Data Structure and Algorithm Analysis(H)
Southern University of Science and Technology
Mengxuan Wu
12212006

Final Review

Mengxuan Wu

1 Sorting Algorithms

1.1 Sorting Based on Comparison

1.1.1 Insertion Sort

Inserting sort maintains a subarray of sorted element. At each iteration, it inserts a new element into the subarray on the correct position.
Runtime: $\Omega(n)$, $O(n^2)$
Correctness: Proof by loop invariant: at the start of each iteration of the for loop, the subarray $A[1..j-1]$ consists of the element originally in $A[1..j-1]$, but in sorted order.
Inplace: Yes
Stable: Yes

```
INSERTION-SORT($A$)
1. for $j = 2$ to $A.length$
2.    $key = A[j]$
3.    $i = j - 1$
4.    while $i > 0$ and $A[i] > key$
5.       $A[i+1] = A[i]$
6.       $i = i - 1$
7.    $A[i+1] = key$
```

1.1.2 Selection Sort

Selection sort maintains a subarray of sorted element. At each iteration, it selects the smallest element in the unsorted subarray, and swap it with the first element in the unsorted subarray.
Runtime: $\Theta(n^2)$
Correctness: Proof by loop invariant: at the start of each iteration of the for loop, the subarray $A[1..i-1]$ consists of the $i-1$ smallest elements.
Inplace: Yes
Stable: No (example: $[2_a, 2_b, 1]$)

```
SELECTION-SORT($A$)
1. for $i = 1$ to $A.length - 1$
2.     $min = i$
3.     for $j = i + 1$ to $A.length$
4.         if $A[j] < A[min]$
5.             $min = j$
6.     exchange $A[i]$ with $A[min]$
```

1.1.3 Merge Sort

Merge sort divides the array into two parts, sort each part recursively, and then merge the two sorted parts.
Runtime: $\Theta(n \log n)$
Correctness:

- Merge: By loop invariant: At the start of each iteration of the for loop, the subarray $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order.
- Merge sort: By induction: The two smaller subarrays are sorted correctly with induction hypothesis, and the merge step merges the two subarrays correctly.

Inplace: No
Stable: Yes

```
MERGE($A, p, q, r$)
1.  $n_1 = q - p + 1$
2.  $n_2 = r - q$
3.  let $L[1 \ldots n_1 + 1]$ and $R[1 \ldots n_2 + 1]$ be new arrays
4.  for $i = 1$ to $n_1$
5.      $L[i] = A[p + i - 1]$
6.  for $j = 1$ to $n_2$
7.      $R[j] = A[q + j]$
8.  $L[n_1 + 1] = \infty$
9.  $R[n_2 + 1] = \infty$
10. $i = 1$
11. $j = 1$
12. for $k = p$ to $r$
13.     if $L[i] \leq R[j]$
14.         $A[k] = L[i]$
15.         $i = i + 1$
16.     else
17.         $A[k] = R[j]$
18.         $j = j + 1$
```

```
MERGE-SORT($A, p, r$)
1.  if $p < r$
2.      $q = \lfloor (p+r)/2 \rfloor$
3.      MERGE-SORT($A, p, q$)
4.      MERGE-SORT($A, q+1, r$)
5.      MERGE($A, p, q, r$)
```

1.1.4 Heap Sort

A heap is a binary tree with the following properties (assume the array starts from index 1):

1. $A[Parent(i)] = \lfloor i/2 \rfloor$.
2. $A[Left(i)] = 2i$.
3. $A[Right(i)] = 2i + 1$.
4. Max-heap property: $A[Parent(i)] \geq A[i]$.
5. Min-heap property: $A[Parent(i)] \leq A[i]$.

Max-heapify:
Assume the binary trees rooted at $Left(i)$ and $Right(i)$ are max-heaps, but max-heap property may be violated at node i. We let the value at node i float down, so that the subtree rooted at node i becomes a max-heap.
Runtime: $O(\log n)$ (height of the tree)
Correctness: By induction: Assume max-heapify works for $h = i - 1$. Now we consider $h = i$. After we swap the value at node i with the value at node *largest*, this maintains the max-heap property for node i. Now one of the subtrees rooted at $Left(i)$ or $Right(i)$ may violate the max-heap property. By induction hypothesis, we know that we can call max-heapify to make the subtree rooted at $Left(i)$ or $Right(i)$ a max-heap.

```
MAX-HEAPIFY($A, i$)
1.  $l = Left(i)$
2.  $r = Right(i)$
3.  if $l \leq A.heap-size$ and $A[l] > A[i]$
4.      $largest = l$
5.  else $largest = i$
6.  if $r \leq A.heap-size$ and $A[r] > A[largest]$
7.      $largest = r$
8.  if $largest \neq i$
9.      exchange $A[i]$ with $A[largest]$
10.     MAX-HEAPIFY($A, largest$)
```

Build-max-heap:

Build-max-heap is used to build a max-heap from an unordered array. We can only do bottom-up, since we need to maintain the max-heap property for all nodes.
Runtime: $O(n)$ The maximum number of nodes of height h is $\lfloor n/2^{h+1} \rfloor$. Hence, the total runtime is $\sum_{h=0}^{\lfloor \log n \rfloor} \lfloor n/2^{h+1} \rfloor O(h) = O(n)$.
Correctness: By loop invariant: At the start of each iteration of the for loop, each node $i+1, i+2, \ldots, n$ have the max-heap property.

```
BUILD-MAX-HEAP($A$)
1. $A.heap-size = A.length$
2. for $i = \lfloor A.length/2 \rfloor$ downto 1
3.     MAX-HEAPIFY($A, i$)
```

Heap-sort:
We first build a max-heap from the array. Then we can extract the maximum element from the heap, and put it at the end of the array. Now we exchange the first element with the last element, decrease the heap size by 1, and call max-heapify to maintain the max-heap property.
Runtime: $\Theta(n \log n)$
Correctness: By loop invariant: At the start of each iteration of the for loop, the subarray $A[1..i]$ is a max-heap that contains the first i smallest elements, and the subarray $A[i+1..n]$ contains the $n-i$ largest elements.
Inplace: Yes
Stable: No (example: $[1_a, 1_b]$)

```
HEAP-SORT($A$)
1. BUILD-MAX-HEAP($A$)
2. for $i = A.length$ downto 2
3.     exchange $A[1]$ with $A[i]$
4.     $A.heap-size = A.heap-size - 1$
5.     MAX-HEAPIFY($A, 1$)
```

1.1.5 Priority Queue

A priority queue is very similar to a heap, but with the following operations:

1. INSERT(Q, x): insert x into Q.
2. MAXIMUM(Q): return the element with the largest key.
3. EXTRACT-MAX(Q): remove and return the element with the largest key.
4. INCREASE-KEY(Q, x, k): increase the key of x to k.

The only new operation is INCREASE-KEY, which can be implemented by "bubbling up" the element. Its runtime is $O(\log n)$.

1.1.6 Quick Sort

Partition:
Partition is used to partition an array into two parts, such that all elements in the first part are smaller than the pivot, and all elements in the second part are larger than the pivot.
Runtime: Worst case: $O(n^2)$, Best case: $O(n \log n)$, Average case: $O(n \log n)$
Correctness: By loop invariant: At the start of each iteration of the for loop, the subarray $A[p..j-1]$ consists of elements smaller than the pivot, and the subarray $A[j+1..r]$ consists of elements larger than the pivot.

```
PARTITION($A, p, r$)
1. $x = A[r]$
2. $i = p - 1$
3. for $j = p$ to $r - 1$
4.     if $A[j] \leq x$
5.         $i = i + 1$
6.         exchange $A[i]$ with $A[j]$
7. exchange $A[i+1]$ with $A[r]$
8. return $i + 1$
```

```
RANDOMIZED-PARTITION($A, p, r$)
1. $i = Random(p, r)$
2. exchange $A[r]$ with $A[i]$
3. return PARTITION($A, p, r$)
```

However, there are some drawbacks of this implementation: this partition algorithm behaves poorly if the input array contains many equal elements.

Quick sort:
Quick sort divides the array into two parts, and sort each part recursively. It is very similar to merge sort, but it does not need to combine the two parts.
Runtime: Worst case: $O(n^2)$, Best case: $O(n \log n)$, Average case: $O(n \log n)$
Correctness: Similar to merge sort. By induction, the two smaller subarrays are sorted correctly with induction hypothesis, and the partition step partitions the array correctly.
Inplace: Yes
Stable: No

```
QUICK-SORT($A, p, r$)
1. if $p < r$
2.     $q$ = PARTITION($A, p, r$)
3.     QUICK-SORT($A, p, q - 1$)
4.     QUICK-SORT($A, q + 1, r$)
```

The randomized version is omitted here.

Randomization:
We can estimate the runtime of quick sort by the expected number of comparisons.
For a partition with fixed ratio, we can prove that the expected number of comparisons is always $O(n \log n)$. Suppose the ratio is $1 : 9$, then we can draw a tree for the partition. We can find at each level, if on both side all nodes exists, the number of comparisons is n. And the maximum number of levels is $\log_{10/9} n$. In total, the number of comparisons is $O(n \log n)$.

The expected number of comparisons for randomized quick sort can be calculated as follows:
Rename the array as $Z_1, Z_2, \ldots, Z_n$, where $Z_1 < Z_2 < \cdots < Z_n$. Then the possibility that Z_i is compared with Z_j is $\frac{2}{j-i+1}$ (suppose $i < j$).
Hence, the expected number of comparisons is:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \leq \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} \leq 2n \sum_{k=2}^{n} \frac{1}{k} \leq 2n \ln n$$

1.1.7 Lower Bound for Sorting Based on Comparison

We can use decision tree to prove the lower bound for sorting.
Let the input size be n, suppose the input is a permutation of $1, 2, \ldots, n$, there are $n!$ possible orders of the input. Since each input must correspond to a leaf in the decision tree, the number of leaves is $n!$. Hence, the height of the decision tree is $\Omega(\log n!)$.
The lower bound of worst case runtime is the length of the longest path in the decision tree, which is the height of the decision tree. By Stirling's approximation, $\log n! = \Omega(n \log n)$.

1.2 Sorting Based on Counting

1.2.1 Counting Sort

Counting Sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k. It works by counting the number of elements of each value.
Runtime: $\Theta(n + k)$
Correctness: By loop invariant: At the start of each iteration of the for loop, the last element in A with value i that has not yet been copied to B belongs to $B[C[i]]$.
Inplace: No
Stable: Yes

```
COUNTING-SORT($A, B, k$)
1.  let $C[0..k]$ be a new array
2.  for $i = 0$ to $k$
3.      $C[i] = 0$
4.  for $j = 1$ to $A.length$
5.      $C[A[j]] = C[A[j]] + 1$
6.  for $i = 1$ to $k$
7.      $C[i] = C[i] + C[i-1]$
8.  for $j = A.length$ downto 1
9.      $B[C[A[j]]] = A[j]$
10.     $C[A[j]] = C[A[j]] - 1$
```

1.2.2 Radix Sort

Radix Sort sorts the elements' digit by digit.
Runtime: $\Theta(d(n+k))$, where d is the number of digits.
Correctness: At each iteration of the for loop, the elements are sorted on the last $i-1$ digits.
Inplace: No
Stable: Yes

```
RADIX-SORT($A, d$)
1.  for $i = 1$ to $d$
2.      use a stable sort to sort array $A$ on digit $i$
```

Attention: radix sort begins from the least significant digit.
When d is a constant and $k = O(n)$, radix sort runs in $\Theta(n)$. In more general cases, we can decide how to choose d and k to make the radix sort faster.
Without loss of generality, we assume the number is in base 2. The for a b bit number, we can divide it into $\lceil b/d \rceil$ groups, each group has d bits. Then we can use counting sort to sort each group. Hence, the runtime is $\Theta(\lceil b/d \rceil(n + 2^d))$.
When $b < \lfloor \log n \rfloor$, we can choose $d = b$. Then the runtime is $\Theta(\frac{b}{b}(n + 2^b)) = \Theta(n)$.
When $b \geq \lfloor \log n \rfloor$, we can choose $d = \lfloor \log n \rfloor$. Then the runtime is $\Theta(\frac{b}{\log n}(n + 2^{\log n})) = \Theta(\frac{bn}{\log n})$.

2 Elementary Data Structures

2.1 Stack

In a stack, only the top element can be accessed. It follows the Last In First Out (LIFO) principle.
Its operations are:

1. PUSH(S, x): insert x into S.
2. POP(S): remove and return the top element of S.

Attention: *S.top* points at the top element of the stack, not an empty element. We initialize $S.top = 0$.

2.2 Queue

In a queue, elements are inserted at the tail, and removed from the head. A queue follows the First In First Out (FIFO) principle.
Its operations are:

1. ENQUEUE(Q, x): insert x into Q.
2. DEQUEUE(Q): remove and return the head element of Q.

Attention: $Q.head$ points at the head element of the queue, but $Q.tail$ points at the empty element after the tail element. We initialize $Q.head = Q.tail = 1$. It should be noted that a queue implemented in this way with an array of size n can only store $n - 1$ elements. Since if it stores n elements, then $Q.head = Q.tail$, which means we cannot distinguish between an empty queue and a full queue.

2.3 Linked List

In a linked list, every element contains two pointers: one points at the previous element, and the other points at the next element. It is also called a doubly linked list.
Linked list makes it easy to insert or delete an element. Here are the pseudocode for insertion and deletion.

```
LIST-INSERT$(L, x)$
1.  $x.next = L.head$
2.  if $L.head \neq NIL$
3.      $L.head.prev = x$
4.  $L.head = x$
5.  $x.prev = NIL$
```

```
LIST-DELETE$(L, x)$
1.  if $x.prev \neq NIL$
2.      $x.prev.next = x.next$
3.  else $L.head = x.next$
4.  if $x.next \neq NIL$
5.      $x.next.prev = x.prev$
```

3 Binary Search Trees

3.1 General Binary Search Tree

A binary search tree is a binary tree in which for each node x, the values of all the keys in the left subtree of x are smaller than the key of x, and the values of all the keys in the right subtree of x are larger than the key of x.
Here are some properties of a binary search tree:

- The **depth** of a node x is the number of edges on the simple path from the root to x. (Top Down)

- A level of a tree is all nodes at the same depth.
- The **height** of a node x is the number of edges on the longest simple downward path from x to a leaf. (Bottom Up)
- The height of a tree is the height of the root.
- A binary search tree is **full** if every node has either zero or two children.
- A binary search tree is **complete** if it is full and all leaves are at the same level.

3.1.1 Successor

The successor of a node x is the node with the smallest key greater than $x.key$. If a node has a right subtree, then its successor is the leftmost node in its right subtree. If it does not, then its successor is the lowest ancestor of x whose left child is also an ancestor of x. Or to say, to find the first "right parent" of x. Notice how this algorithm produce NIL if x is the largest node.

```
TREE-SUCCESSOR($x$)
1. if $x.right \neq NIL$
2.     return TREE-MINIMUM($x.right$)
3. $y = x.p$
4. while $y \neq NIL$ and $x = y.right$
5.     $x = y$
6.     $y = y.p$
7. return $y$
```

The predecessor can be found similarly.

3.1.2 Insert

Insertion is simple, we do binary search to find a position, and then simply append the new node. Notice that we want to keep record of the parent of current node, so that we can find the new node's parent.

```
TREE-INSERT($T, z$)
 1. $x = T.root$
 2. $y = NIL$
 3. while $x \neq NIL$
 4.     $y = x$
 5.     if $z.key < x.key$
 6.         $x = x.left$
 7.     else $x = x.right$
 8. $z.p = y$
 9. if $y = NIL$
10.     $T.root = z$
11. else if $z.key < y.key$
12.     $y.left = z$
13. else $y.right = z$
```

3.1.3 Delete

Deletion is more complicated. We need to consider multiple cases:

1. If z is a leaf, we can simply remove it.
2. If z has only one child, we replace z with its child.
3. If z has two children, we replace z with its successor. Why? Since the right subtree is not empty, the successor must be in the right subtree. The successor cannot have a left child, otherwise the left child will be the successor. Then we can replace z with its successor, and then delete the successor (no left child makes it easy to delete). Since the successor has the smallest key that is larger than $z.key$, it is larger than all keys in the left subtree of z, and smaller than all keys in the right subtree of z. Then the binary search tree property is maintained.

```
TREE-DELETE($T, z$)
1.  if $z.left = NIL$
2.      TRANSPLANT($T, z, z.right$)
3.  else if $z.right = NIL$
4.      TRANSPLANT($T, z, z.left$)
5.  else
6.      $y$ = TREE-MINIMUM($z.right$)
7.      if $y.p \neq z$
8.          TRANSPLANT($T, y, y.right$)
9.          $y.right = z.right$
10.         $y.right.p = y$
11.     TRANSPLANT($T, z, y$)
12.     $y.left = z.left$
13.     $y.left.p = y$
```

3.2 Tree Walk

There are three ways to walk a tree:

1. Preorder: root, left, right.
2. Inorder: left, root, right.
3. Postorder: left, right, root.

Since it takes $\Theta(1)$ time to process each node, the runtime is $\Theta(n)$.

```
INORDER-TREE-WALK($x$)
1.  if $x \neq NIL$
2.      INORDER-TREE-WALK($x.left$)
3.      print $x.key$
4.      INORDER-TREE-WALK($x.right$)
```

```
PREORDER-TREE-WALK($x$)
1. if $x \neq NIL$
2.     print $x.key$
3.     PREORDER-TREE-WALK($x.left$)
4.     PREORDER-TREE-WALK($x.right$)
```

```
POSTORDER-TREE-WALK($x$)
1. if $x \neq NIL$
2.     POSTORDER-TREE-WALK($x.left$)
3.     POSTORDER-TREE-WALK($x.right$)
4.     print $x.key$
```

3.3 AVL Trees

A binary search tree can be unbalanced and degenerate into a linked list, which makes it inefficient. An AVL tree is a self-balancing binary search tree. It requires that the heights of the two child subtrees of any node differ by at most 1. This can be implemented by adding a balance factor to each node.
An AVL tree with n nodes has at most height $1.44 \log n$, which is pretty close to the optimal height $\log n$.

3.3.1 Minimum Number of Nodes in AVL Tree

Consider the minimal number of nodes in an AVL tree of height h. We have:

$$N(h) = N(h-1) + N(h-2) + 1$$

This can be interpreted as: to find the minimal tree of height h, we split the AVL tree into three parts: the root, the left subtree of height $h-1$, and the right subtree of height $h-2$. For the root, it satisfies the AVL property, since the height of the left subtree and the right subtree differ by 1. For the two subtrees, to make the number of nodes minimal, they must be the minimal AVL trees of height $h-1$ and $h-2$.
This number is related to the Fibonacci sequence, and we have $N(h) = F_{h+2} - 1$ (assume the Fibonacci sequence starts with $F_0 = F_1 = 1$).

3.3.2 Insertion

Runtime: $O(\log n)$.
Firstly, we need to define the balance factor of a node x:

$$bal(x) = height(x.left) - height(x.right)$$

Without loss of generality, we assume the insertion inserts a node into the right subtree. We use v to denote the node on the path from the inserted node to the root, and x to denote the right child of v. Here are the cases:
Case 1: $bal(v) = 1$. After insertion, $bal(v) = 0$. The height of v does not change, so we can stop.
Case 2: $bal(v) = 0$. After insertion, $bal(v) = -1$. The height of v increases by 1, so we need to check the parent of v, so we run the algorithm recursively.

Case 3: $bal(v) = -1$. After insertion, $bal(v) = -2$. We need to do a rotation to make v balanced.
Subcase 3.1: The inserted node is in the right subtree of x. We do a left rotation on v. After the rotation, v becomes the left child of x. We then update the balance factor of v and x: $bal(v) = 0$, $bal(x) = 0$. Since the height of v does not change, we can stop.
Subcase 3.2: The inserted node is in the left subtree of x. Let w denote the left child of x. We can find v, x, and w form a zig-zag pattern. We do a right rotation on x, and then a left rotation on v. After the rotation, v becomes the left child of w, and x becomes the right child of w. We then update the balance factor of v, x, and w:

- If $bal(w)$ was 0, then $bal(v) = 0$, $bal(x) = 0$, $bal(w) = 0$.
- If $bal(w)$ was 1, then $bal(v) = 0$, $bal(x) = -1$, $bal(w) = 0$.
- If $bal(w)$ was -1, then $bal(v) = 1$, $bal(x) = 0$, $bal(w) = 0$.

3.3.3 Deletion

Runtime: $O(\log n)$.
Deletion is similar to insertion.
Without loss of generality, we assume the deletion deletes a node from the left subtree. We also need to consider three cases:
Case 1: $bal(v) = 1$. After deletion, $bal(v) = 0$. The height of v decreases by 1, so we need to check the parent of v, so we run the algorithm recursively.
Case 2: $bal(v) = 0$. After deletion, $bal(v) = 1$. The height of v does not change, so we can stop.
Case 3: $bal(v) = -1$. After deletion, $bal(v) = -2$. We need to do a rotation to make v balanced.
Subcase 3.1: $bal(x) \in \{0, -1\}$. The deleted node is in the left subtree of v. We do a left rotation on v. After the rotation, v becomes the left child of x. It depends on the balance factor of x to decide whether we can stop.
Subcase 3.2: $bal(x) = 1$. The deleted node is in the left subtree of v. Let w denote the left child of x. We do a right rotation on x, and then a left rotation on v. After the rotation, v becomes the left child of w, and x becomes the right child of w. The height of v is decreased by 1, so we run the algorithm recursively.

It should be noted that if the deleted operation involves finding the successor, then we need to update the balance factor from the successor's parent.

4 Dynamic Programming

When we want to solve a problem recursively, we may find that some subproblems are solved repeatedly. For example in the Fibonacci sequence, many numbers are calculated repeatedly. So instead of solving the subproblems repeatedly, we can store the solutions in a table, and then use the table to solve the problem.
To implement dynamic programming, we often need the problem to be **optimal substructure** and **overlapping subproblems**.

- Optimal substructure: The solution to the subproblem used within the optimal solution must be optimal.
- Overlapping subproblems: the problem can be broken down into subproblems which are reused several times.

4.1 Rod Cutting

Given a rod of length n and a table of prices p_i for the rod of length i, we want to find the maximum total revenue r_n for the rod. Let r_i denote the maximum total revenue for the rod of length i. Then we have the **Bellman equation**:

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

It should be noted that there is another Bellman equation, but we will not discuss it here:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

4.1.1 Different Implementations

Recursive Implementation

```
CUT-ROD($p, n$)
1. if $n = 0$
2.     return 0
3. $q = -\infty$
4. for $i = 1$ to $n$
5.     $q = \max(q, p_i + \text{CUT-ROD}(p, n - i))$
6. return $q$
```

Runtime: Let $T(n)$ denote the number of times CUT-ROD is called. Then we have $T(n) = 1 + \sum_{i=1}^{n} T(n - i)$. Solve this recurrence relation, we have $T(n) = 2^n$.
Bottom Up Implementation

```
BOTTOM-UP-CUT-ROD($p, n$)
1. $r_0 = 0$
2. for $j = 1$ to $n$
3.     $q = -\infty$
4.     for $i = 1$ to $j$
5.         $q = \max(q, p[i] + r[j - i])$
6.     $r[j] = q$
7. return $r[n]$
```

In dynamic programming, correctly initializing the table is very important. In this case, we initialize $r_0 = 0$.
Runtime: $\Theta(n^2)$.
Top Down Implementation
This is also called **memoization**, we first check whether the subproblem has been solved, and if so, we return the solution.

```
MEMOIZED-CUT-ROD(p, n)
1. if $r_n \geq 0$
2.     return $r_n$
3. $q = -\infty$
4. for $i = 1$ to $n$
5.     $q = \max(q, p_i + \text{MEMOIZED-CUT-ROD}(p, n - i))$
6. $r_n = q$
7. return $q$
```

Runtime: $\Theta(n^2)$.
It is worth noting that although in this particular case, the bottom up implementation is as fast as the top down implementation, in general, the memoization method is faster than the bottom up method. This is because the memoization method only solves the subproblems that are needed, while the bottom up method solves all subproblems.

4.2 Subproblem Graph

We can draw a graph to show the subproblems and their dependencies. For example, in the rod cutting problem, we can draw a graph as follows:

$$\begin{tikzcd}
r_4 \arrow[d] \arrow[dd, bend left] \arrow[ddd, bend left] \arrow[dddd, bend left] \\
r_3 \arrow[d] \arrow[dd, bend right] \arrow[ddd, bend right] \\
r_2 \arrow[d] \arrow[dd, bend left] \\
r_1 \arrow[d] \\
r_0
\end{tikzcd}$$

When we estimate the runtime of a dynamic programming algorithm, it is often linear in the number of vertices and edges in the subproblem graph.

4.3 Reconstruct The Solution

In the rod cutting problem, we want to find the optimal solution, not just the optimal revenue. We can use an auxiliary array s to store the optimal size of the first piece to cut off. Then we can reconstruct the solution by tracing back the array s.
For the bottom up implementation, we can implement a EXTENDED-BOTTOM-UP-CUT-ROD to return both the optimal revenue and the auxiliary array s.

```
EXTENDED-BOTTOM-UP-CUT-ROD($p, n$)
1. $r_0 = 0$
2. for $j = 1$ to $n$
3.     $q = -\infty$
4.     for $i = 1$ to $j$
5.         if $q < p_i + r_{j-i}$
6.             $q = p_i + r_{j-i}$
7.             $s_j = i$
8.     $r_j = q$
9. return $r_n, s$
```

Then we can print the solution by:

```
PRINT-CUT-ROD-SOLUTION($p, n$)
1. $(r, s)$ = EXTENDED-BOTTOM-UP-CUT-ROD($p, n$)
2. while $n > 0$
3.     print $s[n]$
4.     $n = n - s[n]$
```

5 Greedy Algorithms

A greedy algorithm always makes the choice that looks best at the moment. A greedy algorithm always have a corresponding dynamic programming algorithm, but the converse is not true (otherwise we do not need dynamic programming).

5.1 Activity Selection

We have a set of activities $S = \{a_1, a_2, \ldots, a_n\}$, where each activity a_i has a start time s_i and a finish time f_i. Two activities a_i and a_j are **compatible** if $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. The goal is to find the maximum-size subset of mutually compatible activities (**not the set of activities with the maximum total time**).

5.1.1 Dynamic Programming

We can use dynamic programming to solve this problem. Let S_{ij} denote the set of activities that start after a_i finishes and finish before a_j starts, and let A_{ij} denote the maximum-size subset of mutually compatible activities in S_{ij}.
The activity selection problem has optimal substructure: Suppose a_k is in A_{ij}, then $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$. Then $|A_{ij}| = |A_{ik}| + 1 + |A_{kj}|$. Then we have the Bellman equation:

$$c[i, j] = \max_{a_k \in S_{ij}} (c[i, k] + c[k, j] + 1)$$

5.1.2 Greedy Choice

We can also use a greedy algorithm to solve this problem. We first sort the activities by their finish time, and then we choose the first activity. Then we choose the first activity that is compatible with the first activity, and so on.

In other words, we always choose the activity that finishes first.
Loop version:

```
GREEDY-ACTIVITY-SELECTOR($s, f$)
1. $n = s.length$
2. $A = \{a_1\}$
3. $k = 1$
4. for $m = 2$ to $n$
5.    if $s_m \geq f_k$
6.       $A = A \cup \{a_m\}$
7.       $k = m$
8. return $A$
```

Recursive version:

```
RECURSIVE-ACTIVITY-SELECTOR($s, f, k, n$)
1. $m = k + 1$
2. while $m \leq n$ and $s_m < f_k$
3.    $m = m + 1$
4. if $m \leq n$
5.    return $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR($s, f, m, n$)
6. else
7.    return $\emptyset$
```

Runtime: $\Theta(n)$.

5.1.3 Proof of Correctness

We assume that the activities are sorted by their finish time. Let S_k denote the set of activities the starts after a_k finishes, and A_k denote the maximum-size subset of mutually compatible activities in S_k. We want to prove that if a_m is the activity that finishes first in the set S_k, then a_k is in some A_k.
Let a_j denote the activity that finishes first in A_k. If $a_j = a_m$, then we are done. If $a_j \neq a_k$, we can exchange a_j and a_m and not cause any conflict. Since the number of activities in A_k does not change, the new set is still a maximum-size subset of mutually compatible activities in S_k.

5.2 General Scheme of Greedy Algorithms

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.

5.3 Coin Changing

Given a set of coins $S = \{v_1, v_2, \ldots, v_n\}$, where $v_1 < v_2 < \cdots < v_n$, and a value V, we want to find the minimum number of coins that sum to V. This problem does not always have a greedy solution.

The usual greedy choice is to choose the largest coin that does not exceed the remaining value. This sometimes fails, for example: $S = \{1, 3, 4\}$, $V = 6$. The greedy algorithm will choose $4, 1, 1$, but the optimal solution is $3, 3$.

5.4 When Greedy Algorithms Fail

There are many cases when greedy algorithms fail. For example: maze problem, traveling salesman problem, 0-1 knapsack problem, etc.
Take 0-1 knapsack problem as an example. If we have a knapsack with capacity 50 and three items:

- Item 1: $v_1 = 60$, $w_1 = 10$
- Item 2: $v_2 = 100$, $w_2 = 20$
- Item 3: $v_3 = 120$, $w_3 = 30$

If the greedy choice is to choose the item with the largest value per unit weight, then we will choose item 1 and 2, and the total value is 160. However, the optimal solution is to choose item 2 and 3, and the total value is 220.
It should be noted that the **fractional knapsack problem** does have a greedy solution.

6 Graph Algorithms

We usually use two ways to represent a graph: adjacency list and adjacency matrix. To choose which representation to use, we need to consider the density of the graph. A graph is **sparse** if $|E| = o(|V|^2)$, and **dense** if $|E| = \Theta(|V|^2)$. For sparse graphs, we use adjacency list, and for dense graphs, we use adjacency matrix.

6.1 Breadth First Search

In breadth first search, we use a queue to store the vertices. We extract a vertex from the queue, and then add all its adjacent vertices to the queue.
We assign each vertex a color: white, gray, or black.

- White: undiscovered.
- Gray: discovered but not finished.
- Black: finished.

It is worth noting that the two colors gray and black are not necessary and can be combined into one color. The difference is purely for the purpose of analysis.
Each vertex has three attributes: distance, parent, and color.

```
BFS(G, s)
1.  for each vertex $u \in G.V - \{s\}$
2.      $u.color = WHITE$
3.      $u.d = \infty$
4.      $u.\pi = NIL$
5.  $s.color = GRAY$
6.  $s.d = 0$
7.  $s.\pi = NIL$
8.  $Q = \emptyset$
9.  ENQUEUE($Q, s$)
10. while $Q \neq \emptyset$
11.     $u$ = DEQUEUE($Q$)
12.     for each vertex $v \in G.Adj[u]$
13.         if $v.color == WHITE$
14.             $v.color = GRAY$
15.             $v.d = u.d + 1$
16.             $v.\pi = u$
17.             ENQUEUE($Q, v$)
18.     $u.color = BLACK$
```

Runtime: $O(|V| + |E|)$. This is because each vertex is enqueued and dequeued at most once, which takes $O(|V|)$ time. Each edge is examined at most twice, which takes $O(|E|)$ time (assume we use adjacency list).

Correctness: The loop invariant is: At the start of each iteration of the while loop, the queue Q contains all the vertices that are gray. However by the loop invariant only, we cannot prove the correctness of the algorithm.

We can prove the correctness by using the following facts:

- *Let the length of the shortest path from s to u be $d(s, u)$. Then for a graph $G = (V, E)$, if an edge $(u, v) \in E$, then $d(s, v) \leq d(s, u) + 1$.* This is true because the shortest path from s to v cannot be longer than the shortest path from s to u plus the edge (u, v).

- *When the breadth first search ends, for each vertex $v \in V$, $v.d \geq d(s, v)$.* This can be proved by induction. The base case is trivial. For the inductive step, we can find on line 15 that $v.d = u.d + 1$. Together with the fact that $u.d \geq d(s, u)$, we have $v.d = u.d + 1 \geq d(s, u) + 1 \geq d(s, v)$.

- *Suppose the first vertex in the queue is v_1 and the last vertex in the queue is v_k. Then for each $i = 2, 3, \ldots, k$, $v_i.d \geq v_{i-1}.d$ and $v_k.d \leq v_1.d + 1$.* This can be proved by induction. The base case is trivial. When we dequeue v_1, by the inductive hypothesis, we have $v_2.d \geq v_1.d$. Then we can find $v_k.d \leq v_1.d + 1 \leq v_2.d + 1$. When we enqueue v_{k+1}, let the current processing vertex be u. By the inductive hypothesis, we have $v_{k+1}.d = u.d + 1 \leq v_1.d + 1$. Also, $v_k \leq u.d + 1 = v_{k+1}.d$.

- *Suppose v_i and v_j are two vertices in the queue, and v_i is enqueued before v_j. Then $v_i.d \leq v_j.d$.* This can be proved by the last fact.

- *If v and u are neighbors, then $v.d \leq u.d + 1$.* When u is dequeued:

  - If v is white, then v is enqueued, and $v.d = u.d + 1$.
  - If v is gray, then by fact 3, $v.d \leq u.d + 1$.
  - If v is black, then v is enqueued before u, and by fact 4, $v.d \leq u.d$.

- *After the breadth first search ends, for each vertex $v \in V$, $v.d$ is the shortest path from s to v.* This can be proved by contradiction. Suppose a vertex v has a shorter path from s to v, or to say $v.d > d(s, v)$, and v has the shortest mismatched path. Then let vertex u be the last vertex on the shortest path from s to v. By the choice of v, we have $u.d+1 = d(s, u)+1 = d(s, v) < v.d$. By fact 5, we have $v.d \leq u.d+1$, which contradicts the previous inequality.

The shortest path can be found by tracing back the parent pointers. This means the breadth first search tree is a shortest path tree, and is a single source shortest path algorithm.

6.2 Depth First Search

In depth first search, we use a stack to store the vertices. Each vertex has four attributes: discovery time, finish time, parent, and color.

```
DFS($G$)
1. for each vertex $u \in G.V$
2.     $u.color = WHITE$
3.     $u.\pi = NIL$
4. $time = 0$
5. for each vertex $u \in G.V$
6.     if $u.color == WHITE$
7.         DFS-Visit($G, u$)
```

```
DFS-Visit($G, u$)
1.  $time = time + 1$
2.  $u.d = time$
3.  $u.color = GRAY$
4.  for each vertex $v \in G.Adj[u]$
5.      if $v.color == WHITE$
6.          $v.\pi = u$
7.          DFS-Visit($G, v$)
8.  $u.color = BLACK$
9.  $time = time + 1$
10. $u.f = time$
```

Runtime: $O(V + E)$. Each vertex is discovered and finished exactly once, and each edge is examined exactly twice.

6.2.1 Properties of Depth First Search

- **Parenthesis property:** For any two vertices u and v, exactly one of the following three conditions holds:
  - The intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth first forest.
  - The interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$, and u is a descendant of v in a depth first tree.
  - The interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$, and v is a descendant of u in a depth first tree.
- **White-path theorem:** In a depth first forest of a graph $G = (V, E)$, vertex v is a descendant of vertex u if and only if at the time $u.d$ that the search discovers u, there is a path from u to v consisting entirely of white vertices.

Proof:
"$\Rightarrow$": If v is a descendant of u, then v is discovered after u, which means v is white when u is discovered. And this is true for all vertices on the path from u to v, since they are all descendants of u.
"$\Leftarrow$": Proof by contradiction. Let v be the first vertex on the white path but is not a descendant of u. Let w be the predecessor of v on the white path. By the choice of w, we knoe that w is a descendant of u, hence $u.d \leq w.f \leq u.f$. (w and u can be the same vertex.) Since there is a path from w to v, v will be discovered before w finishes, hence $v.d < w.f$. In all, we have $u.d < v.d < w.f \leq u.f$, then v must be a descendant of u.

6.2.2 Classification of Edges

There are four types of edges in a depth first forest:

- **Tree edges:** Edges in the depth first forest. An edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v), or to say v's color is white when edge (u, v) is examined.
- **Back edges:** Edges that connect a vertex to an ancestor in a depth first tree. An edge (u, v) is a back edge if at the time (u, v) is examined, v is colored gray.
- **Forward edges:** Edges that connect a vertex to a descendant in a depth first tree. An edge (u, v) is a forward edge if at the time (u, v) is examined, v is colored black and was discovered later than u.
- **Cross edges:** Edges that connect two vertices in different depth first trees. An edge (u, v) is a cross edge if at the time (u, v) is examined, v is colored black and was discovered earlier than u.

In an undirected graph, there are only tree edges and back edges. This is because there is no way to find a vertex that is colored black.
Let (u, v) be an arbitrary edge of an undirected graph $G = (V, E)$. Without loss of generality, assume that $u.d < v.d$. Hence, the search must discover v before it finishes exploring u. If the first time that (u, v) is examined is from u, then it is a tree edge. Otherwise, (u, v) is a back edge, since u is still gray when (u, v) is examined.

6.2.3 DFS and Cycle

A directed graph is acyclic if and only if a depth first search of the graph yields no back edges.
Proof:
"$\Leftarrow$": If there is a back edge (u, v), then we can find a loop that consists of a path from u to v on the depth first tree, and the edge (u, v).
"$\Rightarrow$": If there is a loop, then there is a back edge. Assume the cycle is $v_1, v_2, \dots, v_k, v_1$. Without loss of generality, assume the first discovered vertex is v_1. Then at time $v_1.d$, all other vertices are colored white. By the white-path theorem, v_k is a descendant of v_1. Therefore, (v_1, v_k) is a back edge.

6.2.4 Topological Sort

A topological sort of a directed acyclic graph $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering. A topological sort can be implemented by depth first search.

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

Runtime: $O(|V| + |E|)$.
Correctness: For any edge (u, v), we need to prove that $u.f > v.f$. Without loss of generality, assume u is discovered before v. Since in an acyclic graph there is no back edge, v must be white or black. If v is white, then v is a descendant of u, and $u.f > v.f$. If v is black, then $v.f$ is already set and $u.f > v.f$.

6.2.5 Strongly Connected Components

A directed graph is **strongly connected** if there is a path from each vertex to every other vertex. Let $G = (V, E)$ be a directed graph, and let $G^T = (V, E^T)$ be the transpose of G, which is the graph formed by reversing all edges in G. Note that G and G^T have the same strongly connected components.

```
STRONGLY-CONNECTED-COMPONENTS($G$)
1. call DFS($G$) to compute finishing times $u.f$ 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 $u.f$
4. output the vertices of each tree in the depth first forest formed in line 3 as a separate strongly connected component
```

Runtime: $O(|V| + |E|)$
Correctness: Proof by induction. An SCC must be in its own tree in the depth first forest. And we know the SCC containing the root of a depth first tree can be correctly output. Then we make induction on the number of SCC in the graph. If there is only one SCC, then the algorithm is correct (it will contain the root of the depth first tree). Then, if we want to output the SCCs of a graph with k SCCs, we can exclude the SCC containing the root of one depth first tree, and output the SCCs of the remaining graph, which has $k - 1$ SCCs.

6.3 Minimum Spanning Tree

A **spanning tree** of a graph $G = (V, E)$ is a tree that is a subgraph of G and contains all vertices of G. A **minimum spanning tree** of a weighted, connected, undirected graph $G = (V, E)$ is a spanning tree of G that has minimum weight.

6.3.1 Generic MST Algorithm

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

Correctness: By loop invariant: A is a subset of some MST.

6.3.2 Cuts

A **cut** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V. An edge (u, v) **crosses** the cut $(S, V - S)$ if one of its endpoints is in S and the other is in $V - S$. A cut **respects** a set A of edges if no edge in A crosses the cut. An edge is a **light edge** crossing a cut if its weight is the minimum of any edge crossing the cut.
We can prove that if a cut respects a set A of edges, then the light edge crossing the cut is safe for A. Suppose $(S, V - S)$ is a cut of G that respects A, and let (u, v) be a light edge crossing the cut. Let T be a minimum spanning tree that contains A. If $(u, v) \in T$, then obviously (u, v) is safe for A. Otherwise, T contains a path from u to v but does not contain (u, v). The path from u to v with (u, v) added forms a cycle. Since the cut crosses the cycle, there must be another edge (x, y) in the cycle that crosses the cut. Since (u, v) is a light edge, we have $w(u, v) \leq w(x, y)$. Then we can replace (x, y) with (u, v) in T to get another spanning tree T'. Noting that (x, y) is not in A, since the cut respects A. Hence, T' contains A. Therefore, (u, v) is safe for A.

6.3.3 Kruskal's Algorithm

Kruskal's algorithm sorts all the edges in nondecreasing order by weight, and then adds the edges that connect two different components in the current forest.

```
KRUSKAL($G, w$)
1. $A = \emptyset$
2. for each vertex $v \in G.V$
3.    MAKE-SET($v$)
4. sort the edges of $G.E$ into nondecreasing order by weight $w$
5. for each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
6.    if FIND-SET($u$) $\neq$ FIND-SET($v$)
7.       $A = A \cup \{(u, v)\}$
8.       UNION($u, v$)
9. return $A$
```

Runtime: $O(|E| \log |V|)$ The major cost is sorting the edges, which takes $O(|E| \log |E|) = O(|E| \log |V|)$ time.

6.3.4 Prim's Algorithm

Prim's algorithm grows a single tree by adding edges to the tree one by one. We use a min-priority queue Q to store the vertices.

```
PRIM($G, w, r$)
1.  for each $u \in G.V$
2.      $u.key = \infty$
3.      $u.\pi = NIL$
4.  $r.key = 0$
5.  $Q = G.V$
6.  while $Q \neq \emptyset$
7.      $u =$ EXTRACT-MIN($Q$)
8.      for each $v \in G.Adj[u]$
9.          if $v \in Q$ and $w(u, v) < v.key$
10.             $v.\pi = u$
11.             DECREASE-KEY($Q, v, w(u, v)$)
```

Runtime: $O(|E| \log |V|)$ Dominated by the DECREASE-KEY operations, which takes $O(|E| \log |V|)$ time.

6.4 Single Source Shortest Path

6.4.1 Dijkstra's Algorithm

We can use Dijkstra's algorithm to find the shortest path from a single source to all other vertices. The idea is very similar to Prim's algorithm.

```
DIJKSTRA($G, w, s$)
1.  $S = \emptyset$
2.  $Q = G.V$
3.  BUILD-MIN-HEAP($Q$)
4.  while $Q \neq \emptyset$
5.      $u =$ EXTRACT-MIN($Q$)
6.      $S = S \cup \{u\}$
7.      for each $v \in G.Adj[u]$
8.          if $v.d > u.d + w(u, v)$
10.             $v.\pi = u$
11.             DECREASE-KEY($Q, v, u.d + w(u, v)$)
```

Runtime: $O(|E| \log |V|)$. The BUILD-MIN-HEAP takes $O(|V|)$ time. All calls for EXTRACT-MIN takes $O(|V| \log |V|)$ time. All calls for DECREASE-KEY takes $O(|E| \log |V|)$ time.
Correctness: Prove by loop invariant: Before each iteration of the while loop, for each vertex $v \in S$, $v.d$ is the weight of the shortest path from s to v. We can proof the

maintenance of the loop invariant by contradiction: If u is the first vertex that does not satisfy $u.d = d(s,u)$ when it is added to S, then we can find a shortest path from s to u. Let x be the last vertex in S on the shortest path and y be the first vertex in $V - S$ on the shortest path. Notice that x might be s and y might be u. By the choice of u, we have $x.d = d(s,x)$ and $y.d = d(s,y)$. Then we have $y.d = d(s,y) \leq d(s,u) \leq u.d$. However, since the queue choose u to be the next vertex, we have $u.d \leq y.d$. This means all the inequalities are equalities, and we get $d(s,u) = d.u$.

7 Related Concepts

7.1 Asymptotic Notation

7.1.1 Definition of Θ

A function $f(n)$ belongs to $\Theta(g(n))$ if there exist constants $0 < c_1 \leq c_2$ and n_0 such that $0 < c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.
An easy way to proof: divide both side by $g(n)$. For example to proof $0 \leq c_1 n^2 \leq n^2 + 2n + 3 \leq c_2 n^2$ for all $n \geq 1$.

$$0 \leq c_1 n^2 \leq n^2 + 2n + 3 \leq c_2 n^2 \to 0 \leq c_1 \leq 1 + \frac{2}{n} + \frac{3}{n^2} \leq c_2$$

Similar to O and Ω. A function $f(n)$ belongs to $O(g(n))$ if there exist constants $0 \leq c$ and n_0 such that $0 \leq f(n) \leq c g(n)$ for all $n \geq n_0$. A function $f(n)$ belongs to $\Omega(g(n))$ if there exist constants $0 \leq c$ and n_0 such that $0 \leq c g(n) \leq f(n)$ for all $n \geq n_0$.
In general, big Θ is used to describe the tight bound of a function, big O is used to describe the upper bound of a function, and big Ω is used to describe the lower bound of a function.

7.1.2 Little o and Little ω

A function $f(n)$ belongs to $o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. A function $f(n)$ belongs to $\omega(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, or equivalently, $g(n)$ belongs to $o(f(n))$.

7.1.3 Important Inequalities

Every polynomial of $\log n$ grows slower than every polynomial of n. That is: $(\log n)^{100} = o(n^{0.01})$.
Every polynomial of n grows slower than every exponential 2^{n^c}. That is: $n^{100} = o(2^{n^{0.01}})$.

7.1.4 Rule of Sum and Rule of Product

Rule of Sum: $f(n) + g(n) = \Theta(\max(f(n), g(n)))$.
Rule of Product: $f(n) \cdot g(n) = \Theta(f(n) \cdot g(n))$.

7.2 RAM Model

The RAM model assumes that each elementary operation takes the same amount of time.

7.3 Common Cost Model

Assume the runtime of an algorithm is the number of elementary operations it performs.

7.4 In place

A sorting algorithm is said to be in place if it requires $O(1)$ additional space. Hence, merge sort is not in place, since it requires $\Omega(n)$ additional space.

7.5 Stable

A sorting algorithm is said to be stable if it preserves the relative order of equal elements.

7.6 Master Theorem

Let $a > 0$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be non-negative function for large enough n. Then the recurrence $T(n) = aT(n/b) + f(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

7.7 Acylic Graph

A directed graph is acyclic if it has no directed cycles.

8 Problems from Homework

Problem 1

Explain why the statement The running time of Algorithm A is at least $O(n^2)$ is meaningless.

Problem 2

Prove by induction that every nonempty binary tree satisfies $|V| = |E| + 1$.