Brute Force

Analysing a recurrence relation -> an equation or inequality that describes a function in terms of its value on smaller inputs

Some brute-force solutions:

- String matching
- Closest pair
- Convex hull
- Hamiltonian path
- Job assignment
- Knapsack problem

Backwards Substitution:

1. Express x(n-1) successively as a function of x(n-2), x(n-3)
2. Derive x(n-j) as a function of j
3. Substitute n-j = base condition

The above equation can be solved by backward substitution:

```
M(n) = M(n-1)+1
Substitute M(n-1) = M(n-2) + 1
-> M(n) = [M(n-2) + 1]+1 = M(n-2) + 2
Substitute M(n-2) = M(n-3) + 1
    M(n) = [M(n-3) + 1] + 2 = M(n-3) + 3
-> Pattern: M(n) = M(n-j) + j
Ultimately: M(n) = M(n-n)+n = M(0) + n = n
```

Example

```
int Mystery(int n such that n > 0):
    if n == 1:
        return 1
    else:
        return 1 + Mystery(n/2)
```

It cuts the search space in half:

```
int countBits(int n such that n > 0):
    if n == 1:
        return 1
    else:
        return 1 + countBits(n/2)
```

This is the addition of 1 on each call to countBits.

$A(1) = 0$ the addition doesn't take place when $n = 1$

$A(n) = A(n/2) + 1$ for $n > 1$

Now let $n = 2^k$ which is the same as saying $k = \log_2 n$

$n/2 = 1/2 * 2^k = 2^{-1} * 2^k = 2^{k-1}$

$A(1) = A(2^0) = 0$

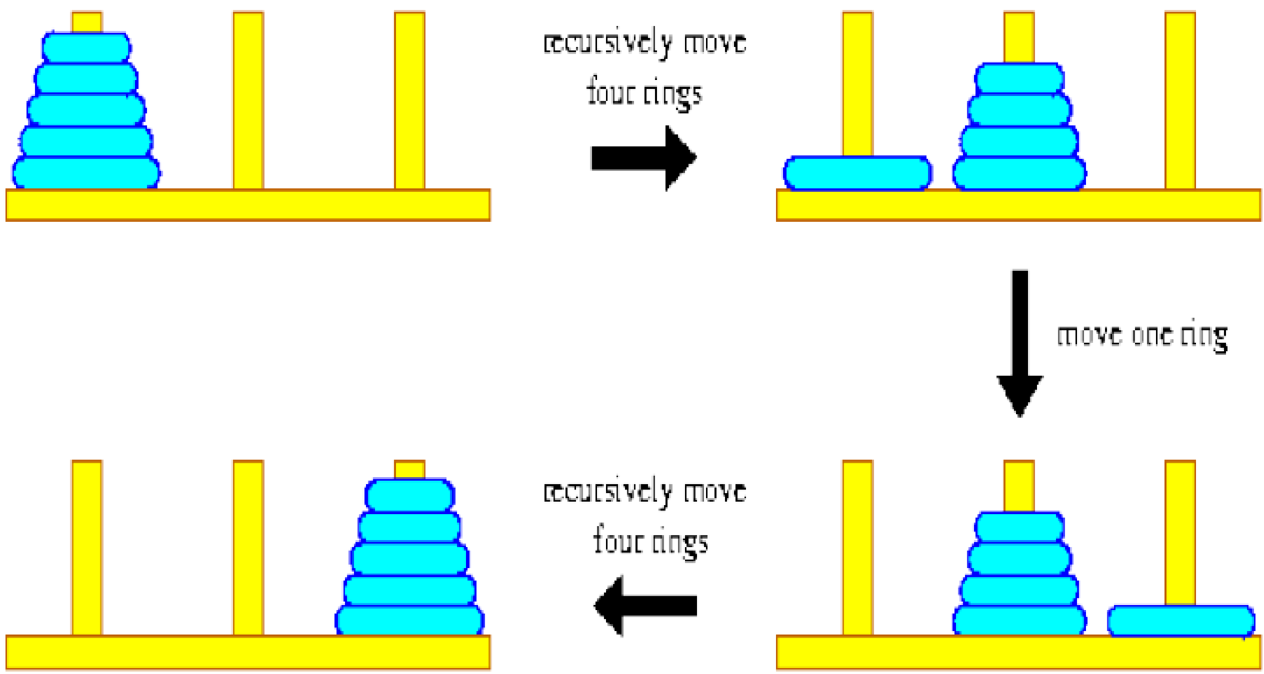
$A(n) = A(2^k) = A(2^{k-1}) + 1$ for $k > 0$

$= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2$

$= A(2^{k-2}) + k = A(2^0) + k = k = \log_2 n \in \Theta(\log n)$

Tower of Hanoi

This will be a walkthrough of the Towers of Hanoi in order to better understand a recurrence relation.




```
void hanoi(int n, int source, int spare, int dest){
    if(n>0){
        hanoi(n-1, source, det, spare);
        cout << "Move disk from " << source << " to " << dest << endl;
        hanoi(n-1,spare,source,dest);
    }
}
```

$P(1) = 1, P(N) = P(N-1) + 1 + P(N-1)$

$P(1) = 1$. Obviously.

$P(N) = P(N-1) + 1 + P(N-1)$

$P(N) = 2P(N-1) + 1$ (eq a)

substitute: $P(N-1) = 2P(N-2)+1$ in (eq a)

$P(N) = 2[\,2\,P(N-2) + 1\,] + 1$

$P(N) = 2^2\,[P(N-2)\,] + 2 + 1$

$P(N) = 2^2[2\,P(N-3) + 1] + 2 + 1$

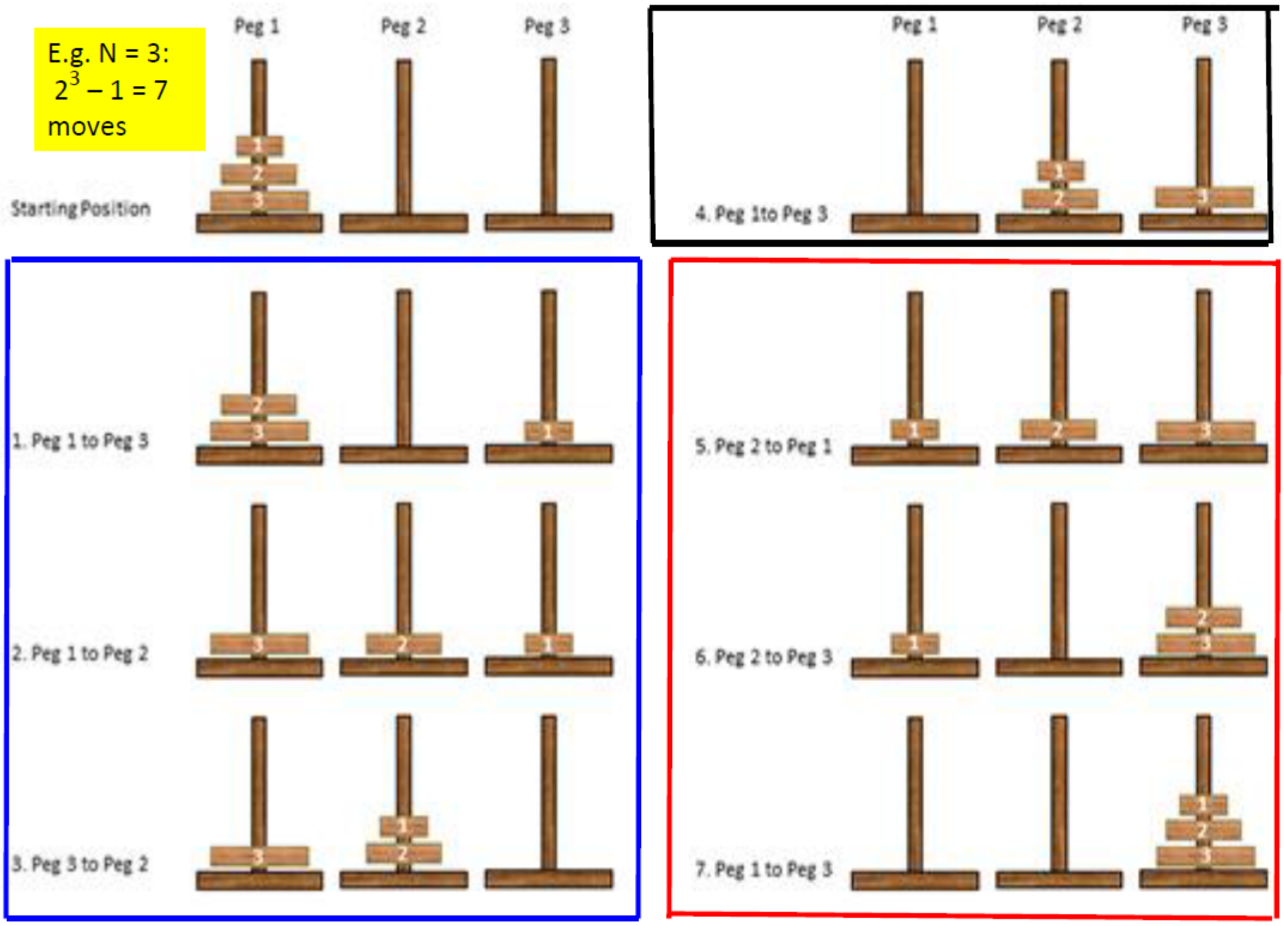
$P(N) = 2^3[P(N-3)] + 2^2 + 2^1 + 1$ etc.

$P(N) = 2^k\,P(N-k) + \boldsymbol{[2^{k-1} + 2^{k-2} + .. + 2 + 1]}$

The italicised part in line above $= 2^k - 1$

Set $k=N-1$: $P(N)= 2^{N-1}P(1) + [2^{N-1}-1] = 2^N-1$

Visual representation:



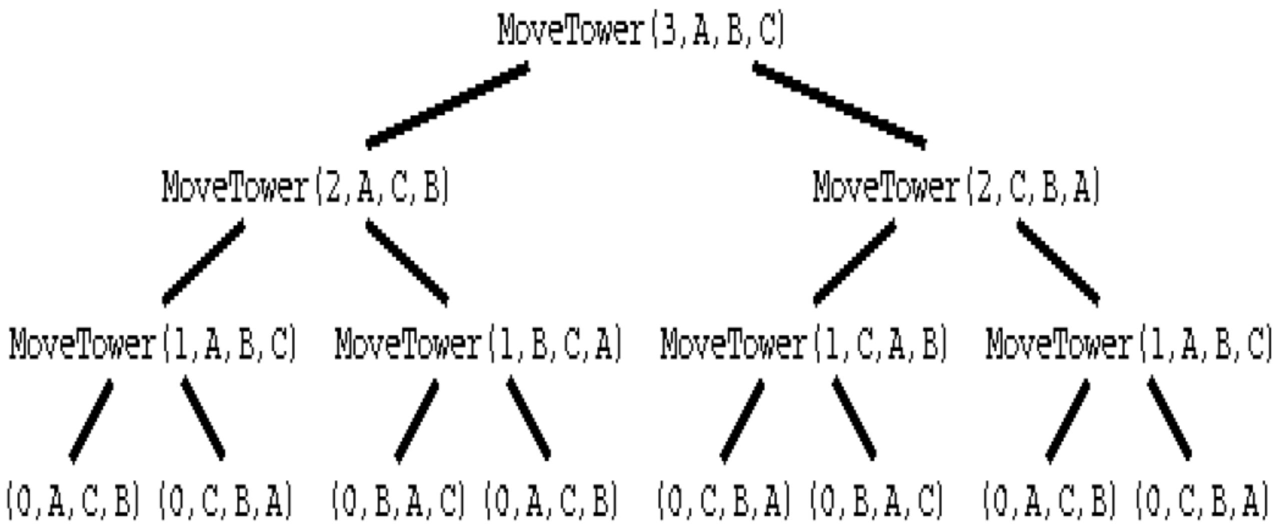
Output:

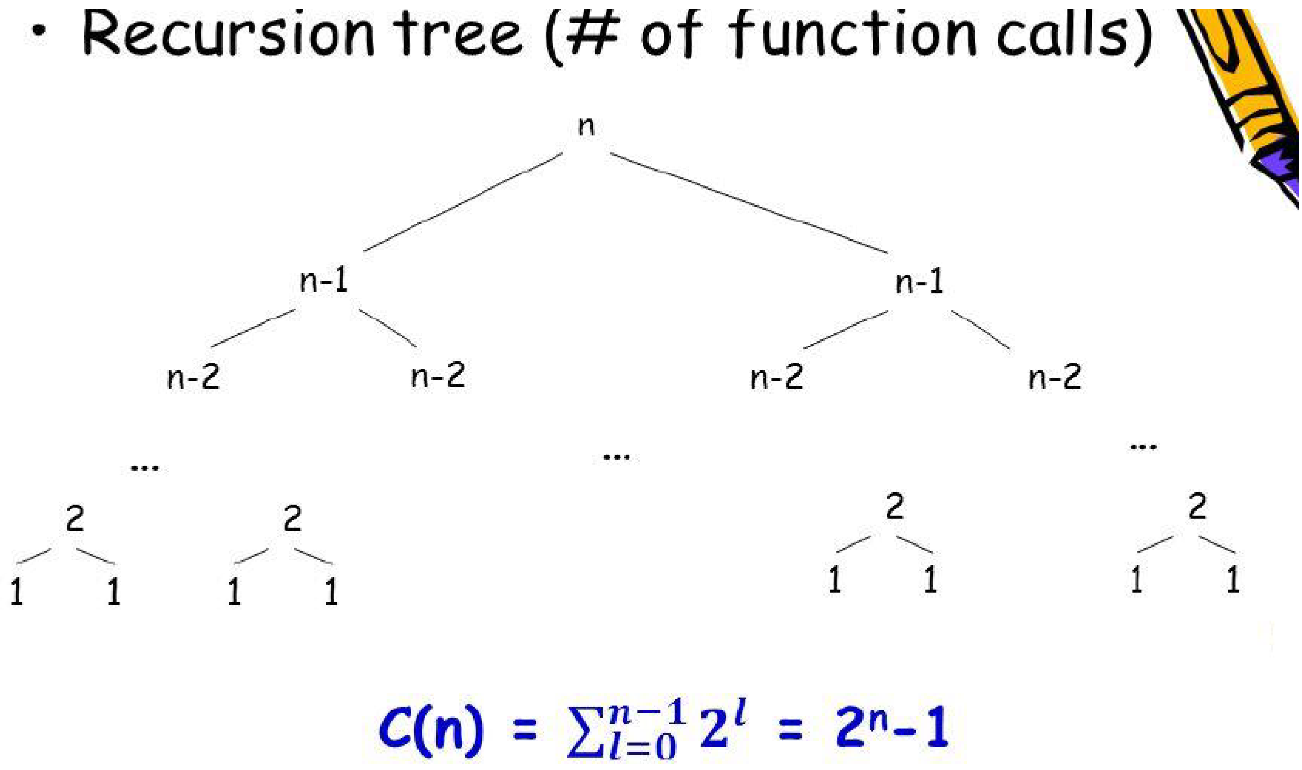
Start here.

Disk 1 from 1 to 3
Disk 2 from 1 to 2
Disk 1 from 3 to 2
Disk 3 from 1 to 3
Disk 1 from 2 to 1
Disk 2 from 2 to 3
Disk 1 from 1 to 3
Disk 4 from 1 to 2
Disk 1 from 3 to 2
Disk 2 from 3 to 1
Disk 1 from 2 to 1
Disk 3 from 3 to 2
Disk 1 from 1 to 3
Disk 2 from 1 to 2
Disk 1 from 3 to 2

Cont in next col...

Disk 5 from 1 to 3
Disk 1 from 2 to 1
Disk 2 from 2 to 3
Disk 1 from 1 to 3
Disk 3 from 2 to 1
Disk 1 from 3 to 2
Disk 2 from 3 to 1
Disk 1 from 2 to 1
Disk 4 from 2 to 3
Disk 1 from 1 to 3
Disk 2 from 1 to 2
Disk 1 from 3 to 2
Disk 3 from 1 to 3
Disk 1 from 2 to 1
Disk 2 from 2 to 3
Disk 1 from 1 to 3





Recursive Fibonacci

```
F(n):
    if n <=1:
        return n
    else
        return F(n-1) + F(n-2)
```

- Basic operation is the addition
- Recurrence relation:
    - A(n) = A(n-1) + 1 + A(n-2)
    - A(n) ∈ $\Theta(1.61803^n)$

Brute Force

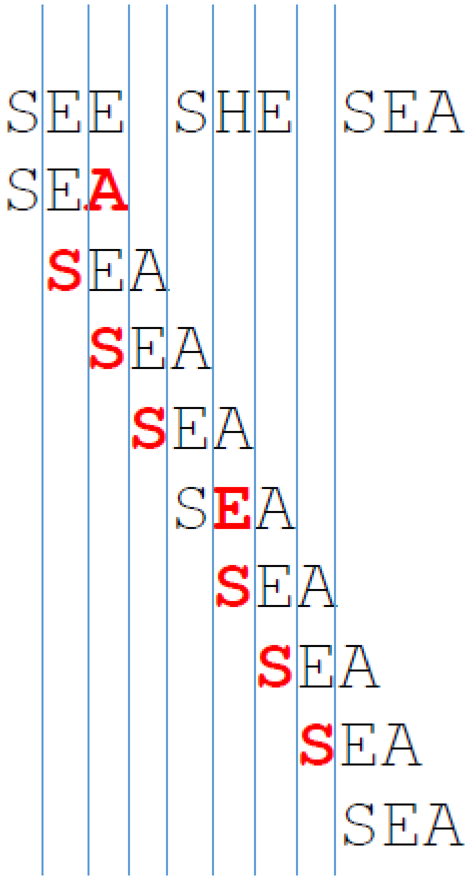
A straightforward approach usually directly based on problem statement and definitions

- Crude but often effective
- Simple
- Widely Applicable
- Sometimes impractically slow
- Try all the possibilities until problem solved
- Loop through each possibility, check if it solves problem

String Search

Brute force pattern match

SEE SHE SEA and we are searching for *SEA*




```
BruteForceStringMatch(T[0...n-1],P[0..m-1]):
//T is the text; P is the pattern we're searching for in the text

for k <- 0 to n-m do {//for reach char in T
    j <- 0
    while j < m and P[j] = T[i+j] do{
        j <- j + 1
        if j = m{
            return k;
        }
    }
}
return -1;
```

My python version of this is here

Worst Case Brute Force

- Worst case: the search string matches every character except the last, for every iteration of the outer loop.
    - E.g.: text = "aaaaaaaaaaaaaaaaaa"
    - Search string = "aaaab"
- Let m = length of search string, n = length of text

- =m(n-m+1) character comparisons
    - Θ(mn) for m much smaller than n (which is what happens in practice)
- Worst case very unlikely with natural language!
- Average case on natural language?

String Matching

- Problem
    - Find a substring in some text that matches a pattern
    - Pattern: a string of m characters to search for
    - Text: a (long) string of n characters to search in

1. Align pattern at beginning of text
2. Moving left to right, compare each character of pattern to the corresponding character in text UNTIL
    - All characters are found to match (successful search):
    - A mismatch detected
3. WHILE pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat step 2

Closest Pair

- Problem
    - Find the two ints that are closest together in a set of 2-D points $P_1 = (x_1,y_1)$,..,$P_n = (X_n,Y_n)$
- Algorithm

```
dmin <- infinity
for i <- 1 to n-1 do
    for j <- i+1 to n do
        d <- sqrt((xi-xk)**2+(yi-yk)**2)
        if d < dmin
            dmin <- d;
            index1 <- i;
            index2 <- j;
return index1, index2
```

- Efficiency: $\Theta(n^2)$

My code for this is here

Convex Hull Problem

- Problem
    - Find the convex hull enclosing n 2-D points
    - Convex Hull: If S is a set of points then the Convex Hull of S is the smallest convex set containing S

- Convex Set: A set of points in the plane is convex if for any two points P and Q, the line segment joining P and Q belongs to the set

Brute Force

- Algorithm
    - For each pair of points p_1 and p_2
    - Determine whether all other points lie to the same side of the straight line through p_1 and p_2
- Efficiency
    - Efficiency: $\Theta(n^3)$
- Strengths

Pros and Cons of Brute Force

- Strengths
    - Wide applicability
    - Simplicity
    - Yields reasonable algorithm for some important problems and standard algorithms for simple computational tasks
    - A good yardstick for better algorithms
    - Sometimes doing better is not worth the bother
- Weakness
    - Rarely produces efficient algorithms
    - Some brute force algorithms are infeasibly slow
    - Note as creative as some other design techniques

Exhaustive Search

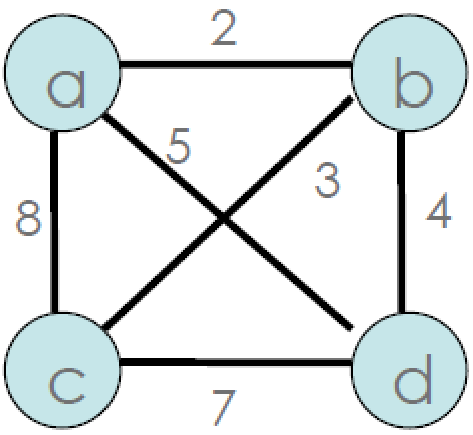
- Definition
    - A brute force solution to the search for an element with a special property
    - Usually among combinatorial objects such a permutations or subsets
    - Suggests generating each and every element of the problem's domain
- Method
    1. Construct a way of listing all potential solutions to the problem in a systematic manner
    2. Evaluate all Solutions one by one (disqualifying infeasible ones) keeping track of the best one found so far
    3. When search ends, announce the winner

Travelling Salesman Problem

- Problem
    - Given n cities with known distances between each pair
    - Find the shortest tour that passes through all the cities exactly once before returning to the starting city
- Alternatively

- Find shortest *Hamiltonian Circuit* in a weighted connected graph

- Example:



Exhaustive Search Implementation

Tour	Cost
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17

- Improvements
  - Start and end at one particular city
  - Remove tours that differ only in direction
- Efficiency
  - `(n-1)!/2 = O(n!)`

Assignment Problem

- Assignment Problem
- n people and n jobs to be done
- Each person is assigned to do exactly one job
- Each job is assigned to exactly one person
- The cost of person i doing job j is C[i,j]
- Find a job assignment with the minimum cost

	Job 1	Job 2	Job 3	Job 4
Person	9	2	7	8
Person	6	4	3	7
Person	5	8	1	8
Person	7	6	9	4

	Job 1	Job 2	Job 3	Job 4
1-2-3-4	Person 1	Person 2	Person 3	Person 4
1-2-4-3	Person 1	Person 2	Person 4	Person 3
1-3-2-4	Person 1	Person 3	Person 2	Person 4
1-2-4-2	Person 1	Person 3	Person 4	Person 2

Solution

- Generate all permutations of n positive integers
- Compute the total cost for that assignment
- Retain the cheapest assignment
- Very inefficient

Knapsack Problem

- **Problem** Given n items
    - Weights: w_1, w_2 ... w_n
    - values: v_1, v_2 .... v_n
    - A knapsack of capacity W
    - Find the most valuable subset of the items that fit into the knapsack
- Example W = 16

Item	Weight	Value
1	2kg	R200
2	5kg	R300
3	10kg	R500
4	5kg	R100

Exhaustive Search Knapsack

Subset	Total Weight	Total Value
1	2kg	R200
2	5kg	R300
3	10kg	R500

Subset	Total Weight	Total Value
4	5kg	R100
1,2	7kg	R500
1,3	12kg	R700
1,4	7kg	R300
2,3	15kg	R800
2,4	10kg	R600
3,4	10kg	R400
1,2,3	17kg	n/a
1,2,4	12kg	R600
1,3,4	17kg	n/a
2,3,4	20kg	n/a
1,2,3,4	22kg	m/a

Efficiency $\Omega(2^n)$

Comments on Exhaustive Search

- Exhaustive search algorithms run in a realistic amount of time **only on very small instances**
- In many cases there are much better alternatives!
- In some cases exhaustive search (or variation) is the only known solution
- and parallel solutions can speed it up

Summary

- Convex hull & Closest pair:
    - All possibilities iterated with nested loops
- Travelling salesman & Job assignment:
    - All possibilities are all possible permutations
- Knapsack problem
    - All possibilities are all the subsets (combinations) of the choices