
Structured Programming
COMP1110/6710
Australian National University
Needs ANU Account!
pollev.com/albertofmartin963
Register for Engagement
Image Courtesy NASA/JPL-Caltech.

Computational complexity

Key **computational resources** required to solve a problem in a computer:

- **Time**
- **Space** (i.e., memory)
- **Energy, communication, file I/O traffic, etc.**

Computational complexity is the study of how problem size affects resource consumption. Key question: ***how resource consumption scales (grows) with increasing problem size?***

Two different dimensions (in this course we will focus on the first):

- **Algorithm Complexity**: study of **a particular algorithm** (subject matter of a field in computer science known as *Analysis of Algorithms*)
- **Problem Complexity**: study of **any possible algorithm** that solves the problem (subject matter of a field in computer science known as *Computational Complexity Theory*)

Algorithm complexity

Australian
National
University

Algorithm complexity (I)

- We need to first identify a parameter that characterizes **problem size**
- Usually denoted as the lower-case letter n
- Examples of what n may denote:
  - the number of elements in a list
  - the number of rows/columns in a matrix (i.e., array of arrays)
  - the number of samples in a data set

Algorithm complexity (II)

- Then, we need to study the algorithm to determine how resource consumption grows as a function of n (the difficult part!)
- Depending on the algorithm at hand, **the value of the input**, not just its size, may influence resource consumption
- This leads to different variants of complexity (*we will focus on the first two*):
  - **Worst-case** analysis considers input of size n that maximizes resource consumption
  - **Best-case** analysis considers input of size n that minimizes resource consumption
  - **Average-case** analysis considers all possible inputs of size n and averages their resource consumption
  - **Amortized analysis** considers a sequence of executions of the algorithm over an input of size n, averaging their resource consumption. Typically useful for data structures that implement operations with internal state, and depending on their internal state, may require a varying amount of resource consumption

Big O notation (formal definition)

Mathematical notation that we use to denote computational complexity

Assume that we have a problem of size n and an algorithm that we know takes $g(n)$ resource units to solve the problem

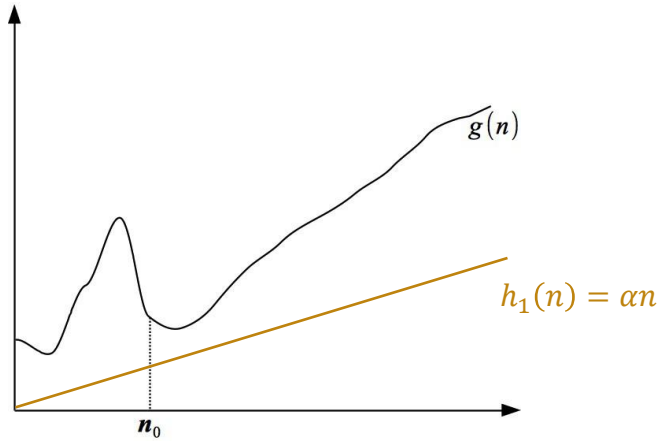
We say that $g(n) \in O(f(n))$ **if and only if:**

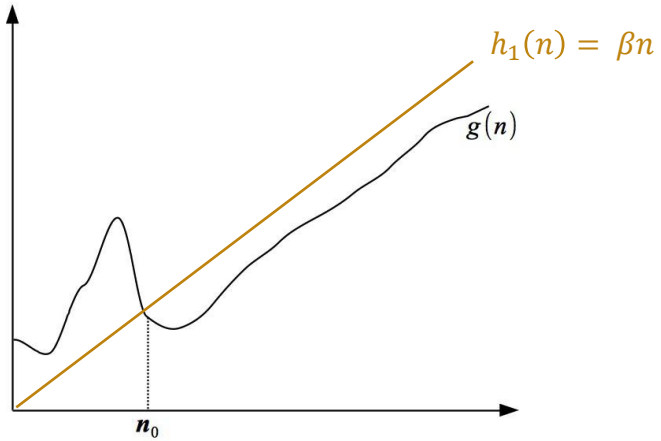
(1) there exist constants $c > 0$ and $n_0 > 0$ such that for all $n > n_0$ we have that $g(n) \leq cf(n)$

(2) $f(n)$ is the function that provides the tightest possible upper bound in (1)

Big O notation (example)

Which function would be $f(n)$ in this case?

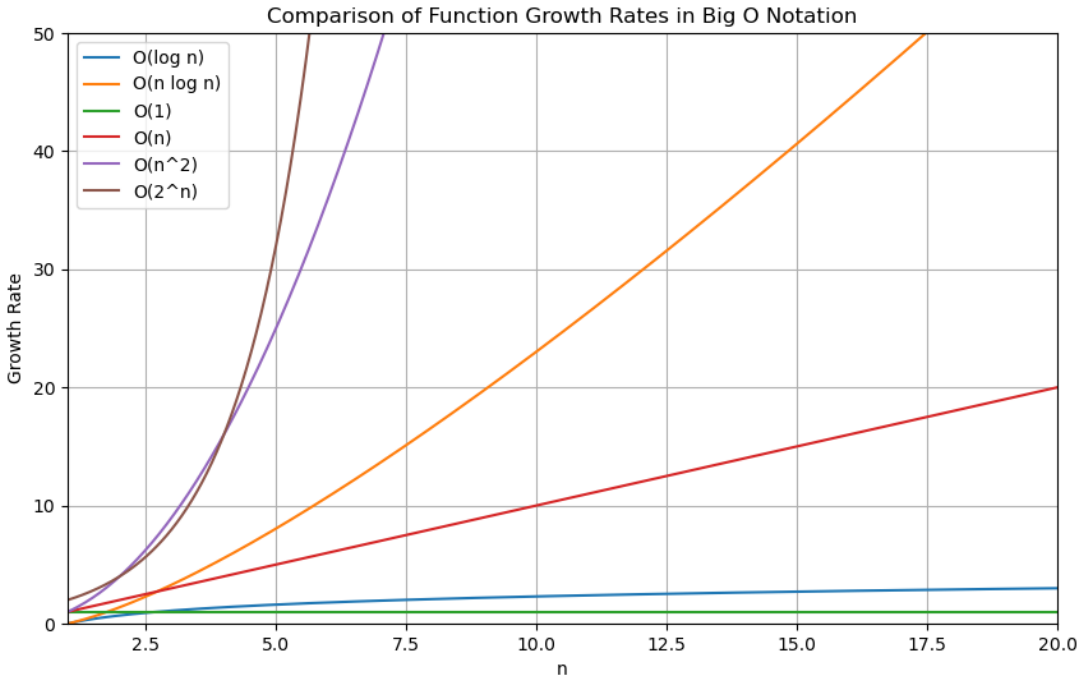




Big O notation (practical considerations)

- $g(n) \in O(f(n))$ roughly means "$g(n)$ grows at the rate of $f(n)$ for large enough n"
- $f(n)$ is determined by the **leading order** of $g(n)$ (i.e., term that dominates as $n \to \infty$)
- **We do not care about constants** when dealing with Big O notation
- For example:
  - $n^2 - 2n + 1$ is $O(n^2)$ (**quadratic** growth rate)
  - $100n$ is $O(n)$ (**linear** growth rate)
  - 10^{18} is $O(1)$ (**constant** growth rate, i.e., no growth with n)

Most common rates of growth (i.e., $f(n)$ functions)



Time complexity

- Time complexity analysis determines the rate of growth of the number of **elementary operations** (e.g., comparisons, arithmetic operations, assignment statements, etc.) that an algorithm requires to solve the problem as problem size grows
- We assume that elementary operations take one unit of time to complete (this is not actually the case when an actual code executes on an actual machine)
- We are not actually concerned with predicting actual computational times when the algorithm runs in a particular computer, e.g., in microseconds (that would be highly machine-dependent, and much harder to estimate with state-of-the-art hardware)
- **Warning:** library function calls do not count as elementary operations. They may indeed trigger a complex algorithm for which you need to know the time complexity when determining the time complexity of the calling algorithm
- Let us start with some simple examples

Summing elements in a list

Assume that the input list has n elements. At which rate does the number of elementary operations growth with n? Use Big O notation to express it.

```
int sum(ArrayList<Integer> list)
{
    int s = 0;                  1
    for (var i: list) {         n
        s += i;                 n
    }
    return s;                   1
}
```

$$g(n) = 2 + 2n \rightarrow g(n) \in O(n)$$

Linear time complexity

Minimum difference among any two elements in a list

Assumption: `values.size()` and `values.get(…)` take unit time

```
int minDiff(ArrayList<Integer> values)
{
  int min =INTEGER.MAX_VALUE;  1
  for (int i=0; i<values.size(); i++) {  n
    for (int j=i+1; j<values.size(); i++) {  n(n − 1)/2
      int diff = Math.abs(values.get(i)-values.get(j))  n(n − 1)/2
      if (diff < min) {  n(n − 1)/2
        min=diff;  n(n − 1)/2
      }
    }
  }
  return min;
}
```

Note that $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$

$$g(n) = 1 + n + 4\frac{n(n-1)}{2} = 1 + n + 2n^2 - 2n = 2n^2 - n + 1$$

Thus, $g(n) \in O(n^2)$ Quadratic time complexity

Question: can we do better?

A more interesting example: Find greatest up to

- Given an unsorted list of integers with n elements, find the largest element in the list $\leq x$, with x being a given input integer number. If all the elements in the list are larger than x, return `null`
- Let us analyse the time complexity of two different approaches:
  1. Directly search the unsorted list
  2. First sort the list in increasing order, then search the sorted list

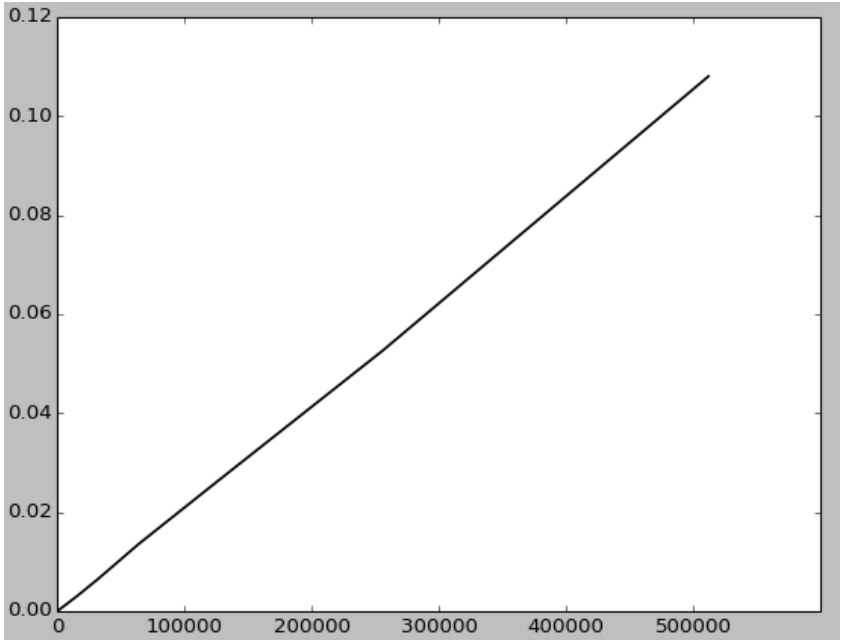
Unsorted Greatest Up To

```
Integer unsortedGreatestUpTo
        (ArrayList<Integer> list, int x)
{
   Integer best = null;
   for (var e: list)
   {
     if (e==x)
       return e;
     if ( e<x && (best == null || e>best))
       best=e;
   }
   return best;
}
```

Time complexity analysis

- Best-case: `list.get(0)==x`
- Worst-case:
  - `list == [x-n, x-n+1, …, x-2, x-1]`
  - We perform all comparisons and update `best` at every loop iteration
  - $g(n) = cn \rightarrow g(n) \in O(n)$ (linear time complexity)

Unsorted Greatest Up To



Actual time measurements versus problem size

Sorted Greatest Up To (Binary search)

```
Integer sortedGreatestUpTo
     (ArrayList<Integer> list, int x)
{
  if (list.isEmpty() || list.get(0) > x)
     return null;
  int lower=0;
  int upper=list.size(); //one past the end
  while (upper-lower>1) {
    int mid = (lower+upper)/2; //int division
    if (list.get(mid) <= x)
      lower = mid;
    else
      upper = mid;
  }
  return list.get(lower);
}
```

Time complexity analysis

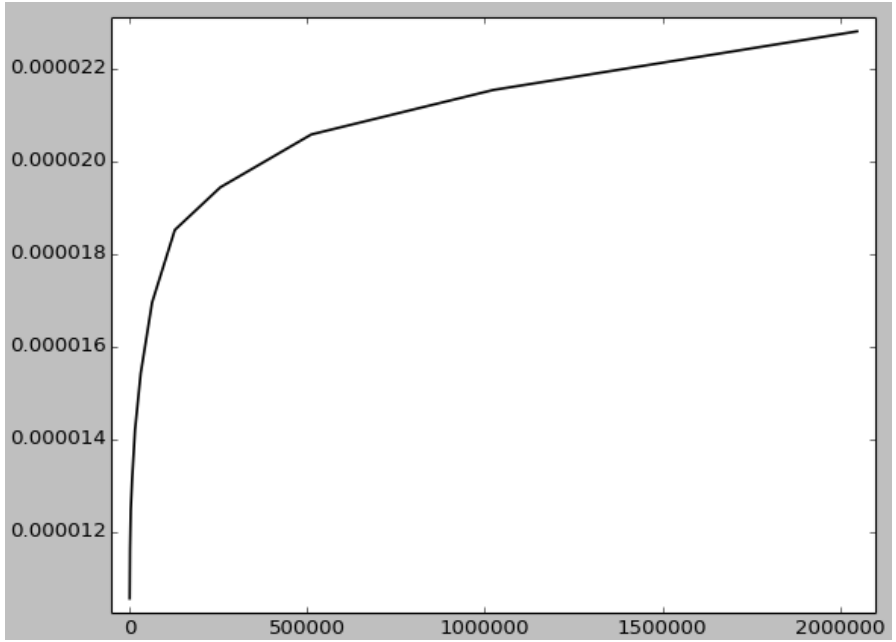
How many loop iterations?

- Initially, `upper-lower==n`
- The difference is halved at every iteration
- Can halve it at most $log_2(n)$ times before it becomes 1
- $g(n) = a\ log_2(n) + b \rightarrow g(n) \in O(\log(n))$
  (logarithmic time complexity)

Note: Loop invariant (assuming no repeated elements)

`list.get(lower) <= x < list.get(upper)`

Sorted Greatest Up To



Actual time measurements versus problem size

Problem complexity

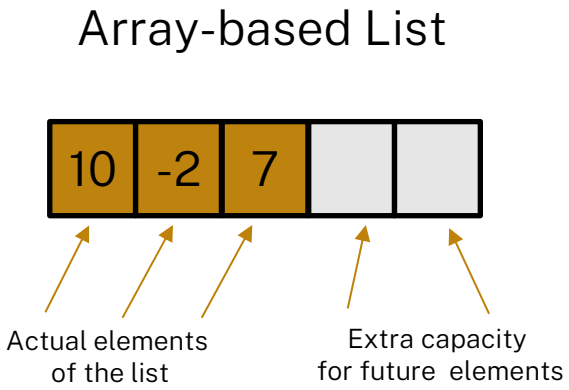
Australian
National
University

Problem complexity

- The complexity of a problem is the amount of resources that **any** algorithm that solves the problem **must** use, in the worst case, as a function of the size of the arguments
- In other words, the complexity of a problem is the **infimum** of the complexities among all algorithms that solve the problem
- For example, it is possible to proof that **any possible sorting algorithm** that uses pair-wise comparisons **needs at least** $O\big(nlog(n)\big)$ **comparisons in the worst case**
- Proving these kind of results is out of the scope of the course, and in general, it requires advances arguments in the mathematical theory of computation

MyArrayList<E>

Let us write our own class implementation of the `List<E>` generic interface using an array of elements of type E, using the concept illustrated below, and study the best-case and worst-case time complexity of different operations



Australian National University

Practice

Design a **generic** class MyArrayList<E>, parameterized by the type of the elements E, that implements a partial set of the methods in the List<E> interface, namely: (1) **add** a new element; (2) **remove** an element; (3) **size**; (4) **isEmpty**; (5) **get**; 6) **set**; (7) **containsAll**; (8) **iterator**. Follow the design recipe! Perform a best-case and worst-case time complexity analysis of each of these methods, and express the result of the analysis using Big O notation.

In order to get you started, we will demonstrate the class definitions, the implementation and time complexity analysis of (1). Then, you will have time to implement and analyse during the workshop the rest of operations, starting with operation (2).