Intro

18 February 2018 18:45

<u>Concerned with:</u>

- design
- time taken to run - steps
    - computation steps for each logical step
- memory usage
    - larger data types increase this
- performance based on large n

The Fibonacci Series:

- Recursion
    - time - 1.6^n logical steps
    - from top to base

- pen and paper
    - store intermediate results in a table
    - from base to top
    - n logical steps
    - n logical RAM slots

- Iterative
    - fixed amount of RAM, only keep previous two values
        - larger integers mean RAM is actually proportional to n
    - from base to top

- Analytic
    - floating precision needs more bits as n increases

- Matrix
    - IMPOTANT - M^n takes $\log_2 n$ steps to compute
        - overhead piecing together results

Complexity Notation

06 April 2018 01:37

Cheat sheet:

Notation	Pronunciation	... (approximately, proportionally)	Program takes
O		Less than	At most this long
Ω	Omega	Greater than	At least this long
Θ	Theta	Equal	Approximately this long

Theta notation

- approximately proportional
- **ignore constant factors**
- interested in **asymptotic behaviour**
  - large n
- **think for worst case**
  - has to be consistent for large n
  - e.g. if depends on even odd n - cannot give theta function

Best complexities:

Smaller
- Constant
- Logarithmic
- polynomial
- exponential
  - (factorial)

Bigger

A function f(n) is **Θ(g(n))** if

there are constants **c1 > 0, c2 > 0** and **n0** (starting n) such that for all **n ≥ n0**

$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$

- f(n) sandwiched between two multiple of g(n)

A function f(n) is **O(g(n))** if

there are constants **c > 0** and **n0** (starting n) such that when **n ≥ n0**

$0 \le f(n) \le cg(n)$

$0 \le f(n) < cg(n)$ - **small** $o(g(n))$

- f(n) sandwiched between 0 and a multiple of g(n)
  - f(n) less than a multiple of g(n)

A function f(n) is **Ω(g(n))** if

there are constants **c > 0** and **n0** (starting n) such that when **n ≥ n0**

$0 \le cg(n) \le f(n)$

$0 \le cg(n) < f(n)$ - $\boldsymbol{\omega(g(n))}$

- f(n) greater than a multiple of g(n)

Divide & Conquer

20 February 2018 12:06

subproblems - solve smaller - then recombine
- requires subproblems to be unrelated
- recombining has a cost
  - increases as you divide further

E.g. $O(n^2)$

$$\text{new cost} = \mathbf{2^k \left(\frac{n}{2^k}\right)^2 + nk} = \frac{n^2}{2^k} + nk$$

- combine cost of O(n)

Subdivide to get to size of 1, $k = \log_2 n$
new cost = n+nlog₂n - O(nlogn)

1. let $n = 2^k$
2. $\mathbf{2k\left(\frac{T(n)}{2^k}\right)^2 + kO(n)}$

Prove if T(n) is O
- by induction
  - assume true for all m < n
  - then show true for n
- for $T(n) = O(test)$
- start from n = 1 or 2, get constraint on constant
- Start:
  - $T(n) \leq 2T\left(\frac{n}{2}\right) + cn$
- $m = \frac{n}{2} < n$
- find $T\left(\frac{n}{2}\right)$
- place into Start
- constant constraint to show P(n)

Master Method:
- runtime of D&C algorithm
- don't use if you don't have a,b,d - then do by hand

$$T(n) = a\,T(\,n/b\,) \;+\; O(n^d) \quad \begin{cases} a & \text{Number of subcases} \\ n/b & \text{Size of each subcase} \\ O(n^d) & \text{How long it takes to combine} \end{cases}$$

for some $a > 0$, $b > 0$ and $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases} \quad = \tilde{O}\left(n^{\max(d,\, \log_b a)}\right)$$

Dynamic Programming

06 April 2018 00:08

divide into non-independent, overlapping subproblems

Optimal substructure - best possible solution built from best possible sub solutions

Once we know the optimal solutions for all the possible subproblems, we can choose the optimal way of combining them

<u>Memoisation</u>

- modify a function to store the result
- if the function is called again with same arguments, can retrieve result instead of re-computing
- easy to do with a hash table

DP with Memoisation:

- top of function, check if result already stored, if so retrieve and return the value - O(1)

- reconstruction
    - store partial results - save on memory use
    - build full solution at end

<u>top-down vs bottom-up</u>

generally bottom-up more efficient practically, but both same complexity wise

top-down:

- recursive function calls
- large stack size as n increases

can use bottom-up when,

- know the order of which results depends on others
- know you have to compute all the values to get final result

How-to:

- start computing results from lowest n
- look-back in data structure
    - use previous results to compute new one
- continue until you reach desired n