Chapter 3: Brute Force and Exhaustive Search

Introduction

The subject of this chapter is *brute force* and its important special case, *exhaustive search*. Brute force can be described as follows:

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

Example: Computing x^n.

By the definition of exponentiation,

$$x^n = \underbrace{x \times \ldots \times x}_{n\ times}$$

This suggests simply computing x^n by multiplying 1 by x n times.

The role of brute force

1. Brute-force is applicable to a very wide variety of problems.
2. A brute-force algorithm can still be useful for solving small-size instances of a problem.
3. The expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a brute-force algorithm can solve those instances with acceptable speed.
4. A brute-force algorithm can serve an important theoretical or educational purpose as a yardstick with which to judge more efficient alternatives for solving a problem.
5. A first application of the brute-force approach often results in an algorithm that can be improved with a modest amount of effort.

Example: Bubble sort - $\Theta(n^2)$

```
Bubblesort(a[1 .. n]) {
   for (i = 2; i ≤ n; i++)
      for (j = n; j ≥ i; j--)
         if (a[j - 1] > a[j])
            a[j - 1] ⇆ a[j];
}
```

Note: We can improve the above algorithm by exploiting some observations and an upgrade version of Bubble sort is Shake sort.

Example: Brute force string matching

```
SequentialStringSearch(T[1 .. n], P[1 .. m]) {
   count = 0;
   for (i = 1; i ≤ n − m + 1; i++) {
      j = 0;
      while (j < m) && (P[1 + j] == T[i + j])
         j++;
      if (j == m)
         count++;
   }
   return count;
}
```

The best-case efficiency: $B(n) =$

The worst-case efficiency: $W(n) =$

The average-case efficiency: $A(n) =$

Find the subsequence with largest sum of elements in an array

Given an array of n integers $a_1, a_2, \dots, a_n$. The task is to find indices i and j with $1 \le i \le j \le n$, such that the sum $\sum_{k=i}^{j} a_k$ is as large as possible. If the array contains all non-positive numbers, then the largest sum is 0.

Brute force version with running time $\Theta(n^3)$

```
MaxContSubSum(a[1 .. n]) {
   maxSum = 0;
   for (i = 1; i ≤ n; i++)
      for (j = i; j ≤ n; j++) {
         curSum = 0;
         for (k = i; k ≤ j; k++)
            curSum += a[k];
         if (curSum > maxSum)
            maxSum = curSum;
      }
   return maxSum;
}
```

Upgrade version with running time $\Theta(n^2)$

```
MaxContSubSum(a[1 .. n]) {
   maxSum = 0;
   for (i = 1; i ≤ n; i++) {
      curSum = 0;
      for (j = i; j ≤ n; j++) {
         curSum += a[j];
         if (curSum > maxSum)
            maxSum = curSum;
      }
   }
   return maxSum;
}
```

Dynamic programming version with running time $\Theta(n)$

```
MaxContSubSum(a[1 .. n]) {
   maxSum = curSum = 0;
   for (j = 1; j ≤ n; j++) {
      curSum += a[j];
      if (curSum > maxSum)
         maxSum = curSum;
      else
         if (curSum < 0)
            curSum = 0;
   }
   return maxSum;
}
```

The change-making problem

Given k denominations: $d_1 < d_2 < \cdots < d_k$ where $d_1 = 1$. Find the minimum number of coins (of certain denominations) that add up to a given amount of money n.

Example: Suppose that there are 4 denominations: $d_1 = 1, d_2 = 5, d_3 = 10, d_4 = 25$ and the amount of money $n = 72$. The minimum number of coins is 6 including two pennies, two dimes, and two quarters.

Idea: Find all k-tuples $\langle c_1, c_2, \dots, c_k \rangle$ such that:

$$c_1 \times d_1 + c_2 \times d_2 + \cdots + c_k \times d_k = n$$

where c_i is the number of coins of denomination d_i. Among these k-tuples we choose the one with the smallest sum $\sum_{i=1}^{k} c_i$.

The efficiency of this algorithm is as follows:

$$T(n) = \frac{n}{d_1} \times \frac{n}{d_2} \times \dots \times \frac{n}{d_k} = \frac{n^k}{d_1 \times d_2 \times \dots \times d_k} \in \Theta(n^k)$$

because $d_1, d_2, \dots, d_k$ are constants.

Closest-Pair Problem

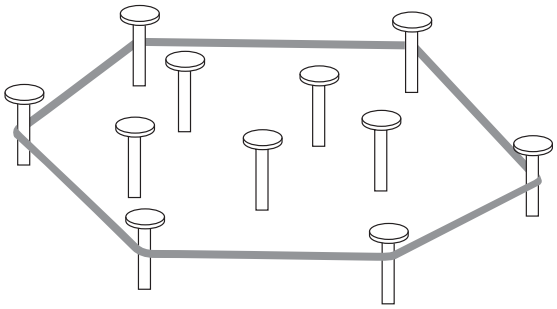
This problem calls for finding the two closest points in a set of n points in the plan.

Algorithm with running time $\Theta(n^2)$

```
BruteForceClosestPoints(P[1 .. n]) {
   dmin = ∞;
   for (i = 1; i ≤ n - 1; i++)
      for (j = i + 1; j ≤ n; j++) {
         d = √((P[i].x − P[j].x)² + (P[i].y − P[j].y)²);
         if (d < dmin) {
            dmin = d;
            point1 = P[i];
            point2 = P[j];
         }
      }
   return <point1, point2>;
}
```

Convex-hull problem

Intuitively, the *convex-hull* of a set of n points in the plane is the smallest *convex polygon* that contains all of them either inside or on its boundary.
Note: Mathematicians call the vertices of such a polygon "extreme points." Finding these points is the way to construct the convex hull for a given set of n points.



A brute force approach is based on the following observation:

"A line segment $\overline{pq}$ connecting two points p and q of a set of n points is a part of the convex hull's boundary if and only if all the other points of the set lie on the same side of the straight line through these two points."

It's known that the straight line through two points (x_1, y_1) and (x_2, y_2) in the coordinate plane can be defined by the equation

$$ax + by = c$$

where $a = y_2 - y_1, b = x_1 - x_2, c = x_1y_2 - x_2y_1$.

Second, such a line divides the plane into two half-planes: for all the points in one of them, $ax + by > c$, while for all the points in the other, $ax + by < c$. For the points on the line itself, $ax + by = c$.

Brute force algorithm with running time $\Theta(n^3)$

```
for (each point pᵢ ∈ S: i = 1 → n - 1)
  for (each point pⱼ ∈ S: j = i + 1 → n) {
    Construct the line p_ip_j;
    if (all the other points in S lie on the same side of
p_ip_j)
       Store the pair of two points <pᵢ, qⱼ>;
  }
```

Now, let's consider another brute force approach whose output is a list of the extreme points in a counterclockwise order.

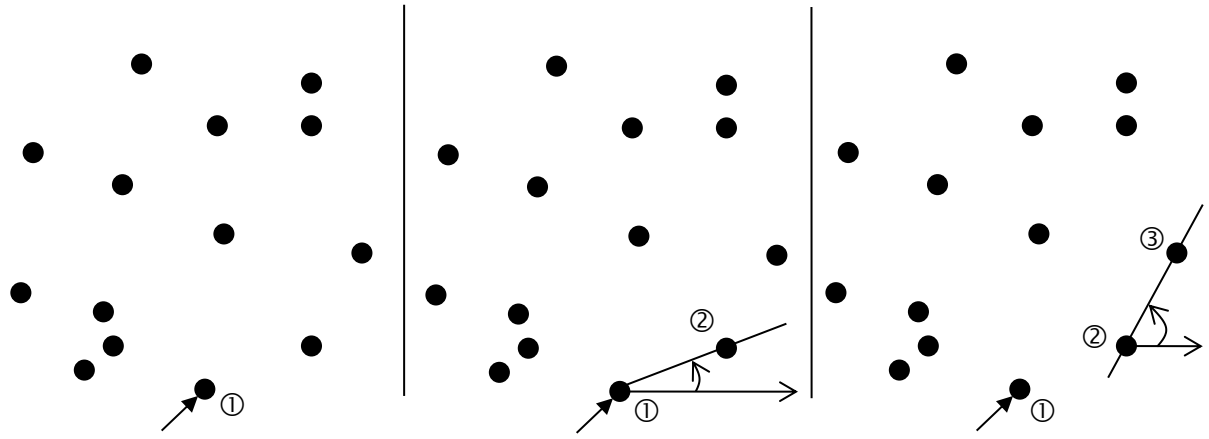
Initially, the point with the lowest *y*-value must be the first extreme point in the result list. If there are two points with the same lowest *y*-value then the leftmost one is the first extreme point.

Assume that p is the current extreme point, how to find out the next one?

Let's denote $curAngle$ the current angle (initially, $curAngle = 0$). The next extreme point q must satisfy the following condition:

$$curAngle \leq (\widehat{\overrightarrow{pq}, 0x}) < (\widehat{\overrightarrow{pr}, 0x}), \forall r \in S: r \neq p, r \neq q$$

Note: The value of $curAngle$ is increased from 0 to 2π.



Algorithm

```
computeAngle(point from, point to) {
  angle = atan2(to.y - from.y, to.x - from.x);
  if (angle < 0)
    angle += 2 * π;
  return angle;
}
```

```
findNextExtremePoint(S, cur, curAngle) {
  minAngle = 2 * π;
  S \= cur;
  for (each point p in S) {
    angle = computeAngle(cur, p);
    if (angle < minAngle && angle ≥ curAngle) {
      next = p;
      minAngle = angle;
    }
  }
  S ∪= cur;
  return [next, minAngle];
}

computeConvexHull(S) {
  convexHull = ∅;
  // Let "first" be the first extreme point
  convexHull ∪= first;
  curAngle = 0;
  point cur = first;
  while (true) {
    [next, curAngle] = findNextExtremePoint(S, cur,
curAngle);
    if (first == next)
      break;
    convexHull ∪= next;
    cur = next;
  }
  return convexHull;
}
```

What is the efficiency of this algorithm? In the worst case, the running time is $\Theta(n^2)$. Otherwise?

Exhaustive Search

Exhaustive search is simply a brute-force approach to combinatorial problems.

Algorithm for finding all subsets of a set of size n

```
for (k = 0; k < 2n; k++)
  Output the binary string of length n representing k;
```

Algorithm for generating all permutations of a set

```
Permutation(pivot, a[1 .. n]) {
  if (pivot == n)
    Output(a);
  else
    for (i = pivot; i ≤ n; i++) {
      a[pivot] ⇆ a[i];
      Permutation(pivot + 1, a);
      a[pivot] ⇆ a[i];
    }
}
Permutation(1, a);
```

Example: Let's consider all permutations of four elements a_1, a_2, a_3, a_4

a_1, a_2, a_3, a_4	a_2, a_1, a_3, a_4	a_3, a_2, a_1, a_4	a_4, a_2, a_3, a_1
a_1, a_2, a_4, a_3	a_2, a_1, a_4, a_3	a_3, a_2, a_4, a_1	a_4, a_2, a_1, a_3
a_1, a_3, a_2, a_4	a_2, a_3, a_1, a_4	a_3, a_1, a_2, a_4	a_4, a_3, a_2, a_1
a_1, a_3, a_4, a_2	a_2, a_3, a_4, a_1	a_3, a_1, a_4, a_2	a_4, a_3, a_1, a_2
a_1, a_4, a_3, a_2	a_2, a_4, a_3, a_1	a_3, a_4, a_1, a_2	a_4, a_1, a_3, a_2
a_1, a_4, a_2, a_3	a_2, a_4, a_1, a_3	a_3, a_4, a_2, a_1	a_4, a_1, a_2, a_3

Let's denote $T(n)$ the number of times the basic operation must be executed to generate all permutations of a set of n elements. The recurrence relation is as follows:

$$T(n) = nT(n-1) + \Theta(n)$$

with the initial condition $T(1) = 0$.

Hint: $T(n) \in \Omega(n!)$

Traveling Salesman Problem

The problem asks to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started.

Note: The problem can also be stated as the problem of finding the shortest Hamiltonian circuit of a weighted graph.

Hamiltonian circuit: Given a graph of n vertices. A Hamiltonian circuit is defined as a sequence of $n+1$ vertices: $v_{i_0}, v_{i_1}, \ldots, v_{i_{n-1}}, v_{i_n}$ such that:

1. $v_{i_0} \equiv v_{i_n}$
2. $\forall j \in [0, n-1]$: v_{i_j} is adjacent to $v_{i_{j+1}}$
3. $\forall j, k \in [0, n-1]$: $v_{i_j} \neq v_{i_k}$ iff $j \neq k$

In words, a Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once.

Idea: With no loss of generality, we can assume that all circuits start and end at one particular vertex v_{i_0}. Thus, we can get $(n-1)!$ *potential tours* by generating all the permutations of $n-1$ intermediate cities and attach v_{i_0} to the beginning and the end of each permutation. Then, we verify if each *potential tours* is a Hamiltonian circuit? If it's true then we compute the tour lengths, and find the shortest among them.

The time efficiency of this algoritm is $\Theta(n!)$.

Sum of subsets problem

The sum of subsets problem consists of finding all subsets of a given set $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$ of n distinct positive integers that sum to a positive integer k.

Example: If $\mathcal{A} = \{3,5,6,7,8,9,10\}$ and $k = 15$, there will be more than one subset whose sum is 15. The subsets are $\{3,5,7\}, \{7,8\}, \{6,9\}, \ldots$

A brute force approach is to generate all subsets of the given set $\mathcal{A}$ and compute the sum of each subset.

The time efficiency of this algoritm is $\Theta(2^n)$.

Knapsach problem

Given n items of known weights $w_1, w_2, \dots, w_n$ and values $v_1, v_2, \dots, v_n$ and a knapsack of capacity C, find the most valuable subset of the items that fit into the knapsack.

The knapsack problem can also be formally stated as follows:

$$maximize \sum_{i=1}^{n} v_i x_i$$

$$subject\ to \sum_{i=1}^{n} w_i x_i \leq C\ where\ x_i \in \{0,1\}, i = 1,..,n$$

A simplistic approach to solving this problem would be to enumerate all subsets of the n items, and select the one that satisfies the constraints and maximizes the profits.

The obvious problem with this strategy is the running time which is at least $\Theta(2^n)$ corresponding to the power set of n items.

Assignment problem

There are n people who need to be assigned to execute n jobs, one person per job. The cost if the i^{th} person is assigned to the j^{th} job is a known quantity $C_{i,j}$ for each pair $i, j = 1,2, \dots, n$. The problem is to find an assignment with the minimum total cost.

Example:

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

We can describe feasible solutions to the assignment problem as n-tuples $\langle j_1, j_2, \dots, j_n \rangle$: $\forall i \in [1,n], j_i \in [1,n]$ in which the i^{th} component (j_i) indicates the column of the element selected in the i^{th} row.

Example: $\langle 2,4,1,3 \rangle$ indicates the assignment of person 1 to job 2, person 2 to job 4, person 3 to job 1, and person 4 to job 3.

The exhaustive-search approach to this problem would require generating all the permutations of integers $1,2,\ldots,n$, computing the total cost of each assignment by summing up the corresponding elements of the cost matrix, and finally selecting the one with the smallest sum.

Of course, the running time of this approach is $\Theta(n!)$.