COMP 3761: Algorithm Analysis and Design

Shidong Shan

BCIT

Learning objectives

1. Describe the brute-force approach
2. State the strengths and weaknesses of the brute-force approach
3. Apply brute-force strategy to solve some common problems and analyze the algorithm complexities: sorting, searching, string matching problems
4. Apply the exhaustive search to typical combinatorial problems, e.g., Traveling Salesman Problem, Knapsack Problem, Assignment Problem.

Brute force design strategy

- A straightforward approach: "Just do it!"
- Based directly on the problem's statement and definitions
- Simple examples:
  1. Computing a^n ($a > 0$, n is a nonnegative integer)

$$a^n = a * a * \ldots * a.$$

  2. Computing $n!$

$$n! = 1 * 2 * 3 * \ldots * n.$$

  3. Sequential search: searching for a key K in an array $A[0..n-1]$

The sorting problem

- Input: a list of n orderable (comparable) items
  - numbers
  - characters
  - strings
- Output: a list sorted in a specified order.
  - nondecreasing
  - nonincreasing

Selection Sort

Pass 1 scan the array to find the smallest element, swap the smallest with the first element

Pass 2 starting with the 2nd element, scan the elements to the right to find the smallest among the rest of the list, i.e., the 2nd smallest element. and swap the 2nd smallest with the 2nd item in the list

- $\cdots$, and so on

Pass i $(0 \leq i \leq n-2)$, find the smallest element in $A[i..n-1]$ and swap it with $A[i]$, i.e., the smallest i items are in their final positions.

- after $n-1$ passes, the list is sorted in nondecreasing order.

Analysis of *SelectionSort*

- **Algorithm** *SelectionSort*$(A[0..n-1])$
  for $i \leftarrow 0..n-2$
    min $\leftarrow i$
    for $j \leftarrow i+1..n-1$
      if $A[j] < A[\text{min}]$, min $\leftarrow j$
    swap $A[i]$ and $A[\text{min}]$
- Example: Apply *SectionSort* on list: 7 3 2 5
- Analysis of time efficiency:
- How many times are the swap operations executed in the worst case?

Bubble sort algorithm

- Compare adjacent elements of the list, exchange adjacent items if they are out of order.

Pass 1 bubble up the largest element to the last position

Pass 2 bubble up the 2nd largest element to the 2nd last position

- $\cdots$, so on

Pass i $(0 \leq i \leq n-2)$, the largest i elements are in their final positions

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

Analysis of *BubbleSort*

- **Algorithm** *BubbleSort*$(A[0..n-1])$
  for $i \leftarrow 0..n-2$
    for $j \leftarrow 0..n-2-i$
      if $A[j+1] < A[j]$, swap $A[j]$ and $A[j+1]$
- Example: Apply *BubbleSort* on list: 67 23 52 35
- Analysis of time efficiency:
- How many times are the swap operations executed in the worst case?

Sorting Properties

- Stable sorting: preserves the relative order of any two equal elements in its input.
  Generally, algorithms that can exchange keys located far apart are not stable but they usually work faster; algorithms that only exchange adjacent elements are stable.
- In place sorting: does not require extra memory, except for possibly a few memory units.

Questions:

- Is selection sort stable? Is it in place?
- Is bubble sort stable? Is it in place?

String matching problem

- Input:
  - Pattern (P): a string of m characters
  - Text(T): a string of n characters ($n \geq m$)
- Output: index of first occurrence of the pattern P in text T.
- Example: T = everyonelikeshim, P = neli

The brute-force algorithm

1. Align pattern at beginning of text
2. Moving from left to right, compare each character of pattern to the corresponding character in text until all characters are found to match (success); or a mismatch is detected
3. While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

Brute force string matching

- **Algorithm** *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

  for $i \leftarrow 0..n-m$

  $j \leftarrow 0$

  while $j < m$ and $P[j] = T[i+j]$

  $j \leftarrow j+1$

  if $j = m$, return i

  return -1.

- **Analysis**:
  - $n-m+1$ different positions for alignment
  - m character comparisons per position
  - time complexity: $\theta(mn)$.

Brute-force strengths and weaknesses

- Strengths:
  - wide applicability
  - simplicity
  - yields reasonable algorithms for some important problems(e.g.,sorting, searching, string matching)
- Weaknesses
  - some brute-force algorithms are unacceptably slow
  - not as constructive as some other design techniques

Method of exhaustive search

A brute-force approach to search for an element with a special property

- generate a list of all potential solutions to the problem in a systematic manner
- evaluate potential solutions one by one, disqualifying infeasible ones
- for an optimization problem, keeping track of the best one found so far
- when search ends, announce the solution(s) found

TSP by exhaustive search

- Input: n cities with known distances between each pair
- Output: find the shortest tour that passes through all the cities exactly once before returning to the starting city.
- Example: Find the shortest tour within 4 cities.
- Time complexity of TSP by exhaustive search: $O((n-1)!)$

Knapsack problem by exhaustive search

- Input: n items
  - weights: $w_1, w_2, \ldots, w_n$
  - values: $v_1, v_2, \ldots, v_n$
  - a knapsack of capacity W
- Output: find most valuable subset of the items that fit into the knapsack.
- **Note**: the number of subsets of an n-element set is 2^n.
- Time complexity: $\Omega(2^n)$

Knapsack problem example

- **Example**: Capacity $W = 16$

item (i)	weight (w_i)	value(v_i)
1	2	20
2	5	30
3	10	50
4	5	10

Assignment problem

- n people, n jobs, one person per job
- $C[i, j]$: cost of assigning person i to job j
- find an assignment that minimizes the total cost.
- **Example**:

Person	job 1	job 2	job 3	job 4
1	9	2	7	8
2	6	4	3	7
3	5	8	1	8
4	7	6	9	4

Assignment problem by exhaustive search

- generate all legitimate assignments
- compute their costs
- select the cheapest one.
- How many assignments are there?
- pose the problem as the one about a cost matrix
- Time complexity: $O(n!)$

Comments on exhaustive search

- Exhaustive-search algorithms run in a realistic amount of time only on very small instances
- In some cases, there are much better alternatives!
  e.g. we will study some of the alternatives later in the course.
- In many cases, exhaustive search or its variation is the only known way to get exact solution

Lab Exercises

- Exercises 3.1: Problem 9
- Exercises 3.2: Problem 5
- Exercises 3.4: Problem 8