Week 2 Notes

Wednesday, January 18, 2023 15:24

Why study algorithms

Simple programming problems
- Has linear scan
- Cannot do much better
- The obvious program works

Algorithms Problems
- Not clear hot to do
- Simple ideas to slow
- Room for optimization

AI programing problems
- Typically hard to state what it is and how it should be implemented

Fibonacci
- Sequence of natural numbers
- n = 0, n = 1, n > 1
- 0, 1, Fn-1 + Fn-2
- Used as a mathematical model for rabbit population

Naïve Algorithm

```
If n <= 1{
    Return n
} Else{
    return fibRecur(n-1) +  fibRecur(n-2)
}
```

- Let T(n) denote the number of lines of code executed by fibRecurs(n)

Why is it slow?
- Computing the same thing over and over again

Efficient Algorithms

Imitate hand computation:

0, 1, 1, 2, 3, 5, 8

$$0 + 1 = 1$$
$$1 + 1 = 2$$
$$1 + 2 = 3$$
$$2 + 3 = 5$$
$$3 + 5 = 8$$

- Every second and third number can find the value after 1+1
    - Like 2+**3=5**

3+5=7

FibList(n)

- Create an arrayF[0…n]
  - F[0] = 0
  - F[1] = 1
  - For I from 2 to n
    - F[i] = F[i-1] + f[i-2]
  - Return F[n]

- T(n) = 2N+2. So T(100) = 202 lines of code
- Easy to compute

<u>GCD Problem</u>

GCDs

- Put fraction a/b in the simplest form
- Divide numerator and denominator by d, to get a/d over b/d
  - Example: GCD of 10, 4 is 2

- Important for Cyptography

Compute GCD

Input: Interger a, b>= 0

Output: gcd(a, b)

- Want to run with large numbers like gcd(37463, 47822)

Function NaiveGCD(a, b)

```
best ← 0
for d from 1 to a + b:
  if d|a and d|b:
    best ← d
return best
```

- Very slow code for anything over 20

<u>Euclidian Algorithm</u>

Function EuclidGCD(a, b)

```
if b = 0:
  return a
a′ ← the remainder when a is
   divided by b
return EuclidGCD(b, a′)
```

- Produces correct results by lemma

Example

$$
\begin{aligned}
&\gcd(3918848, 1653264)\\
=&\gcd(1653264, 612320)\\
=&\gcd(612320, 428624)\\
=&\gcd(428624, 183696)\\
=&\gcd(183696, 61232)\\
=&\gcd(61232, 0)\\
=&61232.
\end{aligned}
$$

Runtime
- Each Step reduces the size of numbers at about a factor of 2
- Take about log(ab) steps
- GCDs of 1-- digit numbers take about 600 steps
- Each step a single division

<u>Computing Runtime</u>
- Figuring out accurate runtime is a huge mess
- In practice, you might not even know some of these details

Asymptotic Notation
- All of these issues can multiply runtimes by (large) constant.

Approximate Runtimes

	n	$n \log n$	n^2	2^n
$n = 20$	1 sec	1 sec	1 sec	1 sec
$n = 50$	1 sec	1 sec	1 sec	13 day
$n = 10^2$	1 sec	1 sec	1 sec	$4 \cdot 10^{13}$ year
$n = 10^6$	1 sec	1 sec	17 min	
$n = 10^9$	1 sec	30 sec	30 year	
max n	10^9	$10^{7.5}$	$10^{4.5}$	30

Big- O

Big-O Notation

Definition

$f(n) = O(g(n))$ (f is Big-O of g) or $f \preceq g$ if there exist constants N and c so that for all $n \geq N$, $f(n) \leq c \cdot g(n)$.

- Using Big-O loses important information about constant multiples
- Big-O is only asymptotic

Overview:
The problems covered over these lectures are created by naïve algorithms that have a slow run time even tho ugh they might seem logical to use. When we sit down to program we don’t nececerarly think our code is going cause problems and be inefficient.
Going over the Fibonacci numbers we looked at examples of how a computer can slow down and take a long time to compute step by step code, we looked at Efficient Algorithms to combat this issue by looking at past results then trying to produce our own