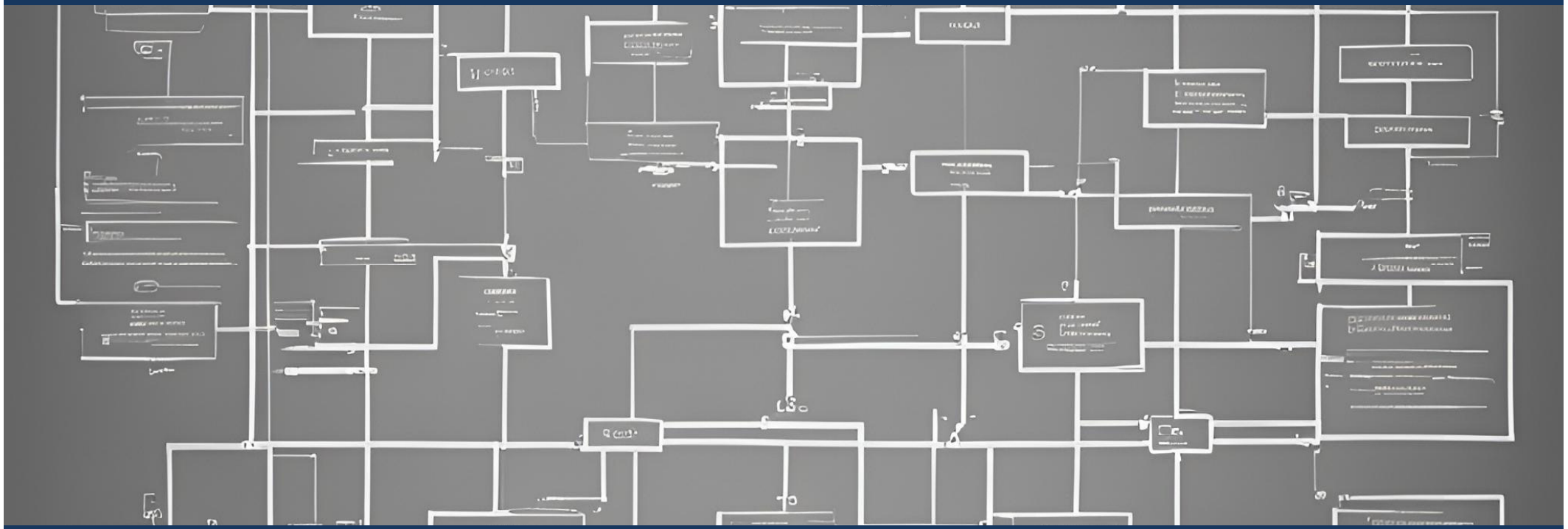
ITM 517 Algorithm

Ja-Hee Kim

Brute-Force Algorithm



Introduction

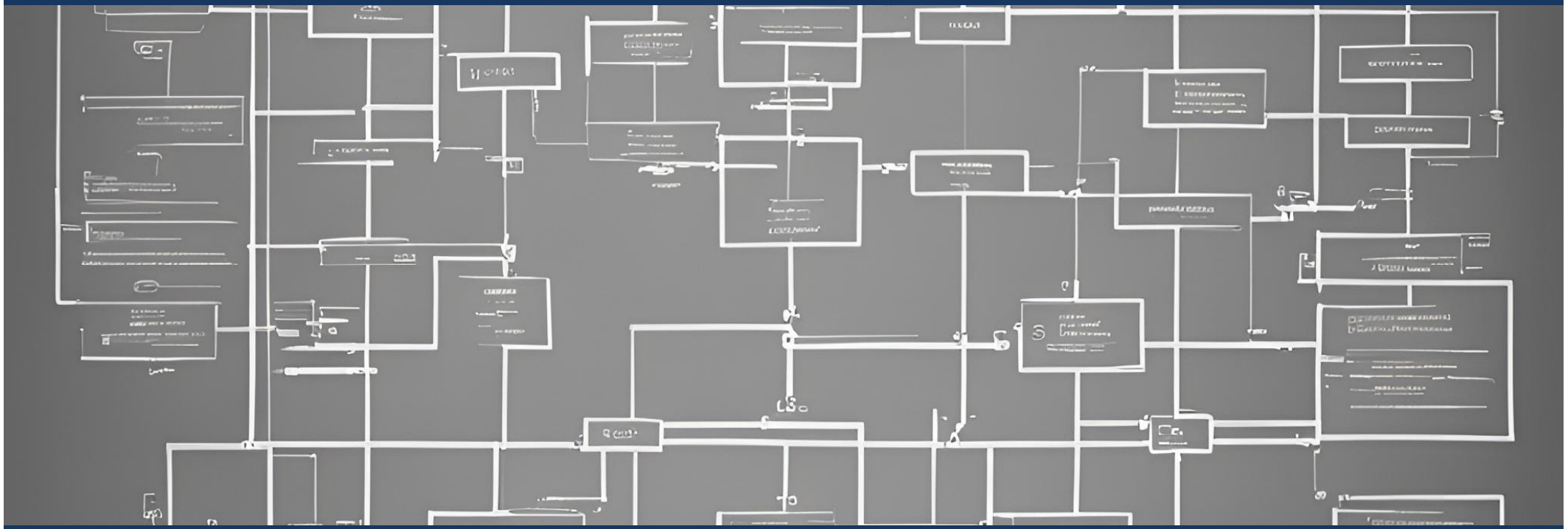
Brute Force

- Based on the problem's statement and definitions of the concepts involved.
- A **straightforward** approach, usually based directly on the problem's statement and definitions of the concepts involved

Strength and Weakness

- Strengths
  - wide applicability
  - **simplicity**
  - yields reasonable algorithms for some important problems

  (e.g., matrix multiplication, sorting, searching, string matching)
- Weaknesses
  - Usually yields **inefficient** algorithms
  - some brute-force algorithms are unacceptably slow
  - not as constructive as some other design techniques



Techniques

Basic techniques

- Optimization problems
- Generate and test
- Backtracking
- Fixing parameters
- Meet in the middle

Optimization problems

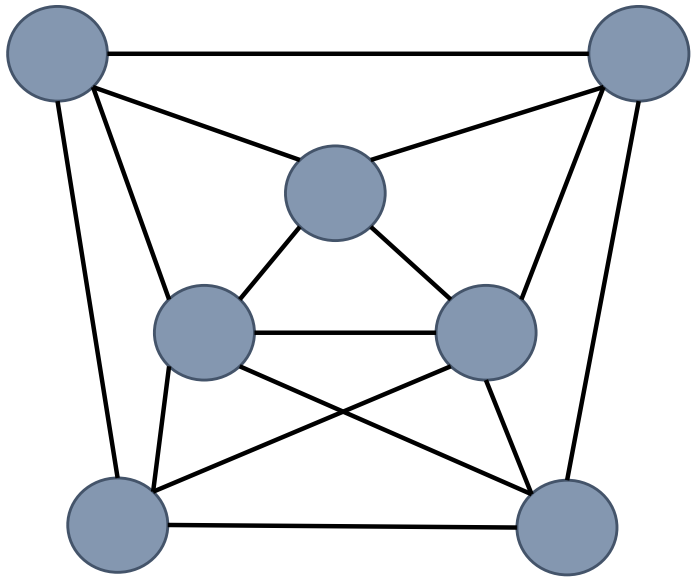
- for a given solution set *S* and a value function *f*, find an $x \in S$, which maximize *f(x)*.
- Example: Find the max value
  - Algorithm: Scan the array to find its maximum element and return it with the maximum element.

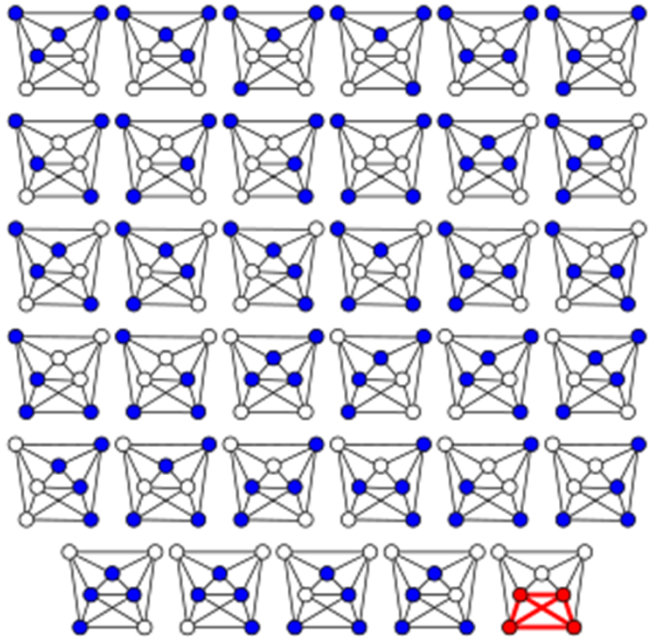
$A[0], \ldots, A[min], \ldots, A[n-1]$

  - Time efficiency: $\Theta(n)$
  - Space efficiency: $\Theta(1)$, **so in place**
  - Stability: **yes**

Generate and test

- generating all solution and test all of them.
- Example: Clique Problem
  - finding cliques (subsets of vertices, all adjacent to each other, also called complete subgraphs) in a graph.
  - Time complexity: $\Theta(2^n)$





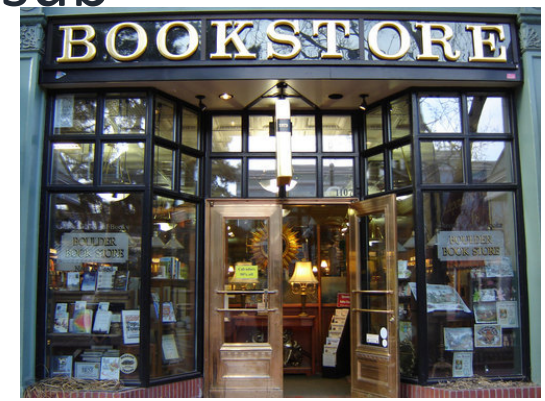
Backtracking

- Constrained problem
- Step
  - It incrementally builds candidates for solutions
  - Abandons a candidate as soon as the candidate cannot satisfy the constraint.
- Example: labyrinth
  - Input size n : the number of intersection
  - Time complexity: $\Theta(b^d)$
  - Space complexity: $\Theta(d)$

  Where b is the branching factor and d is the depth.

Fixing parameters-1

- Instead of testing the entire solution set directly, we fix certain parameters to reduce complexity
- This transforms the problem into a smaller sub-problem that's easier to solve
- Example: Buying books
  - Problem description:
    - to buy *n* books from *m* book shops.
    - Each book is sold by at least one bookstore
    - The price of each book can vary between the different stores.
    - If you order anything from a certain bookstore, you must pay for postage, which
      - may vary between bookstores
      - is the same no matter how many books you decide to order.
    - **Goal: Compute the smallest amount of money you need to pay for all the books.**



Fixing parameters-2

Brute Force Approach (Inefficient)

- Try all possible ways to buy each book from each store
- Time complexity: $\Theta(m^n)$

Fixing Parameters Approach $\Theta(m \times n \cdot 2^m)$

1. **Key insight**: The only decision that matters for each store is "Do we order from this store or not?"
2. **Fixed parameter**: For each store, we decide YES/NO on using it
3. **Optimization**: For each combination of stores, buy each book from the cheapest available store

Worked Example: Buy three books(B1, B2, B3)

Generate all subset of stores: {}, {S1}, {S2}, {S1, S2}

Calculate costs:{S1}: $5 (postage) + $10 (B1) + $20 (B2) + $15 (B3) = $50

{S2}: $3 (postage) + $15 (B1) + $10 (B2) + $20 (B3) = $48

{S1,S2}: $8 (postage) + $10 (B1) + $10 (B2) + $15 (B3) = $43

	Price			
	Postage	B1	B2	B3
Store 1 (S1)	5	10	20	15
Store 2 (S2)	3	15	10	20

Meet in the middle

- To fix half of the parameter space and build some fast structure s.t. when testing the other half of the parameter space.
- Example: subset sum.
  - Given a set of integers S, is there some subset $A \subseteq S$ with a sum equal to T?
  - *n*: the number of elements in set S
  - Time complexity: $\Theta(2^n)$

Time complexity: $\Theta(n2^{n/2})$

```
procedure SUBSETSUM(set S, target T)
    N ← |S|                                                    constant
    left ← N/2
    right ← N − left
    Lset ← the left first elements of S
    Rset ← S \ Lset
    Lsums ← new set
    for each L ⊆ Lset do                                       Θ(2^{n/2})
        Lsums.insert(∑_{l∈L} l)                                Θ(n/2) Θ(n/2 2^{n/2})
    for each R ⊆ Rset do                                       Θ(2^{n/2})
        sum ← ∑_{r∈R} r                                        Θ(n/2) Θ(n/2 2^{n/2})
        if Lsums.contains(T − sum) then                        Constant or Θ(n/2)
            output true
            return
    output false
```

Guideline

- Appropriate Scenarios
  - Prototyping, Simple problems, Correctness critical, Learning/teaching, Small test cases
- Brute Force Variations
  - Generate and Test: for small problems
  - Backtracking: When early pruning is possible
  - Parameter Fixing: When fixing some parameters significantly reduces complexity
  - Meet in the Middle: When problem can be split into two balanced parts

Thanks