Data Structures and Algorithms
Lecture 1: Introduction

Dr Alessio Santamaria

UNIVERSITY
OF SUSSEX

2023/2024

Module organisation

This module consists of:

- 2 hour of lectures per week (total: 22 hours)
- 1 hour of lab per week, from Week 2 (total: 10 hours)

This is a 15 credit module: you're expected to work 150 hours on this! So, 150-22-10 = 118 hours of self-study. (Around 9 hours per week)

Lectures = theoretical presentation of the data structures and algorithms we'll study.

Labs = practical implementation in Python of the above, with applications.

Attendance is recorded with PIN in lectures, and manually in labs.

Assessment

100% exam (Multiple Choice Questions) at the end of term. Online.

You will NOT be assessed on Python code. You will have to answer technical questions and solve theoretical exercises.

At the end of every week I'll release a Canvas quiz with exercises on the topics of that week. (This is unmarked.) Exam questions will be similar to these!
You'll be able to see commented solutions immediately after submitting the quiz.

Support

- Lecturer (questions on lectures, assessment, theoretical concepts)
- Teaching assistants/demonstrators (questions on coding and implementations)
- PAL (anything!)

I'm always available via email (A.Santamaria@sussex.ac.uk). Contact me for an appointment.

Materials:

- Lecture slides
- Panopto recordings
- Lab exercises + solutions
- Weekly Canvas quizzes

If you want to read more about the material, I recommend

- Cormen, Leiserson, Rivest, and Stein: *Introduction to Algorithms*, 4th edition (available online at Sussex Library)
- Goodrich, Tamassia, Goldwasser: *Data Structures and Algorithms in Java*, 6th edition (available physically at Sussex Library)

Prerequisets and looking at the future

This modules **heavily** relies on the following ones:

1. Programming Concepts
2. Mathematical Concepts

Go revise them!

The contents of this module will be used in **all** future modules, particularly in:

1. any module involving programming (including Machine Learning, NLP...)
2. Compilers and Computer Architecture (abstract syntax **trees**, **stack** machines)
3. Program Analysis (algorithm complexity, algorithm design)
4. Computer Networks (graphs)
5. Software Engineering (software design, efficiency)
6. Limits of Computations (algorithm complexity, trees, lists)
7. Individual Project

Motivations

Why study Data Structures and Algorithms?

1. **Problem solving**: you'll learn how to solve various computational problems using different DS and As. The first solution might not be the best!
2. **Efficiency**: you'll learn how to solve problems in an efficient way, not just at any cost whatsoever. Important for scalability.
3. **Technical interviews**: often applicants to tech companies get asked to pick or design the most appropriate DS and/or algorithm to use in a given scenario.
4. **Understanding library functions**: what if you need to design a slight variation of a given library function? You need to know what's the logic behind it.
5. **Real-life situations**: sorting, searching, counting, path finding algorithms are used everywhere and all the time.

And, most importantly...

- **Becoming a Computer *Scientist***: we will delve into one of the cores of the Science of Computing, and this is *awesome*.

Learning outcomes

By the end of this module a successful student should be able to:

- Evidence knowledge of a variety of data structures in terms of their characteristic behaviours and efficiency.
- Implement and apply appropriate data structures for solving program design problems.
- Demonstrate basic knowledge of complexity issues with respect to data manipulation.
- Evidence understanding of a number of fundamental algorithms.

Labs information

- Labs are not designed to be done in one hour. Work on lab sheets on your own **before** coming to your assigned lab.
- In-person labs start from Week 2, but there is a Lab 1 "Getting ready" for you to do on your own. Please read through it **regardless of whether you have experience with Python**.
- Lab materials are published on a dedicated public GitHub repository.
  `https://github.com/bertie-wheen/dsa-2023-4`
  I'll also have them available directly on Canvas, but I recommend you use the GitHub version as it's the main one.
- We aim to publish solutions for Week n and lab for Week $n+1$ on Friday evening of Week n (after final lab).

Table of Contents

Goals

- Solve computational problems
- Prove correctness
- Argue efficiency

Algorithms and computational problems

What is an algorithm?

It is a well-defined computational procedure that takes some value, or set of values, as ***input*** and produces some value, or set of values, as ***output*** in a finite amount of time.

Algorithms are used to solve **computational problems**. A problem specifies for each input belonging to a given set one or many desired outputs.

Example (Sorting problem)

Input: a sequence of n numbers $\langle a_1, \ldots, a_n \rangle$
Output: a permutation (reordering) $\langle a'_1, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

An algorithm for a problem is *correct* if for every valid input for the problem

1. it halts
2. it produces an output as requested by the problem.

In this case we say that the algorithm **solves** the problem.

Example: the birthday problem

Input: a sequence of n people $\langle p_1, \ldots, p_n \rangle$
Output: a pair of two people from the input sequence who are born on the same day and month, if it exists; otherwise `None`.

Proposal for a solving algorithm

1. Define a record A of people, initially empty.
2. **for** $i = 1$ **to** n, check if p_i's birthday matches that of someone in A.
   2.1 If it doesn't, then add p_i to A.
   2.2 If it does match with p_j's birthday, then **return** the pair (p_i, p_j) (and **halt**).
3. **return** `None`.

How to prove that this algorithm solves the birthday problem? By induction.

Recall: induction

Theorem (Principle of Induction)

Let $P(n)$ be a property concerning an arbitrary natural number n. (For instance, "Given n people, our algorithm solves the birthday problem".)
If the following two conditions are satisfied:

1. *$P(0)$ is true,*
2. *for any $k \geq 0$, whenever $P(k)$ is true it also happens that $P(k+1)$ is true*

*then $P(n)$ holds for **all** natural numbers n.*

Condition 1 above is often called *base case*, while condition 2 *inductive step*.
In our case, $P(n)$ is:
"Given a sequence of n people $\langle p_1, \ldots, p_n \rangle$, our algorithm returns a pair of two people in the sequence who are born on the same day and month, if it exists, otherwise it returns `None`".

A proof of correctness

Claim: the property $P(n)$ ="Given a sequence of n people $\langle p_1, \ldots, p_n \rangle$, our algorithm returns a pair of two people in the sequence who are born on the same day and month, if it exists, otherwise it returns `None`" is true for all natural numbers n.

Proof.

We use the Principle of Induction Theorem.

Base case: we need to prove that $P(0)$ is true. Given 0 people as input, the correct output is `None`, which our algorithm indeed returns in this case.

Inductive step. Let $k \geq 0$, suppose that $P(k)$ is true. Is $P(k+1)$ true?

Let $S = \langle p_1, \ldots, p_{k+1} \rangle$ and consider the sub-sequence $S' = \langle p_1, \ldots, p_k \rangle$.

- If S' contains a pair of matching people, then so does S and the algorithm returns correctly by inductive hypothesis. Hence $P(k+1)$ holds.
- Otherwise S' doesn't contain a matching pair; so if S contains a match, then person p_{k+1} must be involved, and the algorithm will return a pair (p_j, p_{k+1}) if there is j such that p_j shares a birthday with p_{k+1}, otherwise it returns `None`. Hence $P(k+1)$ holds. □

Efficiency

An algorithm is said to be more **efficient** than another if it uses fewer resources (time, space, energy...). How can we compare algorithms?
We could simply measure the time they take on a stopwatch. But:

- First we need to implement the algorithms
- We should test them on *lots* of inputs (what about those inputs we haven't chosen?)
- We would need to use the exact same hardware and software configurations.

Instead,

- we use a **model of computation** and we count the number of *primitive operations* as a function of the input size (usually, larger input $\implies$ longer time)
- we compare algorithms based on their **asymptotic performance** in $\mathcal{O}(-)$, $\Omega(-)$, $\Theta(-)$ notation, ignoring constant factors that don't change with the size of the problem input.

Asymptotic performance refresher

Upper bound of asymptotic growth: $\mathcal{O}(-)$

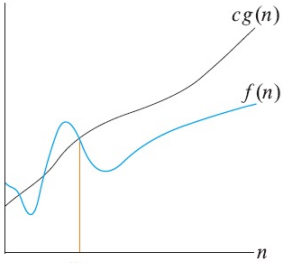
A non-negative function $f(n)$ **is in** $\mathcal{O}(g(n))$ if there exist a real number $c > 0$ and a positive integer n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Lower bound of asymptotic growth: $\Omega(-)$

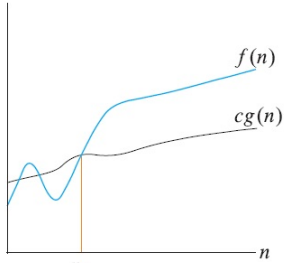
A non-negative function $f(n)$ **is in** $\Omega(g(n))$ if there exist a real number $c > 0$ and a positive integer n_0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

Tight bound of asymptotic growth: $\Theta(-)$

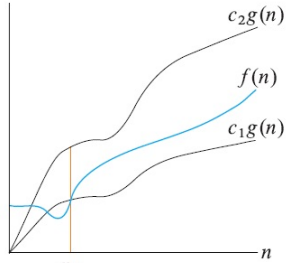
A non-negative function $f(n)$ **is in** $\Theta(g(n))$ if it is in $\mathcal{O}(g(n))$ and in $\Omega(g(n))$.



$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$



$f(n) = \Theta(g(n))$

Common orders of asymptotic complexity

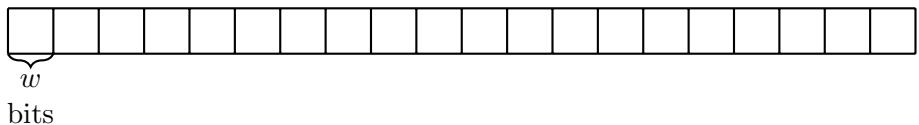
Name	Θ notation	Examples
Constant	$\Theta(1)$	$3 \quad 123,456,789 \quad \sqrt{2} \quad 2024$
Logarithmic	$\Theta(\log n)$	$2024 \log n \quad \log(17n) \quad \log n + 1000$
Linear	$\Theta(n)$	$20n + 5000 \log(3n) - 7$
Linearithmic	$\Theta(n \log n)$	$2n \log(47n) + 1000n - 3$
Quadratic	$\Theta(n^2)$	$3n^2 + 2n \log n - 3000n + 4$
Cubic	$\Theta(n^3)$	$n^3 + 4n^2 - \log n$
Polynomial	$\Theta(n^k)$ $(k \geq 2)$	$2n^{27} + 7000n^2 - 1$
Exponential	$\Theta(2^n)$	$2^{5n+3} + 4000n^3$

- $\log n$ grows slower than any positive power of n
- $n \log n$ grows faster than n but slower than any greater-than-1 power of n
- n^k grows slower than 2^n for every $k > 0$

The w-bit Word-RAM model of computation

We model a computer as an infinite array of machine words called **memory** together with a **processor** that can operate on the memory.

A **machine word** is a sequence of w bits, representing an integer in $\{0, 2^w - 1\}$.



We assume that:

- the processor can perform basic binary operations on two machine words in $\Theta(1)$ time
- given a word a, the processor can access (read/write) the word whose address in memory is a in $\Theta(1)$ time
- the processor can allocate and initialise n consecutive machine words in $\Theta(n)$ time

In order to process an input occupying n machine words in memory, we need to ensure that $2^w \geq n$, that is $w \geq \log n$.

Data structures

The running time of an algorithm depends on the data structure used to store its input data.

Data structure = way to store an arbitrary amount of data, supporting a set of operations to interact with the data.

Interface = set of operations supported by a data structure.

Different data structures can support the same interface, but with different efficiencies per operation.

Static Arrays (sketch)

A static array is a fixed-length sequence of memory addresses, supporting the following operations:

- BUILD_STATICARRAY(n): allocate a new static array of size n initialised to 0 in $\Theta(n)$ time
- GET_AT(i): return the content of the array at index i in $\Theta(1)$ time
- SET_AT(i, x): write x in the array at index i in $\Theta(1)$ time

A Python *tuple* is like a static array without SET_AT(i, x), whereas a Python *list* implements a **dynamic array**.

Input: a static array P of n people as pairs (NAME, BDAY)
Output: a pair of people's names sharing their bday, if it exists, otherwise `None`

```
1 RECORD ← BUILD_STATICARRAY(n)                                  // O(n)
2 for k ← 0 to n − 1 do                                          // O(1) (n times)
3 |   (NAME1, BDAY1) ← P[k]                                      // O(1) (n times)
4 |   for i ← 0 to k do                                          // O(1) (k + 1 times)
5 |   |   (NAME2, BDAY2) ← RECORD.GET_AT(i)                     // O(1) (k + 1 times)
6 |   |   if BDAY1 = BDAY2 then                                 // O(1) (k + 1 times)
7 |   |   |   return (NAME1, NAME2)                             // O(1)
8 |   |   RECORD.SET_AT(k, (NAME1, BDAY1))                      // O(1) (k + 1 times)
9 return None                                                    // O(1)
```

$$\begin{aligned}&\mathcal{O}(n) + n \cdot (2 \cdot \mathcal{O}(1)) + \sum_{k=1}^{n} \big(k \cdot 4 \cdot \mathcal{O}(1)\big) + \mathcal{O}(1) \\ =&\mathcal{O}(n^2)\end{aligned}$$

We can do better using a different data structure for RECORD.

Table of Contents

Interfaces (API, ADT) vs Data Structures

Interface	Data Structure
specification	representation
what data to store	*how* to store data
which operations are supported	algorithms implementing the operations
problem	solution

Two interfaces: sequences and maps

The idea is that we want to store n items.

- *Sequences* maintain the items in an extrinsic (=specified by an external agent) order. Each item has a **rank** in the sequence: first item, second item, ..., last item.
- *Maps* maintain the items using an intrinsic property that depends on the items themselves, normally based on a unique **key** associated with each item.

We have two data structure approaches to solve these interfaces:

1. arrays
2. pointer-based (linked)

The Static Sequence interface

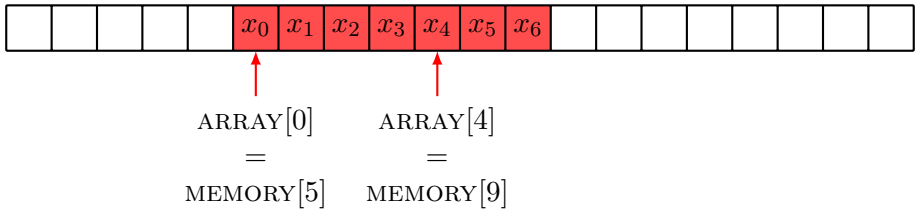
Problem: maintain a sequence of n items $x_0, x_1, \ldots, x_{n-1}$ supporting the following operations.

- BUILD(X): given a collection (an iterable) of n items X, make a sequence of the items in X in their specified order
- LEN(): return n (the number of items in the sequence)
- ITER_SEQ(): output $x_0, x_1, \ldots, x_{n-1}$ in sequence order
- GET_AT(i): return x_i
- SET_AT(i, x): set x_i to be x

A natural solution for the static sequence interface problem is a **static array**.

Static Arrays

Static array *of size* n = consecutive chunk of memory in our RAM model (of exactly n machine words). Hence $\textsc{array}[i] = \textsc{memory}[\textsc{address}(\textsc{array}) + i]$.



Items $x_0, \ldots, x_{n-1}$ are stored consecutively in n machine words so that $\textsc{size}(\textsc{array}) = n$ and $\textsc{array}[i] = x_i$.

Thanks to our model, static arrays ensure that:

1. $\textsc{get\_at}(i)$, $\textsc{set\_at}(i, x)$, $\textsc{len}()$ run in $\mathcal{O}(1)$ time
2. $\textsc{build}(X)$, $\textsc{iter\_seq}()$ run in $\Theta(n)$ time.

The Dynamic Sequence interface

Same operations of static sequence interface, with the addition of:

- INSERT_AT(i, x): make x the new x_i, shifting the original x_i, $x_{i+1}, \ldots, x_{n-1}$ by one on the right
- REMOVE_AT(i): remove x_i and shift $x_{i+1}, \ldots, x_{n-1}$ by one on the left

For instance, INSERT_AT$(2, x)$ would change this:

into this:

where $x'_0 = x_0$, $x'_1 = x_1$, $x'_2 = x$, $x'_3 = x_2$, $\ldots$, $x'_n = x_{n-1}$. If we now call GET_AT(3), we get whatever was x_2 originally.

The Dynamic Sequence interface (continued)

We will also define the following special-case operations:

- INSERT_FIRST(x) and INSERT_LAST(x)
- REMOVE_FIRST() and REMOVE_LAST()

which may be implemented more efficiently if done directly rather than by just using INSERT_AT($0, x$) or INSERT_AT($n-1, x$) etc.
We present three solutions to the Dynamic Sequence interface: static arrays (again!), **linked lists**, and **dynamic arrays**.

Static arrays as dynamic sequences

How can we implement INSERT_AT(i, x) with a static array?

There's only one way...

1. allocate a new array of size $n + 1$
2. copy the original $x_0, \ldots, x_{i-1}$ over
3. write x in place
4. copy the original $x_i, \ldots, x_{n-1}$ over

because static arrays are "static" (the number of machine words of memory to be allocated for the array must be decided upfront)! This costs $\Theta(n)$ time.

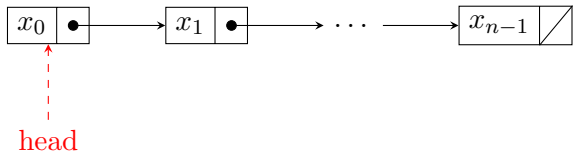
Similarly, the various REMOVE operations cost $\Theta(n)$ time because of new allocation/copying over elements.

So, not great...

Linked Lists

Items are stored in *nodes*: | item | next | where "next" is a **pointer** to (that is, the memory address of) another node. Nodes are simply arrays of size 2.

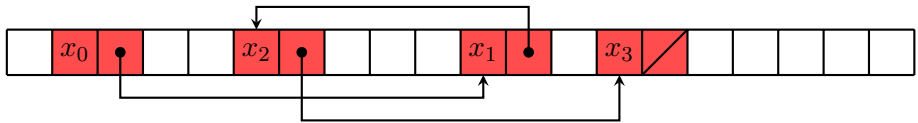
Conceptually, a linked list looks like this:



A Linked List data structure consists of just two data:

1. the first node, containing x_0, called the **head**,
2. the number of items in the list, n.

- Each node occupies 2 machine words of memory
- Consecutive nodes in a linked list don't have to be stored consecutively in memory

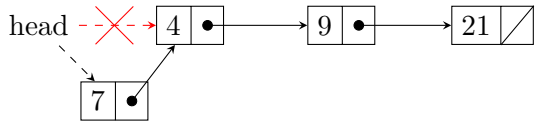


- Getting the next node of a node in a linked list takes $\mathcal{O}(1)$ time
- Inserting or deleting is done by manipulating pointers (no shifting/copying over!)

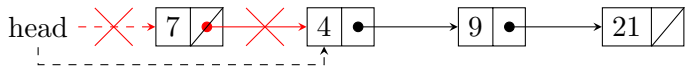
Inserting/deleting at the front of a linked list

It's quick and easy to insert or remove at the front of a linked list.

INSERT_FIRST(7) :

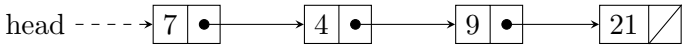


REMOVE_FIRST() :



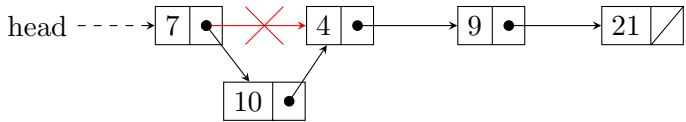
Both cost $\Theta(1)$ time.

Inserting and deleting inside a linked list

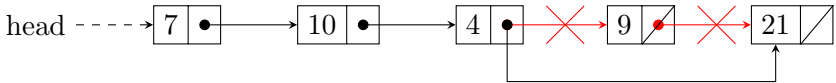


However, inserting and deleting in the middle is more expensive.

INSERT_AT$(1, 10)$:



REMOVE_AT(3) :



INSERT_AT(i, x) and REMOVE_AT(i) cost $\Theta(i)$ because of "pointer hopping".
INSERT_LAST(x) and REMOVE_LAST$()$ cost $\Theta(n)$ time. (Can we improve that?)
Static sequence operations also cost $\Theta(n)$ time in worst case.

Static arrays vs linked lists

Sequence Data Structure	Operations $\mathcal{O}(\cdot)$				
	Container	Static	Dynamic		
	BUILD(X) ITER_SEQ()	GET_AT(i) SET_AT(i, x)	INSERT_FIRST(x) REMOVE_FIRST()	INSERT_LAST(x) REMOVE_LAST()	INSERT_AT(i, x) REMOVE_AT(i)
Static array	n	1	n	n	n
Linked list	n	n	1	n	n

We can improve INSERT_LAST() of a linked list by storing a pointer to the last node and changing accordingly all the other methods (**data structure augmentation**). With this, INSERT_LAST(x) costs $\Theta(1)$ time.

We can augment further to a **doubly linked list** (where each node has a pointer to the previous node too). Now REMOVE_LAST() costs $\Theta(1)$ time.

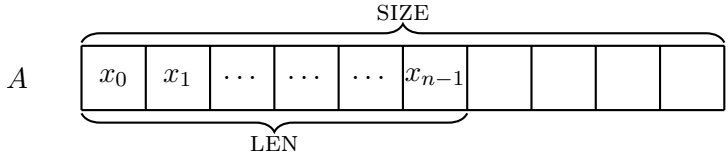
Dynamic Arrays

Remember the problem: we need to store items $x_0, \ldots, x_{n-1}$ and to support the Dyamic Sequence interface.

Dynamic array = like an array but without the constraint $\text{SIZE}(\text{ARRAY}) = n$. We only require that $\text{SIZE}(\text{ARRAY}) = \Theta(n)$ (and of course $\text{SIZE}(\text{ARRAY}) \geq n$).
We do maintain the other constraint that $\text{ARRAY}[i] = x_i$.

A dynamic array data structure consists of three data:

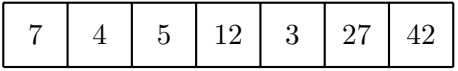
1. an integer, LEN, counting the number of items currently stored
2. an integer, SIZE, such that $\text{LEN} \leq \text{SIZE}$ and $\text{SIZE} = \Theta(n)$,
3. a static array A of size SIZE.



Dynamic arrays: insertion

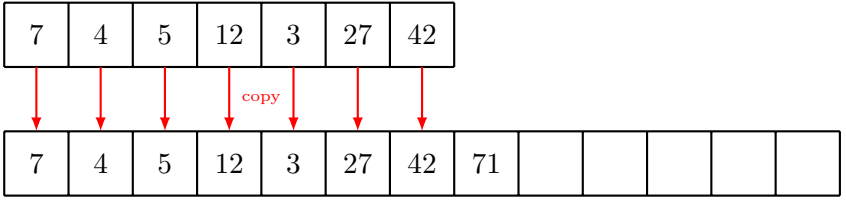
If LEN $<$ SIZE, then INSERT_LAST(x) costs $\Theta(1)$ time.

INSERT_LAST(42) :



If we want to insert a new item and LEN $=$ SIZE, then we first allocate a new, larger static array and copy the elements over. This costs $\Theta(n)$ time.

INSERT_LAST(71) :



Resizing strategy

Increasing the size of the array by a constant value won't help us.
Instead, when calling for INSERT_LAST with LEN = SIZE, we will allocate a new array of size $2 \cdot$ SIZE.

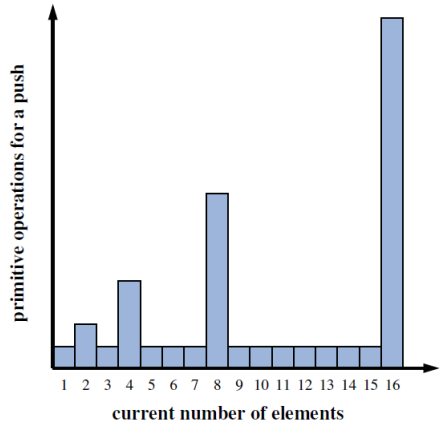
Suppose we start from an empty array and we make n consecutive calls of INSERT_LAST. Then we need to resize the array at insertions 1, 2, 4, 8, 16, ..., k (where k is the highest power of 2 before n, which is $2^{\lfloor \log n \rfloor}$).
Hence, the cost of n consecutive insertions from an empty array is:

$$\Theta\left(\sum_{i=0}^{\log n} 2^i\right) = \Theta\left(2^{(\log n)+1} - 1\right) = \Theta\left(2^{\log n}\right) = \Theta(n).$$

This means *roughly* $\Theta(1)$ cost per insertion!

Amortized analysis

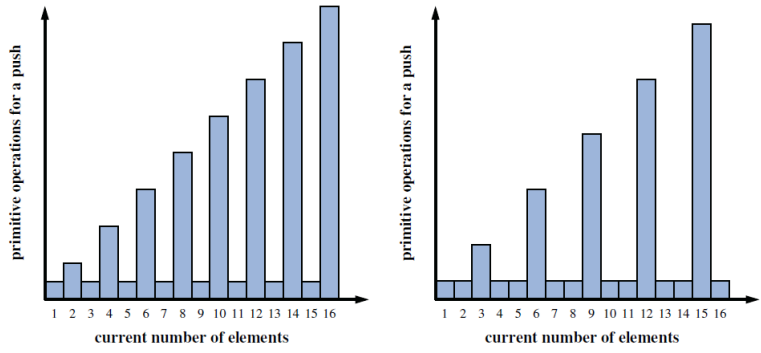
If the new array size is c times the original size, for any $c > 1$, then we can spread (**amortize**) the linear cost of a new allocation + copying over the previous constant-time insertions.



So *on average* we get that each INSERT_LAST(x) operation costs $\mathcal{O}(1)$ time.

Amortized analysis

However, if we just increase the size by a constant amount, this doesn't work!



(a): increasing the size by 2 when array is full
(b): increasing the size by 3 when array is full

Pictures in this and the preceding slide are from Goodrich et al, *Data Structures and Algorithms in Java*, 6th Edition

Shrinking to save space

REMOVE_LAST() costs $\Theta(1)$ time: no re-indexing or shifting needed.
However, this could lead us to waste a lot of memory after many deletions.

Suggestions for a good shrinking strategy?

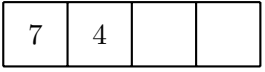
- Proposal: if LEN $<$ SIZE$/2$, halve the array.

REMOVE_LAST() :

Now the worst that can happen is to keep inserting and deleting, and each operation costs $\Theta(n)$ time.

- Proposal 2: if LEN $<$ SIZE$/4$, halve the array.

REMOVE_LAST() :



We get $\Theta(1)$ amortized time again and we ensure LEN $\leq$ SIZE $\leq 4 \cdot$ LEN always.

Data structures for dynamic sequences

Sequence Data Structure	Operations $\mathcal{O}(\cdot)$				
	Container	Static	Dynamic		
	BUILD(X) ITER_SEQ()	GET_AT(i) SET_AT(i, x)	INSERT_FIRST(x) REMOVE_FIRST()	INSERT_LAST(x) REMOVE_LAST()	INSERT_AT(i, x) REMOVE_AT(i)
Static array	n	1	n	n	n
Linked list	n	n	1	n	n
Dynamic array	n	1	n	$1_{(a)}$	n

Notice that INSERT_LAST(x) and REMOVE_LAST() still take $\Theta(n)$ time in worst case scenario in dynamic arrays (with shrinking). But mostly often they take $\Theta(1)$ time.

Exercise: Can you design a data structure that can achieve $\Theta(1)$ worst case time for static operations and $\Theta(1)$ amortized time for INSERT/REMOVE_FIRST?