Algorithm Efficiency

Algorithmic Thinking

Luay Nakhleh

Department of Computer Science

Rice University

All Correct Algorithms Are Not Created Equal

- When presented with a set of correct algorithms for a certain problem, it is natural to choose the most efficient one(s) out of these algorithms.

- (In some cases, we may opt for an algorithm X that's a bit slower than algorithm Y, but that is much, much easier to implement than Y.)

- Efficiency:

  - Time: how fast an algorithm runs

  - Space: how much extra space the algorithm requires

  - There's often a trade-off between the two.

It's All a Function of the Input's Size

- We often investigate the algorithm's complexity (or, efficiency) in terms of some parameter n that indicates the algorithm's input size

- For example, for an algorithm that sorts a list of numbers, the input's size is the number of elements in the list.

- For some algorithms, more than one parameter may be needed to indicates the size of their inputs.

  - For the algorithm IsBipartite, the input size is given by the number of nodes if the graph is represented as an adjacency matrix, whereas the input size is given by the number of nodes and number of edges if the graph is represented by an adjacency list.

It's All a Function of the Input's Size

- Sometimes, more than one choice of a parameter indicating the input size may be possible

- For an algorithm that multiplies two square matrices, one choice is n, the order of the matrix, and another is N, the total number of entries in the matrix.

- Notice that the relation between n and N is easy to establish, so switching between them is straightforward (yet results in different qualitative statements about the efficiency of the algorithm).

- When the matrices being multiplied are not square, N would be more appropriate.

It's All a Function of the Input's Size

- Question: For an algorithm that tests whether a nonnegative integer p is prime, what is the input size?

Units for Measuring Running Time

- We'd like to use a measure that does not depend on extraneous factors such as the speed of a particular computer, dependence on the quality of a program implementing the algorithm and of the compiler used, and the difficulty of clocking the actual running time of the program.

- We usually focus on basic operations that the algorithm executes, and compute the number of times the basic operations are executed.

Orders of Growth

- A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones (2 steps vs. 15 steps is not really a difference when it comes to the running times of algorithms!)

- When analyzing the complexity, or efficiency, of algorithms, we pay special attention to the order of growth of the number of steps of an algorithm on large input sizes.

Orders of Growth

- The complexity analysis framework ignores multiplicative constants and concentrates on the order of growth of the number of steps to within a constant multiple for large-size inputs.

Worst-, Best-, and Average-Case Complexity

- The worst-case complexity of an algorithm is its complexity for the worst-case input of size n, which is an input of size n for which the algorithm runs the longest among all possible inputs of that size.
  - What type of bound does the worst-case analysis provide on the running time?
- The best-case complexity of an algorithm is its complexity for the best-case input of size n, which is an input of size n for which the algorithm runs the fastest among all possible inputs of that size.
- Neither of these two types of analyses provide the necessary information about an algorithm's behavior on a "typical" or "random" input. This is the information that the average-case complexity seeks to provide.

Worst-, Best-, and Average-Case Complexity

- Best-case analysis is usually uninformative about the efficiency of an algorithm.

- Average-case analysis is often very hard to conduct (and, sometimes it is not even clear what the "average" case is).

- Therefore, most analyses focus on the worst case, which is what we will focus on as well.

Worst-case Complexity: Example 1

Algorithm 1: LinearSearch.

Input: An array $A[0 \ldots n-1]$ of integers, and an integer x.
Output: The index of the first element of A that matches x, or -1 if there are no matching elements.

```
1 i ← 0;
2 while i < n and A[i] ≠ x do
3 |   i ← i + 1;
4 if i ≥ n then
5 |   i ← −1;                                  // x was not found in A
6 return i
```

Worst-case Complexity: Example 2

Algorithm 2: MatrixMultiplication.

Input: Two matrices $A[0 \ldots n-1, 0 \ldots k-1]$ and $B[0 \ldots k-1, 0 \ldots m-1]$.
Output: Matrix $C = AB$.

```
1 for i = 0 to n − 1 do
2     for j = 0 to m − 1 do
3         C[i, j] ← 0;
4         for l = 0 to k − 1 do
5             C[i, j] ← C[i, j] + A[i, l] · B[l, j];
6 return C
```

Worst-case Complexity: Example 3

Algorithm 1: IsBipartite.

Input: Undirected graph $g = (V, E)$.
Output: *True* if g is bipartite, and *False* otherwise.

```
1  foreach Non-empty subset V_1 ⊂ V do
2      V_2 ← V \ V_1;
3      bipartite ← True;
4      foreach Edge {u, v} ∈ E do
5          if {u, v} ⊆ V_1 or {u, v} ⊆ V_2 then
6              bipartite ← False;
7              Break;
8      if bipartite = True then
9          return True;
10 return False;
```

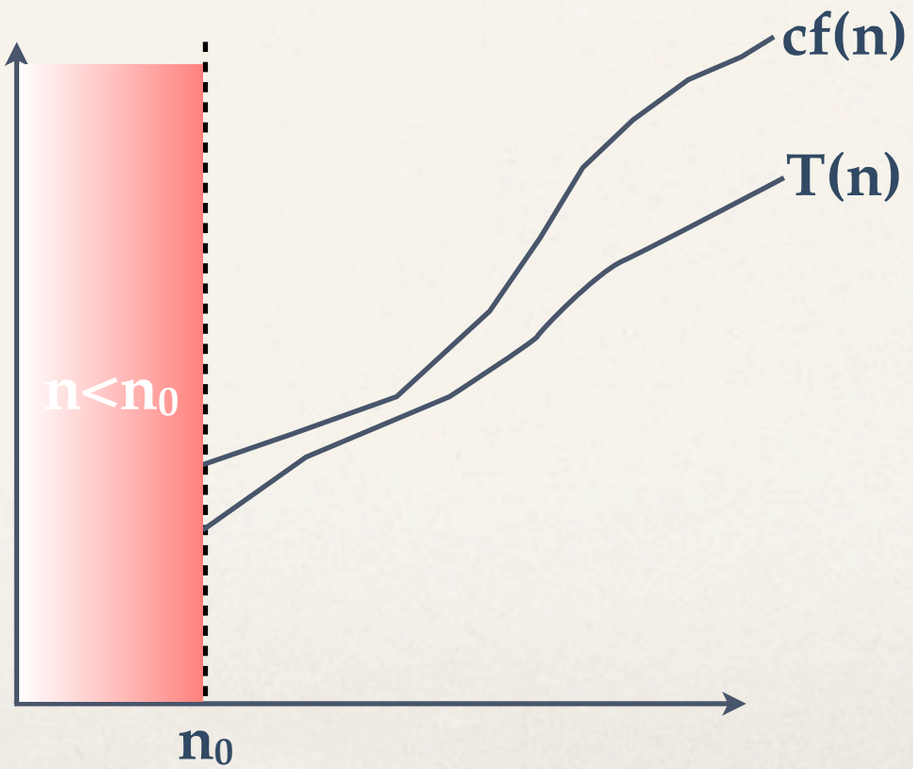
Asymptotic Notations

- As we stated before, in mathematical analysis of efficiency, we ignore multiplicative constants.

- For example,
  - $2n^2+3n-5 \rightarrow n^2+n$
  - $10000n \rightarrow n$
  - $2n^2 \rightarrow n^2$

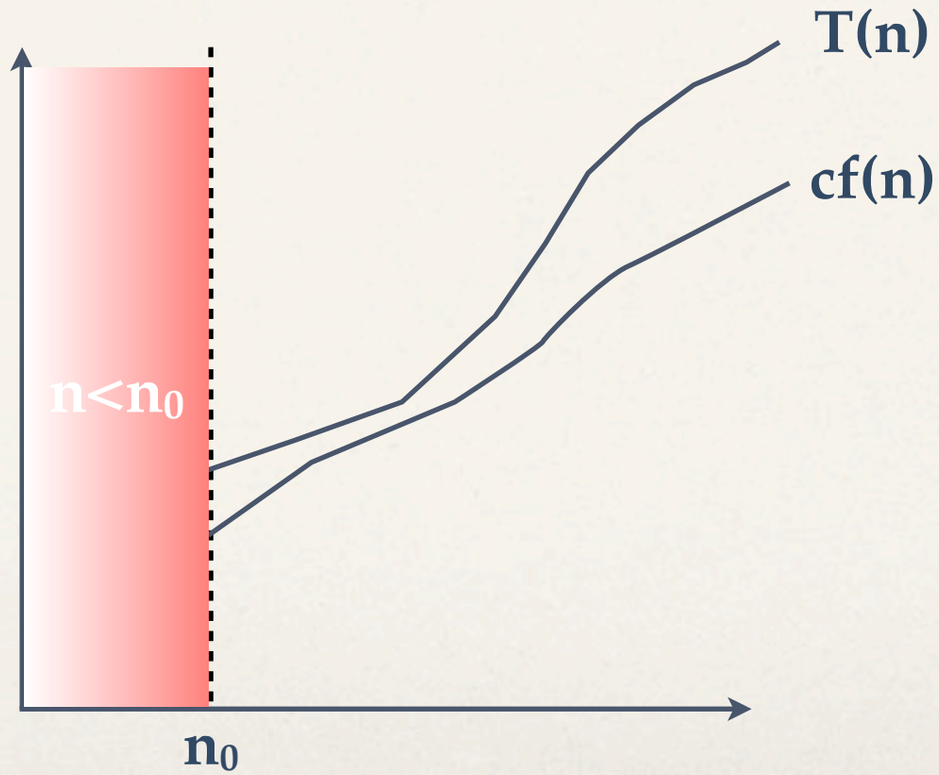
Asymptotic Notations

- Let T(n) and f(n) be two functions.
- We say that T(n) is O(f(n)) if there exist constants $c>0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.



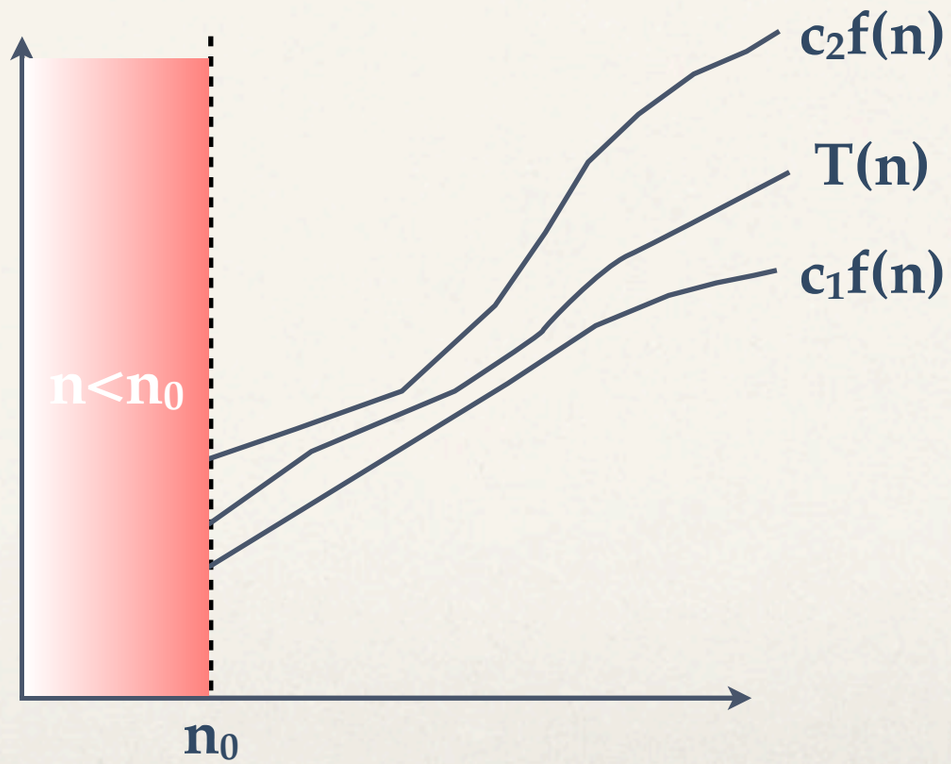
Asymptotic Notations

- Let T(n) and f(n) be two functions.
- We say that T(n) is $\Omega(f(n))$ if there exist constants $c>0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) \geq c \cdot f(n)$.



Asymptotic Notations

- Let T(n) and f(n) be two functions.
- We say that T(n) is $\Theta(f(n))$ if there exist constants $c_1>0$, $c_2>0$, and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$.



Asymptotic Notations

$T(n)$	$f(n)$	$T(n) \stackrel{?}{=} O(f(n))$	$T(n) \stackrel{?}{=} \Omega(f(n))$	$T(n) \stackrel{?}{=} \Theta(f(n))$
10^{1000}	n			
10^{1000}	1			
$(4n^2 - 12n + 9)/(2n - 3)$	n^2			
$(4n^2 - 12n + 9)/(2n - 3)$	n			
$17n - 12$	n			
$n(n+1)/2$	n			
$n(n+1)/2$	n^2			
$n(n+1)/2$	n^3			
$n^2 \log n$	n^2			
$n^2 \log n$	n^3			
$3n^2 + 8n \log n$	n^2			

Asymptotic Notations

- Theorem: If $T_1(n)=O(f_1(n))$ and $T_2(n)=O(f_2(n))$ then $T_1(n)+T_2(n)=O(\max\{f_1(n),f_2(n)\})$.
- Examples:
  - $n + n^2 = O(n^2)$
  - $\log n + n = O(n)$
  - $1 + n + n^2 + 10^6*n^3 + 2*n^4 = O(n^4)$
  - ...
- Proof?

Asymptotic Analysis of Algorithms

Algorithm 3: LinearSearch.

Input: An array $A[0 \ldots n-1]$ of integers, and an integer x.
Output: The index of the first element of A that matches x, or -1 if there are no matching elements.

```
1 i ← 0;
2 while i < n and A[i] ≠ x do
3     i ← i + 1;
4 if i ≥ n then
5     i ← −1;
6 return i
```

Algorithm 4: MatrixMultiplication.

Input: Two matrices $A[0 \ldots n-1, 0 \ldots k-1]$ and $B[0 \ldots k-1, 0 \ldots m-1]$.
Output: Matrix $C = AB$.

```
1 for i ← 0 to n − 1 do
2     for j ← 0 to m − 1 do
3         C[i, j] ← 0;
4         for l ← 0 to k − 1 do
5             C[i, j] ← C[i, j] + A[i, l] · B[l, j];
6 return C
```

Algorithm 1: IsBipartite.

Input: Undirected graph $g = (V, E)$.
Output: *True* if g is bipartite, and *False* otherwise.

```
1  foreach Non-empty subset V1 ⊂ V do
2      V2 ← V \ V1;
3      bipartite ← True;
4      foreach Edge {u, v} ∈ E do
5          if {u, v} ⊆ V1 or {u, v} ⊆ V2 then
6              bipartite ← False;
7              Break;
8      if bipartite = True then
9          return True;
10 return False;
```

Asymptotic Analysis of Algorithms

- Analyzing the computational complexity of many algorithms is not an easy task, and especially so for recursive algorithms.

- We'll see many examples of complexity analysis throughout the semester, and develop an intuition for doing it through the techniques used in these examples.

Empirical Analysis of Algorithms
A General Plan

1. Decide on the efficiency metric M to be measured and the measurement unit (an operation's count vs. a time unit).

2. Decide on characteristics of the input sample (its range, size, and so on).

3. Prepare a program implementing the algorithm for the experimentation.

4. Generate a sample of inputs.

5. Run the algorithm on the sample inputs and record the data observed.

6. Analyze the data obtained.

Empirical Analysis of Algorithms: Issues

- A system's time is typically not very accurate, and you may get somewhat different results on repeated runs of the same program on the same inputs.
  - An obvious remedy is to make a large number of such measurements and take their average.
- If the computer is very fast, the program is very fast, the input is so small, the number precision is limited, etc., the running time may be reported as zero.
  - One possible remedy is to run the program a large number of times and compute the time for all those runs, rather than each single one.
- The reported time may include the time spent by the CPU on other programs.

Empirical Analysis of Algorithms: Issues

- You need to decide on a sample of inputs for the experiment.

- Often, the goal is to use a sample representing a “typical” input; so, you need to understand what a typical input is.

Empirical Analysis of Algorithms: Issues

- The principal strength of the mathematical analysis is its independence of the specific inputs; its principal weakness is its poor reflection of the *exact* running time.

- The principal strength of the empirical analysis lies in its applicability to any algorithm, but its results can depend on the particular sample of instances and the computer used in the experiment (other than it comes "after the fact").

- Remember: Empirical analysis tells about a specific implementation of the abstract algorithm, so you have to be careful about what conclusions you can draw about an algorithm's efficiency from the empirical analysis of a specific implementation. (For deciding on the time it takes to check membership in a set, did you implement your set as a list, a hash table, ...?)