Chapter 9

Rare Topics

Learning is a treasure that will follow its owner everywhere.
— **Chinese Proverb**

Overview and Motivation

In this chapter, we list down rare, 'exotic' topics in Computer Science that may (but usually will not) appear in a typical programming contest. These problems, data structures, and algorithms are mostly one-off unlike the more general topics that have been discussed in Chapters 1-8. Learning the topics in this chapter can be considered as being not 'cost-efficient' because after so much efforts on learning a certain topic, it likely *not* appear in the programming contest. However, we believe that these rare topics will appeal those who really love to expand their knowledge in Computer Science.

Skipping this chapter will not cause a major damage towards the preparation for an ICPC-style programming contest as the probability of appearance of any of these topics is low anyway[1]. However, when these rare topics do appear, contestants with a priori knowledge of those rare topics will have an advantage over others who do not have such knowledge. Some good contestants can probably derive the solution from basic concepts during contest time even if they have only seen the problem for the first time, but usually in a slower pace than those who already know the problem and especially its solution before.

For IOI, many of these rare topics are outside the IOI syllabus [20]. Thus, IOI contestants can choose to defer learning the material in this chapter until they enroll in University.

In this chapter, we keep the discussion for each topic as concise as possible, i.e. most discussions will be just around one or two page(s). Most discussions do not contain sample code as readers who have mastered the content of Chapter 1-8 should not have too much difficulty in translating the algorithms given in this chapter into a working code. We only have a few starred written exercises (without hints/solutions) in this chapter.

These rare topics are listed in alphabetical order in the table of contents at the front of this book. However, if you cannot find the name that we use, please use the indexing feature at the back of this book to check if the alternative names of these topics are listed.

[1]Some of these topics—with low probability—are used as interview questions for IT companies.

9.1 2-SAT Problem

Problem Description

You are given a conjunction of disjunctions ("and of ors") where each disjunction ("the or operation") has two arguments that may be variables or the negation of variables. The disjunctions of pairs are called as 'clauses' and the formula is known as the 2-CNF (Conjunctive Normal Form) formula. The 2-SAT problem is to find a truth (that is, true or false) assignment to these variables that makes the 2-CNF formula true, i.e. every clause has at least one term that is evaluated to true.

Example 1: $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ is satisfiable because we can assign $x_1 = true$ and $x_2 = false$ (alternative assignment is $x_1 = false$ and $x_2 = true$).

Example 2: $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ is not satisfiable. You can try all 8 possible combinations of boolean values of x_1, x_2, and x_3 to realize that none of them can make the 2-CNF formula satisfiable.

Solution(s)

Complete Search

Contestants who only have a vague knowledge of the Satisfiability problem may thought that this problem is an NP-Complete problem and therefore attempt a complete search solution. If the 2-CNF formula has n variables and m clauses, trying all 2^n possible assignments and checking each assignment in $O(m)$ has an overall time complexity of $O(2^n \times m)$. This is likely TLE.

The 2-SAT is a *special case* of Satisfiability problem and it admits a polynomial solution like the one shown below.

Reduction to Implication Graph and Finding SCC

First, we have to realize that a clause in a 2-CNF formula $(a \vee b) \equiv (\neg a \Rightarrow b) \equiv (\neg b \Rightarrow a)$. Thus, given a 2-CNF formula, we can build the corresponding 'implication graph'. Each variable has two vertices in the implication graph, the variable itself and the negation/inverse of that variable[2]. An edge connects one vertex to another if the corresponding variables are related by an implication in the corresponding 2-CNF formula. For the two 2-CNF example formulas above, we have the following implication graphs shown in Figure 9.1.

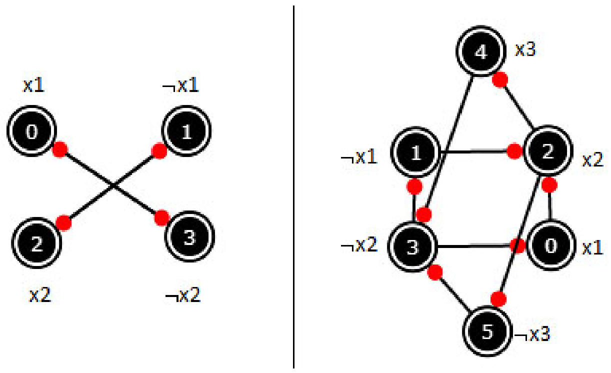

Figure 9.1: The Implication Graph of Example 1 (Left) and Example 2 (Right)

[2]Programming trick: We give a variable an index i and its negation with another index $i + 1$. This way, we can find one from the other by using bit manipulation $i \oplus 1$ where $\oplus$ is the 'exclusive or' operator.

As you can see in Figure 9.1, a 2-CNF formula with n variables (excluding the negation) and m clauses will have $V = \theta(2n) = \theta(n)$ vertices and $E = O(2m) = O(m)$ edges in the implication graph.

Now, a 2-CNF formula is satisfiable if and only if "there is no variable that belongs to the same Strongly Connected Component (SCC) as its negation".

In Figure 9.1—left, we see that there are two SCCs: {0,3} and {1,2}. As there is no variable that belongs to the same SCC as its negation, we conclude that the 2-CNF formula shown in Example 1 is satisfiable.

In Figure 9.1—right, we observe that all six vertices belong to a single SCC. Therefore, we have vertex 0 (that represents x_1) and vertex 1 (that represents[3] $\neg x_1$), vertex 2 (x_2) and vertex 3 ($\neg x_3$), vertex 4 (x_3) and vertex 5 ($\neg x_3$) in the same SCC. Therefore, we conclude that the 2-CNF formula shown in Example 2 is not satisfiable.

To find the SCCs of a directed graph, we can use either Tarjan's SCC algorithm as shown in Section 4.2.9 or Kosaraju's SCC algorithm as shown in Section 9.17.

Exercise 9.1.1*: To find the actual truth assignment, we need to do a bit more work than just checking if there is no variable that belongs to the same SCC as its negation. What are the extra steps required to actually find the truth assignment of a satisfiable 2-CNF formula?

Programming exercises related to 2-SAT problem:

1. ***UVa 10319 - Manhattan **** (the hard part in solving problems involving 2-SAT is in identifying that it is indeed a 2-SAT problem and then building the implication graph; for this problem, we set each street and each avenue as a variable where true means that it can only be used in a certain direction and false means that it can only be used in the other direction; a simple path will be in one of this form: (street $a \wedge$ avenue b) $\vee$ (avenue $c \wedge$ street d); this can be transformed into 2-CNF formula of $(a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d)$; build the implication graph and check if it is satisfiable using the SCC check as shown above; note that there exists a special case where the clause only has one literal, i.e. the simple path uses one street only or one avenue only.)

---

[3]Notice that using the programming trick shown above, we can easily test if vertex 1 and vertex 0 are a variable and its negation by testing if $1 = 0 \oplus 1$.

9.2 Art Gallery Problem

Problem Description

The 'Art Gallery' Problem is a family of related *visibility* problems in computational geometry. In this section, we discuss several variants. The common terms used in the variants discussed below are the simple (not necessarily convex) polygon P to describe the art gallery; a set of points S to describe the guards where each guard is represented by a point in P; a rule that a point $A \in S$ can guard another point $B \in P$ if and only if $A \in S, B \in P$, and line segment AB is contained in P; and a question on whether all points in polygon P are guarded by S. Many variants of this Art Gallery Problem are classified as NP-hard problems. In this book, we focus on the ones that admit polynomial solutions.

1. Variant 1: Determine the upper bound of the smallest size of set S.
2. Variant 2: Determine if $\exists$ a critical point C in polygon P and $\exists$ another point $D \in P$ such that if the guard is at position C, the guard cannot protect point D.
3. Variant 3: Determine if polygon P can be guarded with just one guard.
4. Variant 4: Determine the smallest size of set S if the guards can only be placed at the vertices of polygon P and only the vertices need to be guarded.

Note that there are many more variants and at least one book[4] has been written for it [49].

Solution(s)

1. The solution for variant 1 is a theoretical work of the Art Gallery theorem by Václav Chvátal. He states that $\lfloor n/3 \rfloor$ guards are always sufficient and sometimes necessary to guard a simple polygon with n vertices (proof omitted).
2. The solution for variant 2 involves testing if polygon P is concave (and thus has a critical point). We can use the negation of `isConvex` function shown in Section 7.3.4.
3. The solution for variant 3 can be hard if one has not seen the solution before. We can use the `cutPolygon` function discussed in Section 7.3.6. We cut polygon P with all lines formed by the edges in P in counter clockwise fashion and retain the left side at all times. If we still have a non empty polygon at the end, one guard can be placed in that non empty polygon which can protect the entire polygon P.
4. The solution for variant 4 involves the computation of Minimum Vertex Cover of the 'visibility graph' of polygon P. In general this is another NP-hard problem.

Programming exercises related to Art Gallery problem:

1. ***UVa 00588 - Video Surveillance **** (see variant 3 solution above)
2. **UVa 10078 - Art Gallery *** (see variant 2 solution above)
3. **UVa 10243 - Fire; Fire; Fire *** (variant 4: this problem can be reduced to the Minimum Vertex Cover problem *on Tree*; there is a polynomial DP solution for this variant; the solution has actually been discussed Section 4.7.1)
4. LA 2512 - Art Gallery (see variant 3 solution above plus area of polygon)
5. LA 3617 - How I Mathematician ... (variant 3)

---

[4]PDF version at `http://cs.smith.edu/~orourke/books/ArtGalleryTheorems/art.html`.

9.3 Bitonic Traveling Salesman Problem

Problem Description

The Bitonic Traveling Salesman Problem (TSP) can be described as follows: Given a list of coordinates of n vertices on 2D Euclidean space that are already sorted by x-coordinates (and if tie, by y-coordinates), find a tour that starts from the leftmost vertex, then goes strictly from left to right, and then upon reaching the rightmost vertex, the tour goes strictly from right to left back to the starting vertex. This tour behavior is called 'bitonic'.

The resulting tour may not be the shortest possible tour under the standard definition of TSP (see Section 3.5.2). Figure 9.2 shows a comparison of these two TSP variants. The TSP tour: 0-3-5-6-4-1-2-0 is not a Bitonic TSP tour because although the tour initially goes from left to right (0-3-5-6) and then goes back from right to left (6-4-1), it then makes another left to right (1-2) and then right to left (2-0) steps. The tour: 0-2-3-5-6-4-1-0 is a valid Bitonic TSP tour because we can decompose it into two paths: 0-2-3-5-6 that goes from left to right and 6-4-1-0 that goes back from right to left.

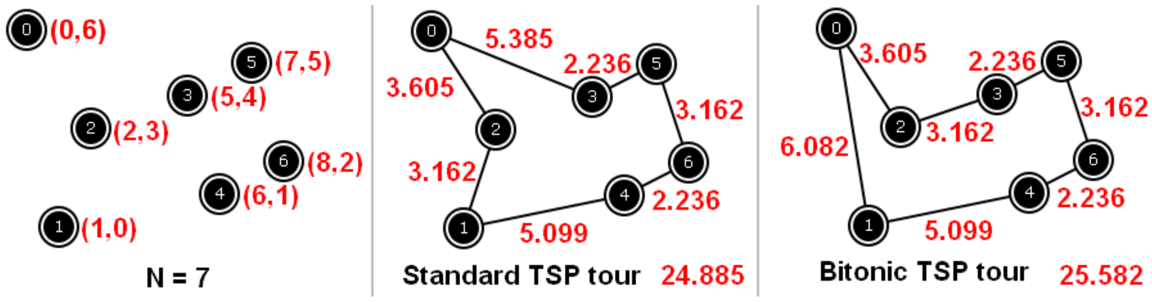

Figure 9.2: The Standard TSP versus Bitonic TSP

Solution(s)

Although a Bitonic TSP tour of a set of n vertices is usually longer than the standard TSP tour, this bitonic constraint allows us to compute a 'good enough tour' in $O(n^2)$ time using Dynamic Programming—as shown below—compared with the $O(2^n \times n^2)$ time for the standard TSP tour (see Section 3.5.2).

The main observation needed to derive the DP solution is the fact that we can (and have to) split the tour into two paths: Left-to-Right (LR) and Right-to-Left (RL) paths. Both paths include vertex 0 (the leftmost vertex) and vertex n-1 (the rightmost vertex). The LR path starts from vertex 0 and ends at vertex n-1. The RL path starts from vertex n-1 and ends at vertex 0.

Remember that all vertices have been sorted by x-coordinates (and if tie, by y-coordinates). We can then consider the vertices one by one. Both LR and RL paths start from vertex 0. Let v be the next vertex to be considered. For each vertex $v \in [1 \ldots n-2]$, we decide whether to add vertex v as the next point of the LR path (to extend the LR path further to the right) or as the previous point the returning RL path (the RL path now starts at v and goes back to vertex 0). For this, we need to keep track of two more parameters: $p1$ and $p2$. Let $p1/p2$ be the current *ending/starting* vertex of the LR/RL path, respectively.

The base case is when vertex $v = n-1$ where we just need to connect the two LR and RL paths with vertex $n-1$.

With these observations in mind, we can write a simple DP solution is like this:

```
double dp1(int v, int p1, int p2) {             // called with dp1(1, 0, 0)
  if (v == n-1)
    return d[p1][v] + d[v][p2];
  if (memo3d[v][p1][p2] > -0.5)
    return memo3d[v][p1][p2];
  return memo3d[v][p1][p2] = min(
    d[p1][v] + dp1(v+1, v, p2),      // extend LR path: p1->v, RL stays: p2
    d[v][p2] + dp1(v+1, p1, v));     // LR stays: p1, extend RL path: p2<-v
}
```

However, the time complexity of `dp1` with three parameters: `(v, p1, p2)` is $O(n^3)$. This is not efficient, as parameter v can be dropped and recovered from $1+max(p1, p2)$ (see this DP optimization technique of dropping one parameter and recovering it from other parameters as shown in Section 8.3.6). The improved DP solution is shown below and runs in $O(n^2)$.

```
double dp2(int p1, int p2) {                         // called with dp2(0, 0)
  int v = 1 + max(p1, p2);   // this single line speeds up Bitonic TSP tour
  if (v == n-1)
    return d[p1][v] + d[v][p2];
  if (memo2d[p1][p2] > -0.5)
    return memo2d[p1][p2];
  return memo2d[p1][p2] = min(
    d[p1][v] + dp2(v, p2),           // extend LR path: p1->v, RL stays: p2
    d[v][p2] + dp2(p1, v));          // LR stays: p1, extend RL path: p2<-v
}
```

Programming exercises related to Bitonic TSP:

1. **UVa 01096 - The Islands *** (LA 4791, World Finals Harbin10, Bitonic TSP variant; print the actual path)
2. ***UVa 01347 - Tour**** (LA 3305, Southeastern Europe 2005; this is the pure version of Bitonic TSP problem, you may want to start from here)

9.4 Bracket Matching

Problem Description

Programmers are very familiar with various form of braces: '()', '{}', '[]', etc as they use braces quite often in their code especially when dealing with if statements and loops. Braces can be nested, e.g. '(())', '{{}}', '[[]]', etc. A well-formed code must have a matched set of braces. The Bracket Matching problem usually involves a question on whether a given set of braces is properly nested. For example, '(())', '({})', '(){}[]' are correctly matched braces whereas '(()', '(}', ')(' are *not* correct.

Solution(s)

We read the brackets one by one from left to right. Every time we encounter a close bracket, we need to match it with the latest open bracket (of the same type). This matched pair is then removed from consideration and the process is continued. This requires a 'Last In First Out' data structure: Stack (see Section 2.2).

We start from an empty stack. Whenever we encounter an open bracket, we push it into the stack. Whenever we encounter a close bracket, we check if it is of the same type with the top of the stack. This is because the top of the stack is the one that has to be matched with the current close bracket. Once we have a match, we pop the topmost bracket from the stack to remove it from future consideration. Only if we manage to reach the last bracket and find that the stack is back to empty, then we know that the brackets are properly nested.

As we examine each of the n braces only once and all stack operations are $O(1)$, this algorithm clearly runs in $O(n)$.

Variant(s)

The number of ways n pairs of parentheses can be correctly matched can be found with Catalan formula (see Section 5.4.3). The optimal way to multiply matrices (i.e. the Matrix Chain Multiplication problem) also involves bracketing. This variant can be solved with Dynamic Programming (see Section 9.20).

Programming exercises related to Bracket Matching:

1. ***UVa 00551 - Nesting a Bunch of ... **** (bracket matching, `stack`, classic)
2. **UVa 00673 - Parentheses Balance *** (similar to UVa 551, classic)
3. ***UVa 11111 - Generalized Matrioshkas **** (bracket matching with some twists)

9.5 Chinese Postman Problem

Problem Description

The Chinese Postman[5]/Route Inspection Problem is the problem of finding the (length of the) shortest tour/circuit that visits every edge of a (connected) undirected weighted graph. If the graph is Eulerian (see Section 4.7.3), then the sum of edge weights along the Euler tour that covers all the edges in the Eulerian graph is the optimal solution for this problem. This is the easy case. But when the graph is non Eulerian, e.g. see the graph in Figure 9.3—left, then this Chinese Postman Problem is harder.

Solution(s)

The important insight to solve this problem is to realize that a non Eulerian graph G must have an *even number* of vertices of odd degree (the Handshaking lemma found by Euler himself). Let's name the subset of vertices of G that have odd degree as T. Now, create a complete graph K_n where n is the size of T. T form the vertices of K_n. An edge (i, j) in K_n has weight which is the *shortest path weight* of a path from i to j, e.g. in Figure 9.3 (middle), edge `2-5` in K_4 has weight $2 + 1 = 3$ from path `2-4-5` and edge `3-4` in K_4 has weight $3 + 1 = 4$ from path `3-5-4`.

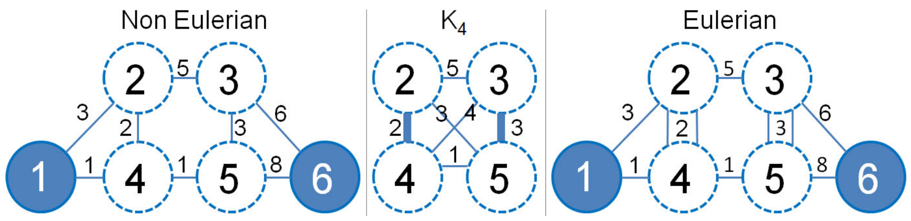

Figure 9.3: An Example of Chinese Postman Problem

Now, if we *double* the edges selected by the *minimum weight perfect matching* on this complete graph K_n, we will convert the non Eulerian graph G to another graph G' which is Eulerian. This is because by doubling those edges, we actually add an edge between a pair of vertices with odd degree (thus making them have even degree afterwards). The *minimum weight* perfect matching ensures that this transformation is done in the *least cost way*. The solution for the minimum weight perfect matching on the K_4 shown in Figure 9.3 (middle) is to take edge `2-4` (with weight 2) and edge `3-5` (with weight 3).

After doubling edge `2-4` and edge `3-5`, we are now back to the easy case of the Chinese Postman Problem. In Figure 9.3 (right), we have an Eulerian graph. The tour is simple in this Eulerian graph. One such tour is: `1->2->4->5->3->6->5->3->2->4->1` with a total weight of `34` (the sum of all edge weight in the modified Eulerian graph G', which is the sum of all edge weight in G plus the cost of the minimum weight perfect matching in K_n).

The hardest part of solving the Chinese Postman Problem is therefore in finding the minimum weight perfect matching on K_n, which is *not* a bipartite graph (a complete graph). If n is small, this can be solved with DP with bitmask technique shown in Section 8.3.1.

Programming exercises related to Chinese Postman Problem:

1. **UVa 10296 - Jogging Trails *** (see the discussion above)

[5]The name is because it is first studied by the Chinese mathematician Mei-Ku Kuan in 1962.

9.6 Closest Pair Problem

Problem Description

Given a set S of n points on a 2D plane, find two points with the closest Euclidean distance.

Solution(s)

Complete Search

A naïve solution computes the distances between all pairs of points and reports the minimum one. However, this requires $O(n^2)$ time.

Divide and Conquer

We can use the following Divide and Conquer strategy to achieve $O(n \log n)$ time.
We perform the following three steps:

1. Divide: We sort the points in set S by their x-coordinates (if tie, by their y-coordinates). Then, we divide set S into two sets of points S_1 and S_2 with a vertical line $x = d$ such that $|S_1| = |S_2|$ or $|S_1| = |S_2| + 1$, i.e. the number of points in each set is balanced.
2. Conquer: If we only have one point in S, we return ∞.
   If we only have two points in S, we return their Euclidean distance.
3. Combine: Let d_1 and d_2 be the smallest distance in S_1 and S_2, respectively. Let d_3 be the smallest distance between all pairs of points (p_1, p_2) where p_1 is a point in S_1 and p_2 is a point in S_2. Then, the smallest distance is $min(d_1, d_2, d_3)$, i.e. the answer may be in the smaller set of points S_1 or in S_2 or one point in S_1 and the other point in S_2, crossing through line $x = d$.

The combine step, if done naïvely, will still run in $O(n^2)$. But this can be optimized. Let $d' = min(d_1, d_2)$. For each point in the left of the dividing line $x = d$, a closer point in the right of the dividing line can only lie within a rectangle with width d' and height $2 \times d'$. It can be proven (proof omitted) that there can be only at most 6 such points in this rectangle. This means that the combine step only require $O(6n)$ operation and the overall time complexity of this divide and conquer solution is $T(n) = 2 \times T(n/2) + O(n)$ which is $O(n \log n)$.

Exercise 9.6.1*: There is a simpler solution other than the classic Divide & Conquer solution shown above. It uses sweep line algorithm. We 'sweep' the points in S from left to right. Suppose the current best answer is d and we are now examining point i. The potential new closest point from i, if any, must have y-coordinate to be within d units of point i. We check all these candidates and update d accordingly (which will be progressively smaller). Implement this solution and analyze its time complexity!

Programming exercises related to Closest Pair problem:

1. **UVa 10245 - The Closest Pair Problem *** (classic, as discussed above)
2. **UVa 11378 - Bey Battle *** (also a closest pair problem)

9.7 Dinic's Algorithm

In Section 4.6, we have seen the potentially unpredictable $O(|f^*|E)$ Ford Fulkerson's method and the preferred $O(VE^2)$ Edmonds Karp's algorithm (finding augmenting paths with BFS) for solving the Max Flow problem. As of year 2013, most (if not all) Max Flow problems in this book are solvable using Edmonds Karp's.

There are several other Max Flow algorithms that have theoretically better performance than Edmonds Karp's. One of them is Dinic's algorithm which runs in $O(V^2E)$. Since a typical flow graph usually has $V < E$ and $E << V^2$, Dinic's worst case time complexity is theoretically better than Edmonds Karp's. Although the authors of this book have not encountered a case where Edmonds Karp's received TLE verdict and Dinic's received AC verdict on the *same* flow graph, it *may be* beneficial to use Dinic's algorithm in programming contests just to be on the safer side.

Dinic's algorithm uses a similar idea as Edmonds Karp's as it also finds augmenting paths iteratively. However, Dinic's algorithm uses the concept of 'blocking flows' to find the augmenting paths. Understanding this concept is the key to extend the easier-to-understand Edmonds Karp's algorithm into Dinic's algorithm.

Let's define `dist[v]` to be the length of the shortest path from the source vertex s to v in the residual graph. Then the level graph of the residual graph is L where edge (u, v) in the residual graph is included in the level graph L iff `dist[v] = dist[u] + 1`. Then, a 'blocking flow' is an $s - t$ flow f such that after sending through flow f from s to t, the level graph L contains no $s - t$ augmenting path anymore.

It has been proven (see [11]) that the number of edges in each blocking flow increases by at least one per iteration. There are at most $V - 1$ blocking flows in the algorithm because there can only be at most $V - 1$ edges along the 'longest' simple path from s to t. The level graph can be constructed by a BFS in $O(E)$ time and a blocking flow in each level graph can be found in $O(VE)$ time. Hence, the worst case time complexity of Dinic's algorithm is $O(V \times (E + VE)) = O(V^2E)$.

Dinic's implementation can be made similar with Edmonds Karp's implementation shown in Section 4.6. In Edmonds Karp's, we run a BFS—which already generates for us the level graph L—but we just use it to find *one* augmenting path by calling `augment(t, INF)` function. In Dinic's algorithm, we need to use the information produced by BFS in a slightly different manner. The key change is this: Instead of finding a blocking flow by running DFS on the level graph L, we can simulate the process by running the `augment` procedure from *each vertex* v that is directly connected to the sink vertex t in the level graph, i.e. edge (v, t) exists in level graph L and we call `augment(v, INF)`. This will find many (but not always all) the required augmenting *paths* that make up the blocking flow of level graph L for us.

Exercise 9.7.1*: Implement the *variant* of Dinic's algorithm *s*tarting from the Edmonds Karp's code given in Section 4.6 using the suggested key change above! Also use the modified Adjacency List as asked in **Exercise 4.6.3.3***. Now, (re)solve various programming exercises listed in Section 4.6! Do you notice any runtime improvements?

Exercise 9.7.2*: Using a data structure called dynamic trees, the running time of finding a blocking flow can be reduced from $O(VE)$ down to $O(E \log V)$ and therefore the overall worst-case time complexity of Dinic's algorithm becomes $O(VE \log V)$. Study and implement this Dinic's implementation variant!

Exercise 9.7.3*: What happens if we use Dinic's algorithm on flow graph that models MCBM problem as discussed in Section 4.7.4? (hint: See Section 9.12).

9.8 Formulas or Theorems

We have encountered some rarely used formulas or theorems in some programming contest problems before. Knowing them will give you an *unfair advantage* over other contestants if one of these rare formulas or theorems is used in the programming contest that you join.

1. Cayley's Formula: There are n^{n-2} spanning trees of a complete graph with n labeled vertices. Example: UVa 10843 - Anne's game.
2. Derangement: A permutation of the elements of a set such that none of the elements appear in their original position. The number of derangements $der(n)$ can be computed as follow: $der(n) = (n-1) \times (der(n-1) + der(n-2))$ where $der(0) = 1$ and $der(1) = 0$. A basic problem involving derangement is UVa 12024 - Hats (see Section 5.6).
3. Erdős Gallai's Theorem gives a necessary and sufficient condition for a finite sequence of natural numbers to be the *degree sequence* of a simple graph. A sequence of non-negative integers $d_1 \geq d_2 \geq \ldots \geq d_n$ can be the degree sequence of a simple graph on n vertices iff $\sum_{i=1}^{n} d_i$ is even and $\sum_{i=1}^{k} d_i \leq k \times (k-1) + \sum_{i=k+1}^{n} min(d_i, k)$ holds for $1 \leq k \leq n$. Example: UVa 10720 - Graph Construction.
4. Euler's Formula for Planar Graph[6]: $V - E + F = 2$, where F is the number of faces[7] of the Planar Graph. Example: UVa 10178 - Count the Faces.
5. Moser's Circle: Determine the number of pieces into which a circle is divided if n points on its circumference are joined by chords with no three internally concurrent. Solution: $g(n) = ^n C_4 + ^n C_2 + 1$. Example: UVa 10213 - How Many Pieces of Land?
6. Pick's Theorem[8]: Let I be the number of integer points in the polygon, A be the area of the polygon, and b be the number of integer points on the boundary, then $A = i + \frac{b}{2} - 1$. Example: UVa 10088 - Trees on My Island.
7. The number of spanning tree of a complete bipartite graph $K_{n,m}$ is $m^{n-1} \times n^{m-1}$. Example: UVa 11719 - Gridlands Airport.

Programming exercises related to *rarely used* Formulas or Theorems:

1. UVa 10088 - Trees on My Island (Pick's Theorem)
2. UVa 10178 - Count the Faces (Euler's Formula, a bit of union find)
3. **UVa 10213 - How Many Pieces ... *** (Moser's circle; the formula is hard to derive; $g(n) = _n C_4 + _n C_2 + 1$)
4. **UVa 10720 - Graph Construction *** (Erdős-Gallai's Theorem)
5. UVa 10843 - Anne's game (Cayley's Formula to count the number of spanning trees of a graph with n vertices is n^{n-2}; use Java BigInteger)
6. UVa 11414 - Dreams (similar to UVa 10720; Erdős-Gallai's Theorem)
7. ***UVa 11719 - Gridlands Airports **** (count the number of spanning tree in a complete bipartite graph; use Java BigInteger)

---

[6]Graph that can be drawn on 2D Euclidean space so that no two edges in the graph cross each other.

[7]When a Planar Graph is drawn without any crossing, any cycle that surrounds a region without any edges reaching from the cycle into the region forms a face.

[8]Found by Georg Alexander Pick.

9.9 Gaussian Elimination Algorithm

Problem Description

A **linear equation** is defined as an equation where the order of the unknowns (variables) is **linear** (a constant or a product of a constant plus the first power of an unknown). For example, equation `X + Y = 2` is linear but equation $X^2 = 4$ is not linear.

A **system of linear equations** is defined as a collection of n unknowns (variables) in (usually) n linear equations, e.g. `X + Y = 2` and `2X + 5Y = 6`, where the solution is $X = 1\frac{1}{3}$, $Y = \frac{2}{3}$. Notice the difference to the **linear diophantine equation** (see Section 5.5.9) as the solution for a **system of linear equations** can be non-integers!

In rare occasions, we may find such system of linear equations in a programming contest problem. Knowing the solution, especially its implementation, may come handy.

Solution(s)

To compute the solution of a **system of linear equations**, one can use techniques like the **Gaussian Elimination** algorithm. This algorithm is more commonly found in Engineering textbooks under the topic of 'Numerical Methods'. Some Computer Science textbooks do have some discussions about this algorithm, e.g. [8]. Here, we show this relatively simple $O(n^3)$ algorithm using a C++ function below.

```
#define MAX_N 100                               // adjust this value as needed
struct AugmentedMatrix { double mat[MAX_N][MAX_N + 1]; };
struct ColumnVector { double vec[MAX_N]; };

ColumnVector GaussianElimination(int N, AugmentedMatrix Aug) {    // O(N^3)
  // input: N, Augmented Matrix Aug, output: Column vector X, the answer
  int i, j, k, l; double t; ColumnVector X;

  for (j = 0; j < N - 1; j++) {              // the forward elimination phase
    l = j;
    for (i = j + 1; i < N; i++)          // which row has largest column value
      if (fabs(Aug.mat[i][j]) > fabs(Aug.mat[l][j]))
        l = i;                                         // remember this row l
    // swap this pivot row, reason: to minimize floating point error
    for (k = j; k <= N; k++)              // t is a temporary double variable
      t = Aug.mat[j][k], Aug.mat[j][k] = Aug.mat[l][k], Aug.mat[l][k] = t;
    for (i = j + 1; i < N; i++)        // the actual forward elimination phase
      for (k = N; k >= j; k--)
        Aug.mat[i][k] -= Aug.mat[j][k] * Aug.mat[i][j] / Aug.mat[j][j];
  }

  for (j = N - 1; j >= 0; j--) {                // the back substitution phase
    for (t = 0.0, k = j + 1; k < N; k++) t += Aug.mat[j][k] * X.vec[k];
    X.vec[j] = (Aug.mat[j][N] - t) / Aug.mat[j][j];   // the answer is here
  }
  return X;
}
```

Source code: `GaussianElimination.cpp/java`

Sample Execution

In this subsection, we show the step-by-step working of 'Gaussian Elimination' algorithm using the following example. Suppose we are given this system of linear equations:

```
  X = 9 - Y - 2Z
2X + 4Y = 1 + 3Z
  3X - 5Z = -6Y
```

First, we need to transform the system of linear equations into the *basic form*, i.e. we reorder the unknowns (variables) in sorted order on the Left Hand Side. We now have:

```
1X + 1Y + 2Z = 9
2X + 4Y - 3Z = 1
3X + 6Y - 5Z = 0
```

Then, we re-write these linear equations as matrix multiplication: $A \times x = b$. This trick is also used in Section 9.21. We now have:

$$\begin{bmatrix} 1 & 1 & 2 \\ 2 & 4 & -3 \\ 3 & 6 & -5 \end{bmatrix} \times \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 9 \\ 1 \\ 0 \end{bmatrix}$$

Later, we will work with both matrix A (of size $N \times N$) and column vector b (of size $N \times 1$). So, we combine them into an $N \times (N + 1)$ 'augmented matrix' (the last column that has three arrows is a comment to aid the explanation):

$$\left[\begin{array}{ccc|c|l} 1 & 1 & 2 & 9 & \rightarrow \text{1X + 1Y + 2Z = 9} \\ 2 & 4 & -3 & 1 & \rightarrow \text{2X + 4Y - 3Z = 1} \\ 3 & 6 & -5 & 0 & \rightarrow \text{3X + 6Y - 5Z = 0} \end{array}\right]$$

Then, we pass this augmented matrix into Gaussian Elimination function above. The first phase is the forward elimination phase. We pick the largest absolute value in column $j = 0$ from row $i = 0$ onwards, then swap that row with row $i = 0$. This (extra) step is just to minimize floating point error. For this example, after swapping row 0 with row 2, we have:

$$\left[\begin{array}{ccc|c|l} \underline{3} & 6 & -5 & 0 & \rightarrow \underline{\text{3X + 6Y - 5Z = 0}} \\ 2 & 4 & -3 & 1 & \rightarrow \text{2X + 4Y - 3Z = 1} \\ \underline{1} & 1 & 2 & 9 & \rightarrow \underline{\text{1X + 1Y + 2Z = 9}} \end{array}\right]$$

The main action done by Gaussian Elimination algorithm in this forward elimination phase is to eliminate variable X (the first variable) from row $i + 1$ onwards. In this example, we eliminate X from row 1 and row 2. Concentrate on the comment "the actual forward elimination phase" inside the Gaussian Elimination code above. We now have:

$$\left[\begin{array}{ccc|c|l} 3 & 6 & -5 & 0 & \rightarrow \text{3X + 6Y - 5Z = 0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0.33} & \mathbf{1} & \rightarrow \underline{\text{0X + 0Y + 0.33Z = 1}} \\ \mathbf{0} & \mathbf{-1} & \mathbf{3.67} & \mathbf{9} & \rightarrow \underline{\text{0X - 1Y + 3.67Z = 9}} \end{array}\right]$$

Then, we continue eliminating the next variable (now variable Y). We pick the largest absolute value in column $j = 1$ from $row = 1$ onwards, then swap that row with row $i = 1$. For this example, after swapping row 1 with row 2, we have the following augmented matrix and it happens that variable Y is already eliminated from row 2:

$$\left[\begin{array}{c|ccc|c|l} \text{row 0} & 3 & 6 & -5 & 0 & \rightarrow \text{3X + 6Y - 5Z = 0} \\ \text{row 1} & 0 & \underline{-1} & 3.67 & 9 & \rightarrow \underline{\text{0X - 1Y + 3.67Z = 9}} \\ \text{row 2} & 0 & \underline{0} & 0.33 & 1 & \rightarrow \underline{\text{0X + 0Y + 0.33Z = 1}} \end{array}\right]$$

Once we have the lower triangular matrix of the augmented matrix all zeroes, we can start the second phase: The back substitution phase. Concentrate on the last few lines in the Gaussian Elimination code above. Notice that after eliminating variable X and Y, there is only variable Z in row 2. We are now sure that $Z = 1/0.33 = 3$.

$$\left[\begin{array}{c|ccc|c|l} \text{row 2} & 0 & 0 & 0.33 & 1 & \rightarrow \text{0X + 0Y + 0.33Z = 1} \rightarrow \text{Z = 1/0.33 = 3} \end{array}\right]$$

Once we have $Z = 3$, we can process row 1.
We get $Y = (9 - 3.67 * 3)/ - 1 = 2$.

$$\left[\begin{array}{c|ccc|c|l} \text{row 1} & 0 & -1 & 3.67 & 9 & \rightarrow \text{0X - 1Y + 3.67Z = 9} \rightarrow \text{Y = (9 - 3.67 * 3) / -1 = 2} \end{array}\right]$$

Finally, once we have $Z = 3$ and $Y = 2$, we can process row 0.
We get $X = (0 - 6 * 2 + 5 * 3)/3 = 1$, done!

$$\left[\begin{array}{c|ccc|c|l} \text{row 0} & 3 & 6 & -5 & 0 & \rightarrow \text{3X + 6Y - 5Z = 0} \rightarrow \text{X = (0 - 6 * 2 + 5 * 3) / 3 = 1} \end{array}\right]$$

Therefore, the solution for the given system of linear equations is $X = 1$, $Y = 2$, and $Z = 3$.

Programming Exercises related to Gaussian Elimination:

1. ***UVa 11319 - Stupid Sequence? **** (solve the system of the first 7 linear equations; then use all 1500 equations for 'smart sequence' checks)

9.10 Graph Matching

Problem Description

Graph matching: Select a subset of edges M of a graph $G(V, E)$ so that no two edges share the same vertex. Most of the time, we are interested in the *Maximum Cardinality* matching, i.e. we want to know the *maximum number of edges* that we can match in graph G. Another common request is the *Perfect* matching where we have both Maximum Cardinality matching and no vertex is left unmatched. Note that if V is odd, it is impossible to have a Perfect matching. Perfect matching can be solved by simply looking for Maximum Cardinality and then checking if all vertices are matched.

There are two important attributes of graph matching problems in programming contests that can (significantly) alter the level of difficulty: Whether the input graph is bipartite (harder otherwise) and whether the input graph is unweighted (harder otherwise). This two characteristics create four variants[9] as outlined below (also see Figure 9.4).

1. Unweighted Maximum Cardinality Bipartite Matching (Unweighted MCBM)
   This is the easiest and the most common variant.

2. Weighted Maximum Cardinality Bipartite Matching (Weighted MCBM)
   This is a similar problem as above, but now the edges in G have weights.
   We usually want the MCBM with the minimum total weight.

3. Unweighted Maximum Cardinality Matching (Unweighted MCM)
   The graph is not guaranteed to be bipartite.

4. Weighted Maximum Cardinality Matching (Weighted MCM)
   This is the hardest variant.

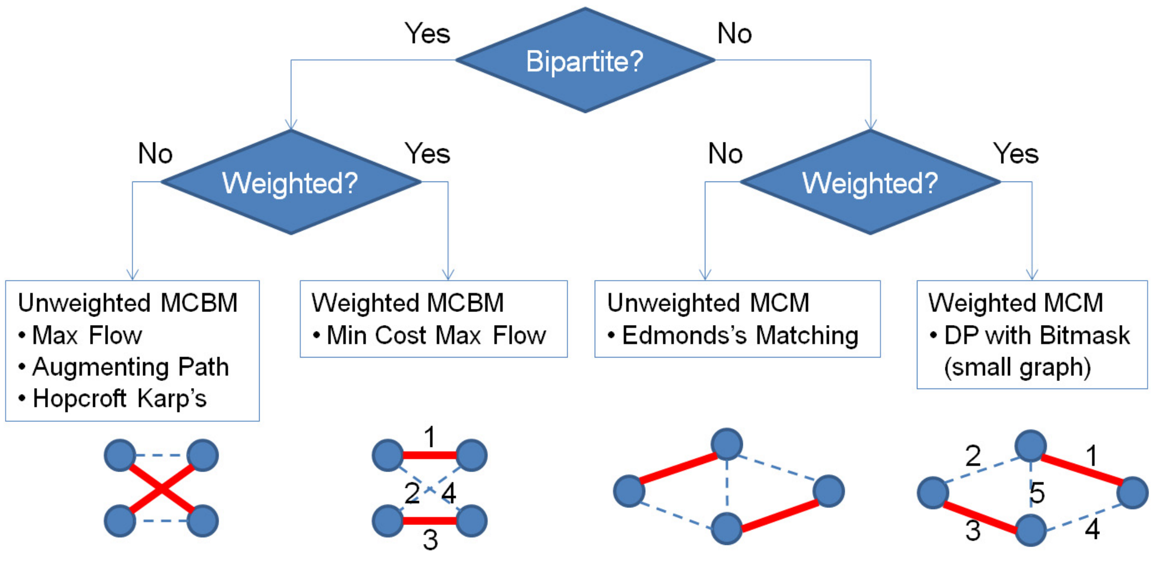

Figure 9.4: The Four Common Variants of Graph Matching in Programming Contests

[9]There are other Graph Matching variants outside these four, e.g. the Stable Marriage problem. However, we will concentrate on these four variants in this section.

Solution(s)

Solutions for Unweighted MCBM

This variant is the easiest and several solutions have been discussed earlier in Section 4.6 (Network Flow) and Section 4.7.4 (Bipartite Graph). The list below summarizes three possible solutions for the Unweighted MCBM problems:

1. Reducing the Unweighted MCBM problem into a Max Flow Problem.
   See Section 4.6 and 4.7.4 for details.
   The time complexity depends on the chosen Max Flow algorithm.
2. $O(V^2 + VE)$ Augmenting Path Algorithm for Unweighted MCBM.
   See Section 4.7.4 for details.
   This is good enough for various contest problems involving Unweighted MCBM.
3. $O(\sqrt{V}E)$ Hopcroft Karp's Algorithm for Unweighted MCBM
   See Section 9.12 for details.

Solutions for Weighted MCBM

When the edges in the bipartite graph are weighted, not all possible MCBMs are optimal. We need to pick one (not necessarily unique) MCBM that has the minimum overall total weight. One possible solution[10] is to reduce the Weighted MCBM problem into a Min Cost Max Flow (MCMF) problem (see Section 9.23).

For example, in Figure 9.5, we show one test case of UVa 10746 - Crime Wave - The Sequel. This is an MCBM problem on a complete bipartite graph $K_{n,m}$, but each edge has associated cost. We add edges from source s to vertices of the left set with capacity 1 and cost 0. We also add edges from vertices of the right set to the sink t also with capacity 1 and cost 0. The directed edges from the left set to the right set has capacity 1 and cost according to the problem description. After having this weighted flow graph, we can run the MCMF algorithm as shown in Section 9.23 to get the required answer: Flow 1 = $0 \rightarrow 2 \rightarrow 4 \rightarrow 8$ with cost 5, Flow 2 = $0 \rightarrow 1 \rightarrow 4 \rightarrow 2$ (cancel flow 2-4) $\rightarrow 6 \rightarrow 8$ with cost 15, and Flow 3 = $0 \rightarrow 3 \rightarrow 5 \rightarrow 8$ with cost 20. The minimum total cost is 5 + 15 + 20 = 40.

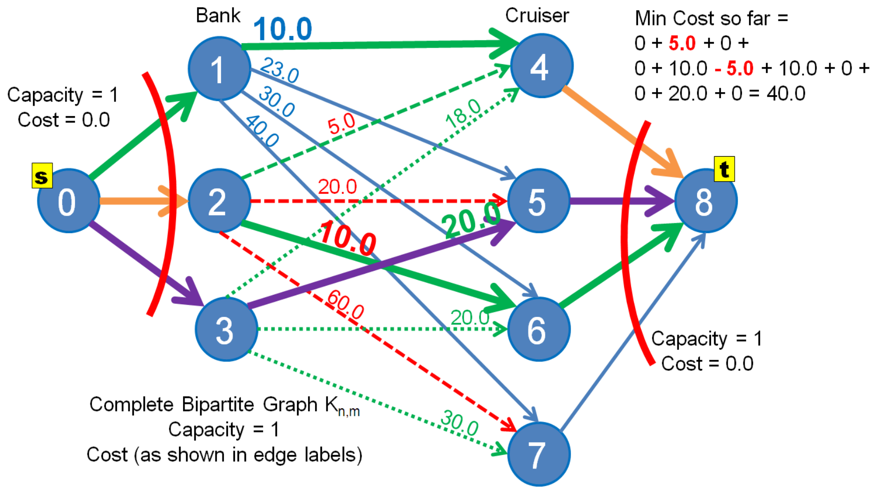

Figure 9.5: A Sample Test Case of UVa 10746: 3 Matchings with Min Cost = 40

[10]Another possible solution if we want to get *perfect* bipartite matching with minimum cost is the Hungarian (Kuhn-Munkres's) algorithm.

Solutions for Unweighted MCM

While the graph matching problem is easy on bipartite graphs, it is hard on general graphs. In the past, computer scientists thought that this variant was another NP-Complete problem until Jack Edmonds published a polynomial algorithm for solving this problem in his 1965 paper titled "Paths, trees, and flowers" [13].

The main issue is that on general graph, we may encounter odd-length augmenting cycles. Edmonds calls such a cycle a 'blossom'. The key idea of Edmonds Matching algorithm is to repeatedly shrink these blossoms (potentially in recursive fashion) so that finding augmenting paths returns back to the easy case as in bipartite graph. Then, Edmonds matching algorithm readjust the matchings when these blossoms are re-expanded (lifted).

The implementation of Edmonds Matching algorithm is not straightforward. Therefore, to make this graph matching variant more manageable, many problem authors limit the size of their unweighted general graphs to be small enough, i.e. $V \le 18$ so that an $O(V \times 2^V)$ DP with bitmask algorithm can be used to solve it (see **Exercise 8.3.1.1**).

Solution for Weighted MCM

This is potentially the hardest variant. The given graph is a general graph and the edges have associated weights. In typical programming contest environment, the most likely solution is the DP with bitmask (see Section 8.3.1) as the problem authors usually set the problem on *a small general graph* only.

Visualization of Graph Matching

To help readers in understanding these graph matching variants and their solutions, we have built the following visualization tool:

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

In this visualization tool, you can draw your own graph and the system will present the correct graph matching algorithm(s) based on the two characteristics: Whether the input graph is bipartite and/or weighted. Note that the visualization of Edmonds's Matching inside our tool is probably one of the first in the world.

Exercise 9.10.1*: Implement Kuhn Munkres's algorithm! (see the original paper [39, 45]).

Exercise 9.10.2*: Implement Edmonds's Matching algorithm! (see the original paper [13]).

Programming exercises related to Graph Matching:

- See some assignment problems (bipartite matching with capacity) in Section 4.6
- See some Unweighted MCBM problems and variants in Section 4.7.4
- See some Weighted MCBM problems in Section 9.23
- Unweighted MCM
  1. ***UVa 11439 - Maximizing the ICPC **** (binary search the answer to get the minimum weight; use this weight to reconstruct the graph; use Edmonds's Matching algorithm to test if we can get perfect matching on general graph)
- See (Un)weighted MCM problems on *small general graph* in Section 8.3 (DP)

9.11 Great-Circle Distance

Problem Description

Sphere is a perfectly round geometrical object in 3D space.

The **Great-Circle Distance** between any two points A and B on sphere is the shortest distance along a path on the **surface of the sphere**. This path is an *arc* of the **Great-Circle** of that sphere that pass through the two points A and B. We can imagine Great-Circle as the resulting circle that appears if we cut the sphere with a plane so that we have two *equal* hemispheres (see Figure 9.6—left and middle).

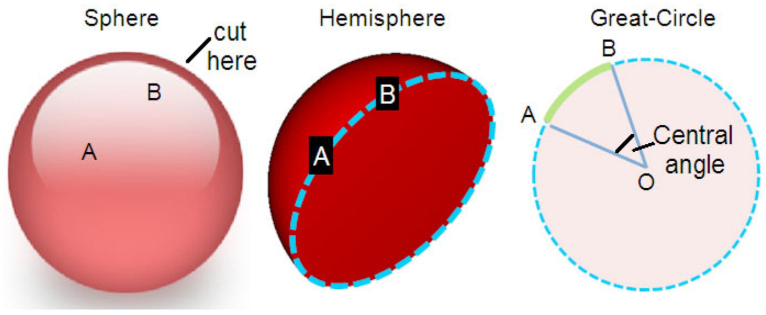

Figure 9.6: L: Sphere, M: Hemisphere and Great-Circle, R: gcDistance (Arc A-B)

Solution(s)

To find the Great-Circle Distance, we have to find the central angle AOB (see Figure 9.6—right) of the Great-Circle where O is the center of the Great-Circle (which is also the center of the sphere). Given the radius of the sphere/Great-Circle, we can then determine the length of arc A-B, which is the required Great-Circle distance.

Although quite rare nowadays, some contest problems involving 'Earth', 'Airlines', etc use this distance measurement. Usually, the two points on the surface of a sphere are given as the Earth coordinates, i.e. the (latitude, longitude) pair. The following library code will help us to obtain the Great-Circle distance given two points on the sphere and the radius of the sphere. We omit the derivation as it is not important for competitive programming.

```
double gcDistance(double pLat, double pLong,
                  double qLat, double qLong, double radius) {
  pLat *= PI / 180; pLong *= PI / 180;     // convert degree to radian
  qLat *= PI / 180; qLong *= PI / 180;
  return radius * acos(cos(pLat)*cos(pLong)*cos(qLat)*cos(qLong) +
                       cos(pLat)*sin(pLong)*cos(qLat)*sin(qLong) +
                       sin(pLat)*sin(qLat)); }
```

Source code: `UVa11817.cpp/java`

Programming exercises related to Great-Circle Distance:

1. UVa 00535 - Globetrotter (`gcDistance`)
2. UVa 10316 - Airline Hub (`gcDistance`)
3. UVa 10897 - Travelling Distance (`gcDistance`)
4. UVa 11817 - Tunnelling The Earth (`gcDistance`; 3D Euclidean distance)

9.12 Hopcroft Karp's Algorithm

Hopcroft Karp's algorithm [28] is another algorithm to solve the Unweighted Maximum Cardinality Bipartite Matching (MCBM) problem on top of the Max Flow based solution (which is longer to code) and the Augmenting Path algorithm (which is the preferred method) as discussed in Section 4.7.4.

In our opinion, the main reason for using the longer-to-code Hopcroft Karp's algorithm instead of the simpler-and-shorter-to-code Augmenting Path algorithm to solve the Unweighted MCBM is its runtime speed. Hopcroft Karp's algorithm runs in $O(\sqrt{V}E)$ which is (much) faster than the $O(VE)$ Augmenting Path algorithm on medium-sized ($V \approx 500$) bipartite (and dense) graphs.

An extreme example is a Complete Bipartite Graph $K_{n,m}$ with $V = n+m$ and $E = n \times m$. On such bipartite graph, the Augmenting Path algorithm has worst case time complexity of $O((n+m) \times n \times m)$. If $m = n$, we have an $O(n^3)$ solution which is only OK for $n \leq 200$.

The main issue with the $O(VE)$ Augmenting Path algorithm is that it may explore the longer augmenting paths first (as it is essentially a 'modified DFS'). This is not efficient. By exploring the *shorter* augmenting paths first, Hopcroft and Karp proved that their algorithm will only run in $O(\sqrt{V})$ iterations [28]. In each iteration, Hopcroft Karp's algorithm executes an $O(E)$ BFS from all the free vertices on the left set and finds augmenting paths of increasing lengths (starting from length 1: a free edge, length 3: a free edge, a matched edge, and a free edge again, length 5, length 7, and so on...). Then, it calls another $O(E)$ DFS to augment those augmenting paths (Hopcroft Karp's algorithm can increase *more than one matching* in one algorithm iteration). Therefore, the overall time complexity of Hopcroft Karp's algorithm is $O(\sqrt{V}E)$.

For the extreme example on Complete Bipartite Graph $K_{n,m}$ shown above, the Hopcroft Karp's algorithm has worst case time complexity of $O(\sqrt{(n+m)} \times n \times m)$. If $m = n$, we have an $O(n^{\frac{5}{2}})$ solution which is OK for $n \leq 600$. Therefore, if the problem author is 'nasty enough' to set $n \approx 500$ and relatively dense bipartite graph for an Unweighted MCBM problem, using Hopcroft Karp's is safer than the standard Augmenting Path algorithm (however, see **Exercise 4.7.4.3*** for a trick to make Augmenting Path algorithm runs 'fast enough' even if the input is a relatively dense and large bipartite graph).

Exercise 9.12.1*: Implement the Hopcroft Karp's algorithm starting from the Augmenting Path algorithm shown in Section 4.7.4 using the idea shown above.

Exercise 9.12.2*: Investigate the similarities and differences of Hopcroft Karp's algorithm and Dinic's algorithm shown in Section 9.7!

9.13 Independent and Edge-Disjoint Paths

Problem Description

Two paths that start from a source vertex s to a sink vertex t are said to be *independent* (vertex-disjoint) if they do not share any vertex apart from s and t.

Two paths that start from a source s to sink t are said to be edge-disjoint if they do not share any edge (but they can share vertices other than s and t).

Given a graph G, find the maximum number of independent and edge-disjoint paths from source s to sink t.

Solution(s)

The problem of finding the (maximum number of) independent paths from source s to sink t can be reduced to the Network (Max) Flow problem. We construct a flow network $N = (V, E)$ from G with vertex capacities, where N is the carbon copy of G except that the capacity of each $v \in V$ is 1 (i.e. each vertex can only be used once—see how to deal with vertex capacity in Section 4.6) and the capacity of each $e \in E$ is also 1 (i.e. each edge can only be used once too). Then run the Edmonds Karp's algorithm as per normal.

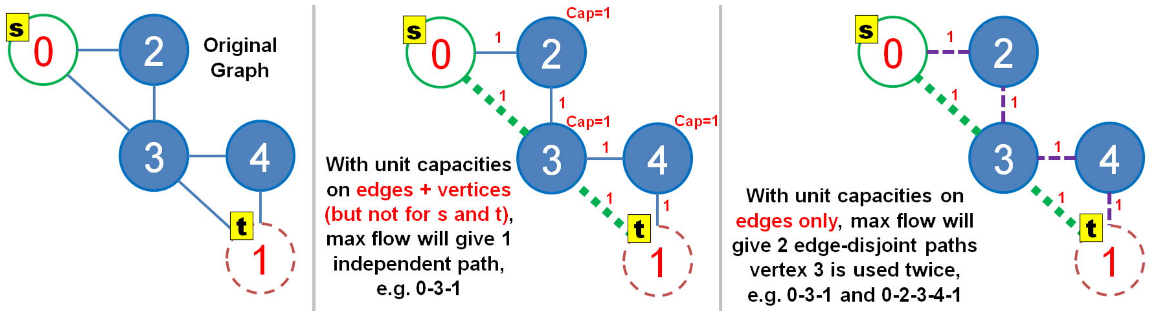
s
0
2
Original Graph
3
4
t
1
Cap=1
1
1
Cap=1
Cap=1
1
1
1
With unit capacities on edges + vertices (but not for s and t), max flow will give 1 independent path, e.g. 0-3-1
With unit capacities on edges only, max flow will give 2 edge-disjoint paths vertex 3 is used twice, e.g. 0-3-1 and 0-2-3-4-1

Figure 9.7: Comparison Between Max Independent Paths vs Max Edge-Disjoint Paths

Finding the (maximum number of) edge-disjoint paths from s to t is similar to finding (maximum) independent paths. The only difference is that this time we do not have any vertex capacity which implies that two edge-disjoint paths can still share the same vertex. See Figure 9.7 for a comparison between maximum independent paths and edge-disjoint paths from $s = 0$ to $t = 6$.

Programming exercises related to Independent and Edge-Disjoint Paths problem:

1. **UVa 00563 - Crimewave *** (check whether the maximum number of independent paths on the flow graph—with unit edge and unit vertex capacity—equals to b banks; analyze the upperbound of the answer to realize that the standard max flow solution suffices even for the largest test case)
2. **UVa 01242 - Necklace *** (LA 4271, Hefei08, to have a necklace, we need to be able to *two* edge-disjoint s-t flows)

9.14 Inversion Index

Problem Description

Inversion index problem is defined as follows: Given a list of numbers, count the minimum number of 'bubble sort' swaps (swap between pair of consecutive items) that are needed to make the list sorted in (usually ascending) order.

For example, if the content of the list is {3, 2, 1, 4}, we need 3 'bubble sort' swaps to make this list sorted in ascending order, i.e. swap (3, 2) to get $\{\underline{2, 3}, 1, 4\}$, swap (3, 1) to get $\{2, \underline{1, 3}, 4\}$, and finally swap (2, 1) to get $\{\underline{1, 2}, 3, 4\}$.

Solution(s)

$O(n^2)$ solution

The most obvious solution is to count how many swaps are needed during the actual running of the $O(n^2)$ bubble sort algorithm.

$O(n \log n)$ solution

The better $O(n \log n)$ Divide and Conquer solution for this inversion index problem is to modify merge sort. During the merge process of merge sort, if the front of the right sorted sublist is taken first rather than the front of the left sorted sublist, we say that 'inversion occurs'. Add inversion index counter by the size of the current left sublist. When merge sort is completed, report the value of this counter. As we only add $O(1)$ steps to merge sort, this solution has the same time complexity as merge sort, i.e. $O(n \log n)$.

On the example above, we initially have: {3, 2, 1, 4}. Merge sort will split this into sublist {3, 2} and {1, 4}. The left sublist will cause one inversion as we have to swap 3 and 2 to get {2, 3}. The right sublist {1, 4} will not cause any inversion as it is already sorted. Now, we merge {2, 3} with {1, 4}. The first number to be taken is 1 from the front of the right sublist. We have two more inversions because the left sublist has two members: {2, 3} that have to be swapped with 1. There is no more inversion after this. Therefore, there are a total of 3 inversions for this example.

Programming exercises related to Inversion Index problem:

1. UVa 00299 - Train Swapping (solvable with $O(n^2)$ bubble sort)
2. **UVa 00612 - DNA Sorting *** (needs $O(n^2)$ `stable_sort`)
3. **UVa 10327 - Flip Sort *** (solvable with $O(n^2)$ bubble sort)
4. UVa 10810 - Ultra Quicksort (requires $O(n \log n)$ merge sort)
5. UVa 11495 - Bubbles and Buckets (requires $O(n \log n)$ merge sort)
6. **UVa 11858 - Frosh Week *** (requires $O(n \log n)$ merge sort; 64-bit integer)

9.15 Josephus Problem

Problem Description

The Josephus problem is a classic problem where initially there are n people numbered from 1, 2, …, n, standing in a circle. Every k-th person is going to be executed and removed from the circle. This count-then-execute process is repeated until there is only one person left and this person will be saved (history said that he was the person named Josephus).

Solution(s)

Complete Search for Smaller Instances

The smaller instances of Josephus problem are solvable with Complete Search (see Section 3.2) by simply simulating the process with help of a cyclic array (or a circular linked list). The larger instances of Josephus problem require better solutions.

Special Case when $k = 2$

There is an elegant way to determine the position of the last surviving person for $k = 2$ using binary representation of the number n. If $n = 1b_1b_2b_3..b_n$ then the answer is $b_1b_2b_3..b_n1$, i.e. we move the most significant bit of n to the back to make it the least significant bit. This way, the Josephus problem with $k = 2$ can be solved in $O(1)$.

General Case

Let $F(n, k)$ denotes the position of the survivor for a circle of size n and with k skipping rule and we number the people from 0, 1, …, $n - 1$ (we will later add +1 to the final answer to match the format of the original problem description above). After the k-th person is killed, the circle shrinks by one to size $n - 1$ and the position of the survivor is now $F(n - 1, k)$. This relation is captured with equation $F(n, k) = (F(n - 1, k) + k)\%n$. The base case is when $n = 1$ where we have $F(1, k) = 0$. This recurrence has a time complexity of $O(n)$.

Other Variants

Josephus problem has several other variants that cannot be name one by one in this book.

Programming exercises related to Josephus problem:

1. UVa 00130 - Roman Roulette (the original Josephus problem)
2. UVa 00133 - The Dole Queue (brute force, similar to UVa 130)
3. UVa 00151 - Power Crisis (the original Josephus problem)
4. UVa 00305 - Joseph (the answer can be precalculated)
5. UVa 00402 - M*A*S*H (modified Josephus, simulation)
6. UVa 00440 - Eeny Meeny Moo (brute force, similar to UVa 151)
7. UVa 10015 - Joseph's Cousin (modified Josephus, dynamic k, variant of UVa 305)
8. ***UVa 10771 - Barbarian tribes **** (brute force, input size is small)
9. ***UVa 10774 - Repeated Josephus **** (repeated case of Josephus when $k = 2$)
10. ***UVa 11351 - Last Man Standing **** (use general case Josephus recurrence)

9.16 Knight Moves

Problem Description

In chess, a knight can move in an interesting 'L-shaped' way. Formally, a knight can move from a cell (r_1, c_1) to another cell (r_2, c_2) in an $n \times n$ chessboard if and only if $(r_1 - r_2)^2 + (c_1 - c_2)^2 = 5$. A common query is the length of shortest moves to move a knight from a starting cell to another target cell. There can be many queries on the same chessboard.

Solution(s)

One BFS per Query

If the chessboard size is small, we can afford to run one BFS per query. For each query, we run BFS from the starting cell. Each cell has at most 8 edges connected to another cells (some cells around the border of the chessboard have less edges). We stop BFS as soon as we reach the target cell. We can use BFS on this shortest path problem as the graph is unweighted (see Section 4.4.2). As there are up to $O(n^2)$ cells in the chessboard, the overall time complexity is $O(n^2 + 8n^2) = O(n^2)$ per query or $O(Qn^2)$ if there are Q queries.

One Precalculated BFS and Handling Special Cases

The solution above is not the most efficient way to solve this problem. If the given chessboard is large and there are several queries, e.g. $n = 1000$ and $Q = 16$ in UVa 11643 - Knight Tour, the approach above will get TLE.

A better solution is to realize that if the chessboard is large enough and we pick two random cells (r_a, c_a) and (r_b, c_b) in the middle of the chessboard with shortest knight moves of d steps between them, shifting the cell positions by a constant factor does not change the answer, i.e. the shortest knight moves from $(r_a + k, c_a + k)$ and $(r_b + k, c_b + k)$ is also d steps, for a constant factor k.

Therefore, we can just run *one* BFS from an arbitrary source cell and do some adjustments to the answer. However, there are a few special (literally) corner cases to be handled. Finding these special cases can be a headache and many Wrong Answers are expected if one does not know them yet. To make this section interesting, we purposely leave this crucial last step as a starred exercise. Try solving UVa 11643 after you get these answers.

Exercise 9.16.1*: Find those special cases and address them. Hints:

1. Separate cases when $3 \leq n \leq 4$ and $n \geq 5$.
2. Literally concentrate on corner cells and side cells.
3. What happen if the starting cell and the target cell are too close?

Programming exercises related to Knight Tour problem:

1. **UVa 00439 - Knight Moves *** (one BFS per query is enough)
2. ***UVa 11643 - Knight Tour **** (the distance between any 2 interesting positions can be obtained by using a pre-calculated BFS table (plus handling of the special corner cases); afterwards, this is just classic TSP problem, see Section 3.5.2)

9.17 Kosaraju's Algorithm

Finding Strongly Connected Components (SCCs) of a directed graph is a classic graph problem that has been discussed in Section 4.2.9. We have seen a modified DFS called Tarjan's algorithm that can solve this problem in efficient $O(V + E)$ time.

In this section, we present another DFS-based algorithm that can be used to find SCCs of a directed graph called the Kosaraju's algorithm. The basic idea of this algorithm is to run DFS *twice*. The *first* DFS is done on the *original directed graph* and record the 'post-order' traversal of the vertices as in finding topological sort[11] in Section 4.2.5. The *second* DFS is done on the *transpose of the original directed graph* using the 'post-order' ordering found by the first DFS. This two passes of DFS is enough to find the SCCs of the directed graph. The C++ implementation of this algorithm is shown below. We encourage readers to read more details on how this algorithm works from another source, e.g. [7].

```
void Kosaraju(int u, int pass) {      // pass = 1 (original), 2 (transpose)
  dfs_num[u] = 1;
  vii neighbor;           // use different Adjacency List in the two passes
  if (pass == 1) neighbor = AdjList[u]; else neighbor = AdjListT[u];
  for (int j = 0; j < (int)neighbor.size(); j++) {
    ii v = neighbor[j];
    if (dfs_num[v.first] == DFS_WHITE)
      Kosaraju(v.first, pass);
  }
  S.push_back(u);       // as in finding topological order in Section 4.2.5
}

// in int main()
  S.clear();  // first pass is to record the 'post-order' of original graph
  dfs_num.assign(N, DFS_WHITE);
  for (i = 0; i < N; i++)
    if (dfs_num[i] == DFS_WHITE)
      Kosaraju(i, 1);
  numSCC = 0;   // second pass: explore the SCCs based on first pass result
  dfs_num.assign(N, DFS_WHITE);
  for (i = N - 1; i >= 0; i--)
    if (dfs_num[S[i]] == DFS_WHITE) {
      numSCC++;
      Kosaraju(S[i], 2); // AdjListT -> the transpose of the original graph
    }

  printf("There are %d SCCs\n", numSCC);
```

Source code: `UVa11838.cpp/java`

Kosaraju's algorithm requires graph transpose routine (or build two graph data structures upfront) that is mentioned briefly in Section 2.4.1 and it needs two passes through the graph data structure. Tarjan's algorithm presented in Section 4.2.9 does not need graph transpose routine and it only needs only one pass. However, these two SCC finding algorithms are equally good and can be used to solve many (if not all) SCC problems listed in this book.

[11]But this may not be a valid topological sort as the input directed graph may be cyclic.

9.18 Lowest Common Ancestor

Problem Description

Given a rooted tree T with n vertices, the Lowest Common Ancestor (LCA) between two vertices u and v, or $LCA(u, v)$, is defined as the lowest vertex in T that has both u and v as descendants. We allow a vertex to be a descendant of itself, i.e. there is a possibility that $LCA(u, v) = u$ or $LCA(u, v) = v$.

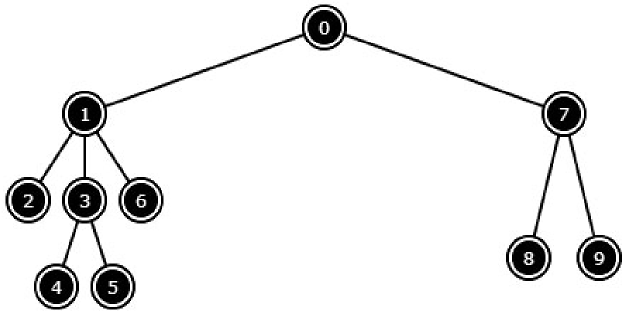

Figure 9.8: An example of a rooted tree T with $n = 10$ vertices

For example, in Figure 9.8, verify that the $LCA(4, 5) = 3$, $LCA(4, 6) = 1$, $LCA(4, 1) = 1$, $LCA(8, 9) = 7$, $LCA(4, 8) = 0$, and $LCA(0, 0) = 0$.

Solution(s)

Complete Search Solution

A naïve solution is to do two steps: From the first vertex u, we go all the way up to the root of T and record all vertices traversed along the way (this can be $O(n)$ if the tree is a very unbalanced). From the second vertex v, we also go all the way up to the root of T, but this time we stop if we encounter a common vertex for the first time (this can also be $O(n)$ if the $LCA(u, v)$ is the root and the tree is very unbalanced). This common vertex is the LCA. This requires $O(n)$ per (u, v) query and can be very slow if there are many queries.

For example, if we want to compute $LCA(4, 6)$ of the tree in Figure 9.8 using this complete search solution, we will first traverse path $4 \to 3 \to 1 \to 0$ and record these 4 vertices. Then, we traverse path $6 \to 1$ and then stop. We report that the LCA is vertex 1.

Reduction to Range Minimum Query

We can reduce the LCA problem into a Range Minimum Query (RMQ) problem (see Section 2.4.3). If the structure of the tree T is not changed throughout all Q queries, we can use the Sparse Table data structure with $O(n \log n)$ construction time and $O(1)$ RMQ time. The details on the Sparse Table data structure is shown in Section 9.33. In this section, we highlight the reduction process from LCA to RMQ.

We can reduce LCA to RMQ in linear time. The key idea is to observe that $LCA(u, v)$ is the shallowest vertex in the tree that is visited between the visits of u and v during a DFS traversal. So what we need to do is to run a DFS on the tree and record information about the depth and the time of visit for every node. Notice that we will visit a total of $2 * n - 1$ vertices in the DFS since the internal vertices will be visited several times. We need to build three arrays during this DFS: `E[0..2*n-2]` (which records the sequence of visited nodes), `L[0..2*n-2]` (which records the depth of each visited node), and `H[0..N-1]` (where `H[i]` records the index of the first occurrence of node `i` in `E`).

The key portion of the implementation is shown below:

```
int L[2*MAX_N], E[2*MAX_N], H[MAX_N], idx;

void dfs(int cur, int depth) {
  H[cur] = idx;
  E[idx] = cur;
  L[idx++] = depth;
  for (int i = 0; i < children[cur].size(); i++) {
    dfs(children[cur][i], depth+1);
    E[idx] = cur;                              // backtrack to current node
    L[idx++] = depth;
  }
}

void buildRMQ() {
  idx = 0;
  memset(H, -1, sizeof H);
  dfs(0, 0);                           // we assume that the root is at index 0
}
```

Source code: `LCA.cpp/java`

For example, if we call `dfs(0, 0)` on the tree in Figure 9.8, we will have[12]:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
H	0	1	2	4	5	7	10	13	14	16	-1	-1	-1	-1	-1	-1	-1	-1	-1
E	0	1	2	1	3	4	3	5	3	*(1)*	6	1	0	7	8	7	9	7	0
L	0	1	2	1	2	<u>3</u>	<u>2</u>	<u>3</u>	<u>2</u>	<u>1</u>	<u>2</u>	1	0	1	2	1	2	1	0

Table 9.1: The Reduction from LCA to RMQ

Once we have these three arrays to work with, we can solve LCA using RMQ. Assume that `H[u]` $<$ `H[v]` or swap u and v otherwise. We notice that the problem reduces to finding the vertex with the smallest depth in `E[H[u]..H[v]]`. So the solution is given by $LCA(u, v)$ = `E[RMQ(H[u], H[v])]` where `RMQ(i, j)` is executed on the `L` array. If using the Sparse Table data structure shown in Section 9.33, it is the `L` array that needs to be processed in the construction phase.

For example, if we want to compute $LCA(4, 6)$ of the tree in Figure 9.8, we will compute `H[4] = 5` and `H[6] = 10` and find the vertex with the smallest depth in `E[5..10]`. Calling `RMQ(5, 10)` on array `L` (see the underlined entries in row `L` of Table 9.1) returns index 9. The value of `E[9]` $= 1$ (see the italicized entry in row `E` of Table 9.1), therefore we report 1 as the answer of $LCA(4, 6)$.

Programming exercises related to LCA:

1. UVa 10938 - Flea circus (Lowest Common Ancestor as outlined above)
2. *UVa 12238 - Ants Colony* (very similar to UVa 10938)

[12]In Section 4.2.1, `H` is named as `dfs_num`.

9.19 Magic Square Construction (Odd Size)

Problem Description

A magic square is a 2D array of size $n \times n$ that contains integers from `[1..`n^2`]` with 'magic' property: The sum of integers in each row, column, and diagonal is the same. For example, for $n = 5$, we can have the following magic square below that has row sums, column sums, and diagonal sums equals to 65.

$$\begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{bmatrix}$$

Our task is to construct a magic square given its size n, assuming that n is odd.

Solution(s)

If we do not know the solution, we may have to use the standard recursive backtracking routine that try to place each integer $\in$ `[1..`n^2`]` one by one. Such Complete Search solution is too slow for large n.

Fortunately, there is a nice 'construction strategy' for magic square of odd size called the 'Siamese (De la Loubère) method'. We start from an empty 2D square array. Initially, we put integer 1 in the middle of the first row. Then we move northeast, wrapping around as necessary. If the new cell is currently empty, we add the next integer in that cell. If the cell has been occupied, we move one row down and continue going northeast. This Siamese method is shown in Figure 9.9. We reckon that deriving this strategy without prior exposure to this problem is likely not straightforward (although not impossible if one stares at the structure of several odd-sized Magic Squares long enough).

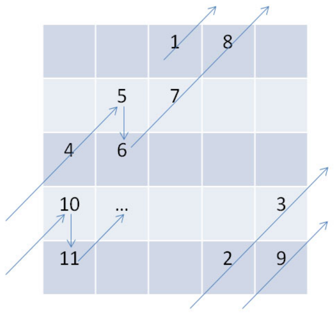

Figure 9.9: The Magic Square Construction Strategy for Odd n

There are other special cases for Magic Square construction of different sizes. It may be unnecessary to learn all of them as most likely it will not appear in programming contest. However, we can imagine some contestants who know such Magic Square construction strategies will have advantage in case such problem appears.

Programming exercises related to Magic Square:

1. ***UVa 01266 - Magic Square **** (follow the given construction strategy)

9.20 Matrix Chain Multiplication

Problem Description

Given n matrices: $A_1, A_2, \ldots, A_n$, each A_i has size $P_{i-1} \times P_i$, output a complete parenthesized product $A_1 \times A_2 \times \ldots \times A_n$ that minimizes the number of scalar multiplications. A product of matrices is called completely parenthesized if it is either:

1. A single matrix
2. The product of 2 completely parenthesized products surrounded by parentheses

For example, given 3 matrices array $P = \{10, 100, 5, 50\}$ (which implies that matrix A_1 has size 10×100, matrix A_2 has size 100×5, and matrix A_3 has size 5×50. We can completely parenthesize these three matrices in two ways:

1. $(A_1 \times (A_2 \times A_3)) = 100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$ scalar multiplications
2. $((A_1 \times A_2) \times A_3) = 10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$ scalar multiplications

From the example above, we can see that the cost of multiplying these 3 matrices—in terms of the number of scalar multiplications—depends on the choice of the complete parenthesization of the matrices. However, exhaustively checking all possible complete parenthesizations is too slow as there are a huge number of such possibilities (for interested reader, there are $Cat(n-1)$ complete parenthesization of n matrices—see Section 5.4.3).

Matrix Multiplication

We can multiple two matrices a of size $p \times q$ and b of size $q \times r$ if the number of columns of a is the same as the number of rows of b (the inner dimension agree). The result of this multiplication is matrix c of size $p \times r$. The cost of such valid matrix multiplication is $O(p \times q \times r)$ multiplications and can be implemented with a short C++ code as follows:

```
#define MAX_N 10                   // increase/decrease this value as needed
struct Matrix { int mat[MAX_N][MAX_N]; };

Matrix matMul(Matrix a, Matrix b, int p, int q, int r) {          // O(pqr)
  Matrix c; int i, j, k;
  for (i = 0; i < p; i++)
    for (j = 0; j < r; j++)
      for (c.mat[i][j] = k = 0; k < q; k++)
        c.mat[i][j] += a.mat[i][k] + b.mat[k][j];
  return c; }
```

For example, if we have 2×3 matrix a and 3×1 matrix b below, we need $2 \times 3 \times 1 = 6$ scalar multiplications.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} \times \begin{bmatrix} b_{1,1} \\ b_{2,1} \\ b_{3,1} \end{bmatrix} = \begin{bmatrix} c_{1,1} = a_{1,1} \times b_{1,1} + a_{1,2} \times b_{2,1} + a_{1,3} \times b_{3,1} \\ c_{2,1} = a_{2,1} \times b_{1,1} + a_{2,2} \times b_{2,1} + a_{2,3} \times b_{3,1} \end{bmatrix}$$

When the two matrices are square matrices of size $n \times n$, this matrix multiplication runs in $O(n^3)$ (see Section 9.21 which is very similar with this one).

Solution(s)

This Matrix Chain Multiplication problem is usually one of the classic example to illustrate Dynamic Programming (DP) technique. As we have discussed DP in details in Section 3.5, we only outline the key ideas here. Note that for this problem, we do not actually multiply the matrices as shown in earlier subsection. We just need to find the optimal complete parenthesization of the n matrices.

Let $cost(i, j)$ where $i < j$ denotes the number of scalar multiplications needed to multiply matrix $A_i \times A_{i+1} \times \ldots \times A_j$. We have the following Complete Search recurrences:

1. $cost(i, j) = 0$ if $i = j$
2. $cost(i, j) = min(cost(i, k) + cost(k + 1, j) + P_{i-1} \times P_k \times P_j), \forall k \in [i \ldots j - 1]$

The optimal cost is stored in $cost(1, n)$. There are $O(n^2)$ different pairs of subproblem (i, j). Therefore, we need a DP table of size $O(n^2)$. Each subproblem requires up to $O(n)$ to be computed. Therefore, the time complexity of this DP solution for Matrix Chain Multiplication problem is $O(n^3)$.

Programming exercises related to Matrix Chain Multiplication:

1. **UVa 00348 - Optimal Array Mult ... *** (as above, output the optimal solution too; note that the optimal matrix multiplication sequence is not unique; e.g. imagine if all matrices are square matrices)

9.21 Matrix Power

Some Definitions and Sample Usages

In this section, we discuss a special case of matrix[13]: The *square matrix*[14]. To be precise, we discuss a special operation of square matrix: The *powers of a square matrix*. Mathematically, $M^0 = I$ and $M^p = \prod_{i=1}^{p} M$. I is the *Identity* matrix[15] and p is the given power of square matrix M. If we can do this operation in $O(n^3 \log p)$—which is the main topic of this subsection, we can solve some more interesting problems in programming contests, e.g.:

- Compute a *single*[16] Fibonacci number $fib(p)$ in $O(\log p)$ time instead of $O(p)$. Imagine if $p = 2^{30}$, $O(p)$ solution will get TLE but $\log_2(p)$ solution just need 30 steps. This is achievable by using the following equality:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^p = \begin{bmatrix} fib(p+1) & \underline{\mathbf{fib(p)}} \\ \underline{\mathbf{fib(p)}} & fib(p-1) \end{bmatrix}$$

  For example, to compute $fib(11)$, we simply multiply the Fibonacci matrix 11 times, i.e. raise it to the power of 11. The answer is in the secondary diagonal of the matrix.

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{11} = \begin{bmatrix} 144 & \underline{\mathbf{89}} \\ \underline{\mathbf{89}} & 55 \end{bmatrix} = \begin{bmatrix} fib(12) & \underline{\mathbf{fib(11)}} \\ \underline{\mathbf{fib(11)}} & fib(10) \end{bmatrix}$$

- Compute the number of paths of length L of a graph stored in an Adjacency Matrix—which is a square matrix—in $O(n^3 \log L)$. Example: See the small graph of size $n = 4$ stored in an Adjacency Matrix M below. The various paths from vertex 0 to vertex 1 with different lengths are shown in entry $M[0][1]$ after M is raised to power L.

```
The graph:   0->1 with length 1: 0->1 (only 1 path)
             0->1 with length 2: impossible
 0--1        0->1 with length 3: 0->1->2->1 (and 0->1->0->1)
    |        0->1 with length 4: impossible
    2--3     0->1 with length 5: 0->1->2->3->2->1 (and 4 others)
```

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} M^2 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 1 \\ 1 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} M^3 = \begin{bmatrix} 0 & 2 & 0 & 1 \\ 2 & 0 & 3 & 0 \\ 0 & 3 & 0 & 2 \\ 1 & 0 & 2 & 0 \end{bmatrix} M^5 = \begin{bmatrix} 0 & 5 & 0 & 3 \\ 5 & 0 & 8 & 0 \\ 0 & 8 & 0 & 5 \\ 3 & 0 & 5 & 0 \end{bmatrix}$$

- Speed-up *some* DP problems as shown later in this section.

[13]A matrix is a rectangular (2D) array of numbers. Matrix of size $m \times n$ has m rows and n columns. The elements of the matrix is usually denoted by the matrix name with two subscripts.

[14]A square matrix is a matrix with the same number of rows and columns, i.e. it has size $n \times n$.

[15]Identity matrix is a matrix with all zeroes except that cells along the main diagonal are all ones.

[16]If we need $fib(n)$ **for all** $n \in$ `[0..n]`, use $O(n)$ DP solution instead.

The Idea of Efficient Exponentiation (Power)

For the sake of discussion, let's assume that built-in library functions like *pow(base, p)* or other related functions that can raise a number *base* to a certain integer power p does not exist. Now, if we do exponentiation 'by definition' as shown below, we will have an inefficient $O(p)$ solution, especially if p is large[17].

```
int normalExp(int base, int p) {    // for simplicity, we use int data type
  int ans = 1;          // we also assume that ans will not exceed 2^31 - 1
  for (int i = 0; i < p; i++) ans *= base;                  // this is O(p)
  return ans; }
```

There is a better solution that uses Divide & Conquer principle. We can express A^p as:

$A^0 = 1$ (base case).
$A^1 = A$ (another base case, but see **Exercise 9.21.1**).
$A^p = A^{p-1} \times A$ if p is odd.
$A^p = (A^{p/2})^2$ if p is even.
As this approach keeps halving the value of p by two, it runs in $O(\log p)$.

For example, by definition: $2^9 = 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \approx O(p)$ multiplications.
But with Divide & Conquer: $2^9 = 2^8 \times 2 = (2^4)^2 \times 2 = ((2^2)^2)^2 \times 2 \approx O(\log p)$ multiplications.

A typical recursive implementation of this Divide & Conquer exponentiation—omitting cases when the answer exceeds the range of 32-bit integer—is shown below:

```
int fastExp(int base, int p) {                                  // O(log p)
       if (p == 0) return 1;
  else if (p == 1) return base;                    // See the Exercise below
  else {           int res = fastExp(base, p / 2); res *= res;
                   if (p % 2 == 1) res *= base;
                   return res; } }
```

Exercise 9.21.1*: Do we actually need the second base case: `if (p == 1) return base;`?

Exercise 9.21.2*: Raising a number to a certain (integer) power can easily cause overflow. An interesting variant is to compute $base^p (\text{mod } m)$. Rewrite function `fastExp(base, p)` into `modPow(base, p, m)` (also see Section 5.3.2 and Section 5.5.8)!

Exercise 9.21.3*: Rewrite the recursive implementation of Divide & Conquer implementation into an iterative implementation. Hint: Continue reading this section.

Square Matrix Exponentiation (Matrix Power)

We can use the same $O(\log p)$ efficient exponentiation technique shown above to perform square matrix exponentiation (matrix power) in $O(n^3 \log p)$ because each matrix multiplication[18] is $O(n^3)$. The *iterative* implementation (for comparison with the recursive implementation shown earlier) is shown below:

[17]If you encounter input size of 'gigantic' value in programming contest problems, like 1B, the problem author is *usually* looking for a logarithmic solution. Notice that $\log_2(1B) \approx \log_2(2^{30})$ is still just 30!

[18]There exists a faster but more complex algorithm for matrix multiplication: The $O(n^{2.8074})$ Strassen's algorithm. Usually we do not use this algorithm for programming contests. Multiplying two Fibonacci matrices shown in Section 9.21 only requires $2^3 = 8$ multiplications as $n = 2$. This can be treated as $O(1)$. Thus, we can compute $fib(p)$ in $O(\log p)$.

```
#define MAX_N 2 // Fibonacci matrix, increase/decrease this value as needed
struct Matrix { int mat[MAX_N][MAX_N]; };      // we will return a 2D array

Matrix matMul(Matrix a, Matrix b) {                                // O(n^3)
  Matrix ans; int i, j, k;
  for (i = 0; i < MAX_N; i++)
    for (j = 0; j < MAX_N; j++)
      for (ans.mat[i][j] = k = 0; k < MAX_N; k++)      // if necessary, use
        ans.mat[i][j] += a.mat[i][k] * b.mat[k][j];    // modulo arithmetic
  return ans; }

Matrix matPow(Matrix base, int p) {                          // O(n^3 log p)
  Matrix ans; int i, j;
  for (i = 0; i < MAX_N; i++) for (j = 0; j < MAX_N; j++)
    ans.mat[i][j] = (i == j);                         // prepare identity matrix
  while (p) {       // iterative version of Divide & Conquer exponentiation
    if (p & 1) ans = matMul(ans, base);     // if p is odd (last bit is on)
    base = matMul(base, base);                             // square the base
    p >>= 1;                                                 // divide p by 2
  }
  return ans; }
```

Source code: `UVa10229.cpp/java`

DP Speed-up with Matrix Power

In this section, we discuss how to derive the required square matrices for two DP problems and show that raising these two square matrices to the required powers can speed-up the computation of the original DP problems.

We start with the 2×2 Fibonacci matrix. We know that $fib(0) = 0$, $fib(1) = 1$, and for $n \geq 2$, we have $fib(n) = fib(n-1) + fib(n-2)$. We can compute $fib(n)$ in $O(n)$ by using Dynamic Programming by computing $fib(n)$ *one by one* progressively from $[2..n]$. However, these DP transitions *can be made faster* by re-writing the Fibonacci recurrence into a matrix form as shown below:

First, we write two versions of Fibonacci recurrence as there are two terms in the recurrence:

$$\begin{aligned} fib(n+1) + fib(n) &= fib(n+2) \\ fib(n) + fib(n-1) &= fib(n+1) \end{aligned}$$

Then, we re-write the recurrence into matrix form:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} fib(n+1) \\ fib(n) \end{bmatrix} = \begin{bmatrix} fib(n+2) \\ fib(n+1) \end{bmatrix}$$

Now we have $a \times fib(n+1) + b \times fib(n) = fib(n+2)$ and $c \times fib(n+1) + d \times fib(n) = fib(n+1)$. Notice that by writing the DP recurrence as shown above, we now have a 2×2 *square matrix*. The appropriate values for a, b, c, and d must be 1, 1, 1, 0 and this is the 2×2 Fibonacci matrix shown earlier. One matrix multiplication advances DP computation of Fibonacci number one step forward. If we multiply this 2×2 Fibonacci matrix p times, we advance DP computation of Fibonacci number p steps forward. We now have:

$$\underbrace{\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \ldots \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}}_{p} \times \begin{bmatrix} fib(n+1) \\ fib(n) \end{bmatrix} = \begin{bmatrix} fib(n+1+p) \\ fib(n+p) \end{bmatrix}$$

For example, if we set $n = 0$ and $p = 11$, and then use $O(\log p)$ matrix power instead of actually multiplying the matrix p times, we have the following calculations:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{11} \times \begin{bmatrix} fib(1) \\ fib(0) \end{bmatrix} = \begin{bmatrix} 144 & 89 \\ 89 & 55 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 144 \\ \underline{\mathbf{89}} \end{bmatrix} = \begin{bmatrix} fib(12) \\ \underline{\mathbf{fib(11)}} \end{bmatrix}$$

This Fibonacci matrix can also be written as shown earlier, i.e.

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{p} = \begin{bmatrix} fib(p+1) & fib(p) \\ fib(p) & fib(p-1) \end{bmatrix}$$

Let's discuss one more example on how to derive the required square matrix for another DP problem: UVa 10655 - Contemplation! Algebra. The problem description is very simple: Given the value of $p = a + b$, $q = a \times b$, and n, find the value of $a^n + b^n$.

First, we tinker with the formula so that we can use $p = a + b$ and $q = a \times b$:

$$a^n + b^n = (a + b) \times (a^{n-1} + b^{n-1}) - (a \times b) \times (a^{n-2} + b^{n-2})$$

Next, we set $X_n = a^n + b^n$ to have $X_n = p \times X_{n-1} - q \times X_{n-2}$.
Then, we write this recurrence twice in the following form:

$$p \times X_{n+1} - q \times X_n = X_{n+2}$$
$$p \times X_n - q \times X_{n-1} = X_{n+1}$$

Then, we re-write the recurrence into matrix form:

$$\begin{bmatrix} p & -q \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} X_{n+1} \\ X_n \end{bmatrix} = \begin{bmatrix} X_{n+2} \\ X_{n+1} \end{bmatrix}$$

If we raise the 2×2 square matrix to the power of n (in $O(\log n)$ time) and then multiply the resulting square matrix with $X_1 = a^1 + b^1 = a + b = p$ and $X_0 = a^0 + b^0 = 1 + 1 = 2$, we have X_{n+1} and X_n. The required answer is X_n. This is faster than $O(n)$ standard DP computation for the same recurrence.

$$\begin{bmatrix} p & -q \\ 1 & 0 \end{bmatrix}^{n} \times \begin{bmatrix} X_1 \\ X_0 \end{bmatrix} = \begin{bmatrix} X_{n+1} \\ X_n \end{bmatrix}$$

Programming Exercises related to Matrix Power:

1. UVa 10229 - Modular Fibonacci (discussed in this section + modulo)
2. ***UVa 10518 - How Many Calls? **** (derive the pattern of the answers for small n; the answer is $2 \times fib(n) - 1$; then use UVa 10229 solution)
3. ***UVa 10655 - Contemplation, Algebra **** (discussed in this section)
4. UVa 10870 - Recurrences (form the required matrix first; power of matrix)
5. ***UVa 11486 - Finding Paths in Grid **** (model as adjacency matrix; raise the adjacency matrix to the power of N in $O(\log N)$ to get the number of paths)
6. *UVa 12470 - Tribonacci* (very similar to UVa 10229; the 3×3 matrix is = [0 1 0; 0 0 1; 1 1 1]; the answer is at matrix[1][1] after it is raised to the power of n and with modulo 1000000009)

9.22 Max Weighted Independent Set

Problem Description

Given a *vertex-weighted* graph G, find the Max *Weighted* Independent Set (MWIS) of G. An Independent Set (IS)[19] is a set of vertices in a graph, no two of which are adjacent. Our task is to select an IS of G with the maximum total (vertex) weight. This is a hard problem on a general graph. However, if the given graph G is a tree or a bipartite graph, we have efficient solutions.

Solution(s)

On Tree

If graph G is a tree[20], we can find the MWIS of G using DP[21]. Let $C(v, taken)$ be the MWIS of the subtree rooted at v if it is *taken* as part of the MWIS. We have the following complete search recurrences:

1. If v is a leaf vertex
   (a) $C(v, true) = w(v)$
   % If leaf v is taken, then the weight of this subtree is the weight of this v.
   (b) $C(v, false) = 0$
   % If leaf v is not taken, then the weight of this subtree is 0.
2. If v is an internal vertex
   (a) $C(v, true) = w(v) + \sum_{\texttt{ch} \in \texttt{children(v)}} C(ch, false)$
   % If root v is taken, we add weight of v but all children of v *cannot* be taken.
   (b) $C(v, false) = \sum_{\texttt{ch} \in \texttt{children(v)}} max(C(ch, true), C(ch, false))$
   % If root v is not taken, children of v may or may not be taken.
   % We return the larger one.

The answer is $max(C(root, 1), C(root, 0))$—take or not take the root. This DP solution just requires $O(V)$ space and $O(V)$ time.

On Bipartite Graph

If the graph G is a bipartite graph, we have to reduce MWIS problem[22], into a Max Flow problem. We assign the original vertex cost (the weight of taking that vertex) as capacity from source to that vertex for the left set of the bipartite graph and capacity from that vertex to sink for right set of the bipartite graph. Then, we give 'infinite' capacity in between any edge in between the left and right sets. The MWIS of this bipartite graph is the weight of all vertex cost minus the max flow value of this flow graph.

[19] For your information, the complement of Independent Set is Vertex Cover.

[20] For most tree-related problems, we need to 'root the tree' first if it is not yet rooted. If the tree does not have a vertex dedicated as the root, pick an arbitrary vertex as the root. By doing this, the subproblems w.r.t subtrees may appear, like in this MWIS problem on Tree.

[21] Some optimization problems on *tree* may be solved with DP techniques. The solution usually involves passing information from/to parent and getting information from/to the children of a rooted tree.

[22] The non-weighted Max Independent Set (MIS) problem on bipartite graph can be reduced into a Max Cardinality Bipartite Matching (MCBM) problem—see Section 4.7.4.

9.23 Min Cost (Max) Flow

Problem Description

The Min Cost Flow problem is the problem of finding the *cheapest* possible way of sending a certain amount of (usually max) flow through a flow network. In this problem, every edge has two attributes: The flow capacity through this edge *and the unit cost* for sending one unit flow through this edge. Some problem authors choose to simplify this problem by setting the edge capacity to a constant integer and only vary the edge cost.

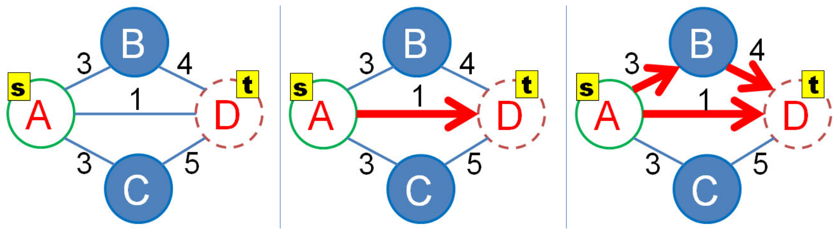

Figure 9.10: An Example of Min Cost Max Flow (MCMF) Problem (UVa 10594 [47])

Figure 9.10—left shows a (modified) instance of UVa 10594. Here, each edge has a uniform capacity of 10 units and a unit cost as shown in the edge label. We want to send 20 units of flow from A to D (note that the max flow of this flow graph is 30 units) which can be satisfied by sending 10 units of flow $A \to D$ with cost $1 \times 10 = 10$ (Figure 9.10—middle); plus another 10 units of flow $A \to B \to D$ with cost $(3 + 4) \times 10 = 70$ (Figure 9.10—right). The total cost is $10 + 70 = 80$ and this is the minimum. Note that if we choose to send the 20 units of flow via $A \to D$ (10 units) and $A \to \underline{C} \to D$ instead, we incur a cost of $1 \times 10 + (3 + \underline{5}) \times 10 = 10 + 80 = 90$. This is higher than the optimal cost of 80.

Solution(s)

The Min Cost (Max) Flow, or in short MCMF, can be solved by replacing the $O(E)$ BFS (to find the shortest—in terms of number of hops—augmenting path) in Edmonds Karp's algorithm into the $O(VE)$ Bellman Ford's (to find the shortest/cheapest—in terms of the *path cost*—augmenting path). We need a shortest path algorithm that can handle negative edge weights as such negative edge weights *may appear* when we cancel a certain flow along a backward edge (as we have to *subtract* the cost taken by this augmenting path as canceling flow means that we do not want to use that edge). See Figure 9.5 for an example.

The needs to use shortest path algorithm like Bellman Ford's slows down the MCMF implementation to around $O(V^2E^2)$ but this is usually compensated by the problem author of most MCMF problems by having smaller input graph constraints.

Programming exercises related to Min Cost (Max) Flow:

1. UVa 10594 - Data Flow (basic min cost max flow problem)
2. **UVa 10746 - Crime Wave - The Sequel *** (min *weighted* bip matching)
3. UVa 10806 - Dijkstra, Dijkstra (send 2 edge-disjoint flows with min cost)
4. ***UVa 10888 - Warehouse **** (BFS/SSSP; min *weighted* bipartite matching)
5. ***UVa 11301 - Great Wall of China **** (modeling, vertex capacity, MCMF)

9.24 Min Path Cover on DAG

Problem Description

The Min Path Cover (MPC) problem on DAG is described as the problem of finding the minimum number of paths to cover *each vertex* on DAG $G = (V, E)$. A path $v_0, v_1, \ldots, v_k$ is said to cover all vertices along its path.

Motivating problem—UVa 1201 - Taxi Cab Scheme: Imagine that the vertices in Figure 9.11.A are passengers, and we draw an edge between two vertices $u - v$ if one taxi can serve passenger u and then passenger v *on time*. The question is: What is the minimum number of taxis that must be deployed to serve *all* passengers?

The answer is two taxis. In Figure 9.11.D, we see one possible optimal solution. One taxi (dotted line) serves passenger 1, passenger 2, and then passenger 4. Another taxi (dashed line) serves passenger 3 and passenger 5. All passengers are served with just two taxis. Notice that there is one more optimal solution: $1 \to 3 \to 5$ and $2 \to 4$.

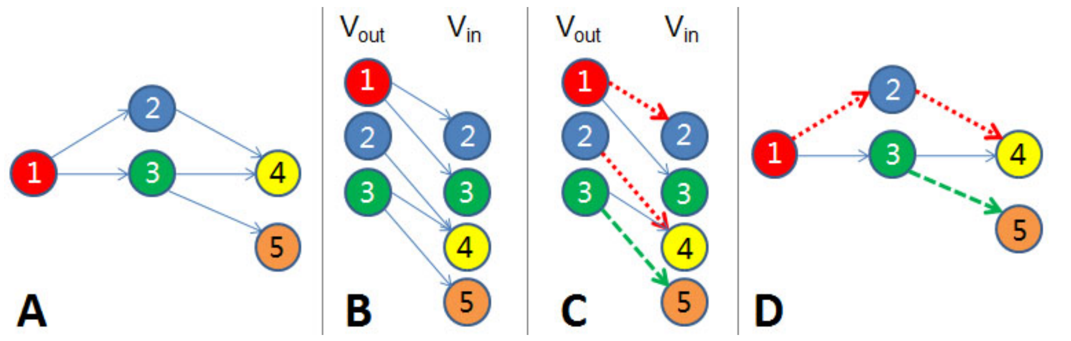

Figure 9.11: Min Path Cover on DAG (from UVa 1201 [47])

Solution(s)

This problem has a polynomial solution: Construct a *bipartite graph* $G' = (V_{out} \bigcup V_{in}, E')$ from G, where $V_{out} = \{v \in V : v$ has positive out-degree$\}$, $V_{in} = \{v \in V : v$ has positive in-degree$\}$, and $E' = \{(u, v) \in (Vout, Vin) : (u, v) \in E\}$. This G' is a bipartite graph. A matching on bipartite graph G' forces us to select at most one outgoing edge from every $u \in V_{out}$ (and similarly at most one incoming edge for $v \in V_{in}$). DAG G initially has n vertices, which can be covered with n paths of length 0 (the vertices themselves). One matching between vertex a and vertex b using edge (a, b) says that we can use one less path as edge $(a, b) \in E'$ can cover both vertices in $a \in V_{out}$ and $b \in V_{in}$. Thus if the MCBM in G' has size m, then we just need $n - m$ paths to cover each vertex in G.

The MCBM in G' that is needed to solve the MPC in G can be solved via several polynomial solutions, e.g. maximum flow solution, augmenting paths algorithm, or Hopcroft Karp's algorithm (see Section 9.10). As the solution for bipartite matching runs in polynomial time, the solution for the MPC in DAG also runs in polynomial time. Note that MPC in general graph is NP-hard.

Programming exercises related to Min Path Cover on DAG:

1. **UVa 01184 - Air Raid *** (LA 2696, Dhaka02, MPC on DAG $\approx$ MCBM)
2. **UVa 01201 - Taxi Cab Scheme *** (LA 3126, NWEurope04, MPC on DAG)

9.25 Pancake Sorting

Problem Description

Pancake Sorting is a classic[23] Computer Science problem, but it is rarely used. This problem can be described as follows: You are given a stack of **N** pancakes. The pancake at the bottom and at the top of the stack has **index 0** and **index N-1**, respectively. The size of a pancake is given by the pancake's diameter (**an integer** $\in$ **[1 .. MAX_D]**). All pancakes in the stack have **different** diameters. For example, a stack A of **N = 5** pancakes: {3, 8, 7, 6, 10} can be visualized as:

```
         4 (top)       10
         3             6
         2             7
         1             8
         0 (bottom)    3
       -----------------------
         index         A
```

Your task is to sort the stack in **descending order**—that is, the largest pancake is at the bottom and the smallest pancake is at the top. However, to make the problem more real-life like, sorting a stack of pancakes can only be done by a sequence of pancake 'flips', denoted by function **flip(i)**. A **flip(i)** move consists of inserting a spatula between two pancakes in a stack (at **index i** and **index N-1**) and flipping (reversing) the pancakes on the spatula (reversing the sub-stack **[i .. N-1]**).

For example, stack A can be transformed to stack B via **flip(0)**, i.e. inserting a spatula between index 0 and 4 then flipping the pancakes in between. Stack B can be transformed to stack C via **flip(3)**. Stack C can be transformed to stack D via **flip(1)**. And so on... Our target is to make the stack sorted in **descending order**, i.e. we want the final stack to be like stack E.

```
4 (top)       10 <--   3  <--   8  <--   6                 3
3             6        8  <--   3        7        ...      6
2             7        7        7        3                 7
1             8        6        6  <--   8                 8
0 (bottom)    3  <--   10       10       10                10
-------------------------------------------------------------
index         A        B        C        D        ...      E
```

To make the task more challenging, you have to compute the **minimum number of flip(i) operations** that you need so that the stack of **N** pancakes is sorted in descending order.

You are given an integer T in the first line, and then T test cases, one in each line. Each test case starts with an integer N, followed by N integers that describe the initial content of the stack. You have to output one integer, the minimum number of **flip(i)** operations to sort the stack.

Constraints: $\mathbf{1 \le T \le 100}$, $\mathbf{1 \le N \le 10}$, and $\mathbf{N \le MAX\_D \le 1000000}$.

[23] Bill Gates (Microsoft founder, former CEO, and current chairman) wrote only one research paper so far, and it is about this pancake sorting [22].

Sample Test Cases

Sample Input

```
7
4   4 3 2 1
8   8 7 6 5 4 1 2 3
5   5 1 2 4 3
5   555555 111111 222222 444444 333333
8   1000000 999999 999998 999997 999996 999995 999994 999993
5   3 8 7 6 10
10  8 1 9 2 0 5 7 3 6 4
```

Sample Output

```
0
1
2
2
0
4
11
```

Explanation

- The first stack is already sorted in descending order.
- The second stack can be sorted with one call of **flip(5)**.
- The third (and also the fourth) input stack can be sorted in descending order by calling **flip(3)** then **flip(1)**: 2 flips.
- The fifth input stack, although contains large integers, is already sorted in descending order, so 0 flip is needed.
- The sixth input stack is actually the sample stack shown in the problem description. This stack can be sorted in descending order using at minimum 4 flips, i.e.
  Solution 1: **flip(0)**, **flip(1)**, **flip(2)**, **flip(1)**: 4 flips.
  Solution 2: **flip(1)**, **flip(2)**, **flip(1)**, **flip(0)**: also 4 flips.
- The seventh stack with **N = 10** is for you to test the runtime speed of your solution.

Solution(s)

First, we need to make an observation that the diameters of the pancake do not really matter. We just need to write simple code to sort these (potentially huge) pancake diameters from [**1..1 million**] and relabel them to [**0..N-1**]. This way, we can describe any stack of pancakes as simply a permutation of N integers.

If we just need to get the pancakes sorted, we can use a non optimal $O(2 \times N - 3)$ Greedy algorithm: Flip the largest pancake to the top, then flip it to the bottom. Flip the second largest pancake to the top, then flip it to the second from bottom. And so on. If we keep doing this, we will be able to have a sorted pancake in $O(2 \times N - 3)$ steps, regardless of the initial state.

However, to get the minimum number of flip operations, we need to be able to model this problem as a Shortest Paths problem on unweighted State-Space graph (see Section 8.2.3). The vertex of this State-Space graph is a permutation of N pancakes. A vertex is connected with unweighted edges to $O(N-1)$ other vertices via various flip operations (minus one as flipping the topmost pancake does not change anything). We can then use BFS from the starting permutation to find the shortest path to the target permutation (where the permutation is sorted in descending order). There are up to $V = O(N!)$ vertices and up to $V = O(N! \times (N-1))$ in this State-Space graph. Therefore, an $O(V+E)$ BFS runs in $O(N \times N!)$ per test case or $O(T \times N \times N!)$ for all test cases. Note that coding such BFS is already a challenging task (see Section 4.4.2 and 8.2.3). But this solution is still too slow for the largest test case.

A simple optimization is to run BFS from the target permutation (sorted descending) to all other permutations **only once**, for all possible **N** in **[1..10]**. This solution has time complexity of roughly $O(10 \times N \times N! + T)$, much faster than before but still too slow for typical programming contest settings.

A better solution is a more sophisticated search technique called 'meet in the middle' (bidirectional BFS) to bring down the search space to a manageable level (see Section 8.2.4). First, we do some preliminary analysis (or we can also look at 'Pancake Number', `http://oeis.org/A058986`) to identify that for the largest test case when $N = 10$, we need *at most* 11 flips to sort any input stack to the sorted one. Therefore, we precalculate BFS from the target permutation to all other permutations for all $\mathbf{N} \in \mathbf{[1..10]}$, but stopping as soon as we reach depth $\lfloor \frac{11}{2} \rfloor = 5$. Then, for each test case, we run BFS from the starting permutation again with maximum depth 5. If we encounter a common vertex with the precalculated BFS from target permutation, we know that the answer is the distance from starting permutation to this vertex plus the distance from target permutation to this vertex. If we do not encounter a common vertex at all, we know that the answer should be the maximum flips: 11. On the largest test case with $N = 10$ for all test cases, this solution has time complexity of roughly $O((10+T) \times 10^5)$, which is now feasible.

Programming exercises related to Pancake Sorting:

1. **UVa 00120 - Stacks Of Flapjacks *** (pancake sorting, greedy version)
2. The Pancake Sorting problem as described in this section.

9.26 Pollard's rho Integer Factoring Algorithm

In Section 5.5.4, we have seen the optimized trial division algorithm that can be used to find the prime factors of integers up to $\approx 9 \times 10^{13}$ (see **Exercise 5.5.4.1**) in *contest environment* (i.e. in 'a few seconds' instead of minutes/hours/days). Now, what if we are given a 64-bit unsigned integer (i.e. up to $\approx 1 \times 10^{19}$) to be factored in contest environment?

For a *faster* integer factorization, one can use the Pollard's rho algorithm [52, 3]. The key idea of this algorithm is that two integers x and y are congruent modulo p (p is one of the factor of n—the integer that we want to factor) with probability 0.5 after 'a few ($1.177\sqrt{p}$) integers' have been randomly chosen.

The theoretical details of this algorithm is probably not that important for Competitive Programming. In this section, we directly provide a working C++ implementation below which can be used to handle composite integer that fit in 64-bit unsigned integers in contest environment. However, Pollard's rho cannot factor an integer n if n is a large prime due to the way the algorithm works. To handle this case, we have to implement a fast (probabilistic) prime testing like the Miller-Rabin's algorithm (see **Exercise 5.3.2.4***).

```
#define abs_val(a) (((a)>0)?(a):-(a))
typedef long long ll;

ll mulmod(ll a, ll b, ll c) { // returns (a * b) % c, and minimize overflow
  ll x = 0, y = a % c;
  while (b > 0) {
    if (b % 2 == 1) x = (x + y) % c;
    y = (y * 2) % c;
    b /= 2;
  }
  return x % c;
}

ll gcd(ll a,ll b) { return !b ? a : gcd(b, a % b); }        // standard gcd

ll pollard_rho(ll n) {
  int i = 0, k = 2;
  ll x = 3, y = 3;                // random seed = 3, other values possible
  while (1) {
    i++;
    x = (mulmod(x, x, n) + n - 1) % n;               // generating function
    ll d = gcd(abs_val(y - x), n);                       // the key insight
    if (d != 1 && d != n) return d;         // found one non-trivial factor
    if (i == k) y = x, k *= 2;
} }

int main() {
  ll n = 2063512844981574047LL;    // we assume that n is not a large prime
  ll ans = pollard_rho(n);          // break n into two non trivial factors
  if (ans > n / ans) ans = n / ans;          // make ans the smaller factor
  printf("%lld %lld\n", ans, n / ans);  // should be: 1112041493 1855607779
} // return 0;
```

We can also implement Pollard's rho algorithm in Java and use the `isProbablePrime` function in Java BigInteger class. This way, we can accept n larger than $2^{64}-1$, e.g. 17798655664295576020099, which is $\approx 2^{74}$, and factor it into $143054969437 \times 124418296927$. However, the runtime of Pollard's rho algorithm increases with larger n. The fact that integer factoring is a very difficult task is still the key concept of modern cryptography.

It is a good idea to test the complete implementation of Pollard's rho algorithm (that is, including the fast probabilistic prime testing algorithm and any other small details) to solve the following two programming exercise problems.

Source code: `Pollardsrho.cpp/java`

Programming exercises related to Pollard's rho algorithm:

1. ***UVa 11476 - Factoring Large(t) ... **** (see the discussion above)
2. POJ 1811 - Prime Test, see `http://poj.org/problem?id=1811`

9.27 Postfix Calculator and Conversion

Algebraic Expressions

There are three types of algebraic expressions: Infix (the natural way for human to write algebraic expressions), Prefix[24] (Polish notation), and Postfix (Reverse Polish notation). In Infix/Prefix/Postfix expressions, an operator is located (in the middle of)/before/after two operands, respectively. In Table 9.2, we show three Infix expressions, their corresponding Prefix/Postfix expressions, and their values.

Infix	Prefix	Postfix	Value
2 + 6 * 3	+ 2 * 6 3	2 6 3 * +	20
(2 + 6) * 3	* + 2 6 3	2 6 + 3 *	24
4 * (1 + 2 * (9 / 3) - 5)	* 4 - + 1 * 2 / 9 3 5	4 1 2 9 3 / * + 5 - *	8

Table 9.2: Examples of Infix, Prefix, and Postfix expressions

Postfix Calculator

Postfix expressions are more computationally efficient than Infix expressions. First, we do not need (complex) parentheses as the precedence rules are already embedded in the Postfix expression. Second, we can also compute partial results as soon as an operator is specified. These two features are not found in Infix expressions.

Postfix expression can be computed in $O(n)$ using Postfix calculator algorithm. Initially, we start with an empty stack. We read the expression from left to right, one token at a time. If we encounter an operand, we will push it to the stack. If we encounter an operator, we will pop the top two items of the stack, do the required operation, and then put the result back to the stack. Finally, when all tokens have been read, we return the top (the only item) of the stack as the final answer.

As each of the n tokens is only processed once and all stack operations are $O(1)$, this Postfix Calculator algorithm runs in $O(n)$.

An example of a Postfix calculation is shown in Table 9.3.

Postfix	Stack (bottom to top)	Remarks
4 1 2 9 3 / * + 5 - *	4 1 2 9 3	The first five tokens are operands
4 1 2 9 3 / * + 5 - *	4 1 2 3	Take 3 and 9, compute 9 / 3, push 3
4 1 2 9 3 / * + 5 - *	4 1 6	Take 3 and 2, compute 2 * 3, push 6
4 1 2 9 3 / * + 5 - *	4 7	Take 6 and 1, compute 1 + 6, push 7
4 1 2 9 3 / * + 5 - *	4 7 5	An operand
4 1 2 9 3 / * + 5 - *	4 7 5	Take 5 and 7, compute 7 - 5, push 2
4 1 2 9 3 / * + 5 - *	4 2	Take 2 and 4, compute 4 * 2, push 8
4 1 2 9 3 / * + 5 - *	8	Return 8 as the answer

Table 9.3: Example of a Postfix Calculation

Exercise 9.27.1*: What if we are given Prefix expressions instead?
How to evaluate a Prefix expression in $O(n)$?

[24]One programming language that uses this expression is Scheme.

Infix to Postfix Conversion

Knowing that Postfix expressions are more computationally efficient than Infix expressions, many compilers will convert Infix expressions in the source code (most programming languages use Infix expressions) into Postfix expressions. To use the efficient Postfix Calculator as shown earlier, we need to be able to convert Infix expressions into Postfix expressions efficiently. One of the possible algorithm is the 'Shunting yard' algorithm invented by Edsger Dijkstra (the inventor of Dijkstra's algorithm—see Section 4.4.3).

Shunting yard algorithm has similar flavor with Bracket Matching (see Section 9.4) and Postfix Calculator above. The algorithm also uses a stack, which is initially empty. We read the expression from left to right, one token at a time. If we encounter an operand, we will immediately output it. If we encounter an open bracket, we will push it to the stack. If we encounter a close bracket, we will output the topmost items of the stack until we encounter an open bracket (but we do not output the open bracket). If we encounter an operator, we will keep outputting and then popping the topmost item of the stack if it has greater than or equal precedence with this operator, or until we encounter an open bracket, then push this operator to the stack. At the end, we will keep outputting and then popping the topmost item of the stack until the stack is empty.

As each of the n tokens is only processed once and all stack operations are $O(1)$, this Shunting yard algorithm runs in $O(n)$.

An example of a Shunting yard algorithm execution is shown in Table 9.4.

Infix	Stack	Postfix	Remarks
$\underline{4} * (1 + 2 * (9 / 3) - 5)$		4	Immediately output
$4 \underline{*} (1 + 2 * (9 / 3) - 5)$	*	4	Put to stack
$4 * \underline{(} 1 + 2 * (9 / 3) - 5)$	* (	4	Put to stack
$4 * (\underline{1} + 2 * (9 / 3) - 5)$	* (	4 1	Immediately output
$4 * (1 \underline{+} 2 * (9 / 3) - 5)$	* (+	4 1	Put to stack
$4 * (1 + \underline{2} * (9 / 3) - 5)$	* (+	4 1 2	Immediately output
$4 * (1 + 2 \underline{*} (9 / 3) - 5)$	* (+ *	4 1 2	Put to stack
$4 * (1 + 2 * \underline{(} 9 / 3) - 5)$	* (+ * (	4 1 2	Put to stack
$4 * (1 + 2 * (\underline{9} / 3) - 5)$	* (+ * (	4 1 2 9	Immediately output
$4 * (1 + 2 * (9 \underline{/} 3) - 5)$	* (+ * (/	4 1 2 9	Put to stack
$4 * (1 + 2 * (9 / \underline{3}) - 5)$	* (+ * (/	4 1 2 9 3	Immediately output
$4 * (1 + 2 * (9 / 3 \underline{)} - 5)$	* (+ *	4 1 2 9 3 /	Only output '/'
$4 * (1 + 2 * (9 / 3) \underline{-} 5)$	* (-	4 1 2 9 3 / * +	Output '*' then '+'
$4 * (1 + 2 * (9 / 3) - \underline{5})$	* (-	4 1 2 9 3 / * + 5	Immediately output
$4 * (1 + 2 * (9 / 3) - 5 \underline{)}$	*	4 1 2 9 3 / * + 5 -	Only output '-'
$4 * (1 + 2 * (9 / 3) - 5)$		4 1 2 9 3 / * + 5 - *	Empty the stack

Table 9.4: Example of an Execution of Shunting yard Algorithm

Programming exercises related to Postfix expression:

1. **UVa 00727 - Equation *** (the classic Infix to Postfix conversion problem)

9.28 Roman Numerals

Problem Description

Roman Numerals is a number system used in ancient Rome. It is actually a Decimal number system but it uses a certain letters of the alphabet instead of digits `[0..9]` (described below), it is not positional, and it does not have a symbol for zero.

Roman Numerals have these 7 basic letters and its corresponding Decimal values: I=1, V=5, X=10, L=50, C=100, D=500, and M=1000. Roman Numerals also have the following letter pairs: IV=4, IX=9, XL=40, XC=90, CD=400, CM=900.

Programming problems involving Roman Numerals usually deal with the conversion from Arabic numerals (the Decimal number system that we normally use everyday) to Roman Numerals and vice versa. Such problems only appear very rarely in programming contests and such conversion can be derived on the spot by reading the problem statement.

Solution(s)

In this section, we provide one conversion library that we have used to solve several programming problems involving Roman Numerals. Although you can derive this conversion code easily, at least you do not have to debug[25] if you already have this library.

```
void AtoR(int A) {
  map<int, string> cvt;
  cvt[1000] = "M"; cvt[900] = "CM"; cvt[500] = "D"; cvt[400] = "CD";
  cvt[100]  = "C"; cvt[90]  = "XC"; cvt[50]  = "L"; cvt[40]  = "XL";
  cvt[10]   = "X"; cvt[9]   = "IX"; cvt[5]   = "V"; cvt[4]   = "IV";
  cvt[1]    = "I";
  // process from larger values to smaller values
  for (map<int, string>::reverse_iterator i = cvt.rbegin();
       i != cvt.rend(); i++)
    while (A >= i->first) {
      printf("%s", ((string)i->second).c_str());
      A -= i->first; }
  printf("\n");
}

void RtoA(char R[]) {
  map<char, int> RtoA;
  RtoA['I'] = 1;   RtoA['V'] = 5;   RtoA['X'] = 10;   RtoA['L'] = 50;
  RtoA['C'] = 100; RtoA['D'] = 500; RtoA['M'] = 1000;

  int value = 0;
  for (int i = 0; R[i]; i++)
    if (R[i+1] && RtoA[R[i]] < RtoA[R[i+1]]) {     // check next char first
      value += RtoA[R[i + 1]] - RtoA[R[i]];               // by definition
      i++; }                                              // skip this char
    else value += RtoA[R[i]];
  printf("%d\n", value);
}
```

[25]If the problem uses different standard of Roman Numerals, you may need to slightly edit our code.

Source code: `UVa11616.cpp/java`

Programming exercises related to Roman Numerals:

1. **UVa 00344 - Roman Digititis *** (count how many Roman characters are used to make all numbers from 1 to N)
2. UVa 00759 - The Return of the ... (Roman number + validity check)
3. **UVa 11616 - Roman Numerals *** (Roman numeral conversion problem)
4. ***UVa 12397 - Roman Numerals **** (conversion, each Roman digit has value)

9.29 Selection Problem

Problem Description

Selection problem is the problem of finding the k-th smallest[26] element of an array of n elements. Another name for selection problem is order statistics. Thus the minimum (smallest) element is the 1-st order statistic, the maximum (largest) element is the n-th order statistic, and the median element is the $\frac{n}{2}$ order statistic (there are 2 medians if n is even).

This selection problem is used as a motivating example in the opening of Chapter 3. In this section, we discuss this problem, its variants, and its various solutions in more details.

Solution(s)

Special Cases: $k = 1$ and $k = n$

Searching the minimum ($k = 1$) or maximum ($k = n$) element of an arbitrary array can be done in $\Omega(n - 1)$ comparisons: We set the first element to be the temporary answer, and then we compare this temporary answer with the other $n - 1$ elements one by one and keep the smaller (or larger, depending on the requirement) one. Finally, we report the answer. $\Omega(n - 1)$ comparisons is the lower bound, i.e. We cannot do better than this. While this problem is easy for $k = 1$ or $k = n$, finding the other order statistics—the general form of selection problem—is more difficult.

$O(n^2)$ algorithm, static data

A naïve algorithm to find the k-th smallest element is to this: Find the smallest element, 'discard' it (e.g. by setting it to a 'dummy large value'), and repeat this process k times. When k is near 1 (or when k is near n), this $O(kn)$ algorithm can still be treated as running in $O(n)$, i.e. we treat k as a 'small constant'. However, the worst case scenario is when we have to find the median ($k = \frac{n}{2}$) element where this algorithm runs in $O(\frac{n}{2} \times n) = O(n^2)$.

$O(n \log n)$ algorithm, static data

A better algorithm is to sort (that is, pre-process) the array first in $O(n \log n)$. Once the array is sorted, we can find the k-th smallest element in $O(1)$ by simply returning the content of index k-1 (0-based indexing) of the sorted array. The main part of this algorithm is the sorting phase. Assuming that we use a good $O(n \log n)$ sorting algorithm, this algorithm runs in $O(n \log n)$ overall.

Expected $O(n)$ algorithm, static data

An even better algorithm for the selection problem is to apply Divide and Conquer paradigm. The key idea of this algorithm is to use the $O(n)$ Partition algorithm (the randomized version) from Quick Sort as its sub-routine.

A randomized partition algorithm: `RandomizedPartition(A, l, r)` is an algorithm to partition a given range `[l..r]` of the array A around a (random) pivot. Pivot $A[p]$ is one of the element of A where $p \in$ `[l..r]`. After partition, all elements $\leq A[p]$ are placed before the pivot and all elements $> A[p]$ are placed after the pivot. The final index of the pivot q is returned. This randomized partition algorithm can be done in $O(n)$.

[26]Note that finding the k-th largest element is equivalent to finding the (n-k+1)-th smallest element.

After performing `q = RandomizedPartition(A, 0, n - 1)`, all elements $\leq A[q]$ will be placed before the pivot and therefore $A[q]$ is now in it's correct order statistic, which is $q+1$. Then, there are only 3 possibilities:

1. $q+1 = k$, $A[q]$ is the desired answer. We return this value and stop.
2. $q+1 > k$, the desired answer is inside the left partition, e.g. in $A[0..q\text{-}1]$.
3. $q+1 < k$, the desired answer is inside the right partition, e.g. in $A[q\text{+}1..n\text{-}1]$.

This process can be repeated recursively on smaller range of search space until we find the required answer. A snippet of C++ code that implements this algorithm is shown below.

```
int RandomizedSelect(int A[], int l, int r, int k) {
  if (l == r) return A[l];
  int q = RandomizedPartition(A, l, r);
       if (q + 1 == k) return A[q];
  else if (q + 1  > k) return RandomizedSelect(A, l, q - 1, k);
  else                 return RandomizedSelect(A, q + 1, r, k);
}
```

This `RandomizedSelect` algorithm runs in expected $O(n)$ time and very unlikely to run in its worst case $O(n^2)$ as it uses randomized pivot at each step. The full analysis involves probability and expected values. Interested readers are encouraged to read other references for the full analysis e.g. [7].

A simplified (but not rigorous) analysis is to assume `RandomizedSelect` divides the array into two at each step and n is a power of two. Therefore it runs `RandomizedPartition` in $O(n)$ for the first round, in $O(\frac{n}{2})$ in the second round, in $O(\frac{n}{4})$ in the third round and finally $O(1)$ in the $1 + \log_2 n$ round. The cost of `RandomizedSelect` is mainly determined by the cost of `RandomizedPartition` as all other steps of `RandomizedSelect` is $O(1)$. Therefore the overall cost is $O(n + \frac{n}{2} + \frac{n}{4} + ... + \frac{n}{n}) = O(n \times (\frac{1}{1} + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{n}) \leq O(2n) = O(n)$.

Library solution for the expected $O(n)$ algorithm, static data

C++ STL has function `nth_element` in `<algorithm>`. This `nth_element` implements the expected $O(n)$ algorithm as shown above. However as of 24 May 2013, we are not aware of Java equivalent for this function.

$O(n \log n)$ pre-processing, $O(\log n)$ algorithm, dynamic data

All solutions presented earlier assume that the given array is static—unchanged for each query of the k-th smallest element. However, if the content of the array is frequently modified, i.e. a new element is added, an existing element is removed, or the value of an existing element is changed, the solutions outlined above become inefficient.

When the underlying data is dynamic, we need to use a balanced Binary Search Tree (see Section 2.3). First, we insert all n elements into a balanced BST in $O(n \log n)$ time. We also augment (add information) about the size of each sub-tree rooted at each vertex. This way, we can find the k-th smallest element in $O(\log n)$ time by comparing k with q—the size of the left sub-tree of the root:

1. If $q + 1 = k$, then the root is the desired answer. We return this value and stop.
2. If $q + 1 > k$, the desired answer is inside the left sub-tree of the root.
3. If $q + 1 < k$, the desired answer is inside the right sub-tree of the root and we are now searching for the $(k-q-1)$-th smallest element in this right sub-tree. This adjustment of k is needed to ensure correctness.

This process—which is similar with the expected $O(n)$ algorithm for static selection problem—can be repeated recursively until we find the required answer. As checking the size of a sub-tree can be done in $O(1)$ if we have properly augment the BST, this overall algorithm runs at worst in $O(\log n)$ time, from root to the deepest leaf of a balanced BST.

However, as we need to augment a balanced BST, this algorithm cannot use built-in C++ STL <`map`>/<`set`> (or Java `TreeMap`/`TreeSet`) as these library code cannot be augmented. Therefore, we need to write our own balanced BST routine (e.g. AVL tree—see Figure 9.12—or Red Black Tree, etc—all of them take some time to code) and therefore such selection problem on *dynamic data* can be quite painful to solve.

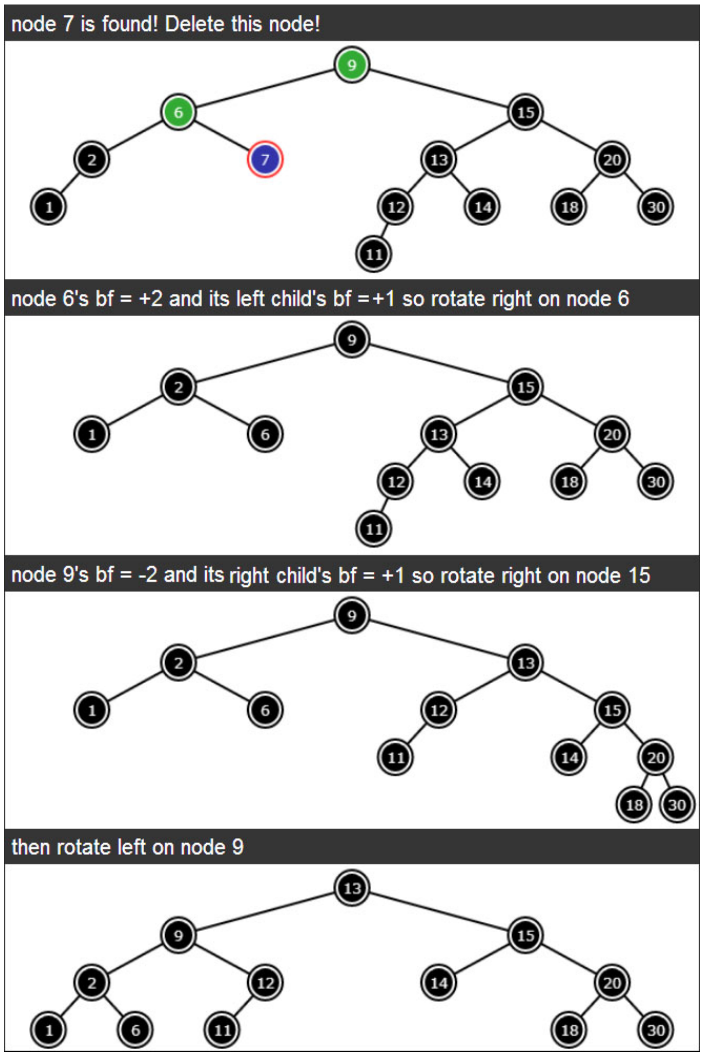
node 7 is found! Delete this node!
node 6's bf = +2 and its left child's bf = +1 so rotate right on node 6
node 9's bf = -2 and its right child's bf = +1 so rotate right on node 15
then rotate left on node 9

Figure 9.12: Example of an AVL Tree Deletion (Delete 7)

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

9.30 Shortest Path Faster Algorithm

Shortest Path Faster Algorithm (SPFA) is an algorithm that utilizes a queue to eliminate redundant operations in Bellman Ford's algorithm. This algorithm was published in Chinese by Duan Fanding in 1994. As of 2013, this algorithm is popular among Chinese programmers but it is not yet well known in other parts of the world.

SPFA requires the following data structures:

1. A graph stored in an Adjacency List: `AdjList` (see Section 2.4.1).
2. `vi dist` to record the distance from source to every vertex.
   (`vi` is our shortcut for `vector<int>`).
3. A `queue<int>` to stores the vertex to be processed.
4. `vi in_queue` to denote if a vertex is in the queue or not.

The first three data structures are the same as Dijkstra's or Bellman Ford's algorithms listed in Section 4.4. The fourth data structure is unique to SPFA. We can write SPFA as follows:

```
// inside int main()
  // initially, only S has dist = 0 and in the queue
  vi dist(n, INF); dist[S] = 0;
  queue<int> q; q.push(S);
  vi in_queue(n, 0); in_queue[S] = 1;

  while (!q.empty()) {
    int u = q.front(); q.pop(); in_queue[u] = 0;
    for (j = 0; j < (int)AdjList[u].size(); j++) {    // all neighbors of u
      int v = AdjList[u][j].first, weight_u_v = AdjList[u][j].second;
      if (dist[u] + weight_u_v < dist[v]) {                // if can relax
        dist[v] = dist[u] + weight_u_v;                           // relax
        if (!in_queue[v]) {                               // add to the queue
          q.push(v);              // only if it is not already in the queue
          in_queue[v] = 1;
  } } } }
```

Source code: `UVa10986.cpp/java`

This algorithm runs in $O(kE)$ where k is a number depending on the graph. The maximum k can be V (which is the same as the time complexity of Bellman Ford's). However, we have tested that for most SSSP problems in UVa online judge that are listed in this book, SPFA (which uses a queue) is as fast as Dijkstra's (which uses a priority queue).

SPFA can deal with negative weight edge. If the graph has no negative cycle, SPFA runs well on it. If the graph has negative cycle(s), SPFA can also detect it as there must be some vertex (those on the negative cycle) that enters the queue for over $V - 1$ times. We can modify the given code above to record the time each vertex enters the queue. If we find that any vertex enters the queue more than $V - 1$ times, we can conclude that the graph has negative cycle(s).

9.31 Sliding Window

Problem Description

There are several variants of Sliding Window problems. But all of them have similar basic idea: 'Slide' a sub-array (that we call a 'window', which can have static or dynamic length) in linear fashion from left to right over the original array of n elements in order to compute something. Some of the variants are:

1. Find the smallest sub-array size (smallest window length) so that the sum of the sub-array is greater than or equal to a certain constant S in $O(n)$? Examples:
   For array $A_1 = \{5, 1, 3, \underline{[5, 10]}, 7, 4, 9, 2, 8\}$ and $S = 15$, the answer is 2 as highlighted.
   For array $A_2 = \{1, 2, \underline{[3, 4, 5]}\}$ and $S = 11$, the answer is 3 as highlighted.

2. Find the smallest sub-array size (smallest window length) so that the elements inside the sub-array contains all integers in range `[1..K]`. Examples:
   For array $A = \{1, \underline{[2, 3, 7, 1, 12, 9, 11, 9, 6, 3, 7, 5, 4]}, 5, 3, 1, 10, 3, 3\}$ and $K = 4$, the answer is 13 as highlighted.
   For the same array $A = \{\underline{[1, 2, 3]}, 7, 1, 12, 9, 11, 9, 6, 3, 7, 5, 4, 5, 3, 1, 10, 3, 3\}$ and $K = 3$, the answer is 3 as highlighted.

3. Find the maximum sum of a certain sub-array with (static) size K. Examples:
   For array $A_1 = \{10, \underline{[50, 30, 20]}, 5, 1\}$ and $K = 3$, the answer is 100 by summing the highlighted sub-array.
   For array $A_2 = \{49, 70, 48, \underline{[61, 60]}, 60\}$ and $K = 2$, the answer is 121 by summing the highlighted sub-array.

4. Find the minimum of *each* possible sub-arrays with (static) size K. Example:
   For array $A = \{0, 5, 5, 3, 10, 0, 4\}$, $n = 7$, and $K = 3$, there are $n-K+1 = 7-3+1 = 5$ possible sub-arrays with size $K = 3$, i.e. $\{0, 5, 5\}$, $\{5, 5, 3\}$, $\{5, 3, 10\}$, $\{3, 10, 0\}$, and $\{10, 0, 4\}$. The minimum of each sub-array is 0, 3, 3, 0, 0, respectively.

Solution(s)

We ignore the discussion of naïve solutions for these Sliding Window variants and go straight to the $O(n)$ solutions to save space. The four solutions below run in $O(n)$ as what we do is to 'slide' a window over the original array of n elements—some with clever tricks.

For variant number 1, we maintain a window that keeps growing (append the current element to the back—the right side—of the window) and add the value of the current element to a running sum or keeps shrinking (remove the front—the left side—of the window) as long as the running sum is $\geq S$. We keep the smallest window length throughout the process and report the answer.

For variant number 2, we maintain a window that keeps growing if range `[1..K]` is not yet covered by the elements of the current window or keeps shrinking otherwise. We keep the smallest window length throughout the process and report the answer. The check whether range `[1..K]` is covered or not can be simplified using a kind of frequency counting. When all integers $\in$ `[1..K]` has non zero frequency, we said that range `[1..K]` is covered. Growing the window increases a frequency of a certain integer that may cause range `[1..K]` to be fully covered (it has no 'hole') whereas shrinking the window decreases a frequency of the removed integer and if the frequency of that integer drops to 0, the previously covered range `[1..K]` is now no longer covered (it has a 'hole').

For variant number 3, we insert the first K integers into the window, compute its sum, and declare the sum as the current maximum. Then we slide the window to the right by adding one element to the right side of the window and removing one element from the left side of the window—thereby maintaining window length to K. We add the sum by the value of the added element minus the value of the removed element and compare with the current maximum sum to see if this sum is the new maximum sum. We repeat this window-sliding process $n - K$ times and report the maximum sum found.

Variant number 4 is quite challenging especially if n is large. To get $O(n)$ solution, we need to use a `deque` (double-ended queue) data structure to model the window. This is because `deque` supports efficient—$O(1)$—insertion and deletion from front and back of the queue (see discussion of `deque` in Section 2.2). This time, we maintain that the window (that is, the `deque`) is sorted in ascending order, that is, the front most element of the `deque` has the minimum value. However, this changes the ordering of elements in the array. To keep track of whether an element is currently still inside the current window or not, we need to remember the index of each element too. The detailed actions are best explained with the C++ code below. This sorted window can shrink from both sides (back and front) and can grow from back, thus necessitating the usage of `deque`[27] data structure.

```
void SlidingWindow(int A[], int n, int K) {
  // ii---or pair<int, int>---represents the pair (A[i], i)
  deque<ii> window; // we maintain ‘window’ to be sorted in ascending order
  for (int i = 0; i < n; i++) {                             // this is O(n)
    while (!window.empty() && window.back().first >= A[i])
      window.pop_back();            // this to keep ‘window’ always sorted

    window.push_back(ii(A[i], i));

    // use the second field to see if this is part of the current window
    while (window.front().second <= i - K)                // lazy deletion
      window.pop_front();

    if (i + 1 >= K)           // from the first window of length K onwards
      printf("%d\n", window.front().first);  // the answer for this window
} }
```

Programming exercises:

1. ***UVa 01121 - Subsequence **** (sliding window variant no 1)
2. ***UVa 11536 - Smallest Sub-Array **** (sliding window variant no 2)
3. IOI 2011 - Hottest (practice task; sliding window variant no 3)
4. IOI 2011 - Ricehub (sliding window++)
5. IOI 2012 - Tourist Plan (practice task; another sliding window variant; the best answer starting from city 0 and ending at city $i \in [0..N\text{-}1]$ is the sum of happiness of the top $K\text{-}i$ cities $\in [0..i]$; use `priority_queue`; output the highest sum)

[27]Note that we do not actually need to use `deque` data structure for variant 1-3 above.

9.32 Sorting in Linear Time

Problem Description

Given an (unsorted) array of n elements, can we sort them in $O(n)$ time?

Theoretical Limit

In general case, the lower bound of generic—comparison-based—sorting algorithm is $\Omega(n \log n)$ (see the proof using decision tree model in other references, e.g. [7]). However, if there is a special property about the n elements, we can have a faster, linear, $O(n)$ sorting algorithm by *not* doing comparison between elements. We will see two examples below.

Solution(s)

Counting Sort

If the array `A` contains n integers with *small* range `[L..R]` (e.g. 'human age' of `[1..99]` years in UVa 11462 - Age Sort), we can use the Counting Sort algorithm. For the explanation below, assume that array `A` is {`2, 5, 2, 2, 3, 3`}. The idea of Counting Sort is as follows:

1. Prepare a 'frequency array' `f` with size `k = R-L+1` and initialize `f` with zeroes.

   On the example array above, we have `L = 2`, `R = 5`, and `k = 4`.

2. We do one pass through array `A` and update the frequency of each integer that we see, i.e. for each `i` $\in$ `[0..n-1]`, we do `f[A[i]-L]++`.

   On the example array above, we have `f[0] = 3, f[1] = 2, f[2] = 0, f[3] = 1`.

3. Once we know the frequency of each integers in that small range,
   we compute the prefix sums of each `i`, i.e. `f[i] = [f-1] + f[i]` $\forall$`i` $\in$ `[1..k-1]`.
   Now, `f[i]` contains the number of elements less than or equal to `i`.

   On the example array above, we have `f[0] = 3, f[1] = 5, f[2] = 5, f[3] = 6`.

4. Next, go backwards from `i = n-1` down to `i = 0`.
   We place `A[i]` at index `f[A[i]-L]-1` as it is the correct location for `A[i]`.
   We decrement `f[A[i]-L]` by one so that the next copy of `A[i]`—if any—will be placed right before the current `A[i]`.

   On the example array above, we first put `A[5] = 3` in index `f[A[5]-2]-1 = f[1]-1 = 5-1 = 4` and decrement `f[1]` to 4.
   Next, we put `A[4] = 3`—the same value as `A[5] = 3`—now in index `f[A[4]-2]-1 = f[1]-1 = 4-1 = 3` and decrement `f[1]` to 3.
   Then, we put `A[3] = 2` in index `f[A[3]-2]-1 = 2` and decrement `f[0]` to 2.
   We repeat the next three steps until we obtain a sorted array: {`2, 2, 2, 3, 3, 5`}.

The time complexity of Counting Sort is $O(n+k)$. When $k = O(n)$, this algorithm theoretically runs in linear time by *not* doing comparison of the integers. However, in programming contest environment, usually k cannot be too large in order to avoid Memory Limit Exceeded. For example, Counting Sort will have problem sorting this array `A` with $n = 3$ that contains {`1, 1000000000, 2`} as it has large k.

Radix Sort

If the array `A` contains n non-negative integers with relatively wide range `[L..R]` but it has relatively small number of digits, we can use the Radix Sort algorithm.

The idea of Radix Sort is simple. First, we make all integers have d digits—where d is the largest number of digits in the largest integer in `A`—by appending zeroes if necessary. Then, Radix Sort will sort these numbers digit by digit, starting with the *least* significant digit to the *most* significant digit. It uses another *stable sort* algorithm as a sub-routine to sort the digits, such as the $O(n + k)$ Counting Sort shown above. For example:

```
Input  | Append | Sort by the  | Sort by the | Sort by the  | Sort by the
d = 4  | Zeroes | fourth digit | third digit | second digit | first digit
 323   | 0323   | 032(2)       | 00(1)3      | 0(0)13       | (0)013
1257   | 1257   | 032(3)       | 03(2)2      | 1(2)57       | (0)322
  13   | 0013   | 001(3)       | 03(2)3      | 0(3)22       | (0)323
 322   | 0322   | 125(7)       | 12(5)7      | 0(3)23       | (1)257
```

For an array of n d-digits integers, we will do an $O(d)$ passes of Counting Sorts which have time complexity of $O(n + k)$ each. Therefore, the time complexity of Radix Sort is $O(d \times (n + k))$. If we use Radix Sort for sorting n 32-bit signed integers ($\approx d = 10$ digits) and k = 10. This Radix Sort algorithm runs in $O(10 \times (n + 10))$. It can still be considered as running in linear time but it has high constant factor.

Considering the hassle of writing the complex Radix Sort routine compared to calling the standard $O(n \log n)$ C++ STL `sort` (or Java `Collections.sort`), this Radix Sort algorithm is rarely used in programming contests. In this book, we only use this combination of Radix Sort and Counting Sort in our Suffix Array implementation (see Section 6.6.4).

Exercise 9.32.1*: What should we do if we want to use Radix Sort but the array `A` contains (at least one) negative number(s)?

Programming exercises related to Sorting in Linear Time:

1. **UVa 11462 - Age Sort *** (standard Counting Sort problem)

9.33 Sparse Table Data Structure

In Section 2.4.3, we have seen that Segment Tree data structure can be used to solve the Range Minimum Query (RMQ) problem—the problem of finding the index that has the minimum element within a range `[i..j]` of the underlying array `A`. It takes $O(n)$ pre-processing time to build the Segment Tree, and once the Segment Tree is ready, each RMQ is just $O(\log n)$. With Segment Tree, we can deal with the *dynamic version* of this RMQ problem, i.e. when the underlying array is updated, we usually only need $O(\log n)$ to update the corresponding Segment Tree structure.

However, some problems involving RMQ never change the underlying array `A` after the first query. This is called the *static* RMQ problem. Although Segment Tree obviously can be used to deal with the static RMQ problem, this static version has an alternative DP solution with $O(n \log n)$ pre-processing time and $O(1)$ per RMQ. One such example is the Lowest Common Ancestor (LCA) problem in Section 9.18.

The key idea of the DP solution is to split `A` into sub arrays of length 2^j for each non-negative integer j such that $2^j \leq n$. We will keep an array `SpT` of size $n \times \log n$ where `SpT[i][j]` stores the index of the minimum value in the sub array starting at index `i` and having length 2^j. This array `SpT` will be sparse as not all of its cells have values (hence the name 'Sparse Table'). We use an abbreviation `SpT` to differentiate this data structure from Segment Tree (`ST`).

To build up the `SpT` array, we use a technique similar to the one used in many Divide and Conquer algorithms such as merge sort. We know that in an array of length 1, the single element is the smallest one. This is our base case. To find out the index of the smallest element in an array of size 2^j, we can compare the values at the indices of the smallest elements in the two distinct sub arrays of size 2^{j-1} and take the index of the smallest element of the two. It takes $O(n \log n)$ time to build up the `SpT` array like this. Please scrutinize the constructor of class `RMQ` shown in the source code below that implements this `SpT` array construction.

It is simple to understand how we would process a query if the length of the range were a power of 2. Since this is exactly the information `SpT` stores, we would just return the corresponding entry in the array. However, in order to compute the result of a query with arbitrary start and end indices, we have to fetch the entry for two smaller sub arrays within this range and take the minimum of the two. Note that these two sub arrays might have to overlap, the point is that we want cover the entire range with two sub arrays and nothing outside of it. This is always possible even if the length of the sub arrays have to be a power of 2. First, we find the length of the query range, which is `j-i+1`. Then, we apply $\log_2$ on it and round down the result, i.e. `k` = $\lfloor \log_2$`(j-i+1)`$\rfloor$. This way, $2^k \leq$ `(j-i+1)`. This simple Figure 9.13 below shows what the two sub arrays might look like. As there is a potentially overlapping sub-problems, this part of the solution is classified as Dynamic Programming.

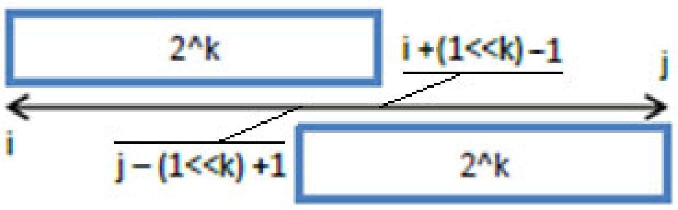

Figure 9.13: Explanation of `RMQ(i, j)`

An example implementation of Sparse Table to solve the static RMQ problem is shown below. You can compare this version with the Segment Tree version shown in Section 2.4.3.

```
#define MAX_N 1000                            // adjust this value as needed
#define LOG_TWO_N 10              // 2^10 > 1000, adjust this value as needed

class RMQ {                                               // Range Minimum Query
private:
  int _A[MAX_N], SpT[MAX_N][LOG_TWO_N];
public:
  RMQ(int n, int A[]) {    // constructor as well as pre-processing routine
    for (int i = 0; i < n; i++) {
      _A[i] = A[i];
      SpT[i][0] = i; // RMQ of sub array starting at index i + length 2^0=1
    }
    // the two nested loops below have overall time complexity = O(n log n)
    for (int j = 1; (1<<j) <= n; j++) // for each j s.t. 2^j <= n, O(log n)
      for (int i = 0; i + (1<<j) - 1 < n; i++)    // for each valid i, O(n)
        if (_A[SpT[i][j-1]] < _A[SpT[i+(1<<(j-1))][j-1]])            // RMQ
          SpT[i][j] = SpT[i][j-1];    // start at index i of length 2^(j-1)
        else                  // start at index i+2^(j-1) of length 2^(j-1)
          SpT[i][j] = SpT[i+(1<<(j-1))][j-1];
  }

  int query(int i, int j) {                              // this query is O(1)
    int k = (int)floor(log((double)j-i+1) / log(2.0));    // 2^k <= (j-i+1)
    if (_A[SpT[i][k]] <= _A[SpT[j-(1<<k)+1][k]]) return SpT[i][k];
    else                                         return SpT[j-(1<<k)+1][k];
} };
```

Source code: `SparseTable.cpp/java`

For the same test case with $n = 7$ and $A = \{18, 17, 13, 19, 15, 11, 20\}$ as in Section 2.4.3, the content of the sparse table `SpT` is as follows:

index	0	1	2
0	0	1	2
1	1	2	2
2	2	2	5
3	3	4	5
4	4	5	empty
5	5	5	empty
6	6	empty	empty

In the first column, we have $j = 0$ that denotes the RMQ of sub array starting at index i with length $2^0 = 1$, we have `SpT[i][j] = i`.

In the second column, we have $j = 1$ that denotes the RMQ of sub array starting at index i with length $2^1 = 2$. Notice that the last row is empty.

In the third column, we have $j = 2$ that denotes the RMQ of sub array starting at index i with length $2^2 = 4$. Notice that the last three rows is empty.

9.34 Tower of Hanoi

Problem Description

The classic description of the problem is as follows: There are three pegs: A, B, and C, as well as n discs, will all discs having different sizes. Starting with all the discs stacked in ascending order on one peg (peg A), your task is to move all n discs to another peg (peg C). No disc may be placed on top of a disc smaller than itself, and only one disc can be moved at a time, from the top of one peg to another.

Solution(s)

There exists a simple recursive backtracking solution for the classic Tower of Hanoi problem. The problem of moving n discs from peg A to peg C with additional peg B as intermediate peg can be broken up into the following sub-problems:

1. Move $n - 1$ discs from peg A to peg B using peg C as the intermediate peg.
   After this recursive step is done, we are left with disc n by itself in peg A.
2. Move disc n from peg A to peg C.
3. Move $n - 1$ discs from peg B to peg C using peg A as the intermediate peg.
   These $n - 1$ discs will be on top of disc n which is now at the bottom of peg C.

Note that step 1 and step 3 above are recursive steps. The base case is when $n = 1$ where we simply move a single disc from the current source peg to its destination peg, bypassing the intermediate peg. A sample C++ implementation code is shown below:

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

void solve(int count, char source, char destination, char intermediate) {
  if (count == 1)
    printf("Move top disc from pole %c to pole %c\n", source, destination);
  else {
    solve(count-1, source, intermediate, destination);
    solve(1, source, destination, intermediate);
    solve(count-1, intermediate, destination, source);
  }
}

int main() {
  solve(3, 'A', 'C', 'B');      // try larger value for the first parameter
} // return 0;
```

The minimum number of moves required to solve a classic Tower of Hanoi puzzle of n discs using this recursive backtracking solution is $2^n - 1$ moves.

Programming exercises related to Tower of Hanoi:

1. **UVa 10017 - The Never Ending ... *** (classical problem)

9.35 Chapter Notes

As of 24 May 2013, Chapter 9 contains 34 rare topics. 10 of them are rare algorithms (highlighted in **bold**). The other 24 are rare problems.

2-SAT Problem	Art Gallery Problem
Bitonic Traveling Salesman Problem	Bracket Matching
Chinese Postman Problem	Closest Pair Problem
Dinic's Algorithm	**Formulas or Theorems**
Gaussian Elimination Algorithm	Graph Matching
Great-Circle Distance	**Hopcroft Karp's Algorithm**
Independent and Edge-Disjoint Paths	Inversion Index
Josephus Problem	Knight Moves
Kosaraju's Algorithm	Lowest Common Ancestor
Magic Square Construction (Odd Size)	Matrix Chain Multiplication
Matrix Power	Max Weighted Independent Set
Min Cost (Max) Flow	Min Path Cover on DAG
Pancake Sorting	**Pollard's rho Integer Factoring Algorithm**
Postfix Calculator and Conversion	Roman Numerals
Selection Problem	Shortest Path Faster Algorithm
Sliding Window	Sorting in Linear Time
Sparse Table Data Structure	Tower of Hanoi

However, after writing so much in the third edition of this book, we become more aware that there are many other Computer Science topics that we have not covered yet.

We close this chapter—and the third edition of this book—by listing down quite a good number of topic keywords that are eventually not included in the third edition of this book due to our-own self-imposed 'writing time limit' of 24 May 2013.

There are many other exotic data structures that are rarely used in programming contests: Fibonacci heap, various hashing techniques (hash tables), heavy-light decomposition of a rooted tree, interval tree, k-d tree, linked list (we purposely avoid this one in this book), radix tree, range tree, skip list, treap, etc.

The topic of Network Flow is much bigger than what we have wrote in Section 4.6 and the several sections in this chapter. Other topics like the Baseball Elimination problem, Circulation problem, Gomory-Hu tree, Push-relabel algorithm, Stoer-Wagner's min cut algorithm, and the rarely known Suurballe's algorithm can be added.

We can add more detailed discussions on a few more algorithms in Section 9.10, namely: Edmonds's Matching algorithm [13], Gale Shapley's algorithm for Stable Marriage problem, and Kuhn Munkres's (Hungarian) algorithm [39, 45].

There are many other mathematics problems and algorithms that can be added, e.g. the Chinese Remainder Theorem, modular multiplicative inverse, Möbius function, several exotic Number Theoretic problems, various numerical methods, etc.

In Section 6.4 and in Section 6.6, we have seen the KMP and Suffix Tree/Array solutions for the String Matching problem. String Matching is a well studied topic and other algorithms exist, like Aho Corasick's, Boyer Moore's, and Rabin Karp's.

In Section 8.2, we have seen several more advanced search techniques. Some programming contest problems are NP-hard (or NP-complete) problems but with small input size. The solution for these problems is usually a creative complete search. We have discussed several NP-hard/NP-complete problems in this book, but we can add more, e.g. Graph Coloring problem, Max Clique problem, Traveling Purchaser problem, etc.

Finally, we list down many other potential topic keywords that can possibly be included in the future editions of this book in alphabetical order, e.g. Burrows-Wheeler Transformation, Chu-Liu Edmonds's Algorithm, Huffman Coding, Karp's minimum mean-weight cycle algorithm, Linear Programming techniques, Malfatti circles, Min Circle Cover problem, Min Diameter Spanning Tree, Min Spanning Tree with one vertex with degree constraint, other computational geometry libraries that are not covered in Chapter 7, Optimal Binary Search Tree to illustrate the Knuth-Yao DP speedup [2], Rotating Calipers algorithm, Shortest Common Superstring problem, Steiner Tree problem, ternary search, Triomino puzzle, etc.

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	-	-	58
Written Exercises	-	-	15*
Programming Exercises	-	-	80