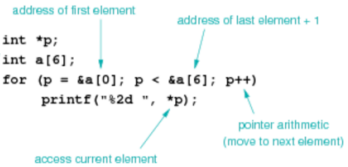
2. 3. gcc -Wall -Werror -std=c11 -c asd/c. char 1, int 4, *pointer 4, float 4, double 8, c%, d%, f%, lf%-double. Pointer: p = &a atoi(*str -> int) atof(*str->float/double)

```
insertionSort(A):
    Input array A[0..n-1] of n elements

    for all i=1..n-1 do
        element=A[i], j=i-1
        while j≥0 and A[j]>element do
            A[j+1]=A[j]
            j=j-1
        end while
        A[j+1]=element
    end for
```

```
#include "Stack.h"
#include <assert.h>

// define the Data Structure
typedef struct {
    char item[MAXITEMS];
    int  top;
} stackRep;

// define the Data Object
static stackRep stackObject;

// set up empty stack
void StackInit() {
    stackObject.top = -1;
}

// check whether stack is empty
int StackIsEmpty() {
    return (stackObject.top < 0);
}
```

```
// insert char on top of stack
void StackPush(char ch) {
    assert(stackObject.top < MAXITEMS-1);
    stackObject.top++;
    int i = stackObject.top;
    stackObject.item[i] = ch;
}

// remove char from top of stack
char StackPop() {
    assert(stackObject.top > -1);
    int i = stackObject.top;
    char ch = stackObject.item[i];
    stackObject.top--;
    return ch;
}
```




4. 5. code 是像固定的函数 f(),main().global 是全局变量，heap 是动态分配的内存，stack 是函数局部变量。双指针 **P 一般矩阵，传参数

An executing C program partitions memory into:

- *code* ... fixed-size, read-only region
  - contains the machine code instructions for the program
- *global data* .. fixed-size, read-write region
  - contain global variables and constant strings
- *heap* ... very large, read-write region
  - contains dynamic data structures created by `malloc()` (see later)
- *stack* ... dynamically-allocated data (function local vars)
  - consists of frames, one for each currently active function
  - each frame contains local variables and house-keeping info

Matrix:

```
float **matrix = malloc(m * sizeof(float *));
assert(matrix != NULL);
int i;
for (i = 0; i < m; i++) {
   matrix[i] = malloc(n * sizeof(float));
   assert(matrix[i] != NULL);
}
```

$4m + 4 \cdot mn$ bytes allocated

```
matrixProduct(A,B):
    Input  n×n matrices A, B
    Output n×n matrix A×B

    for all i=1..n do                          2n+1
        for all j=1..n do                      n(2n+1)
            C[i,j]=0                           n²
            for all k=1..n do                  n²(2n+1)
                C[i,j]=C[i,j]+A[i,k]×B[k,j]    n³×3
            end for
        end for
    end for
    return C                                   1
                                               ----------
                                    Total      7n³+4n²+3n+2
```

6. 7. adjacency matrix 用双指针，DFS array O(V+E2) matrix O(v2) list O(V+E)，BFS O(V+E)。Ham path 是点不重复 O((V-1)!)，Euler path 是边不重复 O(n2)。Components.可到达.

## Comparison of Graph Representations

| | array of edges | adjacency matrix | adjacency list |
|---|---|---|---|
| space usage | $E$ | $V^2$ | $V+E$ |
| initialise | 1 | $V^2$ | $V$ |
| insert edge | 1 | 1 | 1 |
| find/delete edge | $E$ | 1 | $V$ |

Other operations:

| | array of edges | adjacency matrix | adjacency list |
|---|---|---|---|
| disconnected(v)? | $E$ | $V$ | 1 |
| isPath(x,y)? | $E \cdot \log V$ | $V^2$ | $V+E$ |
| copy graph | $E$ | $V^2$ | $E$ |
| destroy graph | 1 | $V$ | $E$ |

### ... Graph ADT (Adjacency Matrix)

Implementation of graph initialisation (adjacency-matrix representation)

```
Graph newGraph(int V) {
   assert(V >= 0);
   int i;

   Graph g = malloc(sizeof(GraphRep));      assert(g != NULL);
   g->nV = V;  g->nE = 0;

   // allocate memory for each row
   g->edges = malloc(V * sizeof(int *));    assert(g->edges != NULL);
   // allocate memory for each column and initialise with 0
   for (i = 0; i < V; i++) {
      g->edges[i] = calloc(V, sizeof(int)); assert(g->edges[i] != NULL);
   }
   return g;
}
```

```
hasPath(G,src,dest):
    Input  graph G, vertices src,dest
    Output true if there is a path from src to dest in G,
           false otherwise

    return dfsPathCheck(G,src,dest)

dfsPathCheck(G,v,dest):
    mark v as visited
    if v=dest then       // found dest
        return true
    else
        for all (v,w)∈edges(G) do
            if w has not been visited then
                return dfsPathCheck(G,w,dest) // found path via w to dest
            end if
        end for
    end if
    return false         // no path from v to dest
```

BFS algorithm (records visiting order, marks vertices as visited when put on q

```
visited[] // array of visiting orders, indexed by vertex 0..

findPathBFS(G,src,dest):
    Input  graph G, vertices src,dest

    for all vertices v∈G do
        visited[v]=-1
    end for
    enqueue src into new queue q
    visited[src]=src
    found=false
    while not found and q is not empty do
        dequeue v from q
        if v=dest then
            found=true
        else
            for each (v,w)∈edges(G) such that visited[w]=-1 do
                enqueue w into q
                visited[w]=v
            end for
        end if
    end while
    if found then
        display path in dest..src order
    end if
```

```
components(G):
    Input graph G

    for all vertices v∈G do
        componentOf[v]=-1
    end for
    compID=0
    for all vertices v∈G do
        if componentOf[v]=-1 then
            dfsComponents(G,v,compID)
            compID=compID+1
        end if
    end for

dfsComponents(G,v,id):
    componentOf[v]=id
    for all vertices w adjacent to v do
        if componentOf[w]=-1 then
            dfsComponents(G,w,id)
        end if
    end for
```

```
hasEulerPath(G,src,dest):
    Input  graph G, vertices src,dest
    Output true if G has Euler path from src to dest
           false otherwise

    if src≠dest then
        if degree(G,src) or degree(G,dest) is even then
            return false
        end if
    else if degree(G,src) is odd then
        return false
    end if
    for all vertices v∈G do
        if v≠src and v≠dest and degree(G,v) is odd then
            return false
        end if
    end for
    return true
```

*Theorem.* *A graph has an Euler circuit* if and only if it is connected and all vertices have even degree

*Theorem.* *A graph has a non-circuitous Euler path* if and only if it is connected and exactly two vertices have odd degree

### ... Transitive Closure

Cost analysis:

- storage: additional $V^2$ items (each item may be 1 bit)
- computation of transitive closure: $V^3$
- computation of `reachable()`: *O(1)* after having generated `tc[][]`

```
visited[] // array [0..nV-1] to keep track of visitec

hasHamiltonianPath(G,src,dest):
    for all vertices v∈G do
        visited[v]=false
    end for
    return hamiltonR(G,src,dest,#vertices(G)-1)

hamiltonR(G,v,dest,d):
    Input G     graph
          v     current vertex considered
          dest  destination vertex
          d     distance "remaining" until path found

    if v=dest then
        if d=0 then return true else return false
    else
        visited[v]=true
        for each (v,w)∈edges(G) ∧ ¬visited[w] do
            if hamiltonR(G,w,dest,d-1) then
                return true
            end if
        end for
    end if
    visited[v]=false            // reset visited mark
    return false
```

### ... Transitive Closure

If we implement the above as:

```
make tc[][] a copy of edges[][]
for all i∈vertices(G) do
   for all s∈vertices(G) do
      for all t∈vertices(G) do
         if tc[s][i]=1 and tc[i][t]=1 then
            tc[s][t]=1
         end if
      end for
   end for
end for
```

then we get an algorithm to convert edges into a *tc*

This is known as *Warshall's algorithm*

```
hasCycle(G):
    Input  graph G
    Output true if G has a cycle, false otherwise

    mark all vertices as unvisited
    for each vertex v∈G do           // make sure
        if v has not been visited then
            if dfsCycleCheck(G,v,v) then
                return true
            end if
        end if
    end for
    return false

dfsCycleCheck(G,v,u):      // look for a cycle th
    mark v as visited
    for each (v,w)∈edges(G) do
        if w has not been visited then
            return dfsCycleCheck(G,w,v)
        else if w≠u then
            return true
        end if
    end for
    return false
```

8. MST(kruskal 是先找最小边，prim 是从一个顶点找最小边条件是一个点在已找到的一个不在),Djs 最小路径,欧拉最小路径和 transitive closure 很像 O(v3)。Maxflow

Relax along() 是 dist(v)+(w,w)<dist(w)那么就替换

```
dist[]  // array of cost of shortest path from s
pred[]  // array of predecessor in shortest path fror

dijkstraSSSP(G,source):
    Input graph G, source node

    initialise dist[] to all ∞, except dist[source]=0
    initialise pred[] to all -1
    vSet=all vertices of G
    while vSet≠∅ do
        find s∈vSet with minimum dist[s]
        for each (s,t,w)∈edges(G) do
            relax along (s,t,w)
        end for
        vSet=vSet\{s}
    end while
```

Data: *G*, `dist[][]`, `path[][]` Algorithm:

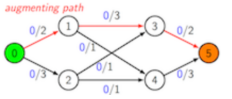
```
dist[][]  // array of cost of shortest path from s to t
path[][]  // array of next node after s on shortest path from s to t

floydAPSP(G):
    Input graph G

    initialise dist[s][t]=0 for each s=t
                         =w for each (s,t,w)∈edges(G)
                         =∞ otherwise
    initialise path[s][t]=t for each (s,t,w)∈edges(G), otherwise to -1
    for all i∈vertices(G) do
        for all s∈vertices(G) do
            for all t∈vertices(G) do
                if dist[s][i]+dist[i][t] < dist[s][t] then
                    dist[s][t]=dist[s][i]+dist[i][t]
                    path[s][t]=path[s][i]
                end if
            end for
        end for
    end for
```

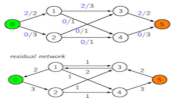
```
maxflow(G):
    Input  flow network G with source s and sink t
    Output maximum flow value

    initialise flow[v][w]=0 for all vertices v, w
    maxflow=0
    while ∃shortest augmenting path visited[] from s to t do
        df = maximum additional flow via visited[]
        // adjust flow so as to represent residual graph
        v=t
        while v≠s do
            flow[visited[v]][v] = flow[visited[v]][v] + df;
            flow[v][visited[v]] = flow[v][visited[v]] - df;
            v=visited[v]
        end while
        maxflow=maxflow+df
    end while
    return maxflow
```

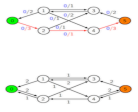
Shortest augmenting path can be found by standard BFS

Maxflow: Augmenting path 是一条可以获得最多 flow 的路径（找最小的限制）eg，flow 是 2。用当前的容量 c-f(v,w),添加另一个方向边容量是是 f(v,w).之后在找新的 Augmenting path 重复这个过程. 最后把箭头反转就是 flow 的流程图。Edmonds 有两个表，一个是 flow 一个是剩余流量

```
KruskalMST(G):
    Input  graph G with n nodes
    Output a minimum spanning tree of G

    MST=empty graph
    sort edges(G) by weight
    for each e∈sortedEdgeList do
        MST = MST ∪ {e}
        if MST has a cyle then
            MST = MST \ {e}
        end if
        if MST has n-1 edges then
            return MST
        end if
    end for
```



### Edmonds-Karp Algorithm

One approach to solving maximum flow problem ...

```
maxflow(G):
```

1. Find a shortest augmenting path
2. Update `flow[][]` so as to represent residual graph
3. Repeat until no augmenting path can be found

$O(v*E2)$

same vertex set $V$

for each edge $v \xrightarrow{c} w \in E$ ...

- $f(v,w) < c$ ⇒ add edge $(v \xrightarrow{c-f(v,w)} w)$ to $E'$
- $f(v,w) > 0$ ⇒ add edge $(v \xleftarrow{f(v,w)} w)$ to $E'$

9. Tree 一般搜索复杂度是 O(logn=height)，design O(logn) n 是节点数量，insert0(n). splay tree 类似 insert 就是插到根但是会出现 double 旋转，每条两次一次性旋转两次，记住标记根

Cost of searching:

| | Array | List | File |
|---|---|---|---|
| Unsorted | O(n) (linear scan) | O(n) (linear scan) | O(n) (linear scan) |
| Sorted | O(log n) (binary search) | O(n) (linear scan) | O(log n) (seek, seek>, ...) |

For binary Trees, several well-defined visiting orders exist:

- *preorder* (NLR) ... visit root, then left subtree, then right subtree
- *inorder* (LNR) ... visit left subtree, then root, then right subtree
- *postorder* (LRN) ... visit left subtree, then right subtree, then root
- *level-order* ... visit root, then all its children, then all their children

```
showBSTreePreorder(t):
  Input tree t

  push t onto new stack S
  while stack is not empty do
    t=pop(S)
    print data(t)
    if right(t) is not empty then
      push right(t) onto S
    end if
    if left(t) is not empty then
      push left(t) onto S
    end if
  end while
```

```
rotateLeft(n2):
  Input  tree n2
  Output n2 rotated to the left

  if n2 is empty or right(n2) is empty then
    return n2
  end if
  n1=right(n2)
  right(n2)=left(n1)
  left(n1)=n2
  return n1
```

```
joinTrees(t1,t2):
  Input  trees t1,t2
  Output t1 and t2 joined together

  if t1 is empty then return t2
  else if t2 is empty then return t1
  else
    curr=t2, parent=NULL
    while left(curr) is not empty do     // find min element in t2
      parent=curr
      curr=left(curr)
    end while
    if parent≠NULL then
      left(parent)=right(curr)  // unlink min element from parent
      right(curr)=t2
    end if
    left(curr)=t1
    return curr                          // curr is new root
  end if
```

```
TreeDelete(t,item):
  Input  tree t, item
  Output t with item deleted

  if t is not empty then          // nothing to do if tree is empty
    if item < data(t) then        // delete item in left subtree
      left(t)=TreeDelete(left(t),item)
    else if item > data(t) then   // delete item in right subtree
      right(t)=TreeDelete(right(t),item)
    else                          // node 't' must be deleted
      if left(t) and right(t) are empty then
        new=empty tree                    // 0 children
      else if left(t) is empty then
        new=right(t)                      // 1 child
      else if right(t) is empty then
        new=left(t)                       // 1 child
      else
        new=joinTrees(left(t),right(t))   // 2 children
      end if
      free memory allocated for t
      t=new
    end if
  end if
  return t
```

... Rebalancing Trees

Implementation of rebalance:

```
rebalance(t):
  Input  tree t with n nodes
  Output t rebalanced

  if n≥3 then
    t=partition(t,⌊n/2⌋)          //
    left(t)=rebalance(left(t))    //
    right(t)=rebalance(right(t))
  end if
  return t
```

```
NewTreeInsert(tree,item):
  Input  tree, item
  Output tree with item randomly inserted

  t=insertAtLeaf(tree,item)
  if #nodes(t) mod k = 0 then
    t=rebalance(t)
  end if
  return t
```

E.g. rebalance after every 20 insertions ⇒ choose *k=20*

... Rebalancing Trees

Implementation of partition operation:

```
partition(tree,i):
  Input  tree with n nodes, index i
  Output tree with i^th item moved to the root

  m=#nodes(left(tree))
  if i < m then
    left(tree)=partition(left(tree),i)
    tree=rotateRight(tree)
  else if i > m then
    right(tree)=partition(right(tree),i-m-1)
    tree=rotateLeft(tree)
  end if
  return tree
```

```
int TreeHeight(Tree t) {
   if (t == NULL) {
      return -1;
   } else {
      int lheight = 1 + TreeHeight(left(t));
      int rheight = 1 + TreeHeight(right(t));
      if (lheight > rheight)
        return lheight;
      else
        return rheight;
   }
}
```

```
TreeTraversal(tree,style):
  Input tree, style of traversal

  if tree is not empty then
    if style="NLR" then
      visit(data(tree))
    end if
    TreeTraversal(left(tree),style)
    if style="LNR" then
      visit(data(tree))
    end if
    TreeTraversal(right(tree),style)
    if style="LRN" then
      visit(data(tree))
    end if
  end if
```

**10. random insert O(logn) worst is O(n), splay tree search O(n) 找到移到根，否则移最接近的, AVL search O(logn) ,234 searche O(heaight = logn), red-black search O(logn)最多 2h 旋转**

```
insertSplay(tree,item):
  Input  tree, item
  Output tree with item splay-inserted

  if tree is empty then return new node containing item
  else if item=data(tree) then return tree
  else if item<data(tree) then
    if left(tree) is empty then
      left(tree)=new node containing item
    else if item<data(left(tree)) then
        // Case 1: left-child of left-child "zig-zig"
      left(left(tree))=insertSplay(left(left(tree)),item)
      tree=rotateRight(tree)
    else if item>data(left(tree)) then
        // Case 2: right-child of left-child "zig-zag"
      right(left(tree))=insertSplay(right(left(tree)),item)
      left(tree)=rotateLeft(left(tree))
    end if
    return rotateRight(tree)
  else    // item>data(tree)
    if right(tree) is empty then
      right(tree)=new node containing item
    else if item<data(right(tree)) then
        // Case 3: left-child of right-child "zag-zig"
      left(right(tree))=insertSplay(left(right(tree)),item)
      right(tree)=rotateRight(right(tree))
    else if item>data(right(tree)) then
        // Case 4: right-child of right-child "zag-zag"
      right(right(tree))=insertSplay(right(right(tree)),item)
      tree=rotateLeft(tree)
    end if
    return rotateLeft(tree)
  end if
```

```
insertAVL(tree,item):
  Input  tree, item
  Output tree with item AVL-inserted

  if tree is empty then
    return new node containing item
  else if item=data(tree) then
    return tree
  else
    if item<data(tree) then
      left(tree)=insertAVL(left(tree),item)
    else if item>data(tree) then
      right(tree)=insertAVL(right(tree),item)
    end if
    if height(left(tree))-height(right(tree)) > 1 then
      if item>data(left(tree)) then
        left(tree)=rotateLeft(left(tree))
      end if
      tree=rotateRight(tree)
    else if height(right(tree))-height(left(tree)) > 1 then
      if item<data(right(tree)) then
        right(tree)=rotateRight(right(tree))
      end if
      tree=rotateLeft(tree)
    end if
    return tree
  end if
```

Searching in 2-3-4 trees:

```
Search(tree,item):
  Input  tree, item
  Output address of item if found in 2-3-4 tree
         NULL otherwise

  if tree is empty then
    return NULL
  else
    i=0
    while i<tree.order-1 ∧ item>tree.data[i] do
      i=i+1   // find relevant slot in data[]
    end while
    if item=tree.data[i] then   // item found
      return address of tree.data[i]
    else      // keep looking in relevant subtree
      return Search(tree.child[i],item)
    end if
  end if
```

```
insert(tree,item):
  Input  2-3-4 tree, item
  Output tree with item inserted

  node=root(tree), parent=NULL
  repeat
    if node.order=4 then
      promote = node.data[1]     // middle value
      nodeL   = new node containing node.data[0]
      nodeR   = new node containing node.data[2]
      if parent=NULL then
        make new 2-node root with promote,nodeL,nodeR
      else
        insert promote,nodeL,nodeR into parent
        increment parent.order
      end if
      node=parent
    end if
    if node is a leaf then
      insert item into node
      increment node.order
    else
      parent=node
      if item<node.data[0] then
        node=node.child[0]
      else if item<node.data[1] then
        node=node.child[1]
      else
        node=node.child[2]
      end if
    end if
  until item inserted
```

... Red-Black Tree Insertion

High-level description of insertion algorithm:

```
insertRB(tree,item,inRight):
  Input  tree, item, inRight indicating direction of last branch
  Output tree with it inserted

  if tree is empty then
    return newNode(item)
  end if
  if left(tree) and right(tree) both are RED then
    split 4-node
  end if
  recursive insert cases (cf. regular BST)
  re-arrange links/colours after insert
  return modified tree

insertRedBlack(tree,item):
  Input  red-black tree, item
  Output tree with item inserted

  tree=insertRB(tree,item,false)
  colour(tree)=BLACK
  return tree
```

```
Tree insertRB(Tree t, Item it, bool inRight) {
   if (t == NULL)
     return newNode(it);
   if (it == data(t))
     return t;
   if ( isRed(left(t)) && isRed(right(t)) ) {
     colour(t) = RED;
     colour(left(t)) = BLACK;
     colour(right(t)) = BLACK;
   }
   if (it < data(t)) {
     left(t) = insertRB(left(t), it, false);
     if (isRed(t) && isRed(left(t)) && inRight) {
       t = rotateRight(t);
     }
     if ( isRed(left(t)) && isRed(left(left(t))) ) {
       t = rotateRight(t);
       colour(t) = BLACK;
       colour(right(t)) = RED;
     }
   } else {
     right(t) = insertRB(right(t), it, true);
     if ( isRed(t) && isRed(right(t)) && !inRight ) {
       t = rotateLeft(t);
     }
     if ( isRed(right(t)) && isRed(right(right(t))) ) {
       t = rotateLeft(t);
       colour(t) = BLACK;
       colour(left(t)) = RED;
     }
   }
   return t;
}

Tree TreeInsert(Tree t, Item it) {
   t = insertRB(t, it, false);
   colour(t) = BLACK;
   return t;
}
```

**split-4: 红黑红 变黑红黑 re-arrange:右红左红用旋转变右红右红，黑右红右红变红黑红**

**11. string: naïve matching O(n*m),Boyer-Moore O(mn+s),KMP O(m+n), Trie: O(d*m) d=26 一般. 红色框是 finish 一个单词，后缀压缩，suffix O(s*m),Huffman O(n+dlogd)**

... Boyer-Moore Algorithm

```
BoyerMooreMatch(T,P,Σ):
  Input  text T of length n, pattern P of length m, alphabet Σ
  Output starting index of a substring of T equal to P
         -1 if no such substring exists

  L=lastOccurenceFunction(P,Σ)
  i=m-1, j=m-1                    // start at end of pattern
  repeat
    if T[i]=P[j] then
      if j=0 then
        return i                  // match found at i
      else
        i=i-1, j=j-1
      end if
    else                          // character-jump
      i=i+m-min(j,1+L[T[i]])
      j=m-1
    end if
  until i≥n
  return -1                       // no match
```

- Biggest jump (m characters ahead) occurs when $L[T[j]] = -1$

Traversing a path, using char-by-char from Key:

```
find(trie,key):
  Input  trie, key
  Output pointer to element in trie if key found
         NULL otherwise

  node=trie
  for each char in key do
    if node.child[char] exists then
      node=node.child[char]  // move down one level
    else
      return NULL
    end if
  end for
  if node.finish then           // "finishing" node reached?
    return node
  else
    return NULL
  end if
```

... Trie Operations

Insertion into Trie:

```
insert(trie,item,key):
  Input  trie, item with key of length m
  Output trie with item inserted

  if trie is empty then
    t=new trie node
  end if
  if m=0 then
    t.finish=true, t.data=item
  else
    t.child[key[0]]=insert(t.child[key[0]],item,key[1..m-1])
  end if
  return t
```

Construction of the failure function is similar to the KMP algorithm itself:

```
failureFunction(P):
  Input  pattern P of length m
  Output failure function for P

  F[0]=0
  i=1, j=0
  while i<m do
    if P[i]=P[j] then   // we have matched j+1 characters
      F[i]=j+1
      i=i+1, j=j+1
    else if j>0 then    // use failure function to shift P
      j=F[j-1]
    else
      F[i]=0            // no match
      i=i+1
    end if
  end while
  return F
```

```
suffixTrieMatch(trie,P):
  Input  compact suffix trie for text T, pattern P of length m
  Output starting index of a substring of T equal to P
         -1 if no such substring exists

  j=0, v=root of trie
  repeat
    // we have matched j+1 characters
    if ∃w∈children(v) such that P[j]=T[start(w)] then
      i=start(w)                  // start(w) is the start index of w
      x=end(w)-i+1                // end(w) is the end index of w
      if m≤x then    // length of suffix ≤ length of the node label?
        if P[j..j+m-1]=T[i..i+m-1] then
          return i-j              // match at i-j
        else
          return -1               // no match
      else if P[j..j+x-1]=T[i..i+x-1] then
        j=j+x, m=m-x              // update suffix start index and length
        v=w                       // move down one level
      else return -1              // no match
      end if
    else
      return -1
    end if
  until v is leaf node
  return -1                       // no match
```

... Knuth-Morris-Pratt Algorithm

```
KMPMatch(T,P):
  Input  text T of length n, pattern P of length m
  Output starting index of a substring of T equal to P
         -1 if no such substring exists

  F=failureFunction(P)
  i=0, j=0                        // start from left
  while i<n do
    if T[i]=P[j] then
      if j=m-1 then
        return i-j                // match found at i-j
      else
        i=i+1, j=j+1
      end if
    else                          // mismatch at P[j]
      if j>0 then
        j=F[j-1]                  // resume comparing P at F[j-1]
      else
        i=i+1
      end if
    end if
  end while
  return -1                       // no match
```

```
HuffmanCode(T):
  Input  string T of size n
  Output optimal encoding tree for T

  compute frequency array
  Q=new priority queue
  for all characters c do
    T=new single-node tree storing c
    join(Q,T) with frequency(c) as key
  end for
  while |Q|≥2 do
    f1=Q.minKey(), T1=leave(Q)
    f2=Q.minKey(), T2=leave(Q)
    T=new tree node with subtrees T1 and T2
    join(Q,T) with f1+f2 as key
  end while
  return leave(Q)
```

**12. random and approximation**

**找 0 值就是 x1*mid>0,让 x1=mid 否则 x2=mid,曲线长度用 a=(end-start)/step 为一个段,之后每段曲线长度是在一段里两点间距离公式，定点覆盖,最少顶点集合 c 覆盖全部边 e.O(v+e) Xn+1=(a*Xn+c)mod m**

**Quicksorted 是选第一个为 low=pivot,left = 最左边 low+1 开始找比 pivot 大的，right 是最右边比 pivot 等于小的。如果 left<right，那么 swap 他们。再循环知道 left>=right.第一个等于 right，right= pivot.再 sorted 左半边右半边.O(n2).随机 quick 是从 low 到 high 随机选取一个 pivot。之后交换第一个与 pivot，其他一样与之前一样。 O(nlogn).**

**Minm cut 压缩是随机选两个点,变成一个点，把所有链接两个点的边连接到新点直到最后剩下两个点为两组 O(E)。Karger 算法是重复压缩 d 次 d=（从 v 个里面选 2 个 Cv2） *lnv，在这之中找最小的 cut. O(e*v2*logv) 。**