# Code Snippets

**Randomize Array prior to using C# QuickSort**

//Randomize array to avoid QuickSort worst-case O(n^2) scenario for 300,000 items

Random rnd = new Random(Environment.TickCount);

int ra, temp;

for (int i = 0; i < n; i++)

{

ra = rnd.Next(0, i+1); //i+1 better than n

temp = a[i];

a[i] = a[ra];

a[ra] = temp;

}

Array.Sort(a);

**Unique Strings in Set (using List<> or Dictionary<,>)**

Dictionary<string, int> d = new Dictionary<string, int>();

//List<string> d = new List<string>();

for (int i = 1; i <= familyName.Length; i++)

for (int j = 1; j <= givenName.Length; j++)

{

string st = familyName.Substring(0, i) + givenName.Substring(0, j);

d[st] = 0;

//if (!d.Contains(st)) d.Add(st);

}

return d.Count;

**Define Array of Lists**

List<int>[] keysInChest = new List<int>[N];

for (i = 0; i < N; i++)

keysInChest[i] = new List<int>();

for (int j = 0; j < 5; j++)

keysInChest[i].Add(int.Parse(s[2 + j]));

**Define List of Arrays**

//Build a tree with List of Integer arrays

List<int[]> graph;

graph = new List<int[]>();

int degree = 5; //Number of levels in tree

for (int i = 0; i <= degree; i++)

graph.Add(new int[(int)Math.Pow(2, degree - i)]);

**SortedList has methods that use a binary search**

//Performance is improved when searching a "SortedList".

//Adding to a SortedList places item in sorted order.

//Methods "Contains" and "IndexOf" use a binary search algorithm.

static SortedList<int, int> pow = new SortedList<int, int>();

static void Insert(int num)

{

int ind = pow.IndexOfKey(num);

if (ind > -1)

{

pow.RemoveAt(ind);

Insert(num + 1);

}

else

pow[num] = 1;

}

**Dictionary**

Dictionary<string, int> d = new Dictionary<string, int>();

foreach (string s in sticker) d[s] = 0;

foreach (string s in sticker) d[s]++;

t = 0;

foreach (int i in d.Values) t += Math.Min(i, 2);

if (t >= 6) return "YES";

return "NO";

Dictionary<int, bool> d = new Dictionary<int, bool>();

**StringBuilder**

using System.Text;

StringBuilder s = new StringBuilder(paragraph);

int n = s.Length;

// s[0] = (char)('A' + (s[0] - 'a'));

// s[0] = char.ToUpper(s[0]);

s[0] = Convert.ToChar(s[0].ToString().ToUpper());

for (int i = 2; i < n; i++)

if (s[i-2] == '.')

s[i] = Convert.ToChar(s[i].ToString().ToUpper());

return s.ToString();

// char[] is vastly simpler and more efficient than StringBuilder,

// if you know a maximum or absolute size of your output string

//Clear Stringbuilder by setting Length = 0

s.Length = 0;

**CharArray**

char[] c = paragraph.ToCharArray();

c[0] = char.ToUpper(c[0]);

for (int i = 2; i < c.Length; i++)

if (c[i-2] == '.')

c[i] = char.ToUpper(c[i]);

return new string(c);

**String[] to Char[,] (2 dimensional)**

public int minimalSteps(string[] board)

{

r = board.Length;

c = board[0].Length;

char[,] bd = new char[r,c];

for (int i = 0; i < r; i++)

for (int j = 0; j < c; j++)

bd[i,j] = board[i][j];

**Returning 2 dimensional array**

return new int[] {i, p};

**String to CharArray & List<string> & CharArray to string**

List<string> f = new List<string>();

foreach (string st in S)

{

char[] a = st.ToCharArray();

Array.Sort(a);

string b = new string(a);

if (f.IndexOf(b) == -1) f.Add(b);

}

return f.Count;

**CharArray & Dictionary<string, bool> & string**

// \*\*\* Alternative way

Dictionary<string, bool> d = new Dictionary<string, bool>();

foreach (string st in S)

{

char[] a = st.ToCharArray();

Array.Sort(a);

string b = new string(a);

d[b] = true;

}

return d.Count;

**List<char> vs. Dictionary (number of unique chars in a string)**

List<char> a = new List<char>();

for (int i = 0; i < n; i++)

if (!a.Contains(bricks[i]))

a.Add(bricks[i]);

int cnt = a.Count;

Dictionary<char, bool> d = new Dictionary<char, bool>();

for (int i = 0; i < n; i++)

d[bricks[i]] = true;

int cnt = d.Count;

**Prime number checker**

static bool isPrime(int n)

{

if (n <= 1) return false;

if (n == 2) return true;

if (n % 2 == 0) return false;

for (int i = 3; i \* i <= n; i += 2)

if (n % i == 0) return false;

return true;

}

**Generate Prime numbers up to n**

static bool[] seive(int n)

{

//Return all primes between 0 and n using Seive of Eratosthenes

bool[] prime = new bool[n + 1];

for (int i = 0; i <= n; i++) prime[i] = true; //15% longer to initialize

prime[0] = false;

prime[1] = false;

for (int i = 2; i \* i <= n; i++)

if (prime[i])

for (int k = i \* i; k <= n; k += i)

prime[k] = false;

return prime;

}

**Power of 2**

Math.Pow(2, qn) // produces a type double

(1 << qn) // q:0 2^0 = 1, q:1 2^1 = 2, q:2 2^2 = 4, q:3 2^3 = 8

if ((it & (1 << qn)) == 0) // produces a type double

//Generate binary (0 or 1) for q digits (q=4, 0..15)

int[] d = new int[q];

int it2 = it;

for (int i = 0; i < q; i++)

{

d[i] = it2 % 2;

it2 /= 2;

}

public int[] getBinaryDigits(int num, int q)

{

//Generate binary (0 or 1) for num using q digits (q=4, 0..15)

//int can hold 32 bit (q = 32) (about 2 billion for num);

//long can hold 64 bit (q = 64) (about 9 quintillion for num)

int[] d = new int[q];

for (int i = 0; i < q; i++)

{

d[i] = num % 2;

num /= 2;

}

return d;

}

**Check all Combinations**

int[] b = {b1,b2,b3,b4};

for (int i = 0; i < 16; i++)

{

int k = i;

int sum = 0;

for (int j = 0; j < 4; j++)

{

//if ((i & (1 << j)) > 0) sum += b[j];

if (k % 2 == 1) sum += b[j];

k /= 2;

}

if (sum == price)

return "POSSIBLE";

}

return "IMPOSSIBLE";

**Distinct items**

string[] st = (Console.ReadLine().Split());

Console.WriteLine(4 - st.Distinct().Count());

//List<string> a = new List<string>();

//foreach (string s in st)

// if (a.IndexOf(s) == -1) a.Add(s);

//Console.WriteLine(4 - a.Count);

List<int> nums = new List<int>();

...

if (!nums.Contains(b)) nums.Add(b);

int cnt = nums.Count;

**Compare 2 strings**

string bday = birthdays[i].Substring(0, 5);

if (String.Compare(bday, date) >= 0) //bday >= date

return bday;

**Palindrome**

private bool IsPal(string word)

{

int n = word.Length;

for (int i = 0; i < n/2; i++)

if (word[i] != word[n - i - 1]) return false;

return true;

}

private static Boolean isPalidrome(int number)

{

int num = number, reverse = 0, digit;

while (num > 0)

{

digit = num % 10;

reverse = reverse \* 10 + digit;

num /= 10;

}

return (number == reverse);

}

**Assigning values to List**

List<long> col = new List<long>(new long[] { r, g, b }); //C# 2.0 syntax

//List<long> col = new List<long>(){ r, g, b }; //C# 3.0 syntax

List<string> name = new List<string>( new string[] {"RED","GREEN","BLUE"});

**Make List<string> into String array**

string[] s = new string[max + 1];

for (int i = 0; i <=max; i++)

for (int j = 0; j < 10; j++)

if (c[j] > i)

s[max-i] += "X";

else

s[max-i] += ".";

return s;

List<string> s = new List<string>();

for (int i = 0; i <= max; i++)

{

string t = "";

for (int j = 0; j < 10; j++)

if (c[j] > i)

t += "X";

else

t += ".";

s.Add(t);

return s.ToArray();

**Make List<int> into Int array**

List<int> ans = new List<int>();

:

return ans.ToArray();

**Make Int array into List<int>**

private int[] Union(int[] A, int[] B)

{

List<int> un = new List<int>(A); //Assign/Convert int[] to List<int>

for (int i = 0; i < B.Length; i++)

if (!un.Contains(B[i])) un.Add(B[i]);

un.Sort();

return un.ToArray();

}

**Concatenate Chars using StringBuilder (for better performance)**

// Time it takes to loop & concatenate/display 100,000 digits using:

// Write('0'); = 6.5 to 7.0 seconds **(only 0.1 in CodeForce)**

// string ns += "0"; = 6.6 to 7.0 seconds

// StringBuilder ns.Append("0") = 0.5 seconds

// Displaying takes 0.5 seconds from complete string

// Thus, actual concatenating of 0's: 6.1 vs. 0.012 seconds

// Thus, SB is 500 times faster than String concat

// Performance is best by using SB/string repeat function:

// StringBuilder - ns.Append('0', n - 1); = 0.002

// string - ns += new String('0', n - 1); = 0.003

**String repeater (Actually Char Repeater)** string sp = new String(' ', max - textIn[i].Length);

**Concatenate Strings using String Join**

//Concatenates the specified elements of a string array, using the

//specified separator between each element.

string[] result = new string[n];

for (long i = 0; i < n; i++)

result[i] = boxes[i].ToString();

return string.Join(" ", result);

**Concatenate Strings using String Join**

public int countStars(string[] result)

{

return String.Join("", result).Replace("-", "").Length;

}

**Days In Month**

//int[] mo = new int[] { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

//if (d < 1 || d > mo[m - 1]) return false;

if (d < 1 || d > DateTime.DaysInMonth(y, m)) return false;

int Days(int Month)

{

if(Month==2)

return 28;

if(Month==4||Month==6||Month==9||Month==11)

return 30;

return 31;

}

**Day # of Year**

DateTime dt = new DateTime(2013, m[i], d[i]);  
 doy2[i] = dt.DayOfYear - 1;

**Dictionary for dates, display max Date occurrence**

Dictionary<string, int> d = new Dictionary<string, int>();

for (int i = 0; i <= n - 10; i++)

{

string df = s.Substring(i, 10);

if (IsValidDate(df))

{

if (d.Keys.Contains(df))

d[df]++;

else

d[df] = 1; //d.Add(df, 1);

}

}

foreach (string key in d.Keys)

if (d[key] > max)

{

max = d[key];

maxDate = key;

}

Console.WriteLine(maxDate);

**Validate date**

// Alternatively, ParseExact will determine validity of a date

// but its performance is not as good as the above code.

try

{

DateTime dt2 = DateTime.ParseExact(dt, "dd-MM-yyyy", null);

}

catch

{

return false;

}

**Convert base 2 to base 10**

public long getBase10 (string b2)

{

//Convert base 2 b2 to base 10 num

long num = 0;

foreach (char ch in b2)

num = num \* 2 + (ch - '0');

return num;

}

**Determine if num is a power of 5**

public bool isPowOf5(long i)

{

while (i > 0 && i % 5 == 0)

i /= 5;

return (i == 1);

}

**Moving in Grid**

private bool IsDominant(string[] stone, int i, int j, int row, int col)

{

if (stone[i][j] == 'o') return true;

int[] dx = new int[] { 0, 0, 1, -1 };

int[] dy = new int[] { 1, -1, 0, 0 };

bool isDominant = true;

for (int k = 0; k < 4; k++)

{

int ni = i + dx[k];

int nj = j + dy[k];

if (ni < 0 || nj < 0 || ni >= row || nj >= col)

continue;

if (stone[ni][nj] == '.')

isDominant = false;

}

return isDominant;

}

**Round double up to Int, with 1e-9 precision**

return (int)Math.Ceiling(sum / tot - 1e-9);

**Convert ASCII to char**

char c = message[i];

if ((c >= 'A' && c <= 'M') || (c >= 'a' && c <= 'm'))

c = (char)(c + 13);

**Count binary 1s in Number (in base 2)**

private static int CountOfOnes(int num)

{

int cnt = 0;

while (num > 0)

{

cnt += num % 2;

num /= 2;

}

return cnt;

}

private static int CountOfOnes(int num)

{

int cnt = 0;

while (num > 0)

{

if ((num & 1) == 1) cnt++;

num >>= 1;

}

return cnt;

}

private static int CountOfOnes(int num)

{

int cnt = 0;

for (int i = 0; i < 20; i++)

if ((num & (1 << i)) > 0) cnt++;

return cnt;

}

private static int CountOfOnes(int num)

{

int cnt = 0;

while (num > 0)

{

cnt++;

num &= (num - 1);

}

return cnt;

}

**Check board positions in 8 directions from spot**

private bool IsSafe(int x, int y, string[] b)

{

//r = rows; c = columns

int[] dx = new int[]{0,1,1,1,0,-1,-1,-1};

int[] dy = new int[]{1,1,0,-1,-1,-1,0,1};

for (int i = 0; i < 8; i++)

{

if (x + dx[i] < 0 || x + dx[i] >= r ||

y + dy[i] < 0 || y + dy[i] >= c) continue;

if (b[x + dx[i]][y + dy[i]] == 'B') return false;

}

return true;

//if ((i+1 == r || b[i+1][j] == '.')

// && (i+1 == r || j+1 == c || b[i + 1][j+1] == '.')

// && (i+1 == r || j-1 < 0 || b[i + 1][j-1] == '.')

// && (j+1 == c || b[i][j+1] == '.')

// && (j-1 < 0 || b[i][j-1] == '.')

// && (i-1 < 0 || b[i-1][j] == '.')

// && (i-1 < 0 || j+1 == c || b[i-1][j+1] == '.')

// && (i-1 < 0 || j-1 < 0 || b[i-1][j-1] == '.'))

// return true;

//return false;

}

**Check board for 5 o's in 8 directions from spot**

int n, dx, dy;

public string win(string[] board)

{

//Check all 8 directions for 5 o's

n = board.Length;

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++)

for (dx = -1; dx <= 1; dx++)

for (dy = -1; dy <= 1; dy++)

{

if (dx == 0 && dy == 0) continue;

if (check(board, i,j)) return "found";

}

return "not found";

}

private bool check(string[] b, int x, int y)

{

int nx, ny;

for (int i = 0; i < 5; i++)

{

nx = x + dx\*i;

ny = y + dy\*i;

if (nx < 0 || nx >= n || ny < 0 || ny >= n || b[nx][ny] != 'o')

return false;

}

return true;

}

**Convert num from base b to Base 10**

private int ConvertToBase10(string s, int b)

{

int sum = 0;

foreach (char c in s)

sum = sum \* b + GetVal(c);

return sum;

}

private int GetVal(char c)

{

return "0123456789ABCDEFGHIJ".IndexOf(c);

//if (c >= '0' && c <= '9') return c - '0';

//else return c - 'A' + 10;

}

**Recursion with 2 ^ numLength options**

public class TennisRallies

{

int numLength, allowed;

string[] forbidden;

public int howMany(int numLength, string[] forbidden, int allowed)

{

this.numLength = numLength;

this.forbidden = forbidden;

this.allowed = allowed;

return CountGoodRallies(0, "");

}

private int CountGoodRallies(int fcount, string curr)

{

for (int i = 0; i < forbidden.Length; i++)

if (curr.EndsWith(forbidden[i])) fcount++;

if (fcount >= allowed) return 0;

if (curr.Length == numLength) return 1;

//return (fcount >= allowed) ? 0 : 1;

return CountGoodRallies(fcount, curr + "c")

+ CountGoodRallies(fcount, curr + "d");

}

}

**Using Masks with 2 ^ numLength options**

public int howMany(int numLength, string[] forbidden, int allowed)

{

int cnt = 0;

for (int i = 0; i < (1 << numLength); i++)

{

StringBuilder s = new StringBuilder();

for (int j = 0; j < numLength; j++)

if ((i & (1 << j)) == 0) s.Append("c");

else s.Append("d");

if (ok(s.ToString(), numLength, forbidden, allowed))

cnt++;

}

return cnt; ;

}

private bool ok(string s, int numLength, string[] forbidden, int allowed)

{

int fcount = 0;

for (int i = 0; i < forbidden.Length; i++)

{

int fl = forbidden[i].Length;

for (int j = 0; j < numLength; j++)

if (j + fl <= numLength

&& s.Substring(j, fl) == forbidden[i])

{

fcount++;

if (fcount >= allowed) return false;

}

}

return true;

}

**Raising b to the power of n modular m (big number)**

long num = GetBigPowerMod(3, n, m);

static long GetBigPowerMod(long b, int n, int m)

{

//num will be b^n mod m; for n, m up to 1e10

//long num = 1

//for (int i = 0; i < n; i++)

// num = (num \* b) % m;

long num = 1;

while (n > 0)

{

if (n % 2 == 1)

num = (num \* b) % m;

b = (b \* b) % m;

n /= 2;

//Console.WriteLine("{0} {1} {2}", num, b, n);

}

return num;

}

**Override Sort Comparison function**

public int[] arrange(int[] numbers)

{

//The best method is always to use the libraries available

//and use an existing sort implementation that allows you

//to override the comparison function.

Array.Sort<int>(numbers, new Comparison<int>(Compare));

return numbers;

}

public int Compare(int a, int b)

{

int ac = CountOfOnes(a);

int bc = CountOfOnes(b);

if (ac < bc) return -1;

if (ac > bc) return 1;

if (a < b) return -1;

if (a > b) return 1;

return 0;

}

**Copy Array to allow changes, and refresh with original data later**

int[] students

int n = students.Length;

int[] studs = new int[n];

Array.Copy(students, studs, n);

**Minimum in single array (row) and single array (col)**

Console.WriteLine(Math.Min(row.Min(), col.Min()));

**Moving in direction on x-y coodinate**

//Start facing North, then turn 90 degrees to the right

int[] dx = new int[] { 0, 1, 0, -1 };

int[] dy = new int[] { 1, 0, -1, 0 };

public int getdist(int T, int[] a)

{

int x = 0, y = 0, dir = 0;

int n = a.Length;

for (int t = 0; t < T; t++)

for (int i = 0; i < n; i++)

{

x += dx[dir] \* a[i];

y += dy[dir] \* a[i];

//x += (int)Math.Cos(dir \* Math.PI / 2) \* a[i];

//y += (int)Math.Sin(dir \* Math.PI / 2) \* a[i];

dir = (dir + a[i]) % 4;

}

return Math.Abs(x) + Math.Abs(y);

}

**GCD and LCM**

//Slow for really big ints

//static int GCD(int a, int b)

//{

// for (int i = Math.Min(a, b); i >= 1; i--)

// if (a % i == 0 && b % i == 0)

// return i;

// return 0;

//}

//GCD\_Euclid

static int GCD(int a, int b)

{

if (b == 0) return a;

return GCD(b, a % b);

}

static int LCM(int a, int b)

{

return a \* b / GCD(a, b);

}

//calculate the gcd using the Euclidean algorithm

int gcd(int a, int b) {

while(b!=0) {

int c = a;

a = b;

b = c%b;

}

return a;

}

//Needed to define as long, since some int did not work??? (see Sept 2013 CodeChef Matrix)

static long GCD(long a, long b)

{

if (b == 0) return a;

return GCD(b, a % b);

}

**Get next Permutation of characters**

bool NextPerm(char[] perm, int n)

{

int i, j;

for (i = n - 1; i > 0; i--)

if (perm[i - 1] < perm[i])

break;

if (i < 1) return false;

for (j = n - 1; j >= 0; j--)

if (perm[i - 1] < perm[j])

break;

{

char t = perm[i - 1];

perm[i - 1] = perm[j];

perm[j] = t;

}

for (int k = (n + i + 1) / 2; k < n; k++)

{

char t = perm[k];

perm[k] = perm[n + i - 1 - k];

perm[n + i - 1 - k] = t;

}

return true;

}

**Recursion - Counting unique Factorizations of n (e.g. 12 = 2\*2\*3, 2\*6, 3\*4)**

public int refactor(int n)

{

return count(n, 2);

}

private int count(int n, int lastFactor)

{

int sum = 0;

for (int i = lastFactor; i \* i <= n; i++)

if (n % i == 0)

sum += 1 + count(n / i, i);

return sum;

}

**Resizing an Array**

int m = Math.Min(50, n);

Array.Resize(ref s, m);

return s;

//Alternate way to resize array by creating another

string[] ns = new string[m];

for (int i = 0; i < m; i++)

ns[i] = s[i];

return ns;

**Precision for Double (eps = 1e-15)**

double d = Math.Abs(dx / dy - da / db);

//Need to round at 1e15 to avoid imprecision at 16th decimal in double

// Otherwise (7/6 - 1/1) > (4/3 - 7/6)

// 0.16666666666666674 > 0.16666666666666652

//d = Math.Floor(d \* 1e15) / 1e15;

double eps = 1e-15; //Need to round at 15th digit for double (15 digits precise)

if (d < min - eps)

{

min = d;

mina = a;

minb = b;

}

**Shift Long 1 (retaining bits for Long size)**

1L << i;

(ulong) 1 << i;

**Reverse string efficiently**

private string Reverse(string text)

{

char[] chars = text.ToCharArray();

Array.Reverse(chars);

return new string(chars);

}

**Reverse integer**

private int Reverse(int x)

{

int r = 0;

while (x > 0)

{

r = r \* 10 + x % 10;

x /= 10;

}

return r;

}

**Check if number is a Square**

static bool IsSquare(long d)

{

long x = (long)Math.Sqrt(d + 0.1);

return x \* x == d;

}

**Binary Search with check for overflow of long number using double**

long maxLong = long.MaxValue; //9e18

while (left < right)

{

n = (left + right + 1) / 2;

sum = n \* (n + 1) \* 2 + n \* inc;

double sumd = (double)n \* (n + 1) \* 2 + n \* inc;

if (sumd < maxLong && sum <= t)

left = n;

else

right = n - 1;

}

**Add MOD if ans became big then 0 then subtracted 1**

ans = (ans\*2) % MOD;

ans--;

if (ans < 0) ans += MOD;

//ans = (ans - 1 + MOD) % MOD; //Add MOD in case ans = 0 - 1

**String Replace**

string s = Console.ReadLine();

string[] a = { "144", "14", "1"};

for (int i = 0; i < 3; i++)

while (s.Contains(a[i]))

s = s.Replace(a[i], "-");

**String Replace**

string s = goal.Replace("z","");

//string s = "";

//for (int i = 0; i < goal.Length; i++)

// if (goal[i] != 'z') s += goal[i];

**Shortest Path**

//Get Shortest Path from i to j; initially graph[i,j] is

// 0 (to self), 1 (connection), or INF (no connection)

//Floyd Warshall algorithm

for (int k = 0; k < n; k++)

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++)

graph[i,j] = Math.Min(graph[i,j],

graph[i,k] + graph[k,j]);

//Find maximum length of all shortest paths

int max = 0;

for (int i = 0; i < n; i++)

for (int j = 0; j < i; j++)

{

if (graph[i,j] >= INF) return -1;

max = Math.Max(max, graph[i,j]);

}

**String Format implied in WriteLine**

Console.WriteLine(i + " " + j + " "+ k);

Console.WriteLine("{0} {1} {2}", i, j, k);

**Reading and Writing Double**

using System.Globalization; //for CultureInfo: 1.234 = 1,234

a = double.Parse(s[i], CultureInfo.InvariantCulture);

d = a - (int)a;

//if (s[i].EndsWith(".000")) //Check if a is integer

if (d < 1e-6)

zero++;

//overload ToString and display 3 decimals

Console.WriteLine(min.ToString("F3", CultureInfo.InvariantCulture));