Fundamental algorithms

Wang Bingbing

May 2, 2010

1 Sorting

Problem Definition. The output sequence is an ordered permutation of the input sequence. When the input is a file, the output is usually a distinct file; when input is an array, the output is usually the same array.

All sorting algorithms can be divided into two groups by whether it is only based on comparisons between elements. The time complexity of all comparison-based sorting algorithms is at least $O(n \lg n)$. Other sorting algorithms can be linear in time.

There is another problem closely related to sorting: selection problem.

1.1 General-Purpose Algorithms

The following algorithms sort an arbitrary n-element sequence.

- Insertion sort
  - efficient for small data sets
  - efficient for almost sorted data sets
  - more efficient than most other simple quadratic algorithms such as selection sort or bubble sort.
  - stable, in-place and online
  - Time complexity
    - Best case: $O(n)$
    - Average case: $O(n^2)$
    - Worst case: $O(n^2)$
- Shell sort
  - Analogous to insertion sort
  - non-stable, in-place

    - Time complexity
        - Worst case: $O(n^{3/2})$ using Hibband's increment sequence

- Heapsort
    - in-place, non-stable
    - Time complexity
        - expected and worst-case: $O(n \lg n)$

- Merge sort
    - stable sorting
    - external sorting
    - Time complexity
        - expected and worst-case: $O(n \lg n)$
    - Space complexiy: $O(n)$

- Quicksort
    - in-place, non-stable
    - works good in virtual memory environment
    - not very good for small input size
    - can be easily improved using good partition scheme such as median-of-3 and tail recursion
    - Time complexity
        - expected: $O(n \lg n)$
        - worst-case: $O(n^2)$
        - best-case: $O(n \lg n)$
    - Space complexity
        - stack depth can be reduced to $O(\lg n)$ using tail recursion

1.2 Special-Purpose Algorithms

These algorithms lead to short and efficient programs for certain inputs.

- Counting sort
    - stable sorting
    - usually used in radix sort
    - Time complexity (assuming n integers ranging from 1 to k)

    - $\Theta(n + k)$
  - Space complexity: $\Theta(k)$

- Radix sort. The following assumes n d-digit numbers in which each digit can take on up to k possible values.
  - Time complexity
    - $\Theta(d(n + k))$
  - Space complexity: $\Theta(k)$
  - LSD radix sort is stable
  - not make effective use of cache

- Bucket sort. Bucket sort runs in linear time when the input is drawn from a uniform distribution over the interval $[0, 1)$. The idear is to divide the interval $[0, 1)$ into n equal-sized subintervals, or *buckets*, and then distribute the n input numbers into the buckets. Since the inputs are uniformly distributed over $[0, 1)$, we don't expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

- Bitmap sort. The bitmap sort uses the fact that the integers to be sorted are from a small range, contain no duplicates, and have no additional data. Running time is $\Theta(n)$.

2 Selection

Problem Definition. The kth order statistic of a set of n elements is the kth smallest element. The selection problem is to find the kth order statistic from a set of n distinct numbers.

There are several special cases of this problem and these special problems can be solved more efficiently using particular method than the general problem. There are some variants of this problem too, such as find the largest k elements or the k smallest elements.

2.1 Linear General-Purpose Algorithm

- Randomized-select, analogous to quicksort, randomly partition and recursively select in one subarray (see Section 9.2 of [CLRS 2001])
  - expected time: $\Theta(n)$, worst-case: $\Theta(n^2)$
  - can be tuned to a iterative version using tail recursion
  - can choose a good pivot by selecting from a small sample of elements

- Median-of-medians algorithm (due to Blum, Floyd, Pratt, Rivest and Tarjan, see Section 9.3 of [CLRS 2001]). The key idea is to choose a good pivot by using median-of-medians method.
  - worst-case time: $\Theta(n)$
  - the constant factor hidden in $\Theta(n)$ is large, great breakthrough in theory but not practical
  - see Problem 9-3 of [CLRS 2001] for a improved version

2.2 Non-linear general selection algorithms

- Sort the elements and return the kth position
  - Time = $\Theta(n \lg n)$ assuming quicksort or heapsort
- Quicksort the first k elements and process the remaining elements once per pass using inerstion sort
  - Time = $\Theta(k \lg k + k(N-k))$
- Make a max-heap of the first k elements and process the remaining elements once per pass maintaining max-heap property
  - Time = $\Theta(k + (N-k)\lg k)$
- Make a max-heap of N elements and extract the k smallest elements
  - Time = $\Theta(N + k \lg N)$

2.3 Special-Purpose Algorithms

- Minimum/Maximum
  - $n-1$ comparisons optimally (see Section 9.1 of [CLRS 2001])
- Simultaneous minimum and maximum
  - $\lceil \frac{3}{2} n \rceil$ comparisons optimally (see Ex. 9.1-2 of [CLRS 2001])
  - compaire pairs of elements from the input first with each other, and then find the minimum from the smaller part and the maximum from the larger part (see Section 9.1 of [CLRS 2001])
- Select second smallest element
  - Tournament sorting (see Ex. 9.1-1 of [CLRS 2001])
  - $n + \lceil \lg n \rceil - 2$ comparisons in the worst case

2.4 Using data structures to select in sublinear time

The strategy to find an order statistic in sublinear time is to store the data in an organized fashion using suitable data structures that facilitate the selection. One such data structure is the order-statistic tree, which dynamically maintains the order statistic of each element (see Chapter 14 of [CLRS 2001]).

2.5 Weighted median

Problem Definition. This is a variant of selection problem, which is specified as follows. For n distinct elements $x_1, x_2, \ldots, x_n$ with positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = 1$, the *weighted(lower) median* is the element x_k satisfying

$$\sum_{x_i < x_k} w_i < \frac{1}{2}$$

and

$$\sum_{x_i > x_k} w_i \leq \frac{1}{2}.$$

This problem can be solved in $\Theta(n)$ worst-case time by using the linear select algorithm as a subroutine.

```
WEIGHTED-MEDIAN($X$)
 1  if $length[X] = 1$
 2     then return $x_1$
 3  elseif $length[X] = 2$
 4     then if $w_1 \geq w_2$
 5             then return $x_1$
 6             else  return $x_2$
 7  else
 8           find the median $x_k$ of $X = x_1, x_2, \ldots, x_n$ by using linear select algorithm
 9           partition the set $X$ around $x_k$
10           compute $W_L = \sum_{x_i<x_k} w_i$ and $W_G = \sum_{x_i>x_k} w_i$
11           if $W_L < 1/2$ and $W_G < 1/2$
12              then return $x_k$
13           elseif $W_L > 1/2$
14              then $w_k \leftarrow w_k + W_G$
15                   $X \leftarrow x_i \in X : x_i \leq x_k$
16                   return WEIGHTED-MEDIAN($X'$)
17           else $w_k \leftarrow w_k + W_L$
18                   $X \leftarrow x_i \in X : x_i \geq x_k$
19                   return WEIGHTED-MEDIAN($X'$)
```

3 Searching

Problem Definition. A search algorithm determines its input is a member of a given set, and possibly retrieve associated information. Efficient search entails representing the set in a clever way. So the searching problem mainly focuses on the various data structures of representing dynamic sets.

3.1 Operations on a dynamic set

Operations on a dynamic set can be grouped into two categories: *queries* (Search(S, k), Minimum(S), Maximum(S), Successor(S, x), Predecessor(S, x)) and *modifying operations* (Insert(S, x), Delete(S, x)).

A dynamic set that only efficiently supports Insert, Delete and Search is called a *dictinary.*

3.2 Data structures of dynamic sets

3.2.1 Arrays (ordered or unordered)

When the array is ordered, Search is efficient (using binary search) but Insert and Delete is inefficient.

When the array is unordered, Insert and Delete is easy but Search is inefficient.

Another inconvenience is that array implementations require that the maximum table size be known ahead of time or that the table undergo amortized growth (see subsection 17.4 of [CLRS 2001]).

3.2.2 Linked list (ordered or unordered)

Linked list is more flexible for modifying operations and we do not have to predict the maximum size of the table in advance.

The disadvantage of linked lists is that we need extra space (for the links) and linked list is inefficient for query operation such as Search because we can not use binary search.

3.2.3 Hashing

Hasing is very efficient for Search operations but is not convenient to implement modifying operations. Therefore it is a good choice if modifying operations are rare and Search operations are frequent.

3.2.4 Binary Search Tree (BST)

BSTs integrate the advantanges of linked lists being efficient for modification, and the advantanges of arrays being efficient to search (binary search).

No matter what node we start at in a height-h binary search tree, k successive PREDECESSOR or SUCCESSOR take $O(k+h)$ time (See Ex. 12.2-8 of [CLRS 2001]). SEARCH is proportionate to the height of the tree.

But when BSTs become quitely unbalanced, the search operation becomes linear in time. So we have to rebalance the tree when necessary. There are three general approaches to providing performance guanrantees in algorithm design: *randomize*, *amortize* and *optimize*. And these approaches can be used to solve the balance problem.

Randomization. A *randomized* algorithm introduces random decision making into the algorithm itself, to reduce dramatically the chance of a worst-case scenario (no matter what the input). Quicksort is a example of this method. *Randomized BSTs* and *skip lists* are two simple ways to use randomization in dynamic set implementations to give efficient implementations of all dynamic set operations. There algorithms are simple and are broadly applicable.

Amortization. An *amortization* approach is to do extra work at one time to avoid more work later, to be able to provide guanranteed upper bounds on the average per-operation cost (the total cost of all operations divided by the number of operations). *Splay BSTs* are an example of using this method.

Optimization. An *optimization* approach is to take the trouble to provide performance guarantees for every operation. Various methods have been develped to take this approach. These methods require that we maintain some structural infomation in the trees to help to balance it. Generally there are two techniques to rebalance the trees: by rotations or by manipulating the node's degrees.

According to above discussion we can divide the various BSTs into three groups, in which those using optimization method can be furter divided into two groups according to its rebalancing schemes.

- Randomization
  - Randomly built BST
  - Treap
  - Skip list
- Amortization
  - Splay trees
- Optimization
  - AVL trees
  - Red-Black trees
  - B-tree (including 2-3 trees and 2-3-4 trees)

3.3 Randomized BSTs

If we insert keys in random order into an empty tree, the constructed tree is a randomized BST and its height is $\Theta(\lg n)$ in expected case. So search is fast in expected case. The randomization property of the BST built this way totally depends on the randomness of the input. But what if the input is not randomized? Yes, we can randomize the input sequence. But what if we do not have all the items at once? There are two ways to circumvent this problem: (1) root insertion with some probability and (2) *Treap.*

Root insertion. The root insertion method is simple: when we insert a new node into the tree of n nodes, the new node should appear at the root with the probability $1/(n+1)$, so we simply make a randomized decision to use root inseriton with that probability. Otherwise, we recursively use the method to insert the new item into the left subtree if the item's key is less than the key at the root, and into the right subtree if the item's key is greater. See subsection 13.1 of [Sedgewick 1998] for more details.

Treap. Treap is a binary search tree. Each node in the tree has a key and has a *priority*, which is a random number chosen when inserted. The nodes in the treap are ordered so that the keys obey the binary search tree property and the priorities obey the min-heap property, hence the word "treap".

Inserting a new node into treap is similar to into the BST, but if min-heap property is violated, we need do rotations to fix it. Expected number of rotations performed when inserting a node into a treap is less than 2. See problem 13-4 of [CLRS 2001] for more details.

3.3.1 Skip list

The structure of skip list is similar to a singly-linked list and its behavior is like a BST. It is also a randomized data structure and is very easy to understand and to be implemented. See subsection 13.5 of [Sedgewick 1998] for more details.

3.3.2 Amortized BSTs

3.3.3 Splay trees

Splay insertion brings newly inserted nodes to the root using the rotations similar to root insertion but *different.* When we insert a node into a BST using splay insertion, we not only bring that node to the root, but also bring the other nodes that we encounter (on the search path) closer to the root. This property holds if we implement the search operation such that it performs the splay transformations during the search.

Splay trees need not store any bookkeeping infomaiton and have a good amortized performance ganrantee: we ganrantee not that each operation is efficient but rather that the average cost of all the operations performed is efficient.

See [Sedgewick 1998] and [Sleator 1985] for more details on splay trees.

3.4 Optimized BSTs

3.4.1 AVL trees

The AVL tree has the property that its subtrees' heights differ at most 1. It was called the balance condition. In general AVL trees are more balanced than a red-black tree, so it is more efficient for look-up-intense applications. Of course it is more difficult to maintain the balance condition.

It needs to store a balance factor or subtree size in each node. Its height is limited to $1.44 \lg n$.

The time complixity of its operations is as follows.

- Insert
  Time $= O(\lg n)$. Using rotations to maintain balance need only one single-rotation or double-rotation.

- Delete
  Time $= O(\lg n)$, $O(\lg n)$ for lookup plus maximum of $O(\lg n)$ rotations.

AVL trees are more rigidly balanced than red-black trees, leading to slower insertion and removal but faster retrieval.

3.4.2 Red-Black trees

Compared with AVL trees, red-black trees loosen the balance condition to get faster insertion and deletion while fast search. Its height is limited to $2 \lg n$. It needs to store the infomation indicating whether the node is red or black.

The time complixity of insertion and deletion is as follows.

- Insert
  Time $= O(\lg n)$, need at most 2 rotations.

- Delete
  Time $= O(\lg n)$, need at most 3 rotations.

3.4.3 B-tree

Each B-tree has a related fixed integer $t \geq 2$ called the *minimum degree* of the B-tree: every node other than the root must have at least $t - 1$ keys (and thus at least t children) and every node (including the root) can contain at most $2t - 1$ keys (and thus at most $2t$ children).

2-3-4 tree is a special B-tree with $t = 2$. 2-3 tree is transformed from 2-3-4 tree. They have a close relation to red-black trees. The red-black tree can be interpreted as a 2-3-4 tree: if we let the black nodes absorb the red nodes which are its children then the resulted tree is a fully balanced tree according to red-black property and is a 2-3-4 tree. For their relations, see 13.4 of [Sedgewick 1998].

Due to low height of B-tree, it is usually implemented on magnetic disks to store huge info to minimize disk I/O operations.

4 Algorithms on strings

There are many problems involving strings such as sorting, matching, searching for a particular string in a long text, longest common substring (LCS), longest duplicated substring or palindromes, etc. To solve these problems efficiently we can preprocess the strings and represent them in a specific data structure. The following are some frequently used data structures to represent strings.

4.1 Data structures on strings

Radix trees for bit strings. This data structure is very convenient to sort the bit strings in time proportionate to the sum of length of all bit strings. See problem 12-2 of [CLRS 2001].

Suffix trees. Suffix tree is a great data structure to attack many problems about strings. A search tree that is built from keys defined by string pointers into a text string is called a *suffix tree*. It represents all suffixes of a string, hence its name. Suffix tree can be represented by any data structure that can admit variable-length keys such as *trie* (section 15.2 of [Sedgewick 1998]), *patricia tries* (section 15.3 of [Sedgewick 1998]), *ternary search trie* (TST, section 15.4 of [Sedgewick 1998]) and BSTs. But trie-based methods are particularly suitable, because (except for the trie methods that do one-way branching at the tails of keys) their running time does not depend on the key length, but rather depends on only the number of digits required to distinguish among the keys. This characteristic lies in direct constrast to, for example, hasing algorithms, which do not apply immediately to this problem because their running time is proportionaly to the key length. There is a on-line linear algorithm to construct the suffix tree ([Ukkonen]).

Suffix array. Suffix array, which is an array of pointers to every character of a string array, is similar to suffix trees but simpler. It can be constructed by initializing each element to point to the corresponding character in the string and then sort it. See section 15.2 of [Bently 2000] for more details.

4.2 Algorithms on strings

- String matching
  - Rabin-Karp algorithm (32.2 of [CLRS 2001])
  - Finit automata (32.3 of [CLRS 2001])
  - Knuth-Morris-Pratt algorithm (32.4 of [CLRS 2001])
  - Suffix array or suffix tree. Preprocess the texts to represent it in a suffix array or suffix tree then process the searching requests. This method is very efficient when the text is fixed and the searching request is frequent.

- Longest Common Substring (LCS)
  - Using suffix array or suffix tree can solve it in linear time. See problem 15.9 of [Bently 2000].
- Longest substring that occurs more than M times
  - Using suffix array can solve it in linear time. See problem 15.8 of [Bently 2000].
- Longest palindrome in a string
  - Similar to the previous problem. It can be solved in linear time using suffix array.

5 Graph Algorithms

5.1 Minimum Spanning Trees (MST)

The two algorithms to compute MST both have the following general forms.

```
Generic-MST($G, w$)
1  $A \leftarrow \emptyset$
2  while $A$ does not form a spanning tree
3        do find an edge $(u, v)$ that is safe for $A$
4           $A \leftarrow A \cup \{(u, v)\}$
5  return $A$
```

In Kruskal's algorithm, the set A is a forest. The safe edge added to A is always a least-weight edge in the graph that connects two distinct components. In Prim's algorithm, the set A forms a single tree. The safe edge added to A is always a least-weight edge connecting the tree to a vertex not in the tree.

5.1.1 Kruskal's Algorithm

The running time of Kruskal's algorithm depends on the implementation of disjoint-set data structure. If the disjoint-set is implemented using a forest with the union-by-rank and path-compression heuristics (see Chapter 21 of [CLRS 2001]) the running time is $O(E \lg E + (V + E)\alpha(V))$ in which $\alpha(V)$ is approximately a constant.

5.1.2 Prim's Algorithm

The runnint time of Prim's algorithm depends on the implementation of priority queue. Time $= \Theta(V) \cdot T_{\text{Extract-Min}} + \Theta(E) \cdot T_{\text{Dcrease-Key}}$

Θ	$T_{\text{Extract-Min}}$	$T_{\text{Decrease-Key}}$	Total
Binary Heap	$O(\lg(V))$	$O(\lg(V))$	$O(E\lg(V))$
Fibonacci Heap	$O(\lg(V))$	$O(1)$	$O(E + V\lg(V))$
	amortized	amortized	worst case

5.2 Single-Source Shortest Path (SSSP)

5.2.1 Dijkstra's algorithm

- require non-negative weight, directed or undirected
- greed algorithm
- similar to Prim's algorithm and breadth first search
- time = $\Theta(V \cdot T_{\text{Extract-Min}} + E \cdot T_{\text{Decrease-Key}})$
  - $O(E\lg(V))$ using binary heap
  - $O(E + V\lg(V))$ using Fibonacci heap

When $w(u, v) = 1$ for all $(u, v) \in E$, BFS can be used to solve SSSP in $O(V + E)$ time.

5.2.2 Bellman-Ford Algorithm

- a general algorithm, can be applied for various graphs
- can determine negative-weight cycles
- Time: $O(VE)$
- can be used to solve a specific linear programming problem: systems of difference constraints in $O(mn)$ time
  - Bellman-Ford maximize $\sum_{i=1}^{n} x_i$ subject to $Ax \leq b$ and $x_i \leq 0$ for all x_i (see Ex. 24.4-8 in [CLRS 2001])
  - Bellman-Ford minimize $\max\{x_i\} - \min\{x_i\}$ subject to $Ax \leq b$ (see Ex. 24.4-9 in [CLRS 2001])

5.2.3 Single-source shortest paths in directed acylic graphs

If the graph is acylic we can compute shortest paths faster than Bellman-Ford by relaxing edges in some order (topological order on the vertices).

```
DAG-SHORTEST-PATHS($G, w, s$)
1  topologically sort the vertices of $G$
2  ▷ Initialize distance of $s$ to each vertex.
3  for each vertex $v \in V[G]$
4          do $d[v] \leftarrow \infty$
5             $\pi[v] \leftarrow$ NIL
6  $d[s] \leftarrow 0$
7  for each vertex $u$, taken in topologically sorted order
8          do for each vertex $v \in Adj[u]$
9                     do RELAX(u, v, w)
```

The running time is $O(V + E)$.

5.3 All-Pairs Shortest Paths

5.3.1 Floyd-Warshall algorithm

The Floyd-Warshall algorithm is a dynamic programming algorithm. The key idea is to define a clever subproblem. Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set $1, 2, \ldots, k$. When $k = 0$, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge, and hence $d_{ij}^{(0)} = w_{ij}$. A recursive definition following the above discussion is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1. \end{cases}$$

Because for any path, all intermediate vertices are in the set $1, 2, \ldots, n$, the matrix $D^{(n)} = (d_{ij}^{(n)})$ gives the final answer: $d_{ij}^{(n)} = \delta(i, j)$ for all $i, j \in V$.

The running time of Floyd-Warshall algorithm is $O(V^3)$ and it can detect the presence of negative-weight cycles by checking whether the diagonal of the matrix $D^{(n)}$ has any negative element.

Transitive closure of a directed graph. The Floy-Warshall algorithm can be modified a little to compute the transitive closure of a directed graph. This method involse the substitution of the logical operations $\vee$ (logical OR) and $\wedge$ (logical AND) for the arithmetic operations min and + in the Floyd-Warshall algorithm. Hence the running time is also $O(V^3)$.

5.3.2 Johnson's algorithm

There is another simple technique to compute all-pairs shortest paths: run Dijkstra's algorithm using each vertex as the source. Then the running time is $O(V^2 \lg V + VE)$ (assume using Fibonacci-heap). But there is a problem: the Dijkstra's algorithm can not be applied if a negative weight exists. The key idea of Johnson's algorithm is to circumvent this problem by ***reweighting*** all edges so that the new weight $\widehat{w}$ of each edge is non-negative. The new set of edge weights $\widehat{w}$ must satisfy two important properties,

1. For all pairs of vertices $u, v \in V$, a path p is a shortest path from u to v using weight function w iff p is a shortest path from u to v using weight function $\widehat{w}$.

2. For all edges (u, v), the new weight $\widehat{w}(u, v)$ is nonnegative.

Johnson's algorithm defines the new weight function as follows

$$\widehat{w}(u, v) = w(u, v) + h(u) - h(v).$$

in which h is a function mapping vertices to real numbers. And this new weight function is guaranteed to satisfy the above first property. Then we want $\widehat{w}(u, v)$ to be nonnegative. This is equivalent to finding some h so that $h(v) - h(u) \leq w(u, v)$ holds. In fact this problem is a special case of linear programming, which can be solved using Bellman-Ford algorithm.

The Johnson's algorithm is described as follows.

S1. Solve the system of difference constraints: $h(v) - h(u) \leq w(u, v)$ by using Bellman-Ford algorithm. If this constraints can not be satisfied then print an error message and exit.

S2. Reweighting. Assign a new weight $\widehat{w}(u, v) = w(u, v) + h(u) - h(v)$ to each edge.

S3. Run Dijkstra's algorithm $|V|$ times to compute $\widehat{\delta}(u, v)$ for all $u, v \in V[G]$.

S4. Compute the shortest path $\delta(u, v)$ of original graph which is equal to $\widehat{\delta}(u, v) + h(v) - h(u)$.

If the min-priority queue in Dijkstra's algorithm is implemented by a Fibonacci heap, the running time of Johnson's algorithm is $O(V^2 \lg V + VE)$. The simpler binary min-heap implementation yields a running time of $O(VE \lg V)$, which is still asymptotically faster than the Flod-Warshall algorithm if the graph is sparse.

5.4 Depth-First search and Topological Sort

Topological sort of a dag can be solved by DFS. The algorithm is as follows.

```
TOPOLOGICAL-SORT($G$)
1  call DFS($G$) to compute finishing times $f[v]$ for each vertex $v$ as each vertex is finised,
          insert it onto the front of a linked list
2  return the linked list of vertices
```

The running time is $\Theta(V + E)$. Note topological sort can be only applied on a directed acyclic graph. There is not a topological sort order if a graph has a cycle. And this situation can be detected by DFS algorithm. A directed graph G is acyclic iff a dpeth-first search of G yields no back edges.

5.5 Depth-First Search and Strongly connected components

Let G^{SCC} be the component graph of a directed graph G and G^T be the transpose of G. The key idea of the algorithm computing strongly connected components by DFS is: first, compute the topologically sorted order of G by which we can deduce the topologically sorted order of G^{SCC} (see Lemma 22.14 of [CLRS 2001]); second, call DFS(G^T) visiting vertices in the topologically sorted order of G^{SCC}, which is the reverse of the topologically sorted order of $(G^T)^{SCC}$. Then the vertices of each tree in the depth-first forest form a separate strongly connected component.

```
STRONGLY-CONNECTED-COMPONENTS($G$)
1  call DFS($G$) to compute finishing times $f[u]$ for each vertex $u$
2  compute $G^T$
3  call DFS($G^T$), but in the main loop of DFS, consider the vertices
           in order of decreasing $f[u]$ (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in
           line 3 as a separate strongly connected component
```

The running time of this algorithm is $\Theta(V + E)$.

For a undirected graph, the connected components can be computed by running DFS once and then the vertices of each tree in the depth-first forest is a separate connected component.

6 Randomness

6.1 Random Sample

Problem Definition. The input consists of two integers m and n, with $m < n$. The output is a sorted list of m random integers in the range $0..n-1$ in which no integer occurs more than once.

6.1.1 Algorithm S1

This algorithm is straightforward: insert random integers into an initially empty set until there are enough. It comes from Section 12.3 of [Bently 2000].

```
Random-Select$(m, n)$
1  $S \leftarrow \emptyset$
2  while $size[S] < m$
3      $\triangleright$ Invariant: $S$ contains a random sample of $size[S]$ integers in
            the range $0..n-1$.
4      do $t \leftarrow$ rand()% $n$
5          if $t \notin S$
6             then $S \leftarrow S \cup \{t\}$
7  print the elements of $S$ in sorted order
```

This algorithm has a flaw: if the random generator `RandInt` is not truly random, the algorithm won't even terminate.

6.1.2 Algorithm K

The following algorithm comes from Section 3.4.2 of Knuth's *Seminumerical Algorithms*.

```
Random-Select$(m, n)$
1  $s \leftarrow m$
2  $r \leftarrow n$
3  for $i \in [0, n)$
4      do if (rand()% $r$) $< s$
5             then print $i$
6                  $s \leftarrow s - 1$
7          $r \leftarrow r - 1$
```

6.1.3 Algorithm S2

Another way to generate a sorted subset of random integers is to shuffle an n-element array that contains the numbers $0..n-1$, and then sort the first m for the output.

This algorithm also comes from Section 12.3 of [Bently 2000].

```
Random-Select$(m, n)$
1  for $i \in [0, n)$
2      do $x[i] \leftarrow i$
3  for $i \in [0, m)$
4      do $j \leftarrow$ RandInt$(i, n-1)$
5          Exchange $x[i] \longleftrightarrow x[j]$
6  Sort $x[0..m-1]$ into increasing order and then print them.
```

This algorithm uses $\Theta(n)$ space and $O(n + m \log m)$ time.

6.1.4 Algorithm F1

This algorithm is a improved version of 6.1.1 due to Robert Floyd. It only uses exactly m calls of `RandInt`. See [Bently 1987] for more details.

```
RANDOM-SELECT($m, n$)
1  $S \leftarrow \emptyset$
2  for $j \leftarrow n - m + 1$ to $n$
3      do $t \leftarrow$ RandInt$(1, j)$
4          if $t \notin S$
5              then
6                  $S \leftarrow S \cup \{t\}$
7              else
8                  $S \leftarrow S \cup \{j\}$
```

This algorithm uses $O(1)$ space and $O(m)$ time.

6.2 Permutation

Problem Definition. The input consists of two integers m and n, with $m < n$. The output is a random sequence of m integers in the range $0..n - 1$ in which no integer occurs more than once.

It is also known as *shuffle*.

6.2.1 Algorithm F2

This algorithm is similar to the previous algorithm F1. See [Bently 1987] for more details.

```
RANDOM-PERMUTATION($m, n$)
1  $S \leftarrow \emptyset$
2  for $j \leftarrow n - m + 1$ to $n$
3      do $t \leftarrow$ RandInt$(1, j)$
4          if $t \notin S$
5              then
6                  prefix $t$ to $S$
7              else
8                  insert $j$ in $S$ after $t$
```

6.2.2 Fisher-Yates shuffle

Randomly permute N elements by exchanging each element e_i with a random element from i to N. It runs in linear time.

```
RANDOM-PERMUTATION($(m, n)$)
1  for $i \in [0, n)$
2        do $x[i] \leftarrow i$
3  for $i \in [0, m)$
4        do $j \leftarrow \text{RandInt}(i, n-1)$
5           Exchange $x[i] \longleftrightarrow x[j]$
```

References

[CLRS 2001] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*, Second Edition, The MIT Press, 2001.

[Sedgewick 1998] Robert Sedgewick, *Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching*, Third Edtion, Addison-Wesley, 1998.

[Sleator 1985] D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM 32*, 1985.

[Ukkonen] Esko Ukkonen, "On-line construction of suffix trees".

[Bently 2000] Jon Bentley, *Programming Pearls*, Second Edition, Addison Wesley, 2000.

[Bently 1987] Jon Bentley, "Programming Pearls: a sample of brilliance", *Communications of the ACM*, Vol. 30:9, Sep. 1987.