Suppose that n= 2^k -1 for some integer k ≥1.

1. Solve this recurrence relation:

T(1)= 1, and

otherwise, T(n)= 1 + T$\left(\frac{n-1}{2}\right)$.

2. Prove that T(n) is $\theta(f(n))$ for a function f(n) that is as simple as possible.

Assignment 2A Programming and Assignment 1A resubmissions are due on Thursday Oct. 12 at 11:55pm.

When you upload your files for Assignment 2A:

1. Make sure you hit SUBMIT EVERY time you revise your files. There is no harm in doing this since I have set connex up to allow an unlimited number of resubmissions.
2. For 2A submissions: make sure that for each of LinkedList.java and BigIntegerList.java, you only have ONE version of each file. Delete old versions. If you have more than one version your code will not compile.

Review Lecture 12:
Assignment 1A: Hints for writing good programs before submitting your programs.
Code submitted for this class should be elegant and efficient (subject to meeting the constraints given: reverse should be a divide and conquer method that splits the list in half).

Midterm tutorial: Tuesday Oct. 17, 7pm-10pm, Elliott 168.

Assignment 2B Written is due at the beginning of class on Monday Oct. 16.

In order to not disadvantage the Friday tutorial students with respect to the midterm exam:

There will be tutorial: Friday Oct. 13

There will be no tutorial: Friday Oct. 20

If you have tutorial on Fridays and cannot attend on Oct. 13, then you are welcome to attend any one of the other sections:

On Friday Oct 13:

B06 ECS 258 F 13:30-14:20

B07 ECS 258 F 14:30-15:20

On Monday Oct. 16:

B01 ECS 258 M 13:30-14:20

B02 ECS 258 M 14:30-15:20

B03 ECS 258 M 15:30-16:20

On Tuesday Oct. 17:

B04 ECS 258 T 09:30-10:20

B05 ECS 258 T 10:30-11:20

Slides with gray backgrounds are taken from:
http://algs4.cs.princeton.edu/lectures/14AnalysisOfAlgorithms-2x2.pdf

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 × 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]`	$c_7\ N^2$

Java takes this time because it initializes arrays.

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 $2N$

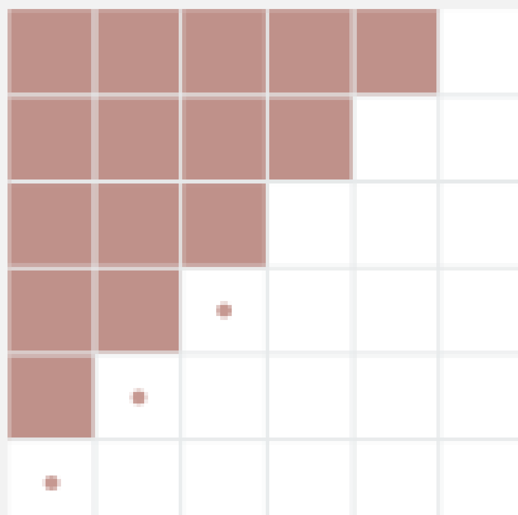
Example: 2-SUM

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 1: 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++;
```

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

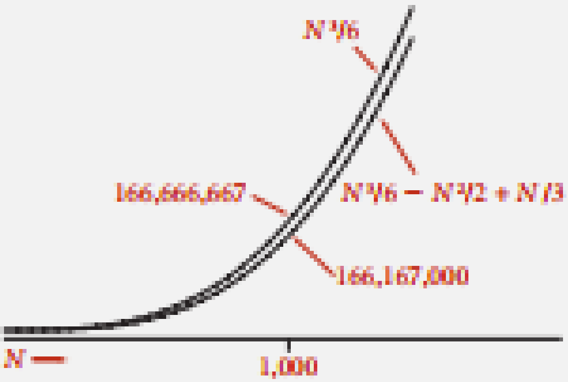
operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	$\frac{1}{2}(N+1)(N+2)$
equal to compare	$\frac{1}{2} N(N-1)$
array access	$N(N-1)$
increment	$\frac{1}{2} N(N-1)$ to $N(N-1)$

cost model = array accesses

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

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



Leading-term approximation

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

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

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

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

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	$½ (N + 1) (N + 2)$	$\sim ½ N^2$
equal to compare	$½ N (N - 1)$	$\sim ½ N^2$
array access	$N (N - 1)$	$\sim N^2$
increment	$½ N (N - 1)$ to $N (N - 1)$	$\sim ½ N^2$ to $\sim N^2$

More widely accepted notation:

Assume that T, f and g are functions mapping the natural numbers {0, 1, 2, 3, ...} into the reals.

Definition: "Big Oh" A function T(n) is in $O(f(n))$ if there exist constants $n_0 \geq 0$, and $c > 0$, such that for all $n \geq n_0$, $T(n) \leq c * f(n)$.

Definition: "Omega" A function T(n) is in $\Omega(f(n))$ if there exist constants $n_0 \geq 0$, and $c > 0$, such that for all $n \geq n_0$, $T(n) \geq c * f(n)$.

Definition: "Theta" The set $\Theta(g(n))$ of functions consists of $\Omega(g(n)) \cap O(g(n))$.

Example: 2-SUM

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.

What is this using θ notation?

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 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.

What is this using θ notation?

Bottom line. Use cost model and tilde notation to simplify counts.

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 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++;
```

Typical usage. With running times.

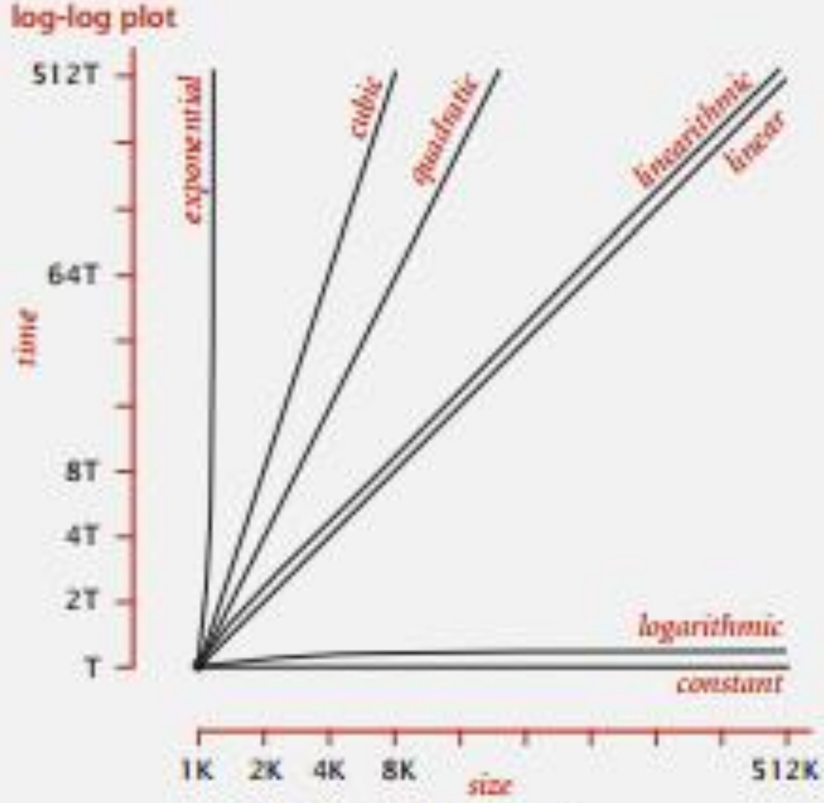
where leading coefficient depends on machine, compiler, JVM, ...

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	$T(N)$

State a recurrence relation T(n) for the Big Oh time complexity of the monday method on the next slide.

If n= 2^k, what is the value of x after the call:

int x= monday(0, n, n);

```
public static int monday(int level, int n, int original_n)
{   int i, j, sum, silly_sum;

    if (n==1) return(original_n);

    silly_sum=0;
    for (i=0; i < n; i++)
            for (j= i+1; j< n; j++)
                silly_sum++;

    sum=   monday(level+1, n/2, original_n);
    sum+= monday(level+1, n/2, original_n);
    sum+= monday(level+1, n/2, original_n);

    for (i=0; i < n; i++) silly_sum++;

    return(sum);
}
```

Binary search: Java implementation

Trivial to implement?

- First binary search published in 1946.
- First bug-free one in 1962.
- Bug in Java's `Arrays.binarySearch()` discovered in 2006.

```
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;
}
```

one "3-way compare"

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

Recursive code:

```
public static int binary_search(
        int level, int key,
        int [] A, int lo, int hi)

 {
          int mid, pos;

     //   Entry is not in the array.

          if (lo > hi) return(-1);

          mid= lo + (hi- lo)/2;
```

```
    if (key < A[mid])
    {
        pos= binary_search(level+1, key,
                                   A, lo, mid-1);
        return(pos);
     }
     else if (key > A[mid])
     {
        pos= binary_search(level+1, key,
                                   A, mid+1, hi);
        return(pos);
     }
     else return(mid);
}
```

0	1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18	20

For this example: A.length= 10

The initial call for a key is:

```
int pos= binary_search(0, key,
                       A, 0, A.length-1);
```

Search for 14:

```
mid= lo + (hi- lo)/2;
A[0 ... 9]  mid = 4
```

0	1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18	20

```
A[5 ... 9]  mid = 7
```

0	1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18	20

Search for 14:

mid= lo + (hi- lo)/2;

A[5 ... 6] mid = 5

0	1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18	20

A[6 ... 6] mid = 6

0	1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18	20

returns 6

Search for 15:

```
mid= lo + (hi- lo)/2;
A[0 ... 9]  mid = 4
```

0	1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18	20

```
A[5 ... 9]  mid = 7
```

0	1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18	20

Search for 15:

```
mid= lo + (hi- lo)/2;
A[5 ... 6]  mid = 5
```

0	1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18	20

```
A[6 ... 6]  mid = 6
```

0	1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18	20

A[7 ... 6] Empty subproblem: lo > hi

```
returns -1
```

On which problem sizes are the left and right subproblems equal in length at every step?
Give a recurrence relation for the time complexity of binary search and solve it.

How much time does binary search take:

1. In the best case?
2. In the worst case for a successful search?
3. On average for a successful search?
4. On average for an unsuccessful search?

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.

this course

Average case. Expected cost for random input.

- Need a model for "random" input.
- Provides a way to predict performance.

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
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

Common mistake. Interpreting big-Oh as an approximate model.

Text: Focus on approximate models: use Tilde-notation

CSC 225: Uses Big Theta, Big Oh and Big Omega notation.