Analysis of Algorithms

Reading Assignment
Chapters 3

Motivation

- Even though we seem to have an abundance of CPU cycles and memory units at our finger tips, program speed and memory use matters when processing large amounts of data
- The running time of a program depends
  - Algorithms and data structures
  - Programming language
  - Compiler/interpreter
  - Operating system
  - Processor and memory
- In algorithm analysis we are primarily interested figuring out how well an algorithm performs with respect to time and space usage regardless of all the other influences
- In other words, we fix the environment within which a program runs and try to analyze the running time independently of the environment
- The goal is to compare the time and space complexity of different algorithms for a given input size

Objectives

- Estimate the running time (function) for a given algorithm
- Appreciate how the running time (function) varies with input size
- Find a measure to compare the quality of algorithms which perform the same task
- Appreciate different complexity classes
- Comparing different growth functions
- Measure running time in terms of basic operations
- Plot and compare growth curves
- Understand Big Oh notation
- Compute and compare Big Oh running times

Basic units

- How shall we assess and quantify the running time of a program?
  - I/O, read/writes fetches/stores
  - Comparisons (for sorting and searching)
  - Assignments
  - Loops
  - Program size/amount of memory
  - Number of calculations
  - Static versus dynamic memory
  - Add, sub, mul, div, sin, cos
- Search and sorting algorithms
  - Comparisons
- Graphics algorithms
  - sin, cos
- CSc 115/160
  - Comparisons and assignments

Running time of an algorithm

- Definition
  - The running time of an algorithm is a function of the size of the input data with units such as comparisons, assignments, arithmetic operations, trigonometric operations. The running time is denoted by T(n) where n is the size of the input to the algorithm.
- Examples of running times
  - $T_1(n) = c_0 n^2$
  - $T_2(n) = c_1 n^3 + c_2 n^2 + c_3 n + c_4$
  - $T_3(n) = c_4 n \lg n + c^4$
  - $T_4(n) = c_5 2^n$

An example

- Example

```
a = 3*n;
cnt = 1;
while (a > 0) {
   a = a - 1;
   cnt = cnt + 1;
}
```

- Basic units
  - Assignments
  - Comparisions
- Analysis
  - T(n) = 2 + while loop
  - = 2 + x(units in loop) + 1 (x = # of iterations)
  - = 2 + x(3) + 1
  - = 2 +3(3n) + 1
  - = 3 + 9n

Linear search

```
int linearSearch(int[] a, int x) {
  int k = 0;
  while (k<a.length) {
    if (a[k] == x) return k;
    k = k + 1;
  }
}
```

- Linear search over **unsorted** array of integers
- Units: comparisons, assignments, no other operations
- Size of the problems
  - n = a.length (size of array)
- Worst-case running time
  - x is not found or found at the last position

T(n) = initialize + while loop
= 1 + while loop
= 1 + x(3) + 1 (x = n)
= 1 + 3n + 1
= 2 + 3n linear function
$T(n) = c_1 n + c_2$ **linear algorithm**

Worst case: T(n) ~ n
Best case: T(n) ~ 1
Expected case: T(n) = n/2

Binary search

```
int binarySearch(int[] a, int x) {
  int l = 0;
  int r = a.length -1;
  while (l<=r) {
    int m = (l+r)/2;
    if (a[m] == x) return m;
    else if (x < a[m]) r = m-1;
    else l = m+1;
  }
  return -1;
}
```

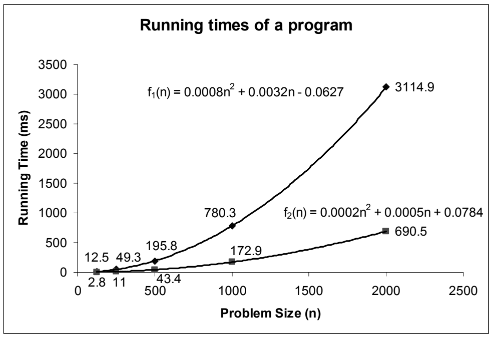
- Binary search over **sorted** array of integers
- Units: comparisons, assignments
- Size of the problems
  - n = a.length (size of array)
- Phone book look up
- Worst case: not found

T(n) = initialize + while loop
= 2 + while loop
= 2 + x(5) + 1 (x = log n)
= 2 + 5log n + 1
= 3 + 5log n
$T(n) = c_1 \log n + c_2$

Logarithmic function

Worst case: T(n) ~ log n
Best case: T(n) ~ 1
Expected case: T(n) = log n

Measurement Example



Methodology Requirements

- We want a methodology for analyzing the running times of algorithms that
  - Takes into account all possible inputs
  - Allows us to evaluate the relative efficiency of any two algorithms in a way this is independent from the hardware and software environment
  - Can be performed by studying a high-level description of the algorithm without actually implementing it or running experiments on it

Another linear algorithm: finding maximum

- **High-level description of an algorithm**
- **Pseudo code**

Algorithm arrayMax(*A*,*n*):
Input: An array A storing n >= 1 integers
Output: The maximum element in A.

currentMax <-- A[0]
for *i* <-- 1 **to** *n* - 1 **do**
if *currentMax* A[*i*] **then** *currentMax* <-- A[*i*]
return *currentMax*

- **Worst case: T(n) ~ n**
- **Best case: T(n) ~ 1**
- **Expected case: T(n) = n/2**

Asymptotic time complexity

- Fundamental measure for the performance of an algorithm
- Study asymptotic growth rates
- Asymptotic
  - Not interested in constants
  - Not interested in small inputs
  - Pure growth rate of the function
  - It essentially removes the "noise" from the running time
- Three sets of functions
- Big Omega $\Omega(g)$
  - Functions that grow at least as fast a g
- Big Theta $\Theta(g)$
  - Functions that grow at the same rate as g
- Big Oh $O(g)$
  - Functions that grow no faster than g

Formal Definition of Big-O Notation

- **Definition**
  - Let f(n) and g(n) be functions mapping nonnegative integers to real numbers. We say that f(n) is O(g(n)) if there is a real constant c > 0 and an integer constant $n_0 \geq 1$ such that f(n) ≤ c·g(n) for every integer $n \geq n_0$.
- We say
  - f(n) is *order* g(n)
  - f(n) is *Big-Oh* of g(n)
- Visually, this says that the f(n) curve must eventually fit under the c·g(n) curve.

Big-O Notation

- We simplify the function by:
  - ignoring all constant coefficients
  - ignoring all but the *dominant term*
    - the dominant term is the one that grows fastest when *n* grows

f(*n*)	O(f(*n*))
$0.3n^2 + 20n + 512$	$O(n^2)$
$0.0001n^4 + 10000n^2$	$O(n^4)$
$3^n + n^2$	$O(3^n)$
$10^n - 5^n + 3^n$	$O(10^n)$
$42\log_2 n$	$O(\log_2 n)$
$7n\log_{10} n + 2n - 12$	$O(n\log_{10} n)$
42	O(1)

Complexity Classes

- When determining the Big-Oh time of a problem, we try to:
  - make the bound as tight as possible
  - make the function as simple as possible
- In practice, this leads to only a handful of important Big-Oh expressions

From least to most complex

Complexity Class	O-notation
Constant	O(1)
log log *n*	O(log log *n*)
Logarithmic	O(log *n*)
Linear	O(*n*)
n log *n*	O(*n* log *n*)
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(2^n)$, $O(3^n)$, …

Famous algorithms

Algorithm	Big O-notation
Hash search	O(1)
Binary search, tree search	O(log *n*)
Linear search, list and tree traversals	O(*n*)
Sorting, Heapsort	O(*n* log *n*)
Bubble sort, insertion sort	$O(n^2)$
Matrix multiplication	$O(n^3)$
Optimal graph coloring	$O(2^n)$, $O(3^n)$, …

Running Time Examples

- An algorithm takes f(n) microseconds (μs) to run

f(n) \ n	2 (2^1)	16 (2^4)	256 (2^8)	1024 (2^{10})	1048576 (2^{20})
1	1 μs	1 μs	1 μs	1 μs	1 μs
$\log_2 n$	1 μs	4 μs	8 μs	10 μs	20 μs
n	2 μs	16 μs	256 μs	1.02 ms	1.05 s
$n \log_2 n$	2 μs	64 μs	2.05 ms	10.2 ms	21 s
n^2	4 μs	256 μs	65.5 ms	1.05 s	1.8 wks
n^3	8 μs	4.1 ms	16.8 s	17.9 min	36559 yrs
2^n	4 μs	65.5 msec	3.7×10^{63} yrs	5.7×10^{294} yrs	2.1×10^{315639} yrs

Estimated lifetime of the sun: only 5×10^9 yrs!

1 μs = 10^{-6} s	1 s = one second	1 wk = 604800 s
1 ms = 10^{-3} s	1 min = 60 s	1 yr = 31557600 s

Big-O Caveats

- Comparisons based on Big-O notation apply only to large problem sizes
  - "large" is an arbitrary term
  - for "small" problem sizes, consider the specific circumstances the algorithm will be running in
    - those constant coefficients we so casually discarded start to matter
  - run experiments on your platform, with your data, to determine the best algorithm (*measurement and tuning*)
- Carefully check whether your data fits the average case
  - otherwise, the worst case time could be important
  - in real-time situations, the worst case time might be crucial
  - sometimes you can easily mould the data to fit an algorithm's best case