Mergesort
1) Divide unsorted list into n sublists of 1 element each
2) Merge sublists to produce new sublists until there is only 1 sublist remaining

```
void merge_sort(int data[], int end1, int end2)
      {
      int * temp = new int[end2];           //Temporary storage of merged arrays
      int i = 0; int j = end1; int k = 0;
      while (i < end1 && j < end2)          //Neither i or j have reached the end of their
arrays
           {
            if(data[i]<data[j])
                 {
                  temp[k]=data[i];
                  i=i+1;
                  k=k+1;
                 }
           else
                 {
                  temp[k]=data[j];
                  j=j+1;
                  k=k+1;               //Move smaller value to temp array
                 }
           }
      while (i<end1)                   //If one array has finished, move the rest of the values
for the other
           {
            temp[k]=data[i];
            i=i+1;
            k=k+1;
           }
      while (j<end2)
           {
            temp[k]=data[j];
            j=j+1;
            k=k+1;
           }
      for(i=0;i<end2;i++)              //save temp back to original array
            data[i]=temp[i];
      int half;
      delete [] temp;
      }

void merge_all(int data[], int size)
      {
      for (int i=1;i<size;i=2*i)            //Double jump size as arrays being merged get
larger.
            for(int j=0;j<size-i;j=j+2*i)   //Increment array number relative to i.
                {
                  int end2 = (2*i<size-j)?2*i:size-j; //Choose which is smaller as end2
                 merge_sort(&(data[j]),i,end2);         //Recursively call merge sort
                }
      }
```

Quicksort
1) Pick a pivot element from list
2) Reorder the list so that all elements with values less than pivot come before pivot, all elements with values greater than pivot come after it.
3) Recursively sort sub list of lesser elements and sub list of greater elements

```
int partition(int Q[], int start, int end)
      {
      if (end-start<1)                    //If the partition is greater than one.
           {
            return -1;                   //Will be used as condition in quick_sort function to end
recursion.
           }
      else
           {
            int pivotp = random_in_range(start,end); //.    Location of pivot
            int pivot = Q[pivotp];                //      Number for pivot
            int L = start;
            int U = end;
            swap(Q[pivotp],Q[L]);
            pivotp = start;
            L+=1;
            while (L < U)
                 {
                  while (Q[L] < pivot && L<U)
                        L+=1;
                  while (Q[U] >= pivot && U>L)
                        U = U - 1;
                 swap (Q[L], Q[U]);
                 if (L<U)
                         L=L+1;  //L will be incremented after every swap unless L and U
point to the same object
                 if (U>L)
                         U=U-1;  //L and U will always point to same object at end of partition
                 }
            if (Q[L]>=pivot)
                 {
                  return L; //If L is already pointing to 1st object greater than pivot, return L.
```

```
                }
            if (Q[L]<pivot)
                {
                  return L+1; //If L is pointing to last object less than pivot, increment L by 1.
                }
            }
      }

void quick_sort (int allthethings[], int start, int end)
      {
       int p = partition(allthethings, start, end);    //First partition goes from zero to the end.
       if (p>=0)
            {
             quick_sort(allthethings, start, p-1);   //Quicksort from start to last object before
pivot
             quick_sort(allthethings, p, end);       //Quicksort from first object after pivot to
end
            }
         }
```

- Best quicksort is log n, approaches n^2 if pivot is bad.
- If data is random, always choose the same pivot (first, last, etc)
- However, most data is almost always already sorted and so if you choose the first element the algorithm approaches worst possible time
- Best case would be the median, which can be found in n.
- However this is usually not necessary, and median of three works better while avoiding problems described above.
- Taking median of elements at left, right, and middle works best.

Quicksort vs. Mergesort
- Mergesort is stable and its worst case is O(nlogn)
- Mergesort works better with linked lists, while quicksort can be implemented on linked lists, there is often poor pivot choices.
- Mergesort requires O(n) extra space for arrays, while quicksort can be partitioned in-place.

General Comparison Sorts
- Elements read through single less than or equal to operator
- Comparison sorts must have lower bound of O(nlogn)
- This is due to limited information available through comparison alone, or the vague algebraic structure of totally ordered sets.
- There are n factorial possible permutations, only one of which is the correct order
- If the algorithm completes after at most f(n) steps, it cannot distinguish more than 2^f(n) cases at a time because keys are distinct and each comparison only has two possible outcomes.
- 2^f(n) > n! or f(n) > log2(n!)

- Stirlings approximation says log2(n!) = O(nlog2n)

Radix Sort
- More efficient for larger numbers that are small in comparison to size of array
- 1) Take least significant digit of each number and sort based on that
  2) Sort based on second least significant digit, etc
- Use modulo function to find specific digits of numbers
- m= 10, n= 10
- Put values in linked list of digits 0-9 based on LSD
- Read into new array in order of LSD 0-9, keeping order that they were put in list
- Mod is incremented by *10 and then divided by 10 to find the next least significant integer
- Keep sorting until maximum integer column is done.

Bucket Sort = Counting Sort
- Not really a sorting algorithm, its used to compliment other sorting algorithms and is not generally used on its own
- Breaks down into smaller lists and sorts internally, then joins back together to sort again
- Items first placed into necessary buckets based on size, then original sorting album orders the buckets.
- Buckets are then joined head to tail.
- You don't want to run bucket sort using recursion

Hash Functions

Contains an array which objects or sorted into, with the array address based on the data key and the array Length.

Requirements for good function:
- Uniform distribution of hash values, in order to reduce collisions which cost time to resolve.
- Can be figured out using statistical tests.
- In open addressing, the hash unction needs to avoid clustering (mapping two or more keys consecutively), which costs a lot to lookup.
- Distribution uniformity is dependent on size of the hash table.
- Cryptic hash functions: Give good functions with any size n.
- Cryptic hash functions use modulo reduction or masking (bitwise operations)
- Masking: To create a hashing function for a hash table often a function is used that has a large domain. To create an index from the output of the function, a modulo can be taken to reduce the size of the domain to match the size of the array; however, it is often faster on many processors to restrict the size of the has table to powers of two sizes and use a bit mask instead.

```
/* THIS is a TERRIBLE hash function, it has two fatal flaws

   int hash(string s, int hashsize)
   { int h = 983475;
     for (int i=0; i<s.length(); i+=1)
       h *= s[i];
     if (h<0)
       h = -h;
     return h % hashsize; }  */


// but THIS is a good one

int hash(string s, int hashsize)
{ int h = 983475;
  for (int i=0; i<s.length(); i+=1)
    h *= h * 0xA5A5 + s[i];              // Multiplying by itself
and a bit mask which allows for a uniform distribution for any
array size.
  if (h<0)
    h = -h;
  if (h<0)
    h = 982754983;                       // Because of twos
compliment to convert between positive and negative?
  return h % hashsize;
}
```

- Keep the table between 25 and 50% percent full and you get an expected average constant tie insert and search operations. Remember to resize the table by doubling and rehashing everything.
- If the load factor is over 50%, probability of collisions and the cost of resolving them will increase, especially with open addressing methods.
- If the load factor is too small, the proportion of unused areas will increase and there will be little reduction in search cost which leads to wasted memory.
- A good hash function should follow the poisson distribution.
- N = number of things being sorted; H = size of the array; lambda = N/H; k is variable, how likely any individual number is (len)

```
int poisson(int n, int h, int len)
{ double lambda = n/(double)h, fact = 1.0;
  double top = exp(- lambda) * pow(lambda, len);             //
(e^-lambda)(lambda raised to length)
```

```
  for (int i = 1; i <= len; i += 1)
    fact *= i;                                              //
factorial of k
  return (int) round(h * top / fact); }                     //
returns numerator over denominator rounded

void hashtable::resize()
        {
        if (N>H/2)                                            //
Resizes only if over half of spaces used.
                {
                H=H*2;                                        //
Table size doubled.
                hashlink**table2=new hashlink*[H];
                for (int i=0; i<H; i+=1)
                        table2[i]=NULL;
                for (int i=0; i<H/2; i+=1)
                        {
                        if(table[i]!=NULL)
                                {
                                string k=table[i]->data->name;
                                table2[hashfunc(k)]=table[i];
                                }
                        }
                delete[]table;
                table=table2;
                }
        }
```

Linear Probing = Open Addressing = Closed Hashing
- form of open addressing, its stored in the array itself when there is a conflict
- When there is a conflict, there is a starting value and a step size, with the starting value incremented by step size until a free value is found or the entire table is traversed
- Results in clustering, which leads to encountering multiple collisions while searching for an open spot. Step size is usually one.

Chaining = Closed Addressing = Open Hashing
- Each slot of the bucket array is a pointer to a linked list to key-value pairs that were hashed to the same location
- Lookup by scanning the list for an entry based on key.
- Insert a new entry to either end of the list.
- Cost is dependent on number of keys per bucket
- Remains effective when there are more items to store than spaces

```
in the bucket
- Performance degrades linearly as opposed to open addressing
methods.

void hashtable::separatechain(city*c,hashlink*h,int count)
        {
        if (h->next==NULL)
                {
                h->next=new hashlink(c,NULL);
                count=count+1;
                cout<<"City: " << c->name << ", Hash Tag: " <<
hashfunc(c->name) << "\n";
                }
        else
                separatechain(c,h->next,count+1);
        }
```

C Functions:

`FILE * fopen ( const char * filename, const char * mode );`

- Opens specified file, and associates with a specified stream
- The file access mode is specified by char parameter:
    - "r" is read: Open file for input operations, file already exists.
    - "w" is write: Create empty file for output operations, writing data to end of file and ignoring repositioning statements.
    - "a" is append: Create/Open file for output at end of file, writing to end of file and ignoring repositioning operations.
    - "r+" is read/update: Open existing file for update (input and output)
    - "w+" is write/update: Create empty file for input and output update. If a file already exists, it is overwritten with blank file
    - "a+" append/update: Open file for update (input or output) with all output operations writing data at the end of the file. Repositioning operations work for input, but not output. Create file if it doesn't exist.

`int fgetc ( FILE * stream );`

- Returns character pointed to by internal file position indicator of specified stream. Indicator is then advanced to the next character.
- Returns EOF at end of the file
- EOF value is -1

`char * fgets ( char * str, int num, FILE * stream );`

- Read characters from *stream* and stores as C string into *num*-1 characters have been read or either the newline or end of file is reached.
- Newline character is copied into str.
- Null character appended after characters copied to str.
- Returns *str* if successful.

```
size_t fread ( void * ptr, size_t size, size_t count, FILE *
stream );
```

- Reads array of *count* elements, each of *size* bytes from specified *stream* and stores them in memory specified by *ptr.*
- Position indicator advanced by total amount of bytes read.
- Total amount of byres read is size*count
- Returns total number of elements successfully read
- If this is different from count parameter there was an error

```
int fseek ( FILE * stream, long int offset, int origin );
```

- Sets position indicator associated with *stream* to a new position.
- New position defined by adding *offset* to reference position *origin.*
- Three constants:
    SEEK_SET = Beginning of file
    SEEK_CUR = Current position of file pointer
    SEEK_END = End of file
- Returns zero if successful, else returns non-zero value

```
int fclose ( FILE * stream );
```

- Closes file associated with *stream* and disassociates it.

```
int printf ( const char * format, ... );
```

- Writes c string pointed by format to standard outputs
- If *format* includes format specifiers, additional arguments are formatted and inserted in resulting string.

Other C Details:
- Strings are arrays of characters with extra '\0' at end
- Anything in "double quotes" is one of those strings in both C and C++
- C++ string assignments actually call s.operator=("string") which converts to a C++ string.
- FILE pointers stein, stout, and steer already exist.

Shortest Path:
struct node:

string name
vector[link-pointer] others
int distance //shortest distance from starting point discovered so far

struct link:
int * to
int length = no_path = -99999;

The whole graph is explored and shortest distance to everywhere is found.

```
 void find_distances(node * A, int so_far)
              // A is where we are now,
              // so_far is how far from the start we already are
   {
     assert(A != NULL);

     if (A->distance != no_path && A->distance <= so_far)
        return;          //Don't continue exploring if there the 
distance already found to a point is less than the      distance 
accumulated to that point.
     A->distance = so_far; //Replace shortest distance if current 
distance is shorter

     int n = A->others.size();
     for (int i=0; i<n; i+=1)      //explore through entire array 
of other locations
      {
         node * next_town = A->others[i]->to;
        int road_length = A->others[i]->length;
        find_distances(next_town, so_far + road_length);
         }         //explore next town in others array, add 
current distance from beginning and the length distance from 
current and next towns
        }

   void find_distances(node * start)
    { find_distances(start, 0);
     }
```

Recursion Elimination
Stacks:
- Operator LIFO (last in first out), elements are inserted and extracted rom the end of the container.
- Elements are pushed/popped from the "back" of the container which is known as the "top".

- Member functions:
    - empty: tell whether container is empty
    - size: Return size
    - top: Access next element
    - push: add elements
    - pop: remove element
    - back: return reference to last element in a vector container

Non-recursive function for finding the sum of a tree:

```
struct stackframe
{ node * t;
  int sum;
  char wwi;
  stackframe(node * a, int b, char c)
  { t=a; sum=b; wwi=c; } };

int run_sum(node * t)
{ vector <stackframe> stack;
  stack.push_back(stackframe(t, 0, 'A'));      //Most recent
location and sum
  int result;
  while (! stack.empty())
  { stackframe & top = stack.back();
    switch (top.wwi)
    { case 'A':
        if (top.t == NULL)
        { result = 0;
          stack.pop_back();
          break; }
        top.wwi = 'B';
        stack.push_back(stackframe(top.t->left, 0, 'A'));
        break;
      case 'B':
        top.sum = result;
        top.sum += top.t->data;
        top.wwi = 'C';
        stack.push_back(stackframe(top.t->right, 0, 'A'));
        break;
      case 'C':
        top.sum += result;
        result = top.sum;
        stack.pop_back();
        break; } }
  return result;
```