Algorithm Complexity

Introduction

- Algorithms need to be analyzed for their efficiency, in terms of
  - Running time ➔ computational complexity
  - Size of used memory ➔ memory complexity
- An algorithm may run faster/slower and use more/less on certain data sets than on others
  ➔ many indicators for assessing the efficiency:
  - Average case
  - Best case (lower bound)
  - Worst case (upper bound)
  - Most common case
- How to measure complexity?
  - Experimental studies
  - Theoretical analysis with pseudo-code, flowcharts

Big-O notation

- Definition: Suppose $f(n)$ and $g(n)$ are non-negative functions of n. Then we say that $f(n)$ is O($g(n)$) if there exists constants $C > 0$ and $N > 0$ such that for all $n > N$, $f(n) \leq Cg(n)$.

- This says that function $f(n)$ grows at a rate no faster than $g(n)$, thus $g(n)$ is an upper bound on $f(n)$

- Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n
  - It represents the computational/memory complexity of algorithms

Examples

- For $f(n) = 3n + 5$ and $g(n) = n$ there are positive constants C and N such that $f(n) \leq Cg(n)$ for $n > N$
  ➔ $3n + 5$ is O(n)

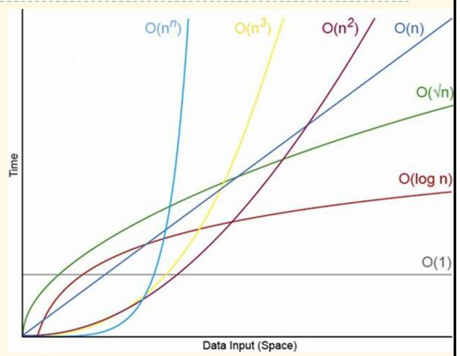
- $3n^2 + 5n + 4$ is O(n^2)

- $n + \sqrt{n}$ is O(n)

- $2^n + n^2$ is O(2^n)

Common Classes

- Constant: O(1)
- Linear: O(n)
- Quadratic: O(n^2)
- Polynomial: O(n^k), $k \geq 1$
- Exponential: O(a^n), $n > 1$
- Logarithmic: O$(\log n)$
- Factorial: O$(n!)$



- Efficiency comparison of classes
  - O(1) < O$(\log n)$ < O$(\sqrt{n})$ < O(n) < O(n^2) < O(n^3) < O(2^n) < O(3^n) < O$(n!)$ < O(n^n)

Asymptotic Notation

- It is correct to say $3n + 2$ is O(n^3), O(2^n)
  ➔ One should make approximation as tight as possible
  ➔ Asymptotic notation

- Simple rule: Drop lower order terms and constant factors
  - $3n + 2$ is O(n)
  - $8n^2\log n + 5n^2 + n$ is O$(n^2\log n)$

Computational Complexity Analysis

- Break down into number of primitive operations

- Example 1: Find the maximum element of an array.
  - Algorithm ArrayMax(*A*, *n*):

```
Algorithm ArrayMax(A, n):
  Input: Array A storing n integers.
  Output: The maximum element in A.

  currentMax ← A[0]
  for i ← 1 to n-1 do
    if currentMax < A[i] then        1 step      n iterations
      currentMax ← A[i]

  return currentMax
```


Computational Complexity Analysis (*cont'd*)

- Example 2: Compute prefix averages.

```
Algorithm PrefixAverages(X):
  Input: Array X of n numbers.
  Output: Array A of n numbers such that A[i]
          is the average of X[0..i].

  Initialize A as array of n numbers.
  for i ← 0 to n-1 do
    a ← 0
    for j ← 0 to i do
      a ← a + X[j]      1 step     i iterations (i = 0..n-1)     n iterations
    A[i] ← a/(i+1)

  return A
```

Computational Complexity Analysis (*cont'd*)

- Example 3: Compute prefix averages.
  - ```
    Algorithm BetterPrefixAverages(X):
      Input: Array X of n numbers.
      Output: Array A of n numbers such that A[i]
              is the average of X[0..i].

      Initialize A as array of n numbers.
      a ← 0
      for i ← 0 to n-1 do                  } n iterations
        a ← a + X[i]            } 1 step
        A[i] ← a/(i+1)

      return A
    ```


Memory Complexity Analysis

- Revisit above examples and determine the memory complexity
    ```