Last Section

An Example

- Task
- Problem analyzing
- Statement of the problem
- Development of a Model
- Design of the algorithm
- Correctness of the algorithm
- Implementation
- Analysis and complexity of the algorithm

Statement of the problem

已知网络中任意两点间建立链路的花费，

需建立一个最小费用的通讯网络,

其中任意节点间可以相互通讯。

Model

Let

G=(V,E) be a connected, undirected graph;

w(u,v) is the cost for connecting *u,v*$\in$*V*;

find an acyclic subset T $\subseteq$ E and connects all vertices in *V*, such that

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$ is minimized

- **Algorithm A** To find a minimum-weight spanning tree T in a weighted, connected network G with N vertices and M edges.

```
Step 0.    [Initialize] Set T ← a network consisting of N
                vertices and no edges;
                set   H ← G.
Step 1.    [Iterate] While T is not a connected network
                do through step 3 od; STOP.
Step 2.    [Pick a lightest edge] Let (U,V) be a lightest
           (cheapest) edge in H;
           if T + (U,V)  has no cycles
           then [add (U,V) to T] set T ← T + (U,V) fi.
Step 3.    [Delete (U, V) from H] Set H ← H – (U, V).
```

Does it work?

- There are several questions we should ask about this algorithm;

  1. Does it always STOP?
  2. When it STOPs, is T always a spanning tree of G?
  3. Is T guaranteed to be a minimum-weight spanning tree?
  4. Is it self-contained (or does it contain hidden, or implicit, sub-algorithms)?
  5. Is it efficient?

Question 4&5

- Step 1 requires a sub-algorithm to determine if a network is connected, and
- step 2 requires a sub-algorithm to decide if a network has a cycle.
- These two steps can make the algorithm ineffective in that these sub-algorithms might be time-consuming.

- **Algorithm B** To find a minimum-weight spanning tree T in a weighted, connected network G with N vertices and M edges.

Step 0. [Initialize] Label all vertices "unchosen"; **set** T ← a network with N vertices and no edges; choose an arbitrary vertex and label it "chosen".

Step 1. [Iterate] **While** there is an unchosen vertex **do** step 2 **od**; STOP.

Step 2. [Pick a lightest edge]

Let (U, V) be a lightest edge between any chosen vertex U and any unchosen vertex V; label V as "chosen"; **and set** T ← T + (U, V).

Now

- Algorithm B is self-contained;
- It is assumed to be more efficient than Algorithm A.

Does it work?

- There are several questions we should ask about this algorithm;

1. Does it always STOP?
2. When it STOPs, is T always a spanning tree of G?
3. Is T guaranteed to be a minimum-weight spanning tree?
4. Is it self-contained (or does it contain hidden, or implicit, sub-algorithms)?
5. Is it efficient?

Complete Development of an Algo.

- 1 Statement of the problem
- 2 Development of a Model
- 3 Design of the algorithm
- 4 Correctness of the algorithm
- 5 Implementation
- 6 Analysis and complexity of the algorithm
- 7 Program testing
- 8 Documentation

Complete Development of an Algo.

- 1 Statement of the problem
- 2 Development of a Model
- 3 Design of the algorithm
- 4 Correctness of the algorithm
- 5 Implementation
- 6 Analysis and complexity of the algorithm
- 7 Program testing
- 8 Documentation

step 2的一个效率更高的实现:

- 保存 未选择节点的一个列表:
  - UNCHSN (I), 初始包括 NUMUN = M – 1 个节点,
- 创建两个额外的数组:
  - LIGHT(I), 等于i*th* 未选择节点 和已选节点之间的最轻的边的费用值
  - VERTEX(I), 该条边上的已选择节点

Flow Chart of Algorithm C

1. Start
2. Initialization
3. Update lightest edge from each unchosen vertex to a chosen vertex
4. Pick a lightest edge e from an unchosen vertex to a chosen vertex
5. Add edge e to MST
6. Delete newly chosen vertex from unchosen list
7. IF No unchosen vertices left THEN Stop
8. GOTO 3

Complete Development of an Algo.

- 1 Statement of the problem
- 2 Development of a Model
- 3 Design of the algorithm
- 4 Correctness of the algorithm
- 5 Implementation
- 6 Analysis and complexity of the algorithm
- 7 Program testing
- 8 Documentation

Analysis of Algorithm C

- Correctness
- Time complexity
  - A rough analysis----$O(N^2)$
  - A detailed analysis
- Simplicity
- Optimality

During your implementation of

Algorithm C for MST

Something Happened

- You found another program runs faster than yours.

- You can’t believe it!
- Prim’s Algo is optimal!

- You are given the algorithm:

```
1.  $A \leftarrow \phi$
2.  **for** each vertex $v \in V[G]$
3.      **do** MAKE-SET ($v$)
4.  sort the edges of $E$ by nondecreasing weight $w$
5.  **for** each edge $(u, v) \in E$, in order by
    nondecreasing weight
6.      **do if** FIND-SET ($u$) $\neq$ FIND-SET ($v$)
7.            **then** $A \leftarrow A \cup \{ (u, v)\}$
8.                  UNION($u$, $v$)
9.  **Return** $A$
```

KRUSKAL(G, w) _Disjoint Set

```
1. $A \leftarrow \phi$
2. for each vertex $v \in V[G]$
3.     do MAKE-SET ($v$)
4. sort the edges of $E$ by nondecreasing weight $w$
5. for each edge $(u, v) \in E$, in order by
   nondecreasing weight
6.      do if FIND-SET ($u$) $\neq$ FIND-SET ($v$)
7.             then $A \leftarrow A \cup \{(u, v)\}$
8.                  UNION($u$, $v$)
9. Return $A$
```

*

Disjoint Set（分离 ;分立）

A ***disjoint-set data structure*** maintains

a collection（群） $S = \{S_1, S_2, \ldots, S_k\}$

of disjoint dynamic sets.

Each set is identified by a ***representative***, which is some member of the set.

Each element of a set is represented by an Object.

In some applications, it doesn't matter which member is used as the representative; we only care that if we ask for the representative of a dynamic set twice without modifying the set between the requests, we get the same answer both times.

In other applications, there may be a prespecified rule for choosing the representative.

Operations on object *x*

- MAKE-SET(*x*) creates a new set whose only member (and thus representative) is pointed to by *x*.

- Union(*x*, *y*) unites the dynamic sets that contain *x* and *y*, say *S*x and *S*y, into a new set that is the union of these two sets. The two sets are assumed to be disjoint prior to the operation. The representative of the resulting set is some member of *S*x ∪ *S*y, although many implementations of UNION choose the representative of either *S*x or *S*y as the new representative. Since we require the sets in the collection to be disjoint, we "destroy" sets *S*x and *S*y, removing them from the collection *S*.

- FIND-SET(*x*) returns a pointer to the representative of the (unique) set containing *x*.

KRUSKAL(*G*, *w*) _Disjoint Set

```
1. $A \leftarrow \phi$
2. for each vertex $v \in V[G]$
3.     do MAKE-SET ($v$)
4. sort the edges of $E$ by nondecreasing weight $w$
5. for each edge $(u, v) \in E$, in order by
   nondecreasing weight
6.      do if FIND-SET ($u$) $\neq$ FIND-SET ($v$)
7.               then $A \leftarrow A \cup \{ (u, v)\}$
8.                     UNION($u$, $v$)
9. Return $A$
```

*

Operations on object *x*

- MAKE-SET(*x*) creates a new set whose only member (and thus representative) is pointed to by *x*.
- Union(*x*, *y*) unites the dynamic sets that contain *x* and *y*, say *S*x and *S*y, into a new set that is the union of these two sets. The two sets are assumed to be disjoint prior to the operation. The representative of the resulting set is some member of *S*x ∪ *S*y, although many implementations of UNION choose the representative of either *S*x or *S*y as the new representative. Since we require the sets in the collection to be disjoint, we "destroy" sets *S*x and *S*y, removing them from the collection *S*.
- FIND-SET(*x*) returns a pointer to the representative of the (unique) set containing *x*.

- *n*, the number of MAKE-SET operations, and
- *m*, the total number of MAKE-SET, UNION, and FIND-SET operations.

Linked-list representation of disjoint sets

- A simple way to implement a disjoint-set data structure is to represent each set by a linked list.
- The first object in each linked list serves as its set's representative.

- Each object contains :
  - A set member
  - A pointer to the object containing the next set member
  - A pointer back to the representative.

Linked-list representation of disjoint sets

- MAKE-SET(x): $O(1)$
- FIND-SET(x): $O(1)$
- UNION:

  Operation UNION(X1,X2),(X2,X3)…(X_{q-1},X_q)

  Number of objects updated: 1,2,3,…*q-1*

  q=m-n; worst: n+q^2*;* $O(\mathrm{m}^2)$

  A sequence of m operations requires Θ(m^2).

- Weighted-union :
  - Each representative also includes the length of the list;
  - And always append the smaller list onto the longer.

Weighted-union

- **Theorem**

  Using the linked-list representation of disjoint sets and

  the weighted-union heuristic,

  a sequence of *m* MAKE-SET, UNION, and FIND-SET operations, *n* of which are MAKE-SET operations,

  takes $O(m+nlgn)$ time.

Proof

Consider a fixed object *x*. We know that each time *x*'s representative pointer was updated, *x* must have started in the smaller set.

The first time *x*'s representative pointer was updated, the resulting set must have had at least 2 members.

Similarly, the next time *x*'s representative pointer was updated, the resulting set must have had at least 4 members. Continuing on, we observe that for any $k \le n$, after *x*'s representative pointer has been updated $\lceil \lg k \rceil$ times, the resulting set must have at least *k* members.

(suppose updated every union)

Since the largest set has at most *n* members, each object's representative pointer has been updated at most $\lceil \lg n \rceil$ times over all the UNION operations. The total time used in updating the *n* objects is thus $O(n \lg n)$.

Weighted-union

- **Theorem**

  Using the linked-list representation of disjoint sets and

  the weighted-union heuristic,

  a sequence of *m* MAKE-SET, UNION, and FIND-SET operations, *n* of which are MAKE-SET operations,

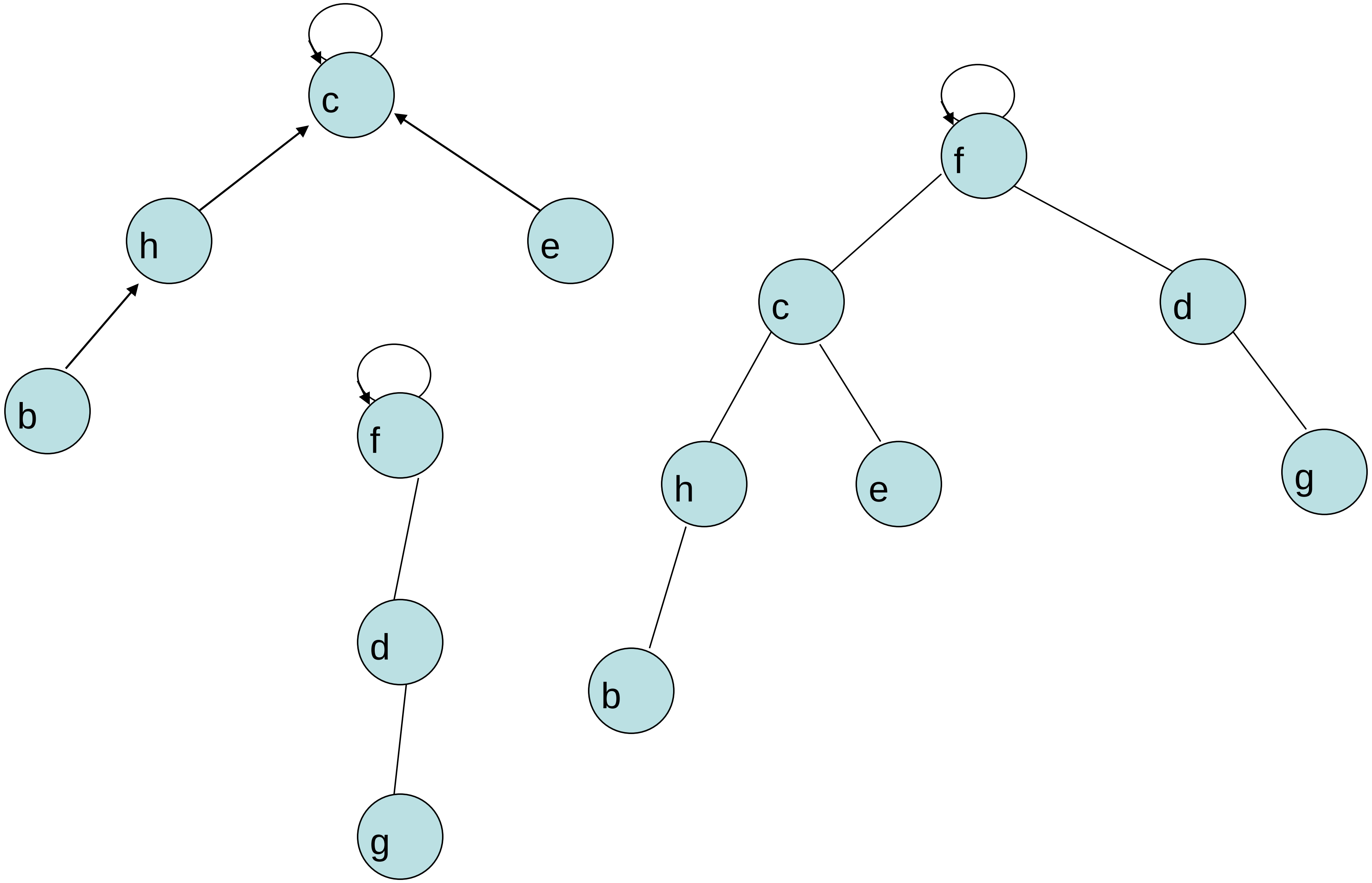
  takes $O(m+n\lg n)$ time.

Disjoint-set Forest

- In a faster implementation of disjoint sets, we represent sets by rooted trees, with each node containing one member and each tree representing one set.

- In a ***disjoint-set forest***, each member points only to its parent. The root of each tree contains the representative and is its own parent.

Disjoint-set Forest



Operations of Disjoint-set Forest

- A MAKE-SET operation simply creates a tree with just one node.
- We perform a FIND-SET operation by chasing parent pointers until we find the root of the tree. (The nodes visited on this path toward the root constitute the ***find path***.)
- A UNION operation causes the root of one tree to point to the root of the other.

Operations of Disjoint-set Forest

- ***Union by rank***

  We maintain a ***rank*** that approximates the logarithm of the subtree size and is also an upper bound on the height of the node. In union by rank, the root with smaller rank is made to point to the root with larger rank during a UNION operation.

- ***Path compression***

  We use it during FIND-SET operations to make each node on the **find path** point directly to the root. Path compression does not change any ranks.

Operations of Disjoint-set Forest

```
MAKE-SET (x)
1. p[x] ← x
2. rank [x] ← 0
```

```
FIND-SET (x)
1. if x ≠ p [x]
2. then p[x] ← FIND-SET(p[x])
3. return p[x]
```

FIND-SET is a two-pass method

pass up the find path to find the root

pass back down the find path to update each node (so that it point directly to the root)

Operations of Disjoint-set Forest

```
UNION ($x$, $y$)
1.  LINK (FIND-SET ($x$), FIND-SET($y$))

LINK ($x$, $y$)
1. if $rank[x] > rank[y]$
2.     then $p[y] \leftarrow x$
3.     else $p[x] \leftarrow y$
4.             if $rank[x] = rank[y]$
5.             then $rank[y] \leftarrow rank[y] + 1$
```

Efficiency

- The efficiency of Disjoint-set Forest with union-by-rank and path compression:

$$o(m\alpha(m,n))$$

$\alpha(m,n)$ is the inverse of Ackermann’s function <=4

KRUSKAL(G, w) _Disjoint Set

```
1. $A \leftarrow \phi$
2. for each vertex $v \in V[G]$
3.     do MAKE-SET ($v$)
4. sort the edges of $E$ by nondecreasing weight $w$
5. for each edge $(u, v) \in E$, in order by
   nondecreasing weight
6.     do if FIND-SET ($u$) $\neq$ FIND-SET ($v$)
7.            then $A \leftarrow A \cup \{(u, v)\}$
8.                 UNION($u$, $v$)
9. Return $A$
```

Complexity

- The disjoint-set-forest implementation with the union-by-rank and path-compression heuristics is the asymptotically fastest implementation known.
- Initialization takes time $O(V)$, and the time to sort the edges in line 4 is $O(E \lg E)$.
- There are $O(E)$ operations on the disjoint-set forest, which in total take $O(E\ \alpha(E,V))$ time, where $\alpha(E,V)$ is the functional inverse of Ackermann's function. Since $\alpha(E,V) = O(\lg E)$, the total running time of Kruskal's algorithm is $O(E \lg E)$.

Which is faster?

- $O(M\lg(M))$ vs. $O(N^2)$
  $O(E\lg(E))$ vs. $O(V^2)$

Can we improve algo. C?

Flow Chart of Algorithm C

1. Start
2. Initialization
3. Update lightest edge from each unchosen vertex to a chosen vertex
4. Pick a lightest edge from an unchosen vertex to a chosen vertex
5. Add edge *e* to MST
6. Delete newly chosen vertex from unchosen list
7. IF No unchosen vertices left THEN Stop
8. GOTO 3

Can we improve algo. C?

- The key to implementing Prim’s algorithm efficiently is to make it easy to select a new edge to be added to the tree formed by the edges in *A*(partial solution edge set).

Another Implementation of Prim's

```
Algorithm D $(G, w, r)$
•   $Q \leftarrow V[G]$
•   for each $u \in Q$
•       do $key[u] \leftarrow \infty$
•   $key[r] \leftarrow 0$
•   $\pi[r] \leftarrow$ NIL
1.  while $Q \neq \Phi$
2.      do u $\leftarrow$ EXTRACT – MIN($Q$)
3.          for each $v \in Adj[u]$
4.              do if $v \in Q$ and $w(u, v) < key[v]$
5.                  then $\pi[v] \leftarrow u$
6.                       $key[v] \leftarrow w[u, v]$
```

Definition

- root *r* ,
- all vertices that are *not* in the tree reside in a priority queue *Q* based on a *key* field.
- For each vertex *v*, *key*[*v*] is the minimum weight of any edge connecting *v* to a vertex in the tree;
- The field $\pi[v]$ names the "parent" of *v* in the tree. During the algorithm, the set *A* is kept implicitly as

$$A = \{\ (v, \pi[v]) : v \in V - \{r\} - Q\ \}$$

- When the algorithm terminates, the priority queue Q is empty; the minimum spanning tree *A* for *G* is thus

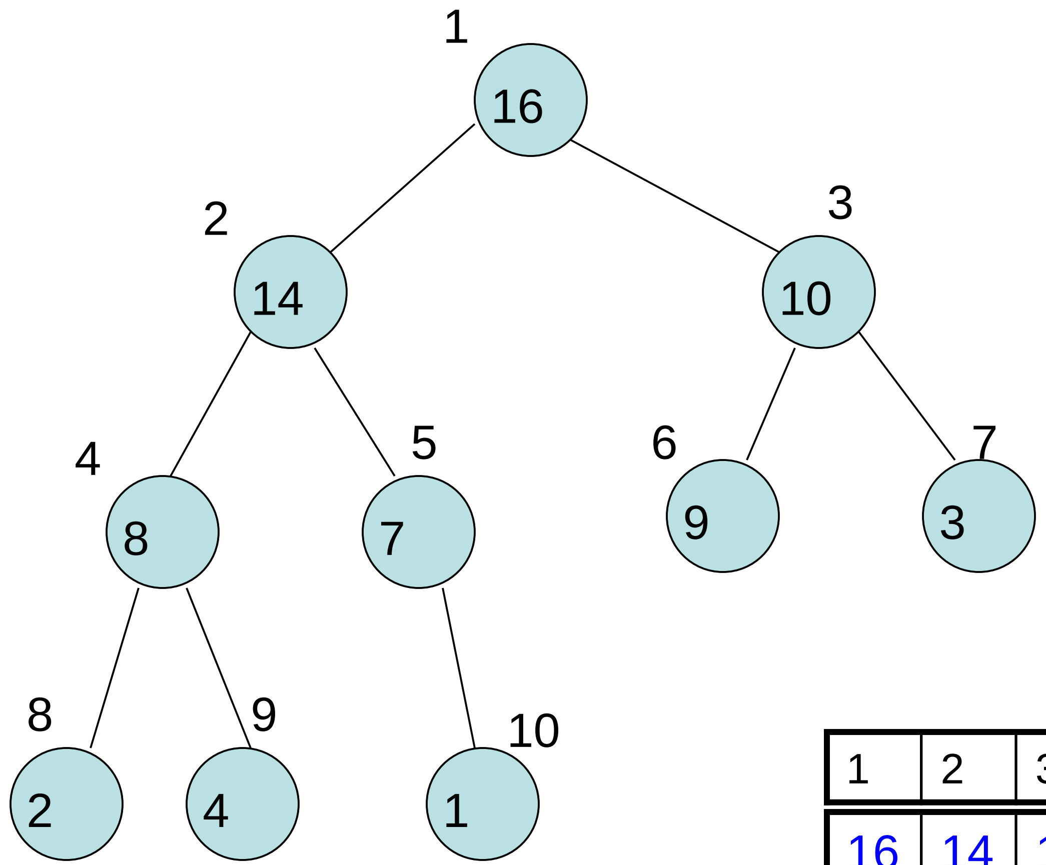
$$A = \{\ (v, \pi[v]) : v \in V - \{r\}\ \}$$

Another Implementation of Prim's

```
Algorithm D ($G$, $w$, $r$)
•   $Q \leftarrow V[G]$
•   for each $u \in Q$
•       do $key[u] \leftarrow \infty$
•   $key[r] \leftarrow 0$
•   $\pi[r] \leftarrow$ NIL
1.  while $Q \neq \Phi$
2.      do u $\leftarrow$ EXTRACT – MIN($Q$)
3.          for each $v \in Adj[u]$
4.              do if $v \in Q$ and $w(u, v) < key[v]$
5.                  then $\pi[v] \leftarrow u$
6.                      $key[v] \leftarrow w[u, v]$
```

The performance of Prim’s algorithm depends on how we implement the priority queue *Q*.

Binary Heap

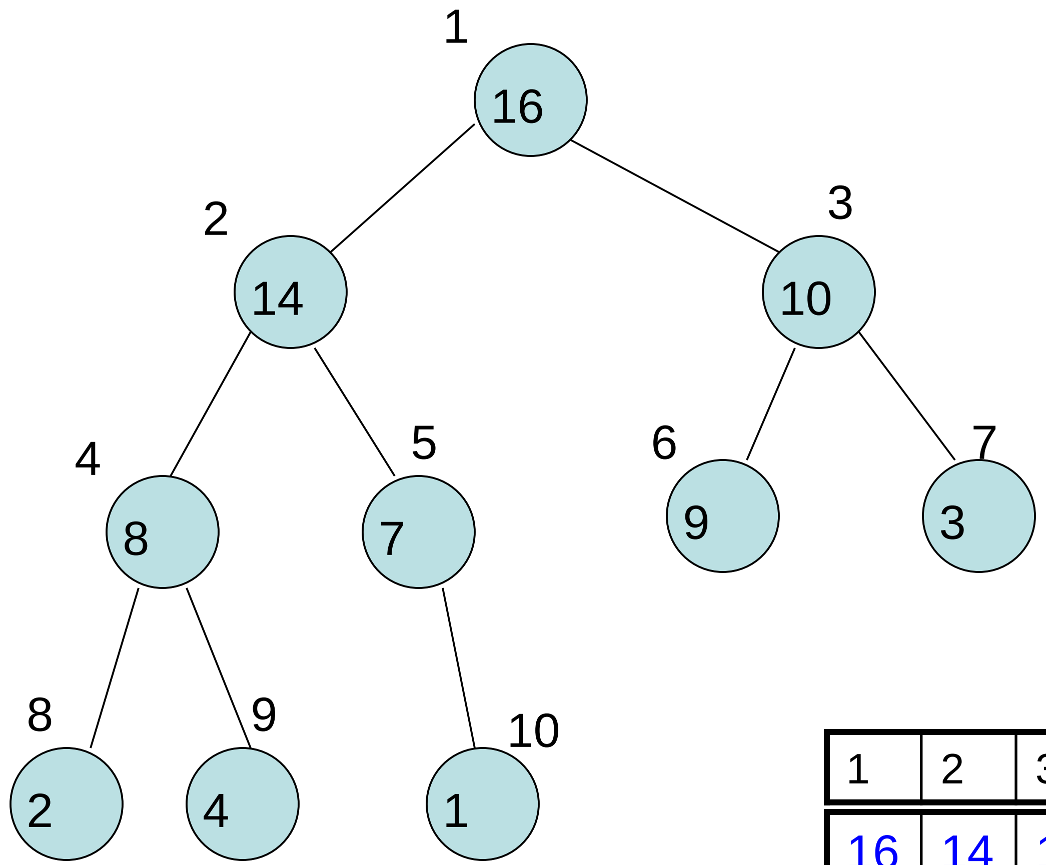


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Binary Heap

- The ***binary heap*** data structure is an array object that can be viewed as a complete binary tree.
- Each node of the tree corresponds to an element of the array that stores the value in the node.
- The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.
- An array *A* that represents a heap is an object with two attributes: *length*[*A*], which is the number of elements in the array, and *heap-size*[*A*], the number of elements in the heap stored within array *A*.
- That is, although *A*[1..*length*[*A*]] may contain valid numbers, no element past *A*[*heap-size*[*A*]], where *heap-size*[*A*] ≤ *length*[*A*], is an element of the heap.

Binary Heap



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

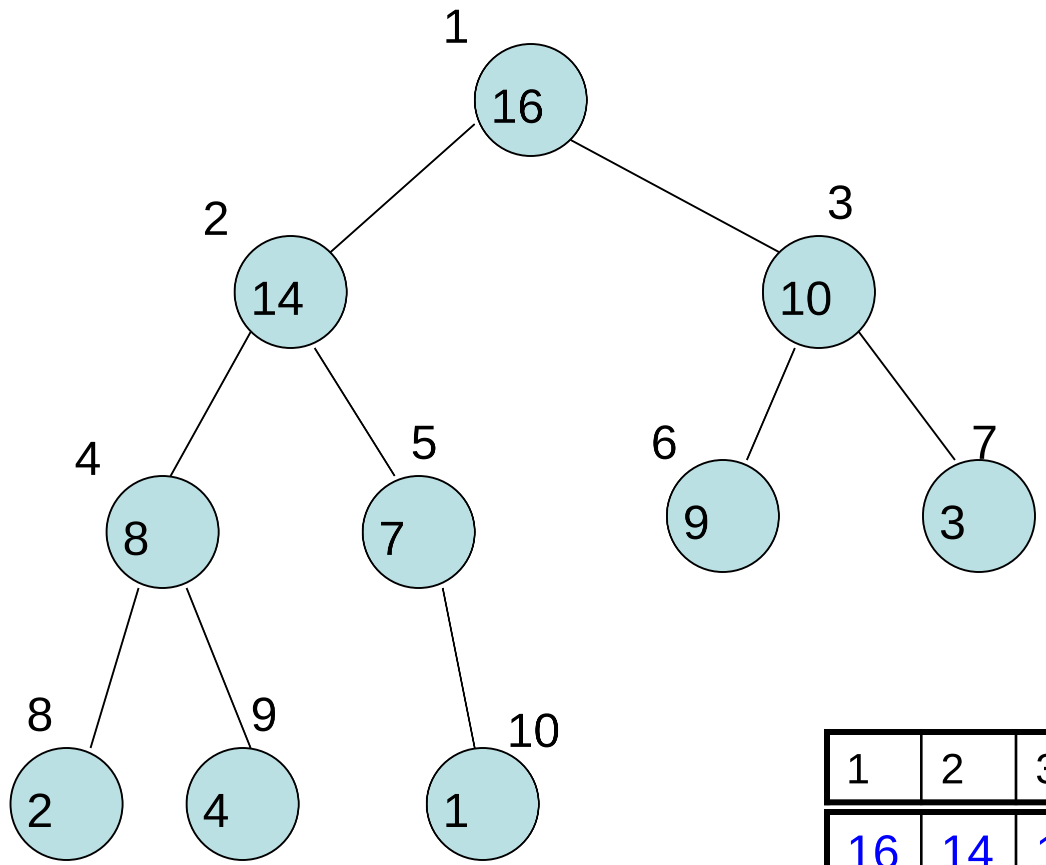
Heap Properties

- The root of the tree is *A*[1], and given the index *i* of a node, the indices of its parent PARENT(*i*), left child LEFT(*i*), and right child RIGHT(*i*) can be computed simply:
  - PARENT(*i*) returns $\lfloor i/2 \rfloor$
  - LEFT(*i*) returns 2i
  - RIGHT(*i*) returns 2i+1

- For every node *i* other than the root

  $A[\text{PARENT}(i)] \geq A[\,i\,]$

Binary Heap



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Efficiency

- We define the ***height*** of a node in a tree to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the tree to be the height of its root.
- Since a heap of *n* elements is based on a complete binary tree, its height is *lg(n).*
- We shall see that the basic operations on heaps run in time at most proportional to the height of the tree and thus take *O* (lg *n*) time.

Important Subroutine

```
HEAPIFY (A, i )
•   l ← LEFT(i)
•   r ← RIGHT (i)
•   if l ≤ heap-size[A] and A[ l ] > A[ i ]
•       then largest ← l
•       else largest ← i
•   if r ≤ heap-size[A] and A[r] > A[largest]
•       then largest ← r
•   if largest ≠ i
•       then exchange A[ i ] & A[largest]
•               HEAPIFY(A, largest)
```

HEAPIFY

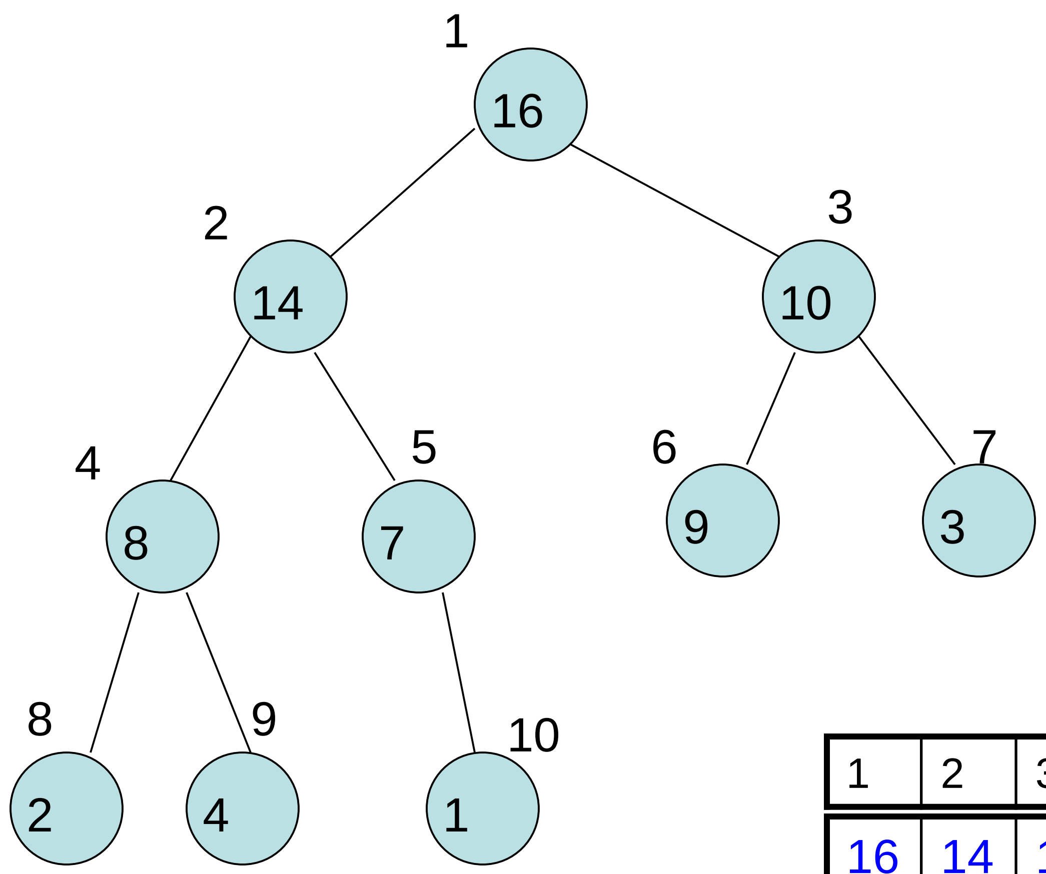
- Assumed :
  - The subtrees rooted at LEFT(*i*) and RIGHT(*i*) are heaps;
  - *A[i]* may be smaller than its children

- HEAPIFY is to let the value at *A[i]* float down in the heap, so that the tree rooted at *i* becomes a heap.

Important Subroutine

```
HEAPIFY (A, i )
•   l ← LEFT(i)
•   r ← RIGHT (i)
•   if l ≤ heap-size[A] and A[ l ] > A[ i ]
•       then largest ← l
•       else largest ← i
•   if r ≤ heap-size[A] and A[r] > A[largest]
•       then largest ← r
•   if largest ≠ i
•       then exchange A[ i ] & A[largest]
•               HEAPIFY(A, largest)
```

Binary Heap



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Running time of HEAPIFY

- $T(n)<T(2n/3)+\Theta(1)$

- According to Master Theorem,

  $T(n)= \Theta(\lg n)$.

*

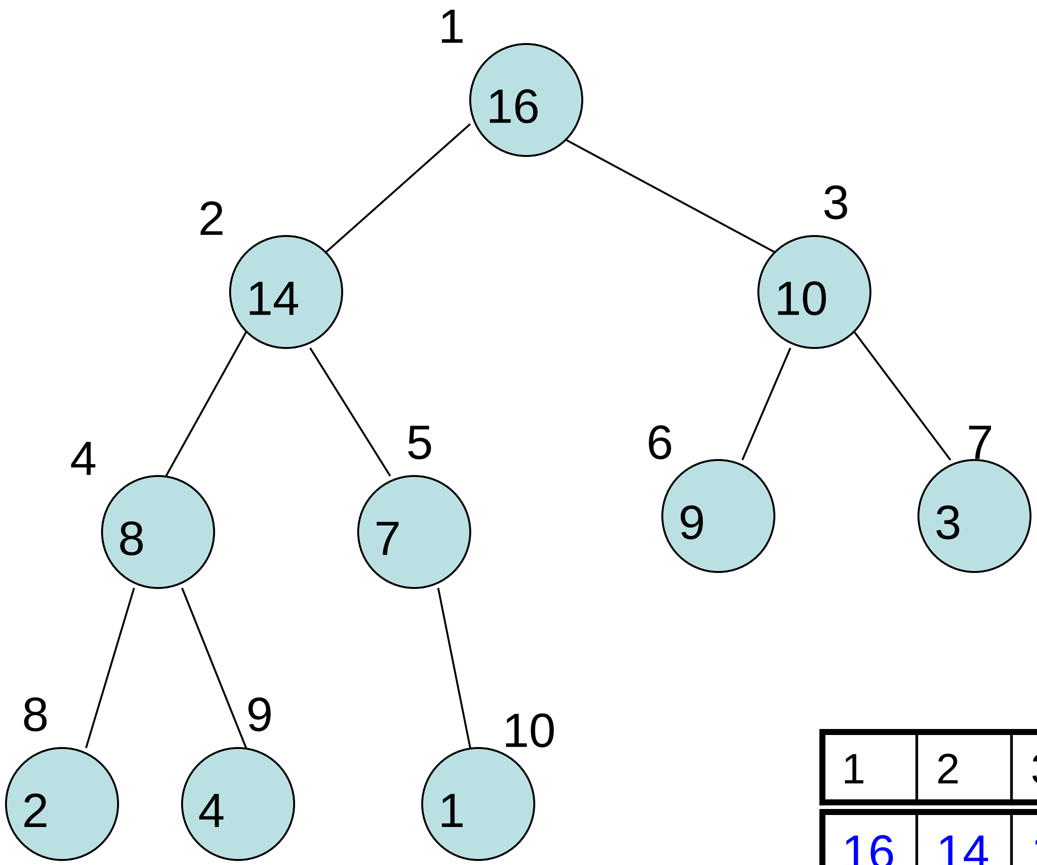
Important Subroutine

```
HEAPIFY ($A$, $i$)
•   $l \leftarrow$ LEFT($i$)
•   $r \leftarrow$ RIGHT ($i$)
•   **if** $l \leq$ *heap-size*[$A$] and $A[\,l\,] > A[\,i\,]$
•       **then** *largest* $\leftarrow l$
•       **else** *largest* $\leftarrow i$
•   **if** $r \leq$ *heap-size*[A] and $A[r] > A[largest]$
•       **then** *largest* $\leftarrow r$
•   **if** *largest* $\neq i$
•       **then** exchange $A[\,i\,]$ & $A[largest]$
•               HEAPIFY($A$, *largest*)
```

Building a heap

```
BUILD-HEAP (A)
•   heap-size[A] ← length[A]
•   for i ← $\lfloor length[A]/2 \rfloor$ downto 1
•       do HEAPIFY (A, i )
```

Binary Heap



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Building a heap

```
BUILD-HEAP (A)
•   heap-size[A] ← length[A]
•   for i ← $\lfloor length[A]/2 \rfloor$ downto 1
•       do HEAPIFY (A, i )
```

O(nlgn)

but the height of a node varies, and the heights of most nodes are small

n-element heap, at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes at height h

thus *O(n)*

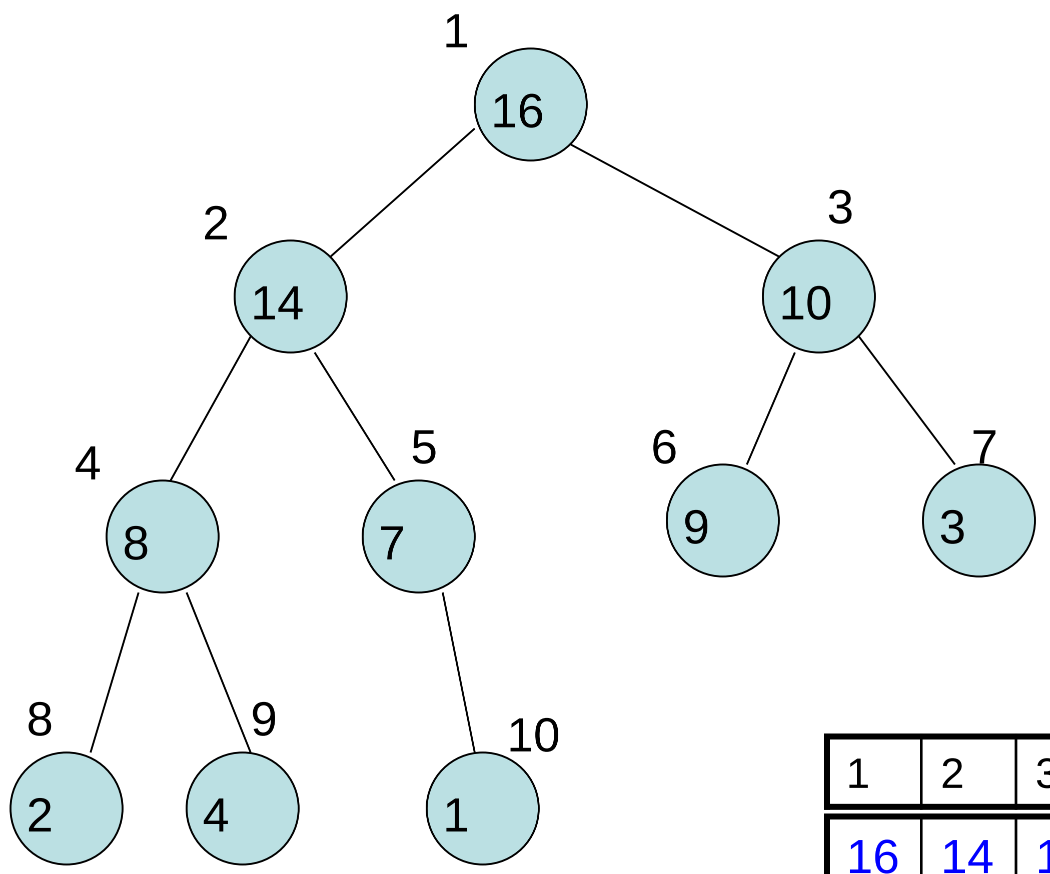
Operation on a heap

HEAP-EXTRACT-MAX (A)

```
•   if heap-size [A] < 1
•      then error "heap underflow"
•   max ← A[1]
•   A[1] ← A[heap-size [A]]
•   heap-size [A] ← heap-size [A] -1
•   HEAPIFY (A, 1)
•   return max
```

O(lgn)

Binary Heap



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Another Implementation of Prim's

```
Algorithm D ($G$, $w$, $r$)
•   $Q \leftarrow V[G]$              // • • • • build heap
•   for each $u \in Q$
•        do $key[u] \leftarrow \infty$
•   $key[r] \leftarrow 0$
•   $\pi[r] \leftarrow$ NIL
1.  while $Q \neq \Phi$
2.      do u $\leftarrow$ EXTRACT – MIN($Q$)          // O(V)O(lgV),
3.           for each $v \in Adj[u]$             //2E
4.               do if $v \in Q$ and $w(u, v) < key[v]$
5.                    then $\pi[v] \leftarrow u$
6.                         $key[v] \leftarrow w[u, v]$
```

O(V), O(V)O(lgV), EO(lgV).

The complexity of Algorithm D is

$O(E \lg V)$

If Fibonacci heap , $O(E+V \lg V)$

Complete Development of an Algo.

- 1 Statement of the problem
- 2 Development of a Model
- 3 Design of the algorithm
- 4 Correctness of the algorithm
- 5 Implementation
- 6 Analysis and complexity of the algorithm
- 7 Program testing
- 8 Documentation

Program Testing

(1) for correctness,
(2) for implementation efficiency, and
(3) for computational complexity.

Taken together, these tests try to provide experimental answers to the questions:

- Does the algorithm work?
- How well does it work?

Testing correctness

- Test it so that it can be used confidently.
- Several ideas that should be part of any testing plan:
  - **Preventive measures (design)**
  - **Spot checks**
  - **Testing all parts of the program**

Test data

- A serious effort should be made to obtain good data for early testing.
  - The initial collection of test data must cover a broad range of cases and correct answers should be known.
  - Some "nonsense" data might also be run to see if such data are handled correctly.
  - Avoid running data that essentially require the program to do the same thing over and over.
  - Test the program on data at the extremes of the input range.
  - Test the program for data that are outside the acceptable input range.
  - If another program is available which solves the same problem, use it as a check against the one being tested. Although it is unlikely, keep in mind that it is possible for both programs to be wrong.

Looking for trouble

- Try everything you can think of to force a program into an error.
- If possible, get someone else to do this for you.
- An unbiased and fresh point of view might be just what is needed to locate errors that you have missed.

Correctness Cont.

- **Time for testing** Testing usually requires a considerable amount of time and should not be rushed. Budget time for it.
- **Retesting** If errors are found during testing, changes must be made to correct them.

- **When to stop testing** There is no general answer to this question. For any particular program, the answer depends on its potential use and the level of confidence desired by the programmer.

Testing Implementation Efficiency

- This kind of testing is concerned with finding ways to make programs run faster or use less storage.
- It is a waste of time for simple programs or programs that will rarely be used, bur it is highly recommended for programs that are likely to be run frequently.

Testing Implementation Efficiency

- Arithmetic operations
- Redundant calculations
  - If a complicated expression is used in more than one place, compute it once and assign it to a variable.
- Order in logical expressions
- Looping elimination
- Loop unrolling
- Inner loop streamlining

Testing Implementation Efficiency

- **Execution profiles**
  - Declare, initialize, range, number of statements

  - The program may be monitored with an assortment of timers and counters, and an output listing is given which shows how many times each statement was executed and the total number of time units spent on each statement.
  - Execution profiles are not only useful for testing implementation efficiency.

Testing Computational Complexity

- It is difficult to test the computational complexity of most algorithms seriously and convincingly.

- There are two major problems:
  - the choice of input data
    - random, plot curve, average and worst…
  - the translation of experimental results into empirical complexity curves
    - Peremeters>=3?,

Complete Development of an Algo.

- 1 Statement of the problem
- 2 Development of a Model
- 3 Design of the algorithm
- 4 Correctness of the algorithm
- 5 Implementation
- 6 Analysis and complexity of the algorithm
- 7 Program testing
- 8 Documentation

Documentation

General Discussion

Situations like the following are common:

A programmer’s first assignment on the new job is to update an important program whose author is no longer present.

The primary purpose of documentation is to help a reader, or even the author, understand a computer program.

For all but the simplest programs, code must be supplemented by both

(1) external documentation and

(2) program documentation.

External Documentation

- In essence, external documentation is anything about the program that is not in the program itself. This is often collected in the form of a user’s manual or technical report.
- Depending on the size and complexity of the program, external documentation can take a variety of forms. Some of these include

(1) flowcharts

(2) user instructions

(3) sample input and output

(4) complete record of the top-down development

(5) verbal statement of the algorithm

(6) references to sources of information

(7) directories of variable and routine names and their roles in the program

(8) discussion of various special features (for example, results of testing, limitations, history, and mathematical background).

Program Documentation

- We do not propose to offer any hard and fast rules for program documentation. There are no such rules. However, we shall present a number of guidelines which are used by many good programmers.
- **Comments should be correct and informative** Although this guideline is obvious, it is often violated. Consider the following code segment:

```
C       CHECK IF K IS LARGER THAN L;
C       THEN GO TO 150
```

- **Write comments as you write the code**
- **Mnemonic names should be used for variables and subroutines** Variable and subroutine names should suggest their function in the algorithm.

Program Documentation

- **Start every program with a prologue** Give the reader useful information at the beginning of every program. The prologue might contain:
- A good descriptive title, author identification, date of last revision, etc.
- A brief description of what the program does, and possibly how it does it (or at least give references)
- A list of all subroutines used, with a brief description of the purpose of each
- A list and description of all variables whose function is not immediately obvious
- Information on error returns and defaults
- Information on input required and output produced

Program Documentation

- **Do not over document** Do not use so much documentation that it is hard to find the program.
- **Avoid bad or overly clever code** The best documentation is clear code. Bad code and overly clever code breed errors and require extra documentation. Do not try to compensate for such code with many comments; rewrite the code.
- **Aim for neatness** It is easier to read clear, well-spaced code than it is to try to work through a congested program.

  Order variable and array declarations, either in some logical sequence or alphabetically. This makes it easier to check whether all variables have been declared, and whether their sizes and types are correct.

  Indent segments of code to reflect levels of logic in a program.

Program Documentation

- **Avoid nonstandard features**
- **Properly define all input and output**
- **Maintenance**
- Many programmers believe that no large program is ever fully tested or free of error.
- There are a variety of other reasons why programs need to be updated, upgraded, or otherwise altered in time. For example, it may be necessary to change the values of program variables, to increase the size of the input that a program can accept, to calculated and output some additional statistical information, to substitute a more efficient algorithm for an existing one, or to recode a program entirely, either for a new machine or system configuration, or in a different programming language.
- This activity falls under the heading of *program maintenance.*
- An appendix should be added to any external documentation. This appendix should thoroughly describe the maintenance operation, including such information as date of changes, reasons for making them, etc.

Divide and Conquer

分治

基本概念

- 分而治之
  - 将原始问题分解为若干子问题，在逐个解决各个子问题的基础上，得到原始问题的解。

- e.g.
  - 图书馆找书
  - 学校评三好学生

基本概念

- 根据如何由分解出的子问题得出原始问题的解，分治策略可分为两种情形：
  - 原始问题的解只存在于分解出的某一个（或某几个）子问题中，则只需要在这一（或这几个）子问题中求解即可；

    图书馆找书

  - 原始问题的解需要由各个子问题的解再经过综合处理得到。

    学校评三好学生

基本概念

- 适当运用分治策略往往可以较快地缩小问题求解的范围，从而加快问题求解的速度。

- 分治策略运用于计算机算法时，往往会出现分解出来的子问题与原始问题类型相同的现象；

  而与原始问题相比，各个子问题的尺寸变小了。这刚好符合递归的特性。

  因此，计算机算法中的分治策略往往与递归联系在一起。

一个问题

- 在一个整数组 $A[1\ldots n]$中，同时寻找最大值和最小值。一种直接的算法如下面所示，它返回一个数对（x, y）, 其中 x 是最小值，y 是最大值。

```
1.  $x \leftarrow A[1]$; $y \leftarrow A[1]$
2.  for $i \leftarrow 2$ to $n$
3.          if $A[i] < x$ then $x \leftarrow A[i]$
4.          if $A[i] > y$ then $y \leftarrow A[i]$
5.  end for
6.  return $(x, y)$
```

- 元素的比较次数是 $2n – 2$。

利用分治策略

- 将数组分割成两半：
  A [1… n/2] 和A [(n/2) +1… n].
  （设 n 为2的幂 ）

- 在每一半中找到最大值和最小值，并返回这两个最小值中的最小值及这两个最大值中的最大值。

MINMAX 算法描述

输入：*n* 个整数元素的数组*A*[1… *n*], *n* 为2的幂。
输出：（*x*, *y*）, *A* 中的最小元素和最大元素。

- *minmax* (1, *n*)

```
Procedure minmax (low, high)
1.    if high – low = 1 then
2.        if A [low] < A [high] then return (A [low], A[high])
3.        else return ( A[high], A [low])
4.        end if
5.    else
6.        mid ← $\lfloor (low + high)/2 \rfloor$
7.        (x1, y1) ← minmax (low, mid)
8.        (x2, y2) ← minmax (mid + 1, high)
9.        x ← min {x1, x2}
10.       y ← max {y1, y2}
11.       return (x, y)
12.   end if
```

MINMAX 效率

- 设 C (n) 表示算法在由n 个元素（n 是2的整数幂）组成的数组上执行的元素比较次数。
- 元素的比较发生于步骤2，9 和10
- 步骤7和步骤8为递归调用，执行的元素比较次数是C (n/2)
- 此算法所做的元素比较次数服从递推关系式
-
- $C(n) = 1$ 若$n = 2$
- $= 2C(n/2) + 2$ 若$n > 2$

- $k = \log n$

直接 vs.. 分治

- 在一个整数组 $A[1...n]$中，同时寻找最大值和最小值。一种直接的算法如下面所示，它返回一个数对（x, y），其中 x 是最小值，y 是最大值。

```
1.  $x \leftarrow A[1]$; $y \leftarrow A[1]$
2.  for $i \leftarrow 2$ to $n$
3.      if $A[i] < x$ then $x \leftarrow A[i]$
4.      if $A[i] > y$ then $y \leftarrow A[i]$
5.  end for
6.  return $(x, y)$
```

- $2n – 2$ vs. $3n/2 - 2$

二分搜索

- 将一给定元素 x 与已排序数组 A (*low*, *high*) 的中间元素 mid 做比较：

- 若 x<A(mid), 则不考虑 A (*mid*, *high*)
- 若 x>A(mid), 则放弃 A (*low*, *mid*)

二分搜索

BINARYSEARCHREC

输入：按非降序排列的*n* 个元素的数组*A* [1…*n*] 和元素*x*.。

输出：如果*x* = *A*[*j*] , 则输出*j*; 否则输出0。

binarysearch (1, *n*)

Procedure *binarysearch* (*low*, *high*)

```
1.  if $low > high$ then return 0
2.  else
3.      $mid \leftarrow \lfloor (low + high)/2 \rfloor$
4.      if $x = A[mid]$ then return $mid$
5.      else if $x < A[mid]$ then return binarysearch ($low$, $mid - 1$)
6.      else return binarysearch ($mid + 1$, $high$)
7.  end if
```

二分搜索 效率

- 当n = 0，即数组为空，算法不执行任何元素的比较
- 当n = 1 时，else 部分被执行，

  且若x ≠ A[mid], 算法将对空数组递归。从而得到当n = 1时，算法执行一次比较**(if then else)**。
- 如果n > 1, 则存在两种可能：当x = A[mid] 时，则算法仅仅执行一次比较；否则算法所需的比较次数是1加上由递归调用数组的前或后一半所执行的比较次数。

二分搜索 效率

- 如果设C(n) 表示算法BINARYSEARCHREC对大小为n 的数组在最坏的情况下所执行的比较次数，则C(n) 可表示为如下递推式
- C(n) = 1 若n = 1
- $\leq 1 + C(\lfloor n/2 \rfloor)$ 若n ≥ 2

- 设对某个整数$k \geq 2$, 满足$2^{k-1} \leq n < 2^k$。 展开上述递推式，可得到
- $C(n) \leq 1 + C(\lfloor n/2 \rfloor)$
- $\leq 2 + C(\lfloor n/4 \rfloor)$
- .
- .
- $\leq (k-1) + C(\lfloor n/2^{k-1} \rfloor)$
- $= (k-1) + 1$
- $= k$

二分搜索 效率

因为$\lfloor \lfloor n/2 \rfloor /2 \rfloor = \lfloor n/4 \rfloor$， 并且$\lfloor n/2^{k-1} \rfloor = 1$ （因为 $2^{k-1} \leqslant n < 2^k$ ），将不等式

$$2^{k-1} \leqslant n < 2^k$$

取对数并两边加 1 得到

$$k \leqslant \log n + 1 \leqslant k + 1$$

或

$$k = \lfloor \log n \rfloor + 1$$

因为 k 是整数。由此得到

$$C(n) \leqslant \lfloor \log n \rfloor + 1$$

二分搜索

- 算法BINARYSEARCHREC在n 个元素组成的已排序数组中搜索某个元素所执行的元素比较次数不超过$\lfloor \log n \rfloor$ + 1。
- 算法BINARYSEARCHREC 的时间复杂性是O(log n).

二分搜索—非递归

BINARYSEARCH

输入：按非降序排列的n 个元素的数组A [1…n] 和元素x.。

输出：如果$x = A[j]$, 则输出j; 否则输出0。

```
1.  $low \leftarrow 1$;  $high \leftarrow n$; $j \leftarrow 0$
2.  **while** ($low <= high$ ) **and** ($j=0$)
3.      $mid \leftarrow \lfloor (low+high)/2 \rfloor$
4.      **if** $x = A[mid]$ **then**  $j \leftarrow mid$
5.      **else if** $x < A[mid]$ **then** $high \leftarrow mid - 1$
6.      **else** $low \leftarrow mid + 1$
7.  **end while**
8.  **return** $j$
```

二分搜索—非递归效率

• 第一次迭代	元素个数	n
• 第二次迭代	元素个数	$\lfloor n/2 \rfloor$
• 第 *j* 次迭代	元素个数	$\lfloor n/2^{j-1} \rfloor$

$\lfloor n/2^{j-1} \rfloor$=1时

j = $\lfloor \lg n \rfloor$ +1

空间 *O*(log n) vs. Θ (1)