CS 2351 Data Structures

Introduction to Algorithms

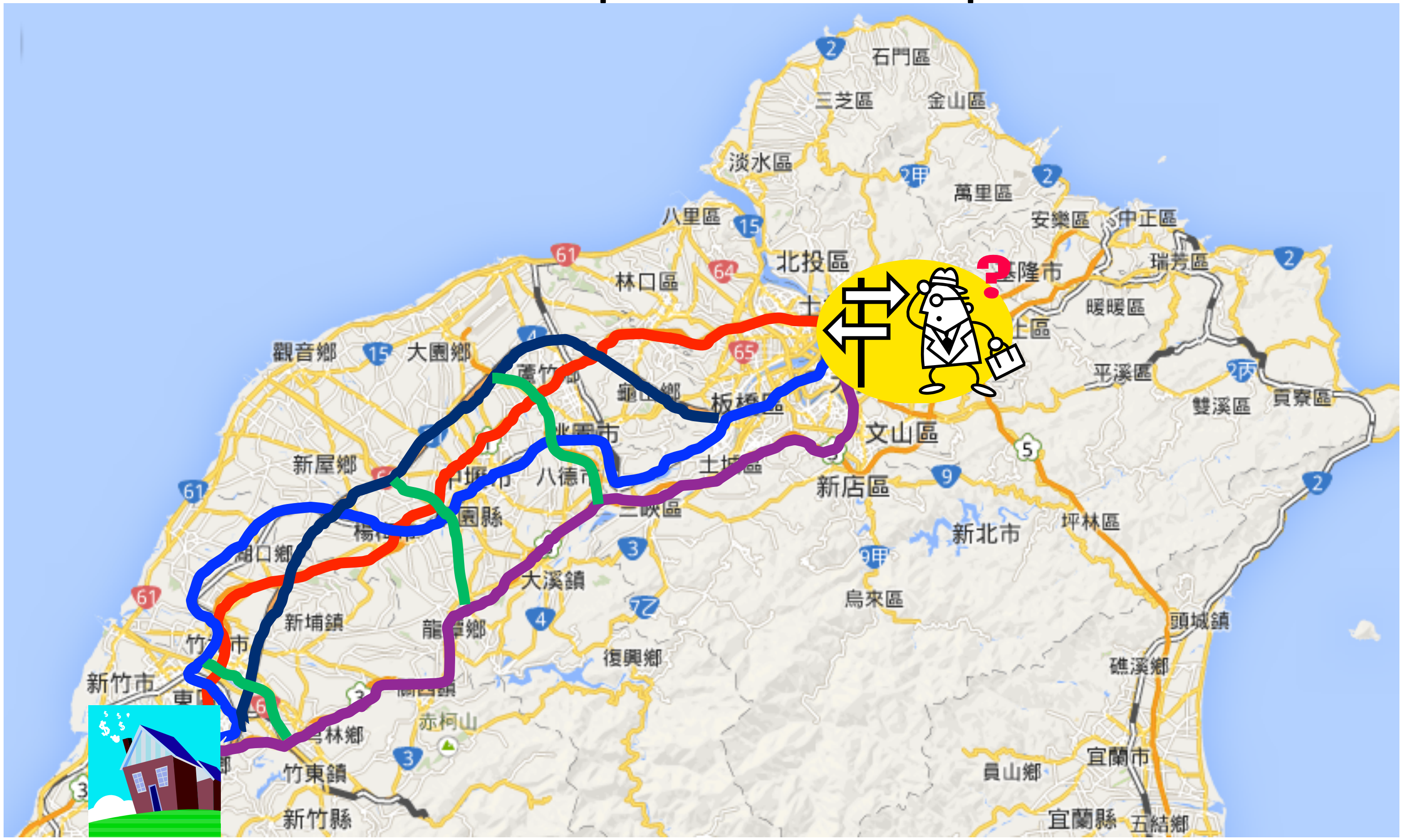
Prof. Chung-Ta King
Department of Computer Science
National Tsing Hua University

Outline

- Data structure and algorithm
  - What is algorithm? How to specify algorithms?
- Designing algorithms
  - Divide-and-conquer, recursion
- Performance, analysis and measurement
  - Concept of Big-O

You have a Problem to Solve

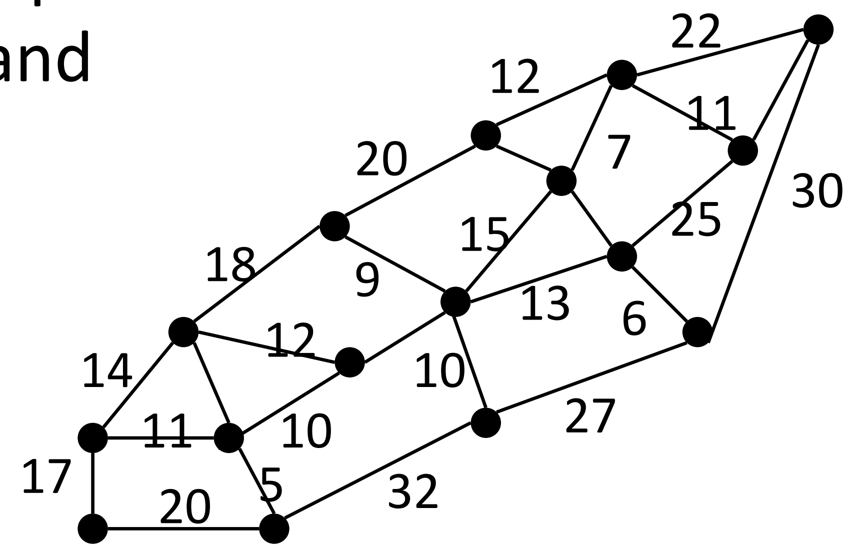
- What is the shortest path from Taipei to Hsinchu?



To Solve the Problem on a Computer

- You must formulate the problem so that the computer can understand
  - What are the inputs?
  - What are the outputs?
  - How to represent and structure the data?

    Data Structure



- You then tell the computer how to solve the problem step-by-step

This is called algorithm

What Is Algorithm?

- An algorithm is a finite set of instructions to solve a computational problem:
  - Must specify every step completely, so a computer can implement it without any further "understanding"
  - Must work for all possible inputs of the problem
  - May have many different algorithms for a problem
- An algorithm must be:
  - *Definiteness*: each instruction is clear and unambiguous
  - *Finiteness*: terminate after a finite number of steps
  - *Effectiveness*: every instruction must be basic and easy to be computed

Algorithm and Data Structure

- When developing an algorithm, the input data and intermediate results must be organized and stored in some structures → data structure
  - It is important to design the data structures and associated operations to provide natural and efficient support for the most important steps in the algorithm, e.g. finding a data
- Selecting a data structure to solve a problem:
  - Analyze your problem to determine the basic operations that must be supported
  - Quantify the resource constraints for each operation
  - Select the data structure best meets these requirements

Algorithm and Efficiency

- As computer scientists, we strive for efficient algo.
- Two aspects of efficiency:
  - good/optimal
    The algorithm produces an ~~efficient~~ output/solution, e.g., find the shortest path from Taipei to Hsinchu

    ← the original problem is often an *optimization* problem
  - The algorithm produces the output/solution efficiently, e.g., sort a set of numbers in the shortest time, find the shortest path from Taipei to Hsinchu in the shortest time

End result versus process/method

- We are more concerned of the efficiency that an algorithm can produce the solution

Algorithm and Efficiency

- An algorithm is *efficient* if it solves the problem within required resource constraints, e.g. time, space
  - Some algorithms solve the problem but are not efficient
- To develop efficient algorithms
  - We must first analyze the problem to determine the performance goals that must be achieved
  - Select the right data structure
  - Work out the algorithm and prove it correct
  - Analyze and estimate performance of resultant algorithm (to be discussed later) to see if perf. goals are achieved

How to Specify Algorithms?

- Natural languages
  - English, Chinese, ...etc.
  - A lot of sentences...
- Graphic representation
  - Flowchart
    - Feasible only if the algorithm is small and simple
- Programming language + few English
  - C++
    - Concise and effective!

Example: Search through a Sorted List

- The problem:
  - Input: $n \geq 1$ distinct integers that are *sorted* in array A[0] … A[n-1], an integer x
  - Output: if x=A[j], return index j
    otherwise return -1

Data structure

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
A	1	3	5	8	9	17	32	50

Ex. For x=9, return index 4
For x=10, return -1

Binary Search Algo in Natural Language

- Let *left* and *right* denote the left and right end indices of the list with initial value 0 and *n*-1
- Let *middle* = (*left*+*right*)/2 be the middle position
- Compare A[*middle*] with *x* and obtain three results:
  - *x* < A[*middle*]: *x* must be somewhere between 0 and *middle*-1 → set *right* to *middle*-1
  - *x* == A[*middle*]: return *middle*
  - *x* > A[*middle*]: *x* must be somewhere between *middle*+1 and *n*-1 → set *left* to *middle*+1
- If *x* is not found and there are still integers to check, recalculate *middle* and repeat above comparisons

Binary Search Algo in C++ and English

```
int BinarySearch(int *A, const int x, const int n)
{ int left=0, right=n-1;

  while (left <= right)
  { // more integers to check
    int middle = (left+right)/2;
    if (x < A[middle])  right = middle-1;
    else if (x > A[middle])  left = middle+1;
    else return middle;
  } // end of while
  return -1; // not found
}
```

Outline

- Data structure and algorithm
- Designing algorithms
  - Divide-and-conquer, recursion
- Performance, analysis and measurement

Designing Algorithms

- There is no single recipe for inventing algorithms
- There are basic rules:
  - Understand your problem well – may require much mathematical analysis!
  - Use existing algorithms (reduction) or algorithmic ideas
- There is a single basic algorithmic technique:

  Divide and Conquer

  - In its simplest form it is simple induction: in order to solve a problem, solve a similar problem of smaller size
  - The key conceptual idea: *think about how to use the smaller solution to get the larger one*

(Prof. Noam Nisan, Hebrew University of Jerusalem)

Induction Expressed as Recursion

- To express induction-styled divide-and-conquer method, recursion is very handy
  - A recursive method is one that contains a call to itself
- Direct recursion:
  - Function calls itself directly
  - Ex.: funcA → funcA
- Indirect recursion:
  - Function A calls other functions that invoke function A
  - Ex.: funcA → funcB → funcA

From Iterative to Recursive

```
int BinarySearch(int *A, const int x, const int n)
{ int left=0, right=n-1;

  while (left <= right)
  { // more integers to check
    int middle = (left+right)/2;
    if (x < A[middle])  right = middle-1;
    else if (x > A[middle])  left = middle+1;
    else return middle;
  } // end of while
  return -1; // not found
}
```

Learn Using an Example

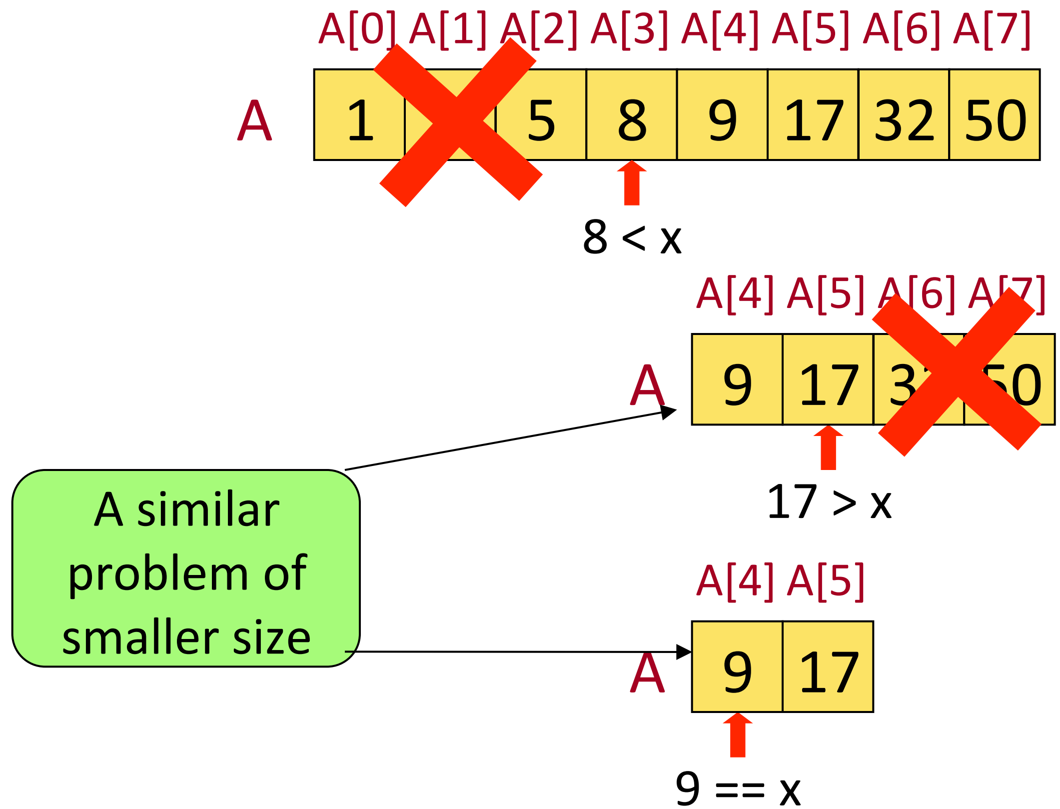
- Search for x=9 in array A[0], …, A[7]:

1st iteration:

2nd iteration:

3rd iteration:

return index 4



Recursive Binary Search

```
int BinarySearch(int *A, const int x,
                 const int left, const int right)
{ // Search A[left],..,A[right] for x
  if (left <= right) { // more to check
    int middle = (left+right)/2;
    if (x < A[middle])
       return BinarySearch(A,x,left,middle-1);
    else if (x > A[middle])
       return BinarySearch(A,x,middle+1,right);
    return middle;
  } // end of if
  return -1; // not found
}
```

Easy If Problem Recursively Defined

- Binomial coefficient

$$C(n, m) = \frac{n!}{m!\,(n-m)!}$$

can be computed by the recursive formula:

$$C(n, m) = C(n-1, m) + C(n-1, m-1)$$

where $C(0, 0) = C(n, n) = 1$

Recursive Binomial Coefficients

```
int BinoCoeff(int n, int m)
{
  // termination conditions
  if (m==n) then return 1;
  else if (m==0) then return 1;

  // recursive step
  else
    return BinoCoeff(n-1,m)+BinoCoeff(n-1,m-1);
}
```

Hints for Recursive Algorithms

To ensure a feasible recursive algorithm, you must stick to the following principles:

- Termination conditions:
  - Your function should return a value or stop at certain condition and stop calling itself
- Decreased parameters:
  - Your parameters should be continuously decreased so that each call brings us one step closer to a termination condition.

Outline

- Data structure and algorithm
- Designing algorithms
- Performance, analysis and measurement
  - Time/space complexity, asymptotic performance, concept of Big-O

You Are Given a Task/Problem

- Make 100,000 arrows in 3 days

× 10萬

You Are Considering Two Options

- Hire 1000 workers, each makes 100 arrows in 3 days, including find and chop the woods
- Borrow the arrows from your enemy (草船借箭)
  - Use 20 boats, each has 30 soldiers and 50 straw figures on the sides

Which option is better?

This is called

performance comparison

(We are comparing efficiency of algorithms)

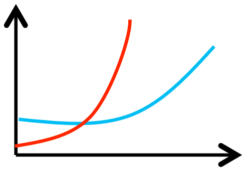
Before You Can Compare Performance

- You must define what you mean by "better"
  - Usually you use some quantitative values to express the goodness, called performance metrics
  - Common metrics: time (latency/throughput), space, power, cost, …
  - *How to define goodness for making arrows?*

- You must be able to analyze/measure performance
  - *How to analyze the cost of making arrows?*
  - *How to measure the cost of making arrows?*

Comparison May Be Meaningless

- If the problem size is too small, performance comparison may not be meaningful or even misleading
- Suppose you only need to make 10 arrows in 3 days
- How about 1000 arrows in 3 days?
- How about 100,000 arrows in 3 days?
  - Apparently, there is a break even point



The comparison is meaningful only if the problem size is large enough

To compare performance, you should consider very <u>large problem size</u> and focus on the effects of <u>growth rate</u>

(You should learn to sharpen your skills in developing large, robust programs)

Same for Comparing Program/Algorithm

- Criteria for performance:
  - Space complexity: How much memory space is used?
  - Time complexity: How much running time is needed?
  - Power/energy

  ➔ Must be considered against problem size
- Two approaches:
  - Performance analysis
    - Machine independent, a prior estimate
  - Performance measurement
    - Machine dependent, a posterior testing

Space Complexity

- Space complexity: $S(P) = C + S_P(I)$
- C is a fixed part:
  - Independent of the number and size of input and output
  - Including code space, space for simple variables, fixed-size structured variables, constants
- $S_P(I)$ is a variable part:
  - Dependent on the particular problem instance, or *Instance Characteristics (I)* (problem size)
  - Including space of referenced variables and recursion stack space

Analyzing Space Complexity

- Should concentrate on estimating $S_P(I)$
  → need to first determine how to quantize instance characteristics
  - Commonly used instance characteristic (I) include number and magnitude of **input** and **output** of the problem

Ex. 1: sorting(A[], n)
Then I= number of integers = n

Ex. 2: find the shortest path
Then I= number of nodes/edges in the graph

Space Complexity: Iterative Summing

```
float Sum(float *a, const int n)
{ float s = 0;
  for(int i=0; i<n; i++)
      s += a[i];
  return s;
}
```

- I = n (number of elements to be summed)
- C = code space + space for **a**, **n** = constant
- $S_{Sum}(I)$ = 0 (**a** stores only the address of array)

➔ S(Sum) = C + $S_{Sum}(I)$ = constant

Space Complexity: Recursive Summing

```
float Rsum(float *a, const int n)
{ float s = a[n-1];
  if (n<=1) return s;
  else return s = s + Rsum(a, n-1);
}
```

- I = n (number of elements to be summed)
- C = constant
- Each recursive call "Rsum" requires 4 words
  - Space for **n**, **a**, return value, return address
- # of calls: Rsum(a, n) → … → Rsum(a, 1) → n calls

→ S(Rsum) = C + S_{Rsum}(n) = constant + 4 words × n

Time Complexity

- Time complexity: $T(P) = C + T_P(I)$
- C is a constant part:
  - Compile time, program load time, …; independent of instance characteristics
- $T_P(I)$ is a variable part:
  - Running time

Focus on run time $T_P(I)$

Time Complexity

- How to evaluate $T_P(I)$?
  - Count every Add, Sub, Multiply, … etc.
  - Practically infeasible because each instruction takes different running time at different machines
- Use "**program step**" to estimate $T_P(I)$
  - "program step" = a segment of code whose execution time is ***independent*** of instance characteristics (I)

```
for(i=0; i<n, i++) → one program step
a=2;                → one program step
```

Yes, they have different execution times. But, when we compare **large problem sizes**, the differences are immaterial

Time Complexity: Iterative Summing

```
float Sum(float *a, const int n)
{ float s = 0;                // 1 step
  for(int i=0; i<n; i++) // n+1 steps
      s += a[i];              // n steps
  return s;                   // 1 step
}
```

- I = n (number of elements to be summed)
- $T_{Sum}(I) = 1 + (n+1) + n + 1 = 2n+3$ steps

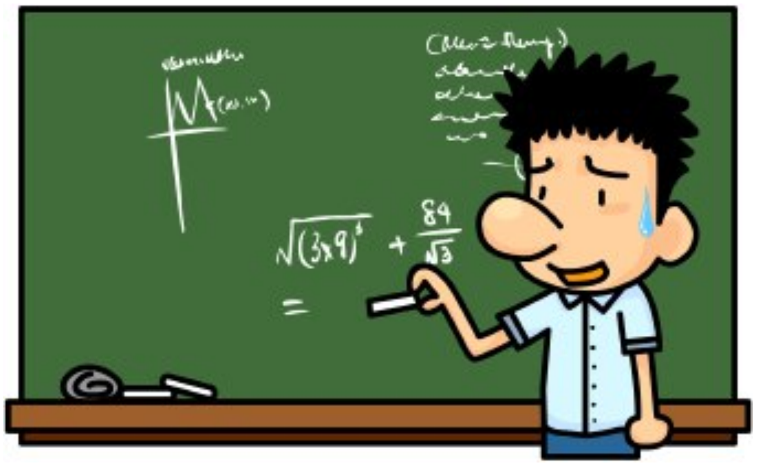
→ $T(Sum) = C + T_{Sum}(n)$ = constant + (2n+3) steps

For **large problem sizes**, n dominates the execution time

Time Complexity: Recursive Summing

```
float Rsum(float *a, const int n)
{ if (n<=1)                                // 1 step
     return a[0];                          // 1 step
  else return (Rsum(a, n-1) + a[n-1]); // 1 step
}
```

- I = n (number of elements to be summed)
- $T_{Rsum}(n)$ = ?



Time Complexity: Recursive Summing

```
float Rsum(float *a, const int n)
{ if (n<=1)                                  // 1 step
      return a[0];                           // 1 step
  else return (Rsum(a,n-1) + a[n-1]);        // 1 step
}
```

- I = n (number of elements to be summed)
- $T_{Rsum}(1) = 2$ steps
- $T_{Rsum}(n) = 2 + T_{Rsum}(n-1)$

  $= 2 + (2 + T_{Rsum}(n-2))$

  $= \ldots$

  $= 2(n-1) + T_{Rsum}(1) = 2n$ steps

Recurrence relations

Observation on Step Counts

- In the previous examples:
  $T_{Sum}(n)$ = 2n + 3 steps
  $T_{Rsum}(n)$ = 2n steps
- Can we say that Rsum is faster than Sum ?
  - **No!**
  - The execution time of each step is inexact and different
- Instead, we focus on "**Growth Rate**" to compare the time complexities of programs
  - *"How the running time changes with changes in the instance characteristics?"*

Program Growth Rate

- For Sum program, $T_{Sum}(n) = 2n + 3$
  → when n is tenfold (10X), $T_{Sum}(n)$ is tenfold (10X)
  - We say that the Sum program runs in **linear** time

- $T_{Rsum}(n) = 2n$ also runs in **linear** time

- Since $T_{Sum}(n)$ and $T_{Rsum}(n)$ have the same growth rate, we say that they are equal in time complexity
- What if $T_{Rsum}(n) = 2^n$?

Mere Growth Rate Is Insufficient

- Two programs with time complexities
  - P1: $c_1 n^2 + c_2 n$
  - P2: $c_3 n$

Which one runs faster?

- Case 1: $c_1 = 1$, $c_2 = 2$, and $c_3 = 100$
  - P1($n^2 + 2n$) ≤ P2($100n$) for $n \leq 98$ → P1 is faster!
- Case 2: $c_1 = 1$, $c_2 = 2$, and $c_3 = 1000$
  - P1($n^2 + 2n$) ≤ P2($1000n$) for $n \leq 998$ → P1 is faster
- No matter what values c_1, c_2 and c_3 are, there will be an n beyond which $c_1 n^2 + c_2 n > c_3 n$ and P2 is faster

Asymptotic Performance

- We should compare the complexity in terms of growth rate for a sufficiently large value of n
- Big-O notation:
  $f(n) = O(g(n))$ iff there exist positive constants c and $n_0 > 0$ such that $f(n) \le c\, g(n)$ for all $n \ge n_0$
- Ex1.: $3n + 2 = O(n)$
  - $3n+2 \le 4n$ for all $n \ge 2$
- Ex2.: $100n + 6 = O(n)$
  - $100n+6 \le 101n$ for all $n \ge 10$
- Ex3.: $10n^2 + 4n + 2 = O(n^2)$
  - $10n^2 + 4n + 2 \le 11 n^2$ for all $n \ge 5$

Properties of Big-O

- $f(n) = O(g(n))$ states $g(n)$ is an ***upper bound*** of $f(n)$
  - $n = O(n) = O(n^{2.5}) = O(n^3)$
  - For the Big-O notation to be informative, $g(n)$ should be ***as small a function of n as possible***!
  - Big-O refers to as ***worst-case running time*** of a program
- Omega (Ω) notation: ***lower bound*** or ***best-case***
  $f(n) = \Omega(g(n))$ iff these exist **$c, n_0>0$** such that **$f(n) \geq c\, g(n)$** for all **all $n \geq n_0$**
- Theta (Θ) notation: ***tight bound*** or ***average-case***
  $f(n) = \Theta(g(n))$ iff **$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$**

Big-O for Polynomial Functions

- Theorm 1.2:
  If $f(n) = a_m n^m + \ldots + a_1 n + a_0$, then $f(n) = O(n^m)$
  - $3n + 2 = O(n)$
  - $100n + 6 = O(n)$
  - $6n^4 + 1000\, n^3 + n^2 = O(n^4)$
- Since Big-O estimates worst-case performance, the "worst term" dominates other terms
  - ***leading constants*** and ***lower-order terms*** do not matter
  - $n^2 + n \log n = O(?)$
  - $O(2^n) + O(n^{10000}) = O(?)$

Names of Common Functions

Complexity	Name
$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n \log n)$	$O(\log n) \leq . \leq O(n^2)$
$O(n^2)$	Quadratic time
$O(n^3)$	Cubic time
$O(n^{100})$	Polynomial time
$O(2^n)$	Exponential time

When n is large enough, the lower terms take more time than the upper ones

Running Times on Computers

- Running times on a 1-billion-steps-per-second computer (1 billion = 10^9)

	f (n)					
n	n	$n \log_2 n$	n^2	n^3	n^{10}	2^n
10	.01 μs	.03 μs	.1 μs	1 μs	10s	1μs
20	.02 μs	.09 μs	.4 μs	8 μs	2.84h	1ms
30	.03 μs	.15 μs	.9 μs	27 μs	6.83d	1s
40	.04 μs	.21 μs	1.6 μs	64 μs	121d	18m
50	.05 μs	.28 μs	2.5 μs	125 μs	3.1y	13d
100	.10 μs	.66 μs	10 μs	1ms	3171y	$4*10^{13}$y
10^3	1 μs	9.96 μs	1 ms	1s	$3.17*10^{13}$y	$32*10^{283}$y
10^4	10 μs	130 μs	100 ms	16.67m	$3.17*10^{23}$y	
10^5	100 μs	1.66 ms	10s	11.57d	$3.17*10^{33}$y	
10^6	1ms	19.92ms	16.67m	31.71y	$3.17*10^{43}$y	

μs = microsecond = 10^{-6}second; ms =milliseconds = 10^{-3}seconds
s = seconds; m = minutes; h = hours; d = days; y = years;

Algorithm impractical if complexity is exponential or high degree polynomial

Performance Measurement

- Obtain actual space and time requirement when running a program
- How to do time measurement?
  - Use system functions such as time()
  - How many data points to measure?
  - To time a short program, it is necessary to repeat it many times, and then take the average
- How to measure average/worst-case time?
- How to determine a sufficiently large instance for asymptotic performance?

Performance Measurement

- Use time(), measured in seconds

```
#include <time.h>
void main()
{
  time_t start = time(NULL);

  // main body of program comes here!

  time_t stop = time(NULL);
  double duration=(double)difftime(stop,start);
}
```

Summary

- An algorithm is a finite set of instructions to solve a computational problem
  - Take some inputs and produce some outputs
  - Right choice of data structures affects algorithm efficiency
- Divide-and-conquer is a common strategy for developing algorithms
  - Recursion is handy for expressing certain type of such algo
- Algorithms are often evaluated using time/space
  - Evaluated using instance characteristics, considering growth rate and large problem size
  - Concept of Big-O