Objectives

- Be able to discuss the issue of when one algorithm is better than another
- Be able to identify the “size of the problem” for a given problem
- Be able to determine when a problem’s size can be thought of in terms of the value of a single integer parameter
- Be able to identify “significant operations” in an algorithm
- Be able to say how a time function T(n) is “measured” by another, simpler, “ideal function” f(n)
- Be able to precisely define the statement “T(n) is O(f(n))
- Be able to define what is meant by the constant function, the linear function, the quadratic function, the cubic function, and in general what is meant by a polynomial function, as ideal functions
- Be able to define what is meant by the log function, the linear log function, an exponential function, and the factorial function, as ideal functions
- Be able to identify the shape
- Given an algorithm or Python function, be able to determine its complexity in “Big-O” terms

Chapter 2 - Analysis

- Algorithm analysis – when is one algorithm better than another?

The “size of the problem”

- Different for each problem being solved
  - Inventory problem
  - Flight scheduling problem
  - DNA sequencing problem
  - Clustering problems, where we look for a partition of a set
  - etc
- But be careful – not every problem can be thought of this way

Significant operations

Big-O Notation and Meaning

- n is the size of the problem
- $T(n)$ is the number of significant operations executed for a problem of size n
- We use a "model function" $f(n)$ and try to make some multiple of $f(n)$ an *upper bound* of $T(n)$. More precisely…
  - $T(n)$ is $O(f(n))$ provided there is some constant C and some value N of n for which, if $n \geq N$, then $T(n) \leq Cf(n)$
- Here we are "measuring a function with a function"
  - But $f(n)$ is not meant to precisely measure $T(n)$. It is just an upper bound on how "bad" $T(n)$ can be.

Some example complexities

- Constant
- Linear
- Log
- Log linear
- Quadratic
- Cubic
- Exponential
- Factorial

Polynomial complexity

- If T(n) is a polynomial, then its complexity is that of its highest-order term

- It doesn't matter if the power is integer or real. It just needs to be non-negative to be called polynomial complexity.

Example from the text

```
a = 5
b = 6
c = 10
for i in range(n):
    for j in range(n):
        x = i * i
        y = j * j
        z = i * j
for k in range(n):
    w = a*k + 45
    v = b*b/k
d = 33
```

Example

```
Algorithm transpose(A: a square matrix)
  for row = 0, size(A)-1
    for col = row+1, size(A[row])-1
      swap A[row][col] with A[col][row]
```

Pragmatics

- In actual use, $T(n)$ is the most important thing, not $f(n)$.
- If n never goes outside a certain range, it may be better to use an "inferior" algorithm

Looking for evidence

- How does an algorithm's complexity show up in a program?
  - Isolate the algorithm in a function
  - Time how long it takes for the function to execute for different sizes of the problem, and thus produce a simulated set of (n, T(n)) pairs
  - Graph the set of pairs
  - Look at the shape of the graph
    - If you know the algorithm's best Big O upper bound f(n), the graph should have a similar shape to a corresponding set of (n, Cf(n)) pairs
    - If you do not know the algorithm's complexity, the shape of the graph is a strong clue

Example: Inserting into the middle of a Python list

```
from time import clock
from random import Random
ran = Random()

def populate(lis,number):
    for i in range(number): lis.append(ran.random())

def insertZeroInMiddle(lis):
    lis.insert(len(lis)//2, 0)

if __name__ == "__main__":
    lis = []
    for listsize in range(10000, 500000, 10000):
        lis.clear()
        populate(lis,listsize)
        begin = clock()
        insertZeroInMiddle(lis)
        end = clock()
        print(str(listsize)+"\t"+str(end-begin))
```