USACO GOLD

Student's Book

DO NOT SHARE

STAR LEAGUE

STAR LEAGUE INC

Copyright© by *Star League Inc*. All rights reserved.

DO NOT SHARE

Dear USACO Gold Contestant

You just passed USACO Silver level and proved that you have the ability of algorithmical thinking and you can implement basic algorithms like binary search, recursion, DFS, BFS etc. and apply these algorithms on solutions of the problems.

Now you have a USACO Gold path in front of you. Knowing the guidelines and your level on the track can help and motivate you to study and finish the USACO Gold content easily. Here is an overall guideline of the USACO Gold curriculum.

1. USACO Silver Review
2. Bitset & Binary operations
3. Data Structure
    a. Stack, Queue, Vector, Set, Map, PriorityQueue- Gold Level Applications
4. Graph Theory
    a. DFS-BFS on graphs
    b. Topological Sort
    c. Minimum Spanning Tree(Prim)
    d. Shortest Path-(Dijkstra, Bellman-Ford)
    e. All Shortest Paths(Floyd-Warshall)
5. Dynamic Programming
    a. Introduction
    b. Basic Problems(Longest Increasing Subsequence, Maximum Subarray Sum)
    c. Knapsack, Coin Change, Subset Sum
    d. DP on Strings Longest Common Sub-sequence, Edit Distance
    e. Number Game
6. Introductory Geometric Algorithms
    a. Points, Line, Slope, Intersection, In-out of shapes, Polygon area
7. Greedy Problems
    a. USACO Gold Greedy Problems

StarLeague wishes the best on your journey to the USACO Platinum level.

DO NOT SHARE

Dynamic Programming

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again. Following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming.

1) Optimal Substructure
2) Overlapping Subproblems

1) Optimal Substructure:

A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

For example, the Shortest Path problem has following optimal substructure property:

If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v. The standard All Pair Shortest Path algorithms like Floyd–Warshall and Bellman–Ford are typical examples of Dynamic Programming.

On the other hand, the Longest Path problem doesn't have the Optimal Substructure property. Here by Longest Path we mean longest simple path (path without cycle) between two nodes. Consider the following unweighted graph. There are two longest paths from q to t: q→r→t and q→s→t. Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path q→r→t is not a combination of longest path from q to r and longest path from r to t, because the longest path from q to r is q→s→t→r and the longest path from r to t is r→q→s→t.

Typically, a greedy algorithm is used to solve a problem with optimal substructure if it can be proved by induction that this is optimal at each step. Otherwise, provided the problem exhibits overlapping subproblems as well, dynamic programming is used.

2) Overlapping Subproblems:

Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn't have common subproblems. If we take example of following recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.

DO NOT SHARE

Program for Fibonacci numbers

The Fibonacci numbers are the numbers in the following integer sequence.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..

In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation

Fn = Fn-1 + Fn-2

with seed values F0 = 0 and F1 = 1.

Given a number n, print n-th Fibonacci Number.

Input : n = 2
Output : 1

Input : n = 9
Output : 34

Write a function *int fib(int n)* that returns F_n. For example, if *n* = 0, then *fib()* should return 0. If n = 1, then it should return 1. For n > 1, it should return $F_{n-1} + F_{n-2}$

Following are different methods to get the nth Fibonacci number.

Method 1 (Use recursion)

A simple method that is a direct recursive implementation mathematical recurrence relation given above.

```
/* simple recursive program for Fibonacci numbers */
int fib(int n)
{
   if ( n <= 1 )
      return n;
   return fib(n-1) + fib(n-2);
}
```

Time Complexity: T(n) = T(n-1) + T(n-2) which is exponential.

We can observe that this implementation does a lot of repeated work (see the following recursion tree). So this is a bad implementation for nth Fibonacci number.

DO NOT SHARE

Recursion tree for execution of fib(5)

```
                        fib(5)
                     /          \
               fib(4)             fib(3)
             /      \             /     \
         fib(3)     fib(2)      fib(2)    fib(1)
        /     \       /    \     /    \
  fib(2)   fib(1)  fib(1) fib(0) fib(1) fib(0)
  /    \
fib(1) fib(0)
```

Extra Space: O(n) if we consider the function call stack size, otherwise O(1).

We can see that the function f(3) is being called 2 times. If we would have stored the value of f(3), then instead of computing it again, we could have reused the old stored value.

Method 2 (Use Dynamic Programming)

We can avoid the repeated work done is the method 1 by storing the Fibonacci numbers calculated so far.

There are following two different ways to store the values so that these values can be reused:
a) Memoization (Top Down)
b) Tabulation (Bottom Up)

a) Memoization (Top Down): The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise we calculate the value and put the result in lookup table so that it can be reused later.

Following is the memoized version for nth Fibonacci Number.

```c
/* C/C++ program for Memoized version for nth Fibonacci number */
#include<stdio.h>
#define NIL -1
#define MAX 100

int lookup[MAX];

/* Function to initialize NIL values in lookup table */
void _initialize()
{
  int i;
  for (i = 0; i < MAX; i++)
    lookup[i] = NIL;
}

/* function for nth Fibonacci number */
int fib(int n)
{
   if (lookup[n] == NIL)
   {
      if (n <= 1)
         lookup[n] = n;
      else
         lookup[n] = fib(n-1) + fib(n-2);
   }

   return lookup[n];
}
```

DO NOT SHARE

b) Tabulation (Bottom Up): The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from table. For example, for the same Fibonacci number, we first calculate fib(0) then fib(1) then fib(2) then fib(3) and so on. So literally, we are building the solutions of subproblems bottomup.

Following is the tabulated version for nth Fibonacci Number.

```
/* C program for Tabulated version */
#include<stdio.h>
int fib(int n)
{
  int f[n+1];
  int i;
  f[0] = 0;   f[1] = 1;
  for (i = 2; i <= n; i++)
      f[i] = f[i-1] + f[i-2];

  return f[n];
}

int main ()
{
  int n = 9;
  printf("Fibonacci number is %d ", fib(n));
  return 0;
}
```

Time Complexity: O(n)

Extra Space: O(n)

Both Tabulated and Memoized store the solutions of subproblems. In Memoized version, table is filled on demand while in Tabulated version, starting from the first entry, all entries are filled one by one.

Method 3 (Space Optimized Method 2)

We can optimize the space used in method 2 by storing the previous two numbers only because that is all we need to get the next Fibonacci number in series.

```
// Fibonacci Series using Space Optimized Method
int fib(int n)
{
  int a = 0, b = 1, c, i;
  if( n == 0)
    return a;
  for (i = 2; i <= n; i++)
  {
     c = a + b;
     a = b;
     b = c;
  }
  return b;
}
```

Time Complexity: O(n)

Extra Space: O(1)

DO NOT SHARE

Tribonacci Numbers

The tribonacci series is a generalization of the Fibonacci sequence where each term is the sum of the three preceding terms.

The Tribonacci Sequence :

0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149, 274, 504, 927, 1705, 3136, 5768, 10609, 19513, 35890, 66012, 121415, 223317, 410744, 755476, 1389537, 2555757, 4700770, 8646064, 15902591, 29249425, 53798080, 98950096, 181997601, 334745777, 615693474, 1132436852… so on

General Form of Tribonacci number:

a(n) = a(n-1) + a(n-2) + a(n-3) with a(0) = a(1) = 0, a(2) = 1.

Given a value N, task is to print first N Tribonacci Numbers.

Examples:

Input : 5
Output : 2

Input : 10
Output : 44

Input : 20
Output : 19513

Exercise 1: A **simple solution** is to simply follow recursive formula and write recursive code for it.

```
int printTribRec(int n)
{
    if (n == 0 || n == 1 || n == 2)
        return 0;

    if (n == 3)
        return 1;
    else
        return printTribRec(n - 1) +
               printTribRec(n - 2) +
               printTribRec(n - 3);
}
```

Time complexity of this solution is exponential.

Exercise 2: A **better solution** is to use Dynamic Programming. Write a solution with O(n) complexity.

DO NOT SHARE

```
int printTrib(int n)
{
    int dp[n];
    dp[0] = dp[1] = 0;
    dp[2] = 1;

    for (int i = 3; i < n; i++)
        dp[i] = dp[i - 1] +
                dp[i - 2] +
                dp[i - 3];

    for (int i = 0; i < n; i++)
        cout << dp[i] << " ";
}
```

Time complexity of above is linear, but it requires extra space.

Exercise 3: We can **optimizes space** used in above solution using three variables to keep track of previous three numbers. Write a solution with O(1) memory complexity.

```
void printTrib(int n)
{
    if (n<=2) return 0;
    if (n<=3) return 1;

    // Initialize first hree numbers
    int first = 0, second = 0, third = 1;
    int curr;

    for (int i = 3; i < n; i++)
    {
        curr = first + second + third;
        first = second;
        second = third;
        third = curr;
    }
    return curr;
}
```

DO NOT SHARE

Ugly Numbers (Goldman-Sachs Interview question)

Ugly numbers are numbers whose only prime factors are 2, 3 or 5. The sequence 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, … shows the first 11 ugly numbers. By convention, 1 is included.
Given a number n, the task is to find n'th Ugly number.

Input : n = 7
Output : 8

Input : n = 10
Output : 12

Input : n = 15
Output : 24

Input : n = 150
Output : 5832

Method 1 (Simple)
Loop for all positive integers until ugly number count is smaller than n, if an integer is ugly than increment ugly number count.
To check if a number is ugly, divide the number by greatest divisible powers of 2, 3 and 5, if the number becomes 1 then it is an ugly number otherwise not.

For example, let us see how to check for 300 is ugly or not. Greatest divisible power of 2 is 4, after dividing 300 by 4 we get 75. Greatest divisible power of 3 is 3, after dividing 75 by 3 we get 25. Greatest divisible power of 5 is 25, after dividing 25 by 25 we get 1. Since we get 1 finally, 300 is ugly number.

```
/*This function divides a by greatest divisible  power of b*/
int maxDivide(int a, int b)
{
  while (a%b == 0)
   a = a/b;
  return a;
}

/* Function to check if a number is ugly or not */
int isUgly(int no)
{
  no = maxDivide(no, 2);
  no = maxDivide(no, 3);
  no = maxDivide(no, 5);

  return (no == 1)? 1 : 0;
}
/* Function to get the nth ugly number*/
int getNthUglyNo(int n)
{
  int i = 1;
  int count = 1;    /* ugly number count */

  /*Check for all integers untill ugly count becomes n*/
  while (n > count)
  {
    i++;
    if (isUgly(i))
      count++;
  }
  return i;
}
```

DO NOT SHARE

This method is not time efficient as it checks for all integers until ugly number count becomes n, but space complexity of this method is O(1)

Method 2 (Use Dynamic Programming)

Here is a time efficient solution with O(n) extra space.

The ugly-number sequence is 1, 2, 3, 4, 5, 6, 8, 9, 10,12, 15, …because every number can only be divided by 2, 3, 5, one way to look at the sequence is to split the sequence to three groups as below:

(1) 1×2, 2×2, 3×2, 4×2, 5×2, …

(2) 1×3, 2×3, 3×3, 4×3, 5×3, …

(3) 1×5, 2×5, 3×5, 4×5, 5×5, …

We can find that every subsequence is the ugly-sequence itself (1, 2, 3, 4, 5, …) multiply 2, 3, 5. Then we use similar merge method as merge sort, to get every ugly number from the three subsequences. Every step we choose the smallest one, and move one step after.

```
1   Declare an array for ugly numbers:  ugly[n]
2   Initialize first ugly no:  ugly[0] = 1
3  Initialize three array index variables i2, i3, i5 to point to    1st element of the ugly array:   i2 = i3 = i5 =0;
4   Initialize 3 choices for the next ugly no:
          next_mulitple_of_2 = ugly[i2]*2;        next_mulitple_of_3 = ugly[i3]*3        next_mulitple_of_5 = ugly[i5]*5;
5   Now go in a loop to fill all ugly numbers till 150:
        For (i = 1; i < 150; i++ )
        {
          next_ugly_no  = Min(next_mulitple_of_2, next_mulitple_of_3, next_mulitple_of_5);
          ugly[i] =  next_ugly_no
          if (next_ugly_no  == next_mulitple_of_2) {
                    i2 = i2 + 1;
                     next_mulitple_of_2 = ugly[i2]*2;
          }
          if (next_ugly_no  == next_mulitple_of_3) {
                    i3 = i3 + 1;
                  next_mulitple_of_3 = ugly[i3]*3;
           }
          if (next_ugly_no  == next_mulitple_of_5){
                i5 = i5 + 1;
                next_mulitple_of_5 = ugly[i5]*5;
           }
        }/* end of for loop */
6.return next_ugly_no
```

Example:

Let us see how it works

initialize ugly[] = | 1 | i2 = i3 i5 = 0;

First iteration

ugly[1] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5) = Min(2, 3, 5) = 2

ugly[] = | 1 | 2 | i2 = 1, i3 = i5 = 0 (i2 got incremented)

Second iteration

ugly[2] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5) = Min(4, 3, 5) = 3

ugly[] = | 1 | 2 | 3 | i2 = 1, i3 = 1, i5 = 0 (i3 got incremented)

Third iteration

ugly[3] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5) = Min(4, 6, 5) = 4

ugly[] = | 1 | 2 | 3 | 4 | i2 = 2, i3 = 1, i5 = 0 (i2 got incremented)

DO NOT SHARE

Fourth iteration

ugly[4] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5) = Min(6, 6, 5) = 5

ugly[] = | 1 | 2 | 3 | 4 | 5 | i2 = 2, i3 = 1, i5 = 1 (i5 got incremented)

Fifth iteration

ugly[4] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5) = Min(6, 6, 10) = 6

ugly[] = | 1 | 2 | 3 | 4 | 5 | 6 | i2 = 3, i3 = 2, i5 = 1 (i2 and i3 got incremented)

```
/* Function to get the nth ugly number*/
unsigned getNthUglyNo(unsigned n)
{
    unsigned ugly[n]; // To store ugly numbers
    unsigned i2 = 0, i3 = 0, i5 = 0;
    unsigned next_multiple_of_2 = 2;
    unsigned next_multiple_of_3 = 3;
    unsigned next_multiple_of_5 = 5;
    unsigned next_ugly_no = 1;

    ugly[0] = 1;
    for (int i=1; i<n; i++)
    {
       next_ugly_no = min(next_multiple_of_2,
                          min(next_multiple_of_3,
                              next_multiple_of_5));
       ugly[i] = next_ugly_no;
       if (next_ugly_no == next_multiple_of_2)
       {
           i2 = i2+1;
           next_multiple_of_2 = ugly[i2]*2;
       }
       if (next_ugly_no == next_multiple_of_3)
       {
           i3 = i3+1;
           next_multiple_of_3 = ugly[i3]*3;
       }
       if (next_ugly_no == next_multiple_of_5)
       {
           i5 = i5+1;
           next_multiple_of_5 = ugly[i5]*5;
       }
    } /*End of for loop (i=1; i<n; i++) */
    return next_ugly_no;
}
```

After 1 2 3 5 the sequence will create it self as multiples of itself with 2,3 and 5. So in each run decide next ugly number by choosing the min of next_mult2, next_mult3 and next_mult5.

At first all of them has index=0; next_mult_X = index_X *X; //X multiplied by next element on the sequence.

DO NOT SHARE

Algorithmic Paradigm: Dynamic Programming

Time Complexity: O(n)

Auxiliary Space: O(n)

Riddle till next lesson

You are given two eggs, and access to a 100-storey building. Both eggs are identical. The aim is to find out the highest floor from which an egg will not break when dropped out of a window from that floor. If an egg is dropped and does not break, it is undamaged and can be dropped again. However, once an egg is broken, that's it for that egg.

If an egg breaks when dropped from floor n, then it would also have broken from any floor above that. If an egg survives a fall, then it will survive any fall shorter than that.

The question is: What strategy should you adopt to minimize the number egg drops it takes to find the solution?. (And what is the worst case for the number of drops it will take?)

Hint 1: How to solve if you have only one egg?

Hint 2: Think recursively.

Hint 3: Try to solve for 10, then 15 then 20 or 21 floors.

References:

https://en.wikipedia.org/wiki/Optimal_substructure

https://www.geeksforgeeks.org/dynamic-programming-set-1/

https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/

http://www.youtube.com/watch?v=V5hZoJ6uK-s

https://www.youtube.com/watch?v=OQ5jsbhAv_M

DO NOT SHARE

Riddle

Wednesday, December 6, 2017 11:48 AM

Let' try to solve the problem for 10 floors.

6 floors

3rd floor → 1,2 gets broken [1,2]/2

3rd floor → survives [4-6]/3 → 5th floor

5th floor → 4th floor gets broken [4]/1

5th floor → survives [6]/1 → 6th floor

10 floors

4th floor → 1,2,3 gets broken [1-3]/3

4th floor → survives [5-10]/6 → 7th floor

7th floor → 5,6 gets broken [5-6]/2

7th floor → survives [8-10]/3 → 9th floor

9th floor → 8 gets broken [8]/1

9th floor → survives [10]/1 → 10

15 floors

5th floor → 1,2,3,4 gets broken [1-4]/4

5th floor → survives [6-15]/10 ← We already know how to solve the question when we have 2 eggs and 10 floors

DO NOT SHARE

General Solution

1st attempt

k^{th} floor

$$\underbrace{1 \quad . \quad . \quad . \quad k-1}_{\text{if gets broken}} \quad k \quad \underbrace{k+1 \qquad\qquad\qquad\qquad n}_{\text{survives}}$$

if gets broken

1 egg $[1..k-1]$ search

try $1..k-1$

survives

2 eggs $[k+1 \ldots n]$ search area.

$\Leftarrow$ find in $k-1$ steps.

2nd attempt

$(k+k-1)^{th}$ floor

$$\underbrace{k+1 \ldots\ldots \quad k+k-2}_{\text{gets broken}} \quad k+k-1 \quad \underbrace{k+k \ldots\ldots\ldots\ldots \quad n}_{\text{survives}}$$

gets broken

1 egg $[k+1 \quad k+k-2]$

$k-2$ search space.

survives

2 eggs $[k+k..n]$

$\Leftarrow$ find in $k-2$ steps.

DO NOT SHARE

Min Cost Path - 2D DP

Given a cost matrix cost[][] and a position (m, n) in cost[][], write a function that returns cost of minimum cost path to reach (m, n) from (0, 0). Each cell of the matrix represents a cost to traverse through that cell. Total cost of a path to reach (m, n) is sum of all the costs on that path (including both source and destination). You can only traverse down, right and diagonally lower cells from a given cell, i.e., from a given cell (i, j), cells (i+1, j), (i, j+1) and (i+1, j+1) can be traversed. You may assume that all costs are positive integers.

1	2	3
4	8	2
1	5	3

For example, in the following figure, what is the minimum cost path to (2, 2)?

The path with minimum cost is highlighted in the following figure. The path is (0, 0) –> (0, 1) –> (1, 2) –> (2, 2). The cost of the path is 8 (1 + 2 + 2 + 3).

Optimal Substructure

The path to reach (m, n) must be through one of the 3 cells: (m-1, n-1) or (m-1, n) or (m, n-1). So minimum cost to reach (m, n) can be written as "minimum of the 3 cells plus cost[m][n]".

minCost(m, n) = min (minCost (m-1, n-1),

minCost (m-1, n),

minCost (m, n-1))

+ cost[m][n]

1	2	3
4	8	2
1	5	3

```
/* Returns cost of minimum cost path from (0,0) to (m, n) in mat[R][C]*/
int minCost(int cost[R][C], int m, int n)
{
   if (n < 0 || m < 0)
      return INT_MAX;
   else if (m == 0 && n == 0)
      return cost[m][n];
   else
      return cost[m][n] + min( minCost(cost, m-1, n-1),
                               minCost(cost, m-1, n),
                               minCost(cost, m, n-1) );
}

/* A utility function that returns minimum of 3 integers */
int min(int x, int y, int z)
{
   if (x < y)
      return (x < z)? x : z;
   else
      return (y < z)? y : z;
}
```

DO NOT SHARE

Overlapping Subproblems

Following is simple recursive implementation of the MCP (Minimum Cost Path) problem. The implementation simply follows the recursive structure mentioned above.

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, there are many nodes which apear more than once. Time complexity of this naive recursive solution is exponential, and it is terribly slow.

```
mC refers to minCost()
                                mC(2, 2)
                        /          |          \
                       /           |           \
               mC(1, 1)          mC(1, 2)            mC(2, 1)
             /     |     \      /     |     \          /     |     \
            /      |      \    /      |      \        /      |      \
      mC(0,0) mC(0,1) mC(1,0) mC(0,1) mC(0,2) mC(1,1) mC(1,0) mC(1,1) mC(2,0)
```

So the MCP problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array tc[][] in bottom up manner.

```
int minCost(int cost[R][C], int m, int n)
{
     int i, j;

     // Instead of following line, we can use int tc[m+1][n+1] or
     // dynamically allocate memoery to save space. The following line is
     // used to keep te program simple and make it working on all compilers.
     int tc[R][C];

     tc[0][0] = cost[0][0];

     /* Initialize first column of total cost(tc) array */
     for (i = 1; i <= m; i++)
        tc[i][0] = tc[i-1][0] + cost[i][0];

     /* Initialize first row of tc array */
     for (j = 1; j <= n; j++)
        tc[0][j] = tc[0][j-1] + cost[0][j];

     /* Construct rest of the tc array */
     for (i = 1; i <= m; i++)
        for (j = 1; j <= n; j++)
            tc[i][j] = min(tc[i-1][j-1],
                           tc[i-1][j],
                           tc[i][j-1]) + cost[i][j];

     return tc[m][n];
}
```

Time Complexity of the DP implementation is O(mn) which is much better than Naive Recursive implementation.

DO NOT SHARE

Binomial Coefficient

Following are common definition of Binomial Coefficients.

1) A binomial coefficient C(n, k) can be defined as the coefficient of X^k in the expansion of (1 + X)^n.
2) A binomial coefficient C(n, k) also gives the number of ways, disregarding order, that k objects can be chosen from among n objects; more formally, the number of k-element subsets (or k-combinations) of an n element set.

The Problem

Write a function that takes two parameters n and k and returns the value of Binomial Coefficient C(n, k). For example, your function should return 6 for n = 4 and k = 2, and it should return 10 for n = 5 and k = 2.

Optimal Substructure

The value of C(n, k) can be recursively calculated using following standard formula for Binomial Coefficients.

C(n, k) = C(n-1, k-1) + C(n-1, k)
C(n, 0) = C(n, n) = 1

Following is a simple recursive implementation that simply follows the recursive structure mentioned above.

```
// A Naive Recursive Implementation
#include<stdio.h>

// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k)
{
  // Base Cases
  if (k==0 || k==n)
    return 1;

  // Recur
  return  binomialCoeff(n-1, k-1) + binomialCoeff(n-1, k);
}
```

Overlapping Subproblems

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for n = 5 an k = 2. The function C(3, 1) is called two times. For large values of n, there will be many common subproblems.

```
                    C(5, 2)
              /                    \
      C(4, 1)                        C(4, 2)
       /   \                        /          \
  C(3, 0)   C(3, 1)            C(3, 1)              C(3, 2)
           /    \              /     \              /     \
    C(2, 0)    C(2, 1)      C(2, 0) C(2, 1)         C(2, 1)  C(2, 2)
              /       \             /   \            /    \
         C(1, 0)  C(1, 1)      C(1, 0)  C(1, 1)   C(1, 0)  C(1, 1)
```

Since same subproblems are called again, this problem has Overlapping Subproblems property. So the Binomial Coefficient problem has both properties of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, re-computations of same subproblems can be avoided by constructing a temporary array C[][] in bottom up manner. Following is Dynamic Programming based implementation.

DO NOT SHARE

```c
// A Dynamic Programming based solution that uses table C[][] to
// calculate the Binomial Coefficient
#include<stdio.h>

// Prototype of a utility function that returns minimum of two integers
int min(int a, int b);

// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k)
{
    int C[n+1][k+1];
    int i, j;

    // Caculate value of Binomial Coefficient in bottom up manner
    for (i = 0; i <= n; i++)
    {
        for (j = 0; j <= min(i, k); j++)
        {
            // Base Cases
            if (j == 0 || j == i)
                C[i][j] = 1;

            // Calculate value using previosly stored values
            else
                C[i][j] = C[i-1][j-1] + C[i-1][j];
        }
    }

    return C[n][k];
}

/* Drier program to test above function*/
int main()
{
    int n = 5, k = 2;
    printf ("Value of C(%d, %d) is %d ", n, k, binomialCoeff(n, k) );
    return 0;
}
```

Time Complexity: O(n*k)

Auxiliary Space: O(n*k)

References:

http://www.csl.mtu.edu/cs4321/www/Lectures/Lecture%2015%20%20Dynamic%20Programming%20Binomial%20Coefficients.htm
https://www.geeksforgeeks.org/dynamic-programming-set-9-binomial-coefficient/

DO NOT SHARE

Binomial Coefficient

Wednesday, January 24, 2018 2:57 PM

$C(n,k)$ number of way disregarding order that k objects can be chosen from among n objects. (Number of k element subsets)

Opt. Substructure

k elements from n elements.

$$C(n,k) = \underbrace{C(n-1,k-1)}_{\text{choose the } n^{th} \text{ element}} + \underbrace{C(n-1,k)}_{\text{do not choose } n^{th} \text{ element}}$$

$$C(n,0) = C(n,n) = 1$$

```
int Comb(n, k)
{
    if k=0 or k=n   return 1
    return Comb(n-1, k-1) + Comb(n-1, k);
}
```

Overlapping Subproblems

$\Rightarrow$ Problem can be solved recursively & it has overlapping subproblems.

$\Rightarrow$ DP

Memoization

```
int f(n,k)
{   if k=0 or k=n   sol[n,k]=1
    if !sol[n,k]
        sol[n,k] = f(n-1,k)+f(n-1,k-1)
    return sol[n,k]

}
```

Tabulation

```
for i: 0 -> n
    for j: 0 -> min(i,k)
        if j=0 or j=i
            C[i][j] = 1
        else
            C[i][j] = C[i-1][j] + C[i-1][j-1]
```

DO NOT SHARE

The Triangle IOI'94

```
      7
     3 8
    8 1 0
   2 7 4 4
  4 5 2 6 5
```
(Figure 1)

Figure 1 shows a number triangle. Write a program that calculates the highest sum of numbers passed on a route that starts at the top and ends somewhere on the base.

- Each step can go either diagonally down to the left or diagonally down to the right.
- The number of rows in the triangle is >1 but <=100.
- The numbers in the triangle, all integers, are between 0 and 9

Find the highest sum value

$$sol(i,j) = \begin{cases} \max\{sol(i+1,j),\ sol(i+1,j+1)\} + mat[i,j] & \text{otherwise} \\ mat[i,j] & i=n \text{ (Base case)} \end{cases}$$

```
int sol (i,j)
{ if i==n   return mat[i,j]
  return sol(i+1,j) + sol(i,j+1)
}
```

(i,j) depends on i+1, j and i+1, j+1

⇒ answers to i^{th} row can be calculated if $i+1^{th}$ row is known.

a) What is the answer for the last row?

Same as values.

```
7
3 8
8 1 0
2 7 4 4
4 5 2 6 5
```

```
30
23 21
20 13 10
7 12 10 10
4 5 2 6 5
```

set n^{th} row to original values.

for row: n-1 → 1

sol[row, col] = mat[row, col] + max(,)

col: 0 → ?

DO NOT SHARE

Longest Increasing Subsequence

Let us discuss Longest Increasing Subsequence (LIS) problem as an example problem that can be solved using Dynamic Programming.

The Longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order. For example, the length of LIS for {10, 22, 9, 33, 21, 50, 41, 60, 80} is 6 and LIS is {10, 22, 33, 50, 60, 80}.

arr[ ]	10	22	9	33	21	50	41	60	80
LIS	1	2		3		4		5	6

More Examples:
Input : arr[] = {3, 10, 2, 1, 20}
Output : Length of LIS = 3
The longest increasing subsequence is 3, 10, 20

Input : arr[] = {3, 2}
Output : Length of LIS = 1
The longest increasing subsequences are {3} and {2}

Input : arr[] = {50, 3, 10, 7, 40, 80}
Output : Length of LIS = 4
The longest increasing subsequence is {3, 7, 40, 80}

Optimal Substructure:

Let arr[0..n-1] be the input array and L(i) be the length of the LIS ending at index i such that arr[i] is the last element of the LIS.

Then, L(i) can be recursively written as:

L(i) = 1 + max(L(j)) where 0 < j < i and arr[j] < arr[i]; or
L(i) = 1, if no such j exists.

To find the LIS for a given array, we need to return max(L(i)) where 0 < i < n.

Thus, we see the LIS problem satisfies the optimal substructure property as the main problem can be solved using solutions to subproblems.
Following is a simple recursive implementation of the LIS problem. It follows the recursive structure discussed above.

```
check (start, nmatches, biggest) {
  int better, i, best=nmatches;
  for (i = start; i < n; i++) {
      if (sequence[i] > biggest) {
          better = check (i, nmatches+1, sequence[i]);
          if (better > best) best = better;
      }
  }
  return best;
}

check (0,0,-1);
```

DO NOT SHARE

Overlapping Subproblems:

Considering the above implementation, following is recursion tree for an array of size 4. lis(n) gives us the length of LIS for arr[].

```
              lis(4)
        /        |
      lis(3)    lis(2)   lis(1)
     /           /
   lis(2) lis(1) lis(1)
   /
lis(1)
```

We can see that there are many subproblems which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation. Following is a tabulated implementation for the LIS problem.

```
int main()
{
    int n,a[20001],dp[20001];
//    ifstream fin("lis.in");
    cin>>n;
    for (int i=0;i<n;i++)
        cin>>a[i];
    dp[0]=0; int lis=0;
    for (int i=0;i<n;i++) {
        dp[i]=1;
        for (int j=0;j<i;j++){
            if (a[i]>a[j])
                dp[i]=max(dp[j]+1,dp[i]);
        }
        if (dp[i]>lis) lis=dp[i];
    }
    cout<<lis<<endl;
    return 0;
}
```

Output:
Length of lis is 5

Note that the time complexity of the above Dynamic Programming (DP) solution is O(N^2) and there is a O(NLogN) solution for the LIS problem.

DO NOT SHARE

Building Bridges (LIS Variations 1)

Consider a 2-D map with a horizontal river passing through its center. There are n cities on the southern bank with x-coordinates a(1) … a(n) and n cities on the northern bank with x coordinates b(1) … b(n). You want to connect as many north-south pairs of cities as possible with bridges such that no two bridges cross. When connecting cities, you can only connect city a(i) on the northern bank to city b(i) on the southern bank. Maximum number of bridges that can be built to connect north-south pairs with the aforementioned constraints.

```
8     1     4     3     5     2     6     7
<---- Cities on the other bank of river---->
--------------------------------------------
  <--------------- River--------------->
--------------------------------------------
1     2     3     4     5     6     7     8
<------- Cities on one bank of river------->
```

The values in the upper bank can be considered as the northern x-coordinates of the cities and the values in the bottom bank can be considered as the corresponding southern x-coordinates of the cities to which the northern x-coordinate city can be connected.

Examples:

```
Input : 6 4 2 1
        2 3 6 5
Output : Maximum number of bridges = 2
Explanation: Let the north-south x-coordinates
be written in increasing order.

1  2  3  4  5  6
  \  \
   \  \        For the north-south pairs
    \  \       (2, 6) and (1, 5)
     \  \      the bridges can be built.
      \  \     We can consider other pairs also,
       \  \    but then only one bridge can be built
        \  \   because more than one bridge built will
         \  \  then cross each other.
          \  \
1  2  3  4  5  6

Input : 8 1 4 3 5 2 6 7
        1 2 3 4 5 6 7 8
Output : Maximum number of bridges = 2
```

DO NOT SHARE

Approach: It is a variation of LIS problem. The following are the steps to solve the problem.

1. Sort the north-south pairs based on increasing order of south x-coordinates.
2. If two south x-coordinates are same, then sort on the basis of increasing order of north coordinates.
3. Now find the Longest Increasing Subsequence of the north x-coordinates.
4. One thing to note that in the increasing subsequence a value can be greater as well as can be equal to its previous value.

We can also sort based on north x-coordinates and find the LIS on the south x coordinates.

```
// north-south coordinates of each City Pair
struct CityPairs
{
    int north, south;
};

// comparison function to sort the given set of CityPairs
bool compare(struct CityPairs a, struct CityPairs b)
{
    if (a.south == b.south)
        return a.north < b.north;
    return a.south < b.south;
}

// function to find the maximum number of bridges that can be built
int maxBridges(struct CityPairs values[], int n)
{
    int lis[n];
    for (int i=0; i<n; i++)
        lis[i] = 1;

    sort(values, values+n, compare);

    // logic of longest increasing subsequence
    // applied on the northern coordinates
    for (int i=1; i<n; i++)
        for (int j=0; j<i; j++)
            if (values[i].north >= values[j].north && lis[i] < 1 + lis[j])
                lis[i] = 1 + lis[j];

    int max = lis[0];
    for (int i=1; i<n; i++)
        if (max < lis[i])
            max = lis[i];

    // max number of bridges that can be built
    return max;
}
```

Time Complexity: $O(N^2)$

References:

https://www.geeksforgeeks.org/dynamic-programming-building-bridges/
https://people.cs.clemson.edu/~bcdean/dp_practice/

DO NOT SHARE

Box Stacking Problem (LIS Variation-2)

You are given a set of n types of rectangular 3-D boxes, where the ith box has height h(i), width w(i) and depth d(i) (all integers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

The Box Stacking problem is a variation of LIS problem. We need to build a maximum height stack.

Following are the key points to note in the problem statement:

1. A box can be placed on top of another box only if both width and depth of the upper placed box are smaller than width and depth of the lower box respectively.
2. We can rotate boxes such that width is smaller than depth. For example, if there is a box with dimensions {1x2x3} where 1 is height, 2×3 is base, then there can be three possibilities, {1x2x3}, {2x1x3} and {3x1x2}
3. We can use multiple instances of boxes. What it means is, we can have two different rotations of a box as part of our maximum height stack.

Following is the **solution** based on DP solution of LIS problem.

1) Generate all 3 rotations of all boxes. The size of rotation array becomes 3 times the size of original array.

For simplicity, we consider depth as always smaller than or equal to width.

2) Sort the above generated 3n boxes in decreasing order of base area.

3) After sorting the boxes, the problem is same as LIS with following optimal substructure property.

MSH(i) = Maximum possible Stack Height with box i at top of stack

MSH(i) = { Max (MSH(j)) + height(i) } where j < i and width(j) > width(i) and depth(j) > depth(i). If there is no such j then MSH(i) = height(i)

4) To get overall maximum height, we return max(MSH(i)) where 0 < i < n

Following is the implementation of the above solution.

DO NOT SHARE

```
struct Box
{ // h --> height, w --> width, d --> depth
  int h, w, d;  // for simplicity of solution, always keep w <= d
};

//Compare function based on Base areas
bool compare1(struct Box a, struct Box  b)
{
    return (( b.d * b.w ) < ( a.d * a.w )) ;
}

/* Returns the height of the tallest stack that can be
   formed with give type of boxes */
int maxStackHeight( Box arr[], int n )
{
   /* Create an array of all rotations of given boxes
      For example, for a box {1, 2, 3}, we consider three
      instances{{1, 2, 3}, {2, 1, 3}, {3, 1, 2}} */
   Box rot[3*n];
   int index = 0;
   for (int i = 0; i < n; i++)
   {
      // Copy the original box
      rot[index].h = arr[i].h;
      rot[index].d = max(arr[i].d, arr[i].w);
      rot[index].w = min(arr[i].d, arr[i].w);
      index++;
      // First rotation of box
      rot[index].h = arr[i].w;
      rot[index].d = max(arr[i].h, arr[i].d);
      rot[index].w = min(arr[i].h, arr[i].d);
      index++;
      // Second rotation of box
      rot[index].h = arr[i].d;
      rot[index].d = max(arr[i].h, arr[i].w);
      rot[index].w = min(arr[i].h, arr[i].w);
      index++;
   }
   // Now the number of boxes is 3n
   n = 3*n;

   /* Sort the array 'rot[]' in non-increasing order of base area */
    sort(rot,rot+n,compare1);

   /* Initialize msh values for all indexes
      msh[i] --> Maximum possible Stack Height with box i on top */
   int msh[n];
   for (int i = 0; i < n; i++ )
      msh[i] = rot[i].h;

   /* Compute optimized msh values in bottom up manner */
   for (int i = 1; i < n; i++ )
      for (int j = 0; j < i; j++ )
         if ( rot[i].w < rot[j].w && rot[i].d < rot[j].d && msh[i] < msh[j] + rot[i].h)
         {
              msh[i] = msh[j] + rot[i].h;
         }

   /* Pick maximum of all msh values */
   int max = -1;
   for ( int i = 0; i < n; i++ )
      if ( max < msh[i] )
         max = msh[i];

   return max;
}
```

DO NOT SHARE

In the above program, given input boxes are {4, 6, 7}, {1, 2, 3}, {4, 5, 6}, {10, 12, 32}. Following are all rotations of the boxes in decreasing order of base area.

10 x 12 x 32	4 x 6 x 7	5 x 4 x 6	1 x 2 x 3
12 x 10 x 32	4 x 5 x 6	7 x 4 x 6	2 x 1 x 3
32 x 10 x 12	6 x 4 x 7	6 x 4 x 5	3 x 1 x 2

The height 60 is obtained by boxes { {3, 1, 2}, {1, 2, 3}, {6, 4, 5}, {4, 5, 6}, {4, 6, 7}, {32, 10, 12}, {10, 12, 32}} in format {h,w,d}

Time Complexity: O(n^2)

Auxiliary Space: O(n)

Asked in: Amazon, Codenation, Microsoft

References:

https://people.cs.clemson.edu/~bcdean/dp_practice/
https://www.geeksforgeeks.org/dynamic-programming-set-21-box-stacking-problem/

DO NOT SHARE

Subset Sum

Monday, February 26, 2018 7:33 PM

Given a set of non negative integers, and a value sum determine if there is a subset with sum equal to given sum.

set[] = {3, 4, 1, 12, 5, 2}, sum = 9. Set[] = {4, 7, 1, 9, 3}, sum = 6

Output : True Output: False

Naive Approach: Generate all subsets and calculate sum.

$O(2^n \cdot n)$ too costly.

∅	0
3	0 3
4	0 3 4 7
1	0 3 4 7 1 4 5 8
5	0 3 4 7 1 4 5 8 5 8 9 12 6 9 10 13

0
0 3
0 3 4 7
0 1 3 4 5 7 8
0 1 3 4 5 6 7 8 9 10 12 13

For each element there are 2 different subset sums.

1) Subset sums without this element
2) Subset sums with this element.

Let SubsetSum(n, X) return
True : if first n elements have a subset with sum equal to X.
False : if among first n elements no subset with sum equal to X.

We can reach to sum X either without using n^{th} element or using n^{th} element

$\Rightarrow$ SubsetSum(n, X) = SubsetSum(n-1, X) || SubsetSum(n-1, X-v[n])

q) What is base case?

with empty set, we can reach only sum 0. $\Rightarrow$ SubsetSum(0,0) = True;
SubsetSum(0,x) = False $\forall x \neq 0$

elements \ sums	0	1	2	3	4	5	6	7	8	9	10
∅	1	0	0	0	0	0	0	0	0	0	0
3	1	0	0	1	0	0	0	0	0	0	0
4	1	0	0	1	1	0	0	1	0	0	0
1	1	1	0	1	1	1	0	1	1	0	0
5	1	1	0	1	1	1	1	1	1	1	1

$f(i,j) = f(i-1, j) \,||\, f(i-1, j-v[i])$

$f(0,0) = T$

DO NOT SHARE

q) Bottom up?

$$dp[0,x] = \begin{cases} True & x = 0 \\ False & x \neq 0 \end{cases}$$

$$dp[i,j] = dp[i-1,j] \;||\; dp[i-1, j-v[i]];$$

q) What is the obj. value?

Can we reach sum value with a subset of n elements $\Rightarrow$ $dp[n,sum]$

q) What if you can use the same element more than once?

$$dp[n,sum] = \underbrace{dp[n-1,sum]}_{\text{don't use this element}} \;||\; \underbrace{dp[n-1,sum-v[n]]}_{\text{use this element once}} \;||\; \underbrace{dp[n,sum-v[n]]}_{\text{use this element again}}$$

q) One of 3 is unnecessary. which one?

if $dp[i,j]$ is True $\Rightarrow$ $dp[i+1,j]$ is also true;

$\Rightarrow$ we don't need to check $dp[n-1,sum-v[n]]$

if that is True, we will get it from $dp[n,sum-v[n]]$

DO NOT SHARE

Coin Change

Given a value N, if we want to make change for N cents, and we have infinite supply of each of S = {S1, S2, .., Sm} valued coins, how many ways can we make the change? The order of coins doesn't matter.

For example, for N = 4 and S = {1,2,3}, there are four solutions: {1,1,1,1},{1,1,2},{2,2},{1,3}. So, output should be 4. For N = 10 and S = {2, 5, 3, 6}, there are five solutions: {2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}.

So the output should be 5.

Optimal Substructure

To count total number solutions, we can divide all set solutions in two sets.

1) Solutions that do not contain mth coin (or Sm).
2) Solutions that contain at least one Sm.

Let count(S[], m, n) be the function to count the number of solutions, then it can be written as sum of count(S[], m-1, n) and count(S[], m, n-Sm).

Therefore, the problem has optimal substructure property as the problem can be solved using solutions to subproblems.

Overlapping Subproblems

Following is a simple recursive implementation of the Coin Change problem. The implementation simply follows the recursive structure mentioned above.

```
int count(int S[], int m, int n )
{
    // If n is 0 then there is 1 solution
    // (do not include any coin)
    if (n == 0) return 1;
    // If n is less than 0 then no solution exists
    if (n < 0)
       return 0;
    // If there are no coins and n
    // is greater than 0, then no
    // solution exist
    if (m <=0 && n >= 1)
        return 0;
    // count is sum of solutions (i)
    // including S[m-1] (ii) excluding S[m-1]
    return count( S, m - 1, n ) + count( S, m, n-S[m-1] );
}
```

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for S = {1, 2, 3} and n = 5.

The function C({1}, 3) is called two times. If we draw the complete tree, then we can see that there are many subproblems being called more than once.

```
C() --> count()
                          C({1,2,3}, 5)
                        /              \
                      /                  \
           C({1,2,3}, 2)                 C({1,2}, 5)
          /      \                      /      \
         /        \                    /        \
C({1,2,3}, -1)  C({1,2}, 2)       C({1,2}, 3)    C({1}, 5)
             /    \            /    \           /     \
            /      \          /      \         /       \
    C({1,2},0)  C({1},2)   C({1,2},1) C({1},3)    C({1}, 4)  C({}, 5)
                  / \     / \        /\         /     \
                 /   \   /   \     /   \       /       \
               .     .  .     .   .     .   C({1}, 3) C({}, 4)
                                            / \
                                           /   \
                                          .     .
```

DO NOT SHARE

Since same subproblems are called again, this problem has Overlapping Subproblems property. So the Coin Change problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array table[][] in bottom up manner.

Dynamic Programming Solution

```
int count( int S[], int m, int n )
{
    int i, j, x, y;
    // We need n+1 rows as the table is constructed
    // in bottom up manner using the base case 0value case (n = 0)
    int table[n+1][m];

    // Fill the enteries for 0 value case (n = 0)
    for (i=0; i<m; i++)
        table[0][i] = 1;

    // Fill rest of the table entries in bottom up manner
    for (i = 1; i < n+1; i++){
        for (j = 0; j < m; j++){
            // Count of solutions including S[j]
            x = (i-S[j] >= 0)? table[i - S[j]][j]: 0;

            // Count of solutions excluding S[j]
            y = (j >= 1)? table[i][j-1]: 0;

            // total count
            table[i][j] = x + y;
        }
    }
    return table[n][m-1];
}
```

Time Complexity: O(mn)

Following is a simplified version of method 2. The auxiliary space required here is O(n) only.

```
int count( int S[], int m, int n )
{
    // table[i] will be storing the number of solutions for
    // value i. We need n+1 rows as the table is constructed
    // in bottom up manner using the base case (n = 0)
    int table[n+1];

    // Initialize all table values as 0
    memset(table, 0, sizeof(table));

    // Base case (If given value is 0)
    table[0] = 1;

    // Pick all coins one by one and update the table[] values
    // after the index greater than or equal to the value of the
    // picked coin
    for(int i=0; i<m; i++)
        for(int j=S[i]; j<=n; j++)
            table[j] += table[j-S[i]];
    return table[n];
}
```

References:

http://www.algorithmist.com/index.php/Coin_Change
https://www.geeksforgeeks.org/dynamic-programming-set-7-coin-change/

DO NOT SHARE

Dividing the Gold

Tuesday, February 27, 2018 1:07 PM

$1 \leq N \leq 250$ gold coins

$1 \leq v_i \leq 2000$ each coin value

Divide into two parts with the smallest difference

Find number of ways to split with this minimun difference

Solution:

Let say $sum = v_1 + v_2 + \dots v_n$

$\Rightarrow$ we want to split the golds as close to sum/2 as possible.

Let $f(n, X)$ be
- True : if first n elements have a subset with sum equal to X.
- False : if among first n elements no subset with sum equal to X.

then $f(n,X) = f(n-1, x) \;||\; f(n-1, x - v_n)$

Sum X can be accessed with out n^{th} coin with n^{th} coin.

a) Obj value:

Find the greatest $X \leq sum/2$ which has $f(n,X)$ is true.

Alternative solution

Let $f(i,x)$ return the maximum value $\leq x$ s.t first elements has a subset with.

return the maximum subset sum value of first i elements $\leq x$

$$f(i,x) = \max \{ f(i-1, x) \;,\; f(i-1, x - v_i) + v_i \}$$

without using i^{th} element what is max — with using i^{th} element what is max.

a) Base case:

$f(0,x)$ with 0 elements max subset sum value is 0 $\quad f(0,x) = 0 \quad \forall x$

Coins = 3,4,1,5

elements	n \ sums	0	1	2	3	4	5	6	7	8	9	10	11
∅	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1	0	0	0	3	3	3	3	3	3	3	3	3
4	2	0	0	0	3	4	4	4	7	7	7	7	7
1	3	0	1	1	3	4	5	5	7	8	8	8	8
5	4	0	1	1	3	4	5	6	7	8	9	10	10

DO NOT SHARE

q) Bottom-up ?

$dp[0,x] = 0$

$dp[i,j] = \max\{dp[i-1,j],\ dp[i-1,j-v_i]+v_i\}$ s.t $dp[i-1,j-v_i]+v_i \leq j$

q) What about # of ways to reach this sum?

Let $f(i,X)$ be number of ways to reach sum X with subsets of first i elements

$$f(i,x) = f(i-1,x) + f(i-1,x-v_i)$$

We can have subsets with sum equal to X as much as

number of ways to reach X without using i^{th} element

and

number of ways to reach $x-v_i$ value without i^{th} element.
(so we use i^{th} element total becomes $x-v_i+v_i=x$)

q) Base case

with 0 elements how many ways we can add up to a value?

to zero sum one way. Empty set. to other sums 0.

$$f(0,x) = \begin{cases} 1 & x=0 \\ 0 & \text{otherwise} \end{cases}$$

Coins = 3,4,1,5

elements	n	0	1	2	3	4	5	6	7	8	9	10	11
Ø	0	1	0	0	0	0	0	0	0	0	0	0	0
3	1	1	0	0	1	0	0	0	0	0	0	0	0
4	2	1	0	0	1	1	0	0	1	0	0	0	0
1	3	1	1	0	1	1+1	1	0	1	1	0	0	0
5	4	1	1	0	1	2	1+1	1	1	1+1	2	1	0

$f[3,4] = f[2,4] + f[2,0]$
$= 1 + 1$
$= 2$

$f[4,5] = f[3,5] + f[3,0]$
$= 1 + 1$
$= 2$

$f[4,9] = f[3,9] + f[3,4]$
$= 0 + 2$
$= 2$

q) Bottom-up

$dp[0,0] = 1$

$dp[i,j] = dp[i-1,j] + dp[i-1,j-v_i]$

DO NOT SHARE

0-1 Knapsack Problem

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don't pick it (0-1 property).

To illustrate this, consider the following example:

Item	Weight	Value
I_0	3	10
I_1	8	4
I_2	9	9
I_3	8	11

The maximum weight the knapsack can hold is 20.

The best set of items from {I0, I1, I2, I3} is {I0, I2, I3}.

A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

1) Optimal Substructure:

To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.

Therefore, the maximum value that can be obtained from n items is max of following two values.

1) Maximum value obtained by n-1 items and W weight (excluding nth item).

2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

2) Overlapping Subproblems

Following is recursive implementation that simply follows the recursive structure mentioned above.

```
// Returns the maximum value that can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
   // Base Case
   if (n == 0 | W == 0)
       return 0;

   // If weight of the nth item is more than Knapsack capacity W, then
   // this item cannot be included in the optimal solution
   if (wt[n-1] > W)
       return knapSack(W, wt, val, n-1);

   // Return the maximum of two cases:
   // (1) nth item included
   // (2) not included
   else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
                    knapSack(W, wt, val, n-1));
}
```

DO NOT SHARE

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, K(1, 1) is being evaluated twice. Time complexity of this naive recursive solution is exponential (2^n).

In the following recursion tree, K() refers to knapSack(). The two parameters indicated in the following recursion tree are n and W.

The recursion tree is for following sample inputs. wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}

```
In the following recursion tree, K() refers to knapSack().  The two
parameters indicated in the following recursion tree are n and W.
The recursion tree is for following sample inputs.
wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}

                     K(3, 2)         ---------> K(n, W)
                  /           \
                /               \
            K(2,2)                K(2,1)
           /      \               /    \
         /          \            /       \
       K(1,2)      K(1,1)       K(1,1)    K(1,0)
       /  \        /   \        /   \
     /     \     /      \     /      \
K(0,2)  K(0,1)  K(0,1)  K(0,0)  K(0,1)   K(0,0)
Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.
```

Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.

Since subproblems are evaluated again, this problem has Overlapping Subproblems property. So the 0-1 Knapsack problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, re-computations of same subproblems can be avoided by constructing a temporary array K[][] in bottom up manner. Following is Dynamic Programming based implementation.

```
// Returns the maximum value that can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
   int i, w;
   int K[n+1][W+1];

   // Build table K[][] in bottom up manner
   for (i = 0; i <= n; i++)
   {
       for (w = 0; w <= W; w++)
       {
           if (i==0 || w==0)
               K[i][w] = 0;
           else if (wt[i-1] <= w)
                 K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w]);
           else
                 K[i][w] = K[i-1][w];
       }
   }

   return K[n][W];
}
```

Time Complexity: O(nW) where n is the number of items and W is the capacity of knapsack.

DO NOT SHARE

Let's work on a larger example:

i	Item	w_i	v_i
0	I_0	4	6
1	I_1	2	4
2	I_2	3	5
3	I_3	1	3
4	I_4	6	9
5	I_5	4	7

W = 10

Item	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	6	6	6	6	6	6	6
1	0	0	4	4	6	6	10	10	10	10	10
2	0	0	4	5	6	9	10	11	11	15	15
3	0	3	4	7	8	9	12	13	14	15	18
4	0	3	4	7	8	9	12	13	14	16	18
5	0	3	4	7	8	10	12	14	15	16	19

Sample Questions:

- Hay4Sale '08 Dec Silver
- Fristeam '09 Mar Silver

More questions on lesson pages.

References:

http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf
http://www.cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf
https://www.geeksforgeeks.org/knapsack-problem/

DO NOT SHARE

Longest Common Substring

Monday, March 12, 2018 11:39 PM

Given two strings X and Y, find the length of longest common substring.

Sample input:

X="abcdxyz" Y="xyzabcd"

X="zxabcdezy" Y="yzabcdezx"

Output:

abcd

yzabcd

Sol 1: Take smaller string say of length n.

It has n^2 substrings.

for each position in bigger string m
check if substring exists n^2 $\Rightarrow O(mn^2)$

Sol 2: Divide the problem into subproblems and use solutions of subproblems to solve the original problem.

Working on strings and subsequences Subproblems can be defined in terms of

Prefixes $s[:i]$ $\forall i$ (n)

Suffixes $s[i:]$ $\forall i$ (n)

Substrings $s[i:j]$ $\forall i \leq j$ (n^2)

Subproblems You can use prefixes or suffixes

Use prefixes as subproblems

Let $f(i,j)$ be LCStr ending at position i,j

$$f(abcd, xbcyd) = 1 + f(abc, xbcy)$$

$$f(abc, xbcy) = 0$$

	∅	x	b	c	y	d
∅	0	0	0	0	0	0
a	0	0	0	0	0	0
b	0	0	1	0	0	0
c	0	0	0	2	0	0
d	0	0	0	0	0	1

$$f(i,j) = \begin{cases} 1 + f(i-1, j-1) & \text{if } s1[i] = s2[j] \\ 0 & \text{otherwise} \end{cases}$$

Base case: $f(0,i) = 0$ $\forall j$
$f(i,0) = 0$ $\forall i$

Use suffixes as Subproblems

Let $f(i,j)$ be LCStr starting at position i,j

$$f(i,j) = \begin{cases} 1 + f(i+1, j+1) & \text{if } s1[i] = s2[j] \\ 0 & \text{otherwise} \end{cases}$$

Base Case: $f(n+1, i) = 0$
$f(i, m+1) = 0$

chars are at $x_1, x_2 \ldots x_n$
$y_1, y_2 \ldots y_n$

DO NOT SHARE

Longest Common Subsequence

LCS Problem Statement: Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. etc are subsequences of “abcdefg”. So a string of length n has 2^n different possible subsequences.

It is a classic computer science problem, the basis of diff (a file comparison program that outputs the differences between two files), and has applications in bioinformatics.

Examples:

LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.

LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

The naive solution for this problem is to generate all subsequences of both given sequences and find the longest matching subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem.

Optimal Substructure:

Let the input sequences be X[0..m-1] and Y[0..n-1] of lengths m and n respectively. And let L(X[0..m-1], Y[0..n-1]) be the length of LCS of the two sequences X and Y. Following is the recursive definition of L(X[0..m-1], Y[0..n-1]).

If last characters of both sequences match (or X[m-1] == Y[n-1]) then
L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

If last characters of both sequences do not match (or X[m-1] != Y[n-1]) then L(X[0..m-1], Y[0..n-1]) = MAX (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2])

Examples:

1. Consider the input strings “AGGTAB” and “GXTXAYB”. Last characters match for the strings. So length of LCS can be written as:
L(“AGGTAB”, “GXTXAYB”) = 1 + L(“AGGTA”, “GXTXAY”)
2. Consider the input strings “ABCDGH” and“AEDFHR. Last characters do not match for the strings. So length of LCS can be written as:
L(“ABCDGH”, “AEDFHR”) = MAX (L(“ABCDG”, “AEDFH**R**”), L(“ABCDG**H**”, “AEDFH”))

	A	G	G	T	A	B
G	-	-	4	-	-	-
X	-	-	-	-	-	-
T	-	-	-	3	-	-
X	-	-	-	-	-	-
A	-	-	-	-	2	-
Y	-	-	-	-	-	-
B	-	-	-	-	-	1

So the LCS problem has optimal substructure property as the main problem can be solved using solutions to subproblems.

```
/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
   if (m == 0 || n == 0)
     return 0;
   if (X[m-1] == Y[n-1])
     return 1 + lcs(X, Y, m-1, n-1);
   else
     return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
}
```

DO NOT SHARE

Overlapping Subproblems:

Following is simple recursive implementation of the LCS problem. The implementation simply follows the recursive structure mentioned above.

Time complexity of the above naive recursive approach is O(2^n) in worst case and worst case happens when all characters of X and Y mismatch i.e., length of LCS is 0.

Considering the above implementation, following is a partial recursion tree for input strings “AXYT” and “AYZX”

```
                        lcs("AXYT", "AYZX")
                      /
          lcs("AXY", "AYZX")            lcs("AXYT", "AYZ")
          /                              /
lcs("AX", "AYZX") lcs("AXY", "AYZ")   lcs("AXY", "AYZ") lcs("AXYT", "AY")
```

In the above partial recursion tree, lcs(“AXY”, “AYZ”) is being solved twice. If we draw the complete recursion tree, then we can see that there are many subproblems which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation. Following is a tabulated implementation for the LCS problem.

```
/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
    int L[m+1][n+1];
    int i, j;

    /* Following steps build L[m+1][n+1] in bottom up fashion. Note
       that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] */
    for (i=0; i<=m; i++)
    {
      for (j=0; j<=n; j++)
      {
        if (i == 0 || j == 0)
          L[i][j] = 0;

        else if (X[i-1] == Y[j-1])
          L[i][j] = L[i-1][j-1] + 1;

        else
          L[i][j] = max(L[i-1][j], L[i][j-1]);
      }
    }

    /* L[m][n] contains length of LCS for X[0..n-1] and Y[0..m-1] */
    return L[m][n];
}
```

Time Complexity of the above implementation is O(mn) which is much better than the worst case time complexity of Naive Recursive implementation.

References: http://www.youtube.com/watch?v=V5hZoJ6uK-s
http://www.algorithmist.com/index.php/Longest_Common_Subsequence
http://www.ics.uci.edu/~eppstein/161/960229.html
http://en.wikipedia.org/wiki/Longest_common_subsequence_problem
https://www.geeksforgeeks.org/longest-common-subsequence/

DO NOT SHARE

String Edit Distance (SED)

Tuesday, March 13, 2018 12:36 AM

Given two strings s1 and s2

Find the minimum number of <u>character edits</u> required to convert s1 into s2.

- → insert c
- → delete c
- → replace c → c'

Sample Input:		Output:	
s1 = SUNDAY	s2 = SATURDAY	3	R(N,R), I(A), I(T)
s1 = cat	s2 = cut	1	R(a,u)

Smaller string solutions, how they can be used for current strings?

Initial Steps

f(SUNDAY, SATURDAY) = f(SUNDA, SATURDA) = f(SUND, SATURD) = f(SUN, SATUR)

we decreased our problem to solving f(SUN, SATUR)

because if last character are same if s1[i] = s2[j] then

f(s1[1..i], s2[1..j]) = f(s1[1..i-1], s2[1..j-1])

what about f(SUN, SATUR)

Work on examples

f(ABCD, EFGH) =

ABCD will be converted into EFGH. One of 3 cases must be applied focusing on the last character. they have to be same.

- Convert ABCD → EFG + Add H to the end of s2. f(ABCD, EFG) + I(H)

DO NOT SHARE

- Delete D from s1 + Convert ABC → EFGH D(D) + f(ABC, EFGH)
- Convert ABC → EFG + Replace the last character R(D,H) + f(ABC, EFG)

Let $f(i,j)$ be the cost of converting $s1[:i]$ to $s2[:j]$

$$f(i,j) = \min \begin{pmatrix} f(i-1,j-1) + 1 & \text{Replace}(s1[i], s2[j]) \\ f(i-1,j) + 1 & \text{Delete}(s1[i]) \\ f(i,j-1) + 1 & \text{Insert}(s2[j]) \end{pmatrix}$$

$$\text{Replacecost}(s1[i], s2[j]) = \begin{cases} 0 & \text{if } s1[i] = s2[j] \\ 1 & \text{otherwise} \end{cases}$$

<u>Base case</u>: $f(\emptyset, s2) = s2.\text{length}$

$f(s1, \emptyset) = s1.\text{length}$

	Ø	S	A	T	U	R	D	A	Y
Ø	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
U	2	1	1	2	2	3	4	5	6
N	3	2	2	2	3	3	4	5	6
D	4	3	3	3	3	4	3	4	5
A	5	4	3	4	4	4	4	3	4
Y	6	5	4	4	5	5	5	4	3

$$f(SUN, SATUR) = \min \begin{pmatrix} f(SU, SATU) + 1 & R(N,R) \\ f(SU, SATUR) + 1 & D(N) \\ f(SUN, SATU) + 1 & I(R) \end{pmatrix}$$

9) What if Insert, Delete and Replace has different costs?

Change +1 with the cost of the corresponding action

$$f(i,j) = \min \begin{pmatrix} f(i-1,j-1) + \text{Replacecost}(s1[i], s2[j]) \\ f(i-1,j) + \text{Delete cost}(s1[i]) \\ f(i,j-1) + \text{Insertcost}(s2[j]) \end{pmatrix}$$

DO NOT SHARE

Landscaping

Tuesday, March 13, 2018 4:39 PM

$1 \leq N \leq$ flowerbeds

A_i units of dirt at pos i

B_i units of dirt is intended

$A_i + 1$	\$X	place one unit of dirt
$A_i - 1$	\$Y	remove one unit of dirt
$A_j - 1, A_i + 1$	\$Z $(\lvert i-j \rvert)$	replace one unit from j to i

$\Rightarrow$ 1 2 3 4 $\quad$ X=100

$\Rightarrow$ 4 3 2 0 $\quad$ Y=200

Z=1

200	remove	one unit of dirt.
$3 \times 3 + 1 \times 1 = 10$	replace	4 units of dirt.
210		

	∅	0	0	0	0	1	1	1	2	2
∅	0	100	200	300	400	500	600	700	800	900
0	200	0	100	200	300	400	500	600	700	800
1	400	200	1	101	201	300	400	500	600	700
1	600	400	201	2	102	201	300	400	500	600
2	800	600	401	202	4	103	202	301	400	500
2	1000	800	601	402	204	5	104	203	301	400
2	1200	1000	801	602	404	205	6	105	203	301
3	1400	1200	1001	802	604	405	206	8	106	204
3	1600	1400	1201	1002	804	605	406	208	9	107
3	1800	1600	1401	1202	1004	805	606	408	209	10
3	2000	1800	1601	1402	1204	1005	806	608	409	210

<u>Base Case:</u> $f(0,j) = j \times X$

$f(i,0) = i \times Y$

$$f(i,j) = \min \begin{cases} f(i-1,j-1) + Z \times \lvert A[i] - B[j] \rvert \\ f(i-1,j) + Y \\ f(i,j-1) + X \end{cases}$$

DO NOT SHARE

Number Game - IOI '96

Tuesday, March 20, 2018 3:07 PM

$N \leq 200$ coins (N even)

v_i value of each coin.

Two player, each picks up either left or right end coin in his turn.

Player with higher total value wins. (Goal: maximize value of coins selected)

Implement the optimal strategy.

N = 6

values: 4 7 2 9 5 2

Solution Let $f(i,j)$ give the max value we can guarantee if remaining coins are between i^{th} to j^{th}

$\Rightarrow$ Base case: $f(i,i) = v_i \quad \forall i$

$f(i,i+1) = \max(v_i, v_{i+1}) \quad \forall i \; 0 < i < n-1$

Recursive formula

$$f(i,j) = \max\left(\underbrace{\min\begin{pmatrix} f(i+2,j) \\ f(i+1,j-1)\end{pmatrix} + v_i}_{\text{pick } v_i}, \underbrace{\min\begin{pmatrix} f(i+1,j-1) \\ f(i,j-2)\end{pmatrix} + v_j}_{\text{pick } v_j}\right)$$

9) Objective value?

what is max sum we can guarantee between $0, n-1 \Rightarrow f(0,n-1)$

Bottom-up

We need DAG

$$dp[i,j] = \max\left(\min(dp[i,j-2], dp[i+1,j-1]) + v_j, \min(dp[i+1,j-1], dp[i+2,j]) + v_i\right)$$

4	7	2	9	5	2
4	7	6	16	11	18
	7	7	11	16	14
		2	9	7	11
			9	9	11
				5	5
					2

$$f(1,3) = \max\left(7 + \min\begin{pmatrix} f(2,2) \\ f(3,3)\end{pmatrix}, 9 + \min\begin{pmatrix} f(1,1) \\ f(2,2)\end{pmatrix}\right)$$

7 2 9

DO NOT SHARE

Longest Increasing Subsequence Size (N log N)

Given an array of random numbers. Find *longest increasing* <u>*subsequence*</u> (LIS) in the array. The size of the list is >20.000 (=> N^2 solution is not good enough to work in 1 second) We need an <u>O(N log N)</u> solution.

For the time being, forget about recursive and DP solutions. Let us take small samples and extend the solution to large instances. Even though it may look complex at first time, once if we understood the logic, coding is simple.

Consider an input array A = {2, 5, 3}. I will extend the array during explanation.

By observation we know that the LIS is either {2, 3} or {2, 5}. ***Note that I am considering only strictly increasing sequences***.

Let us add two more elements, say 7, 11 to the array. These elements will extend the existing sequences.
Now the increasing sequences are {2, 3, 7, 11} and {2, 5, 7, 11} for the input array {2, 5, 3, 7, 11}.

Further, we add one more element, say 8 to the array i.e. input array becomes {2, 5, 3, 7, 11, 8}. Note that the latest element 8 is greater than smallest element of any active sequence (*will discuss shortly about active sequences*). How can we extend the existing sequences with 8? First of all, can 8 be part of LIS? If yes, how? If we want to add 8, it should come after 7 (by replacing 11).

Since the approach is *offline (what we mean by* <u>*offline*</u>*?)*, we are not sure whether adding 8 will extend the series or not. Assume there is 9 in the input array, say {2, 5, 3, 7, 11, 8, 7, 9 …}. We can replace 11 with 8, as there is potentially *best* candidate (9) that can extend the new series {2, 3, 7, 8} or {2, 5, 7, 8}.

Our observation is, assume that the end element of largest sequence is E. We can add (replace) current element A[i] to the existing sequence if there is an element A[j] (j > i) such that E < A[i] < A[j] or (E > A[i] < A[j] – for replace). In the above example, E = 11, A[i] = 8 and A[j] = 9.

In case of our original array {2, 5, 3}, note that we face same situation when we are adding 3 to increasing sequence {2, 5}. I just created two increasing sequences to make explanation simple. Instead of two sequences, 3 can replace 5 in the sequence {2, 5}.

I know it will be confusing, I will clear it shortly!

The question is, when will it be safe to add or replace an element in the existing sequence?

Let us consider another sample A = {2, 5, 3}. Say, the next element is 1. How can it extend the current sequences {2,3} or {2, 5}. Obviously, it can't extend either. Yet, there is a potential that the new smallest element can be start of an LIS. To make it clear, consider the array is {2, 5, 3, 1, 2, 3, 4, 5, 6}. Making 1 as new sequence will create new sequence which is largest.

The observation is, when we encounter new smallest element in the array, it can be a potential candidate to start new sequence.

From the observations, we need to maintain lists of increasing sequences.

In general, we have set of **active lists** of varying length. We are adding an element A[i] to these lists. We scan the lists (for end elements) in decreasing order of their length. We will verify the end elements of all the lists to find a list whose end element is smaller than A[i] (*floor* value).

Our strategy determined by the following conditions,

1. **If A[i] is smallest among all *end* candidates of active lists, we will *start* new active list of length 1.**
2. **If A[i] is largest among all *end* candidates of active lists, we will clone the *largest* active list, and extend it by A[i].**

DO NOT SHARE

3. **If A[i] is in between, we will find a list with *largest end element that is smaller than* A[i]. Clone and extend this list by A[i]. We will discard all other lists of same length as that of this modified list.**

Note that at any instance during our construction of active lists, the following condition is maintained.

"end element of smaller list is smaller than end elements of larger lists" .

It will be clear with an example, let us take example from wiki {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15}.

A[0] = 0. Case 1. There are no active lists, create one.
0.
----------A[1] = 8. Case 2. Clone and extend.
0.
0, 8.
----------A[2] = 4. Case 3. Clone, extend and discard.
0.
0, 4.
0, 8. Discarded
----------A[3] = 12. Case 2. Clone and extend.
0.
0, 4.
0, 4, 12.
----------A[4] = 2. Case 3. Clone, extend and discard.
0.
0, 2.
0, 4. Discarded.
0, 4, 12.
----------A[5] = 10. Case 3. Clone, extend and discard.
0.
0, 2.
0, 2, 10.
0, 4, 12. Discarded.
----------A[6] = 6. Case 3. Clone, extend and discard.
0.
0, 2.
0, 2, 6.
0, 2, 10. Discarded.
----------A[7] = 14. Case 2. Clone and extend.
0.
0, 2.
0, 2, 6.
0, 2, 6, 14.
----------A[8] = 1. Case 3. Clone, extend and discard.
0.
0, 1.
0, 2. Discarded.
0, 2, 6.
0, 2, 6, 14.
----------A[9] = 9. Case 3. Clone, extend and discard.
0.
0, 1.
0, 2, 6.
0, 2, 6, 9.
0, 2, 6, 14. Discarded.
----------A[10] = 5. Case 3. Clone, extend and discard.
0.
0, 1.
0, 1, 5.
0, 2, 6. Discarded.
0, 2, 6, 9.

DO NOT SHARE

----------A[11] = 13. Case 2. Clone and extend.
0.
0, 1.
0, 1, 5.
0, 2, 6, 9.
0, 2, 6, 9, 13.
----------A[12] = 3. Case 3. Clone, extend and discard.
0.
0, 1.
0, 1, 3.
0, 1, 5. Discarded.
0, 2, 6, 9.
0, 2, 6, 9, 13.
----------A[13] = 11. Case 3. Clone, extend and discard.
0.
0, 1.
0, 1, 3.
0, 2, 6, 9.
0, 2, 6, 9, 11.
0, 2, 6, 9, 13. Discarded.
----------A[14] = 7. Case 3. Clone, extend and discard.
0.
0, 1.
0, 1, 3.
0, 1, 3, 7.
0, 2, 6, 9. Discarded.
0, 2, 6, 9, 11.
----------A[15] = 15. Case 2. Clone and extend.
0.
0, 1.
0, 1, 3.
0, 1, 3, 7.
0, 2, 6, 9, 11.
0, 2, 6, 9, 11, 15. <-- LIS List

It is required to understand above strategy to devise an algorithm. Also, ensure we have maintained the condition, "*end element of smaller list is smaller than end elements of larger lists* ". Try with few other examples, before reading further. It is important to understand what happening to end elements.

Algorithm:

Querying length of longest is fairly easy. Note that we are dealing with end elements only. We need not to maintain all the lists. We can store the end elements in an array. Discarding operation can be simulated with replacement, and extending a list is analogous to adding more elements to array.

We will use an auxiliary array to keep end elements. The maximum length of this array is that of input. In the worst case the array divided into N lists of size one (note that it doesn't lead to worst case complexity). To discard an element, we will trace ceil value of A[i] in auxiliary array (again observe the end elements in your rough work) and replace ceil value with A[i]. We extend a list by adding element to auxiliary array. We also maintain a counter to keep track of auxiliary array length. You have learnt Patience Sorting technique partially Here is a proverb, "Tell me and I will forget. Show me and I will remember. Involve me and I will understand." So, pick a suit from deck of cards. Find the longest increasing sub-sequence of cards from the shuffled suit.

You will never forget the approach. http://stackoverflow.com/questions/2631726/how-to-determine-the-longest-increasing-subsequence-usingdynamic-programming

DO NOT SHARE

Given below is code to find length of LIS (updated to C++11 code, no C-style arrays),

```
// Binary search (note boundaries in the caller)
int CeilIndex(std::vector<int> &v, int l, int r, int key) {
    while (r-l > 1) {
    int m = l + (r-l)/2;
    if (v[m] >= key)
        r = m;
    else
        l = m;
    }
    return r;
}

int LongestIncreasingSubsequenceLength(std::vector<int> &v) {
    if (v.size() == 0)
        return 0;

    std::vector<int> tail(v.size(), 0);
    int length = 1; // always points empty slot in tail

    tail[0] = v[0];
    for (size_t i = 1; i < v.size(); i++) {
        if (v[i] < tail[0])          // new smallest value
            tail[0] = v[i];
        else if (v[i] > tail[length-1]) // v[i] extends largest subsequence
            tail[length++] = v[i];
        else
            tail[CeilIndex(tail, -1, length-1, v[i])] = v[i];
    }
    return length;
}
```

Complexity:

The loop runs for N elements. In the worst case (what is worst case input?), we may end up querying ceil value using binary search (log *i*) for many A[i]. Therefore, T(n) < O(log N!) = O(N log N). Analyse to ensure that the upper and lower bounds are also O(N log N). The complexity is THETA (N log N).

DO NOT SHARE

Memory Optimization on Knapsack

Knapsack - Memory Optimization

Tuesday, March 20, 2018 6:00 PM

N items

s_i size of each item

v_i value of each item

Given C capacity of a knapsack

Goal: maximize total value of knapsack

$K(i,j)$ optimal value for filling capacity j knapsack with some subset of first i items

$$K(i,j) = \max\left(K(i-1,j),\ K(i-1,j-s_i)+v_i\right)$$

do not use i^{th} item

use i^{th} item

$1..N$ $1..C$

$N \times C$ memory

size	vol	items	0	1	2	3	4	5	6	7	8	9	10
			0	0	0	0	0	0	0	0	0	0	0
4	6	1	0	0	0	0	6	6	6	6	6	6	6
2	4	2	0	0	4	4	6	6	10	10	10	10	10
3	5	3	0	0	4	5	6	9	10	11	11	15	15
1	3	4	0	3	4	7	8	9	12	13	14	15	18
6	9	5	0	3	4	7	8	9	12	13	14	16	18
4	7	6	0	3	4	7	8	10	12	14	15	16	19

We fill a table of size $N \times C$ objective value is at $K(N,C)$.

In order to calculate $K(i,j)$ we need $K(i-1,j)$, $K(i-1,j-s_i)$ values.

$\Rightarrow$ Keeping previous row is enough to calculate the next row.

$\Rightarrow$ Memory complexity can be reduced to $2 \times C \Rightarrow O(C)$

1) Keep 2 1 dimensional array of size C: prev, cur

```
Repeat n times.
    calculate cur based on prev.
    copy cur to prev
```

2) Define 2D array dp[2][C]

```
dp[0][C] <= Base case
cur = 1    prev = 0
Repeat n times
    for j: 0 -> C
        dp[cur, j] = max(dp[prev, j], dp[prev, j-s_i] + v_i)
        cur = 1-cur    prev = 1-prev
```

3) Define 2D array dp[2][C]

```
dp[0][C] <= Base case
for i: 1 -> N
    for j: 1 -> C
        dp[i%2, j] = max(dp[(i+1)%2, j], dp[(i+1)%2, j-s_i] + v_i)
```

DO NOT SHARE

Outputting the Path

Subset Sum

Tuesday, March 20, 2018 6:22 PM

$N \leq 100$ elements $set_i \leq 100$ $Sum \leq 10000$

Find if any subset of N elements add up to target Sum value.

Output the subset with target sum.

Input	Output
N=6 Sum=11	2 5 4
3 34 4 12 5 2	

Remember Subset sum problem

$S(n,x)$ True if first n elements has a subset with sum X
False if first n elements has no subset with sum X.

$$S(n,x) = S(n-1, x) \;\|\; S(n-1, x-v_n)$$

do not use n^{th} item — use the nth item

Base case: $S(0,i) = \begin{cases} True & i=0 \\ False & otherwise \end{cases}$

elements \ sums	0	1	2	3	4	5	6	7	8	9	10
Ø	1	0	0	0	0	0	0	0	0	0	0
3	1	0	0	1	0	0	0	0	0	0	0
4	1	0	0	1	1	0	0	1	0	0	0
1	1	1	0	1	1	1	0	1	1	0	0
5	1	1	0	1	1	1	1	1	1	1	1

Now we want to find which elements are included

Notice that if $S(i,j)$ is True and $S(i-1,j)$ is False

$\Rightarrow$ i^{th} element is needed to reach sum j.

```
if S[N,Sum] = False
    no Solution
i=N   j=Sum
Repeat until i=0 or j=0
    while S[i-1,j] = True
        i--;
    cout << i
    j = j - v_i;
```

DO NOT SHARE

LCS- Outputting the Path

Tuesday, March 20, 2018 6:52 PM

Remember LCS problem definition:

Given two strings s1 and s2

LCS aims to find the length of Longest Common Subsequence

Now we need to output the common subsequence also.

Recall solution

$$LCS(n,m) = \begin{cases} 1 + LCS(n-1, m-1) & \text{if } s1[n] == s2[m] \\ \max(LCS(n-1,m), LCS(n,m-1)) & \end{cases}$$

Base case: $LCS(0,x) = 0 \quad \forall x < n$

$LCS(x,0) = 0 \quad \forall x < m$

	Ø	G	X	T	X	A	Y	B
Ø	0	0	0	0	0	0	0	0
A	0	0	0	0	0	1	1	1
G	0	1	1	1	1	1	1	1
G	0	1	1	1	1	1	1	1
T	0	1	1	2	2	2	2	2
A	0	1	1	2	2	3	3	3
B	0	1	1	2	2	3	3	4

	Ø	G	X	T	X	A	Y	B
Ø	0	0	0	0	0	0	0	0
A	0	0	0	0	0	1	1	1
G	0	1	1	1	1	1	1	1
G	0	1	1	1	1	1	1	1
T	0	1	1	2	2	2	2	2
A	0	1	1	2	2	3	3	3
B	0	1	1	2	2	3	3	4

repeat

if last characters are same

this char is in the solution

otherwise

which ever value is same go that direction

until LCS value is 0.

```
i:n  j:m
while dp[i,j] ≠0
    if s1[i]==s2[j]
        sol += s1[i]:
        i--  j--
    else if dp[i-1,j]==dp[i,j]
        i--;
    else
        j--;
```

DO NOT SHARE

Computational Geometry Algorithms

Pick's Theorem?

Pick's Theorem is a useful method for determining the area of any polygon whose vertices are points on a lattice, a regularly spaced array of points. While lattices may have points in different arrangements, this essay uses a square lattice to examine Pick's Theorem.

Examples of polygons whose area can be computed by Pick's Theorem are shown below

Although the area of each figure can be calculated using other methods (e.g., partitioning it into smaller pieces or using a surrounding rectangle), Pick's theorem provides a relatively simple alternative. To understand this formula, two definitions are needed:

<u>Boundary Point</u> (*B*): a lattice point on the polygon (including vertices)

<u>Interior Point</u> (*I*): a lattice point in the polygon's interior region

Pick's Theorem uses these definitions to state the area of a polygon whose vertices are lattice points:

$$Area = \frac{B}{2} + I - 1$$

For example, the area of the yellow polygon above requires counting the number of border points (5) and interior points (5). These values are used in the formula:

$$Area = \frac{5}{2} + 5 - 1 = 6.5$$

As an exercise, use Pick's Theorem to calculate the area of the green and blue figures

DO NOT SHARE

Orientation of 3 ordered points

Orientation of an ordered triplet of points in the plane can be

- counter clock wise
- clockwise
- colinear

The following diagram shows different possible orientations of (a,b,c)

If orientation of (p1, p2, p3) is collinear, then orientation of (p3, p2, p1) is also collinear. If orientation of (p1, p2, p3) is clockwise, then orientation of (p3, p2, p1) is counterclockwise and vice versa is also true.

Given three points p1, p2 and p3, find orientation of (p1, p2, p3).

Example:

Input: p1 = {0, 0}, p2 = {4, 4}, p3 = {1, 2}

Output: CounterClockWise

Input: p1 = {0, 0}, p2 = {4, 4}, p3 = {1, 1}

Output: Colinear

How to compute Orientation?

The idea is to use slope.

Slope of line segment (p1, p2):

S1 = (y2 - y1)/(x2 - x1)

Slope of line segment (p2, p3):

S2= (y3 - y2)/(x3 - x2)

DO NOT SHARE

If S1 < S2, the orientation is counterclockwise (left turn) If S1 = S2, the orientation is collinear

If S1 > S2, the orientation is clockwise (right turn)

But comparing the slopes can be cumbersome if the points are vertically linear. We may end up dividing by 0.

Question: How can we overcome this problem?

Using above values of S1 and S2, we can conclude that, the orientation depends on sign of below expression:

(y2 - y1)*(x3 - x2) - (y3 - y2)*(x2 - x1)

Above expression is negative when S1 < S2, i.e., counterclockwiseIt is 0 when S1 = S2, i.e., collinear Also it is positive when S1 > S2, i.e., clockwise

Below is the implementation of above idea.

```
// A C++ program to find orientation of three points
#include <iostream>
using namespace std;

struct Point
{
    int x, y;
};

// To find orientation of ordered triplet (p1, p2, p3).
// The function returns following values
// 0 --> p, q and r are colinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(Point p1, Point p2, Point p3)
{
    // See 10th slides from following link for derivatior
    // of the formula
    int val = (p2.y - p1.y) * (p3.x - p2.x) -
              (p2.x - p1.x) * (p3.y - p2.y);

    if (val == 0) return 0;  // colinear

    return (val > 0)? 1: 2; // clock or counterclock wise
}
```

References:

http://www.dcs.gla.ac.uk/~pat/52233/slides/Geometry1x1.pdf

https://www.geeksforgeeks.org/orientation-3-ordered-points/

DO NOT SHARE

How to check if two given line segments intersect?

Given two line segments (p1, q1) and (p2, q2), find if the given line segments intersect with each other.

Remember notion of **orientation** from previous section. Orientation of an ordered triplet of points in the plane can be

clockwise	**counterclockwise**	**colinear**

How is Orientation useful here?

Two segments (p1,q1) and (p2,q2) intersect if and only if one of the following two conditions is verified

1. *General Case:*

– (p1, q1, p2) and (p1, q1, q2) have different orientations and – (p2, q2, p1) and (p2, q2, q1) have different orientations.

Examples:

Example : Orientations of (p1,q1,p2) and (p1,q1,q2) are different. Orientations of (p2,q2,p1) and (p2,q2,q1) also differnet.

Example: Orientations of (p1,q1,p2) and (p1,q1,q2) are different. Orientations of (p2,q2,p1) and (p2,q2,q1) also different

Example: Orientations of (p1,q1,p2) and (p1,q1,q2) are different. Orientations of (p2,q2,p1) and (p2,q2,q1) are same

Example: Orientations of (p1,q1,p2) and (p1,q1,q2) are different. Orientations of (p2,q2,p1) and (p2,q2,q1) are same.

DO NOT SHARE

2. *Special Case*

– (p1, q1, p2), (p1, q1, q2), (p2, q2, p1), and (p2, q2, q1) are all collinear and

– the x-projections of (p1, q1) and (p2, q2) intersect – the y-projections of (p1, q1) and (p2, q2) intersect

Examples:

Example: All points are collinear. The x-projections pf (p1,q1) and (p2,q2) intersect. The y-projection of (p1,q1) and(p2,q2) also intersect

Example: All points are collinear .The x-projections of (p1,q1) and (p2,q2) not intersect. The y-projection of (p1,q1) and (p2,q2) do not intersect

Following is C++ implementation based on above idea.

```
// The main function that returns true if line segment 'p1q1' and 'p2q2' intersect.
bool doIntersect(Point p1, Point q1, Point p2, Point q2)
{
    // Find the four orientations needed for general and special cases
    int o1 = orientation(p1, q1, p2);
    int o2 = orientation(p1, q1, q2);
    int o3 = orientation(p2, q2, p1);
    int o4 = orientation(p2, q2, q1);

    // General case
    if (o1 != o2 && o3 != o4)
        return true;

    // Special Cases
    // p1, q1 and p2 are collinear and p2 lies on segment p1q1
    if (o1 == 0 && onSegment(p1, p2, q1)) return true;

    // p1, q1 and q2 are collinear and q2 lies on segment p1q1
    if (o2 == 0 && onSegment(p1, q2, q1)) return true;

    // p2, q2 and p1 are collinear and p1 lies on segment p2q2
    if (o3 == 0 && onSegment(p2, p1, q2)) return true;

     // p2, q2 and q1 are collinear and q1 lies on segment p2q2
    if (o4 == 0 && onSegment(p2, q1, q2)) return true;

    return false; // Does not fall in any of the above cases
}
```

References:

http://www.dcs.gla.ac.uk/~pat/52233/slides/Geometry1x1.pdf

https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E.Leiserson, Ronald L. Rivest

DO NOT SHARE

How to check if a given point lies inside or outside a polygon?

Given a polygon and a point 'p', find if 'p' lies inside the polygon or not. The points lying on the border are considered inside.

Following is a simple idea to check whether a point is inside or outside.

1. Draw a horizontal line to the right of each point and extend it to infinity
2. Count the number of times the line intersects with polygon edges.
3. A point is inside the polygon if either count of intersections is odd or point lies on an edge of polygon. If none of the conditions is true, then point lies outside.

How to handle point 'g' in the above figure?

Note that we should returns true if the point lies on the line or same as one of the vertices of the given polygon. To handle this, after checking if the line from 'p' to extreme intersects, we check whether 'p' is colinear with vertices of current line of polygon. If it is collinear, then we check if the point 'p' lies on current side of polygon, if it lies, we return true, else false.
Following is C++ implementation of the above idea.

DO NOT SHARE

```
// Returns true if the point p lies inside the polygon[] with n vertices
bool isInside(Point polygon[], int n, Point p)
{
    // There must be at least 3 vertices in polygon[]
    if (n < 3)  return false;

    // Create a point for line segment from p to infinite
    Point extreme = {INF, p.y};

    // Count intersections of the above line with sides of polygon
    int count = 0, i = 0;
    do
    {
        int next = (i+1)%n;

        // Check if the line segment from 'p' to 'extreme' intersects
        // with the line segment from 'polygon[i]' to 'polygon[next]'
        if (doIntersect(polygon[i], polygon[next], p, extreme))
        {
            // If the point 'p' is collinear with line segment 'i-next',
            // then check if it lies on segment. If it lies, return true,
            // otherwise false
            if (orientation(polygon[i], p, polygon[next]) == 0)
               return onSegment(polygon[i], p, polygon[next]);

            count++;
        }
        i = next;
    } while (i != 0);

    // Return true if count is odd, false otherwise
    return count&1;  // Same as (count%2 == 1)
}
```

Time Complexity: O(n) where n is the number of vertices in the given polygon.

Source: http://www.dcs.gla.ac.uk/~pat/52233/slides/Geometry1x1.pdf

DO NOT SHARE

Introduction.

Geometry is a very important part of most graphics programs, especially computer games, and geometry problems are here to stay. In this article, we'll try to take a bit of the edge off of them and introduce some concepts of geometry problems.

Vectors

Vectors are the basis of a lot of methods for solving geometry problems. Formally, a vector is defined by a direction and a magnitude. In the case of two-dimension geometry, a vector can be represented as pair of numbers, x and y, which gives both a direction and a magnitude. For example, the line segment from (1,3) to (5,1) can be represented by the vector (4,-2). It's important to understand, however, that the vector defines only the direction and magnitude of the segment in this case and does not define the starting or ending locations of the vector.

Vector Addition

There are a number of mathematical operations that can be performed on vectors. The simplest of these is addition: you can add two vectors together and the result is a new vector. If you have two vectors (x_1, y_1) and (x_2, y_2), then the sum of the two vectors is simply (x_1+x_2, y_1+y_2). The image below shows the sum of four vectors. Note that it doesn't matter which order you add them up in – just like regular addition. Throughout these articles, we will use plus and minus signs to denote vector addition and subtraction, where each is simply the piecewise addition or subtraction of the components of the vector.

The sum of vectors A+B+C+D

Dot Product

The addition of vectors is relatively intuitive; a couple of less obvious vector operations are dot and cross products. The dot product of two vectors is simply the sum of the products of the corresponding elements. For example, the dot product of (x_1, y_1) and (x_2, y_2) is $x_1*x_2 + y_1*y_2$. Note that this is not a vector, but is simply a single number (called a scalar). The reason this is useful is that the dot product, $A \cdot B = |A||B|\mathrm{Cos}(\theta)$, where θ is the angle between the A and B. $|A|$ is called the norm of the vector, and in a 2-D geometry problem is simply the length of the vector, $\mathrm{sqrt}(x^2+y^2)$. Therefore, we can calculate $\mathrm{Cos}(\theta) = (A \cdot B)/(|A||B|)$. By using the acos function, we can then find θ. It is useful to recall that $\mathrm{Cos}(90) = 0$ and $\mathrm{Cos}(0) = 1$, as this tells you that a dot product of 0 indicates two perpendicular lines, and that the dot product is greatest when the lines are parallel. A final note about dot products is that they are not limited to

DO NOT SHARE

2-D geometry. We can take dot products of vectors with any number of elements, and the above equality still holds.

Cross Product

An even more useful operation is the cross product. The cross product of two 2-D vectors is $x_1*y_2 - y_1*x_2$Technically, the cross product is actually a vector, and has the magnitude given above, and is directed in the +z direction. Since we're only working with 2-D geometry for now, we'll ignore this fact, and use it like a scalar. Similar to the dot product, `A x B = |A||B|Sin(θ)`. However, θ has a slightly different meaning in this case: `|θ|` is the angle between the two vectors, but θ is negative or positive based on the right-hand rule. In 2-D geometry this means that if A is less than 180 degrees clockwise from B, the value is positive. Another useful fact related to the cross product is that the absolute value of `|A||B|Sin(θ)` is equal to the area of the parallelogram with two of its sides formed by A and B. Furthermore, the triangle formed by A, B and the red line in the diagram has half of the area of the parallelogram, so we can calculate its area from the cross product also.

Parallelogram from A and B

Line-Point Distance

Finding the distance from a point to a line is something that comes up often in geometry problems. Lets say that you are given 3 points, A, B, and C, and you want to find the distance from the point C to the line defined by A and B (recall that a line extends infinitely in either direction). The first step is to find the two vectors from A to B (AB) and from A to C (AC). Now, take the cross product `AB x AC`, and divide by `|AB|`. This gives you the distance (denoted by the red line) as `(AB x AC)/|AB|`. The reason this works comes from some basic high school level geometry. The area of a triangle is found as `base*height/2`. Now, the area of the triangle formed by A, B and C is given by `(AB x AC)/2`. The base of the triangle is formed by AB, and the height of the triangle is the distance from the line to C. Therefore, what we have done is to find twice the area of the triangle using the cross product, and then divided by the length of the base. As always with cross products, the value may be negative, in which case the distance is the absolute value.

DO NOT SHARE

Things get a little bit trickier if we want to find the distance from a line segment to a point. In this case, the nearest point might be one of the endpoints of the segment, rather than the closest point on the line. In the diagram above, for example, the closest point to C on the line defined by A and B is not on the segment AB, so the point closest to C is B. While there are a few different ways to check for this special case, one way is to apply the dot product. First, check to see if the nearest point on the line AB is beyond B (as in the example above) by taking AB · BC. If this value is greater than 0, it means that the angle between AB and BC is between -90 and 90, exclusive, and therefore the nearest point on the segment AB will be B. Similarly, if BA · AC is greater than 0, the nearest point is A. If both dot products are negative, then the nearest point to C is somewhere along the segment. (There is another way to do this, which I'll discuss here).

```
//Compute the dot product AB · BC
    int dot(int[] A, int[] B, int[] C){
        AB = new int[2];
        BC = new int[2];
        AB[0] = B[0]-A[0];
        AB[1] = B[1]-A[1];
        BC[0] = C[0]-B[0];
        BC[1] = C[1]-B[1];
        int dot = AB[0] * BC[0] + AB[1] * BC[1];
        return dot;
    }
    //Compute the cross product AB x AC
    int cross(int[] A, int[] B, int[] C){
        AB = new int[2];
        AC = new int[2];
        AB[0] = B[0]-A[0];
        AB[1] = B[1]-A[1];
        AC[0] = C[0]-A[0];
        AC[1] = C[1]-A[1];
        int cross = AB[0] * AC[1] - AB[1] * AC[0];
        return cross;
    }
    //Compute the distance from A to B
    double distance(int[] A, int[] B){
        int d1 = A[0] - B[0];
        int d2 = A[1] - B[1];
        return sqrt(d1*d1+d2*d2);
    }
    //Compute the distance from AB to C
```

DO NOT SHARE

```
    //if isSegment is true, AB is a segment, not a line.
    double linePointDist(int[] A, int[] B, int[] C, boolean isSegment){
        double dist = cross(A,B,C) / distance(A,B);
        if(isSegment){
            int dot1 = dot(A,B,C);
            if(dot1 > 0)return distance(B,C);
            int dot2 = dot(B,A,C);
            if(dot2 > 0)return distance(A,C);
        }
        return abs(dist);
    }
```

That probably seems like a lot of code, but lets see the same thing with a point class and some operator overloading in C++ or C#. The * operator is the dot product, while ^ is cross product, while + and – do what you would expect.

```
//Compute the distance from AB to C
    //if isSegment is true, AB is a segment, not a line.
    double linePointDist(point A, point B, point C, bool isSegment){
        double dist = ((B-A)^(C-A)) / sqrt((B-A)*(B-A));
        if(isSegment){
            int dot1 = (C-B)*(B-A);
            if(dot1 > 0)return sqrt((B-C)*(B-C));
            int dot2 = (C-A)*(A-B);
            if(dot2 > 0)return sqrt((A-C)*(A-C));
        }
        return abs(dist);
    }
```

Operator overloading is beyond the scope of this article, but I suggest that you look up how to do it if you are a C# or C++ coder, and write your own 2-D point class with some handy operator overloading. It will make a lot of geometry problems a lot simpler.

Polygon Area

Another common task is to find the area of a polygon, given the points around its perimeter. Consider the non-convex polygon below, with 5 points. To find its area we are going to start by triangulating it. That is, we are going to divide it up into a number of triangles. In this polygon, the triangles are ABC, ACD, and ADE. But wait, you protest, not all of those triangles are part of the polygon! We are going to take advantage of the signed area given by the cross product, which will make everything work out nicely. First, we'll take the cross product of `AB x AC` to find the area of ABC. This will give us a negative value, because of the way in which A, B and C are oriented. However, we're still going to add this to our sum, as a negative number. Similarly, we will take the cross product `AC x AD` to find the area of triangle ACD, and we will again get a negative number. Finally, we will take the cross product `AD x AE` and since these three points are oriented in the opposite direction, we will get a positive number. Adding these three numbers (two negatives and a positive) we will end up with a negative number, so will take the absolute value, and that will be area of the polygon.

DO NOT SHARE

The reason this works is that the positive and negative number cancel each other out by exactly the right amount. The area of ABC and ACD ended up contributing positively to the final area, while the area of ADE contributed negatively. Looking at the original polygon, it is obvious that the area of the polygon is the area of ABCD (which is the same as ABC + ABD) minus the area of ADE. One final note, if the total area we end up with is negative, it means that the points in the polygon were given to us in clockwise order. Now, just to make this a little more concrete, lets write a little bit of code to find the area of a polygon, given the coordinates as a 2-D array, p.

```
typedef struct {
   double x,y;
} Point;

double PolygonArea(Point *polygon,int N)
{
   int i,j;
   double area = 0;

   for (i=0;i<N;i++) {
      j = (i + 1) % N;
      area += polygon[i].x * polygon[j].y;
      area -= polygon[i].y * polygon[j].x;
   }

   area /= 2;
   return(area < 0 ? -area : area);
}
```

Notice that if the coordinates are all integers, then the final area of the polygon is one half of an integer.

DO NOT SHARE

```java
public class Point {
    public final int x;
    public final int y;

    // create and initialize a point with given (x, y)
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // return Euclidean distance between this point and that point
    public double distanceTo(Point that) {
        if (that == null) return Double.POSITIVE_INFINITY;
        double dx = this.x - that.x;
        double dy = this.y - that.y;
        return Math.hypot(dx, dy);
    }

   // is a->b->c a counter-clockwise turn?
   // +1 if counter-clockwise, -1 if clockwise, 0 if collinear
   public static int ccw(Point a, Point b, Point c) {
       // return a.x*b.y - a.y*b.x + a.y*c.x - a.x*c.y + b.x*c.y - b.y*c.x;
       double area2 = (b.x - a.x) * (c.y - a.y) - (c.x - a.x) * (b.y - a.y);
       if      (area2 < 0) return -1;
       else if (area2 > 0) return +1;
       else                return  0;
   }

   // is a-b-c collinear?
   public static boolean collinear(Point a, Point b, Point c) {
       return ccw(a, b, c) == 0;
   }

   // is c between a and b?
   // Reference: O' Rourke p. 32
   public static boolean between(Point a, Point b, Point c) {
       if (ccw(a, b, c) != 0) return false;
       if (a.x == b.x && a.y == b.y) {
           return a.x == c.x && a.y == c.y;
       }
       else if (a.x != b.x) {
           // ab not vertical
           return (a.x <= c.x && c.x <= b.x) || (a.x >= c.x && c.x >= b.x);
       }
       else {
           // ab not horizontal
           return (a.y <= c.y && c.y <= b.y) || (a.y >= c.y && c.y >= b.y);
       }
   }
```

DO NOT SHARE

Reference:

https://www.topcoder.com/community/data-science/data-science-tutorials/geometry-concepts-basic-concepts/
http://paulbourke.net/geometry/polygonmesh/source1.c
https://algs4.cs.princeton.edu/91primitives/Point.java.html

The Greedy Algorithm

Tuesday, April 17, 2018 2:25 PM

Basic idea: Build large solutions up from smaller ones.

Keeps only the best solution.

Generally fast, linear to quadratic (mostly NlogN because of sorting)

Fractional Knapsack

- Given a knapsack with a capacity C.
- List of N items with values v1, v2, ... vN and weights w1, w2, ... wN.
- Fill the knapsack with maximum total value. (Allowed to take items in fractional part)

N=4

C=60

Item	Value	Weight
1	280	40
2	100	10
3	120	20
4	120	24

Solution:

i^{th} item $x[i]$ $0 \leq x_i \leq 1$

$x_i . w_i$ contributes to weight

$x_i . v_i$ contributes to value.

Maximize $\sum_{i=1}^{n} x_i . v_i$ subject to constraint $\sum_{i=1}^{n} x_i . w_i = C$

Item	Value	Weight	Ratio
1	280	40	7
2	100	10	10
3	120	20	6
4	120	24	5

C=60

10 2

40 1

10 3

Sort items according to v_i / w_i ratios

for $i:1 \longrightarrow N$

```
if cweight + w[i] ≤ C        //if more space then size of item
{
    x[i] = 1                 // take all of it.
    cweight += w[i]
}
else                         // other wise
    x[i] = (C - cweight)/w[i]  // take as much as you can
```

DO NOT SHARE

Maximum Subarray Sum

Given a list of numbers find the interval with the max sum.

A

$$\text{Max} \sum_{i:s}^{f} A[i]$$

Note that this problem is interesting only if list contains both positive & negative values.

why? what if all values are positive
answer is total of all values
what if all values are negative
answer is the max number in the list

Sol 1: Try all possibilities.

q) How many different subarrays are there

for each start position $i: 1 \to N$
for each end position $j: i+1 \to N$ $\Rightarrow N^2$ different subarrays.

complexity. $N^2 . N \Rightarrow O(N^3)$

q) How can you reduce complexity.
with prefix sums can be reduced to $O(N^2)$

q) Can we reduce more? $O(N)$?
Yes. Try dynamic programming.

Sol 2:

Let $f(i)$ return MSS of first i elements ending at position i.

$f(i+1) =$ either use $A[i+1]$ to extend window ending at position i OR start a new window has only $A[i+1]$

$f(i) = \max(f(i-1) + A[i], A[i])$
Base case : $f(0) = 0$

$dp[0] = 0$
$dp[i] = A[i] + \max(dp[i-1], 0)$

1	2	3	4	5	6	7	8
-2	-3	4	-1	-2	1	5	-3

-2 -3 4 3 1 2 7 4

q) T.C
$O(N)$

DO NOT SHARE

Addition Chain

A.C is a sequence of integers such that the first number is 1 and every subsequent number is sum of two (not necessarily unique) numbers in the list.

1 2 3 5

Find the minimum length chain that ends with a given number.

10	: 1 2 4 8 10	5
5	: 1 2 3 5	4
13	: 1 2 4 6 7 13 1 2 4 8 12 13	6
8	: 1 2 4 8	4
32	: 1 2 4 8 16 32	6
24	: 1 2 4 8 16 24	
26	: 1 2 4 8 16 24 26	

Solution Idea

Maximize cur num. (x2)

Greedily add max number you can until you get N.

a) Does this work? If not find a counter example.

30	: 1 2 4 8 16 24 28 30	8
	: 1 2 4 5 10 15 30	7
	: 1 2 4 6 12 24 30	7

There is no greedy solution to this problem.

Mostly your greedy solution ideas convinces you.

Before implementing greedy solution proove that it actually works.

Croatia '16 7th round question

Given $N \leq 100.000$ numbers as a list. $(v_i \leq 10^6)$

In each step select a number & eliminate its multiples from the list.

What is the min number of steps to empty the list

List: 6 9 11 12 16 33

Number	Remaining list
3	11 16
11	16
2	$\emptyset$

$\Rightarrow$ 3 steps to empty the list.

DO NOT SHARE

9) Does it matter in which order we select the number?

6 9 22 12 16 33

Number	Remaining list
11	6 9 12 16
2	9
3	∅

} 3 steps

Number	Remaining list
2	9 33
3	∅

} 2 steps

while list not empty.
  choose the most eliminating prime
  eliminate numbers in the list

9) Does this approach work? If not find a counter example.

5 7 10 14 20 28

5 eliminates 3
7 eliminates 3
2 eliminates 4

⇒

2	5 7
5	7
7	∅

} 3 steps

Though we can solve without 2.

5	7 14 28
7	∅

} 2 steps

problem does not have a greedy solution.

DO NOT SHARE

MCSS - MCSP (Maximum Cont. Subarray Sum/Product)

Tuesday, April 17, 2018 2:32 PM

Given a list of numbers Find cont. subarray with the largest sum.

9

-2 1 -3 $\underbrace{4\ -1\ 2\ 1}$ -5 4

dp[i] max subarray sum ending with i^{th} element.

$$dp[i] = \max(dp[i-1]+v[i],\ v[i])$$

extend previous window by adding i^{th} element

start a new solution.

9) Do we really need dp array here?

```
val = max(val, 0) + v[i]      // make a greedy choice of
ans = max(ans, val)              0 or positive subarray sum
```

Given a list of numbers Find cont. subarray with the largest product.

N = 6 5 6 -1 2 -2 5

Solution:

for each i,j pair
calc [i:j] $\Rightarrow$ [i:j]
N^2

	5	6	-1	2	-2	5	-40
	5	30	-30	-60	120	600	-24000
		6	-6	-12	24	120	-4800
			-1	-2	4	20	-800
				2	-4	-20	800
					-2	-10	400
						5	-200
max product ending at i :	5	30	-1	2	120	600	800
min product ending at i :	1	6	-30	-60	-4	-20	-24000

Greedy Approach?

for each end position i
Keep min & max product ending at position i

```
val1=1; val2=1;
for i:0 -> N
    nmax = max(v[i], v[i]*val1, v[i]*val2)
    nmin = min(v[i], v[i]*val1, v[i]*val2)
    val1 = nmax
    val2 = nmin
    ans = max(ans, val1, val2)
```

DO NOT SHARE

Barn Repair

Tuesday, April 17, 2018 4:09 PM

$M \leq 50$ max number of boards

$S \leq 200$ stalls

$C \leq S$ occupied stalls

minimum number of stalls must be blocked in order to block all occupied stalls

M	S	C
4	50	18

3 4 6 8	14 15 16 17	21	25 26 27 30 31	40 41 42 43	
6	8		7	4	= 25 → Both solutions are valid.
6	4		11	4	= 25

Solution: Imagine installing one large block in front of occupied stalls.

its size will be : max-min+1

3 4 6 8 14 15 16 17 21 25 26 27 30 31 40 41 42 43

size: 43-3+1

: 41 stalls blocked.

We want to make 4 pieces of blocks.

By choosing 3 gaps, we can divide the block into 4 pieces.

q) Where should we select the gaps.

Since we want to maximize the unblocked stalls.

Always we have to choose maximum gap.

⇒ If we want 2 pieces we have to cancel 31-40 interval.

we want 3 pieces next we have to choose 8-14 interval to cancel.

... and so on.

q) Is it ever possible to choose a smaller gap and reach more unblocked unoccupied?

Assume gaps are

6 4 4 3 9 2.

We want to maximize the total and select 4 of those numbers.

It is never possible to reach a higher sum by skipping bigger number and selecting smaller number.

```
for i: 1 → c
    s[i]
    dif[i] = s[i] - s[i-1]
sort (dif)
```

$unblock = \sum_{i=1}^{m} dif_i$ total of max m gaps

DO NOT SHARE

Selfish Grazing

Tuesday, April 17, 2018 6:35 PM

$N \leq 50.000$ cows

$1 \leq S_i < E_i \leq 10^8$ Start & End of each cow grazing range

Find maximum number of cows that can graze without overlapping grazing areas.

N = 5

DP Solution:

let dp[k] max number of non overlapping cows among first k cows.

$dp[i] = \max(dp[i], dp[j] + 1)$

do not include $i+1^{th}$ cow

use i^{th} cow after j^{th} cow

s.t

cow[j].ending $\leq$ cow[i].start

if cows[] sorted according to ending positions j can be found in $O(\log N)$

otherwise $O(N)$

$dp[0] = 0$

$dp[1] = \max(\underset{0}{dp[0]}, \underset{1}{dp[0]+1}) = 1$

$dp[2] = \max(\underset{1}{dp[1]}, \underset{1+1}{dp[1]+1}) = 2$

$dp[3] = \max(\underset{2}{dp[2]}, dp[2]+1) = 3$

$dp[4] = \max(\underset{3}{dp[3]}, \underset{2+1}{dp[2]+1}) = 3$

$dp[5] = \max(\underset{3}{dp[4]}, \underset{1}{dp[0]+1}) = 3$

Greedy Approach

Sort the cows according to end points. In each step, greedily choose the best cow.

Sort on endpoints.
Go over all cows.
if possible add cow
update end time
update ans

Possible to Include	current end time
✓	0
✓	4
✓	5
✗	5
✗	5

Always take the first available cow
min ending point & start point > current.end.

```
curend = 0      count = 0
for cow: 1 → N
    if cow.start ≥ curend
        curend = cow.end
        count++
```

DO NOT SHARE