Data Structures in C++

Chapter 4

Tim Budd

Oregon State University
Corvallis, Oregon
USA

Outline – Chapter 4

Analyzing Execution Time

Types of Analysis

- Question of termination (previous chapter)
- Wall-clock time (benchmarks)
- Asympototic Analysis

Why Benchmarks Are Not Good Characterizations

- Machine dependent
- Compiler dependent
- Input value dependent
- Programming environment dependent (machine load)

Better Characterizations using Less Information

Question: Why are dictionaries ordered?

Answer: So we can use a more efficient algorithm to look up words.

Why is it more efficient? To search an unordered list of n words takes, worst case, n comparisons. To search an ordered list requires, worst case, $\log n$ comparisons (using binary search, as we do with dictionaries).

Note that we ignore features such as how long it actually takes to do a comparison, as the exact figure is relatively unimportant.

Asymptotic Execution Time

Suppose $f(n)$ is a formula that describes the *exact* execution time of some algorithm for inputs of size n.

We say that the algorithm is $O(g(n))$ (read big-Oh of g of n) if there exists some constants c and n_0 (we need not know what these are, just that they exist), such that:

$$f(n) < c \times g(n), \text{ for all } n > n_0.$$

Searching a dictionary is $O(\log n)$, for example, while searching an unordered list is $O(n)$.

This chapter is devoted to showing how to characterize the asymptotic execution time behavior of algorithms.

Constant Time

We assume that most primitive operations (addition of integers, multiplication, subscripting of arrays) can be performed in a constant time.

We really don't care about the exact figure.

A sequence of constant time operations is still constant time.

```
Card::Card     (suits sv, int rv)
   // initialize a new Card using the
argument values
{
   rank = rv;
   suit = sv;
}
```

Time to do an `if` statement

The time to perform an if statment is determined by

- Time to perform comparison
- Maximum time to perform either if or else statement

```
int min (int a, int b)
    // return the smaller of two arguments
{
    int smallest;

    if (a < b)
        smallest = a;
    else
        smallest = b;
    return smallest;
}
```

This is still constant time.

Simple Loops

A loop that is performing a constant number of iterations is still constant.

```
Deck::Deck ( )
    // initialize a deck by creating all 52 cards
{
    topCard = 0;
    for (int i = 1; i <= 13; i++) {
        Card c1(diamond, i), c2(spade, i),
            c3(heart, i), c4(club, i);
        cards[topCard++] = c1;
        cards[topCard++] = c2;
        cards[topCard++] = c3;
        cards[topCard++] = c4;
        }
}
```

Loops that Depend upon Input

More interesting are loops where the number of executions will depend upon the input. The following is $O(n)$.

```
double minimum (double values [ ], unsigned n)
{
    assert(n > 1);
    double minValue = values [0];

    for (unsigned int i = 0; i < n; i++) {
        if (values[i] < minValue)
            minValue = values[i];
        }
    return minValue;
}
```

Non-obvious terminating condition

The number of iterations need not be quite so obvious. The following will loop, at most $O(\sqrt{n})$ times.

```
int isPrime (unsigned int n)
    // return true if the argument value
    // is prime and false otherwise
{
    for (unsigned int i = 2; i * i < n; i++) {
            // if i is a factor, then not prime
        if (0 == n % i)
            return 0;
        }
        // if we end loop without finding factor
        // then number must be prime
    return 1;
}
```

Nested Loops

When loops are nested, need to determine how many times the innermost statement is excuted.

Easy if loops are independent:

```
void matprod (unsigned int n,
   double & a[n][n], double & b[n][n], double &
c[n][n])
   // multiply the matrix a by b, yielding new
matrix c
{
   for (unsigned int i = 0; i < n; i++) {
      for (unsigned int j = 0; j < n; j++) {
         c [i, j] = 0.0;
         for (unsigned int k = 0; k < n; k++) {
            c[i, j] += a[i, k] * b[k, j];
            }

         }
      }
}
```

More Difficult Nested Loops

Can be more difficult if the running time of inner loops depends upon running time of outer loops.

```
void bubbleSort (double & v[ ], unsigned int n)
{
    for (unsigned int i = n − 1; i > 0; i−−) {
            // move large values to the top
        for (unsigned int j = 0; j < i; j++) {
                // if out of order
            if (v[j] < v[j+1]) {
                    // then swap
                double temp = v[j];
                v[j] = v[j + 1];
                v[j + 1] = temp;
                }
            }
        }
}
```

Try Simulating the Execution for a Few Values

Try simulating the execution on the vector

4 5 3 2

Come up with a conjecture

The first time the outermost loop executes we perform $n-1$ comparisons.
The second time we perform $n-2$ comparisons.
The third time we perform $n-3$ compariosn.
So on until the last time, when we perform 1 comparison.
So the number of times we execute the loop is the sum

$$(n-1)+(n-2)+(n-3)+...+1$$

But what is this amount?
A bit of fiddling might lead us to the formula $\frac{(n-1)n}{2}$. But can be prove this?

Mathematical Induction

A powerful technique we can use in such cases is the method of *mathematical induction.*

- Find (somehow) a conjecture that can be tied to an integer value N.
- Prove as a **base case** that the conjecture holds for N=0, possibly a few more.
- Prove the **induction step** that *if* the result holds for value N, then it *must* also hold for value N+1.

Since it holds for N=0, it must also hold for N=1, and for N=2, and for N=3, and so on.

An Example Proof

Prove that the sum of values from 1 to N is $\frac{n(n+1)}{2}$.

Base case, try 0, 1, 2, 3.

Induction case. Assume it holds for values up to n, and prove for $n+1$.

Requires understanding how the hypothesis for n and $n+1$ are linked. and *reducing* the $n+1$ case to the size n situation.

Assume that the sum of values from 1 to n is $\frac{n(n+1)}{2}$.

Ask what is the sum of the values from 1 to $n+1$.

But this can be written as $(1+2+...+n)+(n+1)$.

Our *induction hypothesis* tell us that we can substitute $\frac{n(n+1)}{2}$ for the first term. The resulting expression is $\frac{n(n+1)}{2}+(n+1)$, or $\frac{n(n+1)}{2}+\frac{2(n+1)}{2}$, which simplifies to $\frac{(n+1)(n+2)}{2}$.

Since this matches our induction hypothesis for $n+1$, we are done.

Another Example

Prove that the sum of powers of 2 is one less than the next higher power.

Relationship between Induction and Recursion

There is a close relationship between mathematical induction and recursion.

- Both begin by identifing *base cases*, that are handled using some other means.
- Both proceed by showing how a large problem can be reduced to a slightly smaller problem *of the same form.*
- The analysis then proceeds by showing first that the base cases are correct, and then *if* the induction formula (or recursive function call) is correct, *then* the larger expression must be correct.

While Loops

Analysis of while loops is similar to for loops.

```
void insertionSort (double v [], unsigned int n)
  // exchange the values in the vector v
  // so they appear in ascending order
{
  for (unsigned int i = 1; i < n; i++) {

    // move element v[i] into place
    unsigned int j = i - 1;
    while (j >= 0 && data[j+1] < data[j]) {
        // swap element
      double temp = v[j];
      v[j] = v[j+1];
      v[j+1] = temp;
        // decrement j
      j = j - 1;
      }
    }
}
```

Simulation of Insertion Sort

Simulate insertion sort on the vector

$$2\ 6\ 4\ 3\ 1$$

Worst Case, Best Case, Average Case

Normally we are interested in describing the worst possible case behavior. What is this for insertion sort? What sort of values cause this behavior? What is the best case behavior? What sort of values cause this behavior?

Sometimes we want to discuss the *average case* behavior, but usually the mathematics of this is very involved.

Another While Loop Algorithm

What can we say about the behavior of the following?

```
unsigned int gcd (unsigned int n, unsigned int m)
    // compute the greatest common divisor
    // of two positive integer values
{
    assert (n > 0 && m > 0);

    while (m != n) {
        if (n > m)
            n = n - m;
        else
            m = m - n;
        }
    return n;
}
```

Binary Search

How about the behavior of the following?

```
unsigned int binarySearch (double v [], unsigned
int n, double value)
{
    unsigned int low = 0;
    unsigned int high = n;
        // repeatedly reduce the area of search
        // until it is just one value
    while (low < high) {
        unsigned mid = (low + high) / 2;
        if (data[mid] < value)
            low = mid + 1;
        else
            high = mid;
        }

        // return the lower value
    return low;
}
```

What is a log?

To the mathematician: $\log_e a = \int_1^a \frac{1}{x} dx$

To a computer scientist:

The log (base n) of a positive value x is *approximately* equal to the number of times that x can be divided by n.

The log (base 2) of a positive value x is *approximately* equal to the number of times that x can be divided in half.

Lots of algorithms work by splitting things in half, so logs come up in many discussions.

Algorithms that Use Algorithms

Algorithms that use algorithms (or functions that use functions), the running time of the function being called must be taken into consideration.

```
void printPrimes (unsigned int n)
    // print numbers between 1 and n
    // indicating which are prime
{
    for (unsigned int i = 2; i <= n; i++) {
        if (isPrime (i))
            cout << i << " is prime\n";
        else
            cout << i << " is not prime\n";
        }
}
```

Recursive Algorithms

Complexity of a recursive algorithms is the product of

- Amount of work done on any one level
- Number of recursive calls

```
void printUnsigned (unsigned int val)
    // print the character representation of
    // the unsigned argument value
{
    if (val < 10)
        printChar (digitChar(val));
    else {
            // print high order part
        printUnsigned (val / 10);
            // print last character
        printChar (digitChar(val % 10));
        }
}
```

Another Interesting Recursive Algorithm

An interesting recursive algorithm for computing a number raised to an integer power is based on the following observations:

$$x^{2n} = (x^2)^n$$
$$x^{2n+1} = (x^2)^n x$$

To compute x^{53} for example, the following sequence of multiplications takes place.
$x^{53} = x * (x^2)^{26}$, $x^{26} = (x^2)^{13}$, $x^{13} = x * x^6$,
$x^6 = (x^2)^3$, $x^3 = x * x^1$, $x^1 = x * x^0$.
Note that the ceiling of the log (base 2) of a number represents the number of times we can divide a number by two.

Algorithm Using These Ideas

Using these ideas we can create a $O(\log n)$ algorithm.

```
//
//   power - raise a double to
//        an integer exponent
//        assumes result does not overflow

double power(double base, unsigned int exponent)
{
    if (exponent == 0)
        return 1.0;
    else if (even(exponent))
        return power(base * base, exponent / 2);
    else
        return power(base * base, exponent / 2)
                * base;
}
```

Difference Between O(n) and O(log n)

The following table illustrates some of the differences in the rate of grown between an $O(n)$ algorithm and a $O(\log n)$ algorithm.

n	time linear alg	time log alg	mults log alg
10	3.2	3.2	6
20	5.8	3.8	7
30	8.5	4.0	9
40	11.1	4.5	8
50	13.7	4.6	9
60	16.4	4.6	10
70	19.0	5.2	10
80	21.6	5.1	9
90	24.2	5.3	11

Not All Problems have Fast Solutions

Remember Towers to Hanoi?

Tower Size	Number of Recursive Calls
T(1)	c
T(2)	two moves of size 1 towers, $2 \times T(1)$, or $2 \times c$
T(3)	two moves of size 2 towers, $2 \times T(2)$, or $2 \times 2 \times c$
T(4)	two moves of size 3 towers, $2 \times T(3)$, or $2 \times 2 \times 2 \times c$
...	
T(n)	two moves of size $n-1$ towers, $2^{n-1} \times c$

How long would it take to move 64 disks?

Rule for Summing Execution Time

When summing execution times, the dominant function is the only important one.

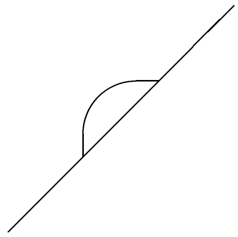
```
void makeIdentityMatrix ( double & m [n][n], unsigned int n)
    // initialize m as an identity matrix
{
        // first make matrix of all zeros
    for (unsigned int i = 0; i < n; i++) {
        for (unsigned int j = 0; j < n; j++) {
            m [i, j] = 0.0;
            }
        }

        // then place ones along diagonal
    for (i = 0; i < n; i++) {
        m [i, i] = 1.0;
        }
}
```

A Ranking of Execution Times

function	*common name*
$n!$	factorial
2^n	exponential
n^d, $d > 3$	polynomial
n^3	cubic
n^2	quadratic
$n\sqrt{n}$	
$n \log n$	
n	linear
$\sqrt{n}$	root–n
$\log n$	logrithmic
1	constant

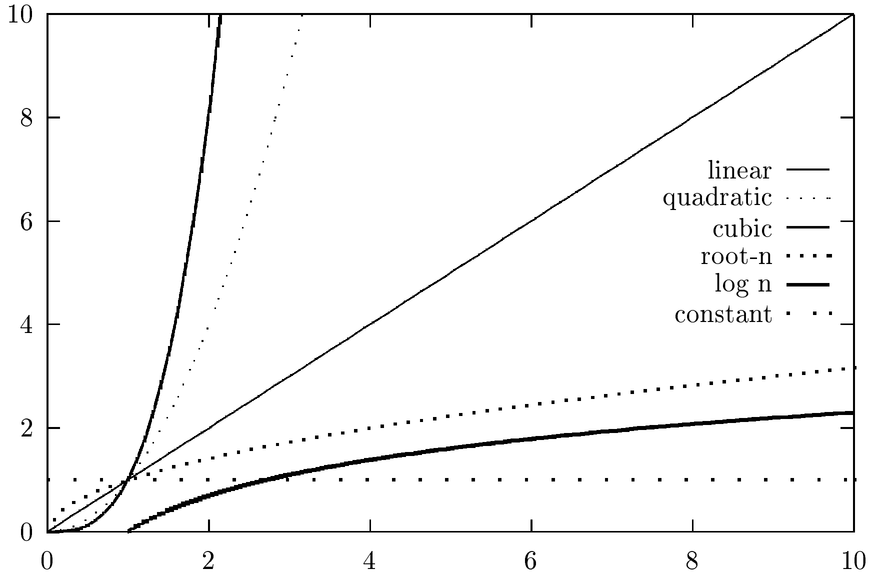
An Intuitive Argument

Here is an intuitive explation of why we can ignore all but the dominant functions. Why to large raindrops fall off of car windshields? Forces, gravity versus surface tension.

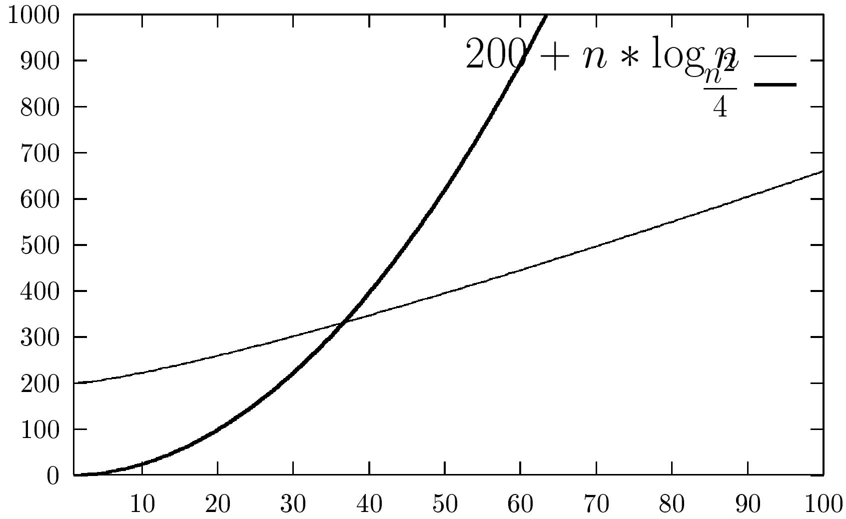


Gravity is $O(n^3)$, while surface tension is $O(n^2)$, where n is the diameter of the drop.

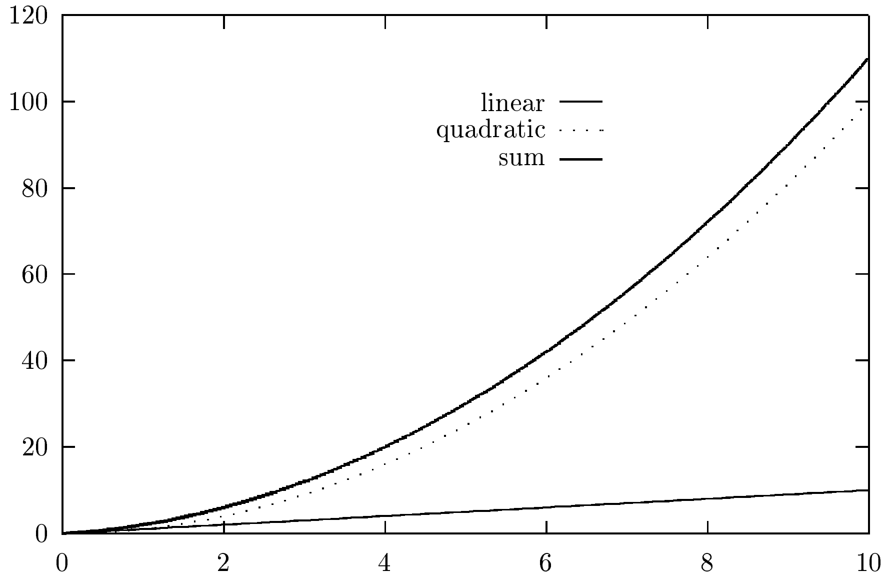
Every Function has a Characteristic Curve



Comparing n log n and n squared



When Adding Figures, Larger Eventually Dominates



Another Way to visual Time

Assume we can perform one operation every micro-second, or 10^6 operations per second. Assume a task that requires input of size 10^5. Here are some typical running times:

function	**running time**
2^n	more than a century
n^3	31.7 years
n^2	2.8 hours
$n\sqrt{n}$	31.6 seconds
$n \log n$	1.2 seconds
n	0.1 seconds
$\sqrt{n}$	3.2×10^{-4} seconds
$\log n$	1.2×10^{-5} seconds