Algorithms for Big Data
CISC5835
Fordham Univ.

Instructor: X. Zhang

Lecture 1

Outline

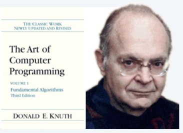
- What is algorithm: word origin, first algorithms, algorithms of today's world
  - Sequential algorithms, Parallel algorithms, approximation algorithms, randomized algorithms
- Scope of the course
- A few algorithms and pseudocode
- Introduction to algorithm analysis: fibonacci seq calculation
  - counting number of "computer steps"
  - recursive formula for running time of recursive algorithm
- Asymptotic notations
- Algorithm running time classes: P, NP

What are Algorithms?

> *" A procedure for solving a mathematical problem (as of finding the greatest common divisor) in a finite number of steps that frequently involves repetition of an operation. "* — *webster.com*

Merriam-Webster

> *" An algorithm is a finite, definite, effective procedure, with some input and some output. "*
>
> — *Donald Knuth*



Etymology. [Knuth, TAOCP]

- *Algorism* = process of doing arithmetic using Arabic numerals.
- A misperception: *algiros* [painful] + *arithmos* [number].
- True origin: Abu 'Abd Allah Muhammad ibn Musa al-Khwarizm was a famous 9th century Persian textbook author who wrote *Kitāb al-jabr wa'l-muqābala*, which evolved into today's high school algebra text.

Goal/Scope of this course

- Goal: provide essential algorithmic background for MS Data Analytics students
  - algorithm analysis: space and time efficiency of algorithms
  - classical algorithms (sorting, searching, selection, graph…)
  - algorithms for big data
  - algorithms implementation in Python
- We will not cover:
  - Machine Learning algorithms (topics for Data Mining, Machine Learning courses)
  - Implementing algorithms in big data cluster environment is left to Big Data Programming

---

Part I: computer algorithms

- a general foundations and background for computer science
  - understand difficulty of problems (P, NP…)
  - understand key data structure (hash, tree)
  - understand time and space efficiency of algorithm
  - Basic algorithms:
    - sorting, searching, selection algorithms
    - algorithmic paradigm: divide & conquer, greedy, dynamic programming, randomization
    - Hashing and universal hashing
    - Graph algorithms/Analytics (path/connectivity/community/centrality analysis)
  - Assumption: whole input can be stored in main memory (organized using some data structure…)

Part II: Big Data Algorithms

- Big Data: volume is too big to be stored in main memory of a single computer
- This class:
  - Stream: m elements from universe of size n, $< x_1, x_2, ..., x_m >= 3, 5, 3, 7, 5, 4, ...$
  - Goal: compute a function of stream (e.g, counting, median, longest increasing sequence…)
    - limited working memory, sublunar in n and m
    - access data sequentially (each element can be accessed only once)
    - process each element quickly
  - Matrix operations and algorithms: for large matrices
  - Such algorithms are randomized and approximate

Outline

- What is algorithm: word origin, first algorithms, algorithms of today's world
- Scope of the course
- A few algorithms and pseudocode
- Introduction to algorithm analysis: fibonacci seq calculation
  - counting number of "computer steps"
  - recursive formula for running time of recursive algorithm
- Asymptotic notations
- Algorithm running time classes: P, NP

Oldest Algorithms

- Al Khwarizmi laid out basic methods for
  - adding, multiplying and dividing numbers
  - extracting square roots
  - calculating digits of pi, …
- These procedures were precise, unambiguous, mechanical, efficient, correct. i.e., they were algorithms, a term coined to honor Al Khwarizmi after decimal system was adopted in Europe many centuries later.

Example: Selection Sort

- **Input**: a list of elements, L[1…n]
- **Output**: rearrange elements in List, so that L[1]<=L[2]<=L[3]<…L[n]
  - Note that “list” is an ADT (could be implemented using array, linked list)
- Ideas (in two sentences)
  - First, find location of smallest element in sub list L[1…n], and swap it with first element in the sublist
  - repeat the same procedure for sublist L[2…n], L[3…n], …, L[n-1…n]

Selection Sort (idea=>pseudocode)

```
for i=1 to n-1
     // find location of smallest element in sub list L[i…n]
     minIndex = i;
     for k=i+1 to n
          if L[k]<L[minIndex]: minIndex=k

     //swap it with first element in the sublist
     if (minIndex!=i)
          swap (L[i], L[minIndex]);

   // Correctness: L[i] is now the i-th smallest element
```

Introduction to algorithm analysis

- Consider calculation of Fibonacci sequence, in particular, the n-th number in sequence:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

Leonardo Fibonacci (c. 1170 – c. 1250)

Fibonacci helped the spread of the decimal system in Europe, primarily through the publication in the early 13th century of his Book of Calculation, the Liber Abaci. (Source: Wikipedia)

Fibonacci Sequence

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
- Formally,

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0. \end{cases}$$

- Problem: How to calculate n-th term, e.g., what is F_{100}, F_{200}?

A recursive algorithm

```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n - 1) + fib1(n - 2)
```

Observation: we reduce a large problem into two smaller problems

- Three questions:
  - Is it correct?
    - yes, as the code mirrors the definition…
  - Resource requirement: How fast is it? Memory requirement?
  - Can we do better? (faster?)

Outline

- What is algorithm: word origin, first algorithms, algorithms of today’s world
- Scope of the course
- A few algorithms and pseudocode
- Introduction to algorithm analysis: fibonacci seq calculation
  - counting number of “computer steps”
  - recursive formula for running time of recursive algorithm
- Asymptotic notations
- Algorithm running time classes: P, NP

Efficiency of algorithms

- We want to solve problems using less resource:
  - **Space**: how much (main) memory is needed?
  - **Time**: how fast can we get the result?
- **Usually, the bigger input, the more memory it takes and the longer it takes**
  - it takes longer to calculate 200-th number in Fibonacci sequence than the 10th number
  - it takes longer to sort larger array
  - it takes longer to multiple two large matrices
- **Efficient algorithms are critical for large input size/problem instance**
  - Finding F_{100}, Searching Web …
- Two different approaches to evaluate efficiency of algorithms: **Measurement vs. analysis**

Experimental approach

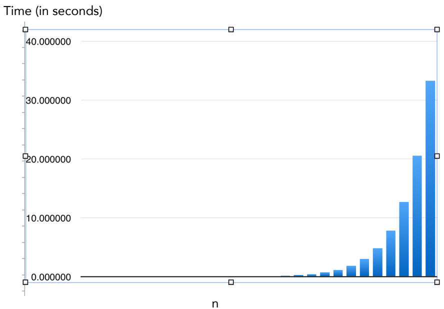
- **Measure how much time elapses from algorithm starts to finishes**
- needs to implement, instrument and deploy

e.g.,

```
import time
….
start_time = time.time()
 BubbleSort (listOfNumbers)  # any code of yours
 end_time = time.time()
 elapsed_time = end_time - start_time
```

Example (Fib1: recursive)

```
n   T(n)ofFib1    F(n)
10      3e-06    55
11      2e-06    89
12      4e-06    144
13      7e-06    233
14      1.1e-05    377
15      1.7e-05    610
16      2.9e-05    987
17      4.7e-05    1597
18      7.6e-05    2584
19      0.000122    4181
20      0.000198    6765
21      0.000318    10946
22      0.000515    17711
23      0.000842    28657
24      0.001413    46368
25      0.002261    75025
26      0.003688    121393
27      0.006264    196418
28      0.009285    317811
29      0.014995    514229
30      0.02429    832040
31      0.039288    1346269
32      0.063543    2178309
33      0.102821    3524578
34      0.166956    5702887
35      0.269394    9227465
36      0.435607    14930352
37      0.701372    24157817
38      1.15612    39088169
39      1.84103    63245986
40      2.9964    102334155
41      4.85536    165580141
42      7.85187    267914296
43      12.6805    433494437
44      20.513    701408733
```



Running time seems to grows exponentially as n increases

Experimental approach

- results are realistic, specific and random
  - specific to language, run time system (Java VM, OS), caching effect, other processes running
  - possible to perform model-fitting to find out T(n): running time of the algorithms given input size
- Cons:
  - time consuming, maybe too late
  - Does not explain why?
- Measurement is important for a "production" system/ end product; but not informative for algorithm efficiency studies/comparison/prediction

Analytic approach

- Is it possible to find out how running time grows when input size grows, **analytically?**
  - Does running time stay constant, increase linearly, logarithmically, quadratically, … exponentially?
- Yes: analyze pseudocode/code to calculate total number of steps in terms of input size, and study its order of growth
  - results are general: not specific to language, run time system, caching effect, other processes sharing computer

Running time analysis

- **Given an algorithm in pseudocode or actual program**
- When the input size is n, what is the total number of computer steps executed by the algorithm, T(n)?
  - **Size of input**: size of an array, polynomial degree, # of elements in a matrix, vertices and edges in a graph, or # of bits in the binary representation of input
  - Computer steps: **arithmetic operations, data movement, control, decision making (if, while), comparison,…**
    - each step take a **constant** amount of time
- Ignore: overhead of function calls (call stack frame allocation, passing parameters, and return values)

Case Studies: Fib1(n)

```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n - 1) + fib1(n - 2)
```

- Let T(n) be number of computer steps needed to compute fib1(n)
  - T(0)=1: when n=0, first step is executed
  - T(1)=2: when n=1, first two steps are executed
  - For n >1, **T(n)=T(n-1)+T(n-2)+3**: first two steps are executed, fib1(n-1) is called (with T(n-1) steps), fib1(n-2) is called (T(n-2) steps), return values are added (1 step)
- Can you see that T(n) > F_n ?

Running Time analysis

```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n - 1) + fib1(n - 2)
```

- Let *T(n) be number of computer steps to compute fib1(n)*
  - **T(0)=1**
  - **T(1)=2**
  - **T(n)=T(n-1)+T(n-2)+3, n>1**
- Analyze running time of recursive algorithm
  - first, write a recursive formula for its running time
  - then, recursive formula => closed formula, asymptotic result

---

Fibonacci numbers

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

$$F_n \geq 2^{\frac{n}{2}} = 2^{0.5n}$$

- Fn is **lower bounded by** $2^{0.5n}$
- In fact, there is a tighter lower bound $2^{0.694n}$
- Recall *T(n): number of computer steps to compute fib1(n),*
  - **T(0)=1**
  - **T(1)=2**
  - **T(n)=T(n-1)+T(n-2)+3, n>1**

$$T(n) > F_n \geq 2^{0.694n}$$

Exponential running time

- Running time of Fib1: $T(n) > 2^{0.694n}$
- Running time of Fib1 is exponential in n
  - calculate F_{200}, it takes at least 2^{138} computer steps
- On NEC Earth Simulator (fastest computer 2002-2004)
  - Executes 40 trillion (10^{12}) steps per second, 40 teraflots
  - Assuming each step takes same amount of time as a “floating point operation”
  - Time to calculate F_{200}: at least 2^{92} seconds, i.e., 1.57×10^{20} years
- Can we throw more computing power to the problem?
  - Moore’s law: computer speeds double about every 18 months (or 2 years according to newer version)

Exponential running time

- Running time of Fib1: $T(n) > 2^{0.694n} = 1.6177^n$
- Moore’s law: computer speeds double about every 18 months (or 2 years according to newer version)
  - If it takes fastest CPU of this year 6 minutes to calculate F_{50},
  - fastest CPU in two years from today can calculate F_{52} in 6 minutes
- Algorithms with exponential running time are not efficient, not scalable
  - not practical solution for large input

Can we do better?

```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n - 1) + fib1(n - 2)
```

- Draw recursive function call tree for fib1(5)
- Observation: wasteful repeated calculation
- Idea: Store solutions to subproblems in array (key of Dynamic Programming)

```
function fib2(n)
if n = 0 return 0
create an array f[0...n]
f[0] = 0, f[1] = 1
for i = 2...n:
    f[i] = f[i - 1] + f[i - 2]
return f[n]
```

Running time fib2(n)

```
function fib2(n)
if n = 0 return 0
create an array f[0...n]
f[0] = 0, f[1] = 1
for i = 2...n:
    f[i] = f[i - 1] + f[i - 2]
return f[n]
```

- Analyze running time of iterative (non-recursive) algorithm:

  T(n)=1 // if n=0 return 0
  +n // create an array of f[0…n]
  +2 // f[0]=0, f[1]=1
  +(n-1) // for loop: repeated for n-1 times
  = 2n+2
- T(n) is a linear function of n, or fib2(n) has linear running time

---

Alternatively…

```
function fib2(n)
if n = 0 return 0
create an array f[0...n]
f[0] = 0, f[1] = 1
for i = 2...n:
    f[i] = f[i - 1] + f[i - 2]
return f[n]
```

Estimation based upon CPU:
takes 1000us,
takes 200n us
each assignment takes 60us
addition and assignment takes 800us…

- How long does it take for fib2(n) finish?

  T(n)=1000 +200n+2*60+(n-1)*800=1000n+320 // in unit of *us*
- Again: T(n) is a linear function of n
  - Constants are not important: different on different computers
  - System effects (caching, OS scheduling) makes it pointless to do such fine-grained analysis anyway!
- Algorithm analysis focuses on how running time grows as problem size grows (constant, linear, quadratic, exponential?)
  - not actual real world time

Summary: Running time analysis

- **Given an algorithm in pseudocode or actual program**
- When the input size is n, how many total number of computer steps are executed?
  - **Size of input**: size of an array, polynomial degree, # of elements in a matrix, vertices and edges in a graph, or # of bits in the binary representation of input
  - Computer steps: **arithmetic operations, data movement, control, decision making (if, while), comparison,…**
    - each step take a **constant** amount of time
- Ignore:
  - Overhead of function calls (call stack frame allocation, passing parameters, and return values)
  - Different execution time for different steps

Time for exercises/examples

1. Reading algorithms in pseudocode
2. Writing algorithms in pseudocode
3. Analyzing algorithms

Algorithm Analysis: Example

- What's the running time of MIN?

Algorithm/Function.: MIN (a[1…n])

input: an array of numbers a[1…n]

output: the minimum number among a[1…n]

```
  m = a[1]
  for i=2 to n:
        if a[i] < m: m = a[i]
  return m
```

  - How do we measure the size of input for this algorithm?
  - How many computer steps when the input's size is n?

Algorithm Analysis: bubble sort

Algorithm/Function.: bubblesort (a[1…n])

input: a list of numbers a[1…n]

output: a sorted version of this list

```
   for endp=n to 2:
         for i=1 to endp-1:
                     if a[i] > a[i+1]:  swap (a[i], a[i+1])
  return a
```

- How do you choose to measure **the size of input**?
  - length of list a, i.e., n
  - the longer the input list, the longer it takes to sort it
- **Problem instance**: a particular input to the algorithm
  - e.g., a[1…6]={1, 4, 6, 2, 7, 3}
  - e.g., a[1…6]={1, 4, 5, 6, 7, 9}

Algorithm Analysis: bubble sort

Algorithm/Function.: bubblesort (a[1…n])
input: an array of numbers a[1…n]
output: a sorted version of this array

```
    for endp=n to 2:
         for i=1 to endp-1:
                  if a[i] > a[i+1]: swap (a[i], a[i+1])
  return a
```

a compute step

- endp=n: inner loop (for j=1 to endp-1) repeats for n-1 times
- endp=n-1: inner loop repeats for n-2 times
- endp=n-2: inner loop repeats for n-3 times
- …
- endp=2: inner loop repeats for 1 times
- Total # of steps: T(n) = (n-1)+(n-2)+(n-3)+…+1=n(n-1)/2

Matrix and Vector

Matrix: a 2D (rectangular) array of numbers, symbols, or expressions, arranged in rows and columns.

e.g., a 2 × 3 matrix $B= \begin{bmatrix} 1 & 9 & -13 \\ 20 & 5 & -6 \end{bmatrix}.$

a $m \times n$ matrix $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$

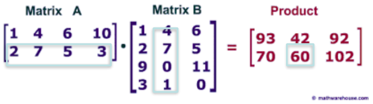
Each element of a matrix is denoted by a variable with two subscripts, $A_{2,1}$ element at second row and first column of a matrix A

Row vector of a matrix is a vector made up of a row of elements from the matrix: [1 9 -13] is a row vector of B

Column vector of a matrix is a vector made up of a column of elements

Matrix Multiplication

Matrix Multiplication: Dimension of A, B, and A x B?



$$\begin{bmatrix} 1 & 4 & 6 & 10 \\ 2 & 7 & 5 & 3 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 & 6 \\ 2 & 7 & 5 \\ 9 & 0 & 11 \\ 3 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 93 & 42 & 92 \\ 70 & \boxed{60} & 102 \end{bmatrix}$$

The (i,j) element of AB is the dot product of i-th row of A with the j-th column of B

*$C_{2,2}$=[2 7 5 3] x [4 7 0 1] = 2*4+7*7+5*0+3*1=60*

$$[\mathbf{AB}]_{i,j} = A_{i,1}B_{1,j} + A_{i,2}B_{2,j} + \cdots + A_{i,n}B_{n,j} = \sum_{r=1}^{n} A_{i,r}B_{r,j},$$

Matrix Multiplication

Matrix Multiplication:

Dimension of A, B, and A x B?

Matrix A · Matrix B = Product

$$\begin{bmatrix}1&4&6&10\\2&7&5&3\end{bmatrix}\cdot\begin{bmatrix}1&4&6\\2&7&5\\9&0&11\\3&1&0\end{bmatrix}=\begin{bmatrix}93&42&92\\70&60&102\end{bmatrix}$$

© mathwarehouse.com

$$[\mathbf{AB}]_{i,j} = A_{i,1}B_{1,j} + A_{i,2}B_{2,j} + \cdots + A_{i,n}B_{n,j} = \sum_{r=1}^{n} A_{i,r}B_{r,j},$$

Total (scalar) multiplication: 4x2x3=24

```
MATRIX-MULTIPLY(A, B)
if A.columns ≠ B.rows
    error "incompatible dimensions"
else let C be a new A.rows × B.columns matrix
    for i = 1 to A.rows
        for j = 1 to B.columns
            $c_{ij} = 0$
            for k = 1 to A.columns
                $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
    return C
```

Total (scalar) multiplication: n_2xn_1xn_3

Algorithm Analysis: Binary Search

Algorithm/Function.: search (a[L…R], value)
input: a list of numbers a[L…R] sorted in ascending order, a number value
output: the index of value in list a (if value is in it), or -1 if not found

```
if (L>R): return -1
m = (L+R)/2
if (a[m]==value):
    return m
else:
    if (a[m]>value):
        return search (a[L…m-1], value)
else:
    return search (a[m+1…R], value)
```

- What's the size of input in this algorithm?
    - length of list a[L…R]

Algorithm Analysis: Binary Search

Algorithm/Function.: search (a[L…R], value)
input: a list of numbers a[L…R] sorted in ascending order, a number value
output: the index of value in list a (if value is in it), or -1 if not found

```
if (L>R): return -1
m = (L+R)/2
if (a[m]==value):
    return m
else:
    if (a[m]>value):
        return search (a[L…m-1], value)
else:
    return search (a[m+1…R], value)
```

- Let T(n) be number of steps to search an list of size n
    - best case (value is in middle point), T(n)=3
    - **worst case (when value is not in list) provides an upper bound**

Algorithm Analysis: Binary Search

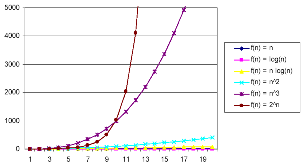
```
Algorithm/Function.: search (a[L…R], value)
input: a list of numbers a[L…R] sorted in ascending order, a number value
output: the index of value in list a (if value is in it), or -1 if not found
  if (L>R): return -1
  m = (L+R)/2
  if (a[m]==value):
     return m
  else:
    if (a[m]>value):
          return search (a[L…m-1], value)
  else:
     return search (a[m+1…R], value)
```

- Let T(n) be number of steps to search an list of size n in worst case
  - T(0)=1 //base case, when L>R
  - T(n)=3+T(n/2) //general case, reduce problem size by half
- Next chapter: master theorem solving T(n)=$\log_2 n$

Outline

- What is algorithm: word origin, first algorithms, algorithms of today's world
  - Sequential algorithms, Parallel algorithms, approximation algorithms, randomized algorithms
- Scope of the course
- A few algorithms and pseudocode
- Introduction to algorithm analysis: fibonacci seq calculation
  - counting number of "computer steps"
  - recursive formula for running time of recursive algorithm
- Asymptotic growth rate, big-O notations
- Algorithm running time classes: P, NP

Growth Rate of functions



- **Growth rate**: How fast f(x) increases as x increases
  - slope (derivative)

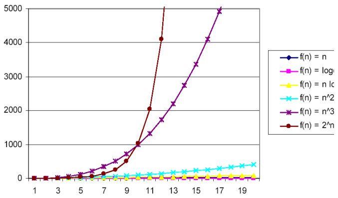
$$\frac{f(x+\Delta x)-f(x)}{\Delta x}$$

- f(x)=2x: constant growth rate (slope is 2)
- $f(x) = 2^x$: growth rate increases as x increases (see figure above)
- $f(x) = log_2 x$: growth rate decreases as x increases

Derivatives of Common Functions

Common Functions	Function	Derivative
Constant	c	0
Line	x	1
	ax	a
Square	x^2	2x
Square Root	√x	$(½)x^{-½}$
Exponential	e^x	e^x
	a^x	$\ln(a)\ a^x$
Logarithms	ln(x)	1/x
	$\log_a(x)$	1 / (x ln(a))

Asymptotic Growth Rate of functions



- **Asymptotic Growth rate**: growth rate of function when $x \to \infty$
  - slope (derivative) when x is very big
- The larger asym. growth rate, the larger f(x) when $x \to \infty$

- e.g., f(x)=2x: asymptotic growth rate is 2
- $f(x) = 2^x$: very big!

(Asymptotic) Growth rate of functions of n (from low to high):
log(n) < n < nlog(n) < n^2 < n^3 < n^4 < ….< 1.5^n < 2^n < 3^n

---

Compare Growth Rate of functions(2)

- Two sorting algorithms:
  - yours: $2n^2 + 100n$
  - your friend: $2n^2$
- Which one is better (for large arrays)?
  - evaluate their ratio when n is large

$$\frac{2n^2 + 100n}{2n^2} = 1 + \frac{100n}{2n^2} = 1 + \frac{50}{n} \to 1, \text{ when } n \to \infty$$

They are same! In general, the lower order term can be dropped.

Focus on Asymptotic Growth Rate

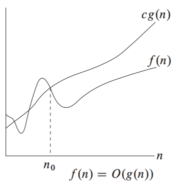
- In answering “How fast T(n) grows as n grows?”, leave out
  - **lower-order terms**
  - constant coefficient: not reliable info. (arbitrarily counts # of computer steps), and hardware difference makes them not important
  - *Note: you still want to optimize your code to bring down constant coefficients. It’s only that they don’t affect “asymptotic growth rate”*
- e.g. bubble sort executes $T(n) = \frac{n(n-1)}{2} = \frac{n^2-n}{2}$ steps to sort a list of n elements
  - bubble sort’s running time, T(n)’s (asymptotic) growth rate is same as n^2, i.e., $T(n) = \Theta(n^2)$
  - bubble sort has a quadratic running time

Outline

- What is algorithm: word origin, first algorithms, algorithms of today’s world
  - Sequential algorithms, Parallel algorithms, approximation algorithms, randomized algorithms
- Scope of the course
- A few algorithms and pseudocode
- Introduction to algorithm analysis: fibonacci seq calculation
  - counting number of “computer steps”
  - recursive formula for running time of recursive algorithm
- Asymptotic growth rate, big-O notations
- Algorithm running time classes: P, NP

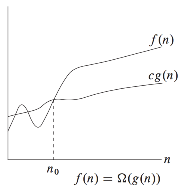
Big-O notation

- f(n) and g(n): two functions from positive integers to positive real numbers



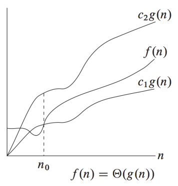
f grows no faster than g, g is asymptotic upper bound of f

$GR(f) \le GR(g)$



f grows no slower than g, g is asymptotic lower bound of f

$GR(f) \ge GR(g)$



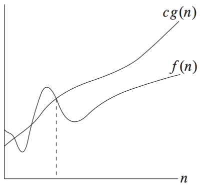
f grows no slower and no faster than g, f grows at same rate as g

$GR(f) == GR(g)$

Big-O notation

- f=O(g) if there is a constant c>0 and n_0, such that for all $n>n_0$, $f(n) \le c \cdot g(n)$
  - f(n) is smaller than some positive constant times g(n) for all n that is large enough
- e.g., $f(n)=100n^2$, $g(n)=n^3$

$$\frac{f(n)}{g(n)} = \frac{100n^2}{n^3} = \frac{100}{n} \le 100$$



$f(n) = O(g(n))$

f(n)=O(g(n)), as there exists c=100, n_0=1, such that for all $n>n_0$, f(n)<=c*g(n)

Looking to bound $\frac{f(n)}{g(n)}$ by a positive constant for all n large enough…

- *Some books write* $f \in O(g)$
  - *O(g) denotes the set of all functions h(n) for which there is a constant c>0, such that* $h(n) \le c \cdot g(n)$

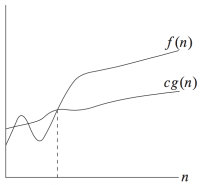
Big-O: Exercise

- For the following four pairs of f(), g(), is f(n)=O(g(n)) ?
  - f(n)=1, g(n)=2n
  - $f(n)=100n^2+8n$, $g(n)=n^2$
  - f(n)=nlog(n), $g(n)=n^2$
  - $f(n) = 2^n, g(n) = 3^n$
  - $f(n) = \frac{(n-1)n}{2}, g(n) = n$

Big-Ω notations

- Consider this pairs of f, g:

$$f(n) = \frac{(n-1)n}{2}, g(n) = n$$



$f(n) = \Omega(g(n))$

  - f(n)=O(g(n)) is not true:

$$\frac{f(n)}{g(n)} = \frac{n-1}{2}$$

  - impossible to find c, n_0, s.t., for all $n>n_0$, $\frac{f(n)}{g(n)} \le c$
  - instead, let c=0.5, n_0=2, then for all $n>=n_0$, $\frac{f(n)}{g(n)} = \frac{n-1}{2} \ge \frac{1}{2}$
- f(n) grows no slower than g(n), i.e., f=Ω(g) (g is asymptotic lower bound of f)
  - if and only if there is a positive constant c, n_0, such that for all n, $f(n) \ge c \cdot g(n)$

Big-Ω notations Exercises

- For following pairs of f(n), g(n), is $f(n) = \Omega(g(n))$
  - $f(n)=100n^2$, $g(n)=n$
  - $f(n)=100n^2+8n$, $g(n)=n^2$
  - $f(n)=2^n$, $g(n)=n^8$

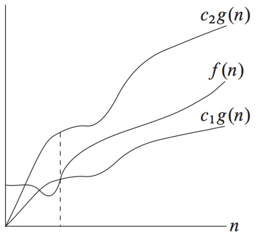
Big-Θ notations

- Consider $f(n)=100n^2+8n$, $g(n)=n^2$

$f(n) = O(g(n)), f(n) = \Omega(g(n))$

- i.e., f grows no faster, an no slower faster than g, f grows a same rate as g asymptotically
- We denote this as $f = \Theta(g)$
  - Def: there are constants c_1, c_2, $n_0>0$, s.t.,

$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \text{ for any } n \geq n_0$



$f(n) = \Theta(g(n))$

f can be sandwiched between g by two constant factors

Big-Θ Exercise

- For following pairs of f and g, is $f(n) = \Theta(g(n))$?
  - (1) $f(n)=10000n^2$, $g(n)=n^2$
  - (2) $f(n) = \frac{0.684c}{2}(n^2+n-2)+n+3, g(n) = n^2$
  - (3) $f(n) = \log_2 n, g(n) = log_{10}n$

mini-summary

- in analyzing running time of algorithms, what's important is scalability (perform well for large input)
  - focus on higher order which dominates lower order parts
    - a three-level nested loop dominates a single-level loop
  - multiplicative constants can be omitted: $14n^2$ becomes n^2

$$14n^2 = \Theta(n^2)$$

  - n^a dominates n^b if a>b, e.g., $n^3 = \Omega(n^{2.5})$
  - any exponential dominates any polynomial:
    - 3^n dominates n^5 $3^n = \Omega(n^5)$
    - any polynomial dominates any logarithms: n dominates $(\log n)^3$
- E.g., $T(n) = 0.56n^3 + 10000n + 0.45 \cdot 3^n = \Theta(3^n)$

Outline

- What is algorithm: word origin, first algorithms, algorithms of today's world
  - Sequential algorithms, Parallel algorithms, approximation algorithms, randomized algorithms
- Scope of the course
- A few algorithms and pseudocode
- Introduction to algorithm analysis: fibonacci seq calculation
  - counting number of "computer steps"
  - recursive formula for running time of recursive algorithm
- Asymptotic growth rate and big-O notations
- Problem complexity class: P, NP

Typical Running Time

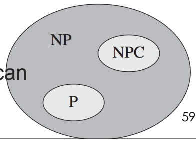
- 1 (constant running time):
  - Instructions are executed once or a few times
- log(n) (logarithmic), e.g., binary search
  - A big problem is solved by cutting original problem in smaller sizes, by a constant fraction at each step
- n (linear): linear search, calculate mean, variance, …
  - A small amount of processing is done on each input element
- n log(n): merge sort
  - A problem is solved by dividing it into smaller problems, solving them independently and combining the solution

Typical Running Time Functions

- n^2 (quadratic): bubble sort
  - Typical for algorithms that process all pairs of data items (double nested loops)
- n^3 (cubic)
  - matrix multiplication
- n^K (polynomial)
- $2^{0.694n}$ (exponential): Fib1
- 2^n (exponential):
  - Few exponential algorithms are appropriate for practical use
- 3^n (exponential), …

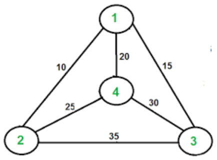
P=NP?

- P: the set of problems that have known polynomial algorithms
- NP: the set of problems for which there exists a polynomial alg. to verify a solution
  - Many NP problems have no polynomial time algorithms … yet, despite intensive research by many
  - Will we ever find one? Not likely…
    - we've tried a long time
    - many problems in NPC (if we can
    - solve one in polynomial, then we can solve all others in polynomial.



NPC: Traveling Salesman Problem

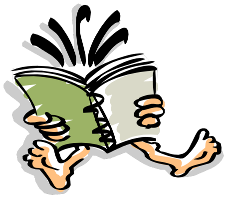
- Given n vertices 1, . . . , n, and all $n(n-1)/2$ distances between them, as well as a budget b.
- Output: find a tour (a cycle that passes through every vertex exactly once) of total cost b or less – or to report that no such tour exists.
- TSP as a search problem
  - given an instance, find a tour within the budget (or report that none exists).
- Usually, TSP is posed as optimization problem
  - find shortest possible tour
  - 1->2->3->4, total cost: 60
- TSP is NP problem



Summary

- This class focused on algorithm running time analysis
- start with running time function, expressing number of computer steps in terms of input size
- Focus on very large problem size, i.e., asymptotic running time
  - big-O notations => focus on dominating terms in running time function
  - Constant, linear, polynomial, exponential time algorithms …
  - NP, NP complete problem

Assignment



- Lab1
- Chapter 0 of DPV