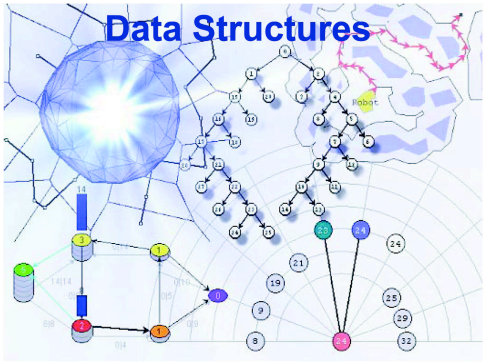
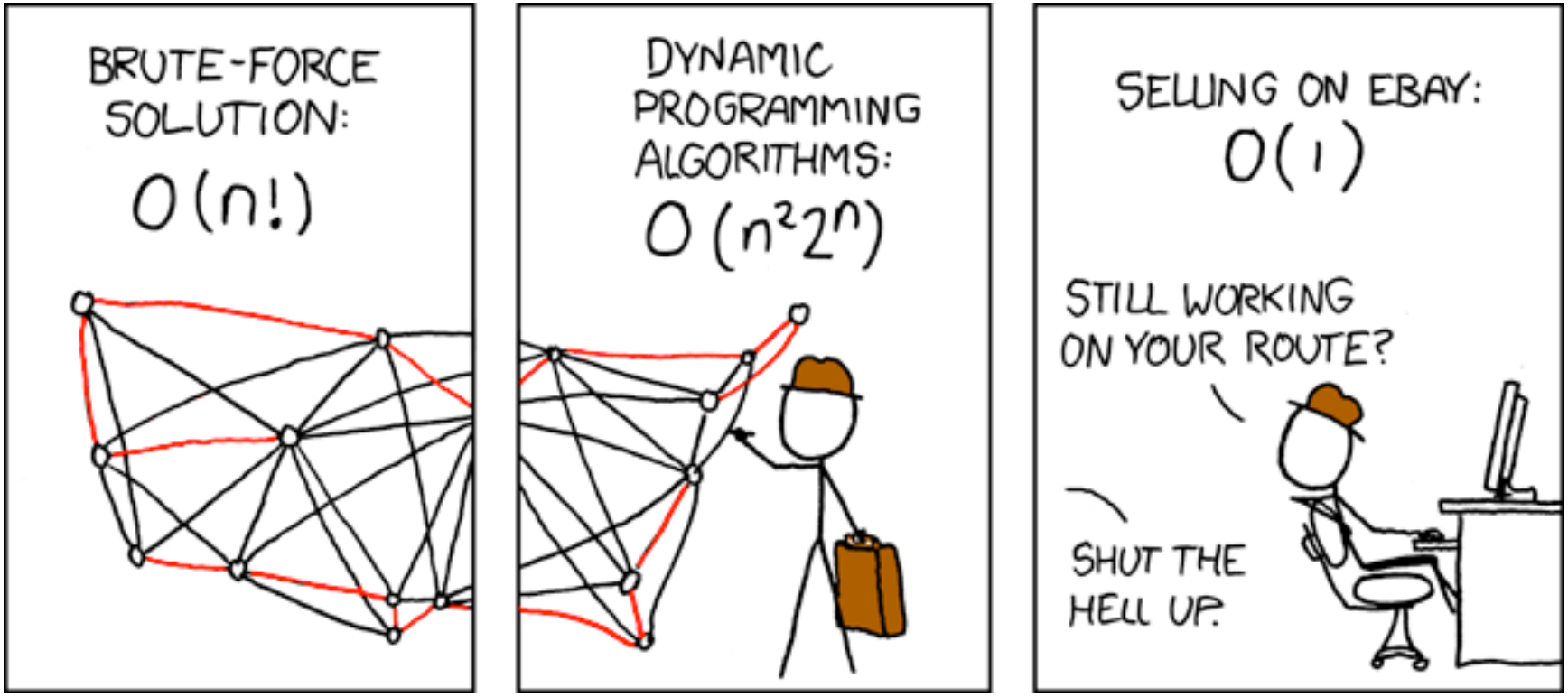
BBM 201
DATA STRUCTURES

Lecture 2:
Recursion & Performance analysis



2017-2018 Fall


BRUTE-FORCE SOLUTION:
O(n!)
DYNAMIC PROGRAMMING ALGORITHMS:
O(n²2ⁿ)
SELLING ON EBAY:
O(1)
STILL WORKING ON YOUR ROUTE?
SHUT THE HELL UP.

System Life Cycle

- Programs pass through a period called ‘system life cycle’, which is defined by the following steps:
  - **1. Requirements:** It is defined by the *inputs* given to the program and *outputs* that will be produced by the program.
  - **2. Analysis:** The first job after defining the requirements is to analyze the problem. There are two ways for this:
    - Bottom-up
    - Top-down

System Life Cycle (cont')

- **3. Design:** Data objects and the possible operations between the objects are defined in this step. Therefore, abstract data objects and the algorithm are defined. They are all independent of the programming language.

- **4. Refinement and Coding:** Algorithms are used in this step in order to do operations on the data objects.

- **5. Verification:** Correctness proofs, testing and error removal are performed in this step.

Cast of Characters

Programmer needs to develop a working solution.

Student might play any or all of these roles someday.

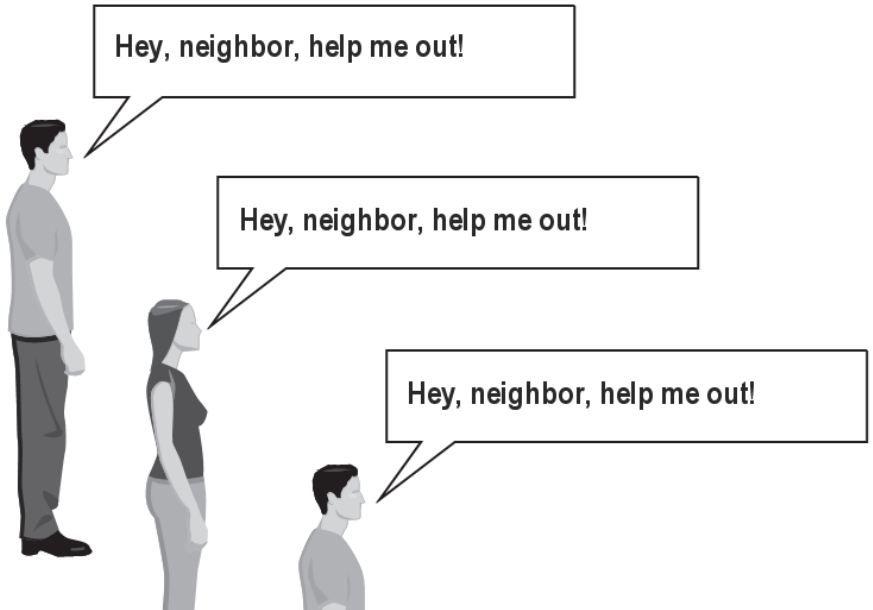
Client wants to solve problem efficiently.

Theoretician wants to understand.

Recursion

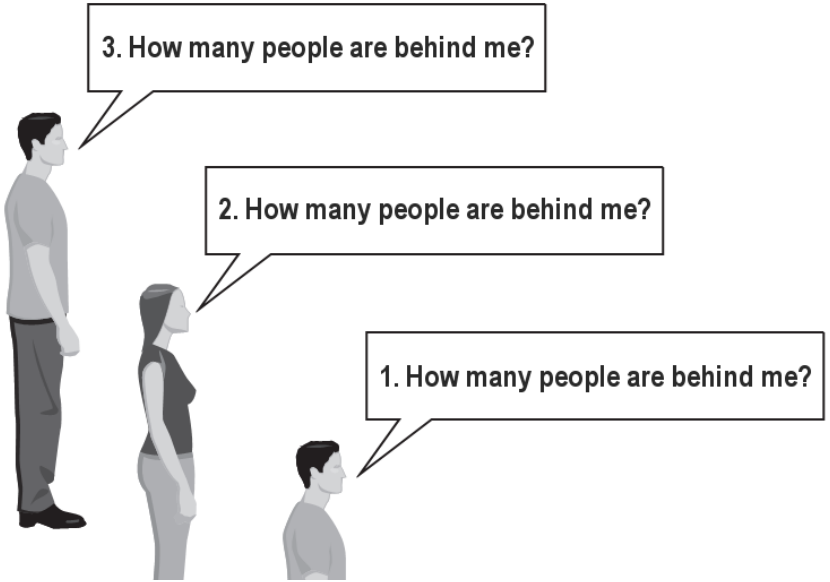
Recursion

- Recursion is all about breaking a big problem into smaller occurrences of that same problem.
  - Each person can solve a small part of the problem.
    - What is a small version of the problem that would be easy to answer?
    - What information from a neighbor might help me?



Recursive Algorithm

- Number of people behind me:
  - If there is someone behind me,
    ask him/her how many people are behind him/her.
    - When they respond with a value **N**, then I will answer **N + 1**.
  - If there is nobody behind me, I will answer **0**.



Recursive Algorithms

- Functions can call themselves (direct recursion)
- A function that calls another function is called by the second function again. (indirect recursion)

Recursion

- Consider the following method to print a line of * characters:

```
// Prints a line containing the given number of
 stars.
// Precondition: n >= 0
void printStars(int n) {
    for (int i = 0; i < n; i++) {
        System.out.print("*");
    }
    System.out.println();   // end the line of output
}
```

- Write a recursive version of this method (that calls itself).
  - Solve the problem <u>without using any loops</u>.
  - Hint: Your solution should print just one star at a time.

A basic case

- What are the cases to consider?
  - What is a very easy number of stars to print without a loop?

```
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else {
        ...
    }
}
```

Handling more cases

- Handling additional cases, with no loops (in a bad way):

```
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else if (n == 2) {
        System.out.print("*");
        System.out.println("*");
    } else if (n == 3) {
        System.out.print("*");
        System.out.print("*");
        System.out.println("*");
    } else if (n == 4) {
        System.out.print("*");
        System.out.print("*");
        System.out.print("*");
        System.out.println("*");
    } else ...
}
```

Handling more cases 2

- Taking advantage of the repeated pattern (somewhat better):

```
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else if (n == 2) {
        System.out.print("*");
        printStars(1);    // prints "*"
    } else if (n == 3) {
        System.out.print("*");
        printStars(2);    // prints "**"
    } else if (n == 4) {
        System.out.print("*");
        printStars(3);    // prints "***"
    } else ...
}
```

Using recursion properly

- Condensing the recursive cases into a single case:

```
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else {
        // recursive case; print one more star
        System.out.print("*");
        printStars(n - 1);
    }
}
```

Even simpler

- The real, even simpler, base case is an n of 0, not 1:

```
public static void printStars(int n) {
    if (n == 0) {
        // base case; just end the line of output
        System.out.println();
    } else {
        // recursive case; print one more star
        System.out.print("*");
        printStars(n - 1);
    }
}
```

Recursive tracing

- Consider the following recursive method:

```
public static int mystery(int n) {
    if (n < 10) {
        return n;
    } else {
        int a = n / 10;
        int b = n % 10;
        return mystery(a + b);
    }
}
```

- What is the result of the following call?

```
mystery(648)
```

A recursive trace

mystery(648):

- `int a = 648 / 10;` `// 64`
- `int b = 648 % 10;` `//  8`
- `return mystery(a + b);` `// mystery(72)`

mystery(72):

- `int a = 72 / 10;` `// 7`
- `int b = 72 % 10;` `// 2`
- `return mystery(a + b);` `// mystery(9)`

mystery(9):

- `return 9;`

Recursive tracing 2

- Consider the following recursive method:

```
public static int mystery(int n) {
    if (n < 10) {
        return (10 * n) + n;
    } else {
        int a = mystery(n / 10);
        int b = mystery(n % 10);
        return (100 * a) + b;
    }
}
```

  - What is the result of the following call?

```
mystery(348)
```

A recursive trace 2

mystery(348)

- int a = mystery(34);
  - int a = mystery(3);
    return (10 * 3) + 3; // 33
  - int b = mystery(4);
    return (10 * 4) + 4; // 44
  - return (100 * 33) + 44; // 3344
- int b = mystery(8);
  return (10 * 8) + 8; // 88
- return (100 * 3344) + 88; // 334488

- What is this method really doing?

Binary Search

- A value is searched in a sorted array. If found, the index of the item is returned, otherwise -1 is returned.

```
int binsearch(int list[], int searchnum, int left, int right)
{
   int middle;
   while(left <= right){
     middle = (left + right)/2;
     switch(compare(list[middle], searchnum)){
        case -1 : left = middle + 1; break;
        case 0; return middle;
        case 1: rigt = middle –1;
     }
    }
   return -1;
}
```

- Macro for ‘COMPARE’:

```
#define COMPARE(x,y) (((x)<(y))? -1 ((x)==(y))?0:1)
```

Binary Search

Let's search for 18:

4	13	18	19	24	27	33	36	39	40
LEFT									RIGHT

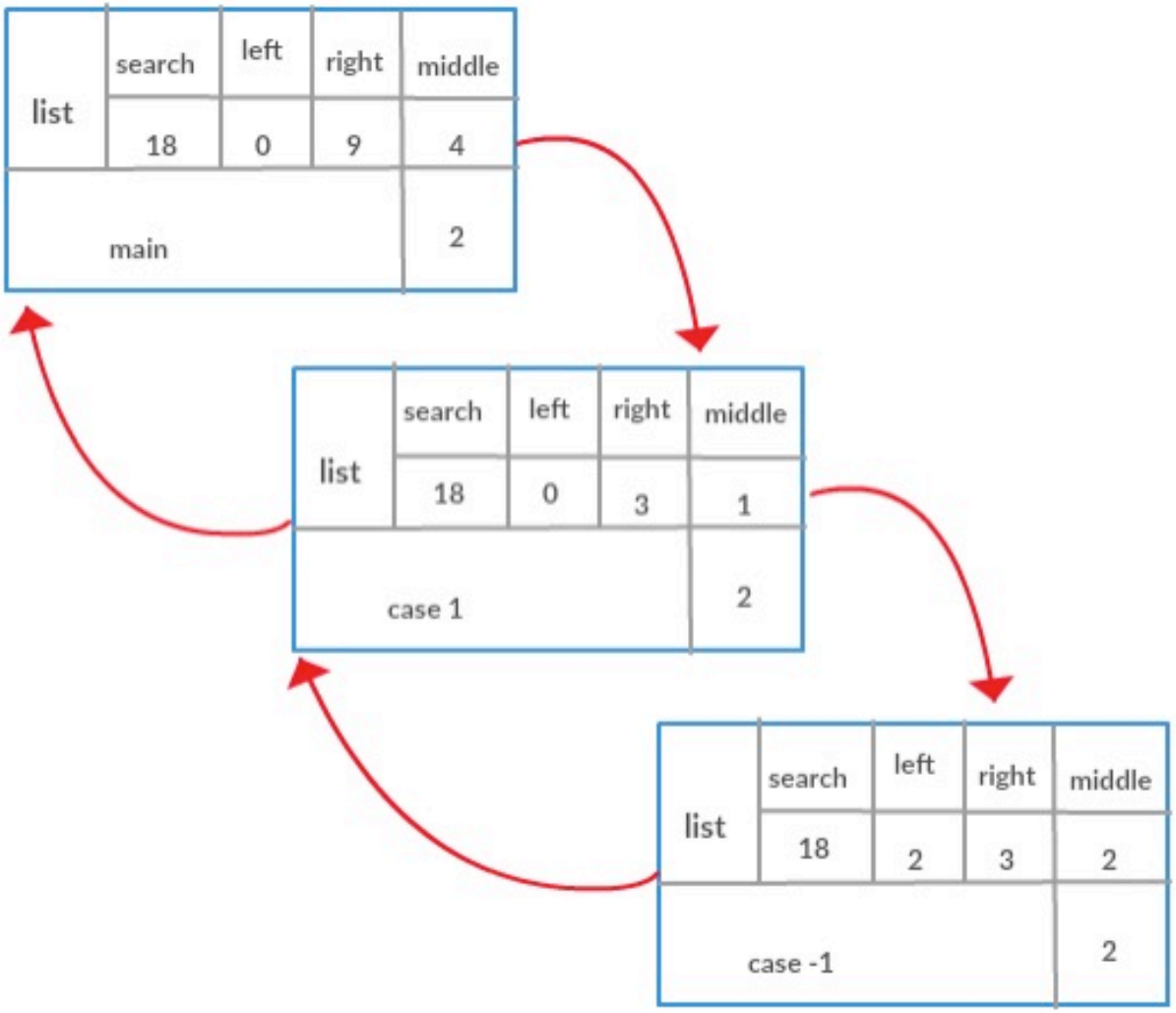
4	13	18	19	24	27	33	36	39	40
LEFT			RIGHT						

4	13	18	19	24	27	33	36	39	40
		LEFT	RIGHT						

Binary Search (revisited)

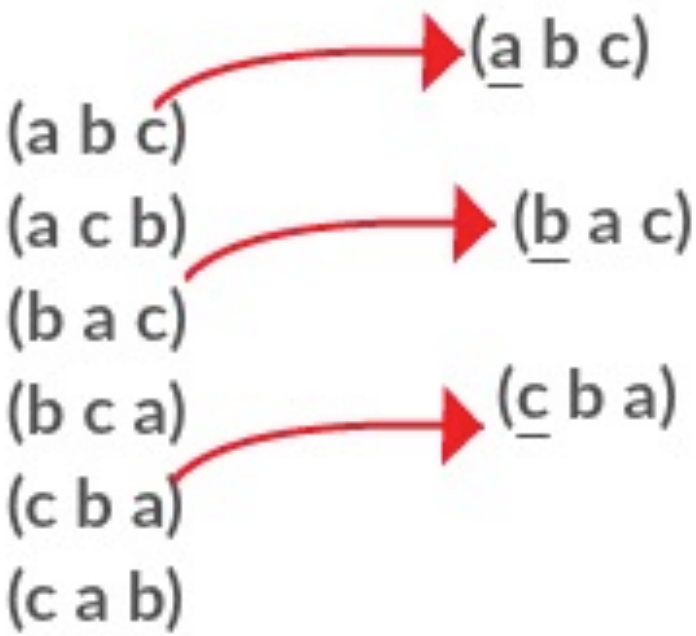
```
int binsearch(int list[], int searchnum, int left, int right)
{
   int middle;
   if(left<=right){
      middle = (left+right)/2;
      switch(COMPARE(list[middle],searchnum)){
         case -1: return binsearch(list, searchnum, middle+1, right);
         case 0: return middle;
         case 1: return binsearch(list, searchnum, left, middle-1);
       }
   }
   return -1;
}
```

Binary Search (revisited)



Permutation problem

- Finding all the permutations of a given set with size n≥1.
  - Remember there are n! different sequences of this set.
  - Example: Find all the permutations of {a,b,c}



Permutation problem

- Example: Find all the permutations of {a,b,c,d}

- All the permutations of {b,c,d} follow ‘a’
- All the permutations of {a,c,d} follow ‘b’
- All the permutations of {a,b,d} follow ‘c’
- All the permutations of {a,b,c} follow ‘d’

Permutation problem

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 10
#define SWAP(x,y,t) ((t)==(x), (x)==(y), (y)==(t))

void perm(char*, int, int);

void main(void)
{
   int n;
   char list[MAX_SIZE];
   printf("\nEnter the values:");
   for(n=0; (list[n]=getchar())!='\n'; n++)
      if(n>=MAX_SIZE){
         printf("\nOverflow error!\n");
         exit(1);
       }
   perm(list,0,n-1);
}
```

Permutation problem

```
void perm(char* list, int i, int n)
{
   int j, temp;
   if(i==n){
      for(j=0; j<=n; j++)
         printf("%c", list[j]);
      printf("\n");
   }
   else{
      for(j=i; j<=n; j++){
         SWAP(list[i], list[j], temp);
         perm(list, i+1, n);
         SWAP(list[i], list[j], temp);
      }
   }
}
```

Factorial Function

- $n! = n*(n-1)*(n-2)*\ldots*2*1$ for any integer n>0

  $0! = 1$

- **<u>Iterative Definition:</u>**
-

```
int fval=1;

for(i=n; i>=1; i--)
     fval = fval*i;
```

Factorial Function

Recursive Definition

- To define n! recursively, n! must be defined in terms of the factorial of a smaller number.

- Observation (problem size is reduced):

$$n! = n * (n-1)!$$

- Base case:

$$0! = 1$$

- We can reach the base case by subtracting 1 from n if n is a positive integer.

Factorial Function

Recursive Definition

Recursive definition

$$n! = 1 \qquad if\ n = 0$$

$$n! = n * (n-1)! \qquad if\ n > 0$$

```
int fact(int n)
{

   if(n=0)
      return (1);
   else
      return (n*fact(n-1));
}
```

This fact function satisfies the four criteria of a recursive solution.

Four Criteria of a Recursive Solution

1. A recursive function calls itself.
   - This action is what makes the solution recursive.
2. Each recursive call solves an identical, but smaller, problem.
   - A recursive function solves a problem by solving another problem that is identical in nature but smaller in size.
3. A test for the base case enables the recursive calls to stop.
   - There must be a case of the problem (known as *base case* or *stopping case*) that is handled differently from the other cases (without recursively calling itself.)
   - In the base case, the recursive calls stop and the problem is solved directly.
4. Eventually, one of the smaller problems must be the base case.
   - The manner in which the size of the problem diminishes ensures that the base case is eventually is reached.

Fibonacci Sequence

- It is the sequence of integers:

t_0		t_1	t_2	t_3	t_4	t_5	t_6
0	1	1	2	3	5	8	...

- Each element in this sequence is the sum of the two preceding elements.
- The specification of the terms in the Fibonacci sequence:

$$t_n = \begin{cases} n & if\ n\ is\ 0\ or\ 1\ (i.e.\ n < 2) \\ t_{n-1} + t_{n-2} & otherwise \end{cases}$$

Fibonacci Sequence

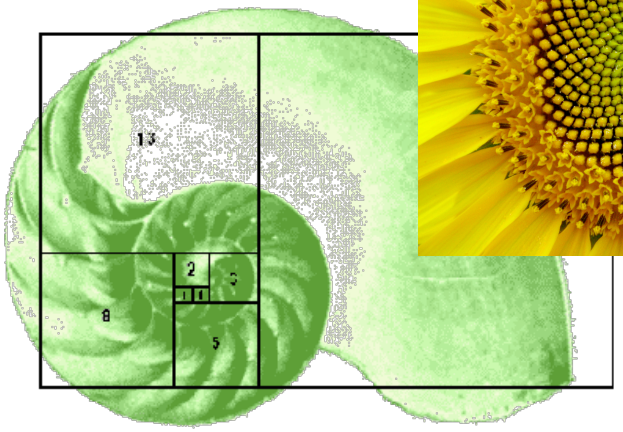
- **Iterative solution**

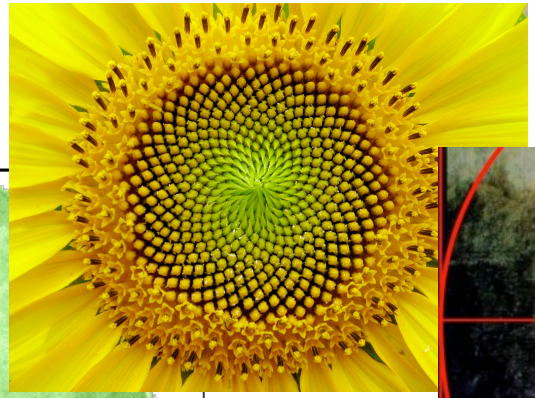
```
int Fib(int n)
{

   int prev1, prev2, temp, j;

   if(n==0 || n==1)
      return n;
   else{
       prev1=0;
       prev2=1;
       for(j=1;j<=n;j++)
       {
           temp = prev1+prev2;
           prev1=prev2;
           prev2=temp;
       }
       return prev2;
   }
}
```

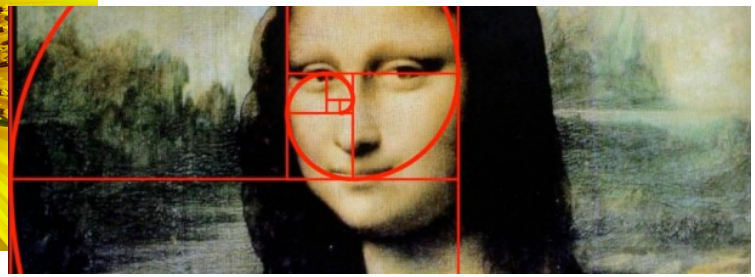
Fibonacci Sequence

- **Recursive solution**

```
int fibonacci(int n)
{

   if(n<2)
      return n;
   else
      return (fibonacci(n-2)+fibonacci(n-1));
}
```



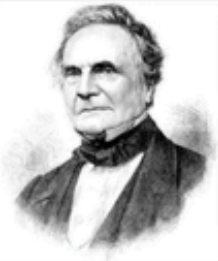


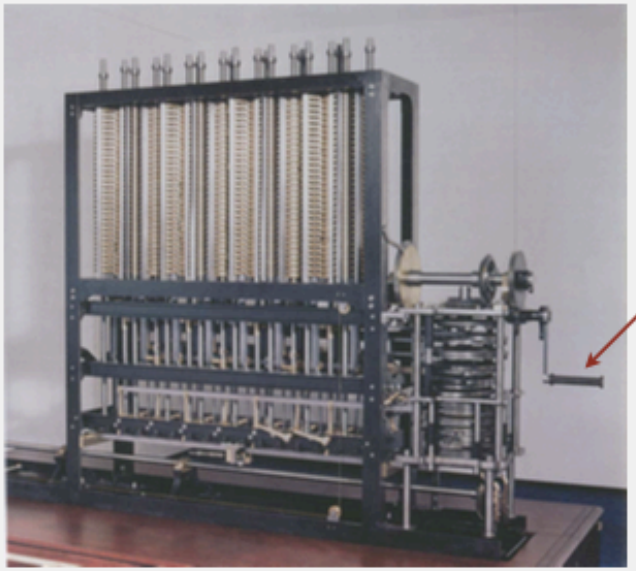


Performance Analysis

Running Time

" As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time? " — *Charles Babbage (1864)*





how many times do you have to turn the crank?

Analytic Engine

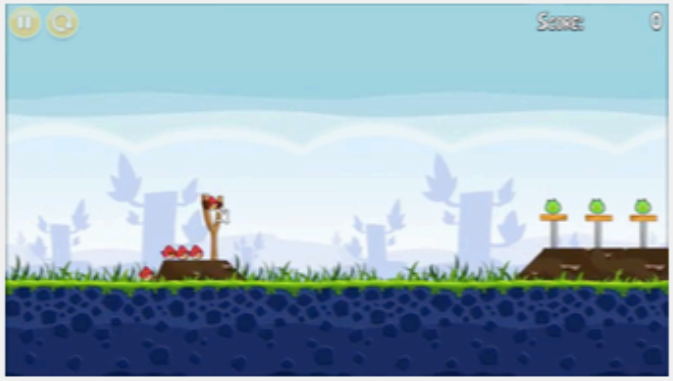
Observations

Example-1

3-SUM. Given N distinct integers, how many triples sum to exactly zero?

```
% more 8ints.txt
8
30 -40 -20 -10 40 0 10 5

% java ThreeSum 8ints.txt
4
```

	a[i]	a[j]	a[k]	sum
1	30	-40	10	0
2	30	-20	-10	0
3	-40	40	0	0
4	-10	0	10	0



Context. Deeply related to problems in computational geometry.

Observations

Example-1: 3-SUM brute force algorithm

```
public class ThreeSum
{
   public static int count(int[] a)
   {
      int N = a.length;
      int count = 0;
      for (int i = 0; i < N; i++)
         for (int j = i+1; j < N; j++)
            for (int k = j+1; k < N; k++)
               if (a[i] + a[j] + a[k] == 0)
                  count++;
      return count;
   }

   public static void main(String[] args)
   {
      In in = new In(args[0]);
      int[] a = in.readAllInts();
      StdOut.println(count(a));
   }
}
```

check each triple

for simplicity, ignore integer overflow

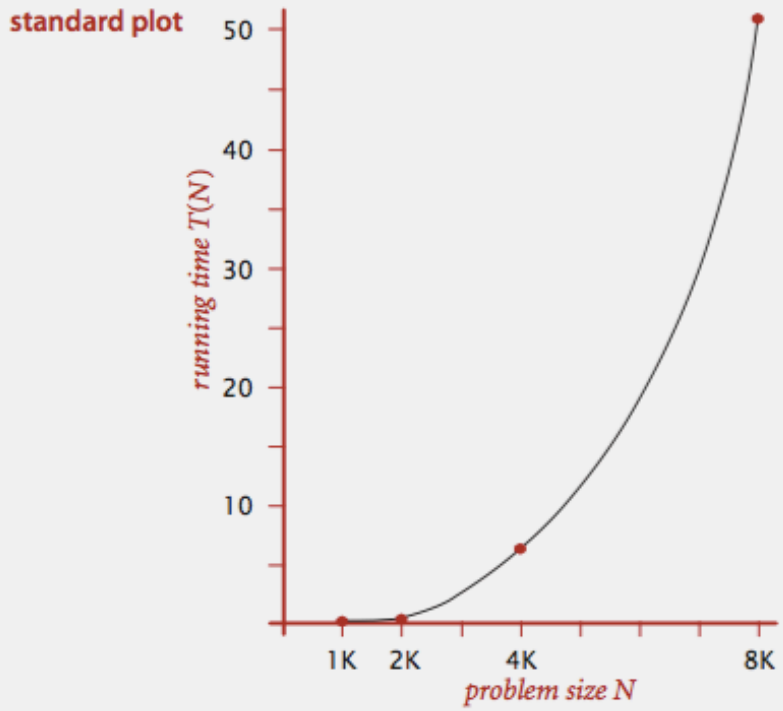
Empirical Analysis

Run the program for various input sizes and measure running time.

N	time (seconds) †
250	0.0
500	0.0
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1
16,000	?

Data Analysis

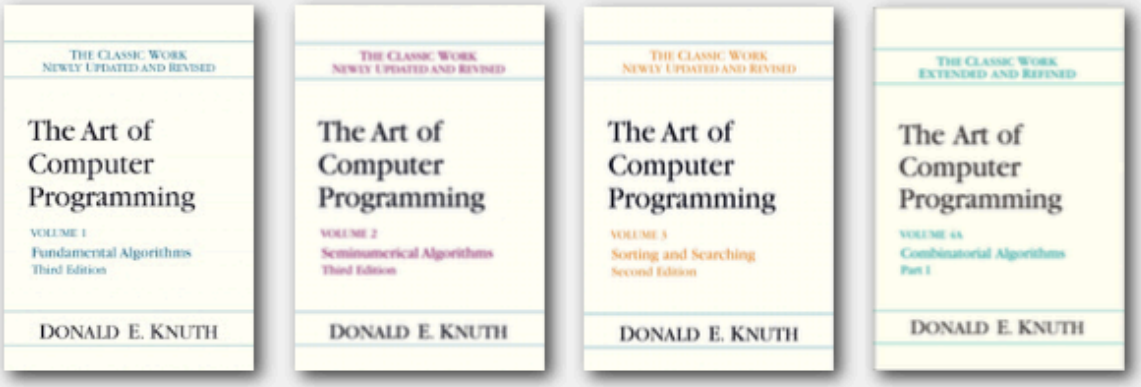
Standard plot. Plot running time $T(N)$ vs. input size N.



Mathematical Models for Running Time

Total running time: sum of cost × frequency for all operations.

- Need to analyze program to determine set of operations.
- Cost depends on machine, compiler.
- Frequency depends on algorithm, input data.



Donald Knuth
1974 Turing Award

In principle, accurate mathematical models are available.

Cost of Basic Operations

Challenge. How to estimate constants.

operation	example	nanoseconds †
integer add	a + b	2.1
integer multiply	a * b	2.4
integer divide	a / b	5.4
floating-point add	a + b	4.6
floating-point multiply	a * b	4.2
floating-point divide	a / b	13.5
sine	Math.sin(theta)	91.3
arctangent	Math.atan2(y, x)	129.0
...	...	...

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM

Cost of Basic Operations

Observation. Most primitive operations take constant time.

operation	example	nanoseconds †
variable declaration	`int a`	c_1
assignment statement	`a = b`	c_2
integer compare	`a < b`	c_3
array element access	`a[i]`	c_4
array length	`a.length`	c_5
1D array allocation	`new int[N]`	$c_6 N$
2D array allocation	`new int[N][N]`	$c_7 N^2$

Caveat. Non-primitive operations often take more than constant time.

novice mistake: abusive string concatenation

Example: 1-Sum

Q. How many instructions as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
   if (a[i] == 0)
      count++;
```

N array accesses

operation	frequency
variable declaration	2
assignment statement	2
less than compare	$N + 1$
equal to compare	N
array access	N
increment	N to $2N$

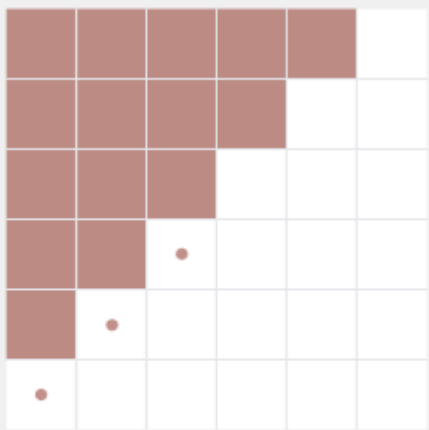
Example: 2-Sum

Q. How many instructions as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      if (a[i] + a[j] == 0)
         count++;
```

$$0+1+2+\ldots+(N-1) = \frac{1}{2}N(N-1) = \binom{N}{2}$$

Pf. [n even]



$$0+1+2+\ldots+(N-1) = \underbrace{\frac{1}{2}N^2}_{\text{half of square}} - \underbrace{\frac{1}{2}N}_{\text{half of diagonal}}$$

Example: 2-Sum

Q. How many instructions as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      if (a[i] + a[j] == 0)
         count++;
```

$$0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2} N (N-1) = \binom{N}{2}$$

operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	$\frac{1}{2} (N+1)(N+2)$
equal to compare	$\frac{1}{2} N (N-1)$
array access	$N (N-1)$
increment	$\frac{1}{2} N (N-1)$ to $N (N-1)$

tedious to count exactly

Simplifying the Calculations

" It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings. " — *Alan Turing*

ROUNDING-OFF ERRORS IN MATRIX PROCESSES

By A. M. TURING

(*National Physical Laboratory, Teddington, Middlesex*)

[Received 4 November 1947]

SUMMARY

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known 'Gauss elimination process', it is found that the errors are normally quite moderate: no exponential build-up need occur.

Simplification: cost model

Cost model. Use some basic operation as a proxy for running time.

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      if (a[i] + a[j] == 0)
         count++;
```

$$0+1+2+\ldots+(N-1) = \frac{1}{2}N(N-1) = \binom{N}{2}$$

operation	frequency
variable declaration	$N+2$
assignment statement	$N+2$
less than compare	$\frac{1}{2}(N+1)(N+2)$
equal to compare	$\frac{1}{2}N(N-1)$
array access	$N(N-1)$
increment	$\frac{1}{2}N(N-1)$ to $N(N-1)$

cost model = array accesses

(we assume compiler/JVM do not optimize any array accesses away!)

Asymptotic Notation: 2-SUM problem

Q. Approximately how many array accesses as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      if (a[i] + a[j] == 0)
         count++;
```

"inner loop"

$$0+1+2+\ldots+(N-1) = \frac{1}{2}N(N-1) = \binom{N}{2}$$

A. $\sim N^2$ array accesses.

Asymptotic Notation: 3-SUM problem

Q. Approximately how many array accesses as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      for (int k = j+1; k < N; k++)
         if (a[i] + a[j] + a[k] == 0)
            count++;
```

"inner loop"

$$\binom{N}{3} = \frac{N(N-1)(N-2)}{3!} \sim \frac{1}{6}N^3$$

A. $\sim \frac{1}{2} N^3$ array accesses.

Binary search (java implementation)

Invariant. If key appears in array a[], then a[lo] ≤ key ≤ a[hi].

```
public static int binarySearch(int[] a, int key)
{
   int lo = 0, hi = a.length - 1;
   while (lo <= hi)
   {
       int mid = lo + (hi - lo) / 2;
       if      (key < a[mid]) hi = mid - 1;
       else if (key > a[mid]) lo = mid + 1;
       else return mid;
   }
   return -1;
}
```

why not mid = (lo + hi) / 2 ?

one "3-way compare"

Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \lg N$ key compares to search in a sorted array of size N.

Def. $T(N)$ = # key compares to binary search a sorted subarray of size $\leq N$.

Binary search recurrence. $T(N) \leq T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.

left or right half (floored division)

possible to implement with one 2-way compare (instead of 3-way)

Pf sketch. [assume N is a power of 2]

$$
\begin{aligned}
T(N) &\leq T(N/2) + 1 && \text{[given]} \\
&\leq T(N/4) + 1 + 1 && \text{[apply recurrence to first term]} \\
&\leq T(N/8) + 1 + 1 + 1 && \text{[apply recurrence to first term]} \\
&\vdots \\
&\leq T(N/N) + \underbrace{1 + 1 + \ldots + 1}_{\lg N} && \text{[stop applying, } T(1) = 1 \text{]} \\
&= 1 + \lg N
\end{aligned}
$$

Types of Analyses

Best case : Lower bound on cost.
Determined by "easiest" input.
Provides a good for all inputs.

Worst case : Upper bound on cost.
Determined by "most difficult" input..
Provides a guarantee for all inputs.

Average case : Expected cost for random input.
Need a model for "random" input.
Provides a way to predict performance.

slide by R. Sedgewick and K. Wayne

Types of Analyses

Ex 1. Array accesses for brute-force 3-SUM.

Best: $\sim \frac{1}{2} N^3$

Average: $\sim \frac{1}{2} N^3$

Worst: $\sim \frac{1}{2} N^3$

Ex 2. Compares for binary search.

Best: ~ 1

Average: $\sim \lg N$

Worst: $\sim \lg N$

Mathematical Models for Running Time

In principle, accurate mathematical models are available.

In practice,

- Formulas can be complicated.
- Advanced mathematics might be required.
- Exact models best left for experts.

costs (depend on machine, compiler)

$$T_N = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E$$

$A =$ array access
$B =$ integer add
$C =$ integer compare
$D =$ increment
$E =$ variable assignment

frequencies
(depend on algorithm, input)

Bottom line. We use approximate models in this course: $T(N) \sim c\,N^3$.

Asymptotic notation: O (Big Oh)

$f(n) = O(g(n))$ iff there exist positive constants c and n_0 such that

$f(n) <= cg(n)$ for all n, $n >= n_0$

$3n+2 = O(n)$	as $3n+2 <= 4n$	for all $n>= 2$
$10n^2+4n+2 = O(n^2)$	as $10n^2 + 4n + 3 <= 11n^2$	for $n>=5$
$6*2^n + n^2 = O(2^n)$	$6*2^n + n^2 <= 7*2^n$	for $n>=4$

$3n+3 = O(n)$	Correct, OK.
$3n+3 = O(n^2)$	Correct, NO!

Question

How many array accesses does the following code fragment make as a function of N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      for (int k = 1; k < N; k = k*2)
         if (a[i] + a[j] >= a[k])
            count++;
```

A. $\sim 3\ N^2$

B. $\sim 3/2\ N^2 \lg N$

C. $\sim 3/2\ N^3$

D. $\sim 3\ N^3$

E. *I don't know.*

Performance Analysis

- There are various criteria in order to evaluate a program.
- The questions:
  - Does the program meet the requirements determined at the beginning?
  - Does it work correctly?
  - Does it have any documentation on the subject of how it works?
  - Does it use the function effectively in order to produce the logical values: TRUE and FALSE?
  - Is the program code readable?
  - Does the program use the primary and the secondary memory effectively?
  - Is the running time of the program acceptable?

Performance Analysis

Performance Evaluation

Space Complexity

Time Complexity

Space Complexity

- Memory space needed by a program to complete its execution:
  - A fixed part → c
  - A variable part → S_p(instance)

$S(P) = c + S_p(\text{instance})$

```
float abc(float a, float b, float c){
   return a+b+b*c+(a+b+c)/(b+c);
}
```

$S_{abc}(I)=0$

```
float sum(float list[], int n){
   float tempsum=0;
   int i;
   for(i=0;i<n;i++){
      tempsum += list[i];
   return tempsum;
}
```

$S_{sum}(n)=0$

Space Complexity

- Recursive function call:

```
float rsum(float list[], int n){
   if(n)
     return rsum(list, n-1)+list[n-1];
   return 0;
}
```

- What is the total variable memory space of this method for an input size of 2000?

Time Complexity

- Total time used by the program is: T(P)
  - Where T(P)=compile time + run (execution) time (T_P)

```
float sum(float list[], int n){
   float tempsum=0;   ..................................  1
   int i;
   for(i=0;i<n;i++){  ..................................  n+1
      tempsum += list[i]; ..............................  n
   return tempsum; ..................................... 1
}

Total step number = 2n+3
```

Time Complexity

```
float rsum(float list[], int n){
   if(n)
     return rsum(list, n-1)+list[n-1];
   return 0;
}
```

- What is the number of steps required to execute this method?

Time Complexity

```
void add(int a[MAX_SIZE]){
   int i,j;
   for(i=0; i<rows; i++)
      for(j=0; j<cols; j++)
         c[i][j]=a[i][j]+b[i][j];
```

- What is the number of steps required to execute this method?

O (Big Oh) Notation

- Time complexity = algorithm complexity
- $f(n)<cg(n) \rightarrow f(n)=O(g(n))$
- $T_{sum}(n)=2n+3 \rightarrow T_{sum}(n)=O(n)$
- $T_{rsum}(n)=2n+2 \rightarrow T_{sum}(n)=O(n)$
- $T_{add}(rows,cols)=2rows*cols+2rows+1 \rightarrow T_{add}(n)=O(rows*cols)$
  $=O(n^2)$

Asymptotic Notation

- Example: $f(n)=3n^3+6n^2+7n$

- What is the algorithm complexity in O notation?

“Magic Square” Problem

- A magic square is a square grid where the numbers in each row and in each column add up to the same number.

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

“Magic Square” Problem

— Based on Coxeter’s rule

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 15

void main(void){
   static int square[MAX_SIZE][MAX_SIZE];
   int i,j,row,column;
   int count;
   int size;

   printf(“\n Enter the square size: “ );
   scanf(“ %d “,&size);
   //Check the square size

   for(i=0; i<size; i++)
      for(j=0; j<size; j++)
         square[i][j]=0;
   square[0][(size-1)/2]=1; //ilk satırın orta elemanı
   i=0;                     //ilk satır
   j=(size-1)/2;            //orta kolon
```

“Magic Square” Problem

— Based on Coxeter’s rule

```
    for(count=2; count<= size*size; count++){
        row=(i-1 < 0) ? (size-1) : (i-1);
        column=(j-1 < 0) ? (size-1) : (j-1);

        if(square[row][column])
            i=(++i)%size;
        else{
            i=row;
            j=column;
        }
        square[i][j]=count;
    }
    printf(“Magix Square size:%d :\n\n “,size);
    for(i=0; i<size; i++)
        for(j=0; j<size; j++)
            printf(“%5d“, square[i][j]);
        printf(“\n“);
    }
}
```

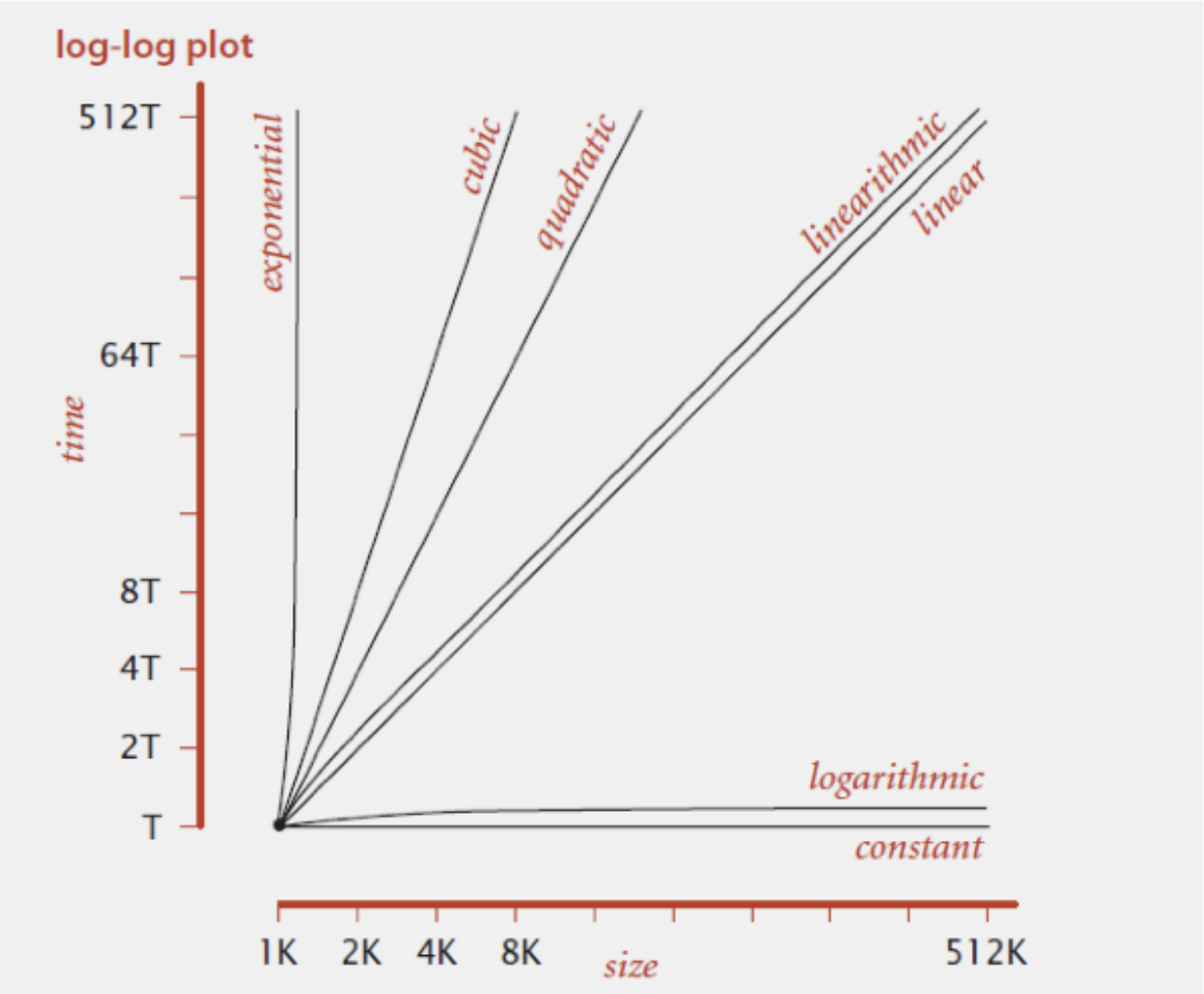
Question

- Example: If the processor can execute 1 billion commands in 1 second, calculate the time required for the ‘magic square’ problem solved by brute force for n=5.

Practical Complexities

Complexity (time)	Name	1	2	4	8	16	32
1	constant	1	1	1	1	1	1
logn	logarithmic	0	1	2	3	4	5
n	linear	1	2	4	8	16	32
nlogn	log linear	0	2	8	24	64	160
n^2	quadratic	1	4	16	64	256	1024
n^3	cubic	1	8	64	512	4096	32768
2^n	exponential	2	4	16	256	65536	4294967296
n!	factorial	1	2	24	40326	20922789888000	$26313*10^{33}$

Practical Complexities



n	f(n)=n	f(n)=$\log_2 n$	f(n)=n^2	f(n)=n^3	f(n)=n^4	f(n)=n^{10}	f(n)=2^n
10	0.01μ	0.03μ	0.1μ	1μ	10μ	10 sec	1μ
20	0.02μ	0.09μ	0.4μ	8μ	160μ	2.84 hr	1 ms
30	0.03μ	0.15μ	0.9μ	27μ	810μ	6.83 d	1 sec
40	0.04μ	0.21μ	1.6μ	64μ	2.56ms	121.36 d	18.3 min
50	0.05μ	0.28μ	2.5μ	125μ	6.25ms	3.1 yr	13 d
100	0.10μ	0.66μ	10μ	1ms	100ms	3171 yr	$4*10^{13}$ yr
1000	1μ	9.96μ	1ms	1sec	16.67min	$3.17*10^{13}$ yr	
10.000	10μ	130.03μ	100ms	16.67min	115.7d	$3.17*10^{23}$ yr	
100.000	100μ	1.66ms	10sec	11.57d	3171yr	$3.17*10^{33}$ yr	
1.000.000	1ms	19.92ms	16.67min	31.71yr	$3.17*10^{7}$ yr	$3.17*10^{43}$ yr	

Survey of Common Running Times

Tilde notation

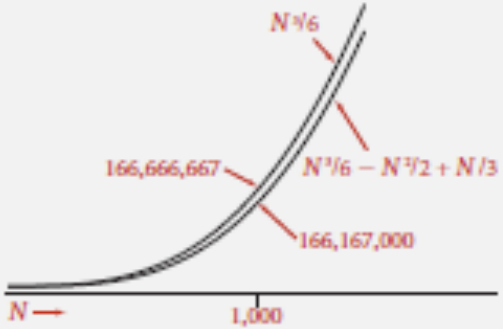
- Estimate running time (or memory) as a function of input size N.
- Ignore lower order terms.
  - when N is large, terms are negligible
  - when N is small, we don't care

Ex 1. $\frac{1}{6}N^3 + 20N + 16 \sim \frac{1}{6}N^3$

Ex 2. $\frac{1}{6}N^3 + 100N^{4/3} + 56 \sim \frac{1}{6}N^3$

Ex 3. $\frac{1}{6}N^3 - \underbrace{\frac{1}{2}N^2 + \frac{1}{3}N}_{\text{discard lower-order terms}} \sim \frac{1}{6}N^3$

(e.g., N = 1000: 166.67 million vs. 166.17 million)



Leading-term approximation

Technical definition. $f(N) \sim g(N)$ means $\lim_{N \to \infty} \frac{f(N)}{g(N)} = 1$

Tilde Notation (~) (same as θ-notation)

- Estimate running time (or memory) as a function of input size N.
- Ignore lower order terms.
  - when N is large, terms are negligible
  - when N is small, we don't care

operation	frequency	tilde notation
variable declaration	$N + 2$	$\sim N$
assignment statement	$N + 2$	$\sim N$
less than compare	$\frac{1}{2}(N + 1)(N + 2)$	$\sim \frac{1}{2} N^2$
equal to compare	$\frac{1}{2} N(N - 1)$	$\sim \frac{1}{2} N^2$
array access	$N(N - 1)$	$\sim N^2$
increment	$\frac{1}{2} N(N - 1)$ to $N(N - 1)$	$\sim \frac{1}{2} N^2$ to $\sim N^2$

Asymptotic notation: Ω (Omega)

$f(n) = O(g(n))$ iff there exist positive constants c and n_0 such that

$f(n) >= cg(n)$ for all n, $n >= n_0$

$3n+2 = \Omega(n)$	as $3n+2 >= 3n$	for all $n>= 1$
$10n^2+4n+2 = \Omega(n^2)$	as $10n^2 + 4n + 2 >= n^2$	for $n>=1$
$6*2^n + n^2 = \Omega(2^n)$	$6*2^n + n^2 >= 2^n$	for $n>=1$

$3n+3 = \Omega(n)$	Correct, OK.
$3n+3 = \Omega(1)$	Correct, NO!

Asymptotic notation: Θ (theta) (same as "~" notation)

$f(n) = O(g(n))$ iff there exist positive constants c_1, c_2, and n_0 such that

$$c_1 g(n) <= f(n) <= c_2 g(n) \text{ for all } n, n >= n_0$$

$3n+2 = \Theta(n)$	as $3n+2 >= 3n$	for all $n >= 1$
$10n^2+4n+2 = \Theta(n^2)$	as $10n^2 + 4n + 2 >= n^2$	for $n>=1$
$6*2^n + n^2 = \Theta(2^n)$	$6*2^n + n^2 >= 2^n$	for $n>=1$

$3n+3 = \Omega(n)$	Correct, OK.
$3n+3 = \Omega(2n)$	Correct, not used!

Linear time: O(n)

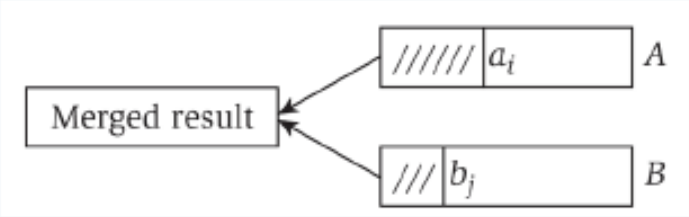
Linear time. Running time is proportional to input size.

Computing the maximum. Compute maximum of n numbers $a_1, \ldots, a_n$.

```
max ← a1
for i = 2 to n {
   if (ai > max)
      max ← ai
}
```

Linear time: O(n)

Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.




```
i = 1, j = 1
while (both lists are nonempty) {
   if (aᵢ ≤ bⱼ) append aᵢ to output list and increment i
   else         append bⱼ to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size n takes $O(n)$ time.

Pf. After each compare, the length of output list increases by 1.

Linearithmic time: O(n log n)

O(n log n) time. Arises in divide-and-conquer algorithms.

Sorting. Mergesort and heapsort are sorting algorithms that perform $O(n \log n)$ compares.

Largest empty interval. Given n time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval when no copies of file arrive?

O(n log n) solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

Quadratic time: $O(n^2)$

Ex. Enumerate all pairs of elements.

Closest pair of points. Given a list of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.

$O(n^2)$ solution. Try all pairs of points.

```
min ← (x₁ - x₂)² + (y₁ - y₂)²
for i = 1 to n {
   for j = i+1 to n {
      d ← (xᵢ - xⱼ)² + (yᵢ - yⱼ)²
      if (d < min)
         min ← d
   }
}
```

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion. [see Chapter 5]

Cubic time: $O(n^3)$

Cubic time. Enumerate all triples of elements.

Set disjointness. Given n sets $S_1, \ldots, S_n$ each of which is a subset of $1, 2, \ldots, n$, is there some pair of these which are disjoint?

$O(n^3)$ solution. For each pair of sets, determine if they are disjoint.

```
foreach set Si {
   foreach other set Sj {
      foreach element p of Si {
         determine whether p also belongs to Sj
      }
      if (no element of Si belongs to Sj)
         report that Si and Sj are disjoint
   }
}
```

Polynomial time: $O(n^k)$

Independent set of size k. Given a graph, are there k nodes such that no two are joined by an edge?

k is a constant

$O(n^k)$ solution. Enumerate all subsets of k nodes.

```
foreach subset S of k nodes {
   check whether S in an independent set
   if (S is an independent set)
      report S is an independent set
   }
}
```

- Check whether S is an independent set takes $O(k^2)$ time.
- Number of k element subsets = $\binom{n}{k} = \frac{n(n-1)(n-2)\times\cdots\times(n-k+1)}{k(k-1)(k-2)\times\cdots\times 1} \leq \frac{n^k}{k!}$
- $O(k^2\, n^k / k!) = O(n^k)$.

poly-time for k=17,
but not practical

Exponential time

Independent set. Given a graph, what is maximum cardinality of an independent set?

$O(n^2 2^n)$ solution. Enumerate all subsets.

```
S* ← φ
foreach subset S of nodes {
   check whether S in an independent set
   if (S is largest independent set seen so far)
      update S* ← S
   }
}
```

Sublinear time

Search in a sorted array. Given a sorted array A of n numbers, is a given number x in the array?

O(log n) solution. Binary search.

```
lo ← 1, hi ← n
while (lo ≤ hi) {
   mid ← (lo + hi) / 2
   if      (x < A[mid]) hi ← mid - 1
   else if (x > A[mid]) lo ← mid + 1
   else return yes
}
return no
```

References

- Kevin Wayne, “Analysis of Algorithms”
- Sartaj Sahni, “Analysis of Algorithms”
- BBM 201 Notes by Mustafa Ege
- Marty Stepp and Helene Martin, Recursion