Algorithm Design & Problem Solving:
Algorithm Analysis

Contents

The challenge

- Q. Will my program be able to solve a large practical input?

Why is my program so slow ?

Why does it run out of memory ?

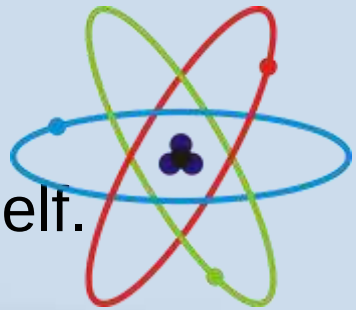
http://algs4.cs.princeton.edu

1.4 Analysis of Algorithms

- *introduction*
- *observations*
- *mathematical models*
- *order-of-growth classifications*
- *theory of algorithms*
- *memory*

Scientific method applied to analysis of algorithms

- A framework for predicting performance and comparing algorithms.
- Scientific method.
  - Observe some feature of the natural world.
  - Hypothesize a model that is consistent with the observations.
  - Predict events using the hypothesis.
  - Verify the predictions by making further observations.
  - Validate by repeating until the hypothesis and observations agree.
- Principles.
  - Experiments must be reproducible.
  - Hypotheses must be falsifiable.
- Feature of the natural world. Computer itself.



Example: 3-SUM

- 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.

3-SUM: brute-force algorithm

```
int find3NumbersforSum(int A[], int arr_size, int sum)
{
    int i,j,k count=0;
    for ( i = 0; i < arr_size - 2; i++) {
        for (j = i + 1; j < arr_size - 1; j++) {
             for (k = j + 1; k < arr_size; k++)
                if (A[i] + A[j] + A[k] == sum)  count++;
        }
    }
    return count;
}
```

check each triple

for simplicity, ignore integer overflow

Measuring the running time

- Q. How to time a program?
- A. Manual.

```
% java ThreeSum 1Kints.txt
```

```
tick tick tick
70
% java ThreeSum 2Kints.txt
```

```
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
528
% java ThreeSum 4Kints.txt
```

```
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
4039
```

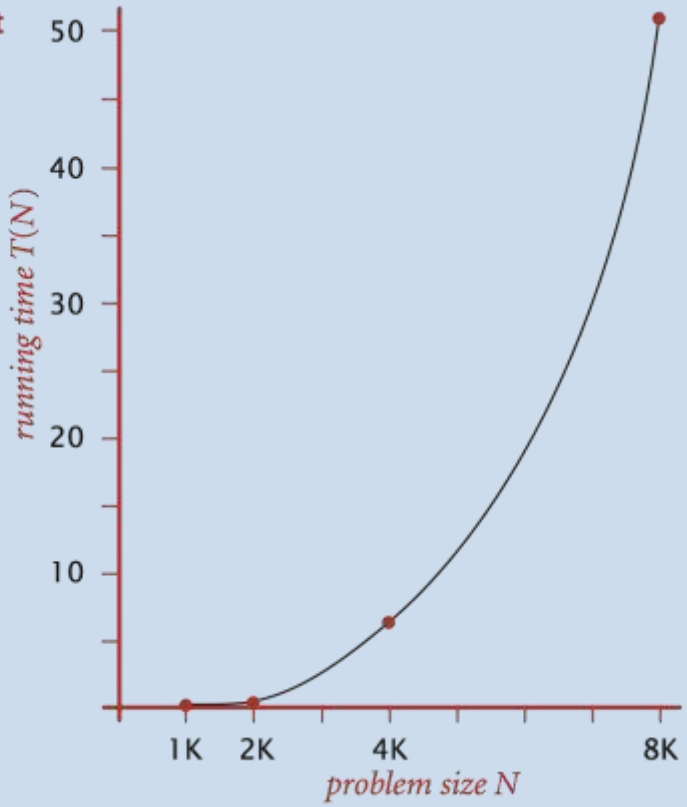
Empirical analysis

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

N	time (seconds) †
250	0
500	0
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1
16,000	?

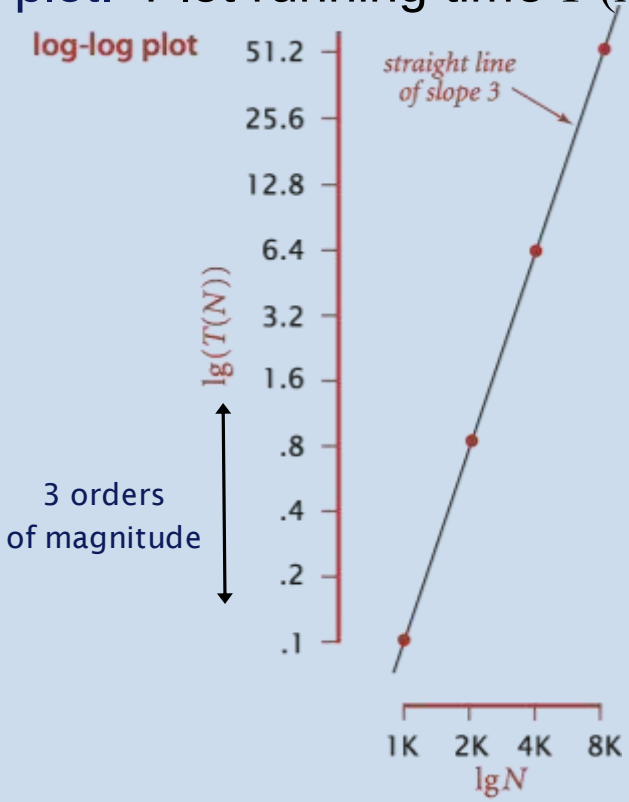
Data analysis

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



Data analysis

- Log-log plot. Plot running time $T(N)$ vs. input size N using log-log scale.



eqn of straight line :-y=bx+c

$\lg(T(N)) = b \lg N + c$

$b = 2.999$

$c = -33.2103$

$T(N) = a N^{b}$, *where* $a = 2^{c}$

power law

- Regression. Fit straight line through data points: $a N^{b}$.

slope

- Hypothesis. The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.

Prediction and validation

- Hypothesis. The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.

"order of growth" of running time is about N^3 [stay tuned]

Predictions.

- 51.0 seconds for $N = 8{,}000$.
- 408.1 seconds for $N = 16{,}000$.

Observations.

N	time (seconds) †
8,000	51.1
8,000	51
8,000	51.1
16,000	410.8

validates hypothesis!

Experimental algorithmics

- System independent effects.
  - Algorithm.
  - Input data.

determines exponent in power law

- System dependent effects.
  - Hardware: CPU, memory, cache, …
  - Software: compiler, interpreter, garbage collector, …
  - System: operating system, network, other apps, …

determines constant in power law

Bad news. Difficult to get precise measurements.

Good news. Much easier and cheaper than other sciences.

e.g., can run huge number of experiments

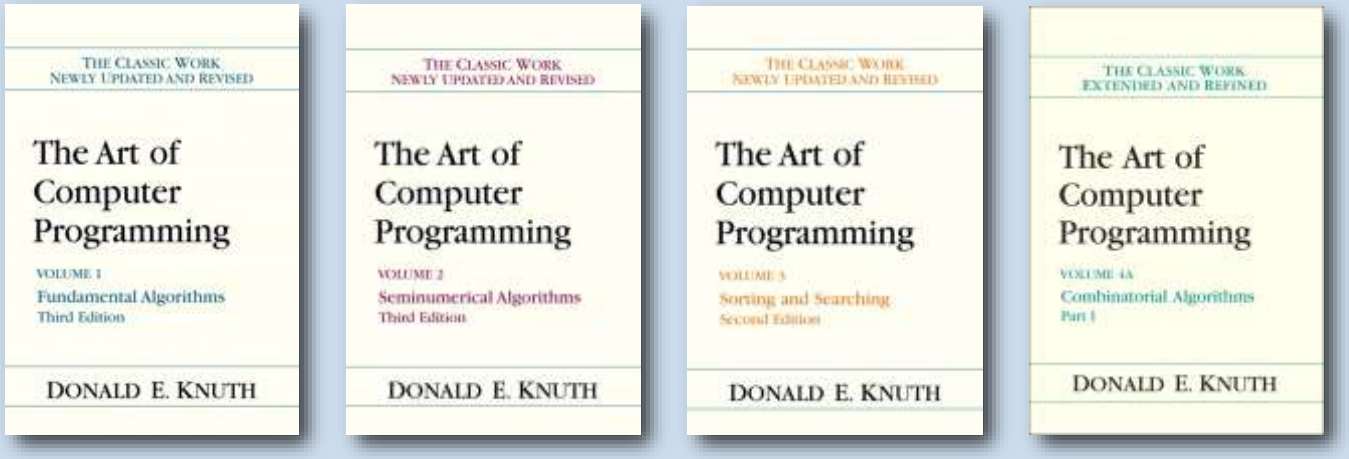
http://algs4.cs.princeton.edu

1.4 Analysis of Algorithms

- *introduction*
- *observations*
- *mathematical models*
- *order-of-growth classifications*
- *theory of algorithms*
- *memory*

Mathematical models for running time

- Total running time: sum of cost $\times$ frequency for all operations.
  - Need to analyze program to determine set of operations.
  - Cost depends on machine, compiler.
  - Frequency depends on algorithm, input data.



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
...	...	...

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

Example: 1-Sum

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

```
int i,count = 0;
for ( 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 (i++ plus count++)	N to $2\,N$

Example: 2-Sum

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

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

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 $N(N-1)$

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

tedious to count exactly

Simplification 2: 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

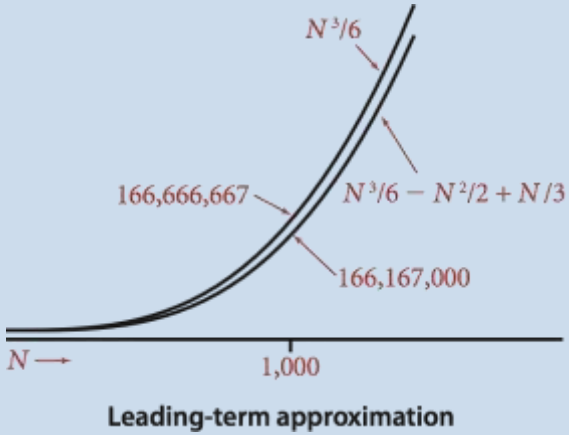
Ex 1. $\frac{1}{6} N^3 + 20 N + 16 \sim \frac{1}{6} N^3$

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

Ex 3. $\frac{1}{6} N^3 - \frac{1}{2} N^2 + \frac{1}{3} N \sim \frac{1}{6} N^3$

discard lower-order terms

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



Leading-term approximation

Technical definition. $f(N) \sim g(N)$ means

$$\lim_{N \to \infty} \frac{f(N)}{g(N)} = 1$$

Simplification 2: 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

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 $\sim N^2$

Example: 2-Sum

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

```
int i,j,count = 0;
for (i = 0; i < N; i++)
  for (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.
- Bottom line. Use cost model and tilde notation to simplify counts.

Example: 3-Sum

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

```
int i,j,k,count = 0;
for ( i = 0; i < N; i++) {
  for ( j = i+1; j < N; j++) {
    for ( 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.
- Bottom line. Use cost model and tilde notation to simplify counts.

3-Sum Approximate Running Time

- Q. How long will it take to run?

A	1	`int i,j,k,count = 0;`
		`for ( i = 0; i < N; i++) {`
B	N	`for ( j = i+1; j < N; j++) {`
C	$\sim N^2/2$	`for ( k = j+1; k < N; k++) {`
D	$\sim n^3/6$	`if (a[i] + a[j] + a[k] == 0)` "inner loop"
E	input dependent	`count++;`

- Only count instructions that execute most frequently – the inner loop (D)
- So, including all the variable init and add ops
  - Total Time T(N) := $(T_1 * A) + (T_2 * B) + (T_3 * C) + (T_4 * D) + (x * E)$
  - Note that E is execution dependent – how many of N's add up to 0

Order of Growth

- T(n) of 3sum function is ~$N^3/6$
- **Order of Growth** of function is N^3

- Note that this reflects what we saw with the power law aN^3 where *a* is some machine dependent factor

http://algs4.cs.princeton.edu

1.4 Analysis of Algorithms

- *introduction*
- *observations*
- *mathematical models*
- *order-of-growth classifications*
- *theory of algorithms*
- *memory*

Common order-of-growth classifications

- Definition. If $f(N) \sim c\, g(N)$ for some constant $c > 0$, then the order of growth of $f(N)$ is $g(N)$.
  - Ignores leading coefficient.
  - Ignores lower-order terms.

- Ex. The order of growth of the running time of this code is N^3.

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

Typical usage. With running times.

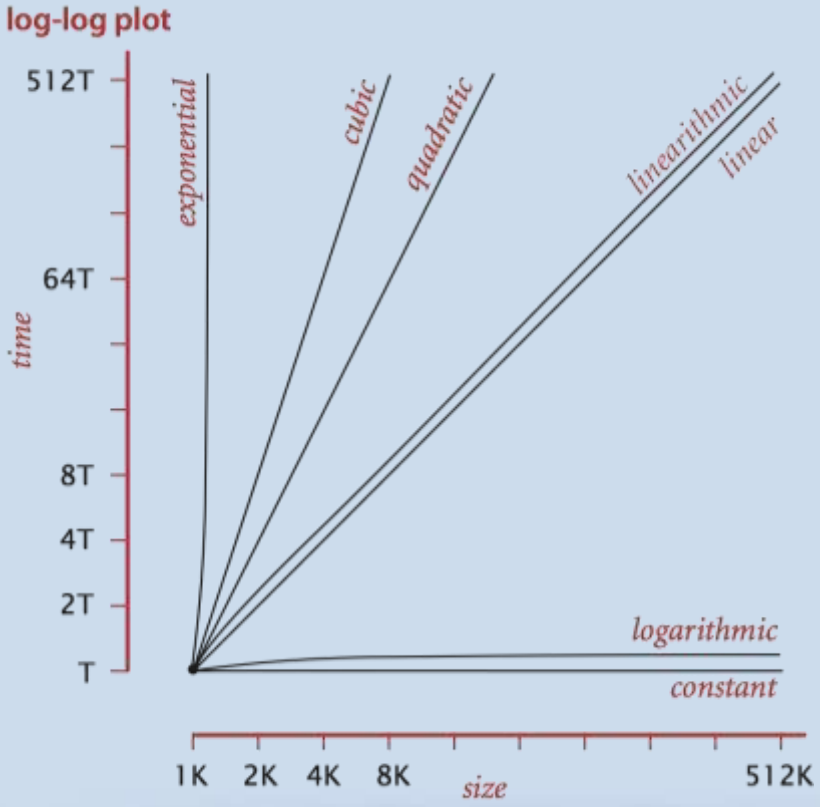
where leading coefficient depends on machine, compiler, JVM, ...

Common order-of-growth classifications

- Good news. The set of functions

  $1,\ \log N,\ N,\ N \log N,\ N^2,\ N^3$, and 2^N

- suffices to describe the order of growth of most common algorithms.



Typical orders of growth

Common order-of-growth classifications

order of growth	name	typical code framework	description	example	$T(2N)$ / $T(N)$
1	**constant**	a = b + c;	statement	add two numbers	1
$\log N$	**logarithmic**	while (N > 1) { N = N / 2; ... }	divide in half	binary search	~ 1
N	**linear**	for (int i = 0; i < N; i++) { ... }	loop	find the maximum	2
$N \log N$	**linearithmic**	[see mergesort lecture]	divide and conquer	mergesort	~ 2

N^2	**quadratic**	for (int i = 0; i < N; i++)
		for (int j = 0; j < N; j++)
		{ ... }
			double loop	check all pairs	4

N^3	**cubic**	for (int i = 0; i < N; i++)
		for (int j = 0; j < N; j++)
		for (int k = 0; k < N; k++)
		{ ... }
			triple loop	check all triples	8

2^N	**exponential**	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$

Practical implications of order-of-growth

growth rate	problem size solvable in minutes			
	1970s	1980s	1990s	2000s
1	any	any	any	any
log N	any	any	any	any
N	millions	tens of millions	hundreds of millions	billions
N log N	hundreds of thousands	millions	millions	hundreds of millions
N^2	hundreds	thousand	thousands	tens of thousands
N^3	hundred	hundreds	thousand	thousands
2^N	20	20s	20s	30

Practical implications of order-of-growth

growth rate	problem size solvable in minutes				time to process millions of inputs			
	1970s	1980s	1990s	2000s	1970s	1980s	1990s	2000s
1	any	any	any	any	instant	instant	instant	instant
log N	any	any	any	any	instant	instant	instant	instant
N	millions	tens of millions	hundreds of millions	billions	minutes	seconds	second	instant
N log N	hundreds of thousands	millions	millions	hundreds of millions	hour	minutes	tens of seconds	seconds
N^2	hundreds	thousand	thousands	tens of thousands	decades	years	months	weeks
N^3	hundred	hundreds	thousand	thousands	never	never	never	millennia

Practical implications of order-of-growth

growth rate	name	description	effect on a program that runs for a few seconds	
			time for 100x more data	size for 100x faster computer
1	constant	independent of input size	–	–
log N	logarithmic	nearly independent of input size	–	–
N	linear	optimal for N inputs	a few minutes	100x
N log N	linearithmic	nearly optimal for N inputs	a few minutes	100x
N^2	quadratic	not practical for large problems	several hours	10x
N^3	cubic	not practical for medium problems	several weeks	4–5x
2^N	exponential	useful only for tiny problems	forever	1x

http://algs4.cs.princeton.edu

1.4 Analysis of Algorithms

- *introduction*
- *observations*
- *mathematical models*
- *order-of-growth classifications*
- *theory of algorithms*
- *memory*

Types of analyses

- Best case. Lower bound on cost.
  - Determined by "easiest" input.
  - Provides a goal 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.

this course

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: $\sim \lg N$

Theory of algorithms

- Goals.
  - Establish “difficulty” of a problem.
  - Develop “optimal” algorithms.

- Approach.
  - Suppress details in analysis: analyze “to within a constant factor.”
  - Eliminate variability in input model: focus on the worst case.

- Upper bound. Performance guarantee of algorithm for any input.
- Lower bound. Proof that no algorithm can do better.
- Optimal algorithm. Lower bound = upper bound (to within a constant factor).

Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for	used to
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N$ $\log N + 3N$ ⋮	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3$ N ⋮	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log$ $N + 3 N$ ⋮	develop lower bounds

Theory of algorithms: example 1

- Goals.
  - Establish "difficulty" of a problem and develop "optimal" algorithms.
  - Ex. 1-SUM = "*Is there a* 0 *in the array?* "
- Upper bound. A specific algorithm.
  - Ex. Brute-force algorithm for 1-SUM: Look at every array entry.
  - Running time of the optimal algorithm for 1-SUM is $O(N)$.
- Lower bound. Proof that no algorithm can do better.
  - Ex. Have to examine all *N* entries (any unexamined one might be 0).
  - Running time of the optimal algorithm for 1-SUM is $\Omega(N)$.
- Optimal algorithm.
  - Lower bound equals upper bound (to within a constant factor).
  - Ex. Brute-force algorithm for 1-SUM is optimal: its running time is $\Theta(N)$.

Theory of algorithms: example 2

- Goals.
  - Establish “difficulty” of a problem and develop “optimal” algorithms.
  - Ex. 3-SUM.

- Upper bound. A specific algorithm.
  - Ex. Brute-force algorithm for 3-SUM.
  - Running time of the optimal algorithm for 3-SUM is $O(N^3)$.

Theory of algorithms: example 2

- Goals.
  - Establish “difficulty” of a problem and develop “optimal” algorithms.
  - Ex. 3-SUM.
- Upper bound. A specific algorithm.
  - Ex. Improved algorithm for 3-SUM.
  - Running time of the optimal algorithm for 3-SUM is $O(N^2 \log N)$.
- Lower bound. Proof that no algorithm can do better.
  - Ex. Have to examine all N entries to solve 3-SUM.
  - Running time of the optimal algorithm for solving 3-SUM is $\Omega(N)$.
- Open problems.
  - Optimal algorithm for 3-SUM?
  - Subquadratic algorithm for 3-SUM?
  - Quadratic lower bound for 3-SUM?

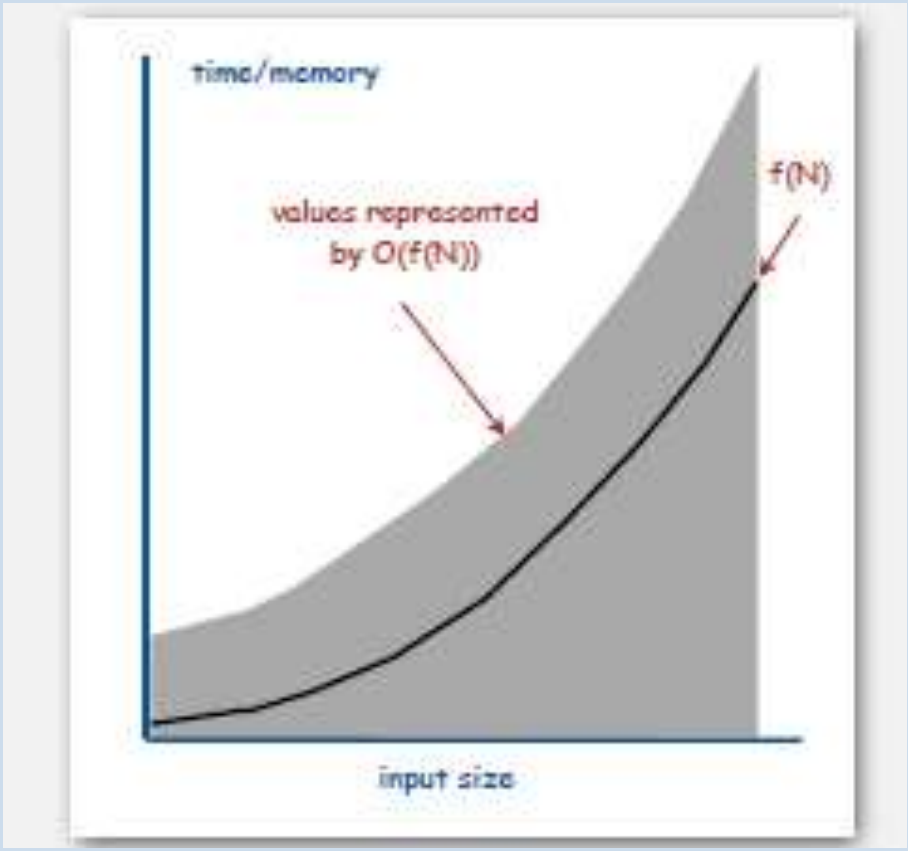
Algorithm design approach

- Start.
  - Develop an algorithm.
  - Prove a lower bound.
- Gap?
  - Lower the upper bound (discover a new algorithm).
  - Raise the lower bound (more difficult).
- Golden Age of Algorithm Design.
  - 1970s-.
  - Steadily decreasing upper bounds for many important problems.
  - Many known optimal algorithms.
- Caveats.
  - Overly pessimistic to focus on worst case?
  - Need better than “to within a constant factor” to predict performance.

Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for	used to
Tilde	leading term	$\sim 10\,N^2$	$10\,N^2$ $10\,N^2 + 22\,N$ $\log N$ $10\,N^2 + 2\,N + 37$	provide approximate model
Big Theta	Asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2}\,N^2$ $10\,N^2$ $5\,N^2 + 22\,N$ $\log N + 3N$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10\,N^2$ $100\,N$ $22\,N \log N + 3\,N$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2}\,N^2$ N^5 $N^3 + 22\,N \log N + 3\,N$	develop lower bounds

big-Oh notation



- Big Oh/Big O
  - Worst Case performance
  - Big Oh is not going to show exact timing of algorithm
  - Just how long it takes to run with its worst case data set in terms of the number of items (N) being processed

Big-O Notation

Test yourself!

What is the worst-case complexity of the each of the following code fragments?

1

```
for (i = 0; i < N; i++)
   …sequence of statements

for (j = 0; j < M; j++) {
   …sequence of statements }
```

How would the complexity change if the second loop went to N instead of M?

2

```
for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
      …sequence of statements
   }
}

for (k = 0; k < N; k++) {
   …sequence of statements
}
```

Give the order of growth (as a function of N) of the running times of each of the following code fragments:

```
int sum = 0;
for (int k = N; k > 0; k = k/2)
   for (int i = 0; i < k; i++)
      sum++;
```

```
int sum = 0;
for (int i = 1; i < N; i = i* 2)
   for(int j = 0; j < i; j++)
      sum++;
```

```
int sum = 0;
for (int i = 1; i < N; i *= 2)
   for (int j = 0; j < N; j++)
      sum++;
```

Give the order of growth (as a function of N) of the running times of each of the following code fragments:

1

```
void function(int n)
{
 if (n==1)
   return;
 for (int i=1; i<=n; i++) {
   for (int j=1; j<=n; j++){
     putchar('*');
     break;
   }
 }
}
```

2

```
void function(int n)
{
    int count = 0;
    for (int i=n/2; i<=n; i++)
        for (int j=1; j<=n; j=2 * j)
            for (int k=1;k<=n;k=k * 2)
                count++;
}
```

3

```
void function(int n)
{
  int count = 0;
  for (int i=n/2; i<=n; i++)
    for (int j=1;j+n/2<=n; j=j++)
      for (int k=1;k<=n;k=k*2)
                count++;
}
```

4

```
void function(int n)
{
  int count = 0;
  for (int i=0; i<n; i++)
    for (int j=i; j< i*i; j++)
      if (j%i == 0){
        for (int k=0; k<j; k++)
          printf("*");
      }
}
```

Thank You !


OLLSCOIL TEICNEOLAÍOCHTA
BHAILE ÁTHA CLIATH
TU DUBLIN
TECHNOLOGICAL
UNIVERSITY DUBLIN