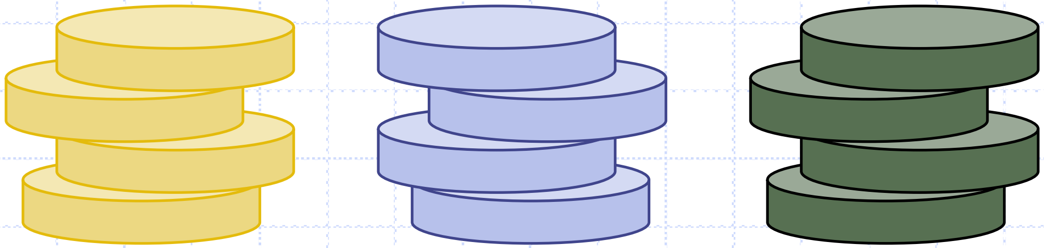
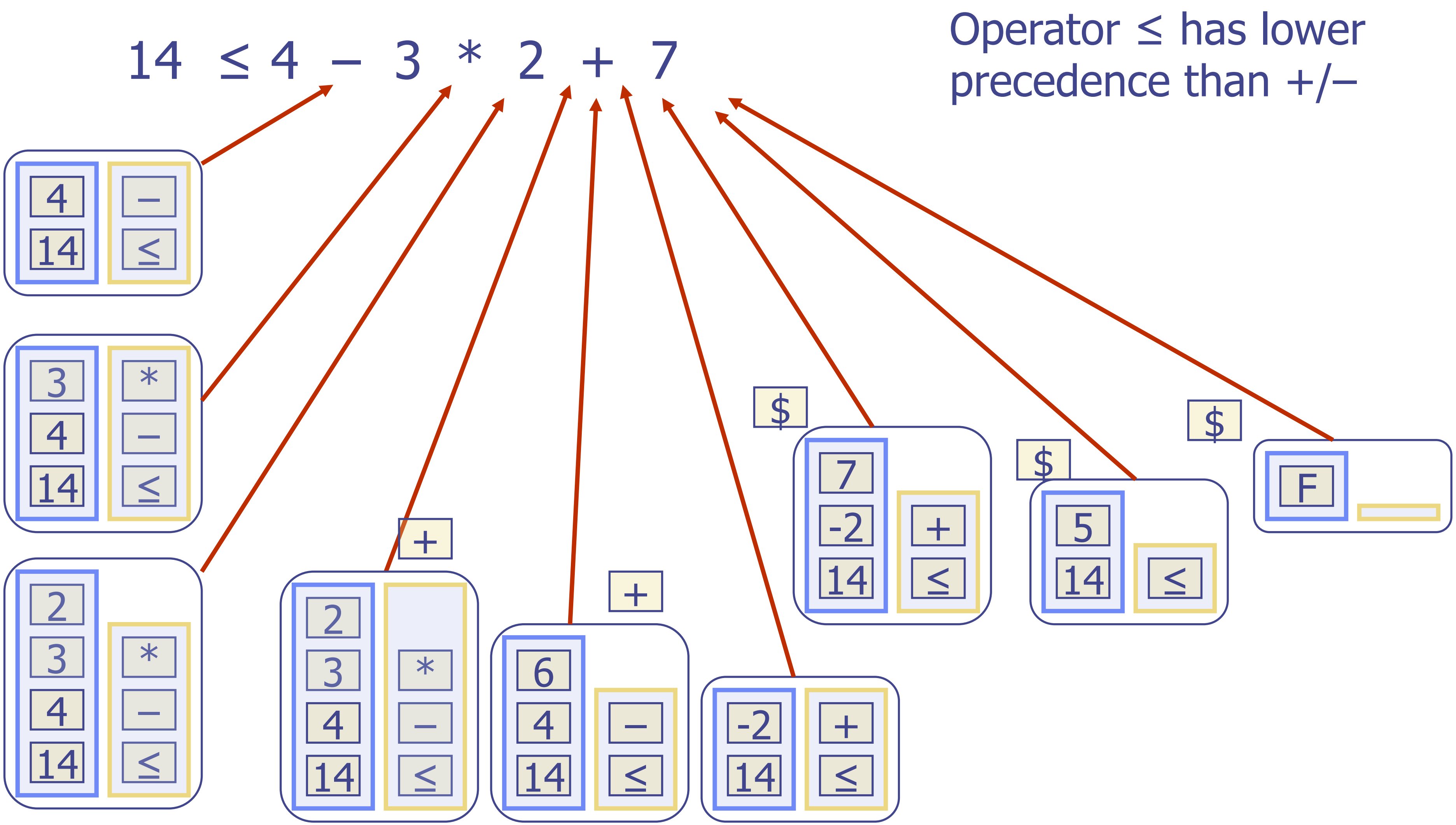
Stacks



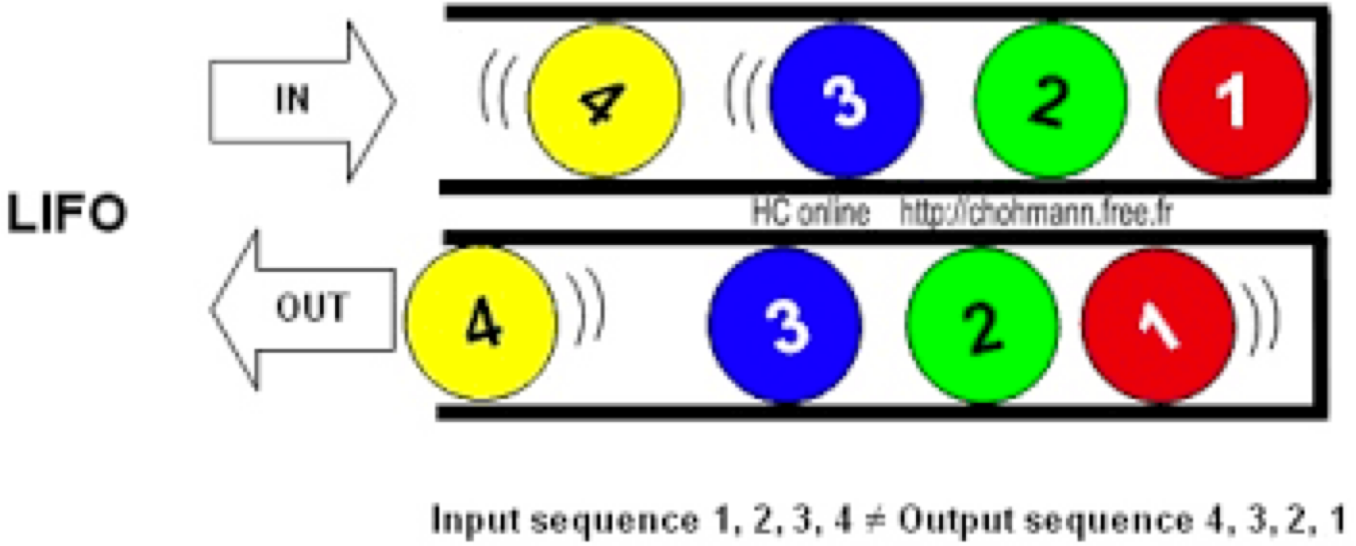
Example: Algorithm on an Example Expression



Overview and Reading

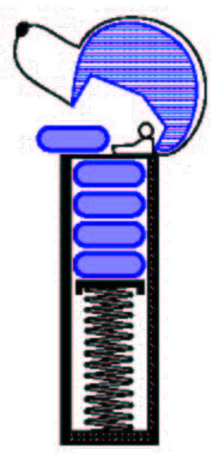
- Reading: Chapter 5.1

- Last-In-First-Out Data Structure



The Stack ADT

- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean empty(): indicates whether no elements are stored



Stack Interface in C++

- C++ interface corresponding to our Stack ADT
- Uses an exception class StackEmpty
- Different from the built-in C++ STL class stack
- STL: Standard Template Library

```
template <typename E>
class Stack  {
public:
   int size() const;
   bool empty() const;
   const E& top() const
         throw(StackEmpty);
   void push(const E& e);
   void pop() throw(StackEmpty);
}
```

Applications of Stacks

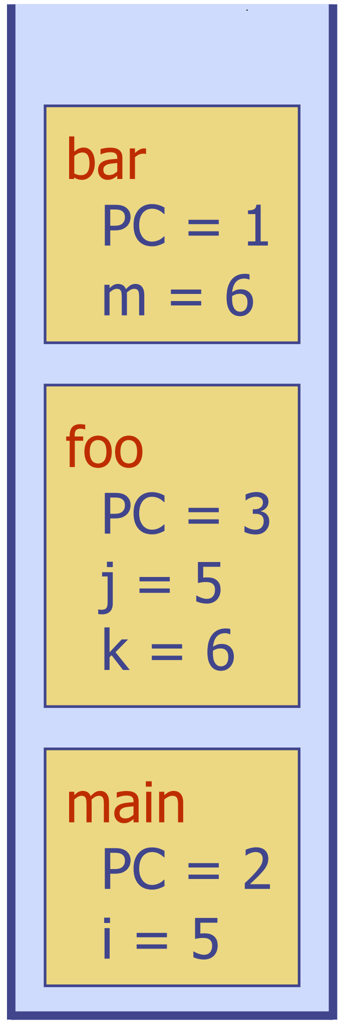
- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the C++ run-time system

- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Example: C++ Run-Time Stack

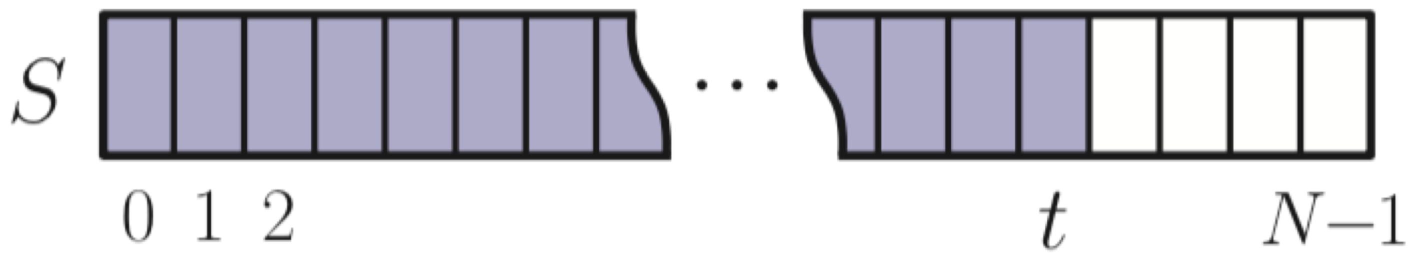
- The C++ run-time system keeps track of the chain of active functions with a stack
- When a function is called, the system pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When the function ends, its frame is popped from the stack and control is passed to the function on top of the stack
- Allows for recursion
- PC: Program Counter

```
main() {
   int i = 5;
   foo(i);
   }

foo(int j) {
   int k;
   k = j+1;
   bar(k);
   }

bar(int m) {
   …
   }
```



Example Implementation: 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



Example Implementation: Array-based Stack

- A simple way of implementing the Stack ADT
- Add elements from left to right
- A variable keeps track of the index of the top element
- The array storing the stack elements may become full
  - A push operation will then throw a StackFull exception
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

Algorithm size():
  return $t + 1$
Algorithm empty():
  return $(t < 0)$
Algorithm top():
  if empty() **then**
    throw StackEmpty exception
  return $S[t]$
Algorithm push(e):
  if size() $= N$ **then**
    throw StackFull exception
  $t \leftarrow t + 1$
  $S[t] \leftarrow e$
Algorithm pop():
  if empty() **then**
    throw StackEmpty exception
  $t \leftarrow t - 1$

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 $O(1)$

- Limitations
  - The maximum size of the stack must be defined a priori and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

- Linked-list based Stack in the text (Chapter 5.1.5)

Array-based Stack in C++

```
template <typename E>
class ArrayStack {
private:
   E* S; // array holding the stack
   int cap; // capacity
   int t; // index of top element
public:
   // constructor given capacity
   ArrayStack(int c) :
       S(new E[c]), cap(c), t(-1) { }
```

```
   void pop() {
     if (empty()) throw StackEmpty
              ("Pop from empty stack");
      t--;
   }
   void push(const E& e) {
      if (size() == cap) throw
           StackFull("Push to full stack");
      S[++ t] = e;
}
…  (other methods of Stack interface)
```

Example use in C++

* indicates top

```
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
```

Stack in C++ STL

```
#include <stack>
using std::stack;                    // make stack accessible
stack<int> myStack;                  // a stack of integers
```

size(): Return the number of elements in the stack.

empty(): Return true if the stack is empty and false otherwise.

push(e): Push e onto the top of the stack.

pop(): Pop the element at the top of the stack.

top(): Return a reference to the element at the top of the stack.

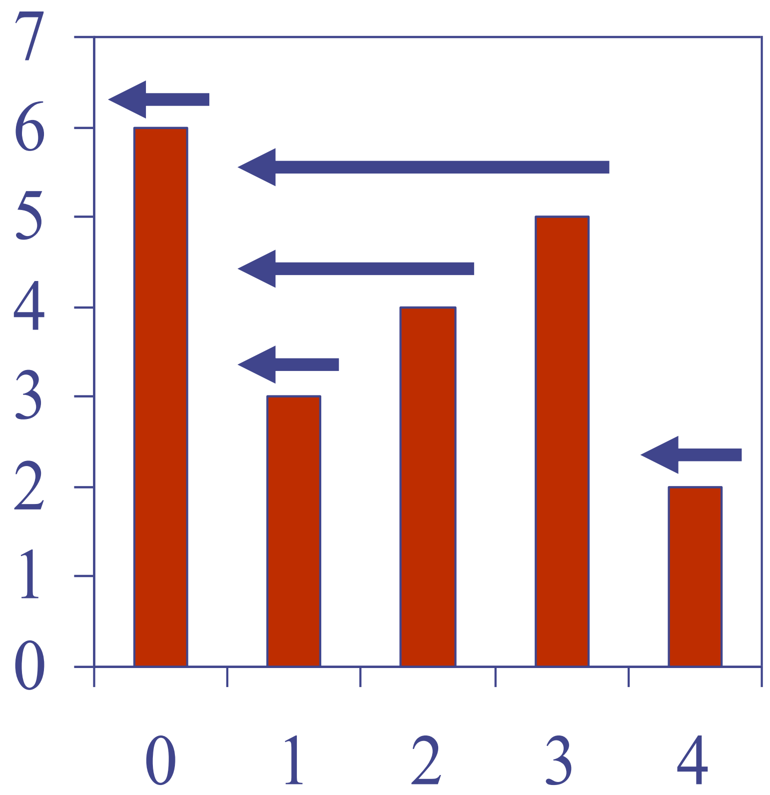
Example: Parentheses Matching

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

- Good Programmer
  - Someone who thinks that stack is a good data structure for the above task

Example: Computing Spans

- 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

Algorithm: span1

				i		
X	6	3	4	5	2
S	1	1	2	3	1

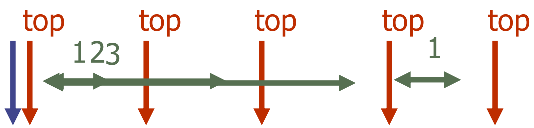
- Loop over *i* = 0, 1, 2, 3, 4
- For each *i*, compute S[i]. How?
  - From X[i] downward on, compute the number of elements which are consecutively smaller than X[i]

Quadratic Algorithm

Algorithm *spans1*(*X*, *n*)	
Input array X of n integers	
Output array S of spans of X	**#**
$S \leftarrow$ new array of n integers	n
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
$s \leftarrow 1$	n
while $s \leq i \wedge X[i - s] \leq X[i]$	$1 + 2 + \ldots + (n - 1)$
$s \leftarrow s + 1$	$1 + 2 + \ldots + (n - 1)$
$S[i] \leftarrow s$	n
return S	1

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

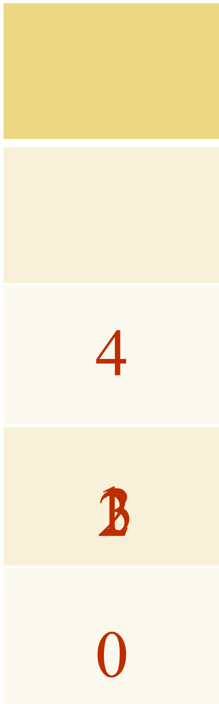
Algorithm: span2



X	6	3	4	5	2
S	1	1	2	3	1

From index 3 to 1,
I am sure that X[4] is the "consecutive largest".
So, please check X[0] after it

From index 2 to 1,
I am sure that X[2] is the "consecutive largest".
So, please check X[0] after it

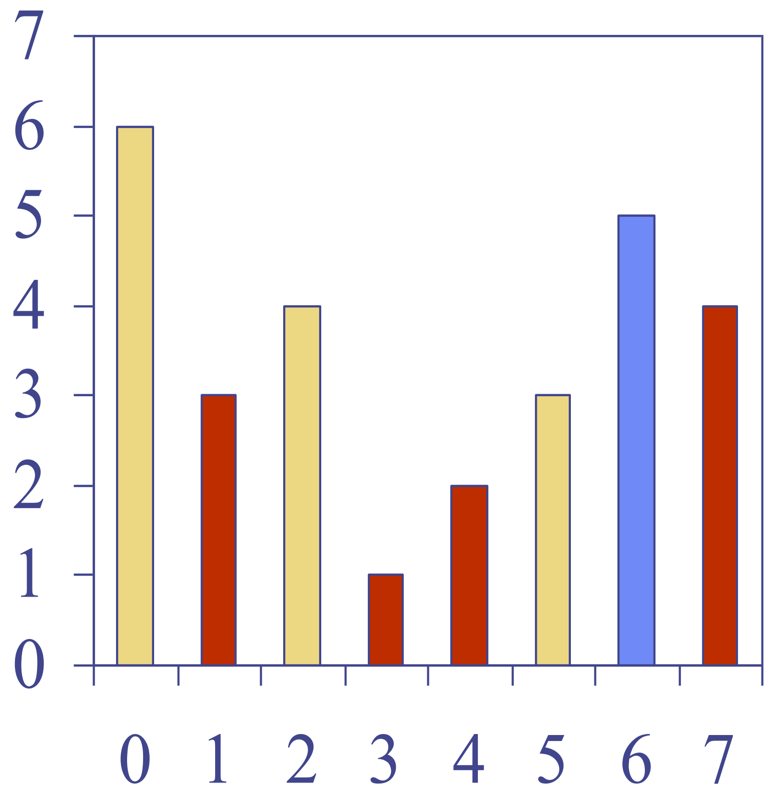


Algorithm ***spans2(X, n)***
    $S \leftarrow$ new array of n integers
    $A \leftarrow$ new empty stack
    for $i \leftarrow 0$ **to** $n - 1$ **do**
        while $(\neg A.empty() \wedge$
                $X[A.top()] \leq X[i]\ )$ **do**
            $A.pop()$
        if $A.empty()$ **then**
            $S[i] \leftarrow i + 1$
        else
            $S[i] \leftarrow i - A.top()$
        $A.push(i)$
    return S

Stack for "index"

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)*	#
$S \leftarrow$ new array of n integers	n
$A \leftarrow$ new empty stack	1
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
while $(\neg A.empty() \wedge$	
$X[A.top()] \leq X[i]\ )$ **do**	n
$A.pop()$	n
if $A.empty()$ **then**	n
$S[i] \leftarrow i + 1$	n
else	
$S[i] \leftarrow i - A.top()$	n
$A.push(i)$	n
return S	1

Questions?