Brute Force and Exhaustive Search

Table of Contents

Brute Force

- **brute force:** straightforward approach to solving problem, "just do it"

    – shouldn't be overlooked: brute force is applicable to wide variety of problems
    – for some problems, produces reasonable algorithms of practical value with no limit on instance size
    – expense of designing more efficient algorithm may not be justified if brute-force can solve with acceptable speed
    – even if inefficient, it may be useful for solving small-size instances of a problem
    – provides baseline to judge more efficient alternatives against

Selection sort

- scan entire list for smallest element

- swap this element with the first element

- repeat from second element, third element, …

- after $n - 1$ passes, list is sorted

```
1  SelectionSort(A[0..n-1])
2  # sort given array by selection sort
```

```
 3  # input: array A[0..n-1] of orderable elements
 4  # output: array A[0..n-1] sorted in non-decreasing order
 5  for i=0 to n-2 do
 6    min = i
 7    for j=i+1 to n-1 do
 8      if A[j] < A[min]:
 9        min = j
10    swap A[i] and A[min]
```

- basic operation: key comparison

- number of times executed depends only on array size

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

- selection sort is $\Theta(n^2)$ for all inputs

- number of key swaps is only $\Theta(n)$ which makes it suitable for swapping small number of large items

Bubble sort

- compare adjacent elements of list
- exchange them if they are out of order: largest element "bubbles up" to end of list
- next pass: 2nd largest element bubbles up
- repeat $n - 1$ times until all elements are sorted

```
1  BubbleSort(A[0..n-1])
2  # sorts a given array by bubble sort
3  # input: array A[0..n-1] of orderable elements
4  # output: array A[0..n-1] sorted in non-decreasing order
5  for i=0 to n-2 do
6      for j=0 to n-2-i do
7          if A[j+1] < A[j]:
8              swap A[j] and A[j+1]
```

- basic operation: key comparison
- number of key comparisons same for all arrays

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2} 1 = \sum_{i=0}^{n-2} [(n-2) - 0 + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

- number of key swaps is dependent on input

- in worst case (decreasing array): same as the number of key comparisons
- can make a simple tweak to improve the algorithm: if there are no exchanges during a pass, the list is sorted and we can stop. It is still $\Theta(n^2)$ on worst and average cases

First application of brute-force approach often results in an algorithm that can be improved with modest effort

Sequential Search

- compare successive elements of a list with a given search key until either a match is found (successful search) or list is exhausted without finding a match (unsuccessful search)
- strength: simplicity
- weakness: inefficiency
- simple enhancement: if you append search key to end of the list, search for the key will have to be successful, and therefore you can eliminate end of list check altogether

```
 1  SequentialSearch2(A[0..n], K)
 2  # implements sequential search with search key as a sentinel
 3  # input: array A of n elements and search key K
 4  # output: index of first element in A[0..n-1] whose value is equal to K
 5  # or -1 if no such element
 6  A[n] = k
 7  i = 0
 8  while A[i] != K do
 9    i++
10
11  if i < n:
12      return i
13  else:
14      return -1
```

- enhancement for sorted input: stop search if element is $\geq$ search key

String matching

- **text:** string of n characters
- **pattern:** string of $m(\leq n)$ characters
- find i, index of leftmost character of text substring matching the pattern
- brute force approach:
  - align pattern against first m characters
  - start matching corresponding pairs of characters from left to right until either all characters match, or a mismatch is found

- – no match: shift pattern one position to right and repeat
- – match: return index

- last possible position there can still be a match: $n - m$

```
1  BruteForceStringMatch(T[0..n-1], P[0..m-1])
2  """
3  implements brute force string matching
4  input: array T[0..n-1] of n characters of text
5         array P[0..m-1] of m characters of pattern
6  output: index of first character that starts a matching substring
7          -1 if search is unsuccessful
8  """
9  for i = 0 to n-m do:
10     j = 0
11     while j < m and P[j] = T[i+j] do:
12         j++
13     if j = m:
14         return i
15 return -1
```

- In worst case, algorithm needs to make $m(n - m + 1)$ comparisons, so $\in O(mn)$j e.g.

```
1  T = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab"
2  P = "aab"
```

- in average case, has been shown to be linear, $\Theta(n)$

Closest-Pair

- find two closest points in a set of n points
- numerical data: typically uses Euclidean distance
- cluster analysis: based on n data points, organise into hierarchy of clusters based on a metric

- – text, non-numerical data: may use other metric, e.g. Hamming distance
- – bottom-up algorithm:
- – begin with each element as separate cluster, merge into successively larger clusters by combining pairs of clusters

- consider 2D closest-pair problem: points (x, y)

- – distance between points $p_i(x_i, y_i), p_j(x_j, y_j)$ is:

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

- brute force approach: compute distance between each pair of points and find a pair with the smallest distance

    - avoid repeating distance computation for pairs of points multiple times $d(p_i, p_j), d(p_j, p_i)$ so only compute for (p_i, p_j) where $i < j$

```
1  BruteforceClosestPair(P):
2  """
3  find distance between two closest points in the plane by brute force
4  input: list of P of n (>= 2) points p1(x1,y1), ... pn(xn, yn)
5  output: distance between closest pair of points
6  """
7  d = infinity
8  for i = 1 to n-1:
9      for j = i + 1 to n do
10         d = min(d sqrt((p[i].x- p[j].x)^2 + (p[i].y-p[j].y)^2))
11  return d
```

- Computing `sqrt` is tricky: to improve this, we can simply compute d^2 and then compute $sqrt(d)$ when we are returning the value
- basic operation: squaring a number, which happens twice for each pair of points
- so the complexity is

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2 = 2 \sum_{i=1}^{n-1} (n - i) = n(n - 1) \in \Theta(n^2)$$

Exhaustive Search

- **exhaustive search:** brute-force approach to combinatorial problems

    - generate each element of the problem domain
    - select those that satisfy all constraints
    - find desired elements (e.g. one that optimises objective function)

- requires algorithm for generating combinatorial objects: this is currently assumed to exist

Travelling Salesman Problem

- **travelling salesman problem (TSP):** find shortest tour through a given set of n settings that visits each city exactly once before returning to starting city

    - represent by weighted graph

- * vertex: city
    - * edge weight: distance
  - with this formulation, problem becomes finding the shortest **Hamiltonian circuit** of the graph

- **Hamiltonian circuit:** cycle that passes through all vertices of graph exactly once

  - can represent as sequence of n+1 adjacent vertices $v_{i_0}, v_{i_1}, ..., v_{i_0}$
    - * v_{i_0} is at the start and end
    - * each vertex is distinct
  - assume, without loss of generality, all circuits start and end at one particular vertex (as they are cycles)
  - we can generate all tours as all permutations of $n - 1$ intermediate cities, compute tour length, and find the shortest one
  - some of the tours differ only by direction: you can then cut the number of vertex permutations in half: only consider permutations in which intermediate vertex *b* precedes vertex *c*

- exhaustive search impractical for TSP for all but very small n

Knapsack Problem

- n items of known weights $w_1, ..., w_n$ and values $v_1, ..., v_n$
- knapsack of capacity W
- find most valuable subset of items that fit into the knapsack
- exhaustive search: generate all subsets of n items given that total weight of each subset doesn't exceed W; find largest value among them

  - number of subsets of n-element set: 2^n
  - $\$⬚⬚(2\^n)$, regardless of how efficiently you generate the subsets
  - extremely inefficient on every input
  - e.g. of an **NP-hard problem**: no polynomial-time algorithms known for any NP-hard problem
    - * many computer scientists believe that such algorithms do not exist, but has not been proven

Assignment Problem

- n people who need to be assigned to execute n jobs, one person per job

- each person is assigned to exactly one job
- each job is assigned to exactly one person

- cost that would accrue if _i_th person is assigned to _j_th job is $C[i,j]$ for each pair $i, j \in [1, ..., n]$

- find an assignment with minimum total cost

- feasible solutions can be described by n-tuples $\langle j_i, .., j_n \rangle$

- j_i indicates job number assigned to ith person

- there is a one-to-one correspondence between feasible assignments and permutations of first n integers

- exhaustive search approach:

  - generate all permutations of $1, ..., n$
  - compute total cost of each assignment
  - select the feasible assignment with the lowest cost
  - number of permutations in general case: $n!$

- exhaustive search impractical for all but very small instances of the problem

- **Hungarian method** is a more efficient algorithm

- most often, there are no known polynomial-time algorithms for problems whose domains grow exponentially (for exact solutions)