130a: Algorithm Analysis

- Foundations of Algorithm Analysis and Data Structures.
- Analysis:
  - How to predict an algorithm's performance
  - How well an algorithm scales up
  - How to compare different algorithms for a problem
- Data Structures
  - How to efficiently store, access, manage data
  - Data structures effect algorithm's performance

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;
   }
  }
  ```

Examples of famous algorithms

- Constructions of Euclid
- Newton's root finding
- Fast Fourier Transform
- Compression (Huffman, Lempel-Ziv, GIF, MPEG)
- DES, 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)

- Amazon hired first *Chief Algorithms Officer* (Udi Manber)

A real-world Problem

- Communication in the Internet
- Message (email, ftp) broken down into IP packets.
- Sender/receiver identified by IP address.
- The packets are routed through the Internet by special computers called Routers.
- Each packet is stamped with its destination address, but not the route.
- Because the Internet topology and network load is constantly changing, routers must discover routes dynamically.
- What should the Routing Table look like?

IP Prefixes and Routing

- Each router is really a switch: it receives packets at several input ports, and appropriately sends them out to output ports.
- Thus, for each packet, the router needs to transfer the packet to that output port that gets it closer to its destination.
- Should each router keep a table: IP address x Output Port?
- How big is this table?
- When a link or router fails, how much information would need to be modified?
- A router typically forwards several million packets/sec!

Data Structures

- The IP packet forwarding is a Data Structure problem!
- Efficiency, scalability is very important.

- Similarly, how does Google find the documents matching your query so fast?
- Uses sophisticated algorithms to create index structures, which are just data structures.
- Algorithms and data structures are ubiquitous.
- With the data glut created by the new technologies, the need to organize, search, and update MASSIVE amounts of information FAST is more severe than ever before.

Algorithms to Process these Data

- Which are the top K sellers?
- Correlation between time spent at a web site and purchase amount?
- Which flows at a router account for > 1% traffic?
- Did source S send a packet in last s seconds?
- Send an alarm if any international arrival matches a profile in the database
- Similarity matches against genome databases
- Etc.

Max Subsequence Problem

- Given a sequence of integers A1, A2, …, An, find the maximum possible sum value of a subsequence Ai, …, Aj.
- Numbers can be negative.
- You want a contiguous chunk with largest sum.

- Example: -2, 11, -4, 13, -5, -2
- The answer is 20 (subseq. A2 through A4).

- We will discuss 4 different algorithms, with time complexities $O(n^3)$, $O(n^2)$, $O(n \log n)$, and $O(n)$.
- With $n = 10^6$, algorithm 1 may take > 10 years; algorithm 4 will take a fraction of a second!

Algorithm 1 for Max Subsequence Sum

- Given $A_1,\dots,A_n$, find the maximum value of $A_i+A_{i+1}+\cdots+A_j$
  0 if the max value is negative

```
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;
```

$O(1)$ $O(1)$ $O(1)$ $O(1)$

$O(j-i)$

$O(\sum_{j=i}^{n-1}(j-i))$

$O(\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}(j-i))$

- Time complexity: $O(n^3)$

Algorithm 2

- Idea: Given sum from i to j-1, we can compute the sum from i to j in constant time.
- This eliminates one nested loop, and reduces the running time to $O(n^2)$.

```
into maxSum = 0;

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

Algorithm 3

- This algorithm uses divide-and-conquer paradigm.
- Suppose we split the input sequence at midpoint.
- The max subsequence is entirely in the left half, entirely in the right half, or it straddles the midpoint.
- Example:

  left half | right half

  4 -3 5 -2 | -1 2 6 -2

- Max in left is 6 (A1 through A3); max in right is 8 (A6 through A7). But straddling max is 11 (A1 thru A7).

Algorithm 3 (cont.)

- Example:

  left half | right half

  4 -3 5 -2 | -1 2 6 -2

- Max subsequences in each half found by recursion.
- How do we find the straddling max subsequence?
- Key Observation:
  - Left half of the straddling sequence is the max subsequence ending with -2.
  - Right half is the max subsequence beginning with -1.

- A linear scan lets us compute these in O(n) time.

Algorithm 3: Analysis

- The divide and conquer is best analyzed through recurrence:

$$T(1) = 1$$
$$T(n) = 2T(n/2) + O(n)$$

- This recurrence solves to $T(n) = O(n \log n)$.

Algorithm 4

2, 3, -2, 1, -5, 4, 1, -3, 4, -1, 2

```
    int maxSum = 0, thisSum = 0;

    for( int j = 0; j < a.size( ); j++ )
    {
        thisSum += a[ j ];

        if ( thisSum > maxSum )
            maxSum = thisSum;
        else if ( thisSum < 0 )
            thisSum = 0;
    }
    return maxSum;
}
```

- Time complexity clearly $O(n)$
- But why does it work? I.e. proof of correctness.

Proof of Correctness

- Max subsequence cannot start or end at a negative Ai.
- More generally, the max subsequence cannot have a prefix with a negative sum.

  Ex: -2 11 -4 13 -5 -2
- Thus, if we ever find that Ai through Aj sums to < 0, then we can advance i to j+1
  - Proof. Suppose j is the first index after i when the sum becomes < 0
  - The max subsequence cannot start at any p between i and j. Because A_i through A_{p-1} is positive, so starting at i would have been even better.

Algorithm 4

```
int maxSum = 0, thisSum = 0;

for( int j = 0; j < a.size( ); j++ )
{
        thisSum += a[ j ];

        if ( thisSum > maxSum )
                maxSum = thisSum;
        else if ( thisSum < 0 )
                thisSum = 0;
}
return maxSum
```

- The algorithm resets whenever prefix is < 0. Otherwise, it forms new sums and updates maxSum in one pass.

Why Efficient Algorithms Matter

- Suppose $N = 10^6$
- A PC can read/process N records in 1 sec.
- But if some algorithm does N*N computation, then it takes 1M seconds = 11 days!!!

- 100 City Traveling Salesman Problem.
  - A supercomputer checking 100 billion tours/sec still requires 10^{100} years!

- Fast factoring algorithms can break encryption schemes. Algorithms research determines what is safe code length. (> 100 digits)

How to Measure Algorithm Performance

- What metric should be used to judge algorithms?
  - Length of the program (lines of code)
  - Ease of programming (bugs, maintenance)
  - Memory required
  - Running time

- **Running time is the dominant standard.**
  - Quantifiable and easy to compare
  - Often the critical bottleneck

Abstraction

- An algorithm may run differently depending on:
  - the hardware platform (PC, Cray, Sun)
  - the programming language (C, Java, C++)
  - the programmer (you, me, Bill Joy)

- While different in detail, all hardware and prog models are equivalent in some sense: **Turing machines**.

- It suffices to count basic operations.

- Crude but valuable measure of algorithm's performance *as a function of input size.*

Average, Best, and Worst-Case

- On which input instances should the algorithm's performance be judged?

- Average case:
  - Real world distributions difficult to predict
- Best case:
  - Seems unrealistic
- Worst case:
  - Gives an absolute guarantee
  - **We will use the worst-case measure.**

Examples

- Vector addition **Z = A+B**

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

  ```
      Z[i] = A[i] + B[i];
  ```

  $T(n) = c\,n$

- Vector (inner) multiplication **z =A*B**

  z = 0;

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

  ```
      z = z + A[i]*B[i];
  ```

  $T(n) = c' + c_1\,n$

Examples

- Vector (outer) multiplication **$Z = A*B^T$**

```
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        Z[i,j] = A[i] * B[j];
```

$T(n) = c_2 n^2;$

- A program does all the above

$T(n) = c_0 + c_1 n + c_2 n^2;$

Simplifying the Bound

- $T(n) = c_k n^k + c_{k-1} n^{k-1} + c_{k-2} n^{k-2} + \ldots + c_1 n + c_o$
  - too complicated
  - too many terms
  - Difficult to compare two expressions, each with 10 or 20 terms
- Do we really need that many terms?

Simplifications

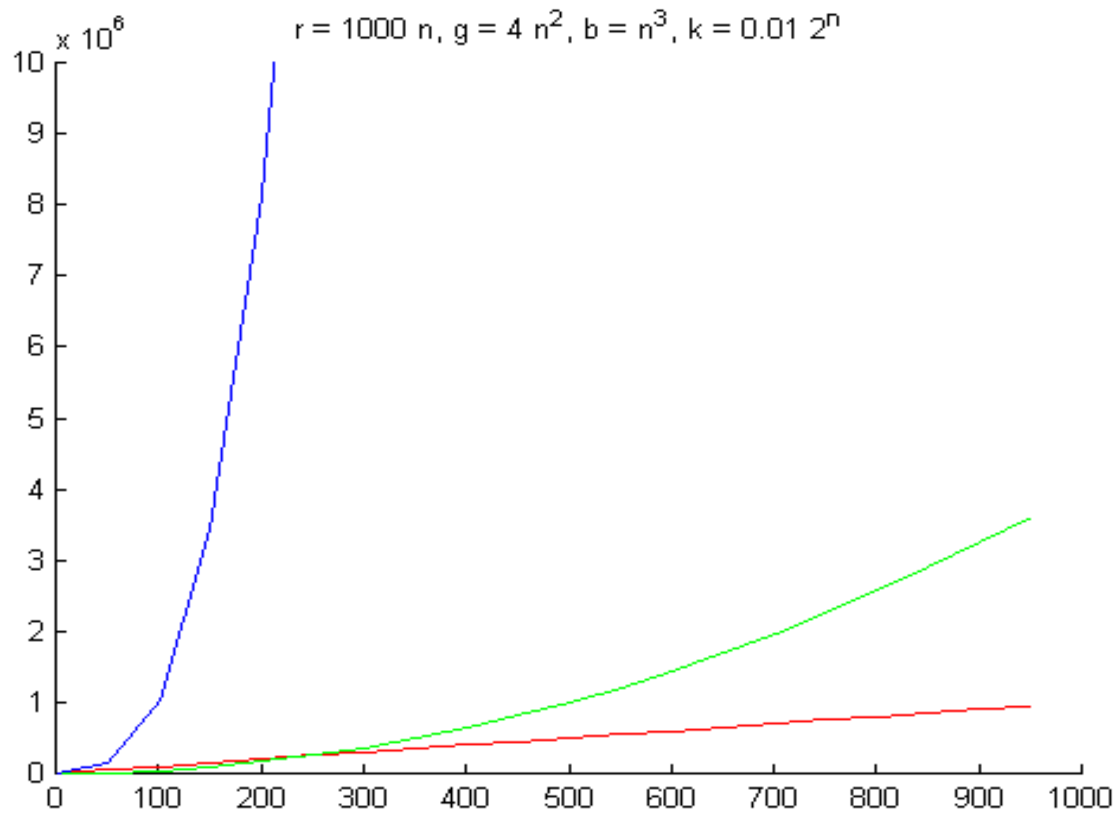
- Keep just one term!
  - the fastest growing term (dominates the runtime)
- No constant coefficients are kept
  - Constant coefficients affected by machines, languages, etc.

- **Asymtotic behavior** (as *n* gets large) is determined entirely by the **leading** term.

  - Example. $T(n) = 10\,n^3 + n^2 + 40n + 800$
    - *If n = 1,000, then T(n) = 10,001,040,800*
    - error is 0.01% if we drop all but the n^3 term
  - In an assembly line the slowest worker determines the throughput rate

Simplification

- Drop the constant coefficient
  - Does not effect the relative order



Simplification

- The *faster* growing term (such as 2^n) *eventually* will outgrow the slower growing terms (e.g., 1000 n) no matter what their coefficients!

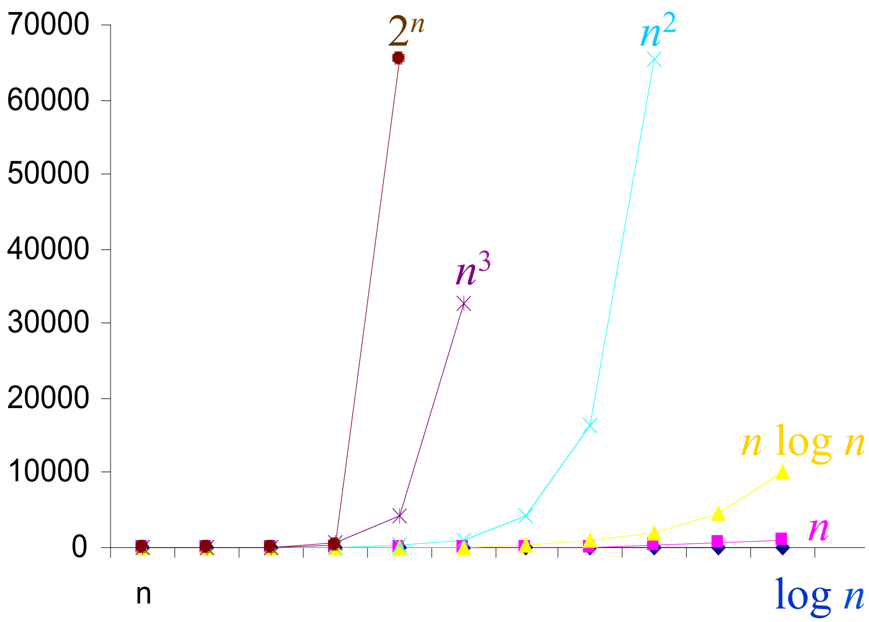
- Put another way, given a certain increase in allocated time, a higher order algorithm will not reap the benefit by solving much larger problem

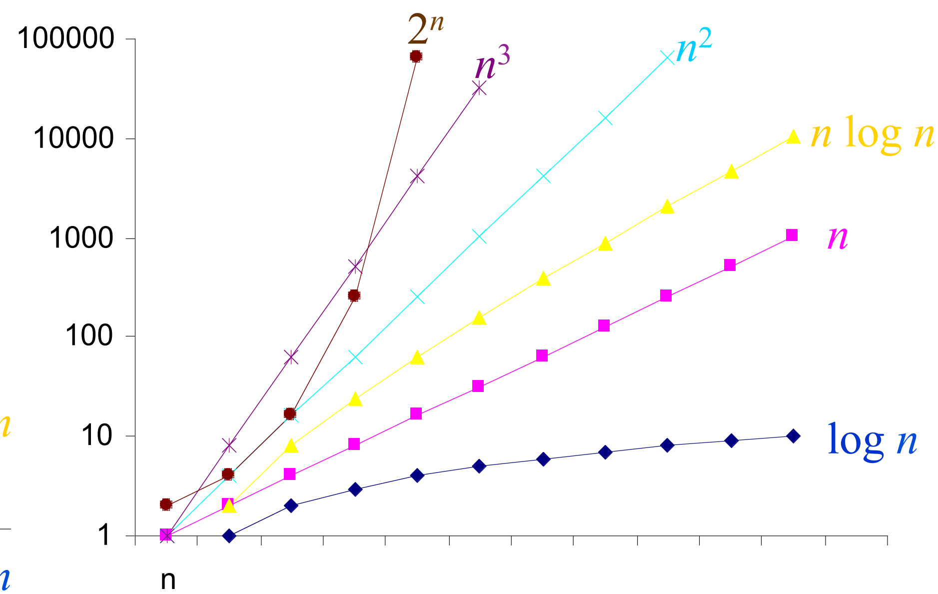
Complexity and Tractability

	$T(n)$						
n	n	$n \log n$	n^2	n^3	n^4	n^{10}	2^n
10	.01µs	.03µs	.1µs	1µs	10µs	10s	1µs
20	.02µs	.09µs	.4µs	8µs	160µs	2.84h	1ms
30	.03µs	.15µs	.9µs	27µs	810µs	6.83d	1s
40	.04µs	.21µs	1.6µs	64µs	2.56ms	121d	18m
50	.05µs	.28µs	2.5µs	125µs	6.25ms	3.1y	13d
100	.1µs	.66µs	10µs	1ms	100ms	3171y	4×10^{13}y
10^3	1µs	9.96µs	1ms	1s	16.67m	3.17×10^{13}y	32×10^{283}y
10^4	10µs	130µs	100ms	16.67m	115.7d	3.17×10^{23}y	
10^5	100µs	1.66ms	10s	11.57d	3171y	3.17×10^{33}y	
10^6	1ms	19.92ms	16.67m	31.71y	3.17×10^{7}y	3.17×10^{43}y	

Assume the computer does 1 billion ops per sec.

log *nnn* log *nn*^{2}*n*^{3} 01011 12248 2481664 382464512





Another View

- More resources (time and/or processing power) translate into large problems solved if complexity is low

T(n)	Problem size solved in 10^3 sec	Problem size solved in 10^4 sec	Increase in Problem size
100n	10	100	10
1000n	1	10	10
$5n^2$	14	45	3.2
N^3	10	22	2.2
2^n	10	13	1.3

Asympotics

T(n)	keep one	drop coef
$3n^2+4n+1$	$3\ n^2$	n^2
$101\ n^2+102$	$101\ n^2$	n^2
$15\ n^2+6n$	$15\ n^2$	n^2
$a\ n^2+bn+c$	$a\ n^2$	n^2

- They all have the same "growth" rate

Caveats

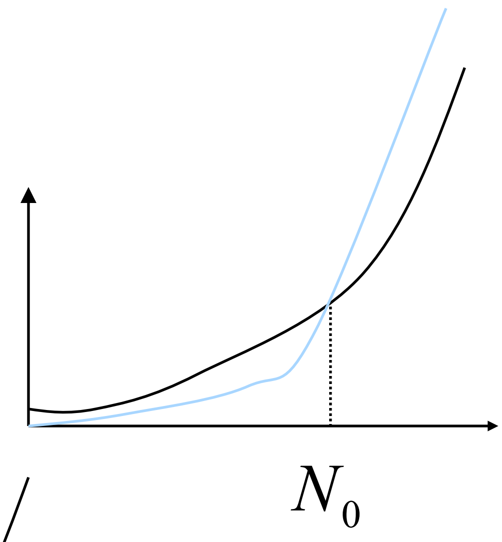
- Follow the spirit, not the letter
  - a 100n algorithm is more expensive than n^2 algorithm when $n < 100$
- Other considerations:
  - a program used only a few times
  - a program run on small data sets
  - ease of coding, porting, maintenance
  - memory requirements

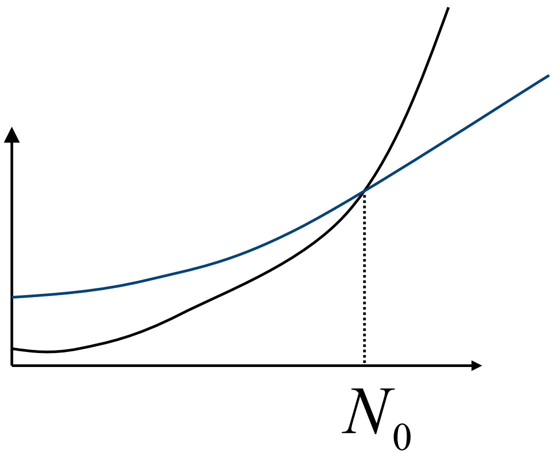
Asymptotic Notations

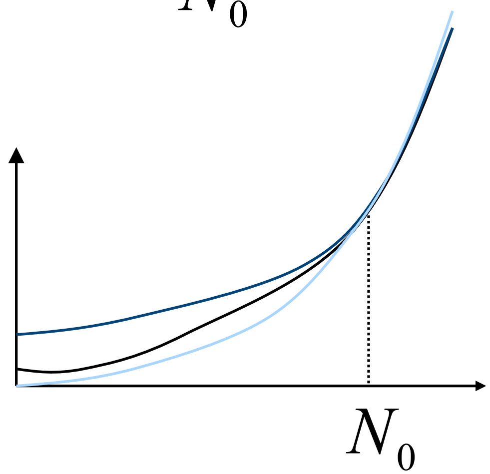
- Big-O, "bounded above by": $T(n) = O(f(n))$
  - For some c and N, $T(n) \leq c \cdot f(n)$ whenever $n > N$.
- Big-Omega, "bounded below by": $T(n) = \Omega(f(n))$
  - For some $c>0$ and N, $T(n) \geq c \cdot f(n)$ whenever $n > N$.
  - Same as $f(n) = O(T(n))$.
- Big-Theta, "bounded above and below": $T(n) = \theta(f(n))$
  - $T(n) = O(f(n))$ and also $T(n) = \Omega(f(n))$
- Little-o, "strictly bounded above": $T(n) = o(f(n))$
  - For some c and N, $T(n) < c \cdot f(n)$ whenever $n > N$
  - $T(n)=O(f(n))$ and $T(n) \neq \theta(f(n))$

By Pictures

- Big-Oh (most commonly used)
  - bounded above
- Big-Omega
  - bounded below
- Big-Theta
  - exactly
- Small-o
  - not as expensive as ...







Example

$$T(n) = n^3 + 2n^2$$

$O(?)$	$\Omega(?)$
∞	0
n^{10}	n
n^5	n^2
n^3	n^3

Examples

$f(n)$	Asymptomic
c	$\Theta(1)$
$\Sigma_{i=1}^{k} c_i n^i$	$\Theta(n^k)$
$\Sigma_{i=1}^{n} i$	$\Theta(n^2)$
$\Sigma_{i=1}^{n} i^2$	$\Theta(n^3)$
$\Sigma_{i=1}^{n} i^k$	$\Theta(n^{k+1})$
$\Sigma_{i=0}^{n} r^i$	$\Theta(r^n)$
$n!$	$\Theta(n(n/e)^n)$
$\Sigma_{i=1}^{n} 1/i$	$\Theta(\log n)$

Summary (Why O(n)?)

- $T(n) = c_k n^k + c_{k-1} n^{k-1} + c_{k-2} n^{k-2} + \ldots + c_1 n + c_o$
- Too complicated
- $O(n^k)$
  - a single term with constant coefficient dropped
- Much simpler, extra terms and coefficients *do not matter* asymptotically
- Other criteria hard to quantify

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 (cont.)

- 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 (cont.)

if (cond) then $O(1)$

  body$_1$ $T_1(n)$

else

  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

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

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
  - Hanoi towers

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

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

n   * factorial(n-1)   T(n)

n-1   * factorial(n-2)   T(n-1)

n-2   * factorial(n-3)   T(n-2)

…

2   *factorial(1)

T(1)

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).

Example: Hanoi Towers

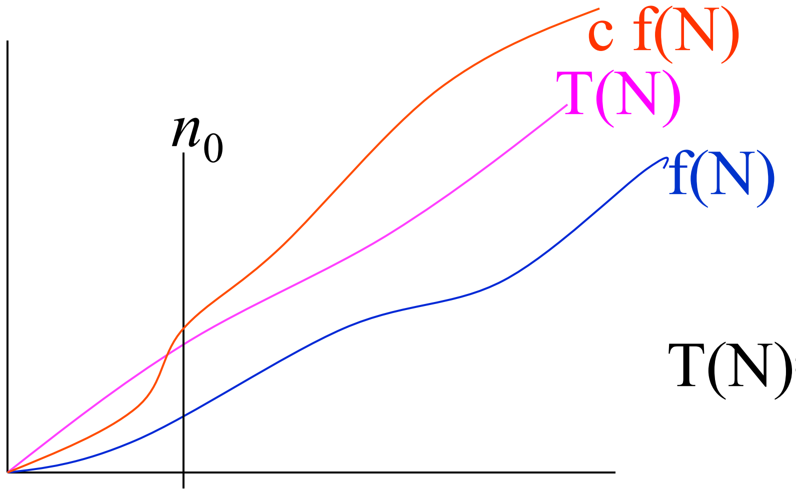
- Hanoi(n,A,B,C) =
- Hanoi(n-1,A,C,B)+Hanoi(1,A,B,C)+Hanoi(n-1,C,B,A)

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

Worst Case, Best Case, and Average Case

```
template<class T>
void SelectionSort(T a[], int n)
{   // Early-terminating version of selection sort
    bool sorted = false;
    for (int size=n; !sorted && (size>1); size--) {
        int pos = 0;
        sorted = true;
        // find largest
        for (int i = 1; i < size; i++)
            if (a[pos] <= a[i]) pos = i;
            else sorted = false; // out of order
        Swap(a[pos], a[size - 1]);
        }
}
```

- Worst Case
- Best Case



$T(N)=O(f(N))$

- T(N)=6N+4 : n0=4 and c=7, f(N)=N
- T(N)=6N+4 <= c f(N) = 7N for N>=4
- 7N+4 = *O*(N)
- 15N+20 = *O*(N)
- N^2=*O*(N)?
- N log N = *O*(N)?
- N log N = *O*(N^2)?
- N^2 = *O*(N log N)?
- N^{10} = *O*(2^N)?
- 6N + 4 = W(N) ? 7N? N+4 ? N^2? N log N?
- N log N = W(N^2)?
- 3 = *O*(1)
- 1000000=*O*(1)
- Sum i = *O*(N)?

An Analogy: Cooking Recipes

- Algorithms are detailed and precise instructions.
- Example: bake a chocolate mousse cake.
  - Convert raw ingredients into processed output.
  - Hardware (PC, supercomputer vs. oven, stove)
  - Pots, pans, pantry are data structures.
- Interplay of hardware and algorithms
  - Different recipes for oven, stove, microwave etc.
- New advances.
  - New models: clusters, Internet, workstations
  - Microwave cooking, 5-minute recipes, refrigeration