Exhaustive Search

Exhaustive Search - Definition

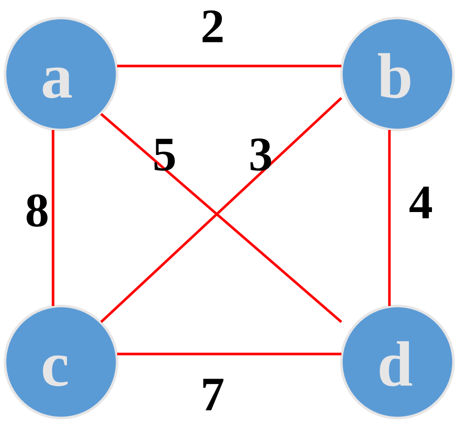
- A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as a permutations, combinations, or subsets of a set.

Method

- Construct a way of listing all potential solutions to the problem in a systematic manner
  - all solutions are eventually listed
  - no solution is repeated
- Evaluate solutions one by one, perhaps disqualifying infeasible ones and keeping track of the best one found so far
- When search ends, announce the winner

Example 1: Traveling salesman 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.
- Alternatively: Find shortest *Hamiltonian circuit* in a weighted connected graph.
- Example:



Traveling salesman by exhaustive search

Tour	Cost
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17

- Efficiency:

Traveling salesman by exhaustive search

Tour	Cost
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17

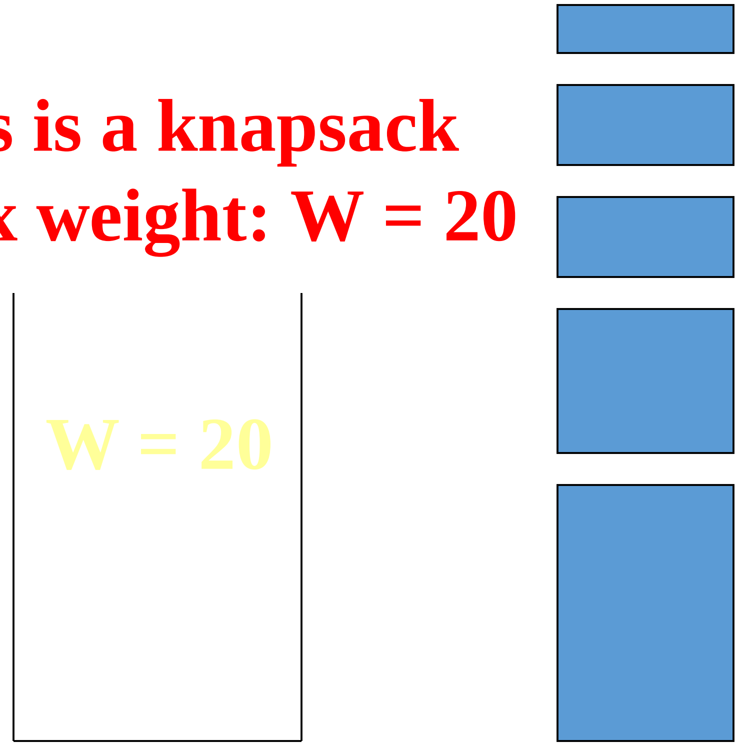
- Efficiency: (n-1)!/2

0-1 Knapsack problem

- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value v_i
- <u>Problem</u>: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem: a picture

This is a knapsack
Max weight: W = 20



Items	Weight w_i	Benefit value v_i
	2	3
	3	4
	4	5
	5	8
	9	10

0-1 Knapsack problem

- Problem, in other words, is to find

$$\max \sum_{i \in T} v_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

- **The problem is called a "*0-1*" problem, because each item must be entirely accepted or rejected.**
- **In the "*Fractional Knapsack Problem,*" we can take fractions of items.**

0-1 Knapsack problem: brute-force approach

Let's first solve this problem with a straightforward algorithm

- We go through all combinations (subsets) and find the one with maximum value and with total weight less or equal to W

Example 2: Knapsack Problem

Given n items:

- weights: $w_1 \quad w_2 \ldots \quad w_n$
- values: $v_1 \quad v_2 \ldots \quad v_n$
- a knapsack of capacity W

Find the most valuable subset of the items that fit into the knapsack

Example:

Knapsack capacity W=16

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

Knapsack by exhaustive search

Subset	*Total weight*	*Total value*
∅	*0*	*$0*
{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

Most valuable subset?

Efficiency:

0-1 Knapsack problem: brute-force approach

- Algorithm:
  - We go through all combinations and find the one with maximum value and with total weight less or equal to *W*

Efficiency:

- Since there are *n* items, there are 2^n possible combinations of items.
- Thus, the running time will be $O(2^n)$

Assignment Problem

- There are n people who need to be assigned to execute n jobs, one person per job.
- C[i, j] : cost that would accrue if i-th person is assigned to j-th job.
- Find an assignment with the minimum total cost.

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

- Select one element in each row so that all selected elements are in different columns and total sum of the selected elements is the smallest
- <j_1, j_2, …, j_n> are feasible solution tuples i-th component indicates the column of the element selected in i-th row e.g., <2, 3, 4, 1>
- Generate all permutations of <1, 2, 3, 4>
- And compute total cost, find the smallest cost

Brute force strengths and weaknesses

- Strengths:
  - wide applicability
  - simplicity
  - yields reasonable algorithms for some important problems
    - sorting; matrix multiplication; closest-pair; convex-hull
  - yields standard algorithms for simple computational tasks and graph traversal problems

Brute force strengths and weaknesses

- Weaknesses:
  - rarely yields efficient algorithms
  - some brute force algorithms unacceptably slow
    - e.g., the recursive algorithm for computing Fibonacci numbers
  - not as constructive/creative as some other design techniques

Selection Sort

- Given n orderable items (e.g., numbers, characters, etc.) how can you rearrange them in non-decreasing order?
- Selection Sort:
  - On the i-th pass (i goes from 0 to n-2) the algo searches for the smallest item among the last n-i elements and swaps it with A_i

```
ALGORITHM SelectionSort(A[0,..n-1])
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]
```

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

already sorted **the last n-i elements**

Example

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

Analysis

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

Bubble sort

- Compare adjacent elements and exchange them if out of order
- Essentially, it bubbles up the largest element to the last position
- $\mathbf{A_0, \ldots \ldots, A_j <\!\!-\!\!> A_{j+1}, \ldots \ldots, A_{n-i-1} \mid A_{n-i} \le \ldots \le A_{n-1}}$

```
ALGORITHM BubbleSort(A[0..n-1])
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]
```