Measuring the Efficiency of Algorithms

We have two algorithms: alg1 and alg2 that solve the same problem. Our application needs a fast running time.

How do we choose between the algorithms?

Issues with this approach:

1. How are the algorithms coded? We want to compare the algorithms, not the implementations.
2. What computer should we use? Results may be sensitive to this choice.
3. What data should we use?

Want: techniques that analyze algorithms independently of specific details such as implementation, hardware, or data.

Solution: analyze the number of operations the algorithm will perform for an input of given size.

Example: copying an array with n elements requires \( \ldots \) operations.

Algorithm A requires \( \frac{n^2}{2} \) operations to solve a problem of size \( n \).

Algorithm B requires \( 5n + 10 \) operations to solve a problem of size \( n \).

Which one would you choose?

For large enough problem size, algorithm B is more efficient.

We focus on the growth rate:

- Algorithm A requires time proportional to \( n^2 \ - O(n^2) \)
- Algorithm B requires time proportional to \( n - O(n) \)

Strictly speaking, \( O(n) \) means the running time is bound by a constant times \( n \).
Common Shapes: Constant
- \(O(1)\)
- examples?

Common Shapes: Linear
- \(O(n)\)
- \(f(n) = an + b\)

Linear
Example: copying an array
\[ a = [0] * \text{len}(b) \]
\[ \text{for } i \in \text{range}(\text{len}(b)) : \]
\[ a = b[i] \]

Same as: \(a = b[:]\)

Just because you can do something in a single statement does not mean it takes \(O(1)\)!

Other Shapes: Sublinear

Common Shapes: logarithm
- \(\log_b n\) is the number \(x\) such that \(b^x = n\)
  - \(2^2 = 8\)
  - \(\log_2 8 = 3\)
  - \(2^4 = 16\)
  - \(\log_2 16 = 4\)
- logarithm is a very slow-growing function

Quadratic
\(O(n^2)\):
\[ \text{for } i \in \text{range}(n) : \]
\[ \quad \text{for } j \in \text{range}(n) : \]
Combinations

Additive complexity:
Suppose your algorithm is composed of two sequential operations one taking $f_1(n)$ operations on inputs of size $n$ and the other taking $f_2(n)$ operations, then the running time for the algorithm is $O(\max(f_1(n), f_2(n)))$.

Combinations

Sequential
- Big-O bound: Steepest growth dominates
- Example: copying of array, followed by binary search
  - $n \times \log(n)$, $O(\log n)$
- Embedded code
  - Big-O bound multiplicative
  - Example: a for loop with $n$ iterations and a body taking $O(\log n)$, $O(\log n)$

Worst and Average Case Time Complexity

Worst case
- just how bad can it get: the maximal number of steps
- our focus in this course

Average case
- amount of time expected “usually”
- In this course we will hand wave when it comes to average case
- Example: searching for an item in an unsorted array

Practical Analysis – Dependent loops

```
  ...  
  for i in range(n) :  
    for j in range(i) :  
      ...  
  i = 0:  inner-loop iters =0
  i = 1:  inner-loop iters =1
  ...  
  i = n-1: inner-loop iters =n-1
  Total = 0 + 1 + 2 + ... + (n-1) = n(n-1)2 / 2 = n(n-1)/2
  f(n) = n(n-1)/2  
  O(n^2)
```

Examples

- Copying a list
  - $O(n)$, where $n$ is size of the list
- Linear search in a list for a particular value
  - Worst case $O(n)$, Average case $O(n)$, where $n$ is the list size
- Enumerate all lowercase strings of length $n$
  - $O(26^n)$
- Insert element in a list
  - Worst case $O(n)$, Average case $O(n)$, where $n$ is the number of elements in the array

Final Comments

- Order-of-magnitude analysis focuses on large problems
- If the problem size is always small, you can probably ignore an algorithm’s efficiency
- Weigh the trade-offs between an algorithm’s time requirements and its memory requirements, expense of programming etc.