Chapter 1:
Stock Spans

Panos Louridas

Athens University of Economics and Business
Real World Algorithms
A Beginners Guide
The MIT Press

Outline

Definition

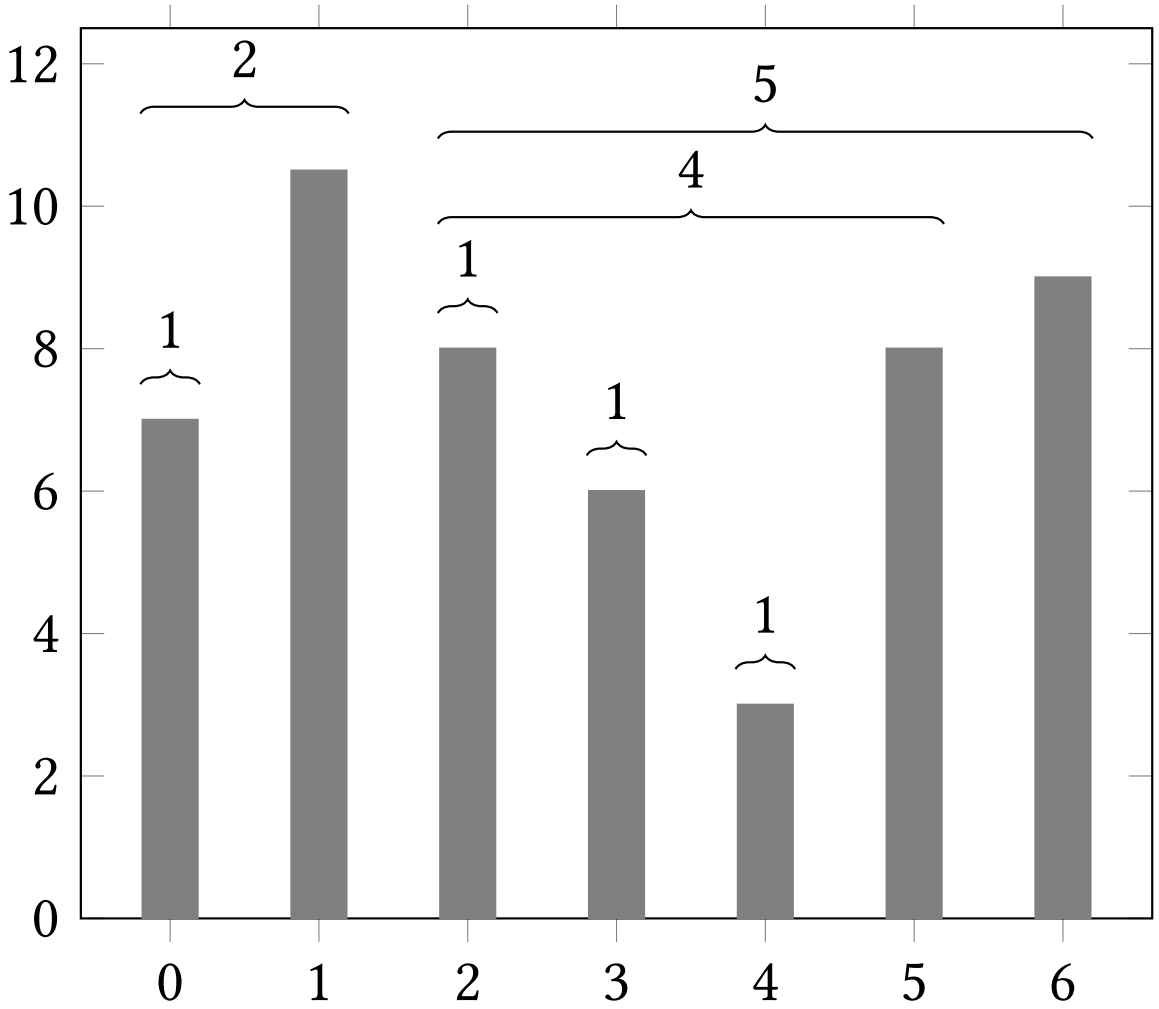
The Stock Span Problem

- Suppose that we have a series of stock quotes.
- For each day we have the closing price for the stock.

Question

For each day, how many consecutive days that lead to that day have a closing price lower than the price on that day?

Example



Solution

- We take each day in turn.
- We assume that the span for that day is equal to one (that is, it holds for that day only).
- We go back over the previous days.
- If the closing price is less than or equal to the price on the day we examine, we increase the span by one.
- Otherwise we have found the span for that day.
- We go to the previous day.

Outline

Definition

An algorithm is a procedure with the following characteristics:

- It comprises a finite sequence of steps.
- It must finish at some finite time.
- Each step must be well defined.
- It takes some input.
- It produces some output.

A Simple Stock Span Algorithm

Algorithm: A Simple Stock Span Algorithm

`SimpleStockSpan`($quotes$) $\longrightarrow$ $spans$

Input: $quotes$, an array with n stock price quotes

Output: $spans$, an array with n stock price spans

```
1  spans ← CreateArray(n)
2  for i ← 0 to n do
3      k ← 1
4      span_end ← FALSE
5      while i − k ≥ 0 and not span_end do
6          if quotes[i − k] ≤ quotes[i] then
7              k ← k + 1
8          else
9              span_end ← TRUE
10     spans[i] ← k
11 return spans
```

Analysis

Suppose we have n values.

- The outer loop (line 2) is executed n times.
- The inner loop (line 5) is executed as many times as necessary per day.
- At worst, if all values are in ascending order, line 7 will be executed:

$$1 + 2 + \cdots + n$$

  times.
- This is equal to:

$$\frac{n(n+1)}{2}$$

Outline

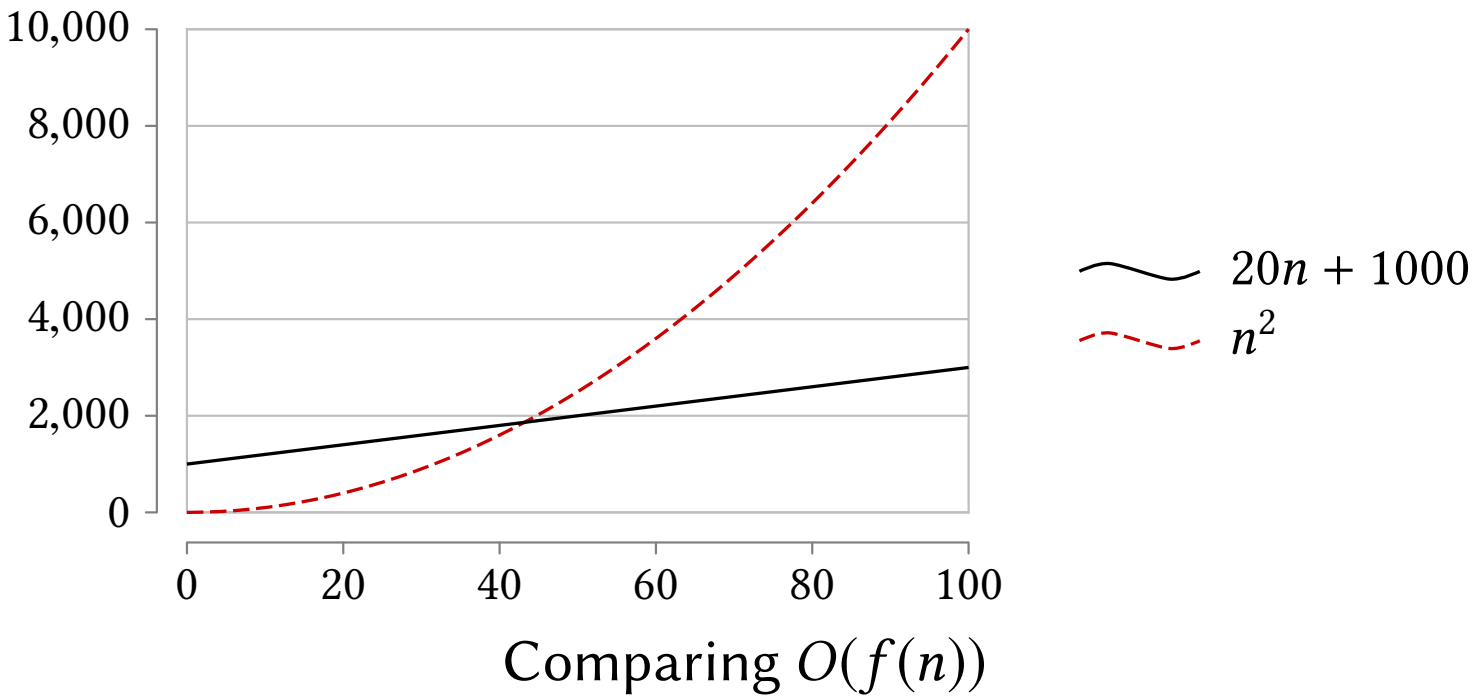
$O(f(n))$ Notation

Definition

Suppose we have a function $f(n)$. If there exists an $n_0 > 0$ such that, for all $n \geq n_0$, we have $0 \leq f(n) \leq cg(n)$, where $c > 0$ is some constant, we say $O(f(n)) = g(n)$.

$O(f(n))$ gives us an *upper limit* on the algorithm's performance.

Comparing $O(f(n))$



Simplifying Complexity Functions

- To calculate an algorithm's complexity we can simplify the functions we use.
- If we have a function like $f(n) = 3n^3 + 5n^2 + 2n + 1000$, then $O(f(n)) = n^3$.
- Why? Because we can always find some c such that $f(n) < cn^3$.
- In general, when we have a function with many parts, the largest one dominates, so we can ignore the rest.
- So, for polynomials we have:

$$O(a_1 n^k + a_2 n^{k-1} + \cdots + a_n n + b) = O(n^k)$$

Complexity Functions

- Most algorithms belong to one of a group of complexity families.
- These are:
  - Constant time / complexity.
  - Logarithmic time / complexity.
  - Linear time / complexity.
  - Loglinear time / complexity.
  - Polynomial time / complexity.
  - Exponential time / complexity.
  - Factorial time / complexity.

Constant Time Algorithms

- The constant function is $f(n) = c$.
- It always takes the same value, for all n.
- As there exist constants c and $n_0 = 1$ such that $0 \leq f(n) \leq cg(n) = c \cdot 1$, we have $O(c) = O(1)$.
- We call such algorithms *constant time algorithms*.
- That does not mean that they take always the same time; it means that in the worst case they take the same time. For example: an algorithm that adds a value y to a value x if $x < 0$.

Example of Constant Time Algorithm

- Accessing an element in a matrix is done in constant time, irrespective of the position of the element.
- If we have a matrix A with n elements, accessing $A[0]$ takes the same time with accessing $A[n-1]$.

Logarithmic Time Algorithms

- The logarithmic function, or simply logarithm, is the function $f(n) = \log_a(n)$ that gives the power to which we must raise a to obtain n: if $y = \log_a(n)$ then $n = a^y$.
- The number a is the *base of the logarithm.*
- We have $x = a^{\log_a x}$, that is, raising to a power and taking a logarithm are inverse functions: $\log_3 27 = 3$ and $3^3 = 27$.
- If $a = 10$ we write $y = \log(n)$.
- If $a = 2$ we write $\lg(n) = \log_2(n)$.
- If $a = e$ we write $\ln(n) = \log_e(n)$, where $e \approx 2.71828$. This is the *natural logarithm.*

Logarithmic Time Algorithms (Contd.)

- We have $\log_a(n) = \log_b(n)/\log_b(a)$. For example, $\lg(n) = \log_{10}(n)/\log_{10}(2)$.
- So we can write simply $O(\log n)$. We also often write $O(\lg n)$.
- Logarithmic time algorithms appear in problems that are solved with divide-and-conquer techniques. Such are the best search algorithms.

Linear Time Algorithms

- The linear function is $f(n) = n$, so $O(n)$.
- Algorithms that run in linear time run in time proportional to the problem of the size they solve.
- An example is searching for something in an unordered list.

Loglinear Time Algorithms

- The loglinear function is $f(n) = n\log(n)$, so we write $O(n\log(n))$.
- These algorithms are usually a combination of a linear and a logarithmic algorithm.
- They may contain a divide-and-conquer strategy, where each piece takes linear time.
- Good sorting algorithms have loglinear complexity.

Polynomial Time Algorithms

- When the function that describes the running time of an algorithm is a *polynomial* $f(n) = (a_1 n^k + a_2 n^{k-1} + \cdots + a_n n + b)$, we say that we have polynomial complexity $O(n^k)$.
- An important group is the algorithms that run in $O(n^2)$ time. These algorithms are called *quadratic time algorithms*.
- Such algorithms are some not particularly efficient sorting algorithms and the classic (school) algorithm for multiplying two n digits algorithms—but note that we can do multiplication faster.

Exponential Time Algorithms

- Exponential time algorithms run in $f(n) = c^n$ time, where c is a constant, so we write $O(c^n)$.
- Note the difference between n^c and c^n!
- The *exponential function* is the special case $c = e$: $f(n) = e^n$.

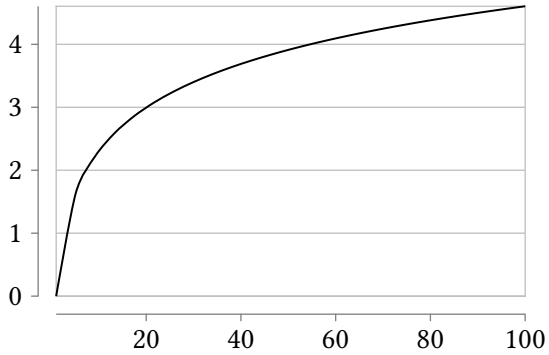
Exponential Time Algorithms (Contd.)

- Exponential algorithms appear when we have a problem of size n, each one of the n inputs can take c different values, and we must examine all possible cases.
- We have c possible values for the first input.
- For each one of them we have c possible values for the second input, so for the first two inputs we have $c \times c = c^2$.
- For each of these c^2 cases we have c possible values for the third input, so in total $c^2 \times c = c^3$.
- For n inputs we get c^n.
- For example, if a password has c characters and the alphabet has n symbols, we will need at most c^n guesses to break the password using a *brute force* method.

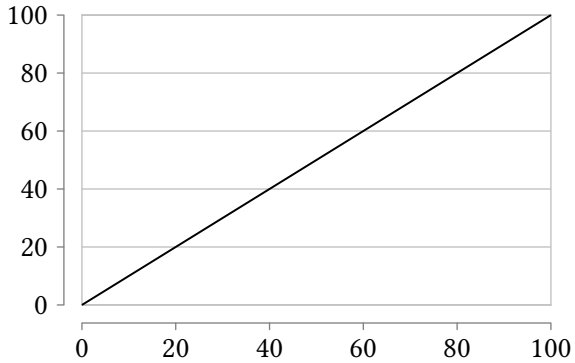
Factorial Time Algorithms

- The *factorial* of a number n is the number $n! = 1 \times 2 \times \cdots \times n$, with $0! = 1$.
- Factorial time algorithms appear when to solve a problem we have examine all the input *permutations*.
- For example, if we have as input $[1, 2, 3]$, the possible permutations are: $[1, 2, 3]$, $[1, 3, 2]$, $[2, 1, 3]$, $[2, 3, 1]$, $[3, 1, 2]$, and $[3, 2, 1]$.
- In general, we have n different values in the first position. In the second position we can have $n - 1$ different values, as we have already used one. In the third position we can have $n - 2$ different values and so on, up to the last position that can take only one value, so in total $n \times (n - 1) \cdots \times 1 = n!$.
- For example, the possible ways to shuffle a deck of cards are $52!$.

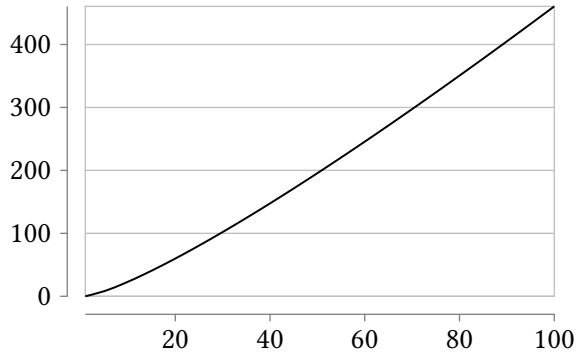
Complexity Graphs



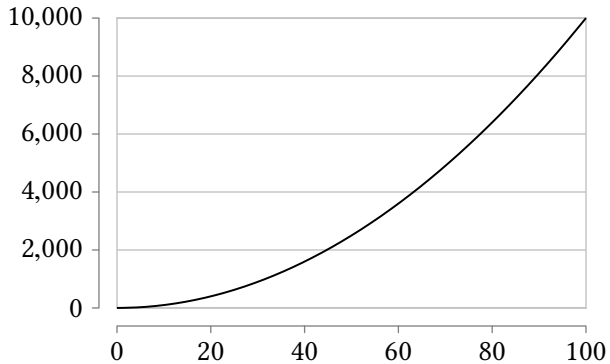
$O(\lg n)$



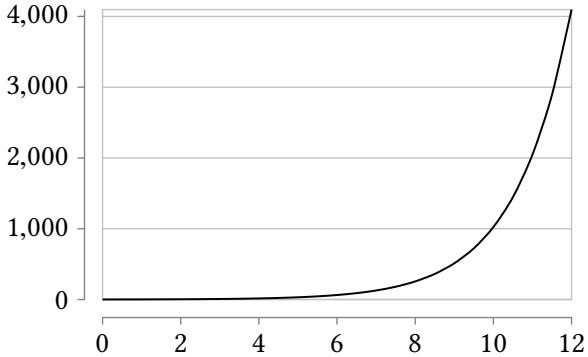
$O(n)$



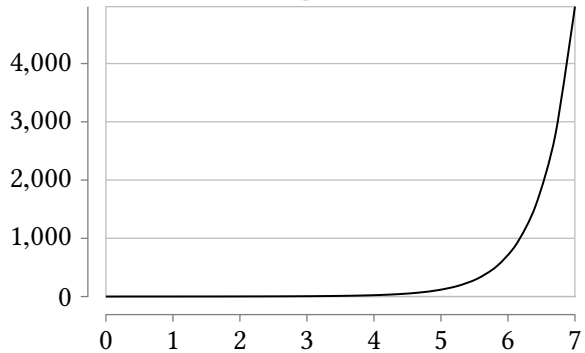
$O(n \lg n)$



$O(n^2)$



$O(2^n)$



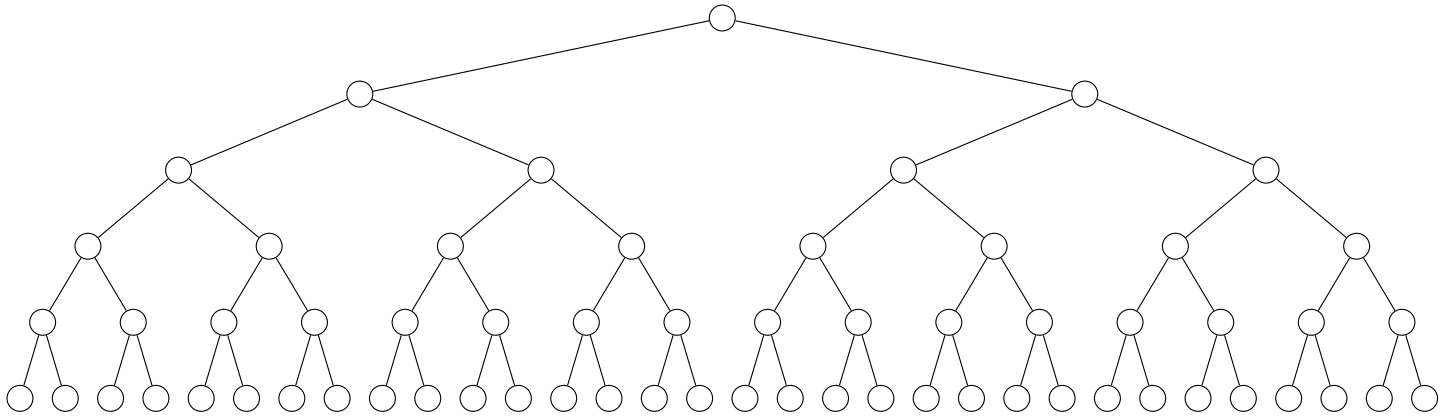
$O(n!)$

Growth of Functions

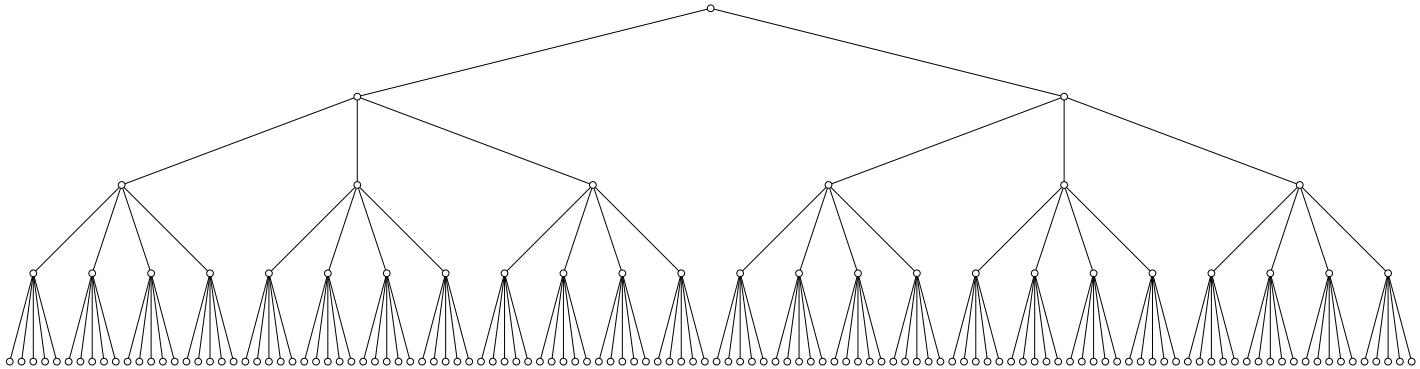
n	1	10	100	1000	1,000,000
$\lg(n)$	0	3.32	6.64	9.97	19.93
n	1	10	100	1000	1,000,000
$n \ln(n)$	0	33.22	664.39	9965.78	1.9×10^7
n^2	1	100	10,000	1,000,000	10^{12}
n^3	1	1000	1,000,000	10^9	10^{18}
2^n	2	1024	1.3×10^{30}	10^{301}	$10^{10^{5.5}}$
$n!$	1	3,628,800	9.33×10^{157}	4×10^{2567}	$10^{10^{6.7}}$

Difference between c^n and $n!$

$2^5 = 32$



$5! = 120$



$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Big Numbers (1)

- 100 billions, or 10^{11}, or 100 *giga*. If you take 100 billion hamburgers and you lay them end to end, you can circle the Earth 216 times, go to the Moon, and come back.
- 1 trillion, or 10^{12}, or 1 *tera*. If you start counting one number per second, you will need 31,000 years to get to one trillion.
- 1 quadrillion, or 10^{15}, or 1 *peta*. According to biologist E. O. Wilson, there are between 1 and 10 quadrillion ants on the planet; i.e., between 1 and 10 petaants.

Big Numbers (2)

- 1 quintillion, or 10^{18}, or one *exa*. That is about the number of grains of sand in 10 large beaches. For example, 10 Cobacabana beaches have one exagrain of sand.
- 1 sextillion, or 10^{21}, or one *zetta*. That is the number of stars in the observable universe.
- 1 septillion, 10^{24}, is one *yotta*.
- 10^{100} is one *googol*.
- $10^{10^{100}}$ is one *googolplex*.

The examples are from Tyson, Neil deGrasse, Michael Abram Strauss, and Richard J. Gott, *Welcome to the Universe: An Astrophysical Tour*, Princeton University Press, 2016.

Acceptable Complexities

- Polynomial complexity ($O(n^k)$) is acceptable, and such algorithms are considered good.
- Given that $O(a_1 n^k + a_2 n^{k-1} + \cdots + a_n n + b) = O(n^k)$, we can always drop all terms except the term with the largest exponent.
- Unfortunately there may not be a polynomial time solution at all.

Non-polynomial Complexity Example

Problem

A salesperson must visit a number of cities, visiting each city exactly once. Each city is directly connected to all other cities. The salesperson must travel the least number of miles.

- To enumerate all possible routes we need $O(n!)$.
- There exist some solutions that take $O(n^2 2^n)$ time.

$\Omega(n)$

Definition

Suppose we have a function $f(n)$. If there exists some $n_0 > 0$ such that for all $n \geq n_0$ we have $f(n) \geq cg(n) \geq 0$, where $c > 0$, then we say that $\Omega(f(n)) = g(n)$.

$\Omega(f(n))$ gives us a *lower limit* to the performance of an algorithm.

$\Theta(n)$

Definition

Suppose we have a function $f(n)$. If and only if there exists some other function $g(n)$ such that $O(f(n)) = g(n)$ and $\Omega(f(n)) = g(n)$, then we say that $\Theta(f(n)) = g(n)$.

Simple Stock Span Algorithm

Algorithm: A Simple Stock Span Algorithm

`SimpleStockSpan`($quotes$) $\rightarrow$ $spans$

Input: $quotes$, an array with n stock price quotes

Output: $spans$, an array with n stock price spans

```
1   spans ← CreateArray(n)
2   for i ← 0 to n do
3       k ← 1
4       span_end ← FALSE
5       while i − k ≥ 0 and not span_end do
6           if quotes[i − k] ≤ quotes[i] then
7               k ← k + 1
8           else
9               span_end ← TRUE
10      spans[i] ← k
11  return spans
```

Analysis (Contd.)

Simple Stock Span Complexity

Line 7 of the algorithm will execute up to

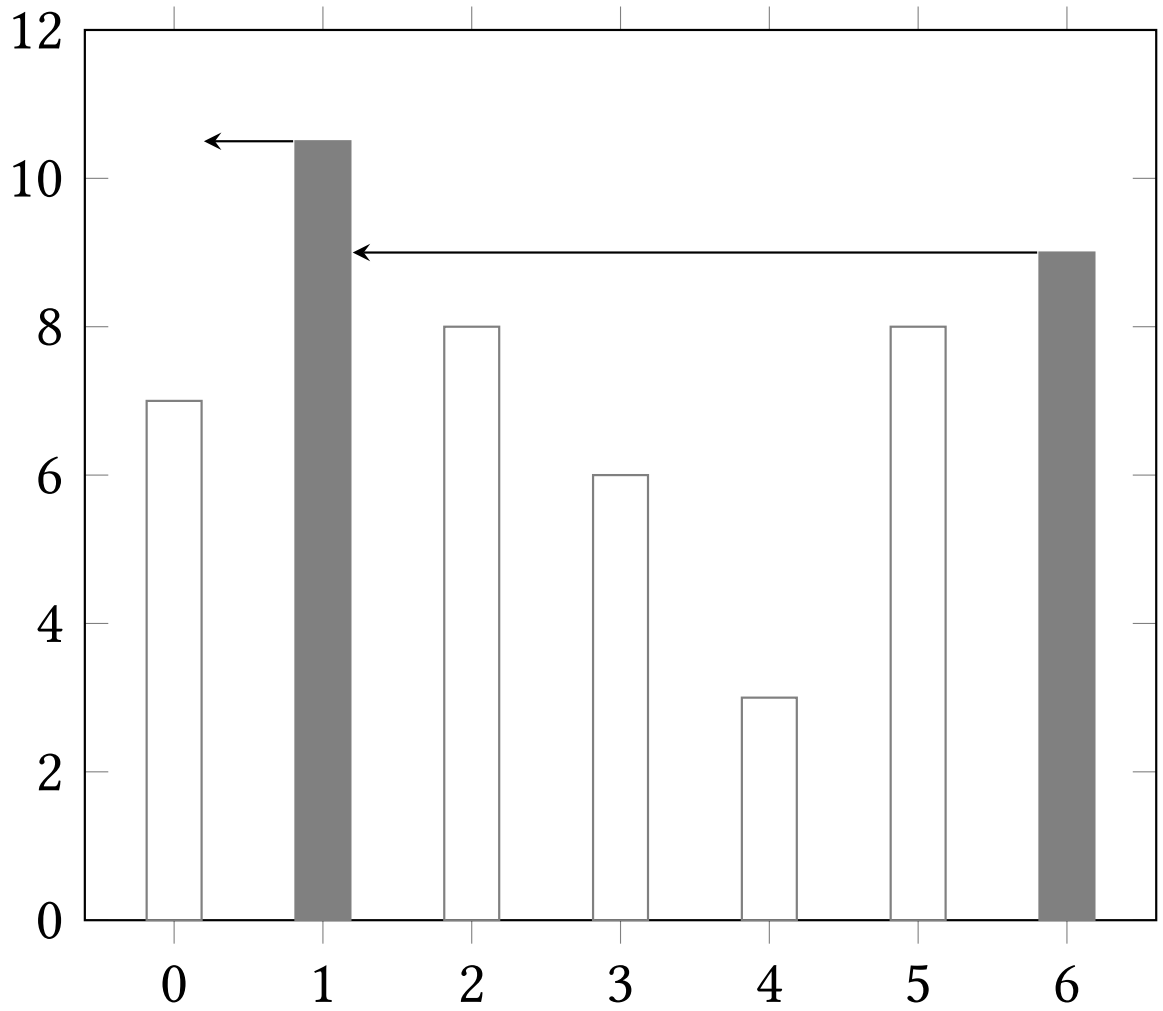
$$\frac{n(n+1)}{2}$$

times, so the overall complexity is:

$$O\left(\frac{n(n+1)}{2}\right) = O(n^2)$$

Can we find a better algorithm?

Back to our Example



Idea

Idea

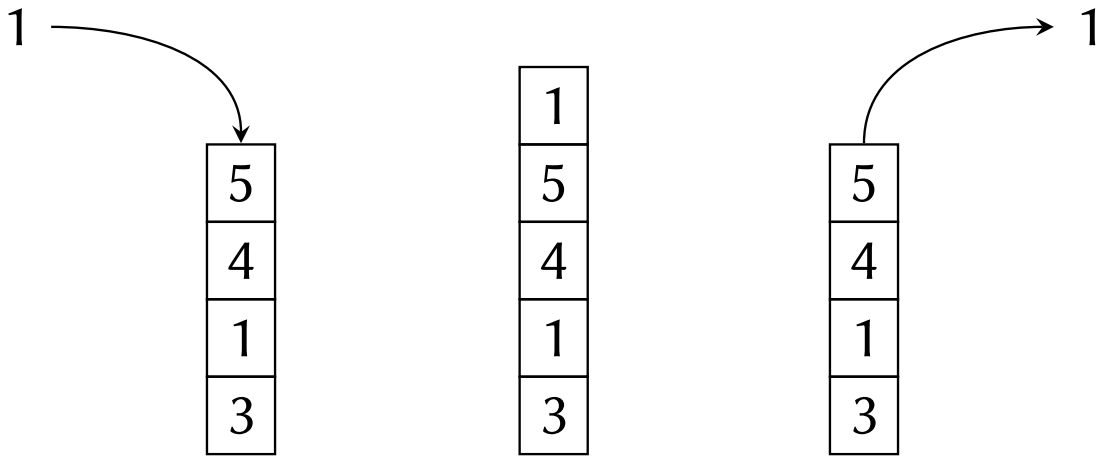
For each day, we only need to examine those days that are above our line-of-sight. We can ignore all other days.

Outline

Stacks

- A *stack* is a *data structure*.
- A data structure is something that contains data.
- A data structure allows specific operations on the data it contains.

Stack Example



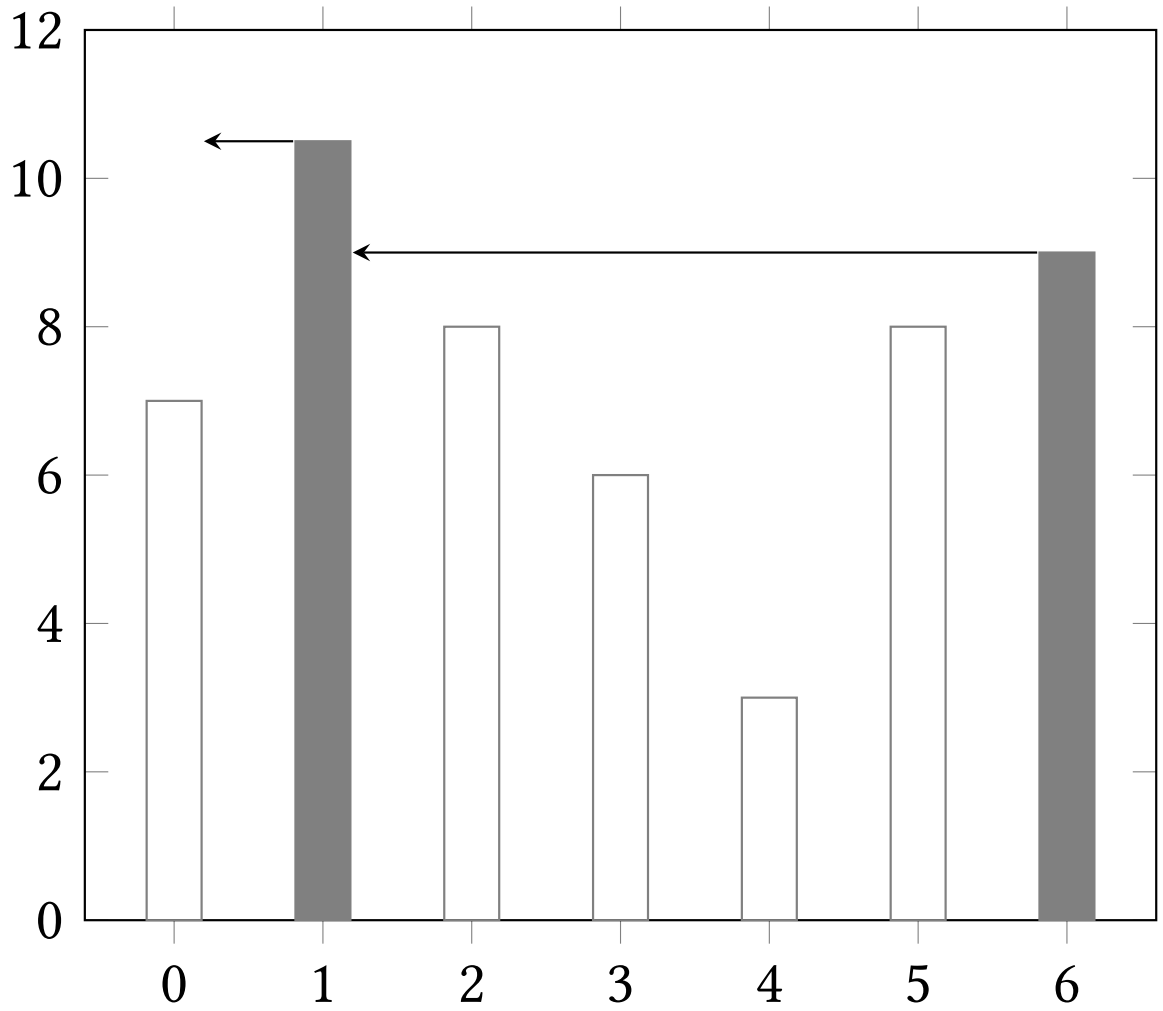
Insertion in a stack: Last In First Out (LIFO)

Stack Operations

- `CreateStack()` creates an empty stack.
- `Push`(S, i) pushes item i on the top of the stack S.
- `Pop`(S) pops the item that is on the top of the stack S. The operation returns the item. If the stack is empty, then the operation is not allowed (we get an error).
- `Top`(S) we get the value of the item on the top of the stack S without removing it. The stack remains the same. If the stack is empty, then the operation again is not allowed and we get an error.
- `IsStackEmpty`(S) returns TRUE if stack S is empty, or FALSE otherwise.

Outline

Back to our Example (Again)

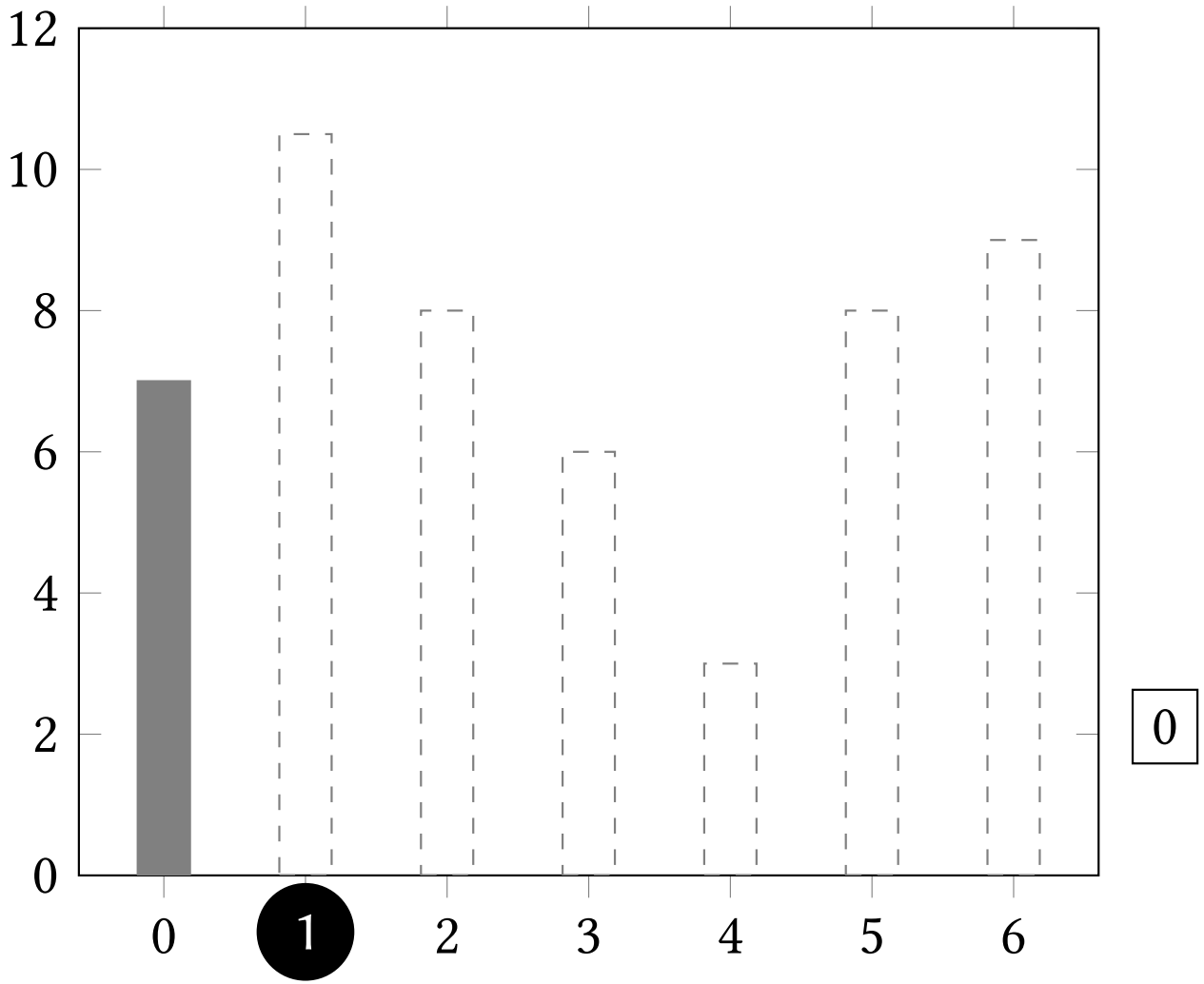


Using a Stack

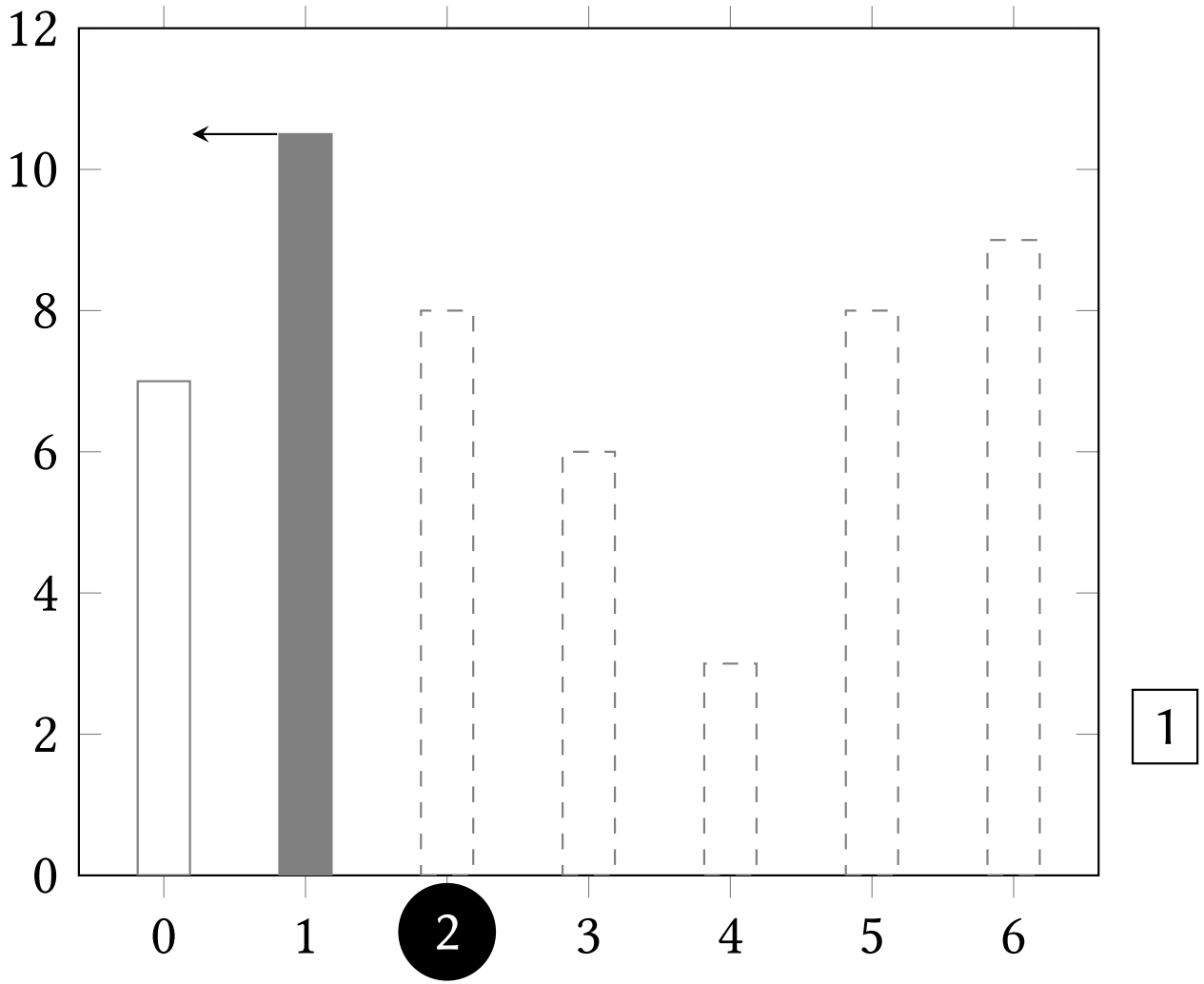
Idea

We will use a stack to keep the days that are above our line-of-sight.

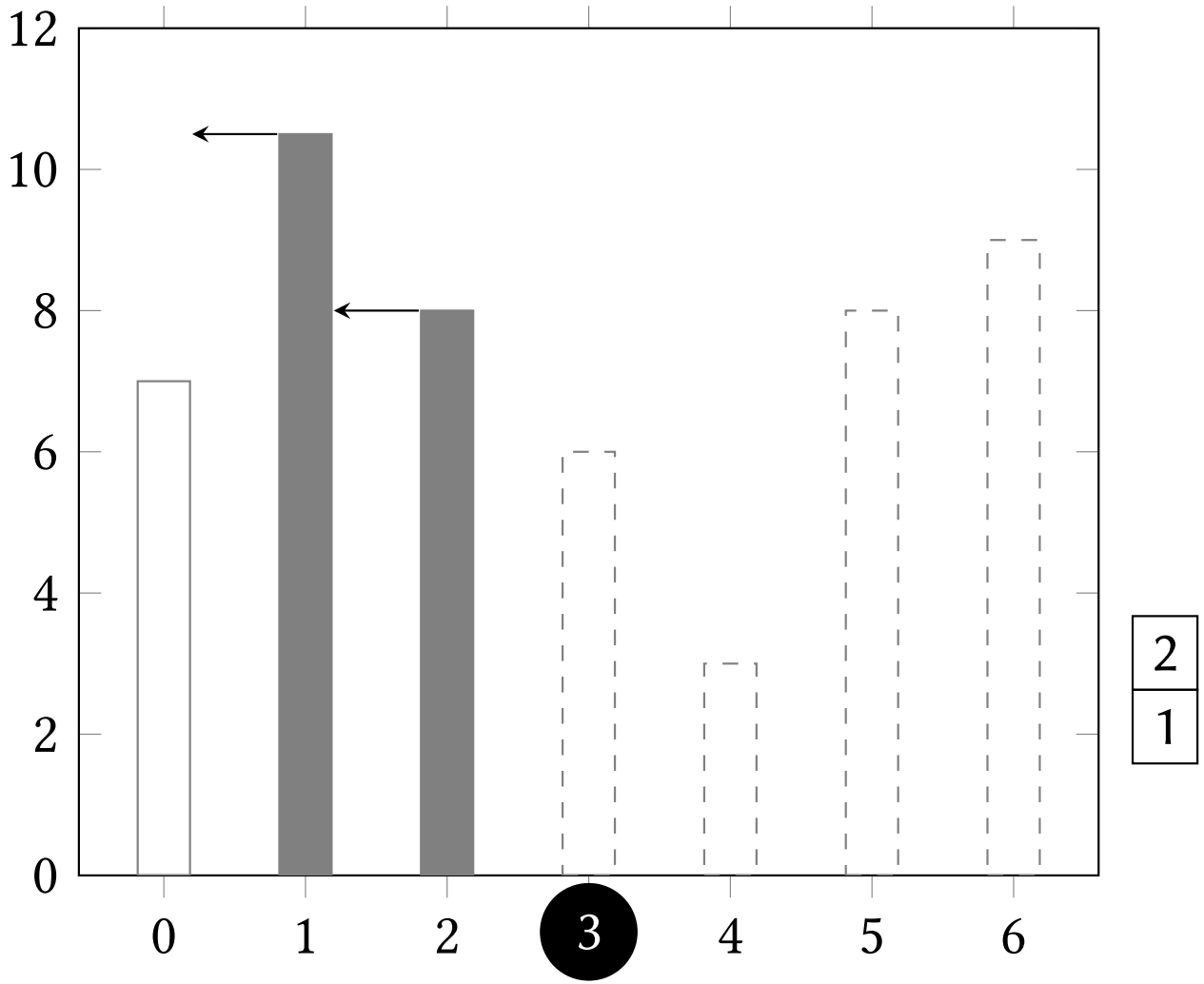
Using a Stack (1)



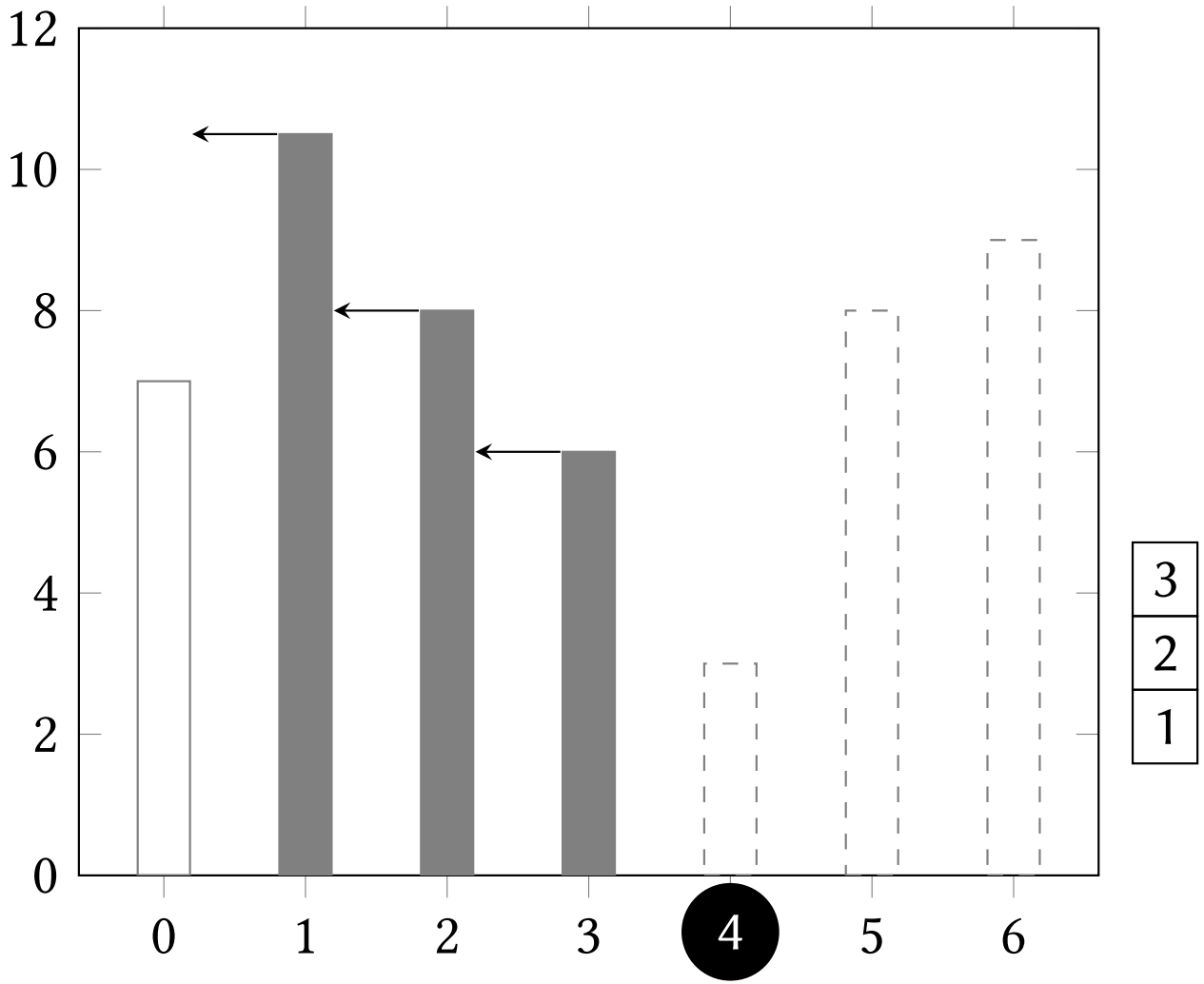
Using a Stack (2)



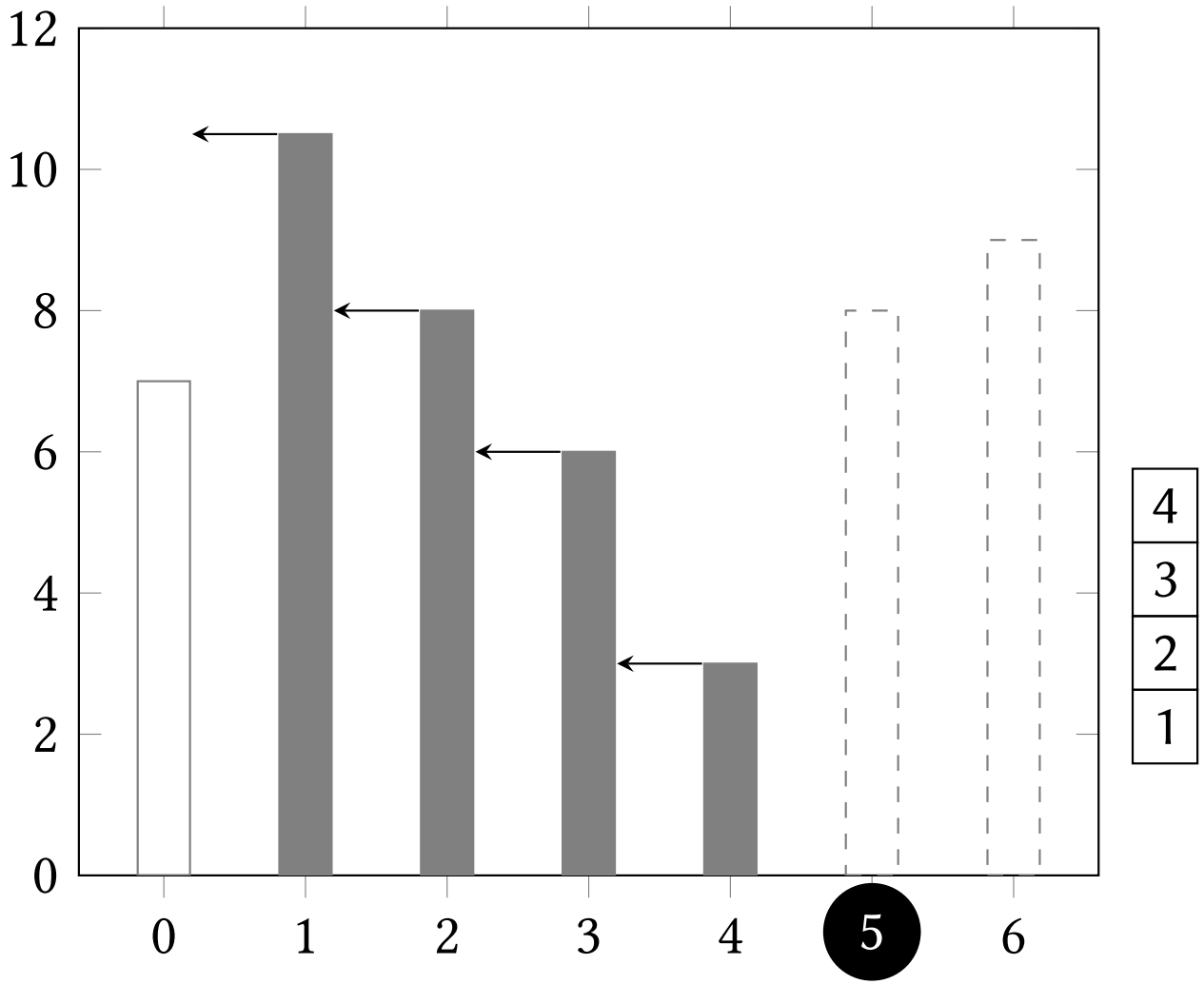
Using a Stack (3)



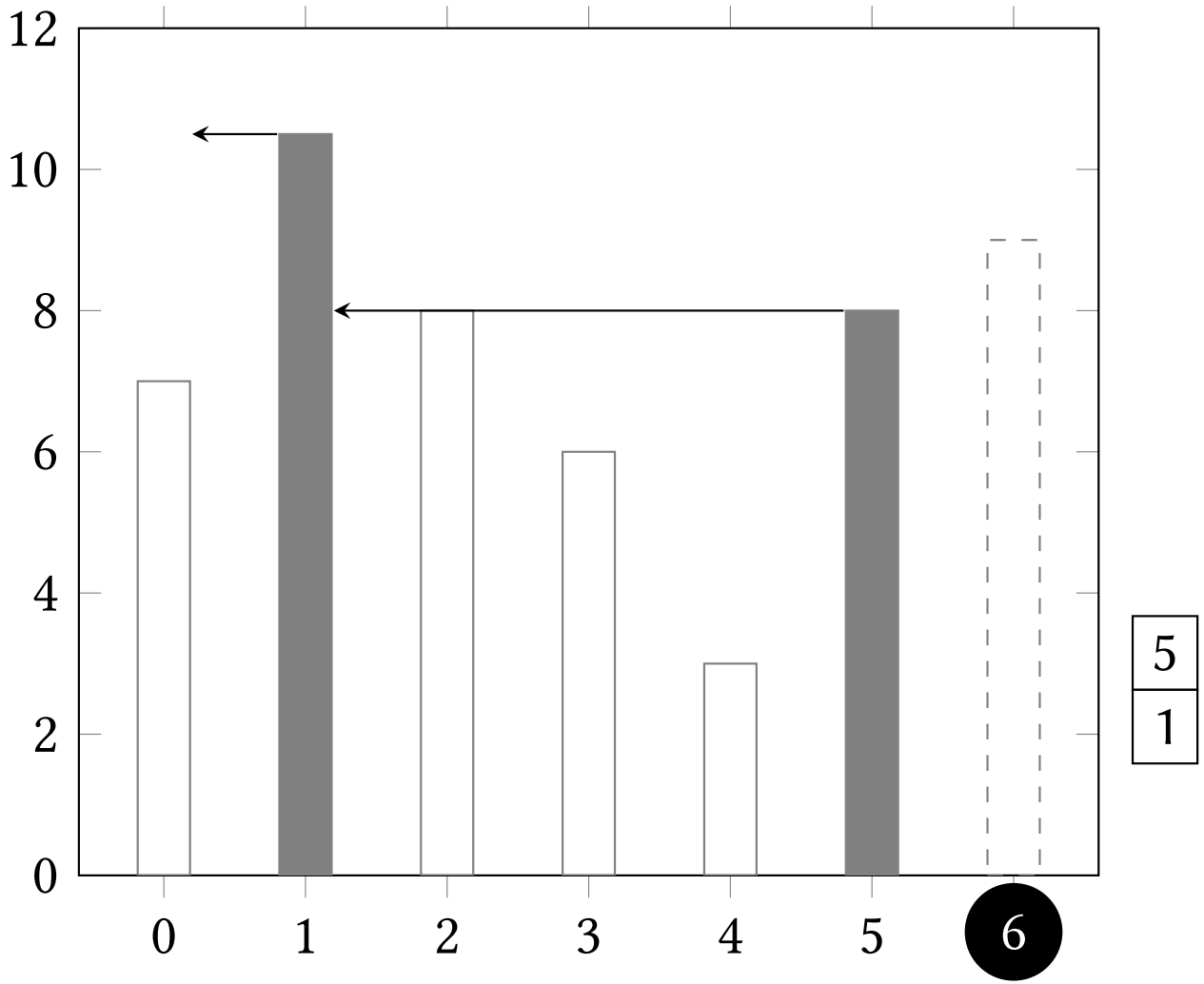
Using a Stack (4)



Using a Stack (5)



Using a Stack (6)



Stack Stock Span Algorithm

Algorithm: Stack Stock Span Algorithm

StackStockSpan($quotes$) $\rightarrow$ $spans$

Input: $quotes$, an array with n stock price quotes

Output: $spans$, an array with n stock price spans

```
1  spans ← CreateArray(n)
2  spans[0] ← 1
3  S ← CreateStack()
4  Push(S, 0)
5  for i ← 1 to n do
6      while not IsStackEmpty(S) and quotes[Top(S)] ≤ quotes[i] do
7          Pop(S)
8      if IsStackEmpty(S) then
9          spans[i] ← i + 1
10     else
11         spans[i] ← i − Top(S)
12     Push(S, i)
13 return spans
```

Analysis of the Improved Algorithm

- The loop starting in line 5 is executed $n - 1$ times.
- For each one of them, the Pop operation in line 7 is executed p_i times.
- In total, for n repetitions it is executed $p_1 + p_2 + \cdots + p_{n-1}$ times.
- We don't know p_i.
- But we do know that each day enters the stack in lines 4, 12 only once.
- So, for sure, $p_1 + p_2 + \cdots + p_{n-1} < n$ (the last day is pushed, but not popped).
- Therefore, the overall complexity is $O(n)$, against $O(n^2)$ we had before.
- The algorithm needs at least n steps, as it will examine n days, so it is also $\Omega(n)$.
- Hence, we finally get $\Theta(n)$.

Short Circuit Evaluation

The correctness of line 6 of the algorithm hinges on *short circuit evaluation*.

- Short circuit evaluation means that the calculation of a logical (boolean) expression stops as soon as we determine its final value.
- **if** $x > 0$ **and** $y > 0$, if $x \leq 0$, then the expression is false, irrespective of y.
- **if** $x > 0$ **or** $y > 0$, if $x > 0$, then the expression is true, irrespective of y.
- In our algorithm, we do not need to worry that the stack may be empty, in which case `Top`(S) would fail: we are guarded by **not** `IsStackEmpty`(S).

Short Circuit Evaluation (Contd.)

operator	a	b	result
and	T	T	T
	T	F	F
	F	T/F	F
or	T	T/F	T
	F	T	T
	F	F	F