Insertion sort pseudocode

```
INSERTION-SORT(A)
for j ← 2 to n
    do key ← A[j]
       ▷ Insert A[j] into the sorted sequence A[1 .. j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

Implementation of merge sort

```
public static void mergeSort(int[] a) {
   int[] tmpArray = new int[a.length];
   mergeSort(a, tmpArray, 0, a.length - 1);
}

private static void mergeSort(int[] a, int[] tmpArray, int left, int right) {
   if (left < right) {
         int center = (left + right) / 2;
         mergeSort(a, tmpArray, left, center);
         mergeSort(a, tmpArray, center + 1, right);
         merge(a, tmpArray, left, center + 1, right);
   }
}
```

Implementation of merge sort

```
private static void merge(int[] a, int[] tmpArray, int leftPos, int rightPos, int rightEnd){
    int leftEnd = rightPos – 1, tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;

    while (leftPos <= leftEnd && rightPos <= rightEnd)
        if (a[leftPos] <= a[rightPos])
            tmpArray[tmpPos++] = a[leftPos++];
        else
            tmpArray[tmpPos++] = a[rightPos++];

    while (leftPos <= leftEnd)
        tmpArray[tmpPos++] = a[leftPos++];

    while (rightPos <= rightEnd)
        tmpArray[tmpPos++] = a[rightPos++];

    for (int i = 0; i < numElements; i++, rightEnd--)
        a[rightEnd] = tmpArray[rightEnd];
}
```

BinarySearch(arr, searchnum, left, right)

1	`if left == right`	$O(1)$
2	`    if arr[left]= searchnum`	$O(1)$
3	`        return left`	$O(1)$
4	`    else`	$O(1)$
5	`        return -1`	$O(1)$
6	`middle = (left + right)/2`	$O(1)$
7	`if arr[middle] = searchnum`	$O(1)$
8	`    return middle`	$O(1)$
9	`elseif arr[middle] < searchnum`	$O(1)$
10	`    return BinarySearch(arr, searchnum, middle+1, right)`	$O(?)$
11	`else`	
12	`    return BinarySearch(arr, searchnum, left, middle -1)`	$O(?)$

SelectionSort(arr, n)

```
1  if n ≤ 1
2    return arr
3  maxnum = arr[0]
4  maxIndex = 0
5  for i = 1 to n -1
6      if maxnum < arr[i]
7         maxnum = arr[i]
8         maxIndex = i
9  arr[maxIndex] = arr[n-1]
10 arr[n-1] = maxnum
11 SelectionSort(arr, n-1)
```

Master theorem: intuition

- Recurrence: $T(n) \leq a \cdot T(n/b) + O(n^d)$
- An algorithm that divides a problem of size n into a subproblems, each of size n / b

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

a: number of subproblems (branching factor)
b: factor by which input size shrinks (shrinking factor)
d: need to do $O(n^d)$ work to create subproblems + "merge" their solutions

Find max subarray crossing midpoint

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4       sum = sum + A[i]
 5       if sum > left-sum
 6            left-sum = sum
 7            max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11       sum = sum + A[j]
12       if sum > right-sum
13            right-sum = sum
14            max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

What is the time cost? Θ(n)

QuickSort

```
private static int median3(int[] a, int left, int right) {
        // Ensure a[left] <= a[center] <= a[right]
        int center = (left + right) / 2;
        if (a[center] < a[left])
                swap (a, left, center);
        if (a[right] < a[left])
                swap (a, left, right);
        if (a[right] < a[center])
                swap (a, center, right);

        // Place pivot at position right - 1
        swap (a, center, right - 1);
        return a[right - 1];
}
```

QuickSort

```
  /* Main quicksort routine */
private static void quicksort(int[ ] a, int left, int right) {
        if (left + CUTOFF <= right) {
                int pivot = median3(a, left, right);
                // Begin partitioning
                int i = left+1, j = right - 2;
                while (true) {
                        while (a[i] < pivot) {i++;}
                        while (a[j] > pivot) {j--;}
                        if (i >= j) break; // i meets j
                        swap (a, i, j);
                }
                swap (a, i, right - 1); // Restore pivot
                quicksort(a, left, i - 1); // Sort small elements
                quicksort(a, i + 1, right); // Sort large elements
        } else
                insertionSort(a, left, right);
}
```

ShellSort with {1,2,4,8,…,n/2}

```
public static void shellSort(int[ ] a) {
    int j;
    for (int gap = a.length/2; gap > 0; gap /=2)
        for (int i = gap; i < a.length; i++) {
            int tmp = a[i];
            for (j = i; j >= gap && tmp < a[j-gap]; j-= gap)
                a[j] = a[j-gap];
            a[j] = tmp;
        }
}
```

CountingSort

Algorithm CountingSort(S)
(values in S are between 0 and m-1)

```
for j ← 0 to m-1 do    // initialize m buckets
   b[j] ← 0
for i ← 0 to n-1 do    // place elements in their appropriate buckets
   b[S[i]] ← b[S[i]]  + 1
i ← 0
for j ← 0 to m-1 do    // place elements in buckets
   for r ← 1 to b[j] do // back in S
         S[i] ← j
         i ← i + 1
```

BucketSort

BUCKET-SORT(A, n)

for i ← 1 **to** n
    do insert A[i] into list B[⌊nA[i]⌋] — O(n)

for i ← 0 **to** n - 1
    do sort list B[i] with QuickSort — Θ(n)

concatenate lists B[0], B[1], . . . , B[n -1]
together in order
return the concatenated lists — O(n)

Θ(n)

Codes (1/2)

```
// items to be sorted are in {0,…,10^d-1},
// i.e., the type of d-digit integers
void radixsort(int A[], int n, int d)
{
   int i;
   for (i=0; i<d; i++)
      bucketsort(A, n, i);
}

// To extract d-th digit of x
int digit(int x, int d)
{
   int i;
   for (i=0; i<d; i++)
      x /= 10; // integer division
   return x%10;
}
```

Codes (2/2)

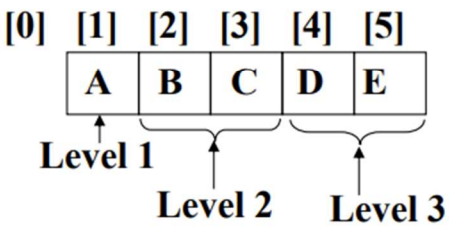
```
void bucketsort(int A[], int n, int d)
// stable-sort according to d-th digit
{
   int i, j;
   Queue *C = new Queue[10];
   for (i=0; i<10; i++) C[i].makeEmpty();
   for (i=0; i<n; i++)
      C[digit(A[i],d)].EnQueue(A[i]);
   for (i=0, j=0; i<10; i++)
      while (!C[i].empty())
      { // copy values from queues to A[]
         C[i].DeQueue(A[j]);
         j++;
      }
}
```

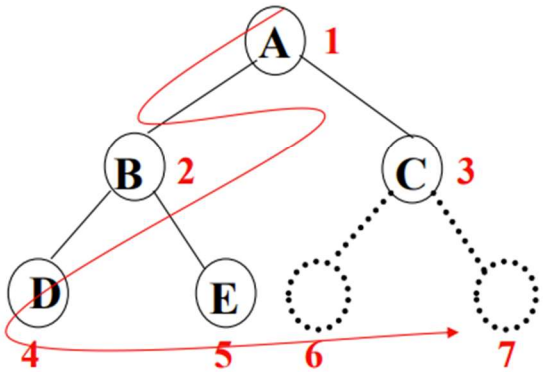
10 classic sorting algorithms

Sorting algorithm	Stability	Time cost			Extra space cost
		Best	Average	Worst	
Bubble sort	√	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Insertion sort	√	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Selection sort	×	O(n)	$O(n^2)$	$O(n^2)$	O(1)
MergeSort	√	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
HeapSort	×	O(nlogn)	O(nlogn)	O(nlogn)	O(1)
QuickSort	×	O(nlogn)	O(nlogn)	$O(n^2)$	O(1)
ShellSort	×	O(n)	$O(n^{1.3})$	$O(n^2)$	O(1)
CountingSort	√	O(n+k)	O(n+k)	O(n+k)	O(k)
BucketSort	√	O(n)	O(n+k)	$O(n^2)$	O(k)
RadixSort	√	O(nk)	O(nk)	O(nk)	O(n)

Binary tree design (ii)

- An array representation
  - Given a complete binary tree with n nodes, for any i-th node, $1 \leq i \leq n$,
    - parent(i) is $\lfloor i/2 \rfloor$
    - leftChild(i) is at $2i$ if $2i \leq n$; otherwise, i has no left child
    - rightChild(i) is at $2i+1$ if $2i+1 \leq n$; otherwise, i has no right child

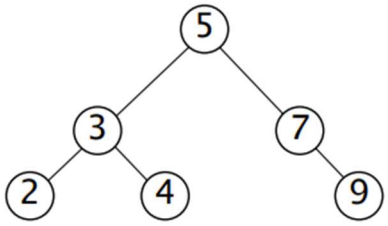




Searching for a key

find(x, k)

```
1.  if x = NIL or k = key [x]
2.       return x
3.  if k < key [x]
4.       return find(left [x], k )
5.  else
6.       return find(right [x], k )
```



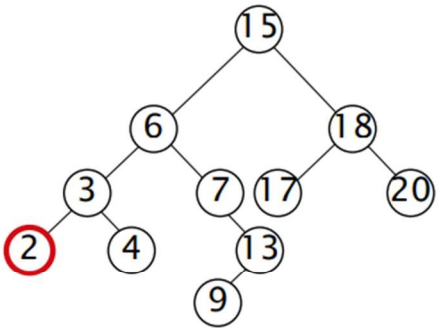
Running time: O(h), where h is the height of tree

Finding the minimum

- Goal: find the minimum value in a BST
  - Following left child pointers from the root, until a NIL is encountered

findMin(x)

```
1.  while left [x] ≠ NIL
2.        do x ← left [x]
3.  return x
```

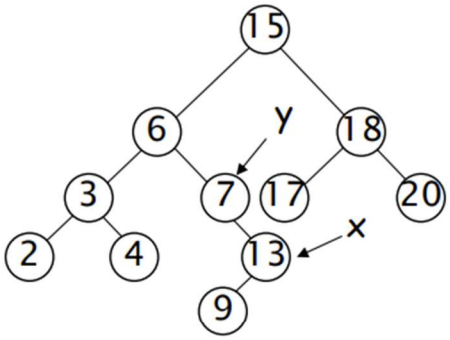


Minimum = 2

Running time: O(h), where h is the height of tree

Successor

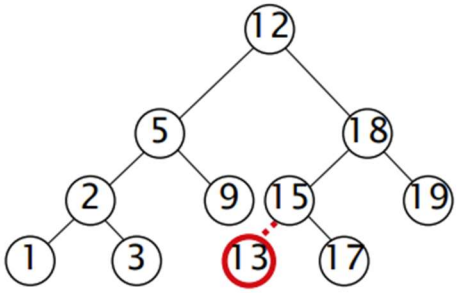
```
successor(x)
1.   if right [x] ≠ NIL
2.       return findMin(right [x])
3.   y ← p[x]
4.   while y ≠ NIL and x = right [y]
5.     do x ← y
6.        y ← p[y]
7.   return y
```



Running time: O(h), where h is the height of tree

Insert algorithm

```
1.   y ← NIL
2.   x ← root [T]
3.   while x ≠ NIL
4.      do y ← x
5.         if key [z] < key [x]
6.            x ← left [x]
7.         else
8.            x ← right [x]
9.   p[z] ← y
10.  if y = NIL
11.     root [T] ← z          ▷ T was empty
12.  else
13.     if key [z] < key [y]
14.        left [y] ← z
15.     else
16.        right [y] ← z
```

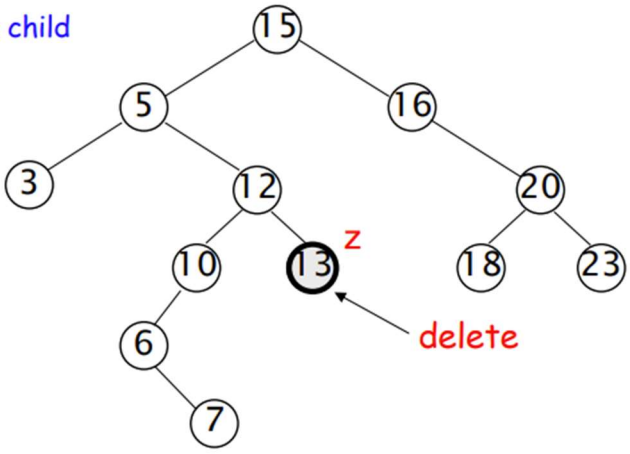


Best-case and worst-case time complexities?

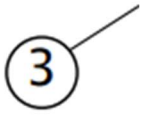
Running time: O(h)

Deletion algorithm

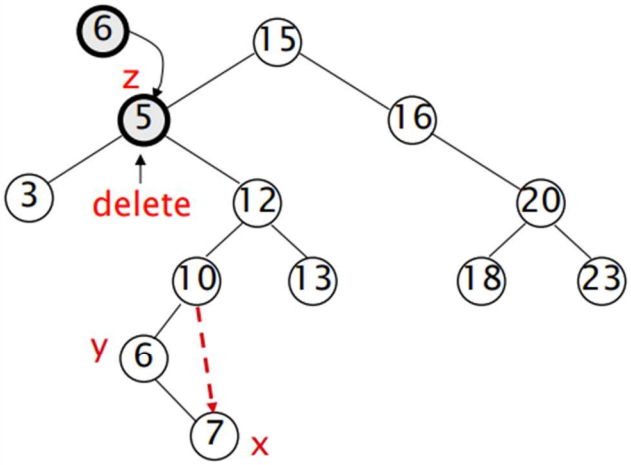
```
1.  if left[z] = NIL and right[z] = NIL    //z has no child
2.      if p[z] = NIL then root[T] = NIL
3.      if z = left[p[z]]
4.          left[p[z]] = NIL
5.      else
6.          right[p[z]] = NIL
```




```
1.  if left[z] = NIL and right[z] ≠ NIL    //z has one right child
2.      y = right[z]
3.      if p[z] = NIL
4.          root[T] = y
5.      else
6.          p[y] = p[z]
7.          if z = left[p[z]]
8.              left[p[z]] = y
9.          else
10.             right[p[z]] = y
11. if left[z] ≠ NIL and right[z] = NIL    //z has one left child
12.     y = left[z]
13.     if p[z] = NIL
14.         root[T] = y
15.     else
16.         p[y] = p[z]
17.         if z = left[p[z]]
18.             left[p[z]] = y
19.         else
20.             right[p[z]] = y
```



```
1.  if left[z] ≠ NIL and right[z] ≠ NIL //z has two children
2.      y ← TREE-SUCCESSOR(z)      //left-most node in right tree
3.      if p[y] = z
4.          right[z] = right[y]
5.          if right[y] ≠ NIL
6.              p[right[y]] = z
7.      else
8.          if right[y] = NIL
9.              left[p[y]] ← NIL
10.         else
11.             x ← right[y]
12.             p[x] ← p[y]
13.             left[p[y]] ← x
14.     key[z] ← key[y] //copy y's data into z
```



Best/worst-case time complexities?

Rebalance implementation

```
private AvlNode<Anytype> insert(Anytype x, AvlNode<Anytype> t ) {
/*1*/     if( t == null )    t = new AvlNode<Anytype>( x, null, null );
/*2*/     else if( x.compareTo( t.element ) < 0 )
          {
                    t.left = insert( x, t.left );
                    if( height( t.left ) - height( t.right ) == 2 )
                              if( x.compareTo( t.left.element ) < 0 )
                                        t = rotateWithLeftChild( t );
                              else
                                        t = doubleWithLeftChild( t );
          }
/*3*/     else if( x.compareTo( t.element ) > 0 )
          {
                    t.right = insert( x, t.right );
                    if( height( t.right ) - height( t.left ) == 2 )
                              if( x.compareTo( t.right.element ) > 0 )
                                        t = rotateWithRightChild( t );
                              else
                                        t = doubleWithRightChild( t );
          }
/*4*/     else
                    ;  // Duplicate; do nothing
          t.height = max( height( t.left ), height( t.right ) ) + 1;
          return t;
}
```

Rebalance implementation

```
private static AvlNode<Anytype> rotateWithLeftChild(AvlNode<Anytype> k2 )
{
        AvlNode<Anytype> k1 = k2.left;
        k2.left = k1.right;
        k1.right = k2;
        k2.height = max(height(k2.left), height(k2.right)) + 1;
        k1.height = max(height( k1.left), k2.height) + 1;
        return k1;
}
```

```
private static AvlNode<Anytype> rotateWithRightChild( AvlNode<Anytype> k1 )
{
    AvlNode<Anytype> k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    k1.height = max( height( k1.left ), height( k1.right ) ) + 1;
    k2.height = max( height( k2.right ), k1.height ) + 1;
    return k2;
}
```

Rebalance implementation

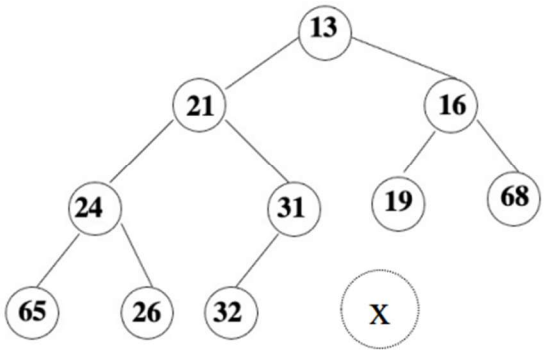
```
private static AvlNode<Anytype> doubleWithLeftChild( AvlNode<Anytype> k3 )
{
        k3.left = rotateWithRightChild( k3.left );
        return rotateWithLeftChild( k3 );
}
```

```
private static AvlNode<Anytype> doubleWithRightChild( AvlNode<Anytype> k1 )
{
        k1.right = rotateWithLeftChild( k1.right );
        return rotateWithRightChild( k1 );
}
```

Binary heap: insert

```
public void insert(ElementType x) throws Exception {
        if (isFull())
                throw new Exception("Overflow");

        // Percolate up
        int hole = ++currentSize;
        while(hole > 1 && x.isHigherPriorityThan(arr[hole/2])) {
                arr[hole] = array[hole / 2];
                hole /= 2;
        }
        arr[hole] = x;
}
```



Binary heap: deleteMin

```
public String deleteMin() {
        if (isEmpty())
                return null;

        String data = arr[1].data;
        arr[1] = arr[currentSize--];

        percolateDown(1);
        return data;
}
```

Binary heap: percolateDown

```
private void percolateDown(int hole) {
        int child;
        ElementType tmp = arr[hole];
        while (hole * 2 <= currentSize) {
                child = hole * 2;
                if (child != currentSize &&
                        arr[child +1].isHigherPriorityThan(arr[child]))
                        child++;
                if (arr[child].isHigherPriorityThan(tmp))
                        arr[hole] = array[child];
                else
                        break;
                hole = child;
        }
        arr[hole] = tmp;
}
```

Linear probing

- For insertion, we probe $h(k)$, $h(k)+1$, $h(k)+2$,$\cdots$, $h(k)+(m-1)$ one by one until we find an empty slot, and insert the record to this slot
  - Formally we probe $h(k,i) = (h(k)+i)\%m$ from $i=0$ to $i=m-1$ until we find an empty slot to insert
- When searching for a record with a certain key,
  - Compute $h(k)$
  - Examine the hash table buckets in order $T[h(k,i)]$ for $0 \le i \le m-1$ until one of the following happens:
    - $T[h(k,i)]$ has the record with key equal to k
    - $T[h(k,i)]$ is empty, then no record contains key k in the hash table

Double hashing

- Deficiency of linear probing
  - Long sequence of occupied slots, which degrades the query efficiency

- Double hashing
  - We have an additional hash function $h' > 0$
  - Insertion: we probe $h(k,i) = \big(h(k) + i \cdot h'(k)\big)\%m$ one by one for i from 0 to $m-1$ until an empty slot is found
  - Search: we search $h(k,i)$ for i from 0 to $m-1$ until one of the following happens:
    - $T[h(k,i)]$ has the record with key equal to k
    - $T[h(k,i)]$ is empty, then no record contains key k in the hash table

Universal hashing

- Let $\mathcal{H}$ be a family of hash functions from $[U]$ to $[m]$
- $\mathcal{H}$ is called universal if the following condition holds:

> Let k_1, k_2 be two distinct integers from $[U]$. By picking a function $h \in \mathcal{H}$ uniformly at random, we guarantee that
>
> $$\Pr[h(k_1) = h(k_2)] \leq \frac{1}{m}$$

  - Then, we choose one from $\mathcal{H}$ uniformly at random and use it as the hash function h for all operations

- Theoretical guarantee with Universal Hashing
  - With a universal hash function h
    - Query time of chaining: $O(1+\alpha)$. (Proof: check appendix)
    - Query time of open addressing: more complicated (Omit), see [1]

Adjacency list vs adjacent matrix

- Adjacency list:
  - Space: $O(n+m)$, save space if the graph is sparse, i.e., $m \ll n^2$
  - Check the existence of an edge (u, v): $O(k)$ time where k is the number of neighbors of v
  - Retrieve the neighbors of a node: $O(k)$ time
  - Add/delete a node: $O(n)$
  - Add/delete an edge: $O(k)$

- Adjacency matrix:
  - Space consumption: $O(n^2)$
  - Check the existence of an edge (u, v): $O(1)$ time
  - Retrieve the neighbors of a node: $O(n)$ time
  - Add/delete a node: $O(n^2)$, (create a new matrix)
  - Add/delete an edge: $O(1)$

BFS: implementation

Algorithm 1: ***BFS(G, s)***

```
1  color← allocate an array of size G.n, initialize with all zeros
2  // Use 0 : white, 1 : gray, and 2: black
3  Q ← an empty queue
4  Q.enqueue(s)
5  color[s] ← 1
6  while !Q.isEmpty()
7   v ← Q.dequeue
8   for u ∈ out-neighbor of v
9      if color[u]=0
10        Q.enqueue(u)
11        color[u]=1
12  color[v] ← 2
13  print v
14 free the array color if necessary
```

```
//adjacency matrix to store the graph
for u = 0 to G.n-1
  if G.adjmatrix[v][u]==1 and color[u]==0
     …

//adjacency list to store the graph
linkedlist_node = G[v].head.next
while linkedlist_node != G[v].tail
   u =linkedlist_node.element
   …
   linkedlist_node = linkedlist_node.next
```

DFS: implementation

Algorithm 1: ***DFS(V, E, s)***

```
1  color← initialize an array of size n with all zero values
2  // Use 0 : white, 1 : gray, and 2: black
3  S ← an empty stack
4  S.push(s)
5  color[s] ← 1
6  while !S.isEmpty()
7     v ← S.top()
8     if v still has white-neighbor u
9            S.push(u)
10           color[u]=1
11    else
12        color[v] ← 2
13        S.pop()
14 Free color array if necessary
```

Prim(V, E, w, r)

```
1.  Q ← ∅                                          ┐
2.  for each u ∈ V                                 │
3.      do key[u] ← ∞                              │ O(V) if Q is implemented as a min-heap
4.         π[u] ← NIL                              │
5.         INSERT(Q, u)                            ┘
6.  DECREASE-KEY(Q, r, 0)        ▶ key[r] ← 0  ← O(lgV)
7.  while Q ≠ ∅                  ← Executed |V| times   ┐ Min-heap
8.        do u ← EXTRACT-MIN(Q)  ← Takes O(lgV)          ┘ operations: O(VlgV)
9.           for each v ∈ Adj[u]      ← Executed O(E) times   ┐
10.             do if v ∈ Q and w(u, v) < key[v]  ← Constant   │ O(ElgV)
11.                then π[v] ← u                  ← Takes O(lgV) │
12.                     DECREASE-KEY(Q, v, w(u, v))              ┘
```

Total time: O(VlgV + ElgV) = O(ElgV)

Prim(V, E, w, r)

```
1.  Q ← ∅
2.  for each u ∈ V
3.      do key[u] ← ∞
4.         π[u] ← NIL
5.         INSERT(Q, u)
6.  DECREASE-KEY(Q, r, 0)        ▶ key[r] ← 0
7.  while Q ≠ ∅
8.        do u ← EXTRACT-MIN(Q)
9.           for (j=0; j<|V|; j++)
10.             if (A[u][j]=1)
11.                if v ∈ Q and w(u, v) < key[v]
12.                   then π[v] ← u
13.                        DECREASE-KEY(Q, v, w(u, v))
```

Total time: $O(VlgV + ElgV+V^2) = O(ElgV+V^2)$

- Lines 1–5: $O(V)$ if Q is implemented as a min-heap
- Line 6: $O(lgV)$
- Line 7: Executed |V| times
- Lines 7–8: Min-heap operations: $O(VlgV)$
- Line 8: Takes $O(lgV)$
- Line 9: Executed $O(V^2)$ times total
- Line 10: Constant
- Lines 12–13: Takes $O(lgV)$; $O(ElgV)$

Algorithm 2: using labels

Assume vertices are 1, 2, …, n, and E >= V

```
1.  Sort all the edges                                   O(ElogE)
2.  for each v ∈ V
3.      label[v] = v                                     O(V)
4.      setArray[v] = {v}
5.  for each edge (u, v) ∈ E
6.      uL = label[u], vL = label[v]                     O(E)
7.      if uL == vL  continue
8.      R.add((u, v))
9.    if setArray[uL].size >= setArray[vL].size
10.     for each vertex w ∈ setArray[vL]                 O(VlogV)
11.           label[w] = uL
12.           setArray[uL].add(w)
13.   else
14.       for each vertex w ∈ setArray[uL]
15.           label[w] = vL
16.           setArray[vL].add(w)
17. Output R
```

O(ElgE)

Dijkstra(G, w, s)

```
1.  INITIALIZE-SINGLE-SOURCE(V, s)  ← Θ(V)
2.  S ← ∅
3.  Q ← V[G]  ← O(V) build min-heap
4.  while Q ≠ ∅  ← Executed O(V) times
5.      do u ← EXTRACT-MIN(Q) ← O(lgV)        } O(VlgV)
6.      S ← S ∪ {u}
7.      for each vertex v ∈ Adj[u] ←O(E) times (total)
8.              do RELAX(u, v, w)                          } O(ElgV)
9.              Update Q (DECREASE_KEY) ← O(lgV)
```

Running time: O(VlgV + ElgV) = O(ElgV)

Floyd's algorithm using a single *D*

```
Floyd
  1. D ← W  // initialize D array to W[ ]
  2. P ← 0    // initialize P array to [0]
  3. for k ← 1 to n
  4.     do for i ← 1 to n
  5.         do for j ← 1 to n
  6.             if (D[ i, j ] > D[ i, k ] + D[ k, j ] )
  7.                 then  D[ i, j ] ← D[ i, k ] + D[ k, j ]
  8.                       P[ i, j ] ← k;
```

} O(|V|³)

Total time cost: O(|V|³)

Bellman-Ford(V, E, w, s)

```
1.  INITIALIZE-SINGLE-SOURCE(V, s)          ← Θ(|V|)
2.  for i ← 1 to |V| - 1                    ← O(|V|)
3.    do for each edge (u, v) ∈ E           ← O(|E|)        } O(|V||E|)
4.         do RELAX(u, v, w)
5.  for each edge (u, v) ∈ E                ← O(|E|)
6.    do if d[v] > d[u] + w(u, v)
7.         then return FALSE
8.   return TRUE
```

Running time: $O(|V||E|)$