EECS 660: FUNDAMENTALS OF COMPUTER ALGORITHMS

MODULE VII: DYNAMIC PROGRAMMING, PART I

DISCLAIMER

- This document is intended to be used for studying EECS 660 Fundamentals of Computer Algorithms, only by students who are currently enrolled in the course.

- This document is a copyright of Dr. Cuncong Zhong. Distribution of this document, or use it for any purpose other than what is stated above, is considered as a copyright infringement. Dr. Cuncong Zhong reserves the right to take necessary legal action.

- If you disagree, please delete the document immediately.

ACKNOWLEDGEMENT

- Many of the figures, unless otherwise stated, come from ***Algorithm Design*** 4th edition, by Jon Kleinberg and Eva Tardos and ***Analysis of Algorithms*** 2nd edition, by Jeffery McConnell.

OUTLINE

- A motivating case: Fibonacci number revisited
- Dynamic programming: three important features
  - overlapping subproblems and memorization
  - optimal substructure
- The knapsack problem revisited
  - optimal substructure explained
  - traceback
  - pseudo-polynomial time complexity

FIBONACCI NUMBER REVISITED

- Recall the Fibonacci problem, where we are trying to compute $f(n) = f(n-1) + f(n-2)$ where $n > 2$, where $f(1) = f(2) = 1$. We can break the computation into subproblems and solve it with a simple recursive algorithm.

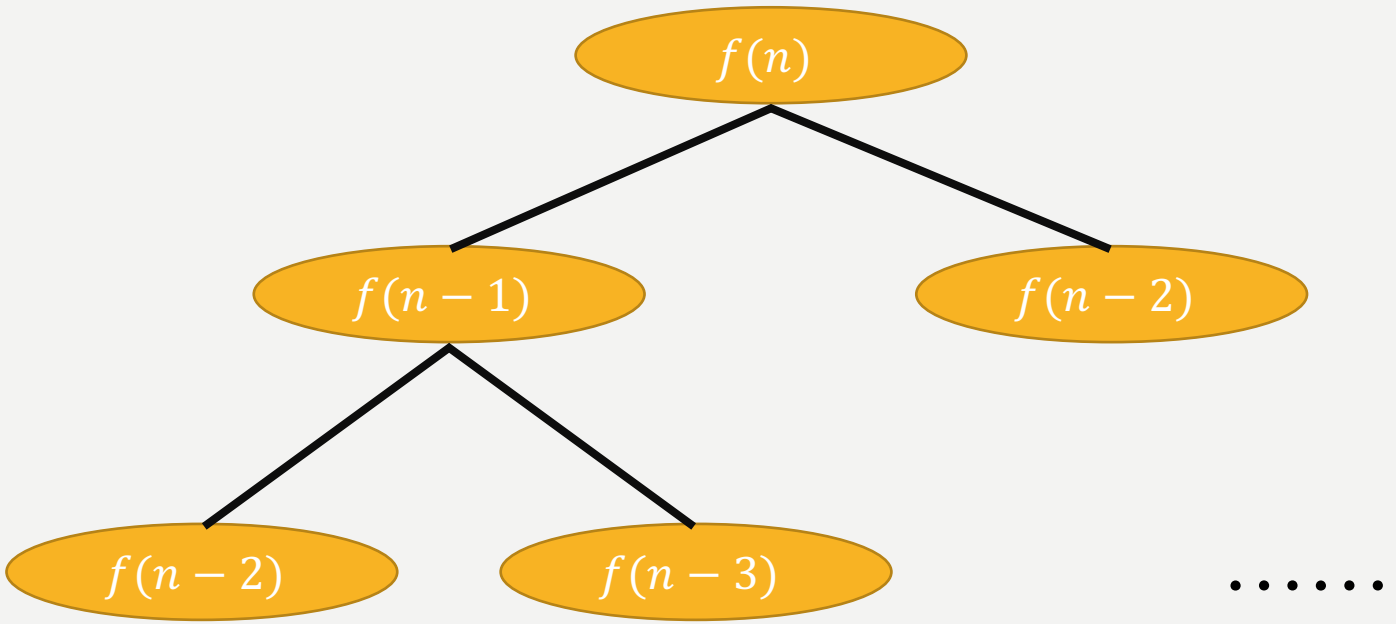
Fibo(n)

If $n < 2$ return 1;

return Fibo($n-1$)+Fibo($n-2$)

FIBONACCI NUMBER REVISITED

- And by unrolling the recursion, we know the time complexity for the algorithm is $O(2^n)$.



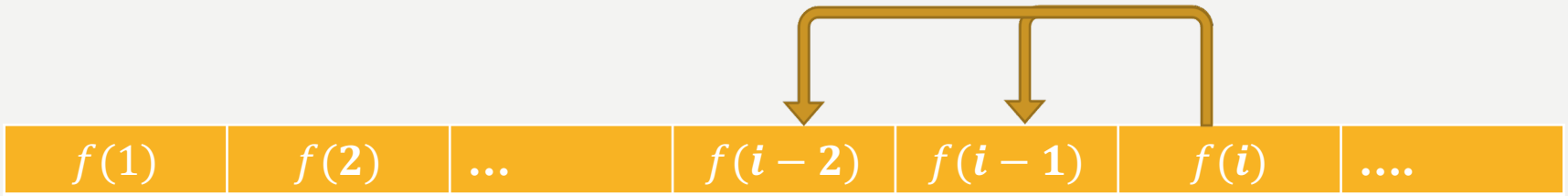
FIBONACCI NUMBER REVISITED

- In fact, there exists a smarter algorithm to compute the Fibonacci number $f(n)$
  - we observe that, to compute $f(n)$, the total number of subproblems we need throughout the entire recursion is $O(n)$, i.e., $f(1), f(2), \ldots, f(n-1)$

- We can simply compute and **memorize** the results of these subproblems, which will lead to a very simple algorithm that runs in $O(n)$.

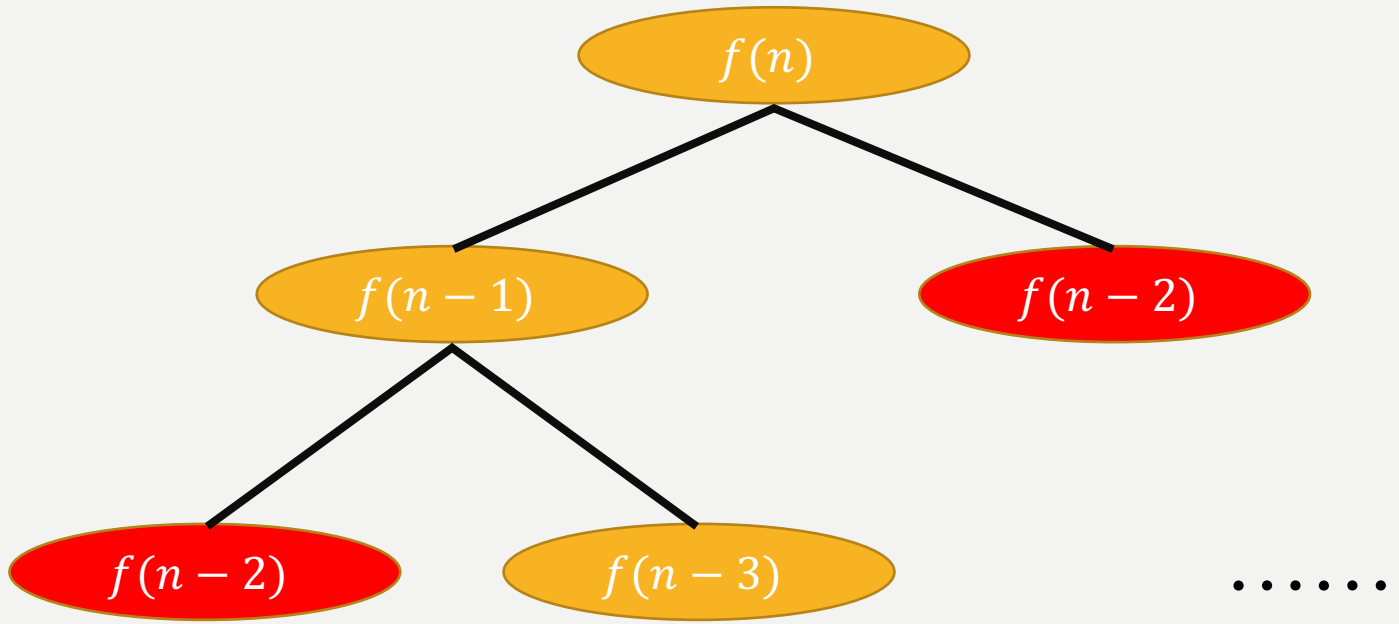
FIBONACCI NUMBER REVISITED

- We simply use a one-dimensional array to store the subproblems $f(1), f(2), \ldots, f(n-1)$. When computing $f(i)$, we simply add up its previous two numbers $f(i-2)$ and $f(i-1)$.



FIBONACCI NUMBER REVISITED

- How are we able to achieve the speedup? Well, we simply **avoid the re-computations of overlapping subproblems**.



DYNAMIC PROGRAMMING

- The above algorithm for computing the Fibonacci number illustrate the major idea of **dynamic programming**. (Here, the term "programming" actually means "planning", or a planning strategy that dynamically adapts to what we have in reality.).

- The idea of identifying overlapping subproblems, solve each of them only once and record the result to avoid re-computation (i.e., **memorization**), is key to the **higher efficiency** of dynamic programming algorithms (compared to direct recursion).
  - if you have developed a recursive or divide and conquer algorithm, try to identify overlapping subproblems and see if it can be solved using dynamic programming
  - we trade space (because we need to memorize the results of the subproblems) for time in dynamic programming, so **we need to analyze both time and space complexity**

OPTIMAL SUBSTRUCTURE

- Another question we need to ask (and in fact need to ask ever since we started learning greedy algorithms and divide and conquer algorithm) is: **which property qualifies a proper subproblem**?

- The sole property that properly defines a subproblem is called **optimal substructure**:
  - optimal structure means that the solution to the larger problem can be obtained from the solutions of its corresponding subproblems
  - this property is also called **subproblem optimality**

OPTIMAL SUBSTRUCTURE

- **Some subproblems have optimal substructure, some subproblems do not:**
  - recall the shortest path problem: a subproblem, defined by any subset of nodes that have been visited (S vs. $V - S$) when traversing from the source, has optimal substructure
  - this is because the shortest paths identified from the subproblem must be the prefixes of the shortest paths identified for larger graph components
  - note that per the optimal substructure definition, "solution to the larger problem can be obtained from the solutions of its corresponding subproblems"; here, **"obtain" does not necessarily mean "directly obtain"**

OPTIMAL SUBSTRUCTURE

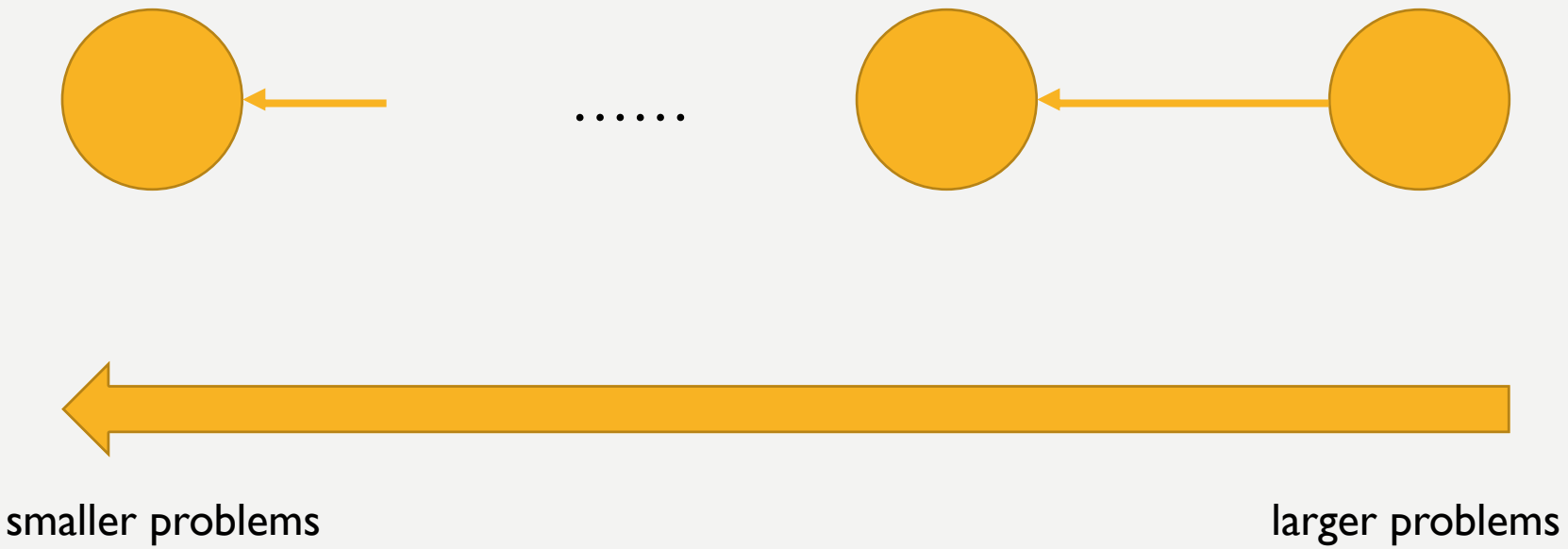
- **Some subproblems have optimal substructure, some subproblems do not:**
  - a subproblem for the stable matching, defined by a subset of men and a subset of women (with matching numbers of men and women), does not have the optimal structure
  - consider you have found a stable matching for the subproblem; then you add in another man and another woman. The newly added man and woman are so charming that they rank the top on everyone else's list. Then, everyone will be willing to break the current matching and match with the new man and woman
  - the already-constructed matching on the subproblem does not mean anything for larger problems

OPTIMAL SUBSTRUCTURE

- **The algorithm design scheme we have learnt so far, including greedy algorithms, divide and conquer algorithms, and dynamic programming algorithms, should all have optimal substructure**.
  - but they do have different substructure in terms of their subproblems
  - when one problem can be solved by greedy algorithms, it implies that solving a larger problem of it only requires the solution of one subproblem
  - when one problem can be solved by divide and conquer algorithms, it implies that solving a larger problem of it only requires the solution of some subproblems
  - when one problem can be solved by dynamic programming algorithms, it implies that solving a larger problem of it only requires the solution of some subproblems, and these subproblems are overlapping

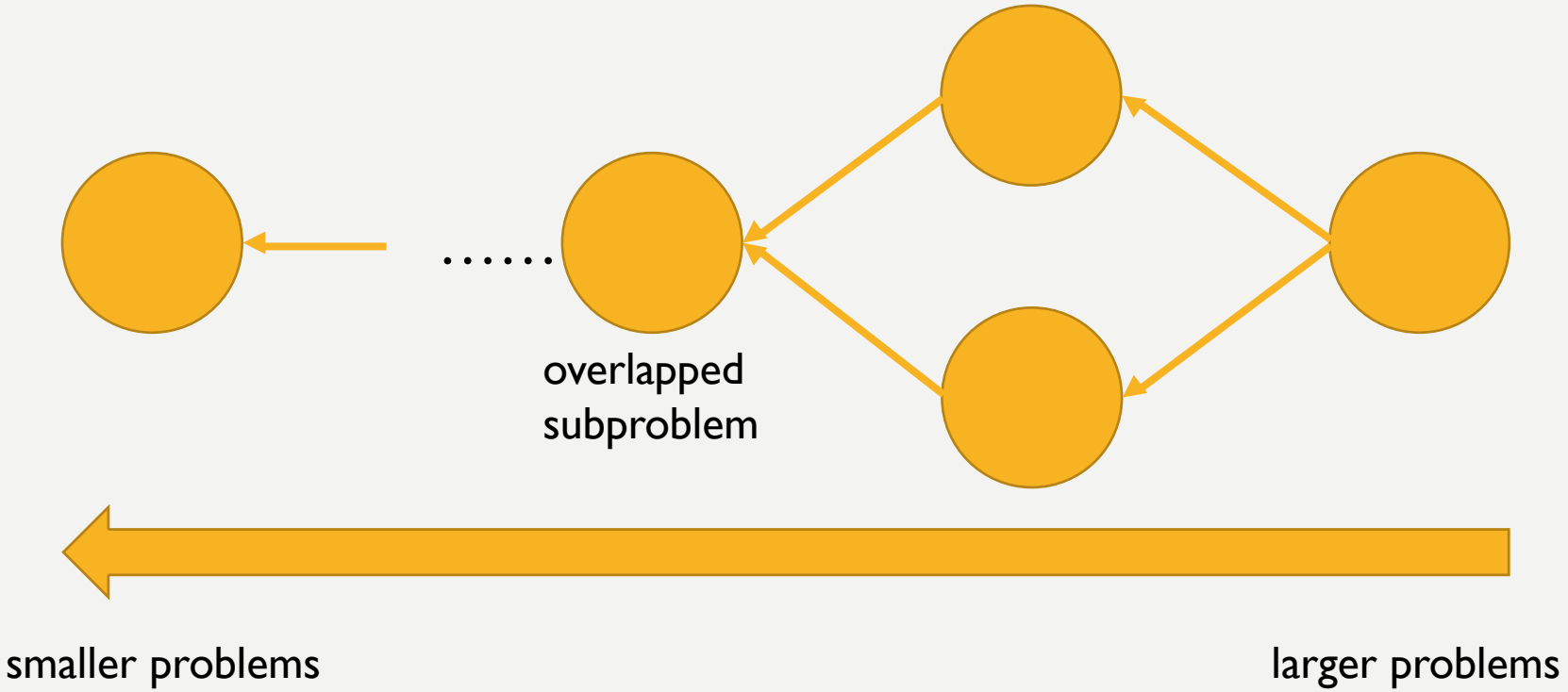
OPTIMAL SUBSTRUCTURE

- Greedy algorithms



OPTIMAL SUBSTRUCTURE

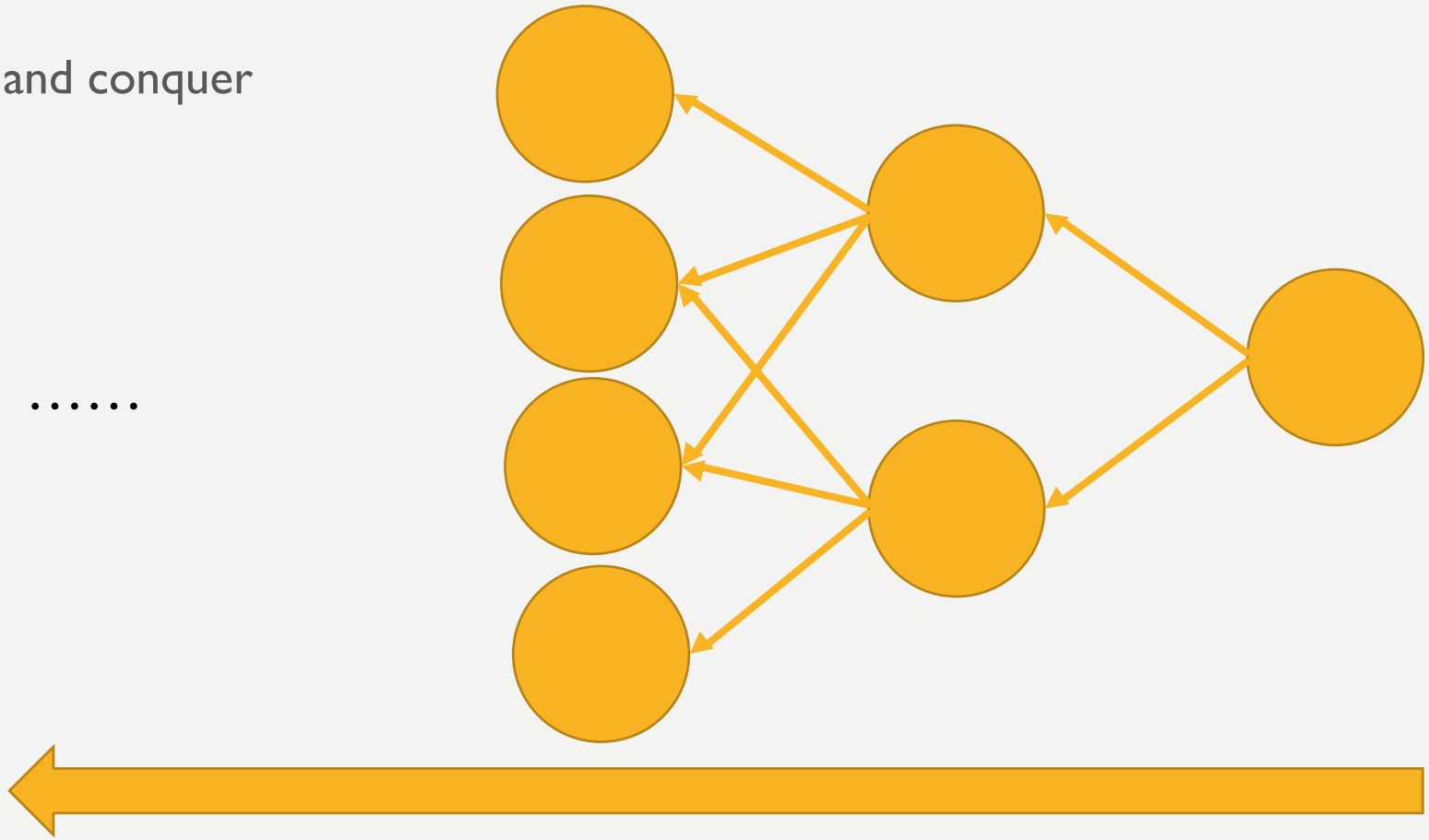
- Dynamic programming



OPTIMAL SUBSTRUCTURE

- Divide and conquer

.....



smaller problems

larger problems

OPTIMAL SUBSTRUCTURE

- In terms of power and generalizability:
  - **divide and conquer > dynamic programming > greedy**
  - for example, if a problem can be solved using greedy algorithm, you can also formulate it in the way of dynamic programming or divide and conquer; but not vice versa

- In terms of efficiency:
  - **greedy > dynamic programming > divide and conquer**

OPTIMAL SUBSTRUCTURE

- Note that:
  - the optimal substructure is not tied to a specific problem
  - it is tied to a specific problem and a specific definition of the subproblem
  - in other words, **for the same problem, some definitions of subproblems does not have optimal substructure, and some definitions do**
  - we will illustrate this idea via revisiting the knapsack problem

THE KNAPSACK PROBLEM REVISITED

- Recall the Knapsack problem:

- You are given a total capacity C, a set of items I, each $i \in I$ is assigned with a weight w_i and a value v_i. The problem seeks to pack a subset of items $I' \subseteq I$, such that:

$$\sum_{i \in I'} w_i \leq C$$

and that the total value

$$\sum_{i \in I'} v_i$$

is maximized.

THE KNAPSACK PROBLEM REVISITED

- And we have shown that greedy algorithm does not work on this case.

- Consider the following instance:

$$C = 12$$
$$w_1 = 8, v_1 = 16$$
$$w_2 = 6, v_2 = 13$$
$$w_3 = 6, v_3 = 10$$
$$w_4 = 5, v_4 = 7$$

THE KNAPSACK PROBLEM REVISITED

- Now, let's try to define the subproblem:
  - we could define the subproblem as the same capacity with a subset of items $(C, I' \subset I)$
  - we could define the subproblem as the same set of items with a smaller capacity $(C' < C, I)$
  - we could define the subproblem as a smaller capacity with a subset of items (C', I')

THE KNAPSACK PROBLEM REVISITED

- The same capacity with a subset of items (C, I')?
  - imagine the item set contains an item i, where $w_i = C, v_i = \infty$, and the subset of items defining the subproblem does not contain the item i
  - what will happen is that no matter what we have chosen for that subproblem, the solution is useless.
  - because the solution to the subproblem clearly does not contain i (not in the subset), while the optimal answer clearly does

THE KNAPSACK PROBLEM REVISITED

- A smaller capacity with the same of items (C', I)?
  - again imagine the item set contains an item i, where $w_i = C, v_i = \infty$, the weight of item i is larger than the capacity defining the subproblem, i.e., C'
  - what will happen is that no matter what we have chosen for that subproblem, the solution is useless.
  - because the solution to the subproblem clearly does not contain i (overweight w.r.t the subproblem capacity C'), while the optimal answer clearly does

THE KNAPSACK PROBLEM REVISITED

- A smaller capacity with a subset of items (C', I')?
  - now, consider another larger problem $(C' + w_i, I' + i)$
  - the solution to this problem, conditioned on that we take item i, is nothing else than including i into the solution obtained for (C', I')
  - this is because, under that condition, we are left with a capacity C' and an item set I', which define the original subproblem
  - caveat: notice that the condition of taking item i does not necessarily lead to the optimal solution

THE KNAPSACK PROBLEM REVISITED

- A smaller capacity with a subset of items (C', I')?
  - again, consider a different larger problem $(C', I' + i)$
  - the solution to this problem, conditioned on that we do NOT take item i, is nothing else than just solution obtained for (C', I')
  - this is because, under that condition, the element i will be excluded from the item set $I' + i$, which defines the original subproblem
  - caveat: notice that the condition of not taking item i does not necessarily lead to the optimal solution either

THE KNAPSACK PROBLEM REVISITED

- A smaller capacity with a subset of items? (cont.)
  - however, we know that the optimal solution must be a specific combination of items
  - for item i, the combination translates to just two conditions
    - conditioned on taking i
    - conditioned on not taking i
  - clearly, the solution of the subproblem (C', I') can be directly used without change to construct the solution $(C' + w_i, I' + i)$ (note: the solution to the subproblem is necessary, but not sufficient)
  - hence, **the subproblem with a smaller capacity and a subset of items does have an optimal substructure**

THE KNAPSACK PROBLEM REVISITED

- Now that we have defined a proper subproblem, the next step would be writing down the recursive function to mathematically describe the relationship between the subproblem and the larger problem.

- We know that:
  - the solution of (C', I') allows us to compute $(C' + w_i, I' + i)$ when i must be included
  - the solution of (C', I') allows us to compute $(C', I' + i)$ when i must not be included
  - but note that recursive functions are written "**backward**", that is, how we compute the larger problem using solutions of the smaller problem, not in the forward way that how we use the solutions of smaller problems to construct the larger problem

THE KNAPSACK PROBLEM REVISITED

- Hence, we can write down the recursive function as the follows:

$$Value(C, I) = max \begin{cases} Value(C - w_i, I - i) + v_i \\ Value(C, I - i) \end{cases}$$

- You might argue that why should we consider item i first, but not item j?
  - it does not matter because putting item i into the knapsack first followed by item j has the same effect as because putting item j into the knapsack first followed by item i
  - so, the order of the items does not matter

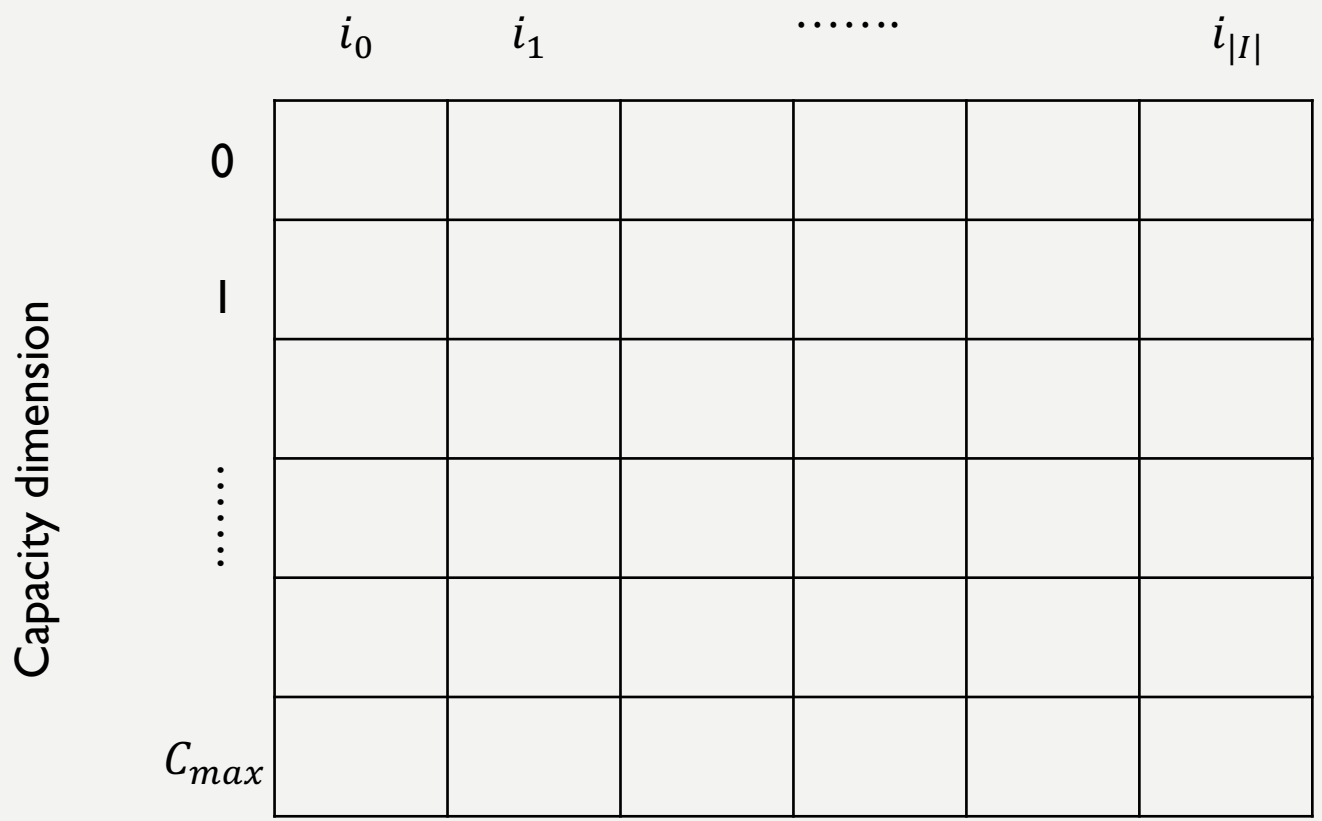
THE KNAPSACK PROBLEM REVISITED

- Now, we shall set up the detailed algorithm to compute the recursive function.

- First, we shall look at the recursive function, and we realize that a subproblem is defined by two factors, i.e., C and I. Hence, if we need to memorize the results of the subproblems, we should use a two-dimensional table
  - you can also look at the function we are computing, i.e., $Value(C, I)$
  - you can simply treat $Value$ as a matrix and (C, I) as its indexes
  - so, it is clearly a 2D matrix

THE KNAPSACK PROBLEM REVISITED

Item dimension: recall that the order does not matter



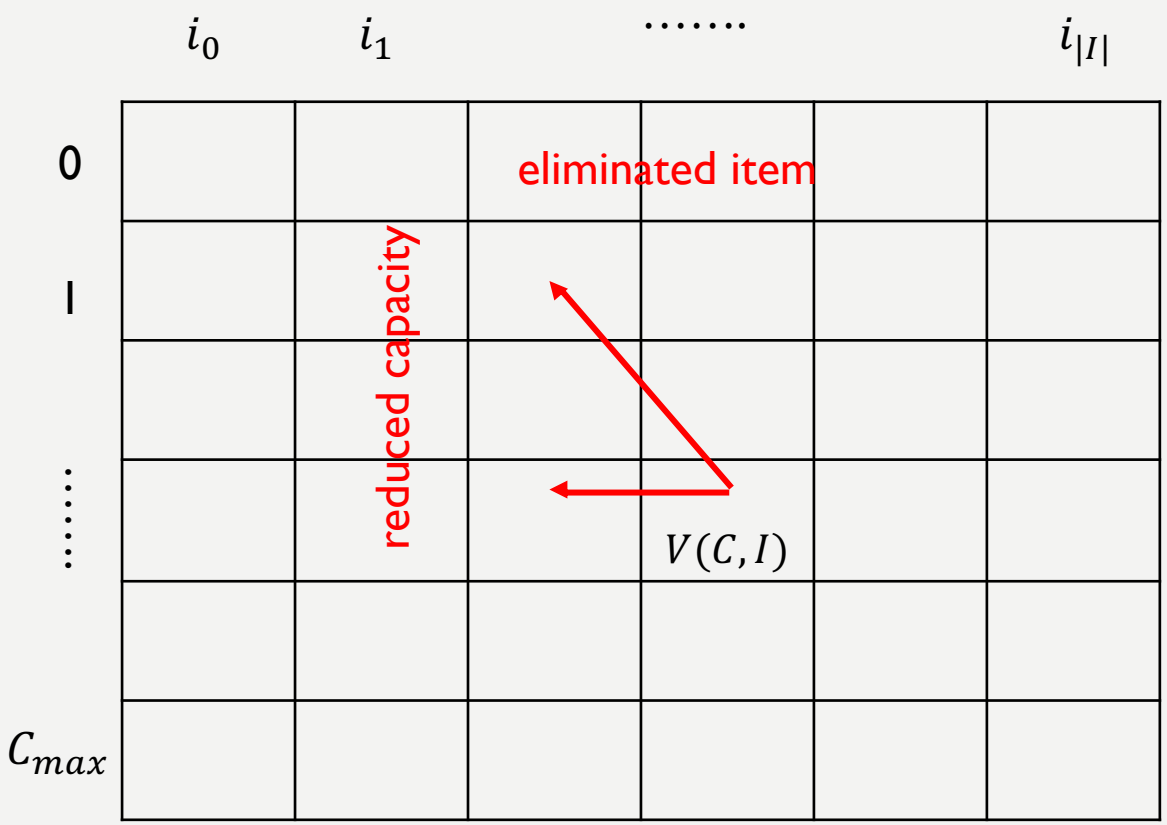
The matrix used for storing subproblem results is also called **dynamic programming table**.

THE KNAPSACK PROBLEM REVISITED

- The next step is to figure out the dependency of subproblem results
  - the dependency is clear in the recursive function, but we are better off if we can visualize it in the context of the dynamic programming table
  - recall the recursive function:

$$Value(C, I) = max \begin{cases} Value(C - w_i, I - i) + v_i \\ Value(C, I - i) \end{cases}$$

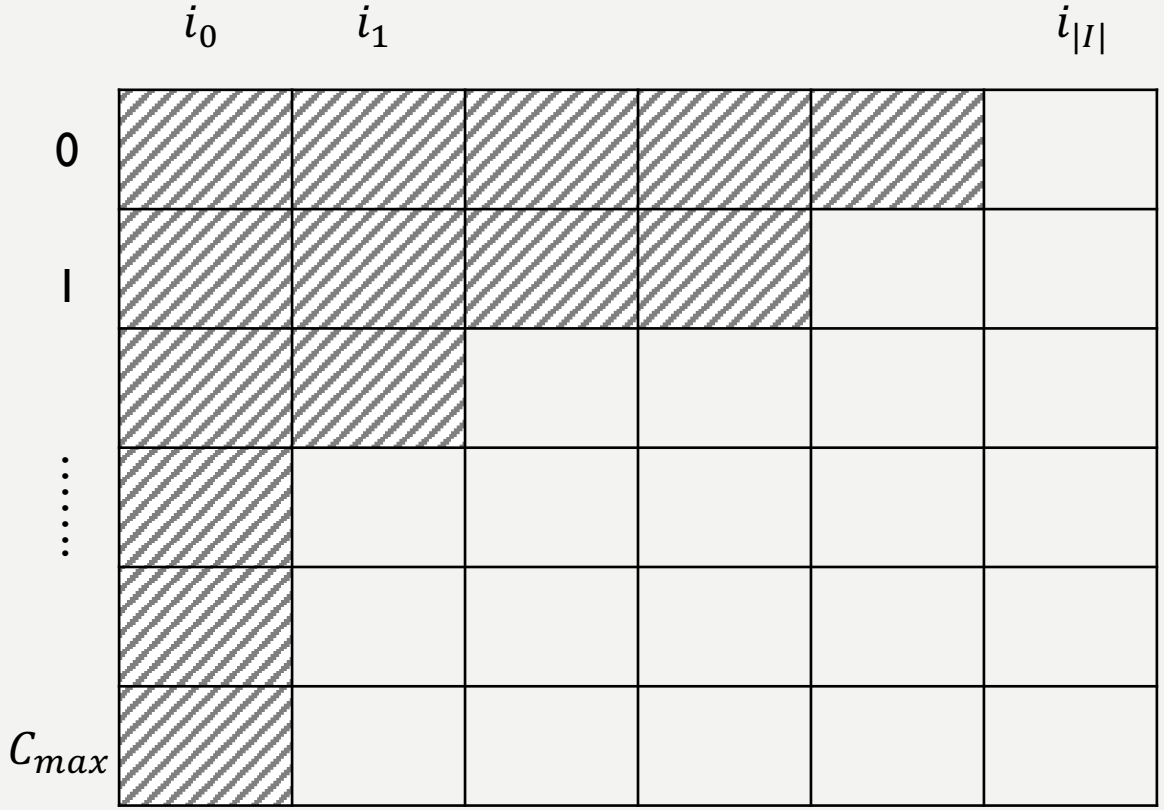
THE KNAPSACK PROBLEM REVISITED



$$Value(C,I) = \begin{cases} Value(C - w_i, I - i) + v_i \\ Value(C, I - i) \end{cases}$$

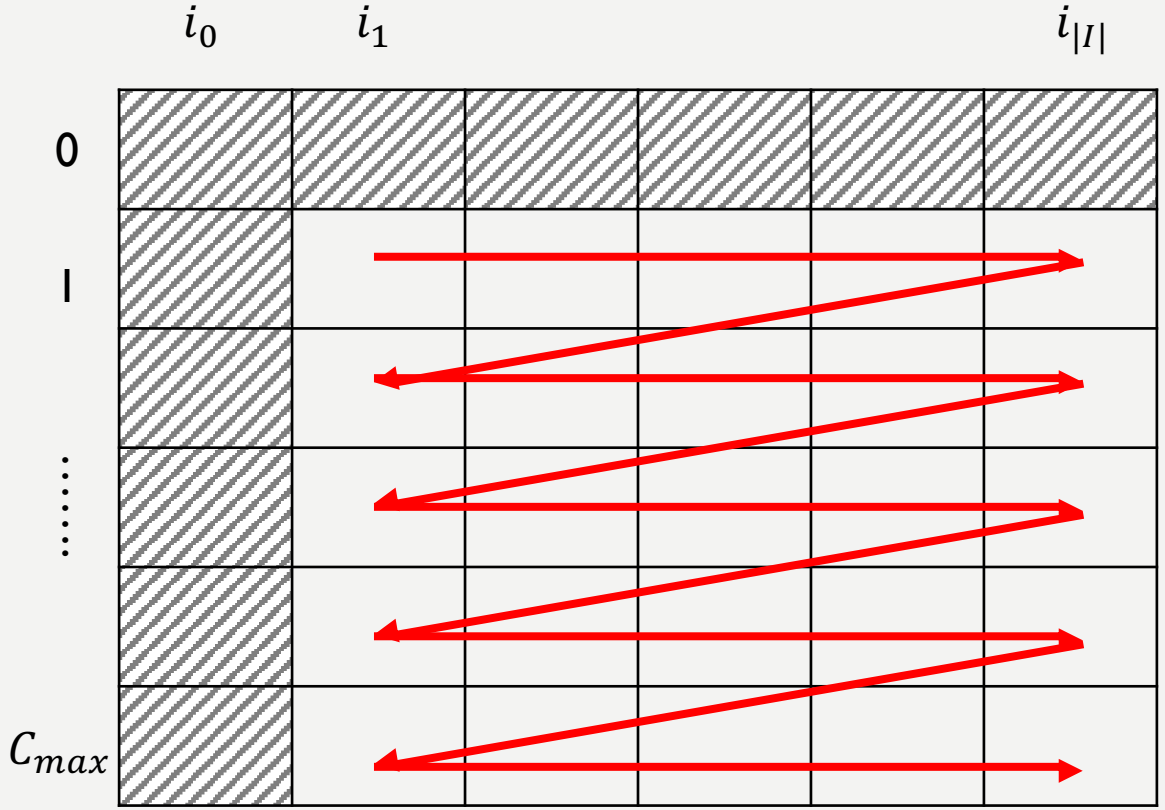
THE KNAPSACK PROBLEM REVISITED

- According the the data dependency in the dynamic programming table, conceptually we should first compute the "northwestern" data blocks before marching "southeast".



THE KNAPSACK PROBLEM REVISITED

- A more "programmable" way to describe the dependency is that, if we have computed the first column and the first row, we should be able to complete the table row-by-row.



THE KNAPSACK PROBLEM REVISITED

- But how do we fill up the first column and the first row?
  - in many cases, the first column and the first row would correspond to special cases of the problem
  - depending on the definition of the dynamic programming table and the specific indexes, we should come up with a **meaningful** way to **initialize** the table
  - in our example, the first row indicates the value we can have if the knapsack has 0 capacity; the answer would obviously be 0, since we cannot put anything inside (in special cases with 0-weighted item, we can initialize the first row as their values' sum)
  - for the first column, in indicates the scenario when we only one item i_0 in the set; clearly, for cells with capacities less than w_0, we cannot take the item and the values would be 0; for cells with capacities equal or greater than w_0, we can take the item and the values would be v_0

THE KNAPSACK PROBLEM REVISITED

- Now we will solve the knapsack instance we had as an example of the algorithm:

$$C = 12$$
$$w_1 = 8, v_1 = 16$$
$$w_2 = 6, v_2 = 13$$
$$w_3 = 6, v_3 = 10$$
$$w_4 = 5, v_4 = 7$$

THE KNAPSACK PROBLEM REVISITED

	i_1	i_2	i_3	i_4
0	0	0	0	0
1	0			
2	0			
3	0			
4	0			
5	0			
6	0			
7	0			
8	16			
9	16			
10	16			
11	16			
12	16			

THE KNAPSACK PROBLEM REVISITED

	i_1	i_2	i_3	i_4
0	0	0	0	0
1	0	$\max(0,0)$		
2	0			
3	0			
4	0			
5	0			
6	0			
7	0			
8	16			
9	16			
10	16			
11	16			
12	16			

note: the first 0 comes from the lest cell, the second 0 comes from the top-left, which does not exist and is assigned with 0

THE KNAPSACK PROBLEM REVISITED

	i_1	i_2	i_3	i_4
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	$\max(0,0+v_4)$
6	0			
7	0			
8	16			
9	16			
10	16			
11	16			
12	16			

note: the first 0 comes from the lest cell, the second 0 comes from the top-left (red cell), which is within the matrix and we can add the value of the corresponding item

THE KNAPSACK PROBLEM REVISITED

	i_1	i_2	i_3	i_4
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	7
6	0	13	13	13
7	0	13	13	13
8	16	16	16	16
9	16	16	16	16
10	16	16	16	16
11	16	16	16	20
12	16	16	23	23

note: the last entry, which corresponds to $Value(C_{max}, I)$, holds the final solution to the entire problem

THE KNAPSACK PROBLEM REVISITED

- The algorithm tells us that the maximum value we can get is 23; and we can easily verify that it is the correct answer.

$$C = 12$$
$$w_1 = 8, v_1 = 16$$
$$w_2 = 6, v_2 = 13$$
$$w_3 = 6, v_3 = 10$$
$$w_4 = 5, v_4 = 7$$

THE KNAPSACK PROBLEM REVISITED

- However, the above algorithm only tells us the maximum value we can achieve, but it does not tell which items should be taken to achieve that maximum value.
- This is common for most of dynamic programming algorithms, that is, they can compute the optimal value, but in its naïve form does not produce the evidence that leads to the optimum.
- We can include an additional step, called **traceback**, to retrieve the evidence.
  - we also need to record auxiliary information in order to use traceback
  - specifically, for each cell, from which subproblem leads to the current solution

THE KNAPSACK PROBLEM REVISITED

	i_1	i_2	i_3	i_4
0	0 (I)	0 (I)	0 (I)	0 (I)
1	0 (I)	0 (L)	0 (L)	0 (L)
2	0 (I)	0 (L)	0 (L)	0 (L)
3	0 (I)	0 (L)	0 (L)	0 (L)
4	0 (I)	0 (L)	0 (L)	0 (L)
5	0 (I)	0 (L)	0 (L)	7 (L)
6	0 (I)	13 (T)	13 (L)	13 (L)
7	0 (I)	13 (T)	13 (L)	13 (L)
8	16 (I)	16 (L)	16 (L)	16 (L)
9	16 (I)	16 (L)	16 (L)	16 (L)
10	16 (I)	16 (L)	16 (L)	16 (L)
11	16 (I)	16 (L)	16 (L)	20 (T)
12	16 (I)	16 (L)	23 (T)	23 (L)

note: "I" indicates the solution is initialized, "L" indicates the solution comes from its left subproblem, "T" indicates the solution comes from its top-left subproblem. Ties are broken to favor "L" (less picked items).

THE KNAPSACK PROBLEM REVISITED

- Once we have stored the auxiliary information, we can begin the traceback.

- However, note that we are trying to reconstruct the evidence that corresponds to the optimal solution of the problem. Therefore, we should begin with the cell that stores the optimal solution. In this case, the very last entry in the bottom-right corner.

THE KNAPSACK PROBLEM REVISITED

	i_1	i_2	i_3	i_4
0	0 (I)	0 (I)	0 (I)	0 (I)
1	0 (I)	0 (L)	0 (L)	0 (L)
2	0 (I)	0 (L)	0 (L)	0 (L)
3	0 (I)	0 (L)	0 (L)	0 (L)
4	0 (I)	0 (L)	0 (L)	0 (L)
5	0 (I)	0 (L)	0 (L)	7 (L)
6	0 (I)	13 (T)	13 (L)	13 (L)
7	0 (I)	13 (T)	13 (L)	13 (L)
8	16 (I)	16 (L)	16 (L)	16 (L)
9	16 (I)	16 (L)	16 (L)	16 (L)
10	16 (I)	16 (L)	16 (L)	16 (L)
11	16 (I)	16 (L)	16 (L)	20 (T)
12	16 (I)	16 (L)	23 (T)	23 (L)

note: we simply follow the L or T pointer, until we reach some I terminal. Every time we follow a T link, we know we should select the corresponding item.

THE KNAPSACK PROBLEM REVISITED

- Correctness: similar to divide and conquer algorithms, the algorithm is correct if the recursion is correct. For this case (and for most cases as well) the correctness if obvious, so we omit the proof here.

- Efficiency:
  - the time complexity is clearly the time for filling up the table plus the time for the traceback
  - to fill the table, we need to compute $O(C|I|)$ entries, while for each entry we need $O(1)$ time
  - for traceback, we are moving either left or up by at least one position, so the time complexity is $O(C + |I|)$
  - the space complexity is clearly $O(C|I|)$

THE KNAPSACK PROBLEM REVISITED

- A note on the time complexity $O(C|I|)$
  - you may think that it is similar to $O(n^2)$
  - however, note that C is a single number rather than a set of items, which need $O(n)$ bits to represent
  - C only needs $O(n)$ bits
  - consider the binary 111, we have used 3 bits to represent $2^3 - 1$
  - now, per definition of time complexity, where the input size is measured by the number of bits on the tap, an n-bit input can represent a number as large as $O(2^n)$
  - so, the time complexity is really $O(2^n|I|)$, which is exponential instead of polynomial

THE KNAPSACK PROBLEM REVISITED

- Algorithms whose time complexity can be written in the form of polynomial but is in fact exponential, are called pseudo-polynomial algorithms.
  - they are in theory classified as exponential algorithms
  - usually, they run faster than strictly exponential algorithms
  - we will discuss the theoretical boundary between polynomial-time complexity and exponential-time complexity later in the class (a new topic termed computational complexity or computational tractability)

SUMMARY

- Fibonacci number revisited
  - a motivating case
- Fundamentals of dynamic programming
  - overlapping subproblems and memorization
  - subproblem optimality
- Knapsack problem revisited
  - defining proper subproblems
  - writing recursion
  - dynamic programming table layout and algorithm design (with proper initialization)
  - traceback
  - pseudo-polynomial