Basics of Algorithm Design Methods

Dr. 何明昕，He Mingxin, Max

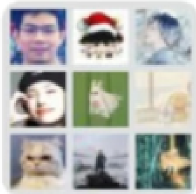
mxhe1 @ yeah.net

Email Subject： (L1-|L2-|L3-) + *last 4 digits of ID* + *Name*: *TOPIC*

Your Lab Class

Sakai: CS203B Fall 2022

数据结构与算法分析B
Data Structures and Algorithm Analysis



CS203B-f22-课程群

该二维码7天内(9月22日前)有效，重新进入将更新

QQ Group for Labs:

813058168

Recap: 3 Dimensions of Typical CS Courses

Most of Courses on CS may be organized by a combination of elements from the following 3 dimensions:

THEORY related: Concepts, Models, Maths, Algorithms, Principles, Mechanisms, Methods, … Need to understand Abstract Things

SYSTEM and TOOLS related: HW, Network, OS, PLs, IDEs, DBMS, Clients, Servers, Virtual Machines, Containers in Cloud, … Need to understand, setup and use them properly

DESIGN related: According to Requirements (Problems), need to use Theory and Tools to shape/create/implement/test Solutions!

Put Them All Together!

Lecture 3

- Basics of Algorithm Design Methods

  (Textbook by Levitin ; Ch1, Ch2 of Text B)
- Lists, Stacks & Queues (1.3 of Text A) (week 3,4)

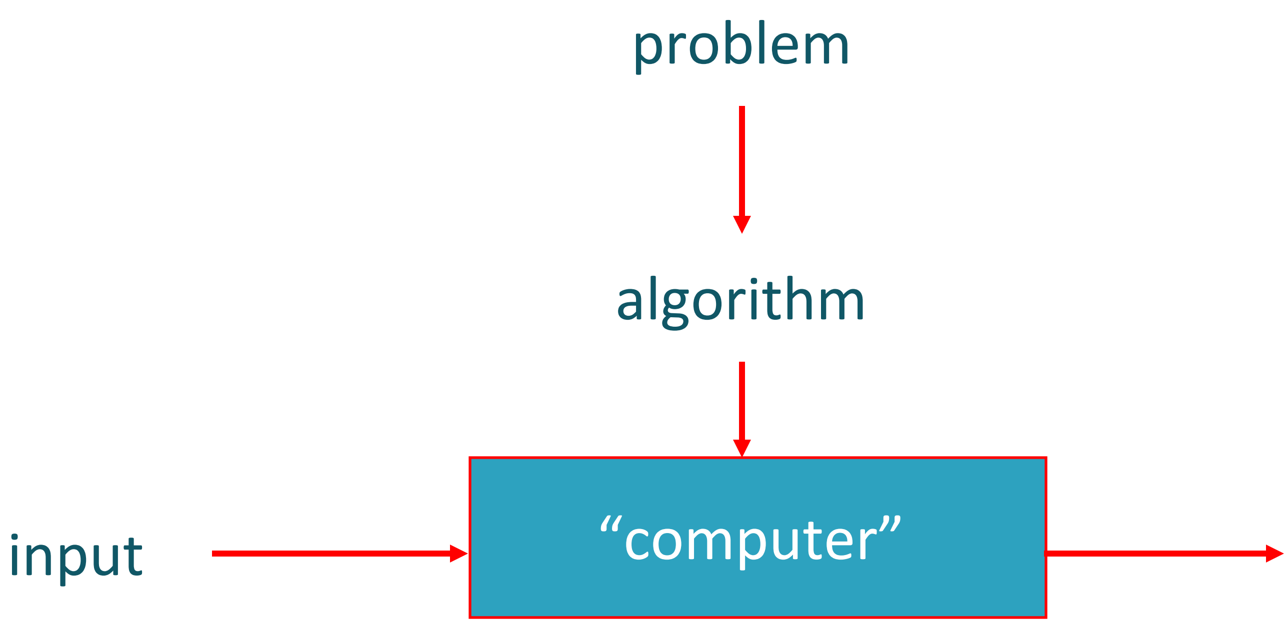
To be discussed in Lecture 5:

- Elementary Sort (2.1 of Text A)

  Selection Sort, Insertion Sort, Shell Sort, Shuffling

What is an Algorithm?

An *algorithm* is a sequence of unambiguous instructions for solving a computation problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.



- Clarity
- Correct
- Terminated
- Time Efficient
- Space Efficient

General Systematic Approach

- Separation of Concerns (关注点分离)
- Problem Decomposition (Divide, 问题分解)
- Stepwise Refinement (Conquer, 逐步求精)

- 将复杂问题做合理的分解，再分别仔细研究问题的不同侧面（关注点），最后综合各方面的结果，合成整体的解决方案。

关注点分离求解问题的一般模式：

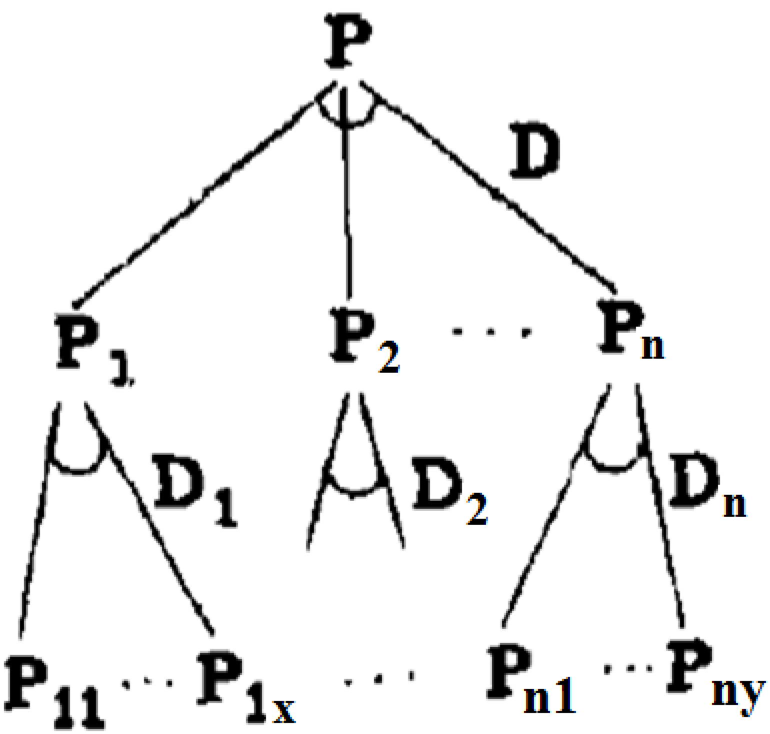
对一个典型的复杂问题 P，通过关注点分离解决问题的基本思路可一般地描述为以下 3 个步骤：

1）先将待解问题 P 分解为不同的关注点 P_1，P_2，…，P_n，即：$P \rightarrow D(P_1, P_2, \cdots, P_n)$，$D$ 表示问题分解策略，即关注点分离方法；

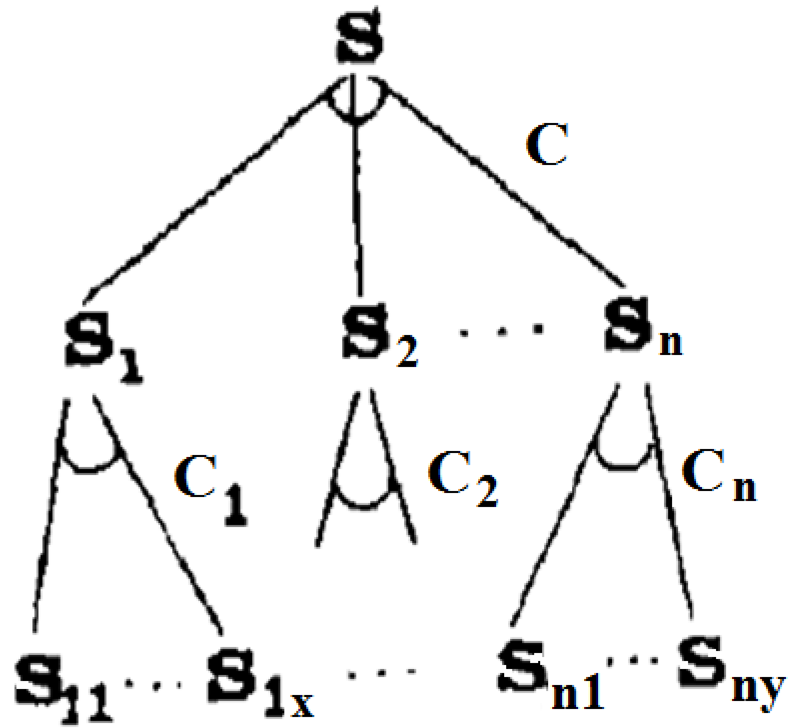
2）对 P_1，P_2，…，P_n 分别考虑，求得各自的解 S_1，S_2，…，S_n；

3）通过对 S_1，S_2，…，S_n 的合成，求得问题 P 的解 S，即：$C(S_1, S_2, \cdots, S_n) \rightarrow S$，$C$ 表示解的合成方法。

解的合成方法与关注点分离策略密切相关，对具体问题做具体的分析，充分掌握问题相关的具体知识，把握住其关键特征，在关节处实施分割，往往事半功倍。



问题分解树



程序生成关系图

Algorithm Design Methods: Techniques/Strategies

- Brute Force
- Divide and Conquer
- Decrease and Conquer
- Transform and Conquer
- Space & Time Tradeoffs
- Greedy Approach
- Dynamic Programming
- Iterative improvement
- Backtracking
- Branch and bound

Important Problem Types

- Sorting
- Searching
- String Processing
- Graph Problems
- Combinatorial Problems
- Geometric Problems
- Numerical Problems

Fundamental Data Structures

- List
  - Array
  - Linked List
  - String
- Stack
- Queue
- Priority Queue
- Graph
- Tree
- Set / Bag
- Map (Dictionary)
- Hashing

Brute Force (暴力法，穷举法)

A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved

Examples:

1. Computing a^n ($a > 0$, n a nonnegative integer)
2. Computing $n!$
3. Multiplying two matrices
4. Searching for a key of a given value in a list

Divide and Conquer (分治法)

- Mathematical Induction Analogy

- Solve the problem by
  - Divide it into smaller parts
  - Solve the smaller parts **recursively**
  - Merge the result of the smaller parts

Sample DandC Code:

```
ResultType DandC(Problem p) {
    if (p is trivial) {
       solve p directly
       return the result
    } else {
       divide p into p₁,p₂,...,pₙ

       for (i = 1 to n)
         rᵢ = DandC(pᵢ)

       combine r₁,r₂,...,rₙ into r
       return r
    }
}
```

Sample DandC Code:

```
ResultType DandC(Problem p) {
    if (p is trivial) {
        solve p directly
        return the result
    } else {
        divide p into p_1,p_2,...,p_n

        for (i = 1 to n)
          r_i = DandC(p_i)

        combine r_1,r_2,...,r_n into r
        return r
    }
}
```

Trivial Case — t_s

Divide — t_d

Recursive — t_r

Combine — t_c

Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Multiplication of large integers
- Matrix multiplication: Strassen’s algorithm
- Closest-pair and convex-hull algorithms
- Binary search: decrease-by-half (or degenerate divide&conq.)

Decrease-and-Conquer (减治法)

1. Reduce problem instance to smaller instance of the same problem
2. Solve smaller instance
3. Extend solution of smaller instance to obtain solution to original instance

- Can be implemented either top-down or bottom-up
- Also referred to as *inductive* or *incremental* approach

3 Types of Decrease and Conquer

- *Decrease by a constant* (usually by 1):
  - insertion sort
  - topological sorting
  - algorithms for generating permutations, subsets

- *Decrease by a constant factor* (usually by half)
  - binary search and bisection method
  - exponentiation by squaring
  - multiplication à la russe

- *Variable-size decrease*
  - Euclid's algorithm
  - selection by partition
  - Nim-like games

What's the difference?

Consider the problem of exponentiation: Compute a^n

- Brute Force: $a^n = a*a*a*a*...*a$
- Divide and conquer: $a^n = a^{n/2} * a^{n/2}$ (more accurately, $a^n = a^{\lfloor n/2 \rfloor} * a^{\lceil n/2 \rceil}$)
- Decrease by one: $a^n = a^{n-1} * a$
- Decrease by constant factor: $a^n = (a^{n/2})^2$ ---- much more efficient algorithm (similar as in Dynamic Programming) – Compare: in divide and conquer we recompute $a^{n/2}$

Transform and Conquer (变治法)

This group of techniques solves a problem by a *transformation* to

- a simpler/more convenient instance of the same problem (*instance simplification*)
- a different representation of the same instance (*representation change*)
- a different problem for which an algorithm is already available (*problem reduction*)

Space-for-Time Trade-offs

Two varieties of space-for-time algorithms:

- *input enhancement* — preprocess the input (or its part) to store some info to be used later in solving the problem
  - counting sorts
  - string searching algorithms
- *prestructuring* — preprocess the input to make accessing its elements easier
  - hashing
  - indexing schemes (e.g., B-trees)

Dynamic Programming (动态规划)

D*ynamic Programming* is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems

- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS
- “Programming” here means “planning”
- Main idea:
  - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
  - solve smaller instances once
  - record solutions in a table
  - extract solution to the initial instance from that table

Greedy Technique (贪婪法)

If solving problem is a series of steps

- *Simply pick the one that "maximizes" the immediate outcome Instead of looking for the long run result.*

Constructs a solution to an *optimization problem* piece by through a sequence of choices that are:

- *feasible*
- *locally optimal*
- *irrevocable*

For some problems, yields an optimal solution for every instance.

For most, does not but can be useful for fast approximations.

Applications of the Greedy Strategy

- Optimal solutions:
  - change making for “normal” coin denominations
  - minimum spanning tree (MST)
  - single-source shortest paths
  - simple scheduling problems
  - Huffman codes

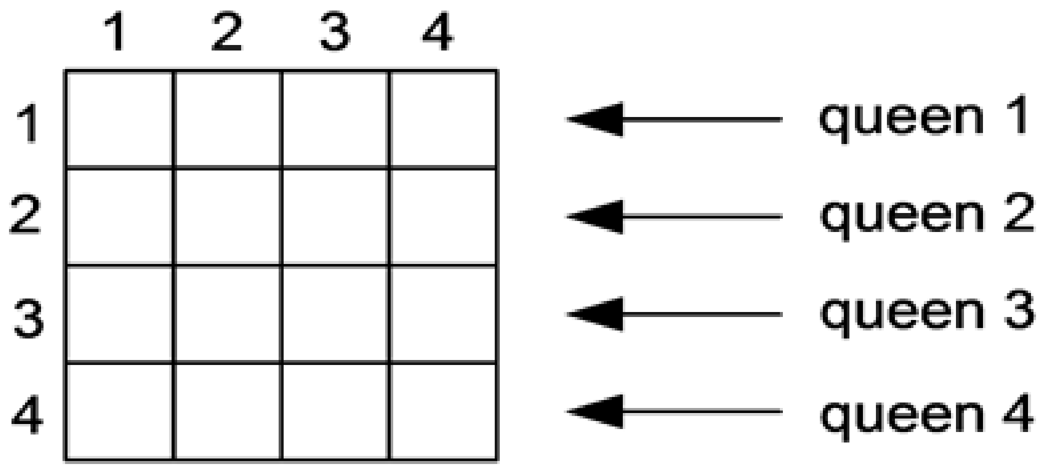
- Approximations:
  - traveling salesman problem (TSP)
  - knapsack problem
  - other combinatorial optimization problems

Backtracking (回溯法)

- Construct the *state-space tree*
  - nodes: partial solutions
  - edges: choices in extending partial solutions
- Explore the state space tree using depth-first search
- “Prune” *nonpromising nodes*
  - stop exploring subtrees rooted at nodes that cannot lead to a solution and backtracks to such a node’s parent to continue the search

Example: *n*-Queens Problem

Place *n* queens on an *n*-by-*n* chess board so that no two of them are in the same row, column, or diagonal



Branch-and-Bound (分枝界限法)

- An enhancement of backtracking
- Applicable to optimization problems
- For each node (partial solution) of a state-space tree, computes a bound on the value of the objective function for all descendants of the node (extensions of the partial solution)
- Uses the bound for:
  - ruling out certain nodes as “non-promising” to prune the tree – if a node’s bound is not better than the best solution seen so far
  - guiding the search through state-space (normally width-first)

Example: Assignment Problem

Select one element in each row of the cost matrix *C* so that:

- no two selected elements are in the same column
- the sum is minimized

Example:

	Job 1	Job 2	Job 3	Job 4
Person *a*	9	2	7	8
Person *b*	6	4	3	7
Person *c*	5	8	1	8
Person *d*	7	6	9	4

<u>Lower bound</u>: Any solution to this problem will have total cost at least: 2 + 3 + 1 + 4 (or 5 + 2 + 1 + 4)

Summary

- Basics of Algorithm Design Methods
  (Textbook by Levitin; Ch1, Ch2 of Text B)
- Lists, Stacks & Queues (1.3 of Text A) (week 3,4)

To be discussed in Lecture 5:

- Elementary Sort (2.1 of Text A)

  Selection Sort, Insertion Sort, Shell Sort, Shuffling