Computational Complexity

Measuring by counting, Big O notation, growth rates, growth rates for list algorithms

Algorithms

An **algorithm** is a step-by-step description of how to solve a problem.

- Example problem: (Search) Find if a particular item belongs to a list.
- Algorithm: Go through the list elements, one-at-a-time starting from the front. For each element, compare it to the item you are searching. Stop when there's a match (true) or when you run off the end of a the list (false – no match).
- Can translate the algorithm into code:

```
Node p = head;
while (p != null) {
   if (item.equals(p.data))
      return true;
   p = p.next;
}
return false;
```

Measuring efficiency

- Algorithms can be coded into different languages and run on different machines.
- You can't measure the efficiency of an algorithm by determining how long it takes to run.
- To measure efficiency, you essentially count single statements. Assume:
  - Single, simple statements (not calls to methods, for example) all take the same amount of time to execute.
  - The amount of time to execute a method depends on:
    - The number of simple statements inside the method body.
    - The time required for a call to any other method inside inside the body.
  - The amount of time required to execute a loop depends on the number of times the loop body executes. Do the same counting as for a method.

Worst case

Algorithm efficiency (time efficiency) is measured by how many simple statements are executed for an input of size *n*. In the worst case:

```
Node p = head;
while (p != null) {
   if (item.equals(p.data))
      return true;
   p = p.next;
}
return false;
```

1 statement

n comparisons for null (loop condition)
n comparisons for equality
n assignments

1 statement

$3n + 2$ statements executed

Statement count & execution time

- The time required for an algorithm to solve a problem is a function *T(n)*, where *n* is the size of the instance of the problem.
- The **worst-case time** is the maximum time taken (number of statement executions) over all instances of size *n*.
- If *T(n)* is a polynomial, then the algorithm is a **polynomial time algorithm**.
- Problems that cannot be solved in polynomial time (or less) are considered to be **intractable**.

Computational complexity

- **Computational complexity** is a branch of theoretical computer science concerned with the difficulty of problems in terms of how many resources are required (time or space) to solve them.
- **Big O notation** gives an estimate of the efficiency of an algorithm. It uses only the *dominant term* of the running time $T(n)$.
- An algorithm with $T(n) = 4n^3 + 265n + 1000000$ would be $O(n^3)$.
- An algorithm with $T(n) = n^3 log_2 n + 5n2 + 345678n – 1289$ would be $O(n^3 \log n)$.

Algorithmic growth rates

This table shows growth rates of common algorithms.

Complexity	Comment	Example Algorithm
$O(1)$	Time independent of input size (constant)	Look up k^{th} element of an array
$O(\log n)$	Time grows by 1 when input size doubles (logarithmic)	Binary search
$O(n)$	Time proportional to input size (linear)	Linear search
$O(n \log n)$	Best time for sorts	Mergesort
$O(n^2)$	Double the input size -> quadruple the time (quadratic)	Elementary sorts (bubble, selection, insertion)
$O(n!)$	Intractable	Brute force Traveling Salesman Problem

Do growth rates matter?

Absolutely, yes! Algorithms can be too slow to be useful.

This table shows f(n) when n is 8.

$f(n)$	Value at n = 8
$f(n) = 1$	1
$f(n) = \log_2 n$	3
$f(n) = n$	8
$f(n) = n \log_2 n$	24
$f(n) = n^2$	64
$f(n) = 2^n$	4,096
$f(n) = n!$	40,320

Complexities of list algorithms

The computational complexity of an algorithm depends in part on the underlying data structure.

Operation	**LinkedList**	**ArrayList**
add(value) // at end	O(n)	O(1) [If not full]
addToFront(value)	O(1)	O(n)
indexOf(value)	O(n)	O(n)
get(index)	O(n)	O(1)
remove(index)	O(n)	O(n)
size()	O(1) or O(n) depending on implementation	O(1)