Brute Force

- Directly do the problem without worrying about costs
- Sometimes simple strategy (as in sorting below)
- Advantage:
  - Simple,
  - general,
  - usually good enough for small input size
- Disadvantage
  - Often takes too much time.

Brute Force

- Selection sort
- Bubble sort
- Sequential Search
- String Matching
- Closest Pair, Convex Hull
- Exhaustive Search

Selection Sort

- Numbers which are already in right position p
- Numbers which are yet to be sorted $s = n - p$
- Find the smallest of the s numbers, and put it in the right place.
- Initially, $p = 0$, $s = n$.

Selection Sort

Selection Sort
    Input: array $A[0 : n-1]$
    For $i \leftarrow 0$ to $n-2$ do
        $minloc \leftarrow i$
        For $j \leftarrow i+1$ to $n-1$ do
            If $A[j] < A[minloc]$, then $minloc \leftarrow j$
        EndFor
        (* Swap $A[i]$ and $A[minloc]$. *)
        $temp \leftarrow A[i]$, $A[i] = A[minloc]$, $A[minloc] = temp$.
    EndFor
End

$|$ 21 29 23 19

19 $|$ 29 23 21

19 21 $|$ 23 29

19 21 23 $|$ 29

19 21 23 29

Analysis

- Consider $C(n)$, number of comparisons.
- In iteration i, takes $n - 1 - i$ comparisons.
- $C(n) = \Sigma_{i=0}^{i=n-2} \Sigma_{j=i+1}^{j=n-1} 1$
- $= \Sigma_{i=0}^{i=n-2} n - 1 - i = n(n-1)/2 \in \Theta(n^2)$

- So complexity is $O(n^2)$ (in fact it is $\Theta(n^2)$)
- For all, average, best, worst
- Space: In place (plus constant).
- Not stable (consider sorting for $5\ 5\ 2$)
- Stable sorting algorithms maintain relative order for equal numbers.

Bubble Sort

- Swap pairwise elements which are out of order, from beginning to end.
- In each round at least one more element is in rightful place (last i elements after i rounds)
- Stable

Bubble Sort

Bubble Sort
Input: $A[0..n-1]$
    For $i \leftarrow 0$ to $n-2$ do
        For $j \leftarrow 0$ to $n-2-i$ do
            If $A[j] > A[j+1]$, then swap $A[j]$ and $A[j+1]$.
        EndFor
    EndFor
End

$$
\begin{array}{ccccccc}
54 & \overset{?}{\leftrightarrow} & 43 & & 69 & & 34 \\
43 & & 54 & \overset{?}{\leftrightarrow} & 69 & & 34 \\
43 & & 54 & & 69 & \overset{?}{\leftrightarrow} & 34 \\
43 & & 54 & & 34 & & 69
\end{array}
$$

Analysis

- In iteration i, takes $n - 1 - i$ comparisons.
- $C(n) = \Sigma_{i=0}^{i=n-2} \Sigma_{j=i+1}^{j=n-1} 1$
- $= \Sigma_{i=0}^{i=n-2} n - 1 - i = n(n-1)/2 \in \Theta(n^2)$
- In place and Stable

Insertion Sort

– Array $A[1:n]$ given
– Progressively, insert the i-th element of the array into $A[1:i-1]$, which is already sorted.
Example:

$5, 3, 7, 9, 2, 6$

$5, 3, 7, 9, 2, 6$

$3, 5, 7, 9, 2, 6$

$3, 5, 7, 9, 2, 6$

$3, 5, 7, 9, 2, 6$

$2, 3, 5, 7, 9, 6$

$2, 3, 5, 6, 7, 9$

Insertion Sort

Input: $A[1:n]$.
Output: Sorted array in increasing order.

Insertion Sort
1. For $i = 2$ to n {
   2. Let $temp = A[i]$
   3. Let $j = i - 1$
   4. While $j \geq 1$ and $A[j] > temp$ Do {
      $A[j+1] = A[j]$
      $j = j - 1$
      }
   5. $A[j+1] = temp$.
   }

End

In the iteration of the For loop at step 1, for a particular value of i:
(a) just before the start of the iteration: $A[1], \ldots, A[i-1]$ are in increasing order,
(b) during the iteration: $A[i]$ is placed in its correct position.
(c) at the end of the iteration: $A[1], \ldots, A[i]$ are in increasing order.

(b): In each iteration of the For loop:
The while loop: “shifts" the numbers greater than $A[i]$ to the right.
Then places $A[i]$ in its correct place.

Complexity:
Number of comparisons/operations:
Best Case: $O(n)$
(happens when the numbers are already in increasing order).
The while loop condition is never true, and thus the algorithm takes a constant amount of time in each iteration of the for loop.

Worst Case:

For the iteration of the For-Loop for a particular value of i:

The while loop is executed at most $i - 1$ times. (* happens for example when the numbers are in reverse sorted order *)

Thus, the whole while loop takes time at most $c_1 * i$

Thus, the for loop iteration for a particular i takes time $\leq c_2 * i$.

Therefore the whole algorithm takes time at most:

$c_2(2 + 3 + 4 + 5 \ldots + n) \leq c_2 * n^2$.

Another method to analyze:
$T(n) \leq C * n + T(n-1)$.
This gives, $T(n) \leq C * (n + n - 1 + \ldots) \leq C * n^2 = O(n^2)$.

- $T($particular input$) =?$
- Size of input
- Worst Case $T(n)$, for a particular size n of input:
  max $\{$ T(input): input has size $n\}$
- Best Case $T(n)$
  min $\{$ T(input): input has size $n\}$
- Average Case $T(n)$

$$\frac{\sum\{\text{ T(input): input has size } n\}}{\text{number of inputs of size } n}$$

- Sometimes Average Case is with probability over different inputs of size n.
- Asymptotic Bounds, are over "all n". Gives good estimate for large enough n, $\ldots$,

Sequential Search

Sequential Search
Input $A[0:n-1]$, Use $A[n]$ as sentinel
Target key: K
$A[n] \leftarrow K$.

    $i \leftarrow 0$
    While $A[i] \neq K$ do
        $i = i + 1$
    EndWhile
    If $i < n$, then "Found" at i.
    Else, "not found"

End

Complexity: $C(n) = n + 1$ for not found.
For found, Best case 1, Worst Case n.

String Matching

Input: $A[0:n-1]$ (string)
Input: $P[0:m-1]$ (pattern)

    For $i \leftarrow 0$ to $n-m$
        Found=True
        For $j \leftarrow 0$ to $m-1$
            If $A[i+j] \neq P[j]$, Then Found=False; break
        EndFor
        If Found, then Location is i; break
    EndFor

- String: THE BIG BROWN FOR JUMPED RIGHT OVER THE LAZY DOG
- Pattern: BROWN
- Complexity $C(n) \leq (n - m + 1)m \in O(mn)$.
- Worst case is actually $C(n) = \Theta(mn)$.
- For random text, the average case is linear $\Theta(n)$ (proof not for this class).

Closest Pair

- A set of points n points in m-dimensional space
- Find the closest pair of points.
- $P_i = (x[i,0], x[i,1], x[i,3], \ldots, x[i,m-1])$.
- $d(P_i, P_j) =$
  $\sqrt{(x[i,0]-x[j,0])^2 + \ldots (x[i,m-1]-x[j,m-1])^2}$
- Consider all possible pairs (i,j), with $i < j$.

Closest Pair

Input: $x[0 : n-1, 0 : m-1]$

$dmin = \infty$

For $i \leftarrow 0$ to $n-2$ do

For $j \leftarrow i+1$ to $n-1$ do

$d = (x[i,0] - x[j,0])^2 + (x[i,1] - x[j,1])^2 + \ldots (x[i,m-1] - x[j,m-1])^2$

If $d < dmin$, then $dmin = d$, $i_1 = i, i_2 = j$.

EndFor

EndFor

minimal distance in $\sqrt{dmin}$ and i_1, i_2 is the minimal distance pair.

End

Complexity: The algorithm does $\Theta(n^2)$ distance calculations — between each pair of points.
Each distance calculation takes $\Theta(m)$ time.
Therefore, $C(n) = \Theta(n^2 m)$.

Convex Hull

- A shape S is convex if for any points P, Q in the shape, every point in the line joining P and Q is also in S. That is, for all λ with $0 \leq \lambda \leq 1$: $\lambda P + (1 - \lambda)Q \in S$.
- Convex Hull of a set of points (at least three). Smallest convex shape S which contains the points. That is, for all convex S' which contain the points, $S \subseteq S'$.
- Theorem: For any finite set of points, Convex Hull is a convex polygon, and its vertices are included in the set of points given
- Hence, we just need to find the extreme pairs of points. The polygon formed using the line segments joining these pair of points will give the convex hull.

- Extreme: All other points are on the same side of the line joint the pair of points.
- For ease, we assume no triplets of points are colinear (at least not in the boundary of the convex hull).

Convex Hull

Input: A set of n Points $P[0 : n-1]$

Output: A convex hull for the set of points

For $i \leftarrow 0$ to $n-2$ do

For $j \leftarrow i+1$ to $n-1$ do

Find the equation of the line passing through $P(i)$ and $P(j)$

Find the “distance” of $P(k)$ from the line found above for all $k \in \{0, 1, 2, \ldots, n-1\} - \{i, j\}$

If all are positive or all are negative, then include the line segment $(P(i), P(j))$ in the convex hull.

(* If only points of convex hull are needed, then include the points $P(i)$ and $P(j)$ *)

EndFor

EndFor

- Line passing through $(x_1, y_1), (x_2, y_2)$.
  $(y - y_1) = \frac{(y_2 - y_1)}{x_2 - x_1}(x - x_1)$.
- Rearrange to put it in form $ax + by + c = 0$.
- Distance of a point (x_0, y_0) from a line $ax + by + c = 0$ is given by $\frac{ax_0 + by_0 + c}{\sqrt{a^2 + b^2}}$
- Count the number of times line equation is evaluated, and distance is calculated.
- Number of times line equation is evaluated: $n(n-1)/2$.
- Number of times distance is evaluated $n(n-1)(n-2)/2$.
- $C(n) \in \Theta(n^3)$.

Exhaustive Search

Generate all possible solutions and choose the correct/best among them

Travelling Salesman Problem

- Input a weighted graph $G = (V, E)$ (undirected)
- Find a simple circuit which goes through all the vertices and has minimum weight.
- Brute Force/Exhaustive approach:
  For each possible order of the vertices (there are $n!$ of them!):
  Find the weight of the circuit formed when the vertices are traversed in that order.
  Then find the one with minimal weight.
- Take exponential time.

Knap Sack Problem

- A set of n items, each having weight and value $W[0:n-1]$, $V[0:n-1]$, and a knapsack size K.
- Find a subset $S \subseteq \{0,1,\ldots,n-1\}$, such that
  $\Sigma_{i \in S} W(i) \leq K$
  and
  $\Sigma_{i \in S} V(i)$ is maximised.
- Exhaustive approach: Consider all possible subsets of $\{0,1,\ldots,n-1\}$ (there are 2^n of them!).
- For each subset as above, check if $\Sigma_{i \in S} W(i) \leq K$, and if so this is a feasible set.
- Among all feasible sets, choose the one which maximises $\Sigma_{i \in S} V(i)$.

Some other examples

- Assigning n jobs to n people. Each person takes some time to do a job.
  Need to find assignment so that the total amount of time is minimised (where each person gets exactly one job to do).
- Cryptography.
  Encoding using a key.
  Decoding using the same (or corresponding) key.
- Brute force/exhaustive approach:
  Try to decode using all possible keys.
- Which one is the right answer: ?
  Usually only one key (or only a few of them) will give meanigful answer.
- Takes exponential time in the length of the keys.

Summary

- Exhaustive search Algorithms take a lot of time, but simple to code
- In many cases there are much better ways to solve the problem
- Brute Force is “exhaustive search”, but perhaps with some “easy” filtering.