ICCS313 - Algorithm Analysis, T1 2019/20

Lecture 2: Brute-Force Algorithms & Exhaustive Search

29 September 2019

Lecturer: Chaya Hiruncharoenvate *Scribe: -*

The first type of algorithms we will look at in this course is the simplest one: brute-force algorithm.

1 Brute-Force Algorithm

Brute force is defined as

> a straightforward approach to solving a problem, usually directly based on the problems statement and definitions of the concepts involved

1.1 Discussion Questions/Class Activity 1

1. What do brute-force algorithms focus on?
2. What trade-offs do brute-force algorithms make?

1.2 Examples

1.2.1 a^n

Compute a^n $(a, n \in \mathbb{N})$

A brute-force approach for this problem is to do compute $a * a * \ldots * a$ ($n-1$ multiplications) using the following algorithm:

```
Power(a, n)
1  pwr = a
2  for i = 2 to n
3      pwr = pwr * a
4  return pwr
```

Review Question: Prove the correctness of this algorithm.

Question: What is the time complexity of $x * y$?
Answer: $\log x * \log y$

Let's look at the cost of these multiplications

Multiplication	Cost
$a * a$	$\log^2 a$
$a^2 * a$	$\log a^2 * \log a = 2\log^2 a$
$a^3 * a$	$\log a^3 * \log a = 3\log^2 a$
$\ldots$	$\ldots$
$a^{n-1} * a$	$\log a^{n-1} * \log a = (n-1)\log^2 a$

The total cost is

$$\begin{aligned} T(a,n) &= \log^2 a + 2\log^2 a + 3\log^2 a + \ldots + (n-1)\log^2 a \\ &= [1+2+3+\ldots+(n-1)]\log^2 a \\ &= \frac{(n-1)n}{2}\log^2 a \\ &\in \Theta(n^2 \log^2 a) \end{aligned}$$

If we set the upper limit for the value of a (e.g. the max value of `int`), then we can treat $\log a$ as a constant. Therefore,

$$T(a,n) = T(n) \in \Theta(n^2)$$

1.2.2 Search

Searching for a given value in a list of positive integers.

Let $A = \{a_1, a_2, \ldots, a_n\}$ where $a_i \in \mathbb{N}$, $b \in \mathbb{N}$

We want to determine if $b \in A$.

A simple brute-force algorithm is as follows:

`Search(A, b)`

```
1 for a_i in A
2     if a_i = b
3         return True
4 return False
```

Question: What is the time complexity of the comparison $x = y$?
Answer: $\min(\log x, \log y) \rightarrow \log x + \log y$

In this example, the cost of our algorithm will heavily based on the number of comparisons make to achieve the result. In contrast with the previous example, we don't know the exact number of iterations the algorithm has to go through to find the result. Therefore, we should conduct a best, worst, and average case analysis of the time complexity of this algorithm.

Best-case Analysis

$$T(A,b) = \log a_1 + \log b$$

Worst-case Analysis

$$T(A,b) = \sum_{i=1}^{n} (\log a_i + \log b)$$

Average-case Analysis

$$T(A,b) = \sum_{i=1}^{n} \left[\frac{i}{n} (\log a_i + \log b) \right]$$

Review Question: Simplify the running time and state asymptotic notations in all cases.

1.2.3 Closest-Pair

The closest-pair problem is to find the two closest points in a set of n points in the Cartesian plane, with the distance between two points measured by the standard Euclidean distance formula.

Brute-force algorithm is to compute the distance between every pair of distinct points and return the indexes of the points for which the distance is the smallest.

Let P be a list of $n \geq 2$ points where $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$

`BruteForceClosestPoints(P)`

```
1  dmin = ∞
2  for i = 1 to n − 1
3      for j = i + 1 to n
4          d = √((x_i − x_j)^2 + (y_i − y_j)^2)
5          if d < dmin
6              dmin = d
7              index1 = i
8              index2 = j
9  return index1, index2
```

The problem size is defined by the number of points, n. The basic operation is the computation of the distance between each pair of points. It needs to be computed for each pair of points, so the best, average, and worst cases are the same.

Before we continue, however, note that there is a way to improve the efficiency of this algorithm significantly. Computing square roots is an expensive operation. But if we think about it a bit, its also completely unnecessary here. Finding the minimum among a collection of square roots is exactly the same as finding the minimum among the original numbers. So it can be removed. This leaves us with the squaring of numbers are our basic operation.

The number of squaring operations can be computed:

$$\begin{aligned} C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2 \\ &= 2 \sum_{i=1}^{n-1} (n-i) \\ &= 2[(n-1) + (n-2) + \ldots + 1] \\ &= n(n-1) \in \Theta(n^2) \end{aligned}$$

Question: What is the running time of this algorithm?

1.3 Strengths & Weaknesses

1.3.1 Strengths

- wide applicability
- simplicity
- yields reasonable algorithms for some important problems (e.g., sorting, searching, matrix multiplication)

1.3.2 Weaknesses

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

Note: Brute force can be a legitimate alternative in view of the human time vs. computer time costs.

2 Exhaustive Search

Exhaustive search is a brute force approach to solving a problem that involves searching for an element with a special property, usually among *combinatorial objects* such permutations, combinations, or subsets of a set.

Method:

- systematically construct all potential solutions to the problem (often using standard algorithms for generating combinatorial objects)
- evaluate solutions one by one, disqualifying infeasible ones and, for optimization problems, keeping track of the best solution found so far
- when search ends, return the (best) solution found

2.1 Examples

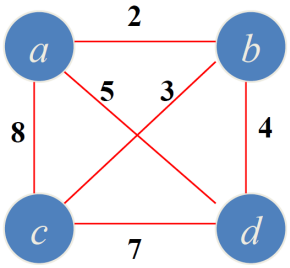
2.1.1 Traveling Salesman Problem (TSP)

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 the shortest *Hamiltonian circuit* in a weighted connected graph. (We will learn all these lingos in Week 4)

If we think about it a bit, we can see that it does not matter where our tour begins and ends its a cycle so we can choose any city/vertex as the starting point and consider all permutation of the other $n1$ vertices.

So an exhaustive search would allow us to consider all $(n1)!$ permutations, compute the cost (total distance traveled) for each, returning the lowest cost circuit found.



Question: Generate all tours and find the shortest tour in the graph above.

Question: What is the time complexity of this exhaustive search TSP algorithm?
Answer: $O(n!)$!

2.1.2 Knapsack Problem

Given n items with
weights: $w_1, w_2, \ldots, w_n$
values: $v_1, v_2, \ldots, v_n$
and a knapsack of capacity W, find most valuable subset of the items that fit into the knapsack

An exhaustive search would generate all possible subsets of the items, calculate whether the weight is feasible, tally up the total value of each subset, and finally select the subset with the highest total value.

Example: Knapsack capacity $W = 16$

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

All possible subsets are

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

2.1.3 Assignment Problem

Given n jobs and n people, and cost C_{ij} of assigning person i to job j. We need to assign one job per person where the cost is minimized.

For example, consider this cost matrix:

	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

The exhaustive search algorithm generates all possible assignments, compute the total cost of each, and select one with the lowest cost.

Question: How many assignments are possible?
Answer: $n!$

Question: What is the cheapest assignment for the example above?

2.2 Comments on Exhaustive Search

- Typically, exhaustive search algorithms run in a realistic amount of time *only on very small instances.*
- For some problems, there are much better alternatives such as shortest paths, minimum spanning tree, assignment problem.
- In many cases, exhaustive search (or variation) is the only known way to solve problem exactly for all its possible instances such as TSP, knapsack problem.