CSC 212
Data Structures and Abstractions
Fall 2015

Lecture 01: Analysis of Algorithms
Algorithm

“Any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.”

[Cormen et al., Introduction to Algorithms, 3rd. Ed.]
Sorting Problem

Input: \( \langle a_1, a_2, \ldots, a_n \rangle \)
Sorting Problem

Input: \( \langle a_1, a_2, \ldots, a_n \rangle \)

Output: \( \langle a'_1, a'_2, \ldots, a'_n \rangle \)

Permutation
Sorting Problem

Input: \( \langle a_1, a_2, \ldots, a_n \rangle \)

Output: \( \langle a'_1, a'_2, \ldots, a'_n \rangle \)

such that \( a'_1 \leq a'_2 \leq \cdots \leq a'_n \)
Understand and Model the Problem
Understand and Model the Problem

Design a Solution
Understand and Model the Problem

Design a Solution

Correct? Efficient? Scalable?
Understand and Model the Problem

Design a Solution

Correct? Efficient? Scalable?

why not?
Understand and Model the Problem

Design a Solution

Correct? Efficient? Scalable?

why not?
\[ F_0 = 0 \]
\[ F_1 = 1 \]
\[ F_n = F_{n-1} + F_{n-2} \]

0 1 1 2 3 5 8 13 21 34 ...
typedef unsigned long int ul_int;
ul_int fib_rec(ul_int n) {
    if (n == 0 || n == 1) {
        return n;
    }
    return fib_rec(n-2) + fib_rec(n-1);
}
ul_int fib_iter(ul_int n) {
  ul_int f;
  ul_int fib[] = {0, 1};

  if (n == 0 || n == 1) {
    return n;
  }

  for (int i = 2 ; i <= n ; i++ ) {
    f = fib[0] + fib[1];
    fib[0] = fib[1];
    fib[1] = f;
  }

  return f;
}
void time_func(ul_int (*f_ptr)(ul_int), ul_int n, char *name) {
    clock_t tic = clock();
    ul_int tt = (*f_ptr)(n);
    clock_t toc = clock();

    double elapsed = (double) (toc-tic) / CLOCKS_PER_SEC;
    printf("%s:	Output value: %ld\t%f seconds\n", name, tt, elapsed);
}

int main(int argc, char **argv) {
    // get argument from command line
    ul_int n = (ul_int) atoi(argv[1]);
    // measure and print time for each call
    time_func(&fib_iter, n, "Iter");
    time_func(&fib_rec, n, "Rec");
}
Recursive
Recursive
Recursive
Recursive
Recursive
Recursive
Recursive
Recursive
Recursive
Recursive
Recursive
Recursive
Recursive
Recursive
Why Analysis of Algorithms?

Classify algorithms/problems
Predict performance/resources
Provide guarantees
Understand underlying principles
Technical Interviews

Arrays and Strings
Linked Lists
Stacks and Queues
Analysis of Algorithms
Sorting and Searching
Dynamic Programming
Trees and Graphs
Bit Manipulation
Recursion
Scalability and Memory
C/C++
Java

[McDowell, Cracking the Code Interview]
Algorithm Analysis

Running Time       Memory/Space
T(n)
Running Time

**Empirical Analysis**
- Run algorithm
- Measure actual time

**Mathematical Model**
- Analyze Algorithm
- Develop Model
Empirical Analysis

- Implement algorithm
- Run on different input sizes
- Record actual running times
- Calculate hypothesis
- Predict and validate
Empirical Analysis

Variations in HW, SW, and OS affect analysis
Implementation details play a role
Difficult for algorithm comparisons
Mathematical Model

High-level analysis — no need to implement
Mathematical Model

High-level analysis — no need to implement

Independent of HW/SW
Mathematical Model

High-level analysis — no need to implement

Independent of HW/SW

Based on counts of \textit{elementary} operations

additions, multiplications, comparisons, etc.
Asymptotic Notation

Ignore machine dependent constants

Interest in execution times with large inputs

growth of $T(n)$ as $n \to \infty$
$T(n) = O(f(n))$ if there are positive constants $c$ and $n_0$ such that $T(n) \leq c \cdot f(n)$ when $n \geq n_0$
\[ T(n) = O(f(n)) \] if there are positive constants \( c \) and \( n_0 \) such that
\[ T(n) \leq c \cdot f(n) \] when \( n \geq n_0 \).
$T(n) = O(f(n))$ if there are positive constants $c$ and $n_0$ such that $T(n) \leq c \cdot f(n)$ when $n \geq n_0$
Big Omega

\[ T(n) = \Omega(f(n)) \] if there are positive constants \( c \) and \( n_0 \) such that 
\[ T(n) \geq c \cdot f(n) \] when \( n \geq n_0 \)
$T(n) = \Omega(f(n))$ if there are positive constants $c$ and $n_0$ such that $T(n) \geq c \cdot f(n)$ when $n \geq n_0$
Big Omega

\[ T(n) = \Omega(f(n)) \text{ if there are positive constants } c \text{ and } n_0 \text{ such that } T(n) \geq c \cdot f(n) \text{ when } n \geq n_0 \]
$T(n) = \Omega(f(n))$ if there are positive constants $c$ and $n_0$ such that $T(n) \geq c \cdot f(n)$ when $n \geq n_0$
Big Theta

\[ T(n) = \Theta(f(n)) \text{ if and only if } \]
\[ T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n)) \]
Big Theta

\[ T(n) = \Theta(f(n)) \text{ if and only if } T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n)) \]
Big Theta

\[ T(n) = \Theta(f(n)) \text{ if and only if } T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n)) \]
Big Theta

\[ T(n) = \Theta(f(n)) \] if and only if \[ T(n) = O(f(n)) \] and \[ T(n) = \Omega(f(n)) \]
Big Theta

\[ T(n) = \Theta(f(n)) \text{ if and only if } T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n)) \]
Big O

Given $T(n) = 3n^2 + 100n + 5$, then $T(n) = O(n^2)$

Proof:

$c = 4, n_0 = 100.05$

for $n \geq 100.05$, then $3n^2 + 100n + 5 \leq 4n^2$
Big O

Given $T(n) = 3n^2 + 100n + 5$, then $T(n) = O(n^2)$

Proof:

$c = 4$, $n_0 = 100.05$

for $n \geq 100.05$, then $3n^2 + 100n + 5 \leq 4n^2$
Given $T(n) = 3n^2 + 100n + 5$, then $T(n) = O(n^2)$

Proof:

$c = 4$, $n_0 = 100.05$

for $n \geq 100.05$, then $3n^2 + 100n + 5 \leq 4n^2$

$T(n)$ is also $O(n^3)$, $O(2^n)$, … but smallest function is preferred
Given $T(n) = 3n^2 + 100n + 5$, then $T(n) = O(n^2)$

Proof:

$c = 4$, $n_0 = 100.05$
for $n \geq 100.05$, then $3n^2 + 100n + 5 \leq 4n^2$

$T(n)$ is also $O(n^3), O(2^n), ...$ but smallest function is preferred

Engineer way …

ignore constants and drop lower order terms
Examples

$10^2 + 3000n + 10$
$21 \log n$
$500 \log n + n^4$
$\sqrt{n} + \log n^{50}$
$4^n + n^{5000}$
$3000n^3 + 3n^{3.5}$
$2^5 + n!$
Order of Growth

\[
T(n) = \Theta(1)
\]
\[
T(n) = \Theta(\log n)
\]
\[
T(n) = \Theta(n)
\]
\[
T(n) = \Theta(n \log n)
\]
\[
T(n) = \Theta(n^2)
\]
\[
T(n) = \Theta(n^3)
\]
\[
T(n) = \Theta(2^n)
\]
Asymptotic Performance

For large values of $n$, an $O(n^2)$ algorithm always beats an $O(n^3)$ algorithm.

In practice, we shouldn’t completely ignore asymptotically slower algorithms.
for (int i = 0 ; i < n ; i ++) {
    // O(1) operations
}

for (int i = 0 ; i < n ; i ++) {
    for (int j = 0 ; j < n ; j ++) {
        // O(1) operations
    }
}

for (int i = 0 ; i < n ; i ++) {
    for (int j = 0 ; j < n*n ; j ++) {
        // O(1) operations
    }
}
\begin{verbatim}
for (int i = 0 ; i < n ; i ++) {
    for (int j = 0 ; j < i ; j ++) {
        // O(1) operations
    }
}

for (int i = 0 ; i < n ; i ++) {
    for (int j = 0 ; j < n ; j ++) {
        for (int k = 0 ; k < n ; k ++) {
            // O(1) operations
        }
    }
}

for (int i = 0 ; i < n ; i ++) {
    for (int j = 0 ; j < i*i ; j ++) {
        for (int k = 0 ; k < j ; k ++) {
            // O(1) operations
        }
    }
}
\end{verbatim}
What to Analyze?

Bounds for algorithms
What to Analyze?

Input Instances

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>25</td>
<td>100</td>
<td>50</td>
<td>100</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

Bounds for algorithms

Worst-case
What to Analyze?

Input Instances

A  B  C  D  E  F

Time

0 s.  25 s.  50 s.  75 s.  100 s.

Worst-case

Best-case

Bounds for algorithms
What to Analyze?

Input Instances

A  B  C  D  E  F

Time

0 s.  25 s.  50 s.  75 s.  100 s.

Worst-case

Average-case

Best-case

Bounds for algorithms