CSCI 333
Data Structures

Chapter 3

15 September, 2003

Notes with the dark blue background were prepared by the textbook author

Clifford A. Shaffer
Department of Computer Science
Virginia Tech

Algorithm Efficiency

There are often many approaches (algorithms) to solve a problem. How do we choose between them?

At the heart of computer program design are two (sometimes conflicting) goals.

1. To design an algorithm that is easy to understand, code, debug.
2. To design an algorithm that makes efficient use of the computer's resources.

Algorithm Efficiency (cont)

Goal (1) is the concern of Software Engineering.

Goal (2) is the concern of data structures and algorithm analysis.

When goal (2) is important, how do we measure an algorithm's cost?

How to Measure Efficiency?

1. Empirical comparison (run programs)
2. Asymptotic Algorithm Analysis

Critical resources:

Factors affecting running time:

For most algorithms, running time depends on "size" of the input.

Running time is expressed as **T**(n) for some function **T** on input size n.

Benchmarks

- Wall clock time
  - Depends on computer
    - processor, memory, compiler
- Often quoted in sales literature
  - SPECmark,
  - SPECrate,
  - TPC (databases)

Best, Worst, Average Cases

Not all inputs of a given size take the same time to run.

Sequential search for K in an array of n integers:

- Begin at first element in array and look at each element in turn until K is found

Best case:

Worst case:

Average case:

Which Analysis to Use?

While average time appears to be the fairest measure, it may be difficult to determine.

When is the worst case time important?

Faster Computer or Algorithm?

What happens when we buy a computer 10 times faster?

$\mathbf{T}(n)$	n	n'	Change	n'/n
$10n$	1,000	10,000	$n' = 10n$	10
$20n$	500	5,000	$n' = 10n$	10
$5n \log n$	250	1,842	$\sqrt{10}\, n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
2^n	13	16	$n' = n + 3$	-----

Asymptotic Analysis: Big-oh

Definition: For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in the set $O(f(n))$ if there exist two positive constants k and M such that $\mathbf{T}(n) <= k\, f(n)$ for all $n > M$.

Usage: The algorithm is in $O(n^2)$ in worst case.

Meaning: For all data sets big enough (i.e., $n>M$), the algorithm always executes in less than $kf(n)$ steps in worst case.

Big-oh Notation (cont)

Big-oh notation indicates an upper bound.

Example: If $\mathbf{T}(n) = 3n^2$ then $\mathbf{T}(n)$ is in $O(n^2)$.

Always try to give the tightest upper bound: While $\mathbf{T}(n) = 3n^2$ is in $O(n^3)$, we prefer $O(n^2)$.

Big-Oh Examples

Example 1: Finding value X in an array (average cost).

$\mathbf{T}(n) = 350/2\, n + 1000$

For all values of $n > 100$,

$350/2\ n + 1000 <= 500\, n$.

Therefore, by the definition, $\mathbf{T}(n)$ is in $O(n)$ for $M = 100$ and $k = 500$.

Big-Oh Examples

Example 2: $\mathbf{T}(n) = 5\ n^2 + 1000\ n$ in average case.

$5\ n^2 + 1000\ n <= 100\ n^2$ for all $n > 100$.

Therefore, $\mathbf{T}(n)$ is in $O(n^2)$ by the definition.

Example 3: $\mathbf{T}(n) = c$. We say this is in $O(1)$.

Big-Omega

Definition: For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in the set $\Omega(g(n))$ if there exist two positive constants k and M such that $\mathbf{T}(n) >= kg(n)$ for all $n > M$.

Meaning: For all data sets big enough (i.e., $n > M$), the algorithm always executes in more than $kg(n)$ steps.

Lower bound.

Big-Omega Example

$\mathbf{T}(n) = 100\ n^2 + 100{,}000\ n$.

$100\ n^2 + 100{,}000\ n >= 10\ n^2$ for all $n > 10$.

Therefore, $\mathbf{T}(n)$ is in $\Omega(n^2)$ by the definition.

We want the greatest lower bound.

Theta Notation

When big-Oh and Ω meet, we indicate this by using Θ (big-Theta) notation.

Definition: An algorithm is said to be $\Theta(h(n))$ if it is in $O(h(n))$ and it is in $\Omega(h(n))$.

A Common Misunderstanding

Confusing worst case with upper bound.

Upper bound refers to a growth rate.

Worst case refers to the worst input from among the choices for possible inputs of a given size.

Simplifying Rules

1. If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.
2. If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$.
3. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $(f_1 + f_2)(n)$ is in $O(\max(g_1(n), g_2(n)))$.
4. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$ then $f_1(n)f_2(n)$ is in $O(g_1(n)g_2(n))$.

Running Time Examples (1)

Example 1: `a = b;`

This assignment takes constant time, so it is $\Theta(1)$.

Example 2:

```
sum = 0;
for (i=1; i<=n; i++)
  sum += n;
```

Running Time Examples (2)

Example 3:

```
sum = 0;
for (j=1; j<=n; j++)
  for (i=1; i<=j; i++)
    sum++;
for (k=0; k<n; k++)
  A[k] = k;
```

Running Time Examples (3)

Example 4:

```
sum1 = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    sum1++;

sum2 = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=i; j++)
    sum2++;
```

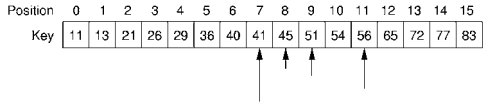
Running Time Examples (4)

Example 5:

```
sum1 = 0;
for (k=1; k<=n; k*=2)
  for (j=1; j<=n; j++)
    sum1++;

sum2 = 0;
for (k=1; k<=n; k*=2)
  for (j=1; j<=k; j++)
    sum2++;
```

Binary Search



How many elements are examined in worst case?

Binary Search

```
// Return position of element in sorted
// array of size n with value K.
int binary(int array[], int n, int K) {
  int l = -1;
  int r = n; // l, r are beyond array bounds
  while (l+1 != r) { // Stop when l, r meet
    int i = (l+r)/2; // Check middle
    if (K < array[i]) r = i;   // Left half
    if (K == array[i]) return i; // Found it
    if (K > array[i]) l = i;   // Right half
  }
  return n; // Search value not in array
}
```

Other Control Statements

`while` loop: Analyze like a `for` loop.

`if` statement: Take greater complexity of `then`/`else` clauses.

`switch` statement: Take complexity of most expensive case.

Subroutine call: Complexity of the subroutine.

Multiple Parameters

Compute the rank ordering for all C pixel values in a picture of P pixels.

```
for (i=0; i<C; i++)  // Initialize count
  count[i] = 0;
for (i=0; i<P; i++)  // Look at all pixels
  count[value(i)]++; // Increment count
sort(count);         // Sort pixel counts
```

If we use P as the measure, then time is $\Theta(P \log P)$.

More accurate is $\Theta(P + C \log C)$.

Space Bounds

Space bounds can also be analyzed with asymptotic complexity analysis.

Time: Algorithm
Space: Size of Data Structure

Space/Time Tradeoff Principle

One can often reduce time if one is willing to sacrifice space, or vice versa.

- Encoding or packing information
  Boolean flags
- Table lookup
  Factorials

Disk-based Space/Time Tradeoff Principle: The smaller you make the disk storage requirements, the faster your program will run.