Algorithms

CS 1025 Computer Science Fundamentals I

Stephen M. Watt

University of Western Ontario
Objectives

• Understand that there can be very different ways to solve the same problem.

• Understand that these ways have different benefits:
  – Simplicity to describe and understand
  – Difficulty to implement and maintain
  – Time cost and space cost to run
Algorithms vs Programs

• An algorithm describes how to do something.
  – It is a precise description.
  – It always works, or specifies exactly when it fails.
  – It terminates on all inputs.

• A program describes the steps to do something.
  – It may or may not give an algorithm.
  – Concerned with practicalities, such as the names of storage locations, whether a loop or recursion is used, ...
• Strictly speaking those are *imperative programs*.

• There are also *declarative programs* that describe *properties* of the answer.

• Then an *algorithm* in the programming language *implementation* provides the *steps* to do the computation.

• E.g. Lex, Prolog, VHDL, YACC
Problem 1

• We will look at a very simple problem and examine two algorithms to solve it.

• The problem we will look at is so simple, you have been doing it since you were about 10 years old.

• The problem is to compute $x$ to the power $n$. 
• Algorithm: Multiply $x$ by itself $n-1$ times.

• Program 1:

```java
double power(double x, int n) {
    double pow = 1;
    while (n-- > 0) pow *= x;
    return pow;
}
```

This is valid Java and valid C.

• Program 2:

```java
double power(double x, int n) {
    if (n == 0) return 1;
    return x * power(x, n-1);
}

// Or:  return n == 0 ? 1 : x*power(x, n-1);
```
• Q: If each product costs $1, how much does it cost to compute power(3.0, 100)?
How Much Does It Cost?

• Q: If each product costs $1, how much does it cost to compute power(3.0, 100) ?

A: $99.
How Much Does It Cost?

• Q: If each product costs $1, how much does it cost to compute $power(3.0, 100)$?

  A: $99.

• The cost is $n-1$ multiplications.

• That’s a lot. Can we do better?
Thinking About The Problem

• Are there any special values that can be computed faster?

• If so, we could compute one of those and then adjust the result...

\[ \text{power}(b, n) = b \times ... \times b \times \text{power}(b, \text{special}_n) \]
A Family of Special Values

• Consider $x^{(2\times k)}$.

• This is $(x^k)^2$. 
A Family of Special Values

• Consider $x^{(2\cdot k)}$.

• This is $(x^k)^2$.

• Can be computed with half the number of operations:

  \[
  t = \text{power}(x,k); \quad \text{pow} = t \cdot t
  \]
A 2\textsuperscript{nd} Algorithm: Repeated Squaring

- If \( n \) is even, then compute \( x^{(n/2)} \).
- If \( n \) is odd, then \( n-1 \) is even. Compute \( x \cdot x^{(n-1)} \).
- Stop at \( n = 0 \). \( x^0 = 1 \).
double power(double x, int n) {
    if (n == 0)
        return 1;
    else if (n % 2 == 0) {
        double t = power(x, n/2);
        return t*t;
    }
    else {
        double t = power(x, n/2);
        return x*t*t;
    }
}
Another Pgm for Repeated Squaring

double power(double x, int n) {
    double pow;
    if (n == 0)
        pow = 1;
    else {
        double t = power(x, n/2);
        pow = t*t;
        if (n % 2 == 1) pow *= x;
    }
    return pow;
}

• Advantages: No code duplication. Single exit point.
How Much Does It Cost?

• Worst case:
  – 2 multiplications at each step.
  – Each step divides the number by 2.
  – The number of steps is therefore $\log_2(n)$
    Need to round that up to the next integer.
  – Cost is proportional to $\log_2(n)$
Why log[2](n) ?

- Suppose we had a problem of size \( n = 1,000,000 \).
- Then solved it in terms of a pb of size 100,000.
- Then solved that in terms of a pb of size 10,000.
- Then solved that in terms of a pb of size 1,000.
- Then solved that in terms of a pb of size 100.
- Then solved that in terms of a pb of size 10.
- Then solved that in terms of a pb of size 1.

- At each stage we remove a zero.
- There are log[10](n) zeros.
- This is true whether this is 10 base ten or 10 base two.

- Splitting the problem size in half at each stage \( \Rightarrow \log[2](n) \)
A Third Algorithm  (Just in case you wondered)

- Use the fact that $x^n = \exp(\log(x^n)) = \exp(n \cdot \log(x))$

- Use standard *numerical approximation* techniques to compute $\exp(x)$ and $\log(x)$.

- This involves computing a quotient where both the numerator and denominator are polynomials of $x$. (Hermite-Pade approximants).

- These do not compute $\exp$ and $\log$, but are approximations.

- It gives an answer that is correct to needed # of digits (e.g. 17)

- Fixed cost. Same for all $n$. 