bits

```
void rbitval(unsigned long *n)
{   // reverse (toggle) each bit
    *n = *n ^ 0xFF;
}

int countbits(unsigned int d)
{
    int c = 0;
    while (d)
    {
        d &= d-1; ← clear LSB
        c++;
    }
    return c;
}

void reverse8bits(unsigned short* d)
{
    unsigned short v = 0;
    v = ((0x80 & *d) >> 7) |
        ((0x40 & *d) >> 5) |
        ((0x20 & *d) >> 3) |
        ((0x10 & *d) >> 1) |
        ((0x08 & *d) << 1) |
        ((0x04 & *d) << 3) |
        ((0x02 & *d) << 5) |
        ((0x01 & *d) << 7);
    *d = v;
}

void changedselectedbits(unsigned long *val, int U, int L)
{
    unsigned long temp = *val;
    unsigned long mask = 0;
    int bits = 0;
    for(int i = 0; i <= (U-L); i++)
        bits++;

    mask = 0xFFFFFFFF << (U+1); // clear out all bits lower than U
    mask += (0xFFFFFFFF >> (32-L)); // clear all bits between U and L
    mask ^= 0xFFFFFFFF; // set all bits between U and L
    printf("mask:%0X\n", mask);
    printf("val:%0X\n", *val);
    printf("reverse %d bits between upper:%d lower:%d \n", bits, U, L);
    temp = temp ^ mask; // XOR: A^A = 0, A^0=A (1^1 = 0 , 1^0 = 1)
    *val = temp;
}
```

String Search Cmp

```
int findmissing(int a[], int size)
{ // Non-repeat, & all positive int
    int i = 0, val = -1, total = 0;
    int sum = (9*(9+1))/2;

    for (i = 0; i< size; i++)
        total += a[i];

    val = sum - total;
    return val;
}
int StrCmp(char* s1, char* s2)
{
    while (*(s1++) == *(s2++))
    {
        if (*s1 == '\0' && *s2 == '\0')
            return 1;
    }
    return 0;
}

void revstring(char* s)
{
    char c;
    int i,j, len;
    len = strlen(s)-1;
    j = strlen(s)/2;

    for (i = 0; i<j; i++, len--)
    {
        c = *(s+len);
        *(s+len) = *(s+i);
        *(s+i) = c;
    }
}

void revstringX(char* s)
{
    int i,j, len;
    len = strlen(s)-1;
    j = strlen(s)/2;

    for (i = 0; i<j; i++, len--)
    {
        *(s+i)   ^= *(s+len);
        *(s+len) ^= *(s+i);
        *(s+i)   ^= *(s+len);
    }
}
```

numbers

```
int ispowerof2(unsigned int d)
{
    // the idea is, a power_of_2 number has only the MSB being 1
    // test: subtract 1 from the val then bit-wise AND with val
    //      if the result is not zero, that means the MSB is still there
    //      showing that one or more lower significant bits were there.
    // thus, non-zero result say that the number is not power of 2.
    if (d & (d-1))
        return 0;
    return 1;   // otherwise
}

void swaptwonums(float* n1, float* n2)
{
    *n1 = *n1 + *n2;
    *n2 = *n1 - *n2;
    *n1 = *n1 - *n2;
}

unsigned int revdecdigit(unsigned int val)
{
    unsigned int ret = 0, rem = 0;
    while (val > 0)
    {
        rem = val % 10; // get the remainder (the least digit)
        ret = ret*10 + rem; // shift the digit by one to the left
        val = val / 10; // get the next upper digit
    }
    return ret;
}

unsigned long changeEndian(unsigned long d)
{
    unsigned long v = 0;
    v = ((d & 0xFF000000) >> 24) |
        ((d & 0x00FF0000) >> 8 ) |
        ((d & 0x000000FF) << 24) |
        ((d & 0x0000FF00) << 8);
    return v;
}

int isprime(unsigned int val)
{
    unsigned long i = 2;
    unsigned int p = sqrt(val);
    printf("p:%i\n", p);
    while (i <= p) {
        if ((val % i) == 0)
            return 0;   // not a prime
        i++;
    }
    return 1; // yes, it is a prime
```

So that the devider value does not need be bigger than one-half of the val.

```
int findapeak(int* a, unsigned* index, unsigned int start, unsigned int len)
{
   unsigned int end = len-1;
   unsigned int m = 0;

   while (start <= end)
    {                                          m
      m = (start + end)/2;
      printf("start: %d, mid:%d, end: %d\n", start, m, end);
      if (m < 1) {
          *index = m;  ← end of search and m == 0 as (end & start) ≤ 1
         return a[m];
     }
      // ascend test
      if (a[m-1] < a[m])
      {
         if (m == end)        // 1. the peak is at the end
         {
             *index = m;
             return a[m];
         }
         if (a[m] > a[m+1])
         {
             *index = m;
             return a[m];
         }
         start = m+1;
      }
      else  // test descend
      if (a[m] > a[m+1])
      {
         if (m == start)     // 2. the peak is at the start
         {
            *index = m;
            return a[m];
         }
         if (a[m] > a[m-1])
         {
            *index = m;
            return a[m];
         }
         end = m-1;
      }
      else
      {
          // there's a dip between two adjacent number
          // just add an offet to the start or end
          start += 1;
      }
    }
    return 0;        // something else
}.
```

button : LED

find x

```
int bsearch(int d, int a[], int size,  int findfirst)
{
// ~~first~~ find x from a sorted array
   int index = -1;
    int length = 0;
   int low = 0, mid = 0;
    int high = size-1;

   // need check for equality of the two as they can meet
    // the equal check is needed as the target can be at the floor or ceiling
   while (low <= high)
   {
       mid = (high + low)/2;
        // use the difference to avoid overflow from the sum
        //mid = (high-low)/2 + low;  // lower offset needed as it raises
      if (d == a[mid])
        {
            index = mid;
            if (findfirst == 1)   // continue search on the position below mid
            {
                high = mid-1;
            }
            else    // position wanted is the highest, so continue on above mid
            {
                low = mid+1;
            }
        }
        else
        {
          if (d < a[mid])
             high = mid-1;
          else
             low = mid+1;
        }
   }
   return index;  //   can be the lowest or highest or the first middle one.
}
```

↑ option to find the lowest index if the flag is 1 otherwise report the one found first

optional

Find Duplicate

```
int getdupunsorted(int ua[], int size, int* duplicates)
{
    // find duplicates in unsorted array when 1 <=  x <= len(array)  pre Condition
    // idea: encode the info (number sign)to a borrowed position in the array
    // suppose an array[ 6,1,5,7, 8,9,9,3, 5,4]
    // first val in array would construct an index: (6-1)=5
    // check if ua[5] has been negated, if so add the orginal val to result set
    // else, it means no duplicate is found,
    // then negate the val in the array pointed by the construct index.
    int val = 0, i, index = - 1;
    // duplicates is a pointer of array to hold the duplicates to be found

    for (i = 0; i < size; i++)
    {
        index = abs(ua[i]) - 1; // since x >= 1, adjust the index from 0 up
        if (ua[index] < 0) { // was it negated before?
            *duplicates = abs(ua[i]); // if so add it to the result set
            duplicates++;
            val++;          -1: has been visited
        }
        else    // a new value from the array was not yet negated, so go do it
            ua[index] = -ua[index]; // the change will be checked
    }
    return val;
}
```

```
//////////////// hash table ////////////////////
int hashc(char c)
{
    return (c - 'a'); // key is made by the val of the character
}
void countrepeatedchar(char* s, int counts[])
{         // find duplicate
    int count = 0, i, index = 0, len = strlen(s) -1;

    if (s == 0) return;

    for (i = 0; i <= len; i++)
    {
        index = hashc((char)(s[i]));  ← use key as an index.
        counts[index]++;
    }
}
```

find X ←→ Rotating

Special case

```
int searchRotated(int left, int right, int d, int a[])
{
// find x from a rotated sorted array
    int mid = 0, index = -1;

    if (a[left] == d) {
        printf("%d is at the start:%d\n", d, left);
        return left;
    }
    if (a[right] == d) {
        printf("%d is at the end:%d\n", d, right);
        return right;
    }
    if (left >= right) {
        printf("%d is not found from %d to %d\n", d, left, right);
        return -1;
    }

    //using recursion has not need to use temporary variable
    // to keep track of boundaries left and right.
    mid = (left+right)/2;
    printf("left:%d, mid:%d, right:%d\n", left, mid, right);
    if (a[mid] == d)                                  m
        return mid;

   if (a[left] < a[mid])                                     S          U S
   {  //  printf("left half is seen sorted\n");
      if (a[left] <= d && d <= a[mid]) // check if d is in left half
        {
           printf("check if %d is in the left of sorted left from mid:%d\n", d, mid);
         return searchRotated(left, mid-1, d, a);
        }
      else   // go right half
      {   //  printf("check if %d is in the right of sorted left from mid:%d\n", d,
      mid);
         return searchRotated(mid+1,right, d, a);
      }
   }
   else                                                       US        S.
   {  // printf("right half is seen sorted\n");
      if(a[mid] <= d && d <= a[right])    // check if d is in right half
        {   // printf("check if %d is in right of sorted right from mid:%d\n", d, mid);
         return searchRotated(mid+1, right, d, a);
        }
      else    // go left half
      {  // printf("check if %d is in left of sorted right from mid:%d\n", d, mid);
         return searchRotated(left, mid-1, d, a);
      }
   }
    return index;
}
```

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

int len;
////Merge sort in non decreasing order
                                  L           R.
void merge(int arr[], int i, int mid, int j) {
    printf("Left: ");
    printArray(arr, i, mid);
    printf(" Right: ");
    printArray(arr, mid + 1, j);
    printf("\n");
    int temp[len]; ← need to hold one half x 2.
    int l = i, r = j;
    int m = mid + 1;
    int k = l;

    while(l <= mid && m <= r) {
        if(arr[l] <= arr[m]) {
            temp[k++] = arr[l++];
        }
        else {
            temp[k++] = arr[m++];
        }
    }

    while(l <= mid)
        temp[k++] = arr[l++]; ← copy the left part

    while(m <= r) {
        temp[k++] = arr[m++]; ← copy the right part.
    }

    for(int i1 = i; i1 <= j; i1++) {
        arr[i1] = temp[i1]; ← finally copy back to the array
    }

    printf("After Merge: ");
    printArray(arr, i, j);
    printf("\n");
                                   start from 0,
}
void mergesort(int arr[], int i, int j) { ← start from array (len-1.)
    int mid = 0;

    if(i < j) {
        mid = (i + j) / 2;
        mergesort(arr, i, mid); ← sort left half.
        mergesort(arr, mid + 1, j); ← sort right half
        merge(arr, i, mid, j);
    }
}
```

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

int main(void)
{
    char *pma1[4] = { "first ", "second "};
    char  ma2[3][20];

    ma2[0][0] = 'T';
    ma2[0][1] = 'o';
    ma2[0][2] = 'm';
    ma2[1][0] = 'H';
    ma2[1][1] = 'u';
    ma2[1][2] = 'a';

    pma1[1] = (char*)("second\0");   // can use direct assignment with pointer type
    pma1[2] = (char*)("third\0");
    memcpy(&ma2[2], "test\0", 5);   // need help from string library for array type

    printf("ma1[1]: %s\n", pma1[1]);
    printf("ma1[2]: %s\n", pma1[2]);
    printf("ma2[0]: %s\n", ma2[0]);
    printf("ma2[2]: %s\n", ma2[2]);
}

int main(void)
{
    char *ma1[4][20];   // needs to specify enough space if assign later
    char  ma2[3][20];

    ma2[0][0] = 'T';
    ma2[0][1] = 'o';
    ma2[0][2] = 'm';
    ma2[1][0] = 'H';
    ma2[1][1] = 'u';
    ma2[1][2] = 'a';

    *ma1[1] = "second\0";   // can use direct assignment
    *ma1[2] = "third\0";
    memcpy(&ma2[2], "test\0", 5);   // need help from string library

    printf("ma1[1]: %s\n", ma1[1][0]);
    printf("ma1[2]: %s\n", ma1[2][0]);
    printf("ma2[0]: %s\n", ma2[0]);
    printf("ma2[2]: %s\n", ma2[2]);
}
```