IM 1003: Computer Programming
Algorithms

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Outline

- **Algorithms**
- Combinatorial problems
- The knapsack problem

Algorithms

- There is an old saying:
  Programming design = Data structures + Algorithms.
- While **Data Structures** and **Algorithms** are two advanced courses, in this semester we will give very brief introductions.
- Today let's talk about algorithms.
- What is an algorithm?

Algorithms

- An algorithm is **a sequence of actions** (steps), arranged in a specific order, that completes a **task**.
  - All steps must be **precise** and **executable**.
- E.g., if the task is to "get 100 in the final exam of Calculus", what is an algorithm for this task?
  - "Writing down correct answers on the answer sheet" is not.
  - "Reading the textbook thoroughly", "completing all the exercises", "have a good sleep in the previous night", "go to the classroom on time", and "be relax and confident" look more like an algorithm.
- Let's see some more concrete examples.

Algorithms

- How to find the maximum number in an array?
- An algorithm is:
  - First set the maximum number to 0.
  - For each element in the array, check whether it is larger than the maximum number.
  - If so, replace the maximum number by the current element. Otherwise, do nothing and check the next element.
  - Once all elements are checked, report the resulting maximum number.
- Note that all the steps are precise and executable.

Pseudocodes

- An algorithm is usually described by **pseudocodes**:
  - A description in words that is organized in a programming style.
  - Use selection, repetition, variables, and indices precisely.
- The pseudocode for the previous algorithm is:

> Consider an array A with n elements
> Set *max* to 0.
> **For** i from 1 to n:
>   **If** $A_i > max$
> $max = A_i$.
> Output *max*.

Pseudocode vs. implementation

- A pseudocode describes an algorithm.
  - It **ignores the syntax issue** of a specific programming language.
  - It can be **implemented** by different programming languages.
- For example, in C++:

```
int array[5] = {1, 2, 3, 4, 8};
int max = 0;

for(int i = 0; i < 5; i++)
{
  if(array[i] > max)
    max = array[i];
}
```


Correctness of algorithms

- For a task, an algorithm may be **right** or **wrong**.
  - Is the algorithm still correct for arrays with negative numbers?

> Consider an array A with n elements
> Set *max* to 0.
> **For** i from 1 to n:
>   **If** $A_i > max$
> $max = A_i$.
> Output *max*.

  - If not, how to modify it?

> Consider an array A with n elements
> Set *max* to $\mathbf{A_1}$.
> **For** i from **2** to n:
>   **If** $A_i > max$
> $max = A_i$.
> Output *max*.

Efficiency of algorithms

- For a task, an correct algorithm may be **efficient** or **inefficient**.
  - Are these two algorithms both correct?
  - Which one is more efficient?

```
Consider an array A with n elements
Set max to A_1.
For i from 2 to n:
  If A_i ≥ max
    max = A_i.
Output max.
```

```
Consider an array A with n elements
Set max to A_1.
For i from 2 to n:
  If A_i > max
    max = A_i.
Output max.
```

- Among all correct algorithms, we want to find one that is efficient.

Efficiency of algorithms

- The efficiency (sometimes called performance) of different algorithms may vary a lot.
- How to find both the maximum **and** minimum numbers in an array?

```
Consider an array A with n elements
Set max to A_1. Set min to A_1.
For i from 2 to n:
  If A_i > max
    max = A_i.
  If A_i < min
    min = A_i.
Output max and min.
```

```
Consider an array A with n elements
Set max to A_1. Set min to A_1.
For i from 2 to n:
  If A_i > max
    max = A_i.
  Else if A_i < min
    min = A_i.
Output max and min.
```

  - Which one is more efficient?

Summary

- An algorithm is a sequence of steps for completing a task.
- An algorithm should first be correct. Then it should be efficient.
- An algorithm is typically described by pseudocodes.
  - Ignore the implementation details when you design your program!

Outline

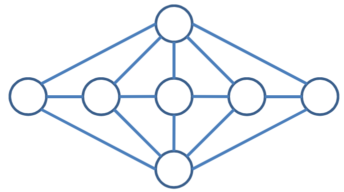
- Algorithms
- **Combinatorial problems**
- The knapsack problem

Combinatorial problems

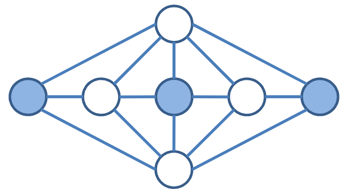
- **Combinatorial problems** (or **discrete problems**) brings many challenges and interesting findings in the field of Computer Science, Operations Research, and various fields of Engineering.
- Roughly speaking, in a combinatorial problem, one tries to find **a subset of** "**items**" such that:
  - The selection **fits a requirement**, or
  - The selection is **optimal** with respect to an objective function.
- In the former case, it is a combinatorial **decision** problem.
- In the latter case, it is a combinatorial **optimization** problem.

Dominating sets

- Consider the following example "**dominating set**":
  - We are given a graph, which contains a set of nodes and set of links.



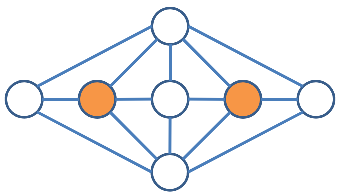
  - A dominating set is a set of nodes D such that all nodes not in D is **adjacent** to at least one node in D.

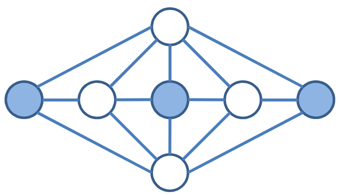


  - For a graph, there may be more than one dominating sets.

Dominating sets

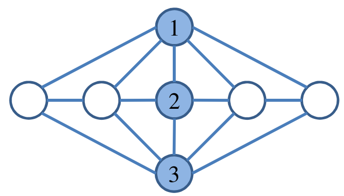
- The decision version of this problem: "Is there any dominating set that contains no more than k nodes?"
- The optimization version of this problem: "Find the dominating set that contains the smallest number of nodes."





Greedy algorithms

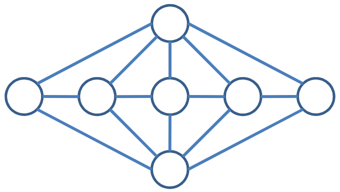
- How would you solve a dominating set problem?
- For a combinatorial problem, typically we may try **a greedy algorithm**:
  - At each step, select one item that "**at this moment**" seems to be the best.
- For the dominating set problem, a greedy algorithm may be:
  - Before all nodes are either in D or adjacent to one node in D, select a node that is not in D and adjacent to most not-in-D nodes.



- Does a greedy algorithm always find an optimal solution?

Complete enumeration

- Another extreme way of solving a combinatorial problem is through a **complete enumeration**.
  - Also called the **brute-force** algorithm.
  - Simply enumerate all the possible selections, compare them, and find the best one.
- Does a complete enumeration always find an optimal solution?
- How many possible selections do we have for this graph?



Exponential-time algorithms

- While a greedy algorithm is efficient, it may not be correct.
- While a complete enumeration is correct, it is too inefficient.
  - Especially when the problem size is large.
- Regarding the dominating set problem, suppose the given graph has n nodes, a complete enumeration needs to evaluate 2^n possible selections.
- Such an algorithm is said to be an **exponential-time** algorithm.
  - Which is not practical for large-scale problems.

Polynomial-time algorithms

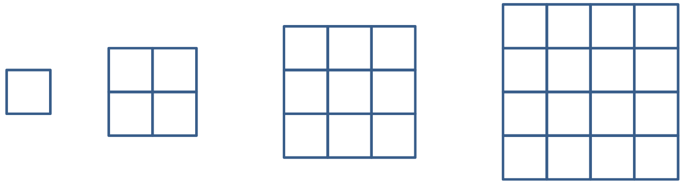
- On the contrary, some algorithms run in **a polynomial time**.
  - The number of actions to be done is at most a polynomial function of the problem size.
- To find the maximum and minimum numbers in a array:
  - At most how many actions will be done?

```
Consider an array A with n elements
Set max to A_1. Set min to A_1.
For i from 1 to n:
  If A_i > max
    max = A_i.
  If A_i < min
    min = A_i.
Output max and min.
```

```
Consider an array A with n elements
Set max to A_1. Set min to A_1.
For i from 1 to n:
  If A_i > max
    max = A_i.
  Else if A_i < min
    min = A_i.
Output max and min.
```


Algorithm complexity

- For the same task, using different algorithms may result in completely different execution time!
- Consider the following example:
  - For n^2 squares arranged into a big square, how many different routes, which do not travel the same edge twice, do we have from the left-top corner to the right-bottom corner?



- Let's watch the video!

Algorithm complexity

- The issues of **algorithm complexity** and efficiency lie at the heart of Computer Science.
  - Will be discussed extensively in Discrete Mathematics, Algorithms, and Theory of Computation.
- At this time, all we need to know is that "among all algorithms, some are better and some are worse."

Outline

- Algorithms
- Combinatorial problems
- **The knapsack problem**

The knapsack problem

- **The knapsack problem** is one of the most fundamental problems in Computer Science.
- It is a problem that is "easy to describe but hard to solve."
- The problem:
  - We are given a knapsack (backpack) and a set of items.
  - These items have various weights and values.
  - We want to select some items to maximize the total value.
  - But the total weight cannot exceed the knapsack capacity.

The knapsack problem

- Problem input:
  - The weight of items: $w_1, w_2, \ldots, w_n$.
  - The value of items: $v_1, v_2, \ldots, v_n$.
  - The weight limit of the knapsack B.
- Problem formulation:
  - Let $x_i = 1$ if item i is selected and 0 otherwise.
  - The problem:

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} v_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^{n} w_i x_i \leq B \\ & x_i \in \{0,1\} \forall i = 1,\ldots,n. \end{aligned}$$

A greedy algorithm

- How to solve the knapsack problem?
- Let's consider the following greedy algorithm:
  - For each unselected item that can be select (selecting it does not exceed the knapsack capacity), select the one which has the **largest v_i / w_i** ratio.
  - Keep doing so until we can select no more item.
- Will the optimal solution be found for the following instance?
  - Knapsack capacity: $B = 6$.
  - 4 items:

i	1	2	3	4
w_i	2	3	4	5
v_i	2	2	4	6

- Any idea to modify this algorithm?

NP-hardness

- Amazingly, no one knows how to solve this problem efficiently!
- It has been shown that the knapsack problem belongs to the class of "**NP-hard**" problems.
  - **No one** has found a method that is better than complete enumeration.
  - Most people believe a polynomial-time algorithm **does not exist**.
- Even the following simplification is NP-hard:
  - Given some items with various weights and a knapsack with a fixed capacity, is there a way of selecting a subset of items to exactly fill the knapsack?
  - Note that this is a decision problem.

NP-hardness

- So what should we do if we really need a solution?
- Fortunately, the knapsack problem is **weakly NP-hard**:
  - There exists pseudo-polynomial algorithms.
  - We will introduce an algorithm based on **dynamic programming**.
  - The algorithm requires selection, repetition, and matrices.
- Given a capacity B and a set of items with weights $w_1, w_2, \ldots, w_n$:
  - We want to determine **whether there is a set** such that items in that set together weigh exactly B.
  - If so, we want to determine **which items** should be selected.

The dynamic programming algorithm

- Let $w = (w_1, w_2, \ldots, w_n)$ be the weight vector and B be the capacity.
- Let $\boldsymbol{P(B, n)}$ be the problem of capacity B and weights $w_1, \ldots, w_n$.
- For the problem with $w = (2, 3, 4, 5)$ and $B = 6$:
  - $P(6, 4)$ is our original problem.
  - $P(6, 3)$ is to fill a knapsack of capacity 6 with $w = (2, 3, 4)$.
  - $P(5, 3)$ is to fill a knapsack of capacity 5 with $w = (2, 3, 4)$.
- The answer of $P(B, n)$ has **three possibilities**:
  - $P(B, n)$ = IMP if these n items cannot fill the knapsack.
  - $P(B, n)$ = NS if they can fill the knapsack by not selecting item n.
  - $P(B, n)$ = S if they can fill the knapsack by selecting item n.

The dynamic programming algorithm

- Example: $w = (2, 3, 4, 5)$ and $B = 6$.
- A problem can be solved by solving "**smaller**" problems:
  - Suppose we know $P(6, 3)$ = NS or S, then we know $P(6, 4)$ = NS.
  - Suppose we know $P(1, 3)$ = NS or S, then we know $P(6, 4)$ = S.
  - Suppose we know $P(6, 3) = P(1, 3)$ = IMP, does $P(6, 4)$ = IMP?
- And also problems with only 1 item is easy:
  - $P(0, 1)$ = NS, $P(2, 1)$ = S, $P(B, 1)$ = IMP for all B that are not 0 or 2.
- So we may do an iterative **bottom-up** solution process:
  - First problems with only 1 item.
  - Then 2 items.
  - And so on.

The dynamic programming algorithm

- Solving this problem with a **matrix**:

w_i / B	0	1	2	3	4	5	6
2	NS	IMP	S	IMP	IMP	IMP	IMP
3	NS	IMP	NS	S	IMP	S	IMP
4	NS	IMP	NS	NS	S	NS	S
5	NS	IMP	NS	NS	NS	S or NS	NS

- The last cell is what we want. The answer is "Yes, we may fill a knapsack of capacity 6 with the four items."
- How to determine the items to be selected?

Implementation

- How to implement this algorithm?
- Prepare a two-dimensional array.
  - Each element records the answer of that subproblem.
- Find the values of the array by a two-level loop.
  - The outer loop checks 1 item, 2 items, …, and n items.
  - The inner loop checks capacity 0, 1, 2, …, and B.
- For each subproblem:
  - If condition 1 is true, write S into this element.
  - If condition 2 is true, write NS into this element.
  - Otherwise, write IMP.
- What if both conditions are true?

Efficiency

- Is this algorithm efficient?
- Typically yes, but no if the knapsack capacity is really large…