

Misc

1. Write a C program to reverse digits of a number

ITERATIVE WAY

Algorithm:

```
Input: num
(1) Initialize rev_num = 0
(2) Loop while num > 0
    (a) Multiply rev_num by 10 and add remainder of num
        divide by 10 to rev_num
        rev_num = rev_num*10 + num%10;
    (b) Divide num by 10
(3) Return rev_num
```

Example:

num = 4562

rev_num = 0

$rev_num = rev_num * 10 + num \% 10 = 2$

$num = num / 10 = 456$

$rev_num = rev_num * 10 + num \% 10 = 20 + 6 = 26$

$num = num / 10 = 45$

$rev_num = rev_num * 10 + num \% 10 = 260 + 5 = 265$

$num = num / 10 = 4$

$rev_num = rev_num * 10 + num \% 10 = 265 + 4 = 2654$

$num = num / 10 = 0$

Program:

```

#include <stdio.h>

/* Iterative function to reverse digits of num*/
int reversDigits(int num)
{
    int rev_num = 0;
    while(num > 0)
    {
        rev_num = rev_num*10 + num%10;
        num = num/10;
    }
    return rev_num;
}

/*Driver program to test reversDigits*/
int main()
{
    int num = 4562;
    printf("Reverse of no. is %d", reversDigits(num));

    getchar();
    return 0;
}

```

Time Complexity: $O(\log(n))$ where n is the input number.

RECURSIVE WAY

Thanks to Raj for adding this to the original post.

```

#include <stdio.h>;

/* Recursive function to reverse digits of num*/
int reversDigits(int num)
{
    static int rev_num = 0;
    static int base_pos = 1;
    if(num > 0)
    {
        reversDigits(num/10);
        rev_num += (num%10)*base_pos;
        base_pos *= 10;
    }
    return rev_num;
}

/*Driver program to test reversDigits*/
int main()
{
    int num = 4562;
    printf("Reverse of no. is %d", reversDigits(num));

    getchar();
    return 0;
}

```

Time Complexity: $O(\log(n))$ where n is the input number

Note that above above program doesn't consider leading zeroes. For example, for 100

program will print 1. If you want to print 001 then see this comment from Maheshwar.

Try extensions of above functions that should also work for floating point numbers.

2. Power Set

Power Set Power set $P(S)$ of a set S is the set of all subsets of S . For example $S = \{a, b, c\}$ then $P(s) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$.

If S has n elements in it then $P(s)$ will have 2^n elements

Algorithm:

Input: Set[], set_size

1. Get the size of power set
 $\text{powet_set_size} = \text{pow}(2, \text{set_size})$
2. Loop for counter from 0 to pow_set_size
 - (a) Loop for $i = 0$ to set_size
 - (i) If i th bit in counter is set
 Print i th element from set for this subset
 - (b) Print separator for subsets i.e., newline

Example:

Set = [a,b,c]

$\text{power_set_size} = \text{pow}(2, 3) = 8$

Run for binary counter = 000 to 111

Value of Counter	Subset
000	-> Empty set
001	-> a
011	-> ab
100	-> c
101	-> ac
110	-> bc
111	-> abc

Program:

```

#include <stdio.h>
#include <math.h>

void printPowerSet(char *set, int set_size)
{
    /*set_size of power set of a set with set_size
    n is (2*n -1)*/
    unsigned int pow_set_size = pow(2, set_size);
    int counter, j;

    /*Run from counter 000..0 to 111..1*/
    for(counter = 0; counter < pow_set_size; counter++)
    {
        for(j = 0; j < set_size; j++)
        {
            /* Check if jth bit in the counter is set
            If set then print jth element from set */
            if(counter & (1<<j))
                printf("%c", set[j]);
        }
        printf("\n");
    }

    /*Driver program to test printPowerSet*/
    int main()
    {
        char set[] = {'a','b','c'};
        printPowerSet(set, 3);

        getchar();
        return 0;
    }
}

```

Time Complexity: $O(2^n \cdot n)$

There are more efficient ways of doing this. Will update here soon with more efficient method.

References:

http://en.wikipedia.org/wiki/Power_set

3. 8 queen problem

The eight queens problem is the problem of placing eight queens on an 8×8 chessboard such that none of them attack one another (no two are in the same row, column, or diagonal). More generally, the n queens problem places n queens on an n×n chessboard.

There are different solutions for the problem.

You can find detailed solutions at

4. Ugly Numbers

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.

Write a program to find and print the 150'th ugly number.

METHOD 1 (Simple)

Thanks to [Nedylko Draganov](#) for suggesting this solution.

Algorithm:

Loop for all positive integers until ugly number count is smaller than n, if an integer is ugly then 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.

Implementation:

```

#include<stdio.h>
#include<stdlib.h>

/*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;
}

/* Driver program to test above functions */
int main()
{
    unsigned no = getNthUglyNo(150);
    printf("150th ugly no. is %d ", no);
    getchar();
    return 0;
}

```

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) $1 \times 2, 2 \times 2, 3 \times 2, 4 \times 2, 5 \times 2, \dots$

(2) $1 \times 3, 2 \times 3, 3 \times 3, 4 \times 3, 5 \times 3, \dots$

(3) $1 \times 5, 2 \times 5, 3 \times 5, 4 \times 5, 5 \times 5, \dots$

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 subsequence. Every step we choose the smallest one, and move one step after.

Algorithm:

```
1 Declare an array for ugly numbers: ugly[150]
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_multiple_of_2 = ugly[i2]*2;
      next_multiple_of_3 = ugly[i3]*3
      next_multiple_of_5 = ugly[i5]*5;
5 Now go in a loop to fill all ugly numbers till 150:
For (i = 1; i < 150; i++)
{
    /* These small steps are not optimized for good
       readability. Will optimize them in C program */
    next_ugly_no = Min(next_multiple_of_2,
                       next_multiple_of_3,
                       next_multiple_of_5);
    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;
    }
    ugly[i] = next_ugly_no
}/* 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 )
```

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 )
```

Will continue same way till $I < 150$

Program:


```

#include<stdio.h>
#include<stdlib.h>
#define bool int

/* Function to find minimum of 3 numbers */
unsigned min(unsigned , unsigned , unsigned );

/* Function to get the nth ugly number*/
unsigned getNthUglyNo(unsigned n)
{
    unsigned *ugly =
        (unsigned *) (malloc (sizeof(unsigned)*n));
    unsigned i2 = 0, i3 = 0, i5 = 0;
    unsigned i;
    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(i=1; i<n; i++)
    {
        next_ugly_no = min(next_multiple_of_2,
                           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;
}

/* Function to find minimum of 3 numbers */
unsigned min(unsigned a, unsigned b, unsigned c)
{
    if(a <= b)
    {
        if(a <= c)
            return a;
        else
            return c;
    }
    if(b <= c)
        return b;
    else
        return c;
}

/* Driver program to test above functions */

```

```
int main()
{
    unsigned no = getNthUglyNo(150);
    printf("%dth ugly no. is %d ", 150, no);
    getchar();
    return 0;
}
```

Algorithmic Paradigm: Dynamic Programming

Time Complexity: $O(n)$

Storage Complexity: $O(n)$

Please write comments if you find any bug in the above program or other ways to solve the same problem.

5. Lucky Numbers

Lucky numbers are subset of integers. Rather than going into much theory, let us see the process of arriving at lucky numbers,

Take the set of integers

1,2,3,4,5,6,7,8,9,10,11,12,14,15,16,17,18,19,.....

First, delete every second number, we get following reduced set.

1,3,5,7,9,11,13,15,17,19,.....

Now, delete every third number, we get

1, 3, 7, 9, 13, 15, 19,.....

Continue this process indefinitely.....

Any number that does NOT get deleted due to above process is called "lucky".

Therefore, set of lucky numbers is 1, 3, 7, 13,.....

Now, given an integer 'n', write a function to say whether this number is lucky or not.

```
bool isLucky(int n)
```

Algorithm:

Before every iteration, if we calculate position of the given no, then in a given iteration, we can determine if the no will be deleted. Suppose calculated position for the given no. is P before some iteration, and each l th no. is going to be removed in this iteration, if $P < l$ then input no is lucky, if P is such that $P \% l == 0$ (l is a divisor of P), then input no is not lucky.

Recursive Way:

```

#include <stdio.h>
#define bool int

/* Returns 1 if n is a lucky no. ohterwise returns 0*/
bool isLucky(int n)
{
    static int counter = 2;

    /*variable next_position is just for readability of
    the program we can remove it and use n only */
    int next_position = n;
    if(counter > n)
        return 1;
    if(n%counter == 0)
        return 0;

    /*calculate next position of input no*/
    next_position -= next_position/counter;

    counter++;
    return isLucky(next_position);
}

/*Driver function to test above function*/
int main()
{
    int x = 5;
    if( isLucky(x) )
        printf("%d is a lucky no.", x);
    else
        printf("%d is not a lucky no.", x);
    getchar();
}

```

Example:

Let's us take an example of 19

1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,15,17,18,19,20,21,.....

1,3,5,7,9,11,13,15,17,19,.....

1,3,7,9,13,15,19,.....

1,3,7,13,15,19,.....

1,3,7,13,19,.....

In next step every 6th no .in sequence will be deleted. 19 will not be deleted after this step because position of 19 is 5th after this step. Therefore, 19 is lucky. Let's see how above C code finds out:

Current function call	Position after this call	Counter for next call	Next Call
isLucky(19)	10	3	isLucky(10)
isLucky(10)	7	4	isLucky(7)
isLucky(7)	6	5	isLucky(6)
isLucky(6)	5	6	isLucky(5)

When isLucky(6) is called, it returns 1 (because counter > n).

Iterative Way:

Please see [this](#) comment for another simple and elegant implementation of the above algorithm.

Please write comments if you find any bug in the given programs or other ways to solve the same problem.

6. Write a program to add two numbers in base 14

Asked by Anshya.

Below are the different ways to add base 14 numbers.

Method 1

Thanks to Raj for suggesting this method.

1. Convert both i/p base 14 numbers to base 10.
2. Add numbers.
3. Convert the result back to base 14.

Method 2

Just add the numbers in base 14 in same way we add in base 10. Add numerals of both numbers one by one from right to left. If there is a carry while adding two numerals, consider the carry for adding next numerals.

Let us consider the presentation of base 14 numbers same as hexadecimal numbers

```
A --> 10
B --> 11
C --> 12
D --> 13
```

Example:

```
num1 =      1  2  A
num2 =      C  D  3
```

1. Add A and 3, we get 13(D). Since 13 is smaller than 14, carry becomes 0 and resultant numeral becomes D

2. Add 2, D and carry(0). we get 15. Since 15 is greater than 13, carry becomes 1 and resultant numeral is $15 - 14 = 1$

3. Add 1, C and carry(1). we get 14. Since 14 is greater than 13, carry becomes 1 and resultant numeral is $14 - 14 = 0$

Finally, there is a carry, so 1 is added as leftmost numeral and the result become:
101D

Implementation of Method 2

```
# include <stdio.h>
# include <stdlib.h>
# define bool int

int getNumeralValue(char );
char getNumeral(int );

/* Function to add two numbers in base 14 */
char *sumBase14(char *num1, char *num2)
{
    int l1 = strlen(num1);
    int l2 = strlen(num2);
    char *res;
    int i;
    int nml1, nml2, res_nml;
    bool carry = 0;

    if(l1 != l2)
    {
        printf("Function doesn't support numbers of different"
               " lengths. If you want to add such numbers then"
               " prefix smaller number with required no. of zeroes");
        getchar();
        assert(0);
    }

    /* Note the size of the allocated memory is one
       more than i/p lengths for the cases where we
       have carry at the last like adding D1 and A1 */
    res = (char *)malloc(sizeof(char)*(l1 + 1));

    /* Add all numerals from right to left */
    for(i = l1-1; i >= 0; i--)
    {
        /* Get decimal values of the numerals of
           i/p numbers*/
        nml1 = getNumeralValue(num1[i]);
        nml2 = getNumeralValue(num2[i]);

        /* Add decimal values of numerals and carry */
        res_nml = carry + nml1 + nml2;

        /* Check if we have carry for next addition
           of numerals */
        if(res_nml >= 14)
        {
            carry = 1;
            res_nml -= 14;
        }
        else
        {
            carry = 0;
        }
        res[i+1] = getNumeral(res_nml);
    }

    /* if there is no carry after last iteration
```

```

    /* if there is no carry after last iteration
    then result should not include 0th character
    of the resultant string */
    if(carry == 0)
        return (res + 1);

    /* if we have carry after last iteration then
    result should include 0th character */
    res[0] = '1';
    return res;
}

/* Function to get value of a numeral
For example it returns 10 for input 'A'
1 for '1', etc */
int getNumeralValue(char num)
{
    if( num >= '0' && num <= '9')
        return (num - '0');
    if( num >= 'A' && num <= 'D')
        return (num - 'A' + 10);

    /* If we reach this line caller is giving
    invalid character so we assert and fail*/
    assert(0);
}

/* Function to get numeral for a value.
For example it returns 'A' for input 10
'1' for 1, etc */
char getNumeral(int val)
{
    if( val >= 0 && val <= 9)
        return (val + '0');
    if( val >= 10 && val <= 14)
        return (val + 'A' - 10);

    /* If we reach this line caller is giving
    invalid no. so we assert and fail*/
    assert(0);
}

/*Driver program to test above functions*/
int main()
{
    char *num1 = "DC2";
    char *num2 = "0A3";

    printf("Result is %s", sumBase14(num1, num2));
    getchar();
    return 0;
}

```

Notes:

Above approach can be used to add numbers in any base. We don't have to do string operations if base is smaller than 10.

You can try extending the above program for numbers of different lengths.

Please comment if you find any bug in the program or a better approach to do the same.

7. Babylonian method for square root

Algorithm:

This method can be derived from (but predates) Newton–Raphson method.

- 1 Start with an arbitrary positive start value x (the closer to the root, the better).
- 2 Initialize $y = 1$.
3. Do following until desired approximation is achieved.
 - a) Get the next approximation for root using average of x and y
 - b) Set $y = n/x$

Implementation:

```
/*Returns the square root of n. Note that the function */
float squareRoot(float n)
{
    /*We are using n itself as initial approximation
    This can definitely be improved */
    float x = n;
    float y = 1;
    float e = 0.000001; /* e decides the accuracy level*/
    while(x - y > e)
    {
        x = (x + y)/2;
        y = n/x;
    }
    return x;
}
```

```
/* Driver program to test above function*/
int main()
{
    int n = 50;
    printf ("Square root of %d is %f", n, squareRoot(n));
    getchar();
}
```

Example:

```
n = 4 /*n itself is used for initial approximation*/
Initialize x = 4, y = 1
Next Approximation x = (x + y)/2 (= 2.500000),
y = n/x (=1.600000)
Next Approximation x = 2.050000,
y = 1.951220
Next Approximation x = 2.000610,
```

```
y = 1.999390
Next Approximation x = 2.000000,
y = 2.000000
Terminate as (x - y) > e now.
```

If we are sure that n is a perfect square, then we can use following method. The method can go in infinite loop for non-perfect-square numbers. For example, for 3 the below while loop will never terminate.

```
/*Returns the square root of n. Note that the function
will not work for numbers which are not perfect squares*/
unsigned int squareRoot(int n)
{
    int x = n;
    int y = 1;
    while(x > y)
    {
        x = (x + y)/2;
        y = n/x;
    }
    return x;
}

/* Driver program to test above function*/
int main()
{
    int n = 49;
    printf (" root of %d is %d", n, squareRoot(n));
    getchar();
}
```

References;

http://en.wikipedia.org/wiki/Square_root

http://en.wikipedia.org/wiki/Babylonian_method#Babylonian_method

Asked by Snehal

Please write comments if you find any bug in the above program/algorithm, or if you want to share more information about Babylonian method.

8. Multiply two integers without using multiplication, division and bitwise operators, and no loops

Asked by Kafil

By making use of recursion, we can multiply two integers with the given constraints.

To multiply x and y , recursively add x y times.

Thanks to [geek4u](#) for suggesting this method.

```
#include<stdio.h>
/* function to multiply two numbers x and y*/
int multiply(int x, int y)
{
    /* 0 multiplied with anything gives 0 */
    if(y == 0)
        return 0;

    /* Add x one by one */
    if(y > 0 )
        return (x + multiply(x, y-1));

    /* the case where y is negative */
    if(y < 0 )
        return -multiply(x, -y);
}

int main()
{
    printf("\n %d", multiply(5, -11));
    getchar();
    return 0;
}
```

Time Complexity: $O(y)$ where y is the second argument to function `multiply()`.

Please write comments if you find any of the above code/algorithm incorrect, or find better ways to solve the same problem.

9. Implement Queue using Stacks

A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be `stack1` and `stack2`. q can be implemented in two ways:

Method 1 (By making `enqueue` operation costly)

This method makes sure that newly entered element is always at the top of `stack1`, so that `deQueue` operation just pops from `stack1`. To put the element at top of `stack1`, `stack2` is used.

```
enqueue(q, x)
1) While stack1 is not empty, push everything from stack1 to stack2.
2) Push x to stack1 (assuming size of stacks is unlimited).
3) Push everything back to stack1.

deQueue(q)
1) If stack1 is empty then error
```

```
2) Pop an item from stack1 and return it
```

Method 2 (By making deQueue operation costly)

In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

```
enqueue(q, x)
    1) Push x to stack1 (assuming size of stacks is unlimited).

dequeue(q)
    1) If both stacks are empty then error.
    2) If stack2 is empty
        While stack1 is not empty, push everything from stack1 to stack2.
    3) Pop the element from stack2 and return it.
```

Method 2 is definitely better than method 1. Method 1 moves all the elements twice in enqueue operation, while method 2 (in dequeue operation) moves the elements once and moves elements only if stack2 is empty.

Implementation of method 2:

```
/* Program to implement a queue using two stacks */
#include<stdio.h>
#include<stdlib.h>

/* structure of a stack node */
struct sNode
{
    int data;
    struct sNode *next;
};

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data);

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref);

/* structure of queue having two stacks */
struct queue
{
    struct sNode *stack1;
    struct sNode *stack2;
};

/* Function to enqueue an item to queue */
void enqueue(struct queue *q, int x)
{
    push(&q->stack1, x);
}

/* Function to dequeue an item from queue */
int dequeue(struct queue *q)
{
    int x;
```

```

/* If both stacks are empty then error */
if(q->stack1 == NULL && q->stack2 == NULL)
{
    printf("Q is empty");
    getchar();
    exit(0);
}

/* Move elements from stack1 to stack 2 only if
   stack2 is empty */
if(q->stack2 == NULL)
{
    while(q->stack1 != NULL)
    {
        x = pop(&q->stack1);
        push(&q->stack2, x);
    }
}

x = pop(&q->stack2);
return x;
}

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data)
{
    /* allocate node */
    struct sNode* new_node =
        (struct sNode*) malloc(sizeof(struct sNode));

    if(new_node == NULL)
    {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*top_ref);

    /* move the head to point to the new node */
    (*top_ref) = new_node;
}

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref)
{
    int res;
    struct sNode *top;

    /*If stack is empty then error */
    if(*top_ref == NULL)
    {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }
    else
    {

```

```

    top = *top_ref;
    res = top->data;
    *top_ref = top->next;
    free(top);
    return res;
}
}

/* Driver function to test above functions */
int main()
{
    /* Create a queue with items 1 2 3*/
    struct queue *q = (struct queue*)malloc(sizeof(struct queue));
    q->stack1 = NULL;
    q->stack2 = NULL;
    enqueue(q, 1);
    enqueue(q, 2);
    enqueue(q, 3);

    /* Dequeue items */
    printf("%d ", dequeue(q));
    printf("%d ", dequeue(q));
    printf("%d ", dequeue(q));

    getchar();
}

```

Queue can also be implemented using one user stack and one Function Call Stack.

Below is modified Method 2 where recursion (or Function Call Stack) is used to implement queue using only one user defined stack.

```

enqueue(x)
    1) Push x to stack1.

deQueue:
    1) If stack1 is empty then error.
    2) If stack1 has only one element then return it.
    3) Recursively pop everything from the stack1, store the popped item
        in a variable res, push the res back to stack1 and return res

```

The step 3 makes sure that the last popped item is always returned and since the recursion stops when there is only one item in *stack1* (step 2), we get the last element of *stack1* in *dequeue()* and all other items are pushed back in step 3.

Implementation of method 2 using Function Call Stack:

```

/* Program to implement a queue using one user defined stack and one F
#include<stdio.h>
#include<stdlib.h>

/* structure of a stack node */
struct sNode
{
    int data;
    struct sNode *next;
};

```

```

/* structure of queue having two stacks */
struct queue
{
    struct sNode *stack1;
};

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data);

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref);

/* Function to enqueue an item to queue */
void enqueue(struct queue *q, int x)
{
    push(&q->stack1, x);
}

/* Function to dequeue an item from queue */
int dequeue(struct queue *q)
{
    int x, res;

    /* If both stacks are empty then error */
    if(q->stack1 == NULL)
    {
        printf("Q is empty");
        getchar();
        exit(0);
    }
    else if(q->stack1->next == NULL)
    {
        return pop(&q->stack1);
    }
    else
    {
        /* pop an item from the stack1 */
        x = pop(&q->stack1);

        /* store the last dequeued item */
        res = dequeue(q);

        /* push everything back to stack1 */
        push(&q->stack1, x);

        return res;
    }
}

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data)
{
    /* allocate node */
    struct sNode* new_node =
        (struct sNode*) malloc(sizeof(struct sNode));

    if(new_node == NULL)
    {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }
}

```

```

/* put in the data */
new_node->data = new_data;

/* link the old list off the new node */
new_node->next = (*top_ref);

/* move the head to point to the new node */
(*top_ref) = new_node;
}

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref)
{
    int res;
    struct sNode *top;

    /*If stack is empty then error */
    if(*top_ref == NULL)
    {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }
    else
    {
        top = *top_ref;
        res = top->data;
        *top_ref = top->next;
        free(top);
        return res;
    }
}

/* Driver function to test above functions */
int main()
{
    /* Create a queue with items 1 2 3*/
    struct queue *q = (struct queue*)malloc(sizeof(struct queue));
    q->stack1 = NULL;

    enqueue(q, 1);
    enqueue(q, 2);
    enqueue(q, 3);

    /* Dequeue items */
    printf("%d ", dequeue(q));
    printf("%d ", dequeue(q));
    printf("%d ", dequeue(q));

    getchar();
}

```

Please write comments if you find any of the above codes/algorithms incorrect, or find better ways to solve the same problem.

10. Check for balanced parentheses in an expression

Given an expression string exp, write a program to examine whether the pairs and the orders of “{”, “}”, “(”, “)”, “[”, “]” are correct in exp. For example, the program should print true for exp = “{[]}{}{[(())]}”) and false for exp = “[()]”

Algorithm:

- 1) Declare a character stack S.
- 2) Now traverse the expression string exp.
 - a) If the current character is a starting bracket ('(', '{' or '[') then push it to stack.
 - b) If the current character is a closing bracket (') or '}' or ']') then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.
- 3) After complete traversal, if there is some starting bracket left in stack then “not balanced”

Implementation:

```
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* structure of a stack node */
struct sNode
{
    char data;
    struct sNode *next;
};

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data);

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref);

/* Returns 1 if character1 and character2 are matching left
and right Parenthesis */
bool isMatchingPair(char character1, char character2)
{
    if(character1 == '(' && character2 == ')')
        return 1;
    else if(character1 == '{' && character2 == '}')
        return 1;
    else if(character1 == '[' && character2 == ']')
        return 1;
    else
        return 0;
}

/*Return 1 if expression has balanced Parenthesis */
bool areParenthesisBalanced(char exp[])
{
    int i = 0;
```

```

/* Declare an empty character stack */
struct sNode *stack = NULL;

/* Traverse the given expression to check matching parenthesis */
while(exp[i])
{
    /*If the exp[i] is a starting parenthesis then push it*/
    if(exp[i] == '{' || exp[i] == '(' || exp[i] == '[')
        push(&stack, exp[i]);

    /* If exp[i] is a ending parenthesis then pop from stack and
       check if the popped parenthesis is a matching pair*/
    if(exp[i] == '}' || exp[i] == ')' || exp[i] == ']')
    {
        /*If we see an ending parenthesis without a pair then return
        if(stack == NULL)
            return 0;

        /* Pop the top element from stack, if it is not a pair
        parenthesis of character then there is a mismatch.
        This happens for expressions like {()} */
        else if ( !isMatchingPair(pop(&stack), exp[i]) )
            return 0;
        }
        i++;
    }

    /* If there is something left in expression then there is a starting
    parenthesis without a closing parenthesis */
    if(stack == NULL)
        return 1; /*balanced*/
    else
        return 0; /*not balanced*/
}

/* UTILITY FUNCTIONS */
/*driver program to test above functions*/
int main()
{
    char exp[100] = "{(){}[]}";
    if(areParenthesisBalanced(exp))
        printf("\n Balanced ");
    else
        printf("\n Not Balanced "); \
    getchar();
}

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data)
{
    /* allocate node */
    struct sNode* new_node =
        (struct sNode*) malloc(sizeof(struct sNode));

    if(new_node == NULL)
    {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }

    /* put in the data */

```



```

/* put in the data */
new_node->data = new_data;

/* link the old list off the new node */
new_node->next = (*top_ref);

/* move the head to point to the new node */
(*top_ref) = new_node;
}

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref)
{
    char res;
    struct sNode *top;

    /*If stack is empty then error */
    if(*top_ref == NULL)
    {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }
    else
    {
        top = *top_ref;
        res = top->data;
        *top_ref = top->next;
        free(top);
        return res;
    }
}

```

Time Complexity: $O(n)$

Auxiliary Space: $O(n)$ for stack.

Please write comments if you find any bug in above codes/algorithms, or find other ways to solve the same problem

11. Reverse a stack using recursion

You are not allowed to use loop constructs like while, for..etc, and you can only use the following ADT functions on Stack S:

isEmpty(S)

push(S)

pop(S)

Solution:

The idea of the solution is to hold all values in Function Call Stack until the stack becomes empty. When the stack becomes empty, insert all held items one by one at the

bottom of the stack.

For example, let the input stack be

```
1  <-- top
2
3
4
```

First 4 is inserted at the bottom.

```
4  <-- top
```

Then 3 is inserted at the bottom

```
4  <-- top
3
```

Then 2 is inserted at the bottom

```
4  <-- top
3
2
```

Then 1 is inserted at the bottom

```
4  <-- top
3
2
1
```

So we need a function that inserts at the bottom of a stack using the above given basic stack function. **//Below is a recursive function that inserts an element at the bottom of a stack.**

```

void insertAtBottom(struct sNode** top_ref, int item)
{
    int temp;
    if(isEmpty(*top_ref))
    {
        push(top_ref, item);
    }
    else
    {
        /* Hold all items in Function Call Stack until we reach end of
        the stack. When the stack becomes empty, the isEmpty(*top_ref)
        becomes true, the above if part is executed and the item is
        inserted at the bottom */
        temp = pop(top_ref);
        insertAtBottom(top_ref, item);

        /* Once the item is inserted at the bottom, push all the
        items held in Function Call Stack */
        push(top_ref, temp);
    }
}

```

//Below is the function that reverses the given stack using insertAtBottom()

```

void reverse(struct sNode** top_ref)
{
    int temp;
    if(!isEmpty(*top_ref))
    {
        /* Hold all items in Function Call Stack until we reach end of
        the stack */
        temp = pop(top_ref);
        reverse(top_ref);

        /* Insert all the items (held in Function Call Stack) one by one
        from the bottom to top. Every item is inserted at the bottom */
        insertAtBottom(top_ref, temp);
    }
}

```

//Below is a complete running program for testing above functions.

```

#include<stdio.h>
#include<stdlib.h>
#define bool int

/* structure of a stack node */
struct sNode
{
    char data;
    struct sNode *next;
};

/* Function Prototypes */
void push(struct sNode** top_ref, int new_data);
int pop(struct sNode** top_ref);
bool isEmpty(struct sNode* top);
void print(struct sNode* top);

```

```

/* Driver program to test above functions */
int main()
{
    struct sNode *s = NULL;
    push(&s, 4);
    push(&s, 3);
    push(&s, 2);
    push(&s, 1);

    printf("\n Original Stack ");
    print(s);
    reverse(&s);
    printf("\n Reversed Stack ");
    print(s);
    getchar();
}

/* Function to check if the stack is empty */
bool isEmpty(struct sNode* top)
{
    return (top == NULL)? 1 : 0;
}

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data)
{
    /* allocate node */
    struct sNode* new_node =
        (struct sNode*) malloc(sizeof(struct sNode));

    if(new_node == NULL)
    {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*top_ref);

    /* move the head to point to the new node */
    (*top_ref) = new_node;
}

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref)
{
    char res;
    struct sNode *top;

    /*If stack is empty then error */
    if(*top_ref == NULL)
    {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }
    else
    {

```

```

    top = *top_ref;
    res = top->data;
    *top_ref = top->next;
    free(top);
    return res;
}
}

/* Function to print a linked list */
void print(struct sNode* top)
{
    printf("\n");
    while(top != NULL)
    {
        printf(" %d ", top->data);
        top = top->next;
    }
}

```

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem.

12. Print all combinations of points that can compose a given number

You can win three kinds of basketball points, 1 point, 2 points, and 3 points. Given a total score n, print out all the combination to compose n.

Examples:

For n = 1, the program should print following:

1

For n = 2, the program should print following:

1 1

2

For n = 3, the program should print following:

1 1 1

1 2

2 1

3

For n = 4, the program should print following:

1 1 1 1

1 1 2

1 2 1

1 3

2 1 1

2 2

3 1

and so on ...

Algorithm:

At first position we can have three numbers 1 or 2 or 3.

First put 1 at first position and recursively call for n-1.

Then put 2 at first position and recursively call for n-2.

Then put 3 at first position and recursively call for n-3.

If n becomes 0 then we have formed a combination that compose n, so print the current combination.

Below is a generalized implementation. In the below implementation, we can change MAX_POINT if there are higher points (more than 3) in the basketball game.

```

#define MAX_POINT 3
#define ARR_SIZE 100
#include<stdio.h>

/* Utility function to print array arr[] */
void printArray(int arr[], int arr_size);

/* The function prints all combinations of numbers 1, 2, ...MAX_POINT
that sum up to n.
i is used in recursion keep track of index in arr[] where next
element is to be added. Initial value of i must be passed as 0 */
void printCompositions(int n, int i)
{
    /* array must be static as we want to keep track
of values stored in arr[] using current calls of
printCompositions() in function call stack*/
    static int arr[ARR_SIZE];

    if (n == 0)
    {
        printArray(arr, i);
    }
    else if(n > 0)
    {
        int k;
        for (k = 1; k <= MAX_POINT; k++)
        {
            arr[i]= k;
            printCompositions(n-k, i+1);
        }
    }
}

/* UTILITY FUNCTIONS */
/* Utility function to print array arr[] */
void printArray(int arr[], int arr_size)
{
    int i;
    for (i = 0; i < arr_size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

/* Driver function to test above functions */
int main()
{
    int n = 5;
    printf("Differnt compositions formed by 1, 2 and 3 of %d are\n", n);
    printCompositions(n, 0);
    getchar();
    return 0;
}

```

Asked by [Aloe](#)

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem.

13. Practice Questions for Recursion | Set 1

Explain the functionality of following functions.

Question 1

```
int fun1(int x, int y)
{
    if(x == 0)
        return y;
    else
        return fun1(x - 1, x + y);
}
```

Answer: The function fun() calculates and returns $((1 + 2 \dots + x-1 + x) + y)$ which is $x(x+1)/2 + y$. For example if x is 5 and y is 2, then fun should return $15 + 2 = 17$.

Question 2

```
void fun2(int arr[], int start_index, int end_index)
{
    if(start_index >= end_index)
        return;
    int min_index;
    int temp;

    /* Assume that minIndex() returns index of minimum value in
       array arr[start_index...end_index] */
    min_index = minIndex(arr, start_index, end_index);

    temp = arr[start_index];
    arr[start_index] = arr[min_index];
    arr[min_index] = temp;

    fun2(arr, start_index + 1, end_index);
}
```

Answer: The function fun2() is a recursive implementation of **Selection Sort**.

Please write comments if you find any of the answers/codes incorrect, or you want to share more information about the topics discussed above.

14. Practice Questions for Recursion | Set 2

Explain the functionality of following functions.

Question 1

```
/* Assume that n is greater than or equal to 1 */
int fun1(int n)
{
    if(n == 1)
        return 0;
    else
        return 1 + fun1(n/2);
}
```

Answer: The function calculates and returns $\lfloor \log_2(n) \rfloor$. For example, if n is between 8 and 15 then fun1() returns 3. If n is between 16 to 31 then fun1() returns 4.

Question 2

```
/* Assume that n is greater than or equal to 0 */
void fun2(int n)
{
    if(n == 0)
        return;

    fun2(n/2);
    printf("%d", n%2);
}
```

Answer: The function fun2() prints binary equivalent of n. For example, if n is 21 then fun2() prints 10101.

Note that above functions are just for practicing recursion, they are not the ideal implementation of the functionality they provide.

Please write comments if you find any of the answers/codes incorrect.

15. Practice Questions for Recursion | Set 3

Explain the functionality of below recursive functions.

Question 1

```
void fun1(int n)
{
    int i = 0;
    if (n > 1)
        fun1(n-1);
    for (i = 0; i < n; i++)
        printf(" * ");
}
```

Answer: Total numbers of stars printed is equal to $1 + 2 + \dots + (n-2) + (n-1) + n$, which is

$$n(n+1)/2.$$

Question 2

```
#define LIMIT 1000
void fun2(int n)
{
    if (n <= 0)
        return;
    if (n > LIMIT)
        return;
    printf("%d ", n);
    fun2(2*n);
    printf("%d ", n);
}
```

Answer: For a positive n , $\text{fun2}(n)$ prints the values of $n, 2n, 4n, 8n \dots$ while the value is smaller than LIMIT . After printing values in increasing order, it prints same numbers again in reverse order. For example $\text{fun2}(100)$ prints 100, 200, 400, 800, 800, 400, 200, 100.

If n is negative, then it becomes a non-terminating recursion and causes overflow.

Please write comments if you find any of the answers/codes incorrect, or you want to share more information about the topics discussed above.

16. Write you own Power without using multiplication(*) and division(/) operators

Method 1 (Using Nested Loops)

We can calculate power by using repeated addition.

For example to calculate 5^6 .

- 1) First 5 times add 5, we get 25. (5^2)
- 2) Then 5 times add 25, we get 125. (5^3)
- 3) Then 5 time add 125, we get 625 (5^4)
- 4) Then 5 times add 625, we get 3125 (5^5)
- 5) Then 5 times add 3125, we get 15625 (5^6)

```

/* Works only if a >= 0 and b >= 0 */
int pow(int a, int b)
{
    if (b == 0)
        return 1;
    int answer = a;
    int increment = a;
    int i, j;
    for(i = 1; i < b; i++)
    {
        for(j = 1; j < a; j++)
        {
            answer += increment;
        }
        increment = answer;
    }
    return answer;
}

/* driver program to test above function */
int main()
{
    printf("\n %d", pow(5, 3));
    getchar();
    return 0;
}

```

Method 2 (Using Recursion)

Recursively add a to get the multiplication of two numbers. And recursively multiply to get a raised to the power b .

```

#include<stdio.h>
/* A recursive function to get a^b
Works only if a >= 0 and b >= 0 */
int pow(int a, int b)
{
    if(b)
        return multiply(a, pow(a, b-1));
    else
        return 1;
}

/* A recursive function to get x*y */
int multiply(int x, int y)
{
    if(y)
        return (x + multiply(x, y-1));
    else
        return 0;
}

/* driver program to test above functions */
int main()
{
    printf("\n %d", pow(5, 3));
    getchar();
    return 0;
}

```

Please write comments if you find any bug in above code/algorithm, or find other ways

to solve the same problem.

17. Print all combinations of balanced parentheses

Write a function to generate all possible n pairs of balanced parentheses.

For example, if n=1

```
{ }
```

for n=2

```
{ } { }
```

```
{ { } }
```

Algorithm:

Keep track of counts of open and close brackets. Initialize these counts as 0.

Recursively call the `_printParenthesis()` function until open bracket count is less than the given n. If open bracket count becomes more than the close bracket count, then put a closing bracket and recursively call for the remaining brackets. If open bracket count is less than n, then put an opening bracket and call `_printParenthesis()` for the remaining brackets.

Thanks to [Shekhu](#) for providing the below code.

```

#include<stdio.h>
#define MAX_SIZE 100

void _printParenthesis(int pos, int n, int open, int close);

/* Wrapper over _printParenthesis() */
void printParenthesis(int n)
{
    if(n > 0)
        _printParenthesis(0, n, 0, 0);
    return;
}

void _printParenthesis(int pos, int n, int open, int close)
{
    static char str[MAX_SIZE];

    if(close == n)
    {
        printf("%s \n", str);
        return;
    }
    else
    {
        if(open > close) {
            str[pos] = '}';
            _printParenthesis(pos+1, n, open, close+1);
        }
        if(open < n) {
            str[pos] = '{';
            _printParenthesis(pos+1, n, open+1, close);
        }
    }
}

/* driver program to test above functions */
int main()
{
    int n = 4;
    printParenthesis(n);
    getchar();
    return 0;
}

```

Please write comments if you find the above codes/algorithms incorrect, or find better ways to solve the same problem.

18. Comma operator should be used carefully

In C and C++, comma is the last operator in [precedence table](#). So comma should be carefully used on right side of an assignment expression. For example, one might expect the output as b = 10 in below program. But program prints b = 20 as assignment has higher precedence over comma and the statement “b = 20, a” becomes equivalent to “(b

= 20), a".

```
#include<stdio.h>
int main()
{
    int a = 10, b;
    b = 20, a;    // b = 20
    printf(" b = %d ", b);
    getchar();
    return 0;
}
```

Putting a bracket with comma makes b = a (or 10).

```
#include<stdio.h>
int main()
{
    int a = 10, b;
    b = (20, a); // b = a
    printf(" b = %d ", b);
    getchar();
    return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

19. How does default virtual behavior differ in C++ and Java ?

Default virtual behavior of methods is opposite in C++ and Java:

In C++, class member methods are non-virtual by default. They can be made virtual by using *virtual* keyword. For example, *Base::show()* is non-virtual in following program and program prints "*Base::show() called*".

```

#include<iostream>

using namespace std;

class Base {
public:
    // non-virtual by default
    void show() {
        cout<<"Base::show() called";
    }
};

class Derived: public Base {
public:
    void show() {
        cout<<"Derived::show() called";
    }
};

int main()
{
    Derived d;
    Base &b = d;
    b.show();
    getch();
    return 0;
}

```

Adding *virtual* before definition of *Base::show()* makes program print "*Derived::show() called*"

In Java, methods are virtual by default and can be made non-virtual by using *final* keyword. For example, in the following java program, *show()* is by default virtual and the program prints "*Derived::show() called*"

```

class Base {

    // virtual by default
    public void show() {
        System.out.println("Base::show() called");
    }
}

class Derived extends Base {
    public void show() {
        System.out.println("Derived::show() called");
    }
}

public class Main {
    public static void main(String[] args) {

```

```

        Base b = new Derived();;
        b.show();
    }
}

```

Unlike C++ non-virtual behavior, if we add *final* before definition of `show()` in *Base*, then the above program fails in compilation.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

20. Practice Questions for Recursion | Set 4

Question 1

Predict the output of following program.

```

#include<stdio.h>
void fun(int x)
{
    if(x > 0)
    {
        fun(--x);
        printf("%d\t", x);
        fun(--x);
    }
}

int main()
{
    int a = 4;
    fun(a);
    getchar();
    return 0;
}

```

Output: 0 1 2 0 3 0 1

```

        fun(4);
        /
        fun(3), print(3), fun(2) (prints 0 1)
        /
        fun(2), print(2), fun(1) (prints 0)
        /
        fun(1), print(1), fun(0) (does nothing)
        /
        fun(0), print(0), fun(-1) (does nothing)

```


Question 2

Predict the output of following program. What does the following fun() do in general?

```
int fun(int a[],int n)
{
    int x;
    if(n == 1)
        return a[0];
    else
        x = fun(a, n-1);
    if(x > a[n-1])
        return x;
    else
        return a[n-1];
}

int main()
{
    int arr[] = {12, 10, 30, 50, 100};
    printf("%d ", fun(arr, 5));
    getchar();
    return 0;
}
```

Output: 100

fun() returns the maximum value in the input array a[] of size n.

Question 3

Predict the output of following program. What does the following fun() do in general?

```
int fun(int i)
{
    if ( i%2 ) return (i++);
    else return fun(fun( i - 1 ));
}

int main()
{
    printf("%d ", fun(200));
    getchar();
    return 0;
}
```

Output: 199

If n is odd then returns n, else returns (n-1). Eg., for n = 12, you get 11 and for n = 11 you get 11. The statement "*return i++;*" returns value of i only as it is a post increment.

Please write comments if you find any of the answers/codes incorrect, or you want to share more information/questions about the topics discussed above.

21. Comparison of Exception Handling in C++ and Java

Both languages use *try*, *catch* and *throw* keywords for exception handling, and meaning of *try*, *catch* and *finally* blocks is also same in both languages. Following are the differences between Java and C++ exception handling.

1) In C++, all types (including primitive and pointer) can be thrown as exception. But in Java only throwable objects (Throwable objects are instances of any subclass of the Throwable class) can be thrown as exception. For example, following type of code works in C++, but similar code doesn't work in Java.

```
#include <iostream>
using namespace std;
int main()
{
    int x = -1;

    // some other stuff
    try {
        // some other stuff
        if( x < 0 )
        {
            throw x;
        }
    }
    catch (int x ) {
        cout << "Exception occurred: thrown value is " << x << endl;
    }
    getch();
    return 0;
}
```

Output:

Exception occurred: thrown value is -1

2) In C++, there is a special catch called “catch all” that can catch all kind of exceptions.

```

#include <iostream>
using namespace std;
int main()
{
    int x = -1;
    char *ptr;

    ptr = new char[256];

    // some other stuff
    try {
        // some other stuff
        if( x < 0 )
        {
            throw x;
        }
        if(ptr == NULL)
        {
            throw " ptr is NULL ";
        }
    }
    catch (...) // catch all
    {
        cout << "Exception occurred: exiting "<< endl;
        exit(0);
    }

    getch();
    return 0;
}

```

Output:

Exception occurred: exiting

In Java, for all practical purposes, we can catch Exception object to catch all kind of exceptions. Because, normally we do not catch Throwable(s) other than Exception(s) (which are Errors)

```

catch(Exception e){
    .....
}

```

3) In Java, there is a block called *finally* that is always executed after the try-catch block. This block can be used to do cleanup work. There is no such block in C++.

```

// creating an exception type
class Test extends Exception { }

class Main {
    public static void main(String args[]) {

        try {
            throw new Test();
        }
    }
}

```

```

        catch(Test t) {
            System.out.println("Got the Test Exception");
        }
        finally {
            System.out.println("Inside finally block ");
        }
    }
}

```

Output:

Got the error

Inside finally block

4) In C++, all exceptions are unchecked. In Java, there are two types of exceptions – checked and unchecked. See [this](#) for more details on checked vs Unchecked exceptions.

5) In Java, a new keyword *throws* is used to list exceptions that can be thrown by a function. In C++, there is no *throws* keyword, the same keyword *throw* is used for this purpose also.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

22. Critical Section

Critical Section:

In simple terms a critical section is group of instructions/statements or region of code that need to be executed atomically ([read this post](#) for atomicity), such as accessing a resource (file, input or output port, global data, etc.).

In concurrent programming, if one thread tries to change the value of shared data at the same time as another thread tries to read the value (i.e. data race across threads), the result is unpredictable.

The access to such shared variable (shared memory, shared files, shared port, etc...) to be synchronized. Few programming languages have built in support for synchronization.

It is critical to understand the importance of race condition while writing kernel mode programming (a device driver, kernel thread, etc.). since the programmer can directly

access and modifying kernel data structures.

A simple solution to critical section can be thought as shown below,

```
acquireLock();  
Process Critical Section  
releaseLock();
```

A thread must acquire a lock prior to executing critical section. The lock can be acquired by only one thread. There are various ways to implement locks in the above pseudo code. Let us discuss them in future articles.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

23. Endian order and binary files

While working with binary files, how do you measure their endianness?

For example, if a programmer is making configuration file in binary format (e.g. on small systems it may not be possible to use XML like text files for configuration, text files require another wrapper/layer over binary files), the same binary file would need to be read on different architectures. In such case endianness issue to be addressed.

Or consider, a binary file is created on little endian machine, can it be read on big endian machine without altering byte order?

We can consider binary file as sequence of increasing addresses starting from low order to high order, each address can store one byte. If we are writing some data to binary file on little endian machine it need not to be altered. Where as if the binary file to be created on big endian machines, the data to be altered.

Note that some file formats define endian ordering. Example are JPEG (big endian) and BMP (little endian).

Related Posts:

1. [Little and Big Endian Mystery](#)
2. [Output of C Programs | Set 14](#)

Article compiled by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

24. Access specifier of methods in interfaces

In Java, all methods in an interface are *public* even if we do not specify *public* with method names. Also, data fields are *public static final* even if we do not mention it with fields names. Therefore, data fields must be initialized.

Consider the following example, *x* is by default *public static final* and *foo()* is *public* even if there are no specifiers.

```
interface Test {  
    int x = 10; // x is public static final and must be initialized here  
    void foo(); // foo() is public  
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

25. When does the worst case of Quicksort occur?

The answer depends on strategy for choosing pivot. In early versions of Quick Sort where leftmost (or rightmost) element is chosen as pivot, the worst occurs in following cases.

- 1) Array is already sorted in same order.
- 2) Array is already sorted in reverse order.
- 3) All elements are same (special case of case 1 and 2)

Since these cases are very common use cases, the problem was easily solved by choosing either a random index for the pivot, choosing the middle index of the partition or (especially for longer partitions) choosing the median of the first, middle and last element of the partition for the pivot. With these modifications, the worst case of Quick sort has less chances to occur, but worst case can still occur if the input array is such that the maximum (or minimum) element is always chosen as pivot.

References:

<http://en.wikipedia.org/wiki/Quicksort>

26. Stability in sorting algorithms

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array. Some sorting algorithms are stable by nature like Insertion sort, Merge Sort, Bubble Sort, etc. And some sorting algorithms are not, like Heap Sort, Quick Sort, etc.

However, any given sorting algo which is not stable can be modified to be stable. There can be sorting algo specific ways to make it stable, but in general, any comparison based sorting algorithm which is not stable by nature can be modified to be stable by changing the key comparison operation so that the comparison of two keys considers position as a factor for objects with equal keys.

References:

<http://www.math.uic.edu/~leon/cs-mcs401-s08/handouts/stability.pdf>

http://en.wikipedia.org/wiki/Sorting_algorithm#Stability

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

27. Applications of Heap Data Structure

Heap Data Structure is generally taught with Heapsort. Heapsort algorithm has limited uses because Quicksort is better in practice. Nevertheless, the Heap data structure itself is enormously used. Following are some uses other than Heapsort.

Priority Queues: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in $O(\log n)$ time. Binomial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also in $O(\log n)$ time which is a $O(n)$ operation in Binary Heap. Heap Implemented priority queues are used in Graph algorithms like [Prim's Algorithm](#) and [Dijkstra's algorithm](#).

Order statistics: The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array. See method 4 and 6 of [this](#) post for details.

References:

<http://net.pku.edu.cn/~course/cs101/2007/resource/Intro2Algorithm/book6/chap07.htm>

http://en.wikipedia.org/wiki/Heap_%28data_structure%29

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

28. Access specifiers for classes or interfaces in Java

In Java, methods and data members of a class/interface can have one of the following four access specifiers. The access specifiers are listed according to their restrictiveness order.

- 1) private
- 2) default (when no access specifier is specified)
- 3) protected
- 4) public

But, the classes and interfaces themselves can have only two access specifiers.

- 1) public
- 2) default (when no access specifier is specified)

We cannot declare class/interface with private or protected access specifiers. For example, following program fails in compilation.

```
//filename: Main.java
protected class Test {}

public class Main {
    public static void main(String args[]) {

    }
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

29. Applications of Queue Data Structure

Queue is used when things don't have to be processed immediately, but have to be processed in **First In First Out** order like **Breadth First Search**. This property of Queue makes it also useful in following kind of scenarios.

- 1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.

2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

See [this](#) for more detailed applications of Queue and Stack.

References:

<http://introcs.cs.princeton.edu/43stack/>

30. Lower bound for comparison based sorting algorithms

The problem of sorting can be viewed as following.

Input: A sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

A sorting algorithm is comparison based if it uses comparison operators to find the order between two numbers. Comparison sorts can be viewed abstractly in terms of decision trees. A decision tree is a **full binary tree** that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size. The execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to a leaf. At each internal node, a comparison $a_i \leq a_j$ is made. The left subtree then dictates subsequent comparisons for $a_i \leq a_j$, and the right subtree dictates subsequent comparisons for $a_i > a_j$. When we come to a leaf, the sorting algorithm has established the ordering. So we can say following about the decision tree.

1) Each of the $n!$ permutations on n elements must appear as one of the leaves of the decision tree for the sorting algorithm to sort properly.

2) Let x be the maximum number of comparisons in a sorting algorithm. The maximum height of the decision tree would be x . A tree with maximum height x has at most 2^x leaves.

After combining the above two facts, we get following relation.

$$n! \leq 2^x$$

Taking Log on both sides.

$$\log n! \leq x$$

```
Since = , we can say
x =
```

Therefore, any comparison based sorting algorithm must make at least $\Omega(n \log_2 n)$ comparisons to sort the input array, and Heapsort and merge sort are asymptotically optimal comparison sorts.

References:

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

31. 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, 141,

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

$$F_n = F_{n-1} + F_{n-2}$$

with seed values

$$F_0 = 0 \quad \text{and} \quad F_1 = 1.$$

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 n th Fibonacci number.

Method 1 (Use recursion)

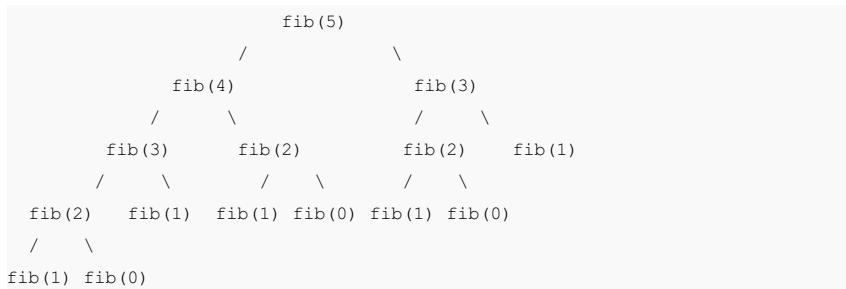
A simple method that is a direct recursive implementation mathematical recurrence relation given above.

```
#include<stdio.h>
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}

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

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.



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

Method 2 (Use Dynamic Programming)

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

```
#include<stdio.h>

int fib(int n)
{
    /* Declare an array to store fibonacci numbers. */
    int f[n+1];
    int i;

    /* 0th and 1st number of the series are 0 and 1*/
    f[0] = 0;
    f[1] = 1;

    for (i = 2; i <= n; i++)
    {
        /* Add the previous 2 numbers in the series
        and store it */
        f[i] = f[i-1] + f[i-2];
    }

    return f[n];
}

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

Time Complexity: $O(n)$

Extra Space: $O(n)$

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 Fibannaci number in series.

```
#include<stdio.h>
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;
}

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

Time Complexity: O(n)

Extra Space: O(1)

Method 4 (Using power of the matrix $\{\{1,1\},\{1,0\}\}$)

This another O(n) which relies on the fact that if we n times multiply the matrix $M = \{\{1,1\}, \{1,0\}\}$ to itself (in other words calculate power(M, n)), then we get the (n+1)th Fibonacci number as the element at row and column (0, 0) in the resultant matrix.

The matrix representation gives the following closed expression for the Fibonacci numbers:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}.$$

```

#include <stdio.h>

/* Helper function that multiplies 2 matrices F and M of size 2*2, and
puts the multiplication result back to F[][] */
void multiply(int F[2][2], int M[2][2]);

/* Helper function that calculates F[][] raise to the power n and puts
result in F[][]
Note that this function is desinged only for fib() and won't work as
power function */
void power(int F[2][2], int n);

int fib(int n)
{
    int F[2][2] = {{1,1},{1,0}};
    if (n == 0)
        return 0;
    power(F, n-1);
    return F[0][0];
}

void multiply(int F[2][2], int M[2][2])
{
    int x = F[0][0]*M[0][0] + F[0][1]*M[1][0];
    int y = F[0][0]*M[0][1] + F[0][1]*M[1][1];
    int z = F[1][0]*M[0][0] + F[1][1]*M[1][0];
    int w = F[1][0]*M[0][1] + F[1][1]*M[1][1];

    F[0][0] = x;
    F[0][1] = y;
    F[1][0] = z;
    F[1][1] = w;
}

void power(int F[2][2], int n)
{
    int i;
    int M[2][2] = {{1,1},{1,0}};

    // n - 1 times multiply the matrix to {{1,0},{0,1}}
    for (i = 2; i <= n; i++)
        multiply(F, M);
}

/* Driver program to test above function */
int main()
{
    int n = 9;
    printf("%d", fib(n));
    getchar();
    return 0;
}

```

Time Complexity: $O(n)$

Extra Space: $O(1)$

Method 5 (Optimized Method 4)

The method 4 can be optimized to work in $O(\log n)$ time complexity. We can do

recursive multiplication to get power(M, n) in the previous method (Similar to the optimization done in [this post](#))

```
#include <stdio.h>

void multiply(int F[2][2], int M[2][2]);

void power(int F[2][2], int n);

/* function that returns nth Fibonacci number */
int fib(int n)
{
    int F[2][2] = {{1,1},{1,0}};
    if (n == 0)
        return 0;
    power(F, n-1);
    return F[0][0];
}

/* Optimized version of power() in method 4 */
void power(int F[2][2], int n)
{
    if (n == 0 || n == 1)
        return;
    int M[2][2] = {{1,1},{1,0}};

    power(F, n/2);
    multiply(F, F);

    if (n%2 != 0)
        multiply(F, M);
}

void multiply(int F[2][2], int M[2][2])
{
    int x = F[0][0]*M[0][0] + F[0][1]*M[1][0];
    int y = F[0][0]*M[0][1] + F[0][1]*M[1][1];
    int z = F[1][0]*M[0][0] + F[1][1]*M[1][0];
    int w = F[1][0]*M[0][1] + F[1][1]*M[1][1];

    F[0][0] = x;
    F[0][1] = y;
    F[1][0] = z;
    F[1][1] = w;
}

/* Driver program to test above function */
int main()
{
    int n = 9;
    printf("%d", fib(9));
    getchar();
    return 0;
}
```

Time Complexity: $O(\text{Log}n)$

Extra Space: $O(\text{Log}n)$ if we consider the function call stack size, otherwise $O(1)$.

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

References:

http://en.wikipedia.org/wiki/Fibonacci_number

<http://www.ics.uci.edu/~epstein/161/960109.html>

32. Practice Questions for Recursion | Set 5

Question 1

Predict the output of following program. What does the following fun() do in general?

```
#include<stdio.h>

int fun(int a, int b)
{
    if (b == 0)
        return 0;
    if (b % 2 == 0)
        return fun(a+a, b/2);
    return fun(a+a, b/2) + a;
}

int main()
{
    printf("%d", fun(4, 3));
    getchar();
    return 0;
}
```

Output: 12

It calculates a*b (a multiplied b).

Question 2

In question 1, if we replace + with * and replace return 0 with return 1, then what does the changed function do? Following is the changed function.

```
#include<stdio.h>

int fun(int a, int b)
{
    if (b == 0)
        return 1;
    if (b % 2 == 0)
        return fun(a*a, b/2);

    return fun(a*a, b/2)*a;
}

int main()
{
    printf("%d", fun(4, 3));
    getchar();
    return 0;
}
```

Output: 64

It calculates a^b (a raised to power b).

Question 3

Predict the output of following program. What does the following fun() do in general?

```
#include<stdio.h>

int fun(int n)
{
    if (n > 100)
        return n - 10;
    return fun(fun(n+11));
}

int main()
{
    printf(" %d ", fun(99));
    getchar();
    return 0;
}
```

Output: 91

```
fun(99) = fun(fun(110)) since 99 > 100
        = fun(100)      since 110 > 100
        = fun(fun(111)) since 100 > 100
        = fun(101)      since 111 > 100
        = 91            since 101 > 100
```

Returned value of fun() is 91 for all integer arguments $n \leq 101$, and $n - 10$ for $n > 101$. This function is known as **McCarthy 91 function**.

Please write comments if you find any of the answers/codes incorrect, or you want to share more information/questions about the topics discussed above.

33. Time Complexity of building a heap

Consider the following algorithm for building a Heap of an input array A.

```
BUILD-HEAP(A)
    heapsize := size(A);
    for i := floor(heapsize/2) downto 1
        do HEAPIFY(A, i);
    end for
END
```

What is the worst case time complexity of the above algo?

Although the worst case complexity looks like $O(n \log n)$, upper bound of time complexity is $O(n)$. See following links for the proof of time complexity.

<http://www.cse.iitk.ac.in/users/sbaswana/Courses/ESO211/heap.pdf/>

http://www.cs.sfu.ca/CourseCentral/307/petra/2009/SLN_2.pdf

34. Dynamic Programming | Set 1 (Overlapping Subproblems Property)

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) Overlapping Subproblems
- 2) Optimal Substructure

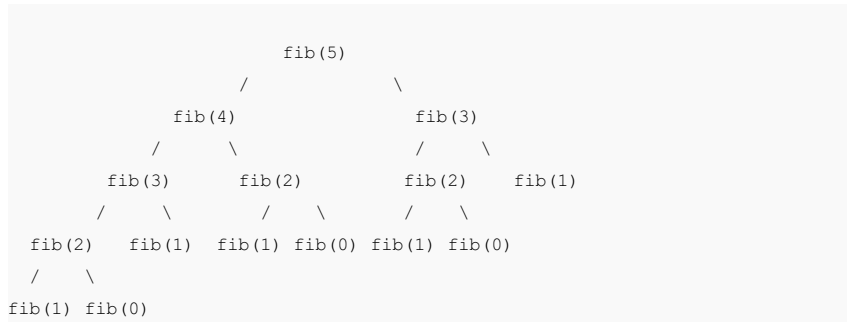
1) Overlapping Subproblems:

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 be 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.

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

Recursion tree for execution of *fib*(5)



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 would have reused the old stored value. 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.

```

/* 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];
}

int main ()
{
    int n = 40;
    _initialize();
    printf("Fibonacci number is %d ", fib(n));
    getchar();
    return 0;
}

```

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.

```

/* 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));
    getchar();
    return 0;
}

```

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. Unlike the tabulated version, all entries of the lookup table are not necessarily filled in memoized version. For example, memoized solution of **LCS problem** doesn't necessarily fill all entries.

To see the optimization achieved by memoized and tabulated versions over the basic recursive version, see the time taken by following runs for 40th Fibonacci number.

Simple recursive program

Memoized version

tabulated version

Also see method 2 of **Ugly Number post** for one more simple example where we have overlapping subproblems and we store the results of subproblems.

We will be covering Optimal Substructure Property and some more example problems in future posts on Dynamic Programming.

Try following questions as an exercise of this post.

- 1) Write a memoized version for LCS problem. Note that the tabular version is given in the CLRS book.
- 2) How would you choose between Memoization and Tabulation?

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

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

35. Dynamic Programming | Set 2 (Optimal Substructure Property)

As we discussed in **Set 1**, following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming.

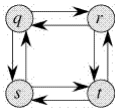
- 1) Overlapping Subproblems
- 2) Optimal Substructure

We have already discussed Overlapping Subproblem property in the **Set 1**. Let us discuss Optimal Substructure property here.

2) 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 given in the **CLRS book**. There are two longest paths from q to t : $q \rightarrow r \rightarrow t$ and $q \rightarrow s \rightarrow t$. Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path $q \rightarrow r \rightarrow t$ 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 $q \rightarrow s \rightarrow t \rightarrow r$.



We will be covering some example problems in future posts on Dynamic Programming.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

http://en.wikipedia.org/wiki/Optimal_substructure
CLRS book

36. Dynamic Programming | Set 3 (Longest Increasing Subsequence)

We have discussed Overlapping Subproblems and Optimal Substructure properties in **Set 1** and **Set 2** respectively.

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, length of LIS for $\{ 10, 22, 9, 33, 21, 50, 41, 60, 80 \}$ is 6 and LIS is $\{10, 22, 33, 50, 60, 80\}$.

Optimal Substructure:

Let $arr[0..n-1]$ be the input array and $L(i)$ be the length of the LIS till index i such that $arr[i]$ is part of LIS and $arr[i]$ is the last element in LIS, then $L(i)$ can be recursively written as.

$L(i) = \{ 1 + \text{Max} (L(j)) \}$ where $j < i$ and $\text{arr}[j] < \text{arr}[i]$ and if there is no such j then $L(i) = 1$

To get LIS of a given array, we need to return $\text{max}(L(i))$ where $0 < i < n$

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

Overlapping Subproblems:

Following is simple recursive implementation of the LIS problem. The implementation simply follows the recursive structure mentioned above. The value of lis ending with every element is returned using `max_ending_here`. The overall lis is returned using pointer to a variable `max`.

```

/* A Naive recursive implementation of LIS problem */
#include<stdio.h>
#include<stdlib.h>

/* To make use of recursive calls, this function must return two things:
1) Length of LIS ending with element arr[n-1]. We use max_ending_here
   for this purpose
2) Overall maximum as the LIS may end with an element before arr[n-1].
   max_ref is used for this purpose.
The value of LIS of full array of size n is stored in *max_ref which is
*/
int _lis( int arr[], int n, int *max_ref)
{
    /* Base case */
    if(n == 1)
        return 1;

    int res, max_ending_here = 1; // length of LIS ending with arr[n-1]

    /* Recursively get all LIS ending with arr[0], arr[1] ... arr[n-2].
       arr[i-1] is smaller than arr[n-1], and max ending with arr[n-1]
       to be updated, then update it */
    for(int i = 1; i < n; i++)
    {
        res = _lis(arr, i, max_ref);
        if (arr[i-1] < arr[n-1] && res + 1 > max_ending_here)
            max_ending_here = res + 1;
    }

    // Compare max_ending_here with the overall max. And update the
    // overall max if needed
    if (*max_ref < max_ending_here)
        *max_ref = max_ending_here;

    // Return length of LIS ending with arr[n-1]
    return max_ending_here;
}

// The wrapper function for _lis()
int lis(int arr[], int n)
{
    // The max variable holds the result
    int max = 1;

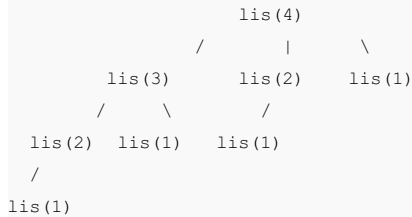
    // The function _lis() stores its result in max
    _lis( arr, n, &max );

    // returns max
    return max;
}

/* Driver program to test above function */
int main()
{
    int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Length of LIS is %d\n", lis( arr, n ));
    getchar();
    return 0;
}

```

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



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.


```

/* Dynamic Programming implementation of LIS problem */
#include<stdio.h>
#include<stdlib.h>

/* lis() returns the length of the longest increasing subsequence in
   arr[] of size n */
int lis( int arr[], int n )
{
    int *lis, i, j, max = 0;
    lis = (int*) malloc ( sizeof( int ) * n );

    /* Initialize LIS values for all indexes */
    for ( i = 0; i < n; i++ )
        lis[i] = 1;

    /* Compute optimized LIS values in bottom up manner */
    for ( i = 1; i < n; i++ )
        for ( j = 0; j < i; j++ )
            if ( arr[i] > arr[j] && lis[i] < lis[j] + 1 )
                lis[i] = lis[j] + 1;

    /* Pick maximum of all LIS values */
    for ( i = 0; i < n; i++ )
        if ( max < lis[i] )
            max = lis[i];

    /* Free memory to avoid memory leak */
    free( lis );

    return max;
}

/* Driver program to test above function */
int main()
{
    int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Length of LIS is %d\n", lis( arr, n ) );

    getchar();
    return 0;
}

```

Note that the time complexity of the above Dynamic Programmig (DP) solution is $O(n^2)$ and there is a $O(n\log n)$ solution for the LIS problem (see [this](#)). We have not discussed the $n\log n$ solution here as the purpose of this post is to explain Dynamic Programmig with a simple example.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

37. Dynamic Programming | Set 4 (Longest Common Subsequence)

We have discussed Overlapping Subproblems and Optimal Substructure properties in [Set 1](#) and [Set 2](#) respectively. We also discussed one example problem in [Set 3](#). Let us discuss Longest Common Subsequence (LCS) problem as one more example problem that can be solved using Dynamic Programming.

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.

1) 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

$$L(X[0..m-1], Y[0..n-1]) = \text{MAX} (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]))$$

Examples:

1) Consider the input strings “AGGTAB” and “GXTXAYB”. Last characters match for the strings. So length of LCS can be written as:

$$L(\text{“AGGTAB”}, \text{“GXTXAYB”}) = 1 + L(\text{“AGGTA”}, \text{“GXTXAY”})$$

2) Consider the input strings “ABCDGH” and “AEDFHR”. Last characters do not match for the strings. So length of LCS can be written as:

$$L(\text{“ABCDGH”}, \text{“AEDFHR”}) = \text{MAX} (L(\text{“ABCDG”}, \text{“AEDFHR”}), L(\text{“ABCDGH”}, \text{“AEDFH”}))$$

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

2) Overlapping Subproblems:

Following is simple recursive implementation of the LCS problem. The implementation

simply follows the recursive structure mentioned above.

```

/* A Naive recursive implementation of LCS problem */
#include<stdio.h>
#include<stdlib.h>

int max(int a, int b);

/* 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));
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Driver program to test above function */
int main()
{
    char X[] = "AGGTAB";
    char Y[] = "GTXAYB";

    int m = strlen(X);
    int n = strlen(Y);

    printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );

    getchar();
    return 0;
}

```

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.

```

/* Dynamic Programming implementation of LCS problem */
#include<stdio.h>
#include<stdlib.h>

int max(int a, int b);

/* 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];
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Driver program to test above function */
int main()
{
    char X[] = "AGGTAB";
    char Y[] = "GTXAYB";

    int m = strlen(X);
    int n = strlen(Y);

    printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );

    getchar();
    return 0;
}

```

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

The above algorithm/code returns only length of LCS. Please see the following post for printing the LCS.

[Printing Longest Common Subsequence](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

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

38. What does 'Space Complexity' mean?

Space Complexity:

The term Space Complexity is misused for Auxiliary Space at many places. Following are the correct definitions of Auxiliary Space and Space Complexity.

Auxiliary Space is the extra space or temporary space used by an algorithm.

Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

For example, if we want to compare standard sorting algorithms on the basis of space, then Auxiliary Space would be a better criteria than Space Complexity. Merge Sort uses $O(n)$ auxiliary space, Insertion sort and Heap Sort use $O(1)$ auxiliary space. Space complexity of all these sorting algorithms is $O(n)$ though.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

39. A Time Complexity Question

What is the time complexity of following function fun()? Assume that $\log(x)$ returns log value in base 2.

```
void fun()
{
    int i, j;
    for (i=1; i<=n; i++)
        for (j=1; j<=log(i); j++)
            printf("GeeksforGeeks");
}
```

Time Complexity of the above function can be written as

$\theta(\log 1) + \theta(\log 2) + \theta(\log 3) + \dots + \theta(\log n)$ which is $\theta(\log n!)$

Order of growth of $\log n!$ and $n \log n$ is same for large values of n , i.e., $\theta(\log n!) = \theta(n \log n)$.

So time complexity of `fun()` is $\theta(n \log n)$.

The expression $\theta(\log n!) = \theta(n \log n)$ can be easily derived from following **Stirling's approximation (or Stirling's formula)**.

$$\log n! = n \log n - n + O(\log(n))$$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

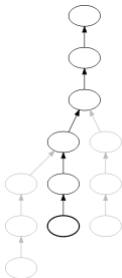
Sources:

http://en.wikipedia.org/wiki/Stirling%27s_approximation

40. Spaghetti Stack

Spaghetti stack

A spaghetti stack is an N-ary tree data structure in which child nodes have pointers to the parent nodes (but not vice-versa)



Spaghetti stack structure is used in situations when records are dynamically pushed and popped onto a stack as execution progresses, but references to the popped records remain in use. Following are some applications of Spaghetti Stack.

Compilers for languages such as C create a spaghetti stack as it opens and closes symbol tables representing block scopes. When a new block scope is opened, a symbol table is pushed onto a stack. When the closing curly brace is encountered, the scope is closed and the symbol table is popped. But that symbol table is remembered, rather than destroyed. And of course it remembers its higher level “parent” symbol table and so on.

Spaghetti Stacks are also used to implement **Disjoint-set data structure**.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Sources:

http://en.wikipedia.org/wiki/Spaghetti_stack

41. Dynamic Programming | Set 5 (Edit Distance)

Continuing further on dynamic programming series, *edit distance* is an interesting algorithm.

Problem: Given two strings of size m , n and set of operations replace (R), insert (I) and delete (D) all at equal cost. Find minimum number of edits (operations) required to convert one string into another.

Identifying Recursive Methods:

What will be sub-problem in this case? Consider finding edit distance of part of the strings, say small prefix. Let us denote them as $[1 \dots i]$ and $[1 \dots j]$ for some $1 < i < m$ and $1 < j < n$. Clearly it is solving smaller instance of final problem, denote it as $E(i, j)$. Our goal is finding $E(m, n)$ and minimizing the cost.

In the prefix, we can right align the strings in three ways $(i, -)$, $(-, j)$ and (i, j) . The hyphen symbol $(-)$ representing no character. An example can make it more clear.

Given strings SUNDAY and SATURDAY. We want to convert SUNDAY into SATURDAY with minimum edits. Let us pick $i = 2$ and $j = 4$ i.e. prefix strings are SUN and SATU respectively (assume the strings indices start at 1). The right most characters can be aligned in three different ways.

Case 1: Align characters U and U. They are equal, no edit is required. We still left with the problem of $i = 1$ and $j = 3$, $E(i-1, j-1)$.

Case 2: Align right character from first string and no character from second string. We need a deletion (D) here. We still left with problem of $i = 1$ and $j = 4$, $E(i-1, j)$.

Case 3: Align right character from second string and no character from first string. We need an insertion (I) here. We still left with problem of $i = 2$ and $j = 3$, $E(i, j-1)$.

Combining all the subproblems minimum cost of aligning prefix strings ending at i and j given by

$E(i, j) = \min([E(i-1, j) + D], [E(i, j-1) + I], [E(i-1, j-1) + R \text{ if } i, j \text{ characters are not same}])$

We still not yet done. What will be base case(s)?

When both of the strings are of size 0, the cost is 0. When only one of the string is zero, we need edit operations as that of non-zero length string. Mathematically,

$$E(0, 0) = 0, E(i, 0) = i, E(0, j) = j$$

Now it is easy to complete recursive method. Go through the code for recursive algorithm (`edit_distance_recursive`).

Dynamic Programming Method:

We can calculate the complexity of recursive expression fairly easily.

$$T(m, n) = T(m-1, n-1) + T(m, n-1) + T(m-1, n) + C$$

The complexity of $T(m, n)$ can be calculated by successive substitution method or solving homogeneous equation of two variables. It will result in an exponential complexity algorithm. It is evident from the recursion tree that it will be solving subproblems again and again. Few strings result in many overlapping subproblems (try the below program with strings *exponential* and *polynomial* and note the delay in recursive method).

We can tabulate the repeating subproblems and look them up when required next time (bottom up). A two dimensional array formed by the strings can keep track of the minimum cost till the current character comparison. The visualization code will help in understanding the construction of matrix.

The time complexity of dynamic programming method is $O(mn)$ as we need to construct the table fully. The space complexity is also $O(mn)$. If we need only the cost of edit, we just need $O(\min(m, n))$ space as it is required only to keep track of the current row and previous row.

Usually the costs D, I and R are not same. In such case the problem can be represented as an acyclic directed graph (DAG) with weights on each edge, and finding shortest path gives edit distance.

Applications:

There are many practical applications of edit distance algorithm, refer [Lucene API](#) for sample. Another example, display all the words in a dictionary that are near proximity to a given word\incorrectly spelled word.

```
// Dynamic Programming implementation of edit distance
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

// Change these strings to test the program
#define STRING_X "SUNDAY"
#define STRING_Y "SATURDAY"

#define SENTINEL (-1)
#define EDIT_COST (1)
```



```

inline
int min(int a, int b) {
    return a < b ? a : b;
}

// Returns Minimum among a, b, c
int Minimum(int a, int b, int c)
{
    return min(min(a, b), c);
}

// Strings of size m and n are passed.
// Construct the Table for X[0...m, m+1], Y[0...n, n+1]
int EditDistanceDP(char X[], char Y[])
{
    // Cost of alignment
    int cost = 0;
    int leftCell, topCell, cornerCell;

    int m = strlen(X)+1;
    int n = strlen(Y)+1;

    // T[m][n]
    int *T = (int *)malloc(m * n * sizeof(int));

    // Initialize table
    for(int i = 0; i < m; i++)
        for(int j = 0; j < n; j++)
            *(T + i * n + j) = SENTINEL;

    // Set up base cases
    // T[i][0] = i
    for(int i = 0; i < m; i++)
        *(T + i * n) = i;

    // T[0][j] = j
    for(int j = 0; j < n; j++)
        *(T + j) = j;

    // Build the T in top-down fashion
    for(int i = 1; i < m; i++)
    {
        for(int j = 1; j < n; j++)
        {
            // T[i][j-1]
            leftCell = *(T + i*n + j-1);
            leftCell += EDIT_COST; // deletion

            // T[i-1][j]
            topCell = *(T + (i-1)*n + j);
            topCell += EDIT_COST; // insertion

            // Top-left (corner) cell
            // T[i-1][j-1]
            cornerCell = *(T + (i-1)*n + (j-1) );

            // edit[(i-1), (j-1)] = 0 if X[i] == Y[j], 1 otherwise
            cornerCell += (X[i-1] != Y[j-1]); // may be replace

            // Minimum cost of current cell
            // Fill in the next cell T[i][j]
            // T[i][j] = min(leftCell, topCell, cornerCell) + cost;

```

```

        *(l + (1)*n + (j)) = Minimum(leftCell, topCell, cornerCell)
    }
}

// Cost is in the cell T[m][n]
cost = *(T + m*n - 1);
free(T);
return cost;
}

// Recursive implementation
int EditDistanceRecursion( char *X, char *Y, int m, int n )
{
    // Base cases
    if( m == 0 && n == 0 )
        return 0;

    if( m == 0 )
        return n;

    if( n == 0 )
        return m;

    // Recurse
    int left = EditDistanceRecursion(X, Y, m-1, n) + 1;
    int right = EditDistanceRecursion(X, Y, m, n-1) + 1;
    int corner = EditDistanceRecursion(X, Y, m-1, n-1) + (X[m-1] != Y[n-1]);

    return Minimum(left, right, corner);
}

int main()
{
    char X[] = STRING_X; // vertical
    char Y[] = STRING_Y; // horizontal

    printf("Minimum edits required to convert %s into %s is %d\n",
        X, Y, EditDistanceDP(X, Y) );
    printf("Minimum edits required to convert %s into %s is %d by recursion\n",
        X, Y, EditDistanceRecursion(X, Y, strlen(X), strlen(Y)));

    return 0;
}

```

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

42. Backtracking | Set 1 (The Knight's tour problem)

Backtracking is an algorithmic paradigm that tries different solutions until finds a solution that “works”. Problems which are typically solved using backtracking technique have following property in common. These problems can only be solved by trying every possible configuration and each configuration is tried only once. A Naive solution for

these problems is to try all configurations and output a configuration that follows given problem constraints. Backtracking works in incremental way and is an optimization over the Naive solution where all possible configurations are generated and tried.

For example, consider the following **Knight's Tour** problem.

The knight is placed on the first block of an empty board and, moving according to the rules of chess, must visit each square exactly once.

Let us first discuss the Naive algorithm for this problem and then the Backtracking algorithm.

Naive Algorithm for Knight's tour

The Naive Algorithm is to generate all tours one by one and check if the generated tour satisfies the constraints.

```
while there are untried tours
{
    generate the next tour
    if this tour covers all squares
    {
        print this path;
    }
}
```

Backtracking works in an incremental way to attack problems. Typically, we start from an empty solution vector and one by one add items (Meaning of item varies from problem to problem. In context of Knight's tour problem, an item is a Knight's move). When we add an item, we check if adding the current item violates the problem constraint, if it does then we remove the item and try other alternatives. If none of the alternatives work out then we go to previous stage and remove the item added in the previous stage. If we reach the initial stage back then we say that no solution exists. If adding an item doesn't violate constraints then we recursively add items one by one. If the solution vector becomes complete then we print the solution.

Backtracking Algorithm for Knight's tour

Following is the Backtracking algorithm for Knight's tour problem.

```
If all squares are visited
    print the solution
Else
    a) Add one of the next moves to solution vector and recursively
       check if this move leads to a solution. (A Knight can make maximum
       eight moves. We choose one of the 8 moves in this step).
    b) If the move chosen in the above step doesn't lead to a solution
       then remove this move from the solution vector and try other
       alternative moves.
    c) If none of the alternatives work then return false (Returning false
```

```
will remove the previously added item in recursion and if false is  
returned by the initial call of recursion then "no solution exists" )
```

Following is C implementation for Knight's tour problem. It prints one of the possible solutions in 2D matrix form. Basically, the output is a 2D 8*8 matrix with numbers from 0 to 63 and these numbers show steps made by Knight.

```
#include<stdio.h>
#define N 8

int solveKTUtil(int x, int y, int movei, int sol[N][N], int xMove[],
               int yMove[]);

/* A utility function to check if i,j are valid indexes for N*N chessboard
int isSafe(int x, int y, int sol[N][N])
{
    if ( x >= 0 && x < N && y >= 0 && y < N && sol[x][y] == -1)
        return 1;
    return 0;
}

/* A utility function to print solution matrix sol[N][N] */
void printSolution(int sol[N][N])
{
    for (int x = 0; x < N; x++)
    {
        for (int y = 0; y < N; y++)
            printf(" %2d ", sol[x][y]);
        printf("\n");
    }
}

/* This function solves the Knight Tour problem using Backtracking. It
function mainly uses solveKTUtil() to solve the problem. It returns false
no complete tour is possible, otherwise return true and prints the tour
Please note that there may be more than one solutions, this function
prints one of the feasible solutions. */
bool solveKT()
{
    int sol[N][N];

    /* Initialization of solution matrix */
    for (int x = 0; x < N; x++)
        for (int y = 0; y < N; y++)
            sol[x][y] = -1;

    /* xMove[] and yMove[] define next move of Knight.
    xMove[] is for next value of x coordinate
    yMove[] is for next value of y coordinate */
    int xMove[8] = { 2, 1, -1, -2, -2, -1, 1, 2 };
    int yMove[8] = { 1, 2, 2, 1, -1, -2, -2, -1 };

    // Since the Knight is initially at the first block
    sol[0][0] = 0;

    /* Start from 0,0 and explore all tours using solveKTUtil() */
    if(solveKTUtil(0, 0, 1, sol, xMove, yMove) == false)
    {
        printf("Solution does not exist");
        return false;
    }
}
```

```

        else
            printSolution(sol);

        return true;
    }

    /* A recursive utility function to solve Knight Tour problem */
    int solveKTUtil(int x, int y, int movei, int sol[N][N], int xMove[N],
                    int yMove[N])
    {
        int k, next_x, next_y;
        if (movei == N*N)
            return true;

        /* Try all next moves from the current coordinate x, y */
        for (k = 0; k < 8; k++)
        {
            next_x = x + xMove[k];
            next_y = y + yMove[k];
            if (isSafe(next_x, next_y, sol))
            {
                sol[next_x][next_y] = movei;
                if (solveKTUtil(next_x, next_y, movei+1, sol, xMove, yMove) ==
                    return true;
                else
                    sol[next_x][next_y] = -1; // backtracking
            }
        }

        return false;
    }

    /* Driver program to test above functions */
    int main()
    {
        solveKT();
        getchar();
        return 0;
    }

```

Output:

```

0  59  38  33  30  17   8  63
37 34  31  60   9  62  29  16
58  1  36  39  32  27  18   7
35 48  41  26  61  10  15  28
42 57   2  49  40  23   6  19
47 50  45  54  25  20  11  14
56 43  52   3  22  13  24   5
51 46  55  44  53   4  21  12

```

Note that Backtracking is not the best solution for the Knight's tour problem. See [this](#) for other better solutions. The purpose of this post is to explain Backtracking with an example.

References:

<http://see.stanford.edu/materials/icspacs106b/H19-RecBacktrackExamples.pdf>
<http://www.cis.upenn.edu/~matuszek/cit594-2009/Lectures/35-backtracking.ppt>
<http://mathworld.wolfram.com/KnightsTour.html>
http://en.wikipedia.org/wiki/Knight%27s_tour

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

43. Backtracking | Set 2 (Rat in a Maze)

We have discussed Backtracking and Knight's tour problem in [Set 1](#). Let us discuss Rat in a [Maze](#) as another example problem that can be solved using Backtracking.

A Maze is given as $N \times N$ binary matrix of blocks where source block is the upper left most block i.e., `maze[0][0]` and destination block is lower rightmost block i.e., `maze[N-1][N-1]`. A rat starts from source and has to reach destination. The rat can move only in two directions: forward and down.

In the maze matrix, 0 means the block is dead end and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze problem. For example, a more complex version can be that the rat can move in 4 directions and a more complex version can be with limited number of moves.

Following is an example maze.

Gray blocks are dead ends (value = 0).

Source			
			Dest.

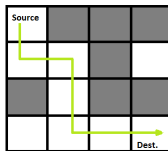
Following is binary matrix representation of the above maze.

```

{1, 0, 0, 0}
{1, 1, 0, 1}
{0, 1, 0, 0}
{1, 1, 1, 1}

```

Following is maze with highlighted solution path.



Following is the solution matrix (output of program) for the above input matrix.

```
{1, 0, 0, 0}
```

```
{1, 1, 0, 0}
```

```
{0, 1, 0, 0}
```

```
{0, 1, 1, 1}
```

All enteries in solution path are marked as 1.

Naive Algorithm

The Naive Algorithm is to generate all paths from source to destination and one by one check if the generated path satisfies the constraints.

```
while there are untried paths
{
    generate the next path
    if this path has all blocks as 1
    {
        print this path;
    }
}
```

Backtrackng Algorithm

```
If destination is reached
    print the solution matrix
Else
    a) Mark current cell in solution matrix as 1.
    b) Move forward in horizontal direction and recursively check if this
        move leads to a solution.
    c) If the move chosen in the above step doesn't lead to a solution
        then move down and check if this move leads to a solution.
    d) If none of the above solutions work then unmark this cell as 0
        (BACKTRACK) and return false.
```

Implementation of Backtracking solution

```
#include<stdio.h>

// Maze size
#define N 4

bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N]);

/* A utility function to print solution matrix sol[N][N] */
```

```

void printSolution(int sol[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            printf(" %d ", sol[i][j]);
        printf("\n");
    }
}

/* A utility function to check if x,y is valid index for N*N maze */
bool isSafe(int maze[N][N], int x, int y)
{
    // if (x,y outside maze) return false
    if(x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1)
        return true;

    return false;
}

/* This function solves the Maze problem using Backtracking. It mainly
solveMazeUtil() to solve the problem. It returns false if no path is
possible, otherwise return true and prints the path in the form of 1s. Please note
that there may be more than one solutions, this function prints one of the
solutions.*/
bool solveMaze(int maze[N][N])
{
    int sol[N][N] = { {0, 0, 0, 0},
                      {0, 0, 0, 0},
                      {0, 0, 0, 0},
                      {0, 0, 0, 0}
                    };

    if(solveMazeUtil(maze, 0, 0, sol) == false)
    {
        printf("Solution doesn't exist");
        return false;
    }

    printSolution(sol);
    return true;
}

/* A recursive utility function to solve Maze problem */
bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N])
{
    // if (x,y is goal) return true
    if(x == N-1 && y == N-1)
    {
        sol[x][y] = 1;
        return true;
    }

    // Check if maze[x][y] is valid
    if(isSafe(maze, x, y) == true)
    {
        // mark x,y as part of solution path
        sol[x][y] = 1;

        /* Move forward in x direction */
        if (solveMazeUtil(maze, x+1, y, sol) == true)
            return true;
    }
}

```



```

        /* If moving in x direction doesn't give solution then
           Move down in y direction */
        if (solveMazeUtil(maze, x, y+1, sol) == true)
            return true;

        /* If none of the above movements work then BACKTRACK:
           unmark x,y as part of solution path */
        sol[x][y] = 0;
        return false;
    }

    return false;
}

// driver program to test above function
int main()
{
    int maze[N][N] = { {1, 0, 0, 0},
                        {1, 1, 0, 1},
                        {0, 1, 0, 0},
                        {1, 1, 1, 1}
    };

    solveMaze(maze);
    getchar();
    return 0;
}

```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

44. Backtracking | Set 3 (N Queen Problem)

We have discussed Knight's tour and Rat in a Maze problems in [Set 1](#) and [Set 2](#) respectively. Let us discuss N Queen as another example problem that can be solved using Backtracking.

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.

	Q		
			Q
Q			
		Q	

The expected output is a binary matrix which has 1s for the blocks where queens are

placed. For example following is the output matrix for above 4 queen solution.

```
{ 0, 1, 0, 0}
{ 0, 0, 0, 1}
{ 1, 0, 0, 0}
{ 0, 0, 1, 0}
```

Naive Algorithm

Generate all possible configurations of queens on board and print a configuration that satisfies the given constraints.

```
while there are untried configurations
{
    generate the next configuration
    if queens don't attack in this configuration then
    {
        print this configuration;
    }
}
```

Backtracking Algorithm

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.

```
1) Start in the leftmost column
2) If all queens are placed
    return true
3) Try all rows in the current column. Do following for every tried row.
    a) If the queen can be placed safely in this row then mark this [row,
        column] as part of the solution and recursively check if placing
        queen here leads to a solution.
    b) If placing queen in [row, column] leads to a solution then return
        true.
    c) If placing queen doesn't lead to a solution then unmark this [row,
        column] (Backtrack) and go to step (a) to try other rows.
3) If all rows have been tried and nothing worked, return false to trigger
    backtracking.
```

Implementation of Backtracking solution

```
#define N 4
#include<stdio.h>

/* A utility function to print solution */
void printSolution(int board[N][N])
{
```

```

    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            printf(" %d ", board[i][j]);
        printf("\n");
    }
}

/* A utility function to check if a queen can be placed on board[row][col]
Note that this function is called when "col" queens are already placed
in columns from 0 to col -1. So we need to check only left side for
attacking queens */
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;

    /* Check this row on left side */
    for (i = 0; i < col; i++)
    {
        if (board[row][i])
            return false;
    }

    /* Check upper diagonal on left side */
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
    {
        if (board[i][j])
            return false;
    }

    /* Check lower diagonal on left side */
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
    {
        if (board[i][j])
            return false;
    }

    return true;
}

/* A recursive utility function to solve N Queen problem */
bool solveNQUtil(int board[N][N], int col)
{
    /* base case: If all queens are placed then return true */
    if (col >= N)
        return true;

    /* Consider this column and try placing this queen in all rows
    one by one */
    for (int i = 0; i < N; i++)
    {
        /* Check if queen can be placed on board[i][col] */
        if ( isSafe(board, i, col) )
        {
            /* Place this queen in board[i][col] */
            board[i][col] = 1;

            /* recur to place rest of the queens */
            if ( solveNQUtil(board, col + 1) == true )
                return true;

            /* If placing queen in board[i][col] doesn't lead to a solution
            then remove it from the board and try other row */
            board[i][col] = 0;
        }
    }

    /* If no row can be placed in this column then return false
    to trigger the backtracking */
    return false;
}

```

```

        then remove queen from board[i][col] */
        board[i][col] = 0; // BACKTRACK
    }
}

/* If queen can not be place in any row in this column col
then return false */
return false;
}

/* This function solves the N Queen problem using Backtracking. It ma
solveNQUtil() to solve the problem. It returns false if queens cannot b
otherwise return true and prints placement of queens in the form of 1s
note that there may be more than one solutions, this function prints on
feasible solutions.*/
bool solveNQ()
{
    int board[N][N] = { {0, 0, 0, 0},
                        {0, 0, 0, 0},
                        {0, 0, 0, 0},
                        {0, 0, 0, 0}
    };

    if ( solveNQUtil(board, 0) == false )
    {
        printf("Solution does not exist");
        return false;
    }

    printSolution(board);
    return true;
}

// driver program to test above function
int main()
{
    solveNQ();

    getchar();
    return 0;
}

```

Sources:

<http://see.stanford.edu/materials/icspaces106b/H19-RecBacktrackExamples.pdf>

http://en.literateprograms.org/Eight_queens_puzzle_%28C%29

http://en.wikipedia.org/wiki/Eight_queens_puzzle

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

45. Check if a number is multiple of 5 without using / and % operators

Given a positive number n, write a function isMultipleof5(int n) that returns true if n is multiple of 5, otherwise false. **You are not allowed to use % and / operators.**

Method 1 (Repeatedly subtract 5 from n)

Run a loop and subtract 5 from n in the loop while n is greater than 0. After the loop terminates, check whether n is 0. If n becomes 0 then n is multiple of 5, otherwise not.

```
#include<stdio.h>

/* assumes that n is a positive integer */
bool isMultipleof5 (int n)
{
    while ( n > 0 )
        n = n - 5;

    if ( n == 0 )
        return true;

    return false;
}

/* Driver program to test above function */
int main()
{
    int n = 19;
    if ( isMultipleof5(n) == true )
        printf("%d is multiple of 5\n", n);
    else
        printf("%d is not a multiple of 5\n", n);

    return 0;
}
```

Method 2 (Convert to string and check the last character)

Convert n to a string and check the last character of the string. If the last character is '5' or '0' then n is multiple of 5, otherwise not.

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>

/* Assuming that integer takes 4 bytes, there can
   be maximum 10 digits in a integer */
# define MAX 11

bool isMultipleof5(int n)
{
    char str[MAX];
    itoa(n, str, MAX);

    int len = strlen(str);

    /* Check the last character of string */
    if ( str[len-1] == '5' || str[len-1] == '0' )
        return true;

    return false;
}

/* Driver program to test above function */
int main()
{
    int n = 19;
    if ( isMultipleof5(n) == true )
        printf("%d is multiple of 5\n", n);
    else
        printf("%d is not a multiple of 5\n", n);

    return 0;
}

```

Thanks to [Baban_Rathore](#) for suggesting this method.

Method 3 (Set last digit as 0 and use floating point trick)

A number n can be a multiple of 5 in two cases. When last digit of n is 5 or 10. If last bit in binary equivalent of n is set (which can be the case when last digit is 5) then we multiply by 2 using $n \ll 1$ to make sure that if the number is multiple of 5 then we have the last digit as 0. Once we do that, our work is to just check if the last digit is 0 or not, which we can do using float and integer comparison trick.

```

#include<stdio.h>

bool isMultipleof5(int n)
{
    /* If n is a multiple of 5 then we make sure that last
       digit of n is 0 */
    if ( (n&1) == 1 )
        n <<= 1;

    float x = n;
    x = ( (int)(x*0.1) ) *10;

    /* If last digit of n is 0 then n will be equal to (int)x */
    if ( (int)x == n )
        return true;

    return false;
}

/* Driver program to test above function */
int main()
{
    int i = 19;
    if ( isMultipleof5(i) == true )
        printf("%d is multiple of 5\n", i);
    else
        printf("%d is not a multiple of 5\n", i);

    getchar();
    return 0;
}

```

Thanks to [darkprince](#) for suggesting this method.

Source: See [this](#) forum thread.

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

46. Design and Implement Special Stack Data Structure | Added Space Optimized Version

Question: Design a Data Structure SpecialStack that supports all the stack operations like push(), pop(), isEmpty(), isFull() and an additional operation getMin() which should return minimum element from the SpecialStack. All these operations of SpecialStack must be O(1). To implement SpecialStack, you should only use standard Stack data structure and no other data structure like arrays, list, .. etc.

Example:

Consider the following SpecialStack

```
16 --> TOP
15
29
19
18
```

When `getMin()` is called it should return 15, which is the minimum element in the current stack.

If we do `pop` two times on stack, the stack becomes

```
29 --> TOP
19
18
```

When `getMin()` is called, it should return 18 which is the minimum in the current stack.

Solution: Use two stacks: one to store actual stack elements and other as an auxiliary stack to store minimum values. The idea is to do `push()` and `pop()` operations in such a way that the top of auxiliary stack is always the minimum. Let us see how `push()` and `pop()` operations work.

Push(int x) // inserts an element x to Special Stack

- 1) push x to the first stack (the stack with actual elements)
- 2) compare x with the top element of the second stack (the auxiliary stack). Let the top element be y.

.....a) If x is smaller than y then push x to the auxiliary stack.

.....b) If x is greater than y then push y to the auxiliary stack.

int Pop() // removes an element from Special Stack and return the removed element

- 1) pop the top element from the auxiliary stack.
- 2) pop the top element from the actual stack and return it.

The step 1 is necessary to make sure that the auxiliary stack is also updated for future operations.

int getMin() // returns the minimum element from Special Stack

- 1) Return the top element of auxiliary stack.

We can see that **all above operations are O(1)**.

Let us see an example. Let us assume that both stacks are initially empty and 18, 19, 29, 15 and 16 are inserted to the SpecialStack.

When we insert 18, both stacks change to following.

Actual Stack


```
18 <--- top
Auxiliary Stack
18 <---- top
```

When 19 is inserted, both stacks change to following.

```
Actual Stack
19 <--- top
18
Auxiliary Stack
18 <---- top
18
```

When 29 is inserted, both stacks change to following.

```
Actual Stack
29 <--- top
19
18
Auxiliary Stack
18 <---- top
18
18
```

When 15 is inserted, both stacks change to following.

```
Actual Stack
15 <--- top
29
19
18
Auxiliary Stack
15 <---- top
18
18
18
```

When 16 is inserted, both stacks change to following.

```
Actual Stack
16 <--- top
15
29
19
18
Auxiliary Stack
15 <---- top
15
18
```

18

18

Following is C++ implementation for SpecialStack class. In the below implementation, SpecialStack inherits from Stack and has one Stack object *min* which work as auxiliary stack.

```
#include<iostream>
#include<stdlib.h>

using namespace std;

/* A simple stack class with basic stack functionalities */
class Stack
{
private:
    static const int max = 100;
    int arr[max];
    int top;
public:
    Stack() { top = -1; }
    bool isEmpty();
    bool isFull();
    int pop();
    void push(int x);
};

/* Stack's member method to check if the stack is iempty */
bool Stack::isEmpty()
{
    if(top == -1)
        return true;
    return false;
}

/* Stack's member method to check if the stack is full */
bool Stack::isFull()
{
    if(top == max - 1)
        return true;
    return false;
}

/* Stack's member method to remove an element from it */
int Stack::pop()
{
    if(isEmpty())
    {
        cout<<"Stack Underflow";
        abort();
    }
    int x = arr[top];
    top--;
    return x;
}

/* Stack's member method to insert an element to it */
void Stack::push(int x)
{
    if(isFull())
    {
        abort();
    }
    arr[top] = x;
    top++;
}
```

```

    {
        cout<<"Stack Overflow";
        abort();
    }
    top++;
    arr[top] = x;
}

```

```

/* A class that supports all the stack operations and one additional
operation getMin() that returns the minimum element from stack at
any time. This class inherits from the stack class and uses an
auxiliary stack that holds minimum elements */

```

```

class SpecialStack: public Stack

```

```

{
    Stack min;
public:
    int pop();
    void push(int x);
    int getMin();
};

```

```

/* SpecialStack's member method to insert an element to it. This method
makes sure that the min stack is also updated with appropriate minimum
values */

```

```

void SpecialStack::push(int x)

```

```

{
    if(isEmpty()==true)
    {
        Stack::push(x);
        min.push(x);
    }
    else
    {
        Stack::push(x);
        int y = min.pop();
        min.push(y);
        if( x < y )
            min.push(x);
        else
            min.push(y);
    }
}

```

```

/* SpecialStack's member method to remove an element from it. This method
removes top element from min stack also. */

```

```

int SpecialStack::pop()

```

```

{
    int x = Stack::pop();
    min.pop();
    return x;
}

```

```

/* SpecialStack's member method to get minimum element from it. */

```

```

int SpecialStack::getMin()

```

```

{
    int x = min.pop();
    min.push(x);
    return x;
}

```

```

/* Driver program to test SpecialStack methods */

```

```

int main()
{

```

```

    SpecialStack s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout<<s.getMin()<<endl;
    s.push(5);
    cout<<s.getMin();
    return 0;
}

```

Output:

10

5

Space Optimized Version

The above approach can be optimized. We can limit the number of elements in auxiliary stack. We can push only when the incoming element of main stack is smaller than or equal to top of auxiliary stack. Similarly during pop, if the pop off element equal to top of auxiliary stack, remove the top element of auxiliary stack. Following is modified implementation of push() and pop().

```

/* SpecialStack's member method to insert an element to it. This method
makes sure that the min stack is also updated with appropriate mini
values */
void SpecialStack::push(int x)
{
    if(isEmpty()==true)
    {
        Stack::push(x);
        min.push(x);
    }
    else
    {
        Stack::push(x);
        int y = min.pop();
        min.push(y);

        /* push only when the incoming element of main stack is smaller
        than or equal to top of auxiliary stack */
        if( x <= y )
            min.push(x);
    }
}

/* SpecialStack's member method to remove an element from it. This method
removes top element from min stack also. */
int SpecialStack::pop()
{
    int x = Stack::pop();
    int y = min.pop();

    /* Push the popped element y back only if it is not equal to x */
    if ( y != x )
        min.push(x);

    return x;
}

```

Thanks to @Venki, @swarup and @Jing Huang for their inputs.

Please write comments if you find the above code incorrect, or find other ways to solve the same problem.

47. Backtracking | Set 4 (Subset Sum)

Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K . We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).

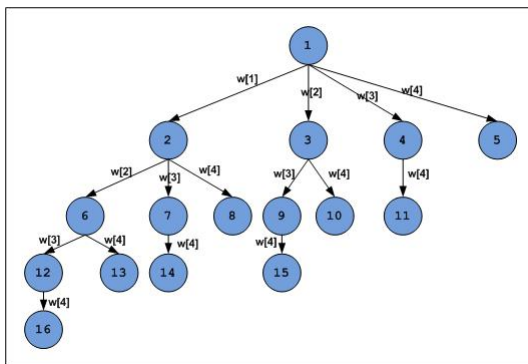
Exhaustive Search Algorithm for Subset Sum

One way to find subsets that sum to K is to consider all possible subsets. A **power set** contains all those subsets generated from a given set. The size of such a power set is 2^N .

Backtracking Algorithm for Subset Sum

Using exhaustive search we consider all subsets irrespective of whether they satisfy given constraints or not. Backtracking can be used to make a systematic consideration of the elements to be selected.

Assume given set of 4 elements, say $w[1] \dots w[4]$. Tree diagrams can be used to design backtracking algorithms. The following tree diagram depicts approach of generating variable sized tuple.



In the above tree, a node represents function call and a branch represents candidate element. The root node contains 4 children. In other words, root considers every element of the set as different branch. The next level sub-trees correspond to the subsets that includes the parent node. The branches at each level represent tuple

element to be considered. For example, if we are at level 1, `tuple_vector[1]` can take any value of four branches generated. If we are at level 2 of left most node, `tuple_vector[2]` can take any value of three branches generated, and so on...

For example the left most child of root generates all those subsets that include `w[1]`. Similarly the second child of root generates all those subsets that includes `w[2]` and excludes `w[1]`.

As we go down along depth of tree we add elements so far, and if the added sum is satisfying explicit constraints, we will continue to generate child nodes further. Whenever the constraints are not met, we stop further generation of sub-trees of that node, and backtrack to previous node to explore the nodes not yet explored. In many scenarios, it saves considerable amount of processing time.

The tree should trigger a clue to implement the backtracking algorithm (try yourself). It prints all those subsets whose sum add up to given number. We need to explore the nodes along the breadth and depth of the tree. Generating nodes along breadth is controlled by loop and nodes along the depth are generated using recursion (post order traversal). Pseudo code given below,

```
if(subset is satisfying the constraint)
    print the subset
    exclude the current element and consider next element
else
    generate the nodes of present level along breadth of tree and
    recur for next levels
```

Following is C implementation of subset sum using variable size tuple vector. Note that the following program explores all possibilities similar to exhaustive search. It is to demonstrate how backtracking can be used. See next code to verify, how we can optimize the backtracking solution.

```
#include <stdio.h>
#include <stdlib.h>

#define ARRAYSIZE(a) (sizeof(a))/(sizeof(a[0]))

static int total_nodes;
// prints subset found
void printSubset(int A[], int size)
{
    for(int i = 0; i < size; i++)
    {
        printf("%*d", 5, A[i]);
    }

    printf("\n");
}

// inputs
// s          - set vector
// t          - tuple vector
// s size     - set size
```

```

// t_size      - tuple size so far
// sum         - sum so far
// ite        - nodes count
// target_sum  - sum to be found
void subset_sum(int s[], int t[],
               int s_size, int t_size,
               int sum, int ite,
               int const target_sum)
{
    total_nodes++;
    if( target_sum == sum )
    {
        // We found subset
        printSubset(t, t_size);
        // Exclude previously added item and consider next candidate
        subset_sum(s, t, s_size, t_size-1, sum - s[ite], ite + 1, target_sum);
        return;
    }
    else
    {
        // generate nodes along the breadth
        for( int i = ite; i < s_size; i++ )
        {
            t[t_size] = s[i];
            // consider next level node (along depth)
            subset_sum(s, t, s_size, t_size + 1, sum + s[i], i + 1, target_sum);
        }
    }
}

// Wrapper to print subsets that sum to target_sum
// input is weights vector and target_sum
void generateSubsets(int s[], int size, int target_sum)
{
    int *tuple_vector = (int *)malloc(size * sizeof(int));

    subset_sum(s, tuple_vector, size, 0, 0, 0, target_sum);

    free(tuple_vector);
}

int main()
{
    int weights[] = {10, 7, 5, 18, 12, 20, 15};
    int size = ARRAYSIZE(weights);

    generateSubsets(weights, size, 35);
    printf("Nodes generated %d\n", total_nodes);
    return 0;
}

```

The power of backtracking appears when we combine explicit and implicit constraints, and we stop generating nodes when these checks fail. We can improve the above algorithm by strengthening the constraint checks and presorting the data. By sorting the initial array, we need not to consider rest of the array, once the sum so far is greater than target number. We can backtrack and check other possibilities.

Similarly, assume the array is presorted and we found one subset. We can generate next node excluding the present node only when inclusion of next node satisfies the

constraints. Given below is optimized implementation (it prunes the subtree if it is not satisfying constraints).

```
#include <stdio.h>
#include <stdlib.h>

#define ARRAYSIZE(a) (sizeof(a))/(sizeof(a[0]))

static int total_nodes;

// prints subset found
void printSubset(int A[], int size)
{
    for(int i = 0; i < size; i++)
    {
        printf("%d", 5, A[i]);
    }

    printf("\n");
}

// qsort compare function
int comparator(const void *pLhs, const void *pRhs)
{
    int *lhs = (int *)pLhs;
    int *rhs = (int *)pRhs;

    return *lhs > *rhs;
}

// inputs
// s          - set vector
// t          - tuple vector
// s_size     - set size
// t_size     - tuple size so far
// sum        - sum so far
// ite       - nodes count
// target_sum - sum to be found
void subset_sum(int s[], int t[],
               int s_size, int t_size,
               int sum, int ite,
               int const target_sum)
{
    total_nodes++;

    if( target_sum == sum )
    {
        // We found sum
        printSubset(t, t_size);

        // constraint check
        if( ite + 1 < s_size && sum - s[ite] + s[ite+1] <= target_sum )
        {
            // Exclude previous added item and consider next candidate
            subset_sum(s, t, s_size, t_size-1, sum - s[ite], ite + 1,
                      target_sum);
        }
        return;
    }
    else
    {
        // constraint check
        if( ite < s_size && sum + s[ite] <= target_sum )
    }
```



```

    if( ite < s_size && sum + s[ite] <= target_sum )
    {
        // generate nodes along the breadth
        for( int i = ite; i < s_size; i++ )
        {
            t[t_size] = s[i];

            if( sum + s[i] <= target_sum )
            {
                // consider next level node (along depth)
                subset_sum(s, t, s_size, t_size + 1, sum + s[i], i
            }
        }
    }
}

// Wrapper that prints subsets that sum to target_sum
void generateSubsets(int s[], int size, int target_sum)
{
    int *tuple_vector = (int *)malloc(size * sizeof(int));

    int total = 0;

    // sort the set
    qsort(s, size, sizeof(int), &comparator);

    for( int i = 0; i < size; i++ )
    {
        total += s[i];
    }

    if( s[0] <= target_sum && total >= target_sum )
    {
        subset_sum(s, tuple_vector, size, 0, 0, 0, target_sum);
    }

    free(tuple_vector);
}

int main()
{
    int weights[] = {15, 22, 14, 26, 32, 9, 16, 8};
    int target = 53;

    int size = ARRAYSIZE(weights);

    generateSubsets(weights, size, target);

    printf("Nodes generated %d\n", total_nodes);

    return 0;
}

```

As another approach, we can generate the tree in fixed size tuple analogs to binary pattern. We will kill the sub-trees when the constraints are not satisfied.

— — — **Venki**. Please write comments if you find anything incorrect, or you want to share

more information about the topic discussed above.

48. Practice Questions for Recursion | Set 6

Question 1

Consider the following recursive C function. Let *len* be the length of the string *s* and *num* be the number of characters printed on the screen, give the relation between *num* and *len* where *len* is always greater than 0.

```
void abc(char *s)
{
    if(s[0] == '\0')
        return;

    abc(s + 1);
    abc(s + 1);
    printf("%c", s[0]);
}
```

Following is the relation between *num* and *len*.

$$\text{num} = 2^{\text{len}} - 1$$

```
s[0] is 1 time printed
s[1] is 2 times printed
s[2] is 4 times printed
s[i] is printed 2^i times
s[strlen(s)-1] is printed 2^(strlen(s)-1) times
total = 1+2+...+2^(strlen(s)-1)
       = (2^strlen(s)) - 1
```

For example, the following program prints 7 characters.

```
#include<stdio.h>

void abc(char *s)
{
    if(s[0] == '\0')
        return;

    abc(s + 1);
    abc(s + 1);
    printf("%c", s[0]);
}

int main()
{
    abc("xyz");
    return 0;
}
```

Thanks to [bharat nag](#) for suggesting this solution.

Question 2

```
#include<stdio.h>
int fun(int count)
{
    printf("%d\n", count);
    if(count < 3)
    {
        fun(fun(fun(++count)));
    }
    return count;
}

int main()
{
    fun(1);
    return 0;
}
```

Output:

```
1
2
3
3
3
3
3
3
```

The main() function calls fun(1). fun(1) prints “1” and calls fun(fun(fun(2))). fun(2) prints “2” and calls fun(fun(fun(3))). So the function call sequence becomes fun(fun(fun(fun(fun(3)))). fun(3) prints “3” and returns 3 (note that count is not incremented and no more functions are called as the if condition is not true for count 3). So the function call sequence reduces to fun(fun(fun(fun(3)))). fun(3) again prints “3” and returns 3. So the function call again reduces to fun(fun(fun(3))) which again prints “3” and reduces to fun(fun(3)). This continues and we get “3” printed 5 times on the screen.

Please write comments if you find any of the answers/codes incorrect, or you want to share more information/questions about the topics discussed above.

49. Dynamic Programming | Set 6 (Min Cost Path)

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.

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

1	2	3
4	8	2
1	5	3

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).

1	2	3
4	8	2
1	5	3

1) 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]”.

$$\text{minCost}(m, n) = \min(\text{minCost}(m-1, n-1), \text{minCost}(m-1, n), \text{minCost}(m, n-1)) + \text{cost}[m][n]$$

2) Overlapping Subproblems

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

```

/* A Naive recursive implementation of MCP(Minimum Cost Path) problem */
#include<stdio.h>
#include<limits.h>
#define R 3
#define C 3

int min(int x, int y, int z);

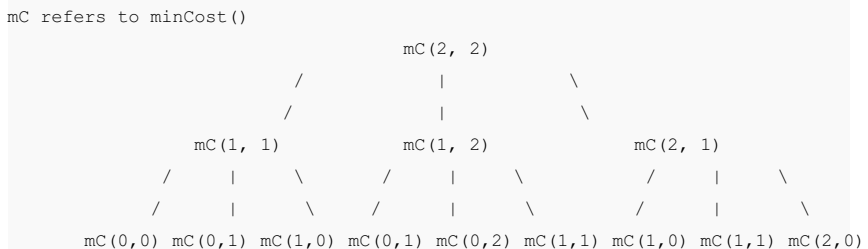
/* Returns cost of minimum cost path from (0,0) to (m, n) in mat[R][C] */
int minCost(int cost[R][C], int m, int n)
{
    if (n < 0 || m < 0)
        return INT_MAX;
    else if (m == 0 && n == 0)
        return cost[m][n];
    else
        return cost[m][n] + min( minCost(cost, m-1, n-1),
         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

/* Driver program to test above functions */
int main()
{
    int cost[R][C] = { {1, 2, 3},
                       {4, 8, 2},
                       {1, 5, 3} };
    printf(" %d ", minCost(cost, 2, 2));
    return 0;
}

```

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 appear more than once. Time complexity of this naive recursive solution is exponential and it is terribly slow.



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.

```
/* Dynamic Programming implementation of MCP problem */
#include<stdio.h>
#include<limits.h>
#define R 3
#define C 3

int min(int x, int y, int z);

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
    // used to keep the program simple and make it working on all comp.
    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];
}

/* 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;
}

/* Driver program to test above functions */
int main()
{
    int cost[R][C] = { {1, 2, 3},
                       {4, 8, 2},
                       {1, 5, 3} };

    printf(" %d ", minCost(cost, 2, 2));
    return 0;
}
```

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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

50. Accessing Grandparent's member in Java

Directly accessing Grandparent's member in Java:

Predict the output of following Java program.

```
// filename Main.java
class Grandparent {
    public void Print() {
        System.out.println("Grandparent's Print()");
    }
}

class Parent extends Grandparent {
    public void Print() {
        System.out.println("Parent's Print()");
    }
}

class Child extends Parent {
    public void Print() {
        super.super.Print(); // Trying to access Grandparent's Print()
        System.out.println("Child's Print()");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.Print();
    }
}
```

Output: Compiler Error

There is error in line “super.super.print();”. In Java, a class cannot directly access the grandparent's members. It is allowed in C++ though. In C++, we can use scope resolution operator (::) to access any ancestor's member in inheritance hierarchy. ***In Java, we can access grandparent's members only through the parent class.*** For example, the following program compiles and runs fine.

```
// filename Main.java
class Grandparent {
    public void Print() {
        System.out.println("Grandparent's Print()");
    }
}

class Parent extends Grandparent {
    public void Print() {
        super.Print();
        System.out.println("Parent's Print()");
    }
}

class Child extends Parent {
    public void Print() {
        super.Print();
        System.out.println("Child's Print()");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.Print();
    }
}
```

Output:

```
Grandparent's Print()
Parent's Print()
Child's Print()
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

51. Comparison of Inheritance in C++ and Java

The purpose of inheritance is same in C++ and Java. Inheritance is used in both languages for reusing code and/or creating is-a relationship. There are following

differences in the way both languages provide support for inheritance.

1) In Java, all classes inherit from the **Object class** directly or indirectly. Therefore, there is always a single inheritance tree of classes in Java, and **Object class** is root of the tree. In Java, if we create a class that doesn't inherit from any class then it automatically inherits from **Object class**. In C++, there is forest of classes; when we create a class that doesn't inherit from anything, we create a new tree in forest.

Following Java example shows that Test class automatically inherits from Object class.

```
class Test {
    // members of test
}

class Main {
    public static void main(String[] args) {
        Test t = new Test();
        System.out.println("t is instanceof Object: " + (t instanceof Object));
    }
}
```

Output:

```
t is instanceof Object: true
```

2) In Java, members of the grandparent class are not directly accessible. See [this G-Fact](#) for more details.

3) The meaning of protected member access specifier is somewhat different in Java. In Java, protected members of a class "A" are accessible in other class "B" of same package, even if B doesn't inherit from A (they both have to be in the same package). For example, in the following program, protected members of A are accessible in B.

```
// filename B.java

class A {
    protected int x = 10, y = 20;
}

class B {
    public static void main(String args[]) {
        A a = new A();
        System.out.println(a.x + " " + a.y);
    }
}
```

4) Java uses *extends* keyword for inheritance. Unlike C++, Java doesn't provide an inheritance specifier like public, protected or private. Therefore, we cannot change the protection level of members of base class in Java, if some data member is public or protected in base class then it remains public or protected in derived class. Like C++, private members of base class are not accessible in derived class.

Unlike C++, in Java, we don't have to remember those rules of inheritance which are combination of base class access specifier and inheritance specifier.

5) In Java, methods are virtual by default. In C++, we explicitly use virtual keyword. See [this G-Fact](#) for more details.

6) Java uses a separate keyword *interface* for interfaces, and *abstract* keyword for abstract classes and abstract functions.

Following is a Java abstract class example.

```
// An abstract class example
abstract class myAbstractClass {

    // An abstract method
    abstract void myAbstractFun();

    // A normal method
    void fun() {
        System.out.println("Inside My fun");
    }
}

public class myClass extends myAbstractClass {
    public void myAbstractFun() {
        System.out.println("Inside My fun");
    }
}
```

Following is a Java interface example

```
// An interface example
public interface myInterface {

    // myAbstractFun() is public and abstract, even if we don't use these keywords
    void myAbstractFun(); // is same as public abstract void myAbstractFun()
}

// Note the implements keyword also.
public class myClass implements myInterface {
    public void myAbstractFun() {
```

```
        System.out.println("Inside My fun");
    }
}
```

7) Unlike C++, Java doesn't support multiple inheritance. A class cannot inherit from more than one class. A class can implement multiple interfaces though.

8) In C++, default constructor of parent class is automatically called, but if we want to call parametrized constructor of a parent class, we must use **Initializer list**. Like C++, default constructor of the parent class is automatically called in Java, but if we want to call parametrized constructor then we must use **super** to call the parent constructor. See following Java example.

```
package main;

class Base {
    private int b;
    Base(int x) {
        b = x;
        System.out.println("Base constructor called");
    }
}

class Derived extends Base {
    private int d;
    Derived(int x, int y) {
        // Calling parent class parameterized constructor
        // Call to parent constructor must be the first line in a Derived class
        super(x);
        d = y;
        System.out.println("Derived constructor called");
    }
}

class Main{
    public static void main(String[] args) {
        Derived obj = new Derived(1, 2);
    }
}
```

Output:

```
Base constructor called
Derived constructor called
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

52. Practice Questions for Recursion | Set 7

Question 1 Predict the output of the following program. What does the following fun() do in general?

```
#include <stdio.h>

int fun ( int n, int *fp )
{
    int t, f;

    if ( n <= 1 )
    {
        *fp = 1;
        return 1;
    }
    t = fun ( n-1, fp );
    f = t + *fp;
    *fp = t;
    return f;
}

int main()
{
    int x = 15;
    printf("%d\n", fun(5, &x));

    return 0;
}
```

Output:

8

The program calculates nth Fibonacci Number. The statement `t = fun (n-1, fp)` gives the (n-1)th Fibonacci number and `*fp` is used to store the (n-2)th Fibonacci Number. Initial value of `*fp` (which is 15 in the above program) doesn't matter. Following recursion tree shows all steps from 1 to 10, for execution of `fun(5, &x)`.

```

              (1) fun(5, fp)
             /           \
        (2) fun(4, fp)   (10) t = 5, f = 8, *fp = 5
           /           \
      (3) fun(3, fp)   (9) t = 3, f = 5, *fp = 3
         /           \
    (4) fun(2, fp)   (8) t = 2, f = 3, *fp = 2
```

```

      /      \
(5) fun(1, fp)   (7) t = 1, f = 2, *fp = 1
/
(6) *fp = 1

```

Question 2: Predict the output of the following program.

```

#include <stdio.h>

void fun(int n)
{
    if(n > 0)
    {
        fun(n-1);
        printf("%d ", n);
        fun(n-1);
    }
}

int main()
{
    fun(4);
    return 0;
}

```

Output

```
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```

```

                fun(4)
                /
            fun(3), print(4), fun(3) [fun(3) prints 1 2 1 3 1 2 1]
            /
        fun(2), print(3), fun(2) [fun(2) prints 1 2 1]
        /
    fun(1), print(2), fun(1) [fun(1) prints 1]
    /
fun(0), print(1), fun(0) [fun(0) does nothing]

```

Please write comments if you find any of the answers/codes incorrect, or you want to share more information/questions about the topics discussed above.

53. Copy Constructor in Java

Like C++, Java also supports copy constructor. But, unlike C++, Java doesn't create a default copy constructor if you don't write your own.

Following is an example Java program that shows a simple use of copy constructor.

```
// filename: Main.java

class Complex {

    private double re, im;

    // A normal parametrized constructor
    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    // copy constructor
    Complex(Complex c) {
        System.out.println("Copy constructor called");
        re = c.re;
        im = c.im;
    }

    // Overriding the toString of Object class
    @Override
    public String toString() {
        return "(" + re + " + " + im + "i)";
    }
}

public class Main {

    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);

        // Following involves a copy constructor call
        Complex c2 = new Complex(c1);

        // Note that following doesn't involve a copy constructor call as
        // non-primitive variables are just references.
        Complex c3 = c2;

        System.out.println(c2); // toString() of c2 is called here
    }
}
```

Output:

```
Copy constructor called  
(10.0 + 15.0i)
```

Now try the following Java program:

```
// filename: Main.java  
  
class Complex {  
  
    private double re, im;  
  
    public Complex(double re, double im) {  
        this.re = re;  
        this.im = im;  
    }  
}  
  
public class Main {  
  
    public static void main(String[] args) {  
        Complex c1 = new Complex(10, 15);  
        Complex c2 = new Complex(c1); // compiler error here  
    }  
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

54. Print all sequences of given length

Given two integers k and n , write a function that prints all the sequences of length k composed of numbers $1, 2, \dots, n$. You need to print these sequences in sorted order.

Examples:

```
Input: k = 2, n = 3
```

```
Output:
```

```
1 1  
1 2  
1 3
```

```
2 1
2 2
2 3
3 1
3 2
3 3
```

Input: $k = 3, n = 4$

Output:

```
1 1 1
1 1 2
1 1 3
1 1 4
1 2 1
.....
.....
4 3 4
4 4 1
4 4 2
4 4 3
4 4 4
```

Method 1 (Simple and Iterative):

The simple idea to print all sequences in sorted order is to start from $\{1\ 1\ \dots\ 1\}$ and keep incrementing the sequence while the sequence doesn't become $\{n\ n\ \dots\ n\}$. Following is the detailed process.

- 1) Create an output array $arr[]$ of size k . Initialize the array as $\{1, 1, \dots, 1\}$.
- 2) Print the array $arr[]$.
- 3) Update the array $arr[]$ so that it becomes immediate successor (to be printed) of itself. For example, immediate successor of $\{1, 1, 1\}$ is $\{1, 1, 2\}$, immediate successor of $\{1, 4, 4\}$ is $\{2, 1, 1\}$ and immediate successor of $\{4, 4, 3\}$ is $\{4\ 4\ 4\}$.
- 4) Repeat steps 2 and 3 while there is a successor array. In other words, while the output array $arr[]$ doesn't become $\{n, n, \dots, n\}$

Now let us talk about how to modify the array so that it contains immediate successor. By definition, the immediate successor should have the same first p terms and larger $(p+1)$ th term. In the original array, $(p+1)$ th term (or $arr[p]$) is smaller than n and all terms after $arr[p]$ i.e., $arr[p+1]$, $arr[p+2]$, \dots $arr[k-1]$ are n .

To find the immediate successor, we find the point p in the previously printed array. To find point p , we start from rightmost side and keep moving till we find a number $arr[p]$ such that $arr[p]$ is smaller than n (or not n). Once we find such a point, we increment it $arr[p]$ by 1 and make all the elements after $arr[p]$ as 1 i.e., we do $arr[p+1] = 1$, $arr[p+2] = 1 \dots arr[k-1] = 1$. Following are the detailed steps to get immediate successor of $arr[]$

- 1) Start from the rightmost term `arr[k-1]` and move toward left. Find the first element `arr[p]` that is not same as `n`.
- 2) Increment `arr[p]` by 1
- 3) Starting from `arr[p+1]` to `arr[k-1]`, set the value of all terms as 1.

```
#include<stdio.h>
```

```
/* A utility function that prints a given arr[] of length size*/
```

```
void printArray(int arr[], int size)
```

```
{
    for(int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return;
}
```

```
/* This function returns 0 if there are no more sequences to be printed,
   modifies arr[] so that arr[] contains next sequence to be printed */
int getSuccessor(int arr[], int k, int n)
```

```
{
    /* start from the rightmost side and find the first number less than n */
    int p = k - 1;
    while (arr[p] == n)
        p--;
```

```
/* If all numbers are n in the array then there is no successor, return 0 */
if (p < 0)
    return 0;
```

```
/* Update arr[] so that it contains successor */
```

```
arr[p] = arr[p] + 1;
for(int i = p + 1; i < k; i++)
    arr[i] = 1;
```

```
return 1;
```

```
}
```

```
/* The main function that prints all sequences from 1, 1, ..1 to n, n, ..n */
void printSequences(int n, int k)
```

```
{
    int *arr = new int[k];
```

```
/* Initialize the current sequence as the first sequence to be printed */
for(int i = 0; i < k; i++)
    arr[i] = 1;
```

```
/* The loop breaks when there are no more successors to be printed */
while(1)
```

```
{
    /* Print the current sequence */
    printArray(arr, k);
```

```
/* Update arr[] so that it contains next sequence to be printed.
   If there are no more sequences then break the loop */
```

```
if(getSuccessor(arr, k, n) == 0)
    break;
```

```
}
```

```
delete(arr); // free dynamically allocated array
return;
```

```
}
```

```
/* Driver Program to test above functions */
int main()
{
    int n = 3;
    int k = 2;
    printSequences(n, k);
    return 0;
}
```

Output:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

Time Complexity: There are total n^k sequences. Printing a sequence and finding its successor take $O(k)$ time. So time complexity of above implementation is $O(k \cdot n^k)$.

Method 2 (Tricky and Recursive)

The recursive function *printSequencesRecur* generates and prints all sequences of length k . The idea is to use one more parameter index. The function *printSequencesRecur* keeps all the terms in `arr[]` same till index, update the value at index and recursively calls itself for more terms after index.

```

#include<stdio.h>

/* A utility function that prints a given arr[] of length size*/
void printArray(int arr[], int size)
{
    for(int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return;
}

/* The core function that recursively generates and prints all sequence
length k */
void printSequencesRecur(int arr[], int n, int k, int index)
{
    int i;
    if (k == 0)
    {
        printArray(arr, index);
    }
    if (k > 0)
    {
        for(i = 1; i<=n; ++i)
        {
            arr[index] = i;
            printSequencesRecur(arr, n, k-1, index+1);
        }
    }
}

/* A function that uses printSequencesRecur() to prints all sequences
from 1, 1, ..1 to n, n, ..n */
void printSequences(int n, int k)
{
    int *arr = new int[k];
    printSequencesRecur(arr, n, k, 0);

    delete(arr); // free dynamically allocated array
    return;
}

/* Driver Program to test above functions */
int main()
{
    int n = 3;
    int k = 2;
    printSequences(n, k);
    return 0;
}

```

Output:

```

1 1
1 2
1 3
2 1
2 2
2 3

```

```
3 1
3 2
3 3
```

Time Complexity: There are n^k sequences and printing a sequence takes $O(k)$ time. So time complexity is $O(k \cdot n^k)$.

Thanks to [mopurizwarriors](#) for suggesting above method. As suggested by [alphayoung](#), we can avoid use of arrays for small sequences that can fit in an integer. Following is the C implementation for same.

```
/* The core function that generates and prints all sequences of length
void printSeqRecur(int num, int pos, int k, int n)
{
    if (pos == k) {
        printf("%d \n", num);
        return;
    }
    for (int i = 1; i <= n; i++) {
        printSeqRecur(num * 10 + i, pos + 1, k, n);
    }
}

/* A function that uses printSequencesRecur() to prints all sequences
from 1, 1, ..1 to n, n, ..n */
void printSequences(int k, int n)
{
    printSeqRecur(0, 0, k, n);
}
```

References:

[Algorithms and Programming: Problems and Solutions by Alexander Shen](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

55. Average of a stream of numbers

Difficulty Level: Rookie

Given a stream of numbers, print average (or mean) of the stream at every point. For example, let us consider the stream as 10, 20, 30, 40, 50, 60, ...

```
Average of 1 numbers is 10.00
Average of 2 numbers is 15.00
Average of 3 numbers is 20.00
Average of 4 numbers is 25.00
Average of 5 numbers is 30.00
```

```
Average of 6 numbers is 35.00
```

```
.....
```

To print mean of a stream, we need to find out how to find average when a new number is being added to the stream. To do this, all we need is count of numbers seen so far in the stream, previous average and new number. Let n be the count, $prev_avg$ be the previous average and x be the new number being added. The average after including x number can be written as $(prev_avg*n + x)/(n+1)$.

```
#include <stdio.h>
```

```
// Returns the new average after including x
float getAvg(float prev_avg, int x, int n)
{
    return (prev_avg*n + x)/(n+1);
}

// Prints average of a stream of numbers
void streamAvg(float arr[], int n)
{
    float avg = 0;
    for(int i = 0; i < n; i++)
    {
        avg = getAvg(avg, arr[i], i);
        printf("Average of %d numbers is %f \n", i+1, avg);
    }
    return;
}

// Driver program to test above functions
int main()
{
    float arr[] = {10, 20, 30, 40, 50, 60};
    int n = sizeof(arr)/sizeof(arr[0]);
    streamAvg(arr, n);

    return 0;
}
```

The above function `getAvg()` can be optimized using following changes. We can avoid the use of `prev_avg` and number of elements by using static variables (Assuming that only this function is called for average of stream). Following is the optimized version.

```

#include <stdio.h>

// Returns the new average after including x
float getAvg (int x)
{
    static int sum, n;

    sum += x;
    return (((float)sum)/++n);
}

// Prints average of a stream of numbers
void streamAvg(float arr[], int n)
{
    float avg = 0;
    for(int i = 0; i < n; i++)
    {
        avg = getAvg(arr[i]);
        printf("Average of %d numbers is %f \n", i+1, avg);
    }
    return;
}

// Driver program to test above functions
int main()
{
    float arr[] = {10, 20, 30, 40, 50, 60};
    int n = sizeof(arr)/sizeof(arr[0]);
    streamAvg(arr, n);

    return 0;
}

```

Thanks to [Abhijeet Deshpande](#) for suggesting this optimized version.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

56. Dynamic Programming | Set 7 (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, ... , S_m} 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.

1) 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 S_m).

2) Solutions that contain at least one S_m .

Let $\text{count}(S[], m, n)$ be the function to count the number of solutions, then it can be written as sum of $\text{count}(S[], m-1, n)$ and $\text{count}(S[], m, n-S_m)$.

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

2) Overlapping Subproblems

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

```
#include<stdio.h>
```

```
// Returns the count of ways we can sum S[0...m-1] coins to get sum n
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
    if (m <= 0 && n >= 1)
        return 0;

    // count is sum of solutions (i) including S[m-1] (ii) excluding S
    return count( S, m - 1, n ) + count( S, m, n-S[m-1] );
}
```

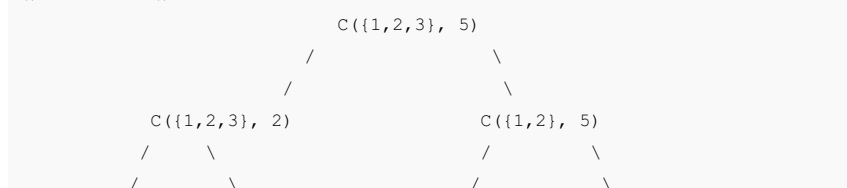
```
// Driver program to test above function
```

```
int main()
{
    int i, j;
    int arr[] = {1, 2, 3};
    int m = sizeof(arr)/sizeof(arr[0]);
    printf("%d ", count(arr, m, 4));
    getchar();
    return 0;
}
```

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()
```




```
#include<stdio.h>
```

```
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
    // the base case 0 value case (n = 0)
    int table[n+1][m];

    // Fill the entries 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];
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 2, 3};
    int m = sizeof(arr)/sizeof(arr[0]);
    int n = 4;
    printf(" %d ", count(arr, m, n));
    return 0;
}
```

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];
}

```

Thanks to [Rohan Laishram](#) for suggesting this space optimized version.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

http://www.algorithmist.com/index.php/Coin_Change

57. Dynamic Programming | Set 8 (Matrix Chain Multiplication)

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

$$(ABC)D = (AB)(CD) = A(BCD) = \dots$$

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10×30 matrix, B is a 30×5 matrix, and C is a 5×60 matrix. Then,

$$\begin{aligned}
 (AB)C &= (10 \times 30 \times 5) + (10 \times 5 \times 60) = 1500 + 3000 = 4500 \text{ operations} \\
 A(BC) &= (30 \times 5 \times 60) + (10 \times 30 \times 60) = 9000 + 18000 = 27000 \text{ operations.}
 \end{aligned}$$

Clearly the first parenthesization requires less number of operations.

Given an array $p[]$ which represents the chain of matrices such that the i th matrix A_i is of dimension $p[i-1] \times p[i]$. We need to write a function `MatrixChainOrder()` that should return the minimum number of multiplications needed to multiply the chain.

Input: `p[] = {40, 20, 30, 10, 30}`

Output: 26000

There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30.

Let the input 4 matrices be A, B, C and D. The minimum number of multiplications are obtained by putting parenthesis in following way

$(A(BC))D \rightarrow 20 \times 30 \times 10 + 40 \times 20 \times 10 + 40 \times 10 \times 30$

Input: `p[] = {10, 20, 30, 40, 30}`

Output: 30000

There are 4 matrices of dimensions 10x20, 20x30, 30x40 and 40x30.

Let the input 4 matrices be A, B, C and D. The minimum number of multiplications are obtained by putting parenthesis in following way

$((AB)C)D \rightarrow 10 \times 20 \times 30 + 10 \times 30 \times 40 + 10 \times 40 \times 30$

Input: `p[] = {10, 20, 30}`

Output: 6000

There are only two matrices of dimensions 10x20 and 20x30. So there is only one way to multiply the matrices, cost of which is $10 \times 20 \times 30$

1) Optimal Substructure:

A simple solution is to place parenthesis at all possible places, calculate the cost for each placement and return the minimum value. In a chain of matrices of size n , we can place the first set of parenthesis in $n-1$ ways. For example, if the given chain is of 4 matrices. let the chain be ABCD, then there are 3 way to place first set of parenthesis: A(BCD), (AB)CD and (ABC)D. So when we place a set of parenthesis, we divide the problem into subproblems of smaller size. Therefore, the problem has optimal substructure property and can be easily solved using recursion.

Minimum number of multiplication needed to multiply a chain of size n = Minimum of all $n-1$ placements (these placements create subproblems of smaller size)

2) Overlapping Subproblems

Following is a recursive implementation that simply follows the above optimal substructure property.

```

/* A naive recursive implementation that simply follows the above opti
substructure property */
#include<stdio.h>
#include<limits.h>

// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int i, int j)
{
    if(i == j)
        return 0;
    int k;
    int min = INT_MAX;
    int count;

    // place parenthesis at different places between first and last ma
    // recursively calculate count of multiplications for each parenthe
    // placement and return the minimum count
    for (k = i; k < j; k++)
    {
        count = MatrixChainOrder(p, i, k) +
                MatrixChainOrder(p, k+1, j) +
                p[i-1]*p[k]*p[j];

        if (count < min)
            min = count;
    }

    // Return minimum count
    return min;
}

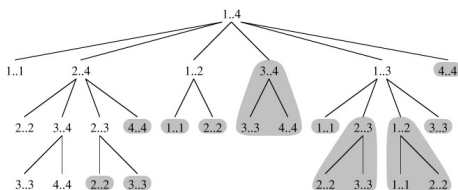
// Driver program to test above function
int main()
{
    int arr[] = {1, 2, 3, 4, 3};
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Minimum number of multiplications is %d ",
           MatrixChainOrder(arr, 1, n-1));

    getchar();
    return 0;
}

```

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for a matrix chain of size 4. The function `MatrixChainOrder(p, 3, 4)` is called two times. We can see that there are many subproblems being called more than once.



Since same subproblems are called again, this problem has Overlapping Subproblems property. So Matrix Chain Multiplication 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 $m[][]$ in bottom up manner.

Dynamic Programming Solution

Following is C/C++ implementation for Matrix Chain Multiplication problem using Dynamic Programming.

```

// See the Cormen book for details of the following algorithm
#include<stdio.h>
#include<limits.h>

// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int n)
{
    /* For simplicity of the program, one extra row and one extra column
       allocated in m[][]. 0th row and 0th column of m[][] are not used */
    int m[n][n];

    int i, j, k, L, q;

    /* m[i,j] = Minimum number of scalar multiplications needed to compute
       the matrix A[i]A[i+1]...A[j] = A[i..j] where dimension of A[i] is
       p[i-1] x p[i] */

    // cost is zero when multiplying one matrix.
    for (i = 1; i < n; i++)
        m[i][i] = 0;

    // L is chain length.
    for (L=2; L<n; L++)
    {
        for (i=1; i<=n-L+1; i++)
        {
            j = i+L-1;
            m[i][j] = INT_MAX;
            for (k=i; k<=j-1; k++)
            {
                // q = cost/scalar multiplications
                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }

    return m[1][n-1];
}

int main()
{
    int arr[] = {1, 2, 3, 4};
    int size = sizeof(arr)/sizeof(arr[0]);

    printf("Minimum number of multiplications is %d ",
           MatrixChainOrder(arr, size));

    getchar();
    return 0;
}

```

Time Complexity: $O(n^3)$

Auxiliary Space: $O(n^2)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

http://en.wikipedia.org/wiki/Matrix_chain_multiplication

<http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Dynamic/chainMatrixMult.h>

58. Dynamic Programming | Set 9 (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 $k = 2$.

1) Optimal Substructure

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

$$\begin{aligned}C(n, k) &= C(n-1, k-1) + C(n-1, k) \\C(n, 0) &= C(n, n) = 1\end{aligned}$$

2) Overlapping Subproblems

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

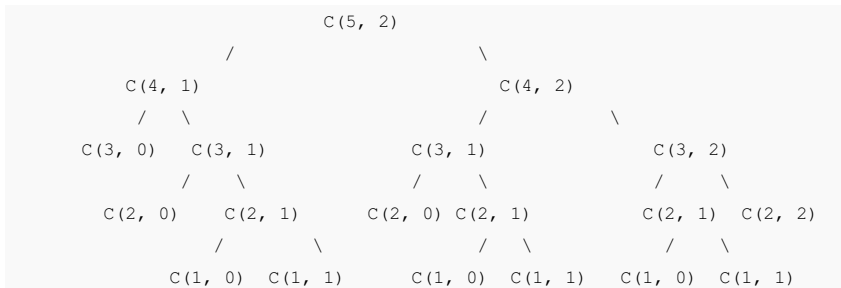
```
// A Naive Recursive Implementation
#include<stdio.h>

// 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);
}

/* 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;
}
```

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



Since same subproblems are called again, this problem has Overlapping Subproblems property. So the Binomial Coefficient 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 $C[][]$ in bottom up manner. Following is Dynamic Programming based implementation.


```

// A Dynamic Programming based solution that uses table C[][] to calculate 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;

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

    return C[n][k];
}

// A utility function to return minimum of two integers
int min(int a, int b)
{
    return (a<b)? a: b;
}

/* Driver 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: $O(n*k)$

Following is a space optimized version of the above code. The following code only uses $O(k)$. Thanks to [AK](#) for suggesting this method.

```
// A space optimized Dynamic Programming Solution
int binomialCoeff(int n, int k)
{
    int* C = (int*)calloc(k+1, sizeof(int));
    int i, j, res;

    C[0] = 1;

    for(i = 1; i <= n; i++)
    {
        for(j = min(i, k); j > 0; j--)
            C[j] = C[j] + C[j-1];
    }

    res = C[k]; // Store the result before freeing memory

    free(C); // free dynamically allocated memory to avoid memory leak

    return res;
}
```

Time Complexity: $O(n \cdot k)$

Auxiliary Space: $O(k)$

References:

<http://www.csl.mtu.edu/cs4321/www/Lectures/Lecture%2015%20-%20Dynamic%20Programming%20Binomial%20Coefficients.htm>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

59. Dynamic Programming | Set 10 (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).

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.

```
/* A Naive recursive implementation of 0-1 Knapsack problem */
#include<stdio.h>

// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// 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) );
}

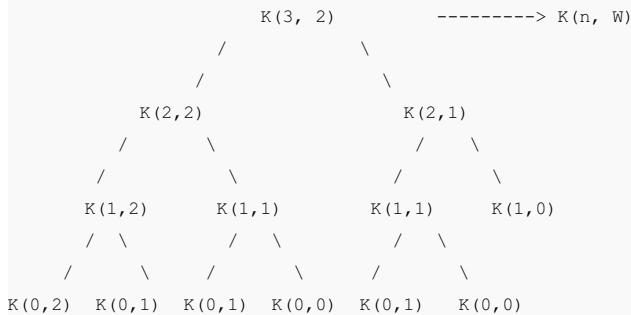
// Driver program to test above function
int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}
```

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 $\text{knapSack}()$. 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}
```



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, recomputations of same subproblems can be avoided by constructing a temporary array $K[][]$ in bottom up manner. Following is Dynamic Programming based implementation.

```

// A Dynamic Programming based solution for 0-1 Knapsack problem
#include<stdio.h>

// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// Returns the maximum value that can be put in a knapsack of capacity
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];
}

int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}

```

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

References:

<http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf>

<http://www.cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

60. Greedy Algorithms | Set 1 (Activity Selection Problem)

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Greedy algorithms are used for optimization problems. An optimization problem can be solved using Greedy if the problem has the following property: *At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem.*

If a Greedy Algorithm can solve a problem, then it generally becomes the best method to solve that problem as the Greedy algorithms are in general more efficient than other techniques like Dynamic Programming. But Greedy algorithms cannot always be applied. For example, Fractional Knapsack problem (See [this](#)) can be solved using Greedy, but **0-1 Knapsack** cannot be solved using Greedy.

Following are some standard algorithms that are Greedy algorithms.

1) Kruskal's Minimum Spanning Tree (MST): In Kruskal's algorithm, we create a MST by picking edges one by one. The Greedy Choice is to pick the smallest weight edge that doesn't cause a cycle in the MST constructed so far.

2) Prim's Minimum Spanning Tree: In Prim's algorithm also, we create a MST by picking edges one by one. We maintain two sets: set of the vertices already included in MST and the set of the vertices not yet included. The Greedy Choice is to pick the smallest weight edge that connects the two sets.

3) Dijkstra's Shortest Path: The Dijkstra's algorithm is very similar to Prim's algorithm. The shortest path tree is built up, edge by edge. We maintain two sets: set of the vertices already included in the tree and the set of the vertices not yet included. The Greedy Choice is to pick the edge that connects the two sets and is on the smallest weight path from source to the set that contains not yet included vertices.

4) Huffman Coding: Huffman Coding is a loss-less compression technique. It assigns variable length bit codes to different characters. The Greedy Choice is to assign least bit length code to the most frequent character.

The greedy algorithms are sometimes also used to get an approximation for Hard optimization problems. For example, **Traveling Salesman Problem** is a NP Hard problem. A Greedy choice for this problem is to pick the nearest unvisited city from the current city at every step. This solution doesn't always produce the best optimal solution, but can be used to get an approximate optimal solution.

Let us consider the **Activity Selection problem** as our first example of Greedy algorithms. Following is the problem statement.

You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

Example:

Consider the following 6 activities.

```
start[] = {1, 3, 0, 5, 8, 5};
finish[] = {2, 4, 6, 7, 9, 9};
```

The maximum set of activities that can be executed

```
by a single person is {0, 1, 3, 4}
```

The greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of previously selected activity. We can sort the activities according to their finishing time so that we always consider the next activity as minimum finishing time activity.

- 1) Sort the activities according to their finishing time
- 2) Select the first activity from the sorted array and print it.
- 3) Do following for remaining activities in the sorted array.
.....a) If the start time of this activity is greater than the finish time of previously selected activity then select this activity and print it.

In the following C implementation, it is assumed that the activities are already sorted according to their finish time.

```
#include<stdio.h>

// Prints a maximum set of activities that can be done by a single
// person, one at a time.
// n --> Total number of activities
// s[] --> An array that contains start time of all activities
// f[] --> An array that contains finish time of all activities
void printMaxActivities(int s[], int f[], int n)
{
    int i, j;

    printf ("Following activities are selected \n");

    // The first activity always gets selected
    i = 0;
    printf("%d ", i);

    // Consider rest of the activities
    for (j = 1; j < n; j++)
    {
        // If this activity has start time greater than or equal to the
        // time of previously selected activity, then select it
        if (s[j] >= f[i])
        {
            printf ("%d ", j);
            i = j;
        }
    }
}

// driver program to test above function
int main()
{
    int s[] = {1, 3, 0, 5, 8, 5};
    int f[] = {2, 4, 6, 7, 9, 9};
    int n = sizeof(s)/sizeof(s[0]);
    printMaxActivities(s, f, n);
    getchar();
    return 0;
}
```

Output:

```
Following activities are selected  
0 1 3 4
```

How does Greedy Choice work for Activities sorted according to finish time?

Let the give set of activities be $S = \{1, 2, 3, \dots, n\}$ and activities be sorted by finish time. The greedy choice is to always pick activity 1. How come the activity 1 always provides one of the optimal solutions. We can prove it by showing that if there is another solution B with first activity other than 1, then there is also a solution A of same size with activity 1 as first activity. Let the first activity selected by B be k, then there always exist $A = \{B - \{k\}\} \cup \{1\}$. (Note that the activities in B are independent and k has smallest finishing time among all. Since k is not 1, $\text{finish}(k) \geq \text{finish}(1)$).

References:

Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
Algorithms by S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani
http://en.wikipedia.org/wiki/Greedy_algorithm

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

61. Dynamic Programming | Set 11 (Egg Dropping Puzzle)

The following is a description of the instance of this famous puzzle involving $n=2$ eggs and a building with $k=36$ floors.

Suppose that we wish to know which stories in a 36-story building are safe to drop eggs from, and which will cause the eggs to break on landing. We make a few assumptions:

-An egg that survives a fall can be used again.
-A broken egg must be discarded.
-The effect of a fall is the same for all eggs.
-If an egg breaks when dropped, then it would break if dropped from a higher floor.
-If an egg survives a fall then it would survive a shorter fall.
-It is not ruled out that the first-floor windows break eggs, nor is it ruled out that the 36th-floor do not cause an egg to break.

If only one egg is available and we wish to be sure of obtaining the right result, the experiment can be carried out in only one way. Drop the egg from the first-floor window; if it survives, drop it from the second floor window. Continue upward until it breaks. In the worst case, this method may require 36 droppings. Suppose 2 eggs are available. What

is the least number of egg-droppings that is guaranteed to work in all cases?
The problem is not actually to find the critical floor, but merely to decide floors from which eggs should be dropped so that total number of trials are minimized.

Source: [Wiki for Dynamic Programming](#)

In this post, we will discuss solution to a general problem with n eggs and k floors. The solution is to try dropping an egg from every floor (from 1 to k) and recursively calculate the minimum number of droppings needed in worst case. The floor which gives the minimum value in worst case is going to be part of the solution.

In the following solutions, we return the minimum number of trails in worst case; these solutions can be easily modified to print floor numbers of every trials also.

1) Optimal Substructure:

When we drop an egg from a floor x , there can be two cases (1) The egg breaks (2) The egg doesn't break.

- 1) If the egg breaks after dropping from x th floor, then we only need to check for floors lower than x with remaining eggs; so the problem reduces to $x-1$ floors and $n-1$ eggs
- 2) If the egg doesn't break after dropping from the x th floor, then we only need to check for floors higher than x ; so the problem reduces to $k-x$ floors and n eggs.

Since we need to minimize the number of trials in *worst* case, we take the maximum of two cases. We consider the max of above two cases for every floor and choose the floor which yields minimum number of trials.

```
k ==> Number of floors
n ==> Number of Eggs
eggDrop(n, k) ==> Minimum number of trails needed to find the critical
                    floor in worst case.
eggDrop(n, k) = 1 + min{max(eggDrop(n - 1, x - 1), eggDrop(n, k - x)):
                        x in {1, 2, ..., k}}
```

2) Overlapping Subproblems

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

```

#include <stdio.h>
#include <limits.h>

// A utility function to get maximum of two integers
int max(int a, int b) { return (a > b)? a: b; }

/* Function to get minimum number of trials needed in worst
case with n eggs and k floors */
int eggDrop(int n, int k)
{
    // If there are no floors, then no trials needed. OR if there is
    // one floor, one trial needed.
    if (k == 1 || k == 0)
        return k;

    // We need k trials for one egg and k floors
    if (n == 1)
        return k;

    int min = INT_MAX, x, res;

    // Consider all droppings from 1st floor to kth floor and
    // return the minimum of these values plus 1.
    for (x = 1; x <= k; x++)
    {
        res = max(eggDrop(n-1, x-1), eggDrop(n, k-x));
        if (res < min)
            min = res;
    }

    return min + 1;
}

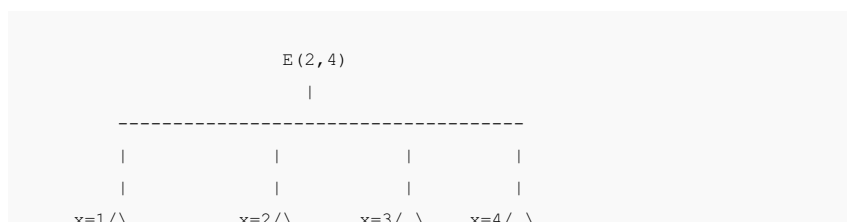
/* Driver program to test to pront printDups*/
int main()
{
    int n = 2, k = 10;
    printf ("\nMinimum number of trials in worst case with %d eggs and
           \"%d floors is %d \n", n, k, eggDrop(n, k));
    return 0;
}

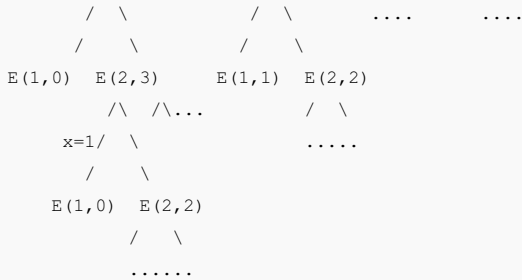
```

Output:

```
Minimum number of trials in worst case with 2 eggs and 10 floors is 4
```

It should be noted that the above function computes the same subproblems again and again. See the following partial recursion tree, $E(2, 2)$ is being evaluated twice. There will many repeated subproblems when you draw the complete recursion tree even for small values of n and k .





Partial recursion tree for 2 eggs and 4 floors.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So Egg Dropping Puzzle 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 `eggFloor[][]` in bottom up manner.

Dynamic Programming Solution

Following is C/C++ implementation for Egg Dropping problem using Dynamic Programming.

```

#include <stdio.h>
#include <limits.h>

// A utility function to get maximum of two integers
int max(int a, int b) { return (a > b)? a: b; }

/* Function to get minimum number of trials needed in worst
case with n eggs and k floors */
int eggDrop(int n, int k)
{
    /* A 2D table where entry eggFloor[i][j] will represent minimum
    number of trials needed for i eggs and j floors. */
    int eggFloor[n+1][k+1];
    int res;
    int i, j, x;

    // We need one trial for one floor and 0 trials for 0 floors
    for (i = 1; i <= n; i++)
    {
        eggFloor[i][1] = 1;
        eggFloor[i][0] = 0;
    }

    // We always need j trials for one egg and j floors.
    for (j = 1; j <= k; j++)
        eggFloor[1][j] = j;

    // Fill rest of the entries in table using optimal substructure
    // property
    for (i = 2; i <= n; i++)
    {
        for (j = 2; j <= k; j++)
        {
            eggFloor[i][j] = INT_MAX;
            for (x = 1; x <= j; x++)
            {
                res = 1 + max(eggFloor[i-1][x-1], eggFloor[i][j-x]);
                if (res < eggFloor[i][j])
                    eggFloor[i][j] = res;
            }
        }
    }

    // eggFloor[n][k] holds the result
    return eggFloor[n][k];
}

/* Driver program to test to print printDups*/
int main()
{
    int n = 2, k = 36;
    printf ("\nMinimum number of trials in worst case with %d eggs and
    \"%d floors is %d \n", n, k, eggDrop(n, k));
    return 0;
}

```

Output:

Minimum number of trials in worst case with 2 eggs and 36 floors is 8

Time Complexity: $O(nk^2)$

Auxiliary Space: $O(nk)$

As an exercise, you may try modifying the above DP solution to print all intermediate floors (The floors used for minimum trail solution).

References:

<http://archive.itejournal.informs.org/Vol4No1/Sniedovich/index.php>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

62. Backtracking | Set 5 (m Coloring Problem)

Given an undirected graph and a number m , determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with same color. Here coloring of a graph means assignment of colors to all vertices.

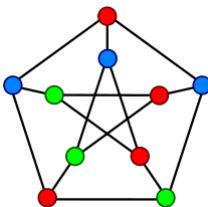
Input:

- 1) A 2D array $\text{graph}[V][V]$ where V is the number of vertices in graph and $\text{graph}[V][V]$ is adjacency matrix representation of the graph. A value $\text{graph}[i][j]$ is 1 if there is a direct edge from i to j , otherwise $\text{graph}[i][j]$ is 0.
- 2) An integer m which is maximum number of colors that can be used.

Output:

An array $\text{color}[V]$ that should have numbers from 1 to m . $\text{color}[i]$ should represent the color assigned to the i th vertex. The code should also return false if the graph cannot be colored with m colors.

Following is an example graph (from [Wiki page](#)) that can be colored with 3 colors.



Naive Algorithm

Generate all possible configurations of colors and print a configuration that satisfies the given constraints.

```
while there are untried conflagrations
{
    generate the next configuration
```

```

    if no adjacent vertices are colored with same color
    {
        print this configuration;
    }
}

```

There will be V^m configurations of colors.

Backtracking Algorithm

The idea is to assign colors one by one to different vertices, starting from the vertex 0. Before assigning a color, we check for safety by considering already assigned colors to the adjacent vertices. If we find a color assignment which is safe, we mark the color assignment as part of solution. If we do not find a color due to clashes then we backtrack and return false.

Implementation of Backtracking solution

```

#include<stdio.h>

// Number of vertices in the graph
#define V 4

void printSolution(int color[]);

/* A utility function to check if the current color assignment
   is safe for vertex v */
bool isSafe (int v, bool graph[V][V], int color[], int c)
{
    for (int i = 0; i < V; i++)
        if (graph[v][i] && c == color[i])
            return false;
    return true;
}

/* A recursive utility function to solve m coloring problem */
bool graphColoringUtil(bool graph[V][V], int m, int color[], int v)
{
    /* base case: If all vertices are assigned a color then
       return true */
    if (v == V)
        return true;

    /* Consider this vertex v and try different colors */
    for (int c = 1; c <= m; c++)
    {
        /* Check if assignment of color c to v is fine*/
        if (isSafe(v, graph, color, c))
        {
            color[v] = c;

            /* recur to assign colors to rest of the vertices */
            if (graphColoringUtil (graph, m, color, v+1) == true)
                return true;

            /* If assigning color c doesn't lead to a solution
               then remove it */
            color[v] = 0;
        }
    }
}

```

```

    }
    /* If no color can be assigned to this vertex then return false */
    return false;
}

/* This function solves the m Coloring problem using Backtracking.
It mainly uses graphColoringUtil() to solve the problem. It returns
false if the m colors cannot be assigned, otherwise return true and
prints assignments of colors to all vertices. Please note that there
may be more than one solutions, this function prints one of the
feasible solutions.*/
bool graphColoring(bool graph[V][V], int m)
{
    // Initialize all color values as 0. This initialization is needed
    // correct functioning of isSafe()
    int *color = new int[V];
    for (int i = 0; i < V; i++)
        color[i] = 0;

    // Call graphColoringUtil() for vertex 0
    if (graphColoringUtil(graph, m, color, 0) == false)
    {
        printf("Solution does not exist");
        return false;
    }

    // Print the solution
    printSolution(color);
    return true;
}

/* A utility function to print solution */
void printSolution(int color[])
{
    printf("Solution Exists:");
    printf("Following are the assigned colors \n");
    for (int i = 0; i < V; i++)
        printf(" %d ", color[i]);
    printf("\n");
}

// driver program to test above function
int main()
{
    /* Create following graph and test whether it is 3 colorable
    (3)---(2)
    |      / \
    |      /   \
    (0)---(1)
    */
    bool graph[V][V] = {{0, 1, 1, 1},
        {1, 0, 1, 0},
        {1, 1, 0, 1},
        {1, 0, 1, 0},
    };
    int m = 3; // Number of colors
    graphColoring (graph, m);
    return 0;
}

```

Output:

Solution Exists: Following are the assigned colors

```
1  2  3  2
```

References:

http://en.wikipedia.org/wiki/Graph_coloring

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

63. Dynamic Programming | Set 12 (Longest Palindromic Subsequence)

Given a sequence, find the length of the longest palindromic subsequence in it. For example, if the given sequence is "BBABCB CAB", then the output should be 7 as "BABCBAB" is the longest palindromic subsequence in it. "BBBBB" and "BBCBB" are also palindromic subsequences of the given sequence, but not the longest ones.

The naive solution for this problem is to generate all subsequences of the given sequence and find the longest palindromic 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 and can efficiently solved using Dynamic Programming.

1) Optimal Substructure:

Let $X[0..n-1]$ be the input sequence of length n and $L(0, n-1)$ be the length of the longest palindromic subsequence of $X[0..n-1]$.

If last and first characters of X are same, then $L(0, n-1) = L(1, n-2) + 2$.

Else $L(0, n-1) = \text{MAX} (L(1, n-1), L(0, n-2))$.

Following is a general recursive solution with all cases handled.

```
// Every single character is a palindrom of length 1
L(i, i) = 1 for all indexes i in given sequence

// IF first and last characters are not same
If (X[i] != X[j]) L(i, j) = max{L(i + 1, j), L(i, j - 1)}

// If there are only 2 characters and both are same
Else if (j == i + 1) L(i, j) = 2
```



```
// If there are more than two characters, and first and last
// characters are same
Else L(i, j) = L(i + 1, j - 1) + 2
```

2) Overlapping Subproblems

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

```
#include<stdio.h>
#include<string.h>

// A utility function to get max of two integers
int max (int x, int y) { return (x > y)? x : y; }

// Returns the length of the longest palindromic subsequence in seq
int lps(char *seq, int i, int j)
{
    // Base Case 1: If there is only 1 character
    if (i == j)
        return 1;

    // Base Case 2: If there are only 2 characters and both are same
    if (seq[i] == seq[j] && i + 1 == j)
        return 2;

    // If the first and last characters match
    if (seq[i] == seq[j])
        return lps (seq, i+1, j-1) + 2;

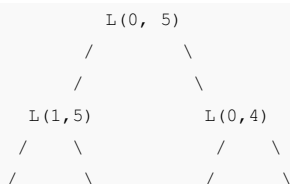
    // If the first and last characters do not match
    return max( lps(seq, i, j-1), lps(seq, i+1, j) );
}

/* Driver program to test above functions */
int main()
{
    char seq[] = "GEEKSFORGEEKS";
    int n = strlen(seq);
    printf ("The length of the LPS is %d", lps(seq, 0, n-1));
    getchar();
    return 0;
}
```

Output:

```
The length of the LPS is 5
```

Considering the above implementation, following is a partial recursion tree for a sequence of length 6 with all different characters.



$L(2, 5)$ $L(1, 4)$ $L(1, 4)$ $L(0, 3)$

In the above partial recursion tree, $L(1, 4)$ 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. Since same subproblems are called again, this problem has Overlapping Subproblems property. So LPS 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 $L[][]$ in bottom up manner.

Dynamic Programming Solution

```

#include<stdio.h>
#include<string.h>

// A utility function to get max of two integers
int max (int x, int y) { return (x > y)? x : y; }

// Returns the length of the longest palindromic subsequence in seq
int lps(char *str)
{
    int n = strlen(str);
    int i, j, cl;
    int L[n][n]; // Create a table to store results of subproblems

    // Strings of length 1 are palindrome of length 1
    for (i = 0; i < n; i++)
        L[i][i] = 1;

    // Build the table. Note that the lower diagonal values of table are
    // useless and not filled in the process. The values are filled in
    // manner similar to Matrix Chain Multiplication DP solution (See
    // http://www.geeksforgeeks.org/archives/15553). cl is length of
    // substring
    for (cl=2; cl<=n; cl++)
    {
        for (i=0; i<=n-cl; i++)
        {
            j = i+cl-1;
            if (str[i] == str[j] && cl == 2)
                L[i][j] = 2;
            else if (str[i] == str[j])
                L[i][j] = L[i+1][j-1] + 2;
            else
                L[i][j] = max(L[i][j-1], L[i+1][j]);
        }
    }

    return L[0][n-1];
}

/* Driver program to test above functions */
int main()
{
    char seq[] = "GEEKS FOR GEEKS";
    int n = strlen(seq);
    printf ("The length of the LPS is %d", lps(seq));
    getchar();
    return 0;
}

```

Output:

```
The length of the LPS is 7
```

Time Complexity of the above implementation is $O(n^2)$ which is much better than the worst case time complexity of Naive Recursive implementation.

This problem is close to the **Longest Common Subsequence (LCS)** problem. In fact, we can use LCS as a subroutine to solve this problem. Following is the two step solution

that uses LCS.

- 1) Reverse the given sequence and store the reverse in another array say `rev[0..n-1]`
 - 2) LCS of the given sequence and `rev[]` will be the longest palindromic sequence.
- This solution is also a $O(n^2)$ solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

<http://users.eecs.northwestern.edu/~dda902/336/hw6-sol.pdf>

64. Dynamic Programming | Set 13 (Cutting a Rod)

Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n . Determine the maximum value obtainable by cutting up the rod and selling the pieces. For example, if length of the rod is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6)

length		1	2	3	4	5	6	7	8

price		1	5	8	9	10	17	17	20

And if the prices are as following, then the maximum obtainable value is 24 (by cutting in eight pieces of length 1)

length		1	2	3	4	5	6	7	8

price		3	5	8	9	10	17	17	20

The naive solution for this problem is to generate all configurations of different pieces and find the highest priced configuration. 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 and can efficiently solved using Dynamic Programming.

1) Optimal Substructure:

We can get the best price by making a cut at different positions and comparing the values obtained after a cut. We can recursively call the same function for a piece obtained after a cut.

Let `cutRoad(n)` be the required (best possible price) value for a rod of length n .

cutRod(n) can be written as following.

$\text{cutRod}(n) = \max(\text{price}[i] + \text{cutRod}(n-i-1))$ for all i in $\{0, 1 \dots n-1\}$

2) Overlapping Subproblems

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

```
// A Naive recursive solution for Rod cutting problem
#include<stdio.h>
#include<limits.h>

// A utility function to get the maximum of two integers
int max(int a, int b) { return (a > b)? a : b;}

/* Returns the best obtainable price for a rod of length n and
price[] as prices of different pieces */
int cutRod(int price[], int n)
{
    if (n <= 0)
        return 0;
    int max_val = INT_MIN;

    // Recursively cut the rod in different pieces and compare different
    // configurations
    for (int i = 0; i < n; i++)
        max_val = max(max_val, price[i] + cutRod(price, n-i-1));

    return max_val;
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 5, 8, 9, 10, 17, 17, 20};
    int size = sizeof(arr)/sizeof(arr[0]);
    printf("Maximum Obtainable Value is %d\n", cutRod(arr, size));
    getchar();
    return 0;
}
```

Output:

Maximum Obtainable Value is 22

Considering the above implementation, following is recursion tree for a Rod of length 4.

```
cR () ----> cutRod ()

                        cR (4)
                       /  |  \  \
                      /   |   \  \
                     /    |    \  \
                    cR (3) cR (2) cR (1) cR (0)
                   /  \  /  \  |
                  /   \| /   \| |
                 /    \| /    \| |
                /     \| /     \| |
               /      \| /      \| |
              /       \| /       \| |
             /        \| /        \| |
            /         \| /         \| |
           /          \| /          \| |
          /           \| /           \| |
         /            \| /            \| |
        /             \| /             \| |
       /              \| /              \| |
      /               \| /               \| |
     /                \| /                \| |
    /                 \| /                 \| |
   /                  \| /                  \| |
  /                   \| /                   \| |
 /                    \| /                    \| |
/                     \| /                     \| |
```

```

      cR(2) cR(1) cR(0) cR(1) cR(0) cR(0)
    /  \      |      |
  /    \      |      |
cR(1) cR(0) cR(0)      cR(0)

```

In the above partial recursion tree, cR(2) is being solved twice. We can see that there are many subproblems which are solved again and again. Since same subproblems are called again, this problem has Overlapping Subproblems property. So the Rod Cutting 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 val[] in bottom up manner.

```

// A Dynamic Programming solution for Rod cutting problem
#include<stdio.h>
#include<limits.h>

```

```

// A utility function to get the maximum of two integers
int max(int a, int b) { return (a > b)? a : b;}

```

```

/* Returns the best obtainable price for a rod of length n and
price[] as prices of different pieces */
int cutRod(int price[], int n)
{
    int val[n+1];
    val[0] = 0;
    int i, j;

    // Build the table val[] in bottom up manner and return the last en
    // from the table
    for (i = 1; i<=n; i++)
    {
        int max_val = INT_MIN;
        for (j = 0; j < i; j++)
            max_val = max(max_val, price[j] + val[i-j-1]);
        val[i] = max_val;
    }
    return val[n];
}

```

```

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 5, 8, 9, 10, 17, 17, 20};
    int size = sizeof(arr)/sizeof(arr[0]);
    printf("Maximum Obtainable Value is %d\n", cutRod(arr, size));
    getchar();
    return 0;
}

```

Output:

```
Maximum Obtainable Value is 22
```

Time Complexity of the above implementation is $O(n^2)$ which is much better than the

worst case time complexity of Naive Recursive implementation.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

65. Josephus problem | Set 1 (A $O(n)$ Solution)

In computer science and mathematics, the **Josephus Problem** (or **Josephus permutation**) is a theoretical problem. Following is the problem statement:

There are n people standing in a circle waiting to be executed. The counting out begins at some point in the circle and proceeds around the circle in a fixed direction. In each step, a certain number of people are skipped and the next person is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last person remains, who is given freedom. Given the total number of persons n and a number k which indicates that $k-1$ persons are skipped and k th person is killed in circle. The task is to choose the place in the initial circle so that you are the last one remaining and so survive.

For example, if $n = 5$ and $k = 2$, then the safe position is 3. Firstly, the person at position 2 is killed, then person at position 4 is killed, then person at position 1 is killed. Finally, the person at position 5 is killed. So the person at position 3 survives.

If $n = 7$ and $k = 3$, then the safe position is 4. The persons at positions 3, 6, 2, 7, 5, 1 are killed in order, and person at position 4 survives.

The problem has following recursive structure.

```
josephus(n, k) = (josephus(n - 1, k) + k - 1) % n + 1  
josephus(1, k) = 1
```

After the first person (k th from beginning) is killed, $n-1$ persons are left. So we call $josephus(n - 1, k)$ to get the position with $n-1$ persons. But the position returned by $josephus(n - 1, k)$ will consider the position starting from $k\%n + 1$. So, we must make adjustments to the position returned by $josephus(n - 1, k)$.

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

```
#include <stdio.h>
```

```
int josephus(int n, int k)
{
    if (n == 1)
        return 1;
    else
        /* The position returned by josephus(n - 1, k) is adjusted because
           recursive call josephus(n - 1, k) considers the original position
           k%n + 1 as position 1 */
        return (josephus(n - 1, k) + k - 1) % n + 1;
}
```

```
// Driver Program to test above function
int main()
{
    int n = 14;
    int k = 2;
    printf("The chosen place is %d", josephus(n, k));
    return 0;
}
```

Output:

```
The chosen place is 13
```

Time Complexity: $O(n)$

The problem can also be solved in $O(k \log n)$ time complexity which is a better solution for large n and small k . We will cover that solution in a separate post.

Source:

http://en.wikipedia.org/wiki/Josephus_problem

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

66. Backtracking | Set 7 (Sudoku)

Given a partially filled 9×9 2D array 'grid[9][9]', the goal is to assign digits (from 1 to 9) to the empty cells so that every row, column, and subgrid of size 3×3 contains exactly one instance of the digits from 1 to 9.

3		6	5	8	4		
5	2						
	8	7				3	1
		3		1		8	
9			8	6	3		5
	5			9		6	
1	3					2	5
						7	4
		5	2	6	3		

Naive Algorithm

The Naive Algorithm is to generate all possible configurations of numbers from 1 to 9 to fill the empty cells. Try every configuration one by one until the correct configuration is found.

Backtracking Algorithm

Like all other **Backtracking problems**, we can solve Sudoku by one by one assigning numbers to empty cells. Before assigning a number, we check whether it is safe to assign. We basically check that the same number is not present in current row, current column and current 3X3 subgrid. After checking for safety, we assign the number, and recursively check whether this assignment leads to a solution or not. If the assignment doesn't lead to a solution, then we try next number for current empty cell. And if none of number (1 to 9) lead to solution, we return false.

```
Find row, col of an unassigned cell
If there is none, return true
For digits from 1 to 9
    a) If there is no conflict for digit at row,col
        assign digit to row,col and recursively try fill in rest of grid
    b) If recursion successful, return true
    c) Else, remove digit and try another
If all digits have been tried and nothing worked, return false
```

Following is C++ implementation for Sudoku problem. It prints the completely filled grid as output.

```
// A Backtracking program in C++ to solve Sudoku problem
#include <stdio.h>

// UNASSIGNED is used for empty cells in sudoku grid
#define UNASSIGNED 0

// N is used for size of Sudoku grid. Size will be NxN
#define N 9

// This function finds an entry in grid that is still unassigned
bool FindUnassignedLocation(int grid[N][N], int &row, int &col);

// Checks whether it will be legal to assign num to the given row,col
bool isSafe(int grid[N][N], int row, int col, int num);

/* Takes a partially filled-in grid and attempts to assign values to
all unassigned locations in such a way to meet the requirements
```

```

for Sudoku solution (non-duplication across rows, columns, and boxes
bool SolveSudoku(int grid[N][N])
{
    int row, col;

    // If there is no unassigned location, we are done
    if (!FindUnassignedLocation(grid, row, col))
        return true; // success!

    // consider digits 1 to 9
    for (int num = 1; num <= 9; num++)
    {
        // if looks promising
        if (isSafe(grid, row, col, num))
        {
            // make tentative assignment
            grid[row][col] = num;

            // return, if success, yay!
            if (SolveSudoku(grid))
                return true;

            // failure, unmake & try again
            grid[row][col] = UNASSIGNED;
        }
    }
    return false; // this triggers backtracking
}

```

/* Searches the grid to find an entry that is still unassigned. If found, the reference parameters row, col will be set the location that is unassigned, and true is returned. If no unassigned entries remain, false is returned. */

```

bool FindUnassignedLocation(int grid[N][N], int &row, int &col)
{
    for (row = 0; row < N; row++)
        for (col = 0; col < N; col++)
            if (grid[row][col] == UNASSIGNED)
                return true;
    return false;
}

```

/* Returns a boolean which indicates whether any assigned entry in the specified row matches the given number. */

```

bool UsedInRow(int grid[N][N], int row, int num)
{
    for (int col = 0; col < N; col++)
        if (grid[row][col] == num)
            return true;
    return false;
}

```

/* Returns a boolean which indicates whether any assigned entry in the specified column matches the given number. */

```

bool UsedInCol(int grid[N][N], int col, int num)
{
    for (int row = 0; row < N; row++)
        if (grid[row][col] == num)
            return true;
    return false;
}

```

/* Returns a boolean which indicates whether any assigned entry

```

/* Returns a boolean which indicates whether any assigned entry
within the specified 3x3 box matches the given number. */
bool UsedInBox(int grid[N][N], int boxStartRow, int boxStartCol, int num)
{
    for (int row = 0; row < 3; row++)
        for (int col = 0; col < 3; col++)
            if (grid[row+boxStartRow][col+boxStartCol] == num)
                return true;
    return false;
}

/* Returns a boolean which indicates whether it will be legal to assign
num to the given row,col location. */
bool isSafe(int grid[N][N], int row, int col, int num)
{
    /* Check if 'num' is not already placed in current row,
    current column and current 3x3 box */
    return !UsedInRow(grid, row, num) &&
           !UsedInCol(grid, col, num) &&
           !UsedInBox(grid, row - row%3, col - col%3, num);
}

/* A utility function to print grid */
void printGrid(int grid[N][N])
{
    for (int row = 0; row < N; row++)
    {
        for (int col = 0; col < N; col++)
            printf("%2d", grid[row][col]);
        printf("\n");
    }
}

/* Driver Program to test above functions */
int main()
{
    // 0 means unassigned cells
    int grid[N][N] = {{3, 0, 6, 5, 0, 8, 4, 0, 0},
                      {5, 2, 0, 0, 0, 0, 0, 0, 0},
                      {0, 8, 7, 0, 0, 0, 0, 3, 1},
                      {0, 0, 3, 0, 1, 0, 0, 8, 0},
                      {9, 0, 0, 8, 6, 3, 0, 0, 5},
                      {0, 5, 0, 0, 9, 0, 6, 0, 0},
                      {1, 3, 0, 0, 0, 0, 2, 5, 0},
                      {0, 0, 0, 0, 0, 0, 0, 7, 4},
                      {0, 0, 5, 2, 0, 6, 3, 0, 0}};

    if (SolveSudoku(grid) == true)
        printGrid(grid);
    else
        printf("No solution exists");

    return 0;
}

```

Output:

```

3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
2 6 3 4 1 5 9 8 7

```

```

9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9

```

References:

<http://see.stanford.edu/materials/icspaces106b/H19-RecBacktrackExamples.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

67. Check whether a given point lies inside a triangle or not

Given three corner points of a triangle, and one more point P. Write a function to check whether P lies within the triangle or not.

For example, consider the following program, the function should return true for P(10, 15) and false for P'(30, 15)



Source: [Microsoft Interview Question](#)

Solution:

Let the coordinates of three corners be (x_1, y_1) , (x_2, y_2) and (x_3, y_3) . And coordinates of the given point P be (x, y)

- 1) Calculate area of the given triangle, i.e., area of the triangle ABC in the above diagram. $\text{Area } A = [x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)]/2$
- 2) Calculate area of the triangle PAB. We can use the same formula for this. Let this area be A_1 .
- 3) Calculate area of the triangle PBC. Let this area be A_2 .
- 4) Calculate area of the triangle PAC. Let this area be A_3 .
- 5) If P lies inside the triangle, then $A_1 + A_2 + A_3$ must be equal to A.

```

#include <stdio.h>
#include <stdlib.h>

/* A utility function to calculate area of triangle formed by (x1, y1)
(x2, y2) and (x3, y3) */
float area(int x1, int y1, int x2, int y2, int x3, int y3)
{
    return abs((x1*(y2-y3) + x2*(y3-y1)+ x3*(y1-y2))/2.0);
}

/* A function to check whether point P(x, y) lies inside the triangle
by A(x1, y1), B(x2, y2) and C(x3, y3) */
bool isInside(int x1, int y1, int x2, int y2, int x3, int y3, int x, int y)
{
    /* Calculate area of triangle ABC */
    float A = area (x1, y1, x2, y2, x3, y3);

    /* Calculate area of triangle PBC */
    float A1 = area (x, y, x2, y2, x3, y3);

    /* Calculate area of triangle PAC */
    float A2 = area (x1, y1, x, y, x3, y3);

    /* Calculate area of triangle PAB */
    float A3 = area (x1, y1, x2, y2, x, y);

    /* Check if sum of A1, A2 and A3 is same as A */
    return (A == A1 + A2 + A3);
}

/* Driver program to test above function */
int main()
{
    /* Let us check whether the point P(10, 15) lies inside the triangle
    formed by A(0, 0), B(20, 0) and C(10, 30) */
    if (isInside(0, 0, 20, 0, 10, 30, 10, 15))
        printf ("Inside");
    else
        printf ("Not Inside");

    return 0;
}

```

Ouput:

```
Inside
```

Exercise: Given coordinates of four corners of a rectangle, and a point P. Write a function to check whether P lies inside the given rectangle or not.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Given a number n , write a function that returns count of numbers from 1 to n that don't contain digit 3 in their decimal representation.

Examples:

Input: $n = 10$

Output: 9

Input: $n = 45$

Output: 31

// Numbers 3, 13, 23, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 43 contain digit 3.

Input: $n = 578$

Output: 385

Solution:

We can solve it recursively. Let $\text{count}(n)$ be the function that counts such numbers.

'msd' --> the most significant digit in n

'd' --> number of digits in n .

$\text{count}(n) = n$ if $n < 3$

$\text{count}(n) = n - 1$ if $3 \leq n < 10$

$\text{count}(n) = \text{count}(\text{msd}) * \text{count}(10^{(d-1)} - 1) +$
 $\text{count}(\text{msd}) +$
 $\text{count}(n \% (10^{(d-1)}))$
 if $n > 10$ and msd is not 3

$\text{count}(n) = \text{count}(\text{msd} * (10^{(d-1)} - 1)) - 1$
 if $n > 10$ and msd is 3

Let us understand the solution with $n = 578$.

$\text{count}(578) = 4 * \text{count}(99) + 4 + \text{count}(78)$

The middle term 4 is added to include numbers 100, 200, 400 and 500.

Let us take $n = 35$ as another example.

$\text{count}(35) = \text{count}(3 * 10 - 1) = \text{count}(29)$

```
#include <stdio.h>
```

```
/* returns count of numbers which are in range from 1 to n and don't contain
as a digit */
int count(int n)
{
    // Base cases (Assuming n is not negative)
    if (n < 3)
        return n;
    if (n >= 3 && n < 10)
        return n-1;

    // Calculate 10^(d-1) (10 raise to the power d-1) where d is
    // number of digits in n. po will be 100 for n = 578
    int po = 1;
    while (n/po > 9)
        po = po*10;

    // find the most significant digit (msd is 5 for 578)
    int msd = n/po;

    if (msd != 3)
        // For 578, total will be 4*count(10^2 - 1) + 4 + count(78)
        return count(msd)*count(po - 1) + count(msd) + count(n%po);
    else
        // For 35, total will be equal to count(29)
        return count(msd*po - 1);
}

// Driver program to test above function
int main()
{
    printf ("%d ", count(578));
    return 0;
}
```

Output:

385

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

69. Magic Square

A **magic square** of order n is an arrangement of n^2 numbers, usually distinct integers, in a square, such that the n numbers in all rows, all columns, and both diagonals sum to the same constant. A magic square contains the integers from 1 to n^2 .

The constant sum in every row, column and diagonal is called the **magic constant** or **magic sum**, M . The magic constant of a normal magic square depends only on n and

has the following value:

$$M = n(n^2+1)/2$$

For normal magic squares of order $n = 3, 4, 5, \dots$, the magic constants are: 15, 34, 65, 111, 175, 260, ...

In this post, we will discuss how programmatically we can generate a magic square of size n . Before we go further, consider the below examples:

Magic Square of size 3

```
-----  
  2   7   6  
  9   5   1  
  4   3   8
```

Sum in each row & each column = $3 \cdot (3^2+1)/2 = 15$

Magic Square of size 5

```
-----  
  9   3  22  16  15  
  2  21  20  14   8  
25  19  13   7   1  
18  12   6   5  24  
11  10   4  23  17
```

Sum in each row & each column = $5 \cdot (5^2+1)/2 = 65$

Magic Square of size 7

```
-----  
20  12   4  45  37  29  28  
11   3  44  36  35  27  19  
 2  43  42  34  26  18  10  
49  41  33  25  17   9   1  
40  32  24  16   8   7  48  
31  23  15  14   6  47  39  
22  21  13   5  46  38  30
```

Sum in each row & each column = $7 \cdot (7^2+1)/2 = 175$

Did you find any pattern in which the numbers are stored?

In any magic square, the first number i.e. 1 is stored at position $(n/2, n-1)$. Let this position be (i,j) . The next number is stored at position $(i-1, j+1)$ where we can consider each row & column as circular array i.e. they wrap around.

Three conditions hold:

1. The position of next number is calculated by decrementing row number of previous number by 1, and incrementing the column number of previous number by 1. At any time,

if the calculated row position becomes -1, it will wrap around to n-1. Similarly, if the calculated column position becomes n, it will wrap around to 0.

2. If the magic square already contains a number at the calculated position, calculated column position will be decremented by 2, and calculated row position will be incremented by 1.

3. If the calculated row position is -1 & calculated column position is n, the new position would be: (0, n-2).

Example:

Magic Square of size 3

```
-----  
2  7  6  
9  5  1  
4  3  8
```

Steps:

1. position of number 1 = $(3/2, 3-1) = (1, 2)$
2. position of number 2 = $(1-1, 2+1) = (0, 0)$
3. position of number 3 = $(0-1, 0+1) = (3-1, 1) = (2, 1)$
4. position of number 4 = $(2-1, 1+1) = (1, 2)$
Since, at this position, 1 is there. So, apply condition 2.
new position = $(1+1, 2-2) = (2, 0)$
5. position of number 5 = $(2-1, 0+1) = (1, 1)$
6. position of number 6 = $(1-1, 1+1) = (0, 2)$
7. position of number 7 = $(0-1, 2+1) = (-1, 3)$ // this is tricky, see condition 3
new position = $(0, 3-2) = (0, 1)$
8. position of number 8 = $(0-1, 1+1) = (-1, 2) = (2, 2)$ //wrap around
9. position of number 9 = $(2-1, 2+1) = (1, 3) = (1, 0)$ //wrap around

Based on the above approach, following is the working code:

```
#include<stdio.h>  
#include<string.h>  
  
// A function to generate odd sized magic squares  
void generateSquare(int n)  
{  
    int magicSquare[n][n];  
  
    // set all slots as 0  
    memset(magicSquare, 0, sizeof(magicSquare));  
  
    // Initialize position for 1  
    int i = n/2;  
    int j = n-1;  
  
    // One by one put all values in magic square  
    for (int num=1; num <= n*n; )  
    {  
        if (i== -1 && j==n) //3rd condition
```

```

    {
        j = n-2;
        i = 0;
    }
    else
    {
        //1st condition helper if next number goes to out of square
        if (j == n)
            j = 0;
        //1st condition helper if next number is goes to out of square
        if (i < 0)
            i=n-1;
    }
    if (magicSquare[i][j]) //2nd condition
    {
        j -= 2;
        i++;
        continue;
    }
    else
        magicSquare[i][j] = num++; //set number

    j++; i--; //1st condition
}

// print magic square
printf("The Magic Square for n=%d:\nSum of each row or column %d:\n",
        n, n*(n*n+1)/2);
for(i=0; i<n; i++)
{
    for(j=0; j<n; j++)
        printf("%3d ", magicSquare[i][j]);
    printf("\n");
}
}

```

```

// Driver program to test above function
int main()
{
    int n = 7; // Works only when n is odd
    generateSquare (n);
    return 0;
}

```

Output:

```

The Magic Square for n=7:
Sum of each row or column 175:

20  12   4  45  37  29  28
11   3  44  36  35  27  19
 2  43  42  34  26  18  10
49  41  33  25  17   9   1
40  32  24  16   8   7  48
31  23  15  14   6  47  39
22  21  13   5  46  38  30

```

NOTE: This approach works only for odd values of n .

References:

http://en.wikipedia.org/wiki/Magic_square

This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

70. Sieve of Eratosthenes

Given a number n , print all primes smaller than or equal to n . It is also given that n is a small number.

For example, if n is 10, the output should be "2, 3, 5, 7". If n is 20, the output should be "2, 3, 5, 7, 11, 13, 17, 19".

The sieve of Eratosthenes is one of the most efficient ways to find all primes smaller than n when n is smaller than 10 million or so (Ref [Wiki](#)).

Following is the algorithm to find all the prime numbers less than or equal to a given integer n by Eratosthenes' method:

1. Create a list of consecutive integers from 2 to n : (2, 3, 4, ..., n).
2. Initially, let p equal 2, the first prime number.
3. Starting from p , count up in increments of p and mark each of these numbers greater than p itself in the list. These numbers will be $2p$, $3p$, $4p$, etc.; note that some of them may have already been marked.
4. Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this number (which is the next prime), and repeat from step 3.

When the algorithm terminates, all the numbers in the list that are not marked are prime.

Following is C++ implementation of the above algorithm. In the following implementation, a boolean array `arr[]` of size n is used to mark multiples of prime numbers.

```

#include <stdio.h>
#include <string.h>

// marks all multiples of 'a' ( greater than 'a' but less than equal to
void markMultiples(bool arr[], int a, int n)
{
    int i = 2, num;
    while ( (num = i*a) <= n )
    {
        arr[ num-1 ] = 1; // minus 1 because index starts from 0.
        ++i;
    }
}

// A function to print all prime numbers smaller than n
void SieveOfEratosthenes(int n)
{
    // There are no prime numbers smaller than 2
    if (n >= 2)
    {
        // Create an array of size n and initialize all elements as 0
        bool arr[n];
        memset(arr, 0, sizeof(arr));

        /* Following property is maintained in the below for loop
        arr[i] == 0 means i + 1 is prime
        arr[i] == 1 means i + 1 is not prime */
        for (int i=1; i<n; ++i)
        {
            if ( arr[i] == 0 )
            {
                //(i+1) is prime, print it and mark its multiples
                printf("%d ", i+1);
                markMultiples(arr, i+1, n);
            }
        }
    }
}

// Driver Program to test above function
int main()
{
    int n = 30;
    printf("Following are the prime numbers below %d\n", n);
    SieveOfEratosthenes(n);
    return 0;
}

```

Output:

```

Following are the prime numbers below 30
2 3 5 7 11 13 17 19 23 29

```

References:

http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

This article is compiled by **Abhinav Priyadarshi** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above

71. Find day of the week for a given date

Write a function that calculates the day of the week for any particular date in the past or future. A typical application is to calculate the day of the week on which someone was born or some other special event occurred.

Following is a simple C function suggested by [Sakamoto, Lachman, Keith and Craver](#) to calculate day. The following function returns 0 for Sunday, 1 for Monday, etc.

```
/* A program to find day of a given date */
#include<stdio.h>
```

```
int dayofweek(int d, int m, int y)
{
    static int t[] = { 0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4 };
    y -= m < 3;
    return ( y + y/4 - y/100 + y/400 + t[m-1] + d ) % 7;
}
```

```
/* Driver function to test above function */
int main()
{
    int day = dayofweek(30, 8, 2010);
    printf ("%d", day);

    return 0;
}
```

Output: 1 (Monday)

See [this](#) for explanation of the above function.

References:

http://en.wikipedia.org/wiki/Determination_of_the_day_of_the_week

This article is compiled by **Dheeraj Jain** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

72. DFA based division

Deterministic Finite Automaton (DFA) can be used to check whether a number “num” is

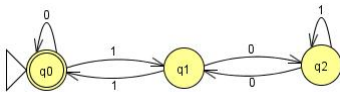
divisible by “k” or not. If the number is not divisible, remainder can also be obtained using DFA.

We consider the binary representation of ‘num’ and build a DFA with k states. The DFA has transition function for both 0 and 1. Once the DFA is built, we process ‘num’ over the DFA to get remainder.

Let us walk through an example. Suppose we want to check whether a given number ‘num’ is divisible by 3 or not. Any number can be written in the form: $\text{num} = 3 \cdot a + b$ where ‘a’ is the quotient and ‘b’ is the remainder.

For 3, there can be 3 states in DFA, each corresponding to remainder 0, 1 and 2. And each state can have two transitions corresponding 0 and 1 (considering the binary representation of given ‘num’).

The transition function $F(p, x) = q$ tells that on reading alphabet x, we move from state p to state q. Let us name the states as 0, 1 and 2. The initial state will always be 0. The final state indicates the remainder. If the final state is 0, the number is divisible.



In the above diagram, double circled state is final state.

1. When we are at state 0 and read 0, we remain at state 0.
2. When we are at state 0 and read 1, we move to state 1, why? The number so formed(1) in decimal gives remainder 1.
3. When we are at state 1 and read 0, we move to state 2, why? The number so formed(10) in decimal gives remainder 2.
4. When we are at state 1 and read 1, we move to state 0, why? The number so formed(11) in decimal gives remainder 0.
5. When we are at state 2 and read 0, we move to state 1, why? The number so formed(100) in decimal gives remainder 1.
6. When we are at state 2 and read 1, we remain at state 2, why? The number so formed(101) in decimal gives remainder 2.

The transition table looks like following:

state	0	1
0	0	1
1	2	0
2	1	2

Let us check whether 6 is divisible by 3?

Binary representation of 6 is 110

state = 0

1. state=0, we read 1, new state=1

2. state=1, we read 1, new state=0

3. state=0, we read 0, new state=0

Since the final state is 0, the number is divisible by 3.

Let us take another example number as 4

state=0

1. state=0, we read 1, new state=1

2. state=1, we read 0, new state=2

3. state=2, we read 0, new state=1

Since, the final state is not 0, the number is not divisible by 3. The remainder is 1.

Note that the final state gives the remainder.

We can extend the above solution for any value of k. For a value k, the states would be 0, 1, ..., k-1. How to calculate the transition if the decimal equivalent of the binary bits seen so far, crosses the range k? If we are at state p, we have read p (in decimal). Now we read 0, new read number becomes $2 \cdot p$. If we read 1, new read number becomes $2 \cdot p + 1$. The new state can be obtained by subtracting k from these values ($2p$ or $2p+1$) where $0 \leq p < k$.

Based on the above approach, following is the working code:

```
#include <stdio.h>
#include <stdlib.h>

// Function to build DFA for divisor k
void preprocess(int k, int Table[][2])
{
    int trans0, trans1;

    // The following loop calculates the two transitions for each state
    // starting from state 0
    for (int state=0; state<k; ++state)
    {
        // Calculate next state for bit 0
        trans0 = state<<1;
        Table[state][0] = (trans0 < k)? trans0: trans0-k;

        // Calculate next state for bit 1
        trans1 = (state<<1) + 1;
        Table[state][1] = (trans1 < k)? trans1: trans1-k;
    }
}

// A recursive utility function that takes a 'num' and DFA (transition
// table) as input and process 'num' bit by bit over DFA
void isDivisibleUtil(int num, int* state, int Table[][2])
{
    // process "num" bit by bit from MSB to LSB
    if (num != 0)
    {
        isDivisibleUtil(num>>1, state, Table);
        *state = Table[*state][num&1];
    }
}
```

```

// The main function that divides 'num' by k and returns the remainder
int isDivisible (int num, int k)
{
    // Allocate memory for transition table. The table will have k*2 e
    int (*Table)[2] = (int (*)[2])malloc(k*sizeof(*Table));

    // Fill the transition table
    preprocess(k, Table);

    // Process 'num' over DFA and get the remainder
    int state = 0;
    isDivisibleUtil(num, &state, Table);

    // Note that the final value of state is the remainder
    return state;
}

// Driver program to test above functions
int main()
{
    int num = 47; // Number to be divided
    int k = 5; // Divisor

    int remainder = isDivisible (num, k);

    if (remainder == 0)
        printf("Divisible\n");
    else
        printf("Not Divisible: Remainder is %d\n", remainder);

    return 0;
}

```

Output:

```
Not Divisible: Remainder is 2
```

DFA based division can be useful if we have a binary stream as input and we want to check for divisibility of the decimal value of stream at any time.

This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

73. Generate integer from 1 to 7 with equal probability

Given a function `foo()` that returns integers from 1 to 5 with equal probability, write a function that returns integers from 1 to 7 with equal probability using `foo()` only. Minimize the number of calls to `foo()` method. Also, use of any other library function is not allowed and no floating point arithmetic allowed.

Solution:

We know `foo()` returns integers from 1 to 5. How we can ensure that integers from 1 to 7 occur with equal probability?

If we somehow generate integers from 1 to a-multiple-of-7 (like 7, 14, 21, ...) with equal probability, we can use modulo division by 7 followed by adding 1 to get the numbers from 1 to 7 with equal probability.

We can generate from 1 to 21 with equal probability using the following expression.

```
5*foo() + foo() - 5
```

Let us see how above expression can be used.

1. For each value of first `foo()`, there can be 5 possible combinations for values of second `foo()`. So, there are total 25 combinations possible.
2. The range of values returned by the above equation is 1 to 25, each integer occurring exactly once.
3. If the value of the equation comes out to be less than 22, return modulo division by 7 followed by adding 1. Else, again call the method recursively. The probability of returning each integer thus becomes 1/7.

The below program shows that the expression returns each integer from 1 to 25 exactly once.

```
#include <stdio.h>

int main()
{
    int first, second;
    for ( first=1; first<=5; ++first )
        for ( second=1; second<=5; ++second )
            printf ("%d \n", 5*first + second - 5);
    return 0;
}
```

Output:

```
1
2
.
.
24
25
```

The below program depicts how we can use `foo()` to return 1 to 7 with equal probability.

```

#include <stdio.h>

int foo() // given method that returns 1 to 5 with equal probability
{
    // some code here
}

int my_rand() // returns 1 to 7 with equal probability
{
    int i;
    i = 5*foo() + foo() - 5;
    if (i < 22)
        return i%7 + 1;
    return my_rand();
}

int main()
{
    printf ("%d ", my_rand());
    return 0;
}

```

This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

74. Dynamic Programming | Set 19 (Word Wrap Problem)

Given a sequence of words, and a limit on the number of characters that can be put in one line (line width). Put line breaks in the given sequence such that the lines are printed neatly. Assume that the length of each word is smaller than the line width.

The word processors like MS Word do task of placing line breaks. The idea is to have balanced lines. In other words, not have few lines with lots of extra spaces and some lines with small amount of extra spaces.

The extra spaces includes spaces put at the end of every line except the last one. The problem is to minimize the following total cost.

Cost of a line = (Number of extra spaces in the line)³
 Total Cost = Sum of costs for all lines

For example, consider the following string and line width M = 15
 "Geeks for Geeks presents word wrap problem"

Following is the optimized arrangement of words in 3 lines
 Geeks for Geeks
 presents word
 wrap problem

The total extra spaces in line 1, line 2 and line 3 are 0, 2 and 3 respectively. So optimal value of total cost is $0 + 2^2 + 3^3 = 13$

Please note that the total cost function is not sum of extra spaces, but sum of cubes (or square is also used) of extra spaces. The idea behind this cost function is to balance the spaces among lines. For example, consider the following two arrangement of same set of words:

1) There are 3 lines. One line has 3 extra spaces and all other lines have 0 extra spaces. Total extra spaces = $3 + 0 + 0 = 3$. Total cost = $3^3 + 0^3 + 0^3 = 27$.

2) There are 3 lines. Each of the 3 lines has one extra space. Total extra spaces = $1 + 1 + 1 = 3$. Total cost = $1^3 + 1^3 + 1^3 = 3$.

Total extra spaces are 3 in both scenarios, but second arrangement should be preferred because extra spaces are balanced in all three lines. The cost function with cubic sum serves the purpose because the value of total cost in second scenario is less.

Method 1 (Greedy Solution)

The greedy solution is to place as many words as possible in the first line. Then do the same thing for the second line and so on until all words are placed. This solution gives optimal solution for many cases, but doesn't give optimal solution in all cases. For example, consider the following string "aaa bb cc dddd" and line width as 6. Greedy method will produce following output.

```
aaa bb
cc
dddd
```

Extra spaces in the above 3 lines are 0, 4 and 1 respectively. So total cost is $0 + 64 + 1 = 65$.

But the above solution is not the best solution. Following arrangement has more balanced spaces. Therefore less value of total cost function.

```
aaa
bb cc
dddd
```

Extra spaces in the above 3 lines are 3, 1 and 1 respectively. So total cost is $27 + 1 + 1 = 29$.

Despite being sub-optimal in some cases, the greedy approach is used by many word processors like MS Word and OpenOffice.org Writer.

Method 2 (Dynamic Programming)

The following Dynamic approach strictly follows the algorithm given in solution of

Cormen book. First we compute costs of all possible lines in a 2D table `lc[][]`. The value `lc[i][j]` indicates the cost to put words from `i` to `j` in a single line where `i` and `j` are indexes of words in the input sequences. If a sequence of words from `i` to `j` cannot fit in a single line, then `lc[i][j]` is considered infinite (to avoid it from being a part of the solution). Once we have the `lc[][]` table constructed, we can calculate total cost using following recursive formula. In the following formula, `C[j]` is the optimized total cost for arranging words from 1 to `j`.

$$c[j] = \begin{cases} 0 & \text{if } j = 0, \\ \min_{1 \leq i \leq j} (c[i-1] + lc[i, j]) & \text{if } j > 0. \end{cases}$$

The above recursion has **overlapping subproblem property**. For example, the solution of subproblem `c(2)` is used by `c(3)`, `C(4)` and so on. So Dynamic Programming is used to store the results of subproblems. The array `c[]` can be computed from left to right, since each value depends only on earlier values.

To print the output, we keep track of what words go on what lines, we can keep a parallel `p` array that points to where each `c` value came from. The last line starts at word `p[n]` and goes through word `n`. The previous line starts at word `p[p[n]]` and goes through word `p[n] - 1`, etc. The function `printSolution()` uses `p[]` to print the solution.

In the below program, input is an array `l[]` that represents lengths of words in a sequence. The value `l[i]` indicates length of the `i`th word (`i` starts from 1) in the input sequence.

```
// A Dynamic programming solution for Word Wrap Problem
```

```
#include <limits.h>
```

```
#include <stdio.h>
```

```
#define INF INT_MAX
```

```
// A utility function to print the solution
```

```
int printSolution (int p[], int n);
```

```
// l[] represents lengths of different words in input sequence. For ex
// l[] = {3, 2, 2, 5} is for a sentence like "aaa bb cc dddd". n is
// l[] and M is line width (maximum no. of characters that can fit in
void solveWordWrap (int l[], int n, int M)
```

```
{
```

```
    // For simplicity, 1 extra space is used in all below arrays
```

```
    // extras[i][j] will have number of extra spaces if words from i
    // to j are put in a single line
```

```
    int extras[n+1][n+1];
```

```
    // lc[i][j] will have cost of a line which has words from
    // i to j
```

```
    int lc[n+1][n+1];
```

```
    // c[i] will have total cost of optimal arrangement of words
    // from 1 to i
```

```
    int c[n+1];
```

```
    // p[] is used to print the solution.
```

```
    int p[n+1];
```

```
    int i, j;
```

```
    // calculate extra spaces in a single line. The value extras[i][j]
```

```

// calculate extra spaces in a single line. The value extra[i][j]
// indicates extra spaces if words from word number i to j are
// placed in a single line
for (i = 1; i <= n; i++)
{
    extras[i][i] = M - l[i-1];
    for (j = i+1; j <= n; j++)
        extras[i][j] = extras[i][j-1] - l[j-1] - 1;
}

// Calculate line cost corresponding to the above calculated extra
// spaces. The value lc[i][j] indicates cost of putting words from
// word number i to j in a single line
for (i = 1; i <= n; i++)
{
    for (j = i; j <= n; j++)
    {
        if (extras[i][j] < 0)
            lc[i][j] = INF;
        else if (j == n && extras[i][j] >= 0)
            lc[i][j] = 0;
        else
            lc[i][j] = extras[i][j]*extras[i][j];
    }
}

// Calculate minimum cost and find minimum cost arrangement.
// The value c[j] indicates optimized cost to arrange words
// from word number 1 to j.
c[0] = 0;
for (j = 1; j <= n; j++)
{
    c[j] = INF;
    for (i = 1; i <= j; i++)
    {
        if (c[i-1] != INF && lc[i][j] != INF && (c[i-1] + lc[i][j]) < c[j])
        {
            c[j] = c[i-1] + lc[i][j];
            p[j] = i;
        }
    }
}

printSolution(p, n);
}

```

```

int printSolution (int p[], int n)
{
    int k;
    if (p[n] == 1)
        k = 1;
    else
        k = printSolution (p, p[n]-1) + 1;
    printf ("Line number %d: From word no. %d to %d \n", k, p[n], n);
    return k;
}

```

```

// Driver program to test above functions
int main()
{
    int l[] = {3, 2, 2, 5};
    int n = sizeof(l)/sizeof(l[0]);
    int M = 6;
}

```

```
    solveWordWrap (1, n, M);  
    return 0;  
}
```

Output:

```
Line number 1: From word no. 1 to 1  
Line number 2: From word no. 2 to 3  
Line number 3: From word no. 4 to 4
```

Time Complexity: $O(n^2)$

Auxiliary Space: $O(n^2)$ The auxiliary space used in the above program can be optimized to $O(n)$ (See the reference 2 for details)

References:

http://en.wikipedia.org/wiki/Word_wrap

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

75. Given a number, find the next smallest palindrome

Given a number, find the next smallest palindrome larger than this number. For example, if the input number is "2 3 5 4 5", the output should be "2 3 6 3 2". And if the input number is "9 9 9", the output should be "1 0 0 1".

The input is assumed to be an array. Every entry in array represents a digit in input number. Let the array be 'num[]' and size of array be 'n'

There can be three different types of inputs that need to be handled separately.

- 1) The input number is palindrome and has all 9s. For example "9 9 9". Output should be "1 0 0 1"
- 2) The input number is not palindrome. For example "1 2 3 4". Output should be "1 3 3 1"
- 3) The input number is palindrome and doesn't have all 9s. For example "1 2 2 1". Output should be "1 3 3 1".

Solution for input type 1 is easy. The output contains $n + 1$ digits where the corner digits are 1, and all digits between corner digits are 0.

Now let us first talk about input type 2 and 3. How to convert a given number to a greater palindrome? To understand the solution, let us first define the following two terms:

Left Side: The left half of given number. Left side of "1 2 3 4 5 6" is "1 2 3" and left side of "1 2 3 4 5" is "1 2"

Right Side: The right half of given number. Right side of "1 2 3 4 5 6" is "4 5 6" and right

side of "1 2 3 4 5" is "4 5"

To convert to palindrome, we can either take the mirror of its left side or take mirror of its right side. However, if we take the mirror of the right side, then the palindrome so formed is not guaranteed to be next larger palindrome. So, we must take the mirror of left side and copy it to right side. But there are some cases that must be handled in different ways. See the following steps.

We will start with two indices i and j . i pointing to the two middle elements (or pointing to two elements around the middle element in case of n being odd). We one by one move i and j away from each other.

Step 1. Initially, ignore the part of left side which is same as the corresponding part of right side. For example, if the number is "8 3 **4 2 2 4** 6 9", we ignore the middle four elements. i now points to element 3 and j now points to element 6.

Step 2. After step 1, following cases arise:

Case 1: Indices i & j cross the boundary.

This case occurs when the input number is palindrome. In this case, we just add 1 to the middle digit (or digits in case n is even) propagate the carry towards MSB digit of left side and simultaneously copy mirror of the left side to the right side.

For example, if the given number is "1 2 9 2 1", we increment 9 to 10 and propagate the carry. So the number becomes "1 3 0 3 1"

Case 2: There are digits left between left side and right side which are not same. So, we just mirror the left side to the right side & try to minimize the number formed to guarantee the next smallest palindrome.

In this case, there can be **two sub-cases**.

2.1) Copying the left side to the right side is sufficient, we don't need to increment any digits and the result is just mirror of left side. Following are some examples of this sub-case.

Next palindrome for "7 **8** 3 3 2 2" is "7 8 3 3 8 7"

Next palindrome for "1 2 **5** 3 2 2" is "1 2 5 5 2 1"

Next palindrome for "1 4 **5** 8 7 6 7 8 3 2 2" is "1 4 5 8 7 6 7 8 5 4 1"

How do we check for this sub-case? All we need to check is the digit just after the ignored part in step 1. This digit is highlighted in above examples. If this digit is greater than the corresponding digit in right side digit, then copying the left side to the right side is sufficient and we don't need to do anything else.

2.2) Copying the left side to the right side is NOT sufficient. This happens when the above defined digit of left side is smaller. Following are some examples of this case.

Next palindrome for "7 **1** 3 3 2 2" is "7 1 4 4 1 7"

Next palindrome for "1 2 **3** 4 6 2 8" is "1 2 3 5 3 2 1"

Next palindrome for "9 4 **1** 8 7 9 7 8 3 2 2" is "9 4 1 8 8 0 8 8 1 4 9"

We handle this subcase like Case 1. We just add 1 to the middle digit (or digits in case n is even) propagate the carry towards MSB digit of left side and simultaneously copy

mirror of the left side to the right side.

```
#include <stdio.h>

// A utility function to print an array
void printArray (int arr[], int n);

// A utility function to check if num has all 9s
int AreAll9s (int num[], int n );

// Returns next palindrome of a given number num[].
// This function is for input type 2 and 3
void generateNextPalindromeUtil (int num[], int n )
{
    // find the index of mid digit
    int mid = n/2;

    // A bool variable to check if copy of left side to right is sufficient
    bool leftsmaller = false;

    // end of left side is always 'mid -1'
    int i = mid - 1;

    // Beginning of right side depends if n is odd or even
    int j = (n % 2)? mid + 1 : mid;

    // Initially, ignore the middle same digits
    while (i >= 0 && num[i] == num[j])
        i--,j++;

    // Find if the middle digit(s) need to be incremented or not (or copy of
    // side is not sufficient)
    if ( i < 0 || num[i] < num[j])
        leftsmaller = true;

    // Copy the mirror of left to right
    while (i >= 0)
    {
        num[j] = num[i];
        j++;
        i--;
    }

    // Handle the case where middle digit(s) must be incremented.
    // This part of code is for CASE 1 and CASE 2.2
    if (leftsmaller == true)
    {
        int carry = 1;
        i = mid - 1;

        // If there are odd digits, then increment
        // the middle digit and store the carry
        if (n%2 == 1)
        {
            num[mid] += carry;
            carry = num[mid] / 10;
            num[mid] %= 10;
            j = mid + 1;
        }
        else
            j = mid;

        while (i >= 0 && carry > 0)
        {
            num[i] += carry;
            carry = num[i] / 10;
            num[i] %= 10;
            i--;
        }
    }
}
```



```

        // Add 1 to the rightmost digit of the left side, propagate the
        // towards MSB digit and simultaneously copying mirror of the
        // to the right side.
        while (i >= 0)
        {
            num[i] += carry;
            carry = num[i] / 10;
            num[i] %= 10;
            num[j++] = num[i--]; // copy mirror to right
        }
    }
}

// The function that prints next palindrome of a given number num[]
// with n digits.
void generateNextPalindrome( int num[], int n )
{
    int i;

    printf("\nNext palindrome is:\n");

    // Input type 1: All the digits are 9, simply o/p 1
    // followed by n-1 0's followed by 1.
    if( AreAll9s( num, n ) )
    {
        printf( "1 " );
        for( i = 1; i < n; i++ )
            printf( "0 " );
        printf( "1" );
    }

    // Input type 2 and 3
    else
    {
        generateNextPalindromeUtil ( num, n );

        // print the result
        printArray (num, n);
    }
}

// A utility function to check if num has all 9s
int AreAll9s( int* num, int n )
{
    int i;
    for( i = 0; i < n; ++i )
        if( num[i] != 9 )
            return 0;
    return 1;
}

/* Utility that prints out an array on a line */
void printArray(int arr[], int n)
{
    int i;
    for (i=0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver Program to test above function
int main()
{

```

```

{
    int num[] = {9, 4, 1, 8, 7, 9, 7, 8, 3, 2, 2};

    int n = sizeof (num)/ sizeof(num[0]);

    generateNextPalindrome( num, n );

    return 0;
}

```

Output:

```

Next palindrome is:
9 4 1 8 8 0 8 8 1 4 9

```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

76. Dynamic Programming | Set 21 (Variations of LIS)

We have discussed Dynamic Programming solution for Longest Increasing Subsequence problem in [this](#) post and a $O(n \log n)$ solution in [this](#) post. Following are commonly asked variations of the standard [LIS problem](#).

1. Building Bridges: Consider a 2-D map with a horizontal river passing through its center. There are n cities on the southern bank with x -coordinates $a(1) \dots a(n)$ and n cities on the northern bank with x -coordinates $b(1) \dots b(n)$. 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 i on the northern bank to city i on the southern bank.

```

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----->

```

Source: [Dynamic Programming Practice Problems](#). The link also has well explained solution for the problem.

2. Maximum Sum Increasing Subsequence: Given an array of n positive integers. Write a program to find the maximum sum subsequence of the given array such that the

integers in the subsequence are sorted in increasing order. For example, if input is {1, 101, 2, 3, 100, 4, 5}, then output should be {1, 2, 3, 100}. The solution to this problem has been published [here](#).

3. The Longest Chain You are given pairs of numbers. In a pair, the first number is smaller with respect to the second number. Suppose you have two sets (a, b) and (c, d), the second set can follow the first set if $b < c$. So you can form a long chain in the similar fashion. Find the longest chain which can be formed. The solution to this problem has been published [here](#).

4. Box Stacking You are given a set of n types of rectangular 3-D boxes, where the i^{th} box has height $h(i)$, width $w(i)$ and depth $d(i)$ (all real numbers). 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.

Source: [Dynamic Programming Practice Problems](#). The link also has well explained solution for the problem.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

77. Make a fair coin from a biased coin

You are given a function `foo()` that represents a biased coin. When `foo()` is called, it returns 0 with 60% probability, and 1 with 40% probability. Write a new function that returns 0 and 1 with 50% probability each. Your function should use only `foo()`, no other library method.

Solution:

We know `foo()` returns 0 with 60% probability. How can we ensure that 0 and 1 are returned with 50% probability?

The solution is similar to [this](#) post. If we can somehow get two cases with equal probability, then we are done. We call `foo()` two times. Both calls will return 0 with 60% probability. So the two pairs (0, 1) and (1, 0) will be generated with equal probability from two calls of `foo()`. Let us see how.

(0, 1): The probability to get 0 followed by 1 from two calls of `foo()` = $0.6 * 0.4 = 0.24$

(1, 0): The probability to get 1 followed by 0 from two calls of `foo()` = $0.4 * 0.6 = 0.24$

So the two cases appear with equal probability. The idea is to return consider only the above two cases, return 0 in one case, return 1 in other case. For other cases [(0, 0) and (1, 1)], recur until you end up in any of the above two cases.

The below program depicts how we can use `foo()` to return 0 and 1 with equal probability.

```
#include <stdio.h>

int foo() // given method that returns 0 with 60% probability and 1 wi
{
    // some code here
}

// returns both 0 and 1 with 50% probability
int my_fun()
{
    int val1 = foo();
    int val2 = foo();
    if (val1 == 0 && val2 == 1)
        return 0; // Will reach here with 0.24 probability
    if (val1 == 1 && val2 == 0)
        return 1; // Will reach here with 0.24 probability
    return my_fun(); // will reach here with (1 - 0.24 - 0.24) probab.
}

int main()
{
    printf ("%d ", my_fun());
    return 0;
}
```

References:

http://en.wikipedia.org/wiki/Fair_coin#Fair_results_from_a_biased_coin

This article is compiled by **Shashank Sinha** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

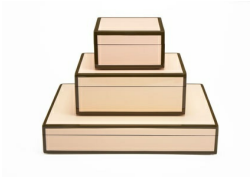
If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

78. Dynamic Programming | Set 22 (Box Stacking Problem)

You are given a set of n types of rectangular 3-D boxes, where the i^{th} box has height $h(i)$, width $w(i)$ and depth $d(i)$ (all real numbers). 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.

Source: <http://people.csail.mit.edu/bdean/6.046/dp/>. The link also has video for explanation of solution.



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. For example, if there is a box with dimensions $\{1 \times 2 \times 3\}$ where 1 is height, 2×3 is base, then there can be three possibilities, $\{1 \times 2 \times 3\}$, $\{2 \times 1 \times 3\}$ and $\{3 \times 1 \times 2\}$.
- 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

$MSH(i) = \{ \text{Max} (MSH(j)) + \text{height}(i) \}$ where $j < i$ and $\text{width}(j) > \text{width}(i)$ and $\text{depth}(j) > \text{depth}(i)$.

If there is no such j then $MSH(i) = \text{height}(i)$

4) To get overall maximum height, we return $\text{max}(MSH(i))$ where $0 < i < n$

Following is C++ implementation of the above solution.

```
/* Dynamic Programming implementation of Box Stacking problem */
#include<stdio.h>
#include<stdlib.h>

/* Representation of a box */
struct Box
{
```

```

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

// A utility function to get minimum of two integers
int min (int x, int y)
{ return (x < y)? x : y; }

// A utility function to get maximum of two integers
int max (int x, int y)
{ return (x > y)? x : y; }

/* Following function is needed for library function qsort(). We
   use qsort() to sort boxes in decreasing order of base area.
   Refer following link for help of qsort() and compare()
   http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */
int compare (const void *a, const void * b)
{
    return ( (*(Box *)b).d * (*(Box *)b).w ) -
           ( (*(Box *)a).d * (*(Box *)a).w );
}

/* Returns the height of the tallest stack that can be formed with given
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] = arr[i];
        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 decreasing order, using library
       function for quick sort */
    qsort (rot, n, sizeof(rot[0]), compare);

    // Uncomment following two lines to print all rotations
    // for (int i = 0; i < n; i++ )
    //     printf("%d x %d x %d\n", rot[i].h, rot[i].w, rot[i].d);

    /* 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;
}

/* Driver program to test above function */
int main()
{
    Box arr[] = { {4, 6, 7}, {1, 2, 3}, {4, 5, 6}, {10, 12, 32} };
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("The maximum possible height of stack is %d\n",
        maxStackHeight (arr, n) );

    return 0;
}

```

Output:

```
The maximum possible height of stack is 60
```

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

Time Complexity: $O(n^2)$

Auxiliary Space: $O(n)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

79. Find the k most frequent words from a file

Given a book of words. Assume you have enough main memory to accommodate all words. design a data structure to find top K maximum occurring words. The data structure should be dynamic so that new words can be added.

A simple solution is to **use Hashing**. Hash all words one by one in a hash table. If a word is already present, then increment its count. Finally, traverse through the hash table and return the k words with maximum counts.

We can **use Trie and Min Heap** to get the k most frequent words efficiently. The idea is to use Trie for searching existing words adding new words efficiently. Trie also stores count of occurrences of words. A Min Heap of size k is used to keep track of k most frequent words at any point of time(Use of Min Heap is same as we used it to find k largest elements in [this](#) post).

Trie and Min Heap are linked with each other by storing an additional field in Trie 'indexMinHeap' and a pointer 'trNode' in Min Heap. The value of 'indexMinHeap' is maintained as -1 for the words which are currently not in Min Heap (or currently not among the top k frequent words). For the words which are present in Min Heap, 'indexMinHeap' contains, index of the word in Min Heap. The pointer 'trNode' in Min Heap points to the leaf node corresponding to the word in Trie.

Following is the complete process to print k most frequent words from a file.

Read all words one by one. For every word, insert it into Trie. Increase the counter of the word, if already exists. Now, we need to insert this word in min heap also. For insertion in min heap, 3 cases arise:

1. The word is already present. We just increase the corresponding frequency value in min heap and call minHeapify() for the index obtained by "indexMinHeap" field in Trie. When the min heap nodes are being swapped, we change the corresponding minHeapIndex in the Trie. Remember each node of the min heap is also having pointer to Trie leaf node.

2. The minHeap is not full. we will insert the new word into min heap & update the root node in the min heap node & min heap index in Trie leaf node. Now, call buildMinHeap().

3. The min heap is full. Two sub-cases arise.

....3.1 The frequency of the new word inserted is less than the frequency of the word stored in the head of min heap. Do nothing.

....3.2 The frequency of the new word inserted is greater than the frequency of the word stored in the head of min heap. Replace & update the fields. Make sure to update the corresponding min heap index of the "word to be replaced" in Trie with -1 as the word is no longer in min heap.

4. Finally, Min Heap will have the k most frequent words of all words present in given file. So we just need to print all words present in Min Heap.

```
// A program to find k most frequent words in a file
#include <stdio.h>
#include <string.h>
#include <ctype.h>

# define MAX_CHARS 26
# define MAX_WORD_SIZE 30

// A Trie node
struct TrieNode
{
    bool isEnd; // indicates end of word
    unsigned frequency; // the number of occurrences of a word
    int indexMinHeap; // the index of the word in minHeap
    TrieNode* child[MAX_CHARS]; // represents 26 slots each for 'a' to
};

// A Min Heap node
struct MinHeapNode
{
    TrieNode* root; // indicates the leaf node of TRIE
    unsigned frequency; // number of occurrences
    char* word; // the actual word stored
};

// A Min Heap
struct MinHeap
{
    unsigned capacity; // the total size a min heap
    int count; // indicates the number of slots filled.
    MinHeapNode* array; // represents the collection of minHeapNodes
};

// A utility function to create a new Trie node
TrieNode* newTrieNode()
{
    // Allocate memory for Trie Node
    TrieNode* trieNode = new TrieNode;

    // Initialize values for new node
    trieNode->isEnd = 0;
    trieNode->frequency = 0;
    trieNode->indexMinHeap = -1;
```

```

    for( int i = 0; i < MAX_CHARS; ++i )
        trieNode->child[i] = NULL;

    return trieNode;
}

// A utility function to create a Min Heap of given capacity
MinHeap* createMinHeap( int capacity )
{
    MinHeap* minHeap = new MinHeap;

    minHeap->capacity = capacity;
    minHeap->count = 0;

    // Allocate memory for array of min heap nodes
    minHeap->array = new MinHeapNode [ minHeap->capacity ];

    return minHeap;
}

// A utility function to swap two min heap nodes. This function
// is needed in minHeapify
void swapMinHeapNodes ( MinHeapNode* a, MinHeapNode* b )
{
    MinHeapNode temp = *a;
    *a = *b;
    *b = temp;
}

// This is the standard minHeapify function. It does one thing extra.
// It updates the minHeapIndex in Trie when two nodes are swapped in
// in min heap
void minHeapify( MinHeap* minHeap, int idx )
{
    int left, right, smallest;

    left = 2 * idx + 1;
    right = 2 * idx + 2;
    smallest = idx;
    if ( left < minHeap->count &&
        minHeap->array[ left ]. frequency <
        minHeap->array[ smallest ]. frequency
    )
        smallest = left;

    if ( right < minHeap->count &&
        minHeap->array[ right ]. frequency <
        minHeap->array[ smallest ]. frequency
    )
        smallest = right;

    if( smallest != idx )
    {
        // Update the corresponding index in Trie node.
        minHeap->array[ smallest ]. root->indexMinHeap = idx;
        minHeap->array[ idx ]. root->indexMinHeap = smallest;

        // Swap nodes in min heap
        swapMinHeapNodes ( &minHeap->array[ smallest ], &minHeap->array
                           [ idx ] );

        minHeapify( minHeap, smallest );
    }
}

```

```

}

// A standard function to build a heap
void buildMinHeap( MinHeap* minHeap )
{
    int n, i;
    n = minHeap->count - 1;

    for( i = ( n - 1 ) / 2; i >= 0; --i )
        minHeapify( minHeap, i );
}

// Inserts a word to heap, the function handles the 3 cases explained :
void insertInMinHeap( MinHeap* minHeap, TrieNode** root, const char* word )
{
    // Case 1: the word is already present in minHeap
    if( (*root)->indexMinHeap != -1 )
    {
        ++( minHeap->array[ (*root)->indexMinHeap ]. frequency );

        // percolate down
        minHeapify( minHeap, (*root)->indexMinHeap );
    }

    // Case 2: Word is not present and heap is not full
    else if( minHeap->count < minHeap->capacity )
    {
        int count = minHeap->count;
        minHeap->array[ count ]. frequency = (*root)->frequency;
        minHeap->array[ count ]. word = new char [strlen( word ) + 1];
        strcpy( minHeap->array[ count ]. word, word );

        minHeap->array[ count ]. root = *root;
        (*root)->indexMinHeap = minHeap->count;

        ++( minHeap->count );
        buildMinHeap( minHeap );
    }

    // Case 3: Word is not present and heap is full. And frequency of root
    // is more than root. The root is the least frequent word in heap,
    // replace root with new word
    else if ( (*root)->frequency > minHeap->array[0]. frequency )
    {
        minHeap->array[ 0 ]. root->indexMinHeap = -1;
        minHeap->array[ 0 ]. root = *root;
        minHeap->array[ 0 ]. root->indexMinHeap = 0;
        minHeap->array[ 0 ]. frequency = (*root)->frequency;

        // delete previously allocated memory and
        delete [] minHeap->array[ 0 ]. word;
        minHeap->array[ 0 ]. word = new char [strlen( word ) + 1];
        strcpy( minHeap->array[ 0 ]. word, word );

        minHeapify ( minHeap, 0 );
    }
}

// Inserts a new word to both Trie and Heap
void insertUtil ( TrieNode** root, MinHeap* minHeap,
                 const char* word, const char* dupWord )
{

```

```

// Base Case
if ( *root == NULL )
    *root = newTrieNode();

// There are still more characters in word
if ( *word != '\0' )
    insertUtil ( &((*root)->child[ tolower( *word ) - 97 ]),
                minHeap, word + 1, dupWord );
else // The complete word is processed
{
    // word is already present, increase the frequency
    if ( (*root)->isEnd )
        ++( (*root)->frequency );
    else
    {
        (*root)->isEnd = 1;
        (*root)->frequency = 1;
    }

    // Insert in min heap also
    insertInMinHeap( minHeap, root, dupWord );
}
}

// add a word to Trie & min heap. A wrapper over the insertUtil
void insertTrieAndHeap(const char *word, TrieNode** root, MinHeap* minH)
{
    insertUtil( root, minHeap, word, word );
}

// A utility function to show results, The min heap
// contains k most frequent words so far, at any time
void displayMinHeap( MinHeap* minHeap )
{
    int i;

    // print top K word with frequency
    for( i = 0; i < minHeap->count; ++i )
    {
        printf( "%s : %d\n", minHeap->array[i].word,
                minHeap->array[i].frequency );
    }
}

// The main function that takes a file as input, add words to heap
// and Trie, finally shows result from heap
void printKMostFreq( FILE* fp, int k )
{
    // Create a Min Heap of Size k
    MinHeap* minHeap = createMinHeap( k );

    // Create an empty Trie
    TrieNode* root = NULL;

    // A buffer to store one word at a time
    char buffer[MAX_WORD_SIZE];

    // Read words one by one from file. Insert the word in Trie and M
    while( fscanf( fp, "%s", buffer ) != EOF )
        insertTrieAndHeap(buffer, &root, minHeap);
}

```

```

// The Min Heap will have the k most frequent words, so print Min Heap
displayMinHeap( minHeap );
}

// Driver program to test above functions
int main()
{
    int k = 5;
    FILE *fp = fopen ("file.txt", "r");
    if (fp == NULL)
        printf ("File doesn't exist ");
    else
        printKMostFreq (fp, k);
    return 0;
}

```

Output:

```

your : 3
well : 3
and : 4
to : 4
Geeks : 6

```

The above output is for a file with following content.

```

Welcome to the world of Geeks

This portal has been created to provide well written well thought and well explained
solutions for selected questions If you like Geeks for Geeks and would like to contribute
here is your chance You can write article and mail your article to contribute at
geeksforgeeks.org See your article appearing on the Geeks for Geeks main page and help
thousands of other Geeks

```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Given a book of words. Assume you have enough main memory to accommodate all words. design a data structure to find top K maximum occurring words. The data structure should be dynamic so that new words can be added.

A simple solution is to **use Hashing**. Hash all words one by one in a hash table. If a word is already present, then increment its count. Finally, traverse through the hash table and return the k words with maximum counts.

We can **use Trie and Min Heap** to get the k most frequent words efficiently. The idea is to use Trie for searching existing words adding new words efficiently. Trie also stores

count of occurrences of words. A Min Heap of size k is used to keep track of k most frequent words at any point of time(Use of Min Heap is same as we used it to find k largest elements in [this](#) post).

Trie and Min Heap are linked with each other by storing an additional field in Trie 'indexMinHeap' and a pointer 'trNode' in Min Heap. The value of 'indexMinHeap' is maintained as -1 for the words which are currently not in Min Heap (or currently not among the top k frequent words). For the words which are present in Min Heap, 'indexMinHeap' contains, index of the word in Min Heap. The pointer 'trNode' in Min Heap points to the leaf node corresponding to the word in Trie.

Following is the complete process to print k most frequent words from a file.

Read all words one by one. For every word, insert it into Trie. Increase the counter of the word, if already exists. Now, we need to insert this word in min heap also. For insertion in min heap, 3 cases arise:

1. The word is already present. We just increase the corresponding frequency value in min heap and call minHeapify() for the index obtained by "indexMinHeap" field in Trie. When the min heap nodes are being swapped, we change the corresponding minHeapIndex in the Trie. Remember each node of the min heap is also having pointer to Trie leaf node.

2. The minHeap is not full. we will insert the new word into min heap & update the root node in the min heap node & min heap index in Trie leaf node. Now, call buildMinHeap().

3. The min heap is full. Two sub-cases arise.

....3.1 The frequency of the new word inserted is less than the frequency of the word stored in the head of min heap. Do nothing.

....3.2 The frequency of the new word inserted is greater than the frequency of the word stored in the head of min heap. Replace & update the fields. Make sure to update the corresponding min heap index of the "word to be replaced" in Trie with -1 as the word is no longer in min heap.

4. Finally, Min Heap will have the k most frequent words of all words present in given file. So we just need to print all words present in Min Heap.

// A program to find k most frequent words in a file

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
```

```
# define MAX_CHARS 26
# define MAX_WORD_SIZE 30
```

// A Trie node

```
struct TrieNode
{
```

```
    bool isEnd; // indicates end of word
    unsigned frequency; // the number of occurrences of a word
    int indexMinHeap; // the index of the word in minHeap
    TrieNode* child[MAX_CHARS]; // represents 26 slots each for 'a' to
```

```

    trieNode->child[MAX_CHARS], // represents 26 slots each for a-z
};

// A Min Heap node
struct MinHeapNode
{
    TrieNode* root; // indicates the leaf node of TRIE
    unsigned frequency; // number of occurrences
    char* word; // the actual word stored
};

// A Min Heap
struct MinHeap
{
    unsigned capacity; // the total size a min heap
    int count; // indicates the number of slots filled.
    MinHeapNode* array; // represents the collection of minHeapNodes
};

// A utility function to create a new Trie node
TrieNode* newTrieNode()
{
    // Allocate memory for Trie Node
    TrieNode* trieNode = new TrieNode;

    // Initialize values for new node
    trieNode->isEnd = 0;
    trieNode->frequency = 0;
    trieNode->indexMinHeap = -1;
    for( int i = 0; i < MAX_CHARS; ++i )
        trieNode->child[i] = NULL;

    return trieNode;
}

// A utility function to create a Min Heap of given capacity
MinHeap* createMinHeap( int capacity )
{
    MinHeap* minHeap = new MinHeap;

    minHeap->capacity = capacity;
    minHeap->count = 0;

    // Allocate memory for array of min heap nodes
    minHeap->array = new MinHeapNode [ minHeap->capacity ];

    return minHeap;
}

// A utility function to swap two min heap nodes. This function
// is needed in minHeapify
void swapMinHeapNodes ( MinHeapNode* a, MinHeapNode* b )
{
    MinHeapNode temp = *a;
    *a = *b;
    *b = temp;
}

// This is the standard minHeapify function. It does one thing extra.
// It updates the minHapIndex in Trie when two nodes are swapped in
// in min heap
void minHeapify( MinHeap* minHeap, int idx )
{

```

```

    int left, right, smallest;

    left = 2 * idx + 1;
    right = 2 * idx + 2;
    smallest = idx;
    if ( left < minHeap->count &&
        minHeap->array[ left ]. frequency <
        minHeap->array[ smallest ]. frequency
    )
        smallest = left;

    if ( right < minHeap->count &&
        minHeap->array[ right ]. frequency <
        minHeap->array[ smallest ]. frequency
    )
        smallest = right;

    if( smallest != idx )
    {
        // Update the corresponding index in Trie node.
        minHeap->array[ smallest ]. root->indexMinHeap = idx;
        minHeap->array[ idx ]. root->indexMinHeap = smallest;

        // Swap nodes in min heap
        swapMinHeapNodes ( &minHeap->array[ smallest ], &minHeap->array
                             minHeapify( minHeap, smallest );
    }
}

// A standard function to build a heap
void buildMinHeap( MinHeap* minHeap )
{
    int n, i;
    n = minHeap->count - 1;

    for( i = ( n - 1 ) / 2; i >= 0; --i )
        minHeapify( minHeap, i );
}

// Inserts a word to heap, the function handles the 3 cases explained :
void insertInMinHeap( MinHeap* minHeap, TrieNode** root, const char* w
{
    // Case 1: the word is already present in minHeap
    if( (*root)->indexMinHeap != -1 )
    {
        ++( minHeap->array[ (*root)->indexMinHeap ]. frequency );

        // percolate down
        minHeapify( minHeap, (*root)->indexMinHeap );
    }

    // Case 2: Word is not present and heap is not full
    else if( minHeap->count < minHeap->capacity )
    {
        int count = minHeap->count;
        minHeap->array[ count ]. frequency = (*root)->frequency;
        minHeap->array[ count ]. word = new char [strlen( word ) + 1];
        strcpy( minHeap->array[ count ]. word, word );

        minHeap->array[ count ]. root = *root;
        (*root)->indexMinHeap = minHeap->count;
    }
}

```



```

        ++( minHeap->count );
        buildMinHeap( minHeap );
    }

    // Case 3: Word is not present and heap is full. And frequency of
    // is more than root. The root is the least frequent word in heap,
    // replace root with new word
    else if ( (*root)->frequency > minHeap->array[0]. frequency )
    {

        minHeap->array[ 0 ]. root->indexMinHeap = -1;
        minHeap->array[ 0 ]. root = *root;
        minHeap->array[ 0 ]. root->indexMinHeap = 0;
        minHeap->array[ 0 ]. frequency = (*root)->frequency;

        // delete previously allocated memory and
        delete [] minHeap->array[ 0 ]. word;
        minHeap->array[ 0 ]. word = new char [strlen( word ) + 1];
        strcpy( minHeap->array[ 0 ]. word, word );

        minHeapify ( minHeap, 0 );
    }
}

// Inserts a new word to both Trie and Heap
void insertUtil ( TrieNode** root, MinHeap* minHeap,
                 const char* word, const char* dupWord )
{
    // Base Case
    if ( *root == NULL )
        *root = newTrieNode();

    // There are still more characters in word
    if ( *word != '\0' )
        insertUtil ( &((*root)->child[ tolower( *word ) - 97 ]),
                    minHeap, word + 1, dupWord );
    else // The complete word is processed
    {
        // word is already present, increase the frequency
        if ( (*root)->isEnd )
            ++( (*root)->frequency );
        else
        {
            (*root)->isEnd = 1;
            (*root)->frequency = 1;
        }

        // Insert in min heap also
        insertInMinHeap( minHeap, root, dupWord );
    }
}

// add a word to Trie & min heap. A wrapper over the insertUtil
void insertTrieAndHeap(const char *word, TrieNode** root, MinHeap* minH
{
    insertUtil( root, minHeap, word, word );
}

// A utility function to show results, The min heap
// contains k most frequent words so far, at any time
void displayMinHeap( MinHeap* minHeap )

```

```

void displayMinHeap( minHeap minHeap )
{
    int i;

    // print top K word with frequency
    for( i = 0; i < minHeap->count; ++i )
    {
        printf( "%s : %d\n", minHeap->array[i].word,
                minHeap->array[i].frequency );
    }
}

// The main funtion that takes a file as input, add words to heap
// and Trie, finally shows result from heap
void printKMostFreq( FILE* fp, int k )
{
    // Create a Min Heap of Size k
    MinHeap* minHeap = createMinHeap( k );

    // Create an empty Trie
    TrieNode* root = NULL;

    // A buffer to store one word at a time
    char buffer[MAX_WORD_SIZE];

    // Read words one by one from file. Insert the word in Trie and M
    while( fscanf( fp, "%s", buffer ) != EOF )
        insertTrieAndHeap(buffer, &root, minHeap);

    // The Min Heap will have the k most frequent words, so print Min
    displayMinHeap( minHeap );
}

// Driver program to test above functions
int main()
{
    int k = 5;
    FILE *fp = fopen ( "file.txt", "r" );
    if (fp == NULL)
        printf ( "File doesn't exist " );
    else
        printKMostFreq (fp, k);
    return 0;
}

```

Output:

```

your : 3
well : 3
and : 4
to : 4
Geeks : 6

```

The above output is for a file with following content.

```

Welcome to the world of Geeks

This portal has been created to provide well written well thought and well explained
solutions for selected questions If you like Geeks for Geeks and would like to contribute

```

here is your chance You can write article and mail your article to contribute at geeksforgeeks.org See your article appearing on the Geeks for Geeks main page and 1 thousands of other Geeks

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

81. Check divisibility by 7

Given a number, check if it is divisible by 7. You are not allowed to use modulo operator, floating point arithmetic is also not allowed.

A simple method is repeated subtraction. Following is another interesting method.

Divisibility by 7 can be checked by a recursive method. A number of the form $10a + b$ is divisible by 7 if and only if $a - 2b$ is divisible by 7. In other words, subtract twice the last digit from the number formed by the remaining digits. Continue to do this until a small number.

Example: the number 371: $37 - (2 \times 1) = 37 - 2 = 35$; $3 - (2 \times 5) = 3 - 10 = -7$; thus, since -7 is divisible by 7, 371 is divisible by 7.

Following is C implementation of the above method

```
// A Program to check whether a number is divisible by 7
#include <stdio.h>

int isDivisibleBy7( int num )
{
    // If number is negative, make it positive
    if( num < 0 )
        return isDivisibleBy7( -num );

    // Base cases
    if( num == 0 || num == 7 )
        return 1;
    if( num < 10 )
        return 0;

    // Recur for ( num / 10 - 2 * num % 10 )
    return isDivisibleBy7( num / 10 - 2 * ( num - num / 10 * 10 ) );
}

// Driver program to test above function
int main()
{
    int num = 616;
    if( isDivisibleBy7(num) )
        printf( "Divisible" );
    else
        printf( "Not Divisible" );
    return 0;
}
```

Output:

```
Divisible
```

How does this work? Let 'b' be the last digit of a number 'n' and let 'a' be the number we get when we split off 'b'.

The representation of the number may also be multiplied by any number relatively prime to the divisor without changing its divisibility. After observing that 7 divides 21, we can perform the following:

```
10.a + b
```

after multiplying by 2, this becomes

```
20.a + 2.b
```

and then

```
21.a - a + 2.b
```

Eliminating the multiple of 21 gives

```
-a + 2b
```

and multiplying by -1 gives

```
a - 2b
```

There are other interesting methods to check divisibility by 7 and other numbers. See following Wiki page for details.

References:

http://en.wikipedia.org/wiki/Divisibility_rule

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

82. Find the largest multiple of 3

Given an array of non-negative integers. Find the largest multiple of 3 that can be formed from array elements.

For example, if the input array is {8, 1, 9}, the output should be "9 8 1", and if the input array is {8, 1, 7, 6, 0}, output should be "8 7 6 0".

Method 1 (Brute Force)

The simple & straight forward approach is to generate all the combinations of the elements and keep track of the largest number formed which is divisible by 3.

Time Complexity: $O(n \times 2^n)$. There will be 2^n combinations of array elements. To compare each combination with the largest number so far may take $O(n)$ time.

Auxiliary Space: $O(n)$ // to avoid integer overflow, the largest number is assumed to be stored in the form of array.

Method 2 (Tricky)

This problem can be solved efficiently with the help of $O(n)$ extra space. This method is based on the following facts about numbers which are multiple of 3.

- 1) A number is multiple of 3 if and only if the sum of digits of number is multiple of 3. For example, let us consider 8760, it is a multiple of 3 because sum of digits is $8 + 7 + 6 + 0 = 21$, which is a multiple of 3.
- 2) If a number is multiple of 3, then all permutations of it are also multiple of 3. For example, since 6078 is a multiple of 3, the numbers 8760, 7608, 7068, are also multiples of 3.
- 3) We get the same remainder when we divide the number and sum of digits of the number. For example, if divide number 151 and sum of it digits 7, by 3, we get the same remainder 1.

What is the idea behind above facts?

The value of $10 \% 3$ and $100 \% 3$ is 1. The same is true for all the higher powers of 10,

because 3 divides 9, 99, 999, ... etc.

Let us consider a 3 digit number n to prove above facts. Let the first, second and third digits of n be 'a', 'b' and 'c' respectively. n can be written as

$$n = 100.a + 10.b + c$$

Since $(10^x)\%3$ is 1 for any x , the above expression gives the same remainder as following expression

$$1.a + 1.b + c$$

So the remainder obtained by sum of digits and 'n' is same.

Following is a solution based on the above observation.

1. Sort the array in non-decreasing order.

2. Take three queues. One for storing elements which on dividing by 3 gives remainder as 0. The second queue stores digits which on dividing by 3 gives remainder as 1. The third queue stores digits which on dividing by 3 gives remainder as 2. Call them as queue0, queue1 and queue2

3. Find the sum of all the digits.

4. Three cases arise:

.....**4.1** The sum of digits is divisible by 3. Dequeue all the digits from the three queues. Sort them in non-increasing order. Output the array.

.....**4.2** The sum of digits produces remainder 1 when divided by 3.

Remove one item from queue1. If queue1 is empty, remove two items from queue2. If queue2 contains less than two items, the number is not possible.

.....**4.3** The sum of digits produces remainder 2 when divided by 3.

Remove one item from queue2. If queue2 is empty, remove two items from queue1. If queue1 contains less than two items, the number is not possible.

5. Finally empty all the queues into an auxiliary array. Sort the auxiliary array in non-increasing order. Output the auxiliary array.

Based on the above, below is the implementation:

```
/* A program to find the largest multiple of 3 from an array of elements */
#include <stdio.h>
#include <stdlib.h>

// A queue node
typedef struct Queue
{
    int front;
    int rear;
    int capacity;
    int* array;
} Queue;
```

```

, -----,

// A utility function to create a queue with given capacity
Queue* createQueue( int capacity )
{
    Queue* queue = (Queue *) malloc (sizeof(Queue));
    queue->capacity = capacity;
    queue->front = queue->rear = -1;
    queue->array = (int *) malloc (queue->capacity * sizeof(int));
    return queue;
}

// A utility function to check if queue is empty
int isEmpty (Queue* queue)
{
    return queue->front == -1;
}

// A function to add an item to queue
void Enqueue (Queue* queue, int item)
{
    queue->array[ ++queue->rear ] = item;
    if ( isEmpty(queue) )
        ++queue->front;
}

// A function to remove an item from queue
int Dequeue (Queue* queue)
{
    int item = queue->array[ queue->front ];
    if( queue->front == queue->rear )
        queue->front = queue->rear = -1;
    else
        queue->front++;
    return item;
}

// A utility function to print array contents
void printArr (int* arr, int size)
{
    int i;
    for (i = 0; i < size; ++i)
        printf ("%d ", arr[i]);
}

/* Following two functions are needed for library function qsort().
Refer following link for help of qsort()
http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */
int compareAsc( const void* a, const void* b )
{
    return *(int*)a > *(int*)b;
}
int compareDesc( const void* a, const void* b )
{
    return *(int*)a < *(int*)b;
}

// This function puts all elements of 3 queues in the auxiliary array
void populateAux (int* aux, Queue* queue0, Queue* queue1,
                  Queue* queue2, int* top )
{
    // Put all items of first queue in aux[]

```

```

while ( !isEmpty(queue0) )
    aux[ (*top)++ ] = Dequeue( queue0 );

// Put all items of second queue in aux[]
while ( !isEmpty(queue1) )
    aux[ (*top)++ ] = Dequeue( queue1 );

// Put all items of third queue in aux[]
while ( !isEmpty(queue2) )
    aux[ (*top)++ ] = Dequeue( queue2 );
}

// The main function that finds the largest possible multiple of
// 3 that can be formed by arr[] elements
int findMaxMultipleOf3( int* arr, int size )
{
    // Step 1: sort the array in non-decreasing order
    qsort( arr, size, sizeof( int ), compareAsc );

    // Create 3 queues to store numbers with remainder 0, 1
    // and 2 respectively
    Queue* queue0 = createQueue( size );
    Queue* queue1 = createQueue( size );
    Queue* queue2 = createQueue( size );

    // Step 2 and 3 get the sum of numbers and place them in
    // corresponding queues
    int i, sum;
    for ( i = 0, sum = 0; i < size; ++i )
    {
        sum += arr[i];
        if ( (arr[i] % 3) == 0 )
            Enqueue( queue0, arr[i] );
        else if ( (arr[i] % 3) == 1 )
            Enqueue( queue1, arr[i] );
        else
            Enqueue( queue2, arr[i] );
    }

    // Step 4.2: The sum produces remainder 1
    if ( (sum % 3) == 1 )
    {
        // either remove one item from queue1
        if ( !isEmpty( queue1 ) )
            Dequeue( queue1 );

        // or remove two items from queue2
        else
        {
            if ( !isEmpty( queue2 ) )
                Dequeue( queue2 );
            else
                return 0;

            if ( !isEmpty( queue2 ) )
                Dequeue( queue2 );
            else
                return 0;
        }
    }

    // Step 4.3: The sum produces remainder 2
    else if ( (sum % 3) == 2 )

```



```

else if ((sum % 3) == 2)
{
    // either remove one item from queue2
    if ( !isEmpty( queue2 ) )
        Dequeue( queue2 );

    // or remove two items from queue1
    else
    {
        if ( !isEmpty( queue1 ) )
            Dequeue( queue1 );
        else
            return 0;

        if ( !isEmpty( queue1 ) )
            Dequeue( queue1 );
        else
            return 0;
    }
}

int aux[size], top = 0;

// Empty all the queues into an auxiliary array.
populateAux (aux, queue0, queue1, queue2, &top);

// sort the array in non-increasing order
qsort (aux, top, sizeof( int ), compareDesc);

// print the result
printArr (aux, top);

return top;
}

// Driver program to test above functions
int main()
{
    int arr[] = {8, 1, 7, 6, 0};
    int size = sizeof(arr)/sizeof(arr[0]);

    if (findMaxMultipleOf3( arr, size ) == 0)
        printf( "Not Possible" );

    return 0;
}

```

The above method can be optimized in following ways.

- 1) We can use Heap Sort or Merge Sort to make the time complexity $O(n \log n)$.
- 2) We can avoid extra space for queues. We know at most two items will be removed from the input array. So we can keep track of two items in two variables.
- 3) At the end, instead of sorting the array again in descending order, we can print the ascending sorted array in reverse order. While printing in reverse order, we can skip the two elements to be removed.

The above code works only if the input arrays has numbers from 0 to 9. It can be easily extended for any positive integer array. We just have to modify the part where we sort

the array in decreasing order, at the end of code.

Time Complexity: $O(n \log n)$, assuming a $O(n \log n)$ algorithm is used for sorting.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

83. Space and time efficient Binomial Coefficient

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 $k = 2$.

We have discussed a $O(n \cdot k)$ time and $O(k)$ extra space algorithm in [this](#) post. The value of $C(n, k)$ can be calculated in $O(k)$ time and $O(1)$ extra space.

$$C(n, k) = \frac{n!}{(n-k)! \cdot k!}$$
$$= \frac{[n \cdot (n-1) \cdot \dots \cdot 1]}{[(n-k) \cdot (n-k-1) \cdot \dots \cdot 1] \cdot [k \cdot (k-1) \cdot \dots \cdot 1]}$$

After simplifying, we get

$$C(n, k) = \frac{[n \cdot (n-1) \cdot \dots \cdot (n-k+1)]}{[k \cdot (k-1) \cdot \dots \cdot 1]}$$

Also, $C(n, k) = C(n, n-k)$ // we can change r to $n-r$ if $r > n-r$

Following implementation uses above formula to calculate $C(n, k)$

```
// Program to calculate C(n ,k)
#include <stdio.h>

// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k)
{
    int res = 1;

    // Since C(n, k) = C(n, n-k)
    if ( k > n - k )
        k = n - k;

    // Calculate value of [n * (n-1) *---* (n-k+1)] / [k * (k-1) *---* 1]
    for (int i = 0; i < k; ++i)
    {
        res *= (n - i);
        res /= (i + 1);
    }

    return res;
}

/* Drier program to test above function*/
int main()
{
    int n = 8, k = 2;
    printf ("Value of C(%d, %d) is %d ", n, k, binomialCoeff(n, k) );
    return 0;
}
```

Value of C(8, 2) is 28

Time Complexity: O(k)

Auxiliary Space: O(1)

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

84. Significance of Pascal's Identity

We know the [Pascal's Identity](#) very well, i.e. $n_{C_r} = n-1_{C_r} + n-1_{C_{r-1}}$

A curious reader might have observed that Pascal's Identity is instrumental in establishing recursive relation in solving binomial coefficients. It is quite easy to prove the above identity using simple algebra. Here I'm trying to explain it's practical significance.

Recap from counting techniques, n_{C_r} means selecting r elements from n elements. Let us

pick a special element k from these n elements, we left with $(n - 1)$ elements.

We can group these r elements selection nC_r into two categories,

- 1) group that contains the element k .
- 2) group that *does not* contain the element k .

Consider first group, the special element k is in all r selections. Since k is part of r elements, we need to choose $(r - 1)$ elements from remaining $(n - 1)$ elements, there are $n-1C_{r-1}$ ways.

Consider second group, the special element k is not there in all r selections, i.e. we will have to select all the r elements from available $(n - 1)$ elements (as we must exclude element k from n). This can be done in $n-1C_r$ ways.

Now it is evident that sum of these two is selecting r elements from n elements.

There can be many ways to prove the above fact. There might be many applications of Pascal's Identity. Please share your knowledge.

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

85. Pascal's Triangle

Pascal's triangle is a triangular array of the binomial coefficients. Write a function that takes an integer value n as input and prints first n lines of the Pascal's triangle. Following are the first 6 rows of Pascal's Triangle.

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

Method 1 ($O(n^3)$ time complexity)

Number of entries in every line is equal to line number. For example, the first line has "1", the second line has "1 1", the third line has "1 2 1",... and so on. Every entry in a line is value of a **Binomial Coefficient**. The value of i th entry in line number $line$ is $C(line, i)$. The value can be calculated using following formula.

```
C(line, i) = line! / ( (line-i)! * i! )
```

A simple method is to run two loops and calculate the value of Binomial Coefficient in inner loop.

```
// A simple O(n^3) program for Pascal's Triangle
#include <stdio.h>
```

```
// See http://www.geeksforgeeks.org/archives/25621 for details of this
int binomialCoeff(int n, int k);
```

```
// Function to print first n lines of Pascal's Triangle
void printPascal(int n)
{
    // Iterate through every line and print entries in it
    for (int line = 0; line < n; line++)
    {
        // Every line has number of integers equal to line number
        for (int i = 0; i <= line; i++)
            printf("%d ", binomialCoeff(line, i));
        printf("\n");
    }
}
```

```
// See http://www.geeksforgeeks.org/archives/25621 for details of this
int binomialCoeff(int n, int k)
```

```
{
    int res = 1;
    if (k > n - k)
        k = n - k;
    for (int i = 0; i < k; ++i)
    {
        res *= (n - i);
        res /= (i + 1);
    }
    return res;
}
```

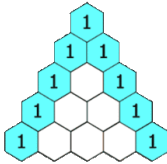
```
// Driver program to test above function
```

```
int main()
{
    int n = 7;
    printPascal(n);
    return 0;
}
```

Time complexity of this method is $O(n^3)$. Following are optimized methods.

Method 2($O(n^2)$ time and $O(n^2)$ extra space)

If we take a closer at the triangle, we observe that every entry is sum of the two values above it. So we can create a 2D array that stores previously generated values. To generate a value in a line, we can use the previously stored values from array.



```
// A O(n^2) time and O(n^2) extra space method for Pascal's Triangle
void printPascal(int n)
{
    int arr[n][n]; // An auxiliary array to store generated pascal triang

    // Iterate through every line and print integer(s) in it
    for (int line = 0; line < n; line++)
    {
        // Every line has number of integers equal to line number
        for (int i = 0; i <= line; i++)
        {
            // First and last values in every row are 1
            if (line == i || i == 0)
                arr[line][i] = 1;
            else // Other values are sum of values just above and left of above
                arr[line][i] = arr[line-1][i-1] + arr[line-1][i];
            printf("%d ", arr[line][i]);
        }
        printf("\n");
    }
}
```

This method can be optimized to use $O(n)$ extra space as we need values only from previous row. So we can create an auxiliary array of size n and overwrite values. Following is another method uses only $O(1)$ extra space.

Method 3 ($O(n^2)$ time and $O(1)$ extra space)

This method is based on method 1. We know that i th entry in a line number $line$ is Binomial Coefficient $C(line, i)$ and all lines start with value 1. The idea is to calculate $C(line, i)$ using $C(line, i-1)$. It can be calculated in $O(1)$ time using the following.

```
C(line, i) = line! / ( (line-i)! * i! )
C(line, i-1) = line! / ( (line - i + 1)! * (i-1)! )

We can derive following expression from above two expressions.
C(line, i) = C(line, i-1) * (line - i + 1) / i
```

So $C(line, i)$ can be calculated from $C(line, i-1)$ in $O(1)$ time

```
// A  $O(n^2)$  time and  $O(1)$  extra space function for Pascal's Triangle
void printPascal(int n)
{
    for (int line = 1; line <= n; line++)
    {
        int C = 1; // used to represent C(line, i)
        for (int i = 1; i <= line; i++)
        {
            printf("%d ", C); // The first value in a line is always 1
            C = C * (line - i) / i;
        }
        printf("\n");
    }
}
```

So method 3 is the best method among all, but it may cause integer overflow for large values of n as it multiplies two integers to obtain values.

This article is compiled by **Rahul** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

86. Greedy Algorithms | Set 3 (Huffman Coding)

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are **Prefix Codes**, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream. Let us understand prefix codes with a counter example. Let there be four characters a , b , c and d , and their corresponding variable length codes be 00 , 01 , 0 and 1 . This coding leads to ambiguity because code assigned to c is prefix of codes assigned to a and b . If the compressed bit stream is 0001 , the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

See [this](#) for applications of Huffman Coding.

There are mainly two major parts in Huffman Coding

- 1) Build a Huffman Tree from input characters.
- 2) Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree

Input is array of unique characters along with their frequency of occurrences and output

is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

2. Extract two nodes with the minimum frequency from the min heap.

3. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

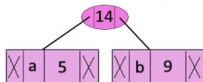
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Let us understand the algorithm with an example:

character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

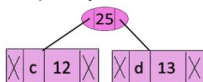
Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency $5 + 9 = 14$.



Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency $12 + 13 = 25$

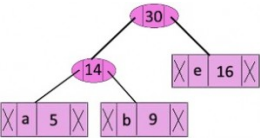


Now min heap contains 4 nodes where 2 nodes are roots of trees with single element

each, and two heap nodes are root of tree with more than one nodes.

character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

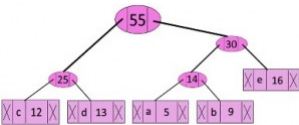
Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency $14 + 16 = 30$



Now min heap contains 3 nodes.

character	Frequency
Internal Node	25
Internal Node	30
f	45

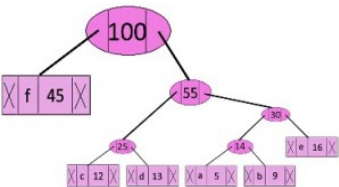
Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency $25 + 30 = 55$



Now min heap contains 2 nodes.

character	Frequency
f	45
Internal Node	55

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency $45 + 55 = 100$



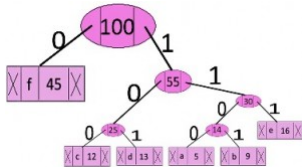
Now min heap contains only one node.

character	Frequency
Internal Node	100

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



The codes are as follows:

character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

```
// C program for Huffman Coding
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
// This constant can be avoided by explicitly calculating height of Hu
#define MAX_TREE_HT 100
```

```
// A Huffman tree node
```

```
struct MinHeapNode
```

```
{
```

```
    char data; // One of the input characters
```

```
    unsigned freq; // Frequency of the character
```

```
    struct MinHeapNode *left, *right; // Left and right child of this node
```

```
};
```

```
// A Min Heap: Collection of min heap (or Huffman tree) nodes
```

```
struct MinHeap
```

```
{
```

```
    unsigned size; // Current size of min heap
```

```
    unsigned capacity; // capacity of min heap
```

```
    struct MinHeapNode **array; // Array of minheap node pointers
```

```
};
```

```
// A utility function allocate a new min heap node with given character
// and frequency of the character
```

```
struct MinHeapNode* newNode(char data, unsigned freq)
```

```
{
```

```
    struct MinHeapNode* temp =
        (struct MinHeapNode*) malloc(sizeof(struct MinHeapNode));
```

```
    temp->left = temp->right = NULL;
```

```
    temp->data = data;
```

```
    temp->freq = freq;
```

```
    return temp;
```

```
}
```

```
// A utility function to create a min heap of given capacity
```

```

// A utility function to create a min heap of given capacity
struct MinHeap* createMinHeap(unsigned capacity)
{
    struct MinHeap* minHeap =
        (struct MinHeap*) malloc(sizeof(struct MinHeap));
    minHeap->size = 0; // current size is 0
    minHeap->capacity = capacity;
    minHeap->array =
        (struct MinHeapNode**) malloc(minHeap->capacity * sizeof(struct MinHeapNode));
    return minHeap;
}

// A utility function to swap two min heap nodes
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
{
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// The standard minHeapify function.
void minHeapify(struct MinHeap* minHeap, int idx)
{
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    if (left < minHeap->size &&
        minHeap->array[left]->freq < minHeap->array[smallest]->freq)
        smallest = left;

    if (right < minHeap->size &&
        minHeap->array[right]->freq < minHeap->array[smallest]->freq)
        smallest = right;

    if (smallest != idx)
    {
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}

// A utility function to check if size of heap is 1 or not
int isSizeOne(struct MinHeap* minHeap)
{
    return (minHeap->size == 1);
}

// A standard function to extract minimum value node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap)
{
    struct MinHeapNode* temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeapify(minHeap, 0);
    return temp;
}

// A utility function to insert a new node to Min Heap
void insertMinHeap(struct MinHeap* minHeap, struct MinHeapNode* minHeapNode)
{
    ++minHeap->size;
    int i = minHeap->size - 1;

```

```

    while (i && minHeapNode->freq < minHeap->array[(i - 1)/2]->freq)
    {
        minHeap->array[i] = minHeap->array[(i - 1)/2];
        i = (i - 1)/2;
    }
    minHeap->array[i] = minHeapNode;
}

// A standard function to build min heap
void buildMinHeap(struct MinHeap* minHeap)
{
    int n = minHeap->size - 1;
    int i;
    for (i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
}

// A utility function to print an array of size n
void printArr(int arr[], int n)
{
    int i;
    for (i = 0; i < n; ++i)
        printf("%d", arr[i]);
    printf("\n");
}

// Utility function to check if this node is leaf
int isLeaf(struct MinHeapNode* root)
{
    return !(root->left) && !(root->right) ;
}

// Creates a min heap of capacity equal to size and inserts all charac
// data[] in min heap. Initially size of min heap is equal to capacity
struct MinHeap* createAndBuildMinHeap(char data[], int freq[], int size)
{
    struct MinHeap* minHeap = createMinHeap(size);
    for (int i = 0; i < size; ++i)
        minHeap->array[i] = newNode(data[i], freq[i]);
    minHeap->size = size;
    buildMinHeap(minHeap);
    return minHeap;
}

// The main function that builds Huffman tree
struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int size)
{
    struct MinHeapNode *left, *right, *top;

    // Step 1: Create a min heap of capacity equal to size. Initially
    // modes equal to size.
    struct MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);

    // Iterate while size of heap doesn't become 1
    while (!isSizeOne(minHeap))
    {
        // Step 2: Extract the two minimum freq items from min heap
        left = extractMin(minHeap);
        right = extractMin(minHeap);

        // Step 3: Create a new internal node with frequency equal to
        // sum of the two nodes frequencies. Make the two extracted no

```

```

        // left and right children of this new node. Add this node to
        // '$' is a special value for internal nodes, not used
        top = newNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
        insertMinHeap(minHeap, top);
    }

    // Step 4: The remaining node is the root node and the tree is complete
    return extractMin(minHeap);
}

// Prints Huffman codes from the root of Huffman Tree. It uses arr[] to
// store codes
void printCodes(struct MinHeapNode* root, int arr[], int top)
{
    // Assign 0 to left edge and recur
    if (root->left)
    {
        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }

    // Assign 1 to right edge and recur
    if (root->right)
    {
        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }

    // If this is a leaf node, then it contains one of the input
    // characters, print the character and its code from arr[]
    if (isLeaf(root))
    {
        printf("%c: ", root->data);
        printArr(arr, top);
    }
}

// The main function that builds a Huffman Tree and print codes by traversing
// the built Huffman Tree
void HuffmanCodes(char data[], int freq[], int size)
{
    // Construct Huffman Tree
    struct MinHeapNode* root = buildHuffmanTree(data, freq, size);

    // Print Huffman codes using the Huffman tree built above
    int arr[MAX_TREE_HT], top = 0;
    printCodes(root, arr, top);
}

// Driver program to test above functions
int main()
{
    char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'};
    int freq[] = {5, 9, 12, 13, 16, 45};
    int size = sizeof(arr)/sizeof(arr[0]);
    HuffmanCodes(arr, freq, size);
    return 0;
}

```

```
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```

Time complexity: $O(n \log n)$ where n is the number of unique characters. If there are n nodes, `extractMin()` is called $2*(n - 1)$ times. `extractMin()` takes $O(\log n)$ time as it calls `minHeapify()`. So, overall complexity is $O(n \log n)$.

If the input array is sorted, there exists a linear time algorithm. We will soon be discussing in our next post.

Reference:

http://en.wikipedia.org/wiki/Huffman_coding

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

87. Greedy Algorithms | Set 4 (Efficient Huffman Coding for Sorted Input)

We recommend to read following post as a prerequisite for this.

Greedy Algorithms | Set 3 (Huffman Coding)

Time complexity of the algorithm discussed in above post is $O(n \log n)$. If we know that the given array is sorted (by non-decreasing order of frequency), we can generate Huffman codes in $O(n)$ time. Following is a $O(n)$ algorithm for sorted input.

1. Create two empty queues.
2. Create a leaf node for each unique character and Enqueue it to the first queue in non-decreasing order of frequency. Initially second queue is empty.
3. Dequeue two nodes with the minimum frequency by examining the front of both queues. Repeat following steps two times
 -a) If second queue is empty, dequeue from first queue.
 -b) If first queue is empty, dequeue from second queue.
 -c) Else, compare the front of two queues and dequeue the minimum.
4. Create a new internal node with frequency equal to the sum of the two nodes

frequencies. Make the first Dequeued node as its left child and the second Dequeued node as right child. Enqueue this node to second queue.

5. Repeat steps#3 and #4 until there is more than one node in the queues. The remaining node is the root node and the tree is complete.

```
// C Program for Efficient Huffman Coding for Sorted input
#include <stdio.h>
#include <stdlib.h>

// This constant can be avoided by explicitly calculating height of Hu
#define MAX_TREE_HT 100

// A node of huffman tree
struct QueueNode
{
    char data;
    unsigned freq;
    struct QueueNode *left, *right;
};

// Structure for Queue: collection of Huffman Tree nodes (or QueueNode)
struct Queue
{
    int front, rear;
    int capacity;
    struct QueueNode **array;
};

// A utility function to create a new QueueNode
struct QueueNode* newNode(char data, unsigned freq)
{
    struct QueueNode* temp =
        (struct QueueNode*) malloc(sizeof(struct QueueNode));
    temp->left = temp->right = NULL;
    temp->data = data;
    temp->freq = freq;
    return temp;
}

// A utility function to create a Queue of given capacity
struct Queue* createQueue(int capacity)
{
    struct Queue* queue = (struct Queue*) malloc(sizeof(struct Queue))
    queue->front = queue->rear = -1;
    queue->capacity = capacity;
    queue->array =
        (struct QueueNode**) malloc(queue->capacity * sizeof(struct QueueNode))
    return queue;
}

// A utility function to check if size of given queue is 1
int isSizeOne(struct Queue* queue)
{
    return queue->front == queue->rear && queue->front != -1;
}

// A utility function to check if given queue is empty
int isEmpty(struct Queue* queue)
{
    return queue->front == -1;
}
```

```

}

// A utility function to check if given queue is full
int isFull(struct Queue* queue)
{
    return queue->rear == queue->capacity - 1;
}

// A utility function to add an item to queue
void enqueue(struct Queue* queue, struct QueueNode* item)
{
    if (isFull(queue))
        return;
    queue->array[++queue->rear] = item;
    if (queue->front == -1)
        ++queue->front;
}

// A utility function to remove an item from queue
struct QueueNode* dequeue(struct Queue* queue)
{
    if (isEmpty(queue))
        return NULL;
    struct QueueNode* temp = queue->array[queue->front];
    if (queue->front == queue->rear) // If there is only one item in
        queue->front = queue->rear = -1;
    else
        ++queue->front;
    return temp;
}

// A utility function to get front of queue
struct QueueNode* getFront(struct Queue* queue)
{
    if (isEmpty(queue))
        return NULL;
    return queue->array[queue->front];
}

/* A function to get minimum item from two queues */
struct QueueNode* findMin(struct Queue* firstQueue, struct Queue* secondQueue)
{
    // Step 3.a: If second queue is empty, dequeue from first queue
    if (isEmpty(firstQueue))
        return dequeue(secondQueue);

    // Step 3.b: If first queue is empty, dequeue from second queue
    if (isEmpty(secondQueue))
        return dequeue(firstQueue);

    // Step 3.c: Else, compare the front of two queues and dequeue min
    if (getFront(firstQueue)->freq < getFront(secondQueue)->freq)
        return dequeue(firstQueue);

    return dequeue(secondQueue);
}

// Utility function to check if this node is leaf
int isLeaf(struct QueueNode* root)
{
    return !(root->left) && !(root->right) ;
}

```



```
// A utility function to print an array of size n
```

```
void printArr(int arr[], int n)
{
    int i;
    for (i = 0; i < n; ++i)
        printf("%d", arr[i]);
    printf("\n");
}
```

```
// The main function that builds Huffman tree
```

```
struct QueueNode* buildHuffmanTree(char data[], int freq[], int size)
{
    struct QueueNode *left, *right, *top;

    // Step 1: Create two empty queues
    struct Queue* firstQueue = createQueue(size);
    struct Queue* secondQueue = createQueue(size);

    // Step 2: Create a leaf node for each unique character and Enqueue
    // the first queue in non-decreasing order of frequency. Initially
    // queue is empty
    for (int i = 0; i < size; ++i)
        enqueue(firstQueue, newNode(data[i], freq[i]));

    // Run while Queues contain more than one node. Finally, first queue
    // be empty and second queue will contain only one node
    while (!(isEmpty(firstQueue) && isSizeOne(secondQueue)))
    {
        // Step 3: Dequeue two nodes with the minimum frequency by exa
        // the front of both queues
        left = findMin(firstQueue, secondQueue);
        right = findMin(firstQueue, secondQueue);

        // Step 4: Create a new internal node with frequency equal to
        // of the two nodes frequencies. Enqueue this node to second q
        top = newNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
        enqueue(secondQueue, top);
    }

    return dequeue(secondQueue);
}
```

```
// Prints huffman codes from the root of Huffman Tree. It uses arr[]
// store codes
```

```
void printCodes(struct QueueNode* root, int arr[], int top)
{
    // Assign 0 to left edge and recur
    if (root->left)
    {
        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }

    // Assign 1 to right edge and recur
    if (root->right)
    {
        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }
}
```

```

    // If this is a leaf node, then it contains one of the input
    // characters, print the character and its code from arr[]
    if (isLeaf(root))
    {
        printf("%c: ", root->data);
        printArr(arr, top);
    }
}

// The main function that builds a Huffman Tree and print codes by traversing
// the built Huffman Tree
void HuffmanCodes(char data[], int freq[], int size)
{
    // Construct Huffman Tree
    struct QueueNode* root = buildHuffmanTree(data, freq, size);

    // Print Huffman codes using the Huffman tree built above
    int arr[MAX_TREE_HT], top = 0;
    printCodes(root, arr, top);
}

// Driver program to test above functions
int main()
{
    char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'};
    int freq[] = {5, 9, 12, 13, 16, 45};
    int size = sizeof(arr)/sizeof(arr[0]);
    HuffmanCodes(arr, freq, size);
    return 0;
}

```

Output:

```

f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111

```

Time complexity: $O(n)$

If the input is not sorted, it need to be sorted first before it can be processed by the above algorithm. Sorting can be done using heap-sort or merge-sort both of which run in $\Theta(n \log n)$. So, the overall time complexity becomes $O(n \log n)$ for unsorted input.

Reference:

http://en.wikipedia.org/wiki/Huffman_coding

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

88. Select a random number from stream, with $O(1)$ space

Given a stream of numbers, generate a random number from the stream. You are allowed to use only $O(1)$ space and the input is in the form of stream, so can't store the previously seen numbers.

So how do we generate a random number from the whole stream such that the probability of picking any number is $1/n$. with $O(1)$ extra space? This problem is a variation of **Reservoir Sampling**. Here the value of k is 1.

- 1) Initialize 'count' as 0, 'count' is used to store count of numbers seen so far in stream.
- 2) For each number 'x' from stream, do following
 -a) Increment 'count' by 1.
 -b) If count is 1, set result as x, and return result.
 -c) Generate a random number from 0 to 'count-1'. Let the generated random number be i.
 -d) If i is equal to 'count - 1', update the result as x.

```

// An efficient program to randomly select a number from stream of num
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// A function to randomly select a item from stream[0], stream[1], ..
int selectRandom(int x)
{
    static int res;    // The resultant random number
    static int count = 0; //Count of numbers visited so far in stream

    count++; // increment count of numbers seen so far

    // If this is the first element from stream, return it
    if (count == 1)
        res = x;
    else
    {
        // Generate a random number from 0 to count - 1
        int i = rand() % count;

        // Replace the prev random number with new number with 1/count
        if (i == count - 1)
            res = x;
    }
    return res;
}

// Driver program to test above function.
int main()
{
    int stream[] = {1, 2, 3, 4};
    int n = sizeof(stream)/sizeof(stream[0]);

    // Use a different seed value for every run.
    srand(time(NULL));
    for (int i = 0; i < n; ++i)
        printf("Random number from first %d numbers is %d \n",
               i+1, selectRandom(stream[i]));

    return 0;
}

```

Output:

```

Random number from first 1 numbers is 1
Random number from first 2 numbers is 1
Random number from first 3 numbers is 3
Random number from first 4 numbers is 4

```

Auxiliary Space: $O(1)$

How does this work

We need to prove that every element is picked with $1/n$ probability where n is the number of items seen so far. For every new stream item x , we pick a random number from 0 to 'count - 1', if the picked number is 'count-1', we replace the previous result with x .

To simplify proof, let us first consider the last element, the last element replaces the

previously stored result with $1/n$ probability. So probability of getting last element as result is $1/n$.

Let us now talk about second last element. When second last element processed first time, the probability that it replaced the previous result is $1/(n-1)$. The probability that previous result stays when n th item is considered is $(n-1)/n$. So probability that the second last element is picked in last iteration is $[1/(n-1)] * [(n-1)/n]$ which is $1/n$.

Similarly, we can prove for third last element and others.

References:

[Reservoir Sampling](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

89. [\[TopTalent.in\]](#) How Tech companies Like Their Résumés

Have all the skills require to be a great Software Engineer? Here are 9 tips to make your résumé look just like how the Tech Companies want them to be. After all, your résumé is the most powerful ship to a perfect job, suitable for your skills.

1. The Opening: If your résumé starts with “*Objective: To utilize my knowledge, skills and abilities as a Software Engineer*” then you can definitely bid the job a farewell.

2. Long = Boring: Unless you have 10+ years of experience, your résumé’s length cannot exceed one page. When the HR professional has only get 15 seconds to read someone’s resume, paragraphs would simply get overlooked. Use bullets which are short (1-2 lines) and ideally, no more than half of the bullets should be 2 lines.

3. Always provide specifics: You can speak about all the positions you held and companies you worked for, but it may all go in vain. Most HR professionals don’t have time to interview you just to bring those specifics out. Focus on specific accomplishments, not responsibilities. Just that one point, arguably, can give you an edge over most people in the pile of résumé.

4. Grammar Voes: I know we have endlessly told this (Read [Your résumé is your sharpest sword](#), keep it sharp and clean), but it deserves repetition. Grievous spelling errors = trash can. As simple as that. In today’s world, poor mastery of the English language is not acceptable – even if your job is related to coding and developing software.

5. Ignore the obvious: You have experience with Windows and MS Office? Please tell the person more, because that definitely is uninteresting. Furthermore, No acronyms on

the résumé unless the word is very popular in the industry.

6. Best format for experience: Unless the company you're applying for is huge, where you are sure that only keyword matching systems can get your résumé from the mail box to the HR official's table please avoid listing every single programming language that you've worked with on your resume. Here's a good example of a format which provides the key words that Applicant Tracking Systems search for, at the same time giving any recruiter a better idea of how you used each technology at each of your jobs.

What it should be like:

Software Engineer, Company X 2007- present:

– Coded a web application using JavaScript, HTML5, and CSS3

– Designed and implemented a JSON/REST API to allow access to the Company database

What it shouldn't be like:

Programming languages known: C, C++, Java, C#, Python, JavaScript, Visual Basic, Visual C#, Visual Basic .NET, and so on. A dozen languages listed together don't make much sense. How can you make it worse? By listing all the versions of a language you've used.

7. More content, less space: Use a good resume template, with columns. This will allow you to fit more content on your resume while making it easier to scan for key information like company names and titles.

3 lines –

Microsoft Corporation

Software Engineer

2008 – 2011

1 line –

Software Engineer

Microsoft Corporation

2008 – 2011

8. Team vs. You: The HR guy knows that most of your work in companies must be team work. But, he is hiring you. So, tell them what you specifically built, created, implemented, designed, programmed. But showing that you are team focussed and good with adapting to team work is also a good thing to do

9. Failure can be great: To conclude, we advise you to stop thinking about what does and doesn't belong on a resume and start thinking about whether something makes you look more or less awesome. For example, a large number of people decide not to include meaty projects because they were for a class assignment/independent

projects/unfinished/unsuccessful. But coding is coding, and your exposure to real work rather than courses at college can put you in a really good position. Links to code (having a GitHub link is a great thing), portfolio work, or publicly visible projects can also act as positive things on your résumé.

This article is powered by **TopTalent.in** - A high end Job portal for students and alumni of Premier Colleges in India. Sign up now for free exclusive access to top notch jobs in India and abroad. Get in touch with them via [facebook](#), [twitter](#) or [linkedin](#). If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks

90. Implement Stack using Queues

The problem is opposite of [this](#) post. We are given a Queue data structure that supports standard operations like enqueue() and dequeue(). We need to implement a Stack data structure using only instances of Queue.

A stack can be implemented using two queues. Let stack to be implemented be 's' and queues used to implement be 'q1' and 'q2'. Stack 's' can be implemented in two ways:

Method 1 (By making push operation costly)

This method makes sure that newly entered element is always at the front of 'q1', so that pop operation just dequeues from 'q1'. 'q2' is used to put every new element at front of 'q1'.

```
push(s, x) // x is the element to be pushed and s is stack
1) Enqueue x to q2
2) One by one dequeue everything from q1 and enqueue to q2.
3) Swap the names of q1 and q2
// Swapping of names is done to avoid one more movement of all elements
// from q2 to q1.

pop(s)
1) Dequeue an item from q1 and return it.
```

Method 2 (By making pop operation costly)

In push operation, the new element is always enqueued to q1. In pop() operation, if q2 is empty then all the elements except the last, are moved to q2. Finally the last element is dequeued from q1 and returned.

```
push(s, x)
1) Enqueue x to q1 (assuming size of q1 is unlimited).
```

```
pop(s)
```

- 1) One by one dequeue everything except the last element from q1 and enqueue to q2
- 2) Dequeue the last item of q1, the dequeued item is result, store it.
- 3) Swap the names of q1 and q2
- 4) Return the item stored in step 2.

```
// Swapping of names is done to avoid one more movement of all elements  
// from q2 to q1.
```

References:

Implement Stack using Two Queues

This article is compiled by **Sumit Jain** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

91. Find the largest multiple of 2, 3 and 5

An array of size n is given. The array contains digits from 0 to 9. Generate the largest number using the digits in the array such that the number is divisible by 2, 3 and 5.

For example, if the arrays is {1, 8, 7, 6, 0}, output must be: 8760. And if the arrays is {7, 7, 7, 6}, output must be: "no number can be formed".

Source: [Amazon Interview | Set 7](#)

This problem is a variation of "[Find the largest multiple of 3](#)".

Since the number has to be divisible by 2 and 5, it has to have last digit as 0. So if the given array doesn't contain any zero, then no solution exists.

Once a 0 is available, extract 0 from the given array. Only thing left is, the number should be divisible by 3 and the largest of all. Which has been discussed [here](#).

Thanks to [shashank](#) for suggesting this solution. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

92. Divide and Conquer | Set 1 (Introduction)

Like **Greedy** and **Dynamic Programming**, Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps.

1. **Divide**: Break the given problem into subproblems of same type.
2. **Conquer**: Recursively solve these subproblems
3. **Combine**: Appropriately combine the answers

Following are some standard algorithms that are Divide and Conquer algorithms.

1) **Binary Search** is a searching algorithm. In each step, the algorithm compares the input element x with the value of the middle element in array. If the values match, return the index of middle. Otherwise, if x is less than the middle element, then the algorithm recurs for left side of middle element, else recurs for right side of middle element.

2) **Quicksort** is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.

3) **Merge Sort** is also a sorting algorithm. The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.

4) **Closest Pair of Points** The problem is to find the closest pair of points in a set of points in x - y plane. The problem can be solved in $O(n^2)$ time by calculating distances of every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solves the problem in $O(n \log n)$ time.

5) **Strassen's Algorithm** is an efficient algorithm to multiply two matrices. A simple method to multiply two matrices need 3 nested loops and is $O(n^3)$. Strassen's algorithm multiplies two matrices in $O(n^{2.8974})$ time.

6) **Cooley–Tukey Fast Fourier Transform (FFT) algorithm** is the most common algorithm for FFT. It is a divide and conquer algorithm which works in $O(n \log n)$ time.

7) **Karatsuba algorithm for fast multiplication** it does multiplication of two n -digit numbers in at most $\frac{n^2}{3}$ single-digit multiplications in general (and exactly $\frac{n^2}{3}$ when n is a power of 2). It is therefore faster than the **classical** algorithm, which requires n^2 single-digit products. If $n = 2^{10} = 1024$, in particular, the exact counts are $3^{10} = 59,049$ and $(2^{10})^2 = 1,048,576$, respectively.

We will publishing above algorithms in separate posts.

Divide and Conquer (D & C) vs Dynamic Programming (DP)

Both paradigms (D & C and DP) divide the given problem into subproblems and solve subproblems. How to choose one of them for a given problem? Divide and Conquer should be used when same subproblems are not evaluated many times. Otherwise

Dynamic Programming or Memoization should be used. For example, Binary Search is a Divide and Conquer algorithm, we never evaluate the same subproblems again. On the other hand, for calculating nth Fibonacci number, Dynamic Programming should be preferred (See [this](#) for details).

References

Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani

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

http://en.wikipedia.org/wiki/Karatsuba_algorithm

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

93. Divide and Conquer | Set 2 (Closest Pair of Points)

We are given an array of n points in the plane, and the problem is to find out the closest pair of points in the array. This problem arises in a number of applications. For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision. Recall the following formula for distance between two points p and q.

$$\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

The Brute force solution is $O(n^2)$, compute the distance between each pair and return the smallest. We can calculate the smallest distance in $O(n \log n)$ time using Divide and Conquer strategy. In this post, a $O(n \times (\log n)^2)$ approach is discussed. We will be discussing a $O(n \log n)$ approach in a separate post.

Algorithm

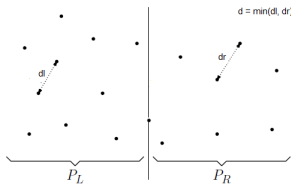
Following are the detailed steps of a $O(n (\log n)^2)$ algorithm.

Input: An array of n points $P[]$

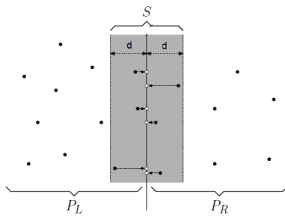
Output: The smallest distance between two points in the given array.

As a pre-processing step, input array is sorted according to x coordinates.

- 1) Find the middle point in the sorted array, we can take $P[n/2]$ as middle point.
- 2) Divide the given array in two halves. The first subarray contains points from $P[0]$ to $P[n/2]$. The second subarray contains points from $P[n/2+1]$ to $P[n-1]$.
- 3) Recursively find the smallest distances in both subarrays. Let the distances be d_l and d_r . Find the minimum of d_l and d_r . Let the minimum be d .



4) From above 3 steps, we have an upper bound d of minimum distance. Now we need to consider the pairs such that one point in pair is from left half and other is from right half. Consider the vertical line passing through $P[n/2]$ and find all points whose x coordinate is closer than d to the middle vertical line. Build an array `strip[]` of all such points.



5) Sort the array `strip[]` according to y coordinates. This step is $O(n \log n)$. It can be optimized to $O(n)$ by recursively sorting and merging.

6) Find the smallest distance in `strip[]`. This is tricky. From first look, it seems to be a $O(n^2)$ step, but it is actually $O(n)$. It can be proved geometrically that for every point in strip, we only need to check at most 7 points after it (note that strip is sorted according to Y coordinate). See [this](#) for more analysis.

7) Finally return the minimum of d and distance calculated in above step (step 6)

Implementation

Following is C/C++ implementation of the above algorithm.

```
// A divide and conquer program in C/C++ to find the smallest distance
// given set of points.
```

```
#include <stdio.h>
#include <float.h>
#include <stdlib.h>
#include <math.h>
```

```
// A structure to represent a Point in 2D plane
```

```
struct Point
{
    int x, y;
};
```

```
/* Following two functions are needed for library function qsort().
Refer: http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */
```

```
// Needed to sort array of points according to X coordinate
```

```

int compareX(const void* a, const void* b)
{
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->x - p2->x);
}
// Needed to sort array of points according to Y coordinate
int compareY(const void* a, const void* b)
{
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->y - p2->y);
}

// A utility function to find the distance between two points
float dist(Point p1, Point p2)
{
    return sqrt( (p1.x - p2.x)*(p1.x - p2.x) +
                 (p1.y - p2.y)*(p1.y - p2.y)
                );
}

// A Brute Force method to return the smallest distance between two po
// in P[] of size n
float bruteForce(Point P[], int n)
{
    float min = FLT_MAX;
    for (int i = 0; i < n; ++i)
        for (int j = i+1; j < n; ++j)
            if (dist(P[i], P[j]) < min)
                min = dist(P[i], P[j]);
    return min;
}

// A utility function to find minimum of two float values
float min(float x, float y)
{
    return (x < y)? x : y;
}

// A utility function to find the distance between the closest points o
// strip of given size. All points in strip[] are sorted accordint to
// y coordinate. They all have an upper bound on minimum distance as d
// Note that this method seems to be a O(n^2) method, but it's a O(n)
// method as the inner loop runs at most 6 times
float stripClosest(Point strip[], int size, float d)
{
    float min = d; // Initialize the minimum distance as d

    qsort(strip, size, sizeof(Point), compareY);

    // Pick all points one by one and try the next points till the dif
    // between y coordinates is smaller than d.
    // This is a proven fact that this loop runs at most 6 times
    for (int i = 0; i < size; ++i)
        for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min;
            if (dist(strip[i], strip[j]) < min)
                min = dist(strip[i], strip[j]);

    return min;
}

// A recursive function to find the smallest distance. The array P con
// all points sorted according to x coordinate

```

```

// all points sorted according to x coordinate
float closestUtil(Point P[], int n)
{
    // If there are 2 or 3 points, then use brute force
    if (n <= 3)
        return bruteForce(P, n);

    // Find the middle point
    int mid = n/2;
    Point midPoint = P[mid];

    // Consider the vertical line passing through the middle point
    // calculate the smallest distance dl on left of middle point and
    // dr on right side
    float dl = closestUtil(P, mid);
    float dr = closestUtil(P + mid, n-mid);

    // Find the smaller of two distances
    float d = min(dl, dr);

    // Build an array strip[] that contains points close (closer than d)
    // to the line passing through the middle point
    Point strip[n];
    int j = 0;
    for (int i = 0; i < n; i++)
        if (abs(P[i].x - midPoint.x) < d)
            strip[j] = P[i], j++;

    // Find the closest points in strip. Return the minimum of d and
    // distance is strip[]
    return min(d, stripClosest(strip, j, d) );
}

// The main function that finds the smallest distance
// This method mainly uses closestUtil()
float closest(Point P[], int n)
{
    qsort(P, n, sizeof(Point), compareX);

    // Use recursive function closestUtil() to find the smallest distance
    return closestUtil(P, n);
}

// Driver program to test above functions
int main()
{
    Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};
    int n = sizeof(P) / sizeof(P[0]);
    printf("The smallest distance is %f ", closest(P, n));
    return 0;
}

```

Output:

```
The smallest distance is 1.414214
```

Time Complexity Let Time complexity of above algorithm be $T(n)$. Let us assume that we use a $O(n \log n)$ sorting algorithm. The above algorithm divides all points in two sets and recursively calls for two sets. After dividing, it finds the strip in $O(n)$ time, sorts the strip in $O(n \log n)$ time and finally finds the closest points in strip in $O(n)$ time. So $T(n)$

can expressed as follows

$$T(n) = 2T(n/2) + O(n) + O(n \log n) + O(n)$$

$$T(n) = 2T(n/2) + O(n \log n)$$

$$T(n) = T(n \times \log n \times \log n)$$

Notes

- 1) Time complexity can be improved to $O(n \log n)$ by optimizing step 5 of the above algorithm. We will soon be discussing the optimized solution in a separate post.
- 2) The code finds smallest distance. It can be easily modified to find the points with smallest distance.
- 3) The code uses quick sort which can be $O(n^2)$ in worst case. To have the upper bound as $O(n (\log n)^2)$, a $O(n \log n)$ sorting algorithm like merge sort or heap sort can be used

References:

<http://www.cs.umd.edu/class/fall2013/cmsc451/Lects/lect10.pdf>

<http://www.youtube.com/watch?v=vS4Zn1a9KUc>

<http://www.youtube.com/watch?v=T3T7T8Ym20M>

http://en.wikipedia.org/wiki/Closest_pair_of_points_problem

94. Abstract Classes in Java

In C++, if a class has at least one pure virtual function, then the class becomes abstract. Unlike C++, in Java, a separate keyword *abstract* is used to make a class abstract.

```
// An example abstract class in Java
abstract class Shape {
    int color;

    // An abstract function (like a pure virtual function in C++)
    abstract void draw();
}
```

Following are some important observations about abstract classes in Java.

- 1) Like C++, in Java, an instance of an abstract class cannot be created, we can have references of abstract class type though.

```
abstract class Base {
    abstract void fun();
}

class Derived extends Base {
    void fun() { System.out.println("Derived fun() called"); }
```

```

}
class Main {
    public static void main(String args[]) {

        // Uncommenting the following line will cause compiler error as the
        // line tries to create an instance of abstract class.
        // Base b = new Base();

        // We can have references of Base type.
        Base b = new Derived();
        b.fun();
    }
}

```

Output:

```

Derived fun() called

```

2) Like C++, an abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of a inherited class is created. For example, the following is a valid Java program.

```

// An abstract class with constructor
abstract class Base {
    Base() { System.out.println("Base Constructor Called"); }
    abstract void fun();
}
class Derived extends Base {
    Derived() { System.out.println("Derived Constructor Called"); }
    void fun() { System.out.println("Derived fun() called"); }
}
class Main {
    public static void main(String args[]) {
        Derived d = new Derived();
    }
}

```

Output:

```

Base Constructor Called
Derived Constructor Called

```

3) In Java, we can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated, but can only be inherited.

```
// An abstract class without any abstract method
abstract class Base {
    void fun() { System.out.println("Base fun() called"); }
}

class Derived extends Base { }

class Main {
    public static void main(String args[]) {
        Derived d = new Derived();
        d.fun();
    }
}
```

Output:

```
Base fun() called
```

4) Abstract classes can also have final methods (methods that cannot be overridden). For example, the following program compiles and runs fine.

```
// An abstract class with a final method
abstract class Base {
    final void fun() { System.out.println("Derived fun() called"); }
}

class Derived extends Base {}

class Main {
    public static void main(String args[]) {
        Base b = new Derived();
        b.fun();
    }
}
```

Output:

```
Derived fun() called
```

Exercise:

1. Is it possible to create abstract and final class in Java?
2. Is it possible to have an abstract method in a final class?
3. Is it possible to inherit from multiple abstract classes in Java?

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

95. Program to find amount of water in a given glass

There are some glasses with equal capacity as 1 litre. The glasses are kept as follows:

```
      1
     2  3
    4  5  6
   7  8  9 10
```

You can put water to only top glass. If you put more than 1 litre water to 1st glass, water overflows and fills equally in both 2nd and 3rd glasses. Glass 5 will get water from both 2nd glass and 3rd glass and so on.

If you have X litre of water and you put that water in top glass, how much water will be contained by jth glass in ith row?

Example. If you will put 2 litre on top.

1st – 1 litre

2nd – 1/2 litre

3rd – 1/2 litre

Source: [Amazon Interview | Set 12](#)

The approach is similar to Method 2 of the [Pascal's Triangle](#). If we take a closer look at the problem, the problem boils down to [Pascal's Triangle](#).

```
              1 ----- 1
            2  3 ----- 2
              4  5  6 ----- 3
             7  8  9 10 ----- 4
```

Each glass contributes to the two glasses down the glass. Initially, we put all water in first glass. Then we keep 1 litre (or less than 1 litre) in it, and move rest of the water to two glasses down to it. We follow the same process for the two glasses and all other glasses till ith row. There will be $i*(i+1)/2$ glasses till ith row.

// Program to find the amount of water in jth glass of ith row

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

// Returns the amount of water in jth glass of ith row

```
float findWater(int i, int j, float X)
{
    // A row number i has maximum i columns. So input column number must
    // be less than i
    if (j > i)
        return 0;
```

```

    printf("Incorrect Input\n");
    exit(0);
}

// There will be i*(i+1)/2 glasses till ith row (including ith row)
float glass[i * (i + 1) / 2];

// Initialize all glasses as empty
memset(glass, 0, sizeof(glass));

// Put all water in first glass
int index = 0;
glass[index] = X;

// Now let the water flow to the downward glasses till the
// amount of water X is not 0 and row number is less than or
// equal to i (given row)
for (int row = 1; row <= i && X != 0.0; ++row)
{
    // Fill glasses in a given row. Number of columns in a row
    // is equal to row number
    for (int col = 1; col <= row; ++col, ++index)
    {
        // Get the water from current glass
        X = glass[index];

        // Keep the amount less than or equal to
        // capacity in current glass
        glass[index] = (X >= 1.0f) ? 1.0f : X;

        // Get the remaining amount
        X = (X >= 1.0f) ? (X - 1) : 0.0f;

        // Distribute the remaining amount to the down two glasses
        glass[index + row] += X / 2;
        glass[index + row + 1] += X / 2;
    }
}

// The index of jth glass in ith row will be i*(i-1)/2 + j - 1
return glass[i*(i-1)/2 + j - 1];
}

// Driver program to test above function
int main()
{
    int i = 2, j = 2;
    float X = 2.0; // Total amount of water

    printf("Amount of water in jth glass of ith row is: %f",
        findWater(i, j, X));

    return 0;
}

```

Output:

```
Amount of water in jth glass of ith row is: 0.500000
```

Time Complexity: $O(i*(i+1)/2)$ or $O(i^2)$

Auxiliary Space: $O(i*(i+1)/2)$ or $O(i^2)$

This article is compiled by **Rahul** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

96. Program to convert a given number to words

Write code to convert a given number into words. For example, if "1234" is given as input, output should be "one thousand two hundred thirty four".

Source: Round 3, Q1 of [Microsoft Interview | Set 3](#)

Following is C implementation for the same. The code supports numbers up-to 4 digits, i.e., numbers from 0 to 9999. Idea is to create arrays that store individual parts of output strings. One array is used for single digits, one for numbers from 10 to 19, one for 20, 30, 40, 50, .. etc, and one for powers of 10.

The given number is divided in two parts: first two digits and last two digits, and the two parts are printed separately.

```
/* Program to print a given number in words. The program handles
   numbers from 0 to 9999 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/* A function that prints given number in words */
void convert_to_words(char *num)
{
    int len = strlen(num); // Get number of digits in given number

    /* Base cases */
    if (len == 0) {
        fprintf(stderr, "empty string\n");
        return;
    }
    if (len > 4) {
        fprintf(stderr, "Length more than 4 is not supported\n");
        return;
    }

    /* The first string is not used, it is to make array indexing simple
    char *single_digits[] = { "zero", "one", "two", "three", "four",
                             "five", "six", "seven", "eight", "nine"}

    /* The first string is not used, it is to make array indexing simple
    char *two_digits[] = {"", "ten", "eleven", "twelve", "thirteen", "fourteen",
                        "fifteen", "sixteen", "seventeen", "eighteen", "nineteen"}

    /* The first two strings are not used, they are to make array indexing simple
    char *tens_multiple[] = {"", "", "twenty", "thirty", "forty", "fifty", "sixty", "seventy", "eighty", "ninety"}
    */
```

```

        "sixty", "seventy", "eighty", "ninety");

char *tens_power[] = {"hundred", "thousand"};

/* Used for debugging purpose only */
printf("\n%s: ", num);

/* For single digit number */
if (len == 1) {
    printf("%s\n", single_digits[*num - '0']);
    return;
}

/* Iterate while num is not '\0' */
while (*num != '\0') {

    /* Code path for first 2 digits */
    if (len >= 3) {
        if (*num - '0' != 0) {
            printf("%s ", single_digits[*num - '0']);
            printf("%s ", tens_power[len-3]); // here len can be 3
        }
        --len;
    }

    /* Code path for last 2 digits */
    else {
        /* Need to explicitly handle 10-19. Sum of the two digits
        used as index of "two_digits" array of strings */
        if (*num == '1') {
            int sum = *num - '0' + *(num + 1) - '0';
            printf("%s\n", two_digits[sum]);
            return;
        }

        /* Need to explicitly handle 20 */
        else if (*num == '2' && *(num + 1) == '0') {
            printf("twenty\n");
            return;
        }

        /* Rest of the two digit numbers i.e., 21 to 99 */
        else {
            int i = *num - '0';
            printf("%s ", i? tens_multiple[i]: "");
            ++num;
            if (*num != '0')
                printf("%s ", single_digits[*num - '0']);
        }
    }
    ++num;
}
}

```

```

/* Driver program to test above function */
int main(void)
{
    convert_to_words("9923");
    convert_to_words("523");
    convert_to_words("89");
    convert_to_words("8989");

    return 0;
}

```

```
return 0;
```

```
}
```

Output:

```
9923: nine thousand nine hundred twenty three
523: five hundred twenty three
89: eighty nine
8989: eight thousand nine hundred eighty nine
```

This article is compiled by **Narendra Kangralkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

97. [TopTalent.in] Dhananjay Sathe Talks About His GSoC Experience And How To Hack It



Google Summer of Code (GSoC) is the brain child of the Google founders Larry Page and Sergey Brin and was first held in the summer of 2005 with an aim to provide some of the most challenging open source projects to the [top talent](#). Not so surprisingly it is one of the most sought after programs and some of the best organizations in the world have now opened up their doors for brilliant students from different parts of the world to collaborate and contribute on

their projects. This year for the first time since inception, the highest number of GSoC participants (227) came from India. IITs, IIITs, BITS Pilani are ruling the roost among colleges. We had a chance to interact with Dhananjay Sathe who did GSoC for the second time in a row. Let's see what he has to say and how you can benefit from his experience.

What is the selection procedure for GSoC and what goes into creating a credible application?

The selection procedure for the Google Summer of Code program is quite straight forward. Technically all you need is to be above the age of 18 and to be enrolled in any academic program in any of the 193 odd countries. This means your CGPA, your branch, your stream (yes even folks perusing law, commerce or any other field can apply) are completely immaterial. Folks perusing research or a PhD can apply too (quite a few actually do).

You essentially need to come up with a new feature set, fix some nagging bugs or port some code in most cases. You need to write a proposal for the same, it is then voted on

both at Google and the concerned organization and they select the most promising applications. So be sure the idea is something nifty, cool and yet achievable in that period.

As for the application itself make sure you give a concrete plan and road-map. Clearly state what your deliverables are and WHEN, HOW you plan to achieve them. Mention all the open source contributions and other programming experiences you have (back it up with links to the code, adds a LOT of credibility). Make sure you have your grammar right, communication is an extremely important part of GSoC. Last but not the least mention why you would like to work on the project and what they, the end users and you stand to gain out of it, this shows the thought process, reasoning and commitment.

How did you decide on your project and what prompted this decision?

I have been an avid linux user and FOSS enthusiast since around 2003 when I first got Linux to boot up on my PC. Using it as a primary OS made me familiar to all the communities, software and norms involved in this kind of development. It also made me aware of what problems one faces and what new stuff could be done. I had this habit, perhaps out of frustration with open source software back in those days, if my app crashed or lacked some functionality I would try to debug it or try and add in the new feature. Samba is a great and powerful tool to have but unfortunately it has a 8500 line man page and can be quite intimidating to new users. I found this a major issue in people around me on campus and I thought samba Gtk was a great way to work on solving that issue. Of course the required background knowledge of Gtk, python and a decent idea of what samba was and how it works were the final things that culminated in my app to Samba.org

What are the benefits of getting into Gsoc and how does it help further while applying for a job?

In one word HUGE! If done right, it is perhaps one of the world's best experiences for aspiring developers. It exposes you to some of the best developers on the planet. You learn a lot more than you can think possible. You get to deal with people from multiple time zones and ideologies. You learn about code development practices, version control, communication, the code base itself and numerous other minor but very important skills required in the real world but completely missing from the scope of formal academics. The benefits show in you, you have much better ideas, practices and experience than most of your contemporaries. Also the folks hiring you realize the value of it all. It gets much simpler to get involved in new open source projects and further your passion and skillset. You develop a ton of contacts and gain access to information and opportunities you otherwise wouldn't know about .Last but not the least, it's a huge brownie point to have something concerned with Google and Open Source on your CV.

How should one prepare if they want to land a GSoC project? How was your journey and what's next?

Follow your passion for software development, learn something new every week, hack it.

The last point is most essential. People often try reading books and learning a new language or toolkit for the sake of it. Don't do that, learn the bare basics, get hold of the source, hack it up and learn as you move ahead. Google has a great search engine and project wikis have a goldmine of information, make the most of it. It's never too late to start, but of course the earlier the better.

I had a blast during my two Summer of Code projects, it was great fun and a lot of learning too. I gained a LOT from it. I will now be working on the next generation of the internet and robotics – cloud robotics at ETH Zurich for my bachelors thesis on some exciting stuff with the open source cloud engine. It's been one heck of a journey.

This article is powered by [TopTalent.in](#) - A high end Job portal for students and alumni of Premier Colleges in India. Sign up now for free exclusive access to top notch jobs in India and abroad. Get in touch with them via [facebook](#), [twitter](#) or [linkedin](#). If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks

98. Efficient program to calculate e^x

The value of **Exponential Function** e^x can be expressed using following **Taylor Series**.

$$e^x = 1 + x/1! + x^2/2! + x^3/3! + \dots$$

How to efficiently calculate the sum of above series?

The series can be re-written as

$$e^x = 1 + (x/1) (1 + (x/2) (1 + (x/3) (\dots)))$$

Let the sum needs to be calculated for n terms, we can calculate sum using following loop.

```
for (i = n - 1, sum = 1; i > 0; --i)
    sum = 1 + x * sum / i;
```

Following is implementation of the above idea.

```
// Efficient program to calculate e raise to the power x
#include <stdio.h>

//Returns approximate value of e^x using sum of first n terms of Taylor
float exponential(int n, float x)
{
    float sum = 1.0f; // initialize sum of series

    for (int i = n - 1; i > 0; --i )
        sum = 1 + x * sum / i;

    return sum;
}

// Driver program to test above function
int main()
{
    int n = 10;
    float x = 1.0f;
    printf("e^x = %f", exponential(n, x));
    return 0;
}
```

Output:

```
e^x = 2.718282
```

This article is compiled by **Rahul** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

99. Dynamic Programming | Set 24 (Optimal Binary Search Tree)

Given a sorted array $keys[0.. n-1]$ of search keys and an array $freq[0.. n-1]$ of frequency counts, where $freq[i]$ is the number of searches to $keys[i]$. Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.

Let us first define the cost of a BST. The cost of a BST node is level of that node multiplied by its frequency. Level of root is 1.

Example 1

Input: $keys[] = \{10, 12\}$, $freq[] = \{34, 50\}$

There can be following two possible BSTs



Frequency of searches of 10 and 12 are 34 and 50 respectively.

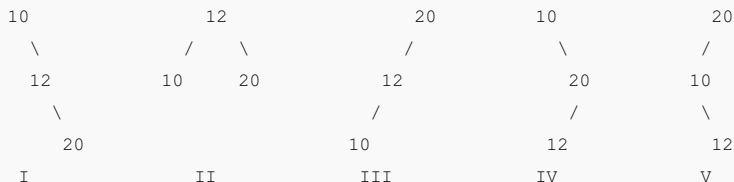
The cost of tree I is $34*1 + 50*2 = 134$

The cost of tree II is $50*1 + 34*2 = 118$

Example 2

Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50}

There can be following possible BSTs



Among all possible BSTs, cost of the fifth BST is minimum.

Cost of the fifth BST is $1*50 + 2*34 + 3*8 = 142$

1) Optimal Substructure:

The optimal cost for freq[i..j] can be recursively calculated using following formula.

$$optCost(i, j) = \sum_{k=i}^j freq[k] + \min_{r=i}^j [optCost(i, r-1) + optCost(r+1, j)]$$

We need to calculate **optCost(0, n-1)** to find the result.

The idea of above formula is simple, we one by one try all nodes as root (r varies from i to j in second term). When we make *r*th node as root, we recursively calculate optimal cost from i to r-1 and r+1 to j.

We add sum of frequencies from i to j (see first term in the above formula), this is added because every search will go through root and one comparison will be done for every search.

2) Overlapping Subproblems

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

```
// A naive recursive implementation of optimal binary search tree prob.
#include <stdio.h>
#include <limits.h>

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j);

// A recursive function to calculate cost of optimal binary search tree
int optCost(int freq[], int i, int j)
{
    // Base cases
    if (j < i)        // If there are no elements in this subarray
        return 0;
    if (j == i)      // If there is one element in this subarray
        return freq[i];

    // Get sum of freq[i], freq[i+1], ... freq[j]
    int fsum = sum(freq, i, j);

    // Initialize minimum value
```

```

int min = INT_MAX;

// One by one consider all elements as root and recursively find co
// of the BST, compare the cost with min and update min if needed
for (int r = i; r <= j; ++r)
{
    int cost = optCost(freq, i, r-1) + optCost(freq, r+1, j);
    if (cost < min)
        min = cost;
}

// Return minimum value
return min + fsum;
}

// The main function that calculates minimum cost of a Binary Search T
// It mainly uses optCost() to find the optimal cost.
int optimalSearchTree(int keys[], int freq[], int n)
{
    // Here array keys[] is assumed to be sorted in increasing order.
    // If keys[] is not sorted, then add code to sort keys, and rearr
    // freq[] accordingly.
    return optCost(freq, 0, n-1);
}

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j)
{
    int s = 0;
    for (int k = i; k <= j; k++)
        s += freq[k];
    return s;
}

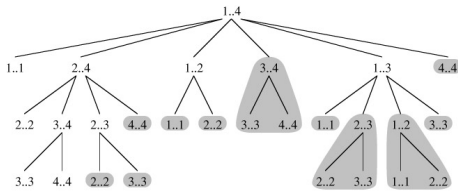
// Driver program to test above functions
int main()
{
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is %d ", optimalSearchTree(keys, freq,
    return 0;
}

```

Output:

```
Cost of Optimal BST is 142
```

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. We can see many subproblems being repeated in the following recursion tree for freq[1..4].



Since same subproblems are called again, this problem has Overlapping Subproblems property. So optimal BST 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 `cost[][]` in bottom up manner.

Dynamic Programming Solution

Following is C/C++ implementation for optimal BST problem using Dynamic Programming. We use an auxiliary array `cost[n][n]` to store the solutions of subproblems. `cost[0][n-1]` will hold the final result. The challenge in implementation is, all diagonal values must be filled first, then the values which lie on the line just above the diagonal. In other words, we must first fill all `cost[i][i]` values, then all `cost[i][i+1]` values, then all `cost[i][i+2]` values. So how to fill the 2D array in such manner? The idea used in the implementation is same as [Matrix Chain Multiplication problem](#), we use a variable 'L' for chain length and increment 'L', one by one. We calculate column number 'j' using the values of 'i' and 'L'.

// Dynamic Programming code for Optimal Binary Search Tree Problem

```
#include <stdio.h>
#include <limits.h>
```

```
// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j);
```

```
/* A Dynamic Programming based function that calculates minimum cost of
a Binary Search Tree. */
```

```
int optimalSearchTree(int keys[], int freq[], int n)
{
```

```
    /* Create an auxiliary 2D matrix to store results of subproblems */
    int cost[n][n];
```

```
    /* cost[i][j] = Optimal cost of binary search tree that can be
       formed from keys[i] to keys[j].
       cost[0][n-1] will store the resultant cost */
```

```
    // For a single key, cost is equal to frequency of the key
```

```
    for (int i = 0; i < n; i++)
        cost[i][i] = freq[i];
```

```
    // Now we need to consider chains of length 2, 3, ... .
    // L is chain length.
```

```
    for (int L=2; L<=n; L++)
    {
```

```
        // i is row number in cost[][]
        for (int i=0; i<=n-L+1; i++)
        {
```

```
            // Get column number j from row number i and chain length
            int j = i+L-1;
```

```

        cost[i][j] = INT_MAX;

        // Try making all keys in interval keys[i..j] as root
        for (int r=i; r<=j; r++)
        {
            // c = cost when keys[r] becomes root of this subtree
            int c = ((r > i)? cost[i][r-1]:0) +
                    ((r < j)? cost[r+1][j]:0) +
                    sum(freq, i, j);
            if (c < cost[i][j])
                cost[i][j] = c;
        }
    }
    return cost[0][n-1];
}

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j)
{
    int s = 0;
    for (int k = i; k <=j; k++)
        s += freq[k];
    return s;
}

// Driver program to test above functions
int main()
{
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is %d ", optimalSearchTree(keys, freq,
    return 0;
}

```

Output:

```
Cost of Optimal BST is 142
```

Notes

1) The time complexity of the above solution is $O(n^4)$. The time complexity can be easily reduced to $O(n^3)$ by pre-calculating sum of frequencies instead of calling sum() again and again.

2) In the above solutions, we have computed optimal cost only. The solutions can be easily modified to store the structure of BSTs also. We can create another auxiliary array of size n to store the structure of tree. All we need to do is, store the chosen 'r' in the innermost loop.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

100. Dynamic Programming | Set 25 (Subset Sum Problem)

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

```
Examples: set[] = {3, 34, 4, 12, 5, 2}, sum = 9
```

```
Output: True //There is a subset (4, 5) with sum 9.
```

Let `isSubSetSum(int set[], int n, int sum)` be the function to find whether there is a subset of `set[]` with sum equal to *sum*. *n* is the number of elements in `set[]`.

The `isSubSetSum` problem can be divided into two subproblems

...a) Include the last element, recur for $n = n-1$, $sum = sum - set[n-1]$

...b) Exclude the last element, recur for $n = n-1$.

If any of the above the above subproblems return true, then return true.

Following is the recursive formula for `isSubSetSum()` problem.

```
isSubSetSum(set, n, sum) = isSubSetSum(set, n-1, sum) ||  
                           isSubSetSum(arr, n-1, sum-set[n-1])
```

Base Cases:

```
isSubSetSum(set, n, sum) = false, if sum > 0 and n == 0
```

```
isSubSetSum(set, n, sum) = true, if sum == 0
```

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

```
// A recursive solution for subset sum problem
#include <stdio.h>

// Returns true if there is a subset of set[] with sum equal to given
bool isSubsetSum(int set[], int n, int sum)
{
    // Base Cases
    if (sum == 0)
        return true;
    if (n == 0 && sum != 0)
        return false;

    // If last element is greater than sum, then ignore it
    if (set[n-1] > sum)
        return isSubsetSum(set, n-1, sum);

    /* else, check if sum can be obtained by any of the following
       (a) including the last element
       (b) excluding the last element */
    return isSubsetSum(set, n-1, sum) || isSubsetSum(set, n-1, sum-set[n-1]);
}

// Driver program to test above function
int main()
{
    int set[] = {3, 34, 4, 12, 5, 2};
    int sum = 9;
    int n = sizeof(set)/sizeof(set[0]);
    if (isSubsetSum(set, n, sum) == true)
        printf("Found a subset with given sum");
    else
        printf("No subset with given sum");
    return 0;
}
```

Output:

```
Found a subset with given sum
```

The above solution may try all subsets of given set in worst case. Therefore time complexity of the above solution is exponential. The problem is in-fact **NP-Complete** (There is no known polynomial time solution for this problem).

We can solve the problem in Pseudo-polynomial time using Dynamic programming.

We create a boolean 2D table subset[][] and fill it in bottom up manner. The value of subset[i][j] will be true if there is a subset of set[0..j-1] with sum equal to i., otherwise false. Finally, we return subset[sum][n]

```
// A Dynamic Programming solution for subset sum problem
#include <stdio.h>
```

```
// Returns true if there is a subset of set[] with sum equal to given
bool isSubsetSum(int set[], int n, int sum)
{
    // The value of subset[i][j] will be true if there is a subset of
    // with sum equal to i
    bool subset[sum+1][n+1];

    // If sum is 0, then answer is true
    for (int i = 0; i <= n; i++)
        subset[0][i] = true;

    // If sum is not 0 and set is empty, then answer is false
    for (int i = 1; i <= sum; i++)
        subset[i][0] = false;

    // Fill the subset table in bottom up manner
    for (int i = 1; i <= sum; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            subset[i][j] = subset[i][j-1];
            if (i >= set[j-1])
                subset[i][j] = subset[i][j] || subset[i - set[j-1]][j-1];
        }
    }

    /* // uncomment this code to print table
    for (int i = 0; i <= sum; i++)
    {
        for (int j = 0; j <= n; j++)
            printf ("%4d", subset[i][j]);
        printf("\n");
    } */

    return subset[sum][n];
}
```

```
// Driver program to test above function
int main()
{
    int set[] = {3, 34, 4, 12, 5, 2};
    int sum = 9;
    int n = sizeof(set)/sizeof(set[0]);
    if (isSubsetSum(set, n, sum) == true)
        printf("Found a subset with given sum");
    else
        printf("No subset with given sum");
    return 0;
}
```

Output:

```
Found a subset with given sum
```

Time complexity of the above solution is $O(\text{sum} \times n)$.

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above.

101. Measure one litre using two vessels and infinite water supply

There are two vessels of capacities 'a' and 'b' respectively. We have infinite water supply. Give an efficient algorithm to make exactly 1 litre of water in one of the vessels. You can throw all the water from any vessel any point of time. Assume that 'a' and 'b' are Coprimes.

Following are the steps:

Let V1 be the vessel of capacity 'a' and V2 be the vessel of capacity 'b' and 'a' is smaller than 'b'.

1) Do following while the amount of water in V1 is not 1.

....a) If V1 is empty, then completely fill V1

....b) Transfer water from V1 to V2. If V2 becomes full, then keep the remaining water in V1 and empty V2

2) V1 will have 1 litre after termination of loop in step 1. Return.

Following is C++ implementation of the above algorithm.

```
/* Sample run of the Algo for V1 with capacity 3 and V2 with capacity 4
1. Fill V1:           V1 = 3, V2 = 0
2. Transfer from V1 to V2, and fill V1:  V1 = 3, V2 = 3
2. Transfer from V1 to V2, and fill V1:  V1 = 3, V2 = 6
3. Transfer from V1 to V2, and empty V2: V1 = 2, V2 = 0
4. Transfer from V1 to V2, and fill V1:  V1 = 3, V2 = 2
5. Transfer from V1 to V2, and fill V1:  V1 = 3, V2 = 5
6. Transfer from V1 to V2, and empty V2: V1 = 1, V2 = 0
7. Stop as V1 now contains 1 litre.
```

Note that V2 was made empty in steps 3 and 6 because it became full */

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

// A utility function to get GCD of two numbers
int gcd(int a, int b) { return b? gcd(b, a % b) : a; }

// Class to represent a Vessel
class Vessel
{
    // A vessel has capacity, and current amount of water in it
    int capacity, current;
public:
    // Constructor: initializes capacity as given, and current as 0
    Vessel(int capacity) { this->capacity = capacity; current = 0; }

    // The main function to fill one litre in this vessel. Capacity of
    // must be greater than this vessel and two capacities must be co-
    void makeOneLitre(Vessel &V2);
```



```

    // Fills vessel with given amount and returns the amount of water
    // transferred to it. If the vessel becomes full, then the vessel
    // is made empty.
    int transfer(int amount);
};

```

```

// The main function to fill one litre in this vessel. Capacity
// of V2 must be greater than this vessel and two capacities
// must be coprime
void Vessel::makeOneLitre(Vessel &V2)
{
    // solution exists iff a and b are co-prime
    if (gcd(capacity, V2.capacity) != 1)
        return;

    while (current != 1)
    {
        // fill A (smaller vessel)
        if (current == 0)
            current = capacity;

        cout << "Vessel 1: " << current << "    Vessel 2: "
              << V2.current << endl;

        // Transfer water from V1 to V2 and reduce current of V1 by
        // the amount equal to transferred water
        current = current - V2.transfer(current);
    }

    // Finally, there will be 1 litre in vessel 1
    cout << "Vessel 1: " << current << "    Vessel 2: "
          << V2.current << endl;
}

```

```

// Fills vessel with given amount and returns the amount of water
// transferred to it. If the vessel becomes full, then the vessel
// is made empty
int Vessel::transfer(int amount)
{
    // If the vessel can accommodate the given amount
    if (current + amount < capacity)
    {
        current += amount;
        return amount;
    }

    // If the vessel cannot accommodate the given amount, then
    // store the amount of water transferred
    int transferred = capacity - current;

    // Since the vessel becomes full, make the vessel
    // empty so that it can be filled again
    current = 0;

    return transferred;
}

// Driver program to test above function
int main()
{
    int a = 3, b = 7; // a must be smaller than b
}

```

```

// Create two vessels of capacities a and b
Vessel V1(a), V2(b);

// Get 1 litre in first vessel
V1.makeOneLitre(V2);

return 0;
}

```

Output:

```

Vessel 1: 3   Vessel 2: 0
Vessel 1: 3   Vessel 2: 3
Vessel 1: 3   Vessel 2: 6
Vessel 1: 2   Vessel 2: 0
Vessel 1: 3   Vessel 2: 2
Vessel 1: 3   Vessel 2: 5
Vessel 1: 1   Vessel 2: 0

```

How does this work?

To prove that the algorithm works, we need to prove that after certain number of iterations in the while loop, we will get 1 litre in V1.

Let 'a' be the capacity of vessel V1 and 'b' be the capacity of V2. Since we repeatedly transfer water from V1 to V2 until V2 becomes full, we will have 'a - b (mod a)' water in V1 when V2 becomes full first time. Once V2 becomes full, it is emptied. We will have 'a - 2b (mod a)' water in V1 when V2 is full second time. We repeat the above steps, and get 'a - nb (mod a)' water in V1 after the vessel V2 is filled and emptied 'n' times. We need to prove that the value of 'a - nb (mod a)' will be 1 for a finite integer 'n'. To prove this, let us consider the following property of coprime numbers.

For any two **coprime integers** 'a' and 'b', the integer 'b' has a **multiplicative inverse** modulo 'a'. In other words, there exists an integer 'y' such that $by \equiv 1 \pmod{a}$ (See 3rd point [here](#)). After '(a - 1)*y' iterations, we will have 'a - [(a-1)*y*b (mod a)]' water in V1, the value of this expression is 'a - [(a - 1) * 1] mod a' which is 1. So the algorithm converges and we get 1 litre in V1.

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

102. Efficient program to print all prime factors of a given number

Given a number n, write an efficient function to print all **prime factors** of n. For example, if the input number is 12, then output should be "2 2 3". And if the input number is 315, then output should be "3 3 5 7".

Following are the steps to find all prime factors.

- 1) While n is divisible by 2, print 2 and divide n by 2.
- 2) After step 1, n must be odd. Now start a loop from i = 3 to square root of n. While i divides n, print i and divide n by i, increment i by 2 and continue.
- 3) If n is a prime number and is greater than 2, then n will not become 1 by above two steps. So print n if it is greater than 2.

```
// Program to print all prime factors
# include <stdio.h>
# include <math.h>

// A function to print all prime factors of a given number n
void primeFactors(int n)
{
    // Print the number of 2s that divide n
    while (n%2 == 0)
    {
        printf("%d ", 2);
        n = n/2;
    }

    // n must be odd at this point. So we can skip one element (Note
    for (int i = 3; i <= sqrt(n); i = i+2)
    {
        // While i divides n, print i and divide n
        while (n%i == 0)
        {
            printf("%d ", i);
            n = n/i;
        }
    }

    // This condition is to handle the case when n is a prime number
    // greater than 2
    if (n > 2)
        printf ("%d ", n);
}

/* Driver program to test above function */
int main()
{
    int n = 315;
    primeFactors(n);
    return 0;
}
```

Output:

3 3 5 7

How does this work?

The steps 1 and 2 take care of composite numbers and step 3 takes care of prime numbers. To prove that the complete algorithm works, we need to prove that steps 1 and 2 actually take care of composite numbers. This is clear that step 1 takes care of even numbers. And after step 1, all remaining prime factor must be odd (difference of two prime factors must be at least 2), this explains why i is incremented by 2.

Now the main part is, the loop runs till square root of n not till. To prove that this optimization works, let us consider the following property of composite numbers.
Every composite number has at least one prime factor less than or equal to square root of itself.

This property can be proved using counter statement. Let a and b be two factors of n such that $a * b = n$. If both are greater than \sqrt{n} , then $a.b > \sqrt{n} * \sqrt{n}$ which contradicts the expression " $a * b = n$ ".

In step 2 of the above algorithm, we run a loop and do following in loop

- a) Find the least prime factor i (must be less than \sqrt{n})
- b) Remove all occurrences i from n by repeatedly dividing n by i .
- c) Repeat steps a and b for divided n and $i = i + 2$. The steps a and b are repeated till n becomes either 1 or a prime number.

Thanks to **Vishwas Garg** for suggesting the above algorithm. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

103. Pattern Searching | Set 8 (Suffix Tree Introduction)

Given a text $txt[0..n-1]$ and a pattern $pat[0..m-1]$, write a function `search(char pat[], char txt[])` that prints all occurrences of $pat[]$ in $txt[]$. You may assume that $n > m$.

Preprocess Pattern or Preprocess Text?

We have discussed the following algorithms in the previous posts:

KMP Algorithm

Rabin Karp Algorithm

Finite Automata based Algorithm

Boyer Moore Algorithm

All of the above algorithms preprocess the pattern to make the pattern searching faster. The best time complexity that we could get by preprocessing pattern is $O(n)$ where n is length of the text. In this post, we will discuss an approach that preprocesses the text. A suffix tree is built of the text. After preprocessing text (building suffix tree of text), we can search any pattern in $O(m)$ time where m is length of the pattern.

Imagine you have stored complete work of **William Shakespeare** and preprocessed it. You can search any string in the complete work in time just proportional to length of the pattern. This is really a great improvement because length of pattern is generally much smaller than text.

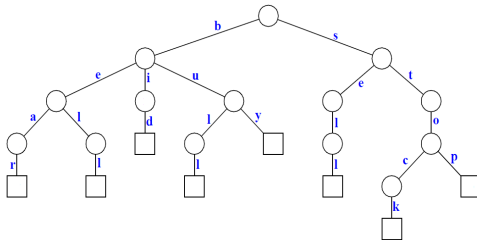
Preprocessing of text may become costly if the text changes frequently. It is good for fixed text or less frequently changing text though.

A Suffix Tree for a given text is a compressed trie for all suffixes of the given text.

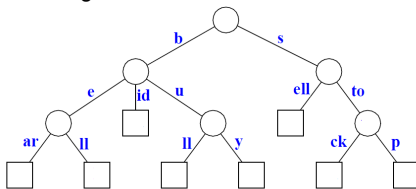
We have discussed **Standard Trie**. Let us understand **Compressed Trie** with the following array of words.

```
{bear, bell, bid, bull, buy, sell, stock, stop}
```

Following is standard trie for the above input set of words.



Following is the compressed trie. Compress Trie is obtained from standard trie by joining chains of single nodes. The nodes of a compressed trie can be stored by storing index ranges at the nodes.



How to build a Suffix Tree for a given text?

As discussed above, Suffix Tree is compressed trie of all suffixes, so following are very abstract steps to build a suffix tree from given text.

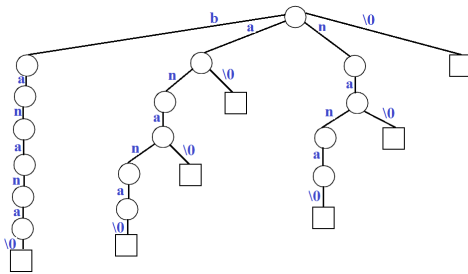
- 1) Generate all suffixes of given text.
- 2) Consider all suffixes as individual words and build a compressed trie.

Let us consider an example text "banana\0" where "\0" is string termination character.

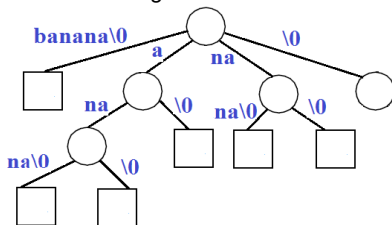
Following are all suffixes of "banana\0"

```
banana\0
anana\0
nana\0
ana\0
na\0
a\0
\0
```

If we consider all of the above suffixes as individual words and build a trie, we get following.



If we join chains of single nodes, we get the following compressed trie, which is the Suffix Tree for given text "banana\0"



Please note that above steps are just to manually create a Suffix Tree. We will be discussing actual algorithm and implementation in a separate post.

How to search a pattern in the built suffix tree?

We have discussed above how to build a Suffix Tree which is needed as a preprocessing step in pattern searching. Following are abstract steps to search a pattern in the built Suffix Tree.

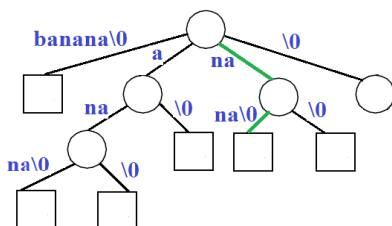
1) Starting from the first character of the pattern and root of Suffix Tree, do following for every character.

.....**a)** For the current character of pattern, if there is an edge from the current node of suffix tree, follow the edge.

.....**b)** If there is no edge, print "pattern doesn't exist in text" and return.

2) If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern, then print "Pattern found".

Let us consider the example pattern as "nan" to see the searching process. Following diagram shows the path followed for searching "nan" or "nana".



How does this work?

Every pattern that is present in text (or we can say every substring of text) must be a prefix of one of all possible suffixes. The statement seems complicated, but it is a simple statement, we just need to take an example to check validity of it.

Applications of Suffix Tree

Suffix tree can be used for a wide range of problems. Following are some famous problems where Suffix Trees provide optimal time complexity solution.

- 1) Pattern Searching
- 2) Finding the longest repeated substring
- 3) Finding the longest common substring
- 4) Finding the longest palindrome in a string

There are many more applications. See [this](#) for more details.

Ukkonen's Suffix Tree Construction is discussed in following articles:

- [Ukkonen's Suffix Tree Construction – Part 1](#)
- [Ukkonen's Suffix Tree Construction – Part 2](#)
- [Ukkonen's Suffix Tree Construction – Part 3](#)
- [Ukkonen's Suffix Tree Construction – Part 4](#)
- [Ukkonen's Suffix Tree Construction – Part 5](#)
- [Ukkonen's Suffix Tree Construction – Part 6](#)

References:

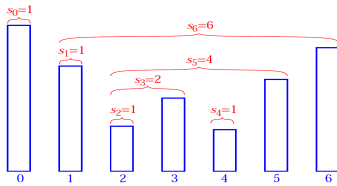
- <http://fbim.fh-regensburg.de/~saj39122/sal/skript/progr/pr45102/Tries.pdf>
- <http://www.cs.ucf.edu/~shzhang/Combio12/lec3.pdf>
- <http://www.allisons.org/ll/AlgDS/Tree/Suffix/>

104. The Stock Span Problem

The stock span problem is a financial problem where we have a series of n daily price quotes for a stock and we need to calculate span of stock's price for all n days.

The span S_i of the stock's price on a given day i is defined as the maximum number of consecutive days just before the given day, for which the price of the stock on the current day is less than or equal to its price on the given day.

For example, if an array of 7 days prices is given as {100, 80, 60, 70, 60, 75, 85}, then the span values for corresponding 7 days are {1, 1, 1, 2, 1, 4, 6}



A Simple but inefficient method

Traverse the input price array. For every element being visited, traverse elements on left of it and increment the span value of it while elements on the left side are smaller.

Following is implementation of this method.

```
// A brute force method to calculate stock span values
#include <stdio.h>
```

```
// Fills array S[] with span values
void calculateSpan(int price[], int n, int S[])
{
    // Span value of first day is always 1
    S[0] = 1;

    // Calculate span value of remaining days by linearly checking prev
    for (int i = 1; i < n; i++)
    {
        S[i] = 1; // Initialize span value

        // Traverse left while the next element on left is smaller than i
        for (int j = i-1; (j>=0)&&(price[i]>=price[j]); j--)
            S[i]++;
    }
}
```

```
// A utility function to print elements of array
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
}
```

```
// Driver program to test above function
int main()
{
    int price[] = {10, 4, 5, 90, 120, 80};
    int n = sizeof(price)/sizeof(price[0]);
    int S[n];

    // Fill the span values in array S[]
    calculateSpan(price, n, S);

    // print the calculated span values
    printArray(S, n);

    return 0;
}
```

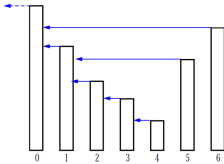
Time Complexity of the above method is $O(n^2)$. We can calculate stock span values in

$O(n)$ time.

A Linear Time Complexity Method

We see that $S[i]$ on day i can be easily computed if we know the closest day preceding i , such that the price is greater than on that day than the price on day i . If such a day exists, let's call it $h(i)$, otherwise, we define $h(i) = -1$.

The span is now computed as $S[i] = i - h(i)$. See the following diagram.



To implement this logic, we use a stack as an abstract data type to store the days i , $h(i)$, $h(h(i))$ and so on. When we go from day $i-1$ to i , we pop the days when the price of the stock was less than or equal to $price[i]$ and then push the value of day i back into the stack.

Following is C++ implementation of this method.

```

// a linear time solution for stock span problem
#include <iostream>
#include <stack>
using namespace std;

// A brute force method to calculate stock span values
void calculateSpan(int price[], int n, int S[])
{
    // Create a stack and push index of first element to it
    stack<int> st;
    st.push(0);

    // Span value of first element is always 1
    S[0] = 1;

    // Calculate span values for rest of the elements
    for (int i = 1; i < n; i++)
    {
        // Pop elements from stack while stack is not empty and top of
        // stack is smaller than price[i]
        while (!st.empty() && price[st.top()] < price[i])
            st.pop();

        // If stack becomes empty, then price[i] is greater than all elements
        // on left of it, i.e., price[0], price[1],..price[i-1]. Else price[i]
        // is greater than elements after top of stack
        S[i] = (st.empty())? (i + 1) : (i - st.top());

        // Push this element to stack
        st.push(i);
    }
}

// A utility function to print elements of array
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}

// Driver program to test above function
int main()
{
    int price[] = {10, 4, 5, 90, 120, 80};
    int n = sizeof(price)/sizeof(price[0]);
    int S[n];

    // Fill the span values in array S[]
    calculateSpan(price, n, S);

    // print the calculated span values
    printArray(S, n);

    return 0;
}

```

Output:

```
1 1 2 4 5 1
```

Time Complexity: $O(n)$. It seems more than $O(n)$ at first look. If we take a closer look, we can observe that every element of array is added and removed from stack at most once. So there are total $2n$ operations at most. Assuming that a stack operation takes $O(1)$ time, we can say that the time complexity is $O(n)$.

Auxiliary Space: $O(n)$ in worst case when all elements are sorted in decreasing order.

References:

[http://en.wikipedia.org/wiki/Stack_\(abstract_data_type\)#The_Stock_Span_Problem](http://en.wikipedia.org/wiki/Stack_(abstract_data_type)#The_Stock_Span_Problem)

<http://crypto.cs.mcgill.ca/~crepeau/CS250/2004/Stack-I.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

105. Design an efficient data structure for given operations

Design a Data Structure for the following operations. The data structure should be efficient enough to accommodate the operations according to their frequency.

- 1) findMin() : Returns the minimum item.
Frequency: Most frequent
- 2) findMax() : Returns the maximum item.
Frequency: Most frequent
- 3) deleteMin() : Delete the minimum item.
Frequency: Moderate frequent
- 4) deleteMax() : Delete the maximum item.
Frequency: Moderate frequent
- 5) Insert() : Inserts an item.
Frequency: Least frequent
- 6) Delete() : Deletes an item.
Frequency: Least frequent.

A **simple solution** is to maintain a sorted array where smallest element is at first position and largest element is at last. The time complexity of findMin(), findMAX() and deleteMax() is $O(1)$. But time complexities of deleteMin(), insert() and delete() will be $O(n)$.

Can we do the most frequent two operations in $O(1)$ and other operations in $O(\text{Log}n)$ time?

The idea is to use two binary heaps (one max and one min heap). The main challenge is, while deleting an item, we need to delete from both min-heap and max-heap. So, we need some kind of mutual data structure. In the following design, we have used doubly linked list as a mutual data structure. The doubly linked list contains all input items and indexes of corresponding min and max heap nodes. The nodes of min and max heaps store addresses of nodes of doubly linked list. The root node of min heap stores the address of minimum item in doubly linked list. Similarly, root of max heap stores address of maximum item in doubly linked list. Following are the details of operations.

1) findMax(): We get the address of maximum value node from root of Max Heap. So this is a $O(1)$ operation.

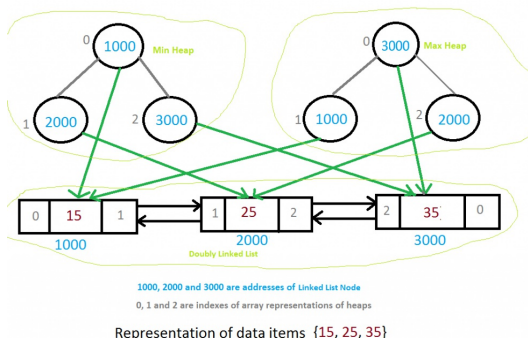
1) findMin(): We get the address of minimum value node from root of Min Heap. So this is a $O(1)$ operation.

3) deleteMin(): We get the address of minimum value node from root of Min Heap. We use this address to find the node in doubly linked list. From the doubly linked list, we get node of Max Heap. We delete node from all three. We can delete a node from doubly linked list in $O(1)$ time. delete() operations for max and min heaps take $O(\text{Log}n)$ time.

4) deleteMax(): is similar to deleteMin()

5) Insert() We always insert at the beginning of linked list in $O(1)$ time. Inserting the address in Max and Min Heaps take $O(\text{Log}n)$ time. So overall complexity is $O(\text{Log}n)$

6) Delete() We first search the item in Linked List. Once the item is found in $O(n)$ time, we delete it from linked list. Then using the indexes stored in linked list, we delete it from Min Heap and Max Heaps in $O(\text{Log}n)$ time. *So overall complexity of this operation is $O(n)$. The Delete operation can be optimized to $O(\text{Log}n)$ by using a balanced binary search tree instead of doubly linked list as a mutual data structure. Use of balanced binary search will not effect time complexity of other operations as it will act as a mutual data structure like doubly Linked List.*



Following is C implementation of the above data structure.

```

// C program for efficient data structure
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A node of doubly linked list
struct LNode
{
    int data;
    int minHeapIndex;
    int maxHeapIndex;
    struct LNode *next, *prev;
};

// Structure for a doubly linked list
struct List
{
    struct LNode *head;
};

// Structure for min heap
struct MinHeap
{
    int size;
    int capacity;
    struct LNode* *array;
};

// Structure for max heap
struct MaxHeap
{
    int size;
    int capacity;
    struct LNode* *array;
};

// The required data structure
struct MyDS
{
    struct MinHeap* minHeap;
    struct MaxHeap* maxHeap;
    struct List* list;
};

// Function to swap two integers
void swapData(int* a, int* b)
{ int t = *a;  *a = *b;  *b = t; }

// Function to swap two List nodes
void swapLNode(struct LNode** a, struct LNode** b)
{ struct LNode* t = *a; *a = *b; *b = t; }

// A utility function to create a new List node
struct LNode* newLNode(int data)
{
    struct LNode* node =
        (struct LNode*) malloc(sizeof(struct LNode));
    node->minHeapIndex = node->maxHeapIndex = -1;
    node->data = data;
    node->prev = node->next = NULL;
    return node;
}

```

```

}

// Utility function to create a max heap of given capacity
struct MaxHeap* createMaxHeap(int capacity)
{
    struct MaxHeap* maxHeap =
        (struct MaxHeap*) malloc(sizeof(struct MaxHeap));
    maxHeap->size = 0;
    maxHeap->capacity = capacity;
    maxHeap->array =
        (struct LNode**) malloc(maxHeap->capacity * sizeof(struct LNode*))
    return maxHeap;
}

// Utility function to create a min heap of given capacity
struct MinHeap* createMinHeap(int capacity)
{
    struct MinHeap* minHeap =
        (struct MinHeap*) malloc(sizeof(struct MinHeap));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array =
        (struct LNode**) malloc(minHeap->capacity * sizeof(struct LNode*))
    return minHeap;
}

// Utility function to create a List
struct List* createList()
{
    struct List* list =
        (struct List*) malloc(sizeof(struct List));
    list->head = NULL;
    return list;
}

// Utility function to create the main data structure
// with given capacity
struct MyDS* createMyDS(int capacity)
{
    struct MyDS* myDS =
        (struct MyDS*) malloc(sizeof(struct MyDS));
    myDS->minHeap = createMinHeap(capacity);
    myDS->maxHeap = createMaxHeap(capacity);
    myDS->list = createList();
    return myDS;
}

// Some basic operations for heaps and List
int isMaxHeapEmpty(struct MaxHeap* heap)
{ return (heap->size == 0); }

int isMinHeapEmpty(struct MinHeap* heap)
{ return heap->size == 0; }

int isMaxHeapFull(struct MaxHeap* heap)
{ return heap->size == heap->capacity; }

int isMinHeapFull(struct MinHeap* heap)
{ return heap->size == heap->capacity; }

int isListEmpty(struct List* list)
{ return !list->head; }

```

```

int hasOnlyOneLNode(struct List* list)
{
    return !list->head->next && !list->head->prev; }

// The standard minHeapify function. The only thing it does extra
// is swapping indexes of heaps inside the List
void minHeapify(struct MinHeap* minHeap, int index)
{
    int smallest, left, right;
    smallest = index;
    left = 2 * index + 1;
    right = 2 * index + 2;

    if ( minHeap->array[left] &&
        left < minHeap->size &&
        minHeap->array[left]->data < minHeap->array[smallest]->data
    )
        smallest = left;

    if ( minHeap->array[right] &&
        right < minHeap->size &&
        minHeap->array[right]->data < minHeap->array[smallest]->data
    )
        smallest = right;

    if (smallest != index)
    {
        // First swap indexes inside the List using address
        // of List nodes
        swapData(&(minHeap->array[smallest]->minHeapIndex),
                &(minHeap->array[index]->minHeapIndex));

        // Now swap pointers to List nodes
        swapLNode(&minHeap->array[smallest],
                 &minHeap->array[index]);

        // Fix the heap downward
        minHeapify(minHeap, smallest);
    }
}

// The standard maxHeapify function. The only thing it does extra
// is swapping indexes of heaps inside the List
void maxHeapify(struct MaxHeap* maxHeap, int index)
{
    int largest, left, right;
    largest = index;
    left = 2 * index + 1;
    right = 2 * index + 2;

    if ( maxHeap->array[left] &&
        left < maxHeap->size &&
        maxHeap->array[left]->data > maxHeap->array[largest]->data
    )
        largest = left;

    if ( maxHeap->array[right] &&
        right < maxHeap->size &&
        maxHeap->array[right]->data > maxHeap->array[largest]->data
    )
        largest = right;
}

```

```

    if (largest != index)
    {
        // First swap indexes inside the List using address
        // of List nodes
        swapData(&maxHeap->array[largest]->maxHeapIndex,
                &maxHeap->array[index]->maxHeapIndex);

        // Now swap pointers to List nodes
        swapLNode(&maxHeap->array[largest],
                &maxHeap->array[index]);

        // Fix the heap downward
        maxHeapify(maxHeap, largest);
    }
}

// Standard function to insert an item in Min Heap
void insertMinHeap(struct MinHeap* minHeap, struct LNode* temp)
{
    if (isMinHeapFull(minHeap))
        return;

    ++minHeap->size;
    int i = minHeap->size - 1;
    while (i && temp->data < minHeap->array[(i - 1) / 2]->data )
    {
        minHeap->array[i] = minHeap->array[(i - 1) / 2];
        minHeap->array[i]->minHeapIndex = i;
        i = (i - 1) / 2;
    }

    minHeap->array[i] = temp;
    minHeap->array[i]->minHeapIndex = i;
}

// Standard function to insert an item in Max Heap
void insertMaxHeap(struct MaxHeap* maxHeap, struct LNode* temp)
{
    if (isMaxHeapFull(maxHeap))
        return;

    ++maxHeap->size;
    int i = maxHeap->size - 1;
    while (i && temp->data > maxHeap->array[(i - 1) / 2]->data )
    {
        maxHeap->array[i] = maxHeap->array[(i - 1) / 2];
        maxHeap->array[i]->maxHeapIndex = i;
        i = (i - 1) / 2;
    }

    maxHeap->array[i] = temp;
    maxHeap->array[i]->maxHeapIndex = i;
}

// Function to find minimum value stored in the main data structure
int findMin(struct MyDS* myDS)
{
    if (isMinHeapEmpty(myDS->minHeap))
        return INT_MAX;

    return myDS->minHeap->array[0]->data;
}

```



```

}

// Function to find maximum value stored in the main data structure
int findMax(struct MyDS* myDS)
{
    if (isMaxHeapEmpty(myDS->maxHeap))
        return INT_MIN;

    return myDS->maxHeap->array[0]->data;
}

// A utility function to remove an item from linked list
void removeLNode(struct List* list, struct LNode** temp)
{
    if (hasOnlyOneLNode(list))
        list->head = NULL;

    else if (!(*temp)->prev) // first node
    {
        list->head = (*temp)->next;
        (*temp)->next->prev = NULL;
    }
    // any other node including last
    else
    {
        (*temp)->prev->next = (*temp)->next;
        // last node
        if ((*temp)->next)
            (*temp)->next->prev = (*temp)->prev;
    }
    free(*temp);
    *temp = NULL;
}

// Function to delete maximum value stored in the main data structure
void deleteMax(struct MyDS* myDS)
{
    MinHeap *minHeap = myDS->minHeap;
    MaxHeap *maxHeap = myDS->maxHeap;

    if (isMaxHeapEmpty(maxHeap))
        return;
    struct LNode* temp = maxHeap->array[0];

    // delete the maximum item from maxHeap
    maxHeap->array[0] =
        maxHeap->array[maxHeap->size - 1];
    --maxHeap->size;
    maxHeap->array[0]->maxHeapIndex = 0;
    maxHeapify(maxHeap, 0);

    // remove the item from minHeap
    minHeap->array[temp->minHeapIndex] = minHeap->array[minHeap->size
    --minHeap->size;
    minHeap->array[temp->minHeapIndex]->minHeapIndex = temp->minHeapIndex;
    minHeapify(minHeap, temp->minHeapIndex);

    // remove the node from List
    removeLNode(myDS->list, &temp);
}

// Function to delete minimum value stored in the main data structure
void deleteMin(struct MyDS* myDS)

```

```

// Function to delete an item from List. The function also
// removes item from min and max heaps
void Delete(struct MyDS* myDS, int item)
{
    MinHeap *minHeap = myDS->minHeap;
    MaxHeap *maxHeap = myDS->maxHeap;

    if (isMinHeapEmpty(minHeap))
        return;
    struct LNode* temp = minHeap->array[0];

    // delete the minimum item from minHeap
    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeap->array[0]->minHeapIndex = 0;
    minHeapify(minHeap, 0);

    // remove the item from maxHeap
    maxHeap->array[temp->maxHeapIndex] = maxHeap->array[maxHeap->size - 1];
    --maxHeap->size;
    maxHeap->array[temp->maxHeapIndex]->maxHeapIndex = temp->maxHeapIndex;
    maxHeapify(maxHeap, temp->maxHeapIndex);

    // remove the node from List
    removeLNode(myDS->list, &temp);
}

```

```

// Function to enList an item to List
void insertAtHead(struct List* list, struct LNode* temp)
{
    if (isListEmpty(list))
        list->head = temp;

    else
    {
        temp->next = list->head;
        list->head->prev = temp;
        list->head = temp;
    }
}

```

```

// Function to delete an item from List. The function also
// removes item from min and max heaps
void Delete(struct MyDS* myDS, int item)
{
    MinHeap *minHeap = myDS->minHeap;
    MaxHeap *maxHeap = myDS->maxHeap;

    if (isListEmpty(myDS->list))
        return;

    // search the node in List
    struct LNode* temp = myDS->list->head;
    while (temp && temp->data != item)
        temp = temp->next;

    // if item not found
    if (!temp || temp && temp->data != item)
        return;

    // remove item from min heap
    minHeap->array[temp->minHeapIndex] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeap->array[temp->minHeapIndex]->minHeapIndex = temp->minHeapIndex;
    minHeapify(minHeap, temp->minHeapIndex);
}

```

```

    // remove item from max heap
    maxHeap->array[temp->maxHeapIndex] = maxHeap->array[maxHeap->size
    --maxHeap->size;
    maxHeap->array[temp->maxHeapIndex]->maxHeapIndex = temp->maxHeapIndex;
    maxHeapify(maxHeap, temp->maxHeapIndex);

    // remove node from List
    removeLNode(myDS->list, &temp);
}

// insert operation for main data structure
void Insert(struct MyDS* myDS, int data)
{
    struct LNode* temp = newLNode(data);

    // insert the item in List
    insertAtHead(myDS->list, temp);

    // insert the item in min heap
    insertMinHeap(myDS->minHeap, temp);

    // insert the item in max heap
    insertMaxHeap(myDS->maxHeap, temp);
}

```

```

// Driver program to test above functions
int main()
{
    struct MyDS *myDS = createMyDS(10);
    // Test Case #1
    /*Insert(myDS, 10);
    Insert(myDS, 2);
    Insert(myDS, 32);
    Insert(myDS, 40);
    Insert(myDS, 5);*/

    // Test Case #2
    Insert(myDS, 10);
    Insert(myDS, 20);
    Insert(myDS, 30);
    Insert(myDS, 40);
    Insert(myDS, 50);

    printf("Maximum = %d \n", findMax(myDS));
    printf("Minimum = %d \n\n", findMin(myDS));

    deleteMax(myDS); // 50 is deleted
    printf("After deleteMax()\n");
    printf("Maximum = %d \n", findMax(myDS));
    printf("Minimum = %d \n\n", findMin(myDS));

    deleteMin(myDS); // 10 is deleted
    printf("After deleteMin()\n");
    printf("Maximum = %d \n", findMax(myDS));
    printf("Minimum = %d \n\n", findMin(myDS));

    Delete(myDS, 40); // 40 is deleted
    printf("After Delete()\n");
    printf("Maximum = %d \n", findMax(myDS));
    printf("Minimum = %d \n", findMin(myDS));
}

```

return 0;

```
}  
    return 0;  
}
```

Output:

```
Maximum = 50  
Minimum = 10  
  
After deleteMax()  
Maximum = 40  
Minimum = 10  
  
After deleteMin()  
Maximum = 40  
Minimum = 20  
  
After Delete()  
Maximum = 30  
Minimum = 20
```

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

106. Generate n-bit Gray Codes

Given a number n, generate bit patterns from 0 to $2^n - 1$ such that successive patterns differ by one bit.

Examples:

```
Following is 2-bit sequence (n = 2)  
00 01 11 10  
Following is 3-bit sequence (n = 3)  
000 001 011 010 110 111 101 100  
And Following is 4-bit sequence (n = 4)  
0000 0001 0011 0010 0110 0111 0101 0100 1100 1101 1111  
1110 1010 1011 1001 1000
```

The above sequences are **Gray Codes** of different widths. Following is an interesting pattern in Gray Codes.

n-bit Gray Codes can be generated from list of (n-1)-bit Gray codes using following

steps.

- 1) Let the list of $(n-1)$ -bit Gray codes be L1. Create another list L2 which is reverse of L1.
- 2) Modify the list L1 by prefixing a '0' in all codes of L1.
- 3) Modify the list L2 by prefixing a '1' in all codes of L2.
- 4) Concatenate L1 and L2. The concatenated list is required list of n -bit Gray codes.

For example, following are steps for generating the 3-bit Gray code list from the list of 2-bit Gray code list.

L1 = {00, 01, 11, 10} (List of 2-bit Gray Codes)

L2 = {10, 11, 01, 00} (Reverse of L1)

Prefix all entries of L1 with '0', L1 becomes {000, 001, 011, 010}

Prefix all entries of L2 with '1', L2 becomes {110, 111, 101, 100}

Concatenate L1 and L2, we get {000, 001, 011, 010, 110, 111, 101, 100}

To generate n -bit Gray codes, we start from list of 1 bit Gray codes. The list of 1 bit Gray code is {0, 1}. We repeat above steps to generate 2 bit Gray codes from 1 bit Gray codes, then 3-bit Gray codes from 2-bit Gray codes till the number of bits becomes equal to n . Following is C++ implementation of this approach.

```

// C++ program to generate n-bit Gray codes
#include <iostream>
#include <string>
#include <vector>
using namespace std;

// This function generates all n bit Gray codes and prints the
// generated codes
void generateGrayarr(int n)
{
    // base case
    if (n <= 0)
        return;

    // 'arr' will store all generated codes
    vector<string> arr;

    // start with one-bit pattern
    arr.push_back("0");
    arr.push_back("1");

    // Every iteration of this loop generates 2*i codes from previously
    // generated i codes.
    int i, j;
    for (i = 2; i < (1<<n); i = i<<1)
    {
        // Enter the previously generated codes again in arr[] in reverse
        // order. Now arr[] has double number of codes.
        for (j = i-1 ; j >= 0 ; j--)
            arr.push_back(arr[j]);

        // append 0 to the first half
        for (j = 0 ; j < i ; j++)
            arr[j] = "0" + arr[j];

        // append 1 to the second half
        for (j = i ; j < 2*i ; j++)
            arr[j] = "1" + arr[j];
    }

    // print contents of arr[]
    for (i = 0 ; i < arr.size() ; i++)
        cout << arr[i] << endl;
}

// Driver program to test above function
int main()
{
    generateGrayarr(4);
    return 0;
}

```

Output

```

000
001
011
010
110

```

111

101

100

This article is compiled by **Ravi Chandra Enaganti**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

107. Interview Experiences at D. E. Shaw & Co.

I had three rounds:

My first round was telephonic interview and it lasted for about an hour and the questions covered entire breadth of core CS subjects, there were questions on Operating Systems, questions on some SQL queries, Computer Networks, Computer Organisation, Digital Design, some algorithms and there were some questions based on the projects i did (that was specified in Resume) i was asked questions on Android since i did a project on Android, and at last they asked me a puzzle on probability.

In The second round (which was face to face interview), the emphasis was mostly on designing algorithms for some new problems, and they expected me to give optimal solutions, and they asked me to design web server and what are the various challenges you will be facing.

The third round was the most critical one and the questions were mostly on C++ concepts and some operating system design issues (e.g., spin locks, semaphores, etc.,) and there were even questions on storage classes and how do you think they might be implemented, there were some questions on algorithms which were simple but there were many in-depth analysis on those questions (e.g., In finding the loop in the linked-list what happens when the fast pointer takes 3 steps instead of 2 does this algorithm does this algorithm converges, what is the condition under which the algorithm converges), they asked me to code some design patterns, and i was asked some puzzles to solve

I got placed in some of the dream companies, and in this journey i should say, <http://geeksforgeeks.org/> has a major role, i am following this site since my first day of graduation and i should say this one will give complete picture of what are the various questions you might be asked in various companies, along with great emphasis on core CS concepts. Thanks geeksforgeeks, i will say this is the one stop that will help you to land into some of the prestigious firms

If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

108. Intersection of n sets

Given n sets of integers of different sizes. Each set may contain duplicates also. How to find the intersection of all the sets. If an element is present multiple times in all the sets, it should be added that many times in the result.

For example, consider there are three sets {1,2,2,3,4} {2,2,3,5,6} {1,3,2,2,6}. The intersection of the given sets should be {2,2,3}

The following is an efficient approach to solve this problem.

1. Sort all the sets.
2. Take the smallest set, and insert all the elements, and their frequencies into a map.
3. For each element in the map do the following
 -a. If the element is not present in any set, ignore it
 -b. Find the frequency of the element in each set (using binary search). If it less than the frequency in the map, update the frequency
 -c. If the element is found in all the sets, add it to the result.

Here is the C++ implementation of the above approach.

```
// C++ program to find intersection of n sets
#include <iostream>
#include <vector>
#include <algorithm>
#include <map>
using namespace std;

// The main function that receives a set of sets as parameter and
// returns a set containing intersection of all sets
vector<int> getIntersection(vector< vector<int> > &sets)
{
    vector<int> result; // To store the resultant set
    int smallSetInd = 0; // Initialize index of smallest set
    int minSize = sets[0].size(); // Initialize size of smallest set

    // sort all the sets, and also find the smallest set
    for (int i = 1 ; i < sets.size() ; i++)
    {
        // sort this set
        sort(sets[i].begin(), sets[i].end());

        // update minSize, if needed
        if (minSize > sets[i].size())
        {
            minSize = sets[i].size();
            smallSetInd = i;
        }
    }
}
```



```

map<int,int> elementsMap;

// Add all the elements of smallest set to a map, if already present
// update the frequency
for (int i = 0; i < sets[smallSetInd].size(); i++)
{
    if (elementsMap.find( sets[smallSetInd][i] ) == elementsMap.end())
        elementsMap[ sets[smallSetInd][i] ] = 1;
    else
        elementsMap[ sets[smallSetInd][i] ]++;
}

// iterate through the map elements to see if they are present in
// remaining sets
map<int,int>::iterator it;
for (it = elementsMap.begin(); it != elementsMap.end(); ++it)
{
    int elem = it->first;
    int freq = it->second;

    bool bFound = true;

    // Iterate through all sets
    for (int j = 0 ; j < sets.size() ; j++)
    {
        // If this set is not the smallest set, then do binary search
        if (j != smallSetInd)
        {
            // If the element is found in this set, then find its frequency
            if (binary_search( sets[j].begin(), sets[j].end(), elem ))
            {
                int lInd = lower_bound(sets[j].begin(), sets[j].end(), elem) - sets[j].begin();
                int rInd = upper_bound(sets[j].begin(), sets[j].end(), elem) - sets[j].begin();

                // Update the minimum frequency, if needed
                if ((rInd - lInd) < freq)
                    freq = rInd - lInd;
            }
            // If the element is not present in any set, then no need
            // to proceed for this element.
            else
            {
                bFound = false;
                break;
            }
        }
    }

    // If element was found in all sets, then add it to result 'frequency'
    if (bFound)
    {
        for (int k = 0; k < freq; k++)
            result.push_back(elem);
    }
}
return result;
}

```

```

// A utility function to print a set of elements
void printset(vector <int> set)
{

```

```

    for (int i = 0 ; i < set.size() ; i++)
        cout<<set[i]<<" ";
}

```

```

// Test case
void TestCase1()
{
    vector < vector <int> > sets;
    vector <int> set1;
    set1.push_back(1);
    set1.push_back(1);
    set1.push_back(2);
    set1.push_back(2);
    set1.push_back(5);

    sets.push_back(set1);

    vector <int> set2;
    set2.push_back(1);
    set2.push_back(1);
    set2.push_back(4);
    set2.push_back(3);
    set2.push_back(5);
    set2.push_back(9);

    sets.push_back(set2);

    vector <int> set3;
    set3.push_back(1);
    set3.push_back(1);
    set3.push_back(2);
    set3.push_back(3);
    set3.push_back(5);
    set3.push_back(6);

    sets.push_back(set3);

    vector <int> r = getIntersection(sets);

    printset(r);
}

// Driver program to test above functions
int main()
{
    TestCase1();
    return 0;
}

```

Output:

```
1 1 5
```

Time Complexity: Let there be 'n' lists, and average size of lists be 'm'. Sorting phase takes $O(n * m * \log m)$ time (Sorting n lists with average length m). Searching phase takes $O(m * n * \log m)$ time . (for each element in a list, we are searching n lists with log m time). So the overall time complexity is $O(n * m * \log m)$.

Auxiliary Space: $O(m)$ space for storing the map.

This article is compiled by **Ravi Chandra Enaganti**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

109. Check if a number is Palindrome

Given an integer, write a function that returns true if the given number is palindrome, else false. For example, 12321 is palindrome, but 1451 is not palindrome.

Let the given number be *num*. A simple method for this problem is to first **reverse digits of *num***, then compare the reverse of *num* with *num*. If both are same, then return true, else false.

Following is an interesting method inspired from method#2 of [this](#) post. The idea is to create a copy of *num* and recursively pass the copy by reference, and pass *num* by value. In the recursive calls, divide *num* by 10 while moving down the recursion tree. While moving up the recursion tree, divide the copy by 10. When they meet in a function for which all child calls are over, the last digit of *num* will be *i*th digit from the beginning and the last digit of copy will be *i*th digit from the end.

```
// A recursive C++ program to check whether a given number is
// palindrome or not
#include <stdio.h>

// A function that returns true only if num contains one digit
int oneDigit(int num)
{
    // comparison operation is faster than division operation.
    // So using following instead of "return num / 10 == 0;"
    return (num >= 0 && num < 10);
}

// A recursive function to find out whether num is palindrome
// or not. Initially, dupNum contains address of a copy of num.
bool isPalUtil(int num, int* dupNum)
{
    // Base case (needed for recursion termination): This statement
    // mainly compares the first digit with the last digit
    if (oneDigit(num))
        return (num == (*dupNum) % 10);

    // This is the key line in this method. Note that all recursive
    // calls have a separate copy of num, but they all share same copy
    // of *dupNum. We divide num while moving up the recursion tree
    if (!isPalUtil(num/10, dupNum))
        return false;

    // The following statements are executed when we move up the
```

```

// recursion call tree
*dupNum /= 10;

// At this point, if num%10 contains i'th digit from beginning,
// then (*dupNum)%10 contains i'th digit from end
return (num % 10 == (*dupNum) % 10);
}

// The main function that uses recursive function isPalUtil() to
// find out whether num is palindrome or not
int isPal(int num)
{
    // If num is negative, make it positive
    if (num < 0)
        num = -num;

    // Create a separate copy of num, so that modifications made
    // to address dupNum don't change the input number.
    int *dupNum = new int(num); // *dupNum = num

    return isPalUtil(num, dupNum);
}

// Driver program to test above functions
int main()
{
    int n = 12321;
    isPal(n)? printf("Yes\n"): printf("No\n");

    n = 12;
    isPal(n)? printf("Yes\n"): printf("No\n");

    n = 88;
    isPal(n)? printf("Yes\n"): printf("No\n");

    n = 8999;
    isPal(n)? printf("Yes\n"): printf("No\n");
    return 0;
}

```

Output:

```

Yes
No
Yes
No

```

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Given an integer, write a function that returns true if the given number is palindrome, else false. For example, 12321 is palindrome, but 1451 is not palindrome.

Let the given number be *num*. A simple method for this problem is to first reverse digits of *num*, then compare the reverse of *num* with *num*. If both are same, then return true, else false.

Following is an interesting method inspired from method#2 of [this](#) post. The idea is to create a copy of *num* and recursively pass the copy by reference, and pass *num* by value. In the recursive calls, divide *num* by 10 while moving down the recursion tree. While moving up the recursion tree, divide the copy by 10. When they meet in a function for which all child calls are over, the last digit of *num* will be *i*th digit from the beginning and the last digit of copy will be *i*th digit from the end.

```
// A recursive C++ program to check whether a given number is
// palindrome or not
#include <stdio.h>

// A function that returns true only if num contains one digit
int oneDigit(int num)
{
    // comparison operation is faster than division operation.
    // So using following instead of "return num / 10 == 0;"
    return (num >= 0 && num < 10);
}

// A recursive function to find out whether num is palindrome
// or not. Initially, dupNum contains address of a copy of num.
bool isPalUtil(int num, int* dupNum)
{
    // Base case (needed for recursion termination): This statement
    // mainly compares the first digit with the last digit
    if (oneDigit(num))
        return (num == (*dupNum) % 10);

    // This is the key line in this method. Note that all recursive
    // calls have a separate copy of num, but they all share same copy
    // of *dupNum. We divide num while moving up the recursion tree
    if (!isPalUtil(num/10, dupNum))
        return false;

    // The following statements are executed when we move up the
    // recursion call tree
    *dupNum /= 10;

    // At this point, if num%10 contains i'th digit from beginning,
    // then (*dupNum)%10 contains i'th digit from end
    return (num % 10 == (*dupNum) % 10);
}

// The main function that uses recursive function isPalUtil() to
// find out whether num is palindrome or not
int isPal(int num)
{
    // If num is negative, make it positive
    if (num < 0)
        num = -num;

    // Create a separate copy of num, so that modifications made
    // to address dupNum don't change the input number.
    int *dupNum = new int(num); // *dupNum = num
}
```

```

    return isPalUtil(num, dupNum);
}

// Driver program to test above functions
int main()
{
    int n = 12321;
    isPal(n)? printf("Yes\n"): printf("No\n");

    n = 12;
    isPal(n)? printf("Yes\n"): printf("No\n");

    n = 88;
    isPal(n)? printf("Yes\n"): printf("No\n");

    n = 8999;
    isPal(n)? printf("Yes\n"): printf("No\n");
    return 0;
}

```

Output:

```

Yes
No
Yes
No

```

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Given an integer, write a function that returns true if the given number is palindrome, else false. For example, 12321 is palindrome, but 1451 is not palindrome.

Let the given number be *num*. A simple method for this problem is to first **reverse digits of num**, then compare the reverse of *num* with *num*. If both are same, then return true, else false.

Following is an interesting method inspired from method#2 of [this](#) post. The idea is to create a copy of *num* and recursively pass the copy by reference, and pass *num* by value. In the recursive calls, divide *num* by 10 while moving down the recursion tree. While moving up the recursion tree, divide the copy by 10. When they meet in a function for which all child calls are over, the last digit of *num* will be ith digit from the beginning and the last digit of copy will be ith digit from the end.

```

// A recursive C++ program to check whether a given number is
// palindrome or not
#include <stdio.h>

// A function that returns true only if num contains one digit
int oneDigit(int num)
{

```

```

{
    // comparison operation is faster than division operation.
    // So using following instead of "return num / 10 == 0;"
    return (num >= 0 && num < 10);
}

// A recursive function to find out whether num is palindrome
// or not. Initially, dupNum contains address of a copy of num.
bool isPalUtil(int num, int* dupNum)
{
    // Base case (needed for recursion termination): This statement
    // mainly compares the first digit with the last digit
    if (oneDigit(num))
        return (num == (*dupNum) % 10);

    // This is the key line in this method. Note that all recursive
    // calls have a separate copy of num, but they all share same copy
    // of *dupNum. We divide num while moving up the recursion tree
    if (!isPalUtil(num/10, dupNum))
        return false;

    // The following statements are executed when we move up the
    // recursion call tree
    *dupNum /= 10;

    // At this point, if num%10 contains i'th digit from beginning,
    // then (*dupNum)%10 contains i'th digit from end
    return (num % 10 == (*dupNum) % 10);
}

// The main function that uses recursive function isPalUtil() to
// find out whether num is palindrome or not
int isPal(int num)
{
    // If num is negative, make it positive
    if (num < 0)
        num = -num;

    // Create a separate copy of num, so that modifications made
    // to address dupNum don't change the input number.
    int *dupNum = new int(num); // *dupNum = num

    return isPalUtil(num, dupNum);
}

// Driver program to test above functions
int main()
{
    int n = 12321;
    isPal(n)? printf("Yes\n"): printf("No\n");

    n = 12;
    isPal(n)? printf("Yes\n"): printf("No\n");

    n = 88;
    isPal(n)? printf("Yes\n"): printf("No\n");

    n = 8999;
    isPal(n)? printf("Yes\n"): printf("No\n");
    return 0;
}

```

Output:

Yes
No
Yes
No

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

112. Dynamic Programming | Set 28 (Minimum insertions to form a palindrome)

Given a string, find the minimum number of characters to be inserted to convert it to palindrome.

Before we go further, let us understand with few examples:

ab: Number of insertions required is 1. **bab**

aa: Number of insertions required is 0. **aa**

abcd: Number of insertions required is 3. **dcbabcd**

abcda: Number of insertions required is 2. **adcbacda** which is same as number of insertions in the substring bcd(Why?).

abcde: Number of insertions required is 4. **edcbabcde**

Let the input string be $str[l.....h]$. The problem can be broken down into three parts:

1. Find the minimum number of insertions in the substring $str[l+1,.....h]$.
2. Find the minimum number of insertions in the substring $str[l,.....h-1]$.
3. Find the minimum number of insertions in the substring $str[l+1,.....h-1]$.

Recursive Solution

The minimum number of insertions in the string $str[l.....h]$ can be given as:

$\text{minInsertions}(str[l+1.....h-1])$ if $str[l]$ is equal to $str[h]$

$\text{min}(\text{minInsertions}(str[l,.....h-1]), \text{minInsertions}(str[l+1.....h])) + 1$ otherwise

How to reuse solutions of subproblems?

We can create a table to store results of subproblems so that they can be used directly if same subproblem is encountered again.

The below table represents the stored values for the string abcde.

```
a b c d e
-----
0 1 2 3 4
0 0 1 2 3
0 0 0 1 2
0 0 0 0 1
0 0 0 0 0
```

How to fill the table?

The table should be filled in diagonal fashion. For the string abcde, 0...4, the following should be order in which the table is filled:

```
Gap = 1:
(0, 1) (1, 2) (2, 3) (3, 4)
```

```
Gap = 2:
(0, 2) (1, 3) (2, 4)
```

```
Gap = 3:
(0, 3) (1, 4)
```

```
Gap = 4:
(0, 4)
```

```

// A Dynamic Programming based program to find minimum number
// insertions needed to make a string palindrome
#include <stdio.h>
#include <string.h>

// A utility function to find minimum of two integers
int min(int a, int b)
{   return a < b ? a : b; }

// A DP function to find minimum number of insertions
int findMinInsertionsDP(char str[], int n)
{
    // Create a table of size n*n. table[i][j] will store
    // minimum number of insertions needed to convert str[i..j]
    // to a palindrome.
    int table[n][n], l, h, gap;

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

    // Fill the table
    for (gap = 1; gap < n; ++gap)
        for (l = 0, h = gap; h < n; ++l, ++h)
            table[l][h] = (str[l] == str[h])? table[l+1][h-1] :
                (min(table[l][h-1], table[l+1][h]) + 1);

    // Return minimum number of insertions for str[0..n-1]
    return table[0][n-1];
}

// Driver program to test above function.
int main()
{
    char str[] = "geeks";
    printf("%d", findMinInsertionsDP(str, strlen(str)));
    return 0;
}

```

Output:

3

Time complexity: $O(N^2)$

Auxiliary Space: $O(N^2)$

Another Dynamic Programming Solution (Variation of Longest Common Subsequence Problem)

The problem of finding minimum insertions can also be solved using Longest Common Subsequence (LCS) Problem. If we find out LCS of string and its reverse, we know how many maximum characters can form a palindrome. We need insert remaining characters. Following are the steps.

- 1) Find the length of LCS of input string and its reverse. Let the length be 'l'.
- 2) The minimum number insertions needed is length of input string minus 'l'.

```

// An LCS based program to find minimum number insertions needed to
// make a string palindrome
#include<stdio.h>
#include <string.h>

/* Utility function to get max of 2 integers */
int max(int a, int b)
{   return (a > b)? a : b; }

/* Returns length of LCS for X[0..m-1], Y[0..n-1].
   See http://goo.gl/bHQVP for details of this function */
int lcs( char *X, char *Y, int m, int n )
{
    int L[n+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];
}

// LCS based function to find minimum number of insertions
int findMinInsertionsLCS(char str[], int n)
{
    // Create another string to store reverse of 'str'
    char rev[n+1];
    strcpy(rev, str);
    strrev(rev);

    // The output is length of string minus length of lcs of
    // str and its reverse
    return (n - lcs(str, rev, n, n));
}

// Driver program to test above functions
int main()
{
    char str[] = "geeks";
    printf("%d", findMinInsertionsLCS(str, strlen(str)));
    return 0;
}

```

Output:

Time complexity of this method is also $O(n^2)$ and this method also requires $O(n^2)$ extra space.

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

113. Dynamic Programming | Set 30 (Dice Throw)

Given n dice each with m faces, numbered from 1 to m , find the number of ways to get sum X . X is the summation of values on each face when all the dice are thrown.

The **Naive approach** is to find all the possible combinations of values from n dice and keep on counting the results that sum to X .

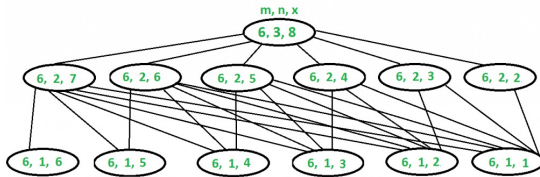
This problem can be efficiently solved using **Dynamic Programming (DP)**.

```
Let the function to find X from n dice is: Sum(m, n, X)
The function can be represented as:
Sum(m, n, X) = Finding Sum (X - 1) from (n - 1) dice plus 1 from nth dice
               + Finding Sum (X - 2) from (n - 1) dice plus 2 from nth dice
               + Finding Sum (X - 3) from (n - 1) dice plus 3 from nth dice
               .....
               .....
               .....
               + Finding Sum (X - m) from (n - 1) dice plus m from nth dice

So we can recursively write Sum(m, n, x) as following
Sum(m, n, X) = Sum(m, n - 1, X - 1) +
               Sum(m, n - 1, X - 2) +
               ..... +
               Sum(m, n - 1, X - m)
```

Why DP approach?

The above problem exhibits overlapping subproblems. See the below diagram. Also, see [this](#) recursive implementation. Let there be 3 dice, each with 6 faces and we need to find the number of ways to get sum 8:



$$\text{Sum}(6, 3, 8) = \text{Sum}(6, 2, 7) + \text{Sum}(6, 2, 6) + \text{Sum}(6, 2, 5) + \\ \text{Sum}(6, 2, 4) + \text{Sum}(6, 2, 3) + \text{Sum}(6, 2, 2)$$

To evaluate $\text{Sum}(6, 3, 8)$, we need to evaluate $\text{Sum}(6, 2, 7)$ which can recursively written as following:

$$\text{Sum}(6, 2, 7) = \text{Sum}(6, 1, 6) + \text{Sum}(6, 1, 5) + \text{Sum}(6, 1, 4) + \\ \text{Sum}(6, 1, 3) + \text{Sum}(6, 1, 2) + \text{Sum}(6, 1, 1)$$

We also need to evaluate $\text{Sum}(6, 2, 6)$ which can recursively written as following:

$$\text{Sum}(6, 2, 6) = \text{Sum}(6, 1, 5) + \text{Sum}(6, 1, 4) + \text{Sum}(6, 1, 3) + \\ \text{Sum}(6, 1, 2) + \text{Sum}(6, 1, 1)$$

.....

$$\text{Sum}(6, 2, 2) = \text{Sum}(6, 1, 1)$$

Please take a closer look at the above recursion. The sub-problems in **RED** are solved first time and sub-problems in **BLUE** are solved again (exhibit overlapping sub-problems). Hence, storing the results of the solved sub-problems saves time.

Following is C++ implementation of Dynamic Programming approach.

```

// C++ program to find number of ways to get sum 'x' with 'n'
// dice where every dice has 'm' faces
#include <iostream>
#include <string.h>
using namespace std;

// The main function that returns number of ways to get sum 'x'
// with 'n' dice and 'm' with m faces.
int findWays(int m, int n, int x)
{
    // Create a table to store results of subproblems. One extra
    // row and column are used for simplicity (Number of dice
    // is directly used as row index and sum is directly used
    // as column index). The entries in 0th row and 0th column
    // are never used.
    int table[n + 1][x + 1];
    memset(table, 0, sizeof(table)); // Initialize all entries as 0

    // Table entries for only one dice
    for (int j = 1; j <= m && j <= x; j++)
        table[1][j] = 1;

    // Fill rest of the entries in table using recursive relation
    // i: number of dice, j: sum
    for (int i = 2; i <= n; i++)
        for (int j = 1; j <= x; j++)
            for (int k = 1; k <= m && k < j; k++)
                table[i][j] += table[i-1][j-k];

    /* Uncomment these lines to see content of table
    for (int i = 0; i <= n; i++)
    {
        for (int j = 0; j <= x; j++)
            cout << table[i][j] << " ";
        cout << endl;
    } */
    return table[n][x];
}

// Driver program to test above functions
int main()
{
    cout << findWays(4, 2, 1) << endl;
    cout << findWays(2, 2, 3) << endl;
    cout << findWays(6, 3, 8) << endl;
    cout << findWays(4, 2, 5) << endl;
    cout << findWays(4, 3, 5) << endl;

    return 0;
}

```

Output:

```

0
2
21
4
6

```

Time Complexity: $O(m * n * x)$ where m is number of faces, n is number of dice and x is

given sum.

We can add following two conditions at the beginning of findWays() to improve performance of program for extreme cases (x is too high or x is too low)

```
// When x is so high that sum can not go beyond x even when we
// get maximum value in every dice throw.
if (m*n <= x)
    return (m*n == x);

// When x is too low
if (n >= x)
    return (n == x);
```

With above conditions added, time complexity becomes $O(1)$ when $x \geq m*n$ or when $x \leq n$.

Exercise:

Extend the above algorithm to find the probability to get $\text{Sum} > X$.

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

114. Expression Evaluation

Evaluate an expression represented by a String. Expression can contain parentheses, you can assume parentheses are well-matched. For simplicity, you can assume only binary operations allowed are +, -, *, and /. Arithmetic Expressions can be written in one of three forms:

Infix Notation: Operators are written between the operands they operate on, e.g. $3 + 4$.

Prefix Notation: Operators are written before the operands, e.g. $+ 3 4$

Postfix Notation: Operators are written after operands.

Infix Expressions are harder for Computers to evaluate because of the additional work needed to decide precedence. Infix notation is how expressions are written and recognized by humans and, generally, input to programs. Given that they are harder to evaluate, they are generally converted to one of the two remaining forms. A very well known algorithm for converting an infix notation to a postfix notation is [Shunting Yard Algorithm by Edgar Dijkstra](#). This algorithm takes as input an Infix Expression and produces a queue that has this expression converted to a postfix notation. Same algorithm can be modified so that it outputs result of evaluation of expression instead of a queue. Trick is using two stacks instead of one, one for operands and one for operators. Algorithm was described succinctly on <http://www.cis.upenn.edu/>

matuszek/cit594-2002/Assignments/5-expressions.htm, and is re-produced here. (Note that credit for succinctness goes to author of said page)

1. While there are still tokens to be read in,
 - 1.1 Get the next token.
 - 1.2 If the token is:
 - 1.2.1 A number: push it onto the value stack.
 - 1.2.2 A variable: get its value, and push onto the value stack.
 - 1.2.3 A left parenthesis: push it onto the operator stack.
 - 1.2.4 A right parenthesis:
 - 1 While the thing on top of the operator stack is not a left parenthesis,
 - 1 Pop the operator from the operator stack.
 - 2 Pop the value stack twice, getting two operands.
 - 3 Apply the operator to the operands, in the correct order.
 - 4 Push the result onto the value stack.
 - 2 Pop the left parenthesis from the operator stack, and discard it.
 - 1.2.5 An operator (call it thisOp):
 - 1 While the operator stack is not empty, and the top thing on the operator stack has the same or greater precedence as thisOp,
 - 1 Pop the operator from the operator stack.
 - 2 Pop the value stack twice, getting two operands.
 - 3 Apply the operator to the operands, in the correct order.
 - 4 Push the result onto the value stack.
 - 2 Push thisOp onto the operator stack.
2. While the operator stack is not empty,
 - 1 Pop the operator from the operator stack.
 - 2 Pop the value stack twice, getting two operands.
 - 3 Apply the operator to the operands, in the correct order.
 - 4 Push the result onto the value stack.
3. At this point the operator stack should be empty, and the value stack should have only one value in it, which is the final result.

It should be clear that this algorithm runs in linear time – each number or operator is pushed onto and popped from Stack only once. Also see

<http://www2.lawrence.edu/fast/GREGGJ/CMSC270/Infix.html>,

<http://faculty.cs.niu.edu/~hutchins/csci241/eval.htm>.

Following is Java implementation of above algorithm.

```
/* A Java program to evaluate a given expression where tokens are separated
by space.
```

```
Test Cases:
```

```
"10 + 2 * 6"          ---> 22
"100 * 2 + 12"        ---> 212
"100 * ( 2 + 12 )"    ---> 1400
```

```

    "100 * ( 2 + 12 ) / 14" ---> 100
*/
import java.util.Stack;

public class EvaluateString
{
    public static int evaluate(String expression)
    {
        char[] tokens = expression.toCharArray();

        // Stack for numbers: 'values'
        Stack<Integer> values = new Stack<Integer>();

        // Stack for Operators: 'ops'
        Stack<Character> ops = new Stack<Character>();

        for (int i = 0; i < tokens.length; i++)
        {
            // Current token is a whitespace, skip it
            if (tokens[i] == ' ')
                continue;

            // Current token is a number, push it to stack for numbers
            if (tokens[i] >= '0' && tokens[i] <= '9')
            {
                StringBuffer sbuf = new StringBuffer();
                // There may be more than one digits in number
                while (i < tokens.length && tokens[i] >= '0' && tokens[i] <= '9')
                    sbuf.append(tokens[i++]);
                values.push(Integer.parseInt(sbuf.toString()));
            }

            // Current token is an opening brace, push it to 'ops'
            else if (tokens[i] == '(')
                ops.push(tokens[i]);

            // Closing brace encountered, solve entire brace
            else if (tokens[i] == ')')
            {
                while (ops.peek() != '(')
                    values.push(applyOp(ops.pop(), values.pop(), values.pop()));
                ops.pop();
            }

            // Current token is an operator.

```

```

        else if (tokens[i] == '+' || tokens[i] == '-' ||
                tokens[i] == '*' || tokens[i] == '/')
        {
            // While top of 'ops' has same or greater precedence to current
            // token, which is an operator. Apply operator on top of 'ops'
            // to top two elements in values stack
            while (!ops.empty() && hasPrecedence(tokens[i], ops.peek()))
                values.push(applyOp(ops.pop(), values.pop(), values.pop()));

            // Push current token to 'ops'.
            ops.push(tokens[i]);
        }
    }

    // Entire expression has been parsed at this point, apply remaining
    // ops to remaining values
    while (!ops.empty())
        values.push(applyOp(ops.pop(), values.pop(), values.pop()));

    // Top of 'values' contains result, return it
    return values.pop();
}

// Returns true if 'op2' has higher or same precedence as 'op1',
// otherwise returns false.
public static boolean hasPrecedence(char op1, char op2)
{
    if (op2 == '(' || op2 == ')')
        return false;
    if ((op1 == '*' || op1 == '/') && (op2 == '+' || op2 == '-'))
        return false;
    else
        return true;
}

// A utility method to apply an operator 'op' on operands 'a'
// and 'b'. Return the result.
public static int applyOp(char op, int b, int a)
{
    switch (op)
    {
        case '+':
            return a + b;
        case '-':

```

```

        return a - b;
    case '*':
        return a * b;
    case '/':
        if (b == 0)
            throw new
                UnsupportedOperationException("Cannot divide by zero");
        return a / b;
    }
    return 0;
}

// Driver method to test above methods
public static void main(String[] args)
{
    System.out.println(EvaluateString.evaluate("10 + 2 * 6"));
    System.out.println(EvaluateString.evaluate("100 * 2 + 12"));
    System.out.println(EvaluateString.evaluate("100 * ( 2 + 12 )"));
    System.out.println(EvaluateString.evaluate("100 * ( 2 + 12 ) / 14"));
}
}

```

Output:

```

22
212
1400
100

```

See [this](#) for a sample run with more test cases.

This article is compiled by **Ciphe**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

115. Dynamic Programming | Set 31 (Optimal Strategy for a Game)

Problem statement: Consider a row of n coins of values $v_1 \dots v_n$, where n is even. We play a game against an opponent by alternating turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can definitely win if we move first.

Note: The opponent is as clever as the user.

Let us understand the problem with few examples:

1. 5, 3, 7, 10 : The user collects maximum value as 15(10 + 5)

2. 8, 15, 3, 7 : The user collects maximum value as 22(7 + 15)

Does choosing the best at each move give an optimal solution?

No. In the second example, this is how the game can finish:

1.

.....User chooses 8.

.....Opponent chooses 15.

.....User chooses 7.

.....Opponent chooses 3.

Total value collected by user is 15(8 + 7)

2.

.....User chooses 7.

.....Opponent chooses 8.

.....User chooses 15.

.....Opponent chooses 3.

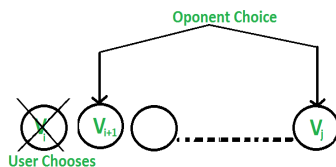
Total value collected by user is 22(7 + 15)

So if the user follows the second game state, maximum value can be collected although the first move is not the best.

There are two choices:

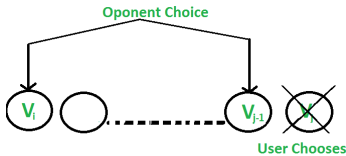
1. The user chooses the i th coin with value V_i : The opponent either chooses $(i+1)$ th coin or j th coin. The opponent intends to choose the coin which leaves the user with minimum value.

i.e. The user can collect the value $V_i + \min(F(i+2, j), F(i+1, j-1))$



2. The user chooses the j th coin with value V_j : The opponent either chooses i th coin or $(j-1)$ th coin. The opponent intends to choose the coin which leaves the user with minimum value.

i.e. The user can collect the value $V_j + \min(F(i+1, j-1), F(i, j-2))$



Following is recursive solution that is based on above two choices. We take the maximum of two choices.

$F(i, j)$ represents the maximum value the user can collect from i 'th coin to j 'th coin.

$$F(i, j) = \text{Max}(V_i + \min(F(i+2, j), F(i+1, j-1)), V_j + \min(F(i+1, j-1), F(i, j-2)))$$

Base Cases

$$\begin{aligned} F(i, j) &= V_i && \text{If } j == i \\ F(i, j) &= \max(V_i, V_j) && \text{If } j == i+1 \end{aligned}$$

Why Dynamic Programming?

The above relation exhibits overlapping sub-problems. In the above relation, $F(i+1, j-1)$ is calculated twice.

```

// C program to find out maximum value from a given sequence of coins
#include <stdio.h>
#include <limits.h>

// Utility functions to get maximum and minimum of two integers
int max(int a, int b) { return a > b ? a : b; }
int min(int a, int b) { return a < b ? a : b; }

// Returns optimal value possible that a player can collect from
// an array of coins of size n. Note that n must be even
int optimalStrategyOfGame(int* arr, int n)
{
    // Create a table to store solutions of subproblems
    int table[n][n], gap, i, j, x, y, z;

    // Fill table using above recursive formula. Note that the table
    // is filled in diagonal fashion (similar to http://goo.gl/PQqoS),
    // from diagonal elements to table[0][n-1] which is the result.
    for (gap = 0; gap < n; ++gap)
    {
        for (i = 0, j = gap; j < n; ++i, ++j)
        {
            // Here x is value of F(i+2, j), y is F(i+1, j-1) and
            // z is F(i, j-2) in above recursive formula
            x = ((i+2) <= j) ? table[i+2][j] : 0;
            y = ((i+1) <= (j-1)) ? table[i+1][j-1] : 0;
            z = (i <= (j-2)) ? table[i][j-2] : 0;

            table[i][j] = max(arr[i] + min(x, y), arr[j] + min(y, z));
        }
    }

    return table[0][n-1];
}

// Driver program to test above function
int main()
{
    int arr1[] = {8, 15, 3, 7};
    int n = sizeof(arr1)/sizeof(arr1[0]);
    printf("%d\n", optimalStrategyOfGame(arr1, n));

    int arr2[] = {2, 2, 2, 2};
    n = sizeof(arr2)/sizeof(arr2[0]);
    printf("%d\n", optimalStrategyOfGame(arr2, n));

    int arr3[] = {20, 30, 2, 2, 2, 10};
    n = sizeof(arr3)/sizeof(arr3[0]);
    printf("%d\n", optimalStrategyOfGame(arr3, n));

    return 0;
}

```

Output:

```

22
4
42

```

Exercise

Your thoughts on the strategy when the user wishes to only win instead of winning with the maximum value. Like above problem, number of coins is even.

Can Greedy approach work quite well and give an optimal solution? Will your answer change if number of coins is odd?

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

116. Random number generator in arbitrary probability distribution fashion

Given n numbers, each with some frequency of occurrence. Return a random number with probability proportional to its frequency of occurrence.

Example:

```
Let following be the given numbers.
```

```
arr[] = {10, 30, 20, 40}
```

```
Let following be the frequencies of given numbers.
```

```
freq[] = {1, 6, 2, 1}
```

```
The output should be
```

```
10 with probability 1/10
```

```
30 with probability 6/10
```

```
20 with probability 2/10
```

```
40 with probability 1/10
```

It is quite clear that the simple random number generator won't work here as it doesn't keep track of the frequency of occurrence.

We need to somehow transform the problem into a problem whose solution is known to us.

One simple method is to take an auxiliary array (say aux[]) and duplicate the numbers according to their frequency of occurrence. Generate a random number(say r) between 0 to Sum-1(including both), where Sum represents summation of frequency array (freq[] in above example). Return the random number aux[r] (Implementation of this method is left as an exercise to the readers).

The limitation of the above method discussed above is huge memory consumption when frequency of occurrence is high. If the input is 997, 8761 and 1, this method is clearly not efficient.

How can we reduce the memory consumption? Following is detailed algorithm that uses $O(n)$ extra space where n is number of elements in input arrays.

1. Take an auxiliary array (say `prefix[]`) of size n .
2. Populate it with prefix sum, such that `prefix[i]` represents sum of numbers from 0 to i .
3. Generate a random number (say r) between 1 to Sum (including both), where Sum represents summation of input frequency array.
4. Find index of Ceil of random number generated in step #3 in the prefix array. Let the index be `indexc`.
5. Return the random number `arr[indexc]`, where `arr[]` contains the input n numbers.

Before we go to the implementation part, let us have quick look at the algorithm with an example:

`arr[]`: {10, 20, 30}

`freq[]`: {2, 3, 1}

`Prefix[]`: {2, 5, 6}

Since last entry in prefix is 6, all possible values of r are [1, 2, 3, 4, 5, 6]

- 1: Ceil is 2. Random number generated is 10.
- 2: Ceil is 2. Random number generated is 10.
- 3: Ceil is 5. Random number generated is 20.
- 4: Ceil is 5. Random number generated is 20.
- 5: Ceil is 5. Random number generated is 20.
- 6: Ceil is 6. Random number generated is 30.

In the above example

10 is generated with probability $2/6$.

20 is generated with probability $3/6$.

30 is generated with probability $1/6$.

How does this work?

Any number `input[i]` is generated as many times as its frequency of occurrence because there exists count of integers in range(`prefix[i - 1]`, `prefix[i]`) is `input[i]`. Like in the above example 3 is generated thrice, as there exists 3 integers 3, 4 and 5 whose ceil is 5.

```

//C program to generate random numbers according to given frequency di
#include <stdio.h>
#include <stdlib.h>

// Utility function to find ceiling of r in arr[l..h]
int findCeil(int arr[], int r, int l, int h)
{
    int mid;
    while (l < h)
    {
        mid = l + ((h - l) >> 1); // Same as mid = (l+h)/2
        (r > arr[mid]) ? (l = mid + 1) : (h = mid);
    }
    return (arr[l] >= r) ? l : -1;
}

// The main function that returns a random number from arr[] according
// distribution array defined by freq[]. n is size of arrays.
int myRand(int arr[], int freq[], int n)
{
    // Create and fill prefix array
    int prefix[n], i;
    prefix[0] = freq[0];
    for (i = 1; i < n; ++i)
        prefix[i] = prefix[i - 1] + freq[i];

    // prefix[n-1] is sum of all frequencies. Generate a random number
    // with value from 1 to this sum
    int r = (rand() % prefix[n - 1]) + 1;

    // Find index of ceiling of r in prefix array
    int indexc = findCeil(prefix, r, 0, n - 1);
    return arr[indexc];
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 2, 3, 4};
    int freq[] = {10, 5, 20, 100};
    int i, n = sizeof(arr) / sizeof(arr[0]);

    // Use a different seed value for every run.
    srand(time(NULL));

    // Let us generate 10 random numbers according to
    // given distribution
    for (i = 0; i < 5; i++)
        printf("%d\n", myRand(arr, freq, n));

    return 0;
}

```

Output: May be different for different runs

4
3
4
4

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

117. Working with Shared Libraries | Set 1

This article is not for those algo geeks. If you are interested in systems related stuff, just read on...

Shared libraries are useful in sharing code which is common across many applications. For example, it is more economic to pack all the code related to TCP/IP implementation in a shared library. However, data can't be shared as every application needs its own set of data. Applications like, browser, ftp, telnet, etc... make use of the shared 'network' library to elevate specific functionality.

Every operating system has its own representation and tool-set to create shared libraries. More or less the concepts are same. On Windows every object file (*.obj, *.dll, *.ocx, *.sys, *.exe etc...) follow a format called Portable Executable. Even shared libraries (called as Dynamic Linked Libraries or DLL in short) are also represented in PE format. The tool-set that is used to create these libraries need to understand the binary format. Linux variants follow a format called Executable and Linkable Format (ELF). The ELF files are position independent (PIC) format. Shared libraries in Linux are referred as shared objects (generally with extension *.so). These are similar to DLLs in Windows platform. Even shared object files follow the ELF binary format.

Remember, the file extensions (*.dll, *.so, *.a, *.lib, etc...) are just for programmer convenience. They don't have any significance. All these are binary files. You can name them as you wish. Yet ensure you provide absolute paths in building applications.

In general, when we compile an application the steps are simple. Compile, Link and Load. However, it is not simple. These steps are more versatile on modern operating systems.

When you link your application against static library, the code is part of your application. There is no dependency. Even though it causes the application size to increase, it has its own advantages. The primary one is speed as there will be no symbol (a program entity) resolution at runtime. Since every piece of code part of the binary image, such applications are independent of version mismatch issues. However, the cost is on fixing an issue in library code. If there is any bug in library code, entire application need to be recompiled and shipped to the client. In case of dynamic libraries, fixing or upgrading the libraries is easy. You just need to ship the updated shared libraries. The application need not to recompile, it only need to re-run. You can design a mechanism where we don't

need to restart the application.

When we link an application against a shared library, the linker leaves some stubs (unresolved symbols) to be filled at application loading time. These stubs need to be filled by a tool called, *dynamic linker* at run time or at application loading time. Again loading of a library is of two types, static loading and dynamic loading. Don't confuse between **static loading** vs **static linking** and **dynamic loading** vs **dynamic linking**.

For example, you have built an application that depends on *libstdc++.so* which is a shared object (dynamic library). How does the application become aware of required shared libraries? (If you are interested, explore the tools *tdump* from Borland tool set, *objdump* or *nm* or *readelf* tools on Linux).

Static loading:

- In static loading, all of those dependent shared libraries are loaded into memory even before the application starts execution. If loading of any shared library fails, the application won't run.
- A dynamic loader examines application's dependency on shared libraries. If these libraries are already loaded into the memory, the library address space is mapped to application virtual address space (VAS) and the dynamic linker does relocation of unresolved symbols.
- If these libraries are not loaded into memory (perhaps your application might be first to invoke the shared library), the loader searches in standard library paths and loads them into memory, then maps and resolves symbols. Again loading is big process, if you are interested write your own loader :).
- While resolving the symbols, if the dynamic linker not able to find any symbol (may be due to older version of shared library), the application can't be started.

Dynamic Loading:

- As the name indicates, dynamic loading is about loading of library on demand.
- For example, if you want a small functionality from a shared library. Why should it be loaded at the application load time and sit in the memory? You can invoke loading of these shared libraries dynamically when you need their functionality. This is called dynamic loading. In this case, the programmer aware of situation 'when should the library be loaded'. The tool-set and relevant kernel provides API to support dynamic loading, and querying of symbols in the shared library.

More details in later articles.

Note: If you come across terms like loadable modules or equivalent terms, don't mix them with shared libraries. They are different from shared libraries. The kernels provide framework to support loadable modules.

Exercise:

1. Assuming you have understood the concepts, How do you design an application (e.g. Banking) which can upgrade to new shared libraries without re-running the application.

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

118. Dynamic Programming | Set 32 (Word Break Problem)

Given an input string and a dictionary of words, find out if the input string can be segmented into a space-separated sequence of dictionary words. See following examples for more details.

This is a famous Google interview question, also being asked by many other companies now a days.

```
Consider the following dictionary
```

```
{ i, like, sam, sung, samsung, mobile, ice,
  cream, icecream, man, go, mango}
```

```
Input:  ilike
```

```
Output: Yes
```

```
The string can be segmented as "i like".
```

```
Input:  ilikesamsung
```

```
Output: Yes
```

```
The string can be segmented as "i like samsung" or "i like sam sung".
```

Recursive implementation:

The idea is simple, we consider each prefix and search it in dictionary. If the prefix is present in dictionary, we recur for rest of the string (or suffix). If the recursive call for suffix returns true, we return true, otherwise we try next prefix. If we have tried all prefixes and none of them resulted in a solution, we return false.

We strongly recommend to see **substr** function which is used extensively in following implementations.

```

// A recursive program to test whether a given string can be segmented
// space separated words in dictionary
#include <iostream>
using namespace std;

/* A utility function to check whether a word is present in dictionary
An array of strings is used for dictionary. Using array of strings
dictionary is definitely not a good idea. We have used for simplicity
the program*/
int dictionaryContains(string word)
{
    string dictionary[] = {"mobile", "samsung", "sam", "sung", "man", "mango",
                           "icecream", "and", "go", "i", "like", "ice", "cream"};
    int size = sizeof(dictionary)/sizeof(dictionary[0]);
    for (int i = 0; i < size; i++)
        if (dictionary[i].compare(word) == 0)
            return true;
    return false;
}

// returns true if string can be segmented into space separated
// words, otherwise returns false
bool wordBreak(string str)
{
    int size = str.size();

    // Base case
    if (size == 0) return true;

    // Try all prefixes of lengths from 1 to size
    for (int i=1; i<=size; i++)
    {
        // The parameter for dictionaryContains is str.substr(0, i)
        // str.substr(0, i) which is prefix (of input string) of
        // length 'i'. We first check whether current prefix is in
        // dictionary. Then we recursively check for remaining string
        // str.substr(i, size-i) which is suffix of length size-i
        if (dictionaryContains( str.substr(0, i) ) &&
            wordBreak( str.substr(i, size-i) ))
            return true;
    }

    // If we have tried all prefixes and none of them worked
    return false;
}

// Driver program to test above functions
int main()
{
    wordBreak("ilikesamsung")? cout <<"Yes\n": cout << "No\n";
    wordBreak("iiiiiii")? cout <<"Yes\n": cout << "No\n";
    wordBreak("")? cout <<"Yes\n": cout << "No\n";
    wordBreak("ilikelikeimangoiii")? cout <<"Yes\n": cout << "No\n";
    wordBreak("samsungandmango")? cout <<"Yes\n": cout << "No\n";
    wordBreak("samsungandmangok")? cout <<"Yes\n": cout << "No\n";
    return 0;
}

```

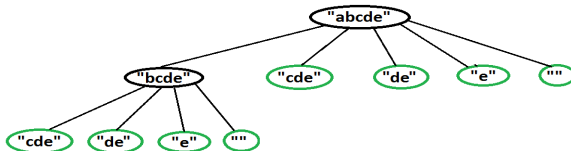
Output:

Yes

Yes
Yes
Yes
Yes
No

Dynamic Programming

Why Dynamic Programming? The above problem exhibits overlapping sub-problems. For example, see the following partial recursion tree for string "abcde" in worst case.



Partial recursion tree for input string "abcde". The subproblems encircled with green color are overlapping subproblems

```
// A Dynamic Programming based program to test whether a given string
// be segmented into space separated words in dictionary
#include <iostream>
#include <string.h>
using namespace std;

/* A utility function to check whether a word is present in dictionary
An array of strings is used for dictionary. Using array of strings
dictionary is definitely not a good idea. We have used for simplicity
the program*/
int dictionaryContains(string word)
{
    string dictionary[] = {"mobile", "samsung", "sam", "sung", "man", "mango",
                           "icecream", "and", "go", "i", "like", "ice", "cream"};
    int size = sizeof(dictionary)/sizeof(dictionary[0]);
    for (int i = 0; i < size; i++)
        if (dictionary[i].compare(word) == 0)
            return true;
    return false;
}

// Returns true if string can be segmented into space separated
// words, otherwise returns false
bool wordBreak(string str)
{
    int size = str.size();
    if (size == 0) return true;

    // Create the DP table to store results of subproblems. The value w
    // will be true if str[0..i-1] can be segmented into dictionary wo
    // otherwise false.
    bool wb[size+1];
    memset(wb, 0, sizeof(wb)); // Initialize all values as false.

    for (int i=1; i<=size; i++)
    {
        // if wb[i] is false, then check if current prefix can make it
        // Current prefix is "str.substr(0, i)"
        if (wb[i] == false && dictionaryContains( str.substr(0, i) ))
```

```

2. wb[i] = false && dictionaryContains( str.substr(0, i) );
   wb[i] = true;

// wb[i] is true, then check for all substrings starting from
// (i+1)th character and store their results.
if (wb[i] == true)
{
    // If we reached the last prefix
    if (i == size)
        return true;

    for (int j = i+1; j <= size; j++)
    {
        // Update wb[j] if it is false and can be updated
        // Note the parameter passed to dictionaryContains() i.
        // substring starting from index 'i' and length 'j-i'
        if (wb[j] == false && dictionaryContains( str.substr(i, j-i) ))
            wb[j] = true;

        // If we reached the last character
        if (j == size && wb[j] == true)
            return true;
    }
}

/* Uncomment these lines to print DP table "wb[]"
for (int i = 1; i <= size; i++)
    cout << " " << wb[i]; */

// If we have tried all prefixes and none of them worked
return false;
}

// Driver program to test above functions
int main()
{
    wordBreak("ilikesamsung")? cout <<"Yes\n": cout << "No\n";
    wordBreak("iiiiiii")? cout <<"Yes\n": cout << "No\n";
    wordBreak("")? cout <<"Yes\n": cout << "No\n";
    wordBreak("ilikelikeimangoiii")? cout <<"Yes\n": cout << "No\n";
    wordBreak("samsungandmango")? cout <<"Yes\n": cout << "No\n";
    wordBreak("samsungandmangok")? cout <<"Yes\n": cout << "No\n";
    return 0;
}

```

Output:

```

Yes
Yes
Yes
Yes
Yes
No

```

Exercise:

The above solutions only finds out whether a given string can be segmented or not.

Extend the above Dynamic Programming solution to print all possible partitions of input

string. See [extended recursive solution](#) for reference.

Examples:

```
Input: ilikeicecreamandmango
```

```
Output:
```

```
i like ice cream and man go
```

```
i like ice cream and mango
```

```
i like icecream and man go
```

```
i like icecream and mango
```

```
Input: ilikesamsungmobile
```

```
Output:
```

```
i like sam sung mobile
```

```
i like samsung mobile
```

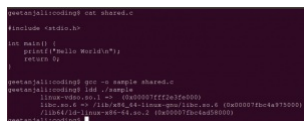
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

119. Working with Shared Libraries | Set 2

We have covered basic information about shared libraries in the [previous post](#). In the current article we will learn how to create shared libraries on Linux.

Prior to that we need to understand how a program is loaded into memory, various (basic) steps involved in the process.

Let us see a typical “Hello World” program in C. Simple Hello World program screen image is given below.



```
root@kali:~/coding# cat shared.c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}

root@kali:~/coding# gcc -o sample shared.c
root@kali:~/coding# ldd sample
linux-vx86.so.1 => (shared)
libc.so.6 => /lib64/libc.so.6 (shared)
/lib64/ld-linux-x86-64.so.2 => /lib64/ld-linux-x86-64.so.2 (shared)
```

We were compiling our code using the command “**gcc -o sample shared.c**” When we compile our code, the compiler won’t resolve implementation of the function **printf()**. It only verifies the syntactical checking. The tool chain leaves a stub in our application which will be filled by dynamic linker. Since printf is standard function the compiler implicitly invoking its shared library. More details down.

We are using **ldd** to list dependencies of our program binary image. In the screen image, we can see our sample program depends on three binary files namely, *linux-vdso.so.1*, *libc.so.6* and */lib64/ld-linux-x86-64.so.2*.

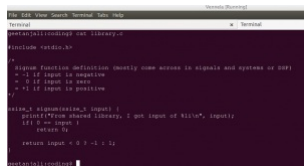
The file `VDSO` is fast implementation of system call interface and some other stuff, it is not our focus (on some older systems you may see different file name in lieu of `*.vdsd.*`). Ignore this file. We have interest in the other two files.

The file `libc.so.6` is C implementation of various standard functions. It is the file where we see `printf` definition needed for our *Hello World*. It is the shared library needed to be loaded into memory to run our Hello World program.

The third file `/lib64/ld-linux-x86-64.so.2` is infact an executable that runs when an application is invoked. When we invoke the program on bash terminal, typically the bash forks itself and replaces its address space with image of program to run (so called fork-exec pair). The kernel verifies whether the `libc.so.6` resides in the memory. If not, it will load the file into memory and does the relocation of `libc.so.6` symbols. It then invokes the dynamic linker (`/lib64/ld-linux-x86-64.so.2`) to resolve unresolved symbols of application code (`printf` in the present case). Then the control transfers to our program *main*. (I have intensionally omitted many details in the process, our focus is to understand basic details).

Creating our own shared library:

Let us work with simple shared library on Linux. Create a file `library.c` with the following content.



```
greet@kali:~/coding$ cat library.c
#include <stdio.h>

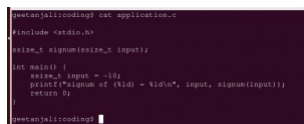
int signum(int input)
{
    //signum function definition: identify some values in signum and systems or not
    // -1 if input is negative
    // 0 if input is zero
    // 1 if input is positive
    if (input < 0)
        return -1;
    else if (input == 0)
        return 0;
    else
        return 1;
}
```

The file `library.c` defines a function *signum* which will be used by our application code. Compile the file `library.c` file using the following command.

gcc -shared -fPIC -o liblibrary.so library.c

The flag `-shared` instructs the compiler that we are building a shared library. The flag `-fPIC` is to generate position independent code (ignore for now). The command generates a shared library `liblibrary.so` in the current working directory. We have our shared object file (shared library name in Linux) ready to use.

Create another file `application.c` with the following content.



```
greet@kali:~/coding$ cat application.c
#include <stdio.h>
#include "library.c"

int main()
{
    int input = -10;
    printf("Signum of %d is %d\n", input, signum(input));
    return 0;
}
```

In the file `application.c` we are invoking the function `signum` which was defined in a

shared library. Compile the application.c file using the following command.

gcc application.c -L /home/geetanjali/coding/ -llibrary -o sample

The flag *-llibrary* instructs the compiler to look for symbol definitions that are not available in the current code (signum function in our case). The option *-L* is hint to the compiler to look in the directory followed by the option for any shared libraries (during link time only). The command generates an executable named as “**sample**”.

If you invoke the executable, the dynamic linker will not be able to find the required shared library. By default it won't look into current working directory. You have to explicitly instruct the tool chain to provide proper paths. The dynamic linker searches standard paths available in the LD_LIBRARY_PATH and also searches in system cache (for details explore the command *ldconfig*). We have to add our working directory to the LD_LIBRARY_PATH environment variable. The following command does the same.

export LD_LIBRARY_PATH=/home/geetanjali/coding/:\$LD_LIBRARY_PATH

You can now invoke our executable as shown.

./sample

Sample output on my system is shown below.



```
sample: error while loading shared libraries: liblibrary.so: cannot open shared object file: No such file or directory
```

Note: The path */home/geetanjali/coding/* is working directory path on my machine. You need to use your working directory path where ever it is being used in the above commands.

Stay tuned, we haven't even explored 1/3rd of shared library concepts. More advanced concepts in the later articles.

Exercise:

It is workbook like article. You won't gain much unless you practice and do some research.

1. Create similar example and write your won function in the shared library. Invoke the function in another application.
2. Is (Are) there any other tool(s) which can list dependent libraries?
3. What is position independent code (PIC)?
4. What is system cache in the current context? How does the directory */etc/ld.so.conf.d/** related in the current context?

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

120. Print all possible strings of length k that can be formed from a set of n characters

Given a set of characters and a positive integer k, print all possible strings of length k that can be formed from the given set.

Examples:

Input:

```
set[] = {'a', 'b'}, k = 3
```

Output:

aaa

aab

aba

abb

baa

bab

bba

bbb

Input:

```
set[] = {'a', 'b', 'c', 'd'}, k = 1
```

Output:

a

b

c

d

For a given set of size n, there will be n^k possible strings of length k. The idea is to start from an empty output string (we call it *prefix* in following code). One by one add all characters to *prefix*. For every character added, print all possible strings with current prefix by recursively calling for k equals to k-1.

Following is Java implementation for same.

```
// Java program to print all possible strings of length k
```

```
class PrintAllKLengthStrings {
```

```
    // Driver method to test below methods
```

```
    public static void main(String[] args) {
```

```
        System.out.println("First Test");
```

```

        char set1[] = {'a', 'b'};
        int k = 3;
        printAllKLength(set1, k);

        System.out.println("\nSecond Test");
        char set2[] = {'a', 'b', 'c', 'd'};
        k = 1;
        printAllKLength(set2, k);
    }

    // The method that prints all possible strings of length k. It is
    // mainly a wrapper over recursive function printAllKLengthRec()
    static void printAllKLength(char set[], int k) {
        int n = set.length;
        printAllKLengthRec(set, "", n, k);
    }

    // The main recursive method to print all possible strings of length k
    static void printAllKLengthRec(char set[], String prefix, int n, int k) {

        // Base case: k is 0, print prefix
        if (k == 0) {
            System.out.println(prefix);
            return;
        }

        // One by one add all characters from set and recursively
        // call for k equals to k-1
        for (int i = 0; i < n; ++i) {

            // Next character of input added
            String newPrefix = prefix + set[i];

            // k is decreased, because we have added a new character
            printAllKLengthRec(set, newPrefix, n, k - 1);
        }
    }
}

```

Output:

```

First Test
aaa
aab
aba

```

abb
baa
bab
bba
bbb

Second Test

a
b
c
d

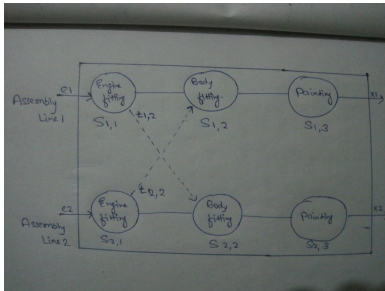
The above solution is mainly generalization of [this post](#).

This article is contributed by **Abhinav Ramana**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

121. Dynamic Programming | Set 34 (Assembly Line Scheduling)

A car factory has two assembly lines, each with n stations. A station is denoted by $S_{i,j}$ where i is either 1 or 2 and indicates the assembly line the station is on, and j indicates the number of the station. The time taken per station is denoted by $a_{i,j}$. Each station is dedicated to some sort of work like engine fitting, body fitting, painting and so on. So, a car chassis must pass through each of the n stations in order before exiting the factory. The parallel stations of the two assembly lines perform the same task. After it passes through station $S_{i,j}$, it will continue to station $S_{i,j+1}$ unless it decides to transfer to the other line. Continuing on the same line incurs no extra cost, but transferring from line i at station $j - 1$ to station j on the other line takes time $t_{i,j}$. Each assembly line takes an entry time e_i and exit time x_i which may be different for the two lines. Give an algorithm for computing the minimum time it will take to build a car chassis.

The below figure presents the problem in a clear picture:



The following information can be extracted from the problem statement to make it simpler:

- Two assembly lines, 1 and 2, each with stations from 1 to n .
- A car chassis must pass through all stations from 1 to n in order (in any of the two assembly lines). i.e. it cannot jump from station i to station j if they are not at one move distance.
- The car chassis can move one station forward in the same line, or one station diagonally in the other line. It incurs an extra cost $t_{i,j}$ to move to station j from line i . No cost is incurred for movement in same line.
- The time taken in station j on line i is $a_{i,j}$.
- $S_{i,j}$ represents a station j on line i .

Breaking the problem into smaller sub-problems:

We can easily find the i th factorial if $(i-1)$ th factorial is known. Can we apply the similar funda here?

If the minimum time taken by the chassis to leave station $S_{i,j-1}$ is known, the minimum time taken to leave station $S_{i,j}$ can be calculated quickly by combining $a_{i,j}$ and $t_{i,j}$.

T1(j) indicates the minimum time taken by the car chassis to leave station j on assembly line 1.

T2(j) indicates the minimum time taken by the car chassis to leave station j on assembly line 2.

Base cases:

The entry time e_i comes into picture only when the car chassis enters the car factory.

Time taken to leave first station in line 1 is given by:

$$T1(1) = \text{Entry time in Line 1} + \text{Time spent in station } S_{1,1}$$

$$T1(1) = e_1 + a_{1,1}$$

Similarly, time taken to leave first station in line 2 is given by:

$$T2(1) = e_2 + a_{2,1}$$

Recursive Relations:

If we look at the problem statement, it quickly boils down to the below observations:

The car chassis at station $S_{1,j}$ can come either from station $S_{1,j-1}$ or station $S_{2,j-1}$.

Case #1: Its previous station is $S_{1,j-1}$

The minimum time to leave station $S_{1,j}$ is given by:

$T1(j)$ = Minimum time taken to leave station $S_{1,j-1}$ + Time spent in station $S_{1,j}$

$$T1(j) = T1(j-1) + a_{1,j}$$

Case #2: Its previous station is $S_{2,j-1}$

The minimum time to leave station $S_{1,j}$ is given by:

$T1(j)$ = Minimum time taken to leave station $S_{2,j-1}$ + Extra cost incurred to change the assembly line + Time spent in station $S_{1,j}$

$$T1(j) = T2(j-1) + t_{2,j} + a_{1,j}$$

The minimum time $T1(j)$ is given by the minimum of the two obtained in cases #1 and #2.

$$T1(j) = \min((T1(j-1) + a_{1,j}), (T2(j-1) + t_{2,j} + a_{1,j}))$$

Similarly the minimum time to reach station $S_{2,j}$ is given by:

$$T2(j) = \min((T2(j-1) + a_{2,j}), (T1(j-1) + t_{1,j} + a_{2,j}))$$

The total minimum time taken by the car chassis to come out of the factory is given by:

T_{min} = min(Time taken to leave station $S_{i,n}$ + Time taken to exit the car factory)

$$T_{min} = \min(T1(n) + x_1, T2(n) + x_2)$$

Why dynamic programming?

The above recursion exhibits overlapping sub-problems. There are two ways to reach station $S_{1,j}$:

1. From station $S_{1,j-1}$
2. From station $S_{2,j-1}$

So, to find the minimum time to leave station $S_{1,j}$ the minimum time to leave the previous two stations must be calculated(as explained in above recursion).

Similarly, there are two ways to reach station $S_{2,j}$:

1. From station $S_{2,j-1}$
2. From station $S_{1,j-1}$

Please note that the minimum times to leave stations $S_{1,j-1}$ and $S_{2,j-1}$ have already been calculated.

So, we need two tables to store the partial results calculated for each station in an assembly line. The table will be filled in bottom-up fashion.

Note:

In this post, the word “leave” has been used in place of “reach” to avoid the confusion. Since the car chassis must spend a fixed time in each station, the word leave suits better.

Implementation:

```
// A C program to find minimum possible time by the car chassis to complete the race
#include <stdio.h>
#define NUM_LINE 2
#define NUM_STATION 4

// Utility function to find minimum of two numbers
int min(int a, int b) { return a < b ? a : b; }

int carAssembly(int a[][NUM_STATION], int t[][NUM_STATION], int *e, int *x)
{
    int T1[NUM_STATION], T2[NUM_STATION], i;

    T1[0] = e[0] + a[0][0]; // time taken to leave first station in line 1
    T2[0] = e[1] + a[1][0]; // time taken to leave first station in line 2

    // Fill tables T1[] and T2[] using the above given recursive relation
    for (i = 1; i < NUM_STATION; ++i)
    {
        T1[i] = min(T1[i-1] + a[0][i], T2[i-1] + t[1][i] + a[0][i]);
        T2[i] = min(T2[i-1] + a[1][i], T1[i-1] + t[0][i] + a[1][i]);
    }

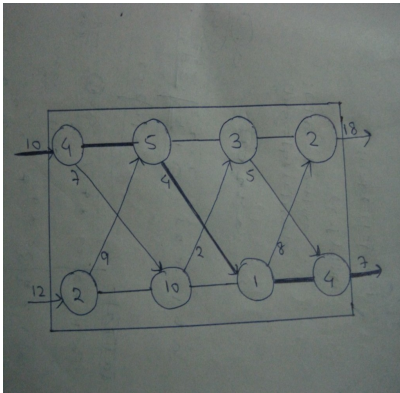
    // Consider exit times and return minimum
    return min(T1[NUM_STATION-1] + x[0], T2[NUM_STATION-1] + x[1]);
}

int main()
{
    int a[][NUM_STATION] = {{4, 5, 3, 2},
                             {2, 10, 1, 4}};
    int t[][NUM_STATION] = {{0, 7, 4, 5},
                             {0, 9, 2, 8}};
    int e[] = {10, 12}, x[] = {18, 7};

    printf("%d", carAssembly(a, t, e, x));

    return 0;
}
```

Output:



The bold line shows the path covered by the car chassis for given input values.

Exercise:

Extend the above algorithm to print the path covered by the car chassis in the factory.

References:

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

This article is compiled by **Aashish Barnwal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

122. Dynamic Programming | Set 35 (Longest Arithmetic Progression)

Given a set of numbers, find the **Length of the Longest Arithmetic Progression (LLAP)** in it.

Examples:

```
set[] = {1, 7, 10, 15, 27, 29}
output = 3
The longest arithmetic progression is {1, 15, 29}

set[] = {5, 10, 15, 20, 25, 30}
output = 6
The whole set is in AP
```

For simplicity, we have assumed that the given set is sorted. We can always add a pre-

processing step to first sort the set and then apply the below algorithms.

A **simple solution** is to one by one consider every pair as first two elements of AP and check for the remaining elements in sorted set. To consider all pairs as first two elements, we need to run a $O(n^2)$ nested loop. Inside the nested loops, we need a third loop which linearly looks for the more elements in **Arithmetic Progression (AP)**. This process takes $O(n^3)$ time.

We can solve this problem in $O(n^2)$ time **using Dynamic Programming**. To get idea of the DP solution, let us first discuss solution of following simpler problem.

Given a sorted set, find if there exist three elements in Arithmetic Progression or not

Please note that, the answer is true if there are 3 or more elements in AP, otherwise false.

To find the three elements, we first fix an element as middle element and search for other two (one smaller and one greater). We start from the second element and fix every element as middle element. For an element $set[j]$ to be middle of AP, there must exist elements ' $set[i]$ ' and ' $set[k]$ ' such that $set[i] + set[k] = 2 * set[j]$ where $0 \leq i < j$ and $j < k \leq n-1$.

How to efficiently find i and k for a given j? We can find i and k in linear time using following simple algorithm.

1) Initialize i as j-1 and k as j+1

2) Do following while $i \geq 0$ and $j \leq n-1$

.....**a)** If $set[i] + set[k]$ is equal to $2 * set[j]$, then we are done.

.....**b)** If $set[i] + set[k] > 2 * set[j]$, then decrement i (do $i--$).

.....**c)** Else if $set[i] + set[k] < 2 * set[j]$, then increment k (do $k++$).

Following is C++ implementation of the above algorithm for the simpler problem.

```

// The function returns true if there exist three elements in AP
// Assumption: set[0..n-1] is sorted.
// The code strictly implements the algorithm provided in the reference
bool arithmeticThree(int set[], int n)
{
    // One by fix every element as middle element
    for (int j=1; j<n-1; j++)
    {
        // Initialize i and k for the current j
        int i = j-1, k = j+1;

        // Find if there exist i and k that form AP
        // with j as middle element
        while (i >= 0 && k <= n-1)
        {
            if (set[i] + set[k] == 2*set[j])
                return true;
            (set[i] + set[k] < 2*set[j])? k++ : i--;
        }
    }

    return false;
}

```

See [this](#) for a complete running program.

How to extend the above solution for the original problem?

The above function returns a boolean value. The required output of original problem is Length of the Longest Arithmetic Progression (**LLAP**) which is an integer value. If the given set has two or more elements, then the value of LLAP is at least 2 (Why?).

The idea is to create a 2D table $L[n][n]$. An entry $L[i][j]$ in this table stores LLAP with $set[i]$ and $set[j]$ as first two elements of AP and $j > i$. The last column of the table is always 2 (Why – see the meaning of $L[i][j]$). Rest of the table is filled from bottom right to top left. To fill rest of the table, j (second element in AP) is first fixed. i and k are searched for a fixed j . If i and k are found such that i, j, k form an AP, then the value of $L[i][j]$ is set as $L[j][k] + 1$. Note that the value of $L[j][k]$ must have been filled before as the loop traverses from right to left columns.

Following is C++ implementation of the Dynamic Programming algorithm.

```

// C++ program to find Length of the Longest AP (llap) in a given sorted set
// The code strictly implements the algorithm provided in the reference
#include <iostream>
using namespace std;

// Returns length of the longest AP subset in a given set
int lengthOfLongestAP(int set[], int n)
{
    if (n <= 2) return n;

    // Create a table and initialize all values as 2. The value of
    // L[i][j] stores LLAP with set[i] and set[j] as first two
    // elements of AP. Only valid entries are the entries where j>i
    int L[n][n];
    int llap = 2; // Initialize the result

    // Fill entries in last column as 2. There will always be

```

```

// two elements in AP with last number of set as second
// element in AP
for (int i = 0; i < n; i++)
    L[i][n-1] = 2;

// Consider every element as second element of AP
for (int j=n-2; j>=1; j--)
{
    // Search for i and k for j
    int i = j-1, k = j+1;
    while (i >= 0 && k <= n-1)
    {
        if (set[i] + set[k] < 2*set[j])
            k++;

        // Before changing i, set L[i][j] as 2
        else if (set[i] + set[k] > 2*set[j])
        {
            L[i][j] = 2, i--;
        }

        else
        {
            // Found i and k for j, LLAP with i and j as first two
            // elements is equal to LLAP with j and k as first two
            // elements plus 1. L[j][k] must have been filled
            // before as we run the loop from right side
            L[i][j] = L[j][k] + 1;

            // Update overall LLAP, if needed
            llap = max(llap, L[i][j]);

            // Change i and k to fill more L[i][j] values for
            // current j
            i--; k++;
        }
    }

    // If the loop was stopped due to k becoming more than
    // n-1, set the remaining entities in column j as 2
    while (i >= 0)
    {
        L[i][j] = 2;
        i--;
    }
}
return llap;
}

```

```

/* Drier program to test above function*/
int main()
{
    int set1[] = {1, 7, 10, 13, 14, 19};
    int n1 = sizeof(set1)/sizeof(set1[0]);
    cout << lenghtOfLongestAP(set1, n1) << endl;

    int set2[] = {1, 7, 10, 15, 27, 29};
    int n2 = sizeof(set2)/sizeof(set2[0]);
    cout << lenghtOfLongestAP(set2, n2) << endl;

    int set3[] = {2, 4, 6, 8, 10};
    int n3 = sizeof(set3)/sizeof(set3[0]);
    cout << lenghtOfLongestAP(set3, n3) << endl;
}

```

```

    return 0;
}

```

Output:

```

4
3
5

```

Time Complexity: $O(n^2)$

Auxiliary Space: $O(n^2)$

References:

<http://www.cs.uiuc.edu/~jeffe/pubs/pdf/arith.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

123. 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.

Before we discuss solution, let us define notion of **orientation**. Orientation of an ordered triplet of points in the plane can be

- counterclockwise
- clockwise
- collinear

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



Note the word 'ordered' here. Orientation of (a, b, c) may be different from orientation of (c, b, a) .

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:

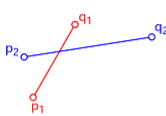
- $(p1, q1, p2)$ and $(p1, q1, q2)$ have different orientations and

– (p_2, q_2, p_1) and (p_2, q_2, q_1) have different orientations

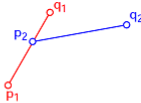
2. Special Case

- (p_1, q_1, p_2) , (p_1, q_1, q_2) , (p_2, q_2, p_1) , and (p_2, q_2, q_1) are all collinear and
- the x-projections of (p_1, q_1) and (p_2, q_2) intersect
- the y-projections of (p_1, q_1) and (p_2, q_2) intersect

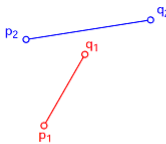
Examples of General Case:



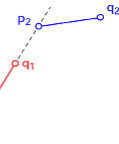
Example 1: Orientations of (p_1, q_1, p_2) and (p_1, q_1, q_2) are different. Orientations of (p_2, q_2, p_1) and (p_2, q_2, q_1) are also different



Example 2: Orientations of (p_1, q_1, p_2) and (p_1, q_1, q_2) are different. Orientations of (p_2, q_2, p_1) and (p_2, q_2, q_1) are also different



Example 3: Orientations of (p_1, q_1, p_2) and (p_1, q_1, q_2) are different. Orientations of (p_2, q_2, p_1) and (p_2, q_2, q_1) are same

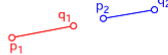


Example 4: Orientations of (p_1, q_1, p_2) and (p_1, q_1, q_2) are different. Orientations of (p_2, q_2, p_1) and (p_2, q_2, q_1) are same

Examples of Special Case:



Example 1: All points are collinear. The x-projections of (p_1, q_1) and (p_2, q_2) intersect. The y-projections of (p_1, q_1) and (p_2, q_2) intersect



Example 2: All points are collinear. The x-projections of (p_1, q_1) and (p_2, q_2) do not intersect. The y-projections of (p_1, q_1) and (p_2, q_2) intersect

Following is C++ implementation based on above idea.

```
// A C++ program to check if two given line segments intersect
#include <iostream>
using namespace std;
```

```
struct Point
```

```
{
    int x;
    int y;
};
```

```
// Given three collinear points p, q, r, the function checks if
// point q lies on line segment 'pr'
```

```
bool onSegment(Point p, Point q, Point r)
{
    if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
        q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
        return true;

    return false;
}
```

```
// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are collinear
// 1 --> Clockwise
```

```

// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(Point p, Point q, Point r)
{
    // See 10th slides from following link for derivation of the formula
    // http://www.dcs.gla.ac.uk/~pat/52233/slides/Geometry1x1.pdf
    int val = (q.y - p.y) * (r.x - q.x) -
              (q.x - p.x) * (r.y - q.y);

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

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

```

```

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

    // p1, q1 and p2 are colinear and q2 lies on segment p1q1
    if (o2 == 0 && onSegment(p1, q2, q1)) return true;

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

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

    return false; // Doesn't fall in any of the above cases
}

```

```

// Driver program to test above functions
int main()
{
    struct Point p1 = {1, 1}, q1 = {10, 1};
    struct Point p2 = {1, 2}, q2 = {10, 2};

    doIntersect(p1, q1, p2, q2)? cout << "Yes\n": cout << "No\n";

    p1 = {10, 0}, q1 = {0, 10};
    p2 = {0, 0}, q2 = {10, 10};
    doIntersect(p1, q1, p2, q2)? cout << "Yes\n": cout << "No\n";

    p1 = {-5, -5}, q1 = {0, 0};
    p2 = {1, 1}, q2 = {10, 10};
    doIntersect(p1, q1, p2, q2)? cout << "Yes\n": cout << "No\n";

    return 0;
}

```


Output:

No
Yes
No

Sources:

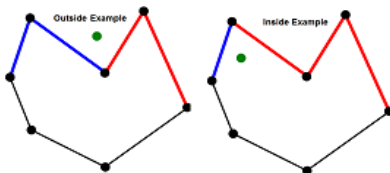
<http://www.dcs.gla.ac.uk/~pat/52233/slides/Geometry1x1.pdf>

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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

124. 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.

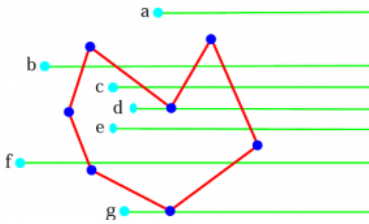


We strongly recommend to see the following post first.

[How to check if two given line segments intersect?](#)

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
- 1) Count the number of times the line intersects with polygon edges.
- 2) 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 return 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 colinear, 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.

```
// A C++ program to check if a given point lies inside a given polygon
// Refer http://www.geeksforgeeks.org/check-if-two-given-line-segments
// for explanation of functions onSegment(), orientation() and doIntersect()
#include <iostream>
using namespace std;

// Define Infinite (Using INT_MAX caused overflow problems)
#define INF 10000

struct Point
{
    int x;
    int y;
};

// Given three colinear points p, q, r, the function checks if
// point q lies on line segment 'pr'
bool onSegment(Point p, Point q, Point r)
{
    if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
        q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
        return true;
    return false;
}

// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are colinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(Point p, Point q, Point r)
{
    int val = (q.y - p.y) * (r.x - q.x) -
              (q.x - p.x) * (r.y - q.y);

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

// The function that returns true if line segment 'p1q1'
// intersects with line segment 'p2q2'
bool doIntersect(Point p1, Point q1, Point p2, Point q2)
{
    // Special Cases
    // 1. One of the line segments is a collinear point
    if (p1.x == p2.x && p1.y == p2.y) return true;
    if (p1.x == q2.x && p1.y == q2.y) return true;
    if (p2.x == q1.x && p2.y == q1.y) return true;
    if (q1.x == q2.x && q1.y == q2.y) return true;

    // 2. General case
    int o1 = orientation(p1, q1, p2);
    int o2 = orientation(p1, q1, q2);
    int o3 = orientation(p2, q2, p1);
    int o4 = orientation(p2, q2, q1);

    if (o1 != o2 && o3 != o4) return true; // General case
    if (o1 == 0 && onSegment(p2, p1, q1)) return true;
    if (o2 == 0 && onSegment(q2, p1, q1)) return true;
    if (o3 == 0 && onSegment(p1, p2, q2)) return true;
    if (o4 == 0 && onSegment(q1, p2, q2)) return true;

    return false;
}
```

```

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

    // p1, q1 and p2 are colinear and q2 lies on segment p1q1
    if (o2 == 0 && onSegment(p1, q2, q1)) return true;

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

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

    return false; // Doesn't fall in any of the above cases
}

// 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 colinear 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)
}

```

```
// Driver program to test above functions
int main()
{
    Point polygon1[] = {{0, 0}, {10, 0}, {10, 10}, {0, 10}};
    int n = sizeof(polygon1)/sizeof(polygon1[0]);
    Point p = {20, 20};
    isInside(polygon1, n, p)? cout << "Yes \n": cout << "No \n";

    p = {5, 5};
    isInside(polygon1, n, p)? cout << "Yes \n": cout << "No \n";

    Point polygon2[] = {{0, 0}, {5, 5}, {5, 0}};
    p = {3, 3};
    n = sizeof(polygon2)/sizeof(polygon2[0]);
    isInside(polygon2, n, p)? cout << "Yes \n": cout << "No \n";

    p = {5, 1};
    isInside(polygon2, n, p)? cout << "Yes \n": cout << "No \n";

    p = {8, 1};
    isInside(polygon2, n, p)? cout << "Yes \n": cout << "No \n";

    Point polygon3[] = {{0, 0}, {10, 0}, {10, 10}, {0, 10}};
    p = {-1, 10};
    n = sizeof(polygon3)/sizeof(polygon3[0]);
    isInside(polygon3, n, p)? cout << "Yes \n": cout << "No \n";

    return 0;
}
```

Output:

```
No
Yes
Yes
Yes
No
No
```

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

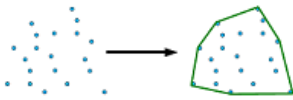
Source:

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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

125. Convex Hull | Set 1 (Jarvis's Algorithm or Wrapping)

Given a set of points in the plane. the convex hull of the set is the smallest convex polygon that contains all the points of it.



We strongly recommend to see the following post first.

[How to check if two given line segments intersect?](#)

The idea of Jarvis's Algorithm is simple, we start from the leftmost point (or point with minimum x coordinate value) and we keep wrapping points in counterclockwise direction. The big question is, given a point p as current point, how to find the next point in output? The idea is to use [orientation\(\)](#) here. Next point is selected as the point that beats all other points at counterclockwise orientation, i.e., next point is q if for any other point r, we have "orientation(p, r, q) = counterclockwise". Following is the detailed algorithm.

1) Initialize p as leftmost point.

2) Do following while we don't come back to the first (or leftmost) point.

.....**a)** The next point q is the point such that the triplet (p, q, r) is counterclockwise for any other point r.

.....**b)** next[p] = q (Store q as next of p in the output convex hull).

.....**c)** p = q (Set p as q for next iteration).

```
// A C++ program to find convex hull of a set of points
// Refer http://www.geeksforgeeks.org/check-if-two-given-line-segments
// for explanation of orientation()
#include <iostream>
using namespace std;

// Define Infinite (Using INT_MAX caused overflow problems)
#define INF 10000

struct Point
{
    int x;
    int y;
};

// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are colinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(Point p, Point q, Point r)
{
    int val = (q.y - p.y) * (r.x - q.x) -
              (q.x - p.x) * (r.y - q.y);

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

// Prints convex hull of a set of n points.
```

```

void convexHull(Point points[], int n)
{
    // There must be at least 3 points
    if (n < 3) return;

    // Initialize Result
    int next[n];
    for (int i = 0; i < n; i++)
        next[i] = -1;

    // Find the leftmost point
    int l = 0;
    for (int i = 1; i < n; i++)
        if (points[i].x < points[l].x)
            l = i;

    // Start from leftmost point, keep moving counterclockwise
    // until reach the start point again
    int p = l, q;
    do
    {
        // Search for a point 'q' such that orientation(p, i, q) is
        // counterclockwise for all points 'i'
        q = (p+1)%n;
        for (int i = 0; i < n; i++)
            if (orientation(points[p], points[i], points[q]) == 2)
                q = i;

        next[p] = q; // Add q to result as a next point of p
        p = q; // Set p as q for next iteration
    } while (p != l);

    // Print Result
    for (int i = 0; i < n; i++)
    {
        if (next[i] != -1)
            cout << "(" << points[i].x << ", " << points[i].y << ")\n";
    }
}

// Driver program to test above functions
int main()
{
    Point points[] = {{0, 3}, {2, 2}, {1, 1}, {2, 1},
                     {3, 0}, {0, 0}, {3, 3}};
    int n = sizeof(points)/sizeof(points[0]);
    convexHull(points, n);
    return 0;
}

```

Output: The output is points of the convex hull.

```

(0, 3)
(3, 0)
(0, 0)
(3, 3)

```

Time Complexity: For every point on the hull we examine all the other points to determine the next point. Time complexity is $\Theta(m * n)$ where n is number of input points

and m is number of output or hull points ($m \leq n$). In worst case, time complexity is $O(n^2)$. The worst case occurs when all the points are on the hull ($m = n$)

We will soon be discussing other algorithms for finding convex hulls.

Sources:

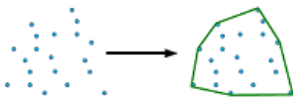
<http://www.cs.uiuc.edu/~jeffe/teaching/373/notes/x05-convexhull.pdf>

<http://www.dcs.gla.ac.uk/~pat/52233/slides/Hull1x1.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

126. Convex Hull | Set 2 (Graham Scan)

Given a set of points in the plane, the convex hull of the set is the smallest convex polygon that contains all the points of it.



We strongly recommend to see the following post first.

[How to check if two given line segments intersect?](#)

We have discussed [Jarvis's Algorithm](#) for Convex Hull. Worst case time complexity of Jarvis's Algorithm is $O(n^2)$. Using Graham's scan algorithm, we can find Convex Hull in $O(n \log n)$ time. Following is Graham's algorithm

Let $\text{points}[0..n-1]$ be the input array.

- 1) Find the bottom-most point by comparing y coordinate of all points. If there are two points with same y value, then the point with smaller x coordinate value is considered. Put the bottom-most point at first position.
- 2) Consider the remaining $n-1$ points and sort them by polar angle in counterclockwise order around $\text{points}[0]$. If polar angle of two points is same, then put the nearest point first.
- 3) Create an empty stack 'S' and push $\text{points}[0]$, $\text{points}[1]$ and $\text{points}[2]$ to S.
- 4) Process remaining $n-3$ points one by one. Do following for every point ' $\text{points}[i]$ '
 - 4.1) Keep removing points from stack while [orientation](#) of following 3 points is not counterclockwise (or they don't make a left turn).
 - a) Point next to top in stack

b) Point at the top of stack

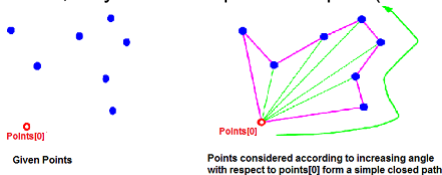
c) points[i]

4.2) Push points[i] to S

5) Print contents of S

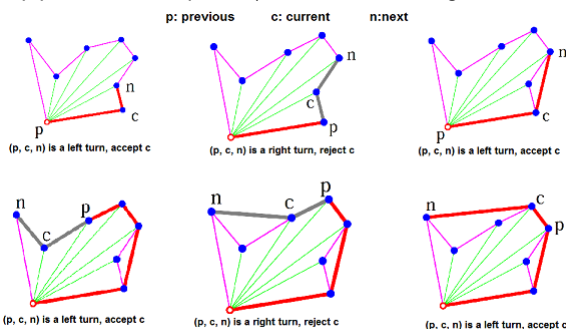
The above algorithm can be divided in two phases.

Phase 1 (Sort points): We first find the bottom-most point. The idea is to pre-process points by sorting them with respect to the bottom-most point. Once the points are sorted, they form a simple closed path (See following diagram).



What should be the sorting criteria? computation of actual angles would be inefficient since trigonometric functions are not simple to evaluate. The idea is to use the orientation to compare angles without actually computing them (See the compare() function below)

Phase 2 (Accept or Reject Points): Once we have the closed path, the next step is to traverse the path and remove concave points on this path. How to decide which point to remove and which to keep? Again, **orientation** helps here. The first two points in sorted array are always part of Convex Hull. For remaining points, we keep track of recent three points, and find the angle formed by them. Let the three points be prev(p), curr(c) and next(n). If orientation of these points (considering them in same order) is not counterclockwise, we discard c, otherwise we keep it. Following diagram shows step by step process of this phase (Source of these diagrams is Ref 2).



In the above algorithm and below code, a stack of points is used to store convex hull points. With reference to the code, p is next-to-top in stack, c is top of stack and n is points[i].

Following is C++ implementation of the above algorithm.

```
// A C++ program to find convex hull of a set of points
// Refer http://www.geeksforgeeks.org/check-if-two-given-line-segments
// for explanation of orientation()
#include <iostream>
```



```

#include <iostream>
#include <stack>
#include <stdlib.h>
using namespace std;

struct Point
{
    int x;
    int y;
};

// A global point needed for sorting points with reference to the first point
// Used in compare function of qsort()
Point p0;

// A utility function to find next to top in a stack
Point nextToTop(stack<Point> &S)
{
    Point p = S.top();
    S.pop();
    Point res = S.top();
    S.push(p);
    return res;
}

// A utility function to swap two points
int swap(Point &p1, Point &p2)
{
    Point temp = p1;
    p1 = p2;
    p2 = temp;
}

// A utility function to return square of distance between p1 and p2
int dist(Point p1, Point p2)
{
    return (p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y);
}

// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are colinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(Point p, Point q, Point r)
{
    int val = (q.y - p.y) * (r.x - p.x) -
              (q.x - p.x) * (r.y - p.y);

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

// A function used by library function qsort() to sort an array of
// points with respect to the first point
int compare(const void *vp1, const void *vp2)
{
    Point *p1 = (Point *)vp1;
    Point *p2 = (Point *)vp2;

    // Find orientation
    int o = orientation(p0, *p1, *p2);
    if (o == 0)

```

```

        return (dist(p0, *p2) >= dist(p0, *p1))? -1 : 1;
    return (o == 2)? -1: 1;
}

// Prints convex hull of a set of n points.
void convexHull(Point points[], int n)
{
    // Find the bottommost point
    int ymin = points[0].y, min = 0;
    for (int i = 1; i < n; i++)
    {
        int y = points[i].y;

        // Pick the bottom-most or chose the left most point in case of tie
        if ((y < ymin) || (ymin == y && points[i].x < points[min].x))
            ymin = points[i].y, min = i;
    }

    // Place the bottom-most point at first position
    swap(points[0], points[min]);

    // Sort n-1 points with respect to the first point. A point p1 comes
    // before p2 in sorted output if p2 has larger polar angle (in
    // counterclockwise direction) than p1
    p0 = points[0];
    qsort(&points[1], n-1, sizeof(Point), compare);

    // Create an empty stack and push first three points to it.
    stack<Point> S;
    S.push(points[0]);
    S.push(points[1]);
    S.push(points[2]);

    // Process remaining n-3 points
    for (int i = 3; i < n; i++)
    {
        // Keep removing top while the angle formed by points next-to-top,
        // top, and points[i] makes a non-left turn
        while (orientation(nextToTop(S), S.top(), points[i]) != 2)
            S.pop();
        S.push(points[i]);
    }

    // Now stack has the output points, print contents of stack
    while (!S.empty())
    {
        Point p = S.top();
        cout << "(" << p.x << ", " << p.y << ")" << endl;
        S.pop();
    }
}

// Driver program to test above functions
int main()
{
    Point points[] = {{0, 3}, {1, 1}, {2, 2}, {4, 4},
                     {0, 0}, {1, 2}, {3, 1}, {3, 3}};
    int n = sizeof(points)/sizeof(points[0]);
    convexHull(points, n);
    return 0;
}

```

Output:

```
(0, 3)
(4, 4)
(3, 1)
(0, 0)
```

Time Complexity: Let n be the number of input points. The algorithm takes $O(n \log n)$ time if we use a $O(n \log n)$ sorting algorithm.

The first step (finding the bottom-most point) takes $O(n)$ time. The second step (sorting points) takes $O(n \log n)$ time. In third step, every element is pushed and popped at most one time. So the third step to process points one by one takes $O(n)$ time, assuming that the stack operations take $O(1)$ time. Overall complexity is $O(n) + O(n \log n) + O(n)$ which is $O(n \log n)$

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
<http://www.dcs.gla.ac.uk/~pat/52233/slides/Hull1x1.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

127. Dynamic Programming | Set 36 (Maximum Product Cutting)

Given a rope of length n meters, cut the rope in different parts of integer lengths in a way that maximizes product of lengths of all parts. You must make at least one cut. Assume that the length of rope is more than 2 meters.

Examples:

```
Input: n = 2
Output: 1 (Maximum obtainable product is 1*1)

Input: n = 3
Output: 2 (Maximum obtainable product is 1*2)

Input: n = 4
Output: 4 (Maximum obtainable product is 2*2)

Input: n = 5
```

```
Output: 6 (Maximum obtainable product is 2*3)
```

```
Input: n = 10
```

```
Output: 36 (Maximum obtainable product is 3*3*4)
```

1) Optimal Substructure:

This problem is similar to [Rod Cutting Problem](#). We can get the maximum product by making a cut at different positions and comparing the values obtained after a cut. We can recursively call the same function for a piece obtained after a cut.

Let $\text{maxProd}(n)$ be the maximum product for a rope of length n . $\text{maxProd}(n)$ can be written as following.

$\text{maxProd}(n) = \max(i*(n-i), \text{maxProdRec}(n-i)*i)$ for all i in $\{1, 2, 3 \dots n\}$

2) Overlapping Subproblems

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

```
// A Naive Recursive method to find maximum product
#include <iostream>
using namespace std;

// Utility function to get the maximum of two and three integers
int max(int a, int b) { return (a > b)? a : b; }
int max(int a, int b, int c) { return max(a, max(b, c)); }

// The main function that returns maximum product obtainable
// from a rope of length n
int maxProd(int n)
{
    // Base cases
    if (n == 0 || n == 1) return 0;

    // Make a cut at different places and take the maximum of all
    int max_val = 0;
    for (int i = 1; i < n; i++)
        max_val = max(max_val, i*(n-i), maxProd(n-i)*i);

    // Return the maximum of all values
    return max_val;
}

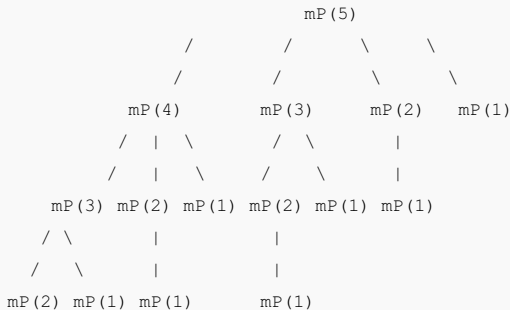
/* Driver program to test above functions */
int main()
{
    cout << "Maximum Product is " << maxProd(10);
    return 0;
}
```

Output:

```
Maximum Product is 36
```

Considering the above implementation, following is recursion tree for a Rope of length 5.

```
mP() ---> maxProd()
```



In the above partial recursion tree, $mP(3)$ is being solved twice. We can see that there are many subproblems which are solved again and again. Since same subproblems are called again, this problem has Overlapping Subproblems property. So the 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 `val[]` in bottom up manner.

```
// A Dynamic Programming solution for Max Product Problem
int maxProd(int n)
{
    int val[n+1];
    val[0] = val[1] = 0;

    // Build the table val[] in bottom up manner and return
    // the last entry from the table
    for (int i = 1; i <= n; i++)
    {
        int max_val = 0;
        for (int j = 1; j <= i/2; j++)
            max_val = max(max_val, (i-j)*j, j*val[i-j]);
        val[i] = max_val;
    }
    return val[n];
}
```

Time Complexity of the Dynamic Programming solution is $O(n^2)$ and it requires $O(n)$ extra space.

A Tricky Solution:

If we see some examples of this problems, we can easily observe following pattern. The maximum product can be obtained by repeatedly cutting parts of size 3 while size is greater than 4, keeping the last part as size of 2 or 3 or 4. For example, $n = 10$, the maximum product is obtained by 3, 3, 4. For $n = 11$, the maximum product is obtained by 3, 3, 3, 2. Following is C++ implementation of this approach.

```

#include <iostream>
using namespace std;

/* The main function that returns the max possible product */
int maxProd(int n)
{
    // n equals to 2 or 3 must be handled explicitly
    if (n == 2 || n == 3) return (n-1);

    // Keep removing parts of size 3 while n is greater than 4
    int res = 1;
    while (n > 4)
    {
        n -= 3;
        res *= 3; // Keep multiplying 3 to res
    }
    return (n * res); // The last part multiplied by previous parts
}

/* Driver program to test above functions */
int main()
{
    cout << "Maximum Product is " << maxProd(10);
    return 0;
}

```

Output:

```
Maximum Product is 36
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

128. How to check if a given number is Fibonacci number?

Given a number 'n', how to check if n is a Fibonacci number.

A simple way is to [generate Fibonacci numbers](#) until the generated number is greater than or equal to 'n'. Following is an interesting property about Fibonacci numbers that can also be used to check if a given number is Fibonacci or not.

*A number is Fibonacci if and only if one or both of $(5*n^2 + 4)$ or $(5*n^2 - 4)$ is a perfect square* (Source: [Wiki](#)). Following is a simple program based on this concept.

```

// C++ program to check if x is a perfect square
#include <iostream>
#include <math.h>
using namespace std;

// A utility function that returns true if x is perfect square
bool isPerfectSquare(int x)
{
    int s = sqrt(x);
    return (s*s == x);
}

// Returns true if n is a Fibonacci Number, else false
bool isFibonacci(int n)
{
    // n is Fibonacci if one of 5*n*n + 4 or 5*n*n - 4 or both
    // is a perfect square
    return isPerfectSquare(5*n*n + 4) ||
           isPerfectSquare(5*n*n - 4);
}

// A utility function to test above functions
int main()
{
    for (int i = 1; i <= 10; i++)
        isFibonacci(i)? cout << i << " is a Fibonacci Number \n":
                        cout << i << " is a not Fibonacci Number \n" ;
    return 0;
}

```

Output:

```

1 is a Fibonacci Number
2 is a Fibonacci Number
3 is a Fibonacci Number
4 is a not Fibonacci Number
5 is a Fibonacci Number
6 is a not Fibonacci Number
7 is a not Fibonacci Number
8 is a Fibonacci Number
9 is a not Fibonacci Number
10 is a not Fibonacci Number

```

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

129. Given n line segments, find if any two segments intersect

We have discussed the problem to detect if **two given line segments intersect or not**. In this post, we extend the problem. Here we are given n line segments and we need to find

out if any two line segments intersect or not.

Naive Algorithm A naive solution to solve this problem is to check every pair of lines and check if the pair intersects or not. We can check two line segments in $O(1)$ time. Therefore, this approach takes $O(n^2)$.

Sweep Line Algorithm: We can solve this problem in $O(n \log n)$ time using Sweep Line Algorithm. The algorithm first sorts the end points along the x axis from left to right, then it passes a vertical line through all points from left to right and checks for intersections. Following are detailed steps.

1) Let there be n given lines. There must be $2n$ end points to represent the n lines. Sort all points according to x coordinates. While sorting maintain a flag to indicate whether this point is left point of its line or right point.

2) Start from the leftmost point. Do following for every point

....a) If the current point is a left point of its line segment, check for intersection of its line segment with the segments just above and below it. And add its line to *active* line segments (line segments for which left end point is seen, but right end point is not seen yet). Note that we consider only those neighbors which are still active.

....b) If the current point is a right point, remove its line segment from active list and check whether its two active neighbors (points just above and below) intersect with each other.

The step 2 is like passing a vertical line from all points starting from the leftmost point to the rightmost point. That is why this algorithm is called Sweep Line Algorithm. The Sweep Line technique is useful in many other geometric algorithms like calculating the 2D Voronoi diagram

What data structures should be used for efficient implementation?

In step 2, we need to store all active line segments. We need to do following operations efficiently:

- a) Insert a new line segment
- b) Delete a line segment
- c) Find predecessor and successor according to y coordinate values

The obvious choice for above operations is Self-Balancing Binary Search Tree like AVL Tree, Red Black Tree. With a Self-Balancing BST, we can do all of the above operations in $O(\log n)$ time.

Also, in step 1, instead of sorting, we can use min heap data structure. Building a min heap takes $O(n)$ time and every extract min operation takes $O(\log n)$ time (See this).

PseudoCode:

The following pseudocode doesn't use heap. It simply sort the array.

```
sweepLineIntersection(Points[0..2n-1]) :
```

```
1. Sort Points[] from left to right (according to x coordinate)
```


2. Create an empty Self-Balancing BST T. It will contain all active line Segments ordered by y coordinate.

```
// Process all 2n points
```

3. for i = 0 to 2n-1

```
    // If this point is left end of its line
    if (Points[i].isLeft)
        T.insert(Points[i].line()) // Insert into the tree

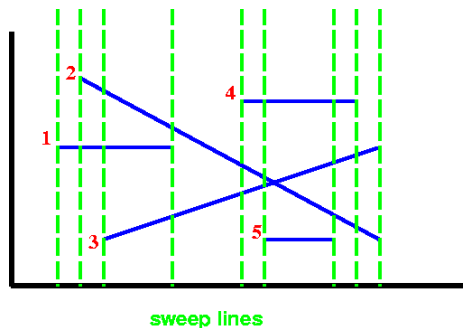
    // Check if this points intersects with its predecessor and successor
    if ( doIntersect(Points[i].line(), T.pred(Points[i].line()) )
        return true
    if ( doIntersect(Points[i].line(), T.succ(Points[i].line()) )
        return true

    else // If it's a right end of its line
        // Check if its predecessor and successor intersect with each other
        if ( doIntersect(T.pred(Points[i].line(), T.succ(Points[i].line()) )
            return true
        T.delete(Points[i].line()) // Delete from tree
```

4. return False

Example:

Let us consider the following example taken from [here](#). There are 5 line segments 1, 2, 3, 4 and 5. The dotted green lines show sweep lines.



Following are steps followed by the algorithm. All points from left to right are processed one by one. We maintain a self-balancing binary search tree.

Left end point of line segment 1 is processed: 1 is inserted into the Tree. The tree contains 1. No intersection.

Left end point of line segment 2 is processed: Intersection of 1 and 2 is checked. 2 is

inserted into the Tree. No intersection. The tree contains 1, 2.

Left end point of line segment 3 is processed: Intersection of 3 with 1 is checked. No intersection. 3 is inserted into the Tree. The tree contains 2, 1, 3.

Right end point of line segment 1 is processed: 1 is deleted from the Tree. Intersection of 2 and 3 is checked. Intersection of 2 and 3 is reported. The tree contains 2, 3. Note that the above pseudocode returns at this point. We can continue from here to report all intersection points.

Left end point of line segment 4 is processed: Intersections of line 4 with lines 2 and 3 are checked. No intersection. 4 is inserted into the Tree. The tree contains 2, 4, 3.

Left end point of line segment 5 is processed: Intersection of 5 with 3 is checked. No intersection. 4 is inserted into the Tree. The tree contains 2, 4, 3, 5.

Right end point of line segment 5 is processed: 5 is deleted from the Tree. The tree contains 2, 4, 3.

Right end point of line segment 4 is processed: 4 is deleted from the Tree. The tree contains 2, 4, 3. Intersection of 2 with 3 is checked. Intersection of 2 with 3 is reported. The tree contains 2, 3. Note that the intersection of 2 and 3 is reported again. We can add some logic to check for duplicates.

Right end point of line segment 2 and 3 are processed: Both are deleted from tree and tree becomes empty.

Time Complexity: The first step is sorting which takes $O(n \log n)$ time. The second step process $2n$ points and for processing every point, it takes $O(\log n)$ time. Therefore, overall time complexity is $O(n \log n)$

References:

<http://www.cs.uiuc.edu/~jeffe/teaching/373/notes/x06-sweepline.pdf>

<http://courses.csail.mit.edu/6.006/spring11/lectures/lec24.pdf>

<http://www.youtube.com/watch?v=dePDHVovJIE>

<http://www.eecs.wsu.edu/~cook/aa/lectures/l25/node10.html>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

130. Find the maximum distance covered using n bikes

There are n bikes and each can cover 100 km when fully fueled. What is the maximum amount of distance you can go using n bikes? You may assume that all bikes are similar

and a bike takes 1 litre to cover 1 km.

You have n bikes and using one bike you can only cover 100 km. so if n bikes start from same point and run simultaneously you can go only 100 km. Let's think bit differently, trick is when you want to cover maximum distance, you should always try to waste minimum fuel. Minimum wastage of fuel means to run minimum number of bikes. Instead of parallel running of n bikes, you can think of serially running them. That means if you transfer some amount of fuel from last bike to another bikes and throw the last bike i.e., don't run the last bike after certain point. But the question is, after what distance the fuel transfer has to be done so that the maximum distance is covered and fuel tank of remaining bikes do not overflow. Let us take following base cases and then generalize the solution.

Base Case 1: There is one bike This is simple, we can cover 100 kms only.

Base Case 2: There are two bikes: What is the maximum distance we can cover when there are 2 bikes? To maximize the distance, we must drop second bike at some point and transfer its fuel to first bike. Let we do the transfer after x kms.

```
Total distance covered = Distance covered by 100 ltr in first bike +  
                          Distance covered by fuel transferred from first bike.
```

Remaining fuel in second bike is $100 - x$. If we transfer this much fuel to first bike, then the total distance would become $100 + 100 - x$ which is $200 - x$. So our task is to maximize $200 - x$. The constraint is, $100 - x$ must be less than or equal to the space created in first bike after x kms, i.e., $100 - x \leq x$. The value of $200 - x$ becomes maximum when x is minimum. The minimum possible value of x is 50. So we are able to travel 150 kms.

Base Case 3: There are three bikes:

Let the first transfer is done after x kms. After x distance all bikes contain $100 - x$ amount of fuel. If we take $100 - x$ amount of fuel from 3rd bike and distribute it among 1st and 2nd bike so that fuel tanks of 1st and 2nd bikes get full. So $100 - x \leq 2 * x$; or, $x = 33.333$ so we should transfer the remaining fuel of third bike and distribute that amount of fuel among 1st and 2nd bike after exactly 33.33 km.

Let us generalize it. If we take a closer look at above cases, we can observe that if there are n bikes, then the first transfer is done (or a bike is dropped) after $100/n$ kms. To generalize it more, when we have x litre remaining fuel in every bike and n remaining bikes, we drop a bike after x/n kms.

Following is C implementation of a general function.

```
#include <stdio.h>
```

```
// Returns maximum distance that can be traveled by n bikes and given fuel in every bike
double maxDistance(int n, int fuel)
{
    // dist_covered is the result of this function
    double dist_covered = 0;

    while (n > 0)
    {
        // after ever fuel/n km we are discarding one bike and filling
        // all the other bikes with fuel/n liters of fuel i.e. to their
        // maximum limit (100 litre)

        dist_covered += (double)fuel / n;

        n -= 1; // reduce number of bikes
    }
    return dist_covered;
}
```

```
// Driver program to test above function
int main()
{
    int n = 3; // number of bikes
    int fuel = 100;
    printf("Maximum distance possible with %d bikes is %f",
           n, maxDistance(n, fuel));
    return 0;
}
```

Output:

```
Maximum distance possible with 3 bikes is 183.333333
```

This article is contributed by **Shamik Mitra**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

131. Closest Pair of Points | O(nlogn) Implementation

We are given an array of n points in the plane, and the problem is to find out the closest pair of points in the array. This problem arises in a number of applications. For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision. Recall the following formula for distance between two points p and q.

$$\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

We have discussed a [divide and conquer solution](#) for this problem. The time complexity of the implementation provided in the previous post is $O(n (\log n)^2)$. In this post, we

discuss an implementation with time complexity as $O(n \log n)$.

Following is a recap of the algorithm discussed in the previous post.

- 1) We sort all points according to x coordinates.
- 2) Divide all points in two halves.
- 3) Recursively find the smallest distances in both subarrays.
- 4) Take the minimum of two smallest distances. Let the minimum be d.
- 5) Create an array strip[] that stores all points which are at most d distance away from the middle line dividing the two sets.
- 6) Find the smallest distance in strip[].
- 7) Return the minimum of d and the smallest distance calculated in above step 6.

The great thing about the above approach is, if the array strip[] is sorted according to y coordinate, then we can find the smallest distance in strip[] in $O(n)$ time. In the implementation discussed in previous post, strip[] was explicitly sorted in every recursive call that made the time complexity $O(n (\log n)^2)$, assuming that the sorting step takes $O(n \log n)$ time.

In this post, we discuss an implementation where the time complexity is $O(n \log n)$. The idea is to presort all points according to y coordinates. Let the sorted array be Py[]. When we make recursive calls, we need to divide points of Py[] also according to the vertical line. We can do that by simply processing every point and comparing its x coordinate with x coordinate of middle line.

Following is C++ implementation of $O(n \log n)$ approach.

```
// A divide and conquer program in C++ to find the smallest distance f
// given set of points.

#include <iostream>
#include <float.h>
#include <stdlib.h>
#include <math.h>
using namespace std;

// A structure to represent a Point in 2D plane
struct Point
{
    int x, y;
};

/* Following two functions are needed for library function qsort().
Refer: http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */

// Needed to sort array of points according to X coordinate
int compareX(const void* a, const void* b)
{
    Point *p1 = (Point *)a, *p2 = (Point *)b;
```

```

        return (p1->x - p2->x);
    }
    // Needed to sort array of points according to Y coordinate
    int compareY(const void* a, const void* b)
    {
        Point *p1 = (Point *)a, *p2 = (Point *)b;
        return (p1->y - p2->y);
    }

    // A utility function to find the distance between two points
    float dist(Point p1, Point p2)
    {
        return sqrt( (p1.x - p2.x)*(p1.x - p2.x) +
                     (p1.y - p2.y)*(p1.y - p2.y) );
    }

    // A Brute Force method to return the smallest distance between two points in P[] of size n
    float bruteForce(Point P[], int n)
    {
        float min = FLT_MAX;
        for (int i = 0; i < n; ++i)
            for (int j = i+1; j < n; ++j)
                if (dist(P[i], P[j]) < min)
                    min = dist(P[i], P[j]);
        return min;
    }

    // A utility function to find minimum of two float values
    float min(float x, float y)
    {
        return (x < y)? x : y;
    }

    // A utility function to find the distance between the closest points of a
    // strip of given size. All points in strip[] are sorted according to
    // y coordinate. They all have an upper bound on minimum distance as d
    // Note that this method seems to be a O(n^2) method, but it's a O(n)
    // method as the inner loop runs at most 6 times
    float stripClosest(Point strip[], int size, float d)
    {
        float min = d; // Initialize the minimum distance as d

        // Pick all points one by one and try the next points till the difference
        // between y coordinates is smaller than d.
        // This is a proven fact that this loop runs at most 6 times
        for (int i = 0; i < size; ++i)
            for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)
                if (dist(strip[i], strip[j]) < min)
                    min = dist(strip[i], strip[j]);

        return min;
    }

    // A recursive function to find the smallest distance. The array Px contains
    // all points sorted according to x coordinates and Py contains all points
    // sorted according to y coordinates
    float closestUtil(Point Px[], Point Py[], int n)
    {
        // If there are 2 or 3 points, then use brute force
    }

```

```

    if (n <= 3)
        return bruteForce(Px, n);

    // Find the middle point
    int mid = n/2;
    Point midPoint = Px[mid];

    // Divide points in y sorted array around the vertical line.
    // Assumption: All x coordinates are distinct.
    Point Pyl[mid+1]; // y sorted points on left of vertical line
    Point Pyr[n-mid-1]; // y sorted points on right of vertical line
    int li = 0, ri = 0; // indexes of left and right subarrays
    for (int i = 0; i < n; i++)
    {
        if (Py[i].x <= midPoint.x)
            Pyl[li++] = Py[i];
        else
            Pyr[ri++] = Py[i];
    }

    // Consider the vertical line passing through the middle point
    // calculate the smallest distance dl on left of middle point and
    // dr on right side
    float dl = closestUtil(Px, Pyl, mid);
    float dr = closestUtil(Px + mid, Pyr, n-mid);

    // Find the smaller of two distances
    float d = min(dl, dr);

    // Build an array strip[] that contains points close (closer than d)
    // to the line passing through the middle point
    Point strip[n];
    int j = 0;
    for (int i = 0; i < n; i++)
        if (abs(Py[i].x - midPoint.x) < d)
            strip[j] = Py[i], j++;

    // Find the closest points in strip. Return the minimum of d and
    // distance is strip[]
    return min(d, stripClosest(strip, j, d) );
}

// The main function that finds the smallest distance
// This method mainly uses closestUtil()
float closest(Point P[], int n)
{
    Point Px[n];
    Point Py[n];
    for (int i = 0; i < n; i++)
    {
        Px[i] = P[i];
        Py[i] = P[i];
    }

    qsort(Px, n, sizeof(Point), compareX);
    qsort(Py, n, sizeof(Point), compareY);

    // Use recursive function closestUtil() to find the smallest distance
    return closestUtil(Px, Py, n);
}

// Driver program to test above functions

```

```
// Driver program to test above functions
int main()
{
    Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}}
    int n = sizeof(P) / sizeof(P[0]);
    cout << "The smallest distance is " << closest(P, n);
    return 0;
}
```

Output:

```
The smallest distance is 1.41421
```

Time Complexity: Let Time complexity of above algorithm be $T(n)$. Let us assume that we use a $O(n \log n)$ sorting algorithm. The above algorithm divides all points in two sets and recursively calls for two sets. After dividing, it finds the strip in $O(n)$ time. Also, it takes $O(n)$ time to divide the P_y array around the mid vertical line. Finally finds the closest points in strip in $O(n)$ time. So $T(n)$ can be expressed as follows

$$T(n) = 2T(n/2) + O(n) + O(n) + O(n)$$

$$T(n) = 2T(n/2) + O(n)$$

$$T(n) = T(n \log n)$$

References:

<http://www.cs.umd.edu/class/fall2013/cmsc451/Lects/lect10.pdf>

<http://www.youtube.com/watch?v=vS4Zn1a9KUc>

<http://www.youtube.com/watch?v=T3T7T8Ym20M>

http://en.wikipedia.org/wiki/Closest_pair_of_points_problem

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

132. Russian Peasant Multiplication

Given two integers, write a function to multiply them without using multiplication operator.

There are many other ways to multiply two numbers (For example, see [this](#)). One interesting method is the [Russian peasant algorithm](#). The idea is to double the first number and halve the second number repeatedly till the second number doesn't become 1. In the process, whenever the second number becomes odd, we add the first number to result (result is initialized as 0)

The following is simple algorithm.

Let the two given numbers be 'a' and 'b'

1) Initialize result 'res' as 0.

2) Do following while 'b' is greater than 0


```
a) If 'b' is odd, add 'a' to 'res'
b) Double 'a' and halve 'b'
3) Return 'res'.
```

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

```
// A method to multiply two numbers using Russian Peasant method
unsigned int russianPeasant(unsigned int a, unsigned int b)
{
    int res = 0; // initialize result

    // While second number doesn't become 1
    while (b > 0)
    {
        // If second number becomes odd, add the first number to result
        if (b & 1)
            res = res + a;

        // Double the first number and halve the second number
        a = a << 1;
        b = b >> 1;
    }
    return res;
}
```

```
// Driver program to test above function
int main()
{
    cout << russianPeasant(18, 1) << endl;
    cout << russianPeasant(20, 12) << endl;
    return 0;
}
```

Output:

```
18
240
```

How does this work?

The value of $a*b$ is same as $(a*2)^{(b/2)}$ if b is even, otherwise the value is same as $((a*2)^{(b/2)} + a)$. In the while loop, we keep multiplying 'a' with 2 and keep dividing 'b' by 2. If 'b' becomes odd in loop, we add 'a' to 'res'. When value of 'b' becomes 1, the value of 'res' + 'a', gives us the result.

Note that when 'b' is a power of 2, the 'res' would remain 0 and 'a' would have the multiplication. See the reference for more information.

Reference:

<http://mathforum.org/dr.math/faq/faq.peasant.html>

This article is compiled by **Shalki Agarwal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

133. Generate all unique partitions of an integer

Given a positive integer n , generate all possible unique ways to represent n as sum of positive integers.

```
Input: n = 2
Output:
2
1 1

Input: n = 3
Output:
3
2 1
1 1 1
Note: 2+1 and 1+2 are considered as duplicates.

Input: n = 4
Output:
4
3 1
2 2
2 1 1
1 1 1 1
```

We strongly recommend you to minimize the browser and try this yourself first.

Solution: We print all partition in sorted order and numbers within a partition are also printed in sorted order (as shown in the above examples). The idea is to get next partition using the values in current partition. We store every partition in an array $p[]$. We initialize $p[]$ as n where n is the input number. In every iteration, we first print $p[]$ and then update $p[]$ to store the next partition. So the main problem is to get next partition from a given partition.

Steps to get next partition from current partition

We are given current partition in $p[]$ and its size. We need to update $p[]$ to store next partition. Values in $p[]$ must be sorted in non-increasing order.

- 1) Find the rightmost non-one value in $p[]$ and store the count of 1's encountered before a non-one value in a variable rem_val (It indicates sum of values on right side to be updated). Let the index of non-one value be k .
- 2) Decrease the value of $p[k]$ by 1 and increase rem_val by 1. Now there may be two cases:

a) If $p[k]$ is more than or equal to rem_val . This is a simple case (we have the sorted order in new partition). Put rem_val at $p[k+1]$ and $p[0...k+1]$ is our new partition.

b) Else (This is an interesting case, take initial $p[]$ as $\{3, 1, 1, 1\}$, $p[k]$ is decreased from 3 to 2, rem_val is increased from 3 to 4, the next partition should be $\{2, 2, 2\}$).

Copy $p[k]$ to next position, increment k and reduce count by $p[k]$ while $p[k]$ is less than rem_val . Finally, put rem_val at $p[k+1]$ and $p[0...k+1]$ is our new partition. This step is like dividing rem_val in terms of $p[k]$ (4 is divided in 2's).

Following is C++ implementation of above algorithm.

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

// A utility function to print an array p[] of size 'n'
void printArray(int p[], int n)
{
    for (int i = 0; i < n; i++)
        cout << p[i] << " ";
    cout << endl;
}

void printAllUniqueParts(int n)
{
    int p[n]; // An array to store a partition
    int k = 0; // Index of last element in a partition
    p[k] = n; // Initialize first partition as number itself

    // This loop first prints current partition, then generates next
    // partition. The loop stops when the current partition has all 1s
    while (true)
    {
        // print current partition
        printArray(p, k+1);

        // Generate next partition

        // Find the rightmost non-one value in p[]. Also, update the
        // rem_val so that we know how much value can be accommodated
        int rem_val = 0;
        while (k >= 0 && p[k] == 1)
        {
            rem_val += p[k];
            k--;
        }

        // if k < 0, all the values are 1 so there are no more partitions
        if (k < 0) return;

        // Decrease the p[k] found above and adjust the rem_val
        p[k]--;
        rem_val++;

        // If rem_val is more, then the sorted order is violated. Divide
        // rem_val in different values of size p[k] and copy these values
        // different positions after p[k]
        while (rem_val > p[k])
        {
            p[k+1] = p[k];
            rem_val -= p[k];
            k++;
        }
        p[k+1] = rem_val;
    }
}
```

```

        rem_val = rem_val - p[k];
        k++;
    }

    // Copy rem_val to next position and increment position
    p[k+1] = rem_val;
    k++;
}
}

```

```

// Driver program to test above functions
int main()
{
    cout << "All Unique Partitions of 2 \n";
    printAllUniqueParts(2);

    cout << "\nAll Unique Partitions of 3 \n";
    printAllUniqueParts(3);

    cout << "\nAll Unique Partitions of 4 \n";
    printAllUniqueParts(4);

    return 0;
}

```

Output:

```

All Unique Partitions of 2
2
1 1

All Unique Partitions of 3
3
2 1
1 1 1

All Unique Partitions of 4
4
3 1
2 2
2 1 1
1 1 1 1

```

This article is contributed by **Hariprasad NG**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

134. Print all possible paths from top left to bottom right of a $m \times n$ matrix

The problem is to print all the possible paths from top left to bottom right of a $m \times n$ matrix with the constraints that **from each cell you can either move only to right or down**.

The algorithm is a simple recursive algorithm, from each cell first print all paths by going down and then print all paths by going right. Do this recursively for each cell encountered.

Following is C++ implementation of the above algorithm.

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

/* mat: Pointer to the starting of mXn matrix
   i, j: Current position of the robot (For the first call use 0,0)
   m, n: Dimention of given the matrix
   pi: Next index to be filed in path array
   *path[0..pi-1]: The path traversed by robot till now (Array to hold
                  path need to have space for at least m+n elements) */
void printAllPathsUtil(int *mat, int i, int j, int m, int n, int *path)
{
    // Reached the bottom of the matrix so we are left with
    // only option to move right
    if (i == m - 1)
    {
        for (int k = j; k < n; k++)
            path[pi + k - j] = *((mat + i*n) + k);
        for (int l = 0; l < pi + n - j; l++)
            cout << path[l] << " ";
        cout << endl;
        return;
    }

    // Reached the right corner of the matrix we are left with
    // only the downward movement.
    if (j == n - 1)
    {
        for (int k = i; k < m; k++)
            path[pi + k - i] = *((mat + k*n) + j);
        for (int l = 0; l < pi + m - i; l++)
            cout << path[l] << " ";
        cout << endl;
        return;
    }

    // Add the current cell to the path being generated
    path[pi] = *((mat + i*n) + j);

    // Print all the paths that are possible after moving down
    printAllPathsUtil(mat, i+1, j, m, n, path, pi + 1);

    // Print all the paths that are possible after moving right
    printAllPathsUtil(mat, i, j+1, m, n, path, pi + 1);

    // Print all the paths that are possible after moving diagonal
    // printAllPathsUtil(mat, i+1, j+1, m, n, path, pi + 1);
}
```

```
// The main function that prints all paths from top left to bottom right
// in a matrix 'mat' of size mXn
void printAllPaths(int *mat, int m, int n)
{
    int *path = new int[m+n];
    printAllPathsUtil(mat, 0, 0, m, n, path, 0);
}

// Driver program to test above functions
int main()
{
    int mat[2][3] = { {1, 2, 3}, {4, 5, 6} };
    printAllPaths(*mat, 2, 3);
    return 0;
}
```

Output:

```
1 4 5 6
1 2 5 6
1 2 3 6
```

Note that in the above code, the last line of `printAllPathsUtil()` is commented, If we uncomment this line, we get all the paths from the top left to bottom right of a $n \times m$ matrix if the diagonal movements are also allowed. And also if moving to some of the cells are not permitted then the same code can be improved by passing the restriction array to the above function and that is left as an exercise.

This article is contributed by **Hariprasad NG**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

135. Analysis of Algorithm | Set 4 (Solving Recurrences)

In the previous post, we discussed [analysis of loops](#). Many algorithms are recursive in nature. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes. For example in [Merge Sort](#), to sort a given array, we divide it in two halves and recursively repeat the process for the two halves. Finally we merge the results. Time complexity of Merge Sort can be written as $T(n) = 2T(n/2) + cn$. There are many other algorithms like Binary Search, Tower of Hanoi, etc.

There are mainly three ways for solving recurrences.

1) Substitution Method: We make a guess for the solution and then we use mathematical induction to prove the the guess is correct or incorrect.

For example consider the recurrence $T(n) = 2T(n/2) + n$

We guess the solution as $T(n) = O(n \log n)$. Now we use induction to prove our guess.

We need to prove that $T(n) \leq cn \log n$. We can assume that it is true for values smaller than n .

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\leq cn/2 \log(n/2) + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \\ &\leq cn \log n \end{aligned}$$

2) Recurrence Tree Method: In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically a arithmetic or geometric series.

For example consider the recurrence relation

$$T(n) = T(n/4) + T(n/2) + cn^2$$

$$\begin{array}{cc} & cn^2 \\ & / \quad \backslash \\ T(n/4) & \quad T(n/2) \end{array}$$

If we further break down the expression $T(n/4)$ and $T(n/2)$, we get following recursion tree.

$$\begin{array}{ccccccc} & & cn^2 & & & & \\ & & / \quad \backslash & & & & \\ & c(n^2)/16 & & c(n^2)/4 & & & \\ & / \quad \backslash & & / \quad \backslash & & & \\ T(n/16) & & T(n/8) & & T(n/8) & & T(n/4) \end{array}$$

Breaking down further gives us following

$$\begin{array}{ccccccccccc} & & & & cn^2 & & & & & & \\ & & & & / \quad \backslash & & & & & & \\ & & c(n^2)/16 & & & c(n^2)/4 & & & & & \\ & & / \quad \backslash & & & / \quad \backslash & & & & & \\ c(n^2)/256 & & c(n^2)/64 & & c(n^2)/64 & & c(n^2)/16 & & & & \\ & / \quad \backslash & & / \quad \backslash & & / \quad \backslash & & / \quad \backslash & & & \end{array}$$

To know the value of $T(n)$, we need to calculate sum of tree nodes level by level. If we sum the above tree level by level,

we get the following series

$$T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + \dots$$

The above series is geometrical progression with ratio $5/16$.

To get an upper bound, we can sum the infinite series.

We get the sum as $(n^2)/(1 - 5/16)$ which is $O(n^2)$

3) Master Method:

Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

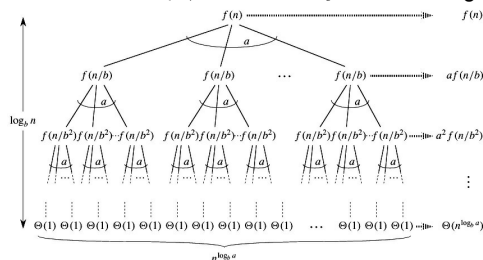
$$T(n) = aT(n/b) + f(n) \text{ where } a \geq 1 \text{ and } b > 1$$

There are following three cases:

1. If $f(n) = \Theta(n^c)$ where $c < \log_b a$ then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^c)$ where $c = \log_b a$ then $T(n) = \Theta(n^c \log n)$
3. If $f(n) = \Theta(n^c)$ where $c > \log_b a$ then $T(n) = \Theta(f(n))$

How does this work?

Master method is mainly derived from recurrence tree method. If we draw recurrence tree of $T(n) = aT(n/b) + f(n)$, we can see that the work done at root is $f(n)$ and work done at all leaves is $\Theta(n^c)$ where $c = \log_b a$. And the height of recurrence tree is $\log_b n$



In recurrence tree method, we calculate total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2). If work done at root is asymptotically more, then our result becomes work done at root (Case 3).

Examples of some standard algorithms whose time complexity can be evaluated using Master Method

Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$. It falls in case 2 as c is 1 and $\log_b a$ is also 1. So the solution is $\Theta(n \log n)$

Binary Search: $T(n) = T(n/2) + \Theta(1)$. It also falls in case 2 as c is 0 and $\log_b a$ is also 0. So the solution is $\Theta(\log n)$

Notes:

1) It is not necessary that a recurrence of the form $T(n) = aT(n/b) + f(n)$ can be solved using Master Theorem. The given three cases have some gaps between them. For example, the recurrence $T(n) = 2T(n/2) + n/\log n$ cannot be solved using master method.

2) Case 2 can be extended for $f(n) = \Theta(n^c \log^k n)$.

If $f(n) = \Theta(n^c \log^k n)$ for some constant $k \geq 0$ and $c = \log_b a$, then $T(n) = \Theta(n^c \log^{k+1} n)$

Practice Problems and Solutions on Master Theorem.

References:

http://en.wikipedia.org/wiki/Master_theorem

MIT Video Lecture on Asymptotic Notation | Recurrences | Substitution, Master Method
Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

136. Find if two rectangles overlap

Given two rectangles, find if the given two rectangles overlap or not.

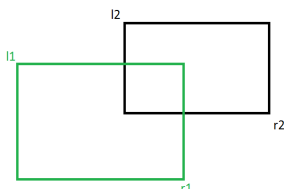
Note that a rectangle can be represented by two coordinates, top left and bottom right. So mainly we are given following four coordinates.

l1: Top Left coordinate of first rectangle.

r1: Bottom Right coordinate of first rectangle.

l2: Top Left coordinate of second rectangle.

r2: Bottom Right coordinate of second rectangle.



We need to write a function `bool doOverlap(l1, r1, l2, r2)` that returns true if the two given rectangles overlap.

One solution is to one by one pick all points of one rectangle and **see if the point lies inside the other rectangle or not**. This can be done using the algorithm discussed [here](#).

Following is a simpler approach. Two rectangles **do not** overlap if one of the following

conditions is true.

- 1) One rectangle is above top edge of other rectangle.
- 2) One rectangle is on left side of left edge of other rectangle.

We need to check above cases to find out if given rectangles overlap or not. Following is C++ implementation of the above approach.

```
#include<stdio.h>

struct Point
{
    int x, y;
};

// Returns true if two rectangles (l1, r1) and (l2, r2) overlap
bool doOverlap(Point l1, Point r1, Point l2, Point r2)
{
    // If one rectangle is on left side of other
    if (l1.x > r2.x || l2.x > r1.x)
        return false;

    // If one rectangle is above other
    if (l1.y < r2.y || l2.y < r1.y)
        return false;

    return true;
}

/* Driver program to test above function */
int main()
{
    Point l1 = {0, 10}, r1 = {10, 0};
    Point l2 = {5, 5}, r2 = {15, 0};
    if (doOverlap(l1, r1, l2, r2))
        printf("Rectangles Overlap");
    else
        printf("Rectangles Don't Overlap");
    return 0;
}
```

Output:

```
Rectangles Overlap
```

Time Complexity of above code is $O(1)$ as the code doesn't have any loop or recursion.

This article is compiled by **Aman Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

137. Tail Recursion

What is tail recursion?

A recursive function is tail recursive when recursive call is the last thing executed by the function. For example the following C++ function print() is tail recursive.

```
// An example of tail recursive function
void print(int n)
{
    if (n < 0) return;
    cout << " " << n;

    // The last executed statement is recursive call
    print(n-1);
}
```

Why do we care?

The tail recursive functions considered better than non tail recursive functions as tail-recursion can be optimized by compiler. The idea used by compilers to optimize tail-recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function's stack frame is of no use.

Can a non-tail recursive function be written as tail-recursive to optimize it?

Consider the following function to calculate factorial of n. It is a non-tail-recursive function. Although it looks like a tail recursive at first look. If we take a closer look, we can see that the value returned by fact(n-1) is used in fact(n), so the call to fact(n-1) is not the last thing done by fact(n)

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

// A NON-tail-recursive function. The function is not tail
// recursive because the value returned by fact(n-1) is used in
// fact(n) and call to fact(n-1) is not the last thing done by fact(n)
unsigned int fact(unsigned int n)
{
    if (n == 0) return 1;

    return n*fact(n-1);
}

// Driver program to test above function
int main()
{
    cout << fact(5);
    return 0;
}
```

The above function can be written as a tail recursive function. The idea is to use one more argument and accumulate the factorial value in second argument. When n reaches 0, return the accumulated value.

```

#include<iostream>
using namespace std;

// A tail recursive function to calculate factorial
unsigned factTR(unsigned int n, unsigned int a)
{
    if (n == 0) return a;
    return factTR(n-1, n*a);
}

// A wrapper over factTR
unsigned int fact(unsigned int n)
{
    return factTR(n, 1);
}

// Driver program to test above function
int main()
{
    cout << fact(5);
    return 0;
}

```

We will soon be discussing more on tail recursion.

References:

http://en.wikipedia.org/wiki/Tail_call

<http://c2.com/cgi/wiki?TailRecursion>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

138. Backtracking | Set 8 (Solving Cryptarithmic Puzzles)

Newspapers and magazines often have crypt-arithmetic puzzles of the form:

```

  SEND
+ MORE
-----
 MONEY
-----

```

The goal here is to assign each letter a digit from 0 to 9 so that the arithmetic works out correctly. The rules are that all occurrences of a letter must be assigned the same digit, and no digit can be assigned to more than one letter.

- First, create a list of all the characters that need assigning to pass to Solve

- If all characters are assigned, return true if puzzle is solved, false otherwise
- Otherwise, consider the first unassigned character
- for (every possible choice among the digits not in use)

make that choice and then recursively try to assign the rest of the characters

if recursion successful, return true

if !successful, unmake assignment and try another digit

If all digits have been tried and nothing worked, return false to trigger backtracking

```
/* ExhaustiveSolve
 * -----
 * This is the "not-very-smart" version of cryptarithmic solver. It takes
 * the puzzle itself (with the 3 strings for the two addends and sum) as a
 * string of letters as yet unassigned. If no more letters to assign
 * then we've hit a base-case, if the current letter-to-digit mapping solves
 * the puzzle, we're done, otherwise we return false to trigger backtracking.
 * If we have letters to assign, we take the first letter from that list and
 * try assigning it the digits from 0 to 9 and then recursively working
 * through solving puzzle from here. If we manage to make a good assignment
 * that works, we've succeeded, else we need to unassign that choice and try
 * another digit. This version is easy to write, since it uses a simple
 * approach (quite similar to permutations if you think about it) but it's
 * not so smart because it doesn't take into account the structure of the
 * puzzle constraints (for example, once the two digits for the addends
 * have been assigned, there is no reason to try anything other than the correct
 * digit for the sum) yet it tries a lot of useless combos regardless
 */
bool ExhaustiveSolve(puzzleT puzzle, string lettersToAssign)
{
    if (lettersToAssign.empty()) // no more choices to make
        return PuzzleSolved(puzzle); // checks arithmetic to see if works
    for (int digit = 0; digit <= 9; digit++) // try all digits
    {
        if (AssignLetterToDigit(lettersToAssign[0], digit))
        {
            if (ExhaustiveSolve(puzzle, lettersToAssign.substr(1)))
                return true;
            UnassignLetterFromDigit(lettersToAssign[0], digit);
        }
    }
    return false; // nothing worked, need to backtrack
}
```

The algorithm above actually has a lot in common with the permutations algorithm, it pretty much just creates all arrangements of the mapping from characters to digits and tries each until one works or all have been successfully tried. For a large puzzle, this could take a while.

A smarter algorithm could take into account the structure of the puzzle and avoid going down dead-end paths. For example, if we assign the characters starting from the ones place and moving to the left, at each stage, we can verify the correctness of what we have so far before we continue onwards. This definitely complicates the code but leads to a tremendous improvement in efficiency, making it much more feasible to solve large puzzles.

Below pseudocode in this case has more special cases, but the same general design

- Start by examining the rightmost digit of the topmost row, with a carry of 0
- If we are beyond the leftmost digit of the puzzle, return true if no carry, false otherwise
- If we are currently trying to assign a char in one of the addends
If char already assigned, just recur on row beneath this one, adding value into sum
If not assigned, then
 - for (every possible choice among the digits not in use)
make that choice and then on row beneath this one, if successful, return true
if !successful, unmake assignment and try another digit
 - return false if no assignment worked to trigger backtracking
- Else if trying to assign a char in the sum
- If char assigned & matches correct,
recur on next column to the left with carry, if success return true,
- If char assigned & doesn't match, return false
- If char unassigned & correct digit already used, return false
- If char unassigned & correct digit unused,
assign it and recur on next column to left with carry, if success return true
- return false to trigger backtracking

Source:

<http://see.stanford.edu/materials/icspaces106b/H19-RecBacktrackExamples.pdf>

139. Program for nth Catalan Number

Catalan numbers are a sequence of natural numbers that occurs in many interesting counting problems like following.

- 1) Count the number of expressions containing n pairs of parentheses which are correctly matched. For $n = 3$, possible expressions are $((()))$, $()(())$, $()()()$, $((())())$, $((())())$.
- 2) Count the number of possible Binary Search Trees with n keys (See [this](#))
- 3) Count the number of full binary trees (A rooted binary tree is full if every vertex has either two children or no children) with $n+1$ leaves.

See [this](#) for more applications.

The first few Catalan numbers for $n = 0, 1, 2, 3, \dots$ are **1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ...**

Recursive Solution

Catalan numbers satisfy the following recursive formula.

$$C_0 = 1 \quad \text{and} \quad C_{n+1} = \sum_{i=0}^n C_i C_{n-i} \quad \text{for } n \geq 0;$$

Following is C++ implementation of above recursive formula.

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

// A recursive function to find nth catalan number
unsigned long int catalan(unsigned int n)
{
    // Base case
    if (n <= 1) return 1;

    // catalan(n) is sum of catalan(i)*catalan(n-i-1)
    unsigned long int res = 0;
    for (int i=0; i<n; i++)
        res += catalan(i)*catalan(n-i-1);

    return res;
}

// Driver program to test above function
int main()
{
    for (int i=0; i<10; i++)
        cout << catalan(i) << " ";
    return 0;
}
```

Output :

```
1 1 2 5 14 42 132 429 1430 4862
```

Time complexity of above implementation is equivalent to nth catalan number.

$$T(n) = \sum_{i=0}^{n-1} T(i) * T(n-i) \quad \text{for } n \geq 0;$$

The value of nth catalan number is exponential that makes the time complexity exponential.

Dynamic Programming Solution

We can observe that the above recursive implementation does a lot of repeated work (we can the same by drawing recursion tree). Since there are overlapping subproblems, we can use dynamic programming for this. Following is a Dynamic programming based implementation in C++.

```

#include<iostream>
using namespace std;

// A dynamic programming based function to find nth
// Catalan number
unsigned long int catalanDP(unsigned int n)
{
    // Table to store results of subproblems
    unsigned long int catalan[n+1];

    // Initialize first two values in table
    catalan[0] = catalan[1] = 1;

    // Fill entries in catalan[] using recursive formula
    for (int i=2; i<=n; i++)
    {
        catalan[i] = 0;
        for (int j=0; j<i; j++)
            catalan[i] += catalan[j] * catalan[i-j-1];
    }

    // Return last entry
    return catalan[n];
}

// Driver program to test above function
int main()
{
    for (int i = 0; i < 10; i++)
        cout << catalanDP(i) << " ";
    return 0;
}

```

Output:

```
1 1 2 5 14 42 132 429 1430 4862
```

Time Complexity: Time complexity of above implementation is $O(n^2)$

Using Binomial Coefficient

We can also use the below formula to find nth catalan number in $O(n)$ time.

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

We have discussed a $O(n)$ approach to find binomial coefficient nCr .


```

#include<iostream>
using namespace std;

// Returns value of Binomial Coefficient C(n, k)
unsigned long int binomialCoeff(unsigned int n, unsigned int k)
{
    unsigned long int res = 1;

    // Since C(n, k) = C(n, n-k)
    if (k > n - k)
        k = n - k;

    // Calculate value of [n*(n-1)*---*(n-k+1)] / [k*(k-1)*---*1]
    for (int i = 0; i < k; ++i)
    {
        res *= (n - i);
        res /= (i + 1);
    }

    return res;
}

// A Binomial coefficient based function to find nth catalan
// number in O(n) time
unsigned long int catalan(unsigned int n)
{
    // Calculate value of 2nCn
    unsigned long int c = binomialCoeff(2*n, n);

    // return 2nCn/(n+1)
    return c/(n+1);
}

// Driver program to test above functions
int main()
{
    for (int i = 0; i < 10; i++)
        cout << catalan(i) << " ";
    return 0;
}

```

Output:

```
1 1 2 5 14 42 132 429 1430 4862
```

Time Complexity: Time complexity of above implementation is O(n).

We can also use below formula to find nth catalan number in O(n) time.

$$C_n = \frac{(2n)!}{(n+1)!n!} = \prod_{k=2}^n \frac{n+k}{k} \quad \text{for } n \geq 0.$$

References:

http://en.wikipedia.org/wiki/Catalan_number

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

140. Count trailing zeroes in factorial of a number

Given an integer n , write a function that returns count of trailing zeroes in $n!$.

Examples:

Input: $n = 5$

Output: 1

Factorial of 5 is 20 which has one trailing 0.

Input: $n = 20$

Output: 4

Factorial of 20 is 2432902008176640000 which has 4 trailing zeroes.

Input: $n = 100$

Output: 24

A simple method is to first calculate factorial of n , then count trailing 0s in the result (We can count trailing 0s by repeatedly dividing the factorial by 10 till the remainder is 0).

The above method can cause overflow for a slightly bigger numbers as factorial of a number is a big number (See factorial of 20 given in above examples). The idea is to consider **prime factors** of a factorial n . A trailing zero is always produced by prime factors 2 and 5. If we can count the number of 5s and 2s, our task is done. Consider the following examples.

$n = 5$: There is one 5 and 3 2s in prime factors of $5!$ ($2 * 2 * 2 * 3 * 5$). So count of trailing 0s is 1.

$n = 11$: There are two 5s and three 2s in prime factors of $11!$ ($2^8 * 3^4 * 5^2 * 7$). So count of trailing 0s is 2.

We can easily observe that the number of 2s in prime factors is always more than or equal to the number of 5s. So if we count 5s in prime factors, we are done. *How to count total number of 5s in prime factors of $n!$?* A simple way is to calculate $\text{floor}(n/5)$. For example, $7!$ has one 5, $10!$ has two 5s. It is done yet, there is one more thing to consider. Numbers like 25, 125, etc have more than one 5. For example if we consider $28!$, we get one extra 5 and number of 0s become 6. Handling this is simple, first divide n by 5 and remove all single 5s, then divide by 25 to remove extra 5s and so on. Following is the summarized formula for counting trailing 0s.

```
Trailing 0s in  $n!$  = Count of 5s in prime factors of  $n!$   
=  $\text{floor}(n/5) + \text{floor}(n/25) + \text{floor}(n/125) + \dots$ 
```

Following is C++ program based on above formula.

```
// C++ program to count trailing 0s in n!
#include <iostream>
using namespace std;

// Function to return trailing 0s in factorial of n
int findTrailingZeros(int n)
{
    // Initialize result
    int count = 0;

    // Keep dividing n by powers of 5 and update count
    for (int i=5; n/i>=1; i *= 5)
        count += n/i;

    return count;
}

// Driver program to test above function
int main()
{
    int n = 100;
    cout << "Count of trailing 0s in " << 100
        << "! is " << findTrailingZeros(n);
    return 0;
}
```

Output:

```
Count of trailing 0s in 100! is 24
```

This article is contributed by **Rahul Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

141. Horner's Method for Polynomial Evaluation

Given a polynomial of the form $c_n x^n + c_{n-1} x^{n-1} + c_{n-2} x^{n-2} + \dots + c_1 x + c_0$ and a value of x , find the value of polynomial for a given value of x . Here c_n, c_{n-1}, \dots are integers (may be negative) and n is a positive integer.

Input is in the form of an array say *poly[]* where *poly[0]* represents coefficient for x^n and *poly[1]* represents coefficient for x^{n-1} and so on.

Examples:

```
// Evaluate value of 2x^3 - 6x^2 + 2x - 1 for x = 3
Input: poly[] = {2, -6, 2, -1}, x = 3
Output: 5
```

```
// Evaluate value of  $2x^3 + 3x + 1$  for  $x = 2$   
Input: poly[] = {2, 0, 3, 1}, x = 2  
Output: 23
```

A naive way to evaluate a polynomial is to one by one evaluate all terms. First calculate x^n , multiply the value with c_n , repeat the same steps for other terms and return the sum.

Time complexity of this approach is $O(n^2)$ if we use a simple loop for evaluation of x^n . Time complexity can be improved to $O(n \log n)$ if we use [O\(Logn\) approach for evaluation of \$x^n\$](#) .

Horner's method can be used to evaluate polynomial in $O(n)$ time. To understand the method, let us consider the example of $2x^3 - 6x^2 + 2x - 1$. The polynomial can be evaluated as $((2x - 6)x + 2)x - 1$. The idea is to initialize result as coefficient of x^n which is 2 in this case, repeatedly multiply result with x and add next coefficient to result. Finally return result.

Following is C++ implementation of Horner's Method.

```
#include <iostream>  
using namespace std;  
  
// returns value of poly[0]x(n-1) + poly[1]x(n-2) + .. + poly[n-1]  
int horner(int poly[], int n, int x)  
{  
    int result = poly[0]; // Initialize result  
  
    // Evaluate value of polynomial using Horner's method  
    for (int i=1; i<n; i++)  
        result = result*x + poly[i];  
  
    return result;  
}  
  
// Driver program to test above function.  
int main()  
{  
    // Let us evaluate value of  $2x^3 - 6x^2 + 2x - 1$  for  $x = 3$   
    int poly[] = {2, -6, 2, -1};  
    int x = 3;  
    int n = sizeof(poly)/sizeof(poly[0]);  
    cout << "Value of polynomial is " << horner(poly, n, x);  
    return 0;  
}
```

Output:

```
Value of polynomial is 5
```

Time Complexity: $O(n)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

142. Write a function that generates one of 3 numbers according to given probabilities

You are given a function `rand(a, b)` which generates equiprobable random numbers between `[a, b]` inclusive. Generate 3 numbers `x, y, z` with probability $P(x), P(y), P(z)$ such that $P(x) + P(y) + P(z) = 1$ using the given `rand(a,b)` function.

The idea is to utilize the equiprobable feature of the `rand(a,b)` provided. **Let the given probabilities be in percentage form, for example $P(x)=40\%, P(y)=25\%, P(z)=35\%$.**

Following are the detailed steps.

- 1) Generate a random number between 1 and 100. Since they are equiprobable, the probability of each number appearing is $1/100$.
- 2) Following are some important points to note about generated random number 'r'.
 - a) 'r' is smaller than or equal to $P(x)$ with probability $P(x)/100$.
 - b) 'r' is greater than $P(x)$ and smaller than or equal $P(x) + P(y)$ with $P(y)/100$.
 - c) 'r' is greater than $P(x) + P(y)$ and smaller than or equal 100 (or $P(x) + P(y) + P(z)$) with probability $P(z)/100$.

```
// This function generates 'x' with probability px/100, 'y' with
// probability py/100 and 'z' with probability pz/100:
// Assumption: px + py + pz = 100 where px, py and pz lie
// between 0 to 100
int random(int x, int y, int z, int px, int py, int pz)
{
    // Generate a number from 1 to 100
    int r = rand(1, 100);

    // r is smaller than px with probability px/100
    if (r <= px)
        return x;

    // r is greater than px and smaller than or equal to px+py
    // with probability py/100
    if (r <= (px+py))
        return y;

    // r is greater than px+py and smaller than or equal to 100
    // with probability pz/100
    else
        return z;
}
```

This function will solve the purpose of generating 3 numbers with given three probabilities.

This article is contributed by **Harsh Agarwal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed

above

143. Find the smallest number whose digits multiply to a given number n

Given a number 'n', find the smallest number 'p' such that if we multiply all digits of 'p', we get 'n'. The result 'p' should have minimum two digits.

Examples:

```
Input: n = 36
Output: p = 49
// Note that 4*9 = 36 and 49 is the smallest such number

Input: n = 100
Output: p = 455
// Note that 4*5*5 = 100 and 455 is the smallest such number

Input: n = 1
Output: p = 11
// Note that 1*1 = 1

Input: n = 13
Output: Not Possible
```

For a given n, following are the two cases to be considered.

Case 1: $n < 10$ When n is smaller than n, the output is always $n+10$. For example for n = 7, output is 17. For n = 9, output is 19.

Case 2: $n \geq 10$ Find all factors of n which are between 2 and 9 (both inclusive). The idea is to start searching from 9 so that the number of digits in result are minimized. For example 9 is preferred over 33 and 8 is preferred over 24.

Store all found factors in an array. The array would contain digits in non-increasing order, so finally print the array in reverse order.

Following is C implementation of above concept.

```

#include<stdio.h>

// Maximum number of digits in output
#define MAX 50

// prints the smallest number whose digits multiply to n
void findSmallest(int n)
{
    int i, j=0;
    int res[MAX]; // To store digits of result in reverse order

    // Case 1: If number is smaller than 10
    if (n < 10)
    {
        printf("%d", n+10);
        return;
    }

    // Case 2: Start with 9 and try every possible digit
    for (i=9; i>1; i--)
    {
        // If current digit divides n, then store all
        // occurrences of current digit in res
        while (n%i == 0)
        {
            n = n/i;
            res[j] = i;
            j++;
        }
    }

    // If n could not be broken in form of digits (prime factors of n
    // are greater than 9)
    if (n > 10)
    {
        printf("Not possible");
        return;
    }

    // Print the result array in reverse order
    for (i=j-1; i>=0; i--)
        printf("%d", res[i]);
}

// Driver program to test above function
int main()
{
    findSmallest(7);
    printf("\n");

    findSmallest(36);
    printf("\n");

    findSmallest(13);
    printf("\n");

    findSmallest(100);
    return 0;
}

```

Output:

```
17
49
Not possible
455
```

This article is contributed by **Ashish Bansal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

144. Dynamic Programming | Set 37 (Boolean Parenthesization Problem)

Given a boolean expression with following symbols.

Symbols

```
'T' ---> true
'F' ---> false
```

And following operators filled between symbols

Operators

```
&    ---> boolean AND
|    ---> boolean OR
^    ---> boolean XOR
```

Count the number of ways we can parenthesize the expression so that the value of expression evaluates to true.

Let the input be in form of two arrays one contains the symbols (T and F) in order and other contains operators (&, | and ^)

Examples:

```
Input: symbol[]    = {T, F, T}
       operator[]  = {^, &}
```

Output: 2

The given expression is " $T \wedge F \wedge T$ ", it evaluates true in two ways " $((T \wedge F) \wedge T)$ " and " $(T \wedge (F \wedge T))$ "

```
Input: symbol[]    = {T, F, F}
       operator[]  = {^, |}
```

Output: 2

The given expression is " $T \wedge F | F$ ", it evaluates true in two ways " $((T \wedge F) | F)$ " and " $(T \wedge (F | F))$ ".


```
Input: symbol[]    = {T, T, F, T}
      operator[]  = { |, &, ^ }
```

Output: 4

The given expression is "T | T & F ^ T", it evaluates true in 4 ways ((T|T) & (F^T)), (T| (T& (F^T))), (((T|T) &F) ^T) and (T| ((T&F) ^T)).

Solution:

Let **T(i, j)** represents the number of ways to parenthesize the symbols between i and j (both inclusive) such that the subexpression between i and j evaluates to true.

$$T(i, j) = \sum_{k=i}^{j-1} \begin{cases} T(i, k) * T(k+1, j) & \text{If operator[k] is '&'} \\ Total(i, k) * Total(k+1, j) - F(i, k) * F(k+1, j) & \text{If operator[k] is '|'} \\ T(i, k) * F(k+1, j) + F(i, k) * T(k+1, j) & \text{If operator[k] is '^'} \end{cases}$$

Total(i, j) = T(i, j) + F(i, j)

Let **F(i, j)** represents the number of ways to parenthesize the symbols between i and j (both inclusive) such that the subexpression between i and j evaluates to false.

$$F(i, j) = \sum_{k=i}^{j-1} \begin{cases} Total(i, k) * Total(k+1, j) - T(i, k) * T(k+1, j) & \text{If operator[k] is '&'} \\ F(i, k) * F(k+1, j) & \text{If operator[k] is '|'} \\ T(i, k) * T(k+1, j) + F(i, k) * F(k+1, j) & \text{If operator[k] is '^'} \end{cases}$$

Total(i, j) = T(i, j) + F(i, j)

Base Cases:

```
T(i, i) = 1 if symbol[i] = 'T'
T(i, i) = 0 if symbol[i] = 'F'

F(i, i) = 1 if symbol[i] = 'F'
F(i, i) = 0 if symbol[i] = 'T'
```

If we draw recursion tree of above recursive solution, we can observe that it many overlapping subproblems. Like other [dynamic programming problems](#), it can be solved by filling a table in bottom up manner. Following is C++ implementation of dynamic programming solution.

```
#include<iostream>
#include<cstring>
using namespace std;
```

```
// Returns count of all possible parenthesizations that lead to
// result true for a boolean expression with symbols like true
// and false and operators like &, | and ^ filled between symbols
int countParenth(char symb[], char oper[], int n)
{
    int F[n][n], T[n][n];

    // Fill diagonal entries first
    // All diagonal entries in T[i][i] are 1 if symbol[i]
    // is T (true). Similarly, all F[i][i] entries are 1 if
    // symbol[i] is F (False)
    for (int i = 0; i < n; i++)
    {
        F[i][i] = (symb[i] == 'F')? 1: 0;
```

```

        T[i][i] = (symbols[i] == 'T')? 1: 0;
    }

    // Now fill T[i][i+1], T[i][i+2], T[i][i+3]... in order
    // And F[i][i+1], F[i][i+2], F[i][i+3]... in order
    for (int gap=1; gap<n; ++gap)
    {
        for (int i=0, j=gap; j<n; ++i, ++j)
        {
            T[i][j] = F[i][j] = 0;
            for (int g=0; g<gap; g++)
            {
                // Find place of parenthesization using current value
                // of gap
                int k = i + g;

                // Store Total[i][k] and Total[k+1][j]
                int tik = T[i][k] + F[i][k];
                int tkj = T[k+1][j] + F[k+1][j];

                // Follow the recursive formulas according to the current
                // operator
                if (oper[k] == '&')
                {
                    T[i][j] += T[i][k]*T[k+1][j];
                    F[i][j] += (tik*tkj - T[i][k]*T[k+1][j]);
                }
                if (oper[k] == '|')
                {
                    F[i][j] += F[i][k]*F[k+1][j];
                    T[i][j] += (tik*tkj - F[i][k]*F[k+1][j]);
                }
                if (oper[k] == '^')
                {
                    T[i][j] += F[i][k]*T[k+1][j] + T[i][k]*F[k+1][j];
                    F[i][j] += T[i][k]*T[k+1][j] + F[i][k]*F[k+1][j];
                }
            }
        }
    }

    return T[0][n-1];
}

```

```

// Driver program to test above function
int main()
{
    char symbols[] = "TTFT";
    char operators[] = "|&^";
    int n = strlen(symbols);

    // There are 4 ways
    // ((T|T)&(F^T)), (T|(T&(F^T))), (((T|T)&F)^T) and T|((T&F)^T))
    cout << countParenth(symbols, operators, n);
    return 0;
}

```

Output:

Time Complexity: $O(n^3)$

Auxiliary Space: $O(n^2)$

References:

http://people.cs.clemson.edu/~bcddean/dp_practice/dp_9.swf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

145. Calculate the angle between hour hand and minute hand

This problem is known as **Clock angle problem** where we need to find angle between hands of an analog clock at .

Examples:

```
Input:  h = 12:00, m = 30.00
```

```
Output: 165 degree
```

```
Input:  h = 3.00, m = 30.00
```

```
Output: 75 degree
```

The idea is to take 12:00 ($h = 12$, $m = 0$) as a reference. Following are detailed steps.

- 1) Calculate the angle made by hour hand with respect to 12:00 in h hours and m minutes.
- 2) Calculate the angle made by minute hand with respect to 12:00 in h hours and m minutes.
- 3) The difference between two angles is the angle between two hands.

How to calculate the two angles with respect to 12:00?

The minute hand moves 360 degree in 60 minute (or 6 degree in one minute) and hour hand moves 360 degree in 12 hours (or 0.5 degree in 1 minute). In h hours and m minutes, the minute hand would move $(h*60 + m)*6$ and hour hand would move $(h*60 + m)*0.5$.

```
// C program to find angle between hour and minute hands
#include <stdio.h>
#include <stdlib.h>

// Utility function to find minimum of two integers
int min(int x, int y) { return (x < y)? x: y; }

int calcAngle(double h, double m)
{
    // validate the input
    if (h < 0 || m < 0 || h > 12 || m > 60)
        printf("Wrong input");

    if (h == 12) h = 0;
    if (m == 60) m = 0;

    // Calculate the angles moved by hour and minute hands
    // with reference to 12:00
    int hour_angle = 0.5 * (h*60 + m);
    int minute_angle = 6*m;

    // Find the difference between two angles
    int angle = abs(hour_angle - minute_angle);

    // Return the smaller angle of two possible angles
    angle = min(360-angle, angle);

    return angle;
}

// Driver program to test above function
int main()
{
    printf("%d \n", calcAngle(9, 60));
    printf("%d \n", calcAngle(3, 30));
    return 0;
}
```

Output:

```
90
75
```

Exercise: Find all times when hour and minute hands get superimposed.

This article is contributed by **Ashish Bansal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

146. An Interesting Method to Generate Binary Numbers from 1 to n

Given a number n, write a function that generates and prints all binary numbers with decimal values from 1 to n.

Examples:

```
Input: n = 2
```

```
Output: 1, 10
```

```
Input: n = 5
```

```
Output: 1, 10, 11, 100, 101
```

A simple method is to run a loop from 1 to n, call decimal to binary inside the loop.

Following is an interesting method that uses [queue data structure](#) to print binary numbers. Thanks to [Vivek](#) for suggesting this approach.

- 1) Create an empty queue of strings
- 2) Enqueue the first binary number "1" to queue.
- 3) Now run a loop for generating and printing n binary numbers.
 -a) Dequeue and Print the front of queue.
 -b) Append "0" at the end of front item and enqueue it.
 -c) Append "1" at the end of front item and enqueue it.

Following is C++ implementation of above algorithm.

```

// C++ program to generate binary numbers from 1 to n
#include <iostream>
#include <queue>
using namespace std;

// This function uses queue data structure to print binary numbers
void generatePrintBinary(int n)
{
    // Create an empty queue of strings
    queue<string> q;

    // Enqueue the first binary number
    q.push("1");

    // This loops is like BFS of a tree with 1 as root
    // 0 as left child and 1 as right child and so on
    while (n-->0)
    {
        // print the front of queue
        string s1 = q.front();
        q.pop();
        cout << s1 << "\n";

        string s2 = s1; // Store s1 before changing it

        // Append "0" to s1 and enqueue it
        q.push(s1.append("0"));

        // Append "1" to s2 and enqueue it. Note that s2 contains
        // the previous front
        q.push(s2.append("1"));
    }
}

// Driver program to test above function
int main()
{
    int n = 10;
    generatePrintBinary(n);
    return 0;
}

```

Output:

```

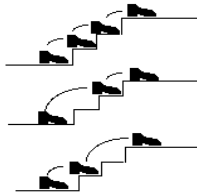
1
10
11
100
101
110
111
1000
1001
1010

```

This article is contributed by **Abhishek**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

147. Count ways to reach the n'th stair

There are n stairs, a person standing at the bottom wants to reach the top. The person can climb either 1 stair or 2 stairs at a time. Count the number of ways, the person can reach the top.



Consider the example shown in diagram. The value of n is 3. There are 3 ways to reach the top. The diagram is taken from [Easier Fibonacci puzzles](#)

More Examples:

Input: $n = 1$

Output: 1

There is only one way to climb 1 stair

Input: $n = 2$

Output: 2

There are two ways: (1, 1) and (2)

Input: $n = 4$

Output: 5

(1, 1, 1, 1), (1, 1, 2), (2, 1, 1), (1, 2, 1), (2, 2)

We can easily find recursive nature in above problem. The person can reach n 'th stair from either $(n-1)$ 'th stair or from $(n-2)$ 'th stair. Let the total number of ways to reach n 'th stair be 'ways(n)'. The value of 'ways(n)' can be written as following.

$$\text{ways}(n) = \text{ways}(n-1) + \text{ways}(n-2)$$

The above expression is actually the expression for [Fibonacci numbers](#), but there is one thing to notice, the value of ways(n) is equal to fibonacci($n+1$).

ways(1) = fib(2) = 1

ways(2) = fib(3) = 2

ways(3) = fib(4) = 3

So we can use function for fibonacci numbers to find the value of ways(n). Following is C++ implementation of the above idea.

```
// A C program to count number of ways to reach n't stair when  
// a person can climb 1, 2, ..m stairs at a time.  
#include<stdio.h>
```

```
// A simple recursive program to find n'th fibonacci number  
int fib(int n)  
{  
    if (n <= 1)  
        return n;  
    return fib(n-1) + fib(n-2);  
}  
  
// Returns number of ways to reach s'th stair  
int countWays(int s)  
{  
    return fib(s + 1);  
}
```

```
// Driver program to test above functions  
int main ()  
{  
    int s = 4;  
    printf("Number of ways = %d", countWays(s));  
    getchar();  
    return 0;  
}
```

Output:

```
Number of ways = 5
```

The time complexity of the above implementation is exponential (golden ratio raised to power n). It can be optimized to work in $O(\log n)$ time using the previously [discussed Fibonacci function optimizations](#).

Generalization of the above problem

How to count number of ways if the person can climb up to m stairs for a given value m?

For example if m is 4, the person can climb 1 stair or 2 stairs or 3 stairs or 4 stairs at a time.

We can write the recurrence as following.

$$\text{ways}(n, m) = \text{ways}(n-1, m) + \text{ways}(n-2, m) + \dots + \text{ways}(n-m, m)$$

Following is C++ implementation of above recurrence.


```
// A C program to count number of ways to reach n't stair when
// a person can climb either 1 or 2 stairs at a time
#include<stdio.h>
```

```
// A recursive function used by countWays
int countWaysUtil(int n, int m)
{
    if (n <= 1)
        return n;
    int res = 0;
    for (int i = 1; i<=m && i<=n; i++)
        res += countWaysUtil(n-i, m);
    return res;
}

// Returns number of ways to reach s'th stair
int countWays(int s, int m)
{
    return countWaysUtil(s+1, m);
}
```

```
// Driver program to test above functions
int main ()
{
    int s = 4, m = 2;
    printf("Nuber of ways = %d", countWays(s, m));
    return 0;
}
```

Output:

```
Number of ways = 5
```

The time complexity of above solution is exponential. It can be optimized to $O(mn)$ by using dynamic programming. Following is dynamic programming based solution. We build a table `res[]` in bottom up manner.

```
// A C program to count number of ways to reach n't stair when
// a person can climb 1, 2, ..m stairs at a time
#include<stdio.h>
```

```
// A recursive function used by countWays
int countWaysUtil(int n, int m)
{
    int res[n];
    res[0] = 1; res[1] = 1;
    for (int i=2; i<n; i++)
    {
        res[i] = 0;
        for (int j=1; j<=m && j<=i; j++)
            res[i] += res[i-j];
    }
    return res[n-1];
}

// Returns number of ways to reach s'th stair
int countWays(int s, int m)
{
    return countWaysUtil(s+1, m);
}
```

```
// Driver program to test above functions
int main ()
{
    int s = 4, m = 2;
    printf("Nuber of ways = %d", countWays(s, m));
    return 0;
}
```

Output:

```
Number of ways = 5
```

This article is contributed by **Abhishek**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

148. Data Structure for Dictionary and Spell Checker?

Which data structure can be used for efficiently building a word dictionary and Spell Checker?

The answer depends upon the functionalities required in Spell Checker and availability of memory. For example following are few possibilities.

Hashing is one simple option for this. We can put all words in a hash table. Refer [this](#) paper which compares hashing with self-balancing Binary Search Trees and Skip List, and shows that hashing performs better.

Hashing doesn't support operations like prefix search. Prefix search is something where a user types a prefix and your dictionary shows all words starting with that prefix. Hashing also doesn't support efficient printing of all words in dictionary in alphabetical order and nearest neighbor search.

If we want both operations, look up and prefix search, **Trie** is suited. With Trie, we can support all operations like insert, search, delete in $O(n)$ time where n is length of the word to be processed. Another advantage of Trie is, we can print all words in alphabetical order which is not possible with hashing.

The disadvantage of Trie is, it requires lots of space. If space is concern, then **Ternary Search Tree** can be preferred. In Ternary Search Tree, time complexity of search operation is $O(h)$ where h is height of the tree. Ternary Search Trees also supports other operations supported by Trie like prefix search, alphabetical order printing and nearest neighbor search.

If we want to support suggestions, like google shows "*did you mean ...*", then we need to find the closest word in dictionary. The closest word can be defined as the word that can be obtained with minimum number of character transformations (add, delete, replace). A Naive way is to take the given word and generate all words which are 1 distance (1 edit or 1 delete or 1 replace) away and one by one look them in dictionary. If nothing found, then look for all words which are 2 distant and so on. There are many complex algorithms for this. As per [the wiki page](#), The most successful algorithm to date is Andrew Golding and Dan Roth's Window-based spelling correction algorithm.

See [this](#) for a simple spell checker implementation.

This article is compiled by **Piyush**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

149. Connect n ropes with minimum cost

There are given n ropes of different lengths, we need to connect these ropes into one rope. The cost to connect two ropes is equal to sum of their lengths. We need to connect the ropes with minimum cost.

For example if we are given 4 ropes of lengths 4, 3, 2 and 6. We can connect the ropes in following ways.

- 1) First connect ropes of lengths 2 and 3. Now we have three ropes of lengths 4, 6 and 5.
- 2) Now connect ropes of lengths 4 and 5. Now we have two ropes of lengths 6 and 9.
- 3) Finally connect the two ropes and all ropes have connected.

Total cost for connecting all ropes is $5 + 9 + 15 = 29$. This is the optimized cost for connecting ropes. Other ways of connecting ropes would always have same or more cost. For example, if we connect 4 and 6 first (we get three strings of 3, 2 and 10), then connect 10 and 3 (we get two strings of 13 and 2). Finally we connect 13 and 2. Total cost in this way is $10 + 13 + 15 = 38$.

We strongly recommend to minimize the browser and try this yourself first.

If we observe the above problem closely, we can notice that the lengths of the ropes which are picked first are included more than once in total cost. Therefore, the idea is to connect smallest two ropes first and recur for remaining ropes. This approach is similar to [Huffman Coding](#). We put smallest ropes down the tree so that they can be repeated multiple times rather than the longer ropes.

Following is complete algorithm for finding the minimum cost for connecting n ropes.

Let there be n ropes of lengths stored in an array `len[0..n-1]`

- 1) Create a min heap and insert all lengths into the min heap.
- 2) Do following while number of elements in min heap is not one.
 -a) Extract the minimum and second minimum from min heap
 -b) Add the above two extracted values and insert the added value to the min-heap.
- 3) Return the value of only left item in min heap.

Following is C++ implementation of above algorithm.

```
// C++ program for connecting n ropes with minimum cost
#include <iostream>
using namespace std;

// A Min Heap: Collection of min heap nodes
struct MinHeap
{
    unsigned size; // Current size of min heap
    unsigned capacity; // capacity of min heap
    int *harr; // Array of minheap nodes
};

// A utility function to create a min heap of given capacity
struct MinHeap* createMinHeap(unsigned capacity)
{
    struct MinHeap* minHeap = new MinHeap;
    minHeap->size = 0; // current size is 0
    minHeap->capacity = capacity;
    minHeap->harr = new int[capacity];
    return minHeap;
}

// A utility function to swap two min heap nodes
void swapMinHeapNode(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// The standard minHeapify function.
void minHeapify(struct MinHeap* minHeap, int idx)
```

```

{
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    if (left < minHeap->size &&
        minHeap->harr[left] < minHeap->harr[smallest])
        smallest = left;

    if (right < minHeap->size &&
        minHeap->harr[right] < minHeap->harr[smallest])
        smallest = right;

    if (smallest != idx)
    {
        swapMinHeapNode(&minHeap->harr[smallest], &minHeap->harr[idx])
        minHeapify(minHeap, smallest);
    }
}

// A utility function to check if size of heap is 1 or not
int isSizeOne(struct MinHeap* minHeap)
{
    return (minHeap->size == 1);
}

// A standard function to extract minimum value node from heap
int extractMin(struct MinHeap* minHeap)
{
    int temp = minHeap->harr[0];
    minHeap->harr[0] = minHeap->harr[minHeap->size - 1];
    --minHeap->size;
    minHeapify(minHeap, 0);
    return temp;
}

// A utility function to insert a new node to Min Heap
void insertMinHeap(struct MinHeap* minHeap, int val)
{
    ++minHeap->size;
    int i = minHeap->size - 1;
    while (i && (val < minHeap->harr[(i - 1)/2]))
    {
        minHeap->harr[i] = minHeap->harr[(i - 1)/2];
        i = (i - 1)/2;
    }
    minHeap->harr[i] = val;
}

// A standard function to build min heap
void buildMinHeap(struct MinHeap* minHeap)
{
    int n = minHeap->size - 1;
    int i;
    for (i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
}

// Creates a min heap of capacity equal to size and inserts all values
// from len[] in it. Initially size of min heap is equal to capacity
struct MinHeap* createAndBuildMinHeap(int len[], int size)
{
    struct MinHeap* minHeap = createMinHeap(size);

```

```

    struct minHeap minHeap = createMinHeap(size);
    for (int i = 0; i < size; ++i)
        minHeap->harr[i] = len[i];
    minHeap->size = size;
    buildMinHeap(minHeap);
    return minHeap;
}

// The main function that returns the minimum cost to connect n ropes
// lengths stored in len[0..n-1]
int minCost(int len[], int n)
{
    int cost = 0; // Initialize result

    // Create a min heap of capacity equal to n and put all ropes in it
    struct MinHeap* minHeap = createAndBuildMinHeap(len, n);

    // Iterate while size of heap doesn't become 1
    while (!isSizeOne(minHeap))
    {
        // Extract two minimum length ropes from min heap
        int min = extractMin(minHeap);
        int sec_min = extractMin(minHeap);

        cost += (min + sec_min); // Update total cost

        // Insert a new rope in min heap with length equal to sum
        // of two extracted minimum lengths
        insertMinHeap(minHeap, min+sec_min);
    }

    // Finally return total minimum cost for connecting all ropes
    return cost;
}

// Driver program to test above functions
int main()
{
    int len[] = {4, 3, 2, 6};
    int size = sizeof(len)/sizeof(len[0]);
    cout << "Total cost for connecting ropes is " << minCost(len, size);
    return 0;
}

```

Output:

```
Total cost for connecting ropes is 29
```

Time Complexity: Time complexity of the algorithm is $O(n \log n)$ assuming that we use a $O(n \log n)$ sorting algorithm. Note that heap operations like insert and extract take $O(\log n)$ time.

Algorithmic Paradigm: Greedy Algorithm

A simple implementation with STL in C++

Following is a simple implementation that uses `priority_queue` available in STL. Thanks to Pango89 for providing below code.

```

#include<iostream>
#include<queue>
using namespace std;

int minCost(int arr[], int n)
{
    // Create a priority queue ( http://www.cplusplus.com/reference/queue/priority-queue/ )
    // By default 'less' is used which is for decreasing order
    // and 'greater' is used for increasing order
    priority_queue< int, vector<int>, greater<int> > pq(arr, arr+n);

    // Initialize result
    int res = 0;

    // While size of priority queue is more than 1
    while (pq.size() > 1)
    {
        // Extract shortest two ropes from pq
        int first = pq.top();
        pq.pop();
        int second = pq.top();
        pq.pop();

        // Connect the ropes: update result and
        // insert the new rope to pq
        res += first + second;
        pq.push(first + second);
    }

    return res;
}

// Driver program to test above function
int main()
{
    int len[] = {4, 3, 2, 6};
    int size = sizeof(len)/sizeof(len[0]);
    cout << "Total cost for connecting ropes is " << minCost(len, size);
    return 0;
}

```

Output:

```
Total cost for connecting ropes is 29
```

This article is compiled by **Abhishek**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

150. How to efficiently implement k stacks in a single array?

We have discussed [space efficient implementation of 2 stacks in a single array](#). In this post, a general solution for k stacks is discussed. Following is the detailed problem

statement.

Create a data structure kStacks that represents k stacks. Implementation of kStacks should use only one array, i.e., k stacks should use the same array for storing elements. Following functions must be supported by kStacks.

push(int x, int sn) → pushes x to stack number 'sn' where sn is from 0 to k-1

pop(int sn) → pops an element from stack number 'sn' where sn is from 0 to k-1

Method 1 (Divide the array in slots of size n/k)

A simple way to implement k stacks is to divide the array in k slots of size n/k each, and fix the slots for different stacks, i.e., use arr[0] to arr[n/k-1] for first stack, and arr[n/k] to arr[2n/k-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be n.

The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in arr[]. For example, say the k is 2 and array size (n) is 6 and we push 3 elements to first and do not push anything to second second stack. When we push 4th element to first, there will be overflow even if we have space for 3 more elements in array.

Method 2 (A space efficient implementation)

The idea is to use two extra arrays for efficient implementation of k stacks in an array. This may not make much sense for integer stacks, but stack items can be large for example stacks of employees, students, etc where every item is of hundreds of bytes. For such large stacks, the extra space used is comparatively very less as we use two integer arrays as extra space.

Following are the two extra arrays are used:

1) top[]: This is of size k and stores indexes of top elements in all stacks.

2) next[]: This is of size n and stores indexes of next item for the items in array arr[]. Here arr[] is actual array that stores k stacks.

Together with k stacks, a stack of free slots in arr[] is also maintained. The top of this stack is stored in a variable 'free'.

All entries in top[] are initialized as -1 to indicate that all stacks are empty. All entries next[i] are initialized as i+1 because all slots are free initially and pointing to next slot. Top of free stack, 'free' is initialized as 0.

Following is C++ implementation of the above idea.

```
// A C++ program to demonstrate implementation of k stacks in a single
// array in time and space efficient way
#include<iostream>
#include<climits>
using namespace std;

// A C++ class to represent k stacks in a single array of size n
class kStacks
{
    int *arr; // Array of size n to store actual content to be stored
```



```

int arr; // Array of size n to store actual contents to be stored
int *top; // Array of size k to store indexes of top elements of
int *next; // Array of size n to store next entry in all stacks
           // and free list

int n, k;
int free; // To store beginning index of free list
public:
    //constructor to create k stacks in an array of size n
    kStacks(int k, int n);

    // A utility function to check if there is space available
    bool isFull() { return (free == -1); }

    // To push an item in stack number 'sn' where sn is from 0 to k-1
    void push(int item, int sn);

    // To pop an from stack number 'sn' where sn is from 0 to k-1
    int pop(int sn);

    // To check whether stack number 'sn' is empty or not
    bool isEmpty(int sn) { return (top[sn] == -1); }
};

//constructor to create k stacks in an array of size n
kStacks::kStacks(int k1, int n1)
{
    // Initialize n and k, and allocate memory for all arrays
    k = k1, n = n1;
    arr = new int[n];
    top = new int[k];
    next = new int[n];

    // Initialize all stacks as empty
    for (int i = 0; i < k; i++)
        top[i] = -1;

    // Initialize all spaces as free
    free = 0;
    for (int i=0; i<n-1; i++)
        next[i] = i+1;
    next[n-1] = -1; // -1 is used to indicate end of free list
}

// To push an item in stack number 'sn' where sn is from 0 to k-1
void kStacks::push(int item, int sn)
{
    // Overflow check
    if (isFull())
    {
        cout << "\nStack Overflow\n";
        return;
    }

    int i = free; // Store index of first free slot

    // Update index of free slot to index of next slot in free list
    free = next[i];

    // Update next of top and then top for stack number 'sn'
    next[i] = top[sn];
    top[sn] = i;

    // Put the item in array

```

```

    arr[i] = item;
}

// To pop an from stack number 'sn' where sn is from 0 to k-1
int kStacks::pop(int sn)
{
    // Underflow check
    if (isEmpty(sn))
    {
        cout << "\nStack Underflow\n";
        return INT_MAX;
    }

    // Find index of top item in stack number 'sn'
    int i = top[sn];

    top[sn] = next[i]; // Change top to store next of previous top

    // Attach the previous top to the beginning of free list
    next[i] = free;
    free = i;

    // Return the previous top item
    return arr[i];
}

/* Driver program to test twStacks class */
int main()
{
    // Let us create 3 stacks in an array of size 10
    int k = 3, n = 10;
    kStacks ks(k, n);

    // Let us put some items in stack number 2
    ks.push(15, 2);
    ks.push(45, 2);

    // Let us put some items in stack number 1
    ks.push(17, 1);
    ks.push(49, 1);
    ks.push(39, 1);

    // Let us put some items in stack number 0
    ks.push(11, 0);
    ks.push(9, 0);
    ks.push(7, 0);

    cout << "Popped element from stack 2 is " << ks.pop(2) << endl;
    cout << "Popped element from stack 1 is " << ks.pop(1) << endl;
    cout << "Popped element from stack 0 is " << ks.pop(0) << endl;

    return 0;
}

```

Output:

```

Popped element from stack 2 is 45
Popped element from stack 1 is 39
Popped element from stack 0 is 7

```

Time complexities of operations push() and pop() is O(1).

The best part of above implementation is, if there is a slot available in stack, then an item can be pushed in any of the stacks, i.e., no wastage of space.

This article is contributed by **Sachin**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

151. Print squares of first n natural numbers without using *, / and -

Given a natural number 'n', print squares of first n natural numbers without using *, / and -.

```
Input:  n = 5
```

```
Output: 0 1 4 9 16
```

```
Input:  n = 6
```

```
Output: 0 1 4 9 16 25
```

We strongly recommend to minimize the browser and try this yourself first.

Method 1: The idea is to calculate next square using previous square value. Consider the following relation between square of x and (x-1). We know square of (x-1) is $(x-1)^2 - 2*x + 1$. We can write x^2 as

$$x^2 = (x-1)^2 + 2*x - 1$$

$$x^2 = (x-1)^2 + x + (x - 1)$$

When writing an iterative program, we can keep track of previous value of x and add the current and previous values of x to current value of square. This way we don't even use the '-' operator.

```
// C++ program to print squares of first 'n' natural numbers
// without using *, / and -
#include<iostream>
using namespace std;
```

```
void printSquares(int n)
{
    // Initialize 'square' and previous value of 'x'
    int square = 0, prev_x = 0;

    // Calculate and print squares
    for (int x = 0; x < n; x++)
    {
        // Update value of square using previous value
        square = (square + x + prev_x);

        // Print square and update prev for next iteration
        cout << square << " ";
        prev_x = x;
    }
}
```

```
// Driver program to test above function
int main()
{
    int n = 5;
    printSquares(n);
}
```

Output:

```
0 1 4 9 16
```

Method 2: Sum of first n odd numbers are squares of natural numbers from 1 to n . For example 1, $1+3$, $1+3+5$, $1+3+5+7$, $1+3+5+7+9$,

Following is C++ program based on above concept. Thanks to Aadithya Umashanker and raviteja for suggesting this method.

```
// C++ program to print squares of first 'n' natural numbers
// without using *, / and -
#include<iostream>
using namespace std;
```

```
void printSquares(int n)
{
    // Initialize 'square' and first odd number
    int square = 0, odd = 1;

    // Calculate and print squares
    for (int x = 0; x < n; x++)
    {
        // Print square
        cout << square << " ";

        // Update 'square' and 'odd'
        square = square + odd;
        odd = odd + 2;
    }
}
```

```
// Driver program to test above function
int main()
{
    int n = 5;
    printSquares(n);
}
```

Output:

```
0 1 4 9 16
```

This article is contributed by **Sachin**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

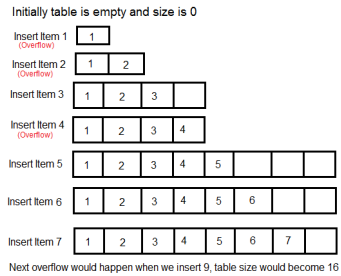
152. Analysis of Algorithm | Set 5 (Amortized Analysis Introduction)

Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation.

The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets and Splay Trees.

Let us consider an example of a simple hash table insertions. How do we decide table size? There is a trade-off between space and time, if we make hash-table size big, search time becomes fast, but space required becomes high.

- 1) Allocate memory for a larger table of size, typically twice the old table.
- 2) Cop the contents of old table to new table.
- 3) Free the old table.



What is the time complexity of n insertions using the above scheme?

Item No.	1	2	3	4	5	6	7	8	9	10
Table Size	1	2	4	4	8	8	8	8	16	16
Cost	1	2	3	1	5	1	1	1	9	1

Amortized Cost = $O(1)$

Following are few important notes.

2) The above Amortized Analysis done for Dynamic Array example is called **Aggregate Method**. There are two more powerful ways to do Amortized analysis called **Accounting Method** and **Potential Method**. We will be discussing the other two methods in separate posts.

3) The amortized analysis doesn't involve probability. There is also another different notion of average case running time where algorithms use randomization to make them faster and expected running time is faster than the worst case running time. These algorithms are analyzed using Randomized Analysis. Examples of these algorithms are

Randomized Quick Sort, Quick Select and Hashing. We will soon be covering Randomized analysis in a different post.

Sources:

Berkeley Lecture 35: Amortized Analysis

MIT Lecture 13: Amortized Algorithms, Table Doubling, Potential Method

<http://www.cs.cornell.edu/courses/cs3110/2011sp/lectures/lec20-amortized/amortized.htm>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

153. How to efficiently implement k Queues in a single array?

We have discussed [efficient implementation of k stack in an array](#). In this post, same for queue is discussed. Following is the detailed problem statement.

Create a data structure kQueues that represents k queues. Implementation of kQueues should use only one array, i.e., k queues should use the same array for storing elements. Following functions must be supported by kQueues.

enqueue(int x, int qn) → adds x to queue number 'qn' where qn is from 0 to k-1

dequeue(int qn) → deletes an element from queue number 'qn' where qn is from 0 to k-1

Method 1 (Divide the array in slots of size n/k)

A simple way to implement k queues is to divide the array in k slots of size n/k each, and fix the slots for different queues, i.e., use arr[0] to arr[n/k-1] for first queue, and arr[n/k] to arr[2n/k-1] for queue2 where arr[] is the array to be used to implement two queues and size of array be n.

The problem with this method is inefficient use of array space. An enqueue operation may result in overflow even if there is space available in arr[]. For example, consider k as 2 and array size n as 6. Let we enqueue 3 elements to first and do not enqueue anything to second second queue. When we enqueue 4th element to first queue, there will be overflow even if we have space for 3 more elements in array.

Method 2 (A space efficient implementation)

The idea is similar to the [stack post](#), here we need to use three extra arrays. In stack post, we needed to extra arrays, one more array is required because in queues, enqueue() and dequeue() operations are done at different ends.

Following are the three extra arrays are used:

- 1) **front[]**: This is of size k and stores indexes of front elements in all queues.
- 2) **rear[]**: This is of size k and stores indexes of rear elements in all queues.
- 2) **next[]**: This is of size n and stores indexes of next item for all items in array arr[].

Here arr[] is actual array that stores k stacks.

Together with k queues, a stack of free slots in arr[] is also maintained. The top of this stack is stored in a variable 'free'.

All entries in front[] are initialized as -1 to indicate that all queues are empty. All entries next[i] are initialized as i+1 because all slots are free initially and pointing to next slot. Top of free stack, 'free' is initialized as 0.

Following is C++ implementation of the above idea.

```
// A C++ program to demonstrate implementation of k queues in a single
// array in time and space efficient way
#include<iostream>
#include<climits>
using namespace std;

// A C++ class to represent k queues in a single array of size n
class kQueues
{
    int *arr;    // Array of size n to store actual content to be stored
    int *front;  // Array of size k to store indexes of front elements
    int *rear;   // Array of size k to store indexes of rear elements
    int *next;   // Array of size n to store next entry in all queues
                // and free list

    int n, k;
    int free;   // To store beginning index of free list
public:
    //constructor to create k queue in an array of size n
    kQueues(int k, int n);

    // A utility function to check if there is space available
    bool isFull() { return (free == -1); }

    // To enqueue an item in queue number 'qn' where qn is from 0 to k-1
    void enqueue(int item, int qn);

    // To dequeue an item from queue number 'qn' where qn is from 0 to k-1
    int dequeue(int qn);

    // To check whether queue number 'qn' is empty or not
    bool isEmpty(int qn) { return (front[qn] == -1); }
};

// Constructor to create k queues in an array of size n
kQueues::kQueues(int k1, int n1)
{
    // Initialize n and k, and allocate memory for all arrays
    k = k1, n = n1;
    arr = new int[n];
    front = new int[k];
    rear = new int[k];
    next = new int[n];

    // Initialize all queues as empty
```



```

    for (int i = 0; i < k; i++)
        front[i] = -1;

    // Initialize all spaces as free
    free = 0;
    for (int i=0; i<n-1; i++)
        next[i] = i+1;
    next[n-1] = -1; // -1 is used to indicate end of free list
}

// To enqueue an item in queue number 'qn' where qn is from 0 to k-1
void kQueues::enqueue(int item, int qn)
{
    // Overflow check
    if (isFull())
    {
        cout << "\nQueue Overflow\n";
        return;
    }

    int i = free; // Store index of first free slot

    // Update index of free slot to index of next slot in free list
    free = next[i];

    if (isEmpty(qn))
        front[qn] = i;
    else
        next[rear[qn]] = i;

    next[i] = -1;

    // Update next of rear and then rear for queue number 'qn'
    rear[qn] = i;

    // Put the item in array
    arr[i] = item;
}

// To dequeue an from queue number 'qn' where qn is from 0 to k-1
int kQueues::dequeue(int qn)
{
    // Underflow check
    if (isEmpty(qn))
    {
        cout << "\nQueue Underflow\n";
        return INT_MAX;
    }

    // Find index of front item in queue number 'qn'
    int i = front[qn];

    front[qn] = next[i]; // Change top to store next of previous top

    // Attach the previous front to the beginning of free list
    next[i] = free;
    free = i;

    // Return the previous front item
    return arr[i];
}

```

```

/* Driver program to test kStacks class */
int main()
{
    // Let us create 3 queue in an array of size 10
    int k = 3, n = 10;
    kQueues ks(k, n);

    // Let us put some items in queue number 2
    ks.enqueue(15, 2);
    ks.enqueue(45, 2);

    // Let us put some items in queue number 1
    ks.enqueue(17, 1);
    ks.enqueue(49, 1);
    ks.enqueue(39, 1);

    // Let us put some items in queue number 0
    ks.enqueue(11, 0);
    ks.enqueue(9, 0);
    ks.enqueue(7, 0);

    cout << "Dequeued element from queue 2 is " << ks.dequeue(2) << endl;
    cout << "Dequeued element from queue 1 is " << ks.dequeue(1) << endl;
    cout << "Dequeued element from queue 0 is " << ks.dequeue(0) << endl;

    return 0;
}

```

Output:

```

Dequeued element from queue 2 is 15
Dequeued element from queue 1 is 17
Dequeued element from queue 0 is 11

```

Time complexities of enqueue() and dequeue() is $O(1)$.

The best part of above implementation is, if there is a slot available in queue, then an item can be enqueued in any of the queues, i.e., no wastage of space. This method requires some extra space. Space may not be an issue because queue items are typically large, for example queues of employees, students, etc where every item is of hundreds of bytes. For such large queues, the extra space used is comparatively very less as we use three integer arrays as extra space.

This article is contributed by **Sachin**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

154. Mobile Numeric Keypad Problem

Given the mobile numeric keypad. You can only press buttons that are up, left, right or down to the current button. You are not allowed to press bottom row corner buttons (i.e. * and #). Given a number N, find out the number of possible numbers of given length.



Examples:

For N=1, number of possible numbers would be 10 (0, 1, 2, 3,, 9)

For N=2, number of possible numbers would be 36

Possible numbers: 00,08 11,12,14 22,21,23,25 and so on.

If we start with 0, valid numbers will be 00, 08 (count: 2)

If we start with 1, valid numbers will be 11, 12, 14 (count: 3)

If we start with 2, valid numbers will be 22, 21, 23,25 (count: 4)

If we start with 3, valid numbers will be 33, 32, 36 (count: 3)

If we start with 4, valid numbers will be 44,41,45,47 (count: 4)

If we start with 5, valid numbers will be 55,54,52,56,58 (count: 5)

.....

.....

We need to print the count of possible numbers.

We strongly recommend to minimize the browser and try this yourself first.

N = 1 is trivial case, number of possible numbers would be 10 (0, 1, 2, 3,, 9)

For N > 1, we need to start from some button, then move to any of the four direction (up, left, right or down) which takes to a valid button (should not go to *, #). Keep doing this until N length number is obtained (depth first traversal).

Recursive Solution:

Mobile Keypad is a rectangular grid of 4X3 (4 rows and 3 columns)

Lets say Count(i, j, N) represents the count of N length numbers starting from position (i, j)

```
If N = 1
    Count(i, j, N) = 10
Else
    Count(i, j, N) = Sum of all Count(r, c, N-1) where (r, c) is new
                    position after valid move of length 1 from current
                    position (i, j)
```

Following is C implementation of above recursive formula.

```
// A Naive Recursive C program to count number of possible numbers
// of given length
#include <stdio.h>

// left, up, right, down move from current location
int row[] = {0, 0, -1, 0, 1};
int col[] = {0, -1, 0, 1, 0};
```

```

// Returns count of numbers of length n starting from key position
// (i, j) in a numeric keyboard.
int getCountUtil(char keypad[][3], int i, int j, int n)
{
    if (keypad == NULL || n <= 0)
        return 0;

    // From a given key, only one number is possible of length 1
    if (n == 1)
        return 1;

    int k=0, move=0, ro=0, co=0, totalCount = 0;

    // move left, up, right, down from current location and if
    // new location is valid, then get number count of length
    // (n-1) from that new position and add in count obtained so far
    for (move=0; move<5; move++)
    {
        ro = i + row[move];
        co = j + col[move];
        if (ro >= 0 && ro <= 3 && co >=0 && co <= 2 &&
            keypad[ro][co] != '*' && keypad[ro][co] != '#')
        {
            totalCount += getCountUtil(keypad, ro, co, n-1);
        }
    }

    return totalCount;
}

// Return count of all possible numbers of length n
// in a given numeric keyboard
int getCount(char keypad[][3], int n)
{
    // Base cases
    if (keypad == NULL || n <= 0)
        return 0;
    if (n == 1)
        return 10;

    int i=0, j=0, totalCount = 0;
    for (i=0; i<4; i++) // Loop on keypad row
    {
        for (j=0; j<3; j++) // Loop on keypad column
        {
            // Process for 0 to 9 digits
            if (keypad[i][j] != '*' && keypad[i][j] != '#')
            {
                // Get count when number is starting from key
                // position (i, j) and add in count obtained so far
                totalCount += getCountUtil(keypad, i, j, n);
            }
        }
    }

    return totalCount;
}

```

```

// Driver program to test above function
int main(int argc, char *argv[])
{
    char keypad[4][3] = {{ '1', '2', '3' },
                        { '4', '5', '6' },

```

```

        {'7', '8', '9'},
        {'*', '0', '#'}}};
printf("Count for numbers of length %d: %d\n", 1, getCount(keypad,
printf("Count for numbers of length %d: %d\n", 2, getCount(keypad,
printf("Count for numbers of length %d: %d\n", 3, getCount(keypad,
printf("Count for numbers of length %d: %d\n", 4, getCount(keypad,
printf("Count for numbers of length %d: %d\n", 5, getCount(keypad,

return 0;
}

```

Output:

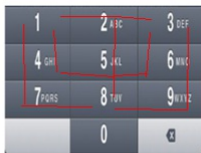
```

Count for numbers of length 1: 10
Count for numbers of length 2: 36
Count for numbers of length 3: 138
Count for numbers of length 4: 532
Count for numbers of length 5: 2062

```

Dynamic Programming

There are many repeated traversal on smaller paths (traversal for smaller N) to find all possible longer paths (traversal for bigger N). See following two diagrams for example. In this traversal, for N = 4 from two starting positions (buttons '4' and '8'), we can see there are few repeated traversals for N = 2 (e.g. 4 -> 1, 6 -> 3, 8 -> 9, 8 -> 7 etc).



Few traversals starting for button 8 for N = 4

e.g. 8 -> 7 -> 4 -> 1, 8 -> 9 -> 6 -> 3
8 -> 5 -> 4 -> 1, 8 -> 5 -> 6 -> 3
8 -> 5 -> 2 -> 2, 8 -> 5 -> 2 -> 3



Few traversals starting from button 5 for N = 4

e.g. 5 -> 8 -> 7 -> 4, 5 -> 8 -> 9 -> 6
5 -> 4 -> 1 -> 2, 5 -> 6 -> 3 -> 2
5 -> 2 -> 1 -> 4, 5 -> 2 -> 3 -> 6

Since the problem has both properties: **Optimal Substructure** and **Overlapping Subproblems**, it can be efficiently solved using dynamic programming.

Following is C program for dynamic programming implementation.

```

// A Dynamic Programming based C program to count number of
// possible numbers of given length

```

```
#include <stdio.h>
```

```
// Return count of all possible numbers of length n
// in a given numeric keyboard
int getCount(char keypad[][3], int n)
{
    if(keypad == NULL || n <= 0)
        return 0;
    if(n == 1)
        return 10;

    // left, up, right, down move from current location
    int row[] = {0, 0, -1, 0, 1};
    int col[] = {0, -1, 0, 1, 0};

    // taking n+1 for simplicity - count[i][j] will store
    // number count starting with digit i and length j
    int count[10][n+1];
    int i=0, j=0, k=0, move=0, ro=0, co=0, num = 0;
    int nextNum=0, totalCount = 0;

    // count numbers starting with digit i and of lengths 0 and 1
    for (i=0; i<=9; i++)
    {
        count[i][0] = 0;
        count[i][1] = 1;
    }

    // Bottom up - Get number count of length 2, 3, 4, ... , n
    for (k=2; k<=n; k++)
    {
        for (i=0; i<4; i++) // Loop on keypad row
        {
            for (j=0; j<3; j++) // Loop on keypad column
            {
                // Process for 0 to 9 digits
                if (keypad[i][j] != '*' && keypad[i][j] != '#')
                {
                    // Here we are counting the numbers starting with
                    // digit keypad[i][j] and of length k keypad[i][j]
                    // will become 1st digit, and we need to look for
                    // (k-1) more digits
                    num = keypad[i][j] - '0';
                    count[num][k] = 0;

                    // move left, up, right, down from current location
                    // and if new location is valid, then get number
                    // count of length (k-1) from that new digit and
                    // add in count we found so far
                    for (move=0; move<5; move++)
                    {
                        ro = i + row[move];
                        co = j + col[move];
                        if (ro >= 0 && ro <= 3 && co >=0 && co <= 2 &&
                            keypad[ro][co] != '*' && keypad[ro][co] != '#'
                        )
                        {
                            nextNum = keypad[ro][co] - '0';
                            count[num][k] += count[nextNum][k-1];
                        }
                    }
                }
            }
        }
    }
}
```

```

    }
}

// Get count of all possible numbers of length "n" starting
// with digit 0, 1, 2, ..., 9
totalCount = 0;
for (i=0; i<=9; i++)
    totalCount += count[i][n];
return totalCount;
}

// Driver program to test above function
int main(int argc, char *argv[])
{
    char keypad[4][3] = {{ '1', '2', '3' },
                        { '4', '5', '6' },
                        { '7', '8', '9' },
                        { '*', '0', '#' } };

    printf("Count for numbers of length %d: %d\n", 1, getCount(keypad, 1));
    printf("Count for numbers of length %d: %d\n", 2, getCount(keypad, 2));
    printf("Count for numbers of length %d: %d\n", 3, getCount(keypad, 3));
    printf("Count for numbers of length %d: %d\n", 4, getCount(keypad, 4));
    printf("Count for numbers of length %d: %d\n", 5, getCount(keypad, 5));

    return 0;
}

```

Output:

```

Count for numbers of length 1: 10
Count for numbers of length 2: 36
Count for numbers of length 3: 138
Count for numbers of length 4: 532
Count for numbers of length 5: 2062

```

A Space Optimized Solution:

The above dynamic programming approach also runs in $O(n)$ time and requires $O(n)$ auxiliary space, as only one for loop runs n times, other for loops runs for constant time. We can see that n th iteration needs data from $(n-1)$ th iteration only, so we need not keep the data from older iterations. We can have a space efficient dynamic programming approach with just two arrays of size 10. Thanks to Nik for suggesting this solution.

```

// A Space Optimized C program to count number of possible numbers
// of given length
#include <stdio.h>

```

```

// Return count of all possible numbers of length n
// in a given numeric keypad
int getCount(char keypad[][3], int n)
{
    if(keypad == NULL || n <= 0)
        return 0;
    if(n == 1)
        return 10;

    // odd[i], even[i] arrays represent count of numbers starting
    // with digit i for any length j
    int odd[10], even[10];
}

```

```

int odd[10], even[10];
int i = 0, j = 0, useOdd = 0, totalCount = 0;

for (i=0; i<=9; i++)
    odd[i] = 1; // for j = 1

for (j=2; j<=n; j++) // Bottom Up calculation from j = 2 to n
{
    useOdd = 1 - useOdd;

    // Here we are explicitly writing lines for each number 0
    // to 9. But it can always be written as DFS on 4X3 grid
    // using row, column array valid moves
    if(useOdd == 1)
    {
        even[0] = odd[0] + odd[8];
        even[1] = odd[1] + odd[2] + odd[4];
        even[2] = odd[2] + odd[1] + odd[3] + odd[5];
        even[3] = odd[3] + odd[2] + odd[6];
        even[4] = odd[4] + odd[1] + odd[5] + odd[7];
        even[5] = odd[5] + odd[2] + odd[4] + odd[8] + odd[6];
        even[6] = odd[6] + odd[3] + odd[5] + odd[9];
        even[7] = odd[7] + odd[4] + odd[8];
        even[8] = odd[8] + odd[0] + odd[5] + odd[7] + odd[9];
        even[9] = odd[9] + odd[6] + odd[8];
    }
    else
    {
        odd[0] = even[0] + even[8];
        odd[1] = even[1] + even[2] + even[4];
        odd[2] = even[2] + even[1] + even[3] + even[5];
        odd[3] = even[3] + even[2] + even[6];
        odd[4] = even[4] + even[1] + even[5] + even[7];
        odd[5] = even[5] + even[2] + even[4] + even[8] + even[6];
        odd[6] = even[6] + even[3] + even[5] + even[9];
        odd[7] = even[7] + even[4] + even[8];
        odd[8] = even[8] + even[0] + even[5] + even[7] + even[9];
        odd[9] = even[9] + even[6] + even[8];
    }
}

// Get count of all possible numbers of length "n" starting
// with digit 0, 1, 2, ..., 9
totalCount = 0;
if(useOdd == 1)
{
    for (i=0; i<=9; i++)
        totalCount += even[i];
}
else
{
    for (i=0; i<=9; i++)
        totalCount += odd[i];
}
return totalCount;
}

```

// Driver program to test above function

```

int main()
{
    char keypad[4][3] = {{ '1', '2', '3' },
                        { '4', '5', '6' },
                        { '7', '8', '9' },

```



```

        {'*', '0', '#'}
    };
    printf("Count for numbers of length %d: %d\n", 1, getCount(keypad,
    printf("Count for numbers of length %d: %d\n", 2, getCount(keypad,
    printf("Count for numbers of length %d: %d\n", 3, getCount(keypad,
    printf("Count for numbers of length %d: %d\n", 4, getCount(keypad,
    printf("Count for numbers of length %d: %d\n", 5, getCount(keypad,

    return 0;
}

```

Output:

```

Count for numbers of length 1: 10
Count for numbers of length 2: 36
Count for numbers of length 3: 138
Count for numbers of length 4: 532
Count for numbers of length 5: 2062

```

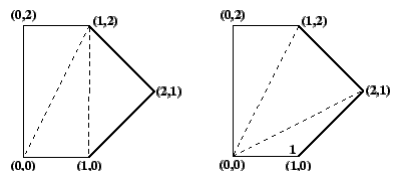
This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

155. Minimum Cost Polygon Triangulation

A triangulation of a convex polygon is formed by drawing diagonals between non-adjacent vertices (corners) such that the diagonals never intersect. The problem is to find the cost of triangulation with the minimum cost. The cost of a triangulation is sum of the weights of its component triangles. Weight of each triangle is its perimeter (sum of lengths of all sides)

See following example taken from [this](#) source.

Two triangulations of the same convex pentagon. The triangulation on the left has a cost of $8 + 2\sqrt{2} + 2\sqrt{5}$ (approximately 15.30), the one on the right has a cost of $4 + 2\sqrt{2} + 4\sqrt{5}$ (approximately 15.77).



This problem has recursive substructure. The idea is to divide the polygon into three parts: a single triangle, the sub-polygon to the left, and the sub-polygon to the right. We try all possible divisions like this and find the one that minimizes the cost of the triangle plus the cost of the triangulation of the two sub-polygons.

```

Let Minimum Cost of triangulation of vertices from i to j be minCost(i, j)
If j <= i + 2 Then

```

```
minCost(i, j) = 0
Else
    minCost(i, j) = Min { minCost(i, k) + minCost(k, j) + cost(i, k, j) }
                        Here k varies from 'i+1' to 'j-1'
```

Cost of a triangle formed by edges (i, j), (j, k) and (k, j) is

```
cost(i, j, k) = dist(i, j) + dist(j, k) + dist(k, j)
```

Following is C++ implementation of above naive recursive formula.

```

// Recursive implementation for minimum cost convex polygon triangulation
#include <iostream>
#include <cmath>
#define MAX 1000000.0
using namespace std;

// Structure of a point in 2D plane
struct Point
{
    int x, y;
};

// Utility function to find minimum of two double values
double min(double x, double y)
{
    return (x <= y)? x : y;
}

// A utility function to find distance between two points in a plane
double dist(Point p1, Point p2)
{
    return sqrt((p1.x - p2.x)*(p1.x - p2.x) +
                (p1.y - p2.y)*(p1.y - p2.y));
}

// A utility function to find cost of a triangle. The cost is considered
// as perimeter (sum of lengths of all edges) of the triangle
double cost(Point points[], int i, int j, int k)
{
    Point p1 = points[i], p2 = points[j], p3 = points[k];
    return dist(p1, p2) + dist(p2, p3) + dist(p3, p1);
}

// A recursive function to find minimum cost of polygon triangulation
// The polygon is represented by points[i..j].
double mTC(Point points[], int i, int j)
{
    // There must be at least three points between i and j
    // (including i and j)
    if (j < i+2)
        return 0;

    // Initialize result as infinite
    double res = MAX;

    // Find minimum triangulation by considering all
    for (int k=i+1; k<j; k++)
        res = min(res, (mTC(points, i, k) + mTC(points, k, j) +
                        cost(points, i, k, j)));

    return res;
}

// Driver program to test above functions
int main()
{
    Point points[] = {{0, 0}, {1, 0}, {2, 1}, {1, 2}, {0, 2}};
    int n = sizeof(points)/sizeof(points[0]);
    cout << mTC(points, 0, n-1);
    return 0;
}

```

15.3006

Following is C++ implementation of dynamic programming solution.

```
// A Dynamic Programming based program to find minimum cost of convex
// polygon triangulation
#include <iostream>
#include <cmath>
#define MAX 1000000.0
using namespace std;

// Structure of a point in 2D plane
struct Point
{
    int x, y;
};

// Utility function to find minimum of two double values
double min(double x, double y)
{
    return (x <= y)? x : y;
}

// A utility function to find distance between two points in a plane
double dist(Point p1, Point p2)
{
    return sqrt((p1.x - p2.x)*(p1.x - p2.x) +
                (p1.y - p2.y)*(p1.y - p2.y));
}

// A utility function to find cost of a triangle. The cost is considered
// as perimeter (sum of lengths of all edges) of the triangle
double cost(Point points[], int i, int j, int k)
{
    Point p1 = points[i], p2 = points[j], p3 = points[k];
    return dist(p1, p2) + dist(p2, p3) + dist(p3, p1);
}
```

```

// A Dynamic programming based function to find minimum cost for convex
// polygon triangulation.
double mTCDP(Point points[], int n)
{
    // There must be at least 3 points to form a triangle
    if (n < 3)
        return 0;

    // table to store results of subproblems. table[i][j] stores cost of
    // triangulation of points from i to j. The entry table[0][n-1] stores
    // the final result.
    double table[n][n];

    // Fill table using above recursive formula. Note that the table
    // is filled in diagonal fashion i.e., from diagonal elements to
    // table[0][n-1] which is the result.
    for (int gap = 0; gap < n; gap++)
    {
        for (int i = 0, j = gap; j < n; i++, j++)
        {
            if (j < i+2)
                table[i][j] = 0.0;
            else
            {
                table[i][j] = MAX;
                for (int k = i+1; k < j; k++)
                {
                    double val = table[i][k] + table[k][j] + cost(points, i, k, j);
                    if (table[i][j] > val)
                        table[i][j] = val;
                }
            }
        }
    }
    return table[0][n-1];
}

// Driver program to test above functions
int main()
{
    Point points[] = {{0, 0}, {1, 0}, {2, 1}, {1, 2}, {0, 2}};
    int n = sizeof(points)/sizeof(points[0]);
    cout << mTCDP(points, n);
    return 0;
}

```

Output:

15.3006

Time complexity of the above dynamic programming solution is $O(n^3)$.

Please note that the above implementations assume that the points of convex polygon are given in order (either clockwise or anticlockwise)

Exercise:

Extend the above solution to print triangulation also. For the above example, the optimal triangulation is 0 3 4, 0 1 3, and 1 2 3.

Sources:

<http://www.cs.utexas.edu/users/djimenez/utsa/cs3343/lecture12.html>

<http://www.cs.utoronto.ca/~heap/Courses/270F02/A4/chains/node2.html>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

156. Find n'th number in a number system with only 3 and 4

Given a number system with only 3 and 4. Find the nth number in the number system. First few numbers in the number system are: 3, 4, 33, 34, 43, 44, 333, 334, 343, 344, 433, 434, 443, 444, 3333, 3334, 3343, 3344, 3433, 3434, 3443, 3444, ...

Source: [Zoho Interview](#)

We strongly recommend to minimize the browser and try this yourself first.

We can generate all numbers with i digits using the numbers with (i-1) digits. The idea is to first add a '3' as prefix in all numbers with (i-1) digit, then add a '4'. For example, the numbers with 2 digits are 33, 34, 43 and 44. The numbers with 3 digits are 333, 334, 343, 344, 433, 434, 443 and 444 which can be generated by first adding a 3 as prefix, then 4.

Following are detailed steps.

```
1) Create an array 'arr[]' of strings size n+1.
2) Initialize arr[0] as empty string. (Number with 0 digits)
3) Do following while array size is smaller than or equal to n
.....a) Generate numbers by adding a 3 as prefix to the numbers generated
        in previous iteration. Add these numbers to arr[]
.....a) Generate numbers by adding a 4 as prefix to the numbers generated
        in previous iteration. Add these numbers to arr[]
```

Thanks to kaushik Lele for suggesting this idea in a comment [here](#). Following is C++ implementation for the same.

```

// C++ program to find n'th number in a number system with only 3 and 4
#include <iostream>
using namespace std;

// Function to find n'th number in a number system with only 3 and 4
void find(int n)
{
    // An array of strings to store first n numbers. arr[i] stores i'th
    string arr[n+1];
    arr[0] = ""; // arr[0] stores the empty string (String with 0 digits)

    // size indicates number of current elements in arr[]. m indicates
    // number of elements added to arr[] in previous iteration.
    int size = 1, m = 1;

    // Every iteration of following loop generates and adds 2*m numbers
    // arr[] using the m numbers generated in previous iteration.
    while (size <= n)
    {
        // Consider all numbers added in previous iteration, add a prefix
        // "3" to them and add new numbers to arr[]
        for (int i=0; i<m && (size+i)<=n; i++)
            arr[size + i] = "3" + arr[size - m + i];

        // Add prefix "4" to numbers of previous iteration and add new
        // numbers to arr[]
        for (int i=0; i<m && (size + m + i)<=n; i++)
            arr[size + m + i] = "4" + arr[size - m + i];

        // Update no. of elements added in previous iteration
        m = m<<1; // Or m = m*2;

        // Update size
        size = size + m;
    }
    cout << arr[n] << endl;
}

// Driver program to test above functions
int main()
{
    for (int i = 1; i < 16; i++)
        find(i);
    return 0;
}

```

Output:

```

3
4
33
34
43
44
333
334
343

```

344
433
434
443
444
3333

This article is contributed by **Raman**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

157. Print all increasing sequences of length k from first n natural numbers

Given two positive integers n and k, print all increasing sequences of length k such that the elements in every sequence are from first n natural numbers.

Examples:

Input: k = 2, n = 3

Output: 1 2
 1 3
 2 3

Input: k = 5, n = 5

Output: 1 2 3 4 5

Input: k = 3, n = 5

Output: 1 2 3
 1 2 4
 1 2 5
 1 3 4
 1 3 5
 1 4 5
 2 3 4
 2 3 5
 2 4 5
 3 4 5

We strongly recommend to minimize the browser and try this yourself first.

It's a good recursion question. The idea is to create an array of length k. The array stores current sequence. For every position in array, we check the previous element and

one by one put all elements greater than the previous element. If there is no previous element (first position), we put all numbers from 1 to n.

Following is C++ implementation of above idea.

```
// C++ program to print all increasing sequences of
// length 'k' such that the elements in every sequence
// are from first 'n' natural numbers.
#include<iostream>
using namespace std;

// A utility function to print contents of arr[0..k-1]
void printArr(int arr[], int k)
{
    for (int i=0; i<k; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// A recursive function to print all increasing sequences
// of first n natural numbers. Every sequence should be
// length k. The array arr[] is used to store current
// sequence.
void printSeqUtil(int n, int k, int &len, int arr[])
{
    // If length of current increasing sequence becomes k,
    // print it
    if (len == k)
    {
        printArr(arr, k);
        return;
    }

    // Decide the starting number to put at current position:
    // If length is 0, then there are no previous elements
    // in arr[]. So start putting new numbers with 1.
    // If length is not 0, then start from value of
    // previous element plus 1.
    int i = (len == 0)? 1 : arr[len-1] + 1;

    // Increase length
    len++;

    // Put all numbers (which are greater than the previous
    // element) at new position.
    while (i<=n)
    {
        arr[len-1] = i;
        printSeqUtil(n, k, len, arr);
        i++;
    }

    // This is important. The variable 'len' is shared among
    // all function calls in recursion tree. Its value must be
    // brought back before next iteration of while loop
    len--;
}

// This function prints all increasing sequences of
// first n natural numbers. The length of every sequence
// must be k. This function mainly uses printSeqUtil()
void printSeq(int n, int k)
```

```

void printSeq(int n, int k)
{
    int arr[k]; // An array to store individual sequences
    int len = 0; // Initial length of current sequence
    printSeqUtil(n, k, len, arr);
}

// Driver program to test above functions
int main()
{
    int k = 3, n = 7;
    printSeq(n, k);
    return 0;
}

```

Output:

```

1 2 3
1 2 4
1 2 5
1 2 6
1 2 7
1 3 4
1 3 5
1 3 6
1 3 7
1 4 5
1 4 6
1 4 7
1 5 6
1 5 7
1 6 7
2 3 4
2 3 5
2 3 6
2 3 7
2 4 5
2 4 6
2 4 7
2 5 6
2 5 7
2 6 7
3 4 5
3 4 6
3 4 7
3 5 6
3 5 7
3 6 7
4 5 6

```

4 5 7
4 6 7
5 6 7

This article is contributed by **Arjun**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

158. Binomial Heap

The main application of **Binary Heap** is as implement priority queue. Binomial Heap is an extension of **Binary Heap** that provides faster union or merge operation together with other operations provided by Binary Heap.

A Binomial Heap is a collection of Binomial Trees

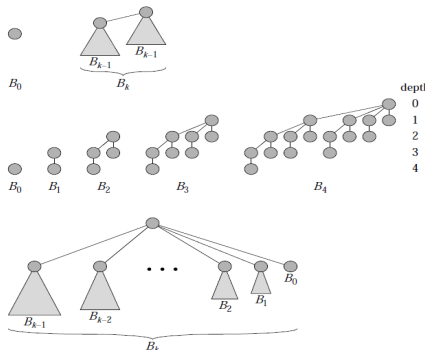
What is a Binomial Tree?

A Binomial Tree of order 0 has 1 node. A Binomial Tree of order k can be constructed by taking two binomial trees of order $k-1$, and making one as leftmost child of other.

A Binomial Tree of order k has following properties.

- It has exactly 2^k nodes.
- It has depth as k .
- There are exactly kC_i nodes at depth i for $i = 0, 1, \dots, k$.
- The root has degree k and children of root are themselves Binomial Trees with order $k-1, k-2, \dots, 0$ from left to right.

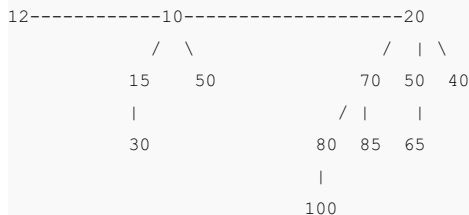
The following diagram is taken from 2nd Edition of **CLRS book**.



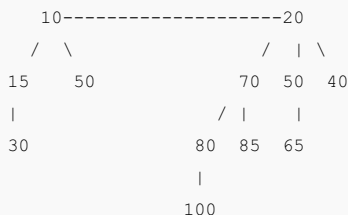
Binomial Heap:

A Binomial Heap is a set of Binomial Trees where each Binomial Tree follows Min Heap property. And there can be at-most one Binomial Tree of any degree.

Examples Binomial Heap:



A Binomial Heap with 13 nodes. It is a collection of 3 Binomial Trees of orders 0, 2 and 3 from left to right.



A Binomial Heap with 12 nodes. It is a collection of 2 Binomial Trees of orders 2 and 3 from left to right.

Binary Representation of a number and Binomial Heaps

A Binomial Heap with n nodes has number of Binomial Trees equal to the number of set bits in Binary representation of n . For example let n be 13, there 3 set bits in binary representation of n (00001101), hence 3 Binomial Trees. We can also relate degree of these Binomial Trees with positions of set bits. With this relation we can conclude that there are $O(\log n)$ Binomial Trees in a Binomial Heap with ' n ' nodes.

Operations of Binomial Heap:

The main operation in Binomial Heap is `union()`, all other operations mainly use this operation. The `union()` operation is to combine two Binomial Heaps into one. Let us first discuss other operations, we will discuss `union` later.

- 1) insert(H, k):** Inserts a key ' k ' to Binomial Heap ' H '. This operation first creates a Binomial Heap with single key ' k ', then calls `union` on H and the new Binomial heap.
- 2) getMin(H):** A simple way to `getMin()` is to traverse the list of root of Binomial Trees and return the minimum key. This implementation requires $O(\log n)$ time. It can be optimized to $O(1)$ by maintaining a pointer to minimum key root.
- 3) extractMin(H):** This operation also uses `union()`. We first call `getMin()` to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally we call `union()` on H and the newly created Binomial Heap. This operation requires $O(\log n)$ time.

4) delete(H): Like Binary Heap, delete operation first reduces the key to minus infinite, then calls `extractMin()`.

5) decreaseKey(H): `decreaseKey()` is also similar to Binary Heap. We compare the decreases key with it parent and if parent's key is less, we swap keys and recur for parent. We stop when we either reach a node whose parent has smaller key or we hit the root node. Time complexity of `decreaseKey()` is $O(\text{Log}n)$.

Union operation in Binomial Heap:

Given two Binomial Heaps H1 and H2, `union(H1, H2)` creates a single Binomial Heap.

1) The first step is to simply merge the two Heaps in non-decreasing order of degrees. In the following diagram, figure(b) shows the result after merging.

2) After the simple merge, we need to make sure that there is at-most one Binomial Tree of any order. To do this, we need to combine Binomial Trees of same order. We traverse the list of merged roots, we keep track of three pointers, `prev`, `x` and `next-x`. There can be following 4 cases when we traverse the list of roots.

—Case 1: Orders of `x` and `next-x` are not same, we simply move ahead.

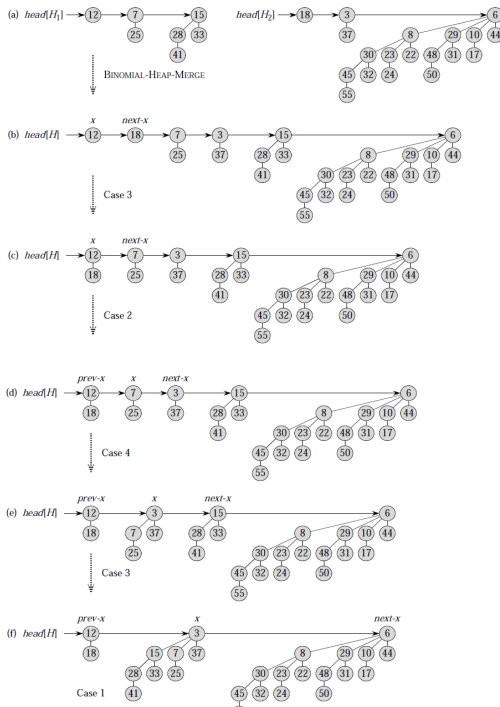
In following 3 cases orders of `x` and `next-x` are same.

—Case 2: If order of `next-next-x` is also same, move ahead.

—Case 3: If key of `x` is smaller than or equal to key of `next-x`, then make `next-x` as a child of `x` by linking it with `x`.

—Case 4: If key of `x` is greater, then make `x` as child of `next`.

The following diagram is taken from 2nd Edition of [CLRS book](#).



How to represent Binomial Heap?

A Binomial Heap is a set of Binomial Trees. A Binomial Tree must be represented in a way that allows sequential access to all siblings, starting from the leftmost sibling (We need this in `insert()` and `extractMin()` and `delete()`). The idea is to represent Binomial Trees as leftmost child and right-sibling representation, i.e., every node stores two pointers, one to the leftmost child and other to the right sibling.

We will soon be discussing implementation of Binomial Heap.

Sources:

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

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

159. Count Distinct Non-Negative Integer Pairs (x, y) that Satisfy the Inequality $x^2 + y^2 < n$

Given a positive number n , count all distinct Non-Negative Integer pairs (x, y) that satisfy the inequality $x*x + y*y < n$.

Examples:

```
Input:  n = 5
Output: 6
The pairs are (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (0, 2)
```

```
Input: n = 6
Output: 8
The pairs are (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (0, 2),
              (1, 2), (2, 1)
```

A **Simple Solution** is to run two loops. The outer loop goes for all possible values of x (from 0 to \sqrt{n}). The inner loops picks all possible values of y for current value of x (picked by outer loop). Following is C++ implementation of simple solution.

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

// This function counts number of pairs (x, y) that satisfy
// the inequality x*x + y*y < n.
int countSolutions(int n)
{
    int res = 0;
    for (int x = 0; x*x < n; x++)
        for (int y = 0; x*x + y*y < n; y++)
            res++;
    return res;
}

// Driver program to test above function
int main()
{
    cout << "Total Number of distinct Non-Negative pairs is "
          << countSolutions(6) << endl;
    return 0;
}
```

Output:

```
Total Number of distinct Non-Negative pairs is 8
```

An upper bound for time complexity of the above solution is $O(n)$. The outer loop runs \sqrt{n} times. The inner loop runs at most \sqrt{n} times.

Using an **Efficient Solution**, we can find the count in $O(\sqrt{n})$ time. The idea is to first find the count of all y values corresponding the 0 value of x . Let count of distinct y values be $yCount$. We can find $yCount$ by running a loop and comparing $yCount*yCount$ with n . After we have initial $yCount$, we can one by one increase value of x and find the next value of $yCount$ by reducing $yCount$.

```

// An efficient C program to find different (x, y) pairs that
// satisfy  $x*x + y*y < n$ .
#include <iostream>
using namespace std;

// This function counts number of pairs (x, y) that satisfy
// the inequality  $x*x + y*y < n$ .
int countSolutions(int n)
{
    int x = 0, yCount, res = 0;

    // Find the count of different y values for x = 0.
    for (yCount = 0; yCount*yCount < n; yCount++) ;

    // One by one increase value of x, and find yCount for
    // current x. If yCount becomes 0, then we have reached
    // maximum possible value of x.
    while (yCount != 0)
    {
        // Add yCount (count of different possible values of y
        // for current x) to result
        res += yCount;

        // Increment x
        x++;

        // Update yCount for current x. Keep reducing yCount while
        // the inequality is not satisfied.
        while (yCount != 0 && (x*x + (yCount-1)*(yCount-1) >= n))
            yCount--;
    }

    return res;
}

// Driver program to test above function
int main()
{
    cout << "Total Number of distinct Non-Negative pairs is "
         << countSolutions(6) << endl;
    return 0;
}

```

Output:

```
Total Number of distinct Non-Negative pairs is 8
```

Time Complexity of the above solution seems more but if we take a closer look, we can see that it is $O(\sqrt{n})$. In every step inside the inner loop, value of yCount is decremented by 1. The value yCount can decrement at most $O(\sqrt{n})$ times as yCount is count y values for $x = 0$. In the outer loop, the value of x is incremented. The value of x can also increment at most $O(\sqrt{n})$ times as the last x is for yCount equals to 1.

This article is contributed by **Sachin Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

160. Algorithm Practice Question for Beginners | Set 1

Consider the following C function.

```
unsigned fun(unsigned n)
{
    if (n == 0) return 1;
    if (n == 1) return 2;

    return fun(n-1) + fun(n-1);
}
```

Consider the following questions for above code ignoring compiler optimization.

- What does the above code do?
- What is the time complexity of above code?
- Can the time complexity of above function be reduced?

What does fun(n) do?

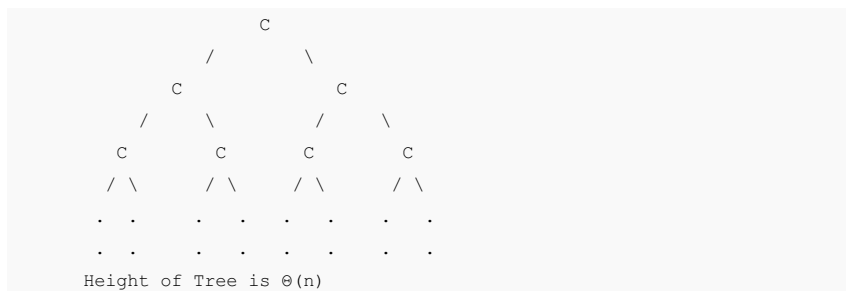
In the above code, fun(n) is equal to $2^{\text{fun}(n-1)}$. So the above function returns 2^n . For example, for $n = 3$, it returns 8, for $n = 4$, it returns 16.

What is the time complexity of fun(n)?

Time complexity of the above function is exponential. Let the Time complexity be $T(n)$. $T(n)$ can be written as following recurrence. Here C is a machine dependent constant.

$$\begin{aligned} T(n) &= T(n-1) + T(n-1) + C \\ &= 2T(n-1) + C \end{aligned}$$

The above recurrence has solution as $\Theta(2^n)$. We can solve it by recurrence tree method. The recurrence tree would be a binary tree with height n and every level would be completely full except possibly the last level.



Can the time complexity of fun(n) be reduced?

A simple way to reduce the time complexity is to make one call instead of 2 calls.

```

unsigned fun(unsigned n)
{
    if (n == 0) return 1;
    if (n == 1) return 2;

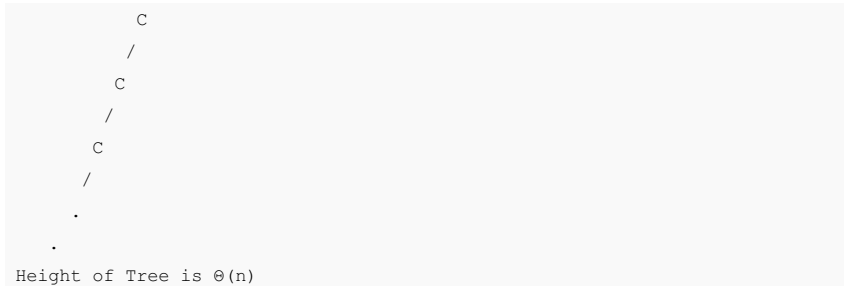
    return 2*fun(n-1);
}

```

Time complexity of the above solution is $\Theta(n)$. Let the Time complexity be $T(n)$. $T(n)$ can be written as following recurrence. Here C is a machine dependent constant.

$$T(n) = T(n-1) + C$$

We can solve it by recurrence tree method. The recurrence tree would be a skewed binary tree (every internal node has only one child) with height n .



The above function can be further optimized using divide and conquer technique to calculate powers.

```

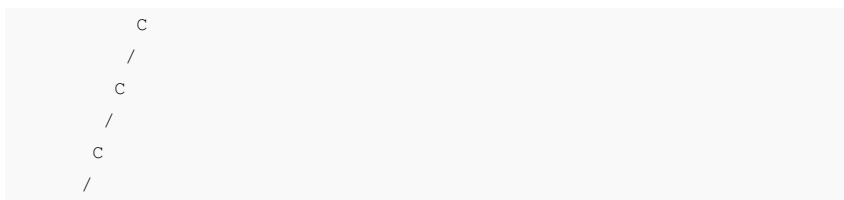
unsigned fun(unsigned n)
{
    if (n == 0) return 1;
    if (n == 1) return 2;
    unsigned x = fun(n/2);
    return (n%2)? 2*x*x: x*x;
}

```

Time complexity of the above solution is $\Theta(\log n)$. Let the Time complexity be $T(n)$. $T(n)$ can be approximately written as following recurrence. Here C is a machine dependent constant.

$$T(n) = T(n/2) + C$$

We can solve it by recurrence tree method. The recurrence tree would be a skewed binary tree (every internal node has only one child) with height $\log(n)$.



```
Height of Tree is  $\Theta(\log n)$ 
```

We can also directly compute $\text{fun}(n)$ using bitwise left shift operator '<<'.

```
unsigned fun(unsigned n)
{
    return 1 << n;
}
```

This article is contributed by **Kartik**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

161. Multiply two polynomials

Given two polynomials represented by two arrays, write a function that multiplies given two polynomials.

Example:

```
Input:  A[] = {5, 0, 10, 6}
        B[] = {1, 2, 4}
Output: prod[] = {5, 10, 30, 26, 52, 24}
```

The first input array represents " $5 + 0x^1 + 10x^2 + 6x^3$ "

The second array represents " $1 + 2x^1 + 4x^2$ "

And Output is " $5 + 10x^1 + 30x^2 + 26x^3 + 52x^4 + 24x^5$ "

We strongly recommend to minimize your browser and try this yourself first.

A simple solution is to one by one consider every term of first polynomial and multiply it with every term of second polynomial. Following is algorithm of this simple method.

```
multiply(A[0..m-1], B[0..n01])
```

- 1) Create a product array `prod[]` of size $m+n-1$.
- 2) Initialize all entries in `prod[]` as 0.
- 3) Travers array `A[]` and do following for every element `A[i]`
...{3.a) Traverse array `B[]` and do following for every element `B[j]`
 `prod[i+j] = prod[i+j] + A[i] * B[j]`
- 4) Return `prod[]`.

The following is C++ implementation of above algorithm.

```
// Simple C++ program to multiply two polynomials
#include <iostream>
```

```
using namespace std;
```

```
// A[] represents coefficients of first polynomial
// B[] represents coefficients of second polynomial
// m and n are sizes of A[] and B[] respectively
int *multiply(int A[], int B[], int m, int n)
{
    int *prod = new int[m+n-1];

    // Initialize the product polynomial
    for (int i = 0; i<m+n-1; i++)
        prod[i] = 0;

    // Multiply two polynomials term by term

    // Take every term of first polynomial
    for (int i=0; i<m; i++)
    {
        // Multiply the current term of first polynomial
        // with every term of second polynomial.
        for (int j=0; j<n; j++)
            prod[i+j] += A[i]*B[j];
    }

    return prod;
}
```

```
// A utility function to print a polynomial
```

```
void printPoly(int poly[], int n)
{
    for (int i=0; i<n; i++)
    {
        cout << poly[i];
        if (i != 0)
            cout << "x^" << i ;
        if (i != n-1)
            cout << " + ";
    }
}
```

```
// Driver program to test above functions
```

```
int main()
{
    // The following array represents polynomial 5 + 10x^2 + 6x^3
    int A[] = {5, 0, 10, 6};

    // The following array represents polynomial 1 + 2x + 4x^2
    int B[] = {1, 2, 4};
    int m = sizeof(A)/sizeof(A[0]);
    int n = sizeof(B)/sizeof(B[0]);

    cout << "First polynomial is \n";
    printPoly(A, m);
    cout << "\nSecond polynomial is \n";
    printPoly(B, n);

    int *prod = multiply(A, B, m, n);

    cout << "\nProduct polynomial is \n";
    printPoly(prod, m+n-1);

    return 0;
}
```

s

Output

```
First polynomial is
5 + 0x^1 + 10x^2 + 6x^3
Second polynomial is
1 + 2x^1 + 4x^2
Product polynomial is
5 + 10x^1 + 30x^2 + 26x^3 + 52x^4 + 24x^5
```

Time complexity of the above solution is $O(mn)$. If size of two polynomials same, then time complexity is $O(n^2)$.

Can we do better?

There are methods to do multiplication faster than $O(n^2)$ time. These methods are mainly based on **divide and conquer**. Following is one simple method that divides the given polynomial (of degree n) into two polynomials one containing lower degree terms (lower than $n/2$) and other containing higher degree terms (higher than or equal to $n/2$)

Let the two given polynomials be A and B.
For simplicity, Let us assume that the given two polynomials are of same degree and have degree in powers of 2, i.e., $n = 2^i$

The polynomial 'A' can be written as $A_0 + A_1 \cdot x^{n/2}$

The polynomial 'B' can be written as $B_0 + B_1 \cdot x^{n/2}$

For example $1 + 10x + 6x^2 - 4x^3 + 5x^4$ can be written as $(1 + 10x) + (6 - 4x + 5x^2) \cdot x^2$

$$\begin{aligned} A * B &= (A_0 + A_1 \cdot x^{n/2}) * (B_0 + B_1 \cdot x^{n/2}) \\ &= A_0 \cdot B_0 + A_0 \cdot B_1 \cdot x^{n/2} + A_1 \cdot B_0 \cdot x^{n/2} + A_1 \cdot B_1 \cdot x^n \\ &= A_0 \cdot B_0 + (A_0 \cdot B_1 + A_1 \cdot B_0) x^{n/2} + A_1 \cdot B_1 \cdot x^n \end{aligned}$$

So the above divide and conquer approach requires 4 multiplications and $O(n)$ time to add all 4 results. Therefore the time complexity is $T(n) = 4T(n/2) + O(n)$. The solution of the recurrence is $O(n^2)$ which is same as the above simple solution.

The idea is to reduce number of multiplications to 3 and make the recurrence as $T(n) = 3T(n/2) + O(n)$

How to reduce number of multiplications?

This requires a little trick similar to **Strassen's Matrix Multiplication**. We do following 3 multiplications.

```

X = (A0 + A1)*(B0 + B1) // First Multiplication
Y = A0B0 // Second
Z = A1B1 // Third

```

The missing middle term in above multiplication equation $A0*B0 + (A0*B1 + A1*B0)x^{n/2} + A1*B1*x^n$ can obtained using below.

$$A0B1 + A1B0 = X - Y - Z$$

So the time taken by this algorithm is $T(n) = 3T(n/2) + O(n)$

The solution of above recurrence is $O(n^{\lg 3})$ which is better than $O(n^2)$.

We will soon be discussing implementation of above approach.

There is a $O(n \log n)$ algorithm also that uses Fast Fourier Transform to multiply two polynomials (Refer [this](#) and [this](#) for details)

Sources:

<http://www.cse.ust.hk/~dekai/271/notes/L03/L03.pdf>

This article is contributed by Harsh. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

162. Job Sequencing Problem | Set 1 (Greedy Algorithm)

Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes single unit of time, so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

Examples:

Input: Four Jobs with following deadlines and profits

JobID	Deadline	Profit
a	4	20
b	1	10
c	1	40
d	1	30

Output: Following is maximum profit sequence of jobs

c, a

Input: Five Jobs with following deadlines and profits

JobID	Deadline	Profit
-------	----------	--------

a	2	100
b	1	19
c	2	27
d	1	25
e	3	15

Output: Following is maximum profit sequence of jobs

c, a, e

We strongly recommend to minimize your browser and try this yourself first.

A **Simple Solution** is to generate all subsets of given set of jobs and check individual subset for feasibility of jobs in that subset. Keep track of maximum profit among all feasible subsets. The time complexity of this solution is exponential.

This is a standard **Greedy Algorithm** problem. Following is algorithm.

```

1) Sort all jobs in decreasing order of profit.
2) Initialize the result sequence as first job in sorted jobs.
3) Do following for remaining n-1 jobs
.....a) If the current job can fit in the current result sequence
           without missing the deadline, add current job to the result.
           Else ignore the current job.
```

The Following is C++ implementation of above algorithm.

```

// Program to find the maximum profit job sequence from a given array
// of jobs with deadlines and profits
#include<iostream>
#include<algorithm>
using namespace std;

// A structure to represent a job
struct Job
{
    char id;        // Job Id
    int dead;       // Deadline of job
    int profit;     // Profit if job is over before or on deadline
};

// This function is used for sorting all jobs according to profit
bool comparison(Job a, Job b)
{
    return (a.profit > b.profit);
}

// Returns minimum number of platforms required
void printJobScheduling(Job arr[], int n)
{
    // Sort all jobs according to decreasing order of profit
    sort(arr, arr+n, comparison);

    int result[n]; // To store result (Sequence of jobs)
    bool slot[n];  // To keep track of free time slots

    // Initialize all slots to be free
```

```

    for (int i=0; i<n; i++)
        slot[i] = false;

    // Iterate through all given jobs
    for (int i=0; i<n; i++)
    {
        // Find a free slot for this job (Note that we start
        // from the last possible slot)
        for (int j=min(n, arr[i].dead)-1; j>=0; j--)
        {
            // Free slot found
            if (slot[j]==false)
            {
                result[j] = i; // Add this job to result
                slot[j] = true; // Make this slot occupied
                break;
            }
        }
    }

    // Print the result
    for (int i=0; i<n; i++)
        if (slot[i])
            cout << arr[result[i]].id << " ";
}

// Driver program to test methods
int main()
{
    Job arr[5] = { {'a', 2, 100}, {'b', 1, 19}, {'c', 2, 27},
                   {'d', 1, 25}, {'e', 3, 15}};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Following is maximum profit sequence of jobs\n";
    printJobScheduling(arr, n);
    return 0;
}

```

Output:

```

Following is maximum profit sequence of jobs
c a e

```

Time Complexity of the above solution is $O(n^2)$. It can be optimized to almost $O(n)$ by using [union-find data structure](#). We will soon be discussing the optimized solution.

Sources:

http://ocw.mit.edu/courses/civil-and-environmental-engineering/1-204-computer-algorithms-in-systems-engineering-spring-2010/lecture-notes/MIT1_204S10_lec10.pdf

This article is contributed by **Shubham**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

163. Birthday Paradox

How many people must be there in a room to make the probability 100% that two people in the room have same birthday?

Answer: 367 (since there are 366 possible birthdays, including February 29).

The above question was simple. Try the below question yourself.

How many people must be there in a room to make the probability 50% that two people in the room have same birthday?

Answer: 23

The number is surprisingly very low. In fact, we need only 70 people to make the probability 99.9 %.

Let us discuss the generalized formula.

What is the probability that two persons among n have same birthday?

Let the probability that two people in a room with n have same birthday be P(same).

P(Same) can be easily evaluated in terms of P(different) where P(different) is the probability that all of them have different birthday.

$$P(\text{same}) = 1 - P(\text{different})$$

P(different) can be written as $1 \times (364/365) \times (363/365) \times (362/365) \times \dots \times (1 - (n-1)/365)$

How did we get the above expression?

Persons from first to last can get birthdays in following order for all birthdays to be distinct:

The first person can have any birthday among 365

The second person should have a birthday which is not same as first person

The third person should have a birthday which is not same as first two persons.

.....

.....

The n'th person should have a birthday which is not same as any of the earlier considered (n-1) persons.

Approximation of above expression

The above expression can be approximated using Taylor's Series.

$$e^x = 1 + x + \frac{x^2}{2!} + \dots$$

provides a first-order approximation for e^x for $x \ll 1$:

$$e^x \approx 1 + x.$$

To apply this approximation to the first expression derived for p(different), set $x = -a / 365$. Thus,

$$e^{-a/365} \approx 1 - \frac{a}{365}.$$

The above expression derived for p(different) can be written as $1 \times (1 - 1/365) \times (1 - 2/365) \times (1 - 3/365) \times \dots \times (1 - (n-1)/365)$

By putting the value of $1 - \frac{a}{365}$ as $e^{-a/365}$, we get following.

$$\begin{aligned} &\approx 1 \times e^{-1/365} \times e^{-2/365} \dots e^{-(n-1)/365} \\ &= 1 \times e^{-(1+2+\dots+(n-1))/365} \\ &= e^{-n(n-1)/2/365} \end{aligned} \tag{1}$$

Therefore,

$$p(\text{same}) = 1 - p(\text{different}) \approx 1 - e^{-n(n-1)/(2 \times 365)}.$$

An even coarser approximation is given by

$$p(\text{same}) \approx 1 - e^{-n^2/(2 \times 365)},$$

By taking Log on both sides, we get the reverse formula.

$$n \approx \sqrt{2 \times 365 \ln \left(\frac{1}{1 - p(\text{same})} \right)}.$$

Using the above approximate formula, we can approximate number of people for a given probability. For example the following C++ function find() returns the smallest n for which the probability is greater than the given p.

C++ Implementation of approximate formula.

The following is C++ program to approximate number of people for a given probability.

```
// C++ program to approximate number of people in Birthday Paradox
// problem
#include <cmath>
#include <iostream>
using namespace std;
```

```
// Returns approximate number of people for a given probability
int find(double p)
{
    return ceil(sqrt(2*365*log(1/(1-p))));
}
```

```
int main()
{
    cout << find(0.70);
}
```

Output:

```
30
```

Source:

http://en.wikipedia.org/wiki/Birthday_problem

Applications:

1) Birthday Paradox is generally discussed with hashing to show importance of collision

handling even for a small set of keys.

2) Birthday Attack

This article is contributed by **Shubham**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

164. How to Implement Forward DNS Look Up Cache?

We have discussed [implementation of Reverse DNS Look Up Cache](#). Forward DNS look up is getting IP address for a given domain name typed in the web browser.

The cache should do the following operations :

1. Add a mapping from URL to IP address
2. Find IP address for a given URL.

There are a few changes from [reverse DNS look up cache](#) that we need to incorporate.

1. Instead of [0-9] and (.) dot we need to take care of [A-Z], [a-z] and (.) dot. As most of the domain name contains only lowercase characters we can assume that there will be [a-z] and (.) 27 children for each trie node.

2. When we type `www.google.in` and `google.in` the browser takes us to the same page. So, we need to add a domain name into trie for the words after `www(.)`. Similarly while searching for a domain name corresponding IP address remove the `www(.)` if the user has provided it.

This is left as an exercise and for simplicity we have taken care of `www.` also.

One solution is to use [Hashing](#). In this post, a [Trie](#) based solution is discussed. One advantage of Trie based solutions is, worst case upper bound is $O(1)$ for Trie, for hashing, the best possible average case time complexity is $O(1)$. Also, with Trie we can implement prefix search (finding all IPs for a common prefix of URLs). The general disadvantage of Trie is large amount of memory requirement.

The idea is to store URLs in Trie nodes and store the corresponding IP address in last or leaf node.

Following is C style implementation in C++.

```
// C based program to implement reverse DNS lookup
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

// There are atmost 27 different chars in a valid URL
// assuming URL consists [a-z] and (.)
#define CHARS 27

// Trie node structure
struct TrieNode {
    char* url;
    struct TrieNode* children[CHARS];
};
```

```

// Maximum length of a valid URL
#define MAX 100

// A utility function to find index of child for a given character 'c'
int getIndex(char c)
{
    return (c == '.') ? 26 : (c - 'a');
}

// A utility function to find character for a given child index.
char getCharFromIndex(int i)
{
    return (i == 26) ? '.' : ('a' + i);
}

// Trie Node.
struct trieNode
{
    bool isLeaf;
    char *ipAdd;
    struct trieNode *child[CHARS];
};

// Function to create a new trie node.
struct trieNode *newTrieNode(void)
{
    struct trieNode *newNode = new trieNode;
    newNode->isLeaf = false;
    newNode->ipAdd = NULL;
    for (int i = 0; i < CHARS; i++)
        newNode->child[i] = NULL;
    return newNode;
}

// This method inserts a URL and corresponding IP address
// in the trie. The last node in Trie contains the ip address.
void insert(struct trieNode *root, char *URL, char *ipAdd)
{
    // Length of the URL
    int len = strlen(URL);
    struct trieNode *pCrawl = root;

    // Traversing over the length of the URL.
    for (int level = 0; level < len; level++)
    {
        // Get index of child node from current character
        // in URL[] Index must be from 0 to 26 where
        // 0 to 25 is used for alphabets and 26 for dot
        int index = getIndex(URL[level]);

        // Create a new child if not exist already
        if (!pCrawl->child[index])
            pCrawl->child[index] = newTrieNode();

        // Move to the child
        pCrawl = pCrawl->child[index];
    }

    //Below needs to be carried out for the last node.
    //Save the corresponding ip address of the URL in the
    //last node of trie.
    pCrawl->isLeaf = true;
    pCrawl->ipAdd = new char[strlen(ipAdd) + 1];

```

```

    pCrawl->ipAdd = new char[strlen(ipAdd) + 1];
    strcpy(pCrawl->ipAdd, ipAdd);
}

// This function returns IP address if given URL is
// present in DNS cache. Else returns NULL
char *searchDNSCache(struct trieNode *root, char *URL)
{
    // Root node of trie.
    struct trieNode *pCrawl = root;
    int len = strlen(URL);

    // Traversal over the length of URL.
    for (int level = 0; level < len; level++)
    {
        int index = getIndex(URL[level]);
        if (!pCrawl->child[index])
            return NULL;
        pCrawl = pCrawl->child[index];
    }

    // If we find the last node for a given ip address,
    // print the ip address.
    if (pCrawl != NULL && pCrawl->isLeaf)
        return pCrawl->ipAdd;

    return NULL;
}

// Driver function.
int main()
{
    char URL[][50] = { "www.samsung.com", "www.samsung.net",
                      "www.google.in"
                    };
    char ipAdd[][MAX] = { "107.108.11.123", "107.109.123.255",
                        "74.125.200.106"
                      };
    int n = sizeof(URL) / sizeof(URL[0]);
    struct trieNode *root = newTrieNode();

    // Inserts all the domain name and their corresponding
    // ip address
    for (int i = 0; i < n; i++)
        insert(root, URL[i], ipAdd[i]);

    // If forward DNS look up succeeds print the url along
    // with the resolved ip address.
    char url[] = "www.samsung.com";
    char *res_ip = searchDNSCache(root, url);
    if (res_ip != NULL)
        printf("Forward DNS look up resolved in cache:\n%s --> %s",
              url, res_ip);
    else
        printf("Forward DNS look up not resolved in cache ");

    return 0;
}

```

Output:

```
Forward DNS look up resolved in cache:
www.samsung.com --> 107.108.11.123
```

This article is contributed by **Kumar Gautam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

165. How to check if an instance of 8 puzzle is solvable?

What is 8 puzzle?

Given a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles in order using the empty space. We can slide four adjacent (left, right, above and below) tiles into the empty space.

1	2	3
4	5	6
7	8	

Goal State

Empty space can be anywhere

How to find if given state is solvable?

Following are two examples, the first example can reach goal state by a series of slides. The second example cannot.

1	8	2
	4	3
7	6	5

Given State

Solvable

We can reach goal state by sliding tiles using blank space.

8	1	2
	4	3
7	6	5

Given State

Not Solvable

We can not reach goal state by sliding tiles using blank space.

Following is simple rule to check if a 8 puzzle is solvable.

It is not possible to solve an instance of 8 puzzle if number of inversions is odd in the input state. In the examples given in above figure, the first example has 10 inversions, therefore solvable. The second example has 11 inversions, therefore unsolvable.

What is inversion?

A pair of tiles form an inversion if the the values on tiles are in reverse order of their appearance in goal state. For example, the following instance of 8 puzzle has two inversions, (8, 6) and (8, 7).

1	2	3
4	—	5
8	6	7

Following is a simple C++ program to check whether a given instance of 8 puzzle is solvable or not. The idea is simple, we count inversions in the given 8 puzzle.

```
// C++ program to check if a given instance of 8 puzzle is solvable or
#include <iostream>
using namespace std;

// A utility function to count inversions in given array 'arr[]'
int getInvCount(int arr[])
{
    int inv_count = 0;
    for (int i = 0; i < 9 - 1; i++)
        for (int j = i+1; j < 9; j++)
            // Value 0 is used for empty space
            if (arr[j] && arr[i] && arr[i] > arr[j])
                inv_count++;
    return inv_count;
}

// This function returns true if given 8 puzzle is solvable.
bool isSolvable(int puzzle[3][3])
{
    // Count inversions in given 8 puzzle
    int invCount = getInvCount((int *)puzzle);

    // return true if inversion count is even.
    return (invCount%2 == 0);
}

/* Driver progra to test above functions */
int main(int argv, int** args)
{
    int puzzle[3][3] = {{1, 8, 2},
                        {0, 4, 3}, // Value 0 is used for empty space
                        {7, 6, 5}};

    isSolvable(puzzle)? cout << "Solvable":
                       cout << "Not Solvable";

    return 0;
}
```

Output:

Solvable

Note that the above implementation uses simple algorithm for inversion count. It is done this way for simplicity. The code can be optimized to $O(n \log n)$ using the [merge sort based algorithm for inversion count](#).

How does this work?

The idea is based on the fact the parity of inversions remains same after a set of moves, i.e., if the inversion count is odd in initial stage, then it remain odd after any sequence of moves and if the inversion count is even, then it remains even after any

sequence of moves. In the goal state, there are 0 inversions. So we can reach goal state only from a state which has even inversion count.

How parity of inversion count is invariant?

When we slide a tile, we either make a row move (moving a left or right tile into the blank space), or make a column move (moving a up or down tile to the blank space).

a) A row move doesn't change the inversion count. See following example

1	2	3	Row Move	1	2	3
4	_	5	----->	_	4	5
8	6	7		8	6	7
Inversion count remains 2 after the move						
1	2	3	Row Move	1	2	3
4	_	5	----->	4	5	_
8	6	7		8	6	7
Inversion count remains 2 after the move						

b) A column move does one of the following three.

.....(i) Increases inversion count by 2. See following example.

1	2	3	Column Move	1	_	3
4	_	5	----->	4	2	5
8	6	7		8	6	7
Inversion count increases by 2 (changes from 2 to 4)						

.....(ii) Decreases inversion count by 2

1	3	4	Column Move	1	3	4
5	_	6	----->	5	2	6
7	2	8		7	_	8
Inversion count decreases by 2 (changes from 5 to 3)						

.....(iii) Keeps the inversion count same.

1	2	3	Column Move	1	2	3
4	_	5	----->	4	6	5
7	6	8		7	_	8
Inversion count remains 1 after the move						

So if a move either increases/decreases inversion count by 2, or keeps the inversion count same, then it is not possible to change parity of a state by any sequence of row/column moves.

Exercise: How to check if a given instance of 15 puzzle is solvable or not. In a 15 puzzle, we have 4x4 board where 15 tiles have a number and one empty space. Note that the above simple rules of inversion count don't directly work for 15 puzzle, the rules

need to be modified for 15 puzzle.

This article is contributed by Ishan. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

166. Find length of period in decimal value of $1/n$

Given a positive integer n , find the period in decimal value of $1/n$. Period in decimal value is number of digits (somewhere after decimal point) that keep repeating.

Examples:

```
Input: n = 3
Output: 1
The value of 1/3 is 0.333333...

Input: n = 7
Output: 6
The value of 1/7 is 0.142857142857142857....

Input: n = 210
Output: 6
The value of 1/210 is 0.0047619047619048....
```

Let us first discuss a simpler problem of finding individual digits in value of $1/n$.

How to find individual digits in value of $1/n$?

Let us take an example to understand the process. For example for $n = 7$. The first digit can be obtained by doing $10/7$. Second digit can be obtained by $30/7$ (3 is remainder in previous division). Third digit can be obtained by $20/7$ (2 is remainder of previous division). So the idea is to get the first digit, then keep taking value of $(\text{remainder} * 10)/n$ as next digit and keep updating remainder as $(\text{remainder} * 10) \% 10$. The complete program is discussed [here](#).

How to find the period?

The period of $1/n$ is equal to the period in sequence of remainders used in the above process. This can be easily proved from the fact that digits are directly derived from remainders.

One interesting fact about sequence of remainders is, all terms in period of this remainder sequence are distinct. The reason for this is simple, if a remainder repeats, then it's beginning of new period.

The following is C++ implementation of above idea.

```

// C++ program to find length of period of 1/n
#include <iostream>
#include <map>
using namespace std;

// Function to find length of period in 1/n
int getPeriod(int n)
{
    // Create a map to store mapping from remainder
    // its position
    map<int, int> mymap;
    map<int, int>::iterator it;

    // Initialize remainder and position of remainder
    int rem = 1, i = 1;

    // Keep finding remainders till a repeating remainder
    // is found
    while (true)
    {
        // Find next remainder
        rem = (10*rem) % n;

        // If remainder exists in mymap, then the difference
        // between current and previous position is length of
        // period
        it = mymap.find(rem);
        if (it != mymap.end())
            return (i - it->second);

        // If doesn't exist, then add 'i' to mymap
        mymap[rem] = i;
        i++;
    }

    // This code should never be reached
    return INT_MAX;
}

// Driver program to test above function
int main()
{
    cout << getPeriod(3) << endl;
    cout << getPeriod(7) << endl;
    return 0;
}

```

Output:

```

1
6

```

We can avoid the use of map or hash using the following fact. For a number n , there can be at most n distinct remainders. Also, the period may not begin from the first remainder as some initial remainders may be non-repetitive (not part of any period). So to make sure that a remainder from a period is picked, start from the $(n+1)$ th remainder and keep looking for its next occurrence. The distance between $(n+1)$ th remainder and its next occurrence is the length of the period.

```
// C++ program to find length of period of 1/n without
// using map or hash
#include <iostream>
using namespace std;
```

```
// Function to find length of period in 1/n
int getPeriod(int n)
{
    // Find the (n+1)th remainder after decimal point
    // in value of 1/n
    int rem = 1; // Initialize remainder
    for (int i = 1; i <= n+1; i++)
        rem = (10*rem) % n;

    // Store (n+1)th remainder
    int d = rem;

    // Count the number of remainders before next
    // occurrence of (n+1)'th remainder 'd'
    int count = 0;
    do {
        rem = (10*rem) % n;
        count++;
    } while(rem != d);

    return count;
}
```

```
// Driver program to test above function
int main()
{
    cout << getPeriod(3) << endl;
    cout << getPeriod(7) << endl;
    return 0;
}
```

Output:

```
1
6
```

Reference:

Algorithms And Programming: Problems And Solutions by Alexander Shen

This article is contributed by **Sachin**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

167. Greedy Algorithm for Egyptian Fraction

Every positive fraction can be represented as sum of unique unit fractions. A fraction is unit fraction if numerator is 1 and denominator is a positive integer, for example $\frac{1}{3}$ is a unit fraction. Such a representation is called Egyptian Fraction as it was used by ancient

Egyptians.

Following are few examples:

Egyptian Fraction Representation of $2/3$ is $1/2 + 1/6$

Egyptian Fraction Representation of $6/14$ is $1/3 + 1/11 + 1/231$

Egyptian Fraction Representation of $12/13$ is $1/2 + 1/3 + 1/12 + 1/156$

We can generate Egyptian Fractions using **Greedy Algorithm**. For a given number of the form ' nr/dr ' where $dr > nr$, first find the greatest possible unit fraction, then recur for the remaining part. For example, consider $6/14$, we first find ceiling of $14/6$, i.e., 3. So the first unit fraction becomes $1/3$, then recur for $(6/14 - 1/3)$ i.e., $4/42$.

Below is C++ implementation of above idea.

```

// C++ program to print a fraction in Egyptian Form using Greedy
// Algorithm
#include <iostream>
using namespace std;

void printEgyptian(int nr, int dr)
{
    // If either numerator or denominator is 0
    if (dr == 0 || nr == 0)
        return;

    // If numerator divides denominator, then simple division
    // makes the fraction in 1/n form
    if (dr%nr == 0)
    {
        cout << "1/" << dr/nr;
        return;
    }

    // If denominator divides numerator, then the given number
    // is not fraction
    if (nr%dr == 0)
    {
        cout << nr/dr;
        return;
    }

    // If numerator is more than denominator
    if (nr > dr)
    {
        cout << nr/dr << " + ";
        printEgyptian(nr%dr, dr);
        return;
    }

    // We reach here dr > nr and dr%nr is non-zero
    // Find ceiling of dr/nr and print it as first
    // fraction
    int n = dr/nr + 1;
    cout << "1/" << n << " + ";

    // Recur for remaining part
    printEgyptian(nr*n-dr, dr*n);
}

// Driver Program
int main()
{
    int nr = 6, dr = 14;
    cout << "Egyptian Fraction Representation of "
        << nr << "/" << dr << " is\n ";
    printEgyptian(nr, dr);
    return 0;
}

```

Output:

```

Egyptian Fraction Representation of 6/14 is
1/3 + 1/11 + 1/231

```

The Greedy algorithm works because a fraction is always reduced to a form where

denominator is greater than numerator and numerator doesn't divide denominator. For such reduced forms, the highlighted recursive call is made for reduced numerator. So the recursive calls keep on reducing the numerator till it reaches 1.

References:

<http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fractions/egyptian.html>

This article is contributed by **Shubham**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

168. Count number of ways to reach a given score in a game

Consider a game where a player can score 3 or 5 or 10 points in a move. Given a total score n , find number of ways to reach the given score.

Examples:

```
Input: n = 20
Output: 4
There are following 4 ways to reach 20
(10, 10)
(5, 5, 10)
(5, 5, 5, 5)
(3, 3, 3, 3, 3, 5)

Input: n = 13
Output: 2
There are following 2 ways to reach 13
(3, 5, 5)
(3, 10)
```

We strongly recommend you to minimize the browser and try this yourself first.

This problem is a variation of [coin change problem](#) and can be solved in $O(n)$ time and $O(n)$ auxiliary space.

The idea is to create a table of size $n+1$ to store counts of all scores from 0 to n . For every possible move (3, 5 and 10), increment values in table.

```
// A C program to count number of possible ways to a given score
// can be reached in a game where a move can earn 3 or 5 or 10
#include <stdio.h>
```

```
// Returns number of ways to reach score n
int count(int n)
{
    // table[i] will store count of solutions for
    // value i.
    int table[n+1], i;

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

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

    // One by one consider given 3 moves and update the table[]
    // values after the index greater than or equal to the
    // value of the picked move
    for (i=3; i<=n; i++)
        table[i] += table[i-3];
    for (i=5; i<=n; i++)
        table[i] += table[i-5];
    for (i=10; i<=n; i++)
        table[i] += table[i-10];

    return table[n];
}
```

```
// Driver program
int main(void)
{
    int n = 20;
    printf("Count for %d is %d\n", n, count(n));

    n = 13;
    printf("Count for %d is %d", n, count(n));
    return 0;
}
```

Output:

```
Count for 20 is 4
Count for 13 is 2
```

Exercise: How to count score when (10, 5, 5), (5, 5, 10) and (5, 10, 5) are considered as different sequences of moves. Similarly, (5, 3, 3), (3, 5, 3) and (3, 3, 5) are considered different.

This article is contributed by **Rajeev Arora**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

169. Visa Interview Experience | Set 6 (On-Campus)

Written Round:

75 minutes Test on Mettle.

Assessment Composition:

1. Coding Skills: 2 Questions (One ques is from DP(easy one like 0-1 knapsack) and other is Activity Selection Problem).
2. Programming: 10 Questions
3. Machine Learning Hadoop MCQ: 6 Questions
4. Networking MCQ: 6 Questions
5. Operating System: 6 Questions
6. Infrastructure: 6 Questions
7. Application Security: 6 Questions

(In coding round everyone had different questions but they mainly cover DP and simple array question).

First Round (35 min):

- First he told me about himself, then asked me “Tell me something interesting about yourself?”
- Then he asked me you have experience in android or iOS app development. Because i don't have any knowledge of about it i simply said no.
- Then he asked me what is your area of interest. (my area of interest : computer networking , ds and algo).
- Then asked me completely describe Tcp/Ip model and OSI layer model thoroughly and difference between them.
- OOPS concepts all.
- PKI (public key infrastructure) and DDOS attack.
- Loadbalancing of servers.
- Complete working of Router .
- Write a code to delete node from singly linked list.

Second Round (1 Hour 10 min):

- First he told me about himself, then asked me about my project.
- Project was the main deciding factor that you will move to next round or not.
- Computer Network Topology, SDLC models, statistics (mean, median, mode, standard deviation formulae).
- What is api give examples also.
- He has given me a real time scenario and asked to develop strategy for making a software.

You have been given unstructured data which is coming from facebook and twitter.

Analyse these data according to “TCS stocks in market and people's view on this”.

Make a strategy to tell that your software will decide that any point of time you should “BUY” or “SELL” the stock.

(A lot of discussion had been done there interviewer just want to know how you are approaching to solution.)

– Testing (white box / black box testing mainly).

he asked one question based on this that if there is memory leakage in your program which testing you will choose and why. (ans : white box testing).

– Finally he asked “You have any question from me”.

Third Round (15 min):

– This round was tech and hr mixed.

– He asked me for which profile you want to go (Networking or Programming) .

– TCP congestion how to remove it.

– A simple 9 ball puzzle , one ball is light in weight . In how many minimum balance you can find out this light weighted ball.

– Finally he asked “You have any question from me”.

Guys mainly focus on networking, information security and project and be confident. Thanks to geeksforgeeks for helping a lot in preparing for placement.

If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

170. Write an iterative $O(\log y)$ function for $\text{pow}(x, y)$

Given an integer x and a positive number y , write a function that computes x^y under following conditions.

a) Time complexity of the function should be $O(\log y)$

b) Extra Space is $O(1)$

Examples:

```
Input: x = 3, y = 5
```

```
Output: 243
```

```
Input: x = 2, y = 5
```

```
Output: 32
```

We strongly recommend to minimize your browser and try this yourself first.

We have discussed [recursive \$O\(\log y\)\$ solution for power](#). The recursive solutions are generally not preferred as they require space on call stack and they involve function call overhead.

Following is C function to compute x^y .

```
#include <stdio.h>
```

```
/* Iterative Function to calculate x raised to the power y in O(logy) */
int power(int x, unsigned int y)
{
    // Initialize result
    int res = 1;

    while (y > 0)
    {
        // If y is even, simply do x square
        if (y%2 == 0)
        {
            y = y/2;
            x = x*x;
        }

        // Else multiply x with result. Note that this else
        // is always executed in the end when y becomes 1
        else
        {
            y--;
            res = res*x;
        }
    }
    return res;
}
```

```
// Driver program to test above functions
int main()
{
    int x = 3;
    unsigned int y = 5;

    printf("Power is %d", power(x, y));

    return 0;
}
```

Output:

```
Power is 243
```

This article is contributed by **Udit Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

171. How to print maximum number of A's using given four keys

This is a famous interview question asked in [Google](#), [Paytm](#) and many other company interviews.

Below is the problem statement.

Imagine you have a special keyboard with the following keys:

Key 1: Prints 'A' on screen

Key 2: (Ctrl-A): Select screen

Key 3: (Ctrl-C): Copy selection to buffer

Key 4: (Ctrl-V): Print buffer on screen appending it
after what has already been printed.

If you can only press the keyboard for N times (with the above four keys), write a program to produce maximum numbers of A's. That is to say, the input parameter is N (No. of keys that you can press), the output is M (No. of As that you can produce).

Examples:

Input: N = 3

Output: 3

We can at most get 3 A's on screen by pressing
following key sequence.

A, A, A

Input: N = 7

Output: 9

We can at most get 9 A's on screen by pressing
following key sequence.

A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V

Input: N = 11

Output: 27

We can at most get 27 A's on screen by pressing
following key sequence.

A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V, Ctrl A,
Ctrl C, Ctrl V, Ctrl V

We strongly recommend to minimize your browser and try this yourself first.

Below are few important points to note.

- a) For $N < 7$, the output is N itself.
- b) Ctrl V can be used multiple times to print current buffer (See last two examples above).

The idea is to compute the optimal string length for N keystrokes by using a simple insight. The sequence of N keystrokes which produces an optimal string length will end with a suffix of Ctrl-A, a Ctrl-C, followed by only Ctrl-V's (For $N > 6$).

The task is to find out the break=point after which we get the above suffix of keystrokes. Definition of a breakpoint is that instance after which we need to only press Ctrl-A, Ctrl-C

once and the only Ctrl-V's afterwards to generate the optimal length. If we loop from N-3 to 1 and choose each of these values for the break-point, and compute that optimal string they would produce. Once the loop ends, we will have the maximum of the optimal lengths for various breakpoints, thereby giving us the optimal length for N keystrokes.

Below is C implementation based on above idea.

```
/* A recursive C program to print maximum number of A's using
   following four keys */
#include<stdio.h>

// A recursive function that returns the optimal length string
// for N keystrokes
int findoptimal(int N)
{
    // The optimal string length is N when N is smaller than 7
    if (N <= 6)
        return N;

    // Initialize result
    int max = 0;

    // TRY ALL POSSIBLE BREAK-POINTS
    // For any keystroke N, we need to loop from N-3 keystrokes
    // back to 1 keystroke to find a breakpoint 'b' after which we
    // will have Ctrl-A, Ctrl-C and then only Ctrl-V all the way.
    int b;
    for (b=N-3; b>=1; b--)
    {
        // If the breakpoint is s at b'th keystroke then
        // the optimal string would have length
        // (n-b-1)*screen[b-1];
        int curr = (N-b-1)*findoptimal(b);
        if (curr > max)
            max = curr;
    }
    return max;
}

// Driver program
int main()
{
    int N;

    // for the rest of the array we will rely on the previous
    // entries to compute new ones
    for (N=1; N<=20; N++)
        printf("Maximum Number of A's with %d keystrokes is %d\n",
            N, findoptimal(N));
}
```

Output:

```
Maximum Number of A's with 1 keystrokes is 1
Maximum Number of A's with 2 keystrokes is 2
Maximum Number of A's with 3 keystrokes is 3
Maximum Number of A's with 4 keystrokes is 4
Maximum Number of A's with 5 keystrokes is 5
```

```
Maximum Number of A's with 6 keystrokes is 6
Maximum Number of A's with 7 keystrokes is 9
Maximum Number of A's with 8 keystrokes is 12
Maximum Number of A's with 9 keystrokes is 16
Maximum Number of A's with 10 keystrokes is 20
Maximum Number of A's with 11 keystrokes is 27
Maximum Number of A's with 12 keystrokes is 36
Maximum Number of A's with 13 keystrokes is 48
Maximum Number of A's with 14 keystrokes is 64
Maximum Number of A's with 15 keystrokes is 81
Maximum Number of A's with 16 keystrokes is 108
Maximum Number of A's with 17 keystrokes is 144
Maximum Number of A's with 18 keystrokes is 192
Maximum Number of A's with 19 keystrokes is 256
Maximum Number of A's with 20 keystrokes is 324
```

The above function computes the same subproblems again and again. Recomputations of same subproblems can be avoided by storing the solutions to subproblems and solving problems in bottom up manner.

Below is Dynamic Programming based C implementation where an auxiliary array `screen[N]` is used to store result of subproblems.

```

/* A Dynamic Programming based C program to find maximum number of A's
that can be printed using four keys */
#include<stdio.h>

// this function returns the optimal length string for N keystrokes
int findoptimal(int N)
{
    // The optimal string length is N when N is smaller than 7
    if (N <= 6)
        return N;

    // An array to store result of subproblems
    int screen[N];

    int b; // To pick a breakpoint

    // Initializing the optimal lengths array for upto 6 input
    // strokes.
    int n;
    for (n=1; n<=6; n++)
        screen[n-1] = n;

    // Solve all subproblems in bottom manner
    for (n=7; n<=N; n++)
    {
        // Initialize length of optimal string for n keystrokes
        screen[n-1] = 0;

        // For any keystroke n, we need to loop from n-3 keystrokes
        // back to 1 keystroke to find a breakpoint 'b' after which we
        // will have ctrl-a, ctrl-c and then only ctrl-v all the way.
        for (b=n-3; b>=1; b--)
        {
            // if the breakpoint is at b'th keystroke then
            // the optimal string would have length
            // (n-b-1)*screen[b-1];
            int curr = (n-b-1)*screen[b-1];
            if (curr > screen[n-1])
                screen[n-1] = curr;
        }
    }

    return screen[N-1];
}

// Driver program
int main()
{
    int N;

    // for the rest of the array we will rely on the previous
    // entries to compute new ones
    for (N=1; N<=20; N++)
        printf("Maximum Number of A's with %d keystrokes is %d\n",
            N, findoptimal(N));
}

```

Output:

```
Maximum Number of A's with 1 keystrokes is 1
```

```
Maximum Number of A's with 2 keystrokes is 2
Maximum Number of A's with 3 keystrokes is 3
Maximum Number of A's with 4 keystrokes is 4
Maximum Number of A's with 5 keystrokes is 5
Maximum Number of A's with 6 keystrokes is 6
Maximum Number of A's with 7 keystrokes is 9
Maximum Number of A's with 8 keystrokes is 12
Maximum Number of A's with 9 keystrokes is 16
Maximum Number of A's with 10 keystrokes is 20
Maximum Number of A's with 11 keystrokes is 27
Maximum Number of A's with 12 keystrokes is 36
Maximum Number of A's with 13 keystrokes is 48
Maximum Number of A's with 14 keystrokes is 64
Maximum Number of A's with 15 keystrokes is 81
Maximum Number of A's with 16 keystrokes is 108
Maximum Number of A's with 17 keystrokes is 144
Maximum Number of A's with 18 keystrokes is 192
Maximum Number of A's with 19 keystrokes is 256
Maximum Number of A's with 20 keystrokes is 324
```

Thanks to **Gaurav Saxena** for providing the above approach to solve this problem.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

This is a famous interview question asked in **Google**, **Paytm** and many other company interviews.

Below is the problem statement.

Imagine you have a special keyboard with the following keys:

Key 1: Prints 'A' on screen

Key 2: (Ctrl-A): Select screen

Key 3: (Ctrl-C): Copy selection to buffer

Key 4: (Ctrl-V): Print buffer on screen appending it
after what has already been printed.

If you can only press the keyboard for N times (with the above four keys), write a program to produce maximum numbers of A's. That is to say, the input parameter is N (No. of keys that you can press), the output is M (No. of As that you can produce).

Examples:

Input: N = 3

Output: 3

We can at most get 3 A's on screen by pressing following key sequence.

A, A, A

Input: N = 7

Output: 9

We can at most get 9 A's on screen by pressing following key sequence.

A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V

Input: N = 11

Output: 27

We can at most get 27 A's on screen by pressing following key sequence.

A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V, Ctrl A, Ctrl C, Ctrl V, Ctrl V

We strongly recommend to minimize your browser and try this yourself first.

Below are few important points to note.

- a) For $N < 7$, the output is N itself.
- b) Ctrl V can be used multiple times to print current buffer (See last two examples above).

The idea is to compute the optimal string length for N keystrokes by using a simple insight. The sequence of N keystrokes which produces an optimal string length will end with a suffix of Ctrl-A, a Ctrl-C, followed by only Ctrl-V's (For $N > 6$).

The task is to find out the break=point after which we get the above suffix of keystrokes. Definition of a breakpoint is that instance after which we need to only press Ctrl-A, Ctrl-C once and the only Ctrl-V's afterwards to generate the optimal length. If we loop from N-3 to 1 and choose each of these values for the break-point, and compute that optimal string they would produce. Once the loop ends, we will have the maximum of the optimal lengths for various breakpoints, thereby giving us the optimal length for N keystrokes.

Below is C implementation based on above idea.


```

/* A recursive C program to print maximum number of A's using
   following four keys */
#include<stdio.h>

// A recursive function that returns the optimal length string
// for N keystrokes
int findoptimal(int N)
{
    // The optimal string length is N when N is smaller than 7
    if (N <= 6)
        return N;

    // Initialize result
    int max = 0;

    // TRY ALL POSSIBLE BREAK-POINTS
    // For any keystroke N, we need to loop from N-3 keystrokes
    // back to 1 keystroke to find a breakpoint 'b' after which we
    // will have Ctrl-A, Ctrl-C and then only Ctrl-V all the way.
    int b;
    for (b=N-3; b>=1; b--)
    {
        // If the breakpoint is s at b'th keystroke then
        // the optimal string would have length
        // (n-b-1)*screen[b-1];
        int curr = (N-b-1)*findoptimal(b);
        if (curr > max)
            max = curr;
    }
    return max;
}

// Driver program
int main()
{
    int N;

    // for the rest of the array we will rely on the previous
    // entries to compute new ones
    for (N=1; N<=20; N++)
        printf("Maximum Number of A's with %d keystrokes is %d\n",
            N, findoptimal(N));
}

```

Output:

```

Maximum Number of A's with 1 keystrokes is 1
Maximum Number of A's with 2 keystrokes is 2
Maximum Number of A's with 3 keystrokes is 3
Maximum Number of A's with 4 keystrokes is 4
Maximum Number of A's with 5 keystrokes is 5
Maximum Number of A's with 6 keystrokes is 6
Maximum Number of A's with 7 keystrokes is 9
Maximum Number of A's with 8 keystrokes is 12
Maximum Number of A's with 9 keystrokes is 16
Maximum Number of A's with 10 keystrokes is 20
Maximum Number of A's with 11 keystrokes is 27

```

```
Maximum Number of A's with 12 keystrokes is 36
Maximum Number of A's with 13 keystrokes is 48
Maximum Number of A's with 14 keystrokes is 64
Maximum Number of A's with 15 keystrokes is 81
Maximum Number of A's with 16 keystrokes is 108
Maximum Number of A's with 17 keystrokes is 144
Maximum Number of A's with 18 keystrokes is 192
Maximum Number of A's with 19 keystrokes is 256
Maximum Number of A's with 20 keystrokes is 324
```

The above function computes the same subproblems again and again. Recomputations of same subproblems can be avoided by storing the solutions to subproblems and solving problems in bottom up manner.

Below is Dynamic Programming based C implementation where an auxiliary array `screen[N]` is used to store result of subproblems.

```

/* A Dynamic Programming based C program to find maximum number of A's
that can be printed using four keys */
#include<stdio.h>

// this function returns the optimal length string for N keystrokes
int findoptimal(int N)
{
    // The optimal string length is N when N is smaller than 7
    if (N <= 6)
        return N;

    // An array to store result of subproblems
    int screen[N];

    int b; // To pick a breakpoint

    // Initializing the optimal lengths array for upto 6 input
    // strokes.
    int n;
    for (n=1; n<=6; n++)
        screen[n-1] = n;

    // Solve all subproblems in bottom manner
    for (n=7; n<=N; n++)
    {
        // Initialize length of optimal string for n keystrokes
        screen[n-1] = 0;

        // For any keystroke n, we need to loop from n-3 keystrokes
        // back to 1 keystroke to find a breakpoint 'b' after which we
        // will have ctrl-a, ctrl-c and then only ctrl-v all the way.
        for (b=n-3; b>=1; b--)
        {
            // if the breakpoint is at b'th keystroke then
            // the optimal string would have length
            // (n-b-1)*screen[b-1];
            int curr = (n-b-1)*screen[b-1];
            if (curr > screen[n-1])
                screen[n-1] = curr;
        }
    }

    return screen[N-1];
}

// Driver program
int main()
{
    int N;

    // for the rest of the array we will rely on the previous
    // entries to compute new ones
    for (N=1; N<=20; N++)
        printf("Maximum Number of A's with %d keystrokes is %d\n",
            N, findoptimal(N));
}

```

Output:

```
Maximum Number of A's with 1 keystrokes is 1
```

```
Maximum Number of A's with 2 keystrokes is 2
Maximum Number of A's with 3 keystrokes is 3
Maximum Number of A's with 4 keystrokes is 4
Maximum Number of A's with 5 keystrokes is 5
Maximum Number of A's with 6 keystrokes is 6
Maximum Number of A's with 7 keystrokes is 9
Maximum Number of A's with 8 keystrokes is 12
Maximum Number of A's with 9 keystrokes is 16
Maximum Number of A's with 10 keystrokes is 20
Maximum Number of A's with 11 keystrokes is 27
Maximum Number of A's with 12 keystrokes is 36
Maximum Number of A's with 13 keystrokes is 48
Maximum Number of A's with 14 keystrokes is 64
Maximum Number of A's with 15 keystrokes is 81
Maximum Number of A's with 16 keystrokes is 108
Maximum Number of A's with 17 keystrokes is 144
Maximum Number of A's with 18 keystrokes is 192
Maximum Number of A's with 19 keystrokes is 256
Maximum Number of A's with 20 keystrokes is 324
```

Thanks to **Gaurav Saxena** for providing the above approach to solve this problem.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

173. Set Cover Problem | Set 1 (Greedy Approximate Algorithm)

Given a universe U of n elements, a collection of subsets of U say $S = \{S_1, S_2, \dots, S_m\}$ where every subset S_i has an associated cost. Find a minimum cost subcollection of S that covers all elements of U .

Example:

$U = \{1, 2, 3, 4, 5\}$

$S = \{S_1, S_2, S_3\}$

$S_1 = \{4, 1, 3\}, \quad \text{Cost}(S_1) = 5$

$S_2 = \{2, 5\}, \quad \text{Cost}(S_2) = 10$

$S_3 = \{1, 4, 3, 2\}, \quad \text{Cost}(S_3) = 3$

Output: Minimum cost of set cover is 13 and

```
set cover is {S2, S3}
```

There are two possible set covers $\{S_1, S_2\}$ with cost 15 and $\{S_2, S_3\}$ with cost 13.

Why is it useful?

It was one of Karp's NP-complete problems, shown to be so in 1972. Other applications: edge covering, vertex cover

Interesting example: IBM finds computer viruses (wikipedia)

Elements- 5000 known viruses

Sets- 9000 substrings of 20 or more consecutive bytes from viruses, not found in 'good' code.

A set cover of 180 was found. It suffices to search for these 180 substrings to verify the existence of known computer viruses.

Another example: Consider General Motors needs to buy a certain amount of varied supplies and there are suppliers that offer various deals for different combinations of materials (Supplier A: 2 tons of steel + 500 tiles for \$x; Supplier B: 1 ton of steel + 2000 tiles for \$y; etc.). You could use set covering to find the best way to get all the materials while minimizing cost

Source: <http://math.mit.edu/~goemans/18434S06/setcover-tamara.pdf>

Set Cover is NP-Hard:

There is no polynomial time solution available for this problem as the problem is a known NP-Hard problem. There is a polynomial time Greedy approximate algorithm, the greedy algorithm provides a Logn approximate algorithm.

2-Approximate Greedy Algorithm:

Let U be the universe of elements, $\{S_1, S_2, \dots, S_m\}$ be collection of subsets of U and $\text{Cost}(S_1), \text{Cost}(S_2), \dots, \text{Cost}(S_m)$ be costs of subsets.

- 1) Let I represents set of elements included so far. Initialize $I = \{\}$
- 2) Do following while I is not same as U .
 - a) Find the set S_i in $\{S_1, S_2, \dots, S_m\}$ whose cost effectiveness is smallest, i.e., the ratio of cost $\text{Cost}(S_i)$ and number of newly added elements is minimum.
Basically we pick the set for which following value is minimum.
$$\frac{\text{Cost}(S_i)}{|S_i - I|}$$
 - b) Add elements of above picked S_i to I , i.e., $I = I \cup S_i$

Example:

Let us consider the above example to understand Greedy Algorithm.

First Iteration:

$I = \{\}$

The per new element cost for $S_1 = \text{Cost}(S_1)/|S_1 - I| = 5/3$

The per new element cost for $S_2 = \text{Cost}(S_2)/|S_2 - I| = 10/2$

The per new element cost for $S_3 = \text{Cost}(S_3)/|S_3 - I| = 3/4$

Since S_3 has minimum value S_3 is added, I becomes $\{1,4,3,2\}$.

Second Iteration:

$I = \{1,4,3,2\}$

The per new element cost for $S_1 = \text{Cost}(S_1)/|S_1 - I| = 5/0$

Note that S_1 doesn't add any new element to I .

The per new element cost for $S_2 = \text{Cost}(S_2)/|S_2 - I| = 10/1$

Note that S_2 adds only 5 to I .

The greedy algorithm provides the optimal solution for above example, but it may not provide optimal solution all the time. Consider the following example.

$S_1 = \{1, 2\}$

$S_2 = \{2, 3, 4, 5\}$

$S_3 = \{6, 7, 8, 9, 10, 11, 12, 13\}$

$S_4 = \{1, 3, 5, 7, 9, 11, 13\}$

$S_5 = \{2, 4, 6, 8, 10, 12, 13\}$

Let the cost of every set be same.

The greedy algorithm produces result as $\{S_3, S_2, S_1\}$

The optimal solution is $\{S_4, S_5\}$

Proof that the above greedy algorithm is Logn approximate.

Let OPT be the cost of optimal solution. Say $(k-1)$ elements are covered before an iteration of above greedy algorithm. The cost of the k 'th element $\leq OPT / (n-k+1)$ (Note that cost of an element is evaluated by cost of its set divided by number of elements added by its set). How did we get this result?

Since k 'th element is not covered yet, there is a S_i that has not been covered before the current step of greedy algorithm and it is there in OPT . Since greedy algorithm picks the most cost effective S_i , per element cost in the picked set in Greedy must be smaller than OPT divided by remaining elements. Therefore cost of k 'th element $\leq OPT/|U-I|$ (Note that $U-I$ is set of not yet covered elements in Greedy Algorithm). The value of $|U-I|$ is $n - (k-1)$ which is $n-k+1$.

Cost of Greedy Algorithm = Sum of costs of n elements

```

[putting  $k = 1, 2 \dots n$  in above formula]
 $\leq (OPT/1 + OPT/2 + \dots + OPT/n)$ 
 $\leq OPT(1 + 1/2 + \dots + 1/n)$ 
[Since  $1 + 1/2 + \dots + 1/n \approx \log n$ ]
 $\leq OPT * \log n$ 

```

Source:

<http://math.mit.edu/~goemans/18434S06/setcover-tamara.pdf>

This article is contributed by **Harshit**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

174. Build Lowest Number by Removing n digits from a given number

Given a string 'str' of digits and an integer 'n', build the lowest possible number by removing 'n' digits from the string and not changing the order of input digits.

Examples:

```
Input: str = "4325043", n = 3
```

```
Output: "2043"
```

```
Input: str = "765028321", n = 5
```

```
Output: "0221"
```

```
Input: str = "121198", n = 2
```

```
Output: "1118"
```

We strongly recommend to minimize your browser and try this yourself first.

The idea is based on the fact that a character among first (n+1) characters must be there in resultant number. So we pick the smallest of first (n+1) digits and put it in result, and recur for remaining characters. Below is complete algorithm.

```
Initialize result as empty string
```

```
res = ""
```

```
buildLowestNumber(str, n, res)
```

```
1) If  $n == 0$ , then there is nothing to remove.
```

```
Append the whole 'str' to 'res' and return
```

```
2) Let 'len' be length of 'str'. If 'len' is smaller or equal  
to n, then everything can be removed
```

```
Append nothing to 'res' and return
```

```
3) Find the smallest character among first (n+1) characters  
   of 'str'. Let the index of smallest character be minIndex.  
   Append 'str[minIndex]' to 'res' and recur for substring after  
   minIndex and for n = n-minIndex  
  
   buildLowestNumber(str[minIndex+1..len-1], n-minIndex).
```

Below is C++ implementation of above algorithm.


```

// C++ program to build the smallest number by removing n digits from
// a given number
#include<iostream>
using namespace std;

// A recursive function that removes 'n' characters from 'str'
// to store the smallest possible number in 'res'
void buildLowestNumberRec(string str, int n, string &res)
{
    // If there are 0 characters to remove from str,
    // append everything to result
    if (n == 0)
    {
        res.append(str);
        return;
    }

    int len = str.length();

    // If there are more characters to remove than string
    // length, then append nothing to result
    if (len <= n)
        return;

    // Find the smallest character among first (n+1) characters
    // of str.
    int minIndex = 0;
    for (int i = 1; i <= n; i++)
        if (str[i] < str[minIndex])
            minIndex = i;

    // Append the smallest character to result
    res.push_back(str[minIndex]);

    // substring starting from minIndex+1 to str.length() - 1.
    string new_str = str.substr(minIndex+1, len-minIndex);

    // Recur for the above substring and n equals to n-minIndex
    buildLowestNumberRec(new_str, n-minIndex, res);
}

// A wrapper over buildLowestNumberRec()
string buildLowestNumber(string str, int n)
{
    string res = "";

    // Note that result is passed by reference
    buildLowestNumberRec(str, n, res);

    return res;
}

// Driver program to test above function
int main()
{
    string str = "121198";
    int n = 2;
    cout << buildLowestNumber(str, n);
    return 0;
}

```

Output:

```
1118>
```

This article is contributed by **Pallav Gurha**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

175. Count possible ways to construct buildings

Given an input number of sections and each section has 2 plots on either sides of the road. Find all possible ways to construct buildings in the plots such that there is a space between any 2 buildings.

Example:

```
N = 1
Output = 4
Place a building on one side.
Place a building on other side
Do not place any building.
Place a building on both sides.

N = 3
Output = 25
3 sections, which means possible ways for one side are
BSS, BSB, SSS, SBS, SSB where B represents a building
and S represents an empty space
Total possible ways are 25, because a way to place on
one side can correspond to any of 5 ways on other side.

N = 4
Output = 64
```

We strongly recommend to minimize your browser and try this yourself first

We can simplify the problem to first calculate for one side only. If we know the result for one side, we can always do square of the result and get result for two sides.

A new building can be placed on a section if section just before it has space. A space can be placed anywhere (it doesn't matter whether the previous section has a building or not).

```
Let countB(i) be count of possible ways with i sections
```

```
        ending with a building.
    countS(i) be count of possible ways with i sections
        ending with a space.

// A space can be added after a building or after a space.
countS(N) = countB(N-1) + countS(N-1)

// A building can only be added after a space.
countB[N] = countS(N-1)

// Result for one side is sum of the above two counts.
result1(N) = countS(N) + countB(N)

// Result for two sides is square of result1(N)
result2(N) = result1(N) * result1(N)
```

Below is C++ implementation of above idea.

```

// C++ program to count all possible way to construct buildings
#include<iostream>
using namespace std;

// Returns count of possible ways for N sections
int countWays(int N)
{
    // Base case
    if (N == 1)
        return 4; // 2 for one side and 4 for two sides

    // countB is count of ways with a building at the end
    // countS is count of ways with a space at the end
    // prev_countB and prev_countS are previous values of
    // countB and countS respectively.

    // Initialize countB and countS for one side
    int countB=1, countS=1, prev_countB, prev_countS;

    // Use the above recursive formula for calculating
    // countB and countS using previous values
    for (int i=2; i<=N; i++)
    {
        prev_countB = countB;
        prev_countS = countS;

        countS = prev_countB + prev_countS;
        countB = prev_countS;
    }

    // Result for one side is sum of ways ending with building
    // and ending with space
    int result = countS + countB;

    // Result for 2 sides is square of result for one side
    return (result*result);
}

// Driver program
int main()
{
    int N = 3;
    cout << "Count of ways for " << N
         << " sections is " << countWays(N);
    return 0;
}

```

Output:

25

Time complexity: $O(N)$

Auxiliary Space: $O(1)$

Algorithmic Paradigm: Dynamic Programming

Optimized Solution:

Note that the above solution can be further optimized. If we take closer look at the

results, for different values, we can notice that the results for two sides are squares of **Fibonacci Numbers**.

N = 1, result = 4 [result for one side = 2]

N = 2, result = 9 [result for one side = 3]

N = 3, result = 25 [result for one side = 5]

N = 4, result = 64 [result for one side = 8]

N = 5, result = 169 [result for one side = 13]

.....
.....

In general, we can say

```
result(N) = fib(N+2)2

fib(N) is a function that returns N'th
        Fibonacci Number.
```

Therefore, we can use **O(LogN) implementation of Fibonacci Numbers** to find number of ways in O(logN) time.

This article is contributed by **GOPINATH**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

176. Iterative Tower of Hanoi

Tower of Hanoi is a mathematical puzzle. It consists of three poles and a number of disks of different sizes which can slide onto any poles. The puzzle starts with the disk in a neat stack in ascending order of size in one pole, the smallest at the top thus making a conical shape. The objective of the puzzle is to move all the disks from one pole (say 'source pole') to another pole (say 'destination pole') with the help of third pole (say auxiliary pole).

The puzzle has the following two rules:

1. You can't place a larger disk onto smaller disk
2. Only one disk can be moved at a time

We've already discussed **recursive solution for Tower of Hanoi**. We have also seen that, for n disks, total $2^n - 1$ moves are required.

Iterative Algorithm:

```

1. Calculate the total number of moves required i.e. "pow(2, n)
   - 1" here n is number of disks.
2. If number of disks (i.e. n) is even then interchange destination
   pole and auxiliary pole.
3. for i = 1 to total number of moves:
    if i%3 == 1:
        legal movement of top disk between source pole and
        destination pole
    if i%3 == 2:
        legal movement top disk between source pole and
        auxiliary pole
    if i%3 == 0:
        legal movement top disk between auxiliary pole
        and destination pole

```

Example:

Let us understand with a simple example with 3 disks:

So, total number of moves required = 7

S

A

D

When i = 1, (i % 3 == 1) legal movement between 'S' and 'D'

When i = 2, (i % 3 == 2) legal movement between 'S' and 'A'

When i = 3, (i % 3 == 0) legal movement between 'A' and 'D' '

When i = 4, (i % 3 == 1) legal movement between 'S' and 'D'

When i = 5, (i % 3 == 2) legal movement between 'S' and 'A'

When i = 6, (i % 3 == 0) legal movement between 'A' and 'D'

When i = 7, (i % 3 == 1) legal movement between 'S' and 'D'

So, after all these destination pole contains all the in order of size.

After observing above iterations, we can think that after a disk other than the smallest disk is moved, the next disk to be moved must be the smallest disk because it is the top disk resting on the spare pole and there are no other choices to move a disk.

```
// C Program for Iterative Tower of Hanoi
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a stack
struct Stack
{
    unsigned capacity;
    int top;
    int *array;
};

// function to create a stack of given capacity.
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack =
        (struct Stack*) malloc(sizeof(struct Stack));
    stack -> capacity = capacity;
    stack -> top = -1;
    stack -> array =
        (int*) malloc(stack -> capacity * sizeof(int));
    return stack;
}

// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{
    return (stack->top == stack->capacity - 1);
}

// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack)
{
    return (stack->top == -1);
}

// Function to add an item to stack. It increases
// top by 1
void push(struct Stack *stack, int item)
{
    if (isFull(stack))
        return;
    stack -> array[++stack -> top] = item;
}

// Function to remove an item from stack. It
// decreases top by 1
int pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack -> array[stack -> top--];
}
```

```

}

// Function to implement legal movement between
// two poles
void moveDisksBetweenTwoPoles(struct Stack *src,
                             struct Stack *dest, char s, char d)
{
    int pole1TopDisk = pop(src);
    int pole2TopDisk = pop(dest);

    // When pole 1 is empty
    if (pole1TopDisk == INT_MIN)
    {
        push(src, pole2TopDisk);
        moveDisk(d, s, pole2TopDisk);
    }

    // When pole2 pole is empty
    else if (pole2TopDisk == INT_MIN)
    {
        push(dest, pole1TopDisk);
        moveDisk(s, d, pole1TopDisk);
    }

    // When top disk of pole1 > top disk of pole2
    else if (pole1TopDisk > pole2TopDisk)
    {
        push(src, pole1TopDisk);
        push(src, pole2TopDisk);
        moveDisk(d, s, pole2TopDisk);
    }

    // When top disk of pole1 < top disk of pole2
    else
    {
        push(dest, pole2TopDisk);
        push(dest, pole1TopDisk);
        moveDisk(s, d, pole1TopDisk);
    }
}

//Function to show the movement of disks
void moveDisk(char fromPeg, char toPeg, int disk)
{
    printf("Move the disk %d from \'%c\' to \'%c\'\n",
           disk, fromPeg, toPeg);
}

//Function to implement TOH puzzle
void tohIterative(int num_of_disks, struct Stack
                 *src, struct Stack *aux,
                 struct Stack *dest)
{
    int i, total_num_of_moves;
    char s = 'S', d = 'D', a = 'A';

    //If number of disks is even, then interchange
    //destination pole and auxiliary pole
    if (num_of_disks % 2 == 0)
    {
        char temp = d;
        d = a;
    }
}

```



```

        a = temp;
    }
    total_num_of_moves = pow(2, num_of_disks) - 1;

    //Larger disks will be pushed first
    for (i = num_of_disks; i >= 1; i--)
        push(src, i);

    for (i = 1; i <= total_num_of_moves; i++)
    {
        if (i % 3 == 1)
            moveDisksBetweenTwoPoles(src, dest, s, d);

        else if (i % 3 == 2)
            moveDisksBetweenTwoPoles(src, aux, s, a);

        else if (i % 3 == 0)
            moveDisksBetweenTwoPoles(aux, dest, a, d);
    }
}

```

```

// Driver Program
int main()
{
    // Input: number of disks
    unsigned num_of_disks = 3;

    struct Stack *src, *dest, *aux;

    // Create three stacks of size 'num_of_disks'
    // to hold the disks
    src = createStack(num_of_disks);
    aux = createStack(num_of_disks);
    dest = createStack(num_of_disks);

    tohIterative(num_of_disks, src, aux, dest);
    return 0;
}

```

Output:

```

Move the disk 1 from 'S' to 'D'
Move the disk 2 from 'S' to 'A'
Move the disk 1 from 'D' to 'A'
Move the disk 3 from 'S' to 'D'
Move the disk 1 from 'A' to 'S'
Move the disk 2 from 'A' to 'D'
Move the disk 1 from 'S' to 'D'

```

References:

http://en.wikipedia.org/wiki/Tower_of_Hanoi#Iterative_solution

This article is contributed by **Anand Barnwal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

