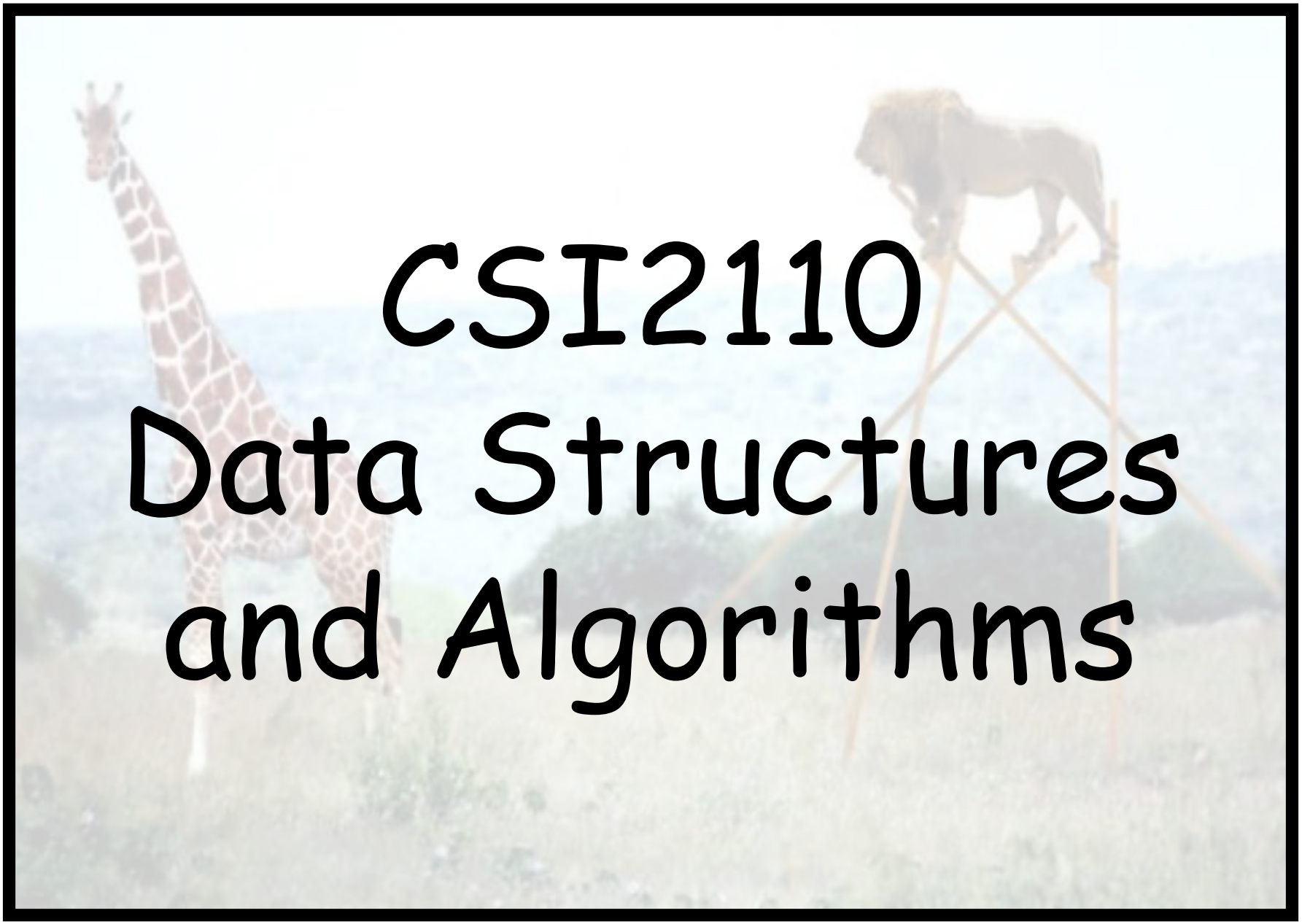

CSI2110
Data Structures
and Algorithms

Trying to overcome the limitations of Array based Stacks and Queues

A Growable Array-Based Stack

Two strategies:

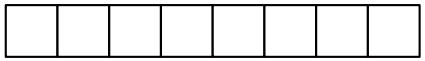
- **tight strategy**
- **growth strategy**

Growth Strategy

A full

Create B

B



Copy A into B

B

Reassign reference A to the new array

A

A Growable Array-Based Stack

Idea: when the array S is full, replace it with a larger one and continue processing push operations.

```
Algorithm push(obj):
        if size() = N then
                A ← new array of length f(N)
                for i ← 0 to N - 1
                        A[i] ← S[i]
                S ← A
        t ← t + 1
        S[t] ← obj
```

- How large should the new array be?
  - **tight strategy** (add a constant): f(N) = N + c
  - **growth strategy** (double up): f(N) = 2N

Tight vs. Growth Strategies: a comparison

- To compare the two strategies, we use the following cost model:

OPERATION	RUN TIME
regular push operation: add one element	1
special push operation: create an array of size f(N), copy N elements, and add one element	f(N)+N+1

Tight Strategy (c=4)

$f(N) = N + c = N + 4$

- start with an array of size 0
- the cost of a special push is

$f(N)+N+1 = 2N + 5$

$= N + c + N + 1$

$= 2N + c + 1$ in general

push	phase	N	cost
1	1	0	5
2	1	4	1
3	1	4	1
4	1	4	1
5	2	4	13
6	2	8	1
7	2	8	1
8	2	8	1
9	3	8	21
10	3	12	1
11	3	12	1
12	3	12	1
13	4	12	29

c =4

5 = 2N + c + 1 = c+1

1
1
1
13 = } c-1

Now N = c

1 2 c + c + 1
1
1
21

Now N = 2c

1
1
1 = 2 (2 c) + c + 1

push	phase	N	cost
1	1	0	5
2	1	4	1
3	1	4	1
4	1	4	1
5	2	4	13
6	2	8	1
7	2	8	1
8	2	8	1
9	3	8	21
10	3	12	1
11	3	12	1
12	3	12	1
13	4	12	29

Phase 1: c+1 + (c-1) = 2c

Phase 2: 2 c + c + 1 + (c-1) = 4c

Phase 3: 2 (2 c) + c + 1 + (c-1) = 6c

Phase i = 2ci

Performance of the Tight Strategy

- We consider k phases, where $k = n/c$
- Each phase corresponds to a new array size
- The cost of phase i is $2ci$
- The total cost of n push operations is the total cost of k phases, with $k = n/c$:

  $$\mathbf{2c\,(1 + 2 + 3 + \ldots + k)},$$

  which is $O(k^2)$ and $\mathbf{O(n^2)}$.

Growth Strategy (double up): f(N) = 2N

- start with an array of size 0, then 1, 2, 4, 8, ...
- the cost of a special push is

f(N)+N+1 =

2N + N + 1 =

3N + 1

for N>0

push	phase	N	cost
1	0	0	1
2	1	1	4
3	2	2	7
4	2	4	1
5	3	4	13
6	3	8	1
7	3	8	1
8	3	8	1
9	4	8	25
10	4	16	1
11	4	16	1
12	4	16	1
...	...	...	...
16	4	16	1
17	5	16	49

Phase i > 1

Special

Normal

...

Normal

2^{i-1}

push	phase	N	cost
1	0	0	1
2	1	1	4
3	2	2	7
4	2	4	1
5	3	4	13
6	3	8	1
7	3	8	1
8	3	8	1
9	4	8	25
10	4	16	1
11	4	16	1
12	4	16	1
...	...	...	...
16	4	16	1
17	5	16	49

Cost of special operations: $3N + 1$

Phase 0: 1 $\quad 0 + 1$

Phase 1: 4 $\quad 3 \cdot 1 + 1$

Phase 2: 7 $\quad 3 \cdot 2 + 1$

Phase 3: 13 $\quad 3 \cdot 4 + 1$

Phase i: $\quad 3 \cdot 2^{i-1} + 1$

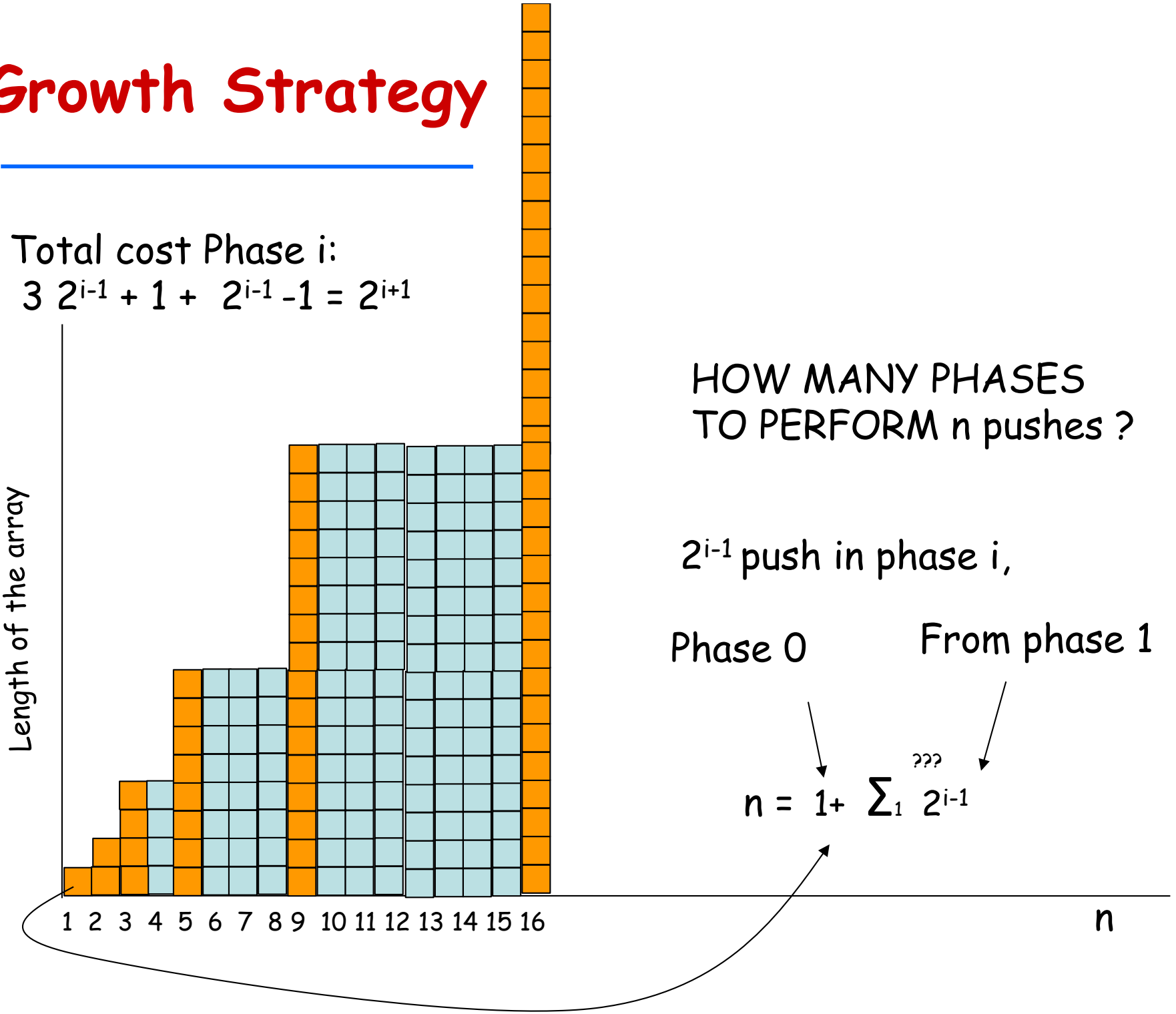
TOT Phase i: $\quad 2^{i-1} + 3 \cdot 2^{i-1} + 1$

$= 2^{i+1}$

Growth Strategy

Total cost Phase i:

$3\ 2^{i-1} + 1 + 2^{i-1} - 1 = 2^{i+1}$



HOW MANY PHASES
TO PERFORM n pushes ?

2^{i-1} push in phase i,

Phase 0 From phase 1

$n = 1 + \sum_{1}^{???} 2^{i-1}$

Performance of the Growth Strategy

- We consider k phases, where $k = \log n$
- Each phase corresponds to a new array size
- The cost of phase i is 2^{i+1}
- The total cost of n push operations is the total cost of k phases, with $k = \log n$
- $2 + 4 + 8 + \ldots + 2^{\log n + 1} = 2\,(\,1 + 2 + \ldots + 2^{\log n}\,)$

  $= 2((2^{\log n + 1} - 1)/(2-1)) = 2\,(\,2(n) - 1\,) = 4n - 2$

- $S = \sum_{i=0}^{m} r^i = 1 + r + r^2 + \ldots + r^m = (r^{m+1} - 1)/(r-1)$
  - $b = a^{\log_a b} \rightarrow b = 2^{\log b} \rightarrow n = 2^{\log n}$
- The growth strategy wins! **O(n)**

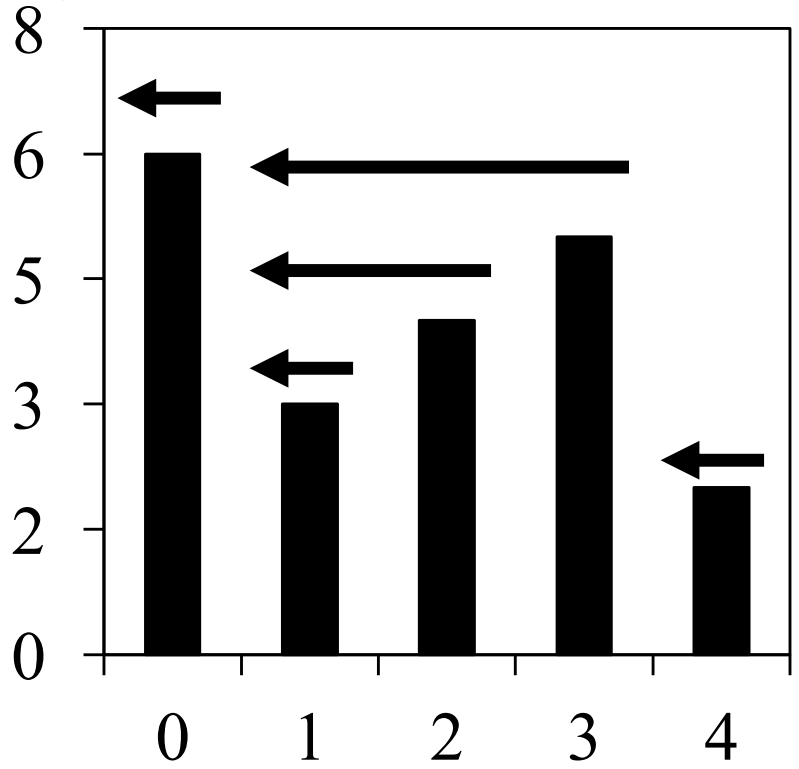
Another Example of the use of a Stack

Computing Spans

Given a series of n daily price quotes for a stock, we call the span of the stock's price on a certain day the maximum number of consecutive days up to the current day that the price of the stock has been less than or equal to its price on that day.

Computing Spans

- 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] ≤ **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

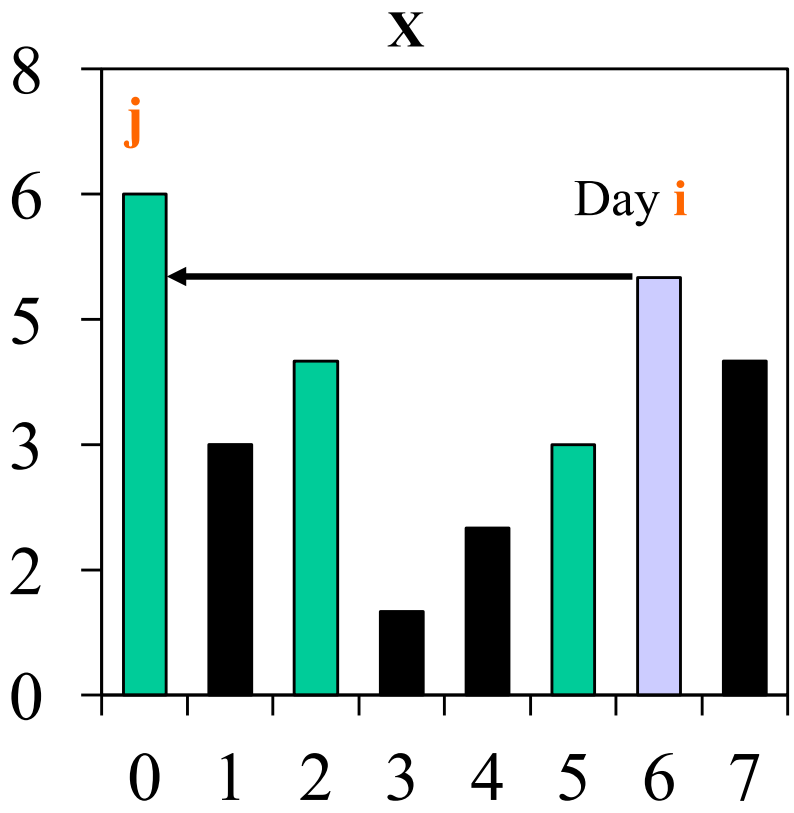
Quadratic Algorithm

```
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
```

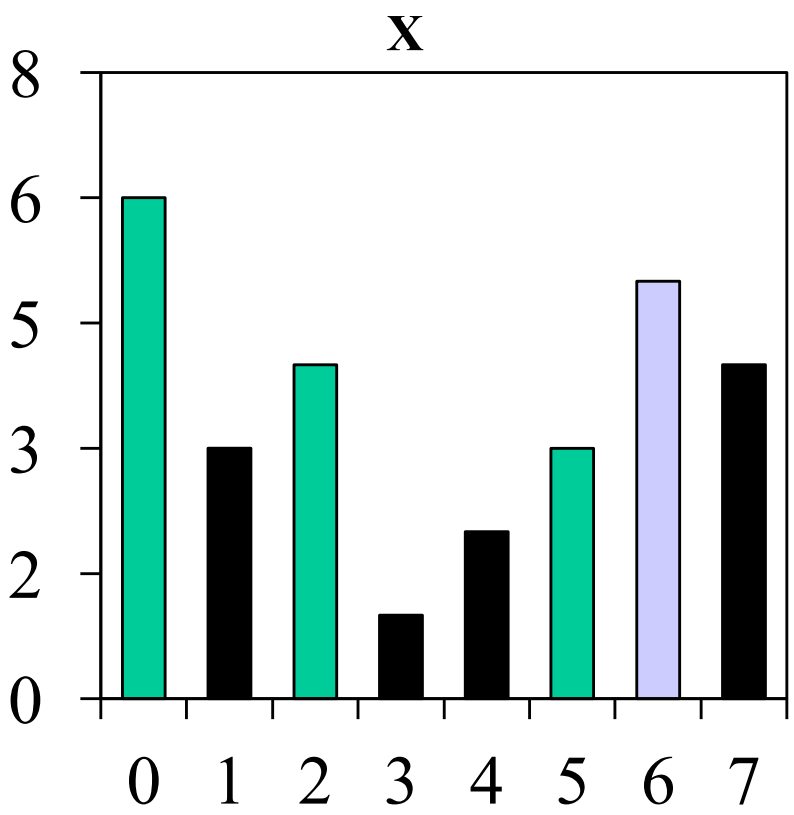
- Algorithm ***spans1*** runs in $O(n^2)$ time

Computing Spans with a Stack

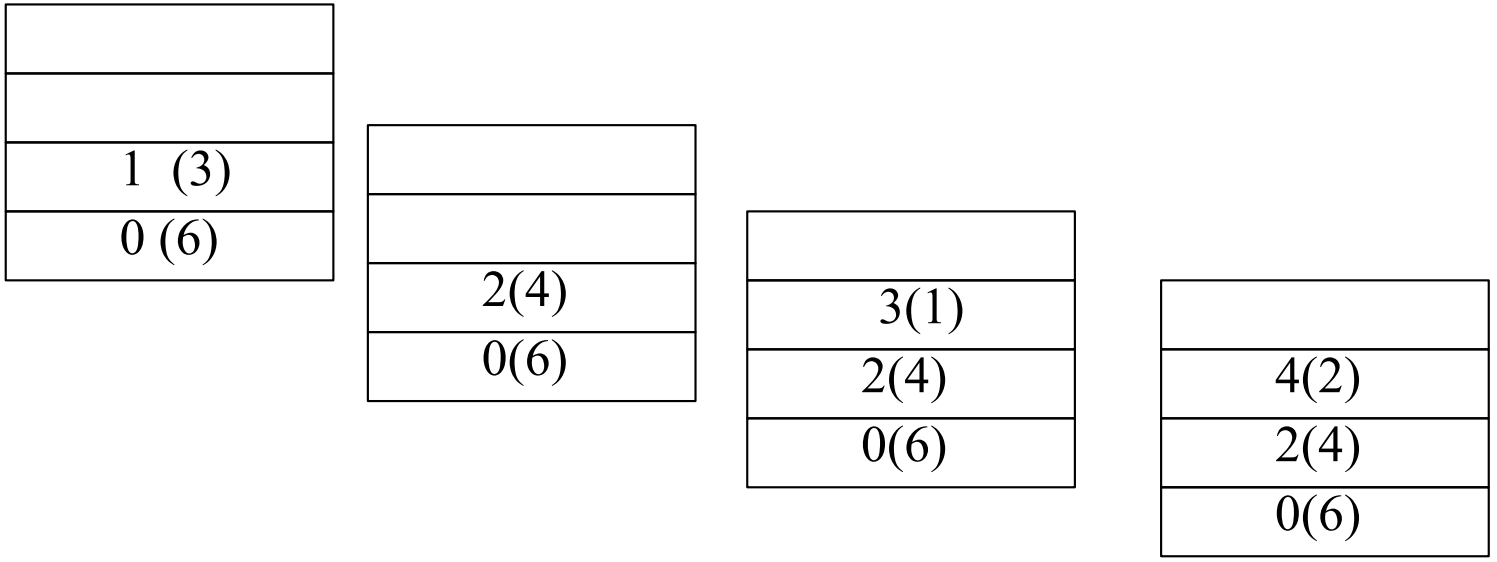
Closest higher element preceding i



$S_i = i - j$



Index: 0, 1, 2, 3, 4, 5, 6, 7
X : 6, 3, 4, 1, 2, 3, 5, 4
S : 1, 1, 2, 1, 2, 3, 6, 1



- 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

    $X[j] > X[i]$

  - We set $S[i] \leftarrow i - j$

  - We push **i** 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 computeSpans2(P):
Input: an n-element array P of numbers
    representing stock prices
Output: an n-element array S of numbers such
    that S[i] is the span of the stock on day i
    Let D be an empty stack
    for i ←0 to n - 1 do k ← 0
     done ← false
     while not(D.isEmpty() or done) do
        if P[i] ≥ P[D.top()] then
           D.pop()
        else done ← true
     if D.isEmpty() then h ← -1
     else h ← D.top()
     S[i] ← i - h
     D.push(i)
    return S
```