The key portion of the implementation is shown below:

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

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

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

Source code: `LCA.cpp/java`

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

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

Table 9.1: The Reduction from LCA to RMQ

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

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

Programming exercises related to LCA:

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

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

9.19 Magic Square Construction (Odd Size)

Problem Description

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

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

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

Solution(s)

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

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

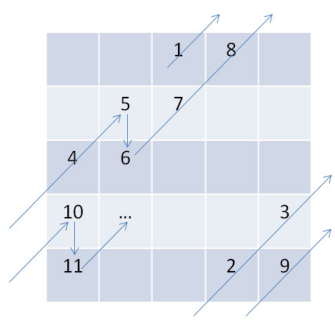


Figure 9.9: The Magic Square Construction Strategy for Odd n

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

Programming exercises related to Magic Square:

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

9.20 Matrix Chain Multiplication

Problem Description

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

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

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

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

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

Matrix Multiplication

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

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

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

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

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

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

Solution(s)

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

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

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

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

Programming exercises related to Matrix Chain Multiplication:

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

9.21 Matrix Power

Some Definitions and Sample Usages

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

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

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

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

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

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

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

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

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

---

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

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

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

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

The Idea of Efficient Exponentiation (Power)

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

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

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

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

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

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

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

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

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

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

Square Matrix Exponentiation (Matrix Power)

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

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

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

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

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

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

Source code: `UVa10229.cpp/java`

DP Speed-up with Matrix Power

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

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

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

$$fib(n+1) + fib(n) = fib(n+2)$$
$$fib(n) + fib(n-1) = fib(n+1)$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Programming Exercises related to Matrix Power:

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

9.22 Max Weighted Independent Set

Problem Description

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

Solution(s)

On Tree

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

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

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

On Bipartite Graph

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

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

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

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

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

9.23 Min Cost (Max) Flow

Problem Description

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

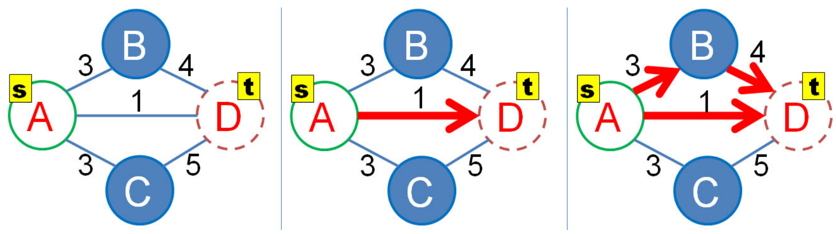


Figure 9.10: An Example of Min Cost Max Flow (MCMF) Problem (UVa 10594 [47])

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

Solution(s)

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

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

Programming exercises related to Min Cost (Max) Flow:

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

9.24 Min Path Cover on DAG

Problem Description

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

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

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

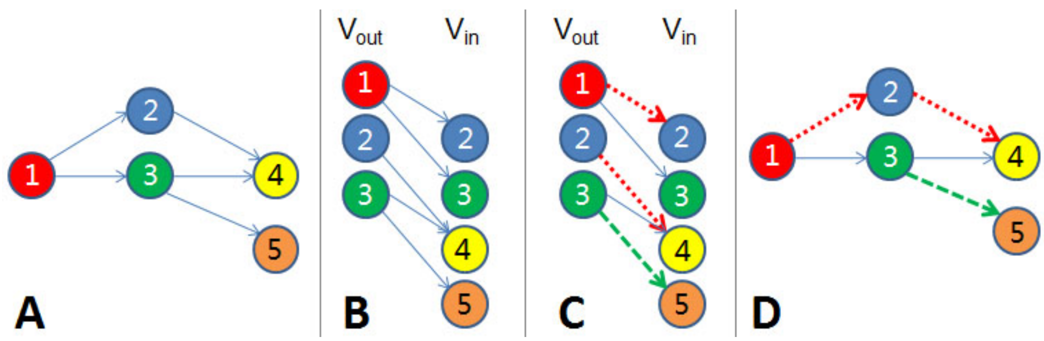


Figure 9.11: Min Path Cover on DAG (from UVa 1201 [47])

Solution(s)

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

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

Programming exercises related to Min Path Cover on DAG:

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

9.25 Pancake Sorting

Problem Description

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

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

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

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

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

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

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

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

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

Sample Test Cases

Sample Input

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

Sample Output

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

Explanation

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

Solution(s)

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

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

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

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

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

Programming exercises related to Pancake Sorting:

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

9.26 Pollard's rho Integer Factoring Algorithm

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

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

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

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

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

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

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

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

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

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

Source code: `Pollardsrho.cpp/java`

Programming exercises related to Pollard's rho algorithm:

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

9.27 Postfix Calculator and Conversion

Algebraic Expressions

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

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

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

Postfix Calculator

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

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

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

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

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

Table 9.3: Example of a Postfix Calculation

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

---

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

Infix to Postfix Conversion

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

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

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

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

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

Table 9.4: Example of an Execution of Shunting yard Algorithm

Programming exercises related to Postfix expression:

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

9.28 Roman Numerals

Problem Description

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

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

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

Solution(s)

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

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

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

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

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

Source code: `UVa11616.cpp/java`

Programming exercises related to Roman Numerals:

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

9.29 Selection Problem

Problem Description

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

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

Solution(s)

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

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

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

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

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

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

Expected $O(n)$ algorithm, static data

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

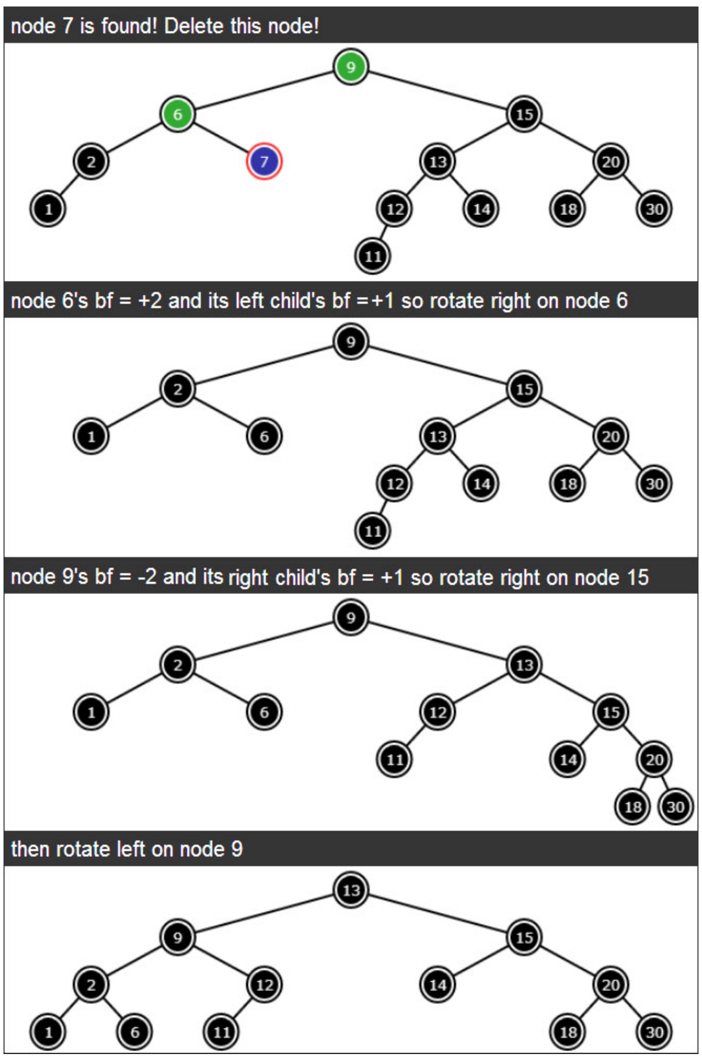


Figure 9.12: Example of an AVL Tree Deletion (Delete 7)

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

9.30 Shortest Path Faster Algorithm

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

SPFA requires the following data structures:

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

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

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

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

Source code: `UVa10986.cpp/java`

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

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

9.31 Sliding Window

Problem Description

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

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

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

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

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

Solution(s)

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

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

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

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

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

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

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

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

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

Programming exercises:

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

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

9.32 Sorting in Linear Time

Problem Description

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

Theoretical Limit

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

Solution(s)

Counting Sort

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

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

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

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

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

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

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

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

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

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

Radix Sort

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

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

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

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

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

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

Programming exercises related to Sorting in Linear Time:

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

9.33 Sparse Table Data Structure

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

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

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

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

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

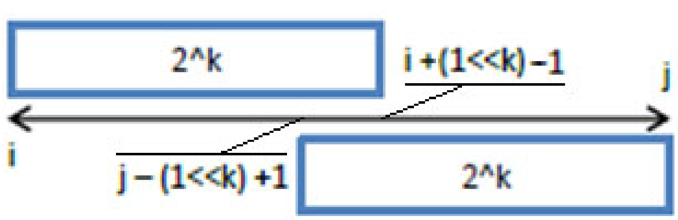


Figure 9.13: Explanation of `RMQ(i, j)`

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

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

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

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

Source code: `SparseTable.cpp/java`

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

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

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

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

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

9.34 Tower of Hanoi

Problem Description

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

Solution(s)

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

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

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

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

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

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

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

Programming exercises related to Tower of Hanoi:

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

9.35 Chapter Notes

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

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

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

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

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

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

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

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

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

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

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

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

Appendix A

uHunt

uHunt (`http://uhunt.felix-halim.net`) is a self-learning tool for UVa online-judge (UVa OJ [47]) created by one of the authors of this book (Felix Halim). The goal is to make solving problems at UVa OJ fun. It achieves the goal by providing:

1. Near real-time feedback and statistics on the recently submitted solutions so that the users can quickly iterate on improving their solutions (see Figure A.1). The users can immediately see the rank of their solutions compared to others in terms of performance. A (wide) gap between the user's solution performance with the best implies that the user still does not know a certain algorithms, data structures, or hacking tricks to get that faster performance. uHunt also has the 'statistics comparer' feature. If you have a *rival* (or a better UVa user that you admire), you can compare your list of solved problems with him/her and then try to solve the problems that your rival can solve.

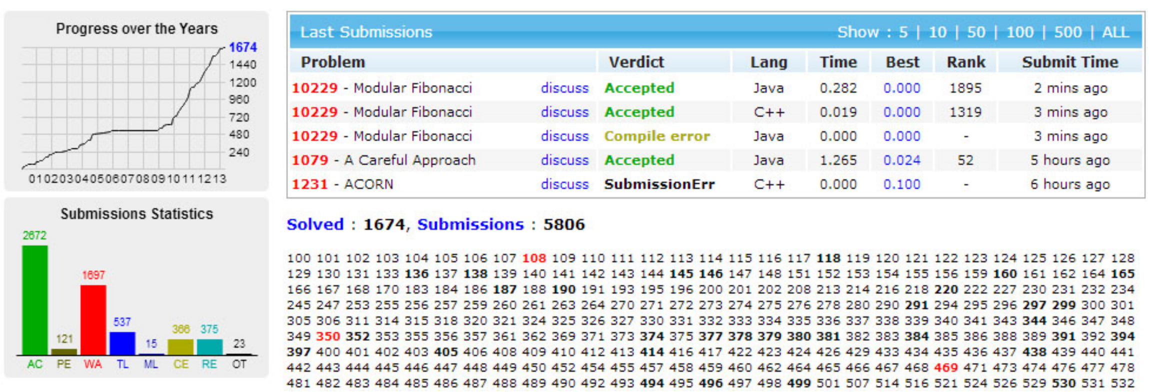


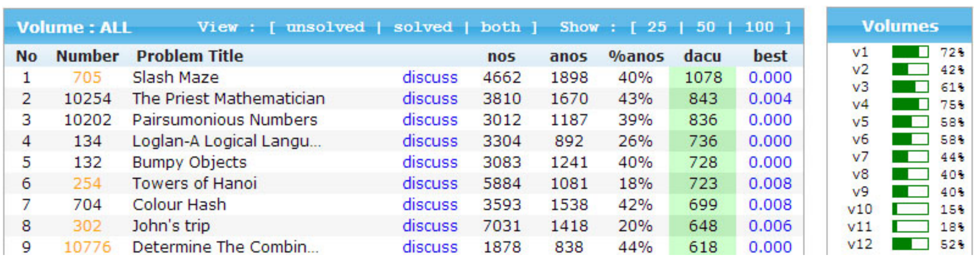
Figure A.1: Steven's statistics as of 24 May 2013

2. Web APIs for other developers to build their own tool. uHunt API has been used to create a full blown contest management system, a command line tool to submit solutions and get feedback through console, and mobile application to see the statistics.

3. A way for the users to help each others. The chat widget on the upper right corner of the page has been used to exchanges ideas and to help each other to solve problems. This gives a conducive environment for learning where user can always ask for help.

4. A selection of the next problems to solve, ordered by increasing difficulty (approximated by the number of distinct accepted users for the problems). This is useful for users who want to solve problems which difficulty matches their current skills. The rationale is this: If a user is still a beginner and he/she needs to build up his/her

confidence, he/she needs to solve problems with gradual difficulty. This is much better than directly attempting hard problems and keep getting non Accepted (AC) responses without knowing what's wrong. The ≈ 149008 UVa users actually contribute statistical information for each problem that can be exploited for this purpose. The easier problems will have higher number of submissions and higher number of AC. However, as a UVa user can still submit codes to a problem even though he/she already gets AC for that problem, then the number of AC alone is not an accurate measure to tell whether a problem is easy or not. An extreme example is like this: Suppose there is a hard problem that is attempted by a single good programmer who submits 50 AC codes just to improve his code's runtime. This problem is not easier than another easier problem where only 49 different users get AC. To deal with this, the default sorting criteria in uHunt is '`dacu`' that stands for '*distinct* accepted users'. The hard problem in the extreme example above only has `dacu` = 1 whereas the easier problem has `dacu` = 49 (see Figure A.3).


Volume : ALL   View : [unsolved | solved | both]   Show : [25 | 50 | 100]

No	Number	Problem Title		nos	anos	%anos	dacu	best
1	705	Slash Maze	discuss	4662	1898	40%	1078	0.000
2	10254	The Priest Mathematician	discuss	3810	1670	43%	843	0.004
3	10202	Pairsumonious Numbers	discuss	3012	1187	39%	836	0.000
4	134	Loglan-A Logical Langu...	discuss	3304	892	26%	736	0.000
5	132	Bumpy Objects	discuss	3083	1241	40%	728	0.000
6	254	Towers of Hanoi	discuss	5884	1081	18%	723	0.008
7	704	Colour Hash	discuss	3593	1538	42%	699	0.008
8	302	John's trip	discuss	7031	1418	20%	648	0.006
9	10776	Determine The Combin...	discuss	1878	838	44%	618	0.000

Volumes

Volume	%
v1	72%
v2	42%
v3	61%
v4	75%
v5	58%
v6	58%
v7	44%
v8	40%
v9	40%
v10	15%
v11	18%
v12	52%

Figure A.2: Hunting the next easiest problems using '`dacu`'

5. A means to create virtual contests. Several users can decide to create a closed contest among them over a set of problems, with a certain contest duration. This is useful for team as well as individual training. Some contests have shadows (i.e. contestants from the past), so that the users can compare their skills to the real contestants in the past.


World Finals Warmup I   Quick Submit

Last Submissions [hide last submissions]   Show : 5 | 10 | 20 | 50 | 100

#	Problem Title	Author Name	Verdict	Lang	Time	Best	Rank	Submit Time
545	12439 February 29	Ivan Reyes (spperman)	Time limit	Java	1.000	0.000	-1	5 secs ago
544	12439 February 29	Bulat S. (soul_rebel)	Wrong answer	C++	0.016	0.000	-1	10 secs ago
543	12435 Consistent Verdicts	Andrej Gajduk (Gajduk)	Wrong answer	Java	0.804	0.292	-1	12 secs ago
542	12439 February 29	Patrick Klitzke (philolo1)	Accepted	C++	0.012	0.000	-1	16 secs ago
541	12439 February 29	Mike Shvets (mike.shvets)	Wrong answer	C++	0.012	0.000	-1	16 secs ago

Contest Ranklist   past submissions   shadow users   Time remaining: 22 hours 8 minutes

#	Author Name	12433	12434	12435	12436	12437	12438	12439	12440	12441	12442	12443	AC / Time
1	panyuchao			0:40	(2) 1:16			(1) 1:11	1:44		0:52		5 / 6:46
2	surwdkgo			0:58				0:11			1:13	0:46	4 / 3:10
3	PMP Forever			0:19		(1) 1:35		0:38			1:05		4 / 3:58
4	Anton Raichuk			(1) 0:41				(1) 0:14	(1) --:--	(4) 1:15	0:35		4 / 4:47
5	Submitor			0:52		(3) 1:49		(1) 0:36			1:18		4 / 5:57

Figure A.3: We can rewind past contests with 'virtual contest'

6. An integration of ≈ 1675 programming exercises in this book from various categories (see Figure A.4). The users can keep track which programming exercises in this book that they have solved and see the progress of their work. These programming exercises can be used even without the book. Now, a user can customize his/her training programme to solve *problems of similar type*! Without such (manual) categorization, this training mode is hard to execute. We also give stars (*) to problems that we consider as **must try *** (up to 3 problems per category).

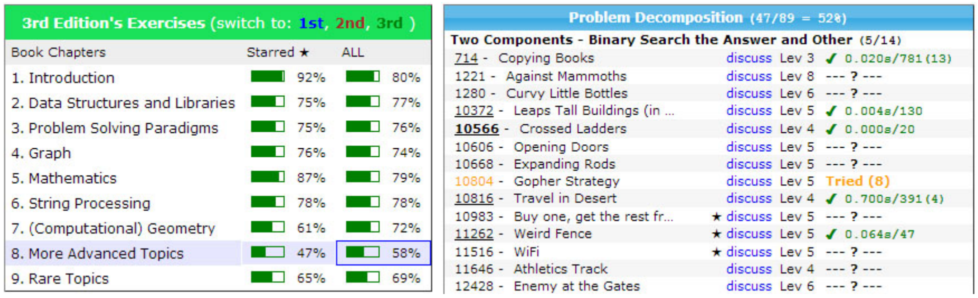


Figure A.4: The programming exercises in this book are integrated in uHunt

Building a web-based tool like uHunt is a computational challenge. There are over ≈ 11796315 submissions from ≈ 149008 users (≈ one submission every few seconds). The statistics and rankings must be updated frequently and such update must be fast. To deal with this challenge, Felix uses lots of advanced data structures (some are beyond this book), e.g. database cracking [29], Fenwick Tree, data compression, etc.

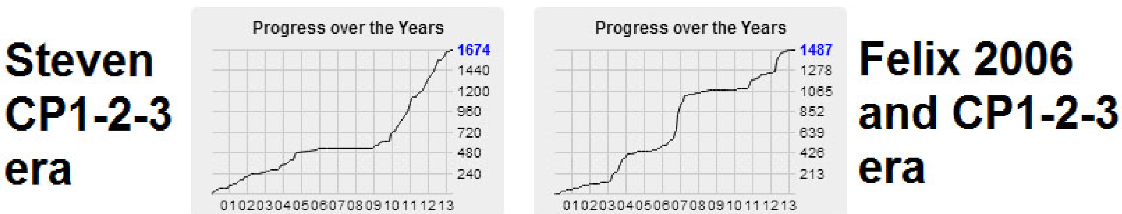


Figure A.5: Steven's & Felix's progress in UVa online judge (2000-present)

We ourselves are using this tool extensively in different stages of our life, as can be seen in Figure A.5. Two major milestones that can be seen from our progress chart are: Felix's intensive training to eventually won ACM ICPC Kaohsiung 2006 with his ICPC team (see Figure A.6) and Steven's intensive problem solving activities in the past four years (late 2009-present) to prepare this book.

Figure A.6: Andrian, Felix, and Andoko Won ACM ICPC Kaohsiung 2006

Appendix B

Credits

The problems discussed in this book are mainly taken from UVa online judge [47], ACM ICPC Live Archive [33], and past IOI tasks (mainly from 2009-2012). So far, we have contacted the following authors (and their current known affiliation as of 2013) to get their permissions (in alphabetical order):

1. Brian C. Dean (Clemson University, America)
2. Colin Tan Keng Yan (National University of Singapore, Singapore)
3. Derek Kisman (University of Waterloo, Canada)
4. Gordon V. Cormack (University of Waterloo, Canada)
5. Howard Cheng (University of Lethbridge, Canada)
6. Jane Alam Jan (Google)
7. Jim Knisely (Bob Jones University, America)
8. Jittat Fakcharoenphol (Kasetsart University, Thailand)
9. Manzurur Rahman Khan (Google)
10. Melvin Zhang Zhiyong (National University of Singapore, Singapore)
11. Michal (Misof) Forišek (Comenius University, Slovakia)
12. Mohammad Mahmudur Rahman (University of South Australia, Australia)
13. Norman Hugh Anderson (National University of Singapore, Singapore)
14. Ondřej Lhoták (University of Waterloo, Canada)
15. Petr Mitrichev (Google)
16. Piotr Rudnicki (University of Alberta, Canada)
17. Rob Kolstad (USA Computing Olympiad)
18. Rujia Liu (Tsinghua University, China)
19. Shahriar Manzoor (Southeast University, Bangladesh)
20. Sohel Hafiz (University of Texas at San Antonio, America)

21. Soo Yuen Jien (National University of Singapore, Singapore)

22. Tan Sun Teck (National University of Singapore, Singapore)

23. TopCoder, Inc (for PrimePairs problem in Section 4.7.4)

A compilation of photos with some of these problem authors that we managed to meet in person is shown below.

However, due to the fact that there are thousands ($\approx$ 1675) of problems listed and discussed in this book, there are many problem authors that we have not manage to contact yet. If you are those problem authors or know the person whose problems are used in this book, please notify us. We keep a more updated copy of this problem credits in our supporting website: `https://sites.google.com/site/stevenhalim/home/credits`

Bibliography

[1] Ahmed Shamsul Arefin. *Art of Programming Contest (from Steven's old Website)*. Gyankosh Prokashoni (Available Online), 2006.

[2] Wolfgang W. Bein, Mordecai J. Golin, Lawrence L. Larmore, and Yan Zhang. The Knuth-Yao Quadrangle-Inequality Speedup is a Consequence of Total-Monotonicity. *ACM Transactions on Algorithms*, 6 (1):17, 2009.

[3] Richard Peirce Brent. An Improved Monte Carlo Factorization Algorithm. *BIT Numerical Mathematics*, 20 (2):176–184, 1980.

[4] Brilliant. Brilliant.
https://brilliant.org/.

[5] Frank Carrano. *Data Abstraction and Problem Solving with C++: Walls and Mirrors*. Addison Wesley, 5th edition, 2006.

[6] Yoeng-jin Chu and Tseng-hong Liu. On the Shortest Arborescence of a Directed Graph. *Science Sinica*, 14:1396–1400, 1965.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithm*. MIT Press, 2nd edition, 2001.

[8] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw Hill, 2008.

[9] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2nd edition, 2000.

[10] Edsger Wybe Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[11] Yefim Dinitz. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Doklady Akademii nauk SSSR*, 11:1277–1280, 1970.

[12] Adam Drozdek. *Data structures and algorithms in Java*. Cengage Learning, 3rd edition, 2008.

[13] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal on Maths*, 17:449–467, 1965.

[14] Jack Edmonds and Richard Manning Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19 (2):248–264, 1972.

[15] Susanna S. Epp. *Discrete Mathematics with Applications*. Brooks-Cole, 4th edition, 2010.

[16] Fabian Ernst, Jeroen Moelands, and Seppo Pieterse. Teamwork in Prog Contests: 3 * 1 = 4.
http://xrds.acm.org/article.cfm?aid=332139.

[17] Project Euler. Project Euler.
http://projecteuler.net/.

[18] Peter M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24 (3):327–336, 1994.

[19] Robert W. Floyd. Algorithm 97: Shortest Path. *Communications of the ACM*, 5 (6):345, 1962.

[20] Michal Forišek. IOI Syllabus.
http://people.ksp.sk/~misof/ioi-syllabus/ioi-syllabus-2009.pdf.

[21] Michal Forišek. The difficulty of programming contests increases. In *International Conference on Informatics in Secondary Schools*, 2010.

[22] William Henry. Gates and Christos Papadimitriou. Bounds for Sorting by Prefix Reversal. *Discrete Mathematics*, 27:47–57, 1979.

[23] Felix Halim, Roland Hock Chuan Yap, and Yongzheng Wu. A MapReduce-Based Maximum-Flow Algorithm for Large Small-World Network Graphs. In *ICDCS*, 2011.

[24] Steven Halim and Felix Halim. Competitive Programming in National University of Singapore. In *A new learning paradigm: competition supported by technology*. Ediciones Sello Editorial S.L., 2010.

[25] Steven Halim, Roland Hock Chuan Yap, and Felix Halim. Engineering SLS for the Low Autocorrelation Binary Sequence Problem. In *Constraint Programming*, pages 640–645, 2008.

[26] Steven Halim, Roland Hock Chuan Yap, and Hoong Chuin Lau. An Integrated White+Black Box Approach for Designing & Tuning SLS. In *Constraint Programming*, pages 332–347, 2007.

[27] Steven Halim, Koh Zi Chun, Loh Victor Bo Huai, and Felix Halim. Learning Algorithms with Unified and Interactive Visualization. *Olympiad in Informatics*, 6:53–68, 2012.

[28] John Edward Hopcroft and Richard Manning Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2 (4):225–231, 1973.

[29] Stratos Idreos. *Database Cracking: Towards Auto-tuning Database Kernels*. PhD thesis, CWI and University of Amsterdam, 2010.

[30] TopCoder Inc. Algorithm Tutorials.
http://www.topcoder.com/tc?d1=tutorials&d2=alg_index&module=Static.

[31] TopCoder Inc. PrimePairs. Copyright 2009 TopCoder, Inc. All rights reserved.
http://www.topcoder.com/stat?c=problem_statement&pm=10187&rd=13742.

[32] TopCoder Inc. Single Round Match (SRM).
http://www.topcoder.com/tc.

[33] Competitive Learning Institute. ACM ICPC Live Archive. http://livearchive.onlinejudge.org/.

[34] IOI. International Olympiad in Informatics. http://ioinformatics.org.

[35] Giuseppe F. Italiano, Luigi Laura, and Federico Santaroni. Finding Strong Bridges and Strong Articulation Points in Linear Time. *Combinatorial Optimization and Applications*, 6508:157–169, 2010.

[36] Arthur B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5 (11):558562, 1962.

[37] Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted Longest-Common-Prefix Array. In *CPM, LNCS 5577*, pages 181–192, 2009.

[38] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison Wesley, 2006.

[39] Harold W. Kuhn. The Hungarian Method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[40] Anany Levitin. *Introduction to The Design & Analysis of Algorithms*. Addison Wesley, 2002.

[41] Rujia Liu. *Algorithm Contests for Beginners (In Chinese)*. Tsinghua University Press, 2009.

[42] Rujia Liu and Liang Huang. *The Art of Algorithms and Programming Contests (In Chinese)*. Tsinghua University Press, 2003.

[43] Udi Manbers and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22 (5):935–948, 1993.

[44] Gary Lee Miller. Riemann's Hypothesis and Tests for Primality. *Journal of Computer and System Sciences*, 13 (3):300–317, 1976.

[45] James Munkres. Algorithms for the Assignment and Transportation Problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[46] Institute of Mathematics and Lithuania Informatics. Olympiads in Informatics. http://www.mii.lt/olympiads_in_informatics/.

[47] University of Valladolid. Online Judge. http://uva.onlinejudge.org.

[48] USA Computing Olympiad. USACO Training Program Gateway. http://train.usaco.org/usacogate.

[49] Joseph O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, 1987.

[50] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 2nd edition, 1998.

[51] David Pearson. A polynomial-time algorithm for the change-making problem. *Operations Research Letters*, 33 (3):231–234, 2004.

[52] John M. Pollard. A Monte Carlo Method for Factorization. *BIT Numerical Mathematics*, 15 (3):331–334, 1975.

[53] George Pólya. *How to Solve It.* Princeton University Press, 2nd edition, 1957.

[54] Janet Prichard and Frank Carrano. *Data Abstraction and Problem Solving with Java: Walls and Mirrors.* Addison Wesley, 3rd edition, 2010.

[55] Michael Oser Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12 (1):128–138, 1980.

[56] Kenneth H. Rosen. *Elementary Number Theory and its Applications.* Addison Wesley Longman, 4th edition, 2000.

[57] Kenneth H. Rosen. *Discrete Mathematics and its Applications.* McGraw-Hill, 7th edition, 2012.

[58] Robert Sedgewick. *Algorithms in C++, Part 1-5.* Addison Wesley, 3rd edition, 2002.

[59] Steven S. Skiena. *The Algorithm Design Manual.* Springer, 2008.

[60] Steven S. Skiena and Miguel A. Revilla. *Programming Challenges.* Springer, 2003.

[61] SPOJ. Sphere Online Judge.
http://www.spoj.pl/.

[62] Wing-Kin Sung. *Algorithms in Bioinformatics: A Practical Introduction.* CRC Press (Taylor & Francis Group), 1st edition, 2010.

[63] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1 (2):146–160, 1972.

[64] Jeffrey Trevers and Stanley Milgram. An Experimental Study of the Small World Problem. *Sociometry*, 32 (4):425–443, 1969.

[65] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14 (3):249–260, 1995.

[66] Baylor University. ACM International Collegiate Programming Contest.
http://icpc.baylor.edu/icpc.

[67] Tom Verhoeff. 20 Years of IOI Competition Tasks. *Olympiads in Informatics*, 3:149166, 2009.

[68] Adrian Vladu and Cosmin Negruşeri. Suffix arrays - a programming contest approach. In *GInfo*, 2005.

[69] Henry S. Warren. *Hacker's Delight.* Pearson, 1st edition, 2003.

[70] Stephen Warshall. A theorem on Boolean matrices. *Journal of the ACM*, 9 (1):11–12, 1962.

[71] Wikipedia. The Free Encyclopedia.
http://en.wikipedia.org.

Index