CS 170

EFFICIENT ALGORITHMS AND INTRACTABLE PROBLEMS

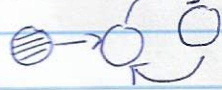
LECTURE 1

8/23/18

• CYCLE DETECTION IN LIST

① 1. Mark first node
   2. While next cell is not marked
   3. Go to next cell

DOES NOT work
- first no doesn't have to be in cycle

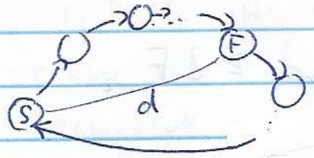


• Claim: Either it has no next cell or detect a cycle

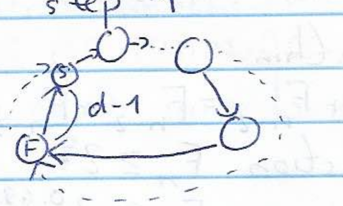
② 1. Initialize 2 pointers
   2. Advance pointer 1 twice
   3. Advance pointer 2 once

• Claim: If ever at the same place, report cycle
=> If no cycle, slow pointer never catches fast one
=> If cycle, both pointers will enter cycle at some point.

step 0

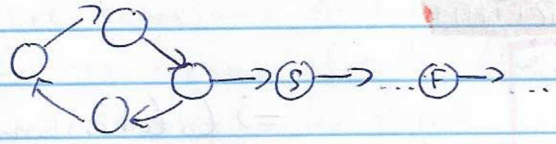


step 1



d decreases at every step by 1

• Runtime: n steps to cycle, n steps to catch up => $O(n)$
• Additional storage: two pointers => $O(1)$
- Careful! - if you start at tail, it won't work



• ALGORITHMS FOR HUMAN GENOME PROJECT

- Reconstruct DNA
- read first, then next, then next, ...
  -> slow and error prone

=> parallel sequencing
  - yields chunks of overlapping DNA
  -> assemble into consistent string

TGAGTAGATA

TGAGT
   AGTAG
      AGATA

- PAGE RANK
  - => Random Surfer Model
    - follow link, follow link, ...
    - occasionally jump to random page
  - page rank = popularity for random surfer
- FIBONACCI NUMBERS

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

- Runtime

$$T(n) = T(n-1) + T(n-2) + 2$$

time for F(n-1) call

time spend in procedure

-> NOTE

$T(n) \geq F(n)$

=> bad algorithm

$F_n = F_{n-1} + F_{n-2} = F_{n-2} + F_{n-3} + F_{n-2} \geq 2F_{n-2}$

- by induction $F_n \geq 2^{\frac{n}{2}}$

$F_n \approx 2^{0.69n}$

-> therefore

$T(n) \geq 2^{\frac{n}{2}} \approx 2^{0.69n}$

-> exponential algorithm

=> BETTER ALGORITHM

```
def fib(n):
    if n ≤ 1:
        return n
    else
        a=[0,1]
        for i in xrange(2,n+1):
            a.append(a[i-1]+a[i-2])
        return a[n]
```

} so through loop $O(n)$

=> total number of operations
$O(n^2)$

-> computing time complexity
- How many bits are in representation of $F(n)$?
  $\log_2 F_n \approx 0.69$
- How long does it take to compute $F_{n-1} + F_{n-2}$?
  $O(n)$

- How long does Fib take?
  $O(n)$ - n additions
  => at most $O(n^2)$
  - doubling size of n made Fib grow by roughly factor of 4
    cn^2 ... runtime when n
    $c(2n)^2 = 4cn^2$ ... runtime when 2n

• POLYNOMIAL VS EXPONENTIAL
- polynomial has runtime $O(n^k)$
  -> scaling input by c grows runtime bound by c^k
- doubling n, scales runtime by constant for polynomial
  - for exponential, it squares runtime

• ASYMPTOTIC ANALYSIS OF FIBONACCI
- recursive Fib has faster inner loop than iterative Fib
- recursive
  $2^{0.69n}$ -> doubling n, squares runtime
- iterative
  n^2 -> doubling n, multiplies runtime by 4

• REFRESHING ASYMPTOTIC NOTATION
=> ignore constant factors
  $2n^2$ is asymptotically the same as $4n^2 \to O(n^2)$
=> ignore smaller order terms
  $2n^2 + 1000\log(n) \to O(n^2)$
=> upper bound: O
  $n^2 \to O(n^3)$
  $\log n \to O(n)$
  - Formally, for positive function f and g from integers to reals, $g(n) = O(f(n))$, if there is constant c where $g(n) \leq c \cdot f(n)$
=> lower bound: Ω
  $2n^2$ is $\Omega(n^2)$ and $\Omega(n)$
  - Formally, $g(n) = \Omega(f(n))$ if there is constant c where $g(n) \geq cf(n)$
=> both lower and upper bound
  $g(n) = \theta(f(n))$ if $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$

• AL KHWARIZMI - ARITHMETIC

⇒ addition $O(n)$

⇒ multiplication $O(n^2)$

$$\begin{array}{r} 1\,2\,3 \to x \\ \cdot\ 4\,5\,6 \to y \\ \hline \cdots \\ \cdots \\ \hline \cdots \to xy \end{array}$$

n-bit number

- $O(n^2)$
  - because every digin in x must multiply every digit in y at least once

⇒ BETTER ALGORITHM

- if x and y are n-digits → xy is 2n-digits
- k^{th} place of xy - coefficient of 2^k

$$a_k = \sum_{i \le k} x_i y_{k-i} = x_k \cdot y_0 + x_{k-1} \cdot y_1 + \ldots$$

$$x \cdot y = \sum_{k=0}^{2n} 2^k a^k$$

→ number of basic operations

$$\sum_{k \le 2n} \min(k, 2n-k) = O(n^2)$$

→ Recursive Algorithm for Multiplication

$$x = \boxed{x_L \mid x_R} = 2^{\frac{n}{2}} \cdot x_L + x_R$$

$$y = \boxed{y_L \mid y_R} = 2^{\frac{n}{2}} y_L + y_R$$

$$x \cdot y = (2^{\frac{n}{2}} x_L + x_R)(2^{\frac{n}{2}} y_L + y_R)$$
$$= 2^n x_L y_L + 2^{\frac{n}{2}}(x_L y_R + x_R y_L) + x_R y_R$$

↳ multiplication between two numbers that are $\frac{n}{2}$-bits

⇒ recurrence

$$T(n) = 4T(\tfrac{n}{2}) + O(n)$$

↳ add results of 4 multiplicati[ons]

↳ you have to do 4 $\frac{n}{2}$-bit number multiplications

multiply (x, y)

* input: x, y in binary
* output: x·y

n = max (size of x, size of y)

if n = 1 : return xy

x_L, x_R = left and right $[\frac{n}{2}]$ bits of x

y_L, y_R = left and right $[\frac{n}{2}]$ bits

P_1 = multiply (x_L, y_L)

P_2 = multiply (x_R, y_R)

P_3 = multiply $(x_L + x_R, y_L + y_R)$

return $P_1 \cdot 2^n + (P_3 - P_1 - P_2) \cdot 2^{\frac{n}{2}} + P_2$

• NOTE

- it doesn't make sence to recurse all the way down to 1-bit
  → for most processors 16 or 32 multiplication is single procedure

- think about recursion tree
  - $\rightarrow$ a degree 4 - tree of depth $\log_2 n$
  - $\rightarrow \theta(n^2)$ leaves or base cases (digit multiplying digit)
    - one for each pair of digits

- **COMPARING MULTIPLICATION ALGORITHMS**
  - $\Rightarrow$ elementary school multiplication
    - when $n \rightarrow 2n$
    - $\rightarrow$ runtime: $T \rightarrow 4T$ because $T = cn^2$
      $T' = c(2n)^2 = 4cn^2 = 4T$
  - $\Rightarrow$ Python multiplication
    - when $n \rightarrow 2n$
    - $\rightarrow$ runtime $T \rightarrow 3T$
      - asymptotic $T = cn^{\omega}$
        $c(2n)^{\omega} = T' = 3T = 3(cn^{\omega})$
        $\Rightarrow \omega = \log_2 3 = 1.58$
    - $\rightarrow O(n^{\log_2 3})$

Can we do better?
- faster algorithms for multiplication exist
- $\rightarrow$ based on Fourier transform
  - also divide and conquer algorithm

- **GAUSS'S TRICK**
  $(a+bi)(c+di) = (ac-bd)+(ad+bc)i$
  - $\rightarrow$ 4 multiplications

  $$\left.\begin{aligned} P_1 &= (a+b)(c+d) \\ P_2 &= ac \\ P_3 &= bd \end{aligned}\right\} \text{3 multiplications}$$

  - $\Rightarrow (ac-bd) = P_2 - P_3$
  - $\Rightarrow (ad+bc) = P_1 - P_2 - P_3$
  - $\rightarrow$ uses just 3 multiplications (one extra addition)

- **FASTER ALGORITHM FOR MULTIPLICATION**
  - remember $x \cdot y = 2^n x_L y_L + 2^{\frac{n}{2}}(x_L y_R + x_R y_L) + x_R y_R$
  - $\rightarrow$ need to compute 3 terms $(x_L y_R + x_R y_L)$, $x_L y_L$, $x_R y_R$
  - because Gauss's trick we can compute
    $x_L y_L, x_L y_R, x_R y_L, x_R y_R$
    with just 3 multiplications

    $P_1 = (x_L + x_R)(y_L + y_R)$
    $P_2 = x_L y_L$
    $P_3 = x_R y_R$

  $$\boxed{T(n) = 3T\left(\frac{n}{2}\right) + \theta(n) = \theta\left(n^{\log_2 3}\right)}$$

  $\hookrightarrow$ runtime is $\theta(n^{\log_2 3})$ - number of base cases is $n^{\log_2 3}$