Week 1

Computational problem: defines a computational task.
Algorithm: sequence of steps, a solution expressed in terms of steps. Input -> output

- something described in logical way
  Correctness and complexity analysis: formal proof that the algorithm solves the problem, bound by the resources it uses. An analysis of how good the algorithm is
- what's the worst case of the running time
  Pseudocode: control flow, method call, return value

$-\infty$; can use to keep track of largest element
max <-- $-\infty$ is assigning $-\infty$ to max.

- Can use proof by induction to prove the correctness of an algorithm

Naive algorithm:

- Iterate over every pair 0 ≤ i ≤ j < n.
- For each compute A[i] + A[i + 1] + · · · + A[j]
- Return the pair with the maximum value

Naive algorithm with preprocessing: pre-compute sums in new array (takes up more memory)

Efficiency: an algorithm is efficient if it runs in polynomial time: on an instance of size n, it performs no more than p(n) steps for some polynomial p(x).

Asymptotic growth analysis: Big-O notation, a way to summarise how T(n) behaves when n increases. We focus on the the terms that make up T(n) and dominate the running time.

1. $T(n) = O(f(n))$ if there exists $n_0, c > 0$ such that $T(n) <= cf(n)$ for all $n > n_0$.

- smaller than f(n)

2. $T(n) = \Omega(f(n))$ if there exists $n_0, c > 0$ such that $T(n) >= cf(n)$ for all $n > n_0$.

- bigger than f(n)

3. $T(n) = \Theta(f(n))$ if there exists 1 and 2.

- equal to f(n)

Examples of asymptotic growth:

- Polynomial: $O(n^c)$, considered efficient since most algorithms have small c
- Logarithmic: $O(logn)$, typical for search algorithms like Binary Search
- Exponential: $O(2^n)$, typical for brute force algorithms exploring all possible combinations of elements
  Comparison of running times:

size	n	$n \log n$	n^2	n^3	2^n	$n!$
10	< 1s	< 1s	< 1s	<1s	<1s	3s
50	< 1s	< 1s	< 1s	<1s	17m	-
100	< 1s	< 1s	< 1s	1s	35y	-
1,000	< 1s	< 1s	1s	15m	-	-
10,000	< 1s	< 1s	2s	11d	-	-
100,000	< 1s	1s	2h	31y	-	-
1,000,000	1s	10s	4d	-	-	-

Properties of asymptotic growth
Transitivity:

- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$

Sums of functions:

- If $f = O(g)$ and $g = O(h)$ then $f + g = O(h)$
- If $f = \Omega(h)$ then $f + g = \Omega(h)$

Asymptotic analysis is a powerful tool that allows us to ignore unimportant details and focus on what's important.

- Gives us some info about the worst-case behaviour of the algorithm. Useful for making predictions and comparing different algorithms.

Let $T(n)$ be the running time of our algorithm.

We say that $T(n)$ is ...	if ...
constant	$T(n) = \Theta(1)$
logarithmic	$T(n) = \Theta(\log n)$
linear	$T(n) = \Theta(n)$
quasi-linear	$T(n) = \Theta(n \log n)$
quadratic	$T(n) = \Theta(n^2)$
cubic	$T(n) = \Theta(n^3)$
exponential	$T(n) = \Theta(c^n)$

Constant time:
Running time does not depend on the size of the input.

- Assignments ($a \leftarrow 42$)
- Comparisons ($=$, $<$, $>$)
- Boolean operations (and, or, not)
- Basic mathematical operations ($+$, $-$, $*$, $/$)
- Constant sized combinations of the above ($a \leftarrow (2 * b + c)/4$)

Calculating the running time of the Naive Algorithm:

```
curr_val, curr_ans ← 0, (None, None)                                  O(1)
for i ← 0 to n − 1 do                                                  ⎫
  for j ← i to n − 1 do                                                ⎪
    // compute A[i] + A[i + 1] + ··· + A[j]                           ⎪
    s ← 0                                                  } O(1)      ⎪
    for k ← i to j do                                                  ⎪
      s ← s + A[k]                                         } O(1)      ⎬ O(n − i)
    // compare to current maximum                                      ⎪
    if s > curr_val then                                               ⎪
      curr_val, curr_ans ← s, (i, j)                       } O(1)      ⎭
return curr_ans                                                        O(1)
```

Let $T(n)$ be the running time of the Naive Algorithm on an instance of size n.

$$\begin{aligned} T(n) &= O(1) + \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} O(j-i) \\ &= O(1) + \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} O(n) \\ &= O(1) + \sum_{i=0}^{n-1} O(n^2) \\ &= O(1) + O(n^3) \\ &= O(n^3) \end{aligned}$$

For your assessments, you will have to design and analyse an algorithm for a given problem. This always consists of three steps:

- Describe your algorithm: A high level description in English, optionally followed by pseudocode. Never submit code!
- Prove its correctness: A formal proof that the algorithm does what it's supposed to do.
- Analyse its time complexity: A formal proof that the algorithm runs in the time you claim it does. Try to model your own solution after the solution published for the tutorial sheets. You are encouraged to use LATEX.

What we will be using in this class closely follows the Python syntax:

- Arrays: we use zero-based indexing
- Slices: `[i:j:k]` is equivalent to Python's `range(i, j, k)`.
- References: Every non-basic data type is passed by reference

Week 2

Topic: **Lists**

Abstract Data Types (ADT)

Type defined in terms of its data items and associated operations - not its implementation. ADTs are supported by many languages. The implementations are specific to the languages.

e.g. Driving a car. The interface - how you interact with the thing: steering wheel and brakes. The implementation is hidden away from you - the nitty gritty details which may be quite complicated.

The **interface** is the means through which you communicate with the specific problem - the type is defined through the types of data items and associated operations.

Benefits of ADT Approach:

- Code is easier to understand if different issues are separated into different places.
- Client can be considered at a higher, more abstract level.
- Many different systems can use the same library, so you only need to code tricky manipulations once, rather than in every client system.
- There can be choices of implementations with different performance tradeoffs - the client doesn't need to be rewritten extensively to change which implementation it uses.

e.g. For an array, we only need to care about an object's index, no need to consider how the actual process of retrieval works.

An ADT is a specification of the desired behaviour from the POV of the user of the data.

A data structure is a concrete representation of data from the POV of an implementer, not a user.

This is similar to the difference between a computational problem and an algorithm.

ADT in programming

- Given as an abstract base class.
- An abc declares methods usually without providing code and we can't construct instances.
- A data structure implementation is a class that inherits from the abc, provides code for all the required methods and has a constructor
- Client code can have variables that are instances of the data structure class, and can call methods on these variables.

Index-Based Lists (List ADT)

It usually supports the following operations:

- `size()`
- `isEmpty()`
- `get(i)`
- `set(i, e)`
- `add(i, e)`
- `remove(i)`

  So every element (e) has an index (i).

  Example

  A sequence of List operations:

Method	Returned value	List content
add(0,A)	-	[A]
add(0,B)	-	[B, A]
get(1)	A	[B, A]
set(2,C)	"error"	[B, A]
add(2,C)	-	[B, A, C]
add(4,D)	"error"	[B, A, C]
remove(1)	A	[B, C]
add(1,D)	-	[B, D, C]
add(1,E)	-	[B, E, D, C]
get(4)	"error"	[B, E, D, C]
add(4,F)	-	[B, E, D, C, F]
set(2,G)	D	[B, E, G, C, F]

Array-based Lists

An option for implementing the list ADT is to use an array , where stores a reference to the element with index (an object stored in allocated memory - contiguous memory block). If array has size , then we can represent lists of size $n <=$.

`get(i)` and `set(i,e)` methods are easy to implement by accessing

- Must check that is valid $(0 <= \ < n)$
- **==Both operations can be carried out in O(1) time, regardless of the array's size.**

Pseudo-code for get:

```
def get(i):
  # input: index i
  # output: ith element in list
    if i < 0 or i ≥ n then
        return "index out of bound"
    else
        return A[i]
```

Pseudo-code for set:

```
def set(i, e):
  # input: index i and value e
  # do: update ith element in list to e
    if i < 0 or i ≥ n then
        return "index out of bound"
    result ← A[i]
    A[i] ← e
    return result
```

`add(i,e)` : we have to make room for the new `e` by shifting forward n elements.

- Start from the 2nd last/last and shift it to the next index.
- Worst time complexity is O(n)

Pseudo-code for add/insertion:

```
def add(i, e):
    if n = N then
        return "array is full"
    if i < n then
        for j in [n-1, n-2, ... , i] do
            A[j + 1] ← A[j]
    A[i] ← e
    n ← n + 1
```

`remove:` we need to fill the hole left at by shifting backward n elements.

- Worst time complexity is O(n)

  Pseudo-code for removal:

```
def remove(i):
    if i < 0 or i ≥ n
        return "index out of bound"
    e ← A[i]
    if i < n-1
        for j in [i, i+1, ... , n-2] do
            A[j] ← A[j+1]
    n ← n - 1
    return e
```

Limitations:

- can represent lists up to the capacity of the array. (n vs N)
  Space complexity: space used is O(N), but we would like it to be O(n)
  Time complexity:
- both get and set take O(1) time
- both add and remove take O(n) time in the worst case.

Positional Lists

- ADT for a list where we store elements at "positions"
- Position - the abstract notion of place where a single object is stored within a container data structure. The position will keep referring to the same entry even after insertion/deletion happens elsewhere in the collection.

  

  Method:
- `element()` : return the element stored at the position instance.

  Operations:

size()	(int) number of elements in the store
isEmpty()	(boolean) whether the store is empty
first()	return position of first element (null if empty)
last()	return position of last element (null if empty)
before(p)	return position immediately before p (null if p is first)
after(p)	return position immediately after p (null if p last)
insertBefore(p, e)	insert e in front of the element at position p
insertAfter(p, e)	insert e following the element at position p
remove(p)	remove and return the element at position p

Singly Linked List

- A concrete data structure: hold hands with only one person
- A sequence of Nodes, each with a reference to the next node.
- List captured by reference (head) to the first Node
- Each Node stores its element and a link to the next node.

  General advice:
- draw the diagram showing the state.