Algorithm Review:

Time Complexity and Dynamic Programming

March 15, 2018

Materials adapted from "*Algorithms*" 4th Edition by *Robert Sedgewick* and *Kevin Wayne*, Princeton University and HackerEarth Dynamic Programming Tutorial

Table of Content

Analysis of Algorithms

How long will my program take?

Example: ThreeSum

Three Sum Problem

counts the number of triples in an array that sum to 0.

Example: ThreeSum

3 elements

Three Sum Problem

counts the number of triples in an array that sum to 0.

```
def three_sum(arr):
    N = len(arr)
    cnt = 0
    for i in range(0, N):
        for j in range(i + 1, N):
            for k in range(j + 1, N):
                if arr[i] + arr[j] + arr[k] == 0:
                    cnt += 1
    return cnt
```

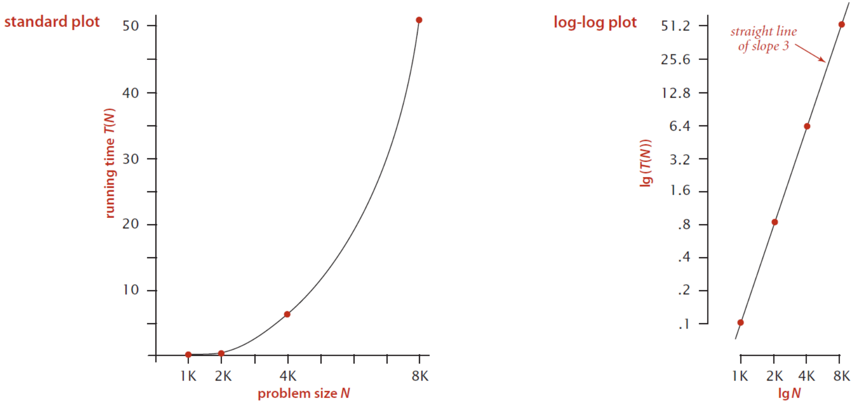
Example: ThreeSum

```
def three_sum(arr):
    N = len(arr)
    cnt = 0
    for i in range(0, N):
        for j in range(i + 1, N):
            for k in range(j + 1, N):
                if arr[i] + arr[j] + arr[k] == 0:
                    cnt += 1
    return cnt
```

The frequency of executions contain the followings:

- 2 for the first two statements
- number of times that `if` statement get executed
- number of times that `cnt+=1` statement get executed

Three Sum Running Time



Analysis of experimental data (the running time of `ThreeSum.count()`)

The `if` statement in ThreeSum

The `if` statement is executed precisely:

$$N(N-1)(N-2)/6$$

times. Through expansion, we obtain:

$$= N^3/6 - N^2/2 + N/3$$

The `if` statement in ThreeSum

The `if` statement is executed precisely: $\binom{n}{3} = 3 \times 2 \times 1$

$$N(N-1)(N-2)/6$$

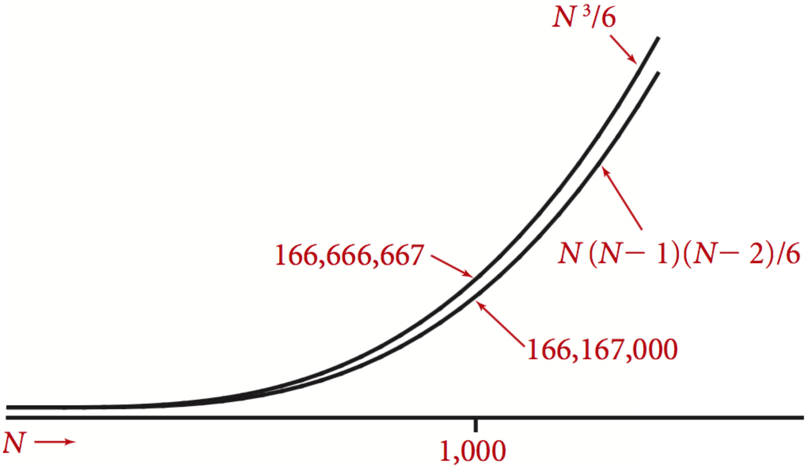
times. Through expansion, we obtain:

$$N^3/6 - N^2/2 + N/3$$

Typically, the terms after the leading term are relatively small for large N. For example, when $N = 1000$,

$$-N^2/2 + N/3 = 499,667$$
$$N^3/6 \approx 166,666,667$$

Leading-term Approximation



Leading-term approximation

Tilde Notation

Tilde Notation ($\sim$) allows to work with approximations, where we throw away low-order terms that complicate formulas and represent a negligible contribution to values of interest.

Definition

We write $\sim f(N)$ to represent any function that, when divided by $f(N)$, approaches 1 as N grows, and we write $g(N) \sim f(N)$ to indicate that $g(N)/f(N)$ approaches 1 as N grows.

$$\lim_{n \to \infty} \frac{g(N)}{f(N)} = 1$$

Tilde Notation

Tilde Notation ($\sim$) allows to work with approximations, where we throw away low-order terms that complicate formulas and represent a negligible contribution to values of interest.

Definition

We write $\sim f(N)$ to represent any function that, when divided by $f(N)$, approaches 1 as N grows, and we write $g(N) \sim f(N)$ to indicate that $g(N)/f(N)$ approaches 1 as N grows.

$$\lim_{n\to\infty} \frac{g(N)}{f(N)} = 1$$

function	tilde approximation
$N^3/6 - N^2/2 + N/3$	$\sim N^3/6$
$N^2/2 - N/2$	$\sim N^2/2$
$\lg N + 1$	$\sim \lg N$

Tilde Notation

Order of growth

Most often, we work with tilde approximations of the form $g(N) \sim af(N)$ where $f(N) = N^b(\log N)^c$ with a, b, and c constants and refer to $f(N)$ as the order of growth of $g(N)$

Tilde Notation

Order of growth

Most often, we work with tilde approximations of the form $g(N) \sim af(N)$ where $f(N) = N^b(\log N)^c$ with a, b, and c constants and refer to $f(N)$ as the order of growth of $g(N)$

function	tilde approximation	order of growth
$N^3/6 - N^2/2 + N/3$	$\sim N^3/6$	N^3
$N^2/2 - N/2$	$\sim N^2/2$	N^2
$\lg N + 1$	$\sim \lg N$	$\lg N$
3	~ 3	1

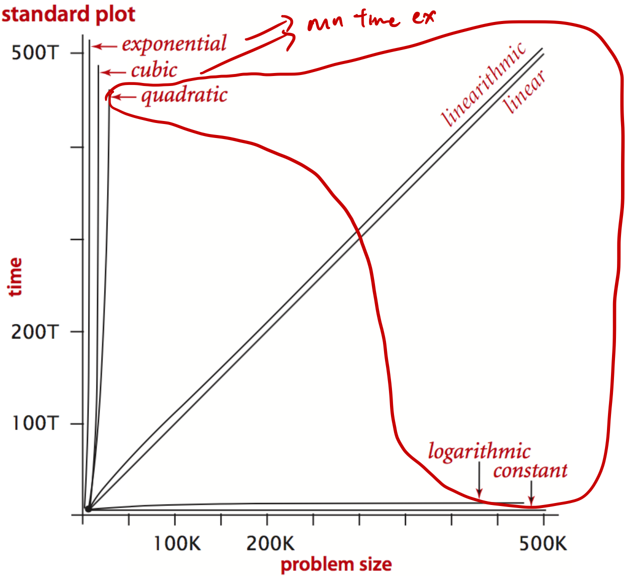
Order of growth

description	function
constant	1
logarithmic	$\log N$
linear	N
linearithmic	$N \log N$
quadratic	N^2
cubic	N^3
exponential	2^N

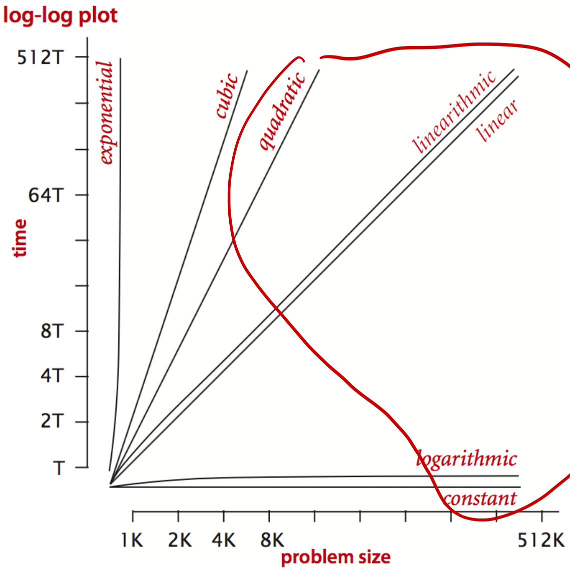
not efficient

Table: Commonly encountered order-of-growth functions

Order of growth: Standard Plot



Order of growth: Log-log Plot



Widely-used Notations

Big-O notation (O)

We say that $f(N) = O(g(N))$ if there exist constants c and N_0 such that $|f(N)| < cg(N)$ for all $N > N_0$.

$$f(N) = N^3/6 - N^2/2 + N/3$$
$$g(N) = N^3$$
$$f(N) = O(g(N))$$

upper bound

Widely-used Notations

Big-O notation (O)

We say that $f(N) = O(g(N))$ if there exist constants c and N_0 such that $|f(N)| < cg(N)$ for all $N > N_0$.

Widely-used Notations

Big-O notation (O)

We say that $f(N) = O(g(N))$ if there exist constants c and N_0 such that $|f(N)| < cg(N)$ for all $N > N_0$.

Big-Omega notation (Ω)

We say that $f(N) = \Omega(g(N))$ if there exist constants c and N_0 such that $|f(N)| > cg(N)$ for all $N > N_0$.

lower bound

Widely-used Notations

Big-O notation (O)

We say that $f(N) = O(g(N))$ if there exist constants c and N_0 such that $|f(N)| < cg(N)$ for all $N > N_0$.

Big-Omega notation (Ω)

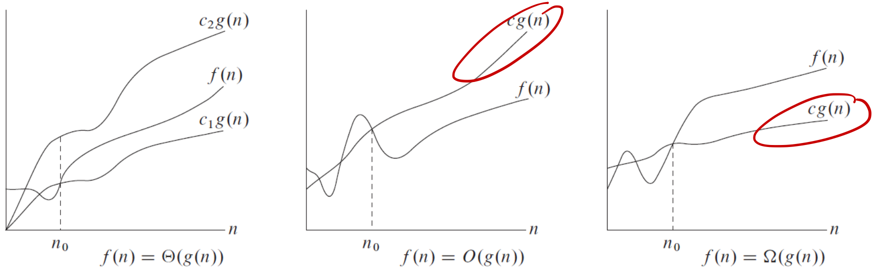
We say that $f(N) = \Omega(g(N))$ if there exist constants c and N_0 such that $|f(N)| > cg(N)$ for all $N > N_0$.

Big-Theta notation (Θ)

We say that $f(N) = \Theta(g(N))$ if $f(N)$ is $O(g(N))$ and $\Omega(g(N))$.

both O & Ω bounded

Graphical Examples



Widely-used Notations

Notation	Provides	Example	Shorthand for	Used to
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2}N^2$, $10N^2$, $5N^2 + 22N\log N$	classify algorithm
Big O	$\Theta(N^2)$ and smaller	$O(N^2)$	$10N^2$, $100N$, $22N\log N + 3N$	develop upper bound
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2}N^2$, N^5, $N^3 + 22N\log N$	develop lower bound

Table of Content

Example[1]: Warm Up

1. writes down "$1 + 1 + 1 + 1 + 1 + 1 + 1 + 1$"

$= 8$

[1] https://www.quora.com/How-should-I-explain-dynamic-programming-to-a-4-year-old/answer/Jonathan-Paulson

Example[1]: Warm Up

1. writes down "$1 + 1 + 1 + 1 + 1 + 1 + 1 + 1$"
2. equals to? 8

[1] https://www.quora.com/How-should-I-explain-dynamic-programming-to-a-4-year-old/answer/Jonathan-Paulson

Example[1]: Warm Up

1. writes down "$1+1+1+1+1+1+1+1$"
2. equals to? 8

[1] https://www.quora.com/How-should-I-explain-dynamic-programming-to-a-4-year-old/answer/Jonathan-Paulson

Example[1]: Warm Up

1. writes down "$1+1+1+1+1+1+1+1$"
2. equals to? 8
3. writes down another "$1+$" on the left

[1] https://www.quora.com/How-should-I-explain-dynamic-programming-to-a-4-year-old/answer/Jonathan-Paulson

Example[1]: Warm Up

1. writes down "$1 + 1 + 1 + 1 + 1 + 1 + 1 + 1$"
2. equals to? 8
3. writes down another "$1+$" on the left
4. the answer is 9

[1] https://www.quora.com/How-should-I-explain-dynamic-programming-to-a-4-year-old/answer/Jonathan-Paulson

Example[1]: Warm Up

1. writes down "$1 + 1 + 1 + 1 + 1 + 1 + 1 + 1$"
2. equals to? 8
3. writes down another "$1+$" on the left
4. the answer is 9

Dynamic Programming (DP) is just a fancy way to say 'remembering stuff to save time later'

[1]https://www.quora.com/How-should-I-explain-dynamic-programming-to-a-4-year-old/answer/Jonathan-Paulson

Dynamic Programming

- avoid repeated work by remembering partial results
- trade space for time
- break a problem down into subproblems.

Fibonacci Numbers

$$f(n) = \begin{cases} 1 & n = 0, 1 \\ f(n-1) + f(n-2) & n \geq 2 \end{cases}$$

The first few numbers: $1, 1, 2, 3, 5, 8, 13, 21 \cdots$ and so on.

$f(0)$ $f(1)$ $f(2)$

$f(2-1) + f(2-2) = f(1) + f(0)$

$= 1 + 1 = 2$

$f(3-1) + f(3-2) = f(2) + f(1)$

$= 2 + 1 = 3$

Fibonacci Numbers

$$f(n) = \begin{cases} 1 & n = 0, 1 \\ f(n-1) + f(n-2) & n \geq 2 \end{cases}$$

The first few numbers: $1, 1, 2, 3, 5, 8, 13, 21 \cdots$ and so on.

```
def fib(n):
  if n < 2:
    return 1
  return fib(n - 1) + fib(n - 2)

query=[100, 20, 1000, 40, 5]
for i in range(len(query)):
  print(fib(query[i]))
```

Fibonacci Numbers

$$f(n) = \begin{cases} 1 & n = 0, 1 \\ f(n-1) + f(n-2) & n \geq 2 \end{cases}$$

The first few numbers: $1, 1, 2, 3, 5, 8, 13, 21 \cdots$ and so on.

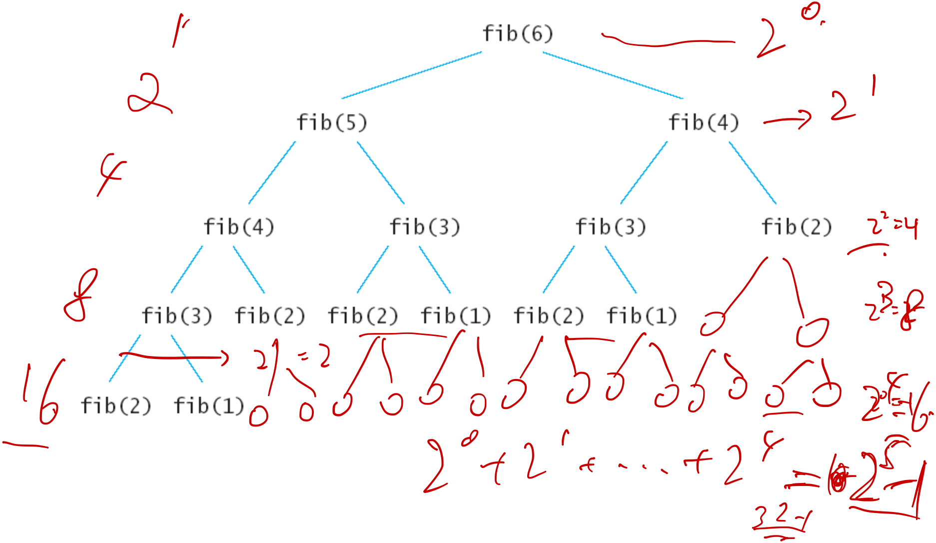
```
def fib(n):
  if n < 2:
    return 1
  return fib(n - 1) + fib(n - 2)

query=[100, 20, 1000, 40, 5]
for i in range(len(query)):
  print(fib(query[i]))
```

Recursion! $O(2^n)$

Fibonacci Numbers

1 1 2 3 5 8 13 21 34



Fibonacci Numbers

$$f[n] = \begin{cases} 1 & n = 0, 1 \\ f[n-1] + f[n-2] & n \geq 2 \end{cases}$$

Dynamic Programming

```
def build_fib(N):
  fib = [None for n in range(N)]
  fib[0] = 1
  fib[1] = 1
  for n in range(2, N):   → N times
    fib[n] = fib[n - 1] + fib[n - 2]
  return fib

fib = build_fib(1001)
query=[100, 20, 1000, 40, 5]
for i in range(len(query)):
  print(fib[query[i]])
```

1 × N

Requires $O(N)$ space and running time of $O(N)$. vs $O(2^n)$

Dynamic Programming

```python
def build_fib(N):
  fib = [None for n in range(N)]
  fib[0] = 1
  fib[1] = 1
  for n in range(2, N):
    fib[n] = fib[n - 1] + fib[n - 2]
  return fib

fib = build_fib(1001)
query=[100, 20, 1000, 40, 5]
for i in range(len(query)):
  print(fib[query[i]])
```

Requires $O(N)$ space and running time of $O(N)$.
Bottom Up approach

Steps for Solving DP Problems

1. Define optimal subproblems
    - E.g., $f[n]$
2. Write down the recurrence that relates optimal subproblems. Compute the value of the optimal solution in bottom-up fashion.
    - E.g., $f[n] = f[n-1] + f[n-2]$
3. Recognize and solve the base cases
    - E.g., $f[0] = f[1] = 1$

Example: Longest Common Subsequence Problem[2]

- given two strings x and y, find the longest common subsequence (LCS) and print its length
- Example:
  - x : `ABCBDAB`
  - y : `BDCABC`
  - `BCAB` is the longest subsequence found in both sequences, so the answer is 4

[2] https://web.stanford.edu/class/cs97si/04-dynamic-programming.pdf

Example: Longest Common Subsequence Problem

- Define optimal subproblems
  - $dp[i][j]$ be the length of the LCS of $x_{1 \ldots i}$ and $y_{1 \ldots j}$
- Write down the recurrence that relates optimal subproblems. Compute the value of the optimal solution in bottom-up fashion.

$$dp[i][j] = \begin{cases} dp[i-1][j-1]+1 & x_i = y_j \\ \max(dp[i-1][j], dp[i][j-1]) & x_i \neq y_j \end{cases}$$

- Recognize and solve the base cases
  - $dp[i][0] = 0, \forall i$
  - $dp[0][j] = 0, \forall j$