Brute Force

From now on, we are going to learn some basic and general strategies in designing algorithms to solve some typical computing problems.

We will analyze the efficiency of these algorithms using the tools learned in the past several classes

We will learn how to design algorithms with better efficiency

First, let's talk about the Brute Force strategies– the simplest

- Brute Force is a straightforward approach to solving a problem, usually directly based on the problem's statement and definitions of the concepts involved
- In many cases, Brute Force does not provide you a very efficient solution

Sorting Algorithm

We have discussed one sorting algorithm: Selection Sort

What are the basic idea behind the Selection Sort algorithm?

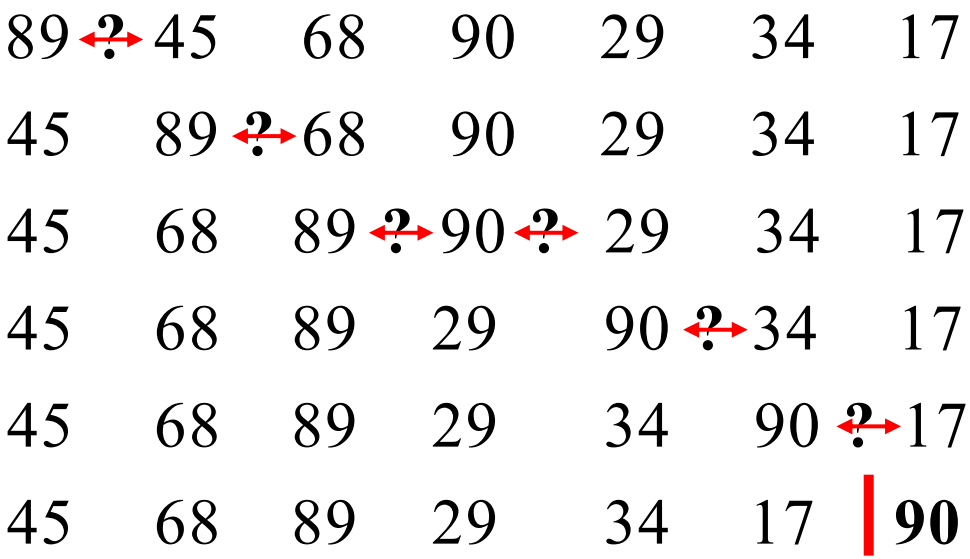
- Scanning the entire given list to find its smallest element and swap it with the first element
- This is a straightforward solution – Brute Force strategy
- What is its time efficiency – $\Theta(n^2)$

An example: sorting the numbers [89 45 68 90 29 34 17]

Another Brute-Force Application: Bubble Sort

Compare adjacent elements and exchange them if they are out of order

This the result after the first pass, which moves the largest as the rightmost element



Algorithm in Pseudocode

ALGORITHM *BubbleSort*$(A[0..n-1])$
// **The algorithm sorts array** $A[0..n-1]$ **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** $A[j+1]$

What is the time efficiency?

$$\sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 \in \Theta(n^2)$$

Is there any way to improve the algorithm?

Sequential Search – Brute Force

Find whether a search key is present in an array

```
// Search key K in A[0..n-1]
ALGORITHM SequentialSearch(A[0..n], K)
A[n] ← K
i ← 0
while A[i] ≠ K do
    i ← i + 1
if i < n return i
elsereturn -1
```

Basic operation? Input size?

What is time efficiency of this algorithm?

Brute-Force String Matching

Find a pattern in the text: Pattern – 'NOT', text – 'NOBODY_NOTICED_HIM'

Typical Applications – 'find' function in the text editor, e.g., MS-Word, Google search

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

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

  $j \leftarrow 0$

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

    $j \leftarrow j+1$

  if $j = m$ **return** i

return -1

What is the time efficiency of this algorithm (the best case, the average case, and the worst case)?

Best case: $\Theta(1)$ Average case: $\Theta(m+n)$ Worst case: $\Theta(mn)$

Exhaustive Search

Combinatorial problem: finding the optimal combination from a finite set of combinatorial objects

A brute-force approach to combinatorial problem

- Involve combinatorial objects such as permutations, combinations, and subsets of a given set
- Two-step solution:
  - Generate every element of the problem's domain
  - Then compare and select the desirable element that satisfies the constraints
- The time complexity is high – usually the complexity grows exponentially with the input size

Exhaustive Search

Three examples

- Traveling salesman problem
- Knapsack problem
- Assignment problem

Introduce the brute-force solutions to these problems

We will explore efficient solutions using advanced algorithm strategies

Traveling Salesman Problem

Problem statement: Find the shortest tour through a given *n* cities that visits each city exactly once before returning to the starting city

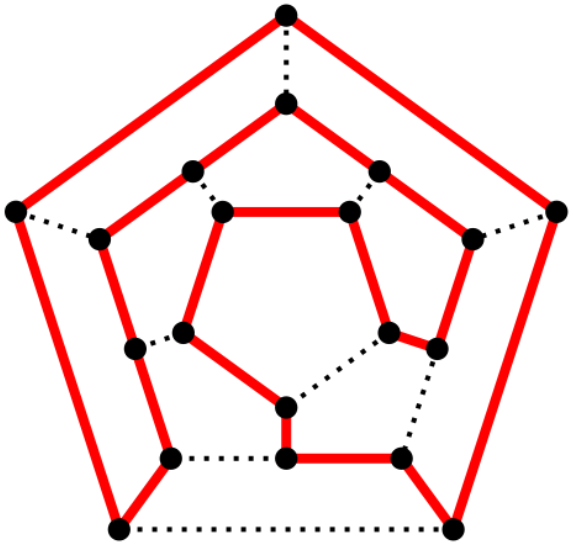
Representation using graph model:

- city → vertex
- road → edge
- length of the road → edge weight.

Formulate the problem: TSP → shortest Hamiltonian Circuit

- a cycle that passes through all the vertices of the graph exactly once

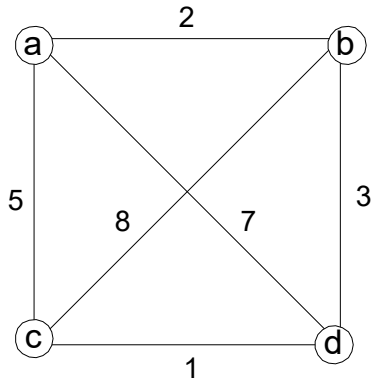
Traveling Salesman Problem



Solution -- Exhaustive search:

- List all the possible Hamiltonian circuits (starting from any vertex)
- Ignore the direction
- How many candidate circuits do we have? → (n-1)!/2
- Very high complexity

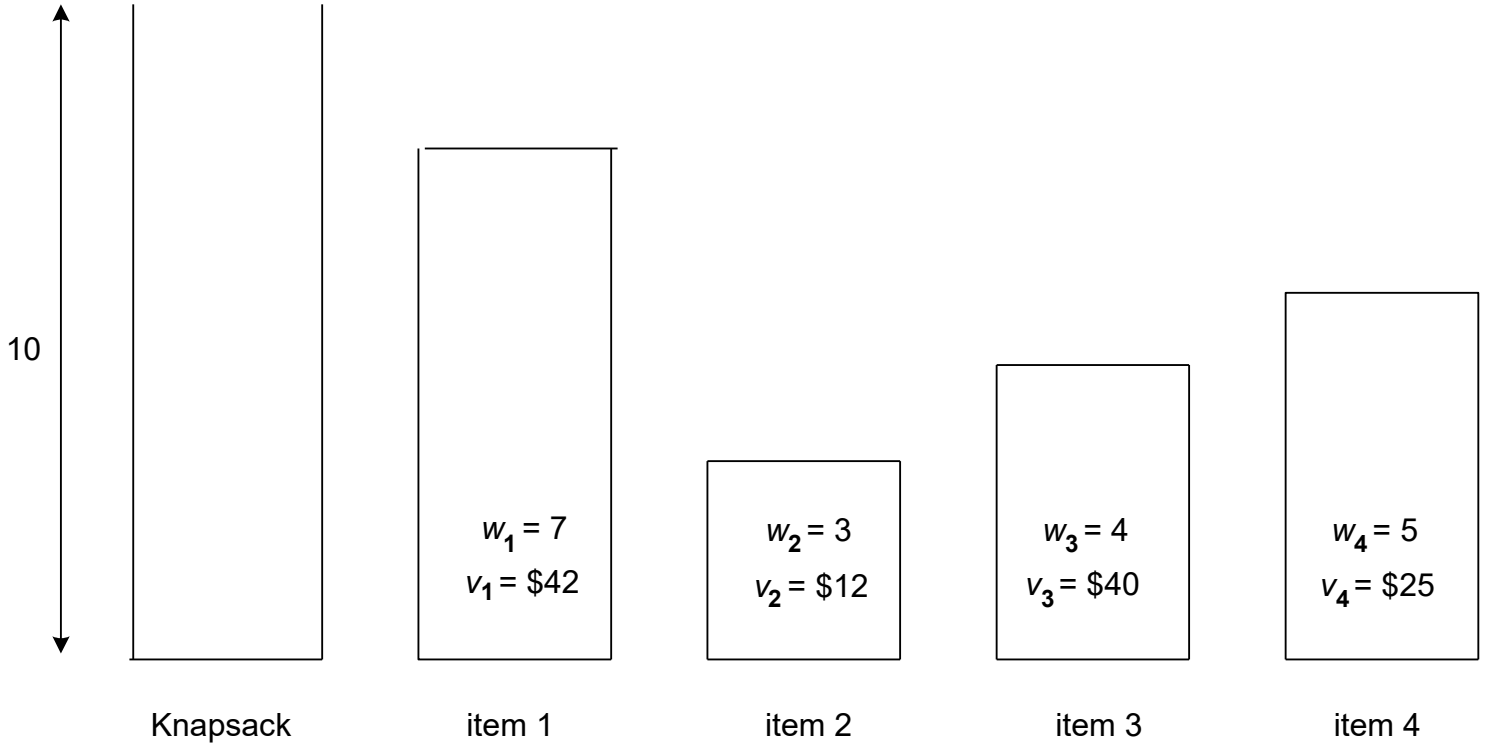
TSP Example



Tour	Length	
a ---> b ---> c --->d ---> a	l = 2 + 8 + 1 + 7 = 18	
a ---> b---> d ---> c ---> a	l = 2 + 3 + 1 + 5 = 11	optimal
a ---> c ---> b ---> d --->a	l = 5 + 8 + 3 + 7 = 23	
a ---> c ---> d ---> b ---> a	l = 5 + 1 + 3 + 2 = 11	optimal
a---> d ---> b ---> c ---> a	l = 7 + 3 + 8 + 5 = 23	
a ---> d ---> c ---> b ---> a	l = 7 + 1 + 8 + 2 = 18	Redundant paths

Knapsack Problem

Given *n* items of known weights w_1, w_2,…, w_n and values v_1, v_2,…, v_n and a knapsack of capacity *W*, find the most valuable subset of the items that fit into the knapsack.

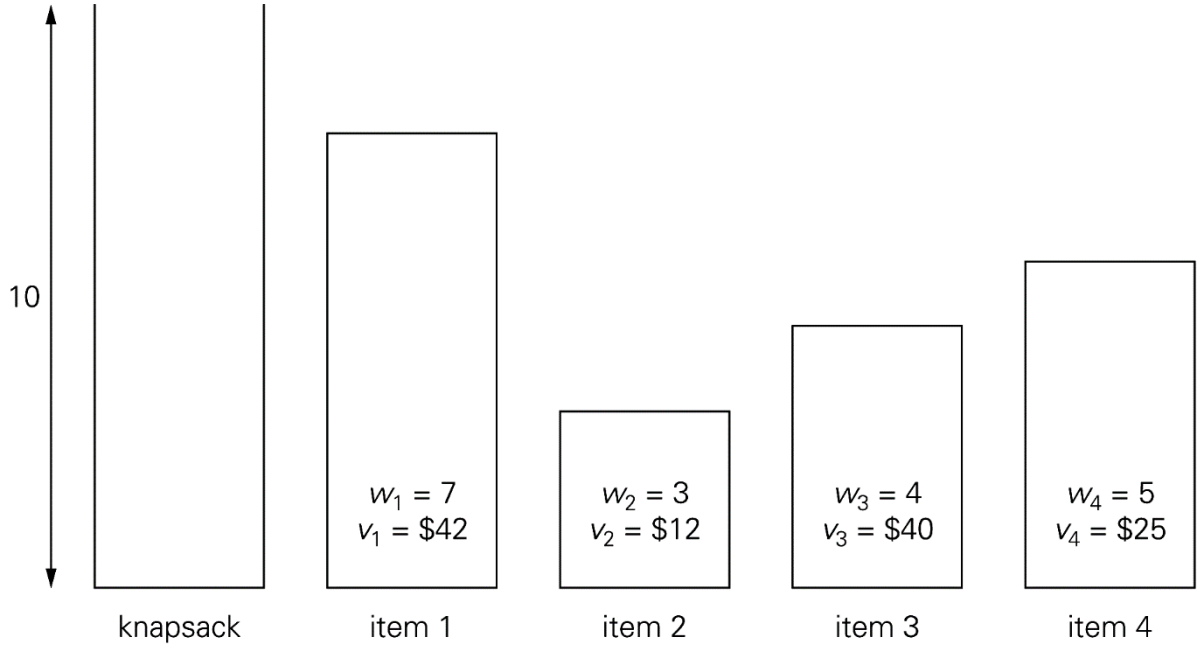


Brute-force Solution to the Knapsack Problem

Exhaustive search:

- Find all subset of the *n* items
- Each item can be selected or not selected to the knapsack
- In total, we have 2^n subsets for n items. Why?
- Select the one with the largest value while satisfies the capacity constraint.
- Complexity 2^n is very high.

Knapsack Example



Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$36
{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

Assignment Problem

***n* people to be assigned to execute *n* jobs, one person per job. *C*[*i,j*] is the cost if person *i* is assigned to job *j*. Find an assignment with the smallest total cost**

	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

Exhaustive search

- How many kinds of different assignments?

The permutation of n persons → $n!$
Very high complexity

Polynomial and non-polynomial Complexity

1	constant
log *n*	logarithmic
n	linear
n log *n*	*n* log *n*
n^2	quadratic
n^3	cubic
2^n	exponential
n!	factorial

Non-polynomial

TSP, Knapsack, and Assignment problem

Summary of Brute Force Strategies

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

In many cases, Brute Force does not provide you a very efficient solution

Examples:

- Sorting algorithms: selection sort and bubble sort $\Theta(n^2)$
- Sequential search $\Theta(n)$ and string match $\Theta(nm)$
- Closest points $\Theta(n^2)$
- Convex polygons $\Theta(n^3)$
- Exhaustive Search for Combinatorial Problem
  - Traveling salesman problem $\Theta(n!)$
  - Knapsack problem $\Theta(2^n)$
  - Assignment problem $\Theta(n!)$

Reading Assignment

Chapter 3.3 Closest-pair and Convex-Hull Problems by Brute Force

Design More Efficient Algorithms

If the exhaustive-search (brute-force) strategy takes non-polynomial time, it does not mean that there exists no polynomial-time algorithm to solve the same problem

In the coming lectures, we are going to learn many such kinds of strategies to design more efficient algorithms.

These new strategies may not be as straightforward as brute-force ones, e.g., the log n –time algorithm to compute a^n using a decrease-and-conquer strategy

Now, Chapter 4: Decrease and Conquer

Reduce problem instance to smaller instance of the same problem and extend solution

- Solve smaller instance
- Extend solution of smaller instance to obtain solution to original problem

Example: $f(n) = f(n-1) + c$

Also referred to as *inductive* or *incremental* approach

Examples of Decrease and Conquer

Decrease by one:

- Insertion sort
- Graph search algorithms:
  - DFS
  - BFS
  - Topological sorting

Decrease by a constant factor

- Binary search

Variable-size decrease

- Euclid's algorithm for computing gcd
- Selection by partition

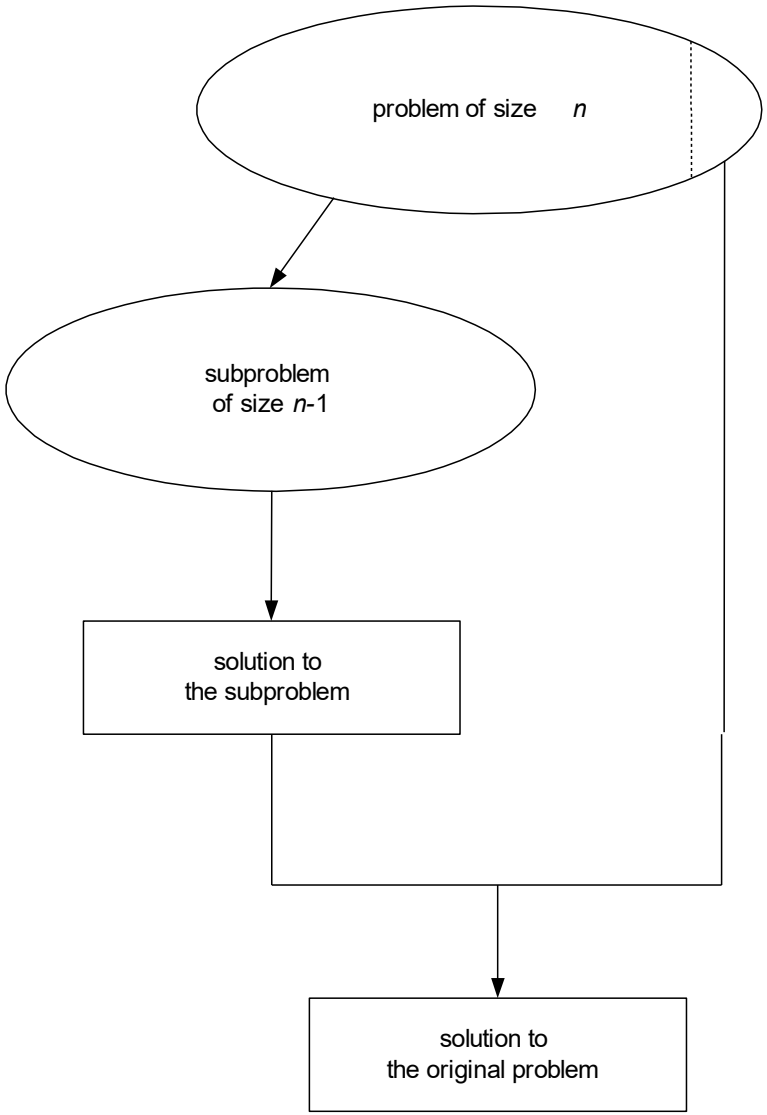
What's the difference?

Consider the problem of exponentiation: Compute a^n

Decrease-by-a-constant

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 1 \\ a & \text{if } n = 1 \end{cases}$$

Efficiency class?



Important Recurrence Types:

One (constant) operation reduces problem size by one.

$T(n) = T(n\text{-}1) + c \qquad T(1) = d$

Solution: $T(n) = (n\text{-}1)c + d$ — *linear, e.g., factorial*

A pass through input reduces problem size by one.

$T(n) = T(n\text{-}1) + cn \qquad T(1) = d$

Solution: $T(n) = [n(n+1)/2 - 1]\, c + d$ — *quadratic, e.g., insertion sort*

One (constant) operation reduces problem size by half.

$T(n) = T(n/2) + c \qquad T(1) = d$

Solution: $T(n) = c \log_2 n + d$ — *logarithmic, e.g., binary search*

Note: you can have similar solution with an arbitrary base b

A pass through input reduces problem size by half.

$T(n) = 2T(n/2) + cn \qquad T(1) = d$

Solution: $T(n) = cn \log_2 n + d\, n$ — $n \log_2 n$, *e.g., mergesort*

Consider the problem of exponentiation: Compute a^n

Decrease-by-a-constant-factor

$$f(n) = \begin{cases} [f(n/2)]^2 & \text{if } n \text{ is even and positive} \\ [f((n-1)/2)]^2 \cdot a & \text{if } n \text{ is odd and greater than } 1 \\ a & \text{if } n = 1 \end{cases}$$

Efficiency class?

Important Recurrence Types:

One (constant) operation reduces problem size by one.

$T(n) = T(n\text{-}1) + c \qquad T(1) = d$

Solution: $T(n) = (n\text{-}1)c + d$ — *linear, e.g., factorial*

A pass through input reduces problem size by one.

$T(n) = T(n\text{-}1) + cn \qquad T(1) = d$

Solution: $T(n) = [n(n+1)/2 - 1]\, c + d$ — *quadratic, e.g., insertion sort*

One (constant) operation reduces problem size by half.

$T(n) = T(n/2) + c \qquad T(1) = d$

Solution: $T(n) = c \log_2 n + d$ — *logarithmic, e.g., binary search*

Note: you can have similar solution with an arbitrary base b

A pass through input reduces problem size by half.

$T(n) = 2T(n/2) + cn \qquad T(1) = d$

Solution: $T(n) = cn \log_2 n + d\, n$ — $n \log_2 n$, *e.g., mergesort*

Variable-size-decrease

A size reduction pattern varies from one iteration of an algorithm to another

Example: Euclid's algorithm for computing the greatest common divisor

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

The arguments on the right-hand side are always smaller than those on the left-hand side

But they are not smaller neither by a constant nor by a constant factor