Lecture 7 - Dynamic Arrays

Computer systems, data structures, and data management (4CM508)

Dr Sam O'Neill

Review

A **data structure** is a way to store data with a collection of supported operations that allow us to manipulate that data.

The collection of supported operations is called an **interface** (Abstract Data Type).

The difference between a data structure and it's interface can be thought of as follows:

- An **interface** is a specification (what the data structure does)
- A **data structure** is a concrete implementation (how its done!).

Sequence Interface

- Maintains a sequence of n items, e.g. `34,25,35,54` or `"sam", "joe", 1`
- Supports sequence operations

e.g. Python List

Sequence Operations

Note that this is not written as Python, we are just describing the type of things (operations) we should be able to do to the sequence.

Name	Description
`create(X)`	create sequence from items in `X`
`size()`	return the length of the sequence
`get(i)`	return the item at index `i`
`set(i,x)`	replace the item at index `i` with `x`
`insert(i,x)`	add `x` to position `i` (this will move all previous items at index `i`, `i+1`,... etc up 1)
`delete(i)`	delete the item at index `i` (this will move all previous items at index `i`, `i+1`,... etc down 1)

Example: Python List

```
demo_list = [2,5,1,66,3,4,23,42]

len(demo_list)          # size() - return the length of the sequence

demo_list[2]            # get(2) - return the item at index 2

demo_list[2] = 34       # set(2,34) - replace the item at index 2 with 34

demo_list.insert(2,999) # insert(2,999) - add 999 to index 2

demo_list.pop(4)        # delete(4) - delete the item at index 4
```

Additional Sequence Operations

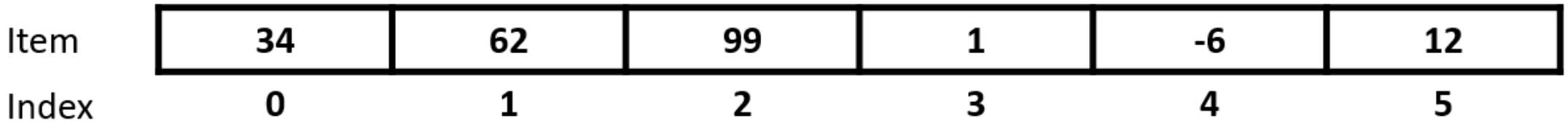
We will also consider the following operations.

Name	Description
`insert_first(x)`	add `x` as the first item. Same as `insert(0,x)`
`delete_first()`	delete the first item. Same as `delete(0)`
`insert_last(x)`	add `x` as the last item. Same as `insert(size(),x)`
`delete_last()`	delete the last item. Same as `delete(size()-1)`

Example: Python List

```
demo_list = [2,5,1,66,3,4,23,42]

demo_list.insert(0,99)    # insert_first(99) - add 99 as the first item

demo_list.pop(0)          # delete_first() - delete the first item

demo_list.append(999)   # insert_last(999) add 999 as the last item

demo_list.pop()           # delete_last() - delete the last item
```

What is an Array?



- A fixed block of contiguous memory
- Stores a fixed number of items with fixed size (number of bytes)
- Items are the same type, e.g. integer

NOTE: Python does not have an array implementation like most common programming languages. It does have an array class, but it doesn't behave completely as expected.

Issues with Arrays

Data Structure	`create(X)`	`get(i)`	`set(i,x)`	`insert(i,x)`	`delete(i)`
Array	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$

Worst-case Complexity

- Dynamic operations such as `insert()` and `delete()` are costly.

Can we do better? The answer is yes and we will see how in this lecture.

Dynamic Arrays

The key idea of dynamic arrays is a very simple one.

- Fix the array size U and track the number of items n
- Resize the array when necessary.

We then have rules for:

- `insert()`
- `delete()`

Insert Rule

When inserting an item we increase the number of items from n to $n+1$, we have two cases:

1. Array has space ($n+1 \leq U$)
    - perform the `insert()`
2. Array is full ($n+1 > U$)
    - **resize (grow)** the array $U = U'$, and then perform `insert()`

Delete Rule

When deleting an item we decrease the number of items from n to $n-1$, we have two cases:

1. Number of items is greater than shrink threshold L, (i.e. $n-1 > L$)
    - perform `delete()`
2. Number of items is less than or equal to threshold L, (i.e. $n-1 \leq L$)
    - **resize (shrink)** the array and then perform `delete()`

Insert - Dynamic Array is NOT full

See slide animation.

Insert - Dynamic Array is full

See slide animation.

Delete - Dynamic Array Above Shrink Threshold

See slide animation.

Delete - Dynamic Array NOT Above Shrink Threshold

See slide animation.

Key Questions

- **What size should we start with?**
  - We will start with a list of size 1.
- **How much do we grow the array by (growth factor)?**
  - We will double the list when it is full. Growth factor of 2.
- **When do we resize for deletion. i.e. lower threshold L**
  - We will shrink the list when we the number of items is $\frac{1}{3}$ of it's current capacity.
- **How much should we shrink the array by?**
  - We will halve the list.

Whilst these might not be exact, the idea of doubling and halving the list and using a threshold of a third are roughly how you would do this.

Summary of Worst-Case Complexity

Data Structure	`create(X)`	`get(i)`	`set(i,x)`	`insert(i,x)`	`delete(i)`
Array	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Dynamic Array	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$

Worst-case Complexity

In the worst case, you will have to do a copy for both `insert()` and `delete`. This scales with n.

Erm... What's the point!?

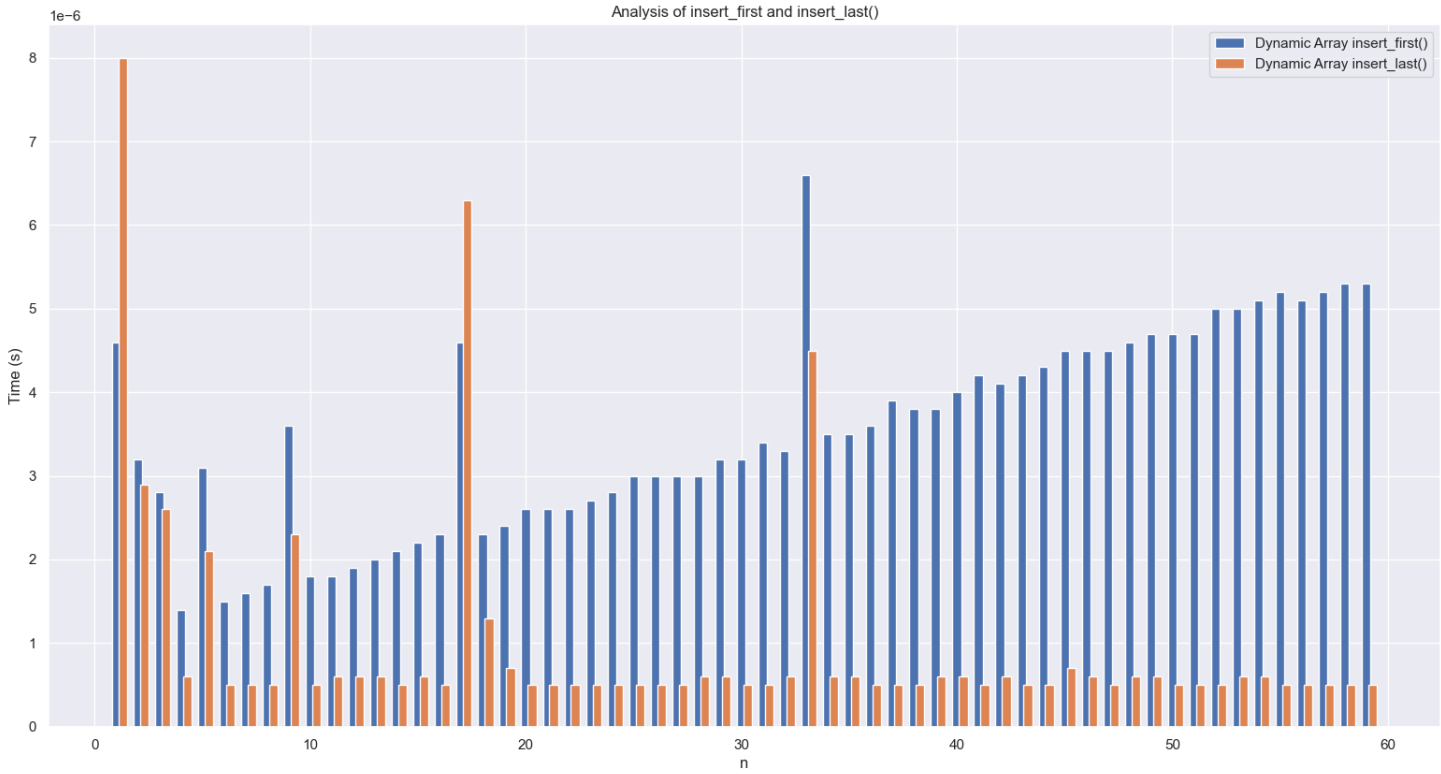
Analysis of `insert_last()` and `insert_first()`

If we look at what happens when we:

1. Keep adding to the front of the dynamic array - `insert_first()`
2. Keep adding to the end of the dynamic array - `insert_last()`

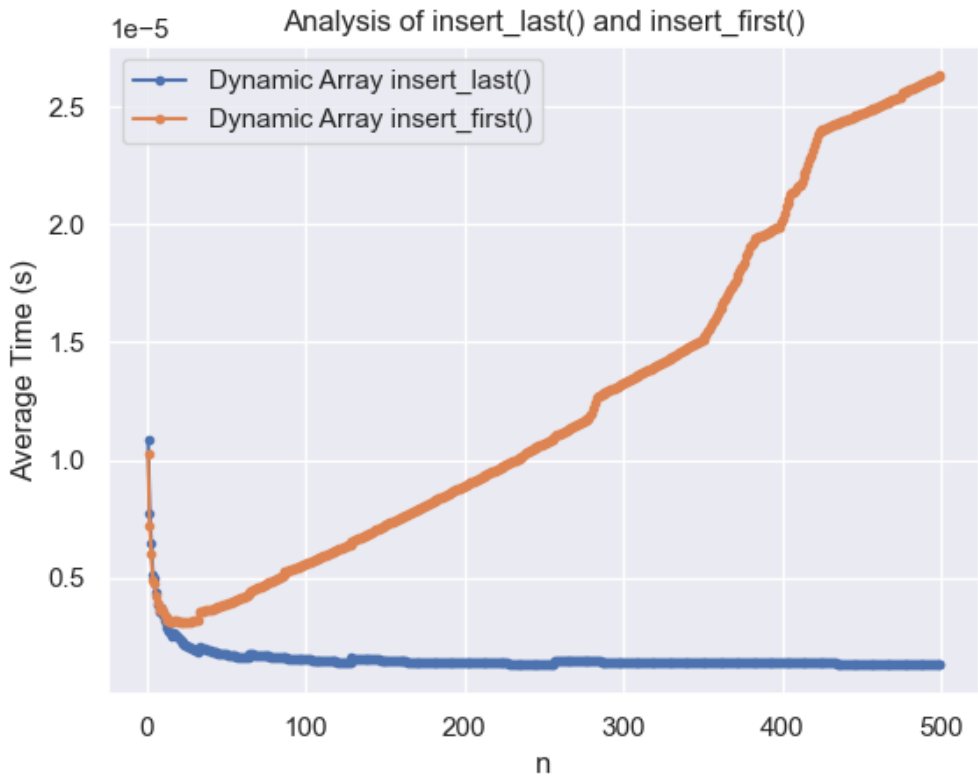
We will see something interesting emerge.

Analysis of `insert_last()` and `insert_first()`



We can see that `insert_last()` seems to at times be very cheap.

If we plot the average.

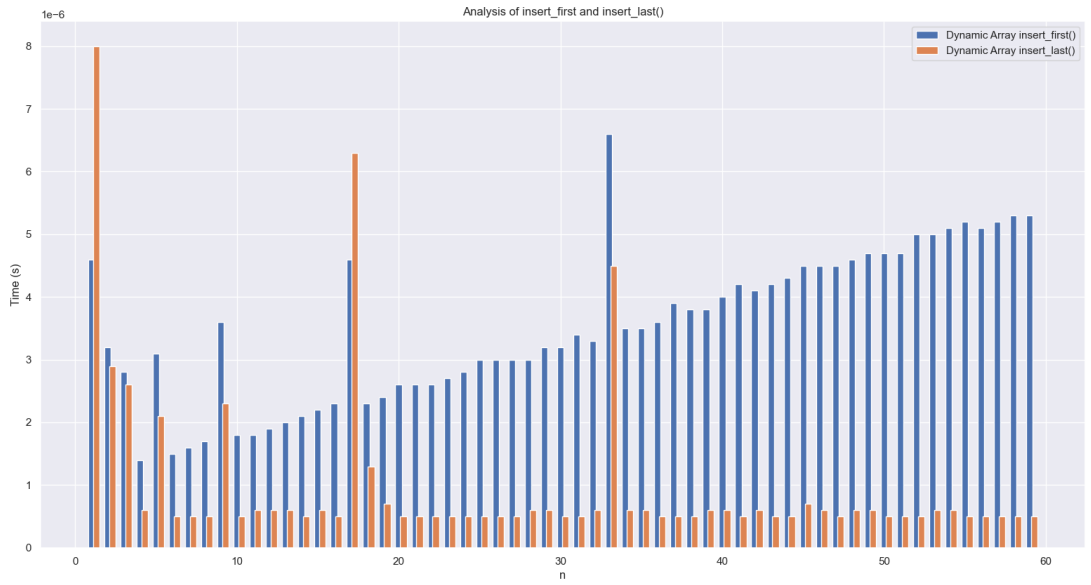


We can see that `insert_last()` doesn't appear to grow like `insert_first()`.

What is going on?

Amortized Analysis of `insert_last()`

- We know that `insert_first()` is $O(n)$ as it must move everything by 1 index to make room at the front.
- What about `insert_last()`.
    - Well if you look at the plot you see a lot of cheap operations.



Key Insight

- Most of the time it is very cheap to add to the end of the list. In fact $O(1)$.
- Occasionally, we have to double the size of the dynamic array which is costly.
- On average doing many `insert_last()` operations will work out to be $O(1)$.
- That means that it does not depend on the size of your list.

Amortized Analysis of `insert_last()`

Amortized analysis:

- used to analyse a single opertation, e.g. `insert_last()`
- Analysed over a sequence of operations.
    - e.g. we keep inserting at the end of the list and look at the average performance of all the operations.

This means we consider all cheap operations and the occasional expensive operations together.

Aggregate Method

We will perform this analysis using the aggregate method.

- Each basic operation costs $£1$
- Charge each `insert_last()`, e.g. $£3$.

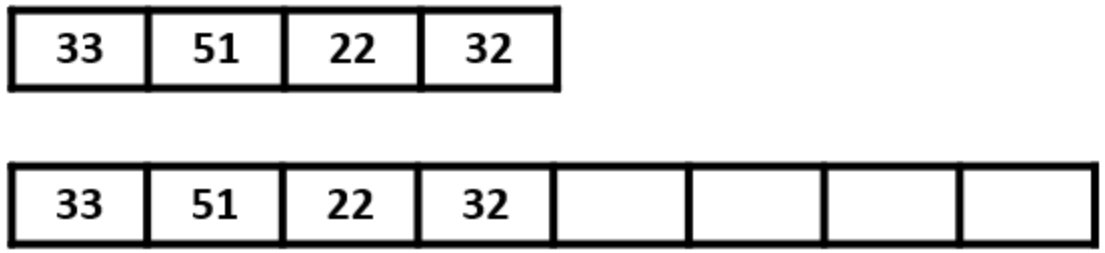
Can we keep enough money in the bank?

We will see that if we charge $£3$ we can!

Important Assumption

We will assume that the cost of growing an array from n to $2n$ is n.

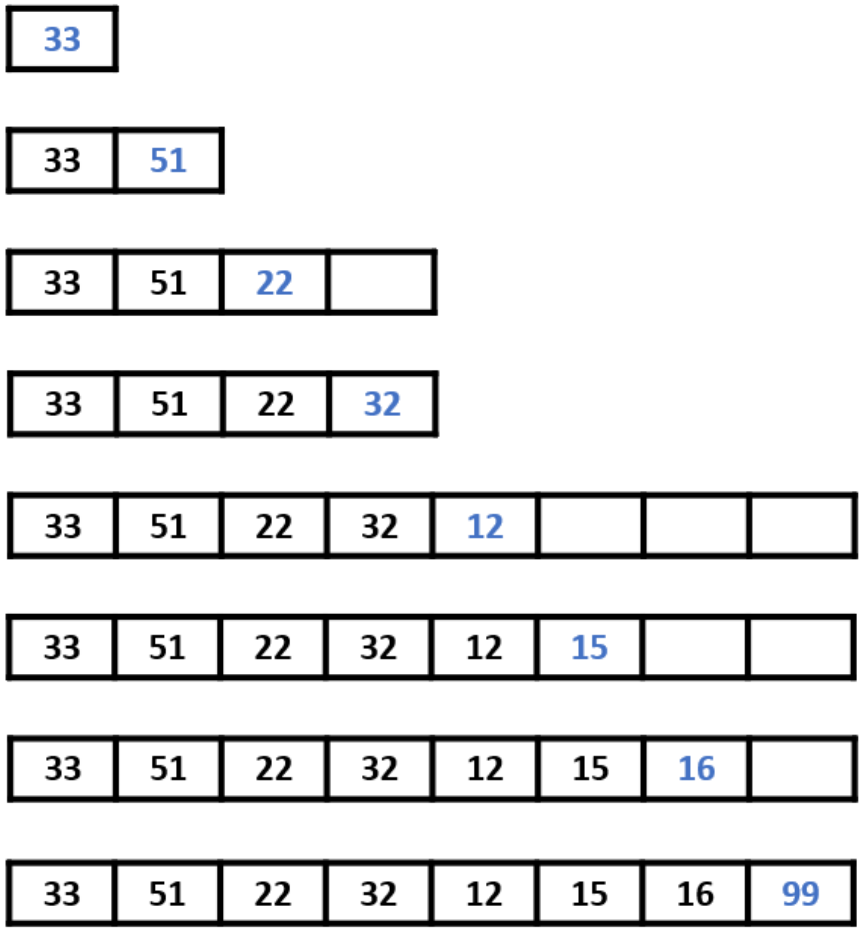
e.g. growing from 4 to 8 will cost 4.



This is a reasonable assumption as different architectures, compilers will allocated memory different.

We will always have to copy over the items!

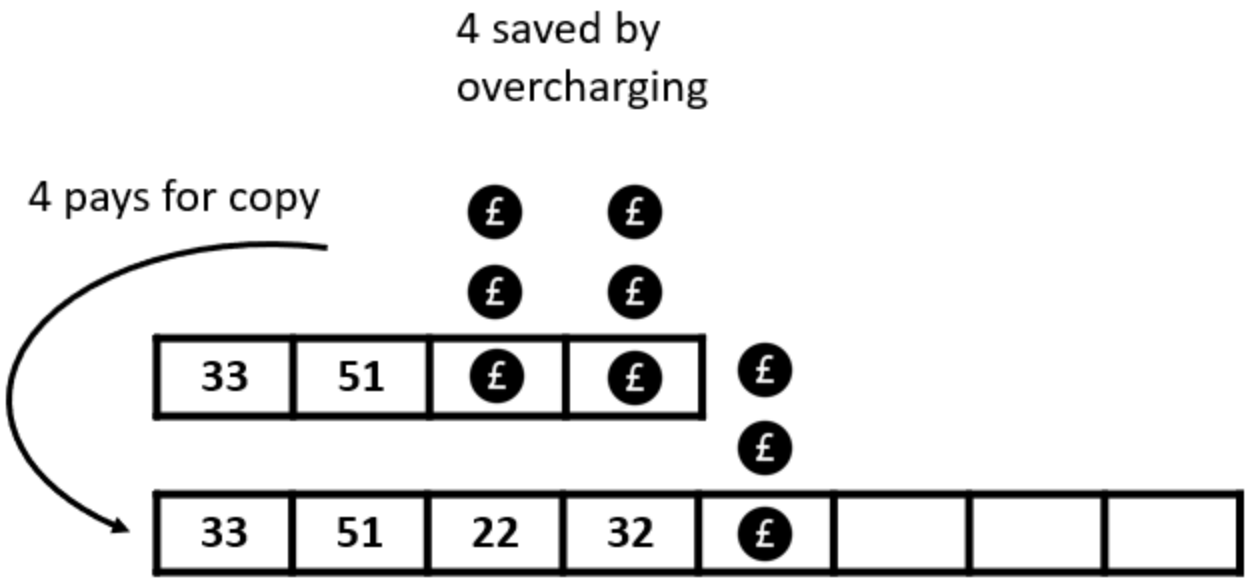
Illustration of Aggregate Method



Charge	Insert Cost	Resize Cost	Money In Bank
3	1	0	2
3	1	1	3
3	1	2	3
3	1	0	5
3	1	4	3
3	1	0	5
3	1	0	7
3	1	0	9

Another Way of Thinking About it

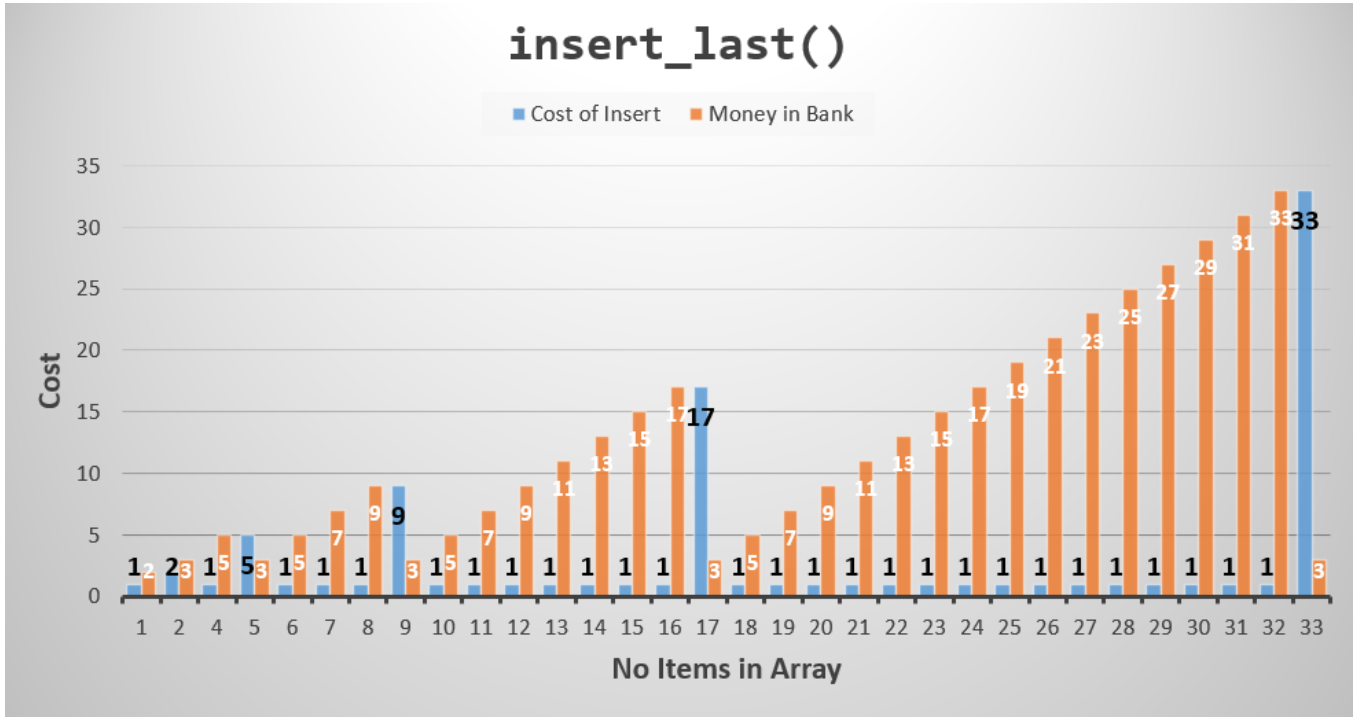
For each cheap `insert_last()` we save $£2$. These add up and pay for the expensive copy.



Hence we **always** pay $£3$ or 3 operations. This is constant time $O(1)$.

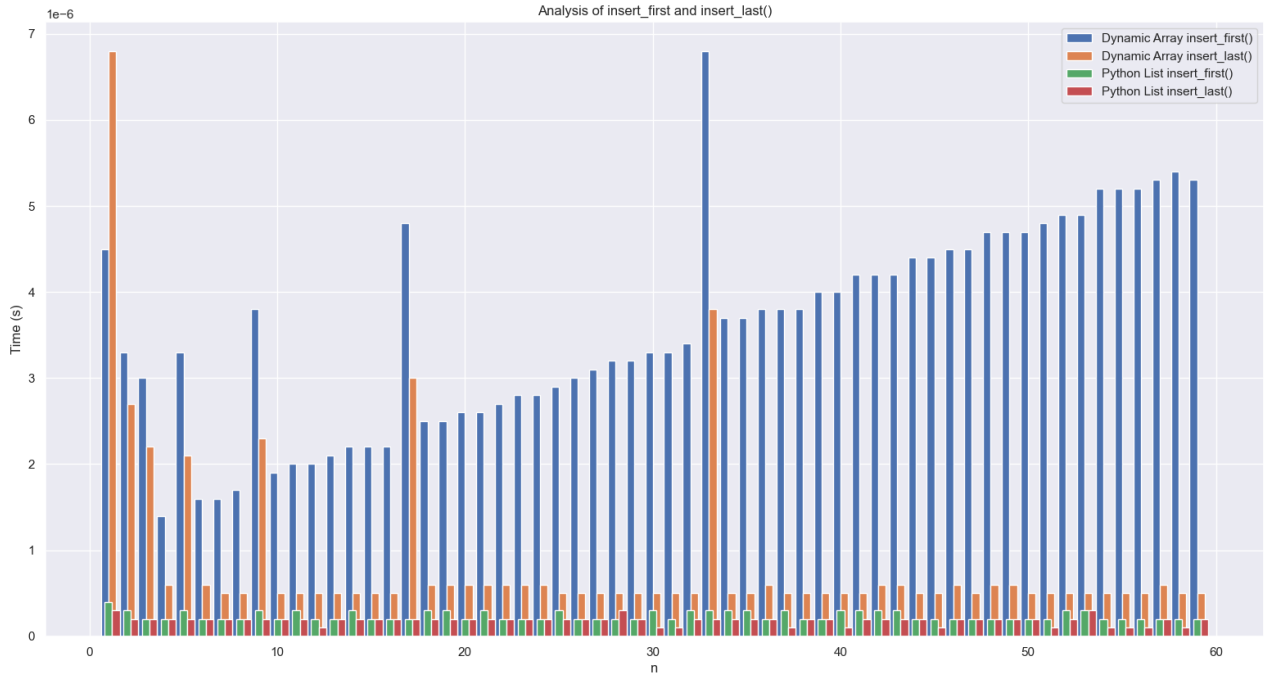
Plot of Costs

Think about charging $£3$ for each `insert_last()`. You can see that the cheap ones pay for the next expensive one.



Python Lists

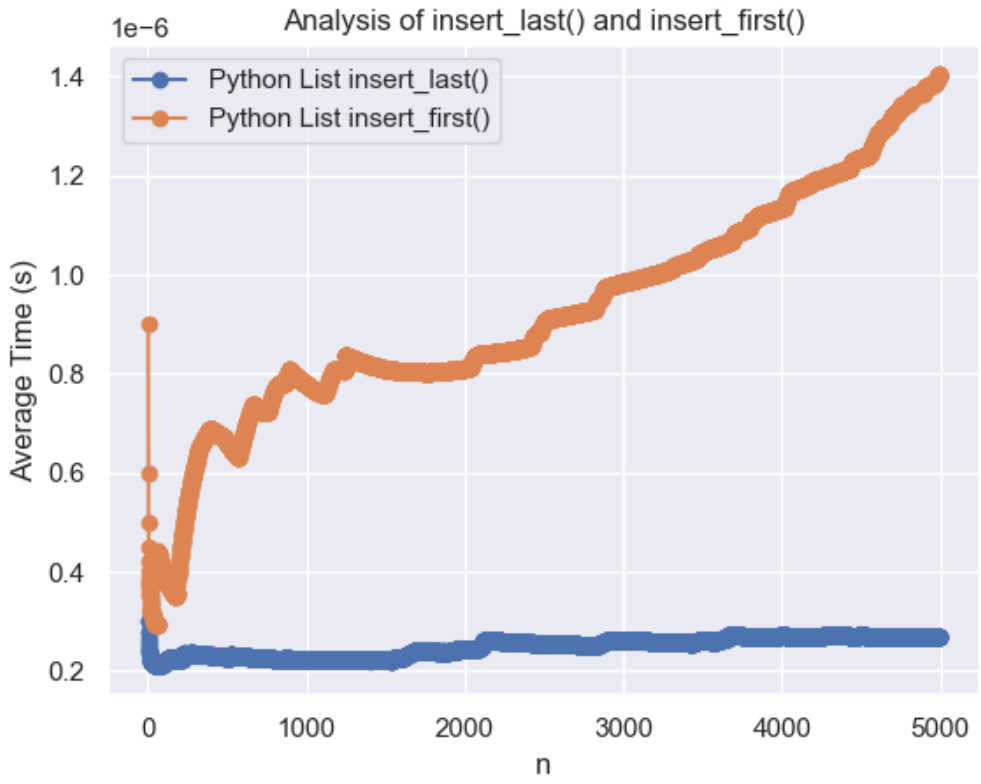
These are highly optimised dynamic arrays. We can plot against our dynamic array.



It looks like `insert_first()` and `insert_last()` are the same for a Python `list`.

Average Time - Python Lists

A more detailed analysis. If we analyse the average over many `insert_first()` and `insert_last()` operations.



Aha! This agrees with our analysis.

Summary

- Dynamic Arrays resize as necessary
- It is cheap to insert and delete at the end of a Dynamic Array
- Python lists are highly optimised Dynamic Arrays

Data Structure	`create(X)`	`get(i)` `set(i,x)`	`insert(i,x)` `delete(i)`	`insert_first(i,x)` `delete_first()`	`insert_last(i,x)` `delete_last()`
Array	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Dynamic Array	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$**

Worst-case Complexity, ****Amoritized Time**

Key Takeaway

If you are doing a lot of appends use a Dynamic Array.

Practical Challenge

There is lots to do on the practical. The additional stuff is:

1. Implement a Dynamic Array in C++ (I have given you a template on replit).
    - It should implement all the sequence operations.

I will post my Python solution on Course Resources.