Lakehead UNIVERSITY

COMP 4433: Algorithm Design and Analysis

Dr. Y. Gu

Feb. 6, 2023 (Lecture 7)

Dynamic Programming (Final Example)

Example 5:

Optimal Binary Search Trees

Optimal Binary Search Trees

Binary search tree is a binary tree, in which the keys in the left subtree is less than the key in the root while keys in the right subtree is greater than the key in the root, and a subtree of binary search tree is also a binary search tree.

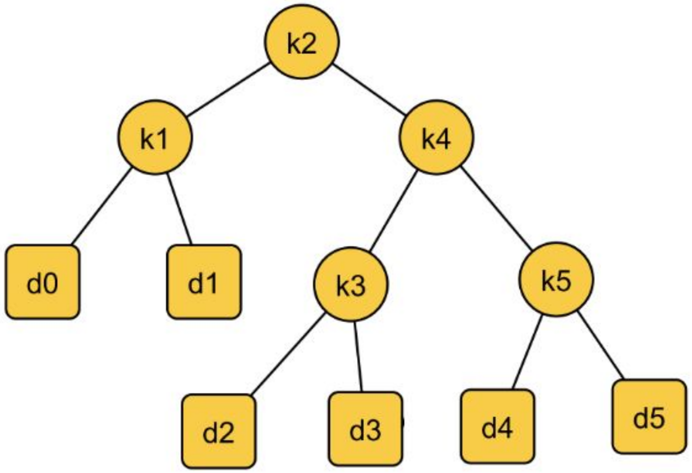
Now we consider a more general case. Suppose we have a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys in sorted order (i.e., $k_1 < k_2 < \cdots < k_n$). For each key k_i, the probability a search will be on k_i is p_i. We wish to build a binary search tree for these keys such that the expected search time (the average search time) is optimal.

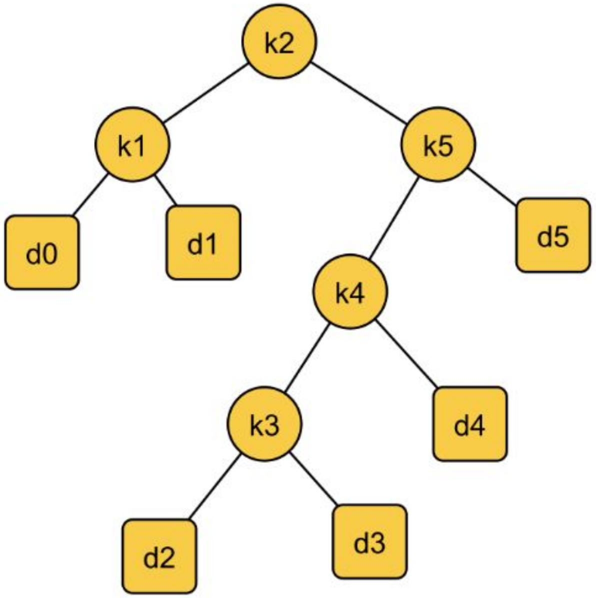
Optimal Binary Search Trees

We also need to consider the search values that are not in K. So we have n+1 dummy keys d_0, d_1, d_2 . . . d_n, where, d_i, 0<i<n, represents the values between k_i and k_{i+1}, d_0 represents the values less than k_1 and d_n represents the values greater than k_n. For each dummy key d_j, we assume the probability for searching according to it is q_j. For each dummy key d_j, we assume the probability for searching according to it is q_j. So, we have

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1.$$

Optimal Binary Search Trees





Optimal Binary Search Trees

Suppose we have already established the binary search tree T (in the tree, dummy keys should be leaves). Then we have the expected cost of a search in T is

$$
\begin{aligned}
E[\text{search cost in } T] &= \sum_{i=1}^{n} (depth_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (depth_T(d_i) + 1) \cdot q_i \\
&= 1 + \sum_{i=1}^{n} depth_T(k_i) \cdot p_i + \sum_{i=0}^{n} depth_T(d_i) \cdot q_i ,
\end{aligned}
$$

where *depthT* denotes a node's depth in the tree T. If the expected search cost is the smallest, then we call T an optimal binary search tree.

Optimal Binary Search Trees

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Two binary search trees are displayed in the previous slide.

The first tree has the expected search cost 2.80 and the second tree has the expected search cost 2.75, which is optimal.

Optimal Binary Search Trees

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

Optimal Binary Search Trees

To construct the tree, we can first construct a binary search tree with the n keys, then add the dummy nodes to leaves. But the number of binary search tree with n nodes is $\Theta(4^n/n^{3/2})$. So exhaustive search is not feasible. We can consider to use dynamic programming.

Dynamic Programming

Step 1: The structure of an optimal binary search tree

Suppose we have constructed an optimal binary search tree. Then each subtree must contain keys in a contiguous range $k_i, k_{i+1}, \ldots, k_j$, for some $1 \leq i \leq j \leq n$. In addition, that subtree must also contains the leaves of dummy keys $d_{i-1}, d_i, \ldots, d_j$.

Therefore we have the optimal substructure: if an optimal binary search tree T has a subtree T′ containing keys $k_i, \ldots, k_j$, then T′ must be optimal as well for subproblem with keys $k_i, \ldots, k_j$ and dummy keys $d_{i-1}, \ldots, d_j$. Otherwise we can replace the subtree with better expected cost and that means that T is not optimal.

Dynamic Programming

Considering the recursive method, if a subtree contains keys $k_i, \ldots, k_j$ and the root is k_r, then the left subtree contains keys $k_i, \ldots, k_{r-1}$ (and dummy keys $d_{i-1}, \ldots, d_{r-1}$) and the right subtree contains keys $k_{r+1}, \ldots, k_j$ (and dummy keys $d_r, \ldots, d_j$).

When the root is i, then the left subtree contains only d_{i-1} and when k_j is the root, its right subtree contains only d_j . We may try every possible key as the root to obtain the optimal subtree.

Dynamic Programming

Step 2: A recursive solution

We can define the values of optimal solution for subtrees as follows. For a subtree with keys $k_i, \ldots,$ k_j, define $e[i, j]$ to be the optimal expected cost of searching, where $i \geq 1$, $i - 1 \leq j \leq n$. Here we define $e[i, i - 1]$ as the subtree with d_{i-1} as a only node. So

$$e[i, i - 1] = q_{i-1}.$$

Dynamic Programming

- When j ≥ i, we need to select a root k_r, which forms two subtrees, one with the keys $d_i, \ldots, d_{r-1}$ and another with the keys $d_{r+1}, \ldots, d_j$.
- For a tree containing keys $k_s, \ldots, k_t$, the optimal value is $e[s, t]$.
- But when it becomes a subtree, the depth of each vertex will increase one. Therefore the the expected costs for this subtree will be

$$e[s,t] + \sum_{l=s}^{t} p_l + \sum_{l=s-1}^{t} q_l.$$

Dynamic Programming

We define

$$w(s,t) = \sum_{l=s}^{t} p_l + \sum_{l=s-1}^{t} q_l$$

Thus, if k_r is the root of an optimal subtree containing keys k_i ,…, k_j , we have

$$e[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j)) .$$

Note that $w(i,j) = w(i,r-1) + p_r + w(r+1,j)$

We have $$e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j) .$$

Dynamic Programming

Now we have the recursive formula for e[i, j].

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i - 1 \\ \min_{i \le r \le j}\{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \le j. \end{cases}$$

To help us to keep the track of the structure of optimal binary search tree, we define root[i, j] to be the index r for which k_r is the root of an optimal binary search tree containing keys k_i ,. . . , k_j .

Dynamic Programming

Step 3: Computing the expected search cost of an optimal BST

Similar to other dynamic programming, we need to use some tables to store the solutions for subproblems. So we define tables *e*, *w* and *root* in the following procedure. For *e* and *w* we need to define $1 \leq i \leq n + 1$, $0 \leq j \leq n$, because we need to record the values of “empty” subtrees (e.g., $e[i, i-1]$, $1 \leq i \leq n$).

Dynamic Programming

```
OPTIMAL-BST(p, q, n)
let e[1..n+1, 0..n], w[1..n+1, 0..n],
        and root[1..n, 1..n] be new tables
for i = 1 to n+1
    e[i, i-1] = q_{i-1}
    w[i, i-1] = q_{i-1}
for l = 1 to n
    for i = 1 to n-l+1
        j = i+l-1
        e[i, j] = ∞
        w[i, j] = w[i, j-1] + p_j + q_j
        for r = i to j
            t = e[i, r-1] + e[r+1, j] + w[i, j]
            if t < e[i, j]
                e[i, j] = t
                root[i, j] = r
return e and root
```

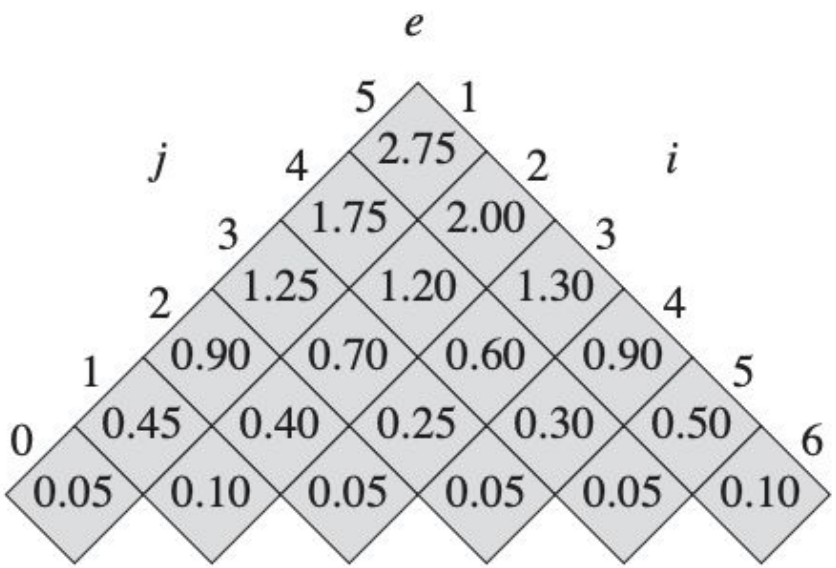
Running Time Discussion

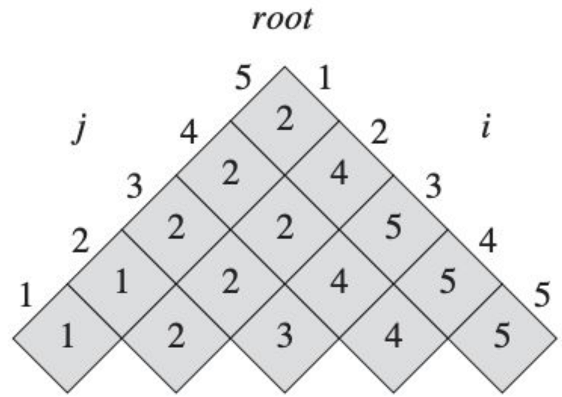
The Optimal-BST procedure takes $\Theta(n^3)$ time.

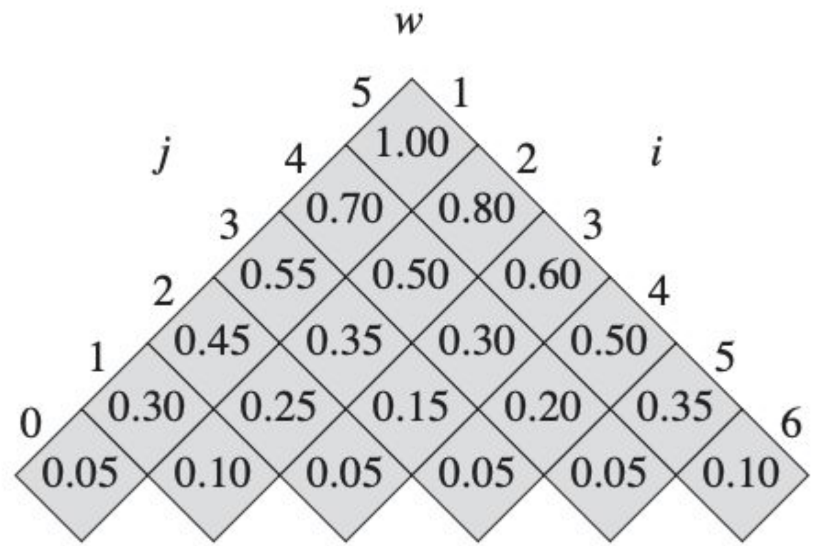
Because the main costs are the three nested for loops, each loop index takes at most n values, the running time is $O(n^3)$.

On the other hand, we can also see that the procedure takes $\Omega(n^3)$ time.

Example


e
5
1
2.75
j
4
2
i
1.75
2.00
3
3
1.25
1.20
1.30
2
4
0.90
0.70
0.60
0.90
1
5
0.45
0.40
0.25
0.30
0.50
0
6
0.05
0.10
0.05
0.05
0.05
0.10


root
5
1
2
j
4
2
i
2
4
3
3
2
2
5
2
4
1
2
4
5
1
5
1
2
3
4
5


w
5
1
1.00
j
4
2
i
0.70
0.80
3
3
0.55
0.50
0.60
2
4
0.45
0.35
0.30
0.50
1
5
0.30
0.25
0.15
0.20
0.35
0
6
0.05
0.10
0.05
0.05
0.05
0.10

Greedy Algorithm

Introduction

- The main idea of greedy algorithm is look some optimal solution locally and then try to extend globally. Usually the greedy algorithm is efficient.
- The greedy algorithm may not achieve optimal solution for the problem.
- We shall arrive at the greedy algorithm by first considering a dynamic programming approach and then showing that we can always make greedy choices to arrive at an optimal solution.

An Activity-Selection Problem

Suppose we have a set $S = \{a_1, a_2, . . . , a_n\}$ of *n* proposed activities that wish to use a resource (for example, a_i are presentations, which need to use one classroom).

Each activity a_i has a start time s_i and a finish time f_i, where $0 \leq s_i < f_i < \infty$. If selected, activity a_i takes place during the time internal $[s_i, f_i)$. Activity a_i and a_j are compatible if $[s_i, f_i) \cap [s_j , f_j) = \varnothing$, that is, if $s_i \geq f_j$ or $s_j \geq f_i$.

In the activity-selection problem, we wish to select a maximum-size subset of mutually compatible activities.

We assume that the activities are sorted in monotonically increasing order of finish time:

$$f_1 \leq f_2 \leq \cdots \leq f_{n-1} \leq f_n.$$

An Activity-Selection Problem

Example: Suppose the activity set S is as follows.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Then the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities. But it is not the largest subset. The subsets $\{a_1, a_4, a_8, a_{11}\}$ or $\{a_2, a_4, a_9, a_{11}\}$ are largest subsets.

An Activity-Selection Problem

We first try to find some recursive method for the optimal subproblems.

Let S_{ij} denote the subset of activities that start after activity a_i finishes and end before a_j starts, and suppose such a maximum set is A_{ij} .

Let $a_k \in A_{ij}$ be an activity, then we claim that $A_{ik} = S_{ik} \cap A_{ij}$ must be an optimal solution of S_{ik}. Otherwise we will be able to improve A_{ij} and A_{ij} would not be optimal. Similarly, $A_{kj} = S_{kj} \cap A_{ij}$ is also optimal.

Therefore, $A_{ij} = A_{ij} \cup \{a_k\} \cup A_{kj}$ and $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$.

An Activity-Selection Problem

Let c[i, j] denote the size of optimal solution for the set S_{ij} , then we have the following formula, i.e. c[i, j]=$|A_{ij}|$, then

$$c[i,j] \quad = \quad \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

- From the above formula, we can develop a dynamic programming.
- We want to use a simpler method to solve the problem with "greedy choice".

An Activity-Selection Problem

- Intuition suggests that we should choose an activity that leaves the resource available for as many other activities as possible.
- We first want to choose a_1 (recall that f_i, $i = 1, . . . , n$, are sorted) because f_1 is the earliest finish time of any activities.
- Let $S_k = \{a_i \in S : s_i \geq f_k\}$ be the set of activities that start after activity a_k finishes.
- If we make the greedy choice of activity a_1, then S_1 remains as the only subproblem to solve.

An Activity-Selection Problem

Before we use the above idea to solve the problem, we want to make sure that the solution will be optimal. We have the following theorem.

Theorem:

Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

An Activity-Selection Problem

Proof.

Let A_k be the maximum-size subset of mutually compatible activities in S_k.

Let a_j be the activity in A_k with the earliest finish time.

If $a_j = a_m$, we are done.

Otherwise, a_m must compatible to all the activities in $A_k \backslash \{a_j\}$ since $f_m \le f_j$. Let $A'_k = A_k \backslash \{a_j\} \cup \{a_m\}$, then $|A'_k| = |A_k|$. So A'_k is maximum-size subset of mutually compatible activities of S_k.

A Recursive Greedy Algorithm

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
m = k + 1
while m ≤ n and s[m] < f[k]          // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return Ø
```

- We can call Recursive-Activity-Selector(s, f, 0, n) to obtain the optimal solution for the problem.
- The running time is $\Theta(n)$: each activity is examined once in while loop. This is assume that s, f are already sorted.
- If it is not sorted, then there are sorting algorithms with running time $O(n \log n)$.

An Iterative Greedy Algorithm

```
GREEDY-ACTIVITY-SELECTOR(s, f)
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}
        k = m
return A
```

- In the procedure, the variable k indexes the most recent addition to A, corresponding to the activity ak.
- It is easy to see that the running time for this procedure is also Θ(n) given that s, f are already sorted.

Summary of Steps

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution.
3. Show that if we make the greedy choice, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice. (Steps 3 and 4 can occur in either order.)
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

Elements of the Greedy Strategy

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

Elements of the Greedy Strategy

Some properties of the problem can be used to see if a greedy algorithm is applicable.

First key ingredient is the greedy-choice property: we can assemble a globally optimal solution by making locally optimal (greedy) choices.

In dynamic programming, we also make choices, but the choices are depends on solved subproblems. In greedy algorithm, we make whatever choice seems best at the moment and then solve the subproblem that remains. So the greedy algorithm is top-down algorithm.

Elements of the Greedy Strategy

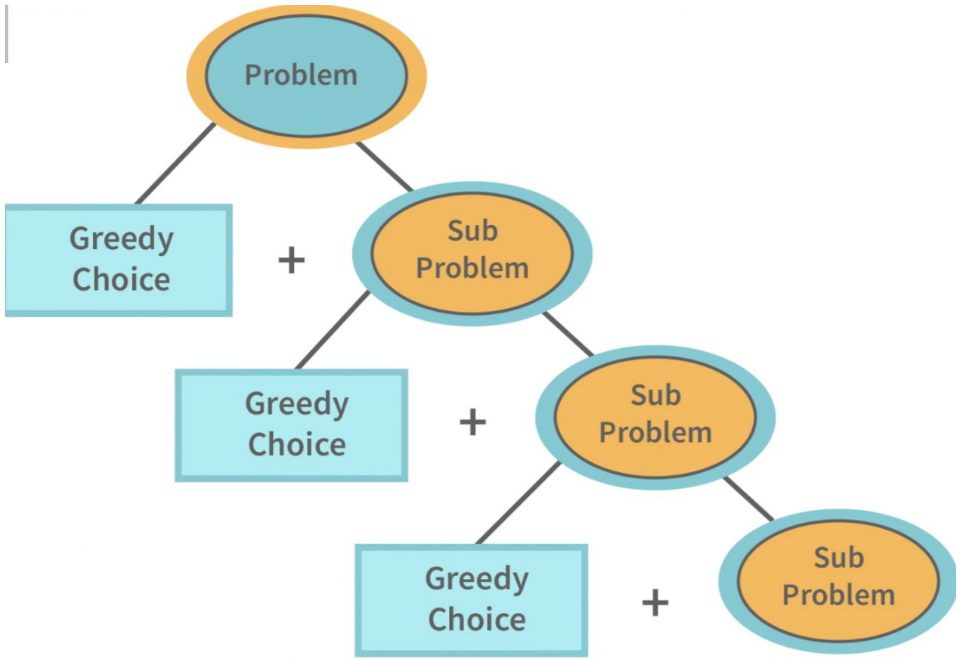
Another thing is the problem exhibits optimal substructure: an optimal solution to the problem contains within it optimal solutions to subproblems.

In greedy algorithm, usually we arrived at a subproblem by having made the greedy choice in the original problem.

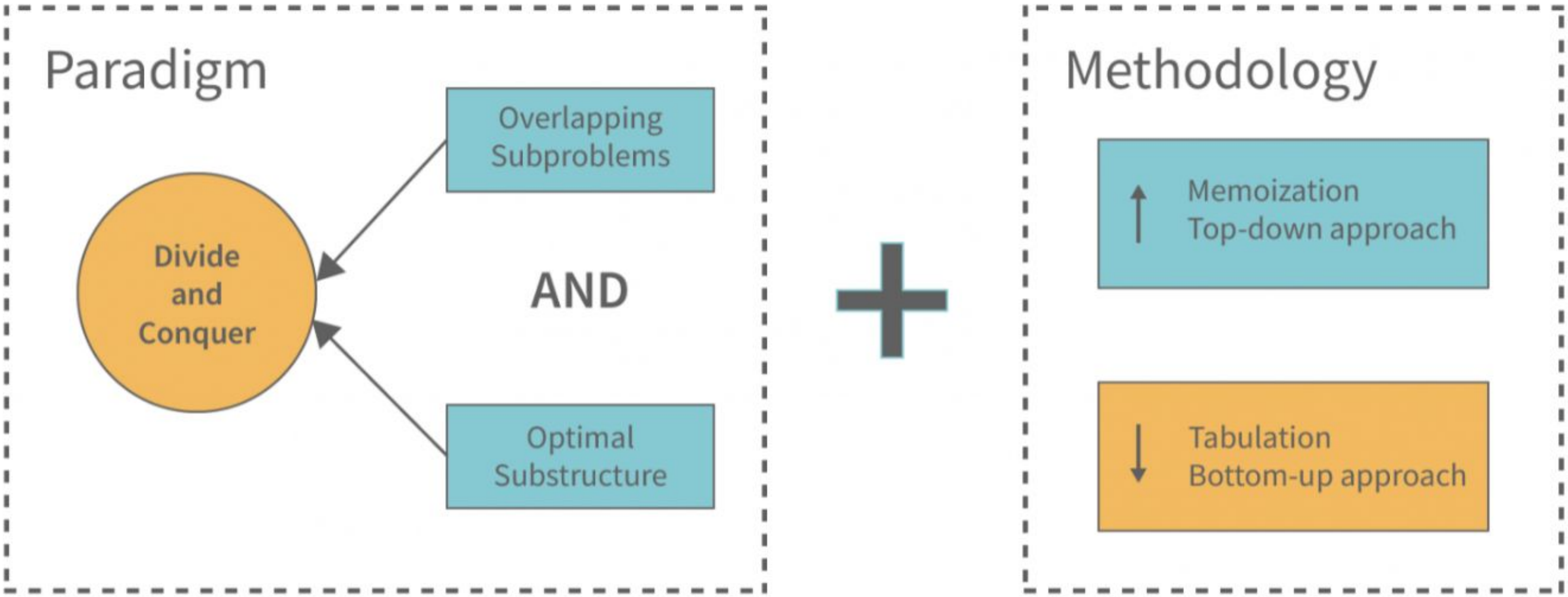
Then we need to prove that an optimal solution to the subproblem combined with the greedy choice already made will yield an optimal solution to the original problem.

Greedy Algorithm v.s. DP

Since both dynamic programming and greedy programming consider the optimal substructures, sometimes we may be confused which method is suitable for the solution



Greedy Algorithm v.s. DP



Greedy Algorithm v.s. DP

Greedy Programming	**Dynamic Programming**
A greedy algorithm chooses the best solution at the moment, in order to ensure a global optimal solution.	In dynamic programming, we look at the current problem and the current solution to determine whether to make a particular choice or not. We then calculate the optimal choice based on previous problems and solutions.
It is not guaranteed that an optimal solution will be obtained in the greedy method.	Because of the nature of Dynamic Programming, it is certain that an optimal solution will be generated.
More Efficient because we never look back to other options.	Less Efficient as compared to a greedy approach becausee it's required DP table to store the answers of calculated states.
A set of overlapping problems cannot be dealt with.	A set of overlapping problems can be dealt with.
No memorization is required.	Memorization is required.
Faster than a dynamic one.	Slower compared to Greedy Algorithm

Greedy Algorithm v.s. DP

Example: The 0-1 knapsack problem is the following. A thief robbing a store finds n items. The ith item is worth v_i dollars and weight w_i pounds, where v_i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack. The problem is which items should he take. (0-1 means for each item take or not take).

In the fractional knapsack problem, the setup is the same, but the thief can take fractions of items, rather than having to take the whole item.

Greedy Algorithm v.s. DP

Both knapsack problems have the optimal substructure property.

- For the 0-1 problem, consider the most valuable load that weighs at most W pounds. If we remove item *j* from this load, the remaining load must be the most valuable load weighing at most $W - w_j$ that the thief can take from the n−1 original items excluding j .
- For the comparable fractional problem, consider that if we remove a weight w of one item j from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ that the thief can take from the n−1 original items plus $w_j - w$ pounds of item j .

Greedy Algorithm v.s. DP

Although the problems are similar, we can solve the fractional knapsack problem by a greedy strategy, but we cannot solve the 0-1 problem by such a strategy.

To solve the fractional problem, we first compute the value per pound v_i/w_i for each item. Obeying a greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound. If the supply of that item is exhausted and he can still carry more, he takes as much as possible of the item with the next greatest value per pound, and so forth, until he reaches his weight limit W . Thus, by sorting the items by value per pound, the greedy algorithm runs in $O(n \log n)$ time.

Greedy Algorithm v.s. DP

The same greedy strategy does not work for the 0-1 knapsack problem.

Consider a small example which has 3 items and a knapsack that can hold 50 pounds. Item 1 weighs 10 pounds and is worth $60. Item 2 weighs 20 pounds and is worth $100. Item 3 weighs 30 pounds and is worth $120. Thus, the value per pound of item 1 is greater than the value per pound of other two items. However, if we take item 1 first, then we will not get the optimal solution.

Greedy Algorithm v.s. DP

In the 0 - 1 problem, when we consider whether to include an item in the knapsack, we must compare the solution to the subproblem that includes the item with the solution to the subproblem that excludes the item before we can make the choice. The problem formulated in this way gives rise to many overlapping subproblems— a hallmark of dynamic programming.

After Class

- After class:

Part III Chapter 12, Part IV 16.1