What is an Algorithm?

- Step-by-step procedure used to solve a problem
- These steps should be capable of being performed by a machine
- Must eventually stop and so produce an answer

Types of Algorithms

- Iterative Algorithms
- Recursive Algorithms (Divide & Conquer)
- Dynamic Programming
- Greedy Algorithms
- Randomized Algorithms
- Approximation Algorithms
- Genetic Algorithms

Examples of famous algorithms

- Constructions of Euclid
- Newton's root finding
- Fast Fourier Transform
- Compression (Huffman, Lempel-Ziv, GIF, MPEG)
- RSA encryption
- Simplex algorithm for linear programming
- Shortest Path Algorithms (Dijkstra, Bellman-Ford)
- Error correcting codes (CDs, DVDs)
- TCP congestion control, IP routing
- Pattern matching (Genomics)
- Search Engines

Role of Algorithms in Modern World

- Enormous amount of data
  - E-commerce (Amazon, Ebay)
  - Network traffic (telecom billing, monitoring)
  - Database transactions (Sales, inventory)
  - Scientific measurements (astrophysics, geology)
  - Sensor networks. RFID tags
  - Bioinformatics (genome, protein bank)

- Computer Scientists focus on problems such as
  - How fast do algorithms run
  - How much memory does the process require

- Example Applications
  - Make the Internet run faster
    - Pink-Degemark's routing algorithm

Algorithm Analysis

- Algorithm analysis is about predicting or computing the resources that an algorithm requires during its execution

- The running time of an algorithm can be thought of as the number of computation steps performed given a particular input.

Why Study Algorithm Analysis?

- Many different algorithms may correctly solve a given problem
- But choice of a particular algorithm may have enormous impact on time and memory used
  - Time vs space tradeoffs are very common
- So we need to understand the mathematical fundamentals needed to analyze algorithms
- Learn how to compare the efficiency of different algorithms in terms of running time and memory usage

Analyzing Running Time

- We will analyze running time (RT) of our algorithms
  - RT: the amount of time it takes for the algorithm to finish execution on a particular input size
  - More precisely, the RT of an algorithm on a particular input is the number of primitive operations or steps executed.
  - We define a step to be a unit of work that can be executed in constant amount of time in a machine.
  - Notice that each line of an algorithm might take a different amount of time.

Efficiency of Algorithms

- Time efficiency
- Space efficiency

- There may be several algorithms to solve the same problem
- Sometimes a trade-off between time and space is necessary

Factors affecting/ masking speed

- Number of input values
- Processor speed
- Number of simultaneous programs etc
- The location of items in a data structure
- Different implementations (Software differences)

Time efficiency

- Count characteristic operations
    - eg arithmetical operations eg multiplications
- Examine critical section/s of algorithm
  - Single statement/group of statements/single operation
  - Central to functioning of algorithm contained in most deeply nested loops of algorithm

Analysis of Algorithms

- We have ways to compare algorithms
  - Generally, the larger the problem, the longer it takes the algorithm to complete
  - Sorting 100,000 elements can take much more time than sorting 1,000 elements
    - and more than 10 times longer
  - the variable *n* suggests the "*n*umber of things"
  - If an algorithm requires $0.025n^2 + 0.012n + 0.0005$ seconds, just plug in a value for n

A Computational Model

- To summarize algorithm runtimes, we can use a computer independent model
  - instructions are executed sequentially
  - count all assignments, comparisons, and increments there is infinite memory
  - every simple instruction takes one unit of time

Another Example: Searching for a number in an array of numbers

- Assume int X[N] of integer is our data set and we are searching for “key”

	Cost	Times
Found = 0;	C0	1
I = 0;	C1	1
while (!found && i < N){	C2	0 <= L < N
If (key ==X[I]) found = 1;	C3	1 <= L <= N
I++;	C4	1 <= L <= N
}		

T(n) = C0 + C1 + L*(C2 + C3 + C4), where 1 <= L <= N is the number of times that the loop is iterated.

Different Cases

- The total cost of sequential search is 3n + 2
  - But is it always exactly 3n + 2 instructions?
  - The last assignment does not always execute
    - But does one assignment really matter?
  - How many times will the loop actually execute?
    - that depends
  - If searchID is found at index 0: _____ iterations
    - best case
  - If searchID is found at index n-1:_____ iterations
    - worst case
  - What is typical?

Different Cases

- Best case -> what are the properties of the data set that will lead to the fewest number of executable statements (steps in the algorithm)

- Worst case -> what are the properties of the data set that will lead to the largest number of executable statements

- Average case -> Usually this means assuming the data is randomly distributed

Example2: Searching for a number in an array of numbers (continued)

- What's the best case? Loop iterates just once => Ω
  - T(n) = C0 + C1 + C2 + C3 + C4
- What's the average (expected) case? Loop iterates N/2 times => Θ
  - T(n) = C0 + C1 + N/2 * (C2 + C3 + C4)
  - Notice that this can be written as T(n) = a + b*n where a, b are constants
- What's the worst case? Loop iterates N times => O
  - T(n) = C0 + C1 + N * (C2 + C3 + C4)
  - Notice that this can be written as T(n) = a + b*n where a, b are constants

Worst Case Analysis of Algorithms

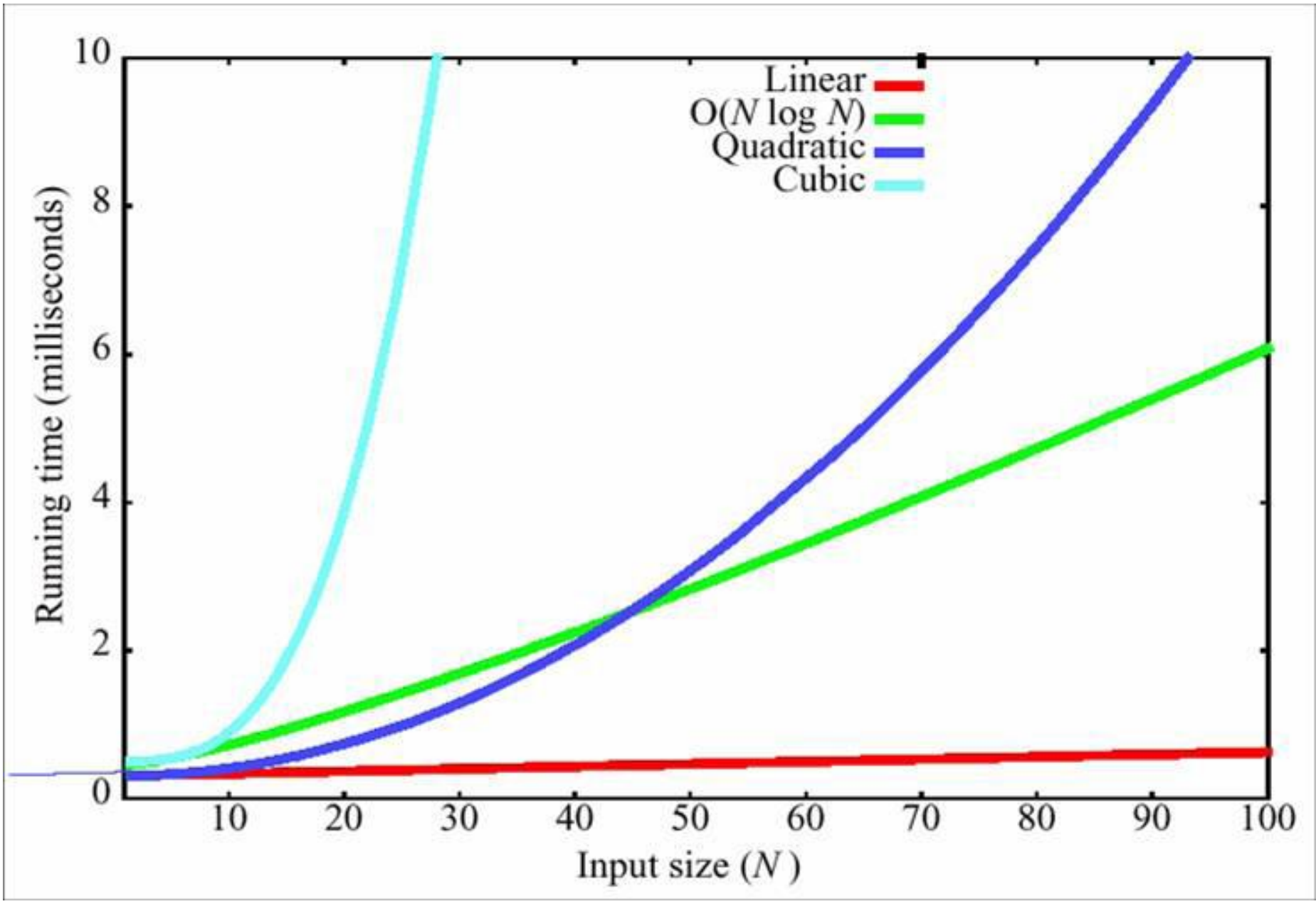
- We will only look at WORST CASE running time of an algorithm. Why?
  - Worst case is an upper bound on the running time. It gives us a guarantee that the algorithm will never take any longer
  - For some algorithms, the worst case happens fairly often. As in this search example, the searched item is typically not in the array, so the loop will iterate N times
  - The “average case” is often roughly as bad as the “worst case”. In our search algorithm, both the average case and the worst case are linear functions of the input size “n”

Common Functions we will encounter

Increasing cost

Name	Big-Oh	Comment
Constant	O(1)	Can't beat it!
Log log	O(loglogN)	Extrapolation search
Logarithmic	O(logN)	Typical time for good searching algorithms
Linear	O(N)	This is about the fastest that an algorithm can run given that we need O(n) just to read the input
N logN	O(NlogN)	Most sorting algorithms
Quadratic	$O(N^2)$	Acceptable when the data size is small (N<1000)
Cubic	$O(N^3)$	Acceptable when the data size is small (N<1000)
Exponential	$O(2^N)$	Only good for really small input sizes (n<=20)

Polynomial time


Linear
O(N log N)
Quadratic
Cubic
Running time (milliseconds)
Input size (N)
0
2
4
6
8
10
10
20
30
40
50
60
70
80
90
100

- Imagine a program that in the worst case takes 400 + 21n2 + 8n3 steps. We simply say this program has cubic complexity.

- Other cubic functions:
  - n3
  - 100n3
  - 100000n3 + 4293n2 + 9400n

- The dominant term is a constant multiplied by n3.
- We say these functions are O(n3) in “big oh” notation.
- The big oh notation represents the growth rate of a function

Runtime Analysis

- Useful rules
  - simple statements (read, write, assign)
    - $O(1)$ (constant)
  - simple operations (+ - * / == > >= < <=
    - $O(1)$
  - sequence of simple statements/operations
    - rule of sums
  - for, do, while loops
    - rules of products

Runtime Analysis

- Two important rules
  - Rule of sums
    - if you do a number of operations in sequence, the runtime is dominated by the most expensive operation
  - Rule of products
    - if you repeat an operation a number of times, the total runtime is the runtime of the operation multiplied by the iteration count

Runtime Analysis

if (cond) then	$O(1)$
$\quad body_1$	$T_1(n)$
else	
$\quad body_2$	$T_2(n)$
endif	

$T(n) = O(max\ (T_1(n),\ T_2(n))$

Runtime Analysis (cont.)

- Method calls
  - A calls B
  - B calls C
  - etc.
- A sequence of operations when call sequences are flattened

$$T(n) = max(T_A(n), T_B(n), T_C(n))$$

Simple Instructions

- Count the simple instructions
  - assignments have cost of 1
  - comparisons have a cost of 1
  - let's count all parts of the loop
- for(int j = 0; j < n; j++)
  - j=0 has a cost of 1, j<n executes n+1 times,and j++ executes n times for a total cost of 2n+2
  - each statement in the repeated part of a loop have have a cost equal to number of iterations

Examples

- Cost
- sum = 0; ---> 1
- sum = sum + next; ---> 1 *Total Cost: 2*
- Cost
- for(int i = 1; i <= n; i++) ---> 1 + n+1 + n = 2n+2
- sum = sum++; ---> n *Total Cost: 3n + 2*

- Cost
- k = 0 ---> 1
- for(int i = 0; i < n; i++) ---> 2n+2
- for(int j = 0; j < n; j++) ---> n(2n+2) = $2n^2$ +2n
- k++; ---> n^2 *Total Cost: $3n^2 + 4n + 3$*

Total Cost of Sequential Search

```
                                                                    Cost
BankAccount searchedForAccount = null;                              ---> 1
int index = 0;                                                      ---> 1
while( index < n &&                          ---> n
      !searchID.equals(accounts[index].getID())) {  ---> n
  index++;                                                                  ---> n
}

if( index < n )                                                     ---> 1
  searchedForAccount = accounts[index];                             ---> 1

// searchedForAccount is null or refers to the desired object
```

- $Total\ cost = 3n+4$

Complexities in Increasing Order

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n \log n)$
5. $O(n^2)$
6. $O(n^3)$
7. $O(2^n)$

For large problems always choose an asymptotically superior algorithm (ie an algorithm of the lowest order possible)

Feasible algorithms are algorithms which are fast enough to be used in practice. (1-4 are always feasible)

Runtimes with for loops

- int n = 1000;
- int[] x = new int[n];

  - O(n)

```
for( int j = 0; j < n; j++ )
 x[j] = 0;
```

  - $O(n^2)$

```
int sum = 0;
for( int j = 0; j < n; j++ )
  for( int k = 0; k < n; k++ )
    sum += j * k;
```

```
int maxSum = 0;

for( int i = 0; i < a.size( ); i++ )
for( int j = i; j < a.size( ); j++ )
{
    int thisSum = 0;
    for( int k = i; k <= j; k++ )
        thisSum += a[ k ];
    if( thisSum > maxSum )
        maxSum   = thisSum;
}
return maxSum;
```

Example

```
for (i=1; i<n; i++)
    if A(i) > maxVal then
        maxVal= A(i);
        maxPos= i;
```

Asymptotic Complexity: O(n)

Example

```
for (i=1; i<n-1; i++)
    for (j=n; j>= i+1;  j--)
        if (A(j-1) > A(j)) then
            temp = A(j-1);
            A(j-1) = A(j);
            A(j) = tmp;
        endif
    endfor
endfor
```

- Asymptotic Complexity is $O(n^2)$

Run Time for Recursive Programs

- T(n) is defined recursively in terms of T(k), k<n
- The recurrence relations allow T(n) to be "unwound" recursively into some base cases (e.g., T(0) or T(1)).

- Examples:
  - Factorial

Example Algorithms

- Two algorithms for computing the Factorial
- Which one is better?

- ```
  int factorial (int n) {
    if (n <= 1)    return 1;
    else   return n * factorial(n-1);
  }
  ```

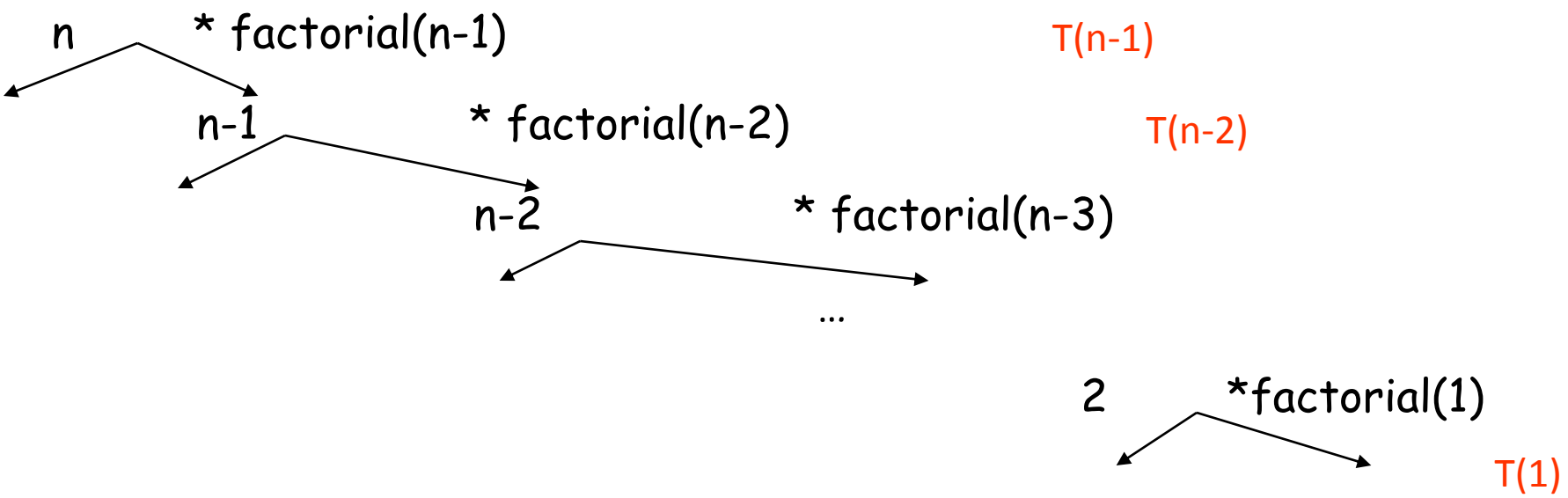
- ```
  int factorial (int n) {
    if (n<=1)    return 1;
    else {
      fact = 1;
      for (k=2; k<=n; k++)
          fact *= k;
      return fact;
    }
  }
  ```

Example: Factorial

```
int factorial (int n) {
  if (n<=1)   return 1;
  else  return n * factorial(n-1);
}
```

factorial (n) = n*n-1*n-2* … *1

$$
\begin{aligned}
&T(n)\\
&= T(n-1) + d\\
&= T(n-2) + d + d\\
&= T(n-3) + d + d + d\\
&= \ldots.\\
&= T(1) + (n-1) * d\\
&= c + (n-1) * d\\
&= O(n)
\end{aligned}
$$



Example: Factorial (cont.)

```
int factorial1(int n) {
   if (n<=1)  return 1;
   else {
     fact = 1;
     for (k=2;k<=n;k++)
         fact *= k;
     return fact;
   }
}
```

$O(1)$

$O(n)$

$O(1)$

- Both algorithms are O(n).