2-SAT Problem

A clause in a 2-CNF $(a \vee b) \equiv (\neg a \Rightarrow b) \equiv (\neg b \Rightarrow a)$. We can build the corresponding 'implication graph'. Each variable has two vertices in the implication graph, the variable itself and the negation/inverse of that variable. An edge connects one vertex to another if the corresponding variables are related by an implication in the corresponding 2-CNF formula. A 2-CNF formula is satisfiable if and only if "there is no variable that belongs to the same Strongly Connected Component (SCC) as its negation".

Art Gallery Problem

Problem Description

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

Variant 1: Determine the upper bound of the smallest size of set S.

Variant 2: Determine if ∃ a critical point C in polygon P and ∃ another point $D \in P$ such that if the guard is at position C, the guard cannot protect point D.

Variant 3: Determine if polygon P can be guarded with just one guard.

Variant 4: Determine the smallest size of set S if the guards can only be placed at the vertices of polygon P and only the vertices need to be guarded.

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

The solution for variant 1 is a theoretical work of the Art Gallery theorem by V´aclav Chv´atal. He states that $\lfloor n/3 \rfloor$ guards are always sufficient and sometimes necessary to guard a simple polygon with n vertices (proof omitted).

The solution for variant 2 involves testing if polygon P is concave (and thus has a critical point). We can use the negation of isConvex function shown in Section 7.3.4.

The solution for variant 3 can be hard if one has not seen the solution before. We can use the cutPolygon function discussed in Section 7.3.6. We cut polygon P with all lines formed by the edges in P in counter clockwise fashion and retain the left side at all times. If we still have a non empty polygon at the end, one guard can be placed in that non empty polygon which can protect the entire polygon P.

The solution for variant 4 involves the computation of Minimum Vertex Cover of the 'visibility graph' of polygon P. In general this is another NP-hard problem.

Bitonic Traveling Salesman Problem

Problem Description

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

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

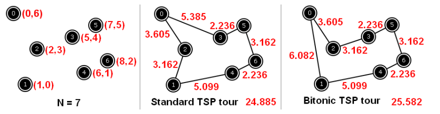


Figure 9.2: The Standard TSP versus Bitonic TSP

Solution(s)

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

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

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

With these observations in mind, we can write a simple DP solution is like this: 339

BITONIC TRAVELING SALESMAN PROBLEM

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

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

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

Bracket Matching

Problem Description

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

Solution(s)

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

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

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

Variant(s)

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

Chinese Postman Problem

Problem Description

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

Solution(s)

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

Figure 9.3: An Example of Chinese Postman Problem

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

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

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

Closest Pair Problem

Problem Description

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

Solution(s)

Complete Search

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

Divide and Conquer

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

Divide: We sort the points in set S by their x-coordinates (if tie, by y-coordinates). Then, we divide set S into two sets of points S_1 and S_2 with a vertical line $x = d$ such that $|S_1| = |S_2|$ or $|S_1| = |S_2| + 1$, i.e. the number of points in each set is balanced.

Conquer: If we only have one point in S, we return ∞. If we only have two points in S, we return their Euclidean distance.

Combine: Let d_1 and d_2 be the smallest distance in S_1 and S_2, respectively. Let d_3 be the smallest distance between all pairs of points (p_1,p_2) where p_1 is a point in S_1 and p_2 is a point in S_2. Then, the smallest distance is $\min(d_1,d_2,d_3)$, i.e. the answer may be in the smaller set of points S_1 or in S_2 or one point in S_1 and the other point in S_2, crossing through line $x = d$.

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

Graph Matching

Solutions for Unweighted MCBM

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

Reducing the Unweighted MCBM problem into a Max Flow Problem. See Section 4.6 and 4.7.4 for details. The time complexity depends on the chosen Max Flow algorithm.

$O(V^2 + VE)$ Augmenting Path Algorithm for Unweighted MCBM. See Section 4.7.4 for details. This is good enough for various contest problems involving Unweighted MCBM. √

O(V E) Hopcroft Karp's Algorithm for Unweighted MCBM See Section 9.12 for details.

Solutions for Weighted MCBM

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

For example, in Figure 9.5, we show one test case of UVa 10746 - Crime Wave - The Sequel. This is an MCBM problem on a complete bipartite graph $K_{n,m}$, but each edge has associated cost. We add edges from source s to vertices of the left set with capacity 1 and cost 0. We also add edges from vertices of the right set to the sink t also with capacity 1 and cost 0. The directed edges from the left set to the right set has capacity 1 and cost according to the problem description. After having this weighted flow graph, we can run the MCMF algorithm as shown in Section 9.23 to get the required answer: Flow 1 = 0 ⟶ 2 ⟶ 4 ⟶ 8 with cost 5, Flow 2 = 0 ⟶ 1 ⟶ 4 ⟶ 2 (cancel flow 2-4) ⟶ 6 ⟶ 8 with cost 15, and Flow 3 =0⟶3⟶5⟶8withcost20. Theminimumtotalcostis5+15+20=40.

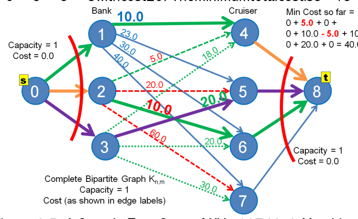
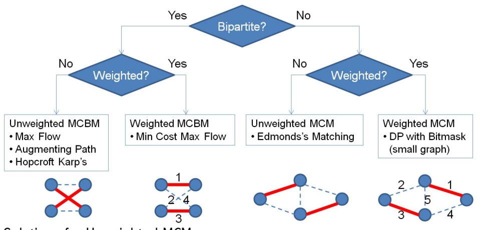


Figure 9.5: A Sample Test Case of UVa 10746: 3 Matchings with Min Cost = 40



Solutions for Unweighted MCM

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

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

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

Solution for Weighted MCM

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

Great-Circle Distance

Problem Description

Sphere is a perfectly round geometrical object in 3D space. The Great-Circle Distance between any two points A and B on sphere is the shortest distance along a path on the surface of the sphere. This path is an arc of the Great- Circle of that sphere that pass through the two points A and B. We can imagine Great-Circle as the resulting circle that appears if we cut the sphere with a plane so that we have two equal hemispheres (see Figure 9.6—left and middle).

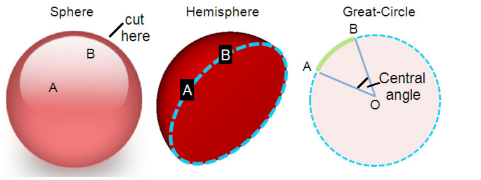


Figure 9.6: L: Sphere, M: Hemisphere and Great-Circle, R: gcDistance (Arc A-B)

Solution(s)

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

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

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

Independent and Edge-Disjoint Paths

Problem Description

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

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

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

Solution(s)

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

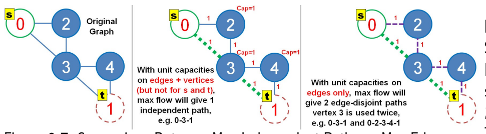

s
0
2
Original Graph
3
4
t
1
With unit capacities on edges + vertices (but not for s and t), max flow will give 1 independent path, e.g. 0-3-1
Cap=1
With unit capacities on edges only, max flow will give 2 edge-disjoint paths vertex 3 is used twice, e.g. 0-3-1 and 0-2-3-4-1

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

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

Inversion Index

Problem Description

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

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

Solution(s) $O(n^2)$ solution

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

O(nlogn) solution

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

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

Josephus Problem

Problem Description

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

Solution(s)   Complete Search for Smaller Instances

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

Special Case when k = 2

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

General Case

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

Magic Square Construction (Odd Size)

Problem Description

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

17 24 1 8 15 | 23 5 7 14 16 | 4 6 13 20 22 | 10 12 19 21 3 |

11 18 25 2 9   Our task is to construct a magic square given its size n, assuming that n is odd.

Solution(s)

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

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

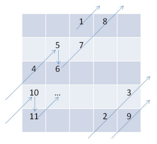


Figure 9.9: The Magic Square Construction Strategy for Odd n

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

Max Weighted Independent Set

Problem Description

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

Solution(s)

On Tree

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

1. If v is a leaf vertex

(a) C(v, true) = w(v)   % If leaf v is taken, then the weight of this subtree is the weight of this v.

(b) C(v,false)=0   % If leaf v is not taken, then the weight of this subtree is 0.

2. If v is an internal vertex

(a) $C(v, true) = w(v) + {}_{ch \in children(v)} C(ch, false)$ % If root v is taken, we add weight of v but all children of v cannot be taken.

(b) $C(v, false) = {}_{ch \in children(v)} \max(C(ch, true), C(ch, false))$ % If root v is not taken, children of v may or may not be taken.   % We return the larger one.

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

On Bipartite Graph

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

9.24 Min Path Cover on DAG

Problem Description

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

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

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

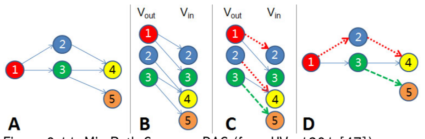


Figure 9.11: Min Path Cover on DAG (from UVa 1201 [47])

Solution(s)

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

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

Pancake Sorting

Problem Description

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

4 (top) 10 36 27 18 0 (bottom) 3

----------------------- index A

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

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

4(top) 10<-- 3 <-- 8 <-- 6 3 3 68<--37...6 277737 1 866<--8 8 0(bottom) 3<-- 10 10 10 10 -- index A B C D ... E

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

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

Constraints: 1 ≤ T ≤ 100, 1 ≤ N ≤ 10, and N ≤ MAX D ≤ 1000000.

Sample Test Cases

Sample Input

7 4 4321 8 87654123 5 51243 5 555555 111111 222222 444444 333333 8 1000000 999999 999998 999997 999996 999995 999994 999993 5 3 8 7 6 10 10 8192057364

Sample Output

0 1 2 2 0 4 11

Explanation

The first stack is already sorted in descending order.

The second stack can be sorted with one call of flip(5).

The third (and also the fourth) input stack can be sorted in descending order by calling flip(3) then flip(1): 2 flips.

The fifth input stack, although contains large integers, is already sorted in descending order, so 0 flip is needed.

The sixth input stack is actually the sample stack shown in the problem description. This stack can be sorted in descending order using at minimum 4 flips, i.e. Solution 1: flip(0), flip(1), flip(2), flip(1): 4 flips. Solution 2: flip(1), flip(2), flip(1), flip(0): also 4 flips.

The seventh stack with N = 10 is for you to test the runtime speed of your solution. Solution(s) First, we need to make an observation that the diameters of the pancake do not really matter. We just need to write simple code to sort these (potentially huge) pancake diameters from [1..1 million] and relabel them to [0..N-1]. This way, we can describe any stack of pancakes as simply a permutation of N integers. If we just need to get the pancakes sorted, we can use a non optimal O(2×N−3) Greedy algorithm: Flip the largest pancake to the top, then flip it to the bottom. Flip the second largest pancake to the top, then flip it to the second from bottom. And so on. If we keep doing this, we will be able to have a sorted pancake in O(2 × N − 3) steps, regardless of the initial state.

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

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

A better solution is a more sophisticated search technique called 'meet in the middle'

(bidirectional BFS) to bring down the search space to a manageable level (see Section

8.2.4). First, we do some preliminary analysis (or we can also look at 'Pancake Number',

http://oeis.org/A058986) to identify that for the largest test case when N = 10, we need

at most 11 flips to sort any input stack to the sorted one. Therefore, we precalculate BFS

from the target permutation to all other permutations for all N ∈ [1..10], but stopping as

soon as we reach depth $\lfloor \frac{11}{2} \rfloor$ = 5. Then, for each test case, we run BFS from the start-

ing permutation again with maximum depth 5. If we encounter a common vertex with the precalculated BFS from target permutation, we know that the answer is the distance from starting permutation to this vertex plus the distance from target permutation to this vertex. If we do not encounter a common vertex at all, we know that the answer should be the maximum flips: 11. On the largest test case with N = 10 for all test cases, this solution has time complexity of roughly $O((10 + T) \times 10^5)$, which is now feasible.

Pollard's rho Integer Factoring Algorithm

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

For a faster integer factorization, one can use the Pollard's rho algorithm [52, 3]. The key idea of this algorithm is that two integers x and y are congruent modulo p (p is one of the factor of n—the integer that we want to factor) with probability 0.5 after 'a few (1.177 $\sqrt{p}$) integers' have been randomly chosen. The theoretical details of this algorithm is probably not that important for Competitive Programming. In this section, we directly provide a working C++ implementation below which can be used to handle composite integer that fit in 64-bit unsigned integers in contest environment. However, Pollard's rho cannot factor an integer n if n is a large prime due to the way the algorithm works. To handle this case, we have to implement a fast (probabilistic) prime testing like the Miller-Rabin's algorithm (see Exercise 5.3.2.4*).

```
#define abs_val(a) (((a)>0)?(a):-(a)) typedef long long ll;
ll mulmod(ll a, ll b, ll c) { // returns (a * b) % c, and minimize overflow ll x = 0, y = a % c;   while (b > 0) {
if (b % 2 == 1) x = (x + y) % c;
y = (y * 2) %
b /= 2; }
return x % c; }
ll gcd(ll a,ll b)
c;
{ return !b ? a : gcd(b, a % b); }
// standard gcd
ll pollard_rho(ll   int i = 0, k = 2;   ll x = 3, y = 3; // random seed = 3, other values possible while (1) {
n) {
i++;   x = (mulmod(x, x, n) + n - 1) % n; ll d = gcd(abs_val(y - x), n);   if (d != 1 && d != n) return d;   if (i == k) y = x, k *= 2;
}}
int main() {   ll n = 2063512844981574047LL;   ll ans = pollard_rho(n);   if (ans > n / ans) ans = n / ans;   printf("%lld %lld\n", ans, n / ans);
}
```

Selection Problem

Problem Description

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

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

O(nlogn) algorithm, static data

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

Expected O(n) algorithm, static data

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

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

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

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

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

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

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

A simplified (but not rigorous) analysis is to assume RandomizedSelect divides the array

into two at each step and n is a power of two. Therefore it runs RandomizedPartition in

O(n) for the first round, in O($\frac{n}{2}$) in the second round, in O($\frac{n}{4}$) in the third round and finally 24

O(1) in the 1 + $\log_2$ n round. The cost of RandomizedSelect is mainly determined by the
cost of RandomizedPartition as all other steps of RandomizedSelect is O(1). Therefore

theoverallcostisO(n+^{n} +^{n} +...+^{n})=O(n×(^{1} +^{1} +^{1}
+...+^{1})≤O(2n)=O(n). 24n 124n

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

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

O(nlogn) pre-processing, O(logn) algorithm, dynamic data

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

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

If q + 1 = k, then the root is the desired answer. We return this value and stop.

If q + 1 > k, the desired answer is inside the left sub-tree of the root.

If q + 1 < k, the desired answer is inside the right sub-tree of the root and we are now searching for the (k−q−1)-th smallest element in this right sub-tree. This adjustment of k is needed to ensure correctness.

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

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

Sliding Window

Problem Description

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

Find the smallest sub-array size (smallest window length) so that the sum of the sub- array is greater than or equal to a certain constant S in O(n)?

Examples: For array A_1 = {5,1,3,[5,10],7,4,9,2,8} and S = 15, the answer is 2 as highlighted. For array A_2 = {1, 2, [3, 4, 5]} and S = 11, the answer is 3 as highlighted.

Find the smallest sub-array size (smallest window length) so that the elements inside the sub-array contains all integers in range [1..K].

Examples: For array A = {1,[2,3,7,1,12,9,11,9,6,3,7,5,4],5,3,1,10,3,3} and K = 4, the an- swer is 13 as highlighted. For the same array A = {[1,2,3],7,1,12,9,11,9,6,3,7,5,4,5,3,1,10,3,3} and K = 3, the answer is 3 as highlighted.

Find the maximum sum of a certain sub-array with (static) size K.

Examples: For array A_1 = {10, [50, 30, 20], 5, 1} and K = 3, the answer is 100 by summing the highlighted sub-array. For array A_2 = {49, 70, 48, [61, 60], 60} and K = 2, the answer is 121 by summing the highlighted sub-array.

Find the minimum of each possible sub-arrays with (static) size K.

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

Solution(s)

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

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

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

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

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

```
void SlidingWindow(int A[], int n, int K) {    // ii---or pair<int, int>---
represents the pair (A[i], i)    deque<ii> window; // we maintain 'window' to
be sorted in ascending order for (int i = 0; i < n; i++) { // this is O(n)
while (!window.empty() && window.back().first >= A[i])
window.pop_back(); // this to keep 'window' always sorted
window.push_back(ii(A[i], i));
// use the second field to see if this is part of the current window while
(window.front().second <= i - K) // lazy deletion
window.pop_front();
if (i + 1 >= K) // from the first window of length K onwards printf("%d\n",
window.front().first); // the answer for this window
}}
```

Sorting in Linear Time

Problem Description

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

Theoretical Limit

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

Solution(s)

Counting Sort

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

Prepare a 'frequency array' f with size k = R-L+1 and initialize f with zeroes. Ontheexamplearrayabove,wehaveL = 2,R = 5,andk = 4.

We do one pass through array A and update the frequency of each integer that we see, i.e. for each i ∈ [0..n-1], we do f[A[i]-L]++. On the example array above, we have f[0] = 3, f[1] = 2, f[2] = 0, f[3] = 1.

Once we know the frequency of each integers in that small range, we compute the prefix sums of each i, i.e. f[i] = [f-1] + f[i] ∀i ∈ [1..k-1]. Now, f[i] contains the number of elements less than or equal to i. On the example array above, we have f[0] = 3, f[1] = 5, f[2] = 5, f[3] = 6.

Next, go backwards from i = n-1 down to i = 0. We place A[i] at index f[A[i]-L]-1 as it is the correct location for A[i]. We decrement f[A[i]-L] by one so that the next copy of A[i]—if any—will be placed right before the current A[i]. On the example array above, we first put A[5] = 3 in index f[A[5]-2]-1 = f[1]-1 = 5-1 = 4 and decrement f[1] to 4. Next, we put A[4] = 3—the same value as A[5] = 3—now in index f[A[4]-2]-1 = f[1]-1 = 4-1 = 3 and decrement f[1] to 3. Then, we put A[3] = 2 in index f[A[3]-2]-1 = 2 and decrement f[0] to 2. We repeat the next three steps until we obtain a sorted array: {2, 2, 2, 3, 3, 5}.

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

Radix Sort

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

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

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

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

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

Tower of Hanoi

Problem Description

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

Solution(s)

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

Move n − 1 discs from peg A to peg B using peg C as the intermediate peg. After this recursive step is done, we are left with disc n by itself in peg A.

MovediscnfrompegAtopegC.

Move n − 1 discs from peg B to peg C using peg A as the intermediate peg. These n−1 discs will be on top of disc n which is now at the bottom of peg C.

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

```
#include <cstdio> using namespace std;
void solve(int count, char source, char destination, char intermediate) { if
(count == 1)
printf("Move top disc from pole %c to pole %c\n", source, destination); else
{
solve(count-1, source, intermediate, destination); solve(1, source,
destination, intermediate); solve(count-1, intermediate, destination, source);
} }
int main() {    solve(3, 'A', 'C', 'B'); // try larger value for the first parameter
} // return 0;
```

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