COMP20003 Algorithms and Data Structures Algorithms

Nir Lipovetzky
Department of Computing and Information Systems
University of Melbourne
Semester 2

Outline of the first few lectures

- Algorithms: general
- This subject: details
- Algorithm efficiency
- Computational complexity
- Data structures
  - Basic data structures
  - Algorithms on basic data structures
  - Complexity analysis of algorithms on basic ds's

Revisit: What is an algorithm?

- A set of steps to accomplish a task.
- Computer algorithms must be:
  - Specific.
  - Correct.
  - Reasonably efficient.

Algorithms and Efficiency

- An algorithm must:
  - be accurate (to within the required tolerance).
  - compute in a "reasonable" amount of time.
- The most accurate algorithm in the world is useless if it takes forever to compute.
  - We are particularly interested in efficiency as the size of the input grows.
- **Why?**

Example algorithm: Compute Fibonacci numbers

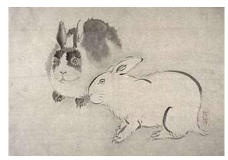
- $F_n = F_{n-1} + F_{n-2}$
  - $F_0 = 0$
  - $F_1 = 1$

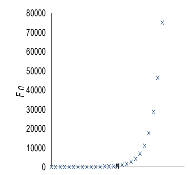
n	F(n-1)	F(n-2)	F(n)
0	-	-	0
1	-	-	1
2	1	0	1
3	1	1	2
4	2	1	3
5	3	2	5
6			8
7	8	5	13
8	13	8	21

- 0,1,1,2,3,5,8,13,21,34,55….

Fibonacci numbers

- Fibonacci numbers grow very quickly:
  - $F_{10} = 55$
  - $F_{15} = 610$
  - $F_{20} = 6765$

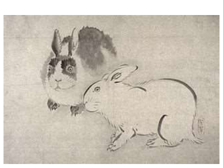




Painting by Kobi, 19th c, public domain

Fibonacci numbers

- The original problem that Fibonacci was investigating (1202):
  - How fast can rabbits breed under ideal circumstances?
  - http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibnat.html#Rabbits



Painting by Kobi, 19th c, public domain

How to compute Fibonacci numbers?

- Given the definition of Fibonacci numbers:
  - $F_n = F_{n-1} + F_{n-2}$
    - $F_0 = 0$
    - $F_1 = 1$
- Does this suggest an easy way to calculate F_n?

Computing Fibonacci Numbers: Scaffolding

```
#include<stdio.h>
#define DEBUG 1

int fib (int n)
{
    if(n==0) return 0;
    if(n==1) return 1;
    return fib(n-1) + fib(n-2);
}
```

```
int main()
{
    int n, ans;

    printf("Enter a number:\n");
    scanf("%d", &n);

    if(DEBUG)
        printf("%d\n",n);
    ans = fib(n);
    printf("Fibonacci of %d is %d\n", n, ans);
}
```

Source: https://jdoodle.com/a/4eF

Naïve Fibonacci algorithm

```
int fib (int n)
{
    if(n==0) return 0;
    if(n==1) return 1;
    return fib(n-1) + fib(n-2);
}
```

Definition:
$F_0 = 0$
$F_1 = 1$
$F_n = F_{n-1} + F_{n-2}$

- Is the algorithm correct? yes
- How long does it take to compute? Next slides

Fibonacci computation

- How to estimate computation effort:
  - Count operations.
  - Count operations as a function of input size.
  - Count operations as a proxy for time.
- $T(n)$ = run time for input n ≈ *number of operations* for input n.
  - **Portable** between machines.
  - Can **compare** algorithms.

Fibonacci computation

- Looking at $T(n)$ as number of operations to calculate the nth Fibonacci number, then
  - $T(n) = T(n-1) + T(n-2) + 3$ *(operations)*
  - $T(1) = T(0) = 1$
- Example: unrolling the loop
  - T(4) = T(3) + T(2) + 3

Fibonacci computation

- Looking at *T(n)* as number of operations to calculate the nth Fibonacci number, then
  - *T(n) = T(n-1) + T(n-2) + 3 (operations)*
  - *T(1) = T(0) = 1*
- Example: unrolling the loop
  - T(4) = T(3) + T(2) + 3
  - = T(2) + T(1) + 3 + T(2) + 3

Fibonacci computation

- Looking at *T(n)* as number of operations to calculate the nth Fibonacci number, then
  - *T(n) = T(n-1) + T(n-2) + 3 (operations)*
  - *T(1) = T(0) = 1*
- Example: unrolling the loop
  - T(4) = T(3) + T(2) + 3
  - = T(2) + T(1) + 3 + T(1) + T(0) + 3 + 3

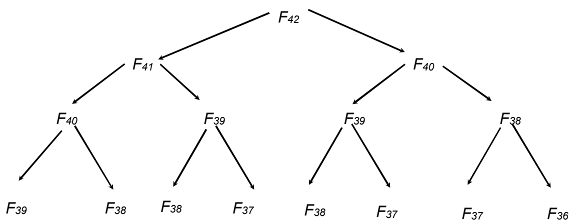
Fibonacci computation

- Looking at *T(n)* as number of operations to calculate the nth Fibonacci number, then
  - *T(n) = T(n-1) + T(n-2) + 3 (operations)*
  - *T(1) = T(0) = 1*
- Example: unrolling the loop
  - T(4) = T(3) + T(2) + 3
  - = T(2) + T(1) + 3 + T(2) + 3
  - = T(1) + T(0) + 3 + T(1) + 3 + T(1) + T(0) + 3 + 3
  - = 1+1+3+1+3+1+1+3+3 = 17 operations

Fibonacci computation

- Looking at *T(n)* as number of operations to calculate the nth Fibonacci number, then
  - *T(n) = T(n-1) + T(n-2) + 3 (operations)*
  - *T(1) = T(0) = 1*
- Example:
  - T(5) = T(4) + T(3)
  - = 17 + T(2) + T(1) + 3
  - = 17 + 9
  - = 26

How many operations to calculate F_{42}?



Any obvious inefficiencies?

Recomputation

Memoization

Store previously computed values

Source: https://jdoodle.com/a/4eM

```
int fib(int n)
{
        int i;
        int fib[n+1];

        fib[0] = 0;
        fib[1] = 1;

        for(i = 2; i <= n; i++)
        {
                fib[i] = fib[i-1] + fib[i-2];
        }
        return (fib[n]);
}/* how many operations to calculate fib(n)? */
```

…or without storing all the intermediate results

Source: https://jdoodle.com/a/4eH

```
int fib(int n)
{
        int result = 0;
        int preOldResult = 1;
        int oldResult = 1;

        if(n <= 0) return 0;
        if(n > 0 && n < 3) return 1;

        for ( int i = 3; i <= n; i++ ){
                result = preOldResult + oldResult;
                preOldResult = oldResult; //Bookkeeping of last 2 results
                oldResult = result;
        }
        return result;
}
```

1-19

Counting operations

- Count of operations as proxy for run time:
  - Advantages?
  - Caveats?
- How long does calculation of fib(2) take
  - Using the naïve algorithm? Use T(2) = 5 ops
  - Using memoization? 5 ops as well. Differences apper for big n!!
- Do we care how long things take for small input *n*?

COMP 20003 Algorithms and Data Structures 1-20

Complexity analysis: general method

- Count operations for $T(n)$ "time" (number of ops) taken for input n.
  - best to identify the most expensive operation and count that operation.
  - we can sometimes trade off space for time.

Complexity analysis: a fine point for fib()

- Assumption:
  - addition of two numbers takes constant time.
  - True *if* both numbers can fit into one computer word: 32 bits, number < 2^{32}.
- But Fibonacci numbers get very large:
  - F_n takes approximately $0.694n$ bits, so
  - To fit in one word, $n < 32/0.694 = 46$
  - $F_{50} = 12,586,269,025 > 12*10^9 > 2^{33}$
  - Assumption not valid for large n.

Closed form for Fibonacci numbers

- Binet's formula:

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

- For large n, $F_n \approx 2^{0.694n}$
- For more on Fibonacci numbers, *see :*
  - http://mathworld.wolfram.com/FibonacciNumber.html

Complexity analysis: Intuition

- Fortunately, most algorithms do deal with smaller numbers.
  - Counting operations usually suffices.
- Always be aware of the assumptions:
  - What is the most expensive operation?
  - Are the operations really constant?
  - What are the inputs and outputs?

Skiena: Algorithm Design Manual

- Chapter 1:
  - Algorithm correctness.
  - Example problems.
- Chapter 2:
  - Chapter 2.1: counting operations

A small diversion: Which C standard?

- C standards:
  - ANSI C (C89)
  - C99 – “substantially” completely supported in gcc 4.5 (with `-std=C99` option on)
  - C11 (current C standard, from 2011) `gcc 4.8`
- On `nutmeg.eng.unimelb.edu.au:`
  - `gcc -v:`
  - `gcc version 4.4.7`

C for gcc on nutmeg

- ANSI C with *some* of the features of C99, *e.g.*
  - Supported:
    - inline functions
    - long long int
  - Not supported:
    - Variable length arrays
    - Doesn’t insist on explicit return type for function
- For all the new features in C99 see:
  http://www.open-std.org/jtc1/sc22/wg14/www/newinc9x.htm

Next section

- Complexity analysis more formally.
- Big-O and related formalisms.