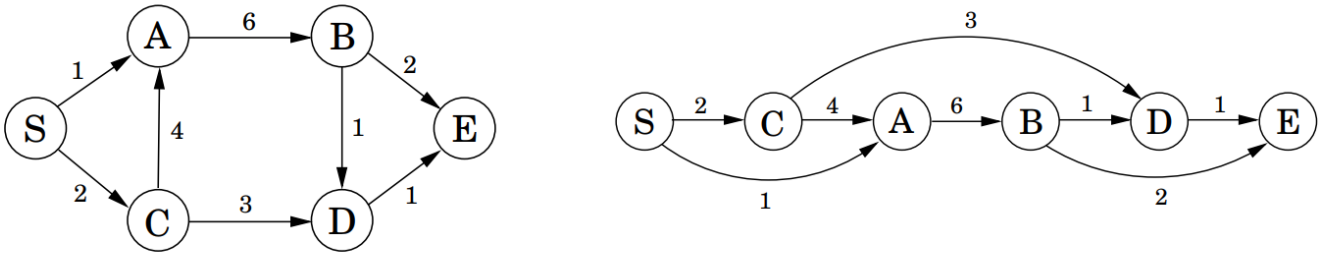
Dynamic Programming

Shortest paths in dags, revisited

- The *special distinguishing feature of a dag* is that its nodes can be ***linearized***; that is, they can be arranged on a line such that ***all edges go from left to right***

- **e.g.**

Figure 6.1 A dag and its linearization (topological ordering).



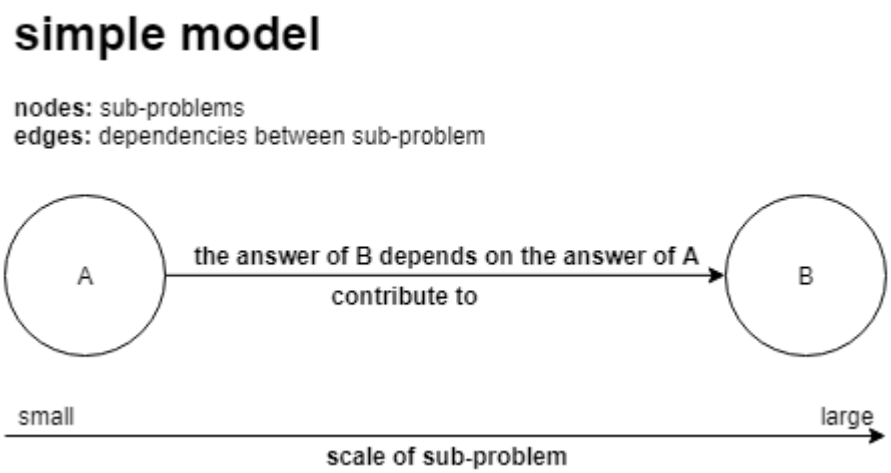
  - **Goal:** find the $distance$ from node S to all other nodes, where $distance$ stands for `the shortest path`
  - Suppose we focus our attention on node D. The only way to reach to it is through its $predecessors, B\ or\ C$
  - Thus we have: dist(D) = $\min\{dist(B)+1, dist(C)+3\}$
  - The similar relation can be applied for every node

Pseudo code

Initialize $dist(v) = \infty$ for all $v \in V$
$dist(s) = 0 \longrightarrow$ can be obtained directedly
for each $v \in V$, in linearized order:
    dist(v) = $\min_{(u,v)\in E}\{dist(u) + l(u,v)\}$

> i.e. for all the incoming edge (u, v) from u to v
> we compute dist(u) + $l(u,v) \rightarrow$ ***the shortest path from s to v via intermediate station u***
> we want the minimum one to reach v regardless what the inermediate station u is

- This algorithm is solving ***a collection of subproblems*** dist(u) : $u \in V$
  - i.e. ultimately we get all distance (shortest path) from s to all other nodes
- ***Important conclusions:***
  - *Dynamic Programming* is a very powerful algorithmic in which **a problem is solved by identifying a collection of subproblems and tackling them one by one, smallest first, using the answers to small problmes to help figure out larger ones, until the whole lot of them is solved.**
  - In dynamic programming we are not given a dag, the dag is *implicit*
    - **nodes:** are the $subproblem$ we define
    - **edges:** are the $dependencies$ between the subproblems
    - If to solve subproblem B we need the answer to subproblem A, then there is a ***(conceptual)*** edge from A to B, in this case, A is thought of as a smaller subproblem B



    - ***subproblems has topological ordering***
    - ***You can solve a subproblem when all subproblems that contribute to it has been solved already"***

Longest increasing subsequences

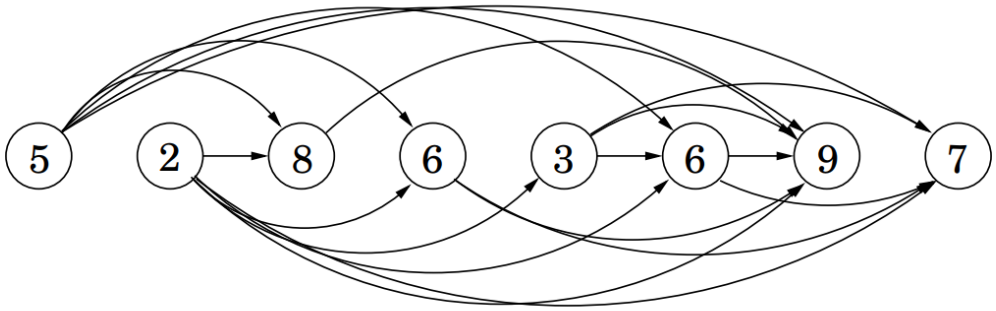
- **Problem definition:**

> Given a sequence of numbers $a_1, \ldots, a_n$, find the length of the longest increasing sequence. For example, the longest incresing subsequence of 5, 2, 8, 6, 3, 6, 9, 7 is 2, 3, 6, 9

- **Restrictions on increasing subsequence:**
  1. **Temporal:** If you pick the $i-th$ element and include it into your solution set, you can only pick elements from $a_{i+1}, \ldots, a_n$ in the future. *You can never pick elements backward*
  2. **Value:** You can only pick elements strictly in increasing order

- **Transfer Problem into DAG**

Figure 6.2 The dag of increasing subsequences.



- **It is the restrctions mentioned (Temporal and Value) above define the edge of our DAG**
  - **Temporal:** (u, v) is always from left to right
  - **Value:** (u, v) is exsited iff v is strictly larger thant u
  - Basically, there is an edge between u and v if and only if
    - u is at the left of v
    - $value(v) > value(u)$
- **L(j) is defined as the length of the longest increasing subsequence ending at j**
- Thus our problem turns into:

$$Problem = max\{L(j)\}, for\ all\ 0 \leq j < n$$

- We can compute L(j) from ***left to right*** such that we already have the answer of all smaller subproblems contributing to L(j) when we want to compute L(j)

- **Pseudo code**

for $j = 0, 2, \ldots, n-1$
  $L(j) = 1 + \max\{L(i)\} : (i, j) \in E$
return $\max_j L(j)$
i.e. for all previous nodes that can contribute current node j, find the maximum contribution

Related to Divide and Conquer

- Is a good idea to solve the problem above using recursion?

    - **NO**, since the subproblem is **highly overlapped** such that each subproblem may be computed for **multiple times**
- What problem is suitable for recursion?
    - when there is **no overlapping between subproblems**, i.e. they are **independent** to each other such that each subproblem will be computed **only once**. *That's why recursion work so well in Divide and Conquer*

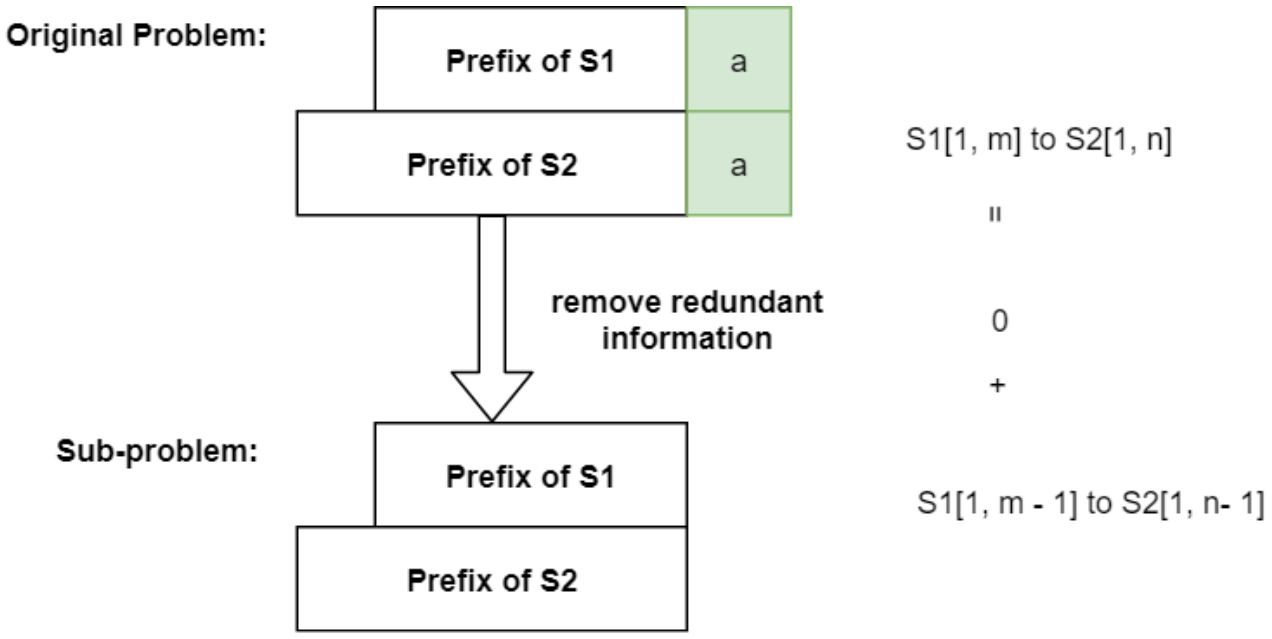
	Dependency between subproblems	Size of subproblems	Height of recursion tree
Divide and Conquer	Independent	drop substantially	small
Dynamic Programming	Overlapped	drop very slowly	large

Edit distance

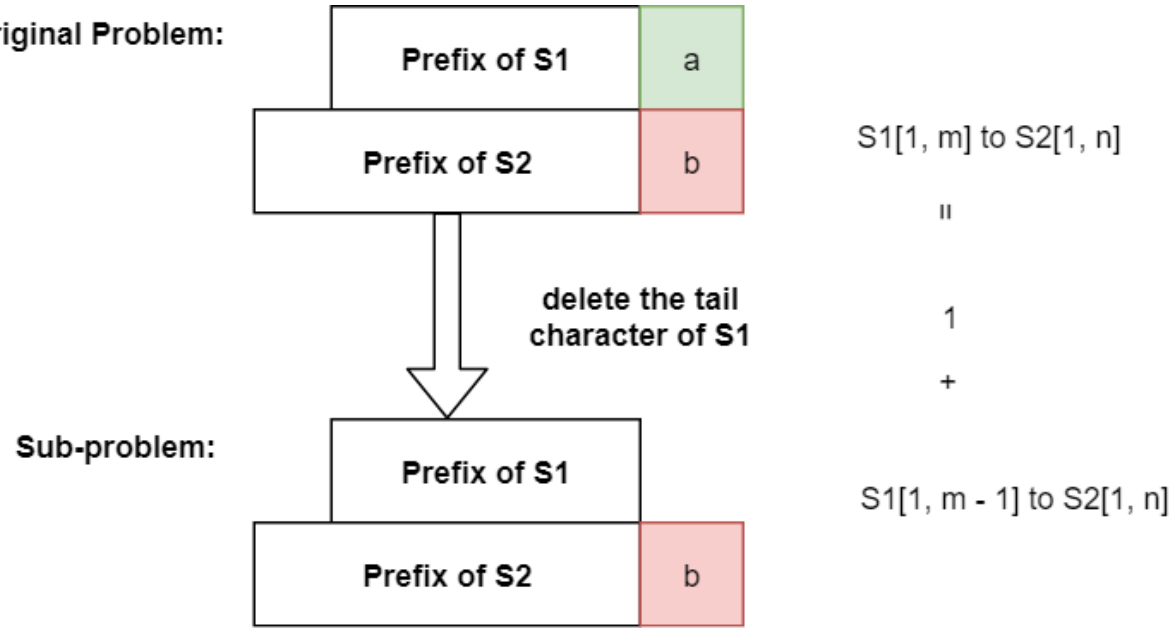
- Given two strings, find the minimum number of steps required to convert word1 to word2.
- 3 operations are allowed to perform on each word
    - *Insert a character*
    - *Delete a character*
    - *Replace a character*
- **Define subproblems**

> Try to *get rid of redundant information* to reduce the size of problem $\Longrightarrow$ get subproblem

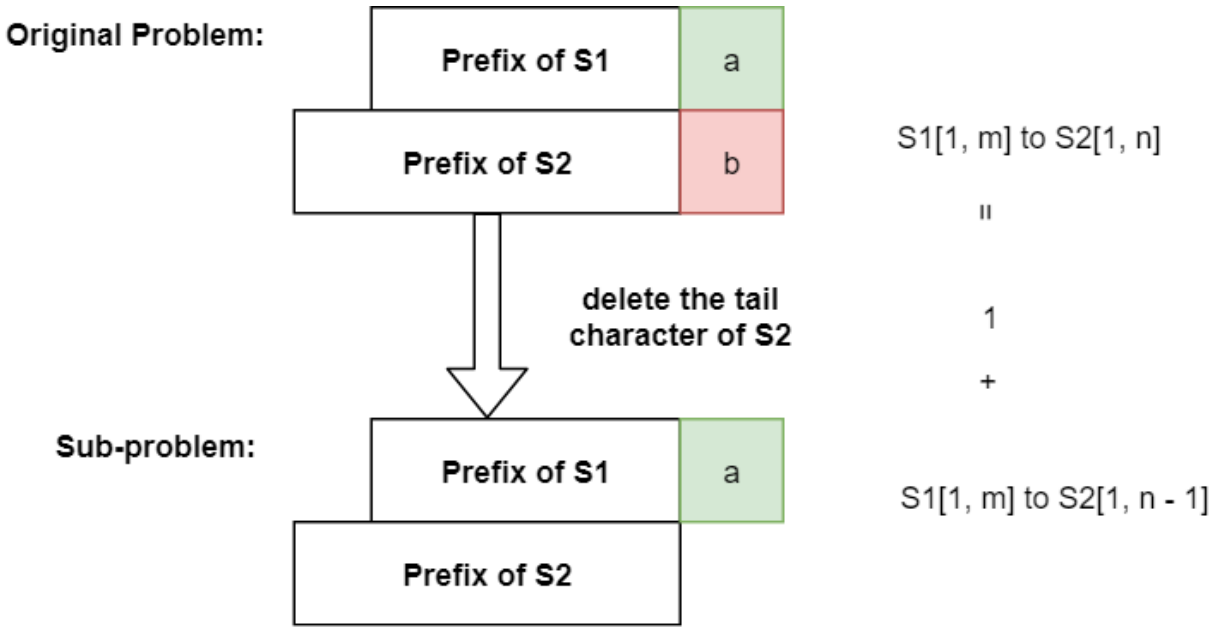
- Suppose we are given two string `S1[1 : m ]` and `S2[1 : n]`
    - **If the last character of S1 is exactly same as the last character of S2**
        - All we need to do is find the minimum operations needed to convert the prefix `S1[1 : m - 1]` to the prefix `S2[1 : n - 1]`



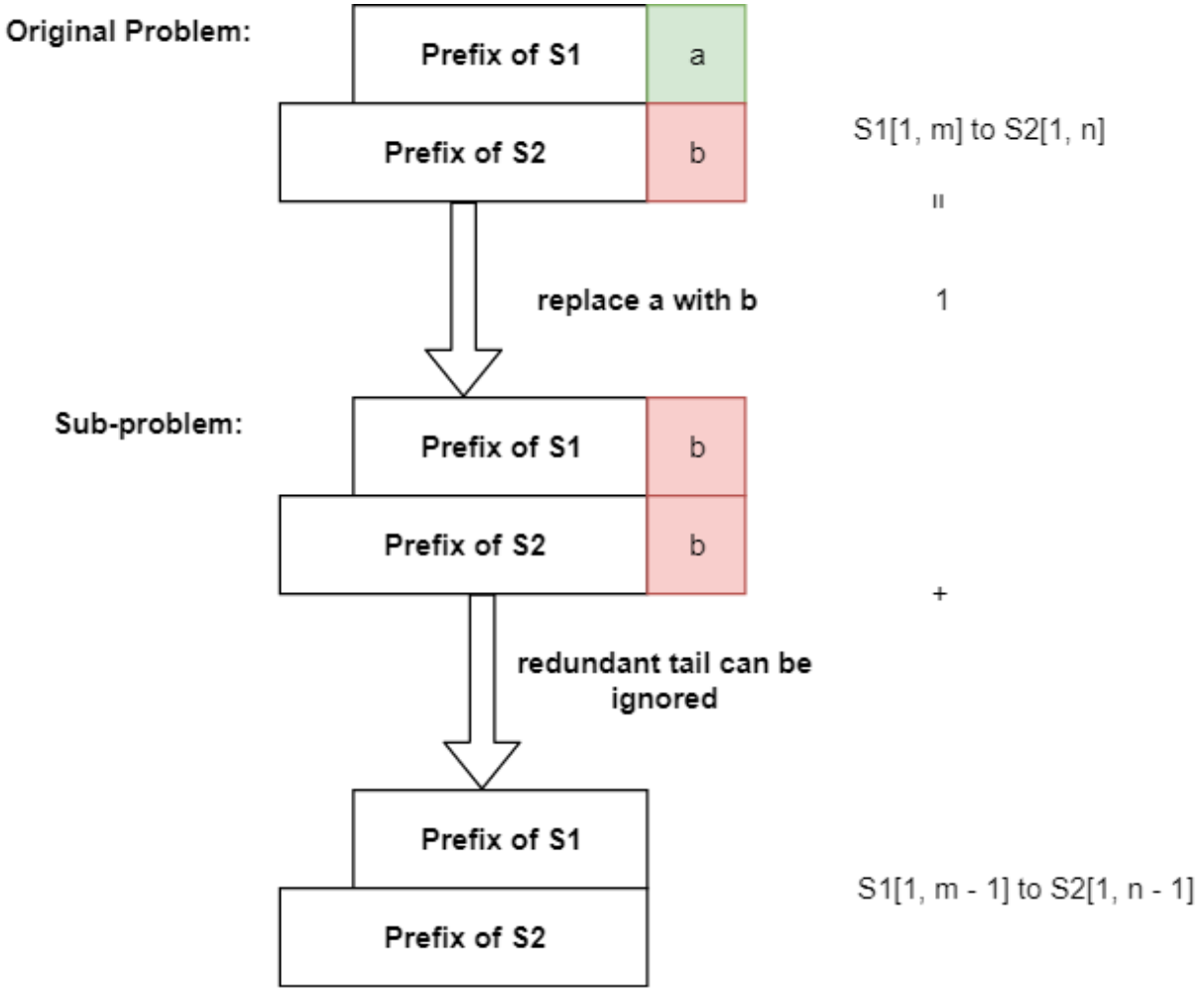
- **Else $\Longrightarrow$ the tail charcters are different**
  - We can do the following 3 operations:
    1. **delete the last character of S1,** and find the minimum operations needed to convert S1[1, m - 1] to S2[1, n];



    2. **delete the last charcter of S2,** and find the minimum operations needed to convert S1[1, m] to S2[1, n - 1];



3. **replace the last character of S1 with the last character of S2,** and find the minimum operations needed to convert S1[1, m - 1] to S2[1, n - 1]



- **Relation of subproblem**

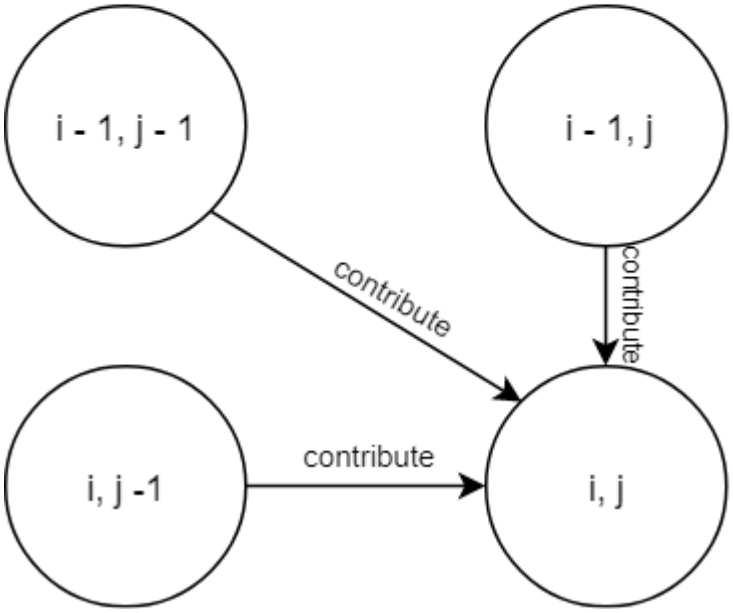
$$Prob(i, j) = Prob(i - 1, j - 1)\ if\ same\ tail$$

$$= min\{Prob(i-1,j), Prob(i,j-1), Prob(i-1,j-1)\} + 1, different\ tail\ char$$

DAG for Edit Distance

Solution to state(i, j) is contributed by solutions to state(i - 1,j), state(i, j - 1), state(i - 1, j - 1)

We need to solve sub-problems in topological ordering, solve state(i - 1, j), state(i, j - 1), state(i - 1, j - 1) first before trying to solve state(i, j)



- **Base case:**
  - Prob(" "," ") = 0
  - Prob(" ", non-empty string) = size of non-empty string $\Longrightarrow$ consecutive insertions
  - Prob(non-empty string, " ") = size of non-empty string $\Longrightarrow$ consecutive deletions
- **Code**

```
class Solution {
public:
    int minDistance(string word1, string word2) {
        int m = word1.size();
        int n = word2.size();
        vector<vector<int>> table(m + 1, vector<int>(n + 1));

        for (int i = 0; i < m + 1; ++i) {//non-empty to empty
            table[i][0] = i;
        }

        for (int j = 0; j < n + 1; ++j) {//empty to non-empty
            table[0][j] = j;
        }

        for (int i = 1; i < m + 1; ++i) {//fill table
            for (int j = 1; j < n + 1; ++j) {
                if (word1[i - 1] == word2[j - 1]) {//same tail char
                    table[i][j] = table[i - 1][j - 1];
                } else {//different tail
                    table[i][j] = min(min(table[i - 1][j - 1], table[i - 1][j]), table[i][j - 1])+
                }
            }
        }

        return table.back().back();
    }
};
```

Common subproblems

> Finding the right subproblem takes *creativity and experimentation.* But there are a few standard choices that seem to arise **repeatedly** in dynamic programming

1. The input is $x_1, x_2, \ldots, x_n$,
   - subproblem is $x_1, x_2, \ldots, x_i \implies$ **prefix**$[1 : i]$
   - Number of subproblems: $O(n)$
2. The input is $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_m$,
   - subproblem is $x_1, x_2, \ldots, x_i$ and $y_1, y_2, \ldots, y_j \implies$ prefix$[1 : i]$, prefix$[1 : j]$
   - Number of subproblems: $O(mn)$
3. The input is $x_1, x_2, \ldots, x_n$,
   - subproble is $x_i, x_{i+1}, \ldots, x_j \implies$ **substring**$[i : j]$
   - Number of subproblems: $O(N^2)$
4. The input is a rooted tree,
   - subproblem is a **rooted subtree**

Knapsack $\rightarrow$ resource-constrained slections

- **Problem definition**
  - ***What is the maximum profit you can gain?***
  - Given a lists of items $i_1, i_2, \ldots, i_n$
    - Each items has a value $v_1, v_2, \ldots, v_n$
    - Each items have a weight $w_1, w_2, \ldots, w_n$
  - Given a single container with **limit capacity** W
- **Example**

Item	Weight	Value
1	6	$30
2	3	$14
3	4	$16
4	2	$9

  The maximum profit you can gain: $48 (two of items) if repetition is allowed
  The maximum profit you can gain: $46 (item 1 and 3) if repetition is probhibit
- **Two versions**
  1. Pick items with repetition
  2. Pick items withouth repetition

Knapsack with repetition

- How to define subproblem,
  - Can we look at fewer items?
    - No, since repetition is allowed. You can always chose item from the whole list regardless what previous item you selected
  - Can we reduce the knapsack capacities?
    - Yes, when a item is selected and put into the knapsack, the capacity will decrease accordingly
- Guess:
  - If the item i is in the optimal solution set,
    - $Prob(W) = Prob(W - w_i) + v_i,\ if\ w_i \leq W$
    - We don't know what the item i is, *so we try them all*

    > i.e.
    > If I pick item 1, what the maximum profit I can gain?

> If I pick item 2, what the maximum profit I can gain?
> ⋮
> If I pick item n, what the maximum profit i can gain?
> **Try them all**, and find the maximum one

DAG for knapsack with repetition

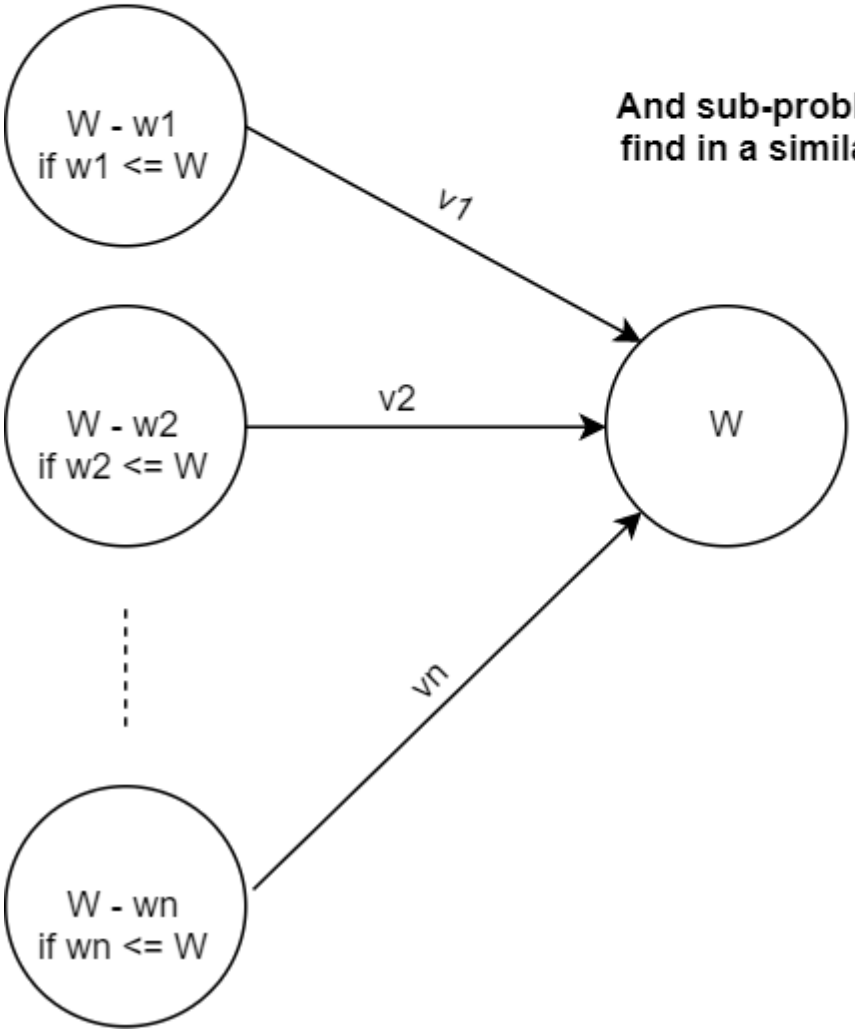
nodes: sub-problems

edge: dependency between sub-problems

value of node: the capacity of your bag

base case: No profit you gain if the capacity is 0

Topological ordering: It can be easily seen that the solution of any sub-problem depend on the solutions of previous sub-problems



And sub-problem(W - wi) can be find in a similar way recursively

> It can be easily see that there are total W subproblems
> Each subproblem can be solved in $O(n)$, i.e. try all items
> The overall complexity: $O(Wn)$

- **Pseudo Code**

$K(0) = 0$
for $w = 1 \ \ to \ \ W$
$\quad K(w) = \max\{K(w - w_i) + v_i : w_i \leq w\}$
$return \ K(w)$

Knapsack without repetition

- **Try to decrease the size of your problem**
  - Can we look at fewer items?
    - Yes and Must, since repetition is not allowed. You cannot chose item that is already in your knapsack;
  - Can we reduce the knapsack capacities?
    - Yes, when a item is selected and put into the knapsack, the capacity will decrease
- **Each time you pick an item and put it into your knapsack,**
  - The capacity of your knapsack decreases $\downarrow$
  - Your choice of items narrows down $\downarrow$
- **Guess:**
  - For each item i you could have **ONLY** two choices,
    1. Pick it up and put into your knapsack if you can ($w_i <= W$) , in this case

$$Prob(W, n) = Prob(W - w_i, n - 1) + v_i$$

    2. Either you don't want the item at all, or the item is not affordable for your knapsack

$$Prob(W, n) = Prob(W, n - 1)$$

  - Prob(W, n) is defined as: "the maximum profit you can gain if you can select items from $i_1, i_2, \ldots, i_n$, and your knapsack has W available space

> **Note that:** the item can been seen as `prefix[1:n]`

DAG for knapsack without repetition

nodes: sub-problems

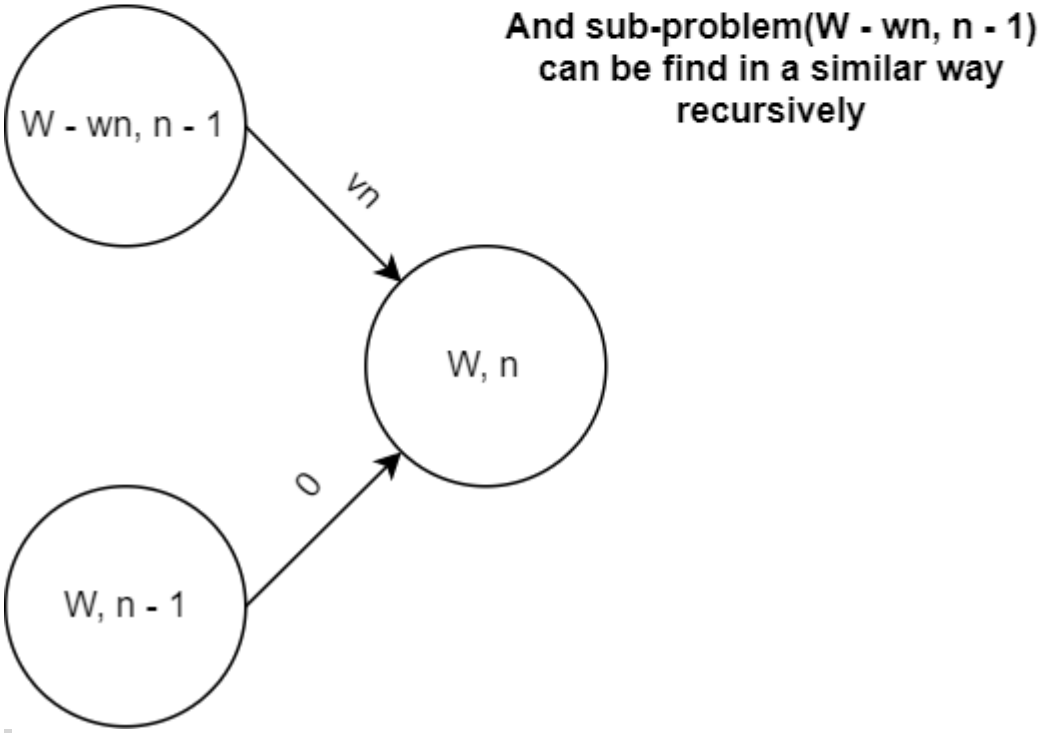
edge: dependency between sub-problems

value of node: # of items available, the capacity of your bag

base case: Prob(0, i) = 0 nothing you can gain if knapsack is full already

Prob(w, 0) = 0 noting you can gain if no item is available

Topological ordering: It can be easily seen that the solution of any sub-problem depend on the solutions of sub-problems with smaller size



> Number of subproblems: $\mathbf{W} * \mathbf{n}$
> Each subproblem can be solved in: $\mathbf{O}(\mathbf{1})$
> The overall complclass: $\mathbf{O}(\mathbf{W}\mathbf{n})$

- **Pseudo Code**

  Create a 2D table and initialize the first row and first column to 0

  $for\ j = 1\ to\ n:$
  $\quad for\ w = 1\ to\ W:$
  $\quad\quad if\ w_j > w$
  $\quad\quad\quad K(w, j) = K(w, j-1)$

$$else$$
$$K(w,j) = \max\{K(w,j-1), K(w-w_j, j-1) + v_j\}$$
$$return K(W,n)$$

Practice

```
class Solution {
public:
    int backPackII(int W, vector<int> &weight, vector<int> &value) {
        // write your code here
        int n = weight.size();
        vector<vector<int>> table(n + 1, vector<int>(W + 1));

        for (int i = 1; i < n + 1; ++i) {
            for (int j = 1; j < W + 1; ++j) {
                if (weight[i - 1] > j) {
                    table[i][j] = table[i - 1][j];
                } else {
                    table[i][j] = max(table[i - 1][j - weight[i - 1]] + value[i - 1], table[i - 1]
                }
            }
        }

        return table.back().back();
    }
};
```

Memoization

getSolution(n) {
    if solution(n) is in hash-table
        return solution(n) ***directly***
    else
        calculate solution(n)
        save solution(n) into hash-table before return it
        return solution(n);
}

> such that each subproblem will be calculated **only once**, thus achieve overall complexity:
>
> $$Number of subproblems * time\ cost\ per\ subproblem$$
>
> otherwise, this kind of recursive function without memoization could cause an exponential runtime