# CS 341 – Algorithms

## Lecture 11 – Dynamic Programming I

23 June 2021

# Today's Plan

1. Framework
2. Weighted interval scheduling
3. Subset-sum and knapsack

# Introduction

On a high level, we can solve a problem by dynamic programming if there is a recurrence relation with only a polynomial number of subproblems.

This is a general and powerful technique, and is also easy to use as it is more systematic.

It extends the ideas in the previous topics including divide and conquer, graph searches, and greedy.

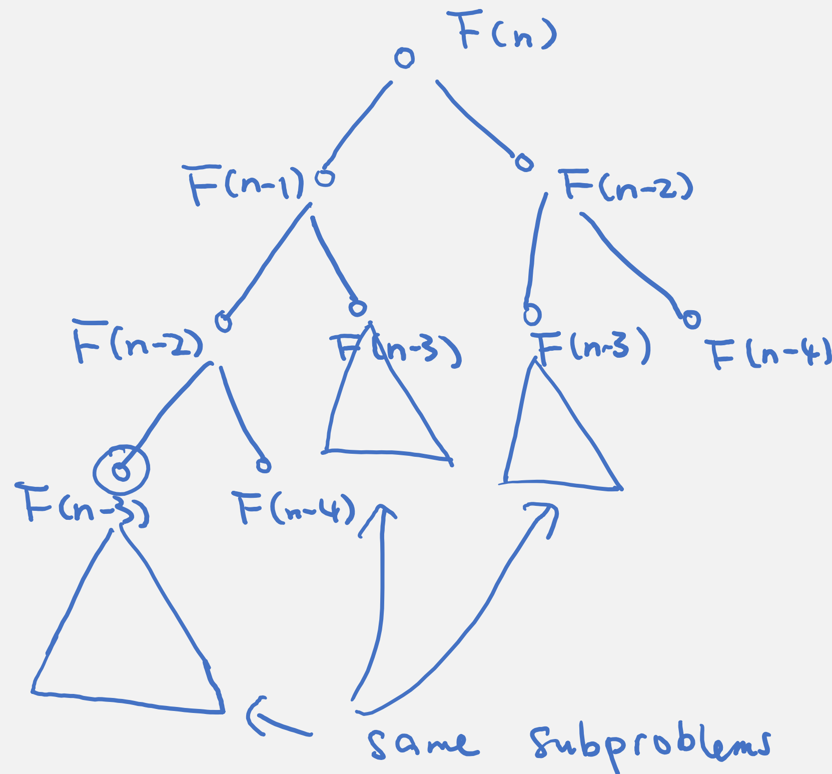
Once you learnt it, you will feel much more confident as a problem solver, as then you can solve interesting and nontrivial problems that were out of reach in a rather routine way.

So this is a topic that I hope you can learn well, and we will do many examples to achieve this goal.

# Toy Example

To illustrate the framework, let's consider the simple problem of computing the Fibonacci sequence.

$$F(n) = F(n-1) + F(n-2). \qquad F(1) = F(2) = 1.$$




$O(1.618^n)$

# Top-Down Memorization

As in BFS/DFS, we can use an array visited[i] to ensure that each subproblem is computed at most once,

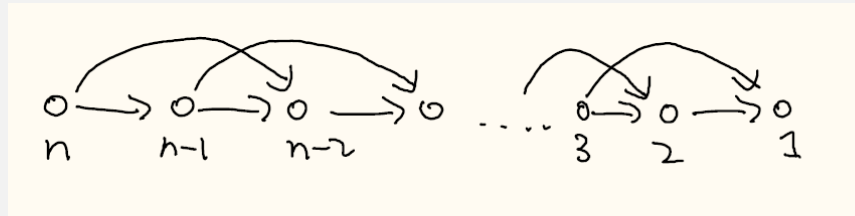
and also we use an array answer[i] to store the value $F(i)$ for future lookup.

```
visited[i] = false   for  1 ≤ i ≤ n   }  main program
F(n).
```

```
F(i)    // recursive function
   if  visited[i] = true ,  return  answer[i].
   if  i = 1  or  i = 2,  return  1.        base case
   answer[i] = F(i-1) + F(i-2).
   visited[i] = true.
   return  answer[i].
```

# Time Complexity of Top-Down Memorization

The analysis is similar to what we did in BFS/DFS.



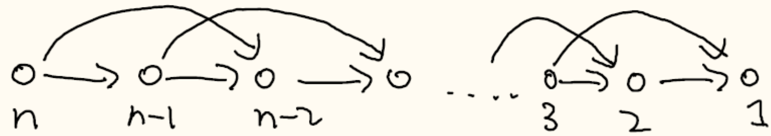

subproblem dependency graph

each subproblem is computed at most once

subproblem is computed. need 2 lookup

total time complexity : O(n) additions

# Bottom-Up Computation




$F(1) = F(2) = 1$

for $3 \leq i \leq n$ do

$F(i) = F(i-1) + F(i-2)$.

$O(n)$ additions. but not $O(n)$ time

the numbers grow exponentially

# Dynamic Programming

So, this is basically the framework of dynamic programming, to store the intermediate values so that we don't need to compute the subproblems again.

From the top-down approach, as long as there is a recurrence with only a polynomial number of subproblems (and polytime processing), then the problem can be solved in polynomial time.

So, the key to designing a dynamic programming algorithm is to come up with a nice recurrence.

We will see that many interesting and seemingly difficult problems have such a nice recurrence.

We will study many examples so that you will acquire the skills to come up with these recurrences.

wiki

The word *dynamic* was chosen by Bellman to capture the time-varying aspect of the problems, and because it sounded impressive.[11]

# Top-Down vs Bottom-Up

In practice, the bottom-up implementation is preferred as it requires no recursion and so more efficient.

In principle, to come up with a bottom-up implementation, we just need to use a topological ordering of the "subproblem dependency graph" to have a correct order to solve the subproblems.

Usually, it is rather straightforward. Sometimes, it requires us to think clearly.

Our main focus will be to come up with the recurrence, as this already implies a polytime algorithm.

We will also mention the bottom-up implementations as much as possible.

DP $\neq$ bottom-up

decouple

# Today's Plan

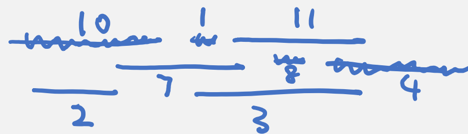
1. Framework
2. Weighted interval scheduling
3. Subset-sum and knapsack

# Weighted Interval Scheduling

**Input**: $n$ intervals $[s_1, f_1], [s_2, f_2], \dots, [s_n, f_n]$, and a weight $w_i$ for each interval $i$.

**Output**: a subset of disjoint intervals that maximizes the total weight.

This is a generalization of the interval scheduling problem in L08.pdf.




$w_i = 1 \ \forall i$

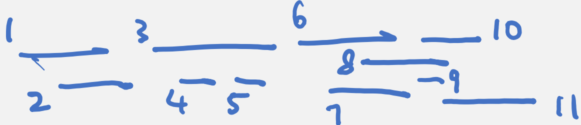
room

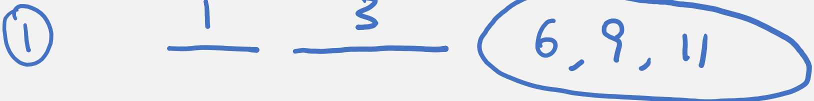
Unlike the special case, there are no known greedy algorithms for this problem.

# Exhaustive Search

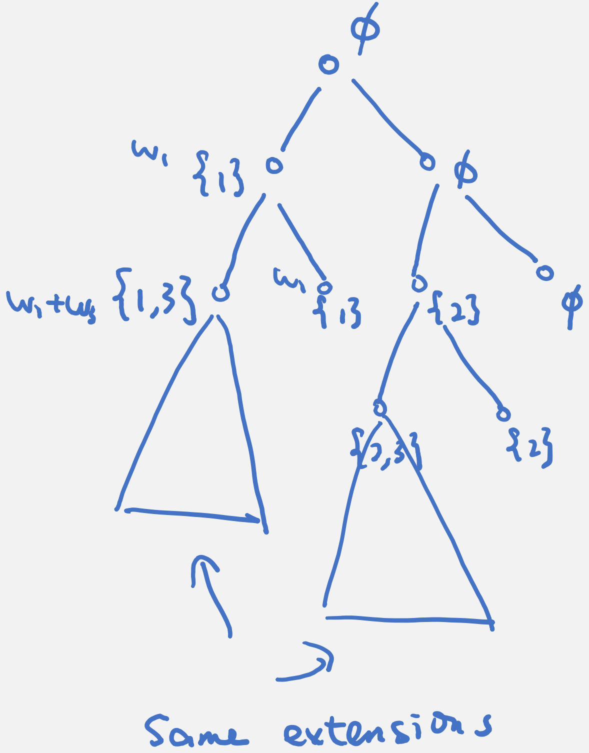
To come up with a good recurrence, first we see how exhaustive search is wasteful and how to improve it.

sort by starting time



last interval chosen

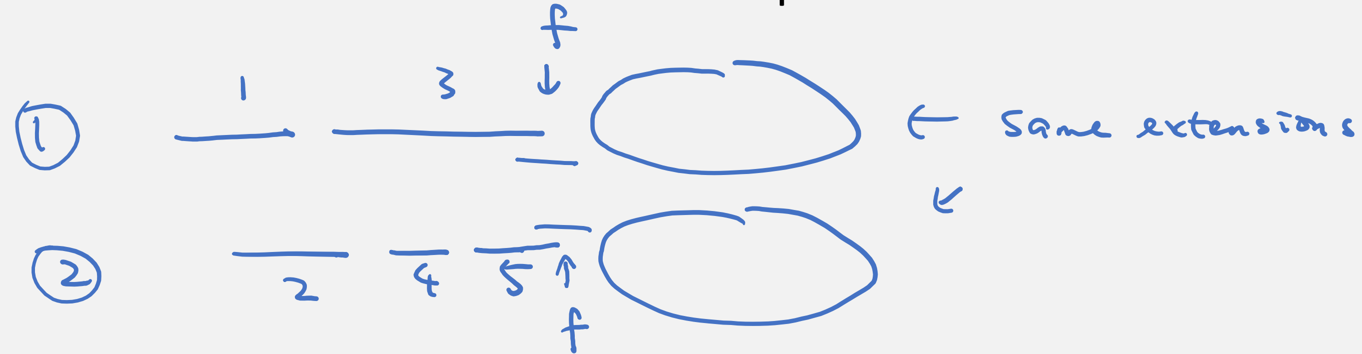



exponential time

# Extending Partial Solutions

We do not need to remember the partial solution that we have chosen so far.

What really matters is the last interval of the current partial solution.




weight of the current partial solution

+ finishing time of the last interval

This suggests that there should be a recurrence with only one parameter!

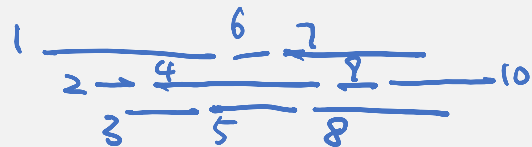
# Good Ordering

For the recurrence, we use a good ordering of the intervals and pre-compute some useful information.

We sort the intervals by starting time so that $s_1 \leq s_2 \leq \cdots \leq s_n$.

For each interval $i$, we define $next(i)$ to be the smallest $j$ such that $j > i$ and $s_j > f_i$

next(i) = n+1 if such a j doesn't exist




next(1) = 6    next(2) = 4

next(3) = 5    next(4) = 9

next(5) = 8    next(6) = 7    . . .

property: for interval i, intervals in {i+1, ..., next(i)-1} overlap with i

(sort by starting time)    intervals in {next(i), ..., n} not overlap with i

# Better Recurrence

Let $opt(i)$ be the maximum total weight of disjoint intervals using intervals in $\{i, \ldots, n\}$ only. $1 \le i \le n$

So, $opt(1)$ is the answer that we would like to compute.

① choose interval 1.

$\Rightarrow$ intervals in $\{2, \ldots, next(1)-1\}$ cannot be overlap.

choose intervals in $\{next(1), \ldots, n\}$ ← opt way to choose disjoint intervals

maximum profit $= w_1 + opt(next(1))$

② not choose interval 1

max profit $= opt(2)$

$$opt(1) = \max\{w_1 + opt(next(1)),\ opt(2)\}.$$

$$opt(i) = \max\{w_i + opt(next(i)),\ opt(i+1)\}.$$

# Algorithm

1. Sort the intervals by non-decreasing starting time so that $s_1 \le s_2 \le \dots \le s_n$.
2. Compute next[i] for $1 \le i \le n$. Set visited[i] = false for $1 \le i \le n$.
3. Return opt(1).

```
opt(i)     // recursive function
    if i = n+1, return 0.          // base case
    if visited[i] = true, return answer[i].
    answer[i] = max { w_i + opt(next[i+1]), opt(i+1) }.
    visited[i] = true.
    return answer[i].
```

# Analysis

Correctness : explanations of recurrence relation. including base cases.

Time complexity : n subproblems

each subproblem, look up two values

$\Rightarrow$ recursion opt(i) can be solved in $O(n)$ time.

sorting $O(n \log n)$ time

next $O(n \log n)$ time $\leftarrow$ binary search

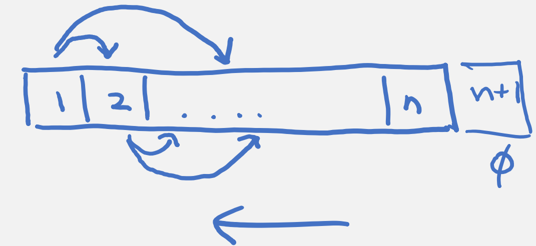
total : $O(n \log n)$

# Bottom-Up Computation

$opt(n+1) = 0$

for $i$ from $n$ downto $1$ do

$$opt(i) = \max\{ w_i + opt(next[i]),\ opt(i+1) \}.$$




This has the same time complexity as the greedy algorithm!

Somehow we can implement an exhaustive search algorithm as efficient as a greedy algorithm!

This is why dynamic programming is so useful and powerful, because it is very systematic
and yet it provides very competitive algorithms.

**Exercise**: Write a program to print out an optimal solution (i.e. the disjoint intervals).

# Today's Plan

1. Framework

2. Weighted interval scheduling

3. Subset-sum and knapsack

Hw 3

midterm

# Subset-Sum

**Input**: $n$ positive integers $a_1, a_2, \dots, a_n$ and an integer $K$.

**Output**: a subset $S \subseteq [n]$ with $\sum_{i \in S} a_i = K$, or report that no such subset exists.

1, 3, 10, 12, 14

$K = 27$? YES $\{3, 10, 14\}$

$K = 29$? YES $\{3, 12, 14\}$

$K = 31$? NO

general version $\sum_{i \in S} a_i \le K$ but maximizes $\sum_{i \in S} a_i$

# Knapsack

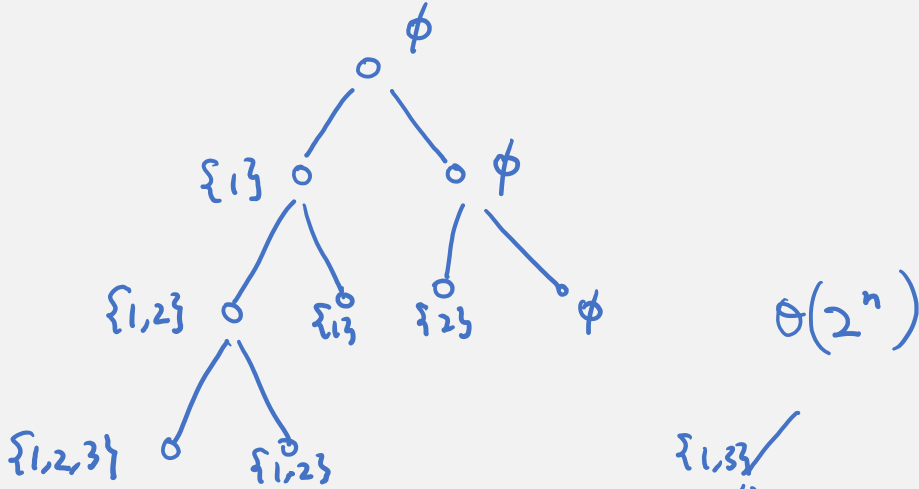
**Input**: $n$ items, each of weight $w_i$ and value $v_i$, and a positive integer $W$.

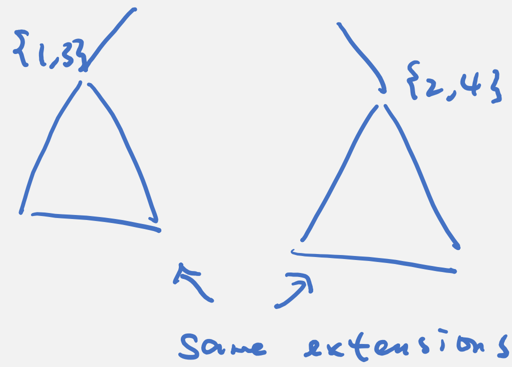
**Output**: a subset $S \subseteq [n]$ with $\sum_{i \in S} w_i \leq W$ that maximizes $\sum_{i \in S} v_i$.

n jobs $w_i$ = time $v_i$ = income

Subset-sum is a special case where $w_i = v_i$

# Exhaustive Search




$a_1 = 3 \quad a_2 = 2 \quad a_3 = 7 \quad a_4 = 8 \mid \ldots$

① $a_1$ and $a_3$ partial sum = 10

② $a_2$ and $a_4$ partial sum = 10

# "Better" Recurrence

The subproblems are $subsum[i][L]$ for $1 \leq i \leq n$ and $1 \leq L \leq K$,

where $subsum[i][L] = true$ iff there is a subset in $\{i, \ldots, n\}$ whose sum is $L$.

answer = subsum[1][K]

compute subsum[i][L]

① choose $a_i$

return true if subsum[i+1][L − $a_i$]

② not choose $a_i$

return true if subsum[i+1][L]

subsum[i][L] = subsum[i+1][L−$a_i$] OR subsum[i+1][L].

# Algorithm

subsum (i, L)

if ( L = 0 ) return true

if ( i > n ) return false // considered all numbers

if ( L < 0 ) return false // over the target

Subsum (i, L) = subsum ( i+1, L − $a_i$ ) OR subsum ( i+1, L ).

# Time Complexity

Correctness : recurrence including base cases

time : $nK$ subproblems

each subproblem , look up two values

total : $O(nK)$.

↑

pseudo-polynomial

$K$ could be $> 2^n$ , worst than naive

NP-complete

# Bottom-Up Computation

subsum[i][L] = false for all $1 \le i \le n$ and $0 \le L \le K$

subsum[n][$a_n$] = subsum[n][0] = true

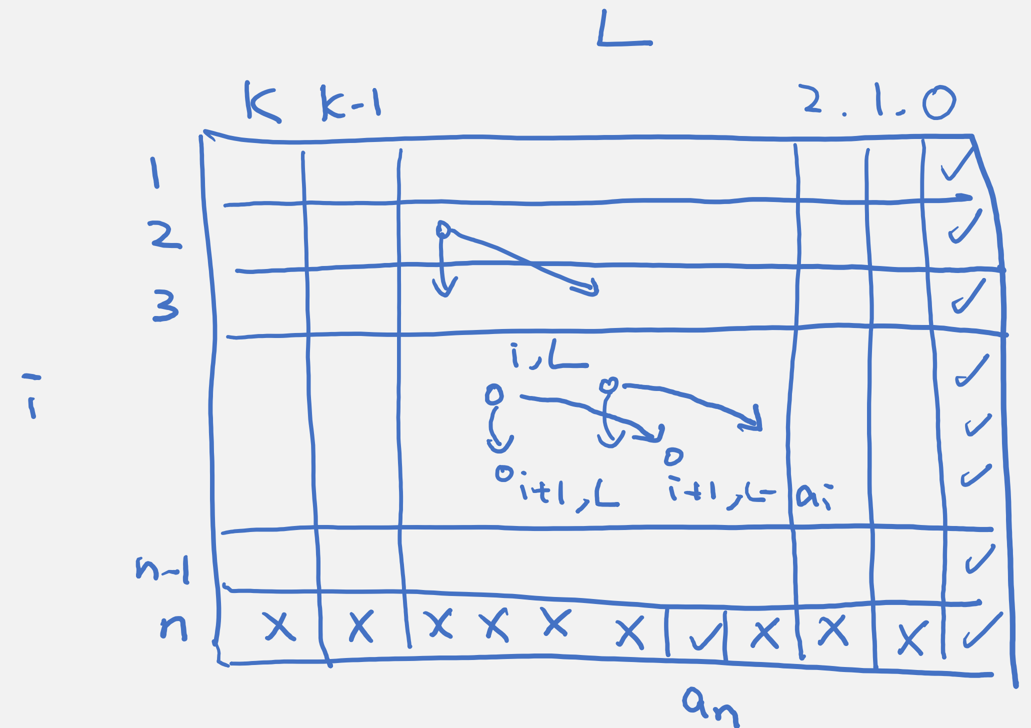
subsum[i][0] = true for all $1 \le i \le n$

for i from n downto 1 do

for L from 1 to K do

if subsum[i+1][L] = true, then subsum[i][L] = true.

if ( $L - a_i \ge 0$ and subsum[i+1][$L-a_i$] = true ), then subsum[i][L] = true.

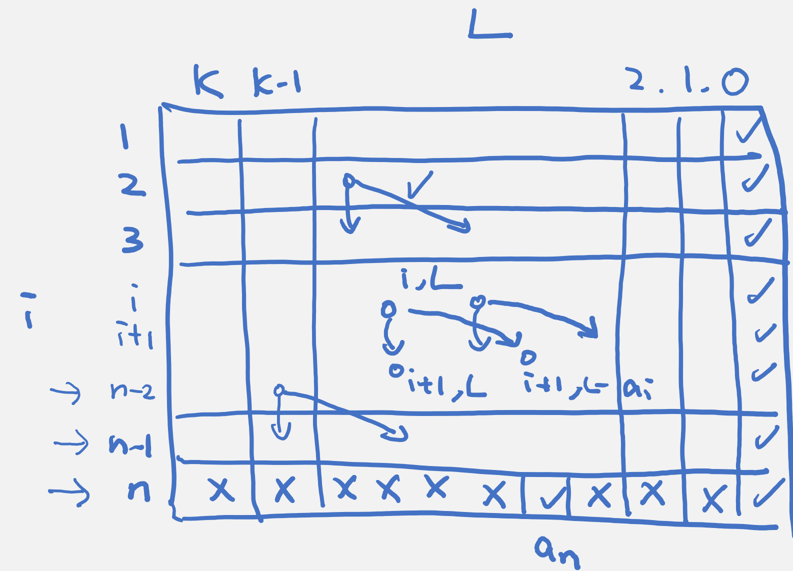


# Top-Down vs Bottom Up

Bottom-up: space efficient implementation

- if decision problem,
  keep two rows
  space $O(K)$, instead of $\Theta(nk)$.
- no recursion
- runtime exactly nk.

top-down: - space $\Theta(nk)$, time $\Theta(nk)$
- easier to write
- doesn't necessarily solve all the subproblems
- take an element first may speed up



# Dynamic Programming and Graph Search

This algorithm can be understood as doing a graph search on the “subproblem dependency graph”.

subsum[1, k] true iff ∃ a directed path from (1, k) to some nodes with “true” base cases.

print a solution: ① parent array ② look at table

# Knapsack: First Approach

**Input**: $n$ items, each of weight $w_i$ and value $v_i$, and a positive integer $W^*$.

**Output**: a subset $S \subseteq [n]$ with $\sum_{i \in S} w_i \leq W^*$ that maximizes $\sum_{i \in S} v_i$.

The subproblems are $knapsack(i, W, V)$, which is true if and only if there is a subset in $\{i, \dots, n\}$ with total weight $W$ and total value $V$.

answer = $\max\{ V \mid knapsack(1, W, V) = \text{true} \text{ and } W \leq W^* \}$

recurrence $knapsack(i, W, V) = knapsack(i+1, W, V)$ OR $knapsack(i+1, W - w_i, V - v_i)$

base cases $knapsack(i, 0, 0)$ true

$knapsack(n+1, \neq 0, \neq 0)$ false

time complexity

$O(n \cdot W^* \cdot \sum_{i=1}^{n} v_i)$

# Better Recurrence

knapsack ( i, w ,201 ) = true
knapsock (i,w, 200) ← don't care

We don't need to store all possible $V$, we just need to store the maximum $V$ achievable.

Define $knapsack(i, W)$ as the maximum value of a subset in $\{i, \ldots, n\}$ with total weight at most $W$.*

More precisely, let $knapsack(i, W) \coloneqq \max_{S \subseteq \{i,\ldots,n\}} \left\{ \sum_{j \in S} v_j \mid \sum_{j \in S} w_j \le W \right\}$.

answer = knapsack (1, w)

recurrence knapsack(i, w) = max { knapsack(i+1, w) [not choose i], $v_i$ + knapsack (i+1, w−$w_i$) [choose i] }

base cases:

knapsack (n+1, ≥0) = 0

knapsack ( i , <0 ) = −∞

# Remaining Steps

The following details are left as exercises.

- Base cases.
- Correctness.
- Time Complexity.
- Top-down implementation. top-down memorization
- Bottom-up implementation.
- Printing a solution.

# Check List for Dynamic Programming Solution

The following is a check list for designing a dynamic programming algorithm.

1. Starting with exhaustive search to get an idea (we may skip this step).

2. Coming up with good subproblems.

3. Write down how to get the answer from the subproblems.

4. Write down the recurrence relations including base cases, with explanation as correctness proof.

5. Write down the algorithm (enough to say top-down memorization).

6. Analyze the time complexity.

7. Print out a solution if necessary.

8. Bottom-up implementation if necessary (e.g. to avoid stack overflow).