Brute Force Algorithms
CS 351, Chapter 3

For most of the algorithms portion of the class we'll focus on specific design strategies to solve problems. One of the simplest is brute force, which can be defined as:

> Brute force is a straightforward approach to solving a problem, usually directly based on the problem's statement and definitions of the concepts involved. Generally it involved iterating through all possible solutions until a valid one is found.

Although it may sound unintelligent, in many cases brute force is the best way to go, as we can rely on the computer's speed to solve the problem for us. Brute force algorithms also present a nice baseline for us to compare our more complex algorithms to.

As a simple example, consider searching through a sorted list of items for some target. Brute force would simply start at the first item, see if it is the target, and if not sequentially move to the next until we either find the target or hit the end of the list. For small lists this is no problem (and would actually be the preferred solution), but for extremely large lists we could use more efficient techniques.

Skipping from text: Selection Sort (already covered), Bubble Sort

Brute Force String Matching

The string matching problem is to find if a pattern P[1..m] occurs within text T[1..n]. Later on we will examine how to compute approximate string matching using dynamic programming. In this case we will examine how to perform exact string matching, and later we will see more efficient methods than the brute force approach. Note that string matching is useful in more cases than just searching for words in text. String matching also applies to other problems, for example, matching DNA patterns in the human genome.

Find all valid shifts of P in T using a loop:

```
Brute-Force-Match(T,P)        // T = length 1-n        P = length 1-m
        n ← length[T]
        m ← length[P]
        for s ← 0 to n-m
                do if P[1..m]=T[s+1,s+m]  then P matches T at index s+1
```

Example:

```
T       =       ILOVEALGORITHMS
P       =               ALGOR
```

O(n+m) in this case, not much duplication of P in T

```
T    =    AAAAAAAAAAAAAA
P    =    AAAAAB
           AAAAAB
                etc.
```

O(mn) in this case, for each of the n characters we have to go through all m chars of P

The brute force or naïve string matcher is slow in some cases, although in many cases it actually works pretty good and should not be ignored, especially since it is easy to implement. For small *n* or cases when the text and pattern differ, this is one of the best methods to use.

Closest-Pair and Convex-Hull Problems

In computational geometry, two well-known problems are to find the closest pair of points and the convex hull of a set of points.

The closest-pair problem, in 2D space, is to find the closest pair of points given a set of n points. Given a list P of n points, $P_1=(x_1,y_1)$, … $P_n=(x_n,y_n)$ we simply do the following:

```
BruteForceClosest(P)
    min ← ∞
    for i = 1 to n-1
        for j = i+1 to n do
            d ← distance(Pi,Pj)    // Use sqrt(distances squared)
            if d < min then
                min ← d
                minPoints = (Pi,Pj)
```

The basic operation is computing the Euclidean distance between all pairs of points and requires $O(n^2)$ runtime. We could arrive at this value more formally by noting:

$$T(n)=\sum_{i=1}^{n-1}\left(\sum_{j=i+1}^{n}C\right)=C\sum_{i=1}^{n-1}\left(\sum_{j=i+1}^{n}1\right)=C\sum_{i=1}^{n-1}(n-i)=C(n-1)(n-i)=\theta(n^2)$$

This requires computing the square root of the sum of squares of the difference between the coordinates in the point. For a large number of points, computing the square root is a very expensive operation and can take a long time to run.

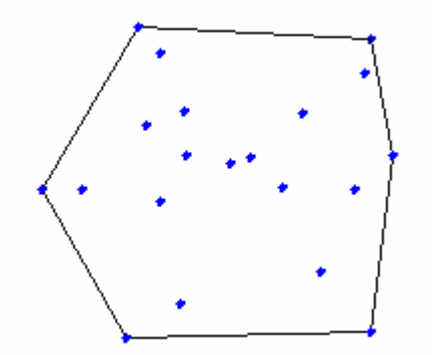
In fact, we don't even need to compute the square root – we can simply ignore the square root and compare the values $(x_i - x_j)^2 + (y_i - y_j)^2$ themselves, since this value is strictly increasing compared to the square root of the value. This results in the same runtime, but would significantly increase the execution speed.

The Convex Hull Problem

In this problem, we want to compute the convex hull of a set of points? What does this mean?

- Formally: It is the smallest convex set containing the points. A convex set is one in which if we connect any two points in the set, the line segment connecting these points must also be in the set.
- Informally: It is a rubber band wrapped around the "outside" points.

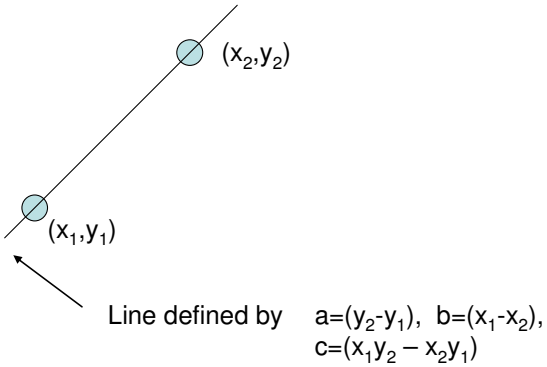
Here is a picture from:
http://www.cs.princeton.edu/~ah/alg_anim/version1/ConvexHull.html
It is an applet so you can play with it to see what a convex hull is if you like.



Theorem: The convex hull of any set S of n>2 points (not all collinear) is a convex polygon with the vertices at some of the points of S.

How could you write a brute-force algorithm to find the convex hull?

In addition to the theorem, also note that a line segment connecting two points P_1 and P_2 is a part of the convex hull's boundary if and only if all the other points in the set lie on the same side of the line drawn through these points. With a little geometry:



For all points above the line, $ax + by > c$, while for all points below the line, $ax + by < c$. Using these formulas, we can determine if two points are on the boundary to the convex hull.

High level pseudocode for the algorithm then becomes:

for each point P_i
    for each point P_j where $P_j \neq P_i$
        Compute the line segment for P_i and P_j
        for every other point P_k where $P_k \neq P_i$ and $P_k \neq P_j$
            If each P_k is on one side of the line segment, label P_i and P_j in the convex hull

What is the runtime for this algorithm?
We will see much faster ones later.

Exhaustive Search

Exhaustive search refers to brute force search for combinatorial problems. We essentially generate each element of the problem domain and see if it satisfies the solution.

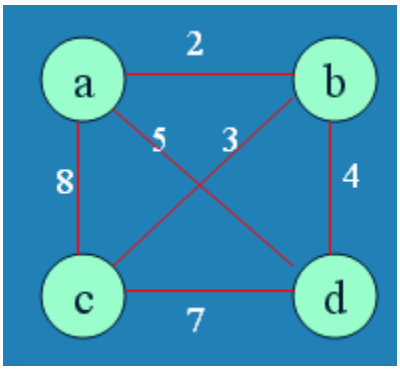
We do the following:

- Construct a way of listing all potential solutions to the problem in a systematic manner
    - all solutions are eventually listed
    - no solution is repeated
- Evaluate solutions one by one, perhaps disqualifying infeasible ones and keeping track of the best one found so far
- When search ends, announce the winner

Previously seen examples from our discussion of P/NP/NP Complete:

Traveling Salesman Problem

Find shortest Hamiltonian circuit in a weighted connected graph.



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

Efficiency?

Knapsack 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 (sum of weights ≤ W)

Example:

item	weight	value
1	2	$20
2	5	$30
3	10	$50
4	5	$10

Knapsack capacity W=16

Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	not feasible
{1,2,4}	12	$60
{1,3,4}	17	not feasible
{2,3,4}	20	not feasible
{1,2,3,4}	22	not feasible

Efficiency? Later we'll see a dynamic programming solution.

Exhaustive search algorithms run in a realistic amount of time only on very small instances.

In many cases there are much better alternatives! In general though we end up with an approximation to the optimal solution instead of the guaranteed optimal solution.

Euler circuits
shortest paths
minimum spanning tree
various AI techniques

In some cases exhaustive search (or variation) is the only known solution.

Brute Force strengths and weaknesses:

Strengths:

- wide applicability
- simplicity
- yields reasonable algorithms for some important problems
  - searching
  - string matching
  - matrix multiplication
- yields standard algorithms for simple computational tasks
  - sum/product of n numbers
  - finding max/min in a list

Weaknesses:

- rarely yields efficient algorithms
- some brute force algorithms unacceptably slow
- not as constructive/creative as some other design techniques