Information Science
9: Computational Complexity in Details
--- more examples ---

Naonori Kakimura
垣村尚徳
kakimura@global.c.u-tokyo.ac.jp

PEAK
Programs in English at Komaba

Image Assignment Results

- Available at
  - http://www.graco.c.u-tokyo.ac.jp/labs/kakimura/Lecture/IS2016/ImagePEAK2016.html

About the Eigenvalue Problem

- Math itself is out of this course's scope
  - One of applications of eigenvalues(linear algebra)

- Final exam does not require minor knowledge
  - NOT asking
    - Memory
      - What "make2d" is?
      - Which is correct "for i in 1..3" or "for i in 1....3"?
        - It may ask: fill in the blank of "for i in 1..(?)"
    - Math knowledge: What "eigenvalue" is?
  - I will explain what they are if they are in the exam

About the Eigenvalue Problem

- Math itself is out of this course’s scope
  - One of applications of eigenvalues(linear algebra)

- Final exam does not require minor knowledge
  - Asking
    - How to read/translate a Ruby program
    - How to design algorithms
    - How to evaluate algorithms

Today's Contents

- Review of complexity order
- Analyzing complexity of algorithms
  - Computing exponential functions
  - Review of computing the number of combinations
- Exercises

Review of Computational Complexity

- Complexity = # Operations
  - Rough estimation of running time BEFORE execution
  - Independent of computer environments

- Ex
  - 1+2+3
    - 2 times
  - 1+2+3+⋯+n
    - n-1 times

Review of Computational Complexity

- Complexity = # Operations

- Summation of n numbers
  - Using the For-loop

```
def sum_loop(n)
    s = 0
    for i in 1..n
        s = s + i
    end
    s
end
```

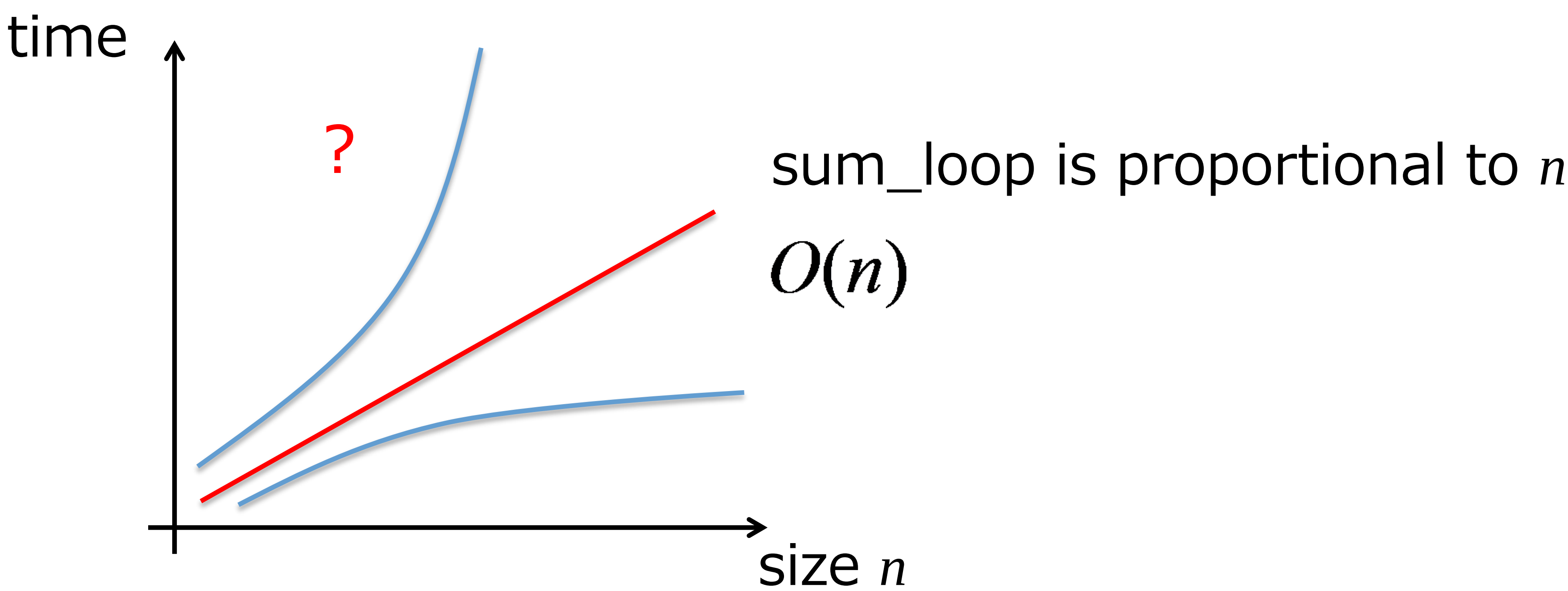
1 operations

n operations
(1 for each i)

Total # operations = n+1
(proportional to n)

Computational Complexity Order

- Interest: proportional relationship btw size & time

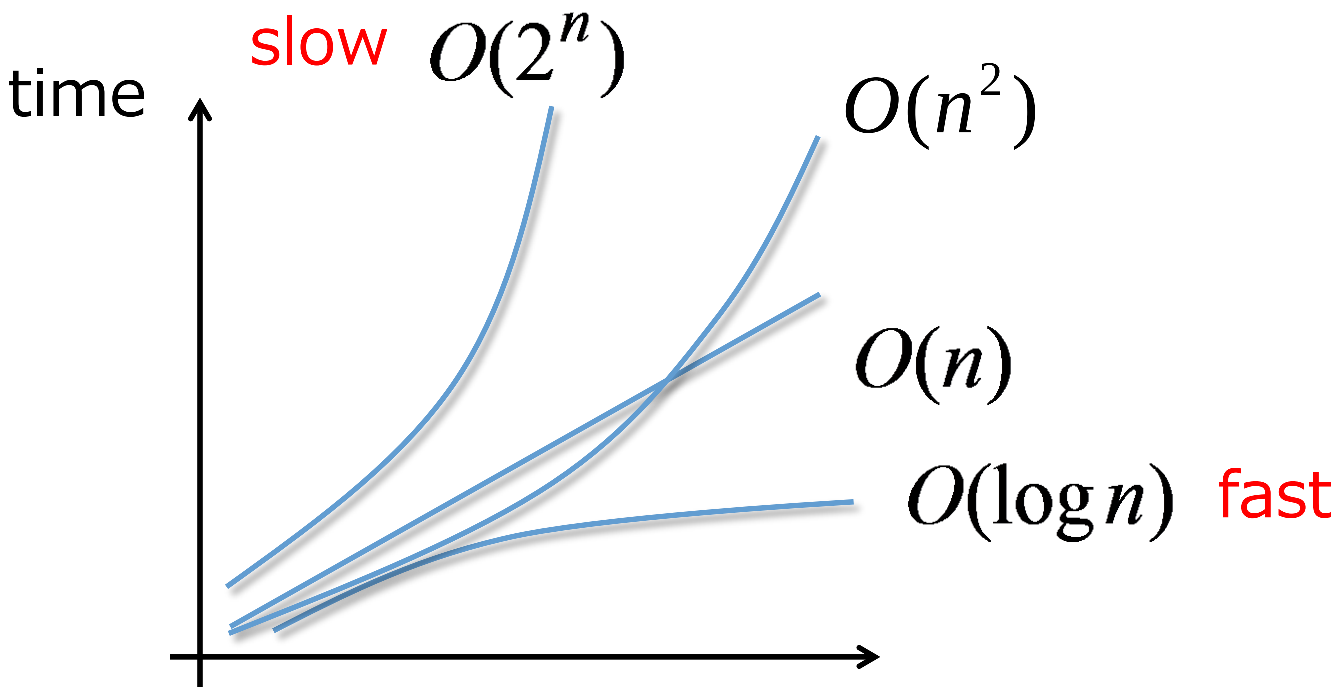


- *Rough* estimation how long it takes
- Use the order notation "O( )"
  - Ex: $\mathbf{O}(n)$, $\mathbf{O}(n^2)$, $\mathbf{O}(\log n)$
    - $\mathbf{O}(\cdot)$: "time is proportional to $\cdot$ "

More precisely
it is an upper bound

Computational Complexity Order

- Interest: proportional relationship btw size-time



Order Notation

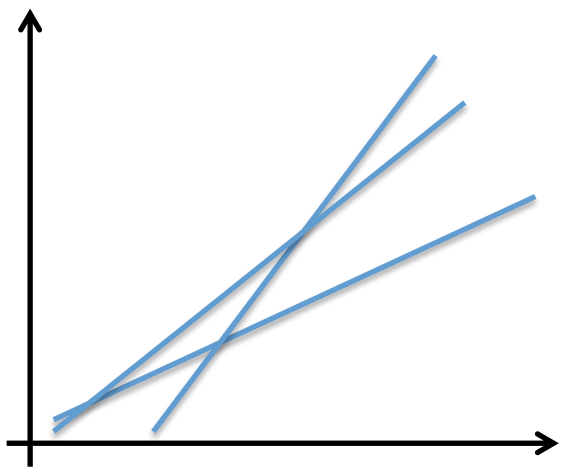
- Use "order" notation instead of detailed eqn
  - Ex: $\mathbf{O}(\boldsymbol{n})$, $\mathbf{O}(\boldsymbol{n}^2)$, $\mathbf{O}(\mathbf{log}\ \boldsymbol{n})$
    - O(・) : "time is proportional to ・ "
  - $\mathbf{O}(\boldsymbol{n})$: $\boldsymbol{n}$ increases 100 times → time 100 times
  - $\mathbf{O}(\boldsymbol{n}^2)$: $\boldsymbol{n}$ increases 100 times → time 10K times
  - $\mathbf{O}(\Phi^{\boldsymbol{n}})$: $\boldsymbol{n}$ increases 100 times → time $\Phi^{99\boldsymbol{n}}$ times
  - $\mathbf{O}(\mathbf{log}\ \boldsymbol{n})$: $\boldsymbol{n}$ increases $\boldsymbol{n}$ times → time 2 times

$$\log(n \times n) = 2 \log n$$

- Point: We do not care small details
  - Ignore coefficients, rounding-up/down
    - $\mathbf{O}(\boldsymbol{n})$ 「proportional to $\boldsymbol{n}$」 when $\boldsymbol{n}$, $\mathbf{2}\boldsymbol{n}$, $\mathbf{100000}\boldsymbol{n}$ times
  - Leave the most dominant term only
    - $\mathbf{O}(\boldsymbol{n}+\mathbf{8})=\mathbf{O}(\boldsymbol{n})$, $\mathbf{O}(\boldsymbol{n}+\mathbf{log}\ \boldsymbol{n})=\mathbf{O}(\boldsymbol{n})$

Computational Complexity Order

- Interest: proportional relationship btw size-time



Don’t distinguish:

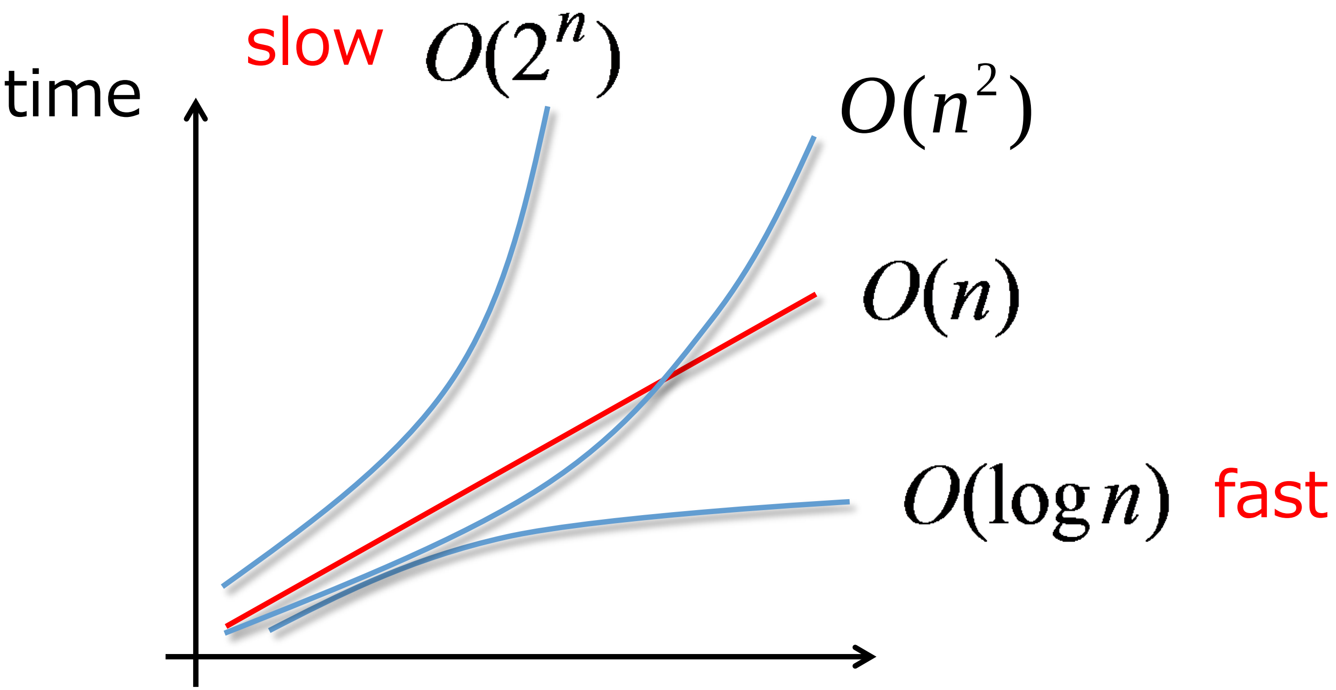
$$\left.\begin{array}{r}\text{Proportional to } n \\ \text{to } 100n \\ \text{to } 10n+40\end{array}\right\} O(n)$$

If n is huge, they are almost same

$$\lim_{n\to\infty}\frac{100n}{n}=const$$

Computational Complexity Order

- Interest: proportional relationship btw size-time



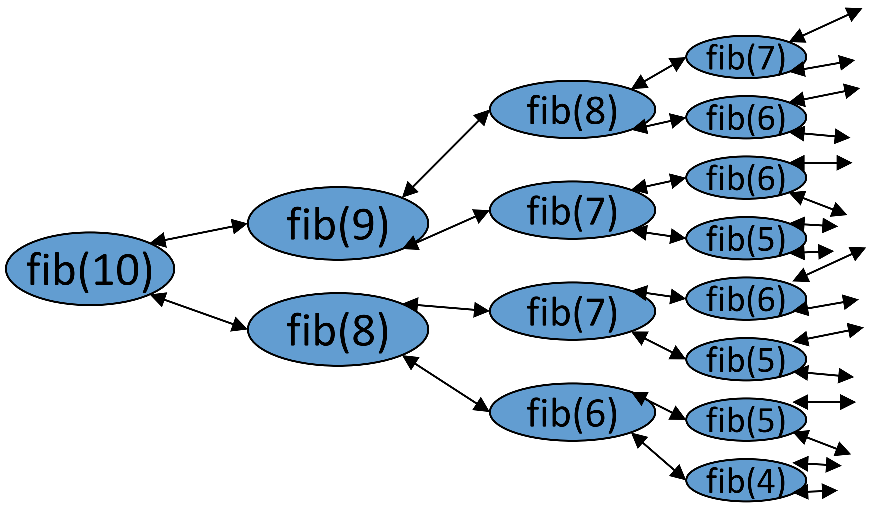
- But $O(n)$ is much different from $O(\log n)$ and $O(n^2)$

$$\lim_{n\to\infty}\frac{n}{\log n}=\infty \qquad \lim_{n\to\infty}\frac{n^2}{n}=\infty$$

Much slower than $O(\log n)$ Much faster than $O(n^2)$

The Fibonacci Case

- Definition-based
  - $\mathbf{O(\Phi^{k})}$



- Enumeration-based
  - $\mathbf{O(k)}$

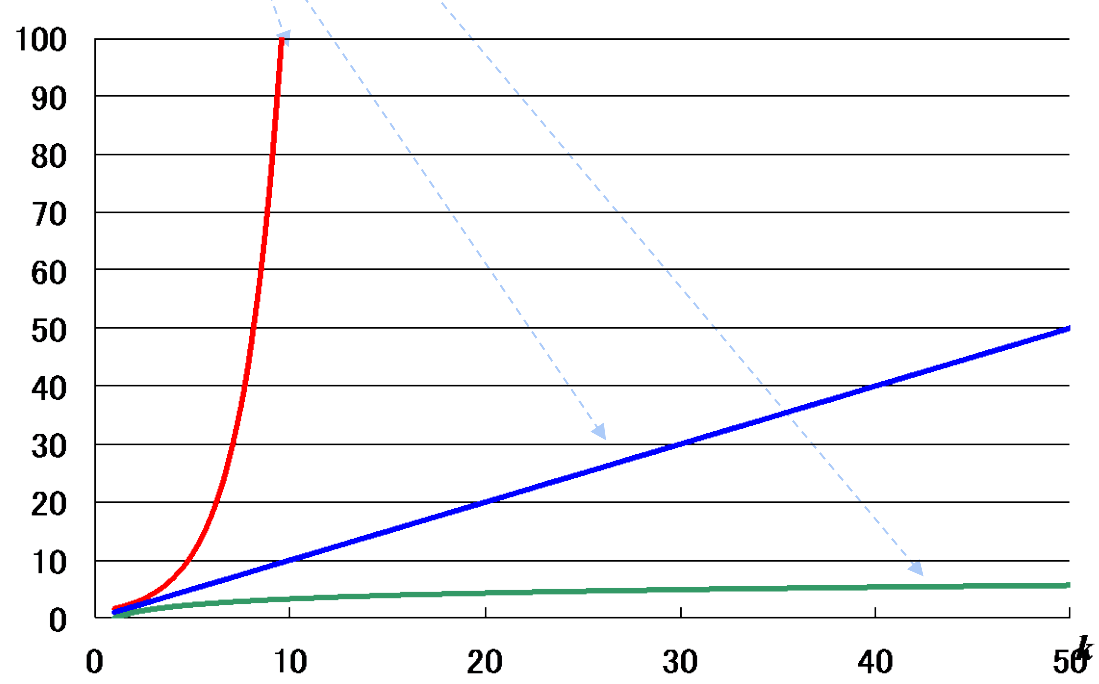
Complexity for Fibonacci Numbers

- Finding the ***k*** th Fibonacci number

$$\phi = \frac{1+\sqrt{5}}{2}$$

  - Definition-based — $\mathbf{O}(\Phi^{k})$
  - Enumeration — $\mathbf{O}(k)$
  - Matrix-computation — $\mathbf{O}(\log k)$

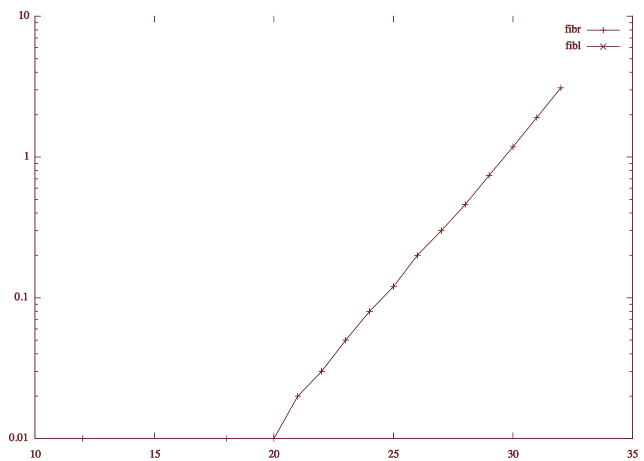
    See Exercise 6.1.6-7

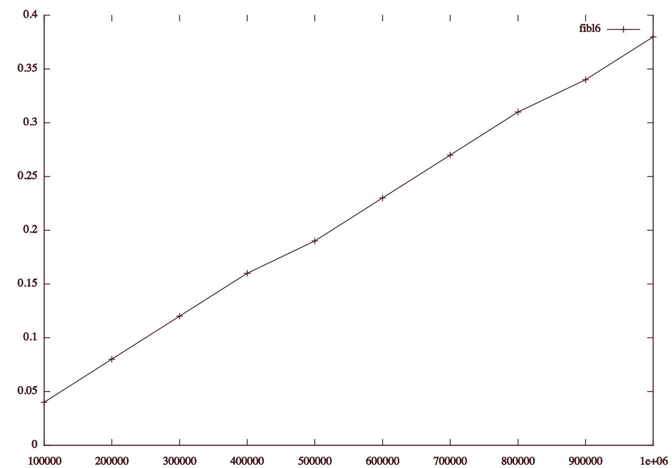


Review: Actual Running Time

- fibr → $\mathbf{O}(\Phi^n)$
  - Actual computational time ∝ an exponential function
- fibl → $\mathbf{O}(n)$
  - Actual computational time ∝ a linear function

Order is a good approximation





Today’s Contents

- Review of complexity order
- Analyzing complexity of algorithms
  - Computing exponential functions
  - Review of computing the number of combinations
- Exercises

More Example: power(a,n)

- Computing an exponential function a^n
  - Three ways to make a program
    - Using for-loop: power_loop(a,n)
    - Using recursion: power_r(a,n)
    - Using efficient recursion: power2(a,n)

Method 1: power_loop(a,n)

- Computing an exponential function a^n
  - Using for-loop: power_loop(a,n)

```
def power_loop(a, n)
    s = 1
    for i in 1..n
        s = s * a
    end
    s
end
```

1 operations

n operations
(1 for each i)

$O(n)$

Method 2: power_r(a,n)

- Computing an Exponential function a^n
  - Using recursion: power_r(a,n)

$$a^n = \begin{cases} a \times a^{n-1} & (n \geq 1) \\ 1 & (n = 0) \end{cases}$$

```
def power_r(a, n)
    if n == 0
        1
    else
        a*power_r(a, n-1)
    end
end
```

(n≧1)

1 operations

+ #operations when n-1

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

Method 2: power_r(a,n)

- Computing an Exponential function a^n
  - Using recursion: power_r(a,n)

$$a^n = \begin{cases} a \times a^{n-1} & (n \geq 1) \\ 1 & (n = 0) \end{cases}$$

```
def power_r(a, n)
    if n == 0
        1
    else
        a*power_r(a, n-1)
    end
end
```

(n≧1)

1 operations

+ 1 operations

+ #operations when n-2

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

Method 2: power_r(a,n)

- Computing an Exponential function a^n
  - Using recursion: power_r(a,n)

$$a^n = \begin{cases} a \times a^{n-1} & (n \geq 1) \\ 1 & (n = 0) \end{cases}$$

```
def power_r(a, n)
    if n == 0
        1
    else
        a*power_r(a, n-1)
    end
end
```

(n≧1)

1 operations

+ 1 operations

+ 1

+⋯

+ #operations when n=0

$$T(n) = O(n)$$

Toward Breaking O(n)

- Consider computing a^8
  - 8 multiplication if we use the previous 2 programs
  - More efficient way (3 times)
    - Compute a^2 from $a \times a$
    - Compute a^4 from $a^2 \times a^2$
    - Compute a^8 from $a^4 \times a^4$

$$a^n = \begin{cases} a^{n/2} \times a^{n/2} & (n : \text{even}, \ \ n \geq 2) \\ a \times a^{n-1} & (n : \text{odd}) \\ 1 & (n = 0) \end{cases}$$

Method 3: power2(a,n)

- Computing an Exponential function a^n
  - Using recursion: power2(a,n)

```
def power2(a, n)
   if n == 0
      1
   elsif n%2 == 0
      (power2(a, n/2))**2
   else
      a*power2(a, n-1)
   end
end
```

$$a^n = \begin{cases} a^{n/2} \times a^{n/2} & (n : \text{even}, n \geq 2) \\ a \times a^{n-1} & (n : \text{odd}) \\ 1 & (n = 0) \end{cases}$$

Examples

- n=35: change of n
  - 35 → 34 → 17 → 16 → 8 → 4 → 2 → 1 → 0

    3rd cond.

    2nd cond.

```
def power2(a, n)
    if n == 0
        1
    elsif n%2 == 0
        (power2(a, n/2))**2
    else
        a*power2(a, n-1)
    end
end
```

Complexity of Method 3: power2(a,n)

- n=35: change of n
  - 35 → 34 $\xrightarrow{\times 1/2}$ 17 → 16 $\xrightarrow{\times 1/2}$ 8 $\xrightarrow{\times 1/2}$ 4 $\xrightarrow{\times 1/2}$ 2 $\xrightarrow{\times 1/2}$ 1 → 0

```
def power2(a, n)
    if n == 0
        1
    elsif  n%2 == 0
        (power2(a, n/2))**2
    else
        a*power2(a, n-1)
    end
end
```

\# of → (being half)

≦ min number t satisfying $\frac{n}{2^t} \leq 1$

⇒ $t = \lceil \log n \rceil$

Rounding up

(# of →) ≦ (# of →) $+ 1$

Have no two consecutive repetition (next time is always →)

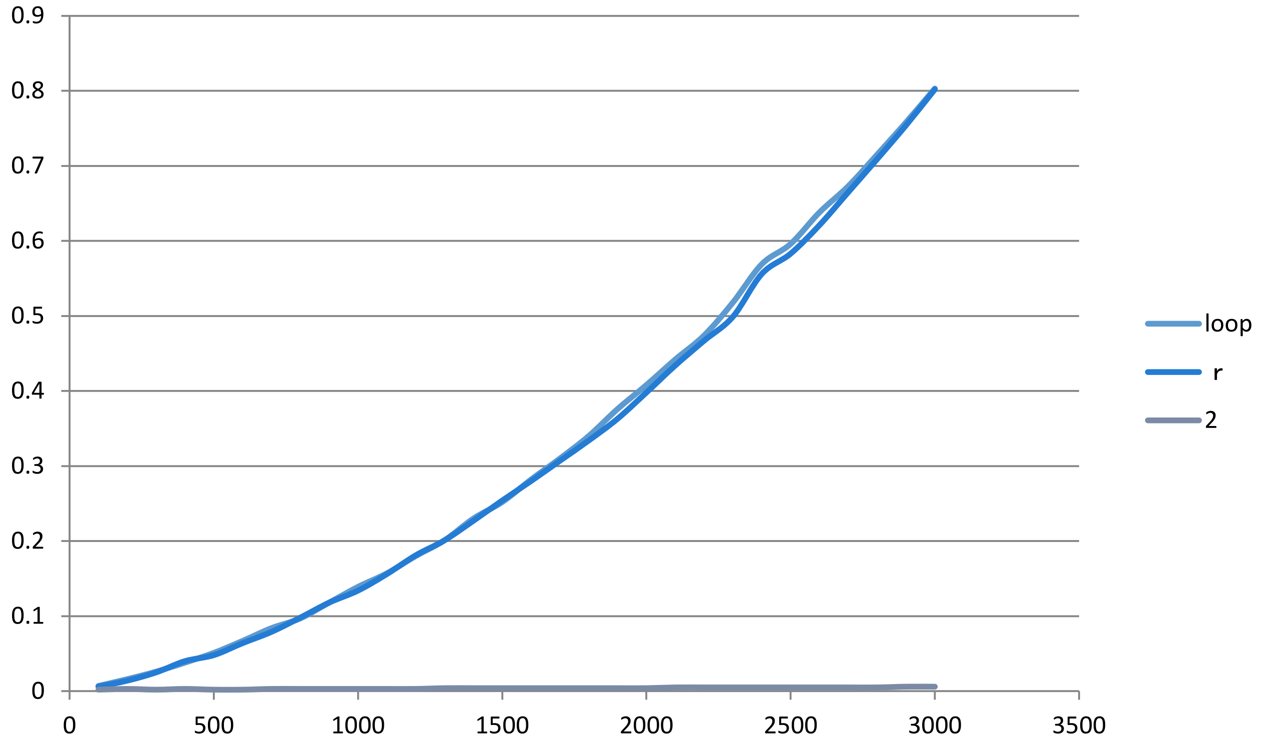
Total complexity: $O(\log n)$ (# iteration ≦ 2 $\log n$ + 2)

Remarks

- Complexity of power2 depends on the input
  - n=32: 7 times
    - n: 32 → 16 → 8 → 4 → 2 → 1 → 0
      - One operation for each n
  - n=31: 10 times
    - n: 31 → 30 → 15 → 14 → 7 → 6 → 3 → 2 → 1 → 0
- Usually estimate the worst case
  - When n=2^m-1, then it takes 2m-1($\simeq$ 2log$n$-1) operations

Actual Computational Time

- Power2 is much faster



Loop, rec: When n is large, we have to do addition of large numbers, which takes time. This makes it slower than a linear function

Today’s Contents

- Review of complexity order
- Analyzing complexity of algorithms
  - Computing exponential functions
  - Review of computing the number of combinations
- Exercises

(Recap) Combination Number ${}_nC_k$

- the number of combinations when we choose k items out of n items

Choose k items out of n elements

$$ {}_nC_k = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots 2\cdot 1} $$

$$ = \frac{n!}{k!(n-k)!} $$

(Recap) Recursive Definition of Combinations

$$
{}_nC_k = \begin{cases} 0 & \text{(when k > n)} \\ 1 & \text{(when k = 0)} \\ {}_{n-1}C_{k-1} + {}_{n-1}C_k & \text{(otherwise)} \end{cases}
$$

```ruby
load ("./make2d.rb")

def combination_loop(n,k)
  c = make2d(n+1,n+1)
  for i in 0..n          # for each row, do the following
    c[i][0] = 1                          # 1st column
    for j in 1..(i-1)    # 2nd to (i-1)th column
      c[i][j] = c[i-1][j-1] + c[i-1][j]
    end
    c[i][i] = 1                          # ith column
  end
  c[n][k]
end
```

combination_loop.rb

Behavior of Algorithm

- Make an (n+1)×(n+1) array
- Fill in the entries from 0th row to nth row
  - One operation for each entry
  - Each entry is scanned once

when n=6

the row is determined by the last row

$n \setminus k$	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

Complexity=# cells

$$O(n^2)$$

The value of ${}_nC_k$

Today's Contents

- Review of complexity order
- Analyzing complexity of algorithms
  - Computing exponential functions
  - Review of computing the number of combinations
- Exercises

Exercise1:

- A certain store has two software *A* and *B* to process experimental data. It is known that for an input of size N,
  - *A* runs in $O(N^2)$ time, and
  - *B* runs in $O(N \log_2(N))$ time.

- When we process 1000-record test data, *A* takes 1 second, while *B* takes 10 seconds.
- The target data has 1-million records.
- Which software is better to process the data? Explain why?

Introduction of Exercise 2: Counting Data

- We want to count the number of items sold
  - At a cash register 
    -  ,  ,  ,  ,  ,…
- Count how many times each item was sold
  -  -- 46
  -  -- 87
  -  -- 53
  -  -- 45

Exercise 2: Formal Description

1. **(Past Exam 2010)** Suppose that an array a has size n and contains m kinds of positive integers. We want to store all the distinct integers of a to b, and also return the frequencies of occurrence in array c. For example, if a=[3,1,4,1,5,9,2,6,5,3], then n is 10 and m is 7. In this case, b contains [3,1,4,5,9,2,6], and c contains [2,2,1,2,1,1,1].

   (a) The following program is a program to compute b and c from a. Describe the computational complexity using n and m. Note that the parameters b and c are supposed to be arrays of size m. We suppose that each entry in array b is initialized to be 0.

```
def intcount(a, b, c)
  for i in 0..(a.length()-1)
    x = a[i]
    j = 0
    while b[j] != 0 && b[j] != x
      j = j + 1
    end
    if b[j] == 0
      b[j] = x
      c[j] = 1
    else
      c[j] = c[j] + 1
    end
  end
end
```

Try to execute the program
with the above a, b, and c

Continued.

(b) Suppose that a is sorted, that is, elements in a is ordered in nondecreasing order. Modifying the above program, make a new function intcount(a,b,c) that runs in $O(n)$ time.

Ex. a=[10, 10, 9, 8, 8, 6, 6, 6, 3, 3, 2, 2, 1]

Hint: suffices to detect the change of numbers in a

```
def intcount(a, b, c)
    Write something
  for i in 0..(a.length()-1)
    Write something
  end
end
```

Deadline of Today’s Exercise

- **By Dec. 14（Wed） 23:59**
  - Explain how you obtain solutions, not only solutions
  - It is OK to submit a hand-written one if you want
    - Write neatly
    - Recommend to scan it and submit through ITC-LMS
    - Or hand in it during the class

Next Session: **Dec. 12**

- Sorting elements
  - Simple sort
  - Merge sort

- No class on Dec. 26
  - Make up on Jan.13 (finish early)
    - Planning to finish the lecture part by Jan 7, but some parts may be in Jan 13
      (Answers of last quizzes, etc)

- Remaining:
  - Dec 12, 19, Jan 7, 13