programming contests is the Binary Search principle. If you want to do well in programming contests, please spend time practicing the various ways to apply it.

Once you are more familiar with the 'Binary Search the Answer' technique discussed in this section, please explore Section 8.4.1 for a few more programming exercises that use this technique with *other algorithm* that we will discuss in the latter parts of this book.

We notice that there are not that many D&C problems outside of our binary search categorization. Most D&C solutions are 'geometry-related' or 'problem specific', and thus cannot be discussed in detail in this book. However, we will encounter some of them in Section 8.4.1 (binary search the answer plus geometry formulas), Section 9.14 (Inversion Index), Section 9.21 (Matrix Power), and Section 9.29 (Selection Problem).

Programming Exercises solvable using Divide and Conquer:

- Binary Search
  1. UVa 00679 - Dropping Balls (binary search; bit manipulation solutions exist)
  2. UVa 00957 - Popes (complete search + binary search: `upper_bound`)
  3. UVa 10077 - The Stern-Brocot ... (binary search)
  4. UVa 10474 - Where is the Marble? (simple: use `sort` and then `lower_bound`)
  5. ***UVa 10567 - Helping Fill Bates **** (store increasing indices of each char of 'S' in 52 vectors; for each query, binary search for the position of the char in the correct vector)
  6. UVa 10611 - Playboy Chimp (binary search)
  7. UVa 10706 - Number Sequence (binary search + some mathematical insights)
  8. UVa 10742 - New Rule in Euphomia (use sieve; binary search)
  9. **UVa 11057 - Exact Sum *** (sort, for price `p[i]`, check if price `(M - p[i])` exists with binary search)
  10. UVa 11621 - Small Factors (generate numbers with factor 2 and/or 3, `sort`, `upper_bound`)
  11. *UVa 11701 - Cantor* (a kind of ternary search)
  12. UVa 11876 - N + NOD (N) (`[lower|upper]_bound` on sorted sequence N)
  13. ***UVa 12192 - Grapevine **** (the input array has special sorted properties; use `lower_bound` to speed up the search)
  14. Thailand ICPC National Contest 2009 - My Ancestor (author: Felix Halim)
- Bisection Method or Binary Search the Answer
  1. **UVa 10341 - Solve It *** (bisection method discussed in this section; for alternative solutions, see `http://www.algorithmist.com/index.php/UVa_10341`)
  2. **UVa 11413 - Fill the ... *** (binary search the answer + simulation)
  3. UVa 11881 - Internal Rate of Return (bisection method)
  4. UVa 11935 - Through the Desert (binary search the answer + simulation)
  5. ***UVa 12032 - The Monkey ... **** (binary search the answer + simulation)
  6. *UVa 12190 - Electric Bill* (binary search the answer + algebra)
  7. IOI 2010 - Quality of Living (binary search the answer)

  Also see: Divide & Conquer for Geometry Problems (see Section 8.4.1)
- Other Divide & Conquer Problems
  1. ***UVa 00183 - Bit Maps **** (simple exercise of Divide and Conquer)
  2. IOI 2011 - Race (D&C; whether the solution path uses a vertex or not)

  Also see: Data Structures with Divide & Conquer flavor (see Section 2.3)

3.4 Greedy

An algorithm is said to be greedy if it makes the locally optimal choice at each step with the hope of eventually reaching the globally optimal solution. In some cases, greedy works—the solution is short and runs efficiently. For *many* others, however, it does not. As discussed in other typical Computer Science textbooks, e.g. [7, 38], a problem must exhibit these two properties in order for a greedy algorithm to work:

1. It has optimal sub-structures.
   Optimal solution to the problem contains optimal solutions to the sub-problems.

2. It has the greedy property (difficult to prove in time-critical contest environment!).
   If we make a choice that seems like the best at the moment and proceed to solve the remaining subproblem, we reach the optimal solution. We will never have to reconsider our previous choices.

3.4.1 Examples

Coin Change - The Greedy Version

Problem description: Given a target amount V cents and a list of denominations of n coins, i.e. we have `coinValue[i]` (in cents) for coin types `i` $\in$ `[0..n-1]`, what is the minimum number of coins that we must use to represent amount V? Assume that we have an unlimited supply of coins of any type. Example: If $n = 4$, `coinValue = {25, 10, 5, 1}` cents[6], and we want to represent $V = 42$ cents, we can use this Greedy algorithm: Select the largest coin denomination which is not greater than the remaining amount, i.e. 42-$\underline{25} = 17 \rightarrow$ 17-$\underline{10}$ $= 7 \rightarrow$ 7-$\underline{5} = 2 \rightarrow$ 2-$\underline{1} = 1 \rightarrow$ 1-$\underline{1} = 0$, a total of 5 coins. This is optimal.

The problem above has the two ingredients required for a successful greedy algorithm:

1. It has optimal sub-structures.
   We have seen that in our quest to represent 42 cents, we used 25+10+5+1+1.
   This is an optimal 5-coin solution to the original problem!
   Optimal solutions to sub-problem are contained within the 5-coin solution, i.e.
   a. To represent 17 cents, we can use 10+5+1+1 (part of the solution for 42 cents),
   b. To represent 7 cents, we can use 5+1+1 (also part of the solution for 42 cents), etc

2. It has the greedy property: Given every amount V, we can greedily subtract from it the largest coin denomination which is not greater than this amount V. It can be proven (not shown here for brevity) that using any other strategies will not lead to an optimal solution, at least for this set of coin denominations.

However, this greedy algorithm does *not* work for *all* sets of coin denominations. Take for example {4, 3, 1} cents. To make 6 cents with that set, a greedy algorithm would choose 3 coins {4, 1, 1} instead of the optimal solution that uses 2 coins {3, 3}. The general version of this problem is revisited later in Section 3.5.2 (Dynamic Programming).

UVa 410 - Station Balance (Load Balancing)

Given $1 \leq C \leq 5$ chambers which can store 0, 1, or 2 specimens, $1 \leq S \leq 2C$ specimens and a list M of the masses of the S specimens, determine which chamber should store each specimen in order to minimize 'imbalance'. See Figure 3.4 for a visual explanation[7].

[6] The presence of the 1-cent coin ensures that we can always make every value.

[7] Since $C \leq 5$ and $S \leq 10$, we can actually use a Complete Search solution for this problem. However, this problem is simpler to solve using the Greedy algorithm.

A = $(\sum_{j=1}^{S} M_j)/C$, i.e. A is the average of the total mass in each of the C chambers.

Imbalance = $\sum_{i=1}^{C} |X_i - A|$, i.e. the sum of differences between the total mass in each chamber w.r.t. A where X_i is the total mass of specimens in chamber i.

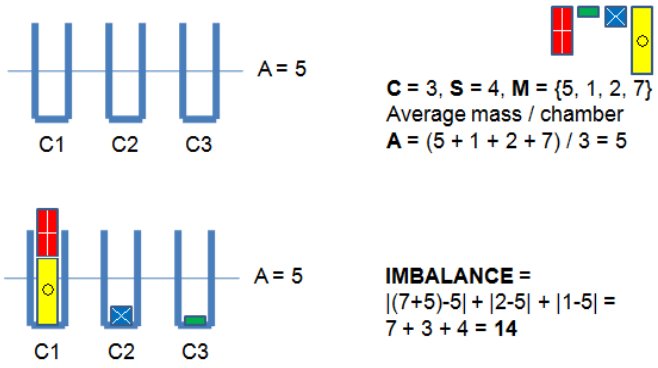


Figure 3.4: Visualization of UVa 410 - Station Balance

This problem can be solved using a greedy algorithm, but to arrive at that solution, we have to make several observations.

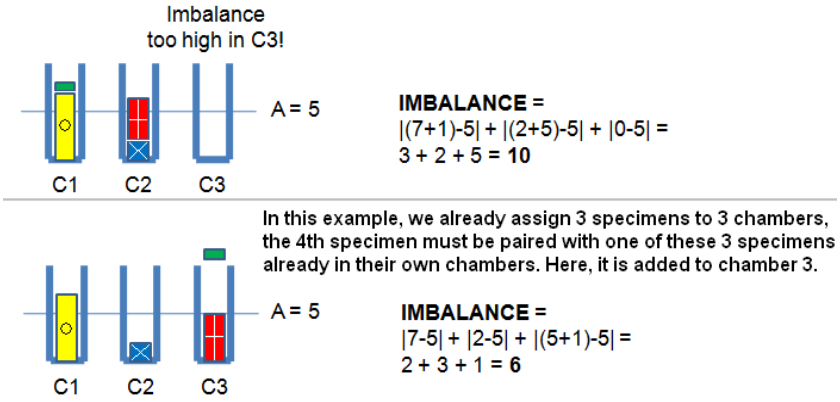


Figure 3.5: UVa 410 - Observations

Observation 1: If there exists an empty chamber, it is usually beneficial and never worse to move one specimen from a chamber with two specimens to the empty chamber! Otherwise, the empty chamber contributes more to the imbalance as shown in Figure 3.5, top.

Observation 2: If $S > C$, then $S - C$ specimens must be paired with a chamber already containing other specimens—the Pigeonhole principle! See Figure 3.5, bottom.

The key insight is that the solution to this problem can be simplified with sorting: if $S < 2C$, add $2C - S$ dummy specimens with mass 0. For example, $C = 3$, $S = 4$, $M = \{5, 1, 2, 7\} \rightarrow C = 3, S = 6, M = \{5, 1, 2, 7, 0, 0\}$. Then, sort the specimens on their mass such that $M_1 \leq M_2 \leq \ldots \leq M_{2C-1} \leq M_{2C}$. In this example, $M = \{5, 1, 2, 7, 0, 0\} \rightarrow \{0, 0, 1, 2, 5, 7\}$. By adding dummy specimens and then sorting them, a greedy strategy becomes 'apparent':

- Pair the specimens with masses $M_1 \& M_{2C}$ and put them in chamber 1, then
- Pair the specimens with masses $M_2 \& M_{2C-1}$ and put them in chamber 2, and so on ...

This greedy algorithm—known as *load balancing*—works! See Figure 3.6.

It is hard to impart the techniques used in deriving this greedy solution. Finding greedy solutions is an art, just as finding good Complete Search solutions requires creativity. A tip that arises from this example: If there is no obvious greedy strategy, try *sorting* the data or introducing some tweak and see if a greedy strategy emerges.

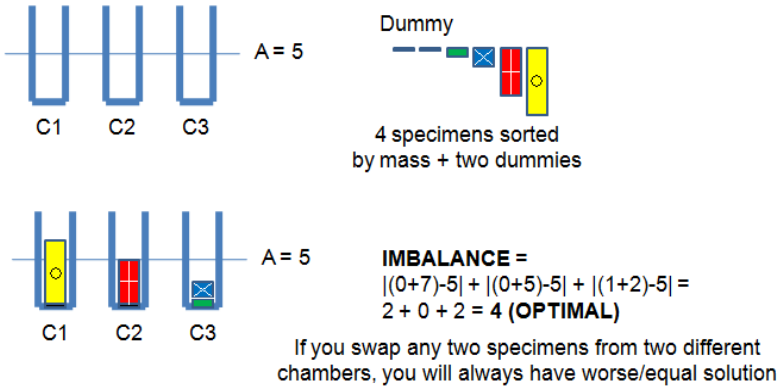


Figure 3.6: UVa 410 - Greedy Solution

UVa 10382 - Watering Grass (Interval Covering)

Problem description: n sprinklers are installed in a horizontal strip of grass L meters long and W meters wide. Each sprinkler is centered vertically in the strip. For each sprinkler, we are given its position as the distance from the left end of the center line and its radius of operation. What is the minimum number of sprinklers that should be turned on in order to water the entire strip of grass? Constraint: $n \leq 10000$. For an illustration of the problem, see Figure 3.7—left side. The answer for this test case is 6 sprinklers (those labeled with {A, B, D, E, F, H}). There are 2 unused sprinklers: {C, G}.

We cannot solve this problem with a brute force strategy that tries all possible subsets of sprinklers to be turned on since the number of sprinklers can go up to 10000. It is definitely infeasible to try all 2^{10000} possible subsets of sprinklers.

This problem is actually a variant of the well known greedy problem called the *interval covering* problem. However, it includes a simple geometric twist. The original interval covering problem deals with intervals. This problem deals with sprinklers that have circles of influence in a horizontal area rather than simple intervals. We first have to transform the problem to resemble the standard interval covering problem.

See Figure 3.7—right side. We can convert these circles and horizontal strips into intervals. We can compute `dx = sqrt(`R^2 `-` $(W/2)^2$`)`. Suppose a circle is centered at `(x, y)`. The interval represented by this circle is `[x-dx..x+dx]`. To see why this works, notice that the additional circle segment beyond `dx` away from x does not completely cover the strip in the horizontal region it spans. If you have difficulties with this geometric transformation, see Section 7.2.4 which discusses basic operations involving a *right triangle*.

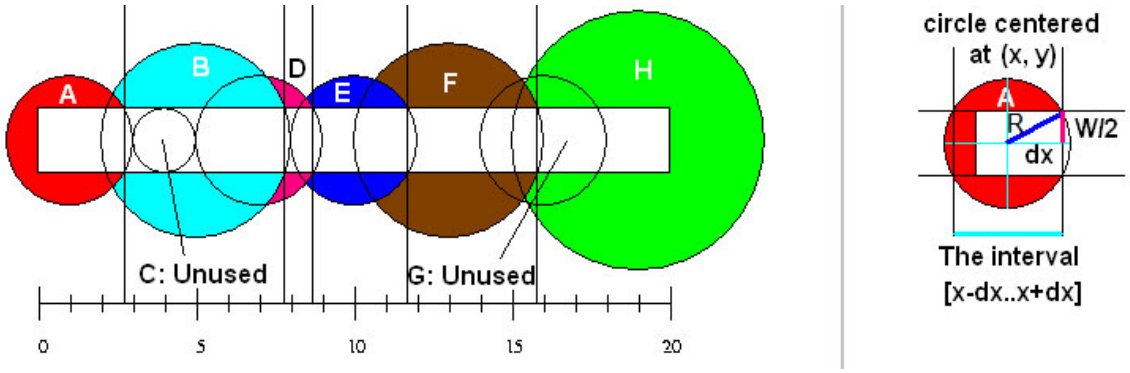


Figure 3.7: UVa 10382 - Watering Grass

Now that we have transformed the original problem into the interval covering problem, we can use the following Greedy algorithm. First, the Greedy algorithm sorts the intervals by *increasing* left endpoint and by *decreasing* right endpoint if ties arise. Then, the Greedy algorithm processes the intervals one at a time. It takes the interval that covers 'as far right as possible' and yet still produces uninterrupted coverage from the leftmost side to the rightmost side of the horizontal strip of grass. It ignores intervals that are already completely covered by other (previous) intervals.

For the test case shown in Figure 3.7—left side, this Greedy algorithm first sorts the intervals to obtain the sequence {A, B, C, D, E, F, G, H}. Then it processes them one by one. First, it takes 'A' (it has to), takes 'B' (connected to interval 'A'), ignores 'C' (as it is embedded inside interval 'B'), takes 'D' (it has to, as intervals 'B' and 'E' are not connected if 'D' is not used), takes 'E', takes 'F', ignores 'G' (as taking 'G' is not 'as far right as possible' and does not reach the rightmost side of the grass strip), takes 'H' (as it connects with interval 'F' and covers more to the right than interval of 'G' does, going beyond the rightmost end of the grass strip). In total, we select 6 sprinklers: {A, B, D, E, F, H}. This is the minimum possible number of sprinklers for this test case.

UVa 11292 - Dragon of Loowater (Sort the Input First)

Problem description: There are n dragon heads and m knights ($1 \leq n, m \leq 20000$). Each dragon head has a *diameter* and each knight has a *height*. A dragon head with diameter **D** can be chopped off by a knight with height **H** if $\mathbf{D} \leq \mathbf{H}$. A knight can only chop off one dragon head. Given a list of diameters of the dragon heads and a list of heights of the knights, is it possible to chop off all the dragon heads? If yes, what is the minimum total height of the knights used to chop off the dragons' heads?

There are several ways to solve this problem, but we will illustrate one that is probably the easiest. This problem is a bipartite matching problem (this will be discussed in more detail in Section 4.7.4), in the sense that we are required to match (pair) certain knights to dragon heads in a maximal fashion. However, this problem can be solved greedily: Each dragon head should be chopped by a knight with the shortest height that is at least as tall as the diameter of the dragon's head. However, the input is given in an arbitrary order. If we sort both the list of dragon head diameters and knight heights in $O(n \log n + m \log m)$, we can use the following $O(min(n, m))$ scan to determine the answer. This is yet another example where sorting the input can help produce the required greedy strategy.

```
gold = d = k = 0; // array dragon+knight are sorted in non decreasing order
while (d < n && k < m) {              // still have dragon heads or knights
  while (dragon[d] > knight[k] && k < m) k++;   // find the required knight
  if (k == m) break;      // no knight can kill this dragon head, doomed :S
  gold += knight[k];                    // the king pay this amount of gold
  d++; k++;                            // next dragon head and knight please
}

if (d == n) printf("%d\n", gold);           // all dragon heads are chopped
else        printf("Loowater is doomed!\n");
```

Exercise 3.4.1.1*: Which of the following sets of coins (all in cents) are solvable using the greedy 'coin change' algorithm discussed in this section? If the greedy algorithm fails on a certain set of coin denominations, determine the smallest counter example V cents on which it fails to be optimal. See [51] for more details about finding such counter examples.

1. $S_1 = \{10, 7, 5, 4, 1\}$
2. $S_2 = \{64, 32, 16, 8, 4, 2, 1\}$
3. $S_3 = \{13, 11, 7, 5, 3, 2, 1\}$
4. $S_4 = \{7, 6, 5, 4, 3, 2, 1\}$
5. $S_5 = \{21, 17, 11, 10, 1\}$

Remarks About Greedy Algorithm in Programming Contests

In this section, we have discussed three classical problems solvable with Greedy algorithms: Coin Change (the special case), Load Balancing, and Interval Covering. For these classical problems, it is helpful to memorize their solutions (for this case, ignore that we have said earlier in the chapter about not relying too much on memorization). We have also discussed an important problem solving strategy usually applicable to greedy problems: Sorting the input data to elucidate hidden greedy strategies.

There are two other classical examples of Greedy algorithms in this book, e.g. Kruskal's (and Prim's) algorithm for the Minimum Spanning Tree (MST) problem (see Section 4.3) and Dijkstra's algorithm for the Single-Source Shortest Paths (SSSP) problem (see Section 4.4.3). There are many more known Greedy algorithms that we have chosen not to discuss in this book as they are too 'problem specific' and rarely appear in programming contests, e.g. Huffman Codes [7, 38], Fractional Knapsack [7, 38], some Job Scheduling problems, etc.

However, today's programming contests (both ICPC and IOI) rarely involve the purely canonical versions of these classical problems. Using Greedy algorithms to attack a 'non classical' problem is usually risky. A Greedy algorithm will normally not encounter the TLE response as it is often lightweight, but instead tends to obtain WA verdicts. Proving that a certain 'non-classical' problem has optimal sub-structure and greedy property during contest time may be difficult or time consuming, so a competitive programmer should usually use this rule of thumb:

If the input size is 'small enough' to accommodate the time complexity of either Complete Search or Dynamic Programming approaches (see Section 3.5), then use these approaches as both will ensure a correct answer. *Only* use a Greedy algorithm if the input size given in the problem statement are too large even for the best Complete Search or DP algorithm.

Having said that, it is increasingly true that problem authors try to set the input bounds of problems that allow for Greedy strategies to be in an ambiguous range so that contestants *cannot* use the input size to quickly determine the required algorithm!

We have to remark that it is quite challenging to come up with new 'non-classical' Greedy problems. Therefore, the number of such novel Greedy problems used in competitive programming is lower than that of Complete Search or Dynamic Programming problems.

Programming Exercises solvable using Greedy
(most hints are omitted to keep the problems challenging):

- Classical, Usually Easier
  1. UVa 00410 - Station Balance (discussed in this section, load balancing)
  2. UVa 01193 - Radar Installation (LA 2519, Beijing02, interval covering)
  3. UVa 10020 - Minimal Coverage (interval covering)
  4. UVa 10382 - Watering Grass (discussed in this section, interval covering)
  5. ***UVa 11264 - Coin Collector ****** (coin change variant)

6. **UVa 11389 - The Bus Driver Problem *** (load balancing)
7. *UVa 12321 - Gas Station* (interval covering)
8. ***UVa 12405 - Scarecrow ** (simpler interval covering problem)
9. IOI 2011 - Elephants (optimized greedy solution can be used up to subtask 3, but the harder subtasks 4 and 5 must be solved using efficient data structure)

- Involving Sorting (Or The Input Is Already Sorted)

1. UVa 10026 - Shoemaker's Problem
2. *UVa 10037 - Bridge*
3. UVa 10249 - The Grand Dinner
4. UVa 10670 - Work Reduction
5. UVa 10763 - Foreign Exchange
6. UVa 10785 - The Mad Numerologist
7. ***UVa 11100 - The Trip, 2007 ****
8. UVa 11103 - WFF'N Proof
9. *UVa 11269 - Setting Problems*
10. **UVa 11292 - Dragon of Loowater ***
11. UVa 11369 - Shopaholic
12. UVa 11729 - Commando War
13. UVa 11900 - Boiled Eggs
14. ***UVa 12210 - A Match Making Problem ****
15. *UVa 12485 - Perfect Choir*

- Non Classical, Usually Harder

1. UVa 00311 - Packets
2. *UVa 00668 - Parliament*
3. UVa 10152 - ShellSort
4. UVa 10340 - All in All
5. UVa 10440 - Ferry Loading II
6. UVa 10602 - Editor Nottobad
7. **UVa 10656 - Maximum Sum (II) ***
8. UVa 10672 - Marbles on a tree
9. UVa 10700 - Camel Trading
10. UVa 10714 - Ants
11. **UVa 10718 - Bit Mask ***
12. *UVa 10982 - Troublemakers*
13. UVa 11054 - Wine Trading in Gergovia
14. **UVa 11157 - Dynamic Frog ***
15. *UVa 11230 - Annoying painting tool*
16. *UVa 11240 - Antimonotonicity*
17. *UVa 11335 - Discrete Pursuit*
18. UVa 11520 - Fill the Square
19. UVa 11532 - Simple Adjacency ...
20. UVa 11567 - Moliu Number Generator
21. *UVa 12482 - Short Story Competition*

3.5 Dynamic Programming

Dynamic Programming (from now on abbreviated as DP) is perhaps the most challenging problem-solving technique among the four paradigms discussed in this chapter. Thus, make sure that you have mastered the material mentioned in the previous chapters/sections before reading this section. Also, prepare to see lots of recursion and recurrence relations!

The key skills that you have to develop in order to master DP are the abilities to determine the problem *state*s and to determine the relationships or *transitions* between current problems and their sub-problems. We have used these skills earlier in recursive backtracking (see Section 3.2.2). In fact, DP problems with small input size constraints may already be solvable with recursive backtracking.

If you are new to DP technique, you can start by assuming that (the 'top-down') DP is a kind of 'intelligent' or 'faster' recursive backtracking. In this section, we will explain the reasons why DP is often faster than recursive backtracking for problems amenable to it.

DP is primarily used to solve *optimization* problems and *counting* problems. If you encounter a problem that says "minimize this" or "maximize that" or "count the ways to do that", then there is a (high) chance that it is a DP problem. Most DP problems in programming contests only ask for the optimal/total value and not the optimal solution itself, which often makes the problem easier to solve by removing the need to backtrack and produce the solution. However, some harder DP problems also require the optimal solution to be returned in some fashion. We will continually refine our understanding of Dynamic Programming in this section.

3.5.1 DP Illustration

We will illustrate the concept of Dynamic Programming with an example problem: UVa 11450 - Wedding Shopping. The abridged problem statement: Given different options for each garment (e.g. 3 shirt models, 2 belt models, 4 shoe models, ...) and a certain *limited* budget, our task is to *buy one model of each garment*. We cannot spend more money than the given budget, but we want to spend *the maximum possible* amount.

The input consists of two integers $1 \leq M \leq 200$ and $1 \leq C \leq 20$, where M is the budget and C is the number of garments that you have to buy, followed by some information about the C garments. For the garment `g` $\in$ `[0..C-1]`, we will receive an integer $1 \leq K \leq 20$ which indicates the number of different models there are for that garment `g`, followed by K integers indicating the price of each model $\in$ `[1..K]` of that garment `g`.

The output is one integer that indicates the maximum amount of money we can spend purchasing one of each garment *without exceeding the budget*. If there is no solution due to the small budget given to us, then simply print "`no solution`".

Suppose we have the following test case A with $M = 20$, $C = 3$:
Price of the 3 models of garment `g = 0` → 6 4 <u>8</u> // the prices are not sorted in the input
Price of the 2 models of garment `g = 1` → 5 <u>10</u>
Price of the 4 models of garment `g = 2` → <u>1</u> 5 3 5

For this test case, the answer is 19, which *may* result from buying the <u>underlined</u> items (8+10+1). This is not unique, as solutions (6+10+3) and (4+10+5) are also optimal.

However, suppose we have this test case B with $M =$ **9 (limited budget)**, $C = 3$:
Price of the 3 models of garment `g = 0` → 6 4 8
Price of the 2 models of garment `g = 1` → 5 10
Price of the 4 models of garment `g = 2` → 1 5 3 5

The answer is then "`no solution`" because even if we buy all the cheapest models for each garment, the total price (4+5+1) = 10 still exceeds our given budget $M = 9$.

In order for us to appreciate the usefulness of Dynamic Programming in solving the above-mentioned problem, let's explore how far the *other* approaches discussed earlier will get us in this particular problem.

Approach 1: Greedy (Wrong Answer)

Since we want to maximize the budget spent, one greedy idea (there are other greedy approaches—which are also WA) is to take the most expensive model for each garment `g` which still fits our budget. For example in test case A above, we can choose the most expensive model 3 of garment `g = 0` with price 8 (`money` is now `20-8 = 12`), then choose the most expensive model 2 of garment `g = 1` with price 10 (`money = 12-10 = 2`), and finally for the last garment `g = 2`, we can only choose model 1 with price 1 as the `money` we have left does not allow us to buy the other models with price 3 or 5. This greedy strategy 'works' for test cases A and B above and produce the same optimal solution (8+10+1) = 19 and "`no solution`", respectively. It also runs very fast[8]: $20 + 20 + \ldots + 20$ for a total of 20 times = 400 operations in the worst case. However, this greedy strategy does not work for many other test cases, such as this *counter-example* below (test case C):

Test case C with $M = 12$, $C = 3$:
3 models of garment `g = 0` → 6 $\underline{4}$ **8**
2 models of garment `g = 1` → $\underline{5}$ 10
4 models of garment `g = 2` → 1 5 $\underline{3}$ 5

The Greedy strategy selects model 3 of garment `g = 0` with price **8** (`money = 12-8 = 4`), causing us to not have enough money to buy any model in garment `g = 1`, thus incorrectly reporting "`no solution`". One optimal solution is $\underline{4+5+3} = 12$, which uses up all of our budget. The optimal solution is not unique as 6+5+1 = 12 also depletes the budget.

Approach 2: Divide and Conquer (Wrong Answer)

This problem is not solvable using the Divide and Conquer paradigm. This is because the sub-problems (explained in the Complete Search sub-section below) are not independent. Therefore, we cannot solve them separately with the Divide and Conquer approach.

Approach 3: Complete Search (Time Limit Exceeded)

Next, let's see if Complete Search (recursive backtracking) can solve this problem. One way to use recursive backtracking in this problem is to write a function `shop(money, g)` with two parameters: The current `money` that we have and the current garment `g` that we are dealing with. The pair `(money, g)` is the *state* of this problem. Note that the order of parameters does not matter, e.g. `(g, money)` is also a perfectly valid state. Later in Section 3.5.3, we will see more discussion on how to select appropriate states for a problem.

We start with `money = M` and garment `g = 0`. Then, we try all possible models in garment `g = 0` (a maximum of 20 models). If model i is chosen, we subtract model i's price from `money`, then repeat the process in a recursive fashion with garment `g = 1` (which can also have up to 20 models), etc. We stop when the model for the last garment `g = C-1` has been chosen. If `money < 0` before we choose a model from garment `g = C-1`, we can prune the infeasible solution. Among all valid combinations, we can then pick the one that results in the smallest non-negative `money`. This maximizes the money spent, which is (`M - money`).

[8]We do not need to sort the prices just to find the model with the maximum price as there are only up to $K \leq 20$ models. An $O(K)$ scan is enough.

We can formally define these Complete Search recurrences (transitions) as follows:

1. If `money < 0` (i.e. money goes negative),
`shop(money, g) =` $-\infty$ (in practice, we can just return a large negative value)

2. If a model from the last garment has been bought, that is, `g = C`,
`shop(money, g) = M - money` (this is the actual money that we spent)

3. In general case, $\forall$ `model` $\in$ `[1..K]` of current garment `g`,
`shop(money, g) = max(shop(money - price[g][model], g + 1))`
We want to maximize this value (Recall that the invalid ones have large negative value)

This solution works correctly, but it is **very slow**! Let's analyze the worst case time complexity. In the largest test case, garment `g = 0` has up to 20 models; garment `g = 1` *also* has up to 20 models and all garments including the last garment `g = 19` ***also*** have up to 20 models. Therefore, this Complete Search runs in $20 \times 20 \times \ldots \times 20$ operations in the worst case, i.e. 20^{20} = a **very large** number. If we can *only* come up with this Complete Search solution, we cannot solve this problem.

Approach 4: Top-Down DP (Accepted)

To solve this problem, we have to use the DP concept as this problem satisfies the two prerequisites for DP to be applicable:

1. This problem has optimal sub-structures[9].
   This is illustrated in the third Complete Search recurrence above: The solution for the sub-problem is part of the solution of the original problem. In other words, if we select model i for garment $g = 0$, for our final selection to be optimal, our choice for garments $g = 1$ and above must also be the optimal choice for a reduced budget of $M - price$, where *price* refers to the price of model i.
2. This problem has overlapping sub-problems.
   This is the key characteristic of DP! The search space of this problem is *not* as big as the rough 20^{20} bound obtained earlier because **many** sub-problems are *overlapping*!

Let's verify if this problem indeed has overlapping sub-problems. Suppose that there are 2 models in a certain garment `g` with the *same* price `p`. Then, a Complete Search will move to the **same** sub-problem `shop(money - p, g + 1)` after picking *either* model! This situation will also occur if some combination of `money` and chosen model's price causes $\texttt{money}_1 - \texttt{p}_1 = \texttt{money}_2 - \texttt{p}_2$ at the same garment `g`. This will—in a Complete Search solution—cause the same sub-problem to be computed *more than once*, an inefficient state of affairs!

So, how many *distinct* sub-problems (a.k.a. **states** in DP terminology) are there in this problem? Only $201 \times 20 = 4020$. There are only 201 possible values for `money` (0 to 200 inclusive) and 20 possible values for the garment `g` (0 to 19 inclusive). Each sub-problem just needs to be computed *once*. If we can ensure this, we can solve this problem *much faster*.

The implementation of this DP solution is surprisingly simple. If we already have the recursive backtracking solution (see the recurrences—a.k.a. **transitions** in DP terminology—shown in the Complete Search approach above), we can implement the **top-down** DP by adding these two additional steps:

1. Initialize[10] a DP 'memo' table with dummy values that are not used in the problem, e.g. '-1'. The DP table should have dimensions corresponding to the problem states.

[9] Optimal sub-structures are also required for Greedy algorithms to work, but this problem lacks the 'greedy property', making it unsolvable with the Greedy algorithm.

[10] For C/C++ users, the `memset` function in <cstring> is a good tool to perform this step.

2. At the start of the recursive function, check if this state has been computed before.
   (a) If it has, simply return the value from the DP memo table, $O(1)$.
   (This the origin of the term 'memoization'.)
   (b) If it has not been computed, perform the computation as per normal (only once) and then store the computed value in the DP memo table so that *further calls* to this sub-problem (state) return immediately.

Analyzing a basic[11] DP solution is easy. If it has M distinct states, then it requires $O(M)$ memory space. If computing one state (the complexity of the DP transition) requires $O(k)$ steps, then the overall time complexity is $O(kM)$. This UVa 11450 problem has $M = 201 \times 20 = 4020$ and $k = 20$ (as we have to iterate through at most 20 models for each garment g). Thus, the time complexity is at most $4020 \times 20 = 80400$ operations per test case, a very manageable calculation.

We display our code below for illustration, especially for those who have never coded a top-down DP algorithm before. Scrutinize this code and verify that it is indeed very similar to the recursive backtracking code that you have seen in Section 3.2.

```
/* UVa 11450 - Wedding Shopping - Top Down */
// assume that the necessary library files have been included
// this code is similar to recursive backtracking code
// parts of the code specific to top-down DP are commented with: 'TOP-DOWN'

int M, C, price[25][25];                  // price[g (<= 20)][model (<= 20)]
int memo[210][25];    // TOP-DOWN: dp table memo[money (<= 200)][g (<= 20)]
int shop(int money, int g) {
  if (money < 0) return -1000000000;     // fail, return a large -ve number
  if (g == C) return M - money;        // we have bought last garment, done
  // if the line below is commented, top-down DP will become backtracking!!
  if (memo[money][g] != -1) return memo[money][g]; // TOP-DOWN: memoization
  int ans = -1;   // start with a -ve number as all prices are non negative
  for (int model = 1; model <= price[g][0]; model++)      // try all models
    ans = max(ans, shop(money - price[g][model], g + 1));
  return memo[money][g] = ans; }     // TOP-DOWN: memoize ans and return it

int main() {            // easy to code if you are already familiar with it
  int i, j, TC, score;
  scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &M, &C);
    for (i = 0; i < C; i++) {
      scanf("%d", &price[i][0]);                  // store K in price[i][0]
      for (j = 1; j <= price[i][0]; j++) scanf("%d", &price[i][j]);
    }
    memset(memo, -1, sizeof memo);    // TOP-DOWN: initialize DP memo table
    score = shop(M, 0);                            // start the top-down DP
    if (score < 0) printf("no solution\n");
    else           printf("%d\n", score);
} } // return 0;
```

[11] Basic means "without fancy optimizations that we will see later in this section and in Section 8.3".

We want to take this opportunity to illustrate another style used in implementing DP solutions (only applicable for C/C++ users). Instead of frequently addressing a certain cell in the memo table, we can use a local *reference* variable to store the memory address of the required cell in the memo table as shown below. The two coding styles are not very different, and it is up to you to decide which style you prefer.

```
int shop(int money, int g) {
  if (money < 0) return -1000000000; // order of >1 base cases is important
  if (g == C) return M - money;  // money can't be <0 if we reach this line
  int &ans = memo[money][g];                  // remember the memory address
  if (ans != -1) return ans;
  for (int model = 1; model <= price[g][0]; model++)
    ans = max(ans, shop(money - price[g][model], g + 1));
  return ans;                // ans (or memo[money][g]) is directly updated
}
```

Source code: `ch3_02_UVa11450_td.cpp/java`

Approach 5: Bottom-Up DP (Accepted)

There is another way to implement a DP solution often referred to as the **bottom-up** DP. This is actually the 'true form' of DP as DP was originally known as the 'tabular method' (computation technique involving a table). The *basic* steps to build bottom-up DP solution are as follows:

1. Determine the required set of parameters that uniquely describe the problem (the state). This step is similar to what we have discussed in recursive backtracking and top-down DP earlier.

2. If there are N parameters required to represent the states, prepare an N dimensional DP table, with one entry per state. This is equivalent to the memo table in top-down DP. However, there are differences. In bottom-up DP, we only need to initialize some cells of the DP table with known initial values (the base cases). Recall that in top-down DP, we initialize the memo table completely with dummy values (usually -1) to indicate that we have not yet computed the values.

3. Now, with the base-case cells/states in the DP table already filled, determine the cells/states that can be filled next (the transitions). Repeat this process until the DP table is complete. For the bottom-up DP, this part is usually accomplished through iterations, using loops (more details about this later).

For UVa 11450, we can write the bottom-up DP as follow: We describe the state of a subproblem with two parameters: The current garment `g` and the current `money`. This state formulation is essentially equivalent to the state in the top-down DP above, except that we have reversed the order to make `g` the first parameter (thus the values of `g` are the row indices of the DP table so that we can take advantage of cache-friendly row-major traversal in a 2D array, see the speed-up tips in Section 3.2.3). Then, we initialize a 2D table (boolean matrix) `reachable[g][money]` of size 20×201. Initially, only cells/states reachable by buying any of the models of the first garment `g = 0` are set to true (in the first row). Let's use test case A above as example. In Figure 3.8, top, the only columns '20-6 = 14', '20-4 = 16', and '20-8 = 12' in row 0 are initially set to true.

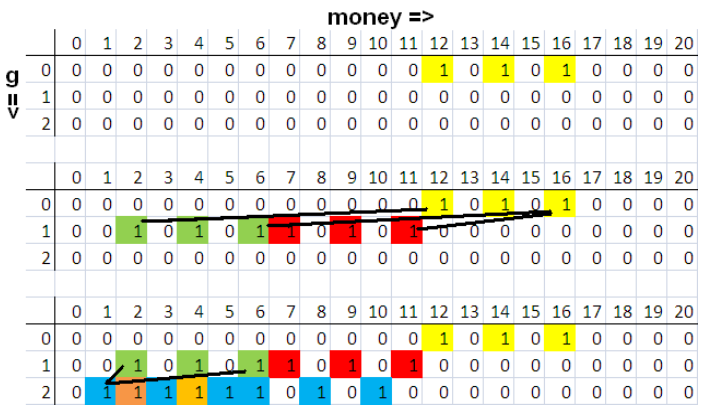


Figure 3.8: Bottom-Up DP (columns 21 to 200 are not shown)

Now, we loop from the second garment `g = 1` (second row) to the last garment `g = C-1 = 3-1 = 2` (third and last row) in row-major order (row by row). If `reachable[g-1][money]` is true, then the next state `reachable[g][money-p]` where `p` is the price of a model of current garment `g` is also reachable as long as the second parameter (`money`) is not negative. See Figure 3.8, middle, where `reachable[0][16]` propagates to `reachable[1][16-5]` and `reachable[1][16-10]` when the model with price 5 and 10 in garment `g = 1` is bought, respectively; `reachable[0][12]` propagates to `reachable[1][12-10]` when the model with price 10 in garment `g = 1` is bought, etc. We repeat this table filling process row by row until we are done with the last row[12].

Finally, the answer can be found in the last row when `g = C-1`. Find the state in that row that is both nearest to index 0 and reachable. In Figure 3.8, bottom, the cell `reachable[2][1]` provides the answer. This means that we can reach state (`money = 1`) by buying some combination of the various garment models. The required final answer is actually `M - money`, or in this case, `20-1 = 19`. The answer is "`no solution`" if there is no state in the last row that is reachable (where `reachable[C-1][money]` is set to true). We provide our implementation below for comparison with the top-down version.

```
/* UVa 11450 - Wedding Shopping - Bottom Up */
// assume that the necessary library files have been included

int main() {
  int g, money, k, TC, M, C;
  int price[25][25];                        // price[g (<= 20)][model (<= 20)]
  bool reachable[25][210];    // reachable table[g (<= 20)][money (<= 200)]

  scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &M, &C);
    for (g = 0; g < C; g++) {
      scanf("%d", &price[g][0]);                // we store K in price[g][0]
      for (money = 1; money <= price[g][0]; money++)
        scanf("%d", &price[g][money]);
    }
```

[12]Later in Section 4.7.1, we will discuss DP as a traversal of an (implicit) DAG. To avoid unnecessary 'backtracking' along this DAG, we have to visit the vertices in their topological order (see Section 4.2.5). The order in which we fill the DP table is a topological ordering of the underlying implicit DAG.

```
    memset(reachable, false, sizeof reachable);         // clear everything
    for (g = 1; g <= price[0][0]; g++)       // initial values (base cases)
      if (M - price[0][g] >= 0)      // to prevent array index out of bound
        reachable[0][M - price[0][g]] = true;  // using first garment g = 0

    for (g = 1; g < C; g++)                     // for each remaining garment
      for (money = 0; money < M; money++) if (reachable[g-1][money])
        for (k = 1; k <= price[g][0]; k++) if (money - price[g][k] >= 0)
          reachable[g][money - price[g][k]] = true;   // also reachable now

    for (money = 0; money <= M && !reachable[C - 1][money]; money++);

    if (money == M + 1) printf("no solution\n");  // last row has no on bit
    else                printf("%d\n", M - money);
  }
} // return 0;
```

Source code: ch3_03_UVa11450_bu.cpp/java

There is an advantage for writing DP solutions in the bottom-up fashion. For problems where we only need the last row of the DP table (or, more generally, the last updated slice of all the states) to determine the solution—including this problem, we can optimize the *memory usage* of our DP solution by sacrificing one dimension in our DP table. For harder DP problems with tight memory requirements, this 'space saving trick' may prove to be useful, though the overall time complexity does not change.

Let's take a look again at Figure 3.8. We only need to store two rows, the current row we are processing and the previous row we have processed. To compute row 1, we only need to know the columns in row 0 that are set to true in `reachable`. To compute row 2, we similarly only need to know the columns in row 1 that are set to true in `reachable`. In general, to compute row g, we only need values from the previous row $g-1$. So, instead of storing a boolean matrix `reachable[g][money]` of size $\underline{20} \times 201$, we can simply store `reachable[2][money]` of size $\underline{2} \times 201$. We can use this programming trick to reference one row as the 'previous' row and another row as the 'current' row (e.g. `prev = 0, cur = 1`) and then swap them (e.g. now `prev = 1, cur = 0`) as we compute the bottom-up DP row by row. Note that for this problem, the memory savings are not significant. For harder DP problems, for example where there might be thousands of garment models instead of 20, this space saving trick can be important.

Top-Down versus Bottom-Up DP

Although both styles use 'tables', the way the bottom-up DP table is filled is different to that of the top-down DP *memo* table. In the top-down DP, the memo table entries are filled 'as needed' through the recursion itself. In the bottom-up DP, we used a correct 'DP table filling order' to compute the values such that the previous values needed to process the current cell have already been obtained. This table filling order is the topological order of the implicit DAG (this will be explained in more detail in Section 4.7.1) in the recurrence structure. For most DP problems, a topological order can be achieved simply with the proper sequencing of some (nested) loops.

For most DP problems, these two styles are equally good and the decision to use a particular DP style is a matter of preference. However, for harder DP problems, one of the

styles can be better than the other. To help you understand which style that you should use when presented with a DP problem, please study the trade-offs between top-down and bottom-up DPs listed in Table 3.2.

Top-Down	**Bottom-Up**
Pros: 1. It is a natural transformation from the normal Complete Search recursion 2. Computes the sub-problems only when necessary (sometimes this is faster)	Pros: 1. Faster if many sub-problems are revisited as there is no overhead from recursive calls 2. Can save memory space with the 'space saving trick' technique
Cons: 1. Slower if many sub-problems are revisited due to function call overhead (this is not usually penalized in programming contests) 2. If there are M states, an $O(M)$ table size is required, which can lead to MLE for some harder problems (except if we use the trick in Section 8.3.4)	Cons: 1. For programmers who are inclined to recursion, this style may not be intuitive 2. If there are M states, bottom-up DP visits and fills the value of *all* these M states

Table 3.2: DP Decision Table

Displaying the Optimal Solution

Many DP problems request only for the value of the optimal solution (like the UVa 11450 above). However, many contestants are caught off-guard when they are also required to print the optimal solution. We are aware of two ways to do this.

The first way is mainly used in the bottom-up DP approach (which is still applicable for top-down DPs) where we store the predecessor information at each state. If there are more than one optimal predecessors and we have to output all optimal solutions, we can store those predecessors in a list. Once we have the optimal final state, we can do backtracking from the optimal final state and follow the optimal transition(s) recorded at each state until we reach one of the base cases. If the problem asks for all optimal solutions, this backtracking routine will print them all. However, most problem authors usually set additional output criteria so that the selected optimal solution is unique (for easier judging).

Example: See Figure 3.8, bottom. The optimal final state is `reachable[2][1]`. The predecessor of this optimal final state is state `reachable[1][2]`. We now backtrack to `reachable[1][2]`. Next, see Figure 3.8, middle. The predecessor of state `reachable[1][2]` is state `reachable[0][12]`. We then backtrack to `reachable[0][12]`. As this is already one of the initial base states (at the first row), we know that an optimal solution is: (20→12) = price 8, then (12→2) = price 10, then (2→1) = price 1. However, as mentioned earlier in the problem description, this problem may have several other optimal solutions, e.g. We can also follow the path: `reachable[2][1]` → `reachable[1][6]` → `reachable[0][16]` which represents another optimal solution: (20→16) = price 4, then (16→6) = price 10, then (6→1) = price 5.

The second way is applicable mainly to the top-down DP approach where we utilize the strength of recursion and memoization to do the same job. Using the top-down DP code shown in Approach 4 above, we will add another function `void print_shop(int money, int g)` that has the same structure as `int shop(int money, int g)` except that it uses the values stored in the memo table to reconstruct the solution. A sample implementation (that only prints out one optimal solution) is shown below:

```
void print_shop(int money, int g) {           // this function returns void
  if (money < 0 || g == C) return;                    // similar base cases
  for (int model = 1; model <= price[g][0]; model++) // which model is opt?
    if (shop(money - price[g][model], g + 1) == memo[money][g]) {
      printf("%d%c", price[g][model], g == C-1 ? '\n' : '-');   // this one
      print_shop(money - price[g][model], g + 1);  // recurse to this state
      break;                                        // do not visit other states
}   }
```

Exercise 3.5.1.1: To verify your understanding of UVa 11450 problem discussed in this section, determine what is the output for test case D below?

Test case D with $M = 25$, $C = 3$:
Price of the 3 models of garment `g = 0` $\rightarrow$ 6 4 8
Price of the 2 models of garment `g = 1` $\rightarrow$ 10 6
Price of the 4 models of garment `g = 2` $\rightarrow$ 7 3 1 5

Exercise 3.5.1.2: Is the following state formulation `shop(g, model)`, where `g` represents the current garment and `model` represents the current model, appropriate and exhaustive for UVa 11450 problem?

Exercise 3.5.1.3: Add the space saving trick to the bottom-up DP code in Approach 5!

3.5.2 Classical Examples

The problem UVa 11450 - Wedding Shopping above is a (relatively easy) non-classical DP problem, where we had to come up with the correct DP states and transitions *by ourself*. However, there are many other *classical* problems with efficient DP solutions, i.e. their DP states and transitions are *well-known*. Therefore, such classical DP problems and their solutions should be mastered by every contestant who wishes to do well in ICPC or IOI! In this section, we list down six classical DP problems and their solutions. Note: Once you understand the basic form of these DP solutions, try solving the programming exercises that enumerate their *variants*.

1. Max 1D Range Sum

Abridged problem statement of UVa 507 - Jill Rides Again: Given an integer array `A` containing $n \leq 20K$ non-zero integers, determine the maximum (1D) range sum of `A`. In other words, find the maximum Range Sum Query (RSQ) between two indices `i` and `j` in `[0..n-1]`, that is: `A[i] + A[i+1] + A[i+2] +...+ A[j]` (also see Section 2.4.3 and 2.4.4).

A Complete Search algorithm that tries all possible $O(n^2)$ pairs of `i` and `j`, computes the required `RSQ(i, j)` in $O(n)$, and finally picks the maximum one runs in an overall time complexity of $O(n^3)$. With n up to $20K$, this is a TLE solution.

In Section 2.4.4, we have discussed the following DP strategy: Pre-process array `A` by computing `A[i] += A[i-1]` $\forall$`i` $\in$ `[1..n-1]` so that `A[i]` contains the sum of integers in subarray `A[0..i]`. We can now compute `RSQ(i, j)` in $O(1)$: `RSQ(0, j) = A[j]` and `RSQ(i, j) = A[j] - A[i-1]` $\forall$`i` $>$ `0`. With this, the Complete Search algorithm above can be made to run in $O(n^2)$. For n up to $20K$, this is still a TLE approach. However, this technique is still useful in other cases (see the usage of this 1D Range Sum in Section 8.4.2).

There is an even better algorithm for this problem. The main part of Jay Kadane's $O(n)$ (can be viewed as a greedy or DP) algorithm to solve this problem is shown below.

```
// inside int main()
  int n = 9, A[] = { 4, -5, 4, -3, 4, 4, -4, 4, -5 };   // a sample array A
  int sum = 0, ans = 0;          // important, ans must be initialized to 0
  for (int i = 0; i < n; i++) {                        // linear scan, O(n)
    sum += A[i];                      // we greedily extend this running sum
    ans = max(ans, sum);                 // we keep the maximum RSQ overall
    if (sum < 0) sum = 0;                   // but we reset the running sum
  }                                              // if it ever dips below 0
  printf("Max 1D Range Sum = %d\n", ans);
```

Source code: ch3_04_Max1DRangeSum.cpp/java

The key idea of Kadane's algorithm is to keep a running sum of the integers seen so far and greedily reset that to 0 if the running sum dips below 0. This is because re-starting from 0 is always better than continuing from a negative running sum. Kadane's algorithm is the required algorithm to solve this UVa 507 problem as $n \leq 20K$.

Note that we can also view this Kadane's algorithm as a DP solution. At each step, we have two choices: We can either leverage the previously accumulated maximum sum, or begin a new range. The DP variable `dp(i)` thus represents the maximum sum of a range of integers that ends with element `A[i]`. Thus, the final answer is the maximum over all the values of `dp(i)` where `i` $\in$ `[0..n-1]`. If zero-length ranges are allowed, then 0 must also be considered as a possible answer. The implementation above is essentially an efficient version that utilizes the space saving trick discussed earlier.

2. Max 2D Range Sum

Abridged problem statement of UVa 108 - Maximum Sum: Given an $n \times n$ ($1 \leq n \leq 100$) square matrix of integers `A` where each integer ranges from `[-127..127]`, find a sub-matrix of `A` with the maximum sum. For example: The 4×4 matrix ($n = 4$) in Table 3.3.A below has a 3×2 sub-matrix on the lower-left with maximum sum of $9 + 2$ - $4 + 1$ - $1 + 8 = 15$.

A				
	0	-2	-7	0
	9	2	-6	2
	-4	1	-4	1
	-1	8	0	-2

B				
	0	-2	-9	-9
	9	9	-4	2
	5	6	-11	-8
	4	13	-4	-3

C				
	0	-2	-9	-9
	9	9	-4	2
	5	6	-11	-8
	4	13	-4	-3

Table 3.3: UVa 108 - Maximum Sum

Attacking this problem naïvely using a Complete Search as shown below does not work as it runs in $O(n^6)$. For the largest test case with $n = 100$, an $O(n^6)$ algorithm is too slow.

```
maxSubRect = -127*100*100;    // the lowest possible value for this problem
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) // start coordinate
  for (int k = i; k < n; k++) for (int l = j; l < n; l++) {    // end coord
    subRect = 0;                       // sum the items in this sub-rectangle
    for (int a = i; a <= k; a++) for (int b = j; b <= l; b++)
      subRect += A[a][b];
    maxSubRect = max(maxSubRect, subRect); }          // the answer is here
```

The solution for the Max 1D Range Sum in the previous subsection can be extended to two (or more) dimensions as long as the inclusion-exclusion principle is properly applied. The only difference is that while we dealt with overlapping sub-ranges in Max 1D Range Sum, we will deal with overlapping sub-matrices in Max 2D Range Sum. We can turn the $n \times n$ input matrix into an $n \times n$ *cumulative sum matrix* where `A[i][j]` no longer contains its own value, but the sum of all items within sub-matrix `(0, 0)` to `(i, j)`. This can be done simultaneously while reading the input and still runs in $O(n^2)$. The code shown below turns the input square matrix (see Table 3.3.A) into a cumulative sum matrix (see Table 3.3.B).

```
scanf("%d", &n);                    // the dimension of input square matrix
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) {
  scanf("%d", &A[i][j]);
  if (i > 0) A[i][j] += A[i - 1][j];           // if possible, add from top
  if (j > 0) A[i][j] += A[i][j - 1];          // if possible, add from left
  if (i > 0 && j > 0) A[i][j] -= A[i - 1][j - 1];     // avoid double count
}                                            // inclusion-exclusion principle
```

With the sum matrix, we can answer the sum of any sub-matrix `(i, j)` to `(k, l)` in $O(1)$ using the code below. For example, let's compute the sum of `(1, 2)` to `(3, 3)`. We split the sum into 4 parts and compute `A[3][3] - A[0][3] - A[3][1] + A[0][1] = -3 - 13 - (-9) + (-2) = -9` as highlighted in Table 3.3.C. With this $O(1)$ DP formulation, the Max 2D Range Sum problem can now be solved in $O(n^4)$. For the largest test case of UVa 108 with $n = 100$, this is still fast enough.

```
maxSubRect = -127*100*100;    // the lowest possible value for this problem
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) // start coordinate
  for (int k = i; k < n; k++) for (int l = j; l < n; l++) {    // end coord
    subRect = A[k][l];      // sum of all items from (0, 0) to (k, l): O(1)
    if (i > 0) subRect -= A[i - 1][l];                              // O(1)
    if (j > 0) subRect -= A[k][j - 1];                              // O(1)
    if (i > 0 && j > 0) subRect += A[i - 1][j - 1];                 // O(1)
    maxSubRect = max(maxSubRect, subRect); }          // the answer is here
```

Source code: `ch3_05_UVa108.cpp/java`

From these two examples—the Max 1D and 2D Range Sum Problems—we can see that not every range problem requires a Segment Tree or a Fenwick Tree as discussed in Section 2.4.3 or 2.4.4. Static-input range-related problems are often solvable with DP techniques. It is also worth mentioning that the solution for a range problem is very natural to produce with bottom-up DP techniques as the operand is already a 1D or a 2D array. We can still write the recursive top-down solution for a range problem, but the solution is not as natural.

3. Longest Increasing Subsequence (LIS)

Given a sequence {`A[0], A[1],..., A[n-1]`}, determine its Longest Increasing Subsequence (LIS)[13]. Note that these 'subsequences' are not necessarily contiguous. Example: $n = 8$, $A = \{\underline{-7}, 10, 9, \underline{2, 3, 8}, 8, 1\}$. The length-4 LIS is {-7, 2, 3, 8}.

[13]There are other variants of this problem, including the Longest *Decreasing* Subsequence and Longest *Non Increasing/Decreasing* Subsequence. The increasing subsequences can be modeled as a Directed Acyclic Graph (DAG) and finding the LIS is equivalent to finding the Longest Paths in the DAG (see Section 4.7.1).

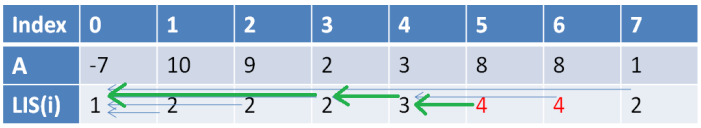


Figure 3.9: Longest Increasing Subsequence

As mentioned in Section 3.1, a naïve Complete Search that enumerates all possible subsequences to find the longest increasing one is too slow as there are $O(2^n)$ possible subsequences. Instead of trying all possible subsequences, we can consider the problem with a different approach. We can write the state of this problem with just one parameter: `i`. Let `LIS(i)` be the LIS ending at index `i`. We know that `LIS(0) = 1` as the first number in `A` is itself a subsequence. For `i` $\geq$ `1`, `LIS(i)` is slightly more complex. We need to find the index `j` such that `j` $<$ `i` and `A[j]` $<$ `A[i]` and `LIS(j)` is the largest. Once we have found this index `j`, we know that `LIS(i) = 1 + LIS(j)`. We can write this recurrence formally as:

1. `LIS(0) = 1` // the base case
2. `LIS(i) = max(LIS(j) + 1)`, $\forall$`j` $\in$ `[0..i-1]` and `A[j]` $<$ `A[i]` // the recursive case, one more than the previous best solution ending at `j` for all `j` $<$ `i`.

The answer is the largest value of `LIS(k)` $\forall$`k` $\in$ `[0..n-1]`.

Now let's see how this algorithm works (also see Figure 3.9):

- `LIS(0)` is 1, the first number in `A` = {-7}, the base case.
- `LIS(1)` is 2, as we can extend `LIS(0)` = {-7} with {10} to form {-7, 10} of length 2.
  The best `j` for `i = 1` is `j = 0`.
- `LIS(2)` is 2, as we can extend `LIS(0)` = {-7} with {9} to form {-7, 9} of length 2.
  We cannot extend `LIS(1)` = {-7, 10} with {9} as it is non increasing.
  The best `j` for `i = 2` is `j = 0`.
- `LIS(3)` is 2, as we can extend `LIS(0)` = {-7} with {2} to form {-7, 2} of length 2.
  We cannot extend `LIS(1)` = {-7, 10} with {2} as it is non-increasing.
  We also cannot extend `LIS(2)` = {-7, 9} with {2} as it is also non-increasing.
  The best `j` for `i = 3` is `j = 0`.
- `LIS(4)` is 3, as we can extend `LIS(3)` = {-7, 2} with {3} to form {-7, 2, 3}.
  This is the best choice among the possibilities.
  The best `j` for `i = 4` is `j = 3`.
- `LIS(5)` is 4, as we can extend `LIS(4)` = {-7, 2, 3} with {8} to form {-7, 2, 3, 8}.
  This is the best choice among the possibilities.
  The best `j` for `i = 5` is `j = 4`.
- `LIS(6)` is 4, as we can extend `LIS(4)` = {-7, 2, 3} with {8} to form {-7, 2, 3, 8}.
  This is the best choice among the possibilities.
  The best `j` for `i = 6` is `j = 4`.
- `LIS(7)` is 2, as we can extend `LIS(0)` = {-7} with {1} to form {-7, 1}.
  This is the best choice among the possibilities.
  The best `j` for `i = 7` is `j = 0`.
- The answers lie at `LIS(5)` or `LIS(6)`; both values (LIS lengths) are 4.
  See that the index `k` where `LIS(k)` is the highest can be anywhere in `[0..n-1]`.

There are clearly many overlapping sub-problems in LIS problem because to compute `LIS(i)`, we need to compute `LIS(j)` $\forall$`j` $\in$ `[0..i-1]`. However, there are only n distinct states, the indices of the LIS ending at index `i`, $\forall$`i` $\in$ `[0..n-1]`. As we need to compute each state with an $O(n)$ loop, this DP algorithm runs in $O(n^2)$.

If needed, the LIS solution(s) can be reconstructed by storing the predecessor information (the arrows in Figure 3.9) and tracing the arrows from index `k` that contain the highest value of `LIS(k)`. For example, `LIS(5)` is the optimal final state. Check Figure 3.9. We can trace the arrows as follow: `LIS(5)` $\rightarrow$ `LIS(4)` $\rightarrow$ `LIS(3)` $\rightarrow$ `LIS(0)`, so the optimal solution (read backwards) is index {0, 3, 4, 5} or {-7, 2, 3, 8}.

The LIS problem can also be solved using the *output-sensitive* $O(n \log k)$ greedy + D&C algorithm (where k is the length of the LIS) instead of $O(n^2)$ by maintaining an array that is *always sorted* and therefore amenable to binary search. Let array `L` be an array such that `L(i)` represents the smallest ending value of all length-`i` LISs found so far. Though this definition is slightly complicated, it is easy to see that it is always ordered—`L(i-1)` will always be smaller than `L(i)` as the second-last element of any LIS (of length-i) is smaller than its last element. As such, we can binary search array `L` to determine the longest possible subsequence we can create by appending the current element `A[i]`—simply find the index of the last element in `L` that is less than `A[i]`. Using the same example, we will update array `L` step by step using this algorithm:

- Initially, at `A[0] = -7`, we have `L = {-7}`.
- We can insert `A[1] = 10` at `L[1]` so that we have a length-2 LIS, `L = {-7, 10}`.
- For `A[2] = 9`, we replace `L[1]` so that we have a 'better' length-2 LIS ending: `L = {-7, 9}`.
  This is a *greedy* strategy. By storing the LIS with smaller ending value, we maximize our ability to further extend the LIS with future values.
- For `A[3] = 2`, we replace `L[1]` to get an 'even better' length-2 LIS ending: `L = {-7, 2}`.
- We insert `A[4] = 3` at `L[2]` so that we have a longer LIS, `L = {-7, 2, 3}`.
- We insert `A[5] = 8` at `L[3]` so that we have a longer LIS, `L = {-7, 2, 3, 8}`.
- For `A[6] = 8`, nothing changes as `L[3] = 8`.
  `L = {-7, 2, 3, 8}` remains unchanged.
- For `A[7] = 1`, we improve `L[1]` so that `L = {-7, 1, 3, 8}`.
  This illustrates how the array `L` is *not* the LIS of `A`. This step is important as there can be longer subsequences *in the future* that may extend the length-2 subsequence at `L[1] = 1`. For example, try this test case: `A = {-7, 10, 9, 2, 3, 8, 8, 1, 2, 3, 4}`. The length of LIS for this test case is 5.
- The answer is the largest length of the sorted array `L` at the end of the process.

Source code: `ch3_06_LIS.cpp/java`

---

4. 0-1 Knapsack (Subset Sum)

Problem[14]: Given n items, each with its own value V_i and weight W_i, $\forall i \in$ `[0..n-1]`, and a maximum knapsack size S, compute the maximum value of the items that we can carry, if we can either[15] ignore or take a particular item (hence the term 0-1 for ignore/take).

[14]This problem is also known as the Subset Sum problem. It has a similar problem description: Given a set of integers and an integer S, is there a (non-empty) subset that has a sum equal to S?

[15]There are other variants of this problem, e.g. the Fractional Knapsack problem with Greedy solution.

Example: $n = 4$, $V = \{100, 70, 50, 10\}$, $W = \{10, 4, 6, 12\}$, $S = 12$.
If we select item 0 with weight 10 and value 100, we cannot take any other item. Not optimal.
If we select item 3 with weight 12 and value 10, we cannot take any other item. Not optimal.
If we select item 1 and 2, we have total weight 10 and total value 120. This is the maximum.

Solution: Use these Complete Search recurrences `val(id, remW)` where `id` is the index of the current item to be considered and `remW` is the remaining weight left in the knapsack:

1. `val(id, 0) = 0` // if `remW = 0`, we cannot take anything else
2. `val(n, remW) = 0` // if `id = n`, we have considered all items
3. if `W[id]` $>$ `remW`, we have no choice but to ignore this item
`val(id, remW) = val(id + 1, remW)`
4. if `W[id]` $\leq$ `remW`, we have two choices: ignore or take this item; we take the maximum
`val(id, remW) = max(val(id + 1, remW), V[id] + val(id + 1, remW - W[id]))`

The answer can be found by calling `value(0, S)`. Note the overlapping sub-problems in this 0-1 Knapsack problem. Example: After taking item 0 and ignoring item 1-2, we arrive at state `(3, 2)`—at the third item (`id = 3`) with two units of weight left (`remW = 2`). After ignoring item 0 and taking item 1-2, we also arrive at the same state `(3, 2)`. Although there are overlapping sub-problems, there are only $O(nS)$ possible distinct states (as `id` can vary between `[0..n-1]` and `remW` can vary between `[0..S]`)! We can compute each of these states in $O(1)$, thus the overall time complexity[16] of this DP solution is $O(nS)$.

Note: The top-down version of this DP solution is often faster than the bottom-up version. This is because not all states are actually visited, and hence the critical DP states involved are actually only a (very small) subset of the entire state space. Remember: The top-down DP only visits *the required states* whereas bottom-up DP visits *all distinct states*. Both versions are provided in our source code library.

Source code: `ch3_07_UVa10130.cpp/java`

5. Coin Change (CC) - The General Version

Problem: Given a target amount V cents and a list of denominations for n coins, i.e. we have `coinValue[i]` (in cents) for coin types `i` $\in$ `[0..n-1]`, what is the minimum number of coins that we must use to represent V? Assume that we have unlimited supply of coins of any type (also see Section 3.4.1).

Example 1: $V = 10$, $n = 2$, `coinValue` $= \{1, 5\}$; We can use:
A. Ten 1 cent coins $= 10 \times 1 = 10$; Total coins used $= 10$
B. One 5 cents coin + Five 1 cent coins $= 1 \times 5 + 5 \times 1 = 10$; Total coins used $= 6$
C. Two 5 cents coins $= 2 \times 5 = 10$; Total coins used $= 2 \rightarrow$ Optimal

We can use the Greedy algorithm if the coin denominations are suitable (see Section 3.4.1). Example 1 above is solvable with the Greedy algorithm. However, for general cases, we have to use DP. See Example 2 below:

Example 2: $V = 7$, $n = 4$, `coinValue` $= \{1, 3, 4, 5\}$
The Greedy approach will produce 3 coins as its result as 5+1+1 = 7, but the optimal solution is actually 2 coins (from 4+3)!

Solution: Use these Complete Search recurrence relations for `change(value)`, where `value` is the remaining amount of cents that we need to represent in coins:

[16] If S is large such that $NS >> 1M$, this DP solution is not feasible, even with the space saving trick!

1. `change(0) = 0` // we need 0 coins to produce 0 cents
2. `change(< 0) = ∞` // in practice, we can return a large positive value
3. `change(value) = 1 + min(change(value - coinValue[i])) ∀i ∈ [0..n-1]`

The answer can be found in the return value of `change(V)`.

<0	0	1	2	3	4	5	6	7	8	9	10
∞	0	1	2	3	4	1	2	3	4	5	2

V = 10, N = 2, coinValue= {1, 5}

Figure 3.10: Coin Change

Figure 4.2.3 shows that:
`change(0) = 0` and `change(< 0) = ∞`: These are the base cases.
`change(1) = 1`, from `1 + change(1-1)`, as `1 + change(1-5)` is infeasible (returns ∞).
`change(2) = 2`, from `1 + change(2-1)`, as `1 + change(2-5)` is also infeasible (returns ∞).
... same thing for `change(3)` and `change(4)`.
`change(5) = 1`, from `1 + change(5-5) = 1` coin, smaller than `1 + change(5-1) = 5` coins.
... and so on until `change(10)`.
The answer is in `change(V)`, which is `change(10) = 2` in this example.

We can see that there are a lot of overlapping sub-problems in this Coin Change problem (e.g. both `change(10)` and `change(6)` require the value of `change(5)`). However, there are only $O(V)$ possible distinct states (as `value` can vary between `[0..V]`)! As we need to try n types of coins per state, the overall time complexity of this DP solution is $O(nV)$.

A variant of this problem is to count *the number of possible (canonical) ways* to get value V cents using a list of denominations of n coins. For example 1 above, the answer is 3: {1+1+1+1+1 + 1+1+1+1+1, 5 + 1+1+1+1+1, 5 + 5}.

Solution: Use these Complete Search recurrence relation: `ways(type, value)`, where `value` is the same as above but we now have one more parameter `type` for the index of the coin type that we are currently considering. This second parameter `type` is important as this solution considers the coin types sequentially. Once we choose to ignore a certain coin type, we should not consider it again to avoid double-counting:

1. `ways(type, 0) = 1` // one way, use nothing
2. `ways(type, <0) = 0` // no way, we cannot reach negative value
3. `ways(n, value) = 0` // no way, we have considered all coin types ∈ `[0..n-1]`
4. `ways(type, value) = ways(type + 1, value) +` // if we ignore this coin type, `ways(type, value - coinValue[type])` // plus if we use this coin type

There are only $O(nV)$ possible distinct states. Since each state can be computed in $O(1)$, the overall time complexity[17] of this DP solution is $O(nV)$. The answer can be found by calling `ways(0, V)`. Note: If the coin values are not changed and you are given many queries with different `V`, then we can choose *not* to reset the memo table. Therefore, we run this $O(nV)$ algorithm once and just perform an $O(1)$ lookup for subsequent queries.

Source code (this coin change variant): `ch3_08_UVa674.cpp/java`

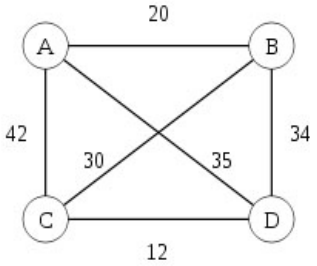
---

[17] If V is large such that $nV >> 1M$, this DP solution is not feasible even with the space saving trick!

6. Traveling Salesman Problem (TSP)

Problem: Given n cities and their pairwise distances in the form of a matrix `dist` of size $n \times n$, compute the cost of making a tour[18] that starts from any city s, goes through all the other $n-1$ cities *exactly once*, and finally returns to the starting city s.

Example: The graph shown in Figure 3.11 has $n = 4$ cities. Therefore, we have $4! = 24$ possible tours (permutations of 4 cities). One of the minimum tours is `A-B-C-D-A` with a cost of `20+30+12+35 = 97` (notice that there can be more than one optimal solution).



	A	B	C	D
A	0	20	42	35
B	20	0	30	34
C	42	30	0	12
D	35	34	12	0

Figure 3.11: A Complete Graph

A 'brute force' TSP solution (either iterative or recursive) that tries all $O((n-1)!)$ possible tours (fixing the first city to vertex A in order to take advantage of symmetry) is only effective when n is at most 12 as $11! \approx 40M$. When $n > 12$, such brute force solutions will get a TLE in programming contests. However, if there are multiple test cases, the limit for such 'brute force' TSP solution is probably just $n = 11$.

We can utilize DP for TSP since the computation of sub-tours is clearly overlapping, e.g. the tour $A - B - C -$(*$n-3$ other cities that finally return to A*) clearly overlaps the tour $A - C - B -$*the same* ($n-3$ *other cities that also return to A*). If we can avoid re-computing the lengths of such sub-tours, we can save a lot of computation time. However, a distinct state in TSP depends on two parameters: The last city/vertex visited `pos` and something that we may have not seen before—a *subset* of visited cities.

There are many ways to represent a set. However, since we are going to pass this set information around as a parameter of a recursive function (if using top-down DP), the representation we use must be lightweight and efficient! In Section 2.2, we have presented a viable option for this usage: The *bitmask*. If we have n cities, we use a binary integer of length n. If bit i is '1' (on), we say that item (city) i is inside the set (it has been visited) and item i is not inside the set (and has not been visited) if the bit is instead '0' (off). For example: `mask`$= 18_{10} = 10010_2$ implies that items (cities) {1, 4} are in[19] the set (and have been visited). Recall that to check if bit i is on or off, we can use `mask & (1 << i)`. To set bit i, we can use `mask |= (1 << i)`.

Solution: Use these Complete Search recurrence relations for `tsp(pos, mask)`:

1. `tsp(pos, `$2^n - 1$`) = dist[pos][0]` // all cities have been visited, return to starting city
// Note: `mask = (1 << n) - 1` or $2^n - 1$ implies that all n bits in `mask` are on.
2. `tsp(pos, mask) = min(dist[pos][nxt] + tsp(nxt, mask | (1 << nxt)))`
// $\forall$ `nxt` $\in$ `[0..n-1]`, `nxt != pos`, and `(mask & (1 << nxt))` is '0' (turned off)
// We basically tries all possible next cities that have not been visited before at each step.

There are only $O(n \times 2^n)$ distinct states because there are n cities and we remember up to 2^n other cities that have been visited in each tour. Each state can be computed in $O(n)$,

[18] Such a tour is called a Hamiltonian tour, which is a cycle in an undirected graph which visits each vertex exactly once and also returns to the starting vertex.

[19] Remember that in `mask`, indices starts from 0 and are counted from the right.

thus the overall time complexity of this DP solution is $O(2^n \times n^2)$. This allows us to solve up to[20] $n \approx 16$ as $16^2 \times 2^{16} \approx 17M$. This is not a huge improvement over the brute force solution but if the programming contest problem involving TSP has input size $11 \leq n \leq 16$, then DP is the solution, not brute force. The answer can be found by calling `tsp(0, 1)`: We start from city 0 (we can start from any vertex; but the simplest choice is vertex 0) and set `mask = 1` so that city 0 is never re-visited again.

Usually, DP TSP problems in programming contests require some kind of graph preprocessing to generate the distance matrix `dist` before running the DP solution. These variants are discussed in Section 8.4.3.

DP solutions that involve a (small) set of Booleans as one of the parameters are more well known as the DP with bitmask technique. More challenging DP problems involving this technique are discussed in Section 8.3 and 9.2.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/rectree.html`

Source code: `ch3_09_UVa10496.cpp/java`

Exercise 3.5.2.1: The solution for the Max 2D Range Sum problem runs in $O(n^4)$. Actually, there exists an $O(n^3)$ solution that combines the DP solution for the Max Range 1D Sum problem on one dimension and uses the same idea as proposed by Kadane on the other dimension. Solve UVa 108 with an $O(n^3)$ solution!

Exercise 3.5.2.2: The solution for the `Range Minimum Query(i, j)` on 1D arrays in Section 2.4.3 uses Segment Tree. This is overkill if the given array is static and unchanged throughout all the queries. Use a DP technique to answer `RMQ(i, j)` in $O(n \log n)$ preprocessing and $O(1)$ per query.

Exercise 3.5.2.3: Solve the LIS problem using the $O(n \log k)$ solution and *also* reconstruct one of the LIS.

Exercise 3.5.2.4: Can we use an iterative Complete Search technique that tries all possible subsets of n items as discussed in Section 3.2.1 to solve the 0-1 Knapsack problem? What are the limitations, if any?

Exercise 3.5.2.5*: Suppose we add one more parameter to this classic 0-1 Knapsack problem. Let K_i denote the number of copies of item i for use in the problem. Example: $n = 2$, $V = \{100, 70\}$, $W = \{5, 4\}$, $K = \{2, 3\}$, $S = 17$ means that there are two copies of item 0 with weight 5 and value 100 and there are three copies of item 1 with weight 4 and value 70. The optimal solution for this example is to take one of item 0 and three of item 1, with a total weight of 17 and total value 310. Solve new variant of the problem assuming that $1 \leq n \leq 500$, $1 \leq S \leq 2000$, $n \leq \sum_{i=0}^{n-1} K_i \leq 40000$! Hint: Every integer can be written as a sum of powers of 2.

Exercise 3.5.2.6*: The DP TSP solution shown in this section can still be *slightly* enhanced to make it able to solve test case with $n = 17$ in contest environment. Show the required minor change to make this possible! Hint: Consider symmetry!

Exercise 3.5.2.7*: On top of the minor change asked in **Exercise 3.5.2.5***, what *other change(s)* is/are needed to have a DP TSP solution that is able to handle $n = 18$ (or even $n = 19$, but with much lesser number of test cases)?

---

[20] As programming contest problems usually require exact solutions, the DP-TSP solution presented here is already one of the best solutions. In real life, the TSP often needs to be solved for instances with thousands of cities. To solve larger problems like that, we have non-exact approaches like the ones presented in [26].

3.5.3 Non-Classical Examples

Although DP is the single most popular problem type with the highest frequency of appearance in recent programming contests, the classical DP problems in their *pure forms* usually never appear in modern ICPCs or IOIs again. We study them to understand DP, but we have to learn to solve many other non-classical DP problems (which may become classic in the near future) and develop our 'DP skills' in the process. In this subsection, we discuss two more non-classical examples, adding to the UVa 11450 - Wedding Shopping problem that we have discussed in detail earlier. We have also selected some easier non-classical DP problems as programming exercises. Once you have cleared most of these problems, you are welcome to explore the more challenging ones in the other sections in this book, e.g. Section 4.7.1, 5.4, 5.6, 6.5, 8.3, 9.2, 9.21, etc.

1. UVa 10943 - How do you add?

Abridged problem description: Given an integer n, how many ways can K non-negative integers less than or equal to n add up to n? Constraints: $1 \leq n, K \leq 100$. Example: For $n = 20$ and $K = 2$, there are 21 ways: $0 + 20$, $1 + 19$, $2 + 18$, $3 + 17$, ..., $20 + 0$.

Mathematically, the number of ways can be expressed as ${}^{(n+k-1)}C_{(k-1)}$ (see Section 5.4.2 about Binomial Coefficients). We will use this simple problem to re-illustrate Dynamic Programming principles that we have discussed in this section, especially the process of deriving appropriate states for a problem and deriving correct transitions from one state to another given the base case(s).

First, we have to determine the parameters of this problem to be selected to represent distinct states of this problem. There are only two parameters in this problem, n and K. Therefore, there are only 4 possible combinations:

1. If we do not choose any of them, we cannot represent a state. This option is ignored.
2. If we choose only n, then we do not know how many numbers $\leq n$ have been used.
3. If we choose only K, then we do not know the target sum n.
4. Therefore, the state of this problem should be represented by a pair (or tuple) (n, K). The order of chosen parameter(s) does not matter, i.e. the pair (K, n) is also OK.

Next, we have to determine the base case(s). It turns out that this problem is very easy when $K = 1$. Whatever n is, there is only *one way* to add exactly one number less than or equal to n to get n: Use n itself. There is no other base case for this problem.

For the general case, we have this recursive formulation which is not too difficult to derive: At state (n, K) where $K > 1$, we can split n into one number $X \in$ `[0..n]` and $n - X$, i.e. $n = X + (n - X)$. By doing this, we arrive at the subproblem $(n - X, K - 1)$, i.e. given a number $n - X$, how many ways can $K - 1$ numbers less than or equal to $n - X$ add up to $n - X$? We can then sum all these ways.

These ideas can be written as the following Complete Search recurrence `ways(n, K)`:

1. `ways(n, 1) = 1` // we can only use 1 number to add up to n, the number n itself
2. `ways(n, K)` = $\sum_{X=0}^{n}$ `ways(n - X, K - 1)` // sum all possible ways, recursively

This problem has overlapping sub-problems. For example, the test case $n = 1, K = 3$ has overlapping sub-problems: The state $(n = 0, K = 1)$ is reached twice (see Figure 4.39 in Section 4.7.1). However, there are only $n \times K$ possible states of (n, K). The cost of computing each state is $O(n)$. Thus, the overall time complexity is $O(n^2 \times K)$. As $1 \leq n, K \leq 100$, this is feasible. The answer can be found by calling `ways(n, K)`.

Note that this problem actually just needs the result modulo $1M$ (i.e. the last 6 digits of the answer). See Section 5.5.8 for a discussion on modulo arithmetic computation.

Source code: `ch3_10_UVa10943.cpp/java`

2. UVa 10003 - Cutting Sticks

Abridged problem statement: Given a stick of length $1 \leq l \leq 1000$ and $1 \leq n \leq 50$ cuts to be made to the stick (the cut coordinates, lying in the range `[0..`l`]`, are given). The cost of a cut is determined by the length of the stick to be cut. Your task is to find a cutting sequence so that the overall cost is minimized.

Example: $l = 100$, $n = 3$, and cut coordinates: `A` $= \{25, 50, 75\}$ (already sorted)

If we cut from left to right, then we will incur cost = 225.
1. First cut is at coordinate 25, total cost so far = 100;
2. Second cut is at coordinate 50, total cost so far = 100 + 75 = 175;
3. Third cut is at coordinate 75, final total cost = 175 + 50 = 225;

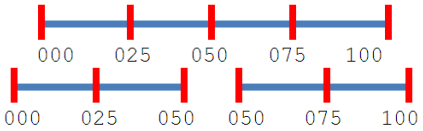


Figure 3.12: Cutting Sticks Illustration

However, the optimal answer is 200.
1. First cut is at coordinate 50, total cost so far = 100; (this cut is shown in Figure 3.12)
2. Second cut is at coordinate 25, total cost so far = 100 + 50 = 150;
3. Third cut is at coordinate 75, final total cost = 150 + 50 = 200;

How do we tackle this problem? An initial approach might be this Complete Search algorithm: Try all possible cutting points. Before that, we have to select an appropriate state definition for the problem: The (intermediate) sticks. We can describe a stick with its two endpoints: `left` and `right`. However, these two values can be very huge and this can complicate the solution later when we want to memoize their values. We can take advantage of the fact that there are only $n+1$ smaller sticks after cutting the original stick n times. The endpoints of each smaller stick can be described by 0, the cutting point coordinates, and l. Therefore, we will add two more coordinates so that `A` $= \{0,$ the original `A`, and $l\}$ so that we can denote a stick by the indices of its endpoints in `A`.

We can then use these recurrences for `cut(left, right)`, where `left`/`right` are the left/right indices of the current stick w.r.t. `A`. Originally, the stick is described by `left = 0` and `right =` n`+1`, i.e. a stick with length `[0..`l`]`:

1. `cut(i-1, i) = 0,` $\forall$`i` $\in$ `[1..`n`+1]` // if `left + 1 = right` where `left` and `right` are the indices in `A`, then we have a stick segment that does not need to be divided further.

2. `cut(left, right) = min(cut(left, i) + cut(i, right) + (A[right]-A[left]))`
$\forall$`i` $\in$ `[left+1..right-1]` // try all possible cutting points and pick the best.
The cost of a cut is the length of the current stick, captured in `(A[right]-A[left])`.
The answer can be found at `cut(0,` n`+1)`.

Now let's analyze the time complexity. Initially, we have n choices for the cutting points. Once we cut at a certain cutting point, we are left with $n-1$ further choices of the second

cutting point. This repeats until we are left with zero cutting points. Trying all possible cutting points this way leads to an $O(n!)$ algorithm, which is impossible for $1 \leq n \leq 50$.

However, this problem has overlapping sub-problems. For example, in Figure 3.12 above, cutting at index 2 (cutting point = 50) produces two states: `(0, 2)` and `(2, 4)`. The same state `(2, 4)` can also be reached by cutting at index 1 (cutting point 25) and then cutting at index 2 (cutting point 50). Thus, the search space is actually not that large. There are only $(n+2) \times (n+2)$ possible left/right indices or $O(n^2)$ distinct states and be memoized. The time to required to compute one state is $O(n)$. Thus, the overall time complexity (of the top-down DP) is $O(n^3)$. As $n \leq 50$, this is a feasible solution.

Source code: `ch3_11_UVa10003.cpp/java`

Exercise 3.5.3.1*: Almost all of the source code shown in this section (LIS, Coin Change, TSP, and UVa 10003 - Cutting Sticks) are written in a top-down DP fashion due to the preferences of the authors of this book. Rewrite them using the bottom-up DP approach.

Exercise 3.5.3.2*: Solve the Cutting Sticks problem in $O(n^2)$. Hint: Use Knuth-Yao DP Speedup by utilizing that the recurrence satisfies Quadrangle Inequality (see [2]).

Remarks About Dynamic Programming in Programming Contests

Basic (Greedy and) DP techniques techniques are always included in popular algorithm textbooks, e.g. Introduction to Algorithms [7], Algorithm Design [38] and Algorithm [8]. In this section, we have discussed six classical DP problems and their solutions. A brief summary is shown in Table 3.4. These classical DP problems, if they are to appear in a programming contest today, will likely occur only as part of bigger and harder problems.

	1D RSQ	2D RSQ	LIS	Knapsack	CC	TSP
State	`(i)`	`(i,j)`	`(i)`	`(id,remW)`	`(v)`	`(pos,mask)`
Space	$O(n)$	$O(n^2)$	$O(n)$	$O(nS)$	$O(V)$	$O(n2^n)$
Transition	subarray	submatrix	all $j < i$	take/ignore	all n coins	all n cities
Time	$O(1)$	$O(1)$	$O(n^2)$	$O(nS)$	$O(nV)$	$O(2^n n^2)$

Table 3.4: Summary of Classical DP Problems in this Section

To help keep up with the growing difficulty and creativity required in these techniques (especially the non-classical DP), we recommend that you also read the TopCoder algorithm tutorials [30] and attempt the more recent programming contest problems.

In this book, we will revisit DP again on several occasions: Floyd Warshall's DP algorithm (Section 4.5), DP on (implicit) DAG (Section 4.7.1), String Alignment (Edit Distance), Longest Common Subsequence (LCS), other DP on String algorithms (Section 6.5), More Advanced DP (Section 8.3), and several topics on DP in Chapter 9.

In the past (1990s), a contestant who is good at DP can become a 'king of programming contests' as DP problems were usually the 'decider problems'. Now, mastering DP is a *basic* requirement! You cannot do well in programming contests without this knowledge. However, we have to keep reminding the readers of this book not to claim that they know DP if they only memorize the solutions of the classical DP problems! Try to master the art of DP problem solving: Learn to determine the states (the DP table) that can uniquely

and efficiently represent sub-problems and also how to fill up that table, either via top-down recursion or bottom-up iteration.

There is no better way to master these problem solving paradigms than solving real programming problems! Here, we list several examples. Once you are familiar with the examples shown in this section, study the newer DP problems that have begun to appear in recent programming contests.

Programming Exercises solvable using Dynamic Programming:

- Max 1D Range Sum
  1. UVa 00507 - Jill Rides Again (standard problem)
  2. **UVa 00787 - Maximum Sub ... *** (max 1D range *product*, be careful with 0, use Java BigInteger, see Section 5.3)
  3. **UVa 10684 - The Jackpot *** (standard problem; easily solvable with the given sample source code)
  4. ***UVa 10755 - Garbage Heap **** (combination of max 2D range sum in two of the three dimensions—see below—and max 1D range sum using Kadane's algorithm on the third dimension)
     See more examples in Section 8.4.
- Max 2D Range Sum
  1. **UVa 00108 - Maximum Sum *** (discussed in this section with sample source code)
  2. UVa 00836 - Largest Submatrix (convert '0' to -INF)
  3. UVa 00983 - Localized Summing for ... (max 2D range sum, get submatrix)
  4. UVa 10074 - Take the Land (standard problem)
  5. UVa 10667 - Largest Block (standard problem)
  6. **UVa 10827 - Maximum Sum on ... *** (copy $n \times n$ matrix into $n \times 2n$ matrix; then this problem becomes a standard problem again)
  7. ***UVa 11951 - Area **** (use long long; max 2D range sum; prune the search space whenever possible)
- Longest Increasing Subsequence (LIS)
  1. UVa 00111 - History Grading (be careful of the ranking system)
  2. UVa 00231 - Testing the Catcher (straight-forward)
  3. UVa 00437 - The Tower of Babylon (can be modeled as LIS)
  4. **UVa 00481 - What Goes Up? *** (use $O(n \log k)$ LIS; print solution; see our sample source code)
  5. UVa 00497 - Strategic Defense Initiative (solution must be printed)
  6. UVa 01196 - Tiling Up Blocks (LA 2815, Kaohsiung03; sort all the blocks in increasing `L[i]`, then we get the classical LIS problem)
  7. UVa 10131 - Is Bigger Smarter? (sort elephants based on decreasing IQ; LIS on increasing weight)
  8. UVa 10534 - Wavio Sequence (must use $O(n \log k)$ LIS twice)
  9. *UVa 11368 - Nested Dolls* (sort in one dimension, LIS in the other)
  10. **UVa 11456 - Trainsorting *** (max(LIS(i) + LDS(i) - 1), $\forall i \in [0 \ldots n\text{-}1]$)
  11. **UVa 11790 - Murcia's Skyline *** (combination of LIS+LDS, weighted)

- 0-1 Knapsack (Subset Sum)
  1. UVa 00562 - Dividing Coins (use a one dimensional table)
  2. UVa 00990 - Diving For Gold (print the solution)
  3. UVa 01213 - Sum of Different Primes (LA 3619, Yokohama06, extension of 0-1 Knapsack, use three parameters: (id, remN, remK) on top of (id, remN))
  4. UVa 10130 - SuperSale (discussed in this section with sample source code)
  5. UVa 10261 - Ferry Loading (s: current car, left, right)
  6. **UVa 10616 - Divisible Group Sum *** (input can be -ve, use long long)
  7. UVa 10664 - Luggage (Subset Sum)
  8. **UVa 10819 - Trouble of 13-Dots *** (0-1 knapsack with 'credit card' twist!)
  9. *UVa 11003 - Boxes* (try all max weight from 0 to $max(weight[i]+capacity[i])$, $\forall i \in [0..n\text{-}1]$; if a max weight is known, how many boxes can be stacked?)
  10. UVa 11341 - Term Strategy (s: id, h_learned, h_left; t: learn module 'id' by 1 hour or skip)
  11. ***UVa 11566 - Let's Yum Cha ****** (English reading problem, actually just a knapsack variant: double each dim sum and add one parameter to check if we have bought too many dishes)
  12. UVa 11658 - Best Coalition (s: id, share; t: form/ignore coalition with id)
- Coin Change (CC)
  1. UVa 00147 - Dollars (similar to UVa 357 and UVa 674)
  2. UVa 00166 - Making Change (two coin change variants in one problem)
  3. **UVa 00357 - Let Me Count The Ways *** (similar to UVa 147/674)
  4. UVa 00674 - Coin Change (discussed in this section with sample source code)
  5. **UVa 10306 - e-Coins *** (variant: each coin has two components)
  6. UVa 10313 - Pay the Price (modified coin change + DP 1D range sum)
  7. UVa 11137 - Ingenuous Cubrency (use long long)
  8. **UVa 11517 - Exact Change *** (a variation to the coin change problem)
- Traveling Salesman Problem (TSP)
  1. **UVa 00216 - Getting in Line *** (TSP, still solvable with backtracking)
  2. **UVa 10496 - Collecting Beepers *** (discussed in this section with sample source code; actually, since $n \leq 11$, this problem is still solvable with recursive backtracking and sufficient pruning)
  3. **UVa 11284 - Shopping Trip *** (requires shortest paths pre-processing; TSP variant where we can go home early; we just need to tweak the DP TSP recurrence a bit: at each state, we have one more option: go home early)
     See more examples in Section 8.4.3 and Section 9.2.
- Non Classical (The Easier Ones)
  1. UVa 00116 - Unidirectional TSP (similar to UVa 10337)
  2. *UVa 00196 - Spreadsheet* (notice that the dependencies of cells are acyclic; we can therefore memoize the direct (or indirect) value of each cell)
  3. *UVa 01261 - String Popping* (LA 4844, Daejeon10, a simple backtracking problem; but we use a `set<string>` to prevent the same state (a substring) from being checked twice)
  4. UVa 10003 - Cutting Sticks (discussed in details in this section with sample source code)
  5. UVa 10036 - Divisibility (must use offset technique as value can be negative)

6. *UVa 10086 - Test the Rods* (s: idx, rem1, rem2; which site that we are now, up to 30 sites; remaining rods to be tested at NCPC; and remaining rods to be tested at BCEW; t: for each site, we split the rods, x rods to be tested at NCPC and $m[i] - x$ rods to be tested at BCEW; print the solution)
7. **UVa 10337 - Flight Planner *** (DP; shortest paths on DAG)
8. UVa 10400 - Game Show Math (backtracking with clever pruning is sufficient)
9. *UVa 10446 - The Marriage Interview* (edit the given recursive function a bit, add memoization)
10. UVa 10465 - Homer Simpson (one dimensional DP table)
11. *UVa 10520 - Determine it* (just write the given formula as a top-down DP with memoization)
12. *UVa 10688 - The Poor Giant* (note that the sample in the problem description is a bit wrong, it should be: 1+(1+3)+(1+3)+(1+3) = 1+4+4+4 = 13, beating 14; otherwise a simple DP)
13. **UVa 10721 - Bar Codes *** (s: n, k; t: try all from 1 to m)
14. UVa 10910 - Mark's Distribution (two dimensional DP table)
15. UVa 10912 - Simple Minded Hashing (s: len, last, sum; t: try next char)
16. **UVa 10943 - How do you add? *** (discussed in this section with sample source code; s: n, k; t: try all the possible splitting points; alternative solution is to use the closed form mathematical formula: $C(n+k-1, k-1)$ which also needs DP, see Section 5.4)
17. *UVa 10980 - Lowest Price in Town* (simple)
18. *UVa 11026 - A Grouping Problem* (DP, similar idea with binomial theorem in Section 5.4)
19. UVa 11407 - Squares (can be memoized)
20. UVa 11420 - Chest of Drawers (s: prev, id, numlck; lock/unlock this chest)
21. UVa 11450 - Wedding Shopping (discussed in details in this section with sample source code)
22. UVa 11703 - sqrt log sin (can be memoized)

- Other Classical DP Problems in this Book
  1. Floyd Warshall's for All-Pairs Shortest Paths problem (see Section 4.5)
  2. String Alignment (Edit Distance) (see Section 6.5)
  3. Longest Common Subsequence (see Section 6.5)
  4. Matrix Chain Multiplication (see Section 9.20)
  5. Max (Weighted) Independent Set (on tree, see Section 9.22)
- Also see Section 4.7.1, 5.4, 5.6, 6.5, 8.3, 8.4 and parts of Chapter 9 for *more* programming exercises related to Dynamic Programming.

3.6 Solution to Non-Starred Exercises

Exercise 3.2.1.1: This is to avoid the division operator so that we only work with integers! If we iterate through `abcde` instead, we may encounter a non-integer result when we compute `fghij = abcde / N`.

Exercise 3.2.1.2: It wil get an AC too as 10! $\approx$ 3 million, about the same as the algorithm presented in Section 3.2.1.

Exercise 3.2.2.1: Modify the backtrack function to resemble this code:

```
void backtrack(int c) {
  if (c == 8 && row[b] == a) {         // candidate sol, (a, b) has 1 queen
    printf("%2d      %d", ++lineCounter, row[0] + 1);
    for (int j = 1; j < 8; j++) printf(" %d", row[j] + 1);
    printf("\n"); }
  for (int r = 0; r < 8; r++)                        // try all possible row
    if (col == b && r != a) continue;                       // ADD THIS LINE
    if (place(r, c)) {          // if can place a queen at this col and row
      row[c] = r; backtrack(c + 1);      // put this queen here and recurse
}   }
```

Exercise 3.3.1.1: This problem can be solved without the 'binary search the answer' technique. Simulate the journey once. We just need to find the largest fuel requirement in the entire journey and make the fuel tank be sufficient for it.

Exercise 3.5.1.1: Garment $g = 0$, take the third model (cost 8); Garment $g = 1$, take the first model (cost 10); Garment $g = 2$, take the first model (cost 7); Money used = 25. Nothing left. Test case C is also solvable with Greedy algorithm.

Exercise 3.5.1.2: No, this state formulation does not work. We need to know how much money we have left at each sub-problem so that we can determine if we still have enough money to buy a certain model of the current garment.

Exercise 3.5.1.3: The modified bottom-up DP code is shown below:

```
#include <cstdio>
#include <cstring>
using namespace std;

int main() {
  int g, money, k, TC, M, C, cur;
  int price[25][25];
  bool reachable[2][210]; // reachable table[ONLY TWO ROWS][money (<= 200)]
  scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &M, &C);
    for (g = 0; g < C; g++) {
      scanf("%d", &price[g][0]);
      for (money = 1; money <= price[g][0]; money++)
        scanf("%d", &price[g][money]);
    }
```

```
    memset(reachable, false, sizeof reachable);
    for (g = 1; g <= price[0][0]; g++)
      if (M - price[0][g] >= 0)
        reachable[0][M - price[0][g]] = true;

    cur = 1;                                        // we start with this row
    for (g = 1; g < C; g++) {
      memset(reachable[cur], false, sizeof reachable[cur]);    // reset row
      for (money = 0; money < M; money++) if (reachable[!cur][money])
        for (k = 1; k <= price[g][0]; k++) if (money - price[g][k] >= 0)
          reachable[cur][money - price[g][k]] = true;
      cur = !cur;                      // IMPORTANT TRICK: flip the two rows
    }

    for (money = 0; money <= M && !reachable[!cur][money]; money++);

    if (money == M + 1) printf("no solution\n");  // last row has no on bit
    else                printf("%d\n", M - money);
} } // return 0;
```

Exercise 3.5.2.1: The $O(n^3)$ solution for Max 2D Range Sum problem is shown below:

```
scanf("%d", &n);                     // the dimension of input square matrix
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) {
  scanf("%d", &A[i][j]);
  if (j > 0) A[i][j] += A[i][j - 1];      // only add columns of this row i
}

maxSubRect = -127*100*100;    // the lowest possible value for this problem
for (int l = 0; l < n; l++) for (int r = l; r < n; r++) {
  subRect = 0;
  for (int row = 0; row < n; row++) {
    // Max 1D Range Sum on columns of this row i
    if (l > 0) subRect += A[row][r] - A[row][l - 1];
    else       subRect += A[row][r];

    // Kadane's algorithm on rows
    if (subRect < 0) subRect = 0;     // greedy, restart if running sum < 0
    maxSubRect = max(maxSubRect, subRect);
} }
```

Exercise 3.5.2.2: The solution is given in Section 9.33.

Exercise 3.5.2.3: The solution is already written inside `ch3_06_LIS.cpp/java`.

Exercise 3.5.2.4: The iterative Complete Search solution to generate and check all possible subsets of size n runs in $O(n \times 2^n)$. This is OK for $n \leq 20$ but too slow when $n > 20$. The DP solution presented in Section 3.5.2 runs in $O(n \times S)$. If S is not that large, we can have a much larger n than just 20 items.

3.7 Chapter Notes

Many problems in ICPC or IOI require a combination (see Section 8.4) of these problem solving strategies. If we have to nominate only one chapter in this book that contestants have to really master, we would choose this one.

In Table 3.5, we compare the four problem solving techniques in their likely results for various problem types. In Table 3.5 and the list of programming exercises in this section, you will see that there are *many more* Complete Search and DP problems than D&C and Greedy problems. Therefore, we recommend that readers concentrate on improving their Complete Search and DP skills.

	BF Problem	D&C Problem	Greedy Problem	DP Problem
BF Solution	AC	TLE/AC	TLE/AC	TLE/AC
D&C Solution	WA	AC	WA	WA
Greedy Solution	WA	WA	AC	WA
DP Solution	MLE/TLE/AC	MLE/TLE/AC	MLE/TLE/AC	AC
Frequency	High	(Very) Low	Low	High

Table 3.5: Comparison of Problem Solving Techniques (Rule of Thumb only)

We will conclude this chapter by remarking that for some real-life problems, especially those that are classified as NP-hard [7], many of the approaches discussed in this section will not work. For example, the 0-1 Knapsack Problem which has an $O(nS)$ DP complexity is too slow if S is big; TSP which has a $O(2^n \times n^2)$ DP complexity is too slow if n is any larger than 18 (see **Exercise 3.5.2.7***). For such problems, we can resort to heuristics or local search techniques such as Tabu Search [26, 25], Genetic Algorithms, Ant-Colony Optimizations, Simulated Annealing, Beam Search, etc. However, all these heuristic-based searches are not in the IOI syllabus [20] and also not widely used in ICPC.

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	32	32 (+0%)	52 (+63%)
Written Exercises	7	16 (+129%)	11+10*=21 (+31%)
Programming Exercises	109	194 (+78%)	245 (+26%)

The breakdown of the number of programming exercises from each section is shown below:

Section	Title	Appearance	% in Chapter	% in Book
3.2	**Complete Search**	112	45%	7%
3.3	Divide and Conquer	23	9%	1%
3.4	Greedy	45	18%	3%
3.5	**Dynamic Programming**	67	27%	4%

Chapter 4

Graph

Everyone is on average ≈ six steps away from any other person on Earth
— **Stanley Milgram - the Six Degrees of Separation experiment in 1969, [64]**

4.1 Overview and Motivation

Many real-life problems can be classified as graph problems. Some have efficient solutions. Some do not have them yet. In this relatively big chapter with lots of figures, we discuss graph problems that commonly appear in programming contests, the algorithms to solve them, and the *practical* implementations of these algorithms. We cover topics ranging from basic graph traversals, minimum spanning trees, single-source/all-pairs shortest paths, network flows, and discuss graphs with special properties.

In writing this chapter, we assume that the readers are *already* familiar with the graph terminologies listed in Table 4.1. If you encounter any unfamiliar term, please read other reference books like [7, 58] (or browse the Internet) and search for that particular term.

Vertices/Nodes	Edges	Set V; size $\|V\|$	Set E; size $\|E\|$	Graph $G(V, E)$
Un/Weighted	Un/Directed	Sparse	Dense	In/Out Degree
Path	Cycle	Isolated	Reachable	Connected
Self-Loop	Multiple Edges	Multigraph	Simple Graph	Sub-Graph
DAG	Tree/Forest	Eulerian	Bipartite	Complete

Table 4.1: List of Important Graph Terminologies

We also assume that the readers have read various ways to represent graph information that have been discussed earlier in Section 2.4.1. That is, we will directly use the terms like: Adjacency Matrix, Adjacency List, Edge List, and implicit graph without redefining them. Please revise Section 2.4.1 if you are not familiar with these graph data structures.

Our research so far on graph problems in recent ACM ICPC (Asia) regional contests reveals that there is at least one (and possibly more) graph problem(s) in an ICPC problem set. However, since the range of graph problems is so big, each graph problem only has a small probability of appearance. So the question is "Which ones do we have to focus on?". In our opinion, there is no clear answer for this question. If you want to do well in ACM ICPC, you have no choice but to study and master all these materials.

For IOI, the syllabus [20] restricts IOI tasks to a subset of material mentioned in this chapter. This is logical as high school students competing in IOI are not expected to be well-versed with too many problem-specific algorithms. To assist the readers aspiring to take part in the IOI, we will mention whether a particular section in this chapter is currently outside the syllabus.

4.2 Graph Traversal

4.2.1 Depth First Search (DFS)

Depth First Search—abbreviated as DFS—is a simple algorithm for traversing a graph. Starting from a distinguished source vertex, DFS will traverse the graph 'depth-first'. Every time DFS hits a branching point (a vertex with more than one neighbors), DFS will choose one of the unvisited neighbor(s) and visit this neighbor vertex. DFS repeats this process and goes deeper until it reaches a vertex where it cannot go any deeper. When this happens, DFS will 'backtrack' and explore another unvisited neighbor(s), if any.

This graph traversal behavior can be implemented easily with the recursive code below. Our DFS implementation uses the help of a *global* vector of integers: `vi dfs_num` to distinguish the state of each vertex. For the simplest DFS implementation, we only use `vi dfs_num` to distinguish between 'unvisited' (we use a constant value `UNVISITED = -1`) and 'visited' (we use another constant value `VISITED = 1`). Initially, all values in `dfs_num` are set to 'unvisited'. We will use `vi dfs_num` for other purposes later. Calling `dfs(u)` starts DFS from a vertex u, marks vertex u as 'visited', and then DFS recursively visits each 'unvisited' neighbor v of u (i.e. edge $u - v$ exists in the graph and `dfs_num[v] == UNVISITED`).

```
typedef pair<int, int> ii; // In this chapter, we will frequently use these
typedef vector<ii> vii; // three data type shortcuts. They may look cryptic
typedef vector<int> vi;   // but they are useful in competitive programming

vi dfs_num;   // global variable, initially all values are set to UNVISITED

void dfs(int u) {     // DFS for normal usage: as graph traversal algorithm
  dfs_num[u] = VISITED;        // important: we mark this vertex as visited
  for (int j = 0; j < (int)AdjList[u].size(); j++) { // default DS: AdjList
    ii v = AdjList[u][j];                 // v is a (neighbor, weight) pair
    if (dfs_num[v.first] == UNVISITED)    // important check to avoid cycle
      dfs(v.first);   // recursively visits unvisited neighbors of vertex u
} }  // for simple graph traversal, we ignore the weight stored at v.second
```

The time complexity of this DFS implementation depends on the graph data structure used. In a graph with V vertices and E edges, DFS runs in $O(V + E)$ and $O(V^2)$ if the graph is stored as Adjacency List and Adjacency Matrix, respectively (see **Exercise 4.2.2.2**).

On the sample graph in Figure 4.1, `dfs(0)`—calling DFS from a starting vertex $u = 0$—will trigger this sequence of visitation: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$. This sequence is 'depth-first', i.e. DFS goes to the deepest possible vertex from the start vertex before attempting another branch (there is none in this case).

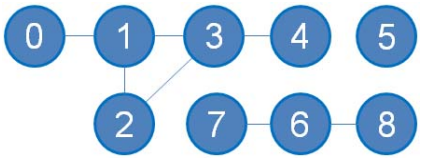


Figure 4.1: Sample Graph

Note that this sequence of visitation depends very much on the way we order the neighbors of a vertex[1], i.e. the sequence $0 \rightarrow 1 \rightarrow 3 \rightarrow 2$ (backtrack to 3) $\rightarrow 4$ is also a possible visitation sequence.

Also notice that one call of `dfs(u)` will only visit all vertices that are *connected* to vertex u. That is why vertices 5, 6, 7, and 8 in Figure 4.1 remain unvisited after calling `dfs(0)`.

[1]For simplicity, we usually just order the vertices based on their vertex numbers, e.g. in Figure 4.1, vertex 1 has vertex {0, 2, 3} as its neighbor, in that order.

The DFS code shown here is very similar to the recursive backtracking code shown earlier in Section 3.2. If we compare the pseudocode of a typical backtracking code (replicated below) with the DFS code shown above, we can see that the main difference is the flagging of visited vertices (states). Backtracking (automatically) un-flag visited vertices (reset the state to previous state) when the recursion backtracks to allow re-visitation of those vertices (states) from another branch. By not revisiting vertices of a general graph (via `dfs_num` checks), DFS runs in $O(V + E)$, but the time complexity of backtracking is exponential.

```
void backtrack(state) {
  if (hit end state or invalid state)             // we need terminating or
    return;    // pruning condition to avoid cycling and to speed up search
  for each neighbor of this state                    // try all permutation
    backtrack(neighbor);
}
```

Sample Application: UVa 11902 - Dominator

Abridged problem description: Vertex X dominates vertex Y if every path from the a start vertex (vertex 0 for this problem) to Y must go through X. If Y is not reachable from the start vertex then Y does not have any dominator. Every vertex reachable from the start vertex dominates itself. For example, in the graph shown in Figure 4.2, vertex 3 dominates vertex 4 since all the paths from vertex 0 to vertex 4 must pass through vertex 3. Vertex 1 does not dominate vertex 3 since there is a path 0-2-3 that does not include vertex 1. Our task: Given a directed graph, determine the dominators of every vertex.

This problem is about reachability tests from a start vertex (vertex 0). Since the input graph for this problem is small ($V < 100$), we can afford to use the following $O(V \times V^2 = V^3)$ algorithm. Run `dfs(0)` on the input graph to record vertices that are reachable from vertex 0. Then to check which vertices are dominated by vertex X, we (temporarily) turn off all the outgoing edges of vertex X and rerun `dfs(0)`. Now, a vertex Y is not dominated by vertex X if `dfs(0)` initially cannot reach vertex Y or `dfs(0)` can reach vertex Y even after all outgoing edges of vertex X are (temporarily) turned off. Vertex Y is dominated by vertex X otherwise. We repeat this process $\forall X \in [0 \ldots V-1]$.

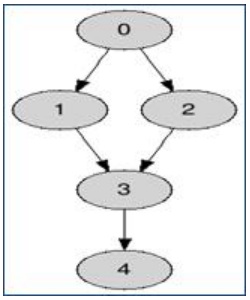


Figure 4.2: UVa 11902

Tips: We do not have to physically delete vertex X from the input graph. We can simply add a statement inside our DFS routine to stop the traversal if it hits vertex X.

4.2.2 Breadth First Search (BFS)

Breadth First Search—abbreviated as BFS—is another graph traversal algorithm. Starting from a distinguished source vertex, BFS will traverse the graph 'breadth-first'. That is, BFS will visit vertices that are direct neighbors of the source vertex (first layer), neighbors of direct neighbors (second layer), and so on, layer by layer.

BFS starts with the insertion of the source vertex s into a queue, then processes the queue as follows: Take out the front most vertex u from the queue, enqueue all unvisited neighbors of u (usually, the neighbors are ordered based on their vertex numbers), and mark them as visited. With the help of the queue, BFS will visit vertex s and all vertices in the connected component that contains s layer by layer. BFS algorithm also runs in $O(V + E)$ and $O(V^2)$

on a graph represented using an Adjacency List and Adjacency Matrix, respectively (again, see **Exercise 4.2.2.2**).

Implementing BFS is easy if we utilize C++ STL or Java API. We use `queue` to order the sequence of visitation and `vector<int>` (or `vi`) to record if a vertex has been visited or not—which at the same time also record the distance (layer number) of each vertex from the source vertex. This distance computation feature is used later to solve a special case of Single-Source Shortest Paths problem (see Section 4.4 and 8.2.3).

```
// inside int main()---no recursion
  vi d(V, INF); d[s] = 0;                 // distance from source s to s is 0
  queue<int> q; q.push(s);                                 // start from source

  while (!q.empty()) {
    int u = q.front(); q.pop();                          // queue: layer by layer!
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];                              // for each neighbor of u
      if (d[v.first] == INF) {         // if v.first is unvisited + reachable
        d[v.first] = d[u] + 1;           // make d[v.first] != INF to flag it
        q.push(v.first);             // enqueue v.first for the next iteration
  } } }
```

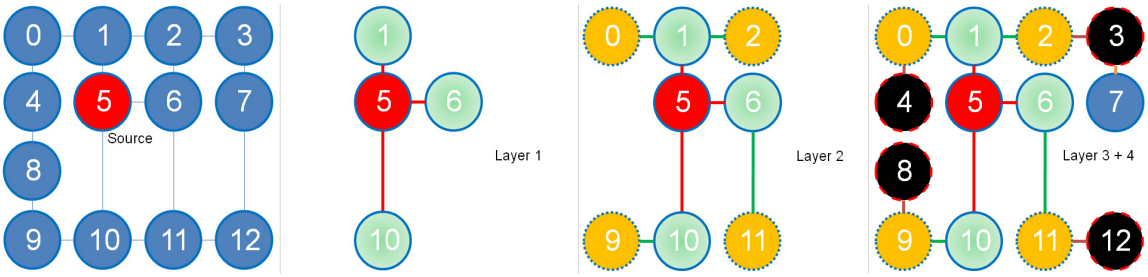


Figure 4.3: Example Animation of BFS

If we run BFS from vertex 5 (i.e. the source vertex $s = 5$) on the connected undirected graph shown in Figure 4.3, we will visit the vertices in the following order:

```
Layer 0:, visit 5
Layer 1:, visit 1, visit 6, visit 10
Layer 2:, visit 0, visit 2, visit 11, visit 9
Layer 3:, visit 4, visit 3, visit 12, visit 8
Layer 4:, visit 7
```

Exercise 4.2.2.1: To show that either DFS or BFS can be used to visit all vertices that are reachable from a source vertex, solve UVa 11902 - Dominator using BFS instead!

Exercise 4.2.2.2: Why do DFS and BFS run in $O(V+E)$ if the graph is stored as Adjacency List and become slower (run in $O(V^2)$) if the graph is stored as Adjacency Matrix? Follow up question: What is the time complexity of DFS and BFS if the graph is stored as Edge List instead? What should we do if the input graph is given as an Edge List and we want to traverse the graph efficiently?

4.2.3 Finding Connected Components (Undirected Graph)

DFS and BFS are not only useful for traversing a graph. They can be used to solve many other graph problems. The first few problems below can be solved with *either* DFS or BFS although some of the last few problems are more suitable for DFS only.

The fact that one single call of `dfs(u)` (or `bfs(u)`) will only visit vertices that are actually connected to u can be utilized to find (and to count the number of) connected components in an *undirected* graph (see further below in Section 4.2.9 for a similar problem on directed graph). We can simply use the following code to restart DFS (or BFS) from one of the remaining unvisited vertices to find the next connected component. This process is repeated until all vertices have been visited and has an overall time complexity of $O(V + E)$.

```
// inside int main()---this is the DFS solution
  numCC = 0;
  dfs_num.assign(V, UNVISITED);    // sets all vertices' state to UNVISITED
  for (int i = 0; i < V; i++)              // for each vertex i in [0..V-1]
    if (dfs_num[i] == UNVISITED)          // if vertex i is not visited yet
      printf("CC %d:", ++numCC), dfs(i), printf("\n");     // 3 lines here!

// For the sample graph in Figure 4.1, the output is like this:
// CC 1: 0 1 2 3 4
// CC 2: 5
// CC 3: 6 7 8
```

Exercise 4.2.3.1: UVa 459 - Graph Connectivity is basically this problem of finding connected components of an undirected graph. Solve it using the DFS solution shown above! However, we can also use Union-Find Disjoint Sets data structure (see Section 2.4.2) or BFS (see Section 4.2.2) to solve this graph problem. How?

4.2.4 Flood Fill - Labeling/Coloring the Connected Components

DFS (or BFS) can be used for other purposes than just finding (and counting the number of) connected components. Here, we show how a *simple tweak* of the $O(V + E)$ `dfs(u)` (we can also use `bfs(u)`) can be used to *label* (also known in CS terminology as 'to color') and count the size of each component. This variant is more famously known as 'flood fill' and usually performed on *implicit* graphs (usually 2D grids).

```
int dr[] = {1,1,0,-1,-1,-1, 0, 1};  // trick to explore an implicit 2D grid
int dc[] = {0,1,1, 1, 0,-1,-1,-1};         // S,SE,E,NE,N,NW,W,SW neighbors

int floodfill(int r, int c, char c1, char c2) {   // returns the size of CC
  if (r < 0 || r >= R || c < 0 || c >= C) return 0;         // outside grid
  if (grid[r][c] != c1) return 0;                  // does not have color c1
  int ans = 1;   // adds 1 to ans because vertex (r, c) has c1 as its color
  grid[r][c] = c2;    // now recolors vertex (r, c) to c2 to avoid cycling!
  for (int d = 0; d < 8; d++)
    ans += floodfill(r + dr[d], c + dc[d], c1, c2);
  return ans;                       // the code is neat due to dr[] and dc[]
}
```

Sample Application: UVa 469 - Wetlands of Florida

Let's see an example below (UVa 469 - Wetlands of Florida). The implicit graph is a 2D grid where the vertices are the cells in the grid and the edges are the connections between a cell and its S/SE/E/NE/N/NW/W/SW cells. 'W' denotes a wet cell and 'L' denotes a land cell. Wet area is defined as *connected cells* labeled with 'W'. We can label (and simultaneously count the size of) a wet area by using floodfill. The example below shows an execution of floodfill from row 2, column 1 (0-based indexing), replacing 'W' to '.'.

We want to make a remark that there are a good number of floodfill problems in UVa online judge [47] with a high profile example: UVa 1103 - Ancient Messages (ICPC World Finals problem in 2011). It may be beneficial for the readers to attempt floodfill problems listed in programming exercises of this section to master this technique!

```
// inside int main()
  // read the grid as a global 2D array + read (row, col) query coordinates
  printf("%d\n", floodfill(row, col, 'W', '.'));  // count size of wet area
                                                  // the returned answer is 12
// LLLLLLLLL         LLLLLLLLL
// LLWWLLWLL         LL..LLWLL   //         The size of connected component
// LWWLLLLLL (R2,C1) L..LLLLLL   //                     (the connected 'W's)
// LWWWLWWLL         L...L..LL   // with one 'W' at (row 2, column 1) is 12
// LLLWWWLLL ======> LLL...LLL
// LLLLLLLLL         LLLLLLLLL   //    Notice that all these connected 'W's
// LLLWWLLWL         LLLWWLLWL   //  are replaced with '.'s after floodfill
// LLWLWLLLL         LLWLWLLLL
// LLLLLLLLL         LLLLLLLLL
```

4.2.5 Topological Sort (Directed Acyclic Graph)

Topological sort (or topological ordering) of a Directed Acyclic Graph (DAG) is a linear ordering of the vertices in the DAG so that vertex u comes before vertex v if edge $(u \rightarrow v)$ exists in the DAG. Every DAG has at least one *and possibly more* topological sort(s).

One application of topological sorting is to find a possible sequence of modules that a University student has to take to fulfill graduation requirement. Each module has certain pre-requisites to be met. These pre-requisites are never cyclic, so they can be modeled as a DAG. Topological sorting this module pre-requisites DAG gives the student a linear list of modules to be taken one after another without violating the pre-requisites constraints.

There are several algorithms for topological sort. The simplest way is to slightly modify the DFS implementation we presented earlier in Section 4.2.1.

```
vi ts;              // global vector to store the toposort in reverse order

void dfs2(int u) {  // different function name compared to the original dfs
  dfs_num[u] = VISITED;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED)
      dfs2(v.first);
  }
  ts.push_back(u); }                  // that's it, this is the only change
```

```
// inside int main()
  ts.clear();
  memset(dfs_num, UNVISITED, sizeof dfs_num);
  for (int i = 0; i < V; i++)       // this part is the same as finding CCs
    if (dfs_num[i] == UNVISITED)
      dfs2(i);
                // alternative, call: reverse(ts.begin(), ts.end()); first
  for (int i = (int)ts.size() - 1; i >= 0; i--)           // read backwards
    printf(" %d", ts[i]);
  printf("\n");

// For the sample graph in Figure 4.4, the output is like this:
// 7 6 0 1 2 5 3 4   (remember that there can be >= 1 valid toposort)
```

In `dfs2(u)`, we append u to the back of a list (vector) of explored vertices only after visiting all the subtrees below u in the DFS spanning tree[2]. We append u to the *back* of this vector because C++ STL `vector` (Java `Vector`) only supports *efficient* $O(1)$ *insertion* from the back. The list will be in reversed order, but we can work around this issue by reversing the print order in the output phase. This simple algorithm for finding (a valid) topological sort is due to Robert Endre Tarjan. It runs in $O(V+E)$ as with DFS as it does the same work as the original DFS plus one constant operation.

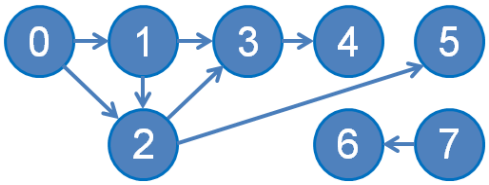


Figure 4.4: An Example of DAG

To complete the discussion about topological sort, we show another algorithm for finding topological sort: Kahn's algorithm [36]. It looks like a 'modified BFS'. Some problems, e.g. UVa 11060 - Beverages, requires this Kahn's algorithm to produce the required topological sort instead of the DFS-based algorithm shown earlier.

```
enqueue vertices with zero incoming degree into a (priority) queue Q;
while (Q is not empty) {
  vertex u = Q.dequeue(); put vertex u into a topological sort list;
  remove this vertex u and all outgoing edges from this vertex;
  if such removal causes vertex v to have zero incoming degree
    Q.enqueue(v); }
```

Exercise 4.2.5.1: Why appending vertex `u` at the back of `vi ts`, i.e. `ts.push_back(u)` in the standard DFS code is enough to help us find the topological sort of a DAG?

Exercise 4.2.5.2: Can you identify another data structure that supports efficient $O(1)$ insertion *from front* so that we do not have to reverse the content of `vi ts`?

Exercise 4.2.5.3: What happen if we run topological sort code above on a non DAG?

Exercise 4.2.5.4: The topological sort code shown above can only generate *one* valid topological ordering of the vertices of a DAG. What should we do if we want to output *all* valid topological orderings of the vertices of a DAG?

[2]DFS spanning tree is discussed in more details in Section 4.2.7.

4.2.6 Bipartite Graph Check

Bipartite graph has important applications that we will see later in Section 4.7.4. In this subsection, we just want to check if a graph is bipartite (or 2/bi-colorable) to solve problems like UVa 10004 - Bicoloring. We can use either BFS or DFS for this check, but we feel that BFS is more natural. The modified BFS code below starts by coloring the source vertex (first layer) with value 0, color the direct neighbors of the source vertex (second layer) with value 1, color the neighbors of direct neighbors (third layer) with value 0 again, and so on, alternating between value 0 and value 1 as the only two valid colors. If we encounter any violation(s) along the way—an edge with two endpoints having the same color, then we can conclude that the given input graph is not a bipartite graph.

```
// inside int main()
  queue<int> q; q.push(s);
  vi color(V, INF); color[s] = 0;
  bool isBipartite = true;  // addition of one boolean flag, initially true
  while (!q.empty() & isBipartite) { // similar to the original BFS routine
    int u = q.front(); q.pop();
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];
      if (color[v.first] == INF) {   // but, instead of recording distance,
        color[v.first] = 1 - color[u];  // we just record two colors {0, 1}
        q.push(v.first); }
      else if (color[v.first] == color[u]) {  // u & v.first has same color
        isBipartite = false; break; } } }    // we have a coloring conflict
```

Exercise 4.2.6.1*: Implement bipartite check using DFS instead!

Exercise 4.2.6.2*: A *simple* graph with V vertices is found out to be a bipartite graph. What is the maximum possible number of edges that this graph has?

Exercise 4.2.6.3: Prove (or disprove) this statement: "Bipartite graph has no odd cycle"!

4.2.7 Graph Edges Property Check via DFS Spanning Tree

Running DFS on a connected graph generates a DFS *spanning tree*[3] (or *spanning forest*[4] if the graph is disconnected). With the help of one more vertex state: `EXPLORED = 2` (visited *but not yet completed*) on top of `VISITED` (visited *and completed*), we can use this DFS spanning tree (or forest) to classify graph edges into three types:

1. Tree edge: The edge traversed by DFS, i.e. an edge from a vertex currently with state: `EXPLORED` to a vertex with state: `UNVISITED`.
2. Back edge: Edge that is part of a cycle, i.e. an edge from a vertex currently with state: `EXPLORED` to a vertex with state: `EXPLORED` too. This is an important application of this algorithm. Note that we usually do not count bi-directional edges as having a 'cycle' (We need to remember `dfs_parent` to distinguish this, see the code below).
3. Forward/Cross edges from vertex with state: `EXPLORED` to vertex with state: `VISITED`. These two type of edges are not typically tested in programming contest problems.

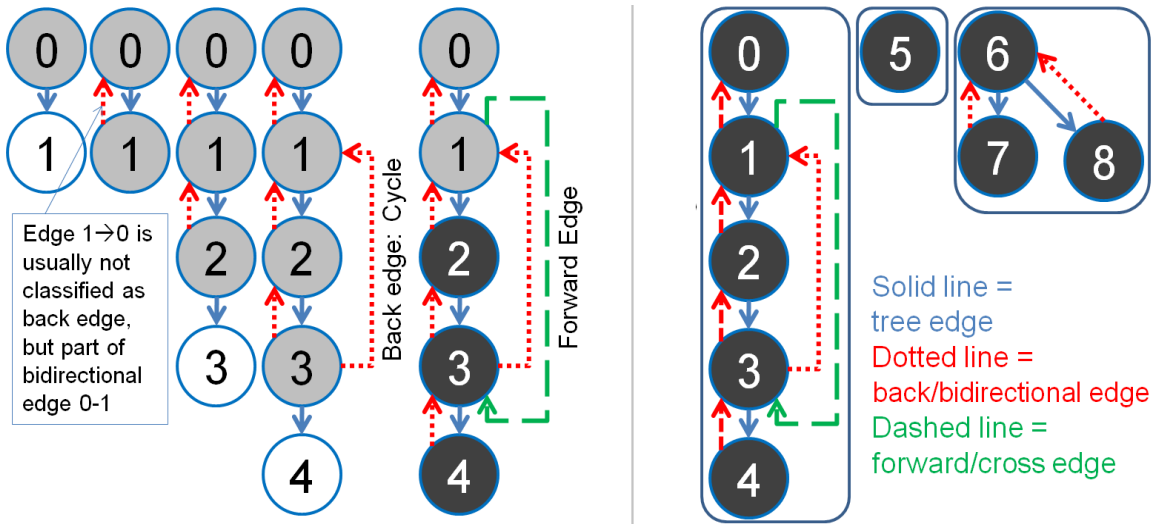


Figure 4.5: Animation of DFS when Run on the Sample Graph in Figure 4.1

Figure 4.5 shows an animation (from left to right) of calling `dfs(0)` (shown in more details), then `dfs(5)`, and finally `dfs(6)` on the sample graph in Figure 4.1. We can see that $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ is a (true) cycle and we classify edge $(3 \rightarrow 1)$ as a back edge, whereas $0 \rightarrow 1 \rightarrow 0$ is not a cycle but it is just a bi-directional edge (0-1). The code for this DFS variant is shown below.

```
void graphCheck(int u) {          // DFS for checking graph edge properties
  dfs_num[u] = EXPLORED;          // color u as EXPLORED instead of VISITED
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED) {  // Tree Edge, EXPLORED->UNVISITED
      dfs_parent[v.first] = u;            // parent of this children is me
      graphCheck(v.first);
    }
    else if (dfs_num[v.first] == EXPLORED) {          // EXPLORED->EXPLORED
      if (v.first == dfs_parent[u])     // to differentiate these two cases
        printf(" Two ways (%d, %d)-(%d, %d)\n", u, v.first, v.first, u);
      else   // the most frequent application: check if the graph is cyclic
        printf(" Back Edge (%d, %d) (Cycle)\n", u, v.first);
    }
    else if (dfs_num[v.first] == VISITED)              // EXPLORED->VISITED
      printf(" Forward/Cross Edge (%d, %d)\n", u, v.first);
  }
  dfs_num[u] = VISITED;       // after recursion, color u as VISITED (DONE)
}

// inside int main()
  dfs_num.assign(V, UNVISITED);
  dfs_parent.assign(V, 0);                                    // new vector
```

[3]A spanning tree of a connected graph G is a tree that spans (covers) all vertices of G but only using a subset of the edges of G.

[4]A disconnected graph G has several connected components. Each component has its own spanning subtree(s). All spanning subtrees of G, one from each component, form what we call a spanning forest.

```
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == UNVISITED)
      printf("Component %d:\n", ++numComp), graphCheck(i); // 2 lines in 1!

// For the sample graph in Figure 4.1, the output is like this:
// Component 1:
//  Two ways (1, 0) - (0, 1)
//  Two ways (2, 1) - (1, 2)
//  Back Edge (3, 1) (Cycle)
//  Two ways (3, 2) - (2, 3)
//  Two ways (4, 3) - (3, 4)
//  Forward/Cross Edge (1, 3)
// Component 2:
// Component 3:
//  Two ways (7, 6) - (6, 7)
//  Two ways (8, 6) - (6, 8)
```

Exercise 4.2.7.1: Perform graph edges property check on the graph in Figure 4.9. Assume that you start DFS from vertex 0. How many back edges that you can find this time?

4.2.8 Finding Articulation Points and Bridges (Undirected Graph)

Motivating problem: Given a road map (undirected graph) with sabotage costs associated to all intersections (vertices) and roads (edges), sabotage either a single intersection or a single road such that the road network breaks down (disconnected) and do so in the least cost way. This is a problem of finding the least cost Articulation Point (intersection) or the least cost Bridge (road) in an undirected graph (road map).

An 'Articulation Point' is defined as *a vertex* in a graph G whose removal (all edges incident to this vertex are also removed) disconnects G. A graph without any articulation point is called 'Biconnected'. Similarly, a 'Bridge' is defined as *an edge* in a graph G whose removal disconnects G. These two problems are usually defined for undirected graphs (they are more challenging for directed graphs and require another algorithm to solve, see [35]).

A naïve algorithm to find articulation points is as follows (can be tweaked to find bridges):

1. Run $O(V+E)$ DFS (or BFS) to count number of connected components (CCs) of the original graph. Usually, the input is a connected graph, so this check will usually gives us one connected component.
2. For each vertex $v \in V$ // $O(V)$
   (a) Cut (remove) vertex v and its incident edges
   (b) Run $O(V+E)$ DFS (or BFS) and see if the number of CCs increases
   (c) If yes, v is an articulation point/cut vertex; Restore v and its incident edges

This naïve algorithm calls DFS (or BFS) $O(V)$ times, thus it runs in $O(V \times (V+E)) = O(V^2 + VE)$. But this is *not* the best algorithm as we can actually just run the $O(V+E)$ DFS *once* to identify all the articulation points and bridges.

This DFS variant, due to John Edward Hopcroft and Robert Endre Tarjan (see [63] and problem 22.2 in [7]), is just another extension from the previous DFS code shown earlier.

We now maintain two numbers: `dfs_num(u)` and `dfs_low(u)`. Here, `dfs_num(u)` now stores the iteration counter when the vertex u is visited *for the first time* (not just for distinguishing `UNVISITED` versus `EXPLORED`/`VISITED`). The other number `dfs_low(u)` stores the lowest `dfs_num` reachable from the current DFS spanning subtree of u. At the beginning, `dfs_low(u) = dfs_num(u)` when vertex u is visited for the first time. Then, `dfs_low(u)` can only be made smaller if there is a cycle (a back edge exists). Note that we do not update `dfs_low(u)` with a back edge (u, v) if v is a direct parent of u.

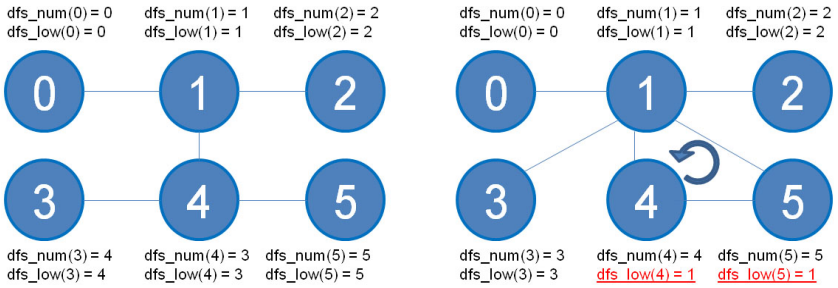


Figure 4.6: Introducing two More DFS Attributes: `dfs_num` and `dfs_low`

See Figure 4.6 for clarity. In both graphs, we run the DFS variant from vertex 0. Suppose for the graph in Figure 4.6—left side, the sequence of visitation is 0 (at iteration 0) → 1 (1) → 2 (2) (backtrack to 1) → 4 (3) → 3 (4) (backtrack to 4) → 5 (5). See that these iteration counters are shown correctly in `dfs_num`. As there is no back edge in this graph, all `dfs_low = dfs_num`.

Suppose for the graph in Figure 4.6—right side, the sequence of visitation is 0 (at iteration 0) → 1 (1) → 2 (2) (backtrack to 1) → 3 (3) (backtrack to 1) → 4 (4) → 5 (5). At this point in the DFS spanning tree, there is an important back edge that forms a cycle, i.e. edge 5-1 that is part of cycle 1-4-5-1. This causes vertices 1, 4, and 5 to be able to reach vertex 1 (with `dfs_num` 1). Thus `dfs_low` of {1, 4, 5} are all 1.

When we are in a vertex u with v as its neighbor and `dfs_low(v)` $\geq$ `dfs_num(u)`, then u is an articulation vertex. This is because the fact that `dfs_low(v)` is *not smaller* than `dfs_num(u)` implies that there is *no back edge* from vertex v that can reach another vertex w with a lower `dfs_num(w)` than `dfs_num(u)`. A vertex w with lower `dfs_num(w)` than vertex u with `dfs_num(u)` implies that w is the ancestor of u in the DFS spanning tree. This means that to reach the ancestor(s) of u from v, one *must* pass through vertex u. Therefore, removing vertex u will disconnect the graph.

However, there is one **special case**: The root of the DFS spanning tree (the vertex chosen as the start of DFS call) is an articulation point only if it has more than one children in the DFS spanning tree (a trivial case that is not detected by this algorithm).

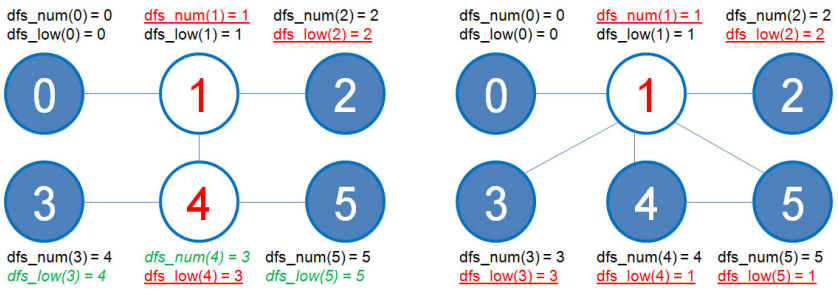


Figure 4.7: Finding Articulation Points with `dfs_num` and `dfs_low`

See Figure 4.7 for more details. On the graph in Figure 4.7—left side, vertices 1 and 4 are articulation points, because for example in edge 1-2, we see that `dfs_low(2)` $\geq$ `dfs_num(1)`

and in edge 4-5, we also see that `dfs_low(5)` $\geq$ `dfs_num(4)`. On the graph in Figure 4.7—right side, only vertex 1 is the articulation point, because for example in edge 1-5, `dfs_low(5)` $\geq$ `dfs_num(1)`.

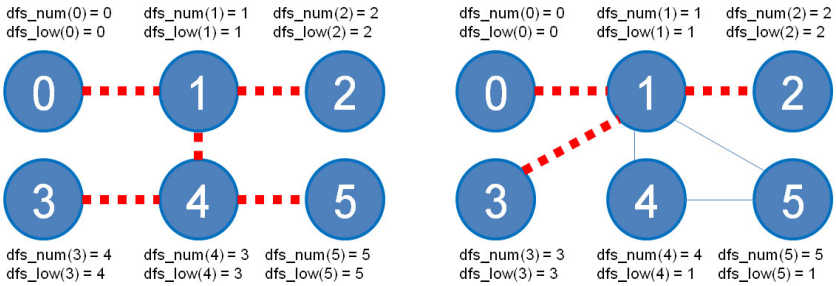


Figure 4.8: Finding Bridges, also with `dfs_num` and `dfs_low`

The process to find bridges is similar. When `dfs_low(v)` > `dfs_num(u)`, then edge u-v is a bridge (notice that we remove the equality test '=' for finding bridges). In Figure 4.8, almost all edges are bridges for the left and right graph. Only edges 1-4, 4-5, and 5-1 are not bridges on the right graph (they actually form a cycle). This is because—for example—for edge 4-5, we have `dfs_low(5)` $\leq$ `dfs_num(4)`, i.e. even if this edge 4-5 is removed, we know for sure that vertex 5 can still reach vertex 1 via *another path* that bypass vertex 4 as `dfs_low(5) = 1` (that other path is actually edge 5-1). The code is shown below:

```
void articulationPointAndBridge(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED) {                    // a tree edge
      dfs_parent[v.first] = u;
      if (u == dfsRoot) rootChildren++;      // special case if u is a root

      articulationPointAndBridge(v.first);

      if (dfs_low[v.first] >= dfs_num[u])         // for articulation point
        articulation_vertex[u] = true;      // store this information first
      if (dfs_low[v.first] > dfs_num[u])                      // for bridge
        printf(" Edge (%d, %d) is a bridge\n", u, v.first);
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);  // update dfs_low[u]
    }
    else if (v.first != dfs_parent[u])  // a back edge and not direct cycle
      dfs_low[u] = min(dfs_low[u], dfs_num[v.first]);  // update dfs_low[u]
} }

// inside int main()
  dfsNumberCounter = 0; dfs_num.assign(V, UNVISITED); dfs_low.assign(V, 0);
  dfs_parent.assign(V, 0); articulation_vertex.assign(V, 0);
  printf("Bridges:\n");
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == UNVISITED) {
      dfsRoot = i; rootChildren = 0; articulationPointAndBridge(i);
      articulation_vertex[dfsRoot] = (rootChildren > 1); }  // special case
```

```
printf("Articulation Points:\n");
for (int i = 0; i < V; i++)
  if (articulation_vertex[i])
    printf(" Vertex %d\n", i);
```

Exercise 4.2.8.1: Examine the graph in Figure 4.1 without running the algorithm above. Which vertices are articulation points and which edges are bridges? Now run the algorithm and verify if the computed `dfs_num` and `dfs_low` of each vertex of Figure 4.1 graph can be used to identify the same articulation points and bridges found manually!

4.2.9 Finding Strongly Connected Components (Directed Graph)

Yet another application of DFS is to find *strongly* connected components in a *directed* graph, e.g. UVa 11838 - Come and Go. This is a different problem to finding connected components in an undirected graph. In Figure 4.9, we have a similar graph to the graph in Figure 4.1, but now the edges are directed. Although the graph in Figure 4.9 looks like it has one 'connected' component, it is actually not a 'strongly connected' component. In directed graphs, we are more interested with the notion of 'Strongly Connected Component (SCC)'. An SCC is defined as such: If we pick any pair of vertices u and v in the SCC, we can find a path from u to v and vice versa. There are actually three SCCs in Figure 4.9, as highlighted with the three boxes: {0}, {1, 3, 2}, and {4, 5, 7, 6}. Note: If these SCCs are contracted (replaced by larger vertices), they form a DAG (also see Section 8.4.3).

There are at least two known algorithms to find SCCs: Kosaraju's—explained in [7] and Tarjan's algorithm [63]. In this section, we adopt Tarjan's version, as it extends naturally from our previous discussion of finding Articulation Points and Bridges—also due to Tarjan. We will discuss Kosaraju's algorithm later in Section 9.17.

The basic idea of the algorithm is that SCCs form subtrees in the DFS spanning tree (compare the original directed graph and the DFS spanning tree in Figure 4.9). On top of computing `dfs_num(u)` and `dfs_low(u)` for each vertex, we also append vertex u to the back of a stack S (here the stack is implemented with a vector) and keep track of the vertices that are currently explored via `vi visited`. The condition to update `dfs_low(u)` is slightly different from the previous DFS algorithm for finding articulation points and bridges. Here, only vertices that currently have `visited` flag turned on (part of the current SCC) that can update `dfs_low(u)`. Now, if we have vertex u in this DFS spanning tree with `dfs_low(u) = dfs_num(u)`, we can conclude that u is the root (start) of an SCC (observe vertex 0, 1, and 4) in Figure 4.9) and the members of those SCCs are identified by popping the current content of stack S until we reach vertex u (the root) of SCC again.

In Figure 4.9, the content of S is $\{0, 1, 3, 2, \underline{4, 5, 7, 6}\}$ when vertex 4 is identified as the root of an SCC (`dfs_low(4) = dfs_num(4) = 4`), so we pop elements in S one by one until we reach vertex 4 and we have this SCC: {6, 7, 5, 4}. Next, the content of S is $\{0, \underline{1, 3, 2}\}$ when vertex 1 is identified as another root of another SCC (`dfs_low(1) = dfs_num(1) = 1`), so we pop elements in S one by one until we reach vertex 1 and we have SCC: {2, 3, 1}. Finally, we have the last SCC with one member only: {0}.

The code given below explores the directed graph and reports its SCCs. This code is basically a tweak of the standard DFS code. The recursive part is similar to standard DFS and the SCC reporting part will run in amortized $O(V)$ times, as each vertex will only belong to one SCC and thus reported only once. In overall, this algorithm still runs in $O(V + E)$.

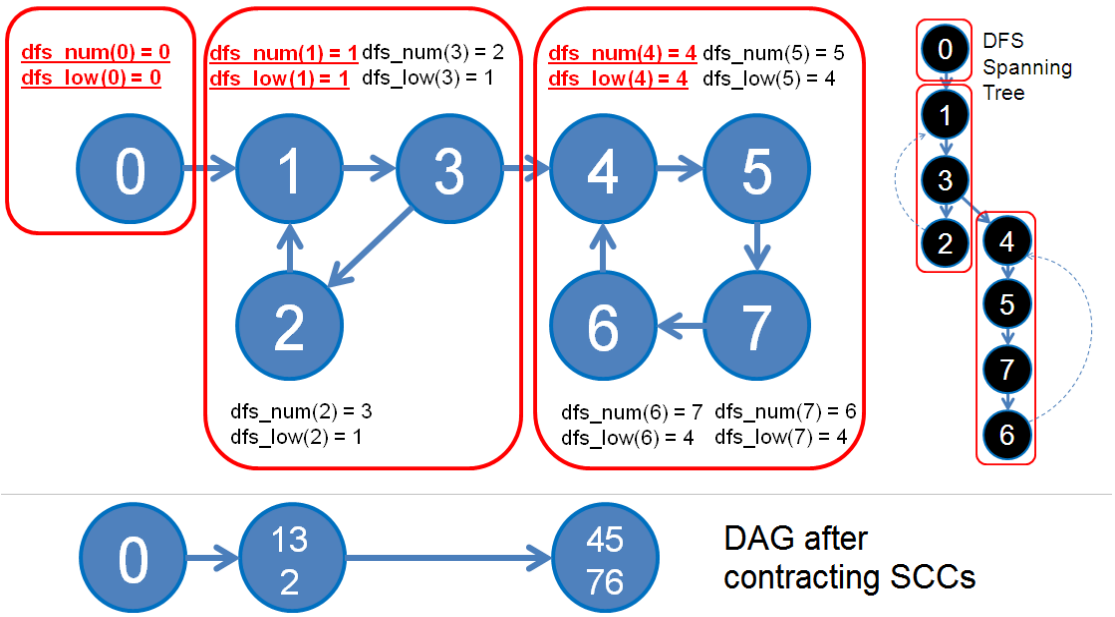


Figure 4.9: An Example of a Directed Graph and its SCCs

```
vi dfs_num, dfs_low, S, visited;                          // global variables

void tarjanSCC(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
  S.push_back(u);       // stores u in a vector based on order of visitation
  visited[u] = 1;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED)
      tarjanSCC(v.first);
    if (visited[v.first])                              // condition for update
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]); }

  if (dfs_low[u] == dfs_num[u]) {    // if this is a root (start) of an SCC
    printf("SCC %d:", ++numSCC);       // this part is done after recursion
    while (1) {
      int v = S.back(); S.pop_back(); visited[v] = 0;
      printf(" %d", v);
      if (u == v) break; }
    printf("\n");
} }

// inside int main()
  dfs_num.assign(V, UNVISITED); dfs_low.assign(V, 0); visited.assign(V, 0);
  dfsNumberCounter = numSCC = 0;
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == UNVISITED)
      tarjanSCC(i);
```

Source code: ch4_01_dfs.cpp/java; ch4_02_UVa469.cpp/java