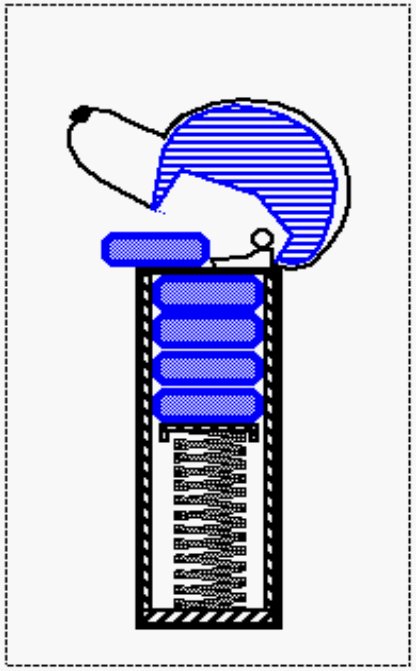
Stacks

- Abstract Data Types (ADTs)
- Stacks
- Interfaces and exceptions
- Java implementation of a stack
- Application to the analysis of a time series
- Growable stacks
- Stacks in the Java virtual machine



Abstract Data Types (ADTs)

- *ADT* is a mathematically specified entity that defines a set of its *instances,* with:
  - a specific *interface* – a collection of signatures of operations that can be invoked on an instance,
  - a set of *axioms* (*preconditions* and *postconditions*) that define the semantics of the operations (i.e., what the operations do to instances of the ADT, but not how)

Abstract Data Types (ADTs)

Types of operations:

- Constructors
- Access functions
- Manipulation procedures

Abstract Data Types

- Why do we need to talk about ADTs in a DS course?
  - They serve as *specifications* of *requirements* for the building blocks of solutions to algorithmic problems
  - Provides a language to talk on a higher level of abstraction
  - ADTs encapsulate *data structures* and algorithms that *implement* them
  - Separate the issues of *correctness* and *efficiency*

Example - Dynamic Sets

- We will deal with ADTs, instances of which are sets of some type of elements.
  - Operations are provided that change the set
- We call such class of ADTs *dynamic sets*

Dynamic Sets (2)

- An example dynamic set ADT
  - Methods:
    - **New():***ADT*
    - **Insert(S:***ADT*, **v:***element***):***ADT*
    - **Delete(S:***ADT*, **v:***element***):***ADT*
    - **IsIn(S:***ADT*, **v:***element***):***boolean*
  - **Insert** and **Delete** – *manipulation* operations
  - **IsIn** – *Access method* method

Dynamic Sets (3)

- Axioms that define the methods:
  - **IsIn(New(), v)** = *false*
  - **IsIn(Insert(S, v), v)** = *true*
  - **IsIn(Insert(S, u), v)** = **IsIn(S, v)**, *if* $v \neq u$
  - **IsIn(Delete(S, v), v)** = *false*
  - **IsIn(Delete(S, u), v)** = **IsIn(S, v)**, *if* $v \neq u$

Other Examples

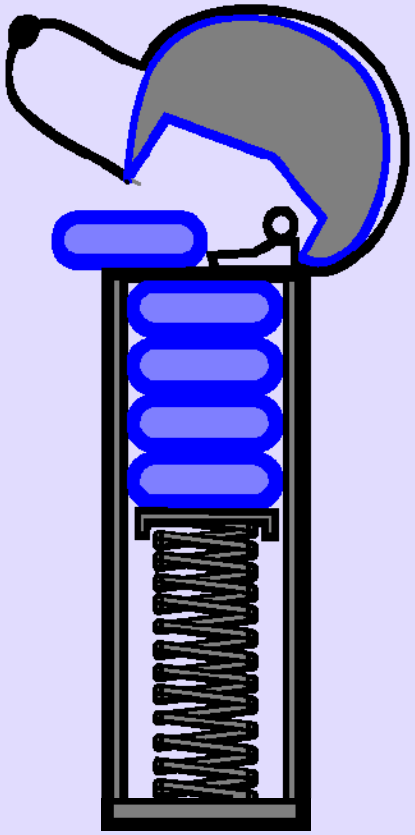
- Simple ADTs:
  - *Queue*
  - *Deque*
  - *Stack*

Stacks

- A **stack** is a container of objects that are inserted and removed according to the last-in-first-out (**LIFO**) principle.
- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as "**pushing**" onto the stack. "**Popping**" off the stack is synonymous with removing an item.

Stacks (2)

- A PEZ ® dispenser as an analogy:



Stacks(3)

- A stack is an ADT that supports four main methods:
  - **new():***ADT* – Creates a new stack
  - **push(S:***ADT*, **o:***element***):***ADT* - Inserts object *o* onto top of stack *S*
  - **pop(S:***ADT***):***ADT* - Removes the top object of stack *S*; if the stack is empty an error occurs
  - **top(S:***ADT***):***element* – Returns the top object of the stack, without removing it; if the stack is empty an error occurs

Stacks(4)

- The following support methods could also be defined:
  - **size(S:*ADT*):*integer*** - Returns the number of objects in stack *S*
  - **isEmpty(S:*ADT*): *boolean*** - Indicates if stack *S* is empty
- Axioms
  - **Pop(Push(S, v)) = S**
  - **Top(Push(S, v)) = v**

Java Stuff

- Given the stack ADT, we need to code the ADT in order to use it in the programs. We need two constructs: interfaces and exceptions.
- An interface is a way to declare what a class is to do. It does not mention how to do it.
  - For an interface, we just write down the method names and the parameters. When specifying parameters, what really matters is their types.
  - Later, when we write a class for that interface, we actually code the content of the methods.
  - Separating interface and implementation is a useful programming technique.

A Stack Interface in Java

The stack data structure is a “built-in” class of Java’s java.util package. But we define our own stack interface:

```
public interface Stack {

// accessor methods
        public int size();
        public boolean isEmpty();
        public Object top()   throws StackEmptyException;

// update methods
        public void push (Object element);
        public Object pop()  throws StackEmptyException;
}
```

Exceptions

- Exceptions are yet another programming construct, useful for handling errors. When we find an error (or an ***exceptional*** case), we just ***throw*** an exception.
- As soon as the exception is thrown, the flow of control exits from the current method and goes to where that method was called from.
- **What is the point of using exceptions?** We can delegate upwards the responsibility of handling an error. Delegating upwards means letting the code who called the current code deal with the problem.

More Exceptions

```
public void eatPizza() throws
StomachAcheException
{...
if (ateTooMuch)
throw new
StomachAcheException("Ouch");
...}
private void simulateMeeting()
{...
try
   {TA.eatPizza();}
catch (StomachAcheException e)
   {system.out.println("somebody has a stomach ache");}
...}
```

So when StomachAcheException is thrown, we exit from method eatPizza() and go to TA.eatpizza().

Even More Exceptions

- The try block and the catch block means that we are listening for exceptions that are specified in the catch parameter.
- Because catch is listening for StomachAcheException, the flow of control will now go to the catch block. And System.out.println will get executed.
- Note that a catch block can contain anything. It does not have to do only System.out.println. We can handle the caught error in any way you like; we can even throw them again.

Exceptions (finally)

- Note that if somewhere in your method, you throw an exception, you need to add a throws clause next to your method name.
- If you never catch an exception, it will propagate upwards and upwards along the chain of method calls until the user sees it.
- What exactly are exceptions in Java? Classes.

```
public class StomachAcheException extends
RuntimeException {
        public StomachAcheException (String err)
                {super(err);}
}
```

Array-Based Stack in Java

- Create a stack using an array by specifying a maximum size *N* for our stack.
- The stack consists of an N-element array *S* and an integer variable *t*, the index of the top element in array S.

- Array indices start at 0, so we initialize *t* to -1

Array-Based Stack in Java (2)

```
public class ArrayStack implements Stack {
        // Implementation of the Stack interface using an array

public static final int CAPACITY = 1024;
                                        // default capacity of stack
private int N;                      // maximum capacity of the stack
private Object S[ ];              // S holds the elements of the stack
private int t = -1;                    // the top element of the stack
public ArrayStack( )  // Initialize the stack with default capacity
        { this(CAPACITY) }
public ArrayStack(int cap)
                              // Initialize the  stack with given capacity
        {N = cap; S = new Object[N]}
```

Array-Based Stack in Java (3)

```
public int size( )                    //Return the current stack size
{return (t + 1)}

public boolean isEmpty( ) //Return true iff  the stack is empty
{return (t < 0)}

public void push(Object obj) throws StackFullException{
                                      //Push a new element on the stack
if (size() == N)
    throw new StackFullException("Stack overflow.")
S[++t] = obj;}
```

Array-Based Stack in Java (4)

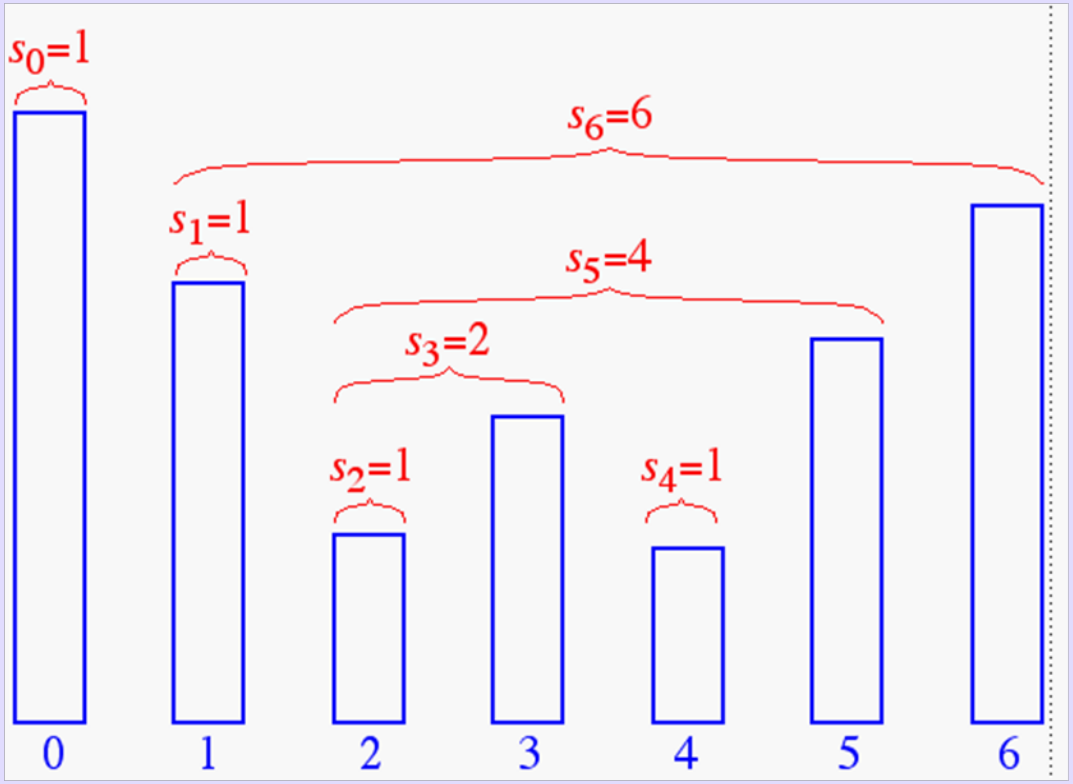
```
public Object top( ) throws StackEmptyException {
                                        // Return the top stack element
if (isEmpty( ))
   throw new StackEmptyException("Stack is empty.");
return S[t]}

public Object pop() throws StackEmptyException {
                                            // Pop off the stack element
Object elem;
if (isEmpty( ))
     throw new StackEmptyException("Stack is Empty.");
elem = S[t];
S[t--] = null;          // Dereference S[top] and decrement top
return elem}}
```

Array-Based Stack in Java (5)

- The array implementation is simple and efficient (methods performed in O(1)).
- There is an upper bound, *N*, on the size of the stack. The arbitrary value *N* may be too small for a given application, or a waste of memory.
- StackEmptyException is required by the interface.
- StackFullException is particular to this implementation.

Application: Time Series

- The span s_i of a stock's price on a certain day i is the maximum number of consecutive days (up to the current day) the price of the stock has been less than or equal to its price on day i



An Inefficient Algorithm

Algorithm computeSpans1(P):
Input: an n-element array P of numbers such that P[i] is the price of the stock on day i
Output: an n-element array S of numbers such that S[i] is the span of the stock on day i

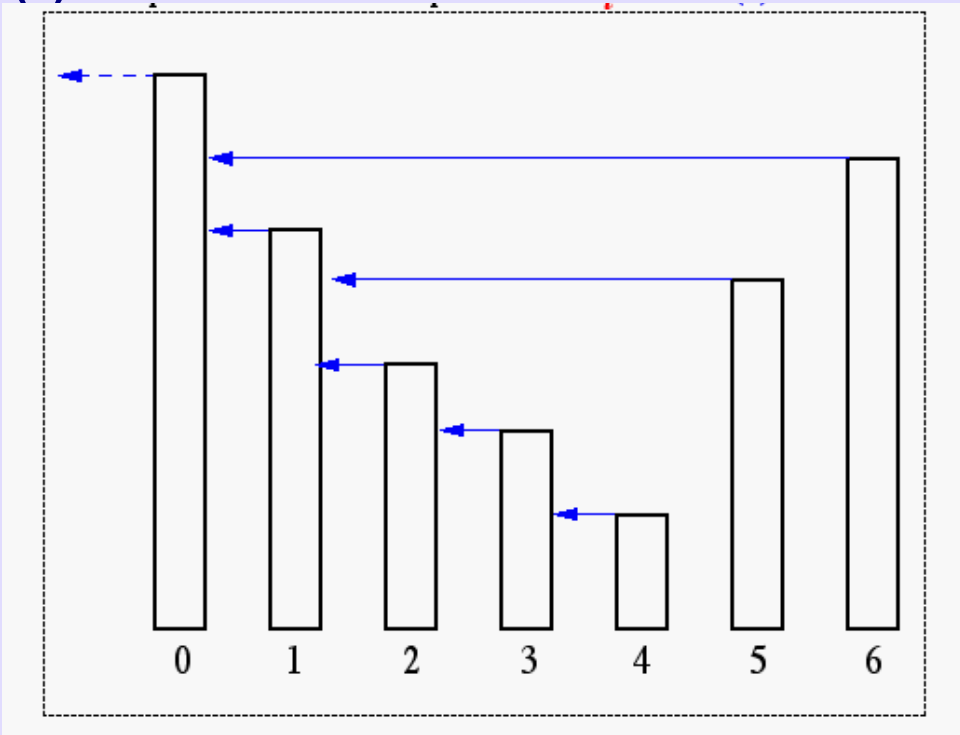
```
for i ← 0 to n - 1 do
    k ← 0; done ← false
    repeat
        if P[i - k] ≤ P[i] then k ← k + 1
                           else done ← true
    until (k = i) or done
    S[i] ← k
return S
```

The running time of this algorithm is $O(n^2)$. Why?

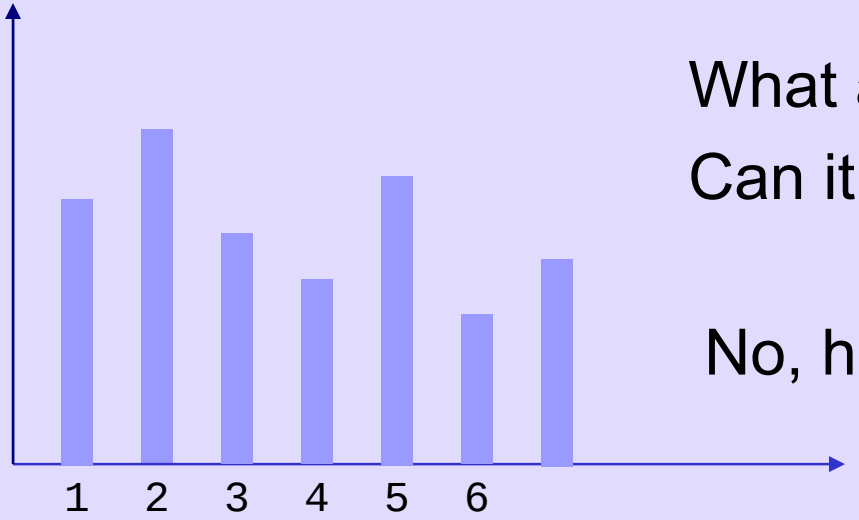
A Stack Can Help

- s_i can be easily computed if we know the closest day preceding i, on which the price is greater than the price on day i. If such a day exists, let's call it h(i), otherwise, we conventionally define h(i) = -1

- In the figure, h(3)=2, h(5)=1 and h(6)=0.
- The span is now computed as $s_i = i - h(i)$



What to do with the Stack?



What are possible values of h(7)?

Can it be 1 or 3 or 4?

No, h(7) can only be 2 or 5 or 6.

- We store indices 2,5,6 in the stack.
- To determine h(7) we compare the price on day 7 with prices on day 6, day 5, day 2 in that order.
- The first price larger than the price on day 7 gives h(7)
- The stack should be updated to reflect the price of day 7
- It should now contains 2,5,7

An Efficient Algorithm

```
Algorithm computeSpans2(P):
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
```

A Growable Array-Based Stack

- Instead of giving up with a StackFullException, we can replace the array S with a larger one and continue processing push operations.

```
Algorithm push(o)
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] ← o
```

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

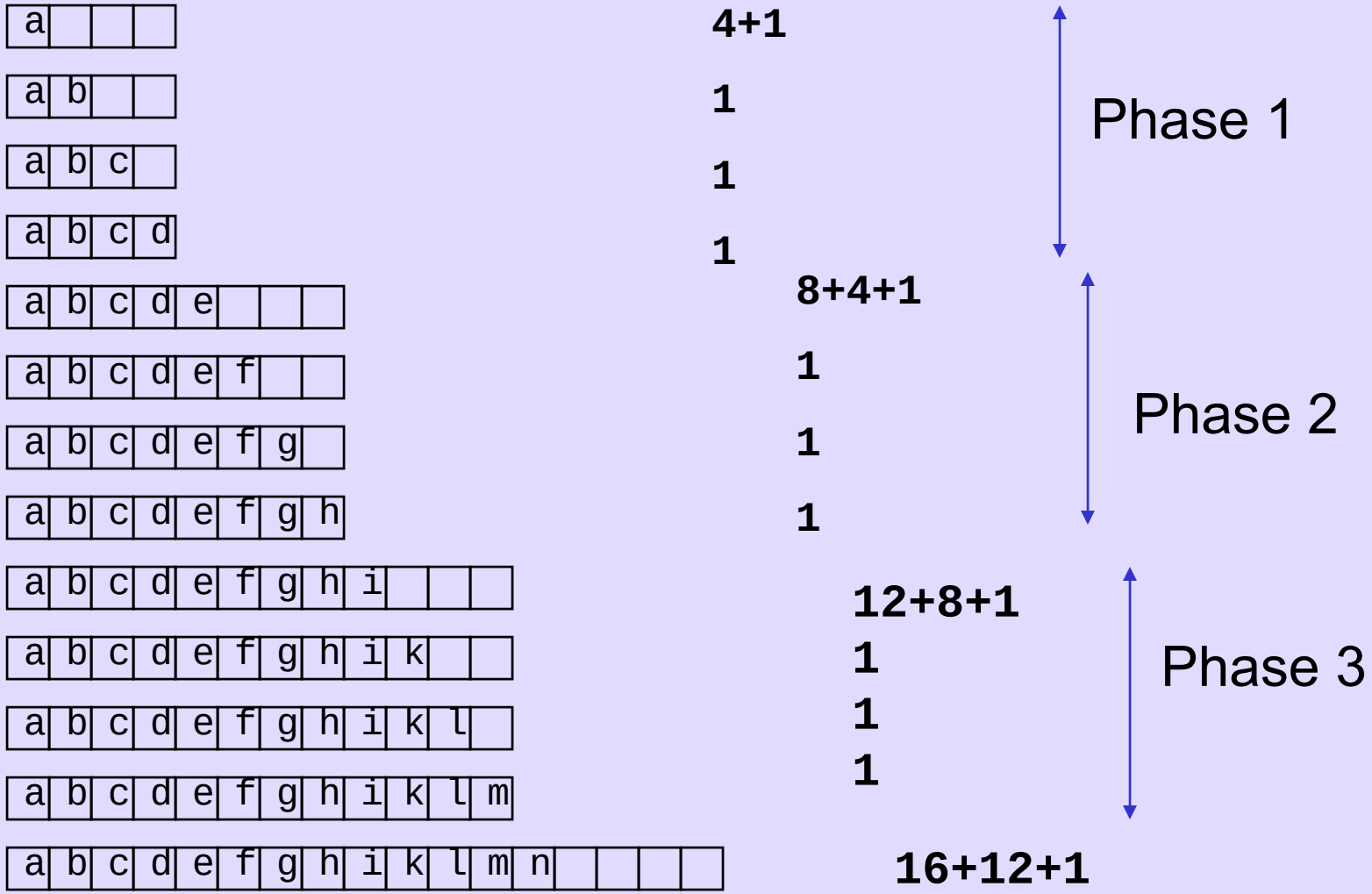
Tight vs. Growth Strategies:
a comparison

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

- A *regular push operation*: adds one element and cost one unit.
- A *special push operation*: create an array of size f(N), copy N elements, and add one element. Costs f(N)+N+1 units

Tight Strategy (c=4)

start with an array of size 0. cost of a special push is 2N + 5



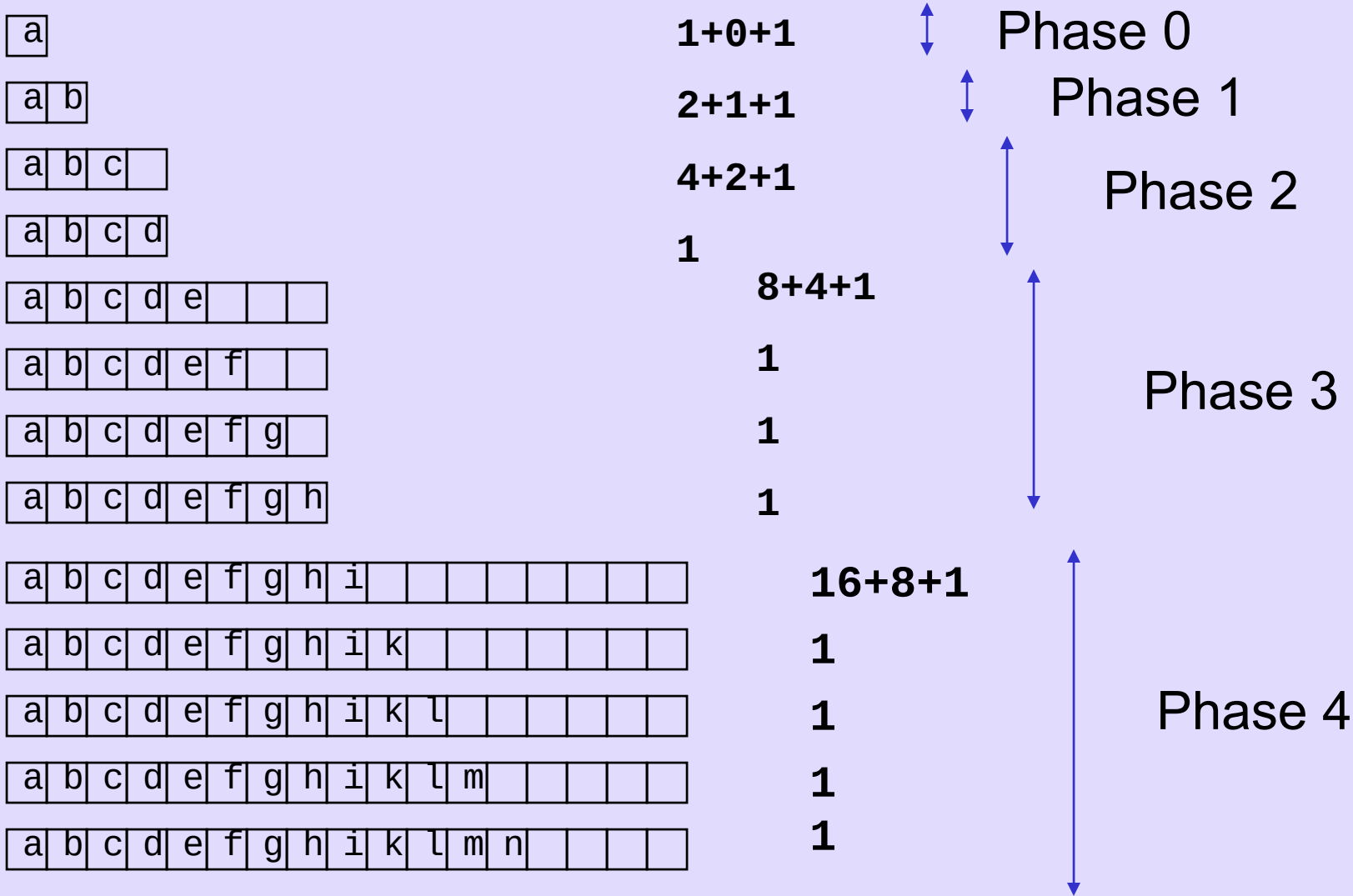
Performance of the Tight Strategy

- In phase i the array has size $c \times i$
- Total cost of phase i is
  - $c \times i$ is the cost of creating the array
  - $c \times (i-1)$ is the cost of copying elements into new array
  - c is the cost of the c pushes.
- Hence, cost of phase i is 2ci
- In each phase we do c pushes. Hence for n pushes we need n/c phases. Total cost of these n/c phases is

  $= 2c\,(1 + 2 + 3 + \ldots + n/c) \approx O(n^2/c)$

Growth Strategy

start with an array of size 0. cost of a special push is 3N + 1



Performance of the Growth Strategy

- In phase i the array has size 2^i
- Total cost of phase i is
  - 2^i is the cost of creating the array
  - 2^{i-1} is the cost of copying elements into new array
  - 2^{i-1} is the cost of the 2^{i-1} pushes done in this phase
- Hence, cost of phase i is 2^{i+1}.
- If we do n pushes, we will have log n phases.
- Total cost of n pushes

$$= 2 + 4 + 8 + \ldots + 2^{\log n+1} = 4n - 1$$

- The growth strategy wins!

Stacks in the *Java Virtual Machine*

- Each process running in a Java program has its own Java Method Stack.
- Each time a method is called, it is pushed onto the stack.
- The choice of a stack for this operation allows Java to do several useful things:
  - Perform recursive method calls
  - Print stack traces to locate an error

Java Method Stack