Analysis and Design of Algorithms

Chapter 3: Brute Force

School of Software Engineering © Yaping zhu

Brute Force

- ***Brute Force*（蛮力法）:** **a straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved.** ⇒ Just do it!
  - ***Example:***
    - Computing a^n ($a > 0$, n and a are nonnegative integer)
      简单地把1和*a*相乘*n*次
    - Computing $n!$
    - Consecutive integer algorithm for gcd (m, n)
    - Searching for a key of a given value in a list
    - Multiplying two matrices based on definition

Brute Force

- 蛮力法的价值:
  - 可能是唯一一种几乎什么问题都可以解决的一般性方法
  - 可以产生一些具备一定价值的算法，不必限制输入规模
  - 如果要解决问题的实例规模不大，蛮力法能够在可接受的速度范围内解决，则不必花费更多代价研究其他高效算法
  - 即使效率通常很低，仍可解决一些小规模问题实例
  - 可作为衡量其他算法效率的准绳

Brute-Force Sorting Alg.

- *Sorting Problem* (排序问题)

Given an array of *n* orderable items (e.g. numbers, characters from some alphabet, character strings), rearrange them in non-decreasing order.

几十种排序算法

https://www.cnblogs.com/onepixel/p/7674659.html

选择排序 **vs** 冒泡排序

Selection Sort （选择排序）

$$A_0 \le A_1 \le \cdots \le A_{i-1} \mid A_i, \ldots, A_{min}, \ldots, A_{n-1}$$

in their final positions   the last $n - i$ elements

- Scan the entire array to find its smallest element and swap it with the first element;
- Starting with the second element, to find the smallest among the next n-1 elements and swap it with the second element;
- Generally, on pass i ($0 \le i \le n$-2), find the smallest element in $A[i..n-1]$ and swap it with $A[i]$;
- After n-1 passes, the array is sorted.

Selection Sort

$$A_0 \le A_1 \le \cdots \le A_{i-1} \;\Big|\; A_i, \ldots, A_{min}, \ldots, A_{n-1}$$

in their final positions | the last $n - i$ elements

ALGORITHM *SelectionSort*($A[0..n-1]$)
    //Sorts a given array by selection sort
    //Input: An array $A[0..n-1]$ of orderable elements
    //Output: Array $A[0..n-1]$ sorted in ascending order
    for $i \leftarrow 0$ **to** $n-2$ **do**
        $min \leftarrow i$
        for $j \leftarrow i+1$ **to** $n-1$ **do**
            if $A[j] < A[min]$ $min \leftarrow j$
        swap $A[i]$ and $A[min]$

Selection Sort

- ***Example:***

Selection Sort on the list {89, 45, 68, 90, 29, 34, 17 }

| 89	45	68	90	29	34	**17**
17 |	45	68	90	**29**	34	89
17	29 |	68	90	45	**34**	89
17	29	34 |	90	**45**	68	89
17	29	34	45 |	90	**68**	89
17	29	34	45	68 |	90	**89**
17	29	34	45	68	89 |	90

每一行代表该算法的一次迭代，也就是说，从尾部到竖线的一遍扫描，找到的最小元素用黑体表示。竖线左边的元素已经位于它们的最终位置，所以在当前和后面的循环中，不必再考虑。

Selection Sort

Analysis of Selection Sort

ALGORITHM *SelectionSort*$(A[0..n-1])$
    //Sorts a given array by selection sort
    //Input: An array $A[0..n-1]$ of orderable elements
    //Output: Array $A[0..n-1]$ sorted in ascending order
    for $i \leftarrow 0$ **to** $n-2$ **do**
        $min \leftarrow i$
        for $j \leftarrow i+1$ **to** $n-1$ **do**
            if $A[j] < A[min]$ $min \leftarrow j$
        swap $A[i]$ and $A[min]$

- **Input size:** number of elements, *n*
- **Basic operation:** key comparison A[*j*] < A[min]
- **Check**：比较的执行次数仅仅依赖于数组的规模，该算法不需考虑最差、平均和最优效率

Selection Sort

ALGORITHM *SelectionSort*$(A[0..n-1])$

```
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in ascending order
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min]   min ← j
    swap A[i] and A[min]
```

- **Time efficiency:** $\Theta(n^2)$

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

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i = (n-1)^2 - \frac{(n-2)(n-1)}{2}$$

$$= \frac{n(n-1)}{2} \in \Theta(n^2)$$

Selection Sort

ALGORITHM *SelectionSort*$(A[0..n-1])$
//Sorts a given array by selection sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in ascending order
for $i \leftarrow 0$ **to** $n-2$ **do**
  $min \leftarrow i$
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[j] < A[min]$ $min \leftarrow j$
  swap $A[i]$ and $A[min]$

- **Number of key swaps:** n-1 次，$\Theta(n)$

在这方面，选择排序优于许多其他排序方法。

Bubble Sort （冒泡排序）

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid \underbrace{A_{n-i} \le \cdots \le A_{n-1}}_{\text{in their final positions}}$$

- Compare adjacent elements of the list and exchange them if they are out of order;
- By doing it repeatedly, we end up "bubbling" the largest element to the last position on the list;
- The next past bubbles up the second largest element, and so on until, after *n*-1 passes, the list is sorted.

Bubble Sort

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid \underbrace{A_{n-i} \le \cdots \le A_{n-1}}_{\text{in their final positions}}$$

ALGORITHM *BubbleSort* (*A* [0…*n*-1])

// Sorts a given array by bubble sort;

// Input: An array *A*[0…*n*-1] of orderable elements

// Output: Array *A*[0…*n*-1] sorted in ascending order

for *i* ← 0 to *n*-2 **do**

for *j* ← 0 to *n*-2-*i* **do**

if *A*[*j*+1] < *A*[*j*] swap *A*[*j*] and *A*[*j*+1]

Bubble Sort

- ***Example:***

Bubble Sort on the list {89, 45, 68, 90, 29, 34, 17 }

89	$\overset{?}{\longleftrightarrow}$	45		68		90		29		34		17
45		89	$\overset{?}{\longleftrightarrow}$	68		90		29		34		17
45		68		89	$\overset{?}{\longleftrightarrow}$	90	$\overset{?}{\longleftrightarrow}$	29		34		17
45		68		89		29		90	$\overset{?}{\longleftrightarrow}$	34		17
45		68		89		29		34		90	$\overset{?}{\longleftrightarrow}$	17
45		68		89		29		34		17	$\vert$	90

45	$\overset{?}{\longleftrightarrow}$	68	$\overset{?}{\longleftrightarrow}$	89	$\overset{?}{\longleftrightarrow}$	29		34		17	$\vert$	90
45		68		29		89	$\overset{?}{\longleftrightarrow}$	34		17	$\vert$	90
45		68		29		34		89	$\overset{?}{\longleftrightarrow}$	17	$\vert$	90
45		68		29		34		17	$\vert$	89		

每次交换了两个元素的位置以后，就另起一行。竖线右边的元素已经位于它们的最终位置，所以后面的循环中就不再考虑。

Bubble Sort

Analysis of Bubble Sort

ALGORITHM *BubbleSort* (A [0…n-1])

// Sorts a given array by bubble sort;

// Input: An array A[0…n-1] of orderable elements

// Output: Array A[0…n-1] sorted in ascending order

for $i \leftarrow 0$ to n-2 **do**

for $j \leftarrow 0$ to n-2-i **do**

if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

- **Input size:** number of elements, n
- **Basic operation:** key comparison
- **Check:** 对于所有规模为 n 的数组，该算法的键值比较次数相同。

Bubble Sort

ALGORITHM *BubbleSort* (*A* [0…*n*-1])

// Sorts a given array by bubble sort;

// Input: An array *A*[0…*n*-1] of orderable elements

// Output: Array *A*[0…*n*-1] sorted in ascending order

for *i* ← 0 to *n*-2 **do**

for *j* ← 0 to *n*-2-*i* **do**

if *A*[*j*+1] < *A*[*j*] swap *A*[*j*] and *A*[*j*+1]

- **Time efficiency:** $\Theta(n^2)$

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

$$= \frac{n(n-1)}{2} \in \Theta(n^2)$$

Bubble Sort

ALGORITHM *BubbleSort* (A [0…n-1])

// Sorts a given array by bubble sort;

// Input: An array A[0…n-1] of orderable elements

// Output: Array A[0…n-1] sorted in ascending order

for $i \leftarrow 0$ to n-2 **do**

for $j \leftarrow 0$ to n-2-i **do**

if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

- **Number of key swaps:** depends on the input （最坏的情况是遇到一个降序排列的数组，此时键比较和键交换的次数相同。）

$$S_{worst}(n) = C(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Improvement: if a pass through the list makes no exchanges, the list has been sorted and we can stop the algorithm.

Brute-Force Searching Alg.

- *Searching Problem* (查找问题)

  The searching problem deals with finding a given value, called a search key, in a given set.

顺序查找 **vs** 字符串匹配

Sequential Search （顺序查找）

ALGORITHM *SequentialSearch2*$(A[0..n], K)$

```
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0..n − 1] whose value is
//        equal to K or −1 if no such element is found
A[n] ← K
i ← 0
while A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

Improvement: 如果给定数组是有序的（非降序），只要遇到一个大于或等于查找键的元素，查找就可以停止了。

String Matching （字符串匹配）

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

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//        an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

Worst case: *O(mn)*

Exhaustive Search（穷举查找）

Searching Problem

- Searching for an element with a special property, in a domain that grows exponentially (or faster) with an instance size.
- Usually involve combinatorial objects such as permutations, combinations, or subsets of a set.
- Many such problems are optimization problems, to find an element that maximizes or minimizes some desired characteristic, such as a path's length or an assignment's cost.

Exhaustive Search

Exhaustive Search— Brute-Force for combinatorial

- Generate a list of all potential solutions to the problem in a systematic manner;
- Selecting those of them that satisfy all the constraints;
- Evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far;
- Search ends, announce the desired solution(s) found (e.g. the one that optimizes some objective function).

Exhaustive Search: *Traveling Salesman Problem*

- *Problem*

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.

- *Idea*
  - Weighted graph:

    vertices: cities;   edge weights: distances

The TSP problem is converted into finding the shortest Hamiltonian circuit in a weighted connected graph.

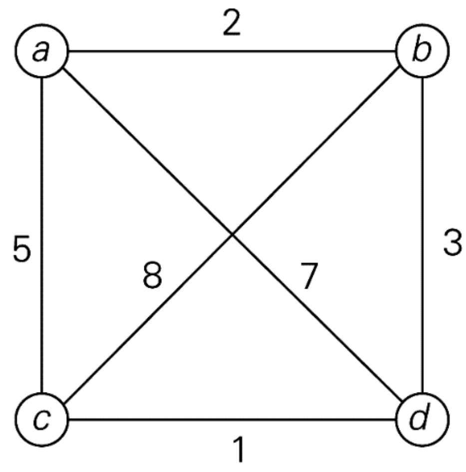
Hamiltonian circuit: a cycle that passes through all the vertices of the graph exactly once.

Exhaustive Search: *Traveling Salesman Problem*

- *Idea*
  - Hamiltonian circuit can be defined as a sequence of n+1 adjacent vertices $v_{i0}, v_{i1}, v_{i2}, \ldots, v_{in-1}, v_{i0}$;
  - Generating all the permutations of n-1 intermediate cities;
  - Computing the tour lengths;
  - Find the shortest among them.

Traveling Salesman Problem

- ***Example:***



Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

Traveling Salesman Problem

Analysis of Exhaustive Search for TSP

- **Number of permutations:** $(n\text{-}1)!$

Exhaustive Search: Knapsack Problem

- *Problem*

Given

weights: $w_1 \quad w_2 \ \ldots \ w_n$

values: $v_1 \quad v_2 \ \ldots \ v_n$

a knapsack of capacity: W

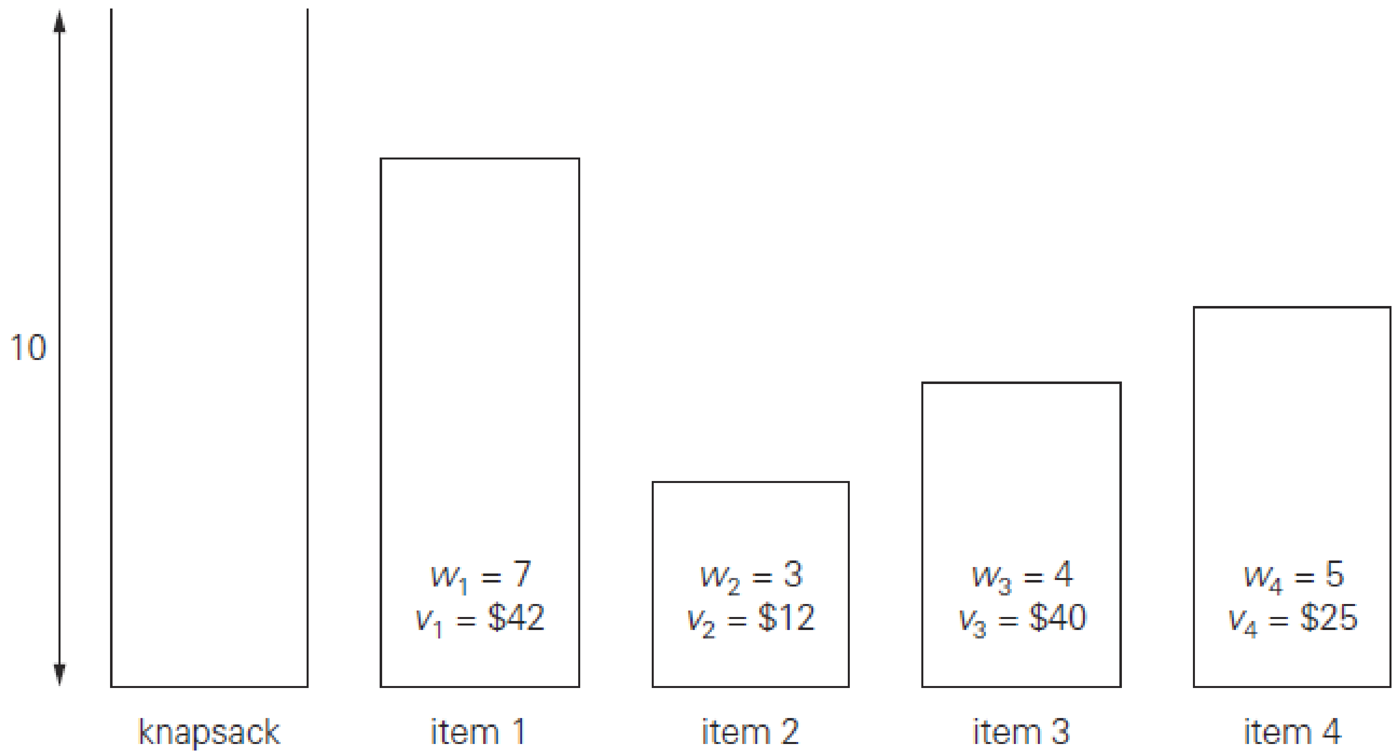
find the most valuable subset of the items that fit into the knapsack.

Exhaustive Search: *Knapsack Problem*

- *Idea*
  - Generating all subsets of the set of *n* items given;
  - Computing the total weight of each feasible subset (i.e. the ones with the total weight not exceeding the knapsack's capacity);
  - Finding a subset of the largest value among them.

Knapsack Problem

- *Example:*



Knapsack Problem

- ***Example:***

Weights: 7, 3, 4, 5

values: $42, $12, $40, $25

knapsack-capacity: 10

Subset	Total weight	Total value
$\varnothing$	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

Knapsack Problem

Analysis of Exhaustive Search for Knapsack

- **Number of subsets for an *n*-element set:** 2^n

For exhaustive search for Knapsack problem and TSP problem,

- examples of so-called *NP*-hard problem
- no polynomial-time algorithm is known for *NP*-hard problem

Exhaustive Search: ***Assignment Problem***

- *Problem*
  - There are n people who need to be assigned to n jobs, one person per job.
  - Each person is assigned to exactly one job, and each job is assigned to exactly one person.
  - The cost of assigning person i to job j is $C[i, j]$.
  - Find an assignment that minimizes the total cost.

Exhaustive Search: Assignment Problem

- *Idea*

Describe the feasible solutions to the Assignment Problem as n-tuples $\langle J_1, \ldots, J_{i,}, \ldots J_n \rangle$ in which the i-th component indicates the column of the element selected in the i-th row (i.e. job number assigned to the i-th person).

- generating all legitimate assignments
- compute their costs
- select the cheapest one

Assignment Problem

- ***Example:***

	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

Pose the problem as the one about a cost matrix:

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

$\langle 1, 2, 3, 4\rangle$ cost = 9 + 4 + 1 + 4 = 18

$\langle 1, 2, 4, 3\rangle$ cost = 9 + 4 + 8 + 9 = 30

$\langle 1, 3, 2, 4\rangle$ cost = 9 + 3 + 8 + 4 = 24

$\langle 1, 3, 4, 2\rangle$ cost = 9 + 3 + 8 + 6 = 26

$\langle 1, 4, 2, 3\rangle$ cost = 9 + 7 + 8 + 9 = 33

$\langle 1, 4, 3, 2\rangle$ cost = 9 + 7 + 1 + 6 = 23

etc.

思考: 可不可以选择每行中最小的元素？

Assignment Problem

Analysis of Exhaustive Search for Assignment

- **Number of permutations:** $n!$

-- *NP-hard problem*

Summary

- 蛮力法是一种简单直接地解决问题的方法，通常直接基于问题的描述和所涉及的概念定义。
- 蛮力法的优点：广泛适用性和简单性；
  蛮力法的缺点：大多效率低。
- 蛮力法一般是得到一个算法，为设计改进算法提供比较依据。
- 穷举法是蛮力法之一，包括旅行商问题，背包问题和分配问题。

Next

- 深度优先搜索
- 广度优先搜索