and *equal* precedence operations.

# Algorithm for Evaluating Expressions

Slide by Matt Stallmann included with permission.

Two stacks:

- opStk holds operators
- valStk holds values
- Use $ as special "end of input" token with lowest precedence

Algorithm doOp()

```
x ← valStk.pop();
y ← valStk.pop();
op ← opStk.pop();
valStk.push( y op x )
```

Algorithm repeatOps( refOp ):

```
while ( valStk.size() > 1 ∧
        prec(refOp) ≤
        prec(opStk.top())
    doOp()
```

Algorithm EvalExp()

Input: a stream of tokens representing an arithmetic expression (with numbers)

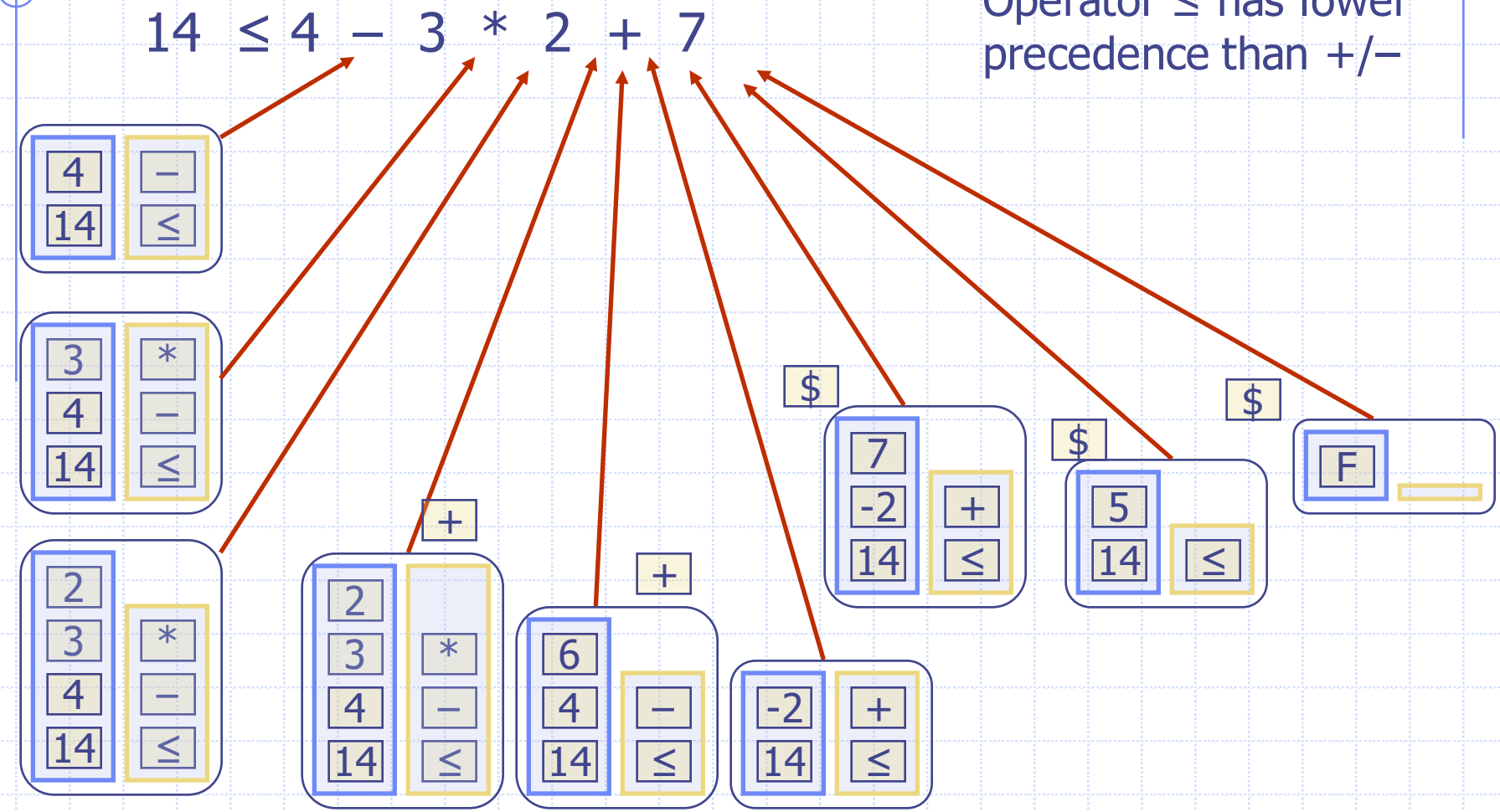
Output: the value of the expression

```
while there's another token z
    if isNumber(z) then
        valStk.push(z)
    else
        repeatOps(z);
        opStk.push(z)
repeatOps($);
return valStk.top()
```

# Algorithm on an Example Expression

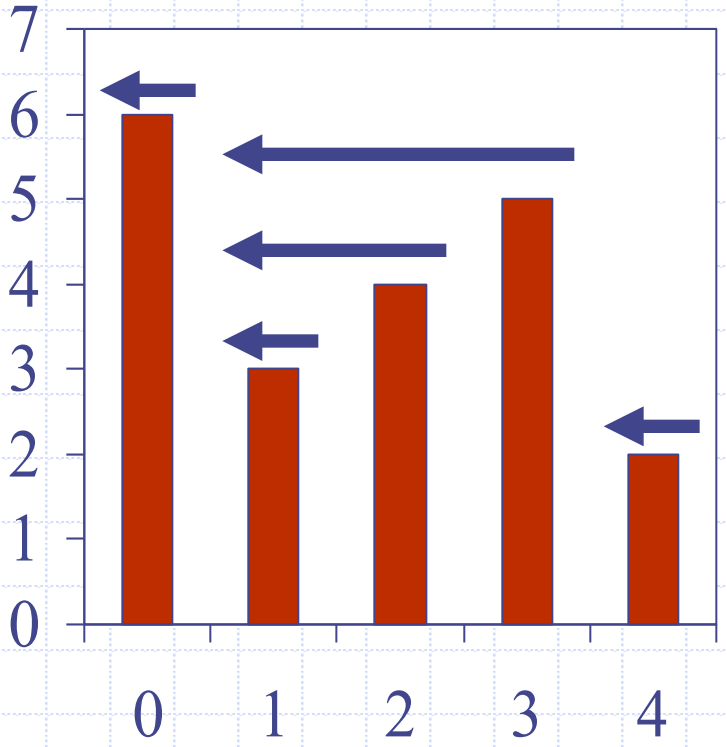
Slide by Matt Stallmann included with permission.

Operator ≤ has lower precedence than +/–



# Computing Spans (not in book)

- Using a stack as an auxiliary data structure in an algorithm
- Given an an array $X$, the span $S[i]$ of $X[i]$ is the maximum number of consecutive elements $X[j]$ immediately preceding $X[i]$ and such that $X[j] \leq X[i]$
- Spans have applications to financial analysis
  - E.g., stock at 52-week high




| $X$ | 6 | 3 | 4 | 5 | 2 |
|---|---|---|---|---|---|
| $S$ | 1 | 1 | 2 | 3 | 1 |

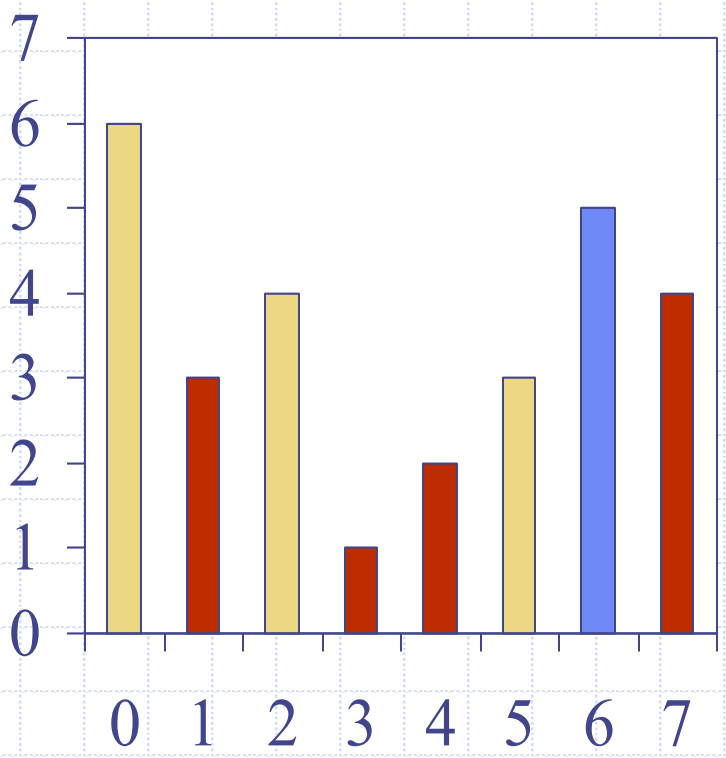
# Quadratic Algorithm

```
Algorithm spans1(X, n)                                  #
    Input array X of n integers
    Output array S of spans of X                        #
    S ← new array of n integers                         n
    for i ← 0 to n – 1 do                               n
        s ← 1                                           n
        while s ≤ i ∧ X[i – s] ≤ X[i]                   1 + 2 + …+ (n – 1)
            s ← s + 1                                   1 + 2 + …+ (n – 1)
        S[i] ← s                                        n
    return S                                            1
```

- Algorithm *spans1* runs in $O(n^2)$ time

# Computing Spans with a Stack

- We keep in a stack the indices of the elements visible when "looking back"
- We scan the array from left to right
  - Let $i$ be the current index
  - We pop indices from the stack until we find index $j$ such that $X[i] < X[j]$
  - We set $S[i] \leftarrow i - j$
  - We push $x$ onto the stack



# Linear Algorithm

- Each index of the array
  - Is pushed into the stack exactly one
  - Is popped from the stack at most once
- The statements in the while-loop are executed at most $n$ times
- Algorithm *spans2* runs in $O(n)$ time

```
Algorithm spans2(X, n)                          #
    S ← new array of n integers                 n
    A ← new empty stack                         1
    for i ← 0 to n – 1 do                       n
        while (¬A.is_empty() ∧
                X[A.top()] ≤ X[i] ) do          n
            A.pop()                             n
        if A.is_empty() then                    n
            S[i] ← i + 1                        n
        else
            S[i] ← i – A.top()                  n
        A.push(i)                               n
    return S                                    1
```