Chapter 8

More Advanced Topics

Genius is one percent inspiration, ninety-nine percent perspiration.
— **Thomas Alva Edison**

8.1 Overview and Motivation

The main purpose of having this chapter is organizational. The first two sections of this chapter contain the harder material from Chapter 3 and 4. In Section 8.2 and 8.3, we discuss the more challenging variants and techniques involving the two most popular problem solving paradigms: Complete Search and Dynamic Programming. Putting these material in the earlier chapters will probably scare off some new readers of this book.

Section 8.4 contains discussions of complex problems that require *more than one* algorithms and/or data structures. These discussions can be confusing for new programmers if they are listed in the earlier chapters. It is more appropriate to discuss them in this chapter, after various (easier) data structures and algorithms have been discussed. Therefore, it is a good idea to read Chapter 1-7 first before reading this section.

We also encourage readers to avoid rote memorization of the solutions but more importantly, please try to understand the key ideas that may be applicable to other problems.

8.2 More Advanced Search Techniques

In Section 3.2, we have discussed various (simpler) iterative and recursive (backtracking) Complete Search techniques. However, some harder problems require *more clever* Complete Search solutions to avoid the Time Limit Exceeded (TLE) verdict. In this section, we discuss some of these techniques with several examples.

8.2.1 Backtracking with Bitmask

In Section 2.2, we have seen that bitmask can be used to model a small set of Boolean. Bitmask operations are very lightweight and therefore every time we need to use a small set of Boolean, we can consider using bitmask technique to speed up our (Complete Search) solution. In this subsection, we give two examples.

The N-Queens Problem, Revisited

In Section 3.2.2, we have discussed UVa 11195 - Another n-Queen Problem. But even after we have improved the left and right diagonal checks by storing the availability of each of the

n rows and the $2 \times n - 1$ left/right diagonals in three `bitset`s, we still get TLE. Converting these three `bitset`s into three bitmasks help a bit, but this is still TLE.

Fortunately, there is a better way to use these row, left diagonal, and right diagonal checks, as described below. This formulation[1] allows for efficient backtracking with bitmask. We will straightforwardly use three bitmasks for `rw`, `ld`, and `rd` to represent the state of the search. The on bits in bitmasks `rw`, `ld`, and `rd` describe which *rows* are attacked in the *next column*, due to *row*, *left diagonal*, or *right diagonal* attacks from previously placed queens, respectively. Since we consider one column at a time, there will only be n possible left/right diagonals, hence we can have three bitmasks of the same length of n bits (compared with $2 \times n - 1$ bits for the left/right diagonals in the earlier formulation in Section 3.2.2).

Notice that although both solutions (the one in Section 3.2.2 and the one above) use the same data structure: Three bitmasks, the one described above is much more efficient. This highlights the need for problem solver to think from various angles.

We first show the short code of this recursive backtracking with bitmask for the (general) n-queens problem with $n = 5$ and then explain how it works.

```
int ans = 0, OK = (1 << 5) - 1;                    // testing for n = 5 queens

void backtrack(int rw, int ld, int rd) {
  if (rw == OK) { ans++; return; }              // if all bits in 'rw' are on
  int pos = OK & (~(rw | ld | rd));       // the '1's in 'pos' are available
  while (pos) {                                 // this loop is faster than O(n)
    int p = pos & -pos;               // Least Significant One---this is fast
    pos -= p;                                          // turn off that on bit
    backtrack(rw | p, (ld | p) << 1, (rd | p) >> 1);               // clever
} }

int main() {
  backtrack(0, 0, 0);                                    // the starting point
  printf("%d\n", ans);                  // the answer should be 10 for n = 5
} // return 0;
```

For $n = 5$, we start with state `(rw, ld, rd) = (0, 0, 0) = (00000, 00000, 00000)`$_2$. This state is shown in Figure 8.1. The variable `OK = (1 << 5) - 1 = (11111)`$_2$ is used both as terminating condition check and to help decide which rows are available for a certain column. The operation `pos = OK & (~(rw | ld | rd))` *combines* the information of which rows in the next column are attacked by the previously placed queens (via row, left diagonal, or right diagonal attacks), *negates* the result, and *combines* it with `OK` to yield the rows that are *available* for the next column. Initially, all rows in column 0 are available.

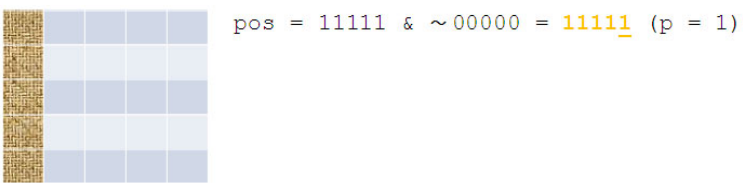


Figure 8.1: 5 Queens problem: The initial state

[1]While this solution is customized for this N-Queens problem, we can probably use parts of the solution for another problem.

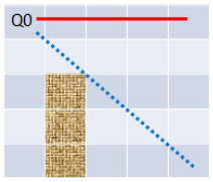
Complete Search (the recursive backtracking) will try all possible rows (that is, all the *on bits* in variable `pos`) of a certain column one by one. Previously in Section 3.2.1, we have seen a way to explore all the on bits of a bitmask in $O(n)$:

```
for (p = 0; p < n; p++)                                          // O(n)
  if (pos && (1 << p))                    // if this bit 'p' is on in 'pos'
    // process p
```

However, this is not the most efficient way. As the recursive backtracking goes deeper, less and less rows are available for selection. Instead of trying all n rows, we can speed up the loop above by just trying all the on bits in variable `pos`. The loop below runs in $O(k)$:

```
while (pos) { // O(k), where k is the number of bits that are on in 'pos'
  int p = pos & -pos;     // determine the Least Significant One in 'pos'
  pos -= p;                                          // turn off that on bit
  // process p
}
```

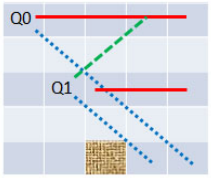
Back to our discussion, for `pos = (11111)`$_2$, we will first start with `p = pos & -pos = 1`, or row 0. After placing the first queen (queen 0) at row 0 of column 0, row 0 is no longer available for the next column 1 and this is quickly captured by bit operation `rw | p` (and also `ld | p` and `rd | p`). Now here is the beauty of this solution. A left/right diagonal increases/decreases the row number that it attacks by one as it changes to the next column, respectively. A shift left/right operation: `(ld | p) << 1` and `(rd | p) >> 1` can nicely capture these behaviour effectively. In Figure 8.2, we see that for the next column 1, row 1 is not available due to left diagonal attack by queen 0. Now only row 2, 3, and 4 are available for column 1. We will start with row 2.




```
rw              = 00001 (1)
ld = 00001 << 1 = 00010 (2)
rd = 00001 >> 1 = 00000|1 (0, the LSB is removed)
---------------------- OR
                  00011 -> NEGATE -> 11100
pos = 11111 & 11100 = 11100 (p = 4)
```

Figure 8.2: 5 Queens problem: After placing the first queen

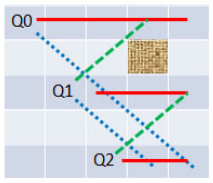
After placing the second queen (queen 1) at row 2 of column 1, row 0 (due to queen 0) and now row 2 are no longer available for the next column 2. The shift left operation for the left diagonal constraint causes row 2 (due to queen 0) and now row 3 to be unavailable for the next column 2. The shift right operation for the right diagonal constraint causes row 1 to be unavailable for the next column 2. Therefore, only row 4 is available for the next column 2 and we have to choose it next (see Figure 8.3).




```
rw              = 00101 (5)
ld = 00110 << 1 = 01100 (12)
rd = 00100 >> 1 = 00010 (2)
---------------------- OR
                  01111 -> NEGATE -> 10000
pos = 11111 & 10000 = 10000 (p = 16)
```

Figure 8.3: 5 Queens problem: After placing the second queen

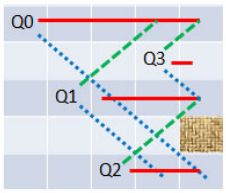
After placing the third queen (queen 2) at row 4 of column 2, row 0 (due to queen 0), row 2 (due to queen 1), and now row 4 are no longer available for the next column 3. The shift left operation for the left diagonal constraint causes row 3 (due to queen 0) and row 4 (due to queen 1) to be unavailable for the next column 3 (there is no row 5—the MSB in bitmask `ld` is unused). The shift right operation for the right diagonal constraint causes row 0 (due to queen 1) and now row 3 to be unavailable for the next column 3. Combining all these information, only row 1 is available for the next column 3 and we have to choose it next (see Figure 8.4).



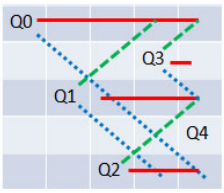

```
rw              =  10101 (21)
ld = 11100 << 1 = 111000 (56, but the MSB is unused)
rd = 10010 >> 1 =  01001 (9)
------------------------ OR
                   111101 -> NEGATE -> 000010
pos = 11111 & 000010 = 00010 (p = 2)
```

Figure 8.4: 5 Queens problem: After placing the third queen

The same explanation is applicable for the fourth and the fifth queen (queen 3 and 4) as shown in Figure 8.5. We can continue this process to get the other 9 solutions for $n = 5$.




```
rw              =   10111 (23)
ld = 111010 << 1 = 1110100 (116, two MSBs are unused)
rd =  01011 >> 1 =   00101 (5)
--------------------------- OR
                   1110111 -> NEGATE -> 0001000
pos = 11111 & 0001000 = 01000 (p = 8)
```




```
rw              =   11111 (we get one solution)
```

Figure 8.5: N-Queens, after placing the fourth and the fifth queens

With this technique, we can solve UVa 11195. We just need to modify the given code above to take the bad cells—which can also be modeled as bitmasks—into consideration. Let's roughly analyze the worst case for $n \times n$ board with no bad cell. Assuming that this recursive backtracking with bitmask has approximately two less rows available at each step, we have a time complexity of $O(n!!)$ where $n!!$ is a notation of multifactorial. For $n = 14$ with no bad cell, the recursive backtracking solution in Section 3.2.2 requires up to $O(14!) \approx 87178M$ operations whereas the recursive backtracking with bitmask above only require around $O(14!!) = 14 \times 12 \times 10 \times \ldots \times 2 = 645120$ operations.

Compact Adjacency Matrix Graph Data Structure

The UVa 11065 - Gentlemen Agreement problem boils down to computation of two integers: The number of Independent Set and the size of the Maximum Independent Set (MIS—see Section 4.7.4 for the problem definition) of a given *general* graph with $V \leq 60$. Finding the MIS of a general graph is an NP-hard problem. Therefore, there is no hope for a polynomial algorithm for this problem.

One solution is the following clever recursive backtracking. The state of the search is a triple: $(i, used, depth)$. The first parameter i implies that we can consider vertices in $[i..V\text{-}1]$ to be included in the Independent Set. The second parameter *used* is a bitmask of length V bits that denotes which vertices are no longer available to be used anymore for the current Independent Set because at least one of their neighbors have been included in the Independent Set. The third parameter *depth* stores the depth of the recursion—which is also the size of the current Independent Set.

There is a clever bitmask trick for this problem that can be used to speed up the solution significantly. Notice that the input graph is small, $V \leq 60$. Therefore, we can store the input graph in an Adjacency Matrix of size up to $V \times V$ (for this problem, we set all cells along the main diagonal of the Adjacency Matrix to true). However, we can compress *one row* of V Booleans ($V \leq 60$) into one bitmask using a 64-bit signed integer.

With this compact Adjacency Matrix $AdjMat$—which is just V rows of 64-bit signed integers—we can use a fast bitmask operation to flag neighbors of vertices efficiently. If we decide to take a free vertex i—i.e. $(used \ \& \ (1 << i)) == 0$, we increase *depth* by one and then use an $O(1)$ bitmask operation: $used \mid AdjMat[i]$ to flag *all* neighbors of i including itself (remember that $AdjMat[i]$ is also a bitmask of length V bits with the i-th bit on).

When all bits in bitmask *used* is turned on, we have just found one more Independent Set. We also record the largest *depth* value throughout the process as this is the size of the Maximum Independent Set of the input graph. The key parts of the code is shown below:

```
void rec(int i, long long used, int depth) {
  if (used == (1 << V) - 1) {               // all intersection are visited
    nS++;                                          // one more possible set
    mxS = max(mxS, depth);                               // size of the set
  }
  else {
    for (int j = i; j < V; j++)
      if (!(used & (1 << j)))          // if intersection j is not yet used
        rec(j + 1, used | AdjMat[j], depth + 1);      // fast bit operation
  }
}

// inside int main()
  // a more powerful, bit-wise adjacency list (for faster set operations)
  for (int i = 0; i < V; i++)
    AdjMat[i] = (1 << i);                                    // i to itself
  for (int i = 0; i < E; i++) {
    scanf("%d %d", &a, &b);
    AdjMat[a] |= (1 << b);
    AdjMat[b] |= (1 << a);
  }
```

Exercise 8.2.1.1*: Sudoku puzzle is another NP-complete problem. The recursive backtracking to find one solution for a standard 9×9 ($n = 3$) Sudoku board can be speed up using bitmask. For each empty cell `(r, c)`, we try putting a digit `[1..`n^2`]` one by one if it is a valid move. The n^2 row, n^2 column, and $n \times n$ square checks can be done with three bitmasks of length n^2 bits. Solve two similar problems: UVa 989 and UVa 10957 with this technique!

8.2.2 Backtracking with Heavy Pruning

Problem I - 'Robots on Ice' in ACM ICPC World Finals 2010 can be viewed as a 'tough test on pruning strategy'. The problem description is simple: Given an $M \times N$ board with 3 check-in points {A, B, C}, find a Hamiltonian[2] path of length $(M \times N)$ from coordinate (0, 0) to coordinate (0, 1). This Hamiltonian path must hit the three check points: A, B, and C at one-quarter, one-half, and three-quarters of the way through its path, respectively. Constraints: $2 \leq M, N \leq 8$.

Example: If given the following 3×6 board with A = (row, col) = (2, 1), B = (2, 4), and C = (0, 4) as in Figure 8.6, then we have two possible paths.

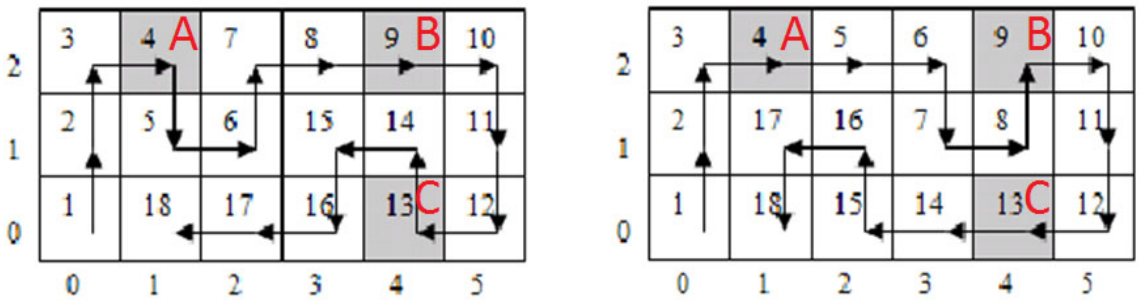


Figure 8.6: Visualization of UVa 1098 - Robots on Ice

A naïve recursive backtracking algorithm will get TLE as there are 4 choices at every step and the maximum path length is $8 \times 8 = 64$ in the largest test case. Trying all 4^{64} possible paths is infeasible. To speed up the algorithm, we must prune the search space if the search:

1. Wanders outside the $M \times N$ grid (obvious),
2. Does not hit the appropriate target check point at 1/4, 1/2, or 3/4 distance—the presence of these three check points actually *reduce* the search space,
3. Hits target check point earlier than the target time,
4. Will not be able to reach the next check point on time from the current position,
5. Will not be able to reach certain coordinates as the current partial path self-block the access to those coordinates. This can be checked with a simple DFS/BFS (see Section 4.2). First, we run DFS/BFS from the goal coordinate (0, 1). If there are coordinates in the $M \times N$ grid that are *not* reachable from (0, 1) and *not yet visited* by the current partial path, we can prune the current partial path.

Exercise 8.2.2.1*: The five pruning strategies mentioned in this subsection are good but actually insufficient to pass the time limit set for LA 4793 and UVa 1098. There is a faster solution for this problem that utilizes the meet in the middle technique (see Section 8.2.4). This example illustrates that the choice of time limit setting may determine which Complete Search solutions are considered as fast enough. Study the idea of meet in the middle technique in Section 8.2.4 and apply it to solve this Robots on Ice problem.

[2]A Hamiltonian path is a path in an undirected graph that visits each vertex exactly once.

8.2.3 State-Space Search with BFS or Dijkstra's

In Section 4.2.2 and 4.4.3, we have discussed two standard graph algorithms for solving the Single-Source Shortest Paths (SSSP) problem. BFS can be used if the graph is unweighted while Dijkstra's should be used if the graph is weighted. The SSSP problems listed in Chapter 4 are still easier in the sense that most of the time we can easily see 'the graph' in the problem description. This is no longer true for some harder graph searching problems listed in this section where the (usually implicit) graphs are no longer trivial to see and the state/vertex can be a complex object. In such case, we usually name the search as 'State-Space Search' instead of SSSP.

When the state is a complex object—e.g. a pair (position, bitmask) in UVa 321 - The New Villa, a quad (row, col, direction, color) in UVa 10047 - The Monocycle, etc—, we normally do not use `vector<int> dist` to store the distance information as in the standard BFS/Dijkstra's implementation. This is because such state may not be easily converted into integer indices. One solution is to use `map<VERTEX-TYPE, int> dist` instead. This trick adds a (small) $\log V$ factor to the time complexity of BFS/Dijkstra's. But for complex State-Space Search, this extra runtime overhead may be acceptable in order to bring down the overall coding complexity. In this subsection, we show one example of such complex State-Space Search.

Exercise 8.2.3.1: How to store `VERTEX-TYPE` if it is a pair, a triple, or a quad of information in both C++ and Java?

Exercise 8.2.3.2: Similar question as in **Exercise 8.2.3.1**, but `VERTEX-TYPE` is a much more complex object, e.g. an array.

Exercise 8.2.3.3: Is it possible that State-Space Search is cast as a maximization problem?

UVa 11212 - Editing a Book

Abridged Problem Description: Given n paragraphs numbered from 1 to n, arrange them in the order of 1, 2, ..., n. With the help of a clipboard, you can press Ctrl-X (cut) and Ctrl-V (paste) several times. You cannot cut twice before pasting, but you can cut several contiguous paragraphs at the same time and these paragraphs will later be pasted in order. What is the minimum number of steps required?

Example 1: In order to make {2, 4, (1), 5, 3, 6} sorted, we cut paragraph (1) and paste it before paragraph 2 to have {1, 2, 4, 5, (3), 6}. Then, we cut paragraph (3) and paste it before paragraph 4 to have {1, 2, 3, 4, 5, 6}. The answer is two steps.

Example 2: In order to make {(3, 4, 5), 1, 2} sorted, we cut three paragraphs at the same time: (3, 4, 5) and paste them after paragraph 2 to have {1, 2, 3, 4, 5}. This is just one single step. This solution is not unique as we can have the following alternative answer: We cut two paragraphs at the same time: (1, 2) and paste them before paragraph 3 to get {1, 2, 3, 4, 5}—this is also one single step.

The loose upper bound of the number of steps required to rearrange these n paragraphs is $O(k)$, where k is the number of paragraphs that are initially in the wrong positions. This is because we can use the following 'trivial' algorithm (which is incorrect): Cut a single paragraph that is in the wrong position and paste that paragraph in the correct position. After k such cut-paste operation, we will definitely have a sorted paragraph. But this may not be the shortest way.

For example, the 'trivial' algorithm above will process {5, 4, 3, 2, 1} as follows: {(5), 4, 3, 2, 1} → {(4), 3, 2, 1, 5} → {(3), 2, 1, 4, 5} → {(2), 1, 3, 4, 5} → {1, 2, 3, 4, 5} of total 4 cut-paste steps. This is not optimal, as we can solve this instance in only 3 steps: {5, 4, (3, 2), 1} → {3, (2, 5), 4, 1} → {3, 4, (1, 2), 5} → {1, 2, 3, 4, 5}.

This problem has a *huge* search space that even for an instance with small $n = 9$, it is near impossible for us to get the answer manually, e.g. We likely will not start drawing the recursion tree just to verify that we need at least 4 steps to sort {5, 4, 9, 8, 7, 3, 2, 1, 6} and at least 5 steps to sort {9, 8, 7, 6, 5, 4, 3, 2, 1}.

The state of this problem is a *permutation* of paragraphs. There are at most $O(n!)$ permutations of paragraphs. With maximum $n = 9$ in the problem statement, this is 9! or 362880. So, the number of vertices of the State-Space graph is not that big actually.

The difficulty of this problem lies in the number of *edges* of the State-Space graph. Given a permutation of length n (a vertex), there are ${}_nC_2$ possible cutting points (index $i, j \in [1..n]$) and there are n possible pasting points (index $k \in [1..(n-(j-i+1))]$). Therefore, for each of the $O(n!)$ vertex, there are about $O(n^3)$ edges connected to it.

The problem actually asked for the shortest path from the source vertex/state (the input permutation) to the destination vertex (a sorted permutation) on this unweighted but huge State-Space graph. The worst case behavior if we run a single $O(V + E)$ BFS on this State-Space graph is $O(n! + (n! * n^3)) = O(n! * n^3)$. For $n = 9$, this is $9! * 93 = 264539520 \approx 265M$ operations. This solution most likely will receive a TLE (or maybe MLE) verdict.

We need a better solution, which we will see in the next Section 8.2.4.

8.2.4 Meet in the Middle (Bidirectional Search)

For some SSSP (but usually State-Space Search) problems on huge graph and we know two vertices: The source vertex/state s and the destination vertex/state t, we may be able to *significantly* reduce the time complexity of the search by searching from *both directions* and hoping that the search will *meet in the middle*. We illustrate this technique by continuing our discussion of the hard UVa 11212 problem.

Before we continue, we need to make a remark that the meet in the middle technique does not always refer to bidirectional BFS. It is a problem solving strategy of 'searching from two directions/parts' that may appear in another form in other difficult searching problem, e.g. see **Exercise 3.2.1.4***.

UVa 11212 - Editing a Book (Revisited)

Although the worst case time complexity of the State-Space Search of this problem is bad, the largest possible answer for this problem is small. When we run BFS on the largest test case with $n = 9$ from the destination state t (the sorted permutation {1, 2, ..., 9}) to reach all other states, we find out that for this problem, the maximum depth of the BFS for $n = 9$ is just 5 (after running it for *a few minutes*—which is TLE in contest environment).

This important information allows us to perform bidirectional BFS by choosing only to go to depth 2 from each direction. While this information is not a necessary condition for us to run a bidirectional BFS, it can help reducing the search space.

There are three possible cases which we discuss below.

Case 1: Vertex s is within two steps away from vertex t (see Figure 8.7).
We first run BFS (max depth of BFS = 2) from the target vertex t to populate distance information from t: `dist_t`. If the source vertex s is already found, i.e. `dist_t[s]` is not `INF`, then we return this value. The possible answers are: 0 (if $s = t$), 1, or 2 steps.

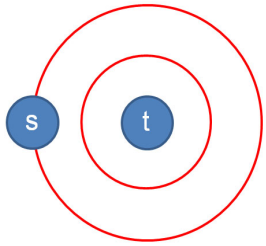


Figure 8.7: Case 1: Example when s is two steps away from t

Case 2: Vertex s is within three to four steps away from vertex t (see Figure 8.8).
If we do not manage to find the source vertex s after Case 1 above, i.e. `dist_t[s] = INF`, we know that s is located further away from vertex t. We now run BFS from the source vertex s (also with max depth of BFS = 2) to populate distance information from s: `dist_s`. If we encounter a common vertex v 'in the middle' during the execution of this second BFS, we know that vertex v is within two layers away from vertex t and s. The answer is therefore `dist_s[v] + dist_t[v]` steps. The possible answers are: 3 or 4 steps.

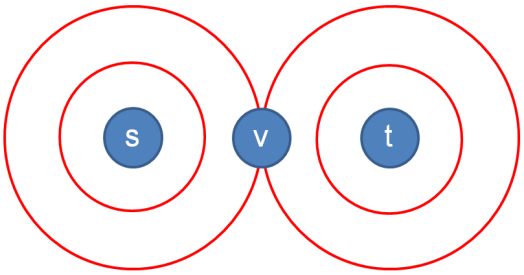


Figure 8.8: Case 2: Example when s is four steps away from t

Case 3: Vertex s is exactly five steps away from vertex t (see Figure 8.9).
If we do not manage to find any common vertex v after running the second BFS in Case 2 above, then the answer is clearly 5 steps that we know earlier as s and t must always be reachable. Stopping at depth 2 allows us to skip computing depth 3, which is *much more time consuming* than computing depth 2.

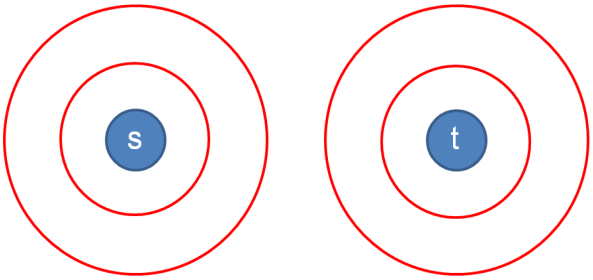


Figure 8.9: Case 3: Example when s is five steps away from t

We have seen that given a permutation of length n (a vertex), there are about $O(n^3)$ branches in this huge State-Space graph. However, if we just run each BFS with at most depth 2, we only execute at most $O((n^3)^2) = O(n^6)$ operations per BFS. With $n = 9$, this is $9^6 = 531441$ operations (this is greater than 9! as there are some overlaps). As the destination vertex t is unchanged throughout the State-Space search, we can compute the first BFS from destination vertex t just once. Then we compute the second BFS from source vertex s per query. Our BFS implementation will have an additional log factor due to the usage of table data structure (e.g. `map`) to store `dist_t` and `dist_s`. This solution is now Accepted.

8.2.5 Informed Search: A* and IDA*

The Basics of A*

Figure 8.10: 15 Puzzle

Complete Search algorithms that we have seen earlier in Chapter 3, 4, and the earlier subsections of this Section are 'uninformed', i.e. all possible states reachable from the current state are *equally good*. For some problems, we do have access to more information (hence the name 'informed search') and we can use the clever A* search that employs heuristic to 'guide' the search direction.

We illustrate this A* search using a well-known 15-puzzle problem. There are 15 slide-able tiles in the puzzle, each with a number from 1 to 15 on it. These 15 tiles are packed into a 4×4 frame with one tile missing. The possible actions are to slide the tile adjacent to the missing tile to the position of that missing tile. Another way of viewing these actions is: "To slide the *blank tile* rightwards, upwards, leftwards, or downwards". The objective of this puzzle is to arrange the tiles so that they looks like Figure 8.10, the 'goal' state.

This seemingly small puzzle is a headache for various search algorithms due to its enormous search space. We can represent a state of this puzzle by listing the numbers of the tiles row by row, left to right into an array of 16 integers. For simplicity, we assign value 0 to the blank tile so the goal state is {1, 2, 3, ..., 14, 15, 0}. Given a state, there can be up to 4 reachable states depending on the position of the missing tile. There are 2/3/4 possible actions if the missing tile is at the 4 corners/8 non-corner sides/4 middle cells, respectively. This is a huge search space.

However, these states are not equally good. There is a nice heuristic for this problem that can help guiding the search algorithm, which is the sum of Manhattan[3] distances between each (non blank) tile in the current state and its location in the goal state. This heuristic gives the lower bound of steps to reach the goal state. By combining the cost so far (denoted by $g(s)$) and the heuristic value (denoted by $h(s)$) of a state s, we have a better idea on where to move next. We illustrate this with a puzzle with starting state A below:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & \underline{\mathbf{0}} \\ 13 & 14 & 15 & 12 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & \underline{\mathbf{0}} \\ 9 & 10 & 11 & \underline{\mathbf{8}} \\ 13 & 14 & 15 & 12 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & \underline{\mathbf{0}} & \underline{\mathbf{11}} \\ 13 & 14 & 15 & 12 \end{bmatrix} \quad D = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & \underline{\mathbf{12}} \\ 13 & 14 & 15 & \underline{\mathbf{0}} \end{bmatrix}$$

The cost of the starting state A is $g(s) = 0$, no move yet. There are three reachable states $\{B, C, D\}$ from this state A with $g(B) = g(C) = g(D) = 1$, i.e. one move. But these three states are *not* equally good:

1. The heuristic value if we slide tile 0 upwards is $h(B) = 2$ as tile 8 and tile 12 are both off by 1. This causes $g(B) + h(B) = 1 + 2 = 3$.
2. The heuristic value if we slide tile 0 leftwards is $h(C) = 2$ as tile 11 and tile 12 are both off by 1. This causes $g(C) + h(C) = 1 + 2 = 3$.
3. But if we slide tile 0 downwards, we have $h(D) = 0$ as all tiles are in their correct position. This causes $g(D) + h(D) = 1 + 0 = 1$, the lowest combination.

[3]The Manhattan distance between two points is the sum of the absolute differences of their coordinates.

If we visit the states in ascending order of $g(s) + h(s)$ values, we will explore the states with the smaller expected cost first, i.e. state D in this example—which is the goal state. This is the essence of the A* search algorithm.

We usually implement this states ordering with the help of a priority queue—which makes the implementation of A* search very similar with the implementation of Dijkstra's algorithm presented in Section 4.4. Note that if $h(s)$ is set to 0 for all states, A* *degenerates* to Dijkstra's algorithm again.

As long as the heuristic function $h(s)$ never overestimates the true distance to the goal state (also known as **admissible heuristic**), this A* search algorithm is optimal. The hardest part in solving search problems using A* search is in finding such heuristic.

Limitations of A*

The problem with A* (and also BFS and Dijkstra's algorithms when used on large State-Space graph) that uses (priority) queue is that the memory requirement can be very huge when the goal state is far from the initial state. For some difficult searching problem, we may have to resort to the following related techniques.

Depth Limited Search

In Section 3.2.2, we have seen recursive backtracking algorithm. The main problem with pure backtracking is this: It may be trapped in an exploration of a very deep path that will not lead to the solution before eventually backtracks after wasting precious runtime.

Depth Limited Search (DLS) places a limit on how deep a backtracking can go. DLS stops going deeper when the depth of the search is longer than what we have defined. If the limit happens to be equal to the depth of the shallowest goal state, then DLS is faster than the general backtracking routine. However, if the limit is too small, then the goal state will be unreachable. If the problem says that the goal state is 'at most d steps away' from the initial state, then use DLS instead of general backtracking routine.

Iterative Deepening Search

If DLS is used wrongly, then the goal state will be unreachable although we have a solution. DLS is usually not used alone, but as part of Iterative Deepening Search (IDS).

IDS calls DLS with *increasing limit* until the goal state is found. IDS is therefore complete and optimal. IDS is a nice strategy that sidesteps the problematic issue of determining the best depth limit by trying all possible depth limits incrementally: First depth 0 (the initial state itself), then depth 1 (those reachable with just one step from the initial state), then depth 2, and so on. By doing this, IDS essentially combines the benefits of lightweight/memory friendly DFS and the ability of BFS that can visit neighboring states layer by layer (see Table 4.2 in Section 4.2).

Although IDS calls DLS many times, the time complexity is still $O(b^d)$ where b is the branching factor and d is the depth of the shallowest goal state. Reason: $O(b^0 + (b^0 + b^1) + (b^0 + b^1 + b^2) + ... + (b^0 + b^1 + b^2 + ... + b^d)) \leq O(c \times b^d) = O(b^d)$.

Iterative Deepening A* (IDA*)

To solve the 15-puzzle problem faster, we can use IDA* (Iterative Deepening A*) algorithm which is essentially IDS with modified DLS. IDA* calls modified DLS to try the all neighboring states in a fixed order (i.e. slide tile 0 rightwards, then upwards, then leftwards, then finally downwards—in that order; we do not use a priority queue). This modified DLS is

stopped not when it has exceeded the depth limit but when its $g(s) + h(s)$ exceeds the best known solution so far. IDA* expands the limit gradually until it hits the goal state.

The implementation of IDA* is not straightforward and we invite readers to scrutinize the given source code in the supporting website.

Source code: `ch8_01_UVa10181.cpp/java`

Exercise 8.2.5.1*: One of the hardest part in solving search problems using A* search is to find the correct admissible heuristic and to compute them efficiently as it has to be repeated many times. List down admissible heuristics that are commonly used in difficult searching problems involving A* algorithm and show how to compute them efficiently! One of them is the Manhattan distance as shown in this section.

Exercise 8.2.5.2*: Solve UVa 11212 - Editing a Book that we have discussed in depth in Section 8.2.3-8.2.4 with A* instead of bidirectional BFS! Hint: First, determine what is a suitable heuristic for this problem.

Programming Exercises solvable with More Advanced Search Techniques:

- More Challenging Backtracking Problems
  1. *UVa 00131 - The Psychic Poker Player* (backtracking with 2^5 bitmask to help deciding which card is retained in hand/exchanged with the top of deck; use 5! to shuffle the 5 cards in hand and get the best value)
  2. *UVa 00710 - The Game* (backtracking with memoization/pruning)
  3. *UVa 00711 - Dividing up* (reduce search space first before backtracking)
  4. *UVa 00989 - Su Doku* (classic Su Doku puzzle; this problem is NP complete but this instance is solvable with backtracking with pruning; use bitmask to speed up the check of available digits)
  5. *UVa 01052 - Bit Compression* (LA 3565 - WorldFinals SanAntonio06, backtracking with some form of bitmask)
  6. **UVa 10309 - Turn the Lights Off *** (brute force the first row in 2^{10}, the rest follows)
  7. *UVa 10318 - Security Panel* (the order is not important, so we can try pressing the buttons in increasing order, row by row, column by column; when pressing one button, only the 3×3 square around it is affected; therefore after we press button (i, j), light $(i-1, j-1)$ must be on (as no button afterward will affect this light); this check can be used to prune the backtracking)
  8. *UVa 10890 - Maze* (looks like a DP problem but the state—involving bitmask—cannot be memoized, fortunately the grid size is 'small')
  9. *UVa 10957 - So Doku Checker* (very similar with UVa 989; if you can solve that one, you can modify your code a bit to solve this one)
  10. **UVa 11195 - Another n-Queen Problem *** (see **Exercise 3.2.1.3*** and the discussion in Section 8.2.1)
  11. ***UVa 11065 - A Gentlemen's Agreement ** (independent set, bitmask helps in speeding up the solution; see the discussion in Section 8.2.1)
  12. *UVa 11127 - Triple-Free Binary Strings* (backtracking with bitmask)
  13. *UVa 11464 - Even Parity* (brute force the first row in 2^{15}, the rest follows)
  14. *UVa 11471 - Arrange the Tiles* (reduce search space by grouping tiles of the same type; recursive backtracking)

- More Challenging State-Space Search with BFS or Dijkstra's
  1. UVa 00321 - The New Villa (s: (position, bitmask 2^{10}), print the path)
  2. *UVa 00658 - It's not a Bug ...* (s: bitmask—whether a bug is present or not, use Dijkstra's as the State-Space graph is weighted)
  3. UVa 00928 - Eternal Truths (s: (row, col, direction, step))
  4. ***UVa 00985 - Round and Round ... **** (4 rotations is the same as 0 rotations, s: (row, col, rotation = [0..3]); find the shortest path from state [1][1][0] to state [R][C][x] where $0 \leq x \leq 3$)
  5. *UVa 01057 - Routing* (LA 3570, World Finals SanAntonio06, use Floyd Warshall's to get APSP information; then model the original problem as another weighted SSSP problem solvable with Dijkstra's)
  6. UVa 01251 - Repeated Substitution ... (LA 4637, Tokyo09, SSSP solvable with BFS)
  7. UVa 01253 - Infected Land (LA 4645, Tokyo09, SSSP solvable with BFS, tedious state modeling)
  8. UVa 10047 - The Monocyle (s: (row, col, direction, color); BFS)
  9. *UVa 10097 - The Color game* (s: (N1, N2); implicit unweighted graph; BFS)
  10. *UVa 10923 - Seven Seas* (s: (ship_position, location of enemies, location of obstacles, steps_so_far); implicit weighted graph; Dijkstra's)
  11. ***UVa 11198 - Dancing Digits **** (s: permutation; BFS; tricky to code)
  12. ***UVa 11329 - Curious Fleas **** (s: bitmask of 26 bits, 4 to describe the position of the die in the 4×4 grid, 16 to describe if a cell has a flea, 6 to describe the sides of the die that has a flea; use `map`; tedious to code)
  13. UVa 11513 - 9 Puzzle (reverse the role of source and destination)
  14. UVa 11974 - Switch The Lights (BFS on implicit unweighted graph)
  15. UVa 12135 - Switch Bulbs (LA 4201, Dhaka08, similar to UVa 11974)
- Meet in the Middle/A*/IDA*
  1. UVa 00652 - Eight (classical sliding block 8-puzzle problem, IDA*)
  2. **UVa 01098 - Robots on Ice *** (LA 4793, World Finals Harbin10, see the discussion in Section 8.2.2; however, there is a faster 'meet in the middle' solution for this problem)
  3. UVa 01217 - Route Planning (LA 3681, Kaohsiung06, solvable with A*/IDA*; test data likely only contains up to 15 stops which already include the starting and the last stop on the route)
  4. **UVa 10181 - 15-Puzzle Problem *** (similar as UVa 652, but this one is larger, we can use IDA*)
  5. UVa 11163 - Jaguar King (another puzzle game solvable with IDA*)
  6. **UVa 11212 - Editing a Book *** (meet in the middle, see Section 8.2.4)
- Also see some more Complete Search problems in Section 8.4

8.3 More Advanced DP Techniques

In Section 3.5, 4.7.1, 5.4, 5.6, and 6.5, we have seen the introduction of Dynamic Programming (DP) technique, several classical DP problems and their solutions, plus a gentle introduction to the easier non classical DP problems. There are several more advanced DP techniques that we have not covered in those sections. Here, we present some of them.

8.3.1 DP with Bitmask

Some of the modern DP problems require a (small) set of Boolean as one of the parameters of the DP state. This is another situation where bitmask technique can be useful (also see Section 8.2.1). This technique is suitable for DP as the integer (that represents the bitmask) can be used as the index of the DP table. We have seen this technique once when we discuss DP TSP (see Section 3.5.2). Here, we give one more example.

UVa 10911 - Forming Quiz Teams

For the abridged problem statement and the solution code of this problem, please refer to the very first problem mentioned in Chapter 1. The grandiose name of this problem is "minimum weight perfect matching on a small general weighted graph". In the general case, this problem is hard. However, if the input size is small, up to $M \leq 20$, then DP with bitmask solution can be used.

The DP with bitmask solution for this problem is simple. The matching state is represented by a `bitmask`. We illustrate this with a small example when $M = 6$. We start with a state where nothing is matched yet, i.e. `bitmask=000000`. If item 0 and item 2 are matched, we can turn on two bits (bit 0 and bit 2) at the same time via this simple bit operation, i.e. `bitmask | (1 << 0) | (1 << 2)`, thus the state becomes `bitmask=000101`. Notice that index starts from 0 and counted from the right. If from this state, item 1 and item 5 are matched next, the state will become `bitmask=100111`. The perfect matching is obtained when the state is all '1's, in this case: `bitmask=111111`.

Although there are many ways to arrive at a certain state, there are only $O(2^M)$ distinct states. For each state, we record the minimum weight of previous matchings that must be done in order to reach this state. We want a perfect matching. First, we find one 'off' bit i using one $O(M)$ loop. Then, we find the best other 'off' bit j from `[`i`+1..`M`-1]` using another $O(M)$ loop and recursively match i and j. This check is again done using bit operation, i.e. we check `if (!(bitmask & (1 << i)))`—and similarly for j. This algorithm runs in $O(M \times 2^M)$. In problem UVa 10911, $M = 2N$ and $2 \leq N \leq 8$, so this DP with bitmask solution is feasible. For more details, please study the code.

Source code: `ch8_02_UVa10911.cpp/java`

In this subsection, we have shown that DP with bitmask technique can be used to solve small instances ($M \leq 20$) of matching on general graph. In general, bitmask technique allows us to represent a small set of up to ≈ 20 items. The programming exercises in this section contain more examples when bitmask is used as *one of the parameters* of the DP state.

Exercise 8.3.1.1: Show the required DP with bitmask solution if we have to deal with "Maximum Cardinality Matching on a small general graph ($V \leq 18$)".

Exercise 8.3.1.2*: Rewrite the code `ch8_02_UVa10911.cpp/java` with the `LSOne` trick shown in Section 8.2.1 to speed it up!

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))).

```
// 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)

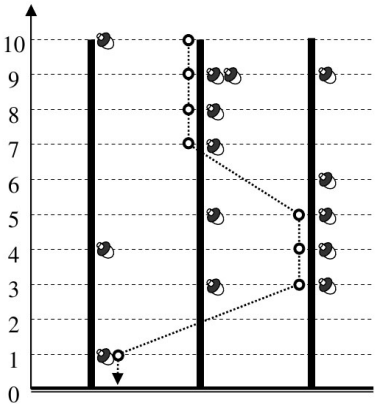


Figure 8.11: The Descent Path

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).

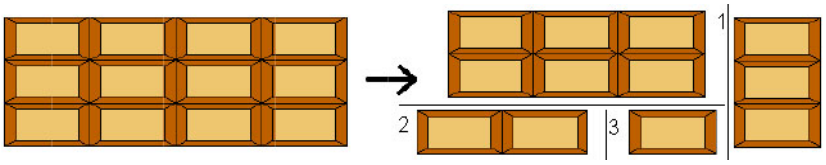


Figure 8.12: Illustration for ACM ICPC WF2010 - J - Sharing Chocolate

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 ≈ 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?

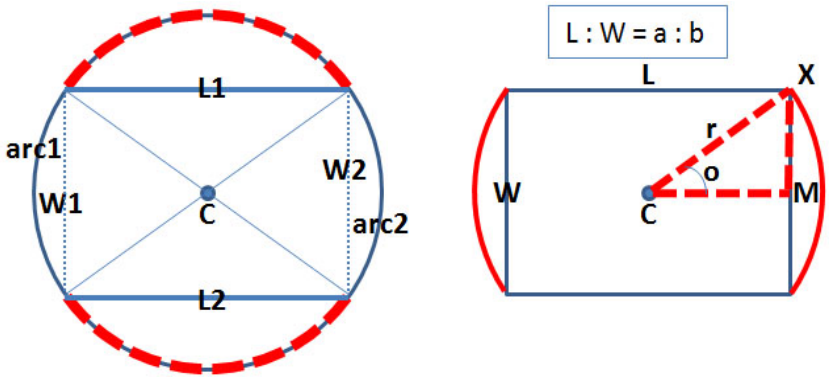


Figure 8.13: Athletics Track (from UVa 11646)

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.

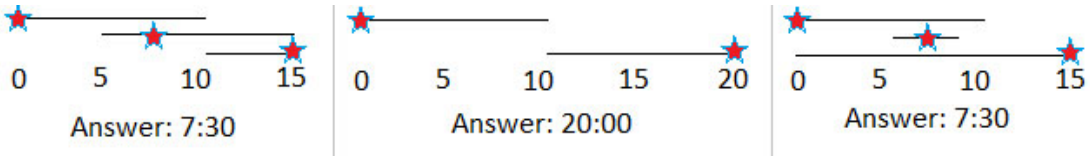


Figure 8.14: Illustration for ACM ICPC WF2009 - A - A Careful Approach

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 → TSP, then DP or backtracking; discussed in this section)
  4. UVa 10944 - Nuts for nuts.. (BFS → 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 → 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 → 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 → 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.—