2025 Fall CSE5025

Combinatorial Optimization

组合优化

Instructor: 刘晟材

Course Info

- Instructor: Shengcai Liu (刘晟材)
  - Office: Room 311, South Tower, CoE (工学院南楼311)
  - Homepage: https://cse.sustech.edu.cn/faculty/~liusc
  - Email: liusc3@sustech.edu.cn
  - Office hour: Friday 16:00-18:00 by email appointment
- Teaching Assistants (TAs)
  - Jinglin Wang (王靖林) Email: 12531330@mail.sustech.edu.cn
  - Zubin Zheng (郑祖彬) Email: 12532580@mail.sustech.edu.cn
- A bilingual course
  - Primary instruction in Chinese (中文讲解); all key terminology provided in both Chinese and English; all course materials in English (英文材料)

Couse Materials

- Lecture Slides (available on Blackboard)
- Reference textbook (optional):

[1] William J. Cook, William H. Cunningham, William R. Pulleyblank, Alexander Schrijver, Combinatorial Optimization.

[2] Eugene Lawler, Combinatorial Optimization: Networks and Matroids.

[3] B. H. Korte, Jens Vygen, Bernhard Korte, Combinatorial Optimization: Theory and Algorithms (Algorithms and Combinatorics).

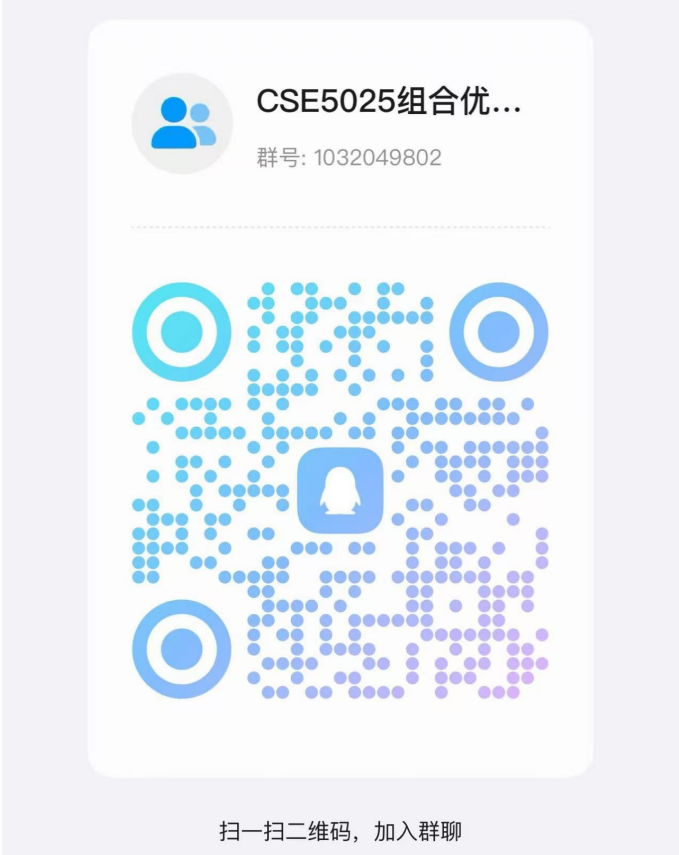
[4] Christos H. Papadimitriou, Kenneth Steiglitz, Combinatorial Optimization: Algorithms and complexity.

Couse Materials

- Blackboard course site: 组合优化 2025秋
  - Resources: Lecture slides, Assignments, Syllabus
  - Only for students in our class

- QQ群号: 1032049802

- Search online to learn more by yourself
  - Google, MathOverflow
- Use LLMs in your study!!!
  - ChatGPT, Gemini, 千问，Deepseek



Grading

- 10% Attendance and exercises (on some lectures)
- 20% Course report
- 30% Assignments (five times in total)
- 40% Final exam

Assignments （作业）

- Submit through BB
- No late assignment will be accepted
  - Unless some special situations (e.g., medical leave) which will be reviewed by the instructor
  - The following excuses will NOT be approved for late submissions: computer crashes, disk crashes, accidental file deletions, lab computer unavailability, and the like. There will be no reply for this kind of late submission requests

Regulations on Academic Misconduct in courses for Undergraduate Students in CSE Department（诚信守则）Important!!!

- If an undergraduate is found to be plagiarized in assignments, projects, or exams
  - for the first time, the assignment, course project, or exam will receive a score of 0;
  - for the second time in the same course, the grade for that course will be 0.
- Any student who is with plagiarism record or refuses to sign & submit the "Undergraduate Students Declaration Form" (有抄袭记录或拒签本科生诚信承诺书):
  - will not be allowed to enroll in the two CS majors through 1+3 mode, and cannot receive any recommendation for postgraduate admission exam exemption and all other academic awards. (影响进系、研究生推免、奖学金)
  - As it may be difficult to determine who actually wrote it when two assignments/projects are identical or nearly identical, the policy will apply to BOTH students, unless one confesses having copied without the other knowing.
- You can find the "Undergraduate Students Declaration Form" (本科生诚信承诺书) on Blackboard，and please submit to TAs through emails by 2025.09.30

Course Objectives of CSE5025

- Not a course for learning programming, but to equip students with
  - an understanding of basic concepts and theories of combinatorial optimization (e.g., network flow theory, submodularity)
  - proficiency in key algorithms (e.g., greedy, dynamic programming, branch-and-bound, heuristic search) and their real-world applications
  - the ability to analyze and solve practical combinatorial optimization problems
  - a sense of what is being studied right now in this area
- Pre-requisites
  - Algorithms and Data Structures
  - Graph Theory and Discrete Mathematics
  - Master at least one programming language, e.g., Python, Java, C

Syllabus (Tentative)

- Introduction to Combinatorial Optimization
- Complexity (P, NP, NP-complete, NP-hard)
- Graph Problems
- Approximation Algorithms
- Integer Programming
- Greedy and Local Search
- Heuristics and Meta-heuristics
- Real-World Applications of Combinatorial Optimization
- Learn to Optimize

Lecture 1:

Introduction to Combinatorial Optimization

Agenda for Today's Lecture (3 Sessions)

- Session 1: The World of Combinatorial Optimization (CO)
  - Definition, core components, and historical context
  - Why CO is crucial: applications and impact
  - Scope: “Easy” vs. “Hard” problems
- Session 2: Dive into the Knapsack Problem (背包问题，简称KP)
  - Problem definition, variations, and applications
- Session 3: Using the KP as a case study to explore:
  - Greedy Algorithm (贪心算法)
  - Approximation Algorithms (近似算法)
  - Dynamic Programming (动态规划，简称DP)
  - Branch & Bound (分支定界法，简称BB)

Combinatorial Optimization (CO)

- CO is a subfield of mathematical optimization that consists of finding an optimal object from a finite set of objects
- In essence, it's the science of making optimal choices from a discrete set of possibilities
- The core problem can be expressed as:

$$\max(\text{or } \min) f(x) \quad s.t. \ \ x \in S$$

S is the feasible set (可行解集合), a finite set of discrete structures (e.g., permutations, graphs, subsets)

x is the solution (解), containing a set of values assigned to all decision variables

$f(x)$ is the objective function (目标函数), a metric we want to maximize or minimize.

Three Core Components of a CO Problem

- Discrete Decision Variables (决策变量):
  - The choices we make are discrete, not continuous. They often answer questions like "yes or no?", "which one?", or "in what order?"
  - *Example*: For each item, do we include it in the knapsack ($x_i = 1$) or not ($x_i = 0$)?
- Constraints (约束):
  - The rules that define which combinations of choices are valid. They determine the feasible set S from the universe of all possible choices
  - *Example*: The total weight of items in the knapsack cannot exceed its capacity
- Objective Function (目标函数):
  - A single, clear metric to evaluate the "goodness" of any feasible solution, which we aim to optimize
  - *Example*: Maximize the total value of the items selected

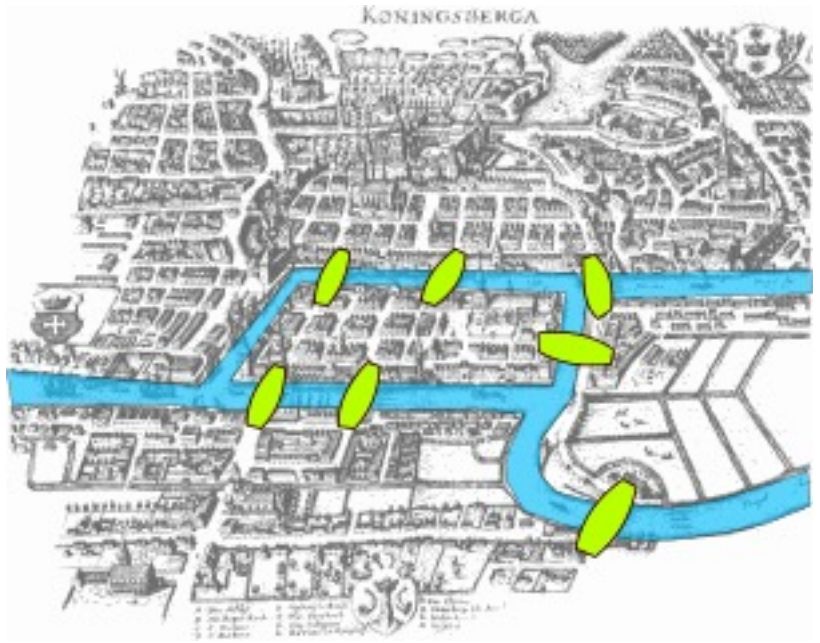
CO vs. Continuous Optimization

Feature	Combinatorial Optimization (CO)	Continuous Optimization
Decision variables	Discrete (integers, binary, permutations)	Continuous (real numbers)
Feasible Set	Finite	A region defined by continuous functions (e.g., $g(x) \leq 0$)
Solution Space	Often rugged, non-convex, with many local optima	Often smooth, sometimes convex
Typical Techniques	Approximation algorithms, Integer Programming, Heuristics, DP	Gradient descent, Newton's Method, Simplex Method
Example	Finding the shortest tour through a set of cities	Training a neural network

A Brief History of CO

- The roots of CO are old, but its formalization is a 20th-century achievement
- 1736: Leonhard Euler's "Seven Bridges of Königsberg" paper lays the foundations of graph theory, a cornerstone of CO

The problem was to devise a walk through the city that would cross each of those bridges once and only once.



A Brief History of CO

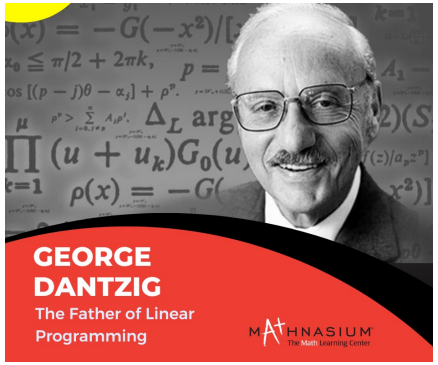
- 1930s-40s: Operations Research emerges during WWII, focusing on optimizing military logistics and resource allocation

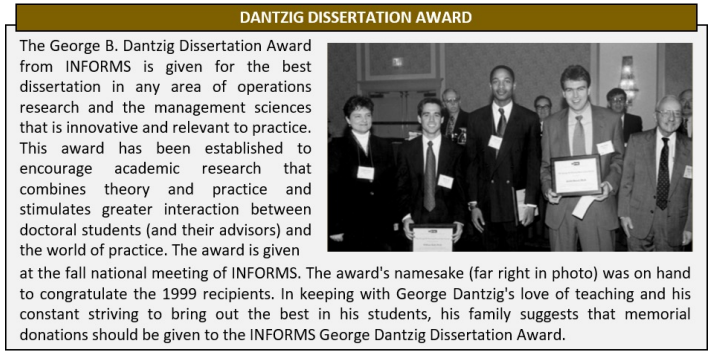
Second World War [edit]

The modern field of operational research arose during World War II.[*dubious – discuss*] In the World War II era, operational research was defined as "a scientific method of providing executive departments with a quantitative basis for decisions regarding the operations under their control".[13] Other names for it included operational analysis (UK Ministry of Defence from 1962)[14] and quantitative management.[15]

During the Second World War close to 1,000 men and women in Britain were engaged in operational research. About 200 operational research scientists worked for the British Army.[16]

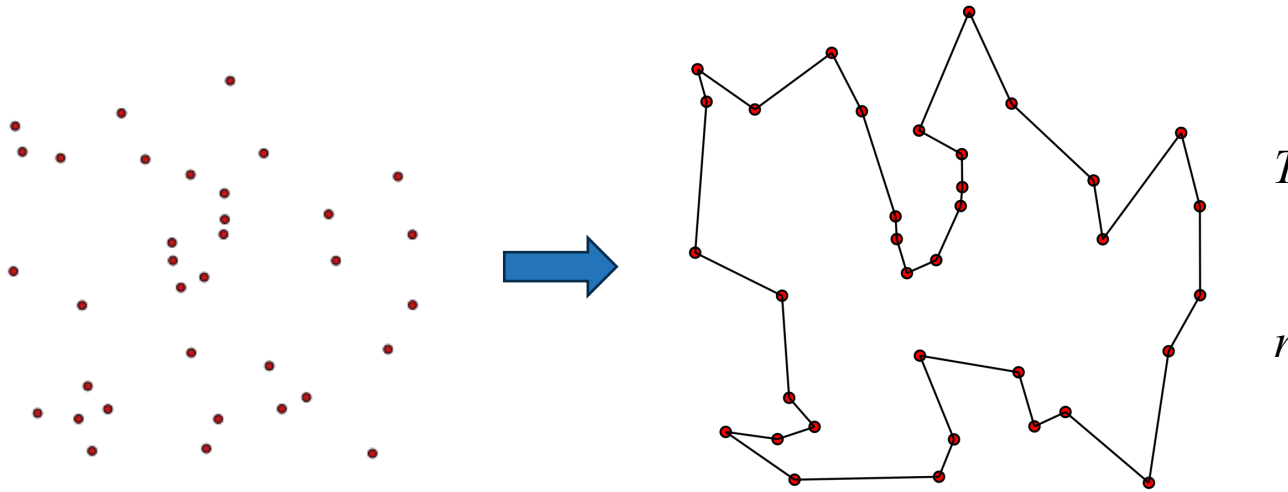
- 1947: George Dantzig develops the Simplex algorithm for Linear Programming, a powerful tool that can solve certain CO problems (if their relaxations are integral).





A Brief History of CO

- 1950s: The Traveling Salesman Problem (TSP) is formally defined and gains attention. TSP is arguably the most widely studied CO problem since then



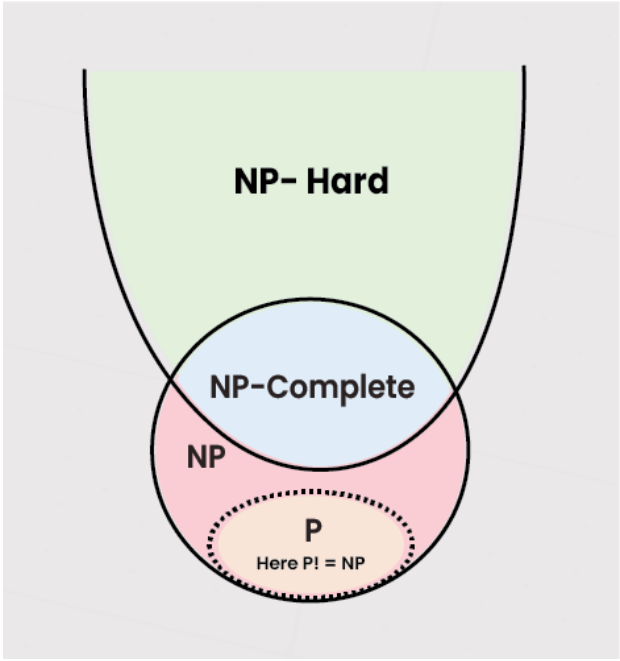
TSP: finds the shortest path that traverses each point exactly once and returns to the starting point

A Brief History of CO

- 1970s: The theory of NP-completeness (Cook, Levin, Karp) provides a formal way to classify “hard” combinatorial problems, revolutionizing the field

Contributions [edit]

The concept of NP–completeness was developed in the late 1960s and early 1970s in parallel by researchers in North America and the Soviet Union. In 1971, Stephen Cook published his paper "The complexity of theorem proving procedures"[2] in conference proceedings of the newly founded ACM Symposium on Theory of Computing. Richard Karp's subsequent paper, "Reducibility among combinatorial problems",[1] generated renewed interest in Cook's paper by providing a list of 21 NP–complete problems. Karp also introduced the notion of completeness used in the current definition of NP–completeness (i.e., by polynomial–time many–one reduction). Cook and Karp each received a Turing Award for this work.



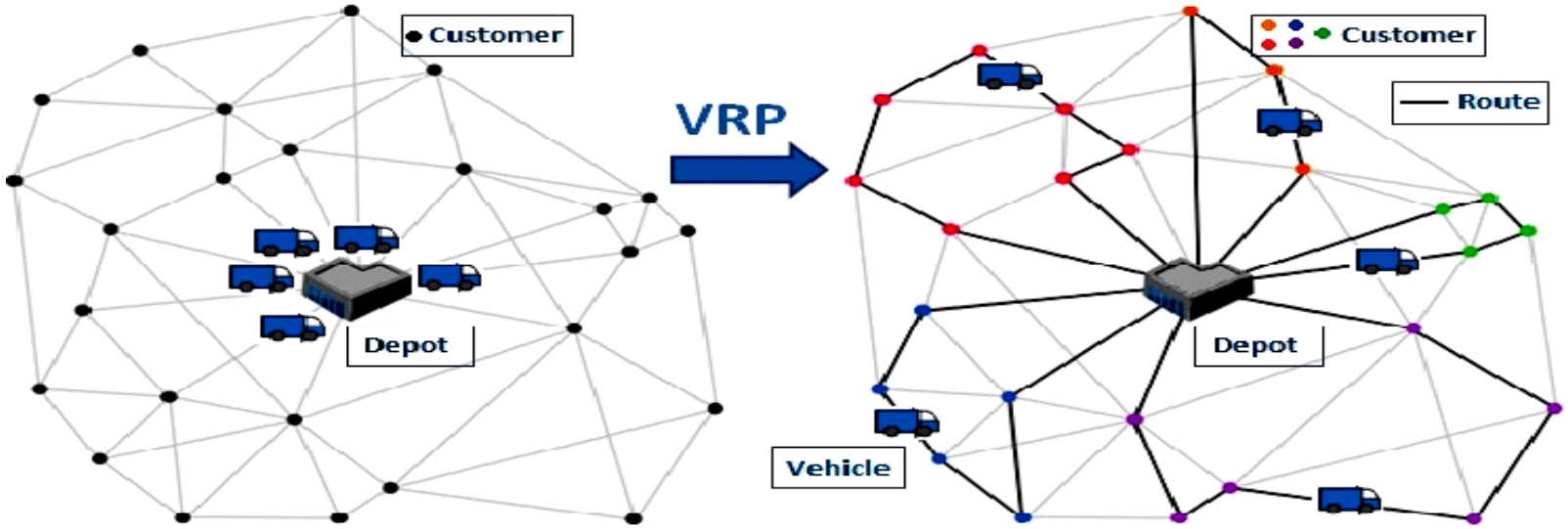
Why is CO Useful?

- CO is the engine behind modern decision-making in nearly every industry. It provides a scientific framework for making the most of limited resources

- Core Impact:
  - Drives Efficiency
  - Reduces Costs
  - Increases Revenue
  - Manages Complexity
  - Automates Decision-Making

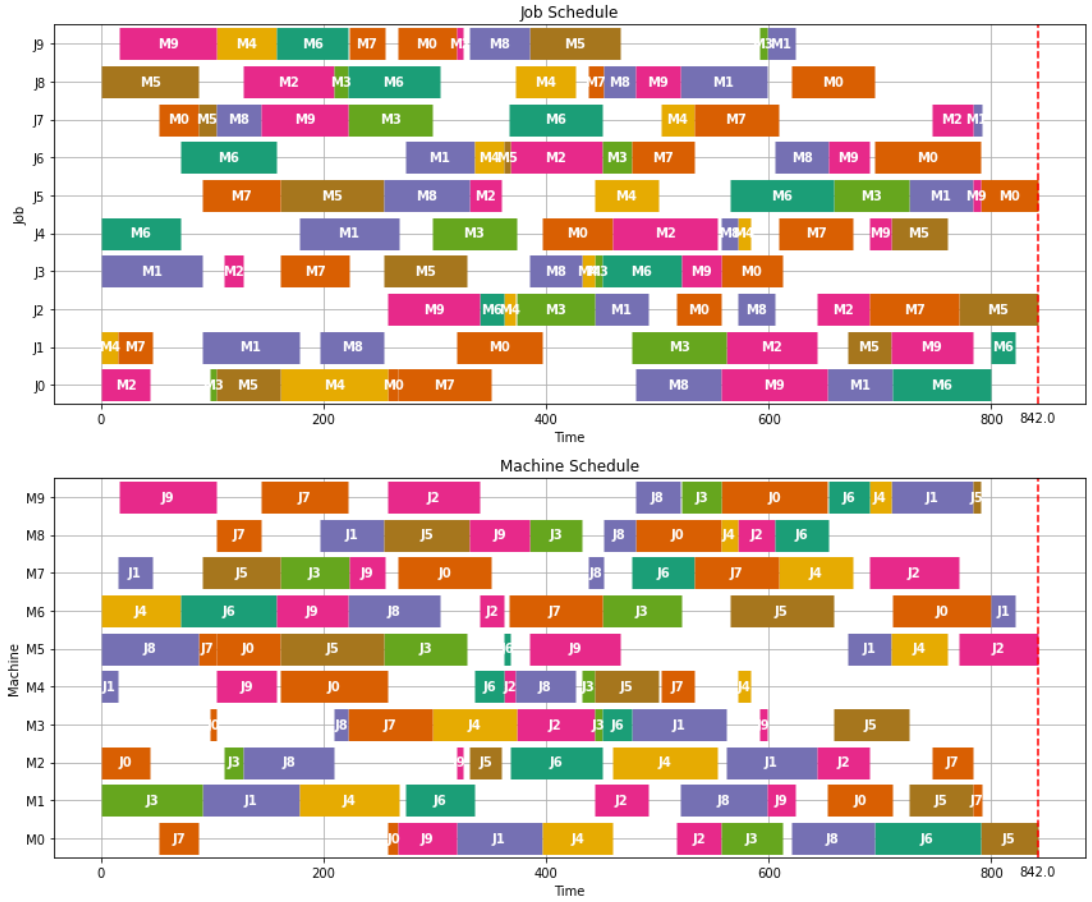
Application (1): Logistics & Supply Chain

- Vehicle Routing Problem (VRP):
  - Problem: For a fleet of delivery vehicles, find the optimal set of routes to serve customers, minimizing total distance or cost
  - Impact: Billions of dollars saved annually by companies like FedEx, UPS, SF-express, CaiNiao, JingDong and Amazon in fuel and labor costs



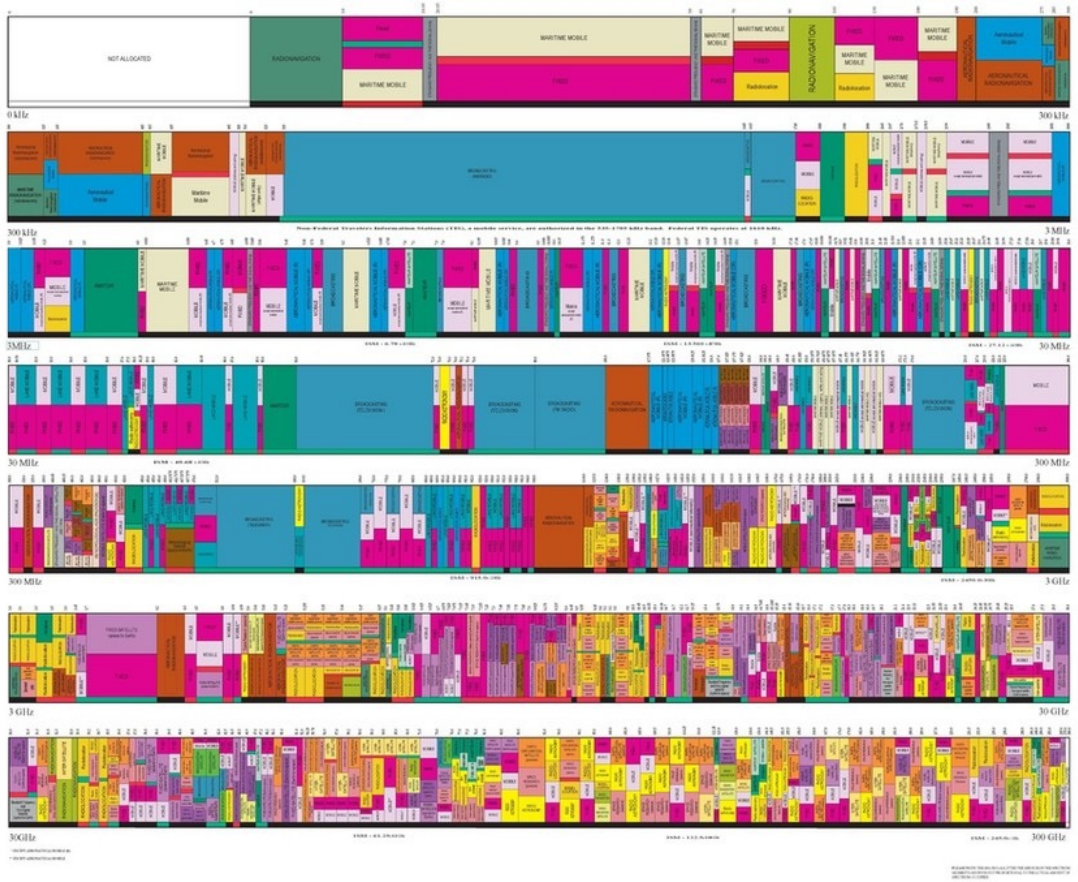
Application (2): Manufacturing

- Job Shop Scheduling:
  - Problem: Given a set of jobs, each with a sequence of operations on different machines, find a schedule that minimizes makespan (total time to complete all jobs)
  - Impact: Maximizes factory throughput, reduces idle time in nearly all manufacturing factories (BYD, DJI, Huawei, etc.)



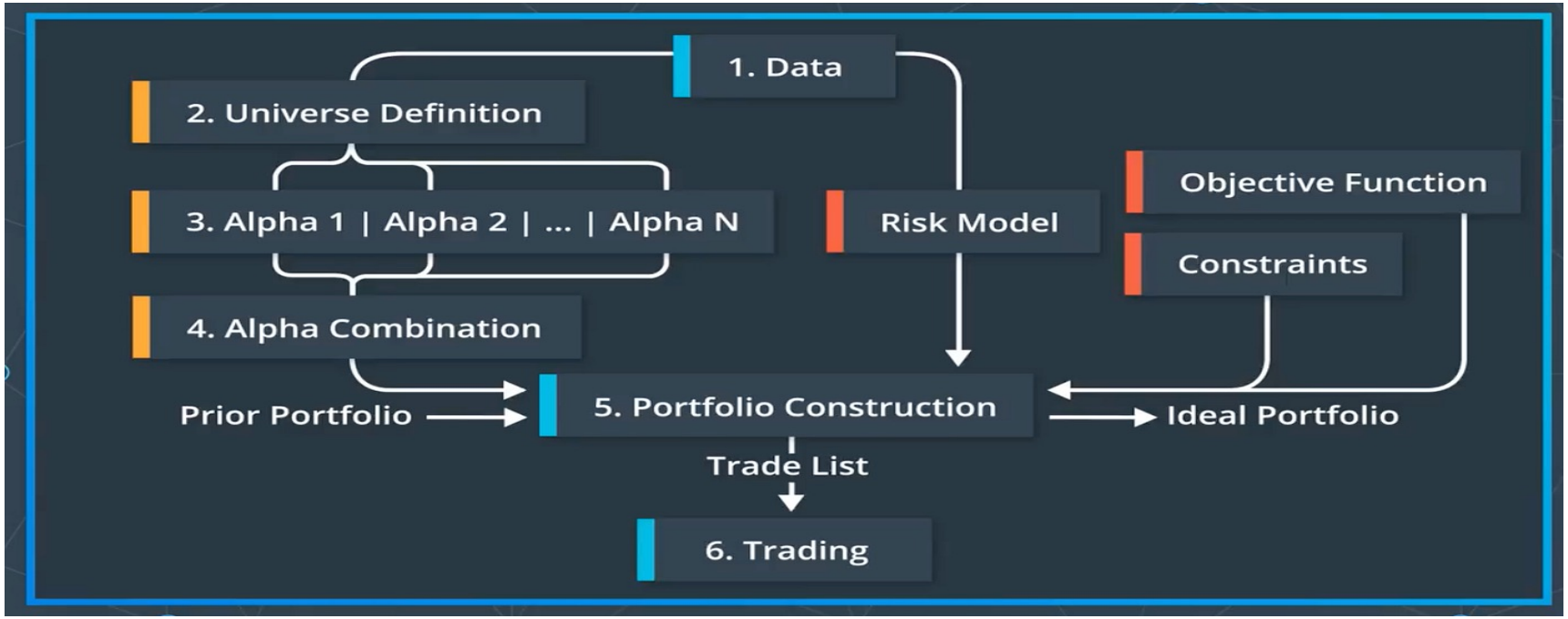
Application (3): Telecommunications

- Frequency Allocation:
  - Problem: Assign radio frequencies to transmitters (e.g., cell towers) to avoid interference and maximize network capacity.
  - Impact: Ensures the quality and reliability of wireless communications by governments



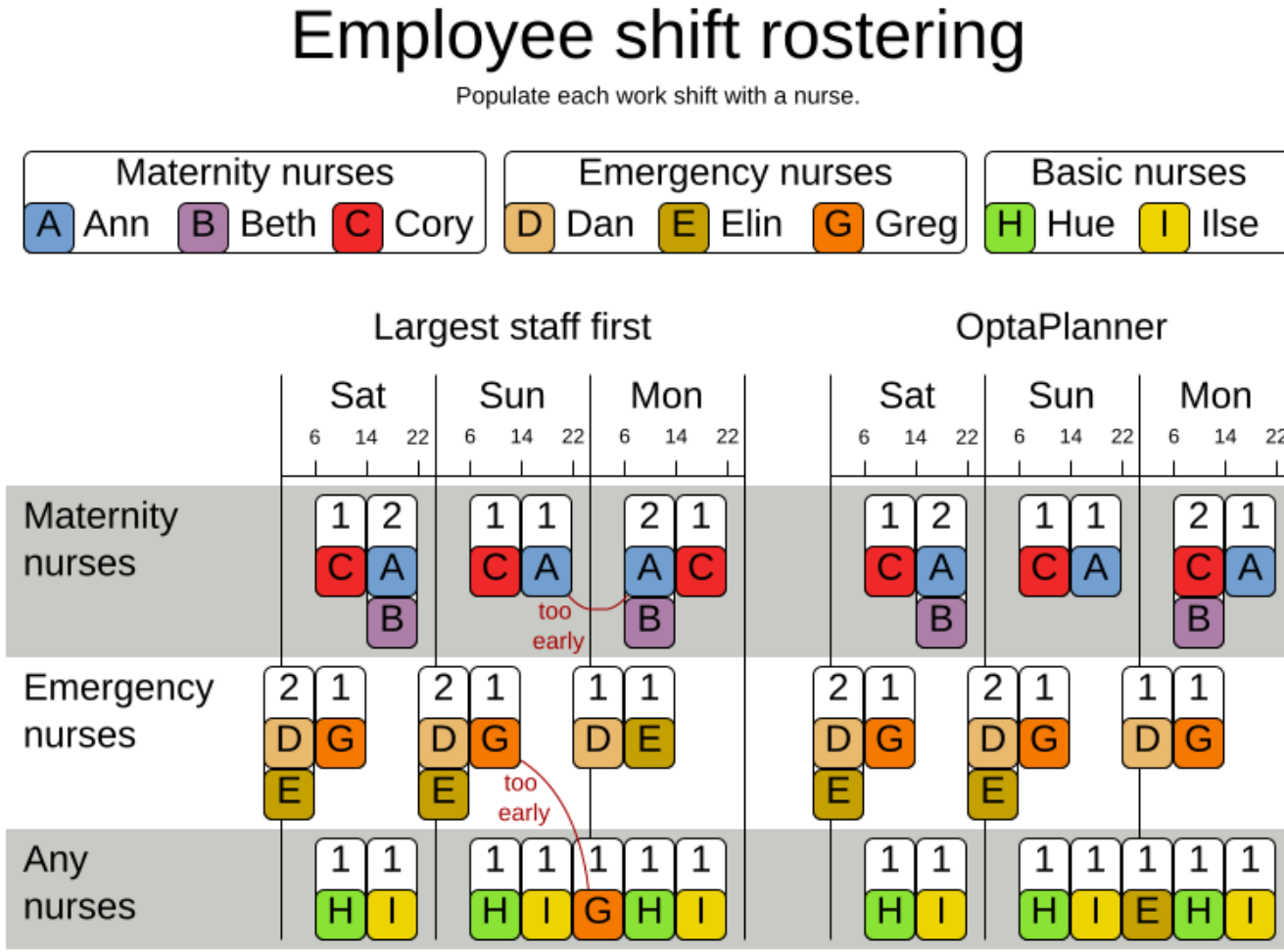
Application (4): Finance

- Portfolio Optimization:
  - Problem: Select a combination of assets (stocks, bonds) to maximize expected return for a given level of risk (variance)
  - Impact: The foundation of modern investment management, e.g., quant trading



Application (5): Healthcare

- Nurse Rostering:
  - Problem: Create a weekly or monthly schedule for nurses that covers all required shifts while respecting nurse preferences, skill levels, and labor laws.
  - Impact: ensures patient safety, improves staff satisfaction, and reduces overtime costs.

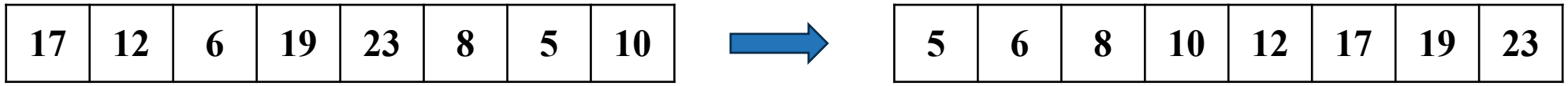


The Challenge of CO: Combinatorial Explosion

- The number of possible solutions in a CO problem can grow astronomically with the problem size
- This is called “combinatorial explosion”

- Consider the TSP: Find the shortest possible route that visits each city and returns to the origin city.
- The number of possible tours for n cities is $(n-1)!$
- It grows even faster than k^n for a given k

So, is Sorting a CO Problem?



- Can we frame sorting as a CO problem? Yes.
- CO Formulation of Sorting:
  - Decision Variables: The ordering (permutation) of the elements in the list
  - Feasible Set S: The set of all possible permutations of the input list
  - Objective Function $f(x)$: Minimize the number of "inversions" (pairs of elements that are in the wrong order). The optimal solution will have 0 inversions

Why Sorting is not Our Focus: "Easy" Problem

- Although sorting fits the CO definition, it is computationally “easy”
- We have very efficient algorithms that are guaranteed to find the optimal solution (the sorted list) in polynomial time
  - For example, Merge Sort and Heap Sort have a worst-case time complexity of $O(n\log n)$
- Problems that can be solved in polynomial time are considered “tractable” and belong to the complexity class ***P***
- This course will not focus on these “easy” problems

The Scope of Our Course: "Hard" Problems

- Our focus is on the vast and challenging world of NP-hard problems
- For NP-hard problems, no polynomial-time algorithm are known to find the optimal solutions
- This includes almost all the famous applications we have discussed and will discuss in this course:
  - Knapsack Problem (KP), VRP, TSP, Job Shop Scheduling, etc.

- The core challenge is that the computation time for finding the exact optimal solution grows al least exponentially with the problem size

Polynomial vs. Exponential Growth

- Why does this distinction matter so much?
- Let's assume a computer can do 1 billion operations per second.

Problem Size (n)	Polynomial (n^3)	Exponential (2^n)	**Factorial ((n-1)!)**
10	1,000 ops (1 µs)	1,024 ops (1 µs)	**362,880 ops (0.36 ms)**
20	8,000 ops (8 µs)	~1 million ops (1 ms)	**~1.2 x 10^{17} ops (~3,857 years)**
30	27,000 ops (27 µs)	~1 billion ops (1 sec)	**~2.4 x 10^{30} ops (~7.7 x 10^{13} years)** *(约77万亿年)*
50	125,000 ops (0.1ms)	**~35.7 years**	**~6.1 x 10^{62} ops (远超宇宙年龄)**
60	216,000 ops (0.2ms)	**~36,559 years**	**~1.4 x 10^{80} ops (远超宇宙年龄)**

Solution Set size is not equal to complexity

- A common misunderstanding

  ✖Solution set size grows exponentially -> The problem is NP-hard

- A counterexample is sorting problem, whose solution set grows super-exponentially

- A problem's hardness is not determined by the number of possible solutions, but by the structural properties of its solution space. This "intrinsic complexity" refers to whether an efficient, guiding principle exists to navigate that space.

The Knapsack Problem (KP)

- You have a bag (knapsack) with a limited weight capacity (e.g., it can only hold 15 kg).
- You find several items of value: a painting, a laptop, jewelry, silverware, etc.
- Each item has a weight and a monetary value.
- You must choose which items to take to maximize the total value without exceeding your knapsack's weight limit.

Formal Definition of KP

- Given
  - A knapsack with a maximum capacity of W
  - A set of n items, where for each item i:
    - it has a weight w_i
    - it has a value v_i

- Objective
  - Select a subset of the items to put into the knapsack such that the sum of their weights does not exceed the capacity W, and the sum of their values is maximized

Variation 1: The 0/1 KP

- This is the most common version
- For each item, you have a binary choice:
  - either you take it (1)
  - or you leave it (0)
- You cannot take a fraction of an item
- You can't take half a laptop or saw off a piece of a painting
- This is the version we will focus on most

0/1 KP: Mathematical Formulation

Decision Variables:

For each item $i = 1, \ldots, n$, we define a binary variable $x_i \in \{0,1\}$:

$$x_i = \begin{cases} 1 & \text{if } i\text{tem } i \text{ is selected} \\ 0 & \text{if } i\text{tem } i \text{ is not selected} \end{cases}$$

Objective Function (Maximize Total Value):

$$\max \sum_{i=1}^{n} v_i x_i$$

Constraint (Respect Weight Capacity):

$$\sum_{i=1}^{n} w_i x_i \leq W$$

Application of 0/1 KP: Capital Budgeting

- The 0/1 KP is a powerful tool for financial decision-making
- **Scenario**: A company has a fixed investment budget of W for the next year. There are n potential projects it can invest in.
- **Model Mapping**:
  - Knapsack Capacity W: The total investment budget
  - Item i: Potential project i
  - Weight w_i: The capital required to undertake project i
  - Value v_i : The expected return from project i
  - Objective: Select the portfolio of projects that maximizes the total expected return without exceeding the budget

Variation 2: The Fractional KP (分数背包问题)

- In this version, items are divisible
- You can take any fraction of an item.
- Analogy: The "items" are piles of gold dust, barrels of oil, or other liquids. You can take 3.7 kg of gold dust if you wish.

- **Mathematical Formulation**: The only change is the variable domain.

$$x_i \in [0,1], \qquad \text{for } i = 1, \ldots, n$$

- This version is significantly easier to solve (as we'll see later).

Variation 3: The Unbounded KP (无界/完全背包问题)

- In this version, there is an unlimited supply of each type of item. You can take as many copies of an item as you can fit.

- **Mathematical Formulation**: The decision variables are non-negative integers

$$x_i \in \{0,1,2,\dots\}, \qquad \text{for } i = 1,\dots,n$$

Application of Unbounded KP: Cutting Stock

- **Scenario**: You have a supply of standard-sized raw material rolls (e.g., steel beams of length W). You need to cut these rolls to maximize profit, given n different types of customer orders for smaller lengths (w_i).
- **Model Mapping:**
  - Knapsack Capacity W : The length of the raw material roll.
  - Item i: An order for a piece of length w_i.
  - Weight w_i : The length of the piece to be cut.
  - Value v_i : The sale price of the cut piece
  - Objective: Determine the combination of cuts that maximizes the value obtained from a single raw material roll, which is equivalent to minimizing wasted material.

A Simple 0/1 KP Example

- Knapsack Capacity $W = 10$ kg
- Available items:

Item	Weight (w_i)	Value (v_i)
1: Water	3 kg	$10
2: Food	4 kg	$40
3: Medkit	5 kg	$30
4: Camera	6 kg	$50

- Total number of subsets is $2^4 = 16$. Let's test a few feasible ones.
- **Combo A:** {Food, Camera}
  - Total Weight = 4 + 6 = 10 kg (Feasible)
  - Total Value = 40 + 50 = **$90**
- **Combo B:** {Water, Medkit}
  - Total Weight = 3 + 5 = 8 kg (Feasible)
  - Total Value = 10 + 30 = $40
- **Combo C:** {Water, Food}
  - Total Weight = 3 + 4 = 7 kg (Feasible)
  - Total Value = 10 + 40 = $50
- **Combo D:** {Water, Camera}
  - Total Weight = 3 + 6 = 9 kg (Feasible)
  - Total Value = 10 + 50 = $60
- **Combo E:** {Food, Medkit}
  - Total Weight = 4 + 5 = 9 kg (Feasible)
  - Total Value = 40 + 30 = $70

Combo A with a value of $90 is the best.
But for a larger number of items, this manual check is impossible

The Challenge of Solving the KP

- The 0/1 KP is NP-hard
- The Unbounded KP is also NP-hard. **Why?**
- The solution space (solution set size) grows exponentially at a rate of 2^n
- For n=30 items, there are 230≈1.07 billion combinations.
- For n=60 items, there are 260≈1.15×1018 combinations.
- We need algorithms that are much smarter than brute-force enumeration.

A Spectrum of Solution Methods

- **First principles for CO**
  - There is no “one-size-fits-all” algorithm for NP-hard problems
  - The choice depends on the trade-off between solution quality and computation time

- We will explore this spectrum using our KP case study:
  - Greedy Algorithm
  - Approximation Algorithms
  - Dynamic Programming
  - Branch and Bound

Method 1: The Greedy Algorithm

- Core Idea: Make the “best” local choice at each step, hoping it will lead to a globally optimal solution.
- It is a myopic approach.

- What is the “best” choice for the KP?
  - Highest Value First? (Might be too heavy and fill the bag for little value elsewhere).
  - Lowest Weight First? (Might fill the bag with light, low-value items).
  - Highest Value-to-Weight Ratio First! This is the most intuitive metric.

The Greedy Strategy: Maximum Density First

Algorithm Steps:

1. For each item i, calculate its value density: $d_i = v_i/w_i$

2. Sort all items in descending order of their density.

3. Iterate through the sorted items. For each item, if it fits into the remaining capacity of the knapsack, add it.

4. Stop when no more items can be added.

Greedy Algorithm on the Fractional KP

The greedy strategy for the Fractional KP

Example: $W = 20$

Item	Weight	Value	Density (v/w)
A	18	25	1.39
B	15	24	1.60
C	10	15	1.50

1. **Sorted by density:** B > C > A.
2. **Take all of Item B.** Remaining capacity: $20 - 15 = 5$. Current value: 24.
3. **Take Item C.** Remaining capacity is 5, but C's weight is 10. So, we take $5/10 = 0.5$ of item C.
   - Weight added: $10 \times 0.5 = 5$.
   - Value added: $15 \times 0.5 = 7.5$.
4. Knapsack is full. **Final optimal value =** $24 + 7.5 = 31.5$.

Greedy is provably optimal in this case. (反证法可以证明)

The Pitfall: Greedy on the 0/1 KP

The same logic often fails for the 0/1 KP because we lose the flexibility of taking fractions.

Counter example: $W = 50$

Item	Weight	Value	Density (v/w)
A	10	60	6.0
B	20	100	5.0
C	30	120	4.0

1. **Greedy by density:**
    1. **Take Item A** (w=10, v=60). Remaining capacity: 40. Total value: 60.
    2. **Take Item B** (w=20, v=100). Remaining capacity: 20. Total value: 160.
    3. **Try Item C** (w=30). Does not fit.

- **Greedy Solution:** {A, B}, Total Value = **160**.
- **Optimal Solution:** {B, C}, Total Weight = 20+30=50, Total Value = 100+120=**220**.
- **Conclusion:** The greedy choice for item A blocked the path to the true optimal solution.

Analysis of the Greedy Algorithm

- Time Complexity:
  - Calculating densities for n items: *O(n)*
  - Sorting the items by density: *O(nlogn)*
  - Iterating through the sorted items: *O(n)*
- Total Time Complexity: *O(nlogn)*, dominated by the sorting step

- **Pros**: Very fast and simple to implement. Optimal for the fractional version.
- **Cons**: Not guaranteed to be optimal for the 0/1 KP or unbounded KP. Can produce arbitrarily bad solutions in some cases (In what case?)

Method 2: Approximation Algorithms

- Core Idea: If finding the optimal solution is too slow, can we find a provably good solution quickly? An approximation algorithm is an algorithm that runs in polynomial time and finds a solution whose objective value is within a certain factor of the optimal value.
- This factor is known as the approximation ratio (近似比) or performance guarantee (性能保证)
- For a maximization problem, an algorithm is a ρ-approximation algorithm (for $\rho \geq 1$) if:

$$\frac{\text{Optimal Value}}{\text{Algorithm's Value}} \leq \rho$$

A Simple 2-Approximation for 0/1 Knapsack

- We can slightly modify our greedy approach to get a performance guarantee

Algorithm Steps:

1. Run the "density-first" greedy algorithm. Let the solution be S_G with total value V_G
2. Find the single most valuable item that fits in the knapsack. Let this solution be S'_G with value V'_G
3. The final result is the better of the two solutions: $\max(V_G, V'_G)$

- This is a 2-approximation algorithm (will be proved later in this course)
- This gives us a powerful trade-off: we sacrifice guaranteed optimality for speed and a worst-case quality bound.

A Note on FPTAS

- For KP, we can do even better than a constant-factor approximation.
  - A Fully Polynomial-Time Approximation Scheme (完全多项式时间近似方案, FPTAS) is an algorithm that can achieve a $(1+\epsilon)$ −approximation for any given $\epsilon > 0$
  - Its running time is polynomial in both n and $1/\epsilon$
- We can get arbitrarily close to the optimal solution (e.g., within 1% or 0.01%) in polynomial time, although the runtime increases as our desired precision increases (i.e., as ϵ gets smaller)
- This makes the KP seem "less hard" than other NP-hard problems like TSP, for which no FPTAS is believed to exist.

Method 3: Dynamic Programming (DP)

- Core Idea: Break down a complex problem into a collection of simpler, overlapping subproblems. Solve each subproblem only once and store its solution. Combine the solutions to subproblems to solve the original problem.

- DP is applicable to problems exhibiting:
  - Optimal Substructure: The optimal solution to the overall problem can be constructed from the optimal solutions of its subproblems.
  - Overlapping Subproblems: The same subproblems are encountered and solved many times during a recursive formulation.

The Principle of Optimality in KP

- Let's consider the optimal solution for n items and capacity W.
  - Case 1: Item n is not in the optimal solution. Then the optimal solution must be the optimal solution for the first $n-1$ items with the same capacity W.
  - Case 2: Item n is in the optimal solution. Then the optimal solution's value is v_n plus the optimal solution for the first $n-1$ items with a reduced capacity of $W - w_n$.
- This demonstrates optimal substructure: The overall optimal solution depends on the optimal solutions to smaller KPs.

Defining the DP State and Recurrence

- To implement DP, we need to define the state of our subproblems.
- Let `dp[i][j]` be the **maximum value achievable using only the first `i` items with a knapsack capacity of `j`**.
- Our goal is to compute `dp[n][W]`.
- The state transition (recurrence relation) is based on the principle of optimality:
  - If we don't take item `i` (or if it doesn't fit, `j < w_i`):
    - `dp[i][j] = dp[i-1][j]`
  - If we do take item `i` (only possible if `j >= w_i`):
    - The value is `v_i + dp[i-1][j - w_i]`
- **The full recurrence is:**

$$dp[i][j] = \begin{cases} dp[i-1][j] & \text{if } w_i > j \\ \max(dp[i-1][j], \quad v_i + dp[i-1][j-w_i]) & \text{if } w_i \leq j \end{cases}$$

DP: Example

- Capacity $W = 5$
- Items:
  - Item 1: $w\_1 = 2, v\_1 = 3$
  - Item 2: $w\_2 = 3, v\_2 = 4$
  - Item 3: $w\_3 = 4, v\_3 = 5$
- We will fill a table of size $(n + 1) \times (W + 1)$.
- Base cases: `dp[0][j] = 0` for all `j` (no items, zero value). `dp[i][0] = 0` for all `i` (zero capacity, zero value).

`dp[i][j]` Table (i: item, j: capacity)

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						

DP: Example – cont'd

- For each capacity `j`, we decide whether to include Item 1.
- `dp[1][j] = max(dp[0][j], v_1 + dp[0][j - w_1])`

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	**3**	**3**	**3**	**3**
2						
3						

- `dp[1][2]`: `max(dp[0][2], 3 + dp[0][0]) = max(0, 3) = 3`
- `dp[1][3]`: `max(dp[0][3], 3 + dp[0][1]) = max(0, 3) = 3`
- ...and so on.

DP: Example – cont'd

- `dp[2][j] = max(dp[1][j], v_2 + dp[1][j - w_2])`

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	**4**	**4**	**7**
3						

- `dp[2][3]`: `max(dp[1][3], 4 + dp[1][0]) = max(3, 4) = 4`
- `dp[2][4]`: `max(dp[1][4], 4 + dp[1][1]) = max(3, 4) = 4`
- `dp[2][5]`: `max(dp[1][5], 4 + dp[1][2]) = max(3, 4+3) = 7`

DP: Example – cont'd

- `dp[3][j] = max(dp[2][j], v_3 + dp[2][j - w_3])`

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	**5**	**7**

- `dp[3][4]`: `max(dp[2][4], 5 + dp[2][0]) = max(4, 5) = 5`
- `dp[3][5]`: `max(dp[2][5], 5 + dp[2][1]) = max(7, 5) = 7`
- **Final Answer:** The maximum value is `dp[3][5] = 7`.

Finding the Items: Backtracking

- The table gives us the optimal *value*, but which items did we pick? We need to backtrack from the final cell `dp[n][W]`.

1. Start at `dp[3][5]=7`. Compare it to `dp[2][5]=7`. They are equal.
   - This means the value came from the `dp[i-1][j]` case. **Item 3 was NOT taken.**
   - We remain at capacity `j=5` and move up to `dp[2][5]`.
2. Now at `dp[2][5]=7`. Compare it to `dp[1][5]=3`. They are not equal.
   - This means the value came from the `v_2 + dp[1][j-w_2]` case. **Item 2 WAS taken.**
   - We reduce capacity `j = 5 - w_2 = 5 - 3 = 2` and move up to `dp[1][2]`.
3. Now at `dp[1][2]=3`. Compare it to `dp[0][2]=0`. They are not equal.
   - This means **Item 1 WAS taken.**
   - We reduce capacity `j = 2 - w_1 = 2 - 2 = 0`. The process terminates.

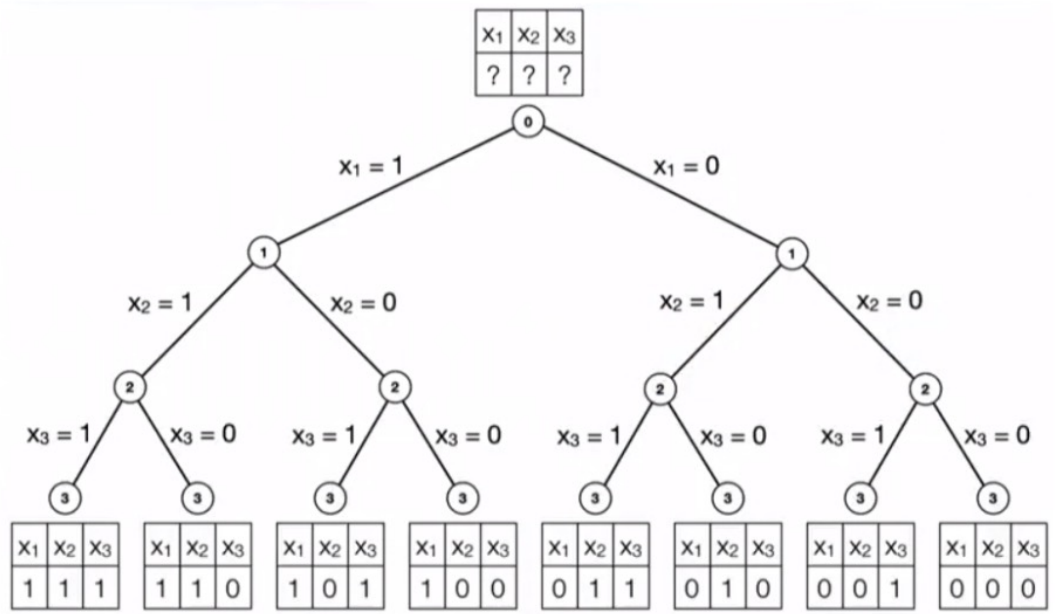
- **Optimal Set:** {Item 1, Item 2}.

Analysis of the DP Algorithm

- Pros:
  - Guarantees finding the globally optimal solution
  - Relatively straightforward to implement
- Cons:
  - Time Complexity: We fill an $n \times W$ table, and each cell takes constant time. So, the complexity is $O(nW)$
  - Pseudo-polynomial Time: The complexity depends on the magnitude of an input number (W), not just the length of the input.
  - True exponential time: $O(n2^{length(W)})$

Method 4: Branch and Bound (B&B)

- Core Idea: A clever, structured way of exploring the search space, which is visualized as a tree. It prunes large parts of the tree that cannot possibly contain the optimal solution.
- It is an exact algorithm.



An Integer Program Example of KP

Three items, $W = 10$

- Item 0, $v_0 = 45,\ w_0 = 5$
- Item 1, $v_1 = 48,\ w_1 = 8$
- Item 2, $v_2 = 35,\ w_2 = 3$

Integer linear program (ILP):

$$\max \quad 45x_0 + 48x_1 + 35x_2$$

$$\text{s.t.} \quad 5x_0 + 8x_1 + 3x_2 \leq 10$$

$$x_i \in \{0, 1\}$$

Branch and Bound (B&B)

- **Branching**: split the problem into a number of subproblems (like DP)
  - For 0/1 KP, this is done by picking an item and creating two branches: one where the item is **included** in the knapsack, and one where it is **excluded**.
- **Bounding**: find an optimistic estimate of the best solution of the subproblem (upper/lower bound for maximization/minimization)
  - At each node in the search tree (representing a subproblem), we calculate an **upper bound** on the best possible solution that can be obtained from this node downwards
  - How to get upper bound? **Relaxation**: Solve the relaxed problem.
- Pruning:
  - Keep track of the best feasible integer solution found so far globally (the incumbent), which gives us a **lower bound**. If a node's upper bound is less than or equal to the current lower bound, prune that node and its entire subtree

Relaxation for KP, version 1

What can we relax?

$$\max \quad 45x_0 + 48x_1 + 35x_2$$

$$\text{s.t.} \quad \cancel{5x_0 + 8x_1 + 3x_2 \leq 10}$$

$$x_i \in \{0, 1\}$$

Relax capacity constraint!

Pack everything ignoring capacity

Relaxation for KP, version 1

What can we relax?

max $45x_0 + 48x_1 + 35x_2$

s.t. ~~$5x_0 + 8x_1 + 3x_2 \leq 10$~~

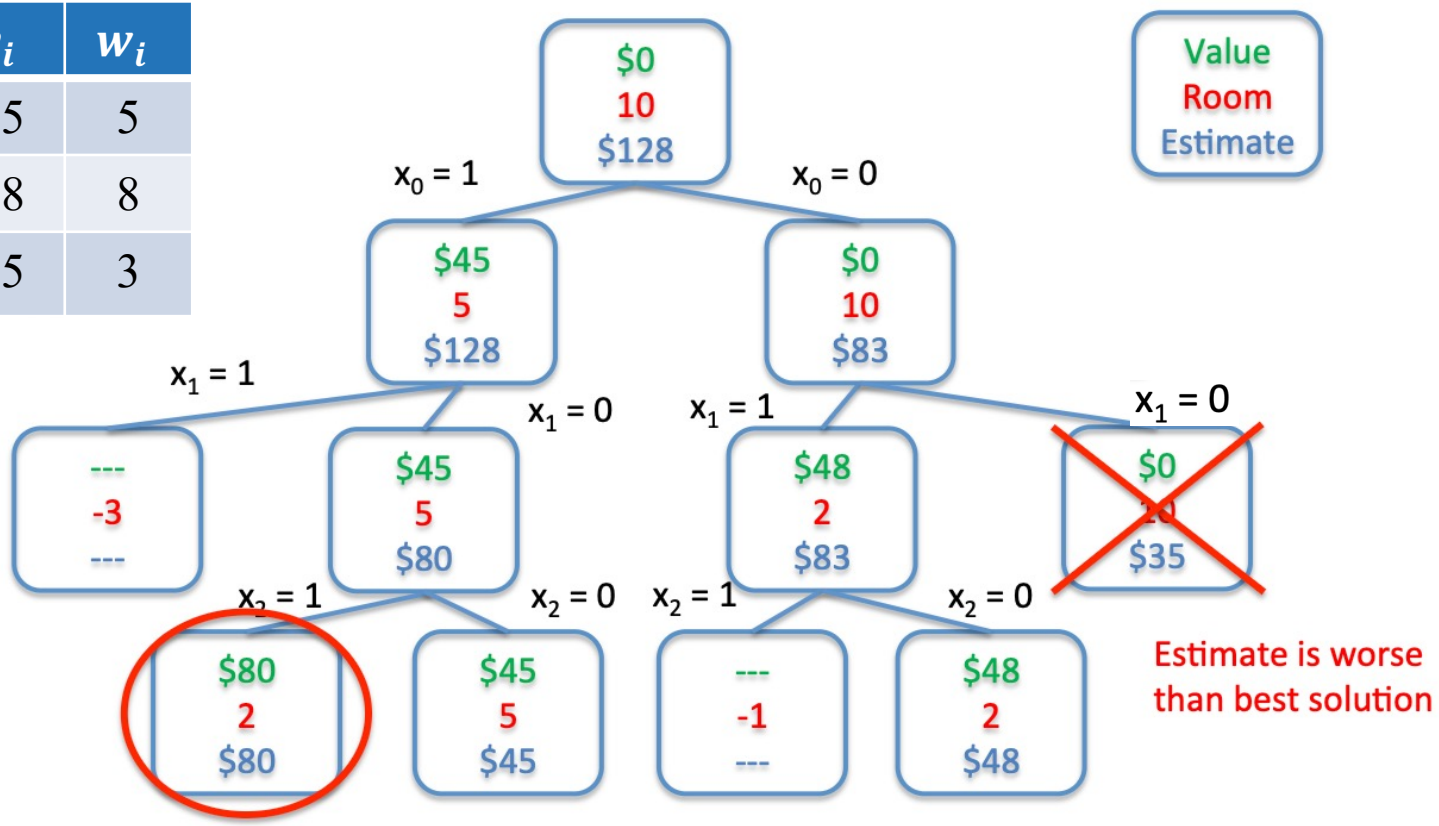
$x_i \in \{0, 1\}$

Relax capacity constraint!

Pack everything ignoring capacity

$W = 10$

i	v_i	w_i
0	45	5
1	48	8
2	35	3



Relaxation for KP, version 2

What else can we relax?

max $45x_0 + 48x_1 + 35x_2$

s.t. $5x_0 + 8x_1 + 3x_2 \le 10$

~~$x_i \in \{0, 1\}$~~ → $x_i \in [0, 1]$

Relax integrality constraint!

Imagine that items are possible to cut in pieces

How do we solve the relaxation?

max $45x_0 + 48x_1 + 35x_2$

s.t. $5x_0 + 8x_1 + 3x_2 \le 10$

$x_i \in [0, 1]$

$v_0 / s_0 = 9$; $v_1 / s_1 = 6$; $v_2 / s_2 = 11.7$

select items 2 and 0

Select ¼ of item 1

Estimation: 92 (version 1: 128; optimal value: 80)

Denote $y_i = x_i v_i$

$$\max \sum_i y_i$$

$$\text{s.t.} \sum_i y_i \frac{s_i}{v_i} \le S$$

$$y_i \le v_i$$

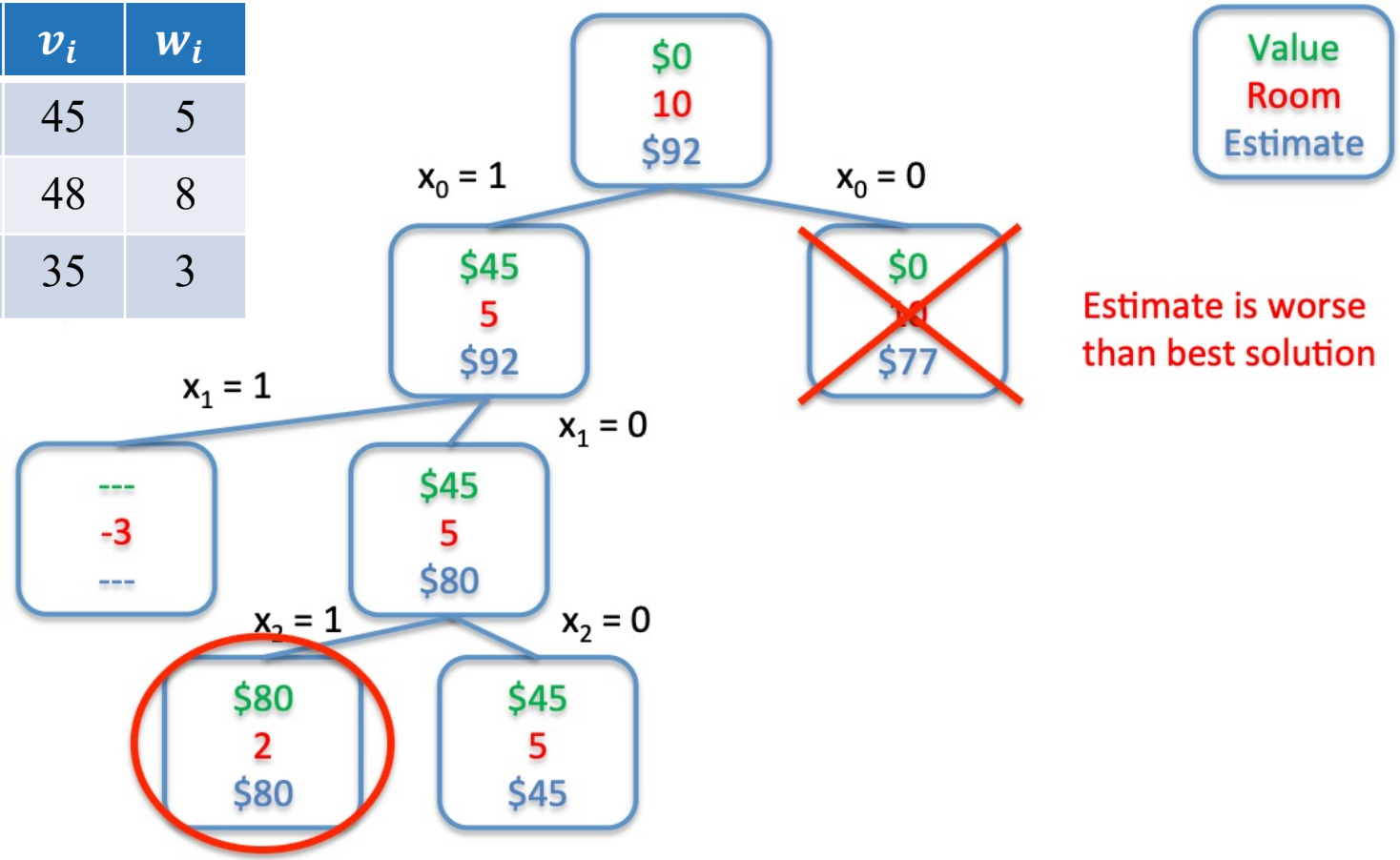
Sort s_i / v_i increasing

Take as much as possible

Relaxation for KP, version 2

$W = 10$

i	v_i	w_i
0	45	5
1	48	8
2	35	3



Analysis of Branch and Bound

- **Pros**:
  - It is an exact algorithm, guaranteed to find the optimal solution.
  - Can be much more efficient than brute force by pruning the search space
  - Its performance is not directly tied to the magnitude of W, so it can be better than DP for problems with very large capacities
- **Cons**:
  - Worst-case complexity is still exponential. The performance heavily depends on the quality of the bounding function and the order of branching
  - Can be more complex to implement than DP

Comparison of Solution Methods for KP

Algorithm Method	Optimality Guarantee	Typical Time Complexity	Key Idea / Best For
Brute Force	Optimal	Exponential ($O(2^n)$)	Simple, but only feasible for tiny problems.
Greedy (by density)	Not for 0/1	Fast ($O(n \log n)$)	Getting a quick, reasonable solution. Optimal for Fractional KP.
Approximation Algorithm	Provably near-optimal	Fast (Polynomial)	When a "good enough" solution is needed quickly with a guarantee.
Dynamic Programming	Optimal	Pseudo-poly ($O(nW)$)	Problems with small-to-moderate integer capacity W.
Branch and Bound	Optimal	Exponential (worst-case)	When exact solutions are needed, especially if W is very large.

Multi-dimensional Knapsack Problem (MKP)

- A more realistic variant where there are multiple constraints.
- Example: An astronaut's pack has limits on both weight and volume.
- Mathematical Model:

$$\max \sum_{i=1}^{n} v_i x_i$$

$$\text{s.t.} \quad \sum_{i=1}^{n} w_{i,k} x_i \leq W_k, \quad \text{for each constraint } k = 1, \ldots, m$$

$$x_i \in \{0, 1\}$$

- MKP is significantly harder to solve than the standard KP

Heuristics and Metaheuristics

- When problems like MKP are too complex even for B&B, we often turn to heuristics and metaheuristics.
- **Heuristic**: A rule-of-thumb or a strategy that leads to a solution, but without a formal proof of quality (e.g., the density-first greedy approach).
- **Metaheuristic**: A high-level framework for building and guiding heuristics to escape local optima and find better solutions. They are problem-agnostic.
- Famous Examples:
  - Simulated Annealing: Mimics the process of cooling metal to reach a low-energy state.
  - Genetic Algorithms: Uses concepts of evolution like selection, crossover, and mutation.
- These will be covered in the late stage during this course

Take-away Messages

- CO has vast applications that drive the modern economy.
- This course focus on NP-hard problems where brute-force is not an option, demanding clever algorithms.
- The KP is a fundamental NP-hard problem, serving as an useful model for resource allocation.
- There is a rich diversity of algorithms for solving CO problems. The choice involves a critical trade-off between solution quality and computation time.
- We have explored the core ideas behind Greedy, DP, B&B, and Approximation Algorithms. These concepts are foundational for the rest of the course.