Code Complexity

Measuring Algorithm Efficiency

Algorithm and Analysis

- **Algorithm**
  - A step-by-step procedure for solving a problem

- **Analysis of Algorithm**
  - To evaluate rigorously the **resources** (**time** and **space**) needed by an algorithm and **represent the result of the evaluation with a formula**
  - We focus more on **time** requirement in our analysis
  - The time requirement of an algorithm is also called the **time complexity** of the algorithm

Measure Actual Running Time?

- We can measure the actual running time of a **program**
  - Use **wall clock time** or **insert timing code** into program

- However, running time is not meaningful when **comparing two algorithms:**
  a. Coded in different languages
  b. Using different data sets
  c. Running on different computers

Counting Operations

- Instead, we count the number of **operations**
  - e.g. *arithmetic*, *assignment*, *comparison*, etc.

- Counting an algorithm's operations is a way to assess its **efficiency**
  - An algorithm's execution time is related to the number of operations it requires

Example: Counting Operations

```
for (int i = 1; i <= n; i++) {
   perform 100 operations;          // A

   for (int j = 1; j <= n; j++) {
      perform 2 operations;         // B
   }
}
```

Total Ops = **A + B** $= \sum_{i=1}^{n} 100 + \sum_{i=1}^{n} (\sum_{j=1}^{n} 2)$

$$= 100n + \sum_{i=1}^{n} 2n \quad = 100n + 2n^2 \quad = 2n^2 + 100n$$

Example: **Counting Operations**

- Knowing the number of operations required by the algorithm, we can state that
  - **Algorithm *X*** takes $2n^2 + 100n$ operations to solve problem of size n

- If the time t needed for one operation is known, then we can state
  - **Algorithm *X*** takes $(2n^2 + 100n)t$ time units

Example: Counting Operations

- However, time ***t*** is directly dependent on the factors mentioned earlier
  - E.g. different languages, compilers and computers

- Instead of tying the analysis to actual time ***t***, we can state
  - ***Algorithm X*** takes time that is **proportional to** $\mathbf{2n^2 + 100n}$ for solving problem of size ***n***

Approximation of Analysis Results

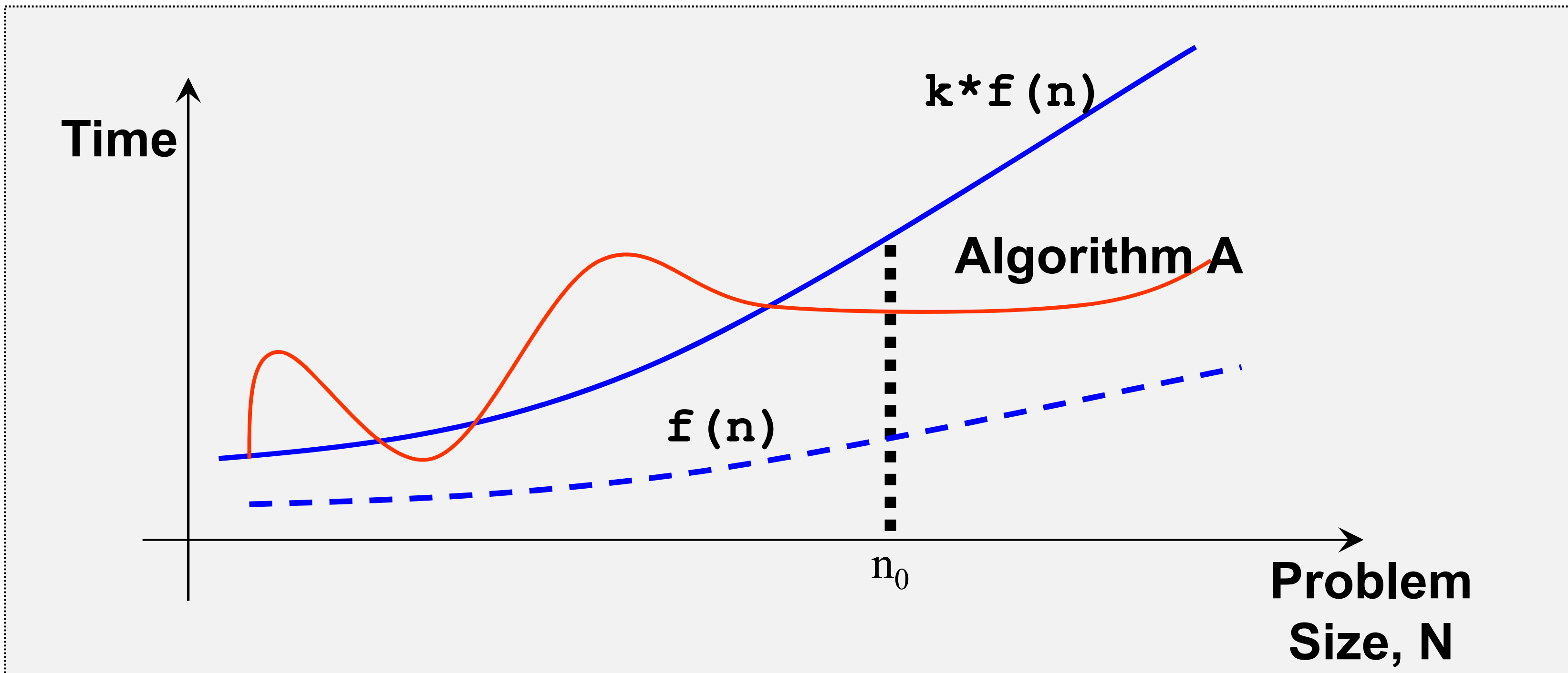
- Suppose the time complexity of
  - Algorithm ***A*** is $\mathbf{3}\boldsymbol{n}^{\mathbf{2}} + 2n + \log n + 30$
  - Algorithm ***B*** is $\mathbf{0.39}\boldsymbol{n}^{\mathbf{3}} + n$
- Intuitively, we know Algorithm *A* will outperform *B*
  - When solving larger problem, i.e. larger ***n***
- The **dominating term** $\mathbf{3}\boldsymbol{n}^{\mathbf{2}}$ and $\mathbf{0.39}\boldsymbol{n}^{\mathbf{3}}$ can tell us approximately how the algorithms perform
- The terms $\boldsymbol{n}^{\mathbf{2}}$ and $\boldsymbol{n}^{\mathbf{3}}$ are even simpler and preferred
- These terms can be obtained through **asymptotic analysis**

Asymptotic Analysis

- An analysis of algorithms that focuses on
  a. Analyzing problems of **large input size**
  b. Consider **only the leading term** of the formula
  c. **Ignore the coefficient** of the leading term

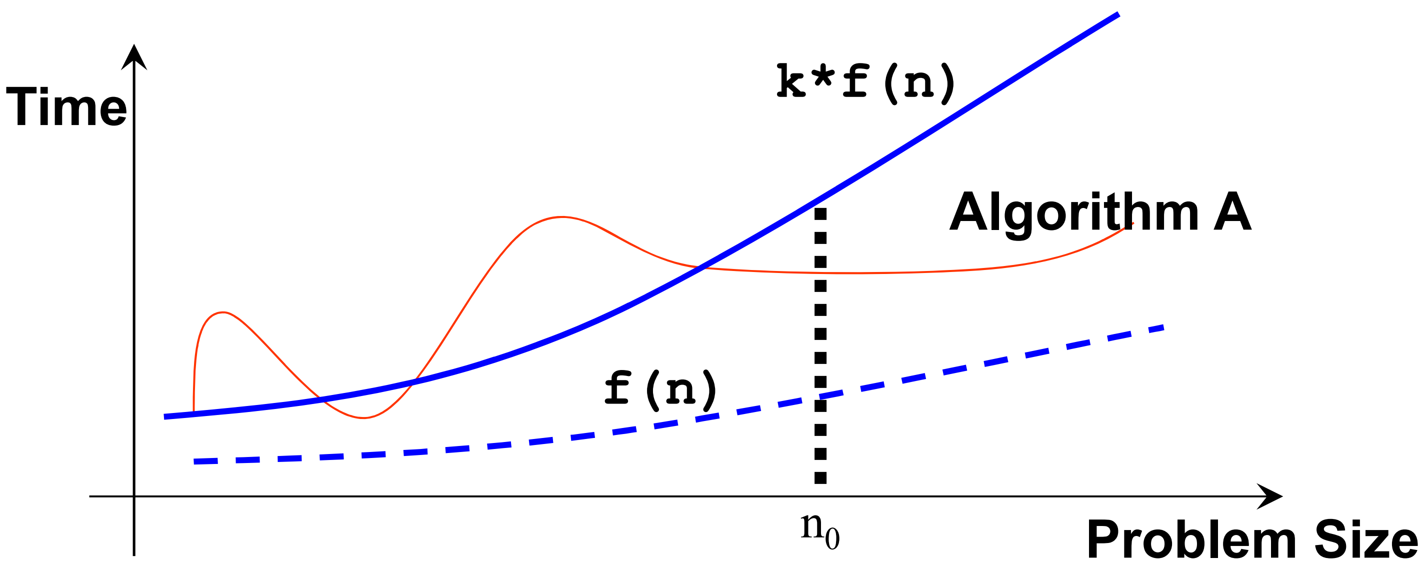
The Big-O Notation: Definition

Algorithm *A* is of **O(*f*(*n*))**
if there exist a **constant *k***, and a **positive integer n_0**
such that Algorithm *A* requires
no more than *k* * *f*(*n*) time units to
solve a **problem of size *n* >= n_0**



The Big-O Notation

- When problem size is larger than n_0, Algorithm *A* is **bounded from above** by $k * f(n)$
- Observations
  - n_0 and k are **not unique**
  - There are many possible $f(n)$



Example: Finding n_0 and k

- Given complexity of Algorithm A is $2n^2 + 100n$
- **Claim:** Algorithm A is of $O(n^2)$
- Solution
  - $2n^2 + 100n < 2n^2 + n^2 = \mathbf{3n^2}$ whenever $\boldsymbol{n > 100}$
  - Set the constants to be $\boldsymbol{k = 3}$ and $\boldsymbol{n_0 = 101}$
  - By definition, we say **Algorithm A** is $\boldsymbol{O(n^2)}$

Corrected after eLecture

- Questions
  - Can we say A is $O(2n^2)$ or $O(3n^2)$?
  - Can we say A is $O(n^3)$?

Growth Terms

- By asymptotic analysis, it is clear that:
  - Coefficient of the **f(n)** can be absorbed into the constant k
  - E.g. **A** is `O(3n²)` with constant k_1

    ➔ **A** is `O(n²)` with constant $k = k_1 * 3$
  - So, **f(n)** can be reduced to function with **coefficient of 1** only
- Such a term is called a **growth term**
- Ordered list of the commonly seen **growth terms**:

$O(1) < O(\lg(n)) < O(n) < O(n \lg(n)) < O(n^2) < O(n^3) < O(2^n)$

"fastest" ***"slowest"***

- "lg" = $\log_2$
- In big-O, log functions of different bases are all the same (why?)

THANK YOU!