Week 3 - exact sequential decision process (SDP) methods for combinatorial optimization, Dynamic Programming (DP)

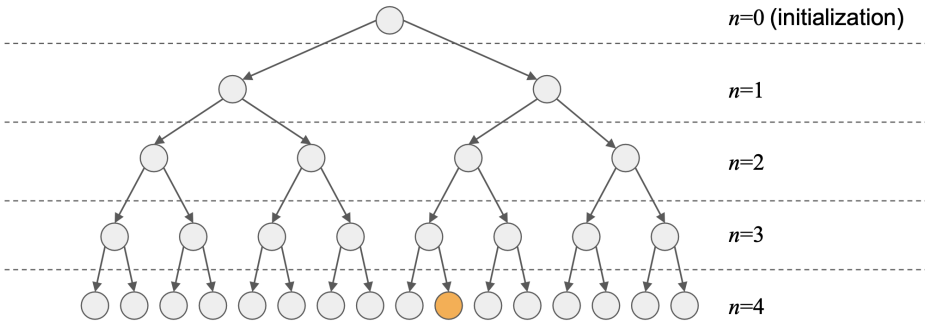
1. exact sequential decision process (SDP) methods for combinatorial optimization
    a. Strategies for constructing exact methods:
        i. Divide and conquer
        ii. Branch and bound
        iii. Dynamic Programming
    b. Sequential decision process (SDP)
        i. Definition
            1. a **recursive** process that scans *N* input data items $x_1, \ldots, x_N$ in sequence, generating new candidate configurations by extending the existing configurations using each input data item in turn
            2. On each iteration, it reduces the set of candidate configurations by removing any that cannot be ultimately extended to an optimal configuration
            3. Finally, it selects an optimal configuration from the remaining candidates
        ii. computational configuration graph
            1. brute-force (a full tree)
            2. greedy (a tree with a single optimal branch at each stage)
            3. dynamic programming (an incomplete tree)
        iii. Pseudocode

```
S = []
for n in range(N):
        # using input data item x_n,
        S.append(configurations) # extend all candidate

        S.remove(configurations) # Remove any cons that

Select an optimal configuration X^* in S
```

iv. Brute-force

1. **Exhaustive (brute-force)** SDP algorithms have no means of applying reduction

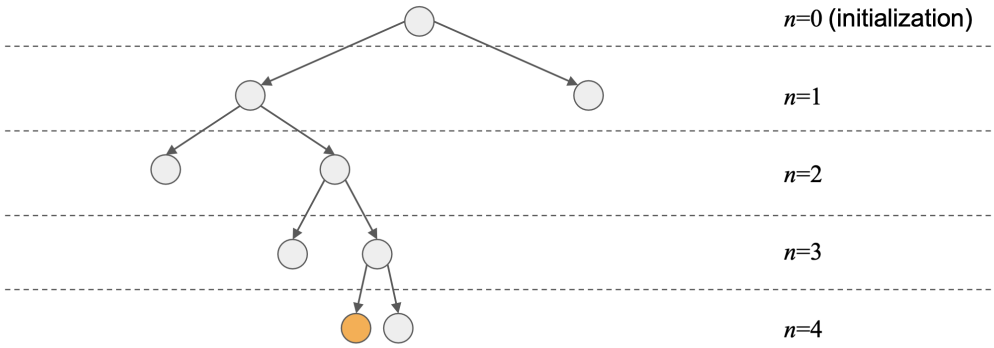
SDP exact: typical exhaustive computation graph



v. Greedy

1. Select the optimal configuration at each stage
2. Complexity: $O(N^2)$
3. Example: insertion sort

SDP exact: typical greedy computation graph



2. Dynamic Programming (DP) - belongs to sequential decision process (SDP)
    a. Definition
        i. Break problems into subproblems and combine their solutions into solutions to larger problems (and not necessarily use recursion)
        ii. It holds the best solutions to a sub problem in a table (memoization), can be used without recomputing
        iii. In contrast to divide-and-conquer, there may be relationships across subproblems
    b. Memoization
        i. **if the same (or similar) problem has to be solved repeatedly, then, if we have previously found the optimal solution x for some part N of the configuration set X , we can store this optimal solution x in memory and then next time when we are asked to find the optimal solution in N , we can solve it in O (1) steps by simply retrieving the stored solution from memory**
    c. **Principle of Optimality:** This principle applies to dynamic programming problems.

> **An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.** (See Bellman, 1957, Chap. III.3.)[10][11][13]

It states that an optimal solution to a problem can be constructed from optimal solutions to its subproblems. In simpler terms, to find the best overall solution, you can **break the problem into smaller parts and find the best solutions for those parts.** You can **then combine these sub-optimal solutions in an optimal way to get the best overall solution**.

d. is used to solve a wide variety of discrete optimization problems
   i. scheduling
   ii. string-editing
   iii. packaging
   iv. inventory management
e. Pseudocode

```
1. Define subproblems
2. Write down the recurrence that relates subproblems
3. Recognize and solve base cases
```

f. For example, Fibonacci Sequence → top-down approach

$$F_0 = 0, F_1 = 1$$
$$For\ n > 1,\ F_n = F_{n-1} + F_{n-2}$$

Let's add memoization

```
def fibonacci(n):
    if n == 0:
        return 0
```

```
    elif n == 1:
        return 1

    # Create a DP table to store Fibonacci numbers
    dp = [0] * (n + 1)
    dp[0] = 0
    dp[1] = 1

    # Fill the DP table using bottom-up approach
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]

    # Return the nth Fibonacci number from the DP table
    return dp[n]
```

g. Maths background: linear and nonlinear functions
    i. Linear functions satisfies the following:
        1. $f(x+y) = f(x) + f(y)$
        2. $f(c*x) = f(c) * f(x)$
    ii. Nonlinear function
        1. doesn't satisfy both conditions above

h. **Bellman recursion**

$$F(X_n^*) = \min_{X' \in S_{n-1}} F(X')$$

i. N number of inputs, k number of ways

Method	Exhaustive	Greedy	Dynamic programming
Applicability	Always	Matroid/ greedoid	Optimality principle
Typical complexity	$O(k^N)$, $O(N!)$	$O(Nk)$, $O(N^k)$	$O(Nk)$, $O(N^k)$

Exercise 1

Consider the following problem:

- A company makes square boxes and triangular boxes. Square boxes take 2 minutes to make and sell for a profit of 4. Triangular boxes take 3 minutes to make and sell for a profit of 5. No two boxes can be created simultaneously. A client wants at least 25 boxes including at least 5 of each type in one hour. What is the best combination of square and triangular boxes to make so that the company makes the most profit from this client?
    - Total profit:

$$F(x) = 4x_1 + 5x_2$$
$$x_1 >= 5,\ x_2 >= 5$$
$$2x_1 + 3x_2 <= 60$$
$$x_1 + x_2 >= 25$$

- Formalize this problem as a canonical optimisation problem, but consider it is ok for the objective to be a function to be maximised instead of minimised. Identify the design variables, the objective function and the constraints.

SDP exact: The greedy search algorithm makes the locally optimal choice at each stage, choosing the closest

SDP exact: O(n^2) insertion sort

▼ Permutation: order matters, n!

N, D, E

N, E, D

E, D, N

....