8.3.2 Compilation of Common (DP) Parameters

After solving lots of DP problems (including recursive backtracking without memoization), contestants will develop a sense of which parameters are commonly selected to represent the states of the DP (or recursive backtracking) problems. Some of them are as follows:

1. Parameter: Index i in an array, e.g. `[`x_0`, `x_1`, ..., `x_i`, ...]`
   Transition: Extend subarray $[0..i]$ (or $[i..n$-$1]$), process i, take item i or not, etc
   Example: 1D Max Sum, LIS, part of 0-1 Knapsack, TSP, etc (see Section 3.5.2)

2. Parameter: Indices (i, j) in two arrays, e.g. `[`x_0`, `x_1`, ..., `x_i`]` + `[`y_0`, `y_1`, ..., `y_j`]`
   Transition: Extend i, j, or both, etc
   Example: String Alignment/Edit Distance, LCS, etc (see Section 6.5)

3. Parameter: Subarray (i, j) of an array `[..., `x_i`, `x_{i+1}`, ..., `x_j`, ...]`
   Transition: Split (i, j) into $(i, k) + (k + 1, j)$ or into $(i, i + k) + (i + k + 1, j)$, etc
   Example: Matrix Chain Multiplication (see Section 9.20), etc

4. Parameter: A vertex (position) in a (usually implicit) DAG
   Transition: Process the neighbors of this vertex, etc
   Example: Shortest/Longest/Counting Paths in/on DAG, etc (Section 4.7.1)

5. Parameter: Knapsack-Style Parameter
   Transition: Decrease (or increase) current value until zero (or until threshold), etc
   Example: 0-1 Knapsack, Subset Sum, Coin Change variants, etc (see Section 3.5.2)
   Note: This parameter is not DP friendly if its range is very high.
   See tips in Section 8.3.3 if the value of this parameter can go negative.

6. Parameter: Small set (usually using bitmask technique)
   Transition: Flag one (or more) item(s) in the set to on (or off), etc
   Example: DP-TSP (see Section 3.5.2), DP with bitmask (see Section 8.3.1), etc

Note that the harder DP problems usually combine two or more parameters to represent distinct states. Try to solve more DP problems listed in this section to build your DP skills.

8.3.3 Handling Negative Parameter Values with Offset Technique

In rare cases, the possible range of a parameter used in a DP state can go negative. This causes an issue for DP solution as we map parameter value into index of a DP table. The indices of a DP table must therefore be non negative. Fortunately, this issue can be dealt easily by using offset technique to make all the indices become non negative again. We illustrate this technique with another non trivial DP problem: Free Parentheses.

UVa 1238 - Free Parentheses (ACM ICPC Jakarta08, LA 4143)

Abridged problem statement: You are given a simple arithmetic expression which consists of only *addition and subtraction* operators, i.e. `1 - 2 + 3 - 4 - 5`. You are free to put any *parentheses* to the expression anywhere and as many as you want as long as the expression is still *valid*. How many *different* numbers can you make? The answer for the simple expression above is 6:

```
1 - 2 + 3 - 4 - 5     =  -7          1 - (2 + 3 - 4 - 5)   =   5
1 - (2 + 3) - 4 - 5   = -13          1 - 2 + 3 - (4 - 5)   =   3
1 - (2 + 3 - 4) - 5   =  -5          1 - (2 + 3) - (4 - 5) =  -3
```

The problem specifies the following constraints: The expression consists of only $2 \leq N \leq 30$ non-negative numbers less than 100, separated by addition or subtraction operators. There is no operator before the first and after the last number.

To solve this problem, we need to make three observations:

1. We only need to put an open bracket after a '-' (negative) sign as doing so will reverse the meaning of subsequent '+' and '-' operators;
2. We can only put X close brackets if we already use X open brackets—we need to store this information to process the subproblems correctly;
3. The maximum value is 100 + 100 + ... + 100 (100 repeated 30 times) = 3000 and the minimum value is 0 - 100 - ... - 100 (one 0 followed by 29 times of negative 100) = -2900—this information also need to be stored, as we will see below.

To solve this problem using DP, we need to determine which set of parameters of this problem represent distinct states. The DP parameters that are easier to identify are these two:

1. '`idx`'—the current position being processed, we need to know where we are now.
2. '`open`'—the number of open brackets so that we can produce a valid expression[4].

But these two parameters are not enough to uniquely identify the state yet. For example, this partial expression: '1-1+1-1...' has `idx = 3` (indices: 0, 1, 2, 3 have been processed), `open = 0` (cannot put close bracket anymore), which sums to 0. Then, '1-(1+1-1)...' also has the same `idx = 3`, `open = 0` and sums to 0. But '1-(1+1)-1...' has the same `idx = 3`, `open = 0`, *but* sums to -2. These two DP parameters does *not* identify unique state yet. We need one more parameter to distinguish them, i.e. the value '`val`'. This skill of identifying the correct set of parameters to represent distinct states is something that one has to develop in order to do well with DP problems. The code and its explanation are shown below:

```
void rec(int idx, int open, int val) {
  if (visited[idx][open][val+3000])   // this state has been reached before
    return;   // the +3000 trick to convert negative indices to [200..6000]
          // negative indices are not friendly for accessing a static array
  visited[idx][open][val+3000] = true;      // set this state to be reached

  if (idx == N)        // last number, current value is one of the possible
    used[val+3000] = true, return;                    // result of expression

  int nval = val + num[idx] * sig[idx] * ((open % 2 == 0) ? 1 : -1);
  if (sig[idx] == -1)       // option 1: put open bracket only if sign is -
    rec(idx + 1, open + 1, nval);                   // no effect if sign is +
  if (open > 0)            // option 2: put close bracket, can only do this
    rec(idx + 1, open - 1, nval);  // if we already have some open brackets
  rec(idx + 1, open, nval);                   // option 3: normal, do nothing
}
```

[4]At `idx = N` (we have processed the last number), it is fine if we still have `open` > 0 as we can dump all the necessary closing brackets at the end of the expression, e.g.: 1 - (2 + 3 - (4 - (5$\underline{)))}$.

```
// Preprocessing: Set a Boolean array 'used' which is initially set to all
// false, then run this top-down DP by calling rec(0, 0, 0)
// The solution is the # of values in array 'used' that are flagged as true
```

As we can see from the code above, we can represent all possible states of this problem with a 3D array: `bool visited[idx][open][val]`. The purpose of this memo table `visited` is to flag if certain state has been visited or not. As '`val`' ranges from -2900 to 3000 (5901 distinct values), we have to offset these range to make the range non-negative. In this example, we use a safe constant +3000. The number of states is $30 \times 30 \times 6001 \approx 5M$ with $O(1)$ processing per state. This is fast enough.

8.3.4 MLE? Consider Using Balanced BST as Memo Table

In Section 3.5.2, we have seen a DP problem: 0-1 Knapsack where the state is $(id, remW)$. Parameter id has range [0..n-1] and parameter $remW$ has range [0..S]. If the problem author sets $n \times S$ to be quite large, it will cause the 2D array (for the DP table) of size $n \times S$ to be too large (Memory Limit Exceeded in programming contests).

Fortunately for problem like this 0-1 Knapsack, if we run the Top-Down DP on it, we will realize that not all of the states are visited (whereas the Bottom-Up DP version will have to explore all states). Therefore, we can trade runtime for smaller space by using a balanced BST (C++ STL `map` or Java `TreeMap`) as the memo table. This balanced BST will *only* record the states that are actually visited by the Top-Down DP. Thus, if there are only k visited states, we will only use $O(k)$ space instead of $n \times S$. The runtime of the Top-Down DP increases by $O(c \times \log k)$ factor. However, note that this trick is rarely useful due to the high constant factor c involved.

8.3.5 MLE/TLE? Use Better State Representation

Our 'correct' DP solution (which produces correct answer but using more computing resources) may be given Memory Limit Exceeded (MLE) or Time Limit Exceeded (TLE) verdict if the problem author used a better state representation and set larger input constraints that break our 'correct' DP solution. If that happens, we have no choice but to find a better DP state representation in order to reduce the DP table size (and subsequently speed up the overall time complexity). We illustrate this technique using an example:

UVa 1231 - ACORN (ACM ICPC Singapore07, LA 4106)

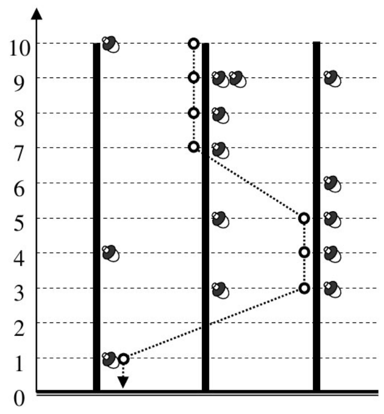

Figure 8.11: The Descent Path

Abridged problem statement: Given t oak trees, the height h of *all* trees, the height f that Jayjay the squirrel loses when it flies from one tree to another, $1 \leq t, h \leq 2000$, $1 \leq f \leq 500$, and the positions of acorns on each of the oak trees: `acorn[tree][height]`, determine the max number of acorns that Jayjay can collect in *one single descent*. Example: if $t = 3, h = 10, f = 2$ and `acorn[tree][height]` as shown in Figure 8.11, the best descent path has a total of 8 acorns (see the dotted line).

Naïve DP Solution: Use a table `total[tree][height]` that stores the best possible acorns collected when Jayjay is on a certain tree at certain height. Then Jayjay recursively tries to either go down (-1) unit on the *same* oak tree or flies (-f) unit(s) to $t-1$ *other* oak trees from this position. On the largest test case, this requires $2000 \times 2000 = 4M$ states and $4M \times 2000 = 8B$ operations. This approach is clearly TLE.

Better DP Solution: We can actually ignore the information: "On which tree Jayjay is currently at" as just memoizing the best among them is sufficient. This is because flying to any other $t-1$ other oak trees decreases Jayjay's height in the same manner. Set a table: `dp[height]` that stores the best possible acorns collected when Jayjay is at this `height`. The bottom-up DP code that requires only $2000 = 2K$ states and time complexity of $2000 \times 2000 = 4M$ is shown below:

```
for (int tree = 0; tree < t; tree++)                          // initialization
  dp[h] = max(dp[h], acorn[tree][h]);

for (int height = h - 1; height >= 0; height--)
  for (int tree = 0; tree < t; tree++) {
    acorn[tree][height] +=
      max(acorn[tree][height + 1],                 // from this tree, +1 above
      ((height + f <= h) ? dp[height + f] : 0)); // from tree at height + f
    dp[height] = max(dp[height], acorn[tree][height]);   // update this too
  }

printf("%d\n", dp[0]);                            // the solution is stored here
```

Source code: `ch8_03_UVa1231.cpp/java`

When the size of naïve DP states are too large that causes the overall DP time complexity to be not-doable, think of another more efficient (but usually not obvious) way to represent the possible states. Using a good state representation is a potential major speed up for a DP solution. Remember that no programming contest problem is unsolvable, the problem author must have known a trick.

8.3.6 MLE/TLE? Drop One Parameter, Recover It from Others

Another known trick to reduce the memory usage of a DP solution (and thereby speed up the solution) is to drop one important parameter which can be recovered by using the other parameter(s). We use one ACM ICPC World Finals problem to illustrate this technique.

UVa 1099 - Sharing Chocolate (ACM ICPC World Finals Harbin10, LA 4794)

Abridged problem description: Given a big chocolate bar of size $1 \leq w, h \leq 100$, $1 \leq n \leq 15$ friends, and the size request of each friend. Can we break the chocolate by using horizontal and vertical cuts so that each friend gets *one piece* of chocolate bar of his chosen size?

For example, see Figure 8.12 (left). The size of the original chocolate bar is $w = 4$ and $h = 3$. If there are 4 friends, each requesting a chocolate piece of size {6, 3, 2, 1}, respectively, then we can break the chocolate into 4 parts using 3 cuts as shown in Figure 8.12 (right).

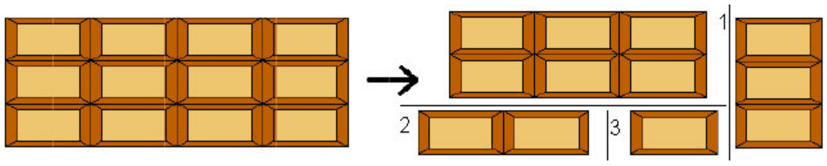

Figure 8.12: Illustration for ACM ICPC WF2010 - J - Sharing Chocolate

For contestants who are already familiar with DP technique, then the following ideas should easily come to mind: First, if sum of all requests is not the same as $w \times h$, then there is no solution. Otherwise, we can represent a distinct state of this problem using three parameters: $(w, h, bitmask)$ where w and h are the current dimension of the chocolate that we are currently considering; and $bitmask$ is the subset of friends that already have chocolate piece of their chosen size. However, a quick analysis shows that this requires a DP table of size $100 \times 100 \times 2^{15} = 327M$. This is too much for programming contest.

A better state representation is to use only two parameters, either: $(w, bitmask)$ or $(h, bitmask)$. Without loss of generality, we adopt $(w, bitmask)$ formulation. With this formulation, we can 'recover' the required value h via `sum(`$bitmask$`) / `w, where `sum(`$bitmask$`)` is the sum of the piece sizes requested by satisfied friends in $bitmask$ (i.e. all the 'on' bits of $bitmask$). This way, we have all the required parameters: w, h, and $bitmask$, but we only use a DP table of size $100 \times 2^{15} = 3M$. This one is doable.

Base cases: If $bitmask$ only contains 1 'on' bit and the requested chocolate size of that person equals to $w \times h$, we have a solution. Otherwise we do not have a solution.

For general cases: If we have a chocolate piece of size $w \times h$ and a current set of satisfied friends $bitmask = bitmask_1 \bigcup bitmask_2$, we can do either horizontal or vertical cut so that one piece is to serve friends in $bitmask_1$ and the other is to serve friends in $bitmask_2$.

The worst case time complexity for this problem is still huge, but with proper pruning, this solution runs within time limit.

Exercise 8.3.6.1*: Solve UVa 10482 - The Candyman Can and UVa 10626 - Buying Coke that use this technique. Determine which parameter is the most effective to be dropped but can still be recovered from other parameters.

Other than several DP problems in Section 8.4, there are a few more DP problems in Chapter 9 which are not listed in this Chapter 8 as they are considered *rare*. They are:

1. Section 9.2: Bitonic Traveling Salesman Problem (we also re-highlight the 'drop one parameter and recover it from others' technique),
2. Section 9.5: Chinese Postman Problem (another usage of DP with bitmask to solve the minimum weight perfect matching on small general weighted graph),
3. Section 9.20: Matrix Chain Multiplication (a classic DP problem),
4. Section 9.21: Matrix Power (we can speed up the DP transitions for *some* rare DP problems from $O(n)$ to $O(\log n)$ by rewriting the DP recurrences as matrix multiplication),
5. Section 9.22: Max Weighted Independent Set (on tree) can be solved with DP,
6. Section 9.33: Sparse Table Data Structure uses DP.

Programming Exercises related to More Advanced DP:

- DP level 2 (slightly harder than those listed in Chapter 3, 4, 5, and 6)
  1. ***UVa 01172 - The Bridges of ... **** (LA 3986, DP non classic, a bit of matching flavor but with left to right and OS type constraints)
  2. **UVa 01211 - Atomic Car Race *** (LA 3404, Tokyo05, precompute array T[L], the time to run a path of length L; DP with one parameter i, where i is the checkpoint where we change tire; if i = n, we do not change the tire)
  3. UVa 10069 - Distinct Subsequences (use Java BigInteger)
  4. UVa 10081 - Tight Words (use doubles)
  5. UVa 10364 - Square (bitmask technique can be used)
  6. *UVa 10419 - Sum-up the Primes* (print path, prime)
  7. *UVa 10536 - Game of Euler* (model the 4×4 board and 48 possible pins as bitmask; then this is a simple two player game; also see Section 5.8)
  8. UVa 10651 - Pebble Solitaire (small problem size; doable with backtracking)
  9. *UVa 10690 - Expression Again* (DP Subset Sum, with negative offset technique, with addition of simple math)
  10. UVa 10898 - Combo Deal (similar to DP + bitmask; store state as integer)
  11. **UVa 10911 - Forming Quiz Teams *** (elaborated in this section)
  12. *UVa 11088 - End up with More Teams* (similar to UVa 10911, but this time it is about matching of *three* persons to one team)
  13. UVa 11832 - Account Book (interesting DP; s: (id, val); use offset to handle negative numbers; t: plus or minus; print solution)
  14. UVa 11218 - KTV (still solvable with complete search)
  15. *UVa 12324 - Philip J. Fry Problem* (must make an observation that sphere $> n$ is useless)
- DP level 3
  1. UVa 00607 - Scheduling Lectures (returns pair of information)
  2. *UVa 00702 - The Vindictive Coach* (the implicit DAG is not trivial)
  3. *UVa 00812 - Trade on Verweggistan* (mix between greedy and DP)
  4. UVa 00882 - The Mailbox ... (s: (lo, hi, mailbox_left); try all)
  5. **UVa 01231 - ACORN *** (LA 4106, Singapore07, DP with dimension reduction, discussed in this section)
  6. **UVa 01238 - Free Parentheses *** (LA 4143, Jakarta08, problem author: Felix Halim, discussed in this section)
  7. UVa 01240 - ICPC Team Strategy (LA 4146, Jakarta08)
  8. UVa 01244 - Palindromic paths (LA 4336, Amritapuri08, store the best path between i, j; the DP table contains strings)
  9. *UVa 10029 - Edit Step Ladders* (use `map` as memo table)
  10. *UVa 10032 - Tug of War* (DP Knapsack with optimization to avoid TLE)
  11. *UVa 10154 - Weights and Measures* (LIS variant)
  12. UVa 10163 - Storage Keepers (try all possible safe line L and run DP; s: id, N_left; t: hire/skip person 'id' for looking at K storage)
  13. UVa 10164 - Number Game (a bit number theory (modulo), backtracking; do memoization on DP state: (sum, taken))

14. UVa 10271 - Chopsticks (Observation: The 3rd chopstick can be any chopstick, we must greedily select adjacent chopstick, DP state: (pos, k_left), transition: Ignore this chopstick, or take this chopstick and the chopstick immediately next to it, then move to pos + 2; prune infeasible states when there are not enough chopsticks left to form triplets.)
15. UVa 10304 - Optimal Binary ... (classical DP, requires 1D range sum and Knuth-Yao speed up to get $O(n^2)$ solution)
16. *UVa 10604 - Chemical Reaction* (the mixing can be done with any pair of chemicals until there are only two chemicals left; memoize the remaining chemicals with help of `map`; sorting the remaining chemicals help increasing the number of hits to the memo table)
17. *UVa 10645 - Menu* (s: (days_left, budget_left, prev_dish, prev_dish_count); the first two parameters are knapsack-style parameter; the last two parameters are used to determine the price of that dish as first, second, and subsequent usage of the dish has different values)
18. UVa 10817 - Headmaster's Headache (s: (id, bitmask); space: $100 \times 2^{2*8}$)
19. *UVa 11002 - Towards Zero* (a simple DP; use negative offset technique)
20. *UVa 11084 - Anagram Division* (using `next_permutation`/brute force is probably not the best approach, there is a DP formulation for this)
21. UVa 11285 - Exchange Rates (maintain the best CAD & USD each day)
22. **UVa 11391 - Blobs in the Board *** (DP with bitmask on 2D grid)
23. *UVa 12030 - Help the Winners* (s: (idx, bitmask, all1, has2); t: try all shoes that has not been matched to the girl that choose dress 'idx')

- DP level 4
    1. UVa 00473 - Raucous Rockers (the input constraint is not clear; therefore use resizeable `vector` and compact states)
    2. **UVa 01099 - Sharing Chocolate *** (LA 4794, World Finals Harbin10, discussed in this section)
    3. **UVa 01220 - Party at Hali-Bula *** (LA 3794, Tehran06; Maximum Independent Set (MIS) problem on tree; DP; also check if the MIS is unique)
    4. UVa 01222 - Bribing FIPA (LA 3797, Tehran06, DP on Tree)
    5. **UVa 01252 - Twenty Questions *** (LA 4643, Tokyo09, DP, s: (bitmask1, bitmask2) where bitmask1 describes the features that we decide to ask and bitmask2 describes the answers of the features that we ask)
    6. *UVa 10149 - Yahtzee* (DP with bitmask; uses card rules; tedious)
    7. UVa 10482 - The Candyman Can (see **Exercise 8.3.6.1***)
    8. UVa 10626 - Buying Coke (see **Exercise 8.3.6.1***)
    9. *UVa 10722 - Super Lucky Numbers* (needs Java BigInteger; DP formulation must be efficient to avoid TLE; state: (N_digits_left, B, first, previous_digit_is_one) and use a bit of simple combinatorics to get the answer)
    10. *UVa 11125 - Arrange Some Marbles* (counting paths in implicit DAG; 8 dimensional DP)
    11. *UVa 11133 - Eigensequence* (the implicit DAG is not trivial)
    12. *UVa 11432 - Busy Programmer* (the implicit DAG is not trivial)
    13. UVa 11472 - Beautiful Numbers (DP state with four parameters)
- Also see some more DP problems in Section 8.4 and in Chapter 9

8.4 Problem Decomposition

While there are only 'a few' basic data structures and algorithms tested in programming contest problems (we believe that many of them have been covered in this book), the harder problems may require a *combination* of two (or more) algorithms and/or data structures. To solve such problems, we must first decompose the components of the problems so that we can solve each component independently. To be able to do so, we must first be familiar with the individual components (the content of Chapter 1 up to Section 8.3).

Although there are ${}_NC_2$ possible combinations of two out of N algorithms and/or data structures, not all of the combinations make sense. In this section, we compile and list down some[5] of the *more common* combinations of two algorithms and/or data structures based on our experience in solving $\approx$ 1675 UVa online judge problems. We end this section with the discussion of the rare combination of *three* algorithms and/or data structures.

8.4.1 Two Components: Binary Search the Answer and Other

In Section 3.3.1, we have seen binary search the answer on a (simple) simulation problem that does not depend on the fancier algorithms listed after Section 3.3.1. Actually, this technique can be combined with some other algorithms in Section 3.4 - Section 8.3. Several variants that we have encountered so far are binary search the answer plus:

- Greedy algorithm (discussed in Section 3.4), e.g. UVa 714, 11516,
- Graph connectivity test (discussed in Section 4.2), e.g. UVa 295, 10876,
- SSSP algorithm (discussed in Section 4.4), e.g. UVa 10816, IOI 2009 (Mecho),
- Max Flow algorithm (discussed in Section 4.6), e.g. UVa 10983,
- MCBM algorithm (discussed in Section 4.7.4), e.g. UVa 1221, 10804, 11262,
- BigInteger operations (discussed in Section 5.3), e.g. UVa 10606,
- Geometry formulas (discussed in Section 7.2), e.g. UVa 1280, 10566, 10668, 11646.

In this section, we write two more examples of using binary search the answer technique. This combination of binary search the answer plus another algorithm can be spotted by asking this question: "If we guess the required answer (in binary search fashion) and assume this answer is true, will the original problem be solvable or not (a True/False question)?".

Binary Search the Answer plus Greedy algorithm

Abridged problem description of UVa 714 - Copying Books: You are given $m \leq 500$ books numbered $1, 2, \ldots, m$ that may have different number of pages $(p_1, p_2, \ldots, p_m)$. You want to make one copy of each of them. Your task is to assign these books among k scribes, $k \leq m$. Each book can be assigned to a single scriber only, and every scriber must get a *continuous sequence* of books. That means, there exists an increasing succession of numbers $0 = b_0 < b_1 < b_2, \cdots < b_{k-1} \leq b_k = m$ such that i-th scriber ($i > 0$) gets a sequence of books with numbers between $b_{i-1} + 1$ and b_i. Each scribe copies pages at the same rate. Thus, the time needed to make one copy of each book is determined by the scriber who is assigned the most work. Now, you want to determine: "What is the minimum number of pages copied by the scriber with the most work?".

[5]This list is not and probably will not be exhaustive.

There exists a Dynamic Programming solution for this problem, but this problem can also be solved by guessing the answer in binary search fashion. We will illustrate this with an example when $m = 9$, $k = 3$ and $p_1, p_2, \ldots, p_9$ are 100, 200, 300, 400, 500, 600, 700, 800, and 900, respectively.

If we guess that the $answer = 1000$, then the problem becomes 'simpler', i.e. If the scriber with the most work can only copy up to 1000 pages, can this problem be solved? The answer is 'no'. We can greedily assign the jobs from book 1 to book m as follows: {100, 200, 300, 400} for scribe 1, {500} for scribe 2, {600} for scribe 3. But if we do this, we still have 3 books {700, 800, 900} unassigned. Therefore the answer must be > 1000.

If we guess $answer = 2000$, then we can greedily assign the jobs as follows: {100, 200, 300, 400, 500} for scribe 1, {600, 700} for scribe 2, and {800, 900} for scribe 3. All books are copied and we still have some slacks, i.e. scribe 1, 2, and 3 still have {500, 700, 300} unused potential. Therefore the answer must be ≤ 2000.

This $answer$ is binary-searchable between $[lo..hi]$ where $lo = max(p_i), \forall i \in [1..m]$ (the number of pages of the thickest book) and $hi = p_1 + p_2 + \ldots + p_m$ (the sum of all pages from all books). And for those who are curious, the optimal $answer$ for the test case in this example is 1700. The time complexity of this solution is $O(m \log hi)$. Notice that this extra log factor is usually negligible in programming contest environment.

Binary Search the Answer plus Geometry formulas

We use UVa 11646 - Athletics Track for another illustration of Binary Search the Answer tecnique. The abridged problem description is as follows: Examine a rectangular soccer field with an athletics track as seen in Figure 8.13—left where the two arcs on both sides (arc1 and arc2) are from the same circle centered in the middle of the soccer field. We want the length of the athletics track (L1 + arc1 + L2 + arc2) to be exactly 400m. If we are given the ratio of the length L and width W of the soccer field to be $a : b$, what should be the actual length L and width W of the soccer field that satisfy the constraints above?

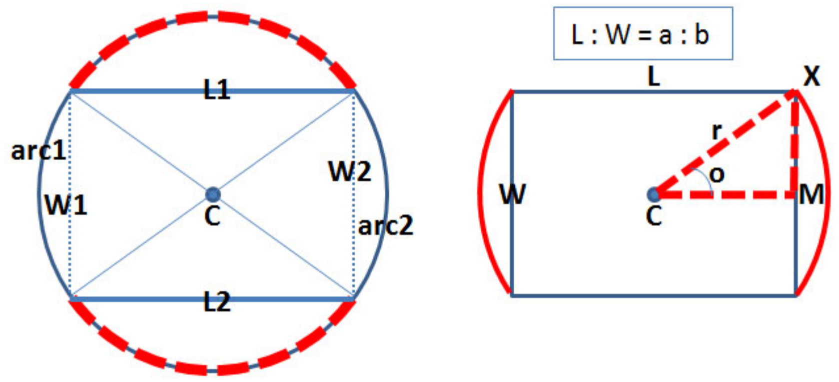

Figure 8.13: Athletics Track (from UVa 11646)

It is quite hard (but not impossible) to obtain the solution with pen and paper strategy (analytical solution), but with the help of a computer and binary search the answer (actually bisection method) technique, we can find the solution easily.

We binary search the value of L. From L, we can get $W = b/a \times L$. The expected length of an arc is $(400 - 2 \times L)/2$. Now we can use Trigonometry to compute the radius r and the angle o via triangle CMX (see Figure 8.13—right). $CM = 0.5 \times L$ and $MX = 0.5 \times W$. With r and o, we can compute the actual arc length. We then compare this value with the expected arc length to decide whether we have to increase or decrease the length L. The snippet of the code is shown below.

```
lo = 0.0; hi = 400.0;           // this is the possible range of the answer
while (fabs(lo - hi) > 1e-9) {
  L = (lo + hi) / 2.0;                             // do bisection method on L
  W = b / a * L;                 // W can be derived from L and ratio a : b
  expected_arc = (400 - 2.0 * L) / 2.0;                    // reference value

  CM = 0.5 * L; MX = 0.5 * W;                       // apply Trigonometry here
  r = sqrt(CM * CM + MX + MX);
  angle = 2.0 * atan(MX / CM) * 180.0 / PI;      // arc's angle = 2x angle o
  this_arc = angle / 360.0 * PI * (2.0 * r);        // compute the arc value

  if (this_arc > expected_arc) hi = L; else lo = L;  // decrease/increase L
}
printf("Case %d: %.12lf %.12lf\n", caseNo++, L, W);
```

Exercise 8.4.1.1*: Prove that other strategies will not be better than the greedy strategy mentioned for the UVa 714 solution above?

Exercise 8.4.1.2*: Derive analytical solution for UVa 11646 instead of using this binary search the answer technique.

8.4.2 Two Components: Involving 1D Static RSQ/RMQ

This combination should be rather easy to spot. The problem involves *another* algorithm to populate the content of a *static* 1D array (that will not be changed anymore once it is populated) and then there will be *many* Range Sum/Minimum/Maximum Queries (RSQ/RMQ) on this static 1D array. Most of the time, these RSQs/RMQs are asked at the output phase of the problem. But sometimes, these RSQs/RMQs are used to speed up the internal mechanism of the other algorithm to solve the problem.

The solution for 1D Static RSQ with Dynamic Programming has been discussed in Section 3.5.2. For 1D Static RMQ, we have the Sparse Table Data Structure (which is a DP solution) that is discussed in Section 9.33. Without this RSQ/RMQ DP speedup, the other algorithm that is needed to solve the problem usually ends up receiving the TLE verdict.

As a simple example, consider a simple problem that asks how many primes there are in various query ranges $[a..b]$ ($2 \leq a \leq b \leq 1000000$). This problem clearly involves Prime Number generation (e.g. Sieve algorithm, see Section 5.5.1). But since this problem has $2 \leq a \leq b \leq 1000000$, we will get TLE if we keep answering each query in $O(b-a+1)$ time by iterating from a to b, especially if the problem author purposely set $b-a+1$ to be near 1000000 at (almost) every query. We need to speed up the output phase into $O(1)$ per query using 1D Static RSQ DP solution.

8.4.3 Two Components: Graph Preprocessing and DP

In this subsection, we want to highlight a problem where graph pre-processing is one of the components as the problem clearly involves some graphs and DP is the other component. We show this combination with two examples.

SSSP/APSP plus DP TSP

We use UVa 10937 - Blackbeard the Pirate to illustrate this combination of SSSP/APSP plus DP TSP. The SSSP/APSP is usually used to transform the input (usually an implicit graph/grid) into another (usually smaller) graph. Then we run Dynamic Programming solution for TSP on the second (usually smaller) graph.

The given input for this problem is shown on the left of the diagram below. This is a 'map' of an island. Blackbeard has just landed at this island and at position labeled with a '@'. He has stashed up to 10 treasures in this island. The treasures are labeled with exclamation marks '!'. There are angry natives labeled with '*'. Blackbeard has to stay away at least 1 square away from the angry natives in any of the eight directions. Blackbeard wants to grab all his treasures and go back to his ship. He can only walk on land '.' cells and not on water '∼' cells nor on obstacle cells '#'.

```
    Input:          Index @ and !       The APSP Distance Matrix
Implicit Graph    Enlarge * with X      A complete (small) graph
  ~~~~~~~~~~         ~~~~~~~~~~           -------------------
  ~~!!!###~~         ~~123###~~           | 0| 1| 2| 3| 4| 5|
  ~##...###~         ~##..X###~           -------------------
  ~#....*##~         ~#..XX*##~         |0| 0|11|10|11| 8| 8|
  ~#!..**~~~         ~#4.X**~~~         |1|11| 0| 1| 2| 5| 9|
  ~~....~~~~   ==>   ~~..XX~~~~   ==>   |2|10| 1| 0| 1| 4| 8|
  ~~~....~~~         ~~~....~~~         |3|11| 2| 1| 0| 5| 9|
  ~~..~..@~~         ~~..~..0~~         |4| 8| 5| 4| 5| 0| 6|
  ~#!.~~~~~~         ~#5.~~~~~~         |5| 8| 9| 8| 9| 6| 0|
  ~~~~~~~~~~         ~~~~~~~~~~         ---------------------
```

This is clearly a TSP problem (see Section 3.5.3), but before we can use DP TSP solution, we have to first transform the input into a distance matrix.

In this problem, we are only interested in the '@' and the '!'s. We give index 0 to '@' and give positive indices to the other '!'s. We enlarge the reach of each '*' by replacing the '.' around the '*' with a 'X'. Then we run BFS on this unweighted implicit graph starting from '@' and all the '!', by only stepping on cells labeled with '.' (land cells). This gives us the All-Pairs Shortest Paths (APSP) distance matrix as shown in the diagram above.

Now, after having the APSP distance matrix, we can run DP TSP as shown in Section 3.5.3 to obtain the answer. In the test case shown above, the optimal TSP tour is: 0-5-4-1-2-3-0 with cost = 8+6+5+1+1+11 = 32.

SCC Contraction plus DP Algorithm on DAG

In some modern problems involving *directed* graph, we have to deal with the Strongly Connected Components (SCCs) of the directed graph (see Section 4.2.9). One of the newer variants is the problem that requires all SCCs of the given directed graph to be *contracted* first to form larger vertices (which we name as super vertices). The original directed graph is not guaranteed to be acyclic, thus we cannot immediately apply DP techniques on such graph. But when the SCCs of a directed graph are contracted, the resulting graph of super vertices is a DAG (see Figure 4.9 for an example). If you recall our discussion in Section 4.7.1, DAG is very suitable for DP techniques as it is acyclic.

UVa 11324 - The Largest Clique is one such problem. This problem in short, is about finding the longest path on the DAG of contracted SCCs. Each super vertex has weight that represents the number of original vertices that are contracted into that super vertex.

8.4.4 Two Components: Involving Graph

This type of problem combinations can be spotted as follows: One clear component is a graph algorithm. However, we need another supporting algorithm, which is usually some sort of mathematics or geometric rule (to build the underlying graph) or even another supporting graph algorithm. In this subsection, we illustrate one such example.

In Section 2.4.1, we have mentioned that for some problems, the underlying graph does not need to be stored in any graph specific data structures (implicit graph). This is possible if we can derive the edges of the graph easily or via some rules. UVa 11730 - Number Transformation is one such problem.

While the problem description is all mathematics, the main problem is actually a Single-Source Shortest Paths (SSSP) problem on unweighted graph solvable with BFS. The underlying graph is generated on the fly during the execution of the BFS. The source is the number S. Then, every time BFS process a vertex u, it enqueues unvisited vertex $u + x$ where x is a prime factor of u that is not 1 or u itself. The BFS layer count when target vertex T is reached is the minimum number of transformations needed to transform S into T according to the problem rules.

8.4.5 Two Components: Involving Mathematics

In this problem combination, one of the components is clearly a mathematics problem, but it is not the only one. It is usually not graph as otherwise it will be classified in the previous subsection above. The other component is usually recursive backtracking or binary search. It is also possible to have two different mathematics algorithms in the same problem. In this subsection, we illustrate one such example.

UVa 10637 - Coprimes is the problem of partitioning S ($0 < S \leq 100$) into t ($0 < t \leq 30$) co-prime numbers. For example, for $S = 8$ and $t = 3$, we can have $1 + 1 + 6$, $1 + 2 + 5$, or $1 + 3 + 4$. After reading the problem description, we will have a strong feeling that this is a mathematics (number theory) problem. However, we will need more than just Sieve of Eratosthenes algorithm to generate the primes and GCD algorithm to check if two numbers are co-prime, but also a recursive backtracking routine to generate all possible partitions.

8.4.6 Two Components: Complete Search and Geometry

Many (computational) geometry problems are brute-force-able (although some requires Divide and Conquer-based solution). When the given input constraints allow for such Complete Search solution, do not hesitate to go for it.

For example, UVa 11227 - The silver bullet boils down into this problem: Given N ($1 \leq N \leq 100$) points on a 2D plane, determine the maximum number of points that are collinear. We can afford to use the following $O(N^3)$ Complete Search solution as $N \leq 100$ (there is a better solution). For each pair of point i and j, we check the other N-2 points if they are collinear with line $i - j$. This solution can be easily written with three nested loops and the `bool collinear(point p, point q, point r)` function shown in Section 7.2.2.

8.4.7 Two Components: Involving Efficient Data Structure

This problem combination usually appear in some 'standard' problem but with *large* input constraint such that we have to use a more efficient data structure to avoid TLE.

For example, UVa 11967-Hic-Hac-Hoe is an extension of a board game Tic-Tac-Toe. Instead of the small 3×3 board, this time the board size is 'infinite'. Thus, there is no way we can record the board using a 2D array. Fortunately, we can store the coordinates of the 'noughts' and 'crosses' in a balanced BST and refer to this BST to check the game state.

8.4.8 Three Components

In Section 8.4.1-8.4.7, we have seen various examples of problems involving two components. In this subsection, we show two examples of rare combinations of three different algorithms and/or data structures.

Prime Factors, DP, Binary Search

UVa 10856 - Recover Factorial can be abridged as follow: "Given N, the number of prime factors in $X!$, what is the minimum possible value of X? ($N \leq 10000001$)". This problem is quite challenging. To make it doable, we have to decompose it into several components.

First, we compute the number of prime factors of an integer i and store it in a table `NumPF[i]` with the following recurrence: If i is a prime, then `NumPF[i] = 1` prime factor; else if $i = PF \times i'$, then `NumPF[i] = 1` + the number of prime factors of i'. We compute this number of prime factors $\forall i \in$ `[1..2703665]`. The upper bound of this range is obtained by trial and error according to the limits given in the problem description.

Then, the second part of the solution is to *accumulate* the number of prime factors of $N!$ by setting `NumPF[i] += NumPF[i-1];` $\forall i \in$ `[1..`N`]`. Thus, `NumPF[N]` contains the number of prime factors of $N!$. This is the DP solution for the 1D Static RSQ problem.

Now, the third part of the solution should be obvious: We can do binary search to find the index X such that `NumPF[X] = N`. If there is no answer, we output "Not possible.".

Complete Search, Binary Search, Greedy

In this write up, we discuss an ACM ICPC World Finals programming problem that combines *three* problem solving paradigms that we have learned in Chapter 3, namely: Complete Search, Divide & Conquer (Binary Search), and Greedy.

ACM ICPC World Finals 2009 - Problem A - A Careful Approach, LA 4445

Abridged problem description: You are given a scenario of airplane landings. There are $2 \leq n \leq 8$ airplanes in the scenario. Each airplane has a time window during which it can safely land. This time window is specified by two integers a_i and b_i, which gives the beginning and end of a closed interval $[a_i..b_i]$ during which the i-th plane can land safely. The numbers a_i and b_i are specified in minutes and satisfy $0 \leq a_i \leq b_i \leq 1440$ (24 hours). In this problem, you can assume that the plane landing time is negligible. Your tasks are:

1. Compute an **order for landing all airplanes** that respects these time windows.
   HINT: order = permutation = Complete Search?

2. Furthermore, the airplane landings should be stretched out **as much as possible** so that the minimum achievable time gap between successive landings is as large as possible. For example, if three airplanes land at 10:00am, 10:05am, and 10:15am, then the smallest gap is five minutes, which occurs between the first two airplanes. Not all gaps have to be the same, but the smallest gap should be as large as possible.
   HINT: Is this similar to 'interval covering' problem (see Section 3.4.1)?

3. Print the answer split into minutes and seconds, rounded to the closest second.

See Figure 8.14 for illustration:
line = the safe landing time window of a plane.
star = the plane's optimal landing schedule.

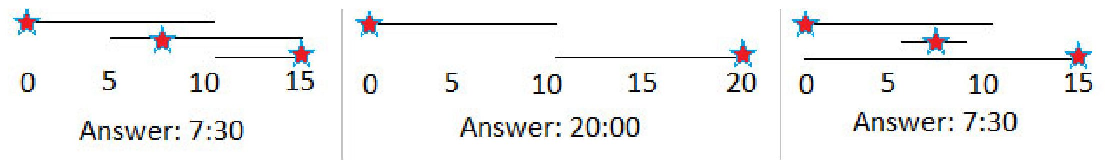

Figure 8.14: Illustration for ACM ICPC WF2009 - A - A Careful Approach

Solution: Since the number of planes is at most 8, an optimal solution can be found by simply trying all 8! = 40320 possible orders for the planes to land. This is the **Complete Search** component of the problem which can be easily implemented using `next_permutation` in C++ STL `algorithm`.

Now, for each specific landing order, we want to know the largest possible landing window. Suppose we guess that the answer is a certain window length L. We can greedily check whether this L is feasible by forcing the first plane to land as soon as possible and the subsequent planes to land in `max(a[that plane], previous landing time +` L`)`. This is the **Greedy** component.

A window length L that is too long/short will cause `lastLanding` (see the code below) to overshoot/undershoot `b[last plane]`, so we have to decrease/increase L. We can binary search the answer L. This is the **Divide and Conquer** component of this problem. As we only want the answer rounded to the nearest integer, stopping binary search when the error $\epsilon <$ 1e-3 is enough. For more details, please study our source code shown below.

```
// World Finals Stockholm 2009, A - A Careful Approach, UVa 1079, LA 4445

#include <algorithm>
#include <cmath>
#include <cstdio>
using namespace std;

int i, n, caseNo = 1, order[8];
double a[8], b[8], L, maxL;

double greedyLanding() {  // with certain landing order, and certain L, try
         // landing those planes and see what is the gap to b[order[n - 1]]
  double lastLanding = a[order[0]];      // greedy, 1st aircraft lands ASAP
  for (i = 1; i < n; i++) {                         // for the other aircrafts
    double targetLandingTime = lastLanding + L;
    if (targetLandingTime <= b[order[i]])
       // can land: greedily choose max of a[order[i]] or targetLandingTime
      lastLanding = max(a[order[i]], targetLandingTime);
    else
      return 1;
  }
  // return +ve value to force binary search to reduce L
  // return -ve value to force binary search to increase L
  return lastLanding - b[order[n - 1]];
}
```

```
int main() {
  while (scanf("%d", &n), n) {                               // 2 <= n <= 8
    for (i = 0; i < n; i++) {   // plane i land safely at interval [ai, bi]
      scanf("%lf %lf", &a[i], &b[i]);
      a[i] *= 60; b[i] *= 60;  // originally in minutes, convert to seconds
      order[i] = i;
    }

    maxL = -1.0;                                // variable to be searched for
    do {                              // permute plane landing order, up to 8!
      double lo = 0, hi = 86400;              // min 0s, max 1 day = 86400s
      L = -1;                              // start with an infeasible solution
      while (fabs(lo - hi) >= 1e-3) {        // binary search L, EPS = 1e-3
        L = (lo + hi) / 2.0;   // we want the answer rounded to nearest int
        double retVal = greedyLanding();                // round down first
        if (retVal <= 1e-2) lo = L;                      // must increase L
        else                hi = L;          // infeasible, must decrease L
      }
      maxL = max(maxL, L);             // get the max over all permutations
    }
    while (next_permutation(order, order + n));     // try all permutations

    // other way for rounding is to use printf format string: %.0lf:%0.2lf
    maxL = (int)(maxL + 0.5);                        // round to nearest second
    printf("Case %d: %d:%0.2d\n", caseNo++, (int)(maxL/60), (int)maxL%60);
  }

  return 0;
}
```

Source code: `ch8_04_UVa1079.cpp/java`

Exercise 8.4.8.1: The given code above is Accepted, but it uses 'double' data type for `lo`, `hi`, and `L`. This is actually unnecessary as all computations can be done in integers. Also, instead of using `while (fabs(lo - hi) >= 1e-3)`, use `for (int i = 0; i < 50; i++)` instead! Please rewrite this code!

Programming Exercises related to Problem Decomposition:

- Two Components - Binary Search the Answer and Other
  1. UVa 00714 - Copying Books (binary search the answer + greedy)
  2. UVa 01221 - Against Mammoths (LA 3795, Tehran06, binary search the answer + MCBM (perfect matching); use the augmenting path algorithm to compute MCBM—see Section 4.7.4)
  3. *UVa 01280 - Curvy Little Bottles* (LA 6027, World Finals Warsaw12, binary search the answer and geometric formula)
  4. *UVa 10372 - Leaps Tall Buildings ...* (binary search the answer + Physics)
  5. UVa 10566 - Crossed Ladders (bisection method)
  6. *UVa 10606 - Opening Doors* (the solution is simply the highest square number $\leq N$, but this problem involves BigInteger; we use a (rather slow) binary search the answer technique to obtain $\sqrt{N}$)
  7. UVa 10668 - Expanding Rods (bisection method)
  8. UVa 10804 - Gopher Strategy (similar to UVa 11262)
  9. UVa 10816 - Travel in Desert (binary search the answer + Dijkstra's)
  10. **UVa 10983 - Buy one, get ... *** (binary search the answer + max flow)
  11. **UVa 11262 - Weird Fence *** (binary search the answer + MCBM)
  12. **UVa 11516 - WiFi *** (binary search the answer + greedy)
  13. UVa 11646 - Athletics Track (the circle is at the center of track)
  14. *UVa 12428 - Enemy at the Gates* (binary search the answer + a bit of graph theory about bridges as outlined in Chapter 4)
  15. IOI 2009 - Mecho (binary search the answer + BFS)
- Two Components - Involving DP 1D RSQ/RMQ
  1. UVa 00967 - Circular (similar to UVa 897, but this time the output part can be speed up using DP 1D range sum)
  2. UVa 10200 - Prime Time (complete search, test if isPrime($n^2 + n + 41$) $\forall n \in$ `[a..b]`; FYI, this prime generating formula $n^2 + n + 41$ was found by Leonhard Euler; for $0 \leq n \leq 40$, it works; however, it does not have good accuracy for larger n; finally use DP 1D RSQ to speed up the solution)
  3. UVa 10533 - Digit Primes (sieve; check if a prime is a digit prime; DP 1D range sum)
  4. UVa 10871 - Primed Subsequence (need 1D Range Sum Query)
  5. **UVa 10891 - Game of Sum *** (Double DP; The first DP is the standard 1D Range Sum Query between two indices: i, j. The second DP evaluates the Decision Tree with state (i, j) and try all splitting points; minimax.)
  6. ***UVa 11105 - Semi-prime H-numbers **** (need 1D Range Sum Query)
  7. **UVa 11408 - Count DePrimes *** (need 1D Range Sum Query)
  8. *UVa 11491 - Erasing and Winning* (greedy, optimized with Sparse Table data structure to deal with the static RMQ)
  9. *UVa 12028 - A Gift from ...* (generate the array; sort it; prepare 1D Range Sum Query; then the solution will be much simpler)

- Two Components - Graph Preprocessing and DP

  1. ***UVa 00976 - Bridge Building **** (use a kind of flood fill to separate north and south banks; use it to compute the cost of installing a bridge at each column; a DP solution should be quite obvious after this preprocessing)
  2. UVa 10917 - A Walk Through the Forest (counting paths in DAG; but first, you have to build the DAG by running Dijkstra's algorithm from 'home')
  3. UVa 10937 - Blackbeard the Pirate (BFS $\rightarrow$ TSP, then DP or backtracking; discussed in this section)
  4. UVa 10944 - Nuts for nuts.. (BFS $\rightarrow$ TSP, then use DP, $n \leq 16$)
  5. **UVa 11324 - The Largest Clique *** (longest paths on DAG; but first, you have to transform the graph into DAG of its SCCs; toposort)
  6. **UVa 11405 - Can U Win? *** (BFS from 'k' & each 'P'—max 9 items; then use DP-TSP)
  7. *UVa 11693 - Speedy Escape* (compute shotest paths information using Floyd Warshall's; then use DP)
  8. UVa 11813 - Shopping (Dijsktra's $\rightarrow$ TSP, then use DP, $n \leq 10$)

- Two Components - Involving Graph

  1. *UVa 00273 - Jack Straw* (line segment intersection and Warshall's transitive closure algorithm)
  2. *UVa 00521 - Gossiping* (build a graph; the vertices are drivers; give an edge between two drivers if they can meet; this is determined with mathematical rule (gcd); if the graph is connected, then the answer is 'yes')
  3. *UVa 01039 - Simplified GSM Network* (LA 3270, World Finals Shanghai05, build the graph with simple geometry; then use Floyd Warshall's)
  4. ***UVa 01092 - Tracking Bio-bots **** (LA 4787, World Finals Harbin10, compress the graph first; do graph traversal from exit using only south and west direction; inclusion-exclusion)
  5. UVa 01243 - Polynomial-time Red... (LA 4272, Hefei08, Floyd Warshall's transitive closure, SCC, transitive reduction of a directed graph)
  6. UVa 01263 - Mines (LA 4846, Daejeon10, geometry, SCC, see two related problems: UVa 11504 & 11770)
  7. UVa 10075 - Airlines (`gcDistance`—see Section 9.11—with APSP)
  8. UVa 10307 - Killing Aliens in Borg Maze (build SSSP graph with BFS, MST)
  9. UVa 11267 - The 'Hire-a-Coder' ... (bipartite check, MST accept -ve weight)
  10. **UVa 11635 - Hotel Booking *** (Dijkstra's + BFS)
  11. UVa 11721 - Instant View ... (find nodes that can reach SCCs with neg cycle)
  12. UVa 11730 - Number Transformation (prime factoring, see Section 5.5.1)
  13. UVa 12070 - Invite Your Friends (LA 3290, Dhaka05, BFS + Dijkstra's)
  14. *UVa 12101 - Prime Path* (BFS, involving prime numbers)
  15. **UVa 12159 - Gun Fight *** (LA 4407, KualaLumpur08, geometry, MCBM)

- Two Components - Involving Mathematics

  1. *UVa 01195 - Calling Extraterrestrial ...* (LA 2565, Kanazawa02, use sieve to generate the list of primes, brute force each prime p and use binary search to find the corresponding pair q)
  2. *UVa 10325 - The Lottery* (inclusion exclusion principle, brute force subset for small $M \leq 15$, lcm-gcd)
  3. UVa 10427 - Naughty Sleepy ... (numbers in $[10^{(k-1)}..10^k-1]$ has k digits)

4. **UVa 10539 - Almost Prime Numbers *** (sieve; get 'almost primes': non prime numbers that are divisible by only a *single* prime number; we can get a list of 'almost primes' by listing the powers of each prime, e.g. 3 is a prime number, so $3^2 = 9$, $3^3 = 27$, $3^4 = 81$, etc are 'almost primes'; we can then sort these 'almost primes'; and then do binary search)
5. **UVa 10637 - Coprimes *** (involving prime numbers and gcd)
6. **UVa 10717 - Mint *** (complete search + GCD/LCM, see Section 5.5.2)
7. *UVa 11282 - Mixing Invitations* (derangement and binomial coefficient, use Java BigInteger)
8. *UVa 11415 - Count the Factorials* (count the number of factors for each integer, use it to find the number of factors for each factorial number and store it in an array; for each query, search in the array to find the first element with that value with binary search)
9. UVa 11428 - Cubes (complete search + binary search)

- Two Components - Complete Search and Geometry

1. *UVa 00142 - Mouse Clicks* (brute force; point-in-rectangle; `dist`)
2. UVa 00184 - Laser Lines (brute force; `collinear` test)
3. UVa 00201 - Square (counting square of various sizes; try all)
4. UVa 00270 - Lining Up (gradient sorting, complete search)
5. UVa 00356 - Square Pegs And Round ... (Euclidean distance, brute force)
6. *UVa 00638 - Finding Rectangles* (brute force 4 corner points)
7. *UVa 00688 - Mobile Phone Coverage* (brute force; chop the region into small rectangles and decide if a small rectangle is covered by an antenna or not; if it is, add the area of that small rectangle to the answer)
8. **UVa 10012 - How Big Is It? *** (try all 8! permutations, Euclidean `dist`)
9. UVa 10167 - Birthday Cake (brute force A and B, `ccw` tests)
10. UVa 10301 - Rings and Glue (circle-circle intersection, backtracking)
11. UVa 10310 - Dog and Gopher (complete search, Euclidean distance `dist`)
12. UVa 10823 - Of Circles and Squares (complete search; check if point inside circles/squares)
13. **UVa 11227 - The silver bullet *** (brute force; `collinear` test)
14. UVa 11515 - Cranes (circle-circle intersection, backtracking)
15. ***UVa 11574 - Colliding Traffic **** (brute force all pairs of boats; if one pair already collide, the answer is 0.0; otherwise derive a quadratic equation to detect when these two boats will collide, if they will; pick the minimum collision time overall; if there is no collision, output 'No collision.')

- Mixed with Efficient Data Structure

1. *UVa 00843 - Crypt Kicker* (backtracking; try mapping each letter to another letter in alphabet; use Trie data structure (see Section 6.6) to speed up if a certain (partial) word is in the dictionary)
2. *UVa 00922 - Rectangle by the Ocean* (first, compute the area of the polygon; then for every pair of points, define a rectangle with those 2 points; use `set` to check whether a third point of the rectangle is on the polygon; check whether it is better than the current best)
3. *UVa 10734 - Triangle Partitioning* (this is actually a geometry problem involving triangle/cosine rule, but we use a data structure that tolerates floating point imprecision due to triangle side normalization to make sure we count each triangle only once)

4. ***UVa 11474 - Dying Tree **** (use union find; first, connect all branches in the tree; next, connect one tree with another tree if any of their branch has distance less than k (a bit of geometry); then, connect any tree that can reach any doctor; finally, check if the first branch of the first/sick tree is connected to any doctor; the code can be quite long; be careful)
5. **UVa 11525 - Permutation *** (can use Fenwick Tree and binary search the answer to find the lowest index i that has $RSQ(1,i) = Si$)
6. ***UVa 11960 - Divisor Game **** (modified Sieve, number of divisors, static Range Maximum Query, solvable with Sparse Table data structure)
7. UVa 11966 - Galactic Bonding (use union find to keep track of the number of disjoint sets/constellations; if Euclidian dist $\leq D$, union the two stars)
8. *UVa 11967 - Hic-Hac-Hoe* (simple brute force, but we need to use C++ STL `map` as we cannot store the actual tic-tac-toe board; we only remember n coordinates and check if there are k consecutive coordinates that belong to any one player)
9. *UVa 12318 - Digital Roulette* (brute force with `set` data structure)
10. *UVa 12460 - Careful teacher* (BFS problem but needs `set` of string data structure to speed up)

- Three Components

1. ***UVa 00295 - Fatman **** (binary search the answer x for this question: if the person is of diameter x, can he go from left to right? for any pair of obstacles (including the top and bottom walls), lay an edge between them if the person cannot go between them and check if the top and bottom wall are disconnected $\rightarrow$ person with diameter x can pass; Euclidian distance)
2. UVa 00811 - The Fortified Forest (LA 5211, World Finals Eindhoven99, `CH`, `perimeter` of polygon, generate all subsets iteratively with bitmask)
3. ***UVa 01040 - The Traveling Judges **** (LA 3271, World Finals Shanghai05, iterative complete search, try all subsets of 2^{20} cities, form MST with those cities with help of Union-Find DS, complex output formatting)
4. UVa 01079 - A Careful Approach (LA 4445, World Finals Stockholm09, discussed in this chapter)
5. *UVa 01093 - Castles* (LA 4788, World Finals Harbin10, try all possible roots, DP on tree)
6. UVa 01250 - Robot Challenge (LA 4607, SoutheastUSA09, geometry, SSSP on DAG $\rightarrow$ DP, DP 1D range sum)
7. UVa 10856 - Recover Factorial (discussed in this section)
8. *UVa 10876 - Factory Robot* (binary search the answer + geometry, Euclidian distance + union find, similar idea with UVa 295)
9. **UVa 11610 - Reverse Prime *** (first, reverse primes less than 10^6; then, append zero(es) if necessary; use Fenwick Tree and binary search)

8.5 Solution to Non-Starred Exercises

Exercise 8.2.3.1: In C++, we can use `pair<int, int>` (short form: `ii`) to store a pair of (integer) information. For triple, we can use `pair<int, ii>`. For quad, we can use `pair<ii, ii>`. In Java, we do not have such feature and thus we have to create a class that implements comparable (so that we can use Java `TreeMap` to store these objects properly).

Exercise 8.2.3.2: We have no choice but to use a class in C++ and Java. For C/C++, struct is also possible. Then, we have to implement a comparison function for such class.

Exercise 8.2.3.3: State-Space Search is essentially an extension of the Single-Source *Shortest* Paths problem, which is a minimization problem. The longest path problem (maximization problem) is NP-hard and usually we do not deal with such variant as the (minimization problem of) State-Space Search is already complex enough to begin with.

Exercise 8.3.1.1: The solution is similar with UVa 10911 solution as shown in Section 1.2. But in the "Maximum Cardinality Matching" problem, there is a possibility that a vertex is *not* matched. The DP with bitmask solution for a small general graph is shown below:

```
int MCM(int bitmask) {
  if (bitmask == (1 << N) - 1)         // all vertices have been considered
    return 0;                               // no more matching is possible
  if (memo[bitmask] != -1)
    return memo[bitmask];

  int p1, p2;
  for (p1 = 0; p1 < N; p1++)                          // find a free vertex p1
    if (!(bitmask & (1 << p1)))
      break;

  // This is the key difference:
  // We have a choice not to match free vertex p1 with anything
  int ans = MCM(bitmask | (1 << p1));

  // Assume that the small graph is stored in an Adjacency Matrix AdjMat
  for (p2 = 0; p2 < N; p2++)   // find neighbors of vertex p1 that are free
    if (AdjMat[p1][p2] && p2 != p1 && !(bitmask & (1 << p2)))
      ans = max(ans, 1 + MCM(bitmask | (1 << p1) | (1 << p2)));

  return memo[bitmask] = ans;
}
```

Exercise 8.4.8.1: Please refer to Section 3.3.1 for the solution.

8.6 Chapter Notes

We have significantly improve this Chapter 8 as promised in the chapter notes of the previous (second) edition. In the third edition, this Chapter 8 roughly has twice the number of pages and exercises because of two reasons. First: We have solved quite a number of harder problems in between the second and the third edition. Second: We have moved some of the harder problems that were previously listed in the earlier chapters (in the second edition) into this chapter—most notably from Chapter 7 (into Section 8.4.6).

In the third edition, this Chapter 8 is no longer the last chapter. We still have one more Chapter 9 where we list down rare topics that rarely appears, but may be of interest for enthusiastic problem solvers.

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	-	15	33 (+120%)
Written Exercises	-	3	5+8*=13 (+333%)
Programming Exercises	-	83	177 (+113%)

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

Section	Title	Appearance	% in Chapter	% in Book
8.2	More Advanced Search	36	20%	2%
8.3	**More Advanced DP**	51	29%	3%
8.4	**Problem Decomposition**	90	51%	5%

—The first time both of us attended ACM ICPC World Finals together.—

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.

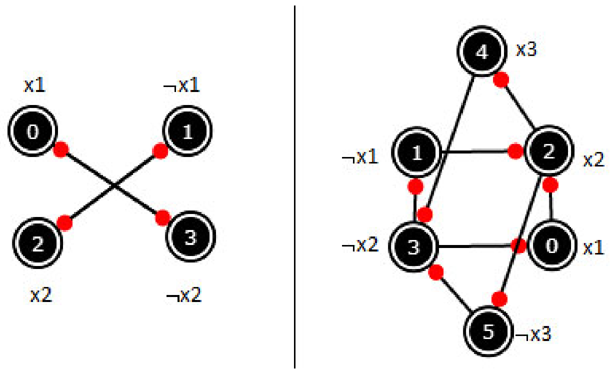

Figure 9.1: The Implication Graph of Example 1 (Left) and Example 2 (Right)

[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.

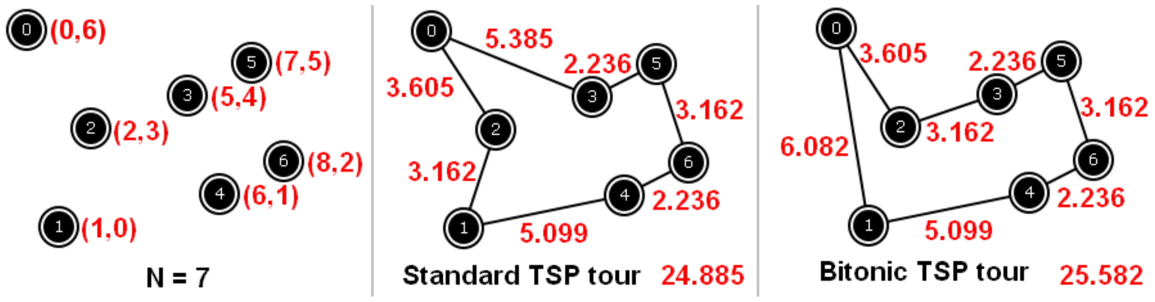

Figure 9.2: The Standard TSP versus Bitonic TSP

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`.

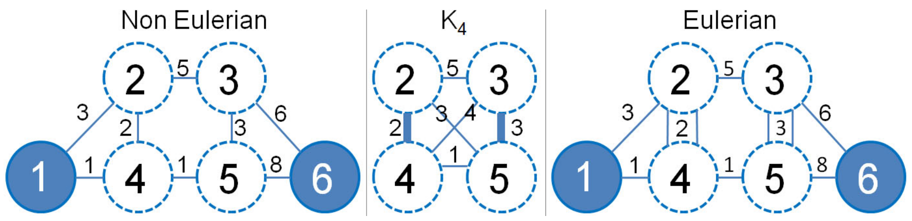

Figure 9.3: An Example of Chinese Postman Problem

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.

```cpp
#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:

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

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 $Z = 1/0.33 = 3$.

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

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

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

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!

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

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.

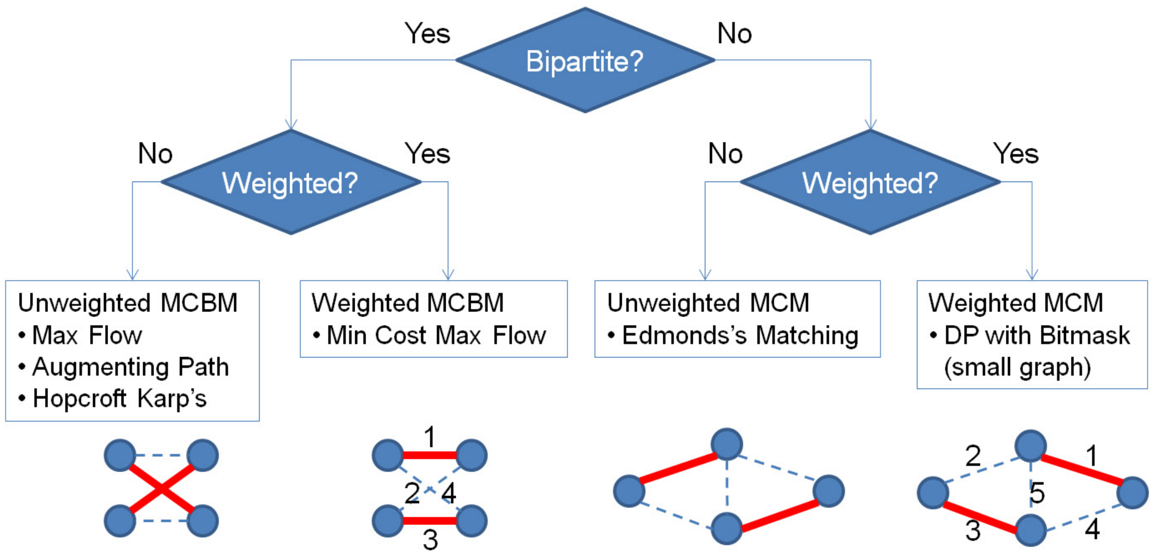

Figure 9.4: The Four Common Variants of Graph Matching in Programming Contests

[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 = $0 \to 2 \to 4 \to 8$ with cost 5, Flow 2 = $0 \to 1 \to 4 \to 2$ (cancel flow 2-4) $\to 6 \to 8$ with cost 15, and Flow 3 = $0 \to 3 \to 5 \to 8$ with cost 20. The minimum total cost is 5 + 15 + 20 = 40.

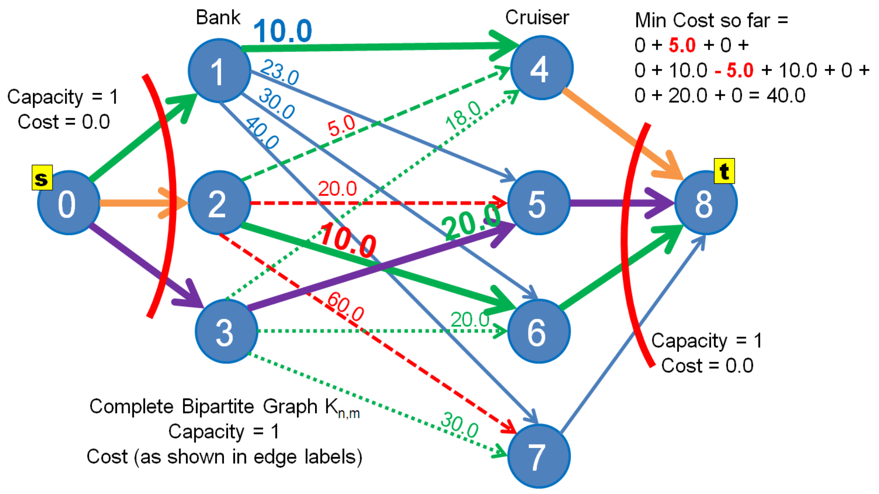

Figure 9.5: A Sample Test Case of UVa 10746: 3 Matchings with Min Cost = 40

[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. $V \leq 18$ 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).

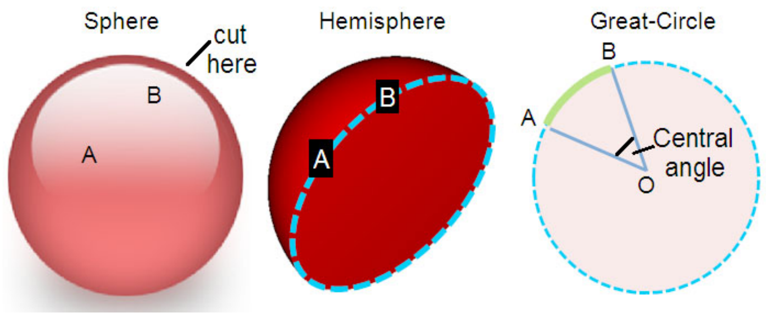

Figure 9.6: L: Sphere, M: Hemisphere and Great-Circle, R: gcDistance (Arc A-B)

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.

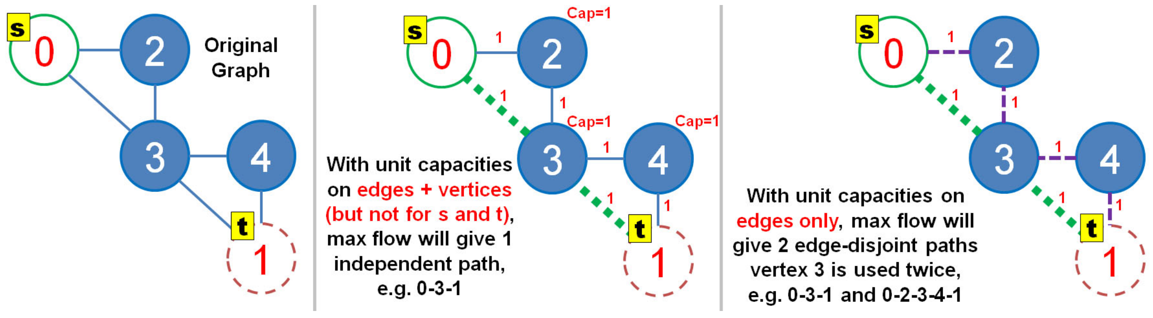
s
0
2
Original Graph
3
4
t
1
Cap=1
1
1
Cap=1
Cap=1
With unit capacities on edges + vertices (but not for s and t), max flow will give 1 independent path, e.g. 0-3-1
With unit capacities on edges only, max flow will give 2 edge-disjoint paths vertex 3 is used twice, e.g. 0-3-1 and 0-2-3-4-1

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 $LCA(u, v) = v$.

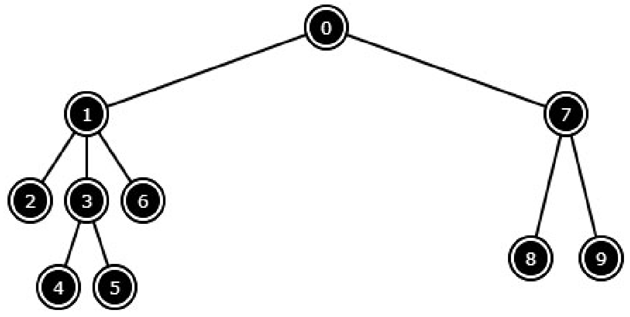

Figure 9.8: An example of a rooted tree T with $n = 10$ vertices

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`).