30th September 2013

System Design for Big Data [tinyurl]

What is tinyurl?

tinyurl [http://tinyurl.com/] is a URL service that users enter a long URL and then the service return a shorter and unique url such as "http://tiny.me/5ie0V2". The highlight part can be any string with 6 letters containing [0-9, a-z, A-Z]. That is, 62^6 ~= 56.8 billions unique strings.

How it works?

On Single Machine
Suppose we have a database which contains three columns: id (auto increment), actual url, and shorten url.

Intuitively, we can design a hash function that maps the actual url to shorten url. But string to string mapping is not easy to compute.

Notice that in the database, each record has a unique id associated with it. What if we convert the id to a shorten url?
Basically, we need a Bijective function [http://en.wikipedia.org/wiki/Bijection] f(x) = y such that

- Each x must be associated with one and only one y;
- Each y must be associated with one and only one x.

In our case, the set of x's are integers while the set of y's are 6-letter-long strings. Actually, each 6-letter-long string can be considered as a number too, a 62-base numeric, if we map each distinct character to a number,

e.g. 0-0, ..., 9-9, 10-a, 11-b, ..., 35-z, 36-A, ..., 61-Z.

Then, the problem becomes Base Conversion [http://en.wikipedia.org/wiki/Base_conversion#Base_conversion] problem which is bijection (if not overflowed :).

```
 public String shorturl(int id, int base, HashMap map) {
  StringBuilder res = new StringBuilder();
  while (id > 0) {
    int digit = id % base;
    res.append(map.get(digit));
    id /= base;
  }
  while (res.length() < 6)  res.append('0');
  return res.reverse().toString();
}
```

For each input long url, the corresponding id is auto generated (in O(1) time). The base conversion algorithm runs in O(k) time where k is the number of digits (i.e. k=6).

On Multiple Machine
Suppose the service gets more and more traffic and thus we need to distributed data onto multiple servers.

We can use Distributed Database [http://en.wikipedia.org/wiki/Distributed_database] . But maintenance for such a db would be much more complicated (replicate data across servers, sync among servers to get a unique id, etc.).

Alternatively, we can use Distributed Key-Value Datastore [http://en.wikipedia.org/wiki/Distributed_data_store] .
Some distributed datastore (e.g. Amazon's Dynamo [http://en.wikipedia.org/wiki/Dynamo_(storage_system)]) uses Consistent Hashing [http://n00tc0d3r.blogspot.com/2013/09/big-data-consistent-hashing.html] to hash servers and inputs into integers and locate the corresponding server using the hash value of the input. We can apply base conversion algorithm on the hash value of the input.

The basic process can be:
Insert

1. Hash an input long url into a single integer;
2. Locate a server on the ring and store the key--longUrl on the server;
3. Compute the shorten url using base conversion (from 10-base to 62-base) and return it to the user.

Retrieve

1. Convert the shorten url back to the key using base conversion (from 62-base to 10-base);
2. Locate the server containing that key and return the longUrl.

Further Readings

- StackOverflow: How to code a url shortener [http://stackoverflow.com/questions/742013/how-to-code-a-url-shortener]

Posted 30th September 2013 by Sophie

Labels: BigData, Design, Java, Maths

28 View comments

30th September 2013

Clone Graph

Clone Graph [http://oj.leetcode.com/problems/clone-graph/]

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

```
class UndirectedGraphNode {
      int label;
      ArrayList neighbors;
      UndirectedGraphNode(int x) { label = x; neighbors = new ArrayList(); }
  };
```

Solution

This is a basic graph problem. It can be solved via either DFS or BFS.
One thing need to pay attention is that this is a graph not a tree, which mean it is possible to contain cycles. If cycles are not handled properly, it could result in infinite loops.

Implementation with DFS

```
private UndirectedGraphNode cloneDFS(UndirectedGraphNode root, HashMap<UndirectedGraphNode, Undirect
  if (root == null) return root;
  UndirectedGraphNode node = new UndirectedGraphNode(root.label);
  visited.put(root, node);

  // DFS
  for (UndirectedGraphNode nb : root.neighbors) {
    if (visited.containsKey(nb)) {
      node.neighbors.add(visited.get(nb));
    } else {
      node.neighbors.add(cloneDFS(nb, visited));
    }
  }

  return node;
}

public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
  return cloneDFS(node, new HashMap<UndirectedGraphNode, UndirectedGraphNode>());
}
```

Implementation with BFS

```
public UndirectedGraphNode cloneBFS(UndirectedGraphNode root) {
  if (root == null) return root;

  ArrayDeque<UndirectedGraphNode> que = new ArrayDeque<UndirectedGraphNode>();
  que.addLast(root);

  HashMap<UndirectedGraphNode, UndirectedGraphNode> visited = new HashMap<UndirectedGraphNode, Undir
  UndirectedGraphNode rootCopy = new UndirectedGraphNode(root.label);
  visited.put(root, rootCopy);

  // BFS
  while (!que.isEmpty()) {
    root = que.removeFirst();
    UndirectedGraphNode node = visited.get(root);
    for (UndirectedGraphNode nb : root.neighbors) {
      if (visited.containsKey(nb)) {
        node.neighbors.add(visited.get(nb));
      } else {
        UndirectedGraphNode n = new UndirectedGraphNode(nb.label);
        node.neighbors.add(n);
        visited.put(nb, n);
        que.addLast(nb);
      }
    }
  }

  return rootCopy;
}
```

In both algorithms, each vertex and each edge are visited constant times. So, both run in O(|V|+|E|) time.

Posted 30th September 2013 by Sophie

Labels: BFS, DFS, Graph, Java

2 View comments

30th September 2013

Topological Sort

Topological Sorting [https://www.spoj.com/problems/TOPOSORT/]

Sandro is a well organised person. Every day he makes a list of things which need to be done and enumerates them from 1 to n. However, some things need to be done before others. In this task you have to find out whether Sandro can solve all his duties and if so, print the correct order.

Input
In the first line you are given an integer n and m (1<=n<=10000, 1<=m<=1000000). On the next m lines there are two distinct integers x and y, (1<=x,y<=10000) describing that job x needs to be done before job y.

Output
Print "Sandro fails." if Sandro cannot complete all his duties on the list. If there is a solution print the correct ordering, the jobs to be done separated by a whitespace. If there are multiple solutions print the one, whose first number is smallest, if there are still multiple solutions, print the one whose second number is smallest, and so on.

Example 1

```
Input:
8 9
1 4
1 2
4 2
4 3
3 2
5 2
3 5
8 2
8 6
Output:
1 4 3 5 7 8 2 6
```

Example 2

```
Input:
2 2
1 2
2 1
Output:
Sandro fails.
```

Solution

Given a set of vertices and a set of directed edges between vertices, Topological Sort [http://en.wikipedia.org/wiki/Toposort] (i.e. toposort) is to produce a linear ordering of vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering.
Toposort only works in Directed Acyclic Graphs [http://en.wikipedia.org/wiki/Directed_acyclic_graph] (DAG). The most common use case is job scheduling.

Toposort results for a DAG may not be unique. For instance, in the below DAG,

[http://2.bp.blogspot.com/-zNES4KSTy6c/Uki25BlpZnI/AAAAAAAAEkc/v1-NLlGnRRw/s1600/180px-Directed_acyclic_graph.png]
valid toposort results include:

- 7, 5, 3, 11, 8, 2, 9, 10 (visual left-to-right, top-to-bottom)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)

Given a graph represented by a list of adjacent lists,

1. Calculate in-degree for the vertex;
2. Enqueue 0 in-degree vertices;
3. For each vertex in queue,
    1. visit the vertex and mark it as visited;
    2. if any of its neighbor has been marked as visited, return error (cycle detected);
    3. otherwise, decrease the in-degree for all of its neighbors and enqueue the ones that in-degree becomes 0;

This algorithm runs in time O(|V|+|E|) since it visits each vertex twice (enqueue and dequeue) and visits each edge once.

```
private static void toposort(ArrayList<ArrayList<Integer>> graph, int[] indegs) {
  int n = indegs.length;
  boolean[] visited = new boolean[n];
  Queue<Integer> que = new PriorityQueue<Integer>(); // use heap s.t. smallest-numbered available vert
  ArrayList<Integer> res = new ArrayList<Integer>(n);

  // enque 0-in-degree nodes
  for (int i=0; i<n; ++i) {
    if (indegs[i] == 0) que.offer(i);
  }

  // bfs
  while (!que.isEmpty()) {
    int node = que.poll();
    res.add(node+1);
    // mark as visited
    visited[node] = true;
    // update its neighbors and enqueue 0-in-degree ones
    for (int nb : graph.get(node)) {
      if (visited[nb]) { // a cycle detected
        break;
      }
      if (--indegs[nb] == 0) que.offer(nb);
    }
  }

  // print
  if (res.size() < n) {
    System.out.println("Sandro fails.");
  } else {
    System.out.println(res);
  }
}

public static void main(String[] args) {
  Scanner in = new Scanner(System.in);
  int n = in.nextInt(), m = in.nextInt();

  // initial graph
  ArrayList<ArrayList<Integer>> graph = new ArrayList<ArrayList<Integer>>(n);
  int[] indegs = new int[n];
  for (int i=0; i<n; ++i) {
    graph.add(new ArrayList<Integer>(n));
  }

  // parse edges
  for (int j=0; j<m; ++j) {
    int v1 = in.nextInt() - 1, v2 = in.nextInt() - 1;
    // add v2 to v1's neighbor list
    graph.get(v1).add(v2);
    // increase v2's indegree
    indegs[v2]++;
  }

  toposort(graph, indegs);
}
```

Note: This algorithm failed SPOJ due to TLE. Maybe C++ version can get through.

<Introduction To Algorithms [http://www.amazon.com/gp/product/0262033844/ref=as_li_ss_il?ie=UTF8&camp=1789&creative=390957&creativeASIN=0262033844&linkCode=as2&tag=n00tc0d3r-20] > provides an alternative algorithm by using DFS. The complexity is also O(|V|+|E|).

Further Readings

- Another blog (in Chinese) [http://www.cnblogs.com/shanyou/archive/2006/11/16/562861.html]
- Linux tsort source code [http://www.opensource.apple.com/source/misc_cmds/misc_cmds-27/tsort/tsort.c]
- Dependency Tracing [http://dep-trace.sourceforge.net/]

Posted 30th September 2013 by Sophie

Labels: Graph, Java, Sort

1 View comments

23rd September 2013

System Design for Big Data [Consistent Hashing]

Suppose you are designing a distributed caching system.

Given n cache hosts, an intuitive hash function is `key % n`. It is simple and commonly used. But it has two major drawbacks:

- It is NOT horizontally scalable.
  Every time when adding one new cache host to the system, all existing mappings are broken. It will be a pain point in maintenance if the caching system contains a lots of data. Also, if the caching system is behind a popular service, it is not easy to schedule a downtime to update all caching mappings.
- It may NOT be load balanced, especially for non-uniformly distributed data.
  In real world, it is less likely that the data is uniformly distributed. Then for the caching system, it results that some caches are hot and saturated while the others idle and almost empty.

In such situations, consistent hashing is a good way to improve the caching system.

What is Consistent Hashing?

Consistent Hashing [http://en.wikipedia.org/wiki/Consistent_hashing] is a hashing strategy such that when the hash table is resized (e.g. a new cache host is added to the system), only k/n keys need to be remapped, where k is the number of keys and n is the number of caches. Recall that in a caching system using the mod as hash function, all keys need to be remapped.

Consistent hashing maps an object to the same cache host if possible. If a cache host is removed, the objects on that host will be shared by other hosts; If a new cache is added, it takes its share from other hosts without touching other shares.

When to use Consistent Hashing?

Consistent hashing is a very useful strategy for distributed caching system and distributed hash tables [http://en.wikipedia.org/wiki/Distributed_hash_table] .
It can reduce the impact of host failures. It can also make the caching system more easier to scale up (and scale down).

Example of uses include:

- Last.fm: memcached client with consistent hashing [http://www.last.fm/user/RJ/journal/2007/04/10/rz_libketama_-_a_consistent_hashing_algo_for_memcache_clients]
- Amazon: internal scalable key-value store, Dynamo [http://s3.amazonaws.com/AllThingsDistributed/sosp/amazon-dynamo-sosp2007.pdf]
- Chord [https://github.com/sit/dht/wiki] : a distributed hash table by MIT

How it works?

As a typical hash function, consistent hashing maps a key or a cache host to an integer.

Suppose the output of the hash function are in the range of [0, 2^128) (e.g. MD5 [http://en.wikipedia.org/wiki/MD5] hash). Image that the integers in the range are placed on a ring such that the values are wrapped around.

Here's how consistent hashing works:

- Given a list of cache servers, hash them to integers in the range.
- To map a key to a server,
  - Hash it to a single integer.
  - Move clockwise on the ring until finding the first cache it encounters.
  - That cache is the one that contains the key.
- See animation below as an example: key1 maps to cache A; key2 maps to cache C.

[http://2.bp.blogspot.com/-FoDbp5aJxmo/Uj9lbZgCMpI/AAAAAAAAEjw/glacbrT174s/s1600/feiche_6.gif]
How Consistent Hashing works

To add a new cache, say D, keys that were originally falling to C will be split and some of them will be moved to D. Other keys don't need to be touched.

To remove a cache or if a cache failed, say C, all keys that were originally mapping to C will fall into A and only those keys need to be moved to A. Other keys don't need to be touched.

Now let's consider the load balance issue.
As we discussed at the beginning, the real data are essentially randomly distributed and thus may not be uniform. It may cause the keys on caches are unbalanced.

To resolve this issue, we add "virtual replicas" for caches.

- For each cache, instead of mapping it to a single point on the ring, we map it to multiple points on the ring, i.e. replicas.
- By doing this, each cache is associated with multiple segments of the ring.

[http://2.bp.blogspot.com/-_sG8zBqb4ug/Uj9RLNk7E8I/AAAAAAAAEkA/S8vGVnqdf5M/s1600/feiche_7.gif]
Virtual Replicas of Caches

If the hash function "mixes well [http://en.wikipedia.org/wiki/Hash_function#Uniformity] ", as the number of replicas increases, the keys will be more balanced. See [3] for a simulation result.

Monotone Keys

If keys are known to be monotonically increased, binary searching can be used to improve the performance of locating a cache for a given key. Then the locate time can be reduced to O(logn).

More Readings

1. The original paper: <Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web> [http://citeseer.ist.psu.edu/karger97consistent.html]
2. A blog post about consistent hashing [http://www.tomkleinpeter.com/2008/03/17/programmers-toolbox-part-3-consistent-hashing/]
3. Another blog post about consistent hashing [http://www.tom-e-white.com/2007/11/consistent-hashing.html]

Posted 23rd September 2013 by Sophie

Labels: BigData, Design

4 View comments

20th September 2013

Inorder Binary Tree Traversal with Constant Space

Inorder Binary Tree Traversal with Constant Space [http://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion-and-without-stack/]

Implementing inorder traversal with O(n) space is pretty straight forward. Could you devise a constant space solution?

Solution

An inorder traversal can be implemented using recursion or stack (see previous post [http://n00tc0d3r.blogspot.com/2013/01/binary-tree-traversals-i.html]). But both methods requires O(h) space, where h is the height of the tree. That said, the worst case space complexity can be O(n).

Morris Traversal [http://en.wikipedia.org/wiki/Tree_traversal#Morris_in-order_traversal_using_threading] introduced a way to implement inorder traversal without recursion or stack.

It first modify the tree to a partial Threaded Binary Tree [http://en.wikipedia.org/wiki/Threaded_binary_tree] where all right child pointers that were null in the original tree are pointed to their inorder successor.
With that information in hand, it is much easier to conduct an inorder traversal: Go to the left most node, and follow right pointers to complete the traversal.
During the traversal, Morris algorithm then fix the modified right pointers and set them back to null.

Here is the algorithm. The code is not too long (surprising :). I would suggest you to walk it through an example to better understand how it works.

```
private void inorderMorris(TreeNode root, ArrayList<Integer> values) {
  TreeNode cur = root;

  while (cur != null) {
    if (cur.left != null) {
      TreeNode pre = cur.left;
      while (pre.right != null && pre.right != cur) {
        pre = pre.right;
      }
      if (pre.right == null) { // set right to successor
        pre.right = cur;
        cur = cur.left;
      } else { // visit and revert the change
        pre.right = null;
        values.add(cur.val);
        cur = cur.right;
      }
    } else { // visit and move to successor
      values.add(cur.val);
      cur = cur.right;
    }
  }
}
```

This algorithm touches each node at most three times: find successor, visit, fix right pointer of pre node. So, it runs in time O(n) and uses O(1) space!

Posted 20th September 2013 by Sophie

Labels: BinaryTree, Java, Tree

1 View comments

9th September 2013

Count Numbers That Has A 4 In It

Count Numbers That Has A 4 In It [http://www.glassdoor.com/Interview/1-Count-the-number-of-positive-integers-less-than-N-that-does-not-contains-digit-4-2-Design-a-data-structure-that-can-d-QTN_499877.htm]

Given a positive integer N, calculate the number of positive integers under N that has at least one digit equal to 4.

For example, given 8, return 1 since only 4 contains digit 4.

Solution

A straightforward way to solve this problem is for each number under N, check every digit of the number for digit 4. If it contains one, count it.

```
private boolean contains4(int num) {
  while (num > 3) {
    if (num % 10 == 4) return true;
    num /= 10;
  }
  return false;
}

public int Count4s(int num) {
  int count = 0;
  for (int i=4; i<num; ++i) {
    if (contains4(i)) ++count;
  }
  return count;
}
```

This algorithm runs in time O(N*k) where k is the number of digits in N.

Think about it.

When you count the numbers skipping digit 4,

- For 0-9, there is one number containing digit 4.
- Same for 10-19, 20-29, 30-39, 50-59, ..., 90-99, except 40-49. So, for 0-99, there are 19=1*9+10 numbers containing digit 4.
- Similarly, for 0-999, there are 271=19*9+100 numbers containing digit 4.
- Therefore, given a number with k digits, we have
  - K_n = K_(n-1) * 9 + 10^(n-1), n>0
  - For n-th digit x, count += (x < 4) ? x * K_(n-1) : (x-1) * K_(n-1) + 10^n
  - For 0-th digit x, count += (x < 4) ? 0 : 1

For example,
count for 3516 is 991=3*271 + ((5-1)*19 + 100) + 1*1 + 1.

Take a look from another point of view.
To calculate the number of integers that do NOT contain digit 4, it is essentially counting in a 9-based numerical system. Then we have

K_n = 10^n - 9^n, n>0

For example,
count for 3516 is 991 = 3*(10^3 - 9^3) + ((5-1)*(10^2 - 9^2)+10^2) + 1*(10 - 9) + 1.
count for 341 is 63 = 3*(10^2 - 9^2) + (4*(10 - 9) + 2) + 0.

Now the algorithm is much simpler:

```
public static int Count4s2(int num) {
  int count = 0;
  for (int nines = 1, tens = 1, remain=0; num>=1; num/=10, nines*=9, tens*=10) {
    int digit = num % 10;
    count += digit*(tens - nines);
    if (digit == 4) { // **4xx: count += xx
      count += (remain + 1);
    } else if (digit > 4) {
      count += tens;
    }
    remain += digit*tens;
  }
  return count;
}
```

This algorithm runs in time O(k).

If you are interested in number theories behind this, this post [http://www.cut-the-knot.org/do_you_know/digit3.shtml] may be helpful for you.

Posted 9th September 2013 by Sophie

Labels: Java, Maths

8 View comments

1st September 2013

Implement LRU Cache

Implement LRU Cache [http://www.geeksforgeeks.org/implement-lru-cache/]

Least Recently Used [http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used] (LRU) caching scheme is to discard the least recently used items first when the cache is full and a newly visted item needs to be added to the cache.
Design a LRU caching scheme that implement the following interface.

```
public interface LruPageCache {
  /** Set the capacity of the cache. */
  public setCapacity(int capacity);

  /** Returns the page number and update cache accordingly.
   *  This time complexity of this method should be O(1).
   */
  public int loadPage(int pageNum);
}
```

Solution

We need a data structure to check whether a page number is in cache in constant time. HashMap [http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html] with each page number as a key can make it.

We also need a data structure to maintain page numbers in cache in the order of their access time. One way to do that is to keep a timestamp field for each record, but we still need to sort them which cannot be done in O(1) time. Alternatively, we can use a linked list to keep all records, and move the newly visited one to the head of the list. To get O(1) time complexity for updating such a linked list, we need a doubly linked list.

Each time when a new page number comes in,

- If it is already in the cache, move the node to the head of the linked list;
- If it is not in the cache, insert it to the head of the linked list and update the current capacity of the cache. If the cache is full, remove the last node of the linked list. (So, we also need a tail pointer. :)

```
public static class LruCacheImpl implements LruPageCache {
  private int capacity = 0;
  private int maxCapacity = 10;
  private DListNode head = null;
  private DListNode tail = null;
  private HashMap<Integer, DListNode> map = new HashMap<Integer, DListNode>();

  /** {@inheritDoc} */
  @Override
  public void setMaxCapacity(final int limit) {
    if (limit < 1) {
      throw InvalidInputException("Max capacity must be positive.");
    }
    maxCapacity = limit;
  }

  /** {@inheritDoc} */
  @Override
  public int loadPage(final int page) {
    final DListNode cur;

    if (map.containsKey(page)) { // cache hit
      cur = map.get(page);
      if (cur != head) {
        remove(cur);
        insertToHead(cur);
      }
      print();
      return cur.val;
    }

    // cache miss
    cur = new DListNode(page);
    insertToHead(cur);
    map.put(page, cur);

    if (capacity == maxCapacity) {
      removeTail();
    } else {
      ++capacity;
    }
    print();

    return cur.val;
  }

  /** Remove the given node from the linked list. */
  private void remove(final DListNode cur) {
    if (cur.pre != null) cur.pre.next = cur.next;
    if (cur.next != null) cur.next.pre = cur.pre;
    if (tail == cur) tail = cur.pre;
  }

  /** Remove the tail of the linked list and return the deleted node. */
  private DListNode removeTail() {
    map.remove(tail.val);
    DListNode last = tail;
    tail = tail.pre;
    tail.next = null;
    if (head == last) head = null;
    return last;
  }
  /** Add the given node to the head of the linked list. */
  private void insertToHead(final DListNode cur) {
    cur.next = head;
    cur.pre = null;
    if (head != null) head.pre = cur;
    head = cur;
    if (tail == null) tail = cur;
  }

  private void print() {
    DListNode cur = head;
    System.out.print("head->");
    while (cur != null) {
      System.out.print(cur.val);
      if (cur == tail) System.out.print(" (tail)");
      else System.out.print("->");
      cur = cur.next;
    }
    System.out.println("");
  }

  /** Doubly Linked list */
  private class DListNode {
    DListNode pre = null;
    DListNode next = null;
    int val;
    DListNode(int v) {
      val = v;
    }
  }
}
```

This scheme provide O(1) time for `loadPage` and uses O(n) spaces where n is the maxCapacity of the cache.

Test outputs:

```
public static void main(String[] args) {
  LruCacheImpl cache = new LruCacheImpl();
  cache.setMaxCapacity(4);
  System.out.println(cache.loadPage(2));
  System.out.println(cache.loadPage(3));
  System.out.println(cache.loadPage(1));
  System.out.println(cache.loadPage(2));
  System.out.println(cache.loadPage(4));
  System.out.println(cache.loadPage(1));
  System.out.println(cache.loadPage(4));
  System.out.println(cache.loadPage(5));
  System.out.println(cache.loadPage(6));
}
-----
output:
head->2 (tail)
2
head->3->2 (tail)
3
head->1->3->2 (tail)
1
head->2->1->3 (tail)
2
head->4->2->1->3 (tail)
4
head->1->4->2->3 (tail)
1
head->4->1->2->3 (tail)
4
head->5->4->1->2 (tail)
5
head->6->5->4->1 (tail)
6
```

Note: Java provides a LinkedHashMap [http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html] class which implements hashmap on a doubly linked list. It is essentially what we have done here except that LinkedHashMap has no capacity limit. So, in real world, if you need an infinite LRU cache, don't reinvent wheel!

Posted 1st September 2013 by Sophie

Labels: Design, Hash, Java, LinkedList

4 View comments

26th August 2013

Implement Iterator for BinaryTree III (Post-order)

Implement Post-order Iterator for Binary Tree

Suppose the data structure for a Tree node is as follows:

```
public class TreeNode {
  int val;
  TreeNode left;
  TreeNode right;
  TreeNode(int x) { val = x; }
}
```

Provide an implementation of the following interface:

```
public interface PostOrderBinaryTreeIterator extends Iterator<Integer> {
  /** Returns the next integer a in the post-order traversal of the given binary tree.
   * For example, given a binary tree below,
   *       4
   *      / \
   *     2   6
   *    / \ / \
   *   1  3 5  7
   * the outputs will be 1, 3, 2, 5, 7, 6, 4.
   */
  public Integer next();

  /** Return true if traversal has not finished; otherwise, return false. */
  public boolean hasNext();
}
```

Solution

A straightforward way is, as we discussed in previous post [http://n00tc0d3r.blogspot.com/2013/01/binary-tree-traversals-i.html] , to traverse the tree with mirrored pre-order, i.e. root-right-left, and store all of them in a stack. Then `next()` just need to pop nodes out from the stack.

By doing that, it requires O(n) extra spaces since pre-process stores all nodes in the tree.

Alternatively, we can do it on the fly somehow.

- Initially, we find the first leaf that is going to be visited first and store all intermediate nodes in a stack;
- Each time we pop out a node from the stack, we check whether it is the left child of the current top of the stack. If so, repeat the step above on the right sub-tree of the current top.

We only need to check whether current is left of top since if it is right, we know top will be the next-to-be-popped node and thus no need to do anything.

```
public class PostOrderBinaryTreeIteratorImpl implements PostOrderBinaryTreeIterator {
  Stack<TreeNode> stack = new Stack<TreeNode>();

  /** find the first leaf in a tree rooted at cur and store intermediate nodes */
  private void findNextLeaf(TreeNode cur) {
    while (cur != null) {
      stack.push(cur);
      if (cur.left != null) {
        cur = cur.left;
      } else {
        cur = cur.right;
      }
    }
  }

  /** Constructor */
  public PostOrderBinaryTreeIterator(TreeNode root) {
    findNextLeaf(root);
  }

  /** {@inheritDoc} */
  @Override
  public boolean hasNext() {
    return !stack.isEmpty();
  }

  /** {@inheritDoc} */
  @Override
  public Integer next() {
    if (!hasNext()) {
      throw new NoSuchElementException("All nodes have been visited!");
    }

    TreeNode res = stack.pop();
    if (!stack.isEmpty()) {
      TreeNode top = stack.peek();
      if (res == top.left) {
        findNextLeaf(top.right); // find next leaf in right sub-tree
      }
    }

    return res.val;
  }
  @Override
  public void remove() {
    throw new UnsupportedOperationException("remove() is not supported.");
  }
}
```

This iterator takes extra spaces for the queue, which is O(h) at worst case, where h is the height of the tree.

With this iterator in hand, an post-order traversal of a binary tree can be implemented as follows.

```
public ArrayList<Integer> postorderTraversal(TreeNode root) {
  PostOrderBinaryTreeIterator iterator = new PostOrderBinaryTreeIteratorImpl(root);
  ArrayList<Integer> results = new ArrayList<Integer>();
  while (iterator.hasNext()) {
    results.add(iterator.next());
  }
  return results;
}
```

Posted 26th August 2013 by Sophie

Labels: BinaryTree, Design, Java, Stack

4 View comments

26th August 2013

Implement Iterator for BinaryTree II (Pre-order)

Implement Pre-order Iterator for Binary Tree

Suppose the data structure for a Tree node is as follows:

```
public class TreeNode {
  int val;
  TreeNode left;
  TreeNode right;
  TreeNode(int x) { val = x; }
}
```

Provide an implementation of the following interface:

```
public interface PreOrderBinaryTreeIterator extends Iterator<Integer> {
  /** Returns the next integer a in the pre-order traversal of the given binary tree.
   * For example, given a binary tree below,
   *       4
   *      / \
   *     2   6
   *    / \ / \
   *   1  3 5  7
   * the outputs will be 4, 2, 1, 3, 6, 5, 7.
   */
  public Integer next();

  /** Return true if traversal has not finished; otherwise, return false.
   */
  public boolean hasNext();
}
```

Solution

The idea is slightly different from we discussed in previous post [http://n00tc0d3r.blogspot.com/2013/01/binary-tree-traversals-i.html] .

In previous post, we visit left nodes on the way to get to the left-most one. But in an iterator, given a top node in the stack, it is not easy to tell whether the left node of the top one has been visited or not. We can make it work by adding flags. But there are a better way to do that.

The reason why we need to keep previous nodes is that when we complete the left sub-tree, we can go to the right sub-tree. That said, what we actually need to keep track are right nodes.

We can use Stack and handle the process more naturally. Each time when we visit a node, we push its right and left children into the stack so that we can access left subtree first and then right subtree. More specifically, we use ArrayDeque [http://docs.oracle.com/javase/6/docs/api/java/util/ArrayDeque.html] which a "resizable-array implementation of the Deque interface". It provides amortized constant time operations such as add, poll, push, and pop, etc.

```
public class PreOrderBinaryTreeIteratorImpl implements PreOrderBinaryTreeIterator {
  Stack<TreeNode> stack = new ArrayDeque<TreeNode>();

  /** Constructor */
  public PreOrderBinaryTreeIterator(TreeNode root) {
    if (root != null) {
      stack.push(root); // add to end of queue
    }
  }

  /** {@inheritDoc} */
  @Override
  public boolean hasNext() {
    return !stack.isEmpty();
  }

  /** {@inheritDoc} */
  @Override
  public Integer next() {
    if (!hasNext()) {
      throw new NoSuchElementException("All nodes have been visited!");
    }

    TreeNode res = stack.pop(); // retrieve and remove the head of queue
    if (res.right != null) stack.push(res.right);
    if (res.left != null) stack.push(res.left);

    return res.val;
  }

  @Override
  public void remove() {
    throw new UnsupportedOperationException("remove() is not supported.");
  }
}
```

This iterator takes extra spaces for the stack, which is O(h) at worst case, where h is the height of the tree.

With this iterator in hand, an pre-order traversal of a binary tree can be implemented as follows.

```
public ArrayList<Integer> preorderTraversal(TreeNode root) {
  PreOrderBinaryTreeIterator iterator = new PreOrderBinaryTreeIteratorImpl(root);
  ArrayList<Integer> results = new ArrayList<Integer>();
  while (iterator.hasNext()) {
    results.add(iterator.next());
  }
  return results;
}
```

Posted 26th August 2013 by Sophie

Labels: BinaryTree, Design, Java, Queue

0 Add a comment

26th August 2013

Implement Iterator for BinaryTree I (In-order)

Implement In-order Iterator for Binary Tree [http://www.glassdoor.com/Interview/In-one-of-the-questions-in-the-interview-they-asked-me-to-build-a-tree-iterator-for-a-binary-tree-QTN_145686.htm]

Suppose the data structure for a Tree node is as follows:

```
public class TreeNode {
  int val;
  TreeNode left;
  TreeNode right;
  TreeNode(int x) { val = x; }
}
```

Provide an implementation of the following interface:

```
public interface InOrderBinaryTreeIterator extends Iterator<Integer> {
  /** Returns the next integer a in the in-order traversal of the given binary tree.
   * For example, given a binary tree below,
   *       4
   *      / \
   *     2   6
   *    / \ / \
   *   1  3 5  7
   * the outputs will be 1, 2, 3, 4, 5, 6, 7.
   */
  public Integer next();

  /** Return true if traversal has not finished; otherwise, return false.
   */
  public boolean hasNext();
}
```

Solution

The idea is still the same as we discussed in previous post [http://n00tc0d3r.blogspot.com/2013/01/binary-tree-traversals-i.html]):

- Find the left-most node of the root and store previous left children in a stack;
- Pop up the top node from the stack;
- If it has a right child, find the lef-most node of the right child and store left children in the stack.

```
public class InOrderBinaryTreeIteratorImpl implements InOrderBinaryTreeIterator {
  Stack<TreeNode> stack = new Stack<TreeNode>();

  /** Push node cur and all of its left children into stack */
  private void pushLeftChildren(TreeNode cur) {
    while (cur != null) {
      stack.push(cur);
      cur = cur.left;
    }
  }

  /** Constructor */
  public InOrderBinaryTreeIterator(TreeNode root) {
    pushLeftChildren(root);
  }

  /** {@inheritDoc} */
  @Override
  public boolean hasNext() {
    return !stack.isEmpty();
  }

  /** {@inheritDoc} */
  @Override
  public Integer next() {
    if (!hasNext()) {
      throw new NoSuchElementException("All nodes have been visited!");
    }

    TreeNode res = stack.pop();
    pushLeftChildren(res.right);

    return res.val;
  }

  @Override
  public void remove() {
    throw new UnsupportedOperationException("remove() is not supported.");
  }
}
```

This iterator takes extra spaces for the stack, which is O(h) at worst case, where h is the height of the tree.

With this iterator in hand, an in-order traversal of a binary tree can be implemented as follows.

```
public ArrayList<Integer> inorderTraversal(TreeNode root) {
  InOrderBinaryTreeIterator iterator = new InOrderBinaryTreeIteratorImpl(root);
  ArrayList<Integer> results = new ArrayList<Integer>();
  while (iterator.hasNext()) {
    results.add(iterator.next());
  }
  return results;
}
```

Posted 26th August 2013 by Sophie

Labels: BinaryTree, Design, Java, Stack

0 Add a comment

19th August 2013 Convert Infix Expression To Postfix Expression

Convert Infix Expression To Postfix Expression [http://geeksquiz.com/stack-set-2-infix-to-postfix/]

Given an infix expression and convert it to a postfix expression.

A mathematical expression, e.g. (3+4), can be notated as Infix [http://en.wikipedia.org/wiki/Infix_notation] ("3+4"), Postfix [http://en.wikipedia.org/wiki/Reverse_Polish_notation] ("34+"), and Prefix [http://en.wikipedia.org/wiki/Polish_notation] ("+34") expressions.
Infix expressions are human readable notations while postfix ones are machine friendly notations. Usually, human inputs are in infix format which are converted to postfix expressions in a computer and then evaluated to get the output result (see previous post [http://n00tc0d3r.blogspot.com/2013/08/evaluate-postfix-expression.html] for evaluation).

For example,
Given "3 + 4", return "34+".
Given "3*(4+5)-6/(1+2)", return "345+*612+/-".

Solution

Intuitively, for digits, we can simply append it to the output string, but for operators, we need a stack to keep previous ones and pop them to the output string as needed.

Let's go through several examples first.

```
"3 + 4 - 5" -> "34+5-"     // pop '+' before '-' since they have the same
"3 + 4 * 5" -> "345*+"     // didn't pop '+' until '*' get popped since '*' has higher precedence than '+'
"3^2^2+4" -> "322^^4+"     // didn't pop '^' until '+' comes in since '^' is
```

precedence [http://en.wikipedia.org/wiki/Order_of_operations]; right-associative [http://en.wikipedia.org/wiki/Operator_associat

Now, we know what we are going to handle this problem:
For each read-in character from the given string,

- If it is a digit, append to output;
- If it is a left parenthesis, push it to stack;
- If it is a right parenthesis, pop out operators from stack and append to output until hit a left parenthesis (if left one doesn't exist, throw exception), also pop out the left parenthesis but no need to append to output;
- If it is an operator,
    - while the top of stack is an operator and is left-associative and has higher or the same precedence, pop it out and append to output;
    - push the new one to stack.

This algorithm is known as Shunting Yard Algorithm [http://en.wikipedia.org/wiki/Shunting_yard_algorithm] . It runs in time O(n) in average and takes O(m) extra spaces in worst case, where n is the length of the given string and m is the number of operators.

```
public static String infixToPostfix(String expr) {
  StringBuilder postfix = new StringBuilder();
  Stack<Character> stack = new Stack<Character>();

  // read in tokens
  for (int i=0; i<expr.length(); ++i) {
    char c = expr.charAt(i);
    if (isDigit(c)) {
      postfix.append(c);
    } else if (isOp(c)) {
      while (isLeftAssociative(c) && !stack.isEmpty() && getPreced(stack.peek()) >= getPreced(c)) {
        postfix.append(stack.pop());
      }
      stack.push(c);
    } else if (c == '(') {
      stack.push(c);
    } else if (c == ')') {
      while (!stack.isEmpty() && stack.peek() != '(') {
        postfix.append(stack.pop());
      }
      if (stack.isEmpty()) {
        throw new IllegalArgumentException("mismatched parentheses.");
      }
      stack.pop(); // pop '(' without adding to output
    } else if (c == ' ') {
      // do nothing
    } else {
      throw new IllegalArgumentException("Invalid input.");
    }
  }

  // empty stack
  while (!stack.isEmpty()) {
    char c = stack.pop();
    if (c == '(') {
      throw new IllegalArgumentException("mismatched parentheses.");
    }
    postfix.append(c);
  }

  return postfix.toString();
}

private static boolean isLeftAssociative(char op) {
  switch (op) {
  case '+':
  case '-':
  case '*':
  case '/':
    return true;
  case '^':
    return false;
  }
  throw new IllegalArgumentException("Invalid input.");
}

private static int getPreced(char op) {
  switch (op) {
  case '+':
  case '-':
    return 1;
  case '*':
  case '/':
    return 2;
  case '^':
    return 3;
  case '(':
  case ')':
    return -1;
  }
  throw new IllegalArgumentException("Invalid input.");
}

private static boolean isDigit(char c) {
  return (c >= '0' && c <= '9');
}

private static boolean isOp(char c) {
  switch (c) {
  case '+':
  case '-':
  case '*':
  case '/':
  case '^':
    return true;
  }
  return false;
}
```

Posted 19th August 2013 by Sophie

Labels: Java, Maths, Stack, String

0 Add a comment

19th August 2013

Evaluate Postfix Expression

Evaluate Postfix Expression [http://www.careercup.com/question?id=15066892]

A Postfix Expression, also called Reverse Polish Notation [http://en.wikipedia.org/wiki/Reverse_Polish_notation] , is a mathematical expression in which every operator follows all of its operands.

For example,
Given "3 4 +", return 7 (= 3+4).
Given "345+*612+/-", return 25 (= 3*(4+5)-6/(1+2)).

You can assume that the result is in the range of a 32-bit integer.

Solution

Since operands come first, we can use a stack to store them. When we hit an operator, pop out two numbers from the stack and execute the operator.

Be careful about the order of the two operands since operator '-' and '/' are not commutative [http://en.wikipedia.org/wiki/Commutative_property] . For example, "43-" should be (4-3), i.e. the firstly popped operand is the right operand and the lastly popped one is the left operand.

```
public int evalPostfixExpression(String expr) {
  Stack<Integer> stack = new Stack<Integer>();

  for (int i=0; i<expr.length(); ++i) {
    char c = expr.charAt(i);
    if (isDigit(c)) {
      stack.push(c - '0');
    } else if (isOp(c)) {
      int b = stack.pop(); // be careful of the order
      int a = stack.pop(); // be careful of the order
      stack.push(execOp(c, a, b));
    } else if (c == ' ') {
      // do nothing
    } else {
      throw new IllegalArgumentException("Invalid input.");
    }
  }

  return stack.pop();
}

private int execOp(char op, int a, int b) {
  switch (op) {
  case '+':
    return (a + b);
  case '-':
    return (a - b);
  case '*':
    return (a * b); // assume that result < Integer.MAX_VALUE
  case '/':
    return (a / b); // assume that result is an integer
  case '^':
    return Math.pow(a, b); // assume that result < Integer.MAX_VALUE
  }
  throw new IllegalArgumentException("Invalid input.");
}

private boolean isDigit(char c) {
  return (c >= '0' && c <= '9');
}

private boolean isOp(char c) {
  switch (c) {
  case '+':
  case '-':
  case '*':
  case '/':
  case '^':
    return true;
  }
  return false;
}
```

Posted 19th August 2013 by Sophie

Labels: Java, Maths, Stack, String

0 Add a comment

19th August 2013

Implement Bounded Blocking Queue

Implement Bounded Blocking Queue [http://www.careercup.com/question?id=14004678]

Write a multithreaded bounded Blocking Queue [http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html] where the capacity of the queue is limited. Implement `size`, `add`, `remove`, and `peek` methods.

There could be multiple producer and consumer threads.
Producers fill up the queue. If the queue is full, producers should wait;
On the other hand, consumers take elements from the queue. If the queue is empty, consumers should wait.

Solution

The make the actions of adding or removing an element from the underlying queue, we need to either use lock or `synchronized` the relative blocks that conduct the actions.

Here is an implementation [http://tutorials.jenkov.com/java-concurrency/blocking-queues.html] with `synchronized`.

One shortcoming of using synchronization is that it only allow one thread access the queue at the same time, either consumer or producer.)

Plus, we need to use `notifyAll` instead of `notify` since there could be multiple waiting producers and consumers and `notify` can wake up any thread which could be a producer or a consumer. This stackoverflow post gives a detailed example to explain notify vs. notifyAll [http://stackoverflow.com/a/3186336/1131594] .

```
public class BoundedBlockingQueue<E> {
   private final Queue<E> queue = new LinkedList<E>();
   private final int capacity;
   private final AtomicInteger count = new AtomicInteger(0);

   public BoundedBlockingQueue(int capacity) {
     if (capacity <= 0)  throw new InvalidArgumentException("The capacity of the queue must be > 0.");
     this.capacity = capacity;
   }

   public int size() {
     return count.get();
   }

   public synchronized void add(E e) throws RuntimeException {
     if (e == null) throw new NullPointerException("Null element is not allowed.");

     int oldCount = -1;
     while (count.get() == capacity) wait();

     queue.add(e);
     oldCount = count.getAndIncrement();
     if (oldCount == 0) {
       notifyAll(); // notify other waiting threads (could be producers or consumers)
     }
   }

   public synchronized E remove() throws NoSuchElementException {
     E e;
```

```
    int oldCount = -1;
    while (count.get() == 0) wait();

    e = queue.remove();
    oldCount = count.getAndDecrement();
    if (oldCount == this.capacity) {
      notifyAll(); // notify other waiting threads (could be producers or consumers)
    }
    return e;
  }

  /* Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. *
  public E peek() {
    if (count.get() == 0) return null;
    synchronized(this) {
      return queue.peek();
    }
  }
}
```

Notice that if the queue was empty before `add` or full before `remove`, we need to notify other waiting threads to unblock them.
We only need to emit such notifications in the above two cases since otherwise there cannot be any waiting threads.

Here is an implementation with locks.

We use two Reentrant Locks [http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html] to replace the use of synchronized methods. With separate locks for put and take, a consumer and a producer can access the queue at the same time (if it is neither empty nor full). A reentrant lock provides the same basic behaviors as a Lock [http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html] does by using synchronized methods and statements. Beyond that, it is owned by the thread last successfully locking and thus when the same thread invokes `lock()` again, it will return immediately without lock it again.

Together with lock, we use Condition [http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Condition.html] to replace the object monitor (wait and notifyAll). A Condition instance is intrinsically bound to a lock. Thus, we can use it to signal threads that are waiting for the associated lock. Even better, multiple condition instances can be associated with one single lock and each instance will have its own wait-thread-set, which means instead of waking up all threads waiting for a lock, we can wake up a predefined subset of such threads. Similar to `wait()`, `Condition.await()` can atomically release the associated lock and suspend the current thread.

We use Atomic Integer [http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html] for the count of elements in the queue to ensure that the count will be updated atomically.

```
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.ReentrantLock;

public class BoundedBlockingQueue<E> {
  private final Queue<E> queue = new LinkedList<E>();
  private final int capacity;
  private final AtomicInteger count = new AtomicInteger(0);

  private final ReentrantLock putLock = new ReentrantLock();
  private final ReentrantLock takeLock = new ReentrantLock();

  private final Condition notFull = putLock.newCondition();
  private final Condition notEmpty = takeLock.newCondition();

  public BoundedBlockingQueue(int capacity) {
    if (capacity <= 0)  throw new InvalidArgumentException("The capacity of the queue must be > 0.");
    this.capacity = capacity;
  }

  public int size() {
    return count.get();
  }

  public void add(E e) throws RuntimeException {
    if (e == null) throw new NullPointerException("Null element is not allowed.");

    int oldCount = -1;
    putLock.lock();
    try {
      // we use count as a wait condition although count isn't protected by a lock
      // since at this point all other put threads are blocked, count can only
      // decrease (via some take thread).
      while (count.get() == capacity) notFull.await();

      queue.add(e);
      oldCount = count.getAndIncrement();
      if (oldCount + 1 < capacity) {
        notFull.signal(); // notify other producers for count change
      }
    } finally {
      putLock.unlock();
    }
    // notify other waiting consumers
    if (oldCount == 0) {
      takeLock.lock();
      try {
        notEmpty.signal();
      } finally {
        takeLock.unlock();
      }
    }
  }

  public E remove() throws NoSuchElementException {
    E e;

    int oldCount = -1;
    takeLock.lock();
    try {
      while (count.get() == 0) notEmpty.await();

      e = queue.remove();
      oldCount = count.getAndDecrement();
      if (oldCount > 1) {
        notEmpty.signal(); // notify other consumers for count change
      }
    } finally {
      takeLock.unlock();
    }

    // notify other waiting producers
    if (oldCount == capacity) {
      putLock.lock();
      try {
        notFull.signal();
      } finally {
        putLock.unlock();
      }
    }

    return e;
  }

  /* Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. *
  public E peek() {
    if (count.get() == 0) return null;

    takeLock.lock();
    try {
      return queue.peek();
    } finally {
      takeLock.unlock();
    }
  }
}
```

Posted 19th August 2013 by Sophie

Labels: Design, Java, MultiThreading, Queue

3 View comments

18th August 2013 Implement Iterator for Perfect Power Generator

Implement Iterator for Perfect Power Generator [http://www.careercup.com/question?id=8591375]

Provide an implementation of the following interface:

```
public interface Powers extends Iterator<Long> {
  /* Returns the next integer a in the arithmetic sequence of integers where
   * a = m^n, m > 1, n > 1, and m and n are both integers
   * Thus, the first few outputs will be 4, 8, 9, 16, 25, 27, 32, 36, etc.
   */
  public Long next();

  /* Resets the sequence to the beginning, such that the next call to next()
   * will return 4.
   */
  public void reset();
}
```

Solution

The key point of this problem is to find the next m^n.

One way is to maintain a min-heap of several power values of prime numbers.

- At beginning, the heap only contains [2, 4]. It is ordered by power values (not base)
- Next, return 4.
  Inside the iterator, increase 4 to 8 by multiplying 2, i.e. remove heap top [2, 4] and add [2, 8].
  Also find prime numbers between old heap top 4 and new heap top 8, which is 3.
  Add [3, 9] to the heap.
- Next, return 8.
  Remove [2, 8] and add [2, 16].
  Find prime numbers between 8 and 9, which is null.
- Next, return 9.
  Remove [3, 9] and add [3, 27].
  Find prime numbers between 9 and 16, which are 5, 7, 11, and 13.
  Add [5, 25], [7, 49], [11, 121], [13, 169] to the heap.
- ... ...

As we discussed in previous post [http://n00tc0d3r.blogspot.com/2013/08/prime-numbers.html] , finding prime numbers under a given number n takes O(n) time. So, as next-value goes up, the running time of `next` method becomes O(n) where n is the returned value.
In the meantime, the size of the min-heap grows up too. As next-value goes up, the space complexity becomes O(m) where m is the number of primes under n.

Another way is to find the next-value on the fly, i.e find a perfect power [http://en.wikipedia.org/wiki/Perfect_power] between n and n*k where n is last return next-value and k is the corresponding base-value.

Given a number n, there are a couple of ways to test whether it is a perfect power.
Suppose n = a^b (b>1)

- Method 1: We have logn = b*loga, i.e. b = logn/loga. So, for each 2 <= a < n/2, check whether logn/loga is an integer.
  Assumes that `Math.log` runs in time O(logn). So this algorithm runs in time O(logn^2).
- Method 2: We also have b < logn+1. So, we can use binary search to find a in [2, n/2) such that a^b == n.
  There are logn b's and binary search over n/2 numbers takes time O(logn). So, this algorithm also runs in time O(logn^2).

So, the overall running time of `next` method is O(nlognlogn).

```
public class PowersImpl implements Powers {
  private long next = 4;
  private int base = 2;

  /* b = logn/loga */
  private int isPerfectPower(long n) {
    for (int a=2; a<n/2; ++a) {
      double b = Math.log(n) / Math.log(a);
      if (b - (int)b == 0) return a;
    }
    return -1;
  }

  /* binary search */
  public int isPerfectPower(long n) {
    for (int b=2; b<(int)(Math.log(n)/Math.log(2)+1); ++b) {
      long l = 2, r = n/2;
      while (l <= r) {
        long mid = l + (r - l) / 2;
        long v = (long)Math.pow(mid, b);
        if (v == n) return (int)mid;
        if (v > n) {
          r = mid - 1;
        } else {
          l = mid + 1;
        }
      }
    }
    return -1;
  }

  @Override
  public Long next() {
    long pre = next;
    next = next*base;
    for (long i=pre+1; i<next; ++i) {
      int a = isPerfectPower(i);
      if (a > 0) {
        next = i;
        base = j;
        break;
      }
    }
    return pre;
  }

  @Override
  public void reset() {
    next = 4; base = 2;
  }
}
```

Posted 18th August 2013 by Sophie

Labels: Array, Design, Java, Maths

1 View comments

15th August 2013

Find First Non-Repeating Character in A String

Find First Unique Character in A String [http://www.geeksforgeeks.org/given-a-string-find-its-first-non-repeating-character/]

Given a string, find the first non-repeating character in the string.

For example, given "geeksforgeeks", return 'f'.

Solution

A brute-force solution is to loop through the string and for each letter, check remaining letters to see whether it has duplicates. The average and worst-case running times are both O(n^2) which is bad.

Another solution is to use a HashMap to track the occurrence of each letter.
We iterate through the string twice: the first one is to compute the occurrences; and the second one is to find the first letter with occurrence = 1.

```
public static Character findFirstUnique(String s) {
  // map: char -- occurrence
  HashMap<Character, Integer> map = new HashMap<Character, Integer>();

  // loop 1: compute occurrence
  for (int i=0; i<s.length(); ++i) {
    char c = s.charAt(i);
    if (map.containsKey(c)) {
      map.put(c, map.get(c)+1);
    } else {
      map.put(c, 1);
    }
  }

  // loop 2: find the first unique
  for (int i=0; i<s.length(); ++i) {
    char c = s.charAt(i);
    if (map.get(c) == 1) return c;
  }

  return null;
}
```

This algorithm runs in time O(2n)=O(n) and takes O(n) spaces.

Do we have to iterate through the entire string twice? For a string like "aaaaaaaaeeeeeeeeeeddddddddf", if we can avoid revisiting all of the duplicates, the running time will be improved (still the same order though).

So, ideally, besides the hashmap for occurrences, if we could have a data structure to store the known unique letters in order and remove ones that have at least one duplicates, then at the end of the first loop the first element is what we are looking for.

What we need for the data structure are:

- O(1) time for operations like insert, remove, contains
- elements are in insertion order

LinkedHashSet [http://docs.oracle.com/javase/6/docs/api/java/util/LinkedHashSet.html] seems to be a good option here. LinkedHashSet can be thought as HashSet plus Doubly Linked List.

```
import java.util.LinkedHashSet;

public Character findFirstUnique(String s) {
  // map: char -- occurrence
  HashMap<Character, Integer> map = new HashMap<Character, Integer>();
  LinkedHashSet<Character> set = new LinkedHashSet<Character>(s.length());

  // loop: compute occurrence
  for (int i=0; i<s.length(); ++i) {
    char c = s.charAt(i);
    if (map.containsKey(c)) {
      set.remove(c);
    } else {
      map.put(c, 1);
      set.add(c);
    }
  }

  // find the first unique in the ordered set
  Iterator<Character> it = set.iterator();
  if (it.hasNext()) return it.next();
  return null;
}
```

This algorithm runs in time O(n) and takes O(2n)=O(n) spaces. Note that it increases the complexity of implementation and maintaining two hash table also requires some extra time.

Is the O(n) time optimal?
Yes. Even to verify whether a given character is unique in the string, it requires to iterate through the entire string and compare each letter, which is O(n) time.

Posted 15th August 2013 by Sophie

Labels: Hash, Java, String

0 Add a comment

12th August 2013

Implement Iterator for Array

Official Java Documents for Iterable Interface: java.lang.Iterable [http://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html] .
Official Java Documents for Iterator Interface: java.util.Iterator [http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html] .

```
public class MyIterable<E> implements Iterable<E> {
  E[] elems;
  int nElems;

  /**
   * Constructor.
   */
  public MyIterable(E[] data) {
    this.elems = data;
    nElems = data.length;
  }

  /**
   * Returns an iterator over a set of elements of type E.
   */
  public Iterator<E> iterator() {
    return new MyIterator<E>();
  }

  private class MyIterator<E> implements Iterator<E> {
    int nextElem = 0;
    boolean hasRemoved = true;

    /**
     * Returns true if the iteration has more elements.
     * In other words, returns true if next() would return an element
     * rather than throwing an exception.
     */
    public boolean hasNext() {
      return nextElem < nElems;
    }

    /**
     * Returns the next element in the iteration.
     *
     * Throws: java.util.NoSuchElementException if the iteration has no more elements
     */
    public E next() {
      if (!hasNext()) {
        throw new NoSuchElementException("No more elements");
      }

      E e = elems[nextElem];
      ++nextElem;
      hasRemoved = false;
      return e;
    }

    /**
     * Removes from the underlying collection the last element returned by this iterator.
     * This method can be called only once per call to next(). The behavior of an iterator
     * is unspecified if the underlying collection is modified while the iteration is in
     * progress in any way other than by calling this method.
     *
     * Throws: java.lang.IllegalStateException if the next method has not yet been called,
     * or the remove method has already been called after the last call to the next method.
     */
    public void remove() {
      if (hasRemoved) {
        throw new IllegalStateException(
        "The remove method can only be called once and after the next method.");
      }

      if (nextElem < nElems - 1) {
        System.arraycopy(elems, nextElem+1, elems, nextElem, nElems-nextElem-1);
      }
      --nElems;
      hasRemoved = true;
    }
  }
}
```

Posted 12th August 2013 by Sophie

Labels: Array, Design, Java

0 Add a comment

12th August 2013

Prime Numbers

Find All Prime Numbers [http://www.geeksforgeeks.org/sieve-of-eratosthenes/]

Given a number n, print all primes smaller than or equal to n. It is also given that n is a small number.

For example, if n is 10, the output should be “2, 3, 5, 7″. If n is 20, the output should be “2, 3, 5, 7, 11, 13, 17, 19″.

Solution

A Prime number [http://en.wikipedia.org/wiki/Prime_number] is a natural number which has exactly two distinct natural number divisors: 1 and itself.

A naive solution is for each number k <= n check whether k is divisible by any number less than k. If not, mark it as a prime number. Obviously, this algorithm takes O(n^2) time.

Let's take another view point of this problem.

Finding all prime numbers is equivalent to find the remaining numbers after removing non-prime numbers. That is, if we remove all even numbers, then all multiples of 3, all multiples of 5, ..., eventually, the remaining numbers are the prime numbers.

```
public void printPrimes(int n) {
  if (n < 2)  return; // A prime number must be >= 2.
  boolean[] flags = new boolean[n+1]; // by default initialized to false

  int rootN = (int)Math.sqrt(n);
  for (int i=2; i<=rootN; ++i) {  // [1]
    // flags[i]==true means it has been marked before and thus all its multiples can be skipped
    if (!flags[i]) {

      // Note: if we get here, i itself is a prime number

      // mark its multiples
      for (int j=i*i; j<=n; j+=i) {  // [2]
        flags[j] = true;
      }
    }
  }

  // print all primes
  System.out.println("Prime numbers less than " + n + ":");
  for (int i=2; i<=n; ++i) {
    if (!flags[i]) System.out.print(i + ", ");
  }
}
```

There are two small optimizations as marked in the code:

1. Since `n = sqrt(n) * sqrt(n)`, for any number greater than `sqrt(n)`, it is either a prime number or it is a multiple of some number that is no greater than `sqrt(n)`.
   Proof: Suppose there is a number, say k<=n, such that it is not a prime number and it is > `sqrt(n)` and it is not a multiple of any number <= `sqrt(n)`. Then it must has at least two divisors such that both of them are > `sqrt(n)`. It implies that k > n, which is a contradiction to our assumption.
2. Similarly, to mark all multiples of i (except itself since it itself is a prime number :), we can start from i*i since for any i*j, where 1<j<i, must have been taken care of in previous rounds.

~~With these optimizations, each number will be marked at most once. Thus, this algorithm runs in time O(n).~~

- ~~For prime numbers > sqrt(n), it touches each once for printing;~~
- ~~For numbers <= sqrt(n), it touches each at most twice, for marking and maybe printing.~~

Update (thanks to Keith): With these optimizations, each number will be marked k_i times, where k_i is the number of prime factors of the number i. E.g. 12 -- two times, one for 2 and the other for 3; 105 - 3 times, one for 3, one for 5 and one for 7.
In total, there are n numbers, for each prime x, we will mark n/x numbers, i.e. the total running time T(n) = n*(1/2 + 1/3 + 1/5 + ...) = n * sum_{i = all primes less than n} (1 / i). This series is called, Prime Harmonic Series [http://en.wikipedia.org/wiki/Prime_harmonic_series] , which has been proved to be asymptotic to loglogn. Therefore, the total

running time T(n) = O(n*loglogn).

This algorithm takes O(n) spaces.

This algorithm is called Sieve of Eratosthenes [http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes] .

Is it a Prime Number? [http://www.careercup.com/question?id=12342686]

Given number n, return true if n is prime otherwise false.

Solution

To verify whether a number is a prime number, we need to check whether it has any divisor other than itself and > 1.

As proved in previous problem, instead of searching divisor among all numbers in (2..n), we only need to check numbers in [2..`sqrt(n)`].

```
public boolean isPrime(int n) {
  if (n < 2) return false; // prime must >=2
  if (n == 2) return true;
  if ((n & 1) == 0) return false; // prime must not be even

  int rootN = (int)Math.sqrt(n);
  for (int i=2; i<=rootN; ++i) {
    if (n % i == 0) return false;
  }
  return true;
}
```

This algorithm runs in time O(sqrt(n)) and uses O(1) space.

Find All Prime Factors [http://www.geeksforgeeks.org/print-all-prime-factors-of-a-given-number/]

Given a number n, print all prime factors of n.

For example, if the input number is 12, then output should be “2 2 3″. And if the input number is 315, then output should be “3 3 5 7″.

Solution

After completing the previous problems, the intuitive solution for this problem is to first find all prime numbers less than `sqrt(n)` and then iterate through those numbers to find all factors of n.

Finding all prime numbers less than `sqrt(n)` takes O(`sqrt(n)`) time and space. Do we really need to find all prime numbers?

No. If we divide n by factors we found, for any following non-prime numbers, they will not be a divisor of n. For example, if we already check 2 and 3 and "remove" them from n, then (n % 6) must **not** equal 0.

```
public static void printPrimeFactors(int n) {
  if (n < 2) return ; // prime must >=2

  System.out.println("Prime factors of " + n + ":");

  while ((n & 1) == 0) {
    System.out.print(2 + " ");
    n /= 2;
  }

  int rootN = (int)Math.sqrt(n);
  for (int i=3; i<=rootN; ++i) {
    while (n % i == 0) {
      System.out.print(i + " ");
      n /= i;
    }
  }

  if (n > 2) { // there is at most one prime factor > sqrt(n)
    System.out.print(n);
  }
}
```

This algorithm runs in O(m) time where m is the number of prime factors.

Posted 12th August 2013 by Sophie

Labels: Java, Maths

4 View comments

12th August 2013

Reservoir Sampling And Variants

Reservoir Sampling [http://www.geeksforgeeks.org/reservoir-sampling/]

Choose a sample of k items from a list S containing n items, where n is either a very large or unknown number. The probability of picking the k items should be k/n.

You can assume n is too large to fit all items in memory.

Solution

Since the sample size is large, we are aiming at an one-pass solution that selects all k samples and only goes through the original data points once.

Let's start from k=1.
Suppose we have had one sample, p, selected from S[0..i-1]. What's the probability to select S[i] to replace the sample? That is,

- P(take S[i] as a sample of i+1 items) = 1/(i+1)
- P(take p as a sample of i items) = 1/i
- P(use S[i] to replace p | given p) = ??

Note that after piping in S[i], there are two possible outputs: p is the sample; or, S[i] is the sample. Either way, the overall probability of selecting a sample is 1/(i+1). So,

- P(pick p as a sample of i+1 items | given p) = i/(i+1)
- P(pick S[i] as a sample of i+1 items | given p) = 1 - i/(i+1) = 1/(i+1)

Now, let's move to the case of k>1.
Suppose we have a reservoir pool A of k samples from S[0..i-1]. From the formula above, it is easy to imply that the probability of pick S[i] as a sample is k/(i+1). So, for each newly piped-in item, we generate a random number, j, of [0, i] and if j < k, replace A[j] with S[i].

```
public void selectKSamples(int[] data, int k) {
  int n = data.length;
  if (n < k) return;
  if (n == k) {
    printArray(data);
    return;
  }

  int[] pool = new int[k];
  for (int i=0; i<k; ++i) {
    pool[i] = data[i];
  }
  // random pick
  for (int i=k; i<n; ++i) {
    int rand = (int)(Math.random() * (i+1)); // generate a random number of [0,i]
    if (rand < k) {
      pool[rand] = data[i];
    }
  }

  printArray(pool);
}

private void printArray(int[] A) {
  for (int num : A) System.out.print(num + " ");
  System.out.println("");
}
```

This algorithm runs in O(n) time and uses O(k) spaces which is optimal.

This technology is called **Reservoir Sampling**. If you want a formal proof of the algorithm, check out this wiki [http://en.wikipedia.org/wiki/Reservoir_sampling] page.

Random Sampling in Linked List

Given a linked list, generate a random node from the list. You are allowed to use only O(1) space.

Solution

The input is in the form of linked list and we are not allowed to use extra spaces, so we cannot check previously visited nodes (well, we can but that will lead to a O(n^2) solution).

This is a variant of the above problem where k=1.

```
public class ListNode {
  public int val;
  public ListNode next;
  ListNode(int val) {
    this.val = val; this.next = null;
  }
}

public ListNode selectASamples(ListNode head) {
  ListNode sample = null;
  for (int count=1; head!=null; head = head.next, ++count) {
    // generate a randome number of [0, count-1]
    if (count == 1 || (int)(Math.random() * count) == count - 1) {
      sample = head;
    }
  }

  return sample;
}
```

This algorithm runs in time O(n) and takes O(1) space.

Posted 12th August 2013 by Sophie

Labels: Java, LinkedList, Maths

2 View comments

22nd July 2013

Optimal Game Strategy: Maximum Coin Value

Maximum Coin Value [http://www.geeksforgeeks.org/dynamic-programming-set-31-optimal-strategy-for-a-game/]

Consider a row of n coins of values (v1, ..., vn), where n is even. We play a game against an opponent by alternating turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can definitely win if we move first.

Note: The opponent is as clever as the user.

Solution - Recursion

Translate the problem into mathematical language.
Let C(i, j) be coin values from i-th to j-th spot in the row and V[i] be the coin value at i-th spot.

- If we pick V[i], the opponent will take C[i+1] or V[j] so that the remaining give us minimal value, and then we continue on C(i+2, j) or C(i+1, j-1);
- If we pick V[j], the opponent will take V[i] or C[j-1] so that the remaining give us minimal value, and then we continue on C(i+1, j-1) or C(i, j-2).

Note that pick up the maximal (i.e. max(V[i], V[j])) in each round may not make you win with the overall maximum at the end. For example, given (10, 20, 8, 7), the maximum amount is 7+20=27.

So, the maximum amount of coins can be obtained from the following recursive formula:

```
maxCoin(i, j) = max(V[i] + min(maxCoin(i+2, j), maxCoin(i+1, j-1))
                    V[j] + min(maxCoin(i+1, j-1), maxCoin(i, j-2)));
maxCoin(i, j) = V[i], if (i == j); maxCoin(i, j) = max(V[i], V[j]), if (i+1 == j).
```

It is easy to get a recursive function based on the above formula.
The running time of this algorithm is O(2^n) since we recalculate maxCoin for subarrays again and again.
Note: T(n) = O(1) + 4*T(n-2) = O(1) + 4*O(1) + 4*4*T(n-4) = ... = 1 + 4 + 4^2 + ... + 4^(n/2) = O(2^n).

Solution - DP

These repeat calculations give us a hint of using DP to store the intermediate results so as to reduce running time.

Suppose we have a 2-D table maxCoin[i, j] for max coin value of coins from i-th to j-th spot in the row. Now let's discuss how to fill up this table. As shown in the picture below, value of cell [i, j] can be calculated on three cells on the previous diagonal. So, we need to fill up the table diagonal by diagonal, from center diagonal to upper-right corner.

[http://4.bp.blogspot.com/-hNUZajDEN1k/UewzbL8imal/AAAAAAAAEjE/t1ufnVy887U/s1600/Screen+Shot+2013-07-21+at+12.15.33+PM.png]

This algorithm runs in time O(n^2) and uses O(n^2) extra space.

```
public int getMaxCoin(int[] coins) {
  int n = coins.length;
  int[][] maxCoin = new int[n][n];

  // fill up the center diagomal
  for (int i=0; i<n; ++i) {
    maxCoin[i][i] = coins[i];
  }

  // fill up the table
  for (int k=1; k<n; ++k) {
    for (int i=0, j=k; i<n-k; ++i, ++j) {
      int left = (i+2 <= j) ? maxCoin[i+2][j] : 0;
      int bottom = (j-2 >= i) ? maxCoin[i][j-2] : 0;
      maxCoin[i][j] = Math.max((coins[i] + Math.min(left, maxCoin[i+1][j-1])),
                     (coins[j] + Math.min(maxCoin[i+1][j-1], bottom)));
    }
  }

  return maxCoin[0][n-1];
}
```

Note: This algorithm works for both cases where n is even or odd.

Solution - Another DP

Actually, since we assume that the opponent is as clever as ourselves, he will use the same strategy to pick up coins. That said,

- If we pick V[i], the money that opponent will get would be C(i+1, j);
- Similarly, if we pick V[j], the opponent would get C(i, j-1).

Now the goal is simply to select a coin such that the money that the opponent could achieve from the remaining coins is minimized.

Given coins from i-th to j-th, suppose we know the money of the opponent, how much can we get?
It will be the total value of all coins minus the coin values of the opponent!

The formula of the maximum coin value can be simplified as:

> maxCoin(i, j) = max(Sum(i, j) - C(i+1, j), Sum(i, j) - C(i, j-1));
> maxCoin(i, j) = V[i], if (i == j); maxCoin(i, j) = max(V[i], V[j]), if (i+1 == j).

Unfortunately, The running time of this recursive algorithm is still O(2^n).
Note: T(n) = O(1) + 2*T(n-1) = O(1) + 2*O(1) + 2*2*T(n-2) = 1 + 2 + 4 + ... + 2^(n-1) = O(2^n).

But this formula can also lead to a DP solution.

Here we use one single n by n matrix for sum and maxCoin: the upper-right triangle for maxCoin and the bottom-left one for sum.

```
public int getMaxCoin(int[] coins) {
  int n = coins.length;
  // For i<j, T[i][j] = maxCoin(i, j)
  // For i>j, T[j][i] = sum(j, i)
  int[][] T = new int[n][n];

  for (int i=0; i<n; ++i) {
    T[i][i] = coins[i];
  }

  for (int k=1; k<n; ++k) {
    for (int i=0, j=k; i<n-k; ++i, ++j) {
      // maxCoin(i, j) = max( Sum(i, j) - C(i+1, j), Sum(i, j) - C(i, j-1) );
      T[j][i] = T[j-1][i] + coins[j];
      T[i][j] = Math.max(T[j][i] - T[i+1][j], T[j][i] - T[i][j-1]);
    }
  }

  return T[0][n-1];
}
```

This algorithm has the same time and space complexity but the formula is simplified and no need to consider odd and even cases.

Posted 22nd July 2013 by Sophie

Labels: DP, Java, Recursion

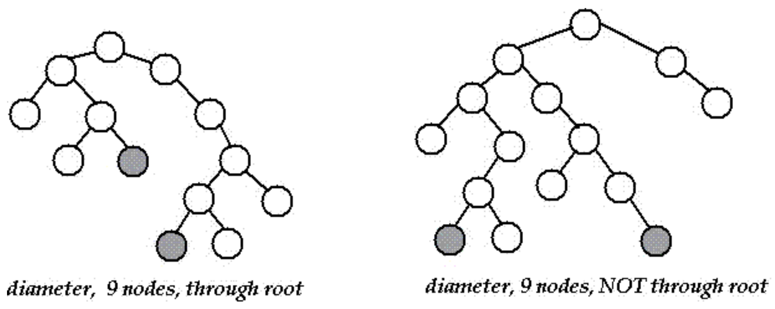
7 View comments

20th July 2013

Diameter of a Binary Tree

Diameter of a Binary Tree [http://www.geeksforgeeks.org/diameter-of-a-binary-tree/]

The **diameter** of a tree (sometimes called the width) is the number of nodes on the longest path between two leaves in the tree. The diagram below shows two trees each with diameter nine, the leaves that form the ends of a longest path are shaded (note that there is more than one path in each tree of length nine, but no path longer than nine nodes).


diameter, 9 nodes, through root
diameter, 9 nodes, NOT through root

[http://geeksforgeeks.org/wp-content/uploads/tree_diameter.GIF]

Solution

This problem is similar to the "Binary Tree Max Path Sum" problem that we discussed in previous post [http://n00tc0d3r.blogspot.com/2013/01/tree-path-sum.html] .

The diameter of a tree T is

> Diameter(T) = max(Diameter(T.left), Diameter(T.right), Height(T.left)+Height(T.right)+1)

We can "translate" this formula to code with a recursive helper method to get height of a given tree, `height(TreeNode root)`. But you may notice that height of subtrees have been repeatedly computed.
Suppose the tree is of height m and root is at level 0. It visits each node at i-th level i+1 times. The worst case running time is O(n^2).

To reduce the running time, either we calculate heights, store height of each node in a HashMap, and run another pass to calculate the diameter; or we calculate height and diameter in the same recursion.

Both algorithms have worst-case running time O(n) but the former one requires O(n) extra space.

Here is an implementation for latter algorithm which runs in time O(n) with O(1) space.
Notice that Java cannot return two values and pass-in arguments are passed in by reference. So, we create an object to store height and diameter so that they could be updated through recursions.

```
private class Data {
  public int height;
  public int diameter;
}

private void diameter(TreeNode root, Data d) {
  if (root == null) {
    d.height = 0; d.diameter = 0; return;
  }
  diameter(root.left, d); // get data in left subtree
  int hLeft = d.height;
  int dLeft = d.diameter;
  diameter(root.right, d); // overwrite with data in right tree
  d.diameter = Math.max(Math.max(dLeft, d.diameter), hLeft+d.height+1);
  d.height = Math.max(hLeft, d.height) + 1;
}

public int diameter(TreeNode root) {
  Data data = new Data();
  diameter(root, data);
  return data.diameter;
}
```

Posted 20th July 2013 by Sophie

Labels: BinaryTree, Java, Recursion

3 View comments

18th July 2013

Find Majority

Find Majority [http://www.geeksforgeeks.org/majority-element/]

A majority element in an array A[] of size n is an element that appears more than n/2 times (and hence there is at most one such element).

Write a function which takes an array and returns the majority element (if it exists), otherwise returns null.
For example:
Given an array of [3, 3, 4, 2, 4, 4, 2, 4, 4], return 4;
Given an array of [3, 3, 4, 2, 4, 4, 2, 4], return null.

Solution

Intuitively, we can scan the array and use a hash map to record the occurrence of each element. Then, find out the one with maximum occurrence by iterating through the key set of the hash map.
This algorithm is a two-pass algorithm and thus it runs in time O(n) with O(n) space usage.

We can also complete it with one-pass if we keep tracking curMax when counting the occurrence. But, still, this requires O(n) extra space.

Another idea is to sort the array in O(nlogn) time and scan the sorted array to find the majority element.
This algorithm can take O(1) extra space (depending on the sorting algorithm) but require time O(nlogn+n) = O(nlogn).
Note: If we know there must exist a majority element, then the middle element, A[(n+1)/2], is the goal.

Think it in this way: If we "cancel" every pair of different elements and if there are any elements left, it must be the majority element.
For example, given [3, 3, 4, 2, 4, 4, 2, 4, 4],

- [3, 3, 4, 2, 4, 4, 2, 4, 4]
- [3, 4, 3, 2, 4, 4, 2, 4, 4]
- [3, 4, 3, 2, 4, 4, 2, 4, 4]

We need two pointers, left and right, pointing to unequal elements, cancel them, and forward the two pointers to next valid positions.
There are two cases:

- left and right are pointer to adjacent elements, then after canceling, move both pointers by 2;
- otherwise, swap right element with the one next to left, i.e. A[left+1], and cancel A[left] and A[left+1], move left by 2 and move right by 1.

After moving pointers and cancellations, the last element in the resulting array will be a candidate of the majority element. We need to verify it by scanning through the array and count the occurrence of that one.

This algorithm runs in linear time using constant space. Here is an implementation.

```
private void swap(int[] data, int a, int b) {
  int temp = data[a];
  data[a] = data[b];
  data[b] = temp;
}
```

```
public Integer findMajority(int[] data) {
  int l = 0, r = 1, n = data.length;
  while (r < n) {
    if (data[r] == data[l]) { // forward r if it is not equal to l
      ++r; continue;
    }
    if (l == r-1) { // "cancel" adjacent elements
      r += 2;
      l += 2;
    } else { // swap r to l+1 and "cancel" l and l+1
      swap(data, r, l+1);
      ++r;
      l += 2;
    }
  }

  // verify if it is majority
  int count = 0;
  for (int i=0; i<n; ++i) {
    if (data[i] == data[n-1]) ++count;
    if (n - count < count) return data[n-1];
  }
  return null;
}
```

The downside of this algorithm is that we modified the original array (so did the sorting algorithm).

There is a better way to do the cancelling: Moore's voting algorithm [http://www.cs.utexas.edu/~moore/best-ideas/mjrty/index.html] . You can find the paper and a step-by-step demo for the algorithm. This stackoverflow post [http://stackoverflow.com/questions/3740371/finding-the-max-repeated-element-in-an-array] gives a graphical explanation.

The basic idea is to have a counter and an index to a candidate of majority element.

- Scan through the array.
    - If the counter is zero, no previous elements or all previous one have been cancelled, then take the next element as the candidate majority element.
    - If the current element is equal to the candidate majority, increase the counter; Otherwise, decrease the counter.
- Scan through the array one more time to verify the candidate.

This algorithm runs in time O(n) using O(1) space without modifying the original array.

```
public static Integer findMajority(int[] data) {
  int major = 0, count = 1, n = data.length;
  for (int i=1; i<n; ++n) {
    if (count == 0) { // all previous ones have been cancelled, start over
      major = i;
      count = 1;
    } else if (data[i] == data[major]) {
      ++count;
    } else {
      --count;
    }
  }

  // verify if it is majority
  count = 0;
  for (int i=0; i<n; ++i) {
    if (data[i] == data[major]) ++count;
    if (n - count < count) return data[major];
  }
  return null;
}
```

Posted 18th July 2013 by Sophie

Labels: Array, Hash, Java

4 View comments

15th July 2013

Word Ladder II

Word Ladder II [http://leetcode.com/onlinejudge#question_126]

(This is a follow up question for Word Ladder [http://n00tc0d3r.blogspot.com/2013/07/word-ladder.html] .)

Given two words (start and end), and a dictionary, find all shortest transformation sequence(s) from start to end, such that:

- Only one letter can be changed at a time
- Each intermediate word must exist in the dictionary

Return 0 if there is no such transformation sequence.
Assume that all words have the same length and contain only lowercase alphabetic characters.

For example, given start = "hit", end = "cog", dict = ["hot","dot","dog","lot","log"], return

```
[
    ["hit","hot","dot","dog","cog"],
    ["hit","hot","lot","log","cog"]
]
```

Solution

As we discussed in previous post, usually BFS is better for finding the minimum length of shortest path while DFS is better for finding the actual shortest path. So, for this problem,

- Use BFS to find the minimum sequence length and store all back traceable paths from a word to its parent word in upper level.
  Note that we need to check the level information to avoid circle in a graph.
  For example, "red" -> "ted" -> "red".
- Use DFS to backtrace all paths from end to start.
  We need to use DFS to get all paths since one word may have multiple parents word from previous level.
  For example, given "hot", "dog", and dictionary ["hot","dog","hog","hop","dot"], "dog" can come from "dot" or "hog".

Suppose n is the number of paths and m is the path length. The DFS method shown below takes time O(n*m) which is optimal since we need at least that amount of time to populate all paths.

The BFS method only explores levels as many as needed and stops as soon as we finish the last level which is at minimum depth and which contains the goal. During BFS, it takes O(26*k) time to find all candidate words in next level that can be transformed to. Overall, in worst case, it may take O(k*N) time where N is the total number of words in the dictionary.

```
public ArrayList<ArrayList<String>> findLadders(String start, String end, HashSet<String> dict) {
  ArrayList<ArrayList<String>> paths = new ArrayList<ArrayList<String>>();
  if (start == null || end == null || start.length() == 0) return paths;

  // maintain a hashmap for visited words
  Map<String, ArrayList<Node>> visited = new HashMap<String, ArrayList<Node>>();

  // BFS to find the minimum sequence length
  getMinLength(start, end, dict, visited);

  // DFS to back trace paths from end to start
  buildPaths(end, start, visited, new LinkedList<String>(), paths);

  return paths;
}

/* Use BFS to find the minimum transformation sequences length from start to end.
  Also store parent nodes from previous level for each visited valid word. */
private void getMinLength(String start, String end, HashSet<String> dict,
    Map<String, ArrayList<Node>> visited) {
  // maintain a queue for words, depth and previous word during BSF
  Queue<Node> queue = new LinkedList<Node>();
  queue.add(new Node(start, 1));
  // BFS
  dict.add(end);
  int lastLevel = 0;
  while (!queue.isEmpty()) {
    Node node = queue.poll();
    if (lastLevel > 0 && node.depth >= lastLevel) break;
    // find transformable words in next level
    for (int i=0; i<node.word.length(); ++i) {
      StringBuilder sb = new StringBuilder(node.word);
      char original = sb.charAt(i);
      for (char c='a'; c<='z'; ++c) {
        if (c == original) continue;
        sb.setCharAt(i, c);
        String s = sb.toString();
        // if hits end, mark the current depth as the last level
        if (s.equals(end)) {
          if (lastLevel == 0) lastLevel = node.depth + 1;
        }
        if (dict.contains(s) && !s.equals(start)) {
          ArrayList<Node> pres = visited.get(s);
          if (pres == null) {
            // enqueue unvisited word
            queue.add(new Node(s, node.depth+1));
            pres = new ArrayList<Node>();
            visited.put(s, pres);
            pres.add(node);
          } else if (pres.get(0).depth == node.depth) {
            // parent nodes should be in the same level - to avoid circle in graph
            pres.add(node);
          }
        }
      }
    }
  }
  return lastLevel;
}

/* Use DFS to back trace all paths from end to start. */
```

```
private void buildPaths(String s, String start, Map<String, ArrayList<Node>> visited,
    LinkedList<String> path, ArrayList<ArrayList<String>> paths) {
  if (s == null || visited == null || path == null || paths == null) return;

  path.add(0, s);
  if (s.equals(start)) {
    ArrayList<String> p = new ArrayList<String>(path);
    paths.add(p);
  } else {
    ArrayList<Node> pres = visited.get(s);
    if (pres != null) {
      for (Node pre : pres) {
        buildPaths(pre.word, start, visited, path, paths);
      }
    }
  }
  path.remove(0);
}

private class Node {
  String word;
  int depth;
  public Node(String w, int d) {
    word = w; depth = d;
  }
}
```

Posted 15th July 2013 by Sophie

Labels: BFS, DFS, Graph, Java, Recursion, String, StringBuilder

1 View comments

15th July 2013

Word Ladder

Word Ladder [http://leetcode.com/onlinejudge#question_127]

Given two words (start and end), and a dictionary, find the length of shortest transformation sequence from start to end, such that:

- Only one letter can be changed at a time
- Each intermediate word must exist in the dictionary

Return 0 if there is no such transformation sequence.
Assume that all words have the same length and contain only lowercase alphabetic characters.

For example, given start = "hit", end = "cog", dict = ["hot","dot","dog","lot","log"],
return 5 since one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog".

Solution - One-time Query

A straightforward way to solve the problem is to use **DFS** such that

- From start, find out "neighbors" that can be transformed from start with one letter change;
- Recursively find out next neighbor, increasing depth and marking words as visited during DFS;
- When hits the end, compare current length with the minimum length and store the smaller one.

Although we skip visited nodes, in worst cases, it may end up with touching every word once. And for each word, to find out which word is a neighbor, we need to check every other word on the count of different letters, which takes O(n*m) time, where n is the number of words in the dictionary and m is the word length. So the total running time for this solution is O(n*n*m).

Notice that a valid transformation allows to change only one letter. So we can modify a word letter by letter and check whether the resulting word is in dict. If so, we find a neighbor.
By doing this, the total running time gets reduced to O(n*m*26) = O(n*m).

```
/* Assume start and end are of the same length. */
public int ladderLength(String start, String end, HashSet<String> dict) {
  if (start == null || end == null || start.length() == 0
      || start.length() != end.length() || start.equals(end))
    return 0;

  minLength = 0;
  DFS(start, end, dict, 1, new HashSet<String>());
  return minLength;
}

private int minLength = 0;
private void DFS(String s, String end, HashSet<String> dict, int length,
    HashSet<String> visited) {
  // mark s as visited
  visited.add(s);

  for (int i=0; i<s.length(); ++i) {
    StringBuilder sb = new StringBuilder(s);
    for (char c='a'; c<='z'; ++c) {
      if (c == s.charAt(i)) continue; // skip itself
      sb.setCharAt(i, c);
      String word = sb.toString();
      // if hits end, return length
      if (word.equals(end)) {
        if (minLength == 0 || length+1 < minLength)
          minLength = length + 1;
      // skip invalid
      } else if (!visited.contains(word) && dict.contains(word)) {
        // DFS
        DFS(word, end, dict, length+1, visited);
      }
    }
  }

  visited.remove(s);
}
```

Another way to solve the problem is to use **BFS** such that

- From start, find out all "neighbors" that can be transformed from start with one letter change;
- Instead of recursion, queue up all neighbors;
- When pop out a word from the queue, mark it as visited and queue up all its neighbors. If we hits the end, we can safely return the depth since BFS ensures that the current depth is the minimum.

In worst cases, there are still chances that we visit each word once. But on average, finding minimum depth with BFS is faster than that with DFS since we don't need to check all possible paths. The first time when we hits the goal in BFS, we know the current depth is the minimum depth.

In the algorithm below, we use another HashMap to maintain visited words. In Java, HashSet [http://docs.oracle.com/javase/6/docs/api/java/util/HashSet.html] is implemented on the Set interface but backed by a HashMap instance. So, in Java, HashMap will be slightly faster than HashSet if both works for you (In C++, on the contrary, set is not implemented on a map and when size gets large, the constant factor of looking up in a hashmap is not ignorable and thus set would be preferred in those cases).

We can also improve the algorithm by removing visited word from dictionary. Thus we don't need to keep another HashMap for visited words. But it needs to modify the dictionary and is not a good design in practise. We may pass in a copy of dictionary if we don't want to modify the original one.

```
/* Assume start and end are of the same length. */
public int ladderLength(String start, String end, HashSet<String> dict) {
  if (start == null || end == null || start.length() == 0
      || start.length() != end.length() || start.equals(end))
    return 0;

  return BFS(start, end, dict);
}

private class PathNode {
  String word;
  int length;
  public PathNode(String w, int l) {
    word = w; length = l;
  }
}
private int BFS(String start, String end, HashSet<String> dict) {
  // maintain a queue for words and path length during BSF
  Queue<PathNode> queue = new LinkedList<PathNode>();
  queue.add(new PathNode(start, 1));
  // maintain a hashmap for visited words
  Map<String, Boolean> visited = new HashMap<String, Boolean>();
  // BFS
  while (!queue.isEmpty()) {
    PathNode node = queue.poll();
    String s = node.word;
    // get current path length
    int length = node.length;
    // mark the word as visited
    visited.put(s, true);
    // find all words that can be transformed from s and store in queue with len+1
    if (dict.size() < 26) { // loop through all words in dict - O(n*m)
```

If we consider the dictionary as a group such that each node contains a word and an edge between two words meaning that they can transformed to each other with one letter change. Then the problem becomes to find the shortest path between two nodes, start and end. There is an existing algorithm to solve that particular problem, Dijkstra's algorithm [http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm] .

Dijkstra's algorithm can efficiently find out shortest paths from one node to every other node in the graph. It can also be used to find out shortest path from one node to another by stopping the algorithm when hits the goal. In this problem, the edge weights are all 1's and it is essentially an improved version of BFS with pre-populated edges.

Unfortunately, in this problem, we have to populate all edges from the given dictionary which requires at least O(n*m) time. But in previous BFS algorithm, we may not need to populate edges for all words, which gives a better performance in this case.

If the dictionary doesn't change frequently and we need to perform such queries several times, Dijkstra's algorithm will outperform BFS one since it only conduct one-time process for all edges. See next section for further discussion.

```
/* Assume start and end are of the same length. */
public int ladderLength(String start, String end, HashSet<String> dict) {
  if (start == null || end == null || start.length() == 0
      || start.length() != end.length() || start.equals(end))
    return 0;

  return Dijkstra(start, end, dict);
}


private int Dijkstra(String start, String end, HashSet<String> dict) {
  dict.add(start);
  dict.add(end);

  // convert to an array
  int n = dict.size();
  String[] words = new String[n];
  dict.toArray(words);

  int ss = -1, ee = -1;
  ArrayList<ArrayList<Integer>> neighbors = new ArrayList<ArrayList<Integer>>(n);
  // initialize
  for (int i=0; i<n; ++i) {
    ArrayList<Integer> neighbor = new ArrayList<Integer>();
    neighbors.add(neighbor);
  }
  // find all "edges" and index to start and end
  for (int i=0; i<n; ++i) {
    if (start.equals(words[i])) ss = i;
    if (end.equals(words[i])) ee = i;
    for (int j=i+1; j<n; ++j) {
      if (isNeighbor(words[i], words[j])) {
        ArrayList<Integer> neighbor = neighbors.get(i);
```

Solution - Repeated Queries

If we are going to run queries on the same dictionary for a large number of times, then it is worthy to preprocess the dictionary such that all shortest distances between all pairs of words in the dictionary are stored in a table. This can be solved use Floyd's algorithm [http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm] using O(n*m*26+n^3) time where n is the total number of words in the dictionary and m is word length.
Note: It will take O(n^2+n*m*26) time to update the distance table with a new word.

With that information in hand, for each query (start, end), we only need to

- Find the indices of starting and ending words in the dictionary.
- Look into the table to get the number of intermediate steps.

That said, for each following query, assuming start and end are both in the dictionary, the running time becomes O(1)!

```
/* Assume start and end are of the same length. */
public int ladderLength(String start, String end, HashSet<String> dict) {
  HashMap<String, Integer> dictIndex = new HashMap<String, Integer>();
  int n = 0;
  for (String word : dict) {
    dictIndex.put(word, n++);
  }

  // preprocess: find shortest dist for all pairs of words in dictionary
  // O(n^3), n is number of words in dictionary
  int[][] dist = Floyd(dictIndex);

  return dist[dictIndex.get(start)][dictIndex.get(end)]+1;
}

/* Calculate distances between words in dictionary and store them in a table.
  This algorithm is simiar to Floyd's algorithm for all pairs shortest paths. */
private int[][] Floyd(final HashMap<String, Integer> dictIndex) {
  int n = dictIndex.size();
  // dist[i][j]=k means words[i] can be transformed to words[j] in k steps
  // we only fill up upper trangle of dist matrix, i.e. i<=j
  int[][] dist = new int[n][n];

  // initialize to n+1 - O(n^2)
  for (int i=0; i<n; ++i) {
    for (int j=0; j<n; ++j) {
      if (i != j) dist[i][j] = n+1;
    }
  }

  // one-step transforms - O(n*m*26), m is word length
  for (String s : dictIndex.keySet()) {
```

Posted 15th July 2013 by Sophie

Labels: BFS, DFS, Graph, Java, Recursion, String, StringBuilder

4 View comments

12th July 2013 Word Search

Word Search [http://leetcode.com/onlinejudge#question_79]

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

For example,
Given board =

```
[
  ["ABCE"],
  ["SFCS"],
  ["ADEE"]
]
```

word = "ABCCED", -> returns true,
word = "SEE", -> returns true,
word = "ABCB", -> returns false.

Solution

Consider the board as a graph where edges connect adjacent cells. Now the problem becomes finding a path in a graph. Given a starting node, we can use DFS or BFS to find a path. But for this problem, we have to find the starting node first.

The basic idea is:

> Loop through every cell in the board and perform DFS to match the word. Since a same cell cannot be reused, we also need to keep track of visited cells during DFS and unmark it if it is not being used.

We can also use BFS, but it requires to store the intermediate results, which in this problem is a tuple (x, y). Creating objects for tuples is quite expensive.

```
public boolean exist(char[][] board, String word) {
  boolean[][] visited = new boolean[board.length][board[0].length];

  // find the starting char
  for (int i=0; i<board.length; ++i) {
    for (int j=0; j<board[0].length; ++j) {
      if (DFS(board, i, j, word, 0, visited)) return true;
    }
  }

  return false;
}

private boolean DFS(char[][] board, int x, int y, String word, int cur, boolean[][] visited) {
  // validate input
  if (visited[x][y] || board[x][y] != word.charAt(cur)) return false;
  if (cur == word.length() - 1) return true;

  // mark the node as visited
  visited[x][y] = true;

  // BFS on its neighbers
  if (x > 0 && DFS(board, x-1, y, word, cur+1, visited)) return true;
  if (x+1 < board.length && DFS(board, x+1, y, word, cur+1, visited)) return true;
  if (y > 0 && DFS(board, x, y-1, word, cur+1, visited)) return true;
  if (y+1 < board[0].length && DFS(board, x, y+1, word, cur+1, visited)) return true;

  // mark the node as unused
  visited[x][y] = false;
  return false;
}
```

There are O(n*m) candidates for starting cell. For each of them we run a DFS with branching factor 3 and depth at most k, where k is the length of the word. The worst case running time for DFS is O(3^k). The worst case total running time is (n*m*2^k).
In this algorithm, we use a boolean matrix to store visited information, which requires O(n*m) extra spaces. If we are allowed to modify the original matrix, we can mark a cell as visited by setting it to some special value, such as "#". By doing this, space complexity gets down to O(1). Again, this method modifies the original matrix which may not be a prefer way in practise.

Another way to solve the problem is

- Preprocess the board as a hash map of <char, a list of cells containing the char>.
  This takes O(n*m) time and O(n*m) spaces.
- Loop through characters in the given word and check whether it is adjacent to previous character.

For example, given word = "SEE" and a board =

```
[
  ["ABCE"],
  ["SFCS"],
  ["ADEE"]
]
```

Convert the board to node sets <'S', [(1, 0), (1, 3)]>, <'E', [(0, 3), (2, 2), (2, 3)]>.
For 'S' at (1, 0), there is no adjacent 'E';
For 'S' at (1, 3), the 2nd 'E' can be (0, 3), but then there is no adjacent 'E' next to the 2nd 'E'; the 2nd 'E' can also be (2, 3) and the 3rd 'E' will be (2, 2).

Notice that we still perform DFS-like actions on those nodes but the size of node set could be smaller than the original matrix.

If this word query is a one-time query, this method won't beat the above one, especially considering the complication of implementation. But if this word query are going to be run for quite a few times, the preprocess can save us a lot of time.

```java
public boolean exist(char[][] board, String word) {
  HashMap<Character, ArrayList<Integer>> allNodes = preprocess(board);
  if (!allNodes.containsKey(word.charAt(0))) return false;

  boolean[] visited = new boolean[board.length*board[0].length];;
  for (int node : allNodes.get(word.charAt(0))) {
    if (expand(allNodes, node, visited, word, 1, board[0].length)) return true;
  }

  return false;
}

private boolean expand(HashMap<Character, ArrayList<Integer>> allNodes, int preNode,
    boolean[] visited, String word, int cur, int m) {
  if (cur == word.length()) return true;
  if (!allNodes.containsKey(word.charAt(cur))) return false;

  visited[preNode] = true;
  for (int node : allNodes.get(word.charAt(cur))) {
    if (!visited[node] && isAdjacent(preNode, node, m) && expand(allNodes, node, visited, word, cur+1,
  }
  visited[preNode] = false;

  return false;
}

private HashMap<Character, ArrayList<Integer>> preprocess(char[][] board) {
  HashMap<Character, ArrayList<Integer>> allNodes = new HashMap<Character, ArrayList<Integer>>();

  for (int i=0; i<board.length; ++i) {
    for (int j=0; j<board[0].length; ++j) {
      ArrayList<Integer> nodes;
      if (!allNodes.containsKey(board[i][j])) {
        nodes = new ArrayList<Integer>();
        allNodes.put(board[i][j], nodes);
      } else {
        nodes = allNodes.get(board[i][j]);
      }
      nodes.add(i*board[0].length+j);
    }
  }

  return allNodes;
}

private boolean isAdjacent(int n1, int n2, int m) {
  int min = Math.min(n1, n2), max = Math.max(n1, n2);
  return ((max-min) == m || (max-min == 1 && (max % m) != 0));
}
```

Posted 12th July 2013 by Sophie

Labels: DFS, Graph, Hash, Java, Matrix, Recursion

0 Add a comment

11th July 2013

Maximum Sum of Non-contiguous Subsequences

Maximum Sum of Non-contiguous Subsequences [http://www.geeksforgeeks.org/maximum-sum-such-that-no-two-elements-are-adjacent/]

Given an array of integers, find the maximum sum of a subsequence with the constraint that no 2 numbers in the sequence should be adjacent in the array.

For example, given [3 2 7 10], return 13 (sum of 3 and 10);
given [3 2 -5 10 7], return 13 (sum of 3 and 7).

If all numbers are negative, return 0.

Solution

The basic idea is to calculate sum of all possible subsequences such that no 2 elements in a sequence would be adjacent in the original array and then keep tracking the maximum sum so far.

Now the problem is how to efficiently find all possible subsequences.

A straightforward way is to loop through each element of the array, recursively find next candidates. For instance, given [3 2 7 10], if we select 3 first, the next candidate can be either 7 or 10, i.e. any number in subarray [7 10]. When hits the end of array, compare the sum with current maxSum and store the greater one.
This algorithm runs in exponential time and there are a lot of space to improve since we revisit each element many times, much more than necessary.
One way to improve is to use backtracking: Store the known maxSum for subarrays we have visited, [0 .. k]. More specifically,

> maxSum[k] = maximum sum of subsequences of array [0 .. k] that include array[k].

Now the algorithm becomes:

- For each element in the array, find the maximum sum of array[k]+maxSum[j] where 0 <= j < k-1.
  Notice that maxSum[k-1] includes array[k-1] and thus it cannot include array[k].
- If all numbers are positive, after we calculate maxSum's, we can simply check the last two values; but numbers are possibly negative, which means the max-sum-sequence may not include several numbers (not just the last one), so we need to loop through all maxSum's to find out the overall maxSum.

```
public int maxSumInSubsequence(int[] data) {
  if (data == null) return 0;
  int n = data.length;

  // maxSum[i] == the maximum sum of subsequences of data[0 .. i] that include data[i]
  int[] maxSum = new int[n];
  for (int i=0; i<n; ++i) {
    maxSum[i] = data[i];
    // maxSum[i-1] includes data[i-1] and thus cannot include data[i]
    for (int j=0; j<i-1; ++j) {
      maxSum[i] = Math.max(data[i] + maxSum[j], maxSum[i]);
    }
  }

  // find the max of all subsequences
  int max = 0;
  for (int i=0; i<n; ++i) {
    max = Math.max(max, maxSum[i]);
  }

  return max;
}
```

This algorithm takes O(n^2) time and O(n) spaces.

In algorithm above, we store maximum sums that including array[k] in maxSum[k] and we need to go back to check previous maxSum's which excludes array[k].

If we keep both includeSum and excludeSum, then we have

> includeSum[k] = excludeSum[k-1] + data[i]
> excludeSum[k] = Max(includeSum[k-1], excludeSum[k-1])

That said, we don't need to keep an array of previous maximum sums. All we need are just previous includeSum and excludeSum.

Here is code for this algorithm:

```
public int maxSumInSubsequence(int[] data) {
  if (data == null) return 0;
  int n = data.length;

  int incl = data[0], excl = 0;
  for (int i=1; i<n; ++i) {
    // current max excluding data[i]
    int exclNew = Math.max(incl, excl);
    // current max including data[i]
    incl = excl + data[i];
    excl = exclNew;
  }

  return Math.max(incl, excl);
}
```

This algorithm runs in time O(n) with O(1) space!

Posted 11th July 2013 by Sophie

Labels: Array, Backtracking, DP, Java, Recursion

4 View comments

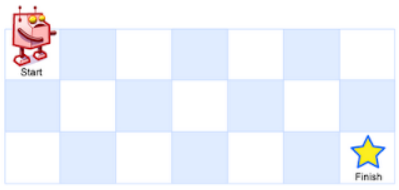
10th July 2013

Unique Paths

Unique Paths [http://leetcode.com/onlinejudge#question_62]

A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).



[http://4.bp.blogspot.com/-Vn8USHnYLuw/Udzw6TEqeLI/AAAAAAAAEig/0xSVzHdb7wc/s1600/robot_maze.png]

How many possible unique paths are there?

Solution - DP

This is naturally a DP problem.

Let PathNum(i, j) represent the number of unique path from (0, 0) to (i, j). As shown in the graph below, we have

```
PathNum(i, j) = PathNum(i-1, j) + PathNum(i, j-1)
```

To get PathNum(m-1, n-1), we can build up such a table row by row. That gives us a solution with O(m*n) running time and O(m*n) spaces. The running time is optimal since we have to visit each cell at least once. But the space usage can be optimized.

[http://3.bp.blogspot.com/-4c8K2-jRyaY/Udzw6R1CgQI/AAAAAAAAEic/PoE7_A88sh8/s1600/Screen+Shot+2013-07-09+at+10.27.02+PM.png]

Notice that we actually only use information from previous row. That said, we can maintain one row and for each cell, add previous cell to itself since itself is the value of cell above it. For the first row, the value of cell above it is 0.
By doing this, the space complexity can be reduced to O(min(m, n))!

```
public int uniquePaths(int m, int n) {
  if (m == 0 || n == 0) return 0;
  int x = Math.max(m, n), y = Math.min(m, n);
  int[] row = new int[y];

  row[0] = 1;

  // fill up the table
  for (int i=0; i<x; ++i) {
    for (int j=1; j<y; ++j) {
      row[j] += row[j-1];
    }
  }

  return row[y-1];
}
```

Solution - Maths

This problem can also be solved using Combinations.

Think about it in this way:
There are in total m+n-2 steps from Start to End and n-1 (or m-1) "turning points" where we move from one column (or row) to the next one. Note that the "turning point" is not necessary to be a turn-left or turn-right point. For instance, moving from (i, j) to (i, j+1) can also be counted as a turning point since it moves to next column.

Now the question becomes how many ways we can choose n-1 (or m-1) turning points from the total steps, i.e. C(m+n-2, n-1). We only need to choose either row- or column-turning points since once those points are selected, the path has been determined. Obviously, the combination can be computed in O(min(m, n)) time with O(1) space.

```
public int uniquePaths(int m, int n) {
  if (m == 0 || n == 0) return 0;

  int x = Math.min(m, n), y = Math.max(m, n);
  double count = 1;
  for (int i=1; i<x; ++i) {
    count *= (y + i - 1);
    count /= i;
  }

  return (int)count;
}
```

Unique Paths with Obstacles [http://leetcode.com/onlinejudge#question_63]

Follow up question:

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as 1 and 0 respectively in the grid. For example, there is one obstacle in the middle of a 3x3 grid as illustrated below.

```
[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
```

The total number of unique paths is 2 in this case.

Solution - DP

With a small modification on the above DP solution, we can solve this problem easily:
Whenever hits an obstacle, set the current cell to 0, meaning that there is no way to get from (0, 0) to (i, j) via (k, l) if (k, l) has an obstacle.

A small optimization is that if the entire row are all 0's, we can safely return 0 since we cannot arrive at end without passing through any cell in a row.

```
public int uniquePathsWithObstacles(int[][] obstacleGrid) {
  int m = obstacleGrid.length;
  if (m == 0) return 0;
  int n = obstacleGrid[0].length;
  if (n == 0) return 0;

  int[] row = new int[n];
  row[0] = 1;
  for (int i=0; i<m; ++i) {
    int total = 0;
    for (int j=0; j<n; ++j) {
      if (obstacleGrid[i][j] == 1) {
        row[j] = 0;
      } else if (j>0) {
        row[j] += row[j-1];
      }
      total += row[j];
    }
    if (total == 0) return 0;
  }

  return row[n-1];
}
```

Posted 10th July 2013 by Sophie

Labels: DP, Java, Maths, Matrix

4 View comments

8th July 2013

System Design for Big Data [Count Occurrences]

Typical Problem Statement

Given a large number (millons or billons) of records (integers, IPs, URLs, query key words, etc.),

- find out top k most frequent records;
- find out duplicates;
- find out whether a given number x is in the set or not.

One common requirement here is how to handle big data efficiently when they are not able to be all loaded into memory.

Typical Strategies

If all data can be loaded into memory, we can

1. Count occurrences using a Hash table or Bitmap (or Bloom Filter) -- O(n)
2. Sort data by their occurrences -- O(nlogn)
3. Return the result

But when it comes to big data, we need to do multi-pass to (pre-)process data and optimize the solution to reduce disk I/O, network traffic, and (of course) running time.

Note: You may want to ask interviewer whether this is an one-time or multi-time call. If this is an one-time call, the preprocess won't save you any time and may introduce a little more complexity by storing processed data. But in most cases, it will be a multi-time call.

The whole process will be something like:

1. For each record, generate a hash key, divide up keys and distribute them to machine `#[hash(record) % N]`. -- O(n)
    - This one-time preprocess can dramatically improve the performance for following calls.
    - Hashing not only provide O(1) access time but also aggregate data with same key onto same machine.
    - One disadvantage of using `mod` is that some machines may get saturated faster than others. If that is the case, it may need to further spread data on that machine into several newly added machines so as to build up a tree-like structure, which costs expensive data migration. Alternatively, we can redesign a better hashing function to replace `mod` such that data could be distributed more evenly.
    - In some cases, we may want "similar data" to go to same machine, and replace `mod` with a filter function can help us to achieve that goal. E.g. Given person records, filter them by locations.

- If the input are documents, consider to use Inverted Index [http://en.wikipedia.org/wiki/Inverted_index] to map words in documents to a list of document ids which contains that word. It is widely used in search engine indexing [http://en.wikipedia.org/wiki/Index_(search_engine)] algorithms.

2. On each machine, use HashMap [http://en.wikipedia.org/wiki/Hashmap] or Bitmap [http://en.wikipedia.org/wiki/Bitmap] to count occurrences. -- O(n/N) with parallel operations
   - Bitmap can be considered as a compact hash map storing true/false values for each key.
   - Bloom Filter [http://en.wikipedia.org/wiki/Bloom_filter] is a special case of Bitmap and can provide strong space advantage over other data structures (with small chances of false positive though).
   - Trie [http://en.wikipedia.org/wiki/Trie] is another space efficient data structure that provides O(m) access time where m is the length of query string (which would be much smaller than the total number of queries).
3. On each machine, use HeapSort [http://en.wikipedia.org/wiki/Heapsort] to retrieve the top k results and output to a small file. -- O(nlogk)
   - If we want the top k most popular records, build up a min-heap of size k with the first k records, scan the remaining records, replace the heap top if the current record is greater than the heap top and then adjust the heap.
   - If we want to the top k smalles value, use a max-heap.
4. Apply MergeSort [http://en.wikipedia.org/wiki/Merge_sort] or other External Sorting [http://en.wikipedia.org/wiki/External_sorting] algorithms over sorted data on each machine to get the final results. -- O(kNlog(kN))

This process sometimes can be referred to MapReduce [http://en.wikipedia.org/wiki/MapReduce] process where Map conducts filtering and sorting and Reduce performs a summary operation.

Exercise Problems

Find Missing Integer (<Cracking Coding Interview [http://www.amazon.com/gp/product/098478280X/ref=as_li_ss_il?ie=UTF8&camp=1789&creative=390957&creativeASIN=098478280X&linkCode=as2&tag=n00tc0d3r-20] > 10.3)

Given a file with 4 billion distinct integers, find out an integer that is not contained in the file.

1. Assume you have 1 GB of memory.
2. Assume you have 10 MB of memory.

Solution

There are 2^32 distinct integers and the file contains 4 billion ~= 2^32 integers.

Given 1 GB memory, i.e. 2^30 bytes = 2^33 bits.
We can map all integers to a Bitmap so that all integers can be loaded into memory. So, we can solve the problem in one-pass.

```
public int findMissingNumber(int[] data) {
  long n = (long)Integer.MAX_VALUE >> 1;
  byte[] bitmap = new byte[(int)(n >> 3)];

  // scan all data
  for (int i=0; i<data.length; ++i) {
    bitmap[data[i]>>3] |= (1 << (data[i] & 7));
  }

  // find missing one
  for (int i=0; i<bitmap.length; ++i) {
    for (int j=0; j<8; ++j) {
      if ((bitmap[i] & (1 << j)) == 0) return (i*8 + j);
    }
  }

  // not found
  return -1;
}
```

Given 10 MB memory, to find a missing integer requires two pass of data.

1. Divide the data into N blocks and count the number of integers in each block. If the block contains the missing integer, the count will be smaller than the size of the block.
   Minimum memory usage is
   N * 4 < 10 * 2^20 (in bytes)
   where N is the number of blocks.
2. For the block that contains the missing integer, apply bitmap over integers in the block to find the missing one.
   Minimum memory usage is
   2^32 / N = B < 10 * 2^20 * 8 (in bits)
   where B is the size of a block.

So, the block size can be in range [2^11, 2^26]. Selecting a middle value such that both passes require less memory, i.e. N*4*8 ~= B, B = 2^18.

```
public int findMissingNumber(int[] data) {
  int blockSize = 1 << 18; // block size 2^18
  int blockNum = 1 << 14; // N = 2^32 / 2^18

  int start = countNumInBlock(data, new int[blockNum], blockSize);
  return findMissingNumInBlock(data, start);
}

/* Count the number of integers in each block and return the one with missing integer. */
private int countNumInBlock(int[] data, int[] counts, int blockSize) {
  for (int i=0; i<data.length; ++i) {
    ++counts[data[i] >> 18];
  }
  for (int i=0; i<counts.length; ++i) {
    if (counts[i] < blockSize) return blockSize * i;
  }

  return -1;
}

/* Find a missing integer from given block [start, start+blockSize). */
private int findMissingNumInBlock(int[] data, int start, int blockSize) {
  byte[] bitmap = new byte[(int)(blockSize >> 3)];

  // scan all data
  int end = start + blockSize;
  for (int i=0; i<data.length; ++i) {
    if (data[i] >= start && data[i] < end)
      bitmap[data[i]>>3] |= (1 << (data[i] & 7));
  }

  // find missing one
  for (int i=0; i<bitmap.length; ++i) {
    for (int j=0; j<8; ++j) {
      if ((bitmap[i] & (1 << j)) == 0) return (i*8 + j);
    }
  }

  // not found
  return -1;
}
```

Find Top k **Most Frequent IP** (<How to Solve Big Data Interview Questions [http://blog.csdn.net/v_july_v/article/details/7382693#t4] >, in Chinese)

Given a large access log of a website that contains 10 billion IPs that access the website in a day, find out k IPs that access the website most frequently.

Solution

Each IP is 32 bits and thus there are at most 2^32 distinct IPs. IPs are stored as Strings "xxx.xxx.xxx.xxx", so each IP uses 15 bytes.

If there is no memory limit, we can build up a hash table that maps each IP to the count of its occurrences, and then iterate through all keys in the hash table and find the top k IPs with largest count (using HeapSort with min-heap).
Such a hash table requires (15+4) bytes for each record and in total takes 19*2^32 bytes ~= 80 GB spaces.

Suppose memory limit is 100 MB.

We solve the problem with two passes:

- Divide up the data set into N blocks.
  Notice that the data set is not sorted, so when we split data, we cannot simple put every n records into a small file. We need to hashing the IP to its corresponding file so that the same IP will always go to the same file.
  100 MB / 19 bytes ~= 5.3 millons. So, let each block have 4 millon = 2^22 IPs.
  One way to hash the IPs is to parse the first two token "xxx.xxx" of an IP, convert it into an integer and use the high 10 bits as the corresponding file id.
- In each block, use a hash table to calculate frequencies and then perform in-place HeapSort to find the first k most frequent IP.
- MergeSort/HeapSort the N sorted arrays to find the overall k most frequent IP.

Find top k largest from a given array.

```
public void findTopKLargest(int[] data, int k) {
  // build min-heap, data[0] is heap top
  for (int i=k-1; i>0; --i) {
    int parent = (i-1)>>2;
    if (data[parent] > data[i]) swap(data, parent, i);
  }

  // pipe the remaining numbers to heap to find the top k largest
  for (int i=k; i<data.length; ++i) {
    if (data[i] > data[0]) { // insert to heap if greater than heap top
      swap(data, 0, i);
      adjustMinHeap(data, k);
    }
  }
}

private void adjustMinHeap(int[] data, int k) {
  int index = 0;
  while (index < ((k-1) >> 1)) {
    int left = (index << 1) + 1, right = left + 1;
    int min = (data[left] > data[right]) ? right : left;
    if (data[index] <= data[min]) break;
    swap(data, index, min);
    index = min;
  }
}
```

```
private void swap(int[] data, int a, int b) {
  int temp = data[a];
  data[a] = data[b];
  data[b] = temp;
}
```

Posted 8th July 2013 by Sophie

Labels: BigData, Design, Hash, Heap, MapReduce, Sort

13 View comments

7th July 2013 Unique Binary Search Trees

Unique Binary Search Trees [http://leetcode.com/onlinejudge#question_96]

Given n, generate all structurally unique BST's (binary search trees) that store values 1...n.

For example,
Given n = 3, there are a total of 5 unique BST's.

```
   1         3     3      2      1
    \       /     /      / \      \
     3     2     1      1   3      2
    /     /       \                 \
   2     1         2                 3
```

Solution - Recursion

Given a number n, how many unique BST's can be constructed?

Think it in a deductive way:

- n=1, one unique BST.
- n=2, pick one value, and the one remaining node can be built as one BST, i.e. Two BSTs.
- ... ...
- n=k, pick one value i, and split the remaining values to two groups: [1 .. i-1] goes to left subtree and [i+1 .. n] goes to right subtree. Then the problem becomes to construct BSTs with left subtree from BST(1, i-1) and right subtree from BST(i+1, n), where BST(1, i-1) and BST(i+1, n) has been computed in previous iterations.

It is easy to get to a recursive solution:

- Given a range [l .. r], if l > r, return a list containing an empty tree.
- Otherwise, for each value k between l and r, inclusively
  - recursively compute BSTs in range [l .. k-1] and range [k+1 .. r]
  - construct BSTs with root of k, left subtree from BST(1, i-1) and right subtree from BST(i+1, n).

```
private ArrayList<TreeNode> genSubTrees(int l, int r) {
  ArrayList<TreeNode> trees = new ArrayList<TreeNode>();
  if (l > r) { // return an empty tree
    trees.add(null);
  } else {
    for (int i=l; i<=r; ++i) {
      ArrayList<TreeNode> lefts = genSubTrees(l, i-1);
      ArrayList<TreeNode> rights = genSubTrees(i+1, r);
      for (TreeNode left : lefts) {
        for (TreeNode right : rights) {
          TreeNode root = new TreeNode(i);
          root.left = left;
          root.right = right;
          trees.add(root);
        }
      }
    }
  }
  return trees;
}

public ArrayList<TreeNode> generateTrees(int n) {
  return genSubTrees(1, n);
}
```

This algorithm runs in exponential time (much greater then the actual total count of unique BSTs) since we recompute BSTs in subrange [i .. j] repeatedly.

Solution - DP

Noticing the repeat computation in the above algorithm gives us a hint of DP: Can we store some intermediate results so as to reuse it rather than recomputing it?

The answer is yes and we can store BSTs of range [i .. i+k] and then for next range [i .. i+k+1] we can reuse previous results!

Now the algorithm becomes:

- Create a table T containing lists of BSTs such that T[i, l] = a list of BSTs of range [i .. i+l].
- Initialize the table with T[i, 0] which are all single node trees.

- Increasing the size of range from 1 to n-1 and fill up the table.
  For each range size l,
  - For each starting value i,
    - For each value k in [i, i+l], construct BSTs with root of k, left subtree from BSTs of [i, k-i-1] and right subtree from BSTs of [k+1, i+l-k-1].
- Finally, T[0, n-1] gives us the result of all BSTs.

```
public ArrayList<TreeNode> generateTrees(int n) {
  if (n < 1) {
    ArrayList<TreeNode> trees = new ArrayList<TreeNode>();
    trees.add(null);
    return trees;
  }

  // T[i,l] contains a list of BSTs of [i .. i+l]
  ArrayList<ArrayList<ArrayList<TreeNode>>> allNumTrees = new ArrayList<ArrayList<ArrayList<TreeNode>>

  // init with single node trees
  for (int i=1; i<=n; ++i) {
    ArrayList<ArrayList<TreeNode>> numTrees = new ArrayList<ArrayList<TreeNode>>(n-i);
    ArrayList<TreeNode> trees = new ArrayList<TreeNode>();
    TreeNode root = new TreeNode(i);
    trees.add(root);
    numTrees.add(trees);
    allNumTrees.add(numTrees);
  }

  // fill up the table
  for (int l=1; l<n; ++l) { // levels
    for (int i=0; i<n-l; ++i) { // starting number
      ArrayList<ArrayList<TreeNode>> numTrees = allNumTrees.get(i);
      ArrayList<TreeNode> trees = new ArrayList<TreeNode>();
      for (int k=i; k<=i+l; ++k) { // root value
        if (k == i) {
          for (TreeNode right : allNumTrees.get(k+1).get(l-1)) {
            TreeNode root = new TreeNode(k+1);
            root.right = right;
            trees.add(root);
          }
        } else if (k == i+l) {
          for (TreeNode left : allNumTrees.get(i).get(l-1)) {
            TreeNode root = new TreeNode(k+1);
            root.left = left;
            trees.add(root);
          }
        } else {
          for (TreeNode left : allNumTrees.get(i).get(k-i-1)) {
            for (TreeNode right : allNumTrees.get(k+1).get(i+l-k-1)) {
              TreeNode root = new TreeNode(k+1);
              root.left = left;
              root.right = right;
              trees.add(root);
            }
          }
        }
      }
      numTrees.add(trees);
    }
  }

  return allNumTrees.get(0).get(n-1);
}
```

This algorithm sacrifice space for time. Since each subtree is only computed and stored once, the time and space complexities are equal to the total number of unique BSTs, which is the Catalan number as proven in previous post [http://n00tc0d3r.blogspot.com/2013/07/unique-binary-search-trees.html] .

There are other ways to build such a table. For example, build up a table of T such that T[i, j] = a list of BSTs of range [i .. j]. In that case, for each T[i, j] where i > j we need to put an empty tree there.

Posted 7th July 2013 by Sophie

Labels: BinarySearchTree, DP, Java, Recursion, Tree

0 Add a comment

5th July 2013

Count Inversions

Count Inversions [http://www.geeksforgeeks.org/counting-inversions/]

Given an integer array, count the number of inversions.

Inversion Count for an array indicates – how far (or close) the array is from being sorted. If array is already sorted then inversion count is 0. If array is sorted in reverse order that inversion count is the maximum.
Formally speaking, two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j

For example, given [22, 48, 12, 35, 57], return 3 since there are three inversions, (22, 12), (48, 12), (48, 35).

Solution

A Brute Force [http://en.wikipedia.org/wiki/Brute_force] solution is:

- For each element in the array, count the number of remaining elements that are smaller than it.

This gives us a O(n^2) solution.

```
public long countInversionBF(int[] A) {
  long count = 0;

  for (int i=0; i<A.length-1; ++i) {
    for (int j=i+1; j<A.length; ++j) {
      if (A[i] > A[j]) ++count;
    }
  }

  return count;
}
```

Can we do better?

What if we sort the array first? We know several sorting algorithms takes time O(nlogn) even in worst case, for example, Merge Sort [https://en.wikipedia.org/wiki/Merge_sort] .
Then in the given example, the array becomes [12, 22, 35, 48, 57]. Notice that from [22, 48, 12, 35, 57]

- 22 moved from [0] to [1], i.e. forward 1 step
- 48 moved from [1] to [3], i.e. forward 2 steps
- 12 moved from [2] to [0], i.e. backward 2 steps
- 35 moved from [3] to [2], i.e. backward 1 step

It looks like if we summarize either the total number of forward steps or that of backward steps, we get the count of inversions.

Think about it. Does this works for all cases? If so, why?
If an element x has an inversion, what does it tells us? That said, among k elements that are behind x, (k-1) of them are no less than x and 1 of them are less than x. Thus, in the sorted array, x need to move 1 step to get the subarray starting from x to the end be sorted. That's essentially the sorting process.

Therefore, we can modify the merge sort algorithm a little bit so as to solve this problem:

- Divide the array into two subarrays;
- Sort each subarray;
- Merge the two sorted subarrays and count the total number of steps for elements that move forwards.

```
/* merge two sorted subarrays A[l..mid) and A[mid..r) into one sorted array
  and count inversions during the process. B is a temporary array for merging. */
private long mergeAndCount(int[] A, int l, int mid, int r, int[] B) {
  long count = 0;

  // copy from A to B
  System.arraycopy(A, l, B, l, r-l);
  // merge
  for (int i=l, h1=l, h2=mid; i<r; ++i) {
    if (h1 >= mid || (h2 < r && B[h2] < B[h1])) {
      count += (h2 - i);
      A[i] = B[h2++];
    } else {
      A[i] = B[h1++];
    }
  }

  return count;
}

/* count inversions in A[l..r) */
public long countInversion(int[] A, int l, int r, int[] B) {
  if (l >= r || l < 0 || r > A.length) return 0;

  long count = 0;
  int mid = l + (r - l) / 2;

  // sort subarrays if needed
  if (mid > l+1) count += countInversion(A, l, mid, B);
  if (mid < r-1) count += countInversion(A, mid, r, B);

  // merge and count inversions
  count += mergeAndCount(A, l, mid, r, B);

  return count;
}
```

The running time of merge sort is O(nlogn) (in all cases, best/average/worst). And thus this algorithm runs in time O(nlogn) and uses O(n) spaces.

Notice that we pass in a temporary array so that we could reuse it, rather than creating a new one in each recursion. Allocating memory is always expensive and we definitely need to avoid doing that repeatedly.

Posted 5th July 2013 by Sophie

Labels: Array, Java, Recursion, Sort

0 Add a comment

3rd July 2013

Count Unique Binary Search Trees

Count Unique Binary Search Trees [http://leetcode.com/onlinejudge#question_96]

Given n, how many structurally unique BST's (binary search trees) that store values 1...n?

For example,
Given n = 3, there are a total of 5 unique BST's.

```
   1         3     3      2      1
    \       /     /      / \      \
     3     2     1      1   3      2
    /     /       \                 \
   2     1         2                 3
```

Solution

Given a number n, how many unique BST's can be constructed?

Think it in a deductive way:

- n=1, count(1) = 1 unique BST.
- n=2, pick one value, and the one remaining node can be built as one BST.
  i.e. count(2) = 2 * count(2-1).
- n=3, pick one value i, and split the remaining values to two groups: [1 .. i-1] goes to left subtree and [i+1 .. n] goes to right subtree.
  i.e. count(3) = count(3-1) + count(1)*count(1) + count(3-1)
- ... ...
- n=k, suppose count(0) = 1, we then have:

  count(k) = sum_{i=0}^(k-1) (count(i)*count(k-1-i))

This formula is known as Segner's recurrence relation [http://mathworld.wolfram.com/SegnersRecurrenceFormula.html] .
To implement it, it is nature to use DP: store previous count values in an array and compute sum on them.

```
public int numTrees(int n) {
  if (n == 0) return 0;
  int[] count = new int[n+1];
  count[0] = 1; count[1] = 1;

  for (int i=2; i<=n; ++i) {
    for (int j=0; j<i; ++j) {
      count[i] += (count[j] * count[i-1-j]);
    }
  }

  return count[n];
}
```

This algorithm runs in time O(n^2) with O(n) spaces.

This problem can be reduced to finding the total number of structurally unique binary tree with n nodes, which can then be reduced to Euler's Polygon Division Problem [http://mathworld.wolfram.com/EulersPolygonDivisionProblem.html] . The result of the above recurrence formula gives the Catalan number [http://en.wikipedia.org/wiki/Catalan_number] such that

count(n) = Combination(2n, n) - Combination(2n, n+1) = (2n)! / ((n+1)!n!)

Note: If you are interested in the proof of Catalan number, wiki [http://en.wikipedia.org/wiki/Catalan_number] provides several neat proofs. :D

```
public int numTrees(int n) {
  if (n == 0) return 0;
  int count = 1;

  for (int i=2; i<=n; ++i) {
    count *= ((2*i - 1) / (i+1));
  }

  return count;
}
```

This algorithm runs in time O(n) with O(1) spaces.

Posted 3rd July 2013 by Sophie

Labels: BinarySearchTree, DP, Java, Maths

0 Add a comment

1st July 2013 Summary: Distributed System Design

Requirements

First of first, you need to be clear about the requirements of the system. This is the basis of the system design and also determines how you could optimize the design (e.g. which can be compromised and which are not).

You can start to ask questions like:
Is it computation-consumed or I/O-consumed?
Is it a data storage system or a request-response system?
What are the typical use cases or workflows? Any special use cases we need to take care of?
What are its typical input and output?
What are its major components?
...

Draw diagrams for typical use cases or workflows and reduce to UML diagrams of the major components and interactions between them.

Add/Remove/Modify components functionalities to optimize the overall workflow. Keep in mind the following principles during design and optimization.

Principles

- **Scalability**: Size!
  - Easy to scale up/down
  - Traffic/load that can be handled
- **Reliability**: Data consistency
  - (Realtime?) data consistency
  - Persistence
- **Performance**: Speed of a website
  - Fast response
  - Low latency
- Availability: Uptime of a website
  - Redundancy for key components
  - Failure detection and recovery
- Manageability: Operational load
  - Easy to update
  - Simple to operate
- Cost
  - Hardware/Software
  - Develop time
  - Operational effort

Techniques

- Caches can cache popular responses from server so as to reduce the number of calls (Local/Distributed Cache such as memcached [http://memcached.org/])
- Proxies can be used to aggregate client requests to server so as to avoid unnecessary reconnections
- Hashing and Revert indexes provides constant time to retrieve data
- Load Balancers can reduce the chances of servers are under load (some are hot while others are idle)
- Queues and Asynchrony calls so that clients don't need to spin there waiting for response

-----
Ref: Scalable Web Architecture and Distributed Systems [http://www.aosabook.org/en/distsys.html]

Posted 1st July 2013 by Sophie

Labels: Design, Summary

1 View comments

30th June 2013 Valid Number

Valid Number [http://leetcode.com/onlinejudge#question_65]

Validate if a given string is numeric.

Solution

The problem description is quite vogue. We need to clarify it first.

Is it possible to have leading/trailing spaces?
Possibly. Leading/trailing spaces can be ignored. That said, "   90   " is okay but "9 0" is not.
Is sign (+/-) accepted?
Yes. "+10", "-3", "10" are all valid.
Is decimals accepted?
Yes. "2.413", ".413", and "143." are all valid.
Is hexadecimal accepted?
No. "0x34AF" can be treated as invalid.
Is it possible to use "E notation"?

Yes. But you can assume power is integer, i.e. "89e9" and "1.23e4" is an valid input but "52e4.2" is not. Also, both power and exponent may have sign and , e.g. "+23e-3".

So, an valid input could be something like

(space)*[+-]?([0-9]+|[0-9]*\.[0-9]+|[0-9]+\.[0-9]*)([eE][+-]?[0-9]+)?(space)*$

where '*' means 0 or more occurrences, '+' means 1 or more occurrence, and '?' means 0 or 1.

A simple solution is to use regex [http://en.wikipedia.org/wiki/Regular_expression] (tutorial [http://www.vogella.com/articles/JavaRegularExpressions/article.html]) to match the given string.

```
public boolean isNumber(String s) {
  return s.matches("^\\s*[+-]?(\\d+|\\d*\\.\\d+|\\d+\\.\\d*)([eE][+-]?\\d+)?\\s*$");
}
```

The running time depends on how complier parses regular expression.

If we are required to actually parse the string, we can reduce the regular expression to a Finite State Machine [http://en.wikipedia.org/wiki/State_machine] and compare whether the given string is equivalent to the pattern.

[http://3.bp.blogspot.com/-Da4X9dKYEDE/Uc_jy0xcg8I/AAAAAAAAEh4/x-sfkvDrmsg/s760/Screen+Shot+2013-06-30+at+12.52.07+AM.png]

As shown in the picture, we have 9 states and several input types. The state machine can then be translated into the following table.

[http://1.bp.blogspot.com/-lXU9XWOG0NM/Uc_ctDIS1iI/AAAAAAAAEhs/wXyNKcMQsOk/s345/Screen+Shot+2013-06-30+at+12.21.49+AM.png]

Now the algorithm is clear:

- Create a table for the state transitions and let the initial state be 0;
- For each character in the input string, update the state based on the type of the character and the current state, return false if input or the state transition is invalid;
- Return true if the end state is one of the final states, otherwise, return false.

```
private enum Type {
  Space(0), Sign(1), Digit(2), Dot(3), Exp(4), Null(-1);
  private int type;
  private Type(int t) { type = t; }
  public int getType() { return type; }
}

public boolean isNumber(String s) {
  int[][] states = {
    {0, 8, -1, -1, 8, -1, -1, 8, 8},
    {2, -1, -1, -1, -1, 6, -1, -1, -1},
    {1, 1, 1, 4, 4, 7, 7, 7, -1},
    {3, 4, 3, -1, -1, -1, -1, -1, -1},
    {-1, 5, -1, -1, 5, -1, -1, -1, -1}
  };

  int state = 0;
  for (Character c : s.toCharArray()) {
    Type inputType = Type.Null;
    if (c == ' ') {
      inputType = Type.Space;
    } else if (c == '+' || c == '-') {
      inputType = Type.Sign;
    } else if (c >= '0' && c <= '9') {
      inputType = Type.Digit;
    } else if (c == '.') {
      inputType = Type.Dot;
    } else if (c == 'e' || c == 'E') {
      inputType = Type.Exp;
    } else {
      return false; // invalid input
    }

    state = states[inputType.getType()][state];
    if (state < 0) { // invalid state transition
      return false;
    }
  }

  return (state == 1 || state == 4 || state == 7 || state == 8);
}
```

We visit each character once and thus this algorithm runs in time O(n). It only takes O(5*9)=O(1) space.

Posted 30th June 2013 by Sophie

Labels: DP, Java, Maths, Regex, String

0 Add a comment

24th June 2013

Triangle Path Sum

Triangle [http://leetcode.com/onlinejudge#question_120]

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

For example, given the following triangle

```
[
     [2],
    [3,4],
   [6,5,7],
  [4,1,8,3]
]
```

The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).

Solution - DFS, Recursion

For each numer at (i, j), the adjacent numbers on the next row below are (i+1, j) and (i+1, j+1).
A recursive solution is to use DFS to find out the minimum sum among all possible sums from top to bottom. Since numbers could be negative, we cannot prune sub-triangle when the current sum is no less than current minimum sum.

```
private int DFS(ArrayList<ArrayList<Integer>> triangle, int row, int column, int sum, int minSum) {
  // add itself
  sum += triangle.get(row).get(column);

  if (row == triangle.size() - 1) { // last row
    if (sum < minSum) return sum;
  } else {
    minSum = DFS(triangle, row+1, column, sum, minSum);
    minSum = DFS(triangle, row+1, column+1, sum, minSum);
  }

  return minSum;
}

public int minimumTotal(ArrayList<ArrayList<Integer>> triangle) {
  return DFS(triangle, 0, 0, 0, Integer.MAX_VALUE);
}
```

This algorithm actually expand the triangle into a full binary tree and check each node constant times. For example, the example triangle becomes

```
[
         [2],
     [3    ,    4],
   [6  , 5    5  , 7],
  [4,1 1,8 1,8 8,3]
]
```

Given a triangle of n rows, the resulted binary tree will have O(2^n) nodes. Thus, it takes O(2^n) time, where n is the total number of rows, and O(n) spaces for DFS.

Solution - DFS, DP

Notice that in the above solution, we revisit each number multiple times which may not be necessary. For example, the sums in the small triangle

```
[
 [5]
[1,8]
]
```

has been recalculated along the path via number 3 and 4.

That hints us to use DP to optimize the algorithm.
Instead of recalculate all the time, we store the known minimun sum from the current node to bottom in a table. Then next time when we hit the same number, it only takes O(1) time to loop up the value.

For the given example, we end up with a min-sum-table as follows.

```
[
     [11],
    [9,10],
   [7,6,10],
  [4,1,8,3 ]
]
```

And the algorithm goes like:

1. For each numer at (i, j) where i is not the last row, check its two adjacent numbers on the next row to find the min of two sums from that adjacent number to bottom.
   1. If the min-sum for (i+1, j) is already in the table, use it; Otherwise, use DFS to calculate it recursively and add it to the table.
   2. Use DFS to calculate the min-sum for (i+1, j+1) recursively and add it to the table.
   3. Set the current min-sum of (i, j) is number at (i, j) plus min(min-sum(i+1, j), min-sum(i+1, j+1).
   4. Return min-sum(i, j).
2. For numbers on the last row, simply return the number itself.

This algorithm doesn't revisit numbers and thus the total running time gets down to O(n^2). This is an optimal solution since we have to visit all numbers at least once to get the minimum sum and there are O(n^2) numbers. But this algorithm takes O(n^2) spaces for the table.

Do we really need to keep all of them minimum sums?
Each time when we check a number on the next row, we actually only need the previously calculated min-sum of that row. That is, for number (i, j), we only need min-sum(i+1, j) which would have been calculated via path from (i, j-1) (if exists).

Revise the previous algorithm a little bit:

1. For each numer at (i, j) where i is not the last row, check its two adjacent numbers on the next row to find the min of two sums from that adjacent number to bottom.
   1. If the row-min-sum exists, i.e. min-sum(i+1, j) has been calculated, use it; Otherwise, use DFS to calculate it recursively and update row-min-sum for row i+1.
   2. Use DFS to calculate the min-sum(i+1, j+1) recursively and update row-min-sum for row i+1.
   3. Update row-min-sum for the current row.
   4. Return min-sum(i, j).
2. For numbers on the last row, simply return the number itself.

By doing this, we only need to store one value for each row. The space complexity goes down to O(n)!

```
private int DFS(ArrayList<ArrayList<Integer>> triangle, int row, int column,
    HashMap<Integer, Integer> rowMin) {
  int min = triangle.get(row).get(column);
  if (row == triangle.size() - 1) { // last row, return itself
    return min;
  }

  if (!rowMin.containsKey(row+1)) { // calculate for first column
    min += Math.min(DFS(triangle, row+1, column, rowMin),
            DFS(triangle, row+1, column+1, rowMin));
  } else {
    min += Math.min(rowMin.get(row+1),
            DFS(triangle, row+1, column+1, rowMin));
  }

  rowMin.put(row, min);

  return min;
}

public int minimumTotal(ArrayList<ArrayList<Integer>> triangle) {
  return DFS(triangle, 0, 0, new HashMap<Integer, Integer>());
}
```

Solution - BFS, DP

An alternative (and better) solution is to use BFS.

1. Traverse the triangle level by level.
2. For each node in a level, compute the minimal sum from path to itself using sums from previous level.

For each node on the boundary, we only visit them once; for node in the middle of the triangle, we visit them twice. Thus, the total running time of this algorithm is O(n^2) and it takes O(n) space to save sums in each level, where n is the total number of rows.

```
public int minimumTotal(ArrayList<ArrayList<Integer>> triangle) {
  int[] preRow = new int[triangle.size()];

  // BFS
  for (ArrayList<Integer> row : triangle) {
    int last = row.size() - 1;
    if (last > 0) preRow[last] = preRow[last-1] + row.get(last);
    int pre = preRow[0];
    preRow[0] += row.get(0);
    for (int i=1; i<last; ++i) {
      // save the previous value so that we can use it for next number
      int temp = preRow[i];
      // either from i-1 or i from the previous row
      preRow[i] = Math.min(preRow[i]+row.get(i), pre+row.get(i));
      pre = temp;
    }
  }

  // find the min
  int res = preRow[0];
  for (int num : preRow) {
    if (num < res) res = num;
  }

  return res;
}
```

Notice that in the above algorithm, we have to save preRow[i] so as to use it for next number. To avoid it, we can do the calculate backwards.

```
public int minimumTotal(ArrayList<ArrayList<Integer>> triangle) {
  int n = triangle.size();
  int[] sum = new int[n];

  // BFS
  for (ArrayList<Integer> row : triangle) {
    int m = row.size();
    for (int i=m-1; i>=0; --i) {
      if (i > 0) { // check edge cases
        sum[i] = (i==m-1) ? sum[i-1] : Math.min(sum[i-1], sum[i]);
      }
      sum[i] += row.get(i);
    }
  }

  // find the min
  int minSum = Integer.MAX_VALUE;
  for (int s : sum) {
    minSum = Math.min(minSum, s);
  }
  return minSum;
}
```

Posted 24th June 2013 by Sophie

Labels: Array, BFS, DFS, DP, Java, Recursion, Tree

1 View comments

24th June 2013

Text Justification

Text Justification [http://leetcode.com/onlinejudge#question_68]

Given an array of words and a length L, format the text such that each line has exactly L characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly L characters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left justified and no extra space is inserted between words.

For example, words: ["This", "is", "an", "example", "of", "text", "justification."], L = 16.
Return the formatted lines as:

```
[
   "This    is    an",
   "example  of text",
   "justification.  "
]
```

Note: Each word is guaranteed not to exceed L in length.

Solution

First, to add several words into a String of length L,

- If left-justify, add word one by one, each followed be a space. If the total length < L, stuff with spaces.
- Otherwise, since we need to add extra space evenly, calculate the number of evenly distributed spaces and also the leftover extra spaces which will be added to the first few slots. Then, add word one by one, each followed by spaces as calculated before.

Now, given a list of words, how to split it into several lines?
We can count the word length until it reaches L. In that case, add those words into one line and start another line. If some words left, put them into the last line using left-justify.

Some special cases need to be consider:

- For any line other than the last line, if it contains only one word, use left-justify;
- If the given word array is empty, return a string stuffed with L spaces.

```
// generate a string of n spaces
private String stuffSpaces(int n) {
  if (n==0) return "";
  StringBuilder ss = new StringBuilder();
  while (n > 0) {
    ss.append(' ');
    --n;
  }
  return ss.toString();
}

// generate a string of length L, containing words[start..end], inclusively
private String justify(String[] words, int start, int end, int total, int L) {
  StringBuilder ss = new StringBuilder();
  if (end == start || end == words.length-1) { // single word or last words
    while (start < end) {
      ss.append(words[start++]);
      ss.append(stuffSpaces(1));
    }
    ss.append(words[end]);
    ss.append(stuffSpaces(L - ss.length()));
  } else {
    int spaces = (L - total) / (end - start);
    int extras = (L - total) % (end - start);
    // buld up full string
    while (start < end) {
      ss.append(words[start++]);
      ss.append(stuffSpaces(spaces));
      if (extras > 0) {
        ss.append(stuffSpaces(1));
        --extras;
      }
    }
    ss.append(words[end]);
  }
  return ss.toString();
}

public ArrayList<String> fullJustify(String[] words, int L) {
  ArrayList<String> res = new ArrayList<String>();

  // count word length
  int len = 0, start = 0;;
  for (int i=0; i<words.length; ++i) {
    len += words[i].length();
    if ((len + i - start) > L) {
      res.add(justify(words, start, i-1, len-=words[i].length(), L));
      // reset
      len = words[i].length();
      start = i;
    }
  }
  // last line
  if (len > 0) {
    res.add(justify(words, start, words.length-1, len, L));
  }

  // if words is empty, stuff results with L spaces
  if (res.size() == 0) {
    res.add(stuffSpaces(L));
  }
  return res;
}
```

This algorithm runs in linear time since it touches each given word O(1) times (count the total length and append to a line).

Posted 24th June 2013 by Sophie

Labels: Array, Java, String, StringBuilder

24th June 2013

Trapping Rain Water

Trapping Rain Water [http://leetcode.com/onlinejudge#question_42]

Given n **non-negative** integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

For example,
Given [0,1,0,2,1,0,1,3,2,1,2,1], return 6.

[http://www.leetcode.com/wp-content/uploads/2012/08/rainwatertrap.png]
The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

Solution

What we can get from the given example?
First, we need to skip initial zeros if exist any since there is not left boundary to trap the water for those buckets.
Then, an intuitive idea is, starting from the first positive bar,

1. Mark it as left and move forward current to next bar.
2. If current < left, calculate the difference, store it in a temporary sum, and move forward to next bar;
3. Otherwise, add the temporary sum to the total volume, and continue to step 1.

But this will fail the case of [2, 1, 0, 1], where the expected result is 1 but 0 will be returned from the above procedure. A fix is when we hit the end of the given array and if left is not current, go backwards to calculate the volume until we hit left.

```
private int getVolume(int[] A, boolean isForward, int end, int[] boundary) {
  // skip zeros
  int cur = 0;
  if (!isForward) cur = A.length - 1;
  while (cur != end && A[cur] == 0) {
    cur += (isForward ? 1 : -1);
  }

  // calculate trapped volume
  int vol = 0, newEnd = cur;
  for (int i=cur, tempSum=0; i!=end; i+=(isForward ? 1 : -1)) {
    if (A[i] >= A[newEnd]) {
      vol += tempSum;
      // reset
      tempSum = 0;
      newEnd = i;
    } else {
      tempSum += (A[newEnd] - A[i]);
    }
  }
  boundary[0] = newEnd;
  return vol;
}

public int trap(int[] A) {
  int[] boundary = new int[1];
  int vol = getVolume(A, true, A.length, boundary);
  vol += getVolume(A, false, boundary[0]-1, boundary);

  return vol;
}
```

This algorithm runs in time O(n) and takes O(1) space.

Solution - Stack

This problem is similar to the problem of finding Largest Rectangle in Histogram [http://n00tc0d3r.blogspot.com/2013/03/largest-rectangle-in-histogram.html] . So, we can use a similar strategy here:

1. Use Stack to store the index of a bar;
2. If the current one is smaller then the top of the stack, push it to stack;
3. Otherwise, pop up the top until stack is empty or top is greater than the current one, add up the volume, push the current one to stack.

The tricky part is what is the volume to be added each time when we pop up a value from the stack.

Consider the following cases for testing the theory:

- [2, 0, 1, 2]
- [2, 1, 0, 2]
- [2, 1, 0, 1, 2]

We kind of add up level by level, as shown in picture below. The height is the smaller one of the left and right boundaries of the current bucket, and the width is the distance between left and right boundary.

[http://4.bp.blogspot.com/-0-8FciGddrE/UcedKc3AV0I/AAAAAAAAEgk/uPE2I-AFXm4/s1600/Screen+Shot+2013-06-23+at+6.12.10+PM.png]

```
public int trap(int[] A) {
  // skip zeros
  int cur = 0;
  while (cur < A.length && A[cur] == 0) ++cur;

  // check each one
  int vol = 0;
  Stack<Integer> stack = new Stack<Integer>();
  while (cur < A.length) {
    while (!stack.isEmpty() && A[cur] >= A[stack.peek()]) {
      int b = stack.pop();
      if (stack.isEmpty()) break;
      vol += ((cur - stack.peek() - 1) * (Math.min(A[cur], A[stack.peek()]) - A[b]));
    }
    stack.push(cur);
    ++cur;
  }

  return vol;
}
```

This algorithm runs in time O(n) since it touches each bar at most twice (push and pop) and it takes at most O(n) space for the stack.

Posted 24th June 2013 by Sophie

Labels: Array, Java, Stack

4 View comments

22nd June 2013

Find Element that Appears Once

Find the element that appears once [http://www.geeksforgeeks.org/find-the-element-that-appears-once/]

Given an array where every element occurs three times, except one element which occurs only once. Find the element that occurs once. Expected time complexity is O(n) and O(1) extra space.
Examples:

Input: arr[] = {12, 1, 12, 3, 12, 1, 1, 2, 3, 3}
Output: 2

Solution

We can sort the elements and then go through the sorted list to find out the number that has no duplicates. It takes O(1) space but runs in O(nlogn) time (There do exist sorting algorithms that have O(n) time complexity but they all requires special elements, e.g. elements are within a small range or have limited digits.).

We can use a hash table to store the occurrences of each element. It takes O(n) time but requires O(n) space.

We cannot use XOR on all elements since all elements occurring odd times (Reminder: n XOR n = 0 and n XOR n XOR n = n).

Notice that all other numbers occurs three times. That said, if we sum them up, the sum must be multiple of 3. Can we take advantage of this?
If the singleton number is less than 3, say {2, 2, 2, 1}, we can simply sum them up and mod by 3. But there is little chance of such situation...
What if we sum up each bit of numbers? Then for each digit, mod the sum by 3 and the left-over must be the bit in the single element.

```
public static int findSingle(int[] array) {
  int result = 0;

  // Note: Java integer size is always 32-bit
  for (int i=0; i<32; ++i) {
    int mask = (1 << i);
    int sum = 0;
    for (int j=0; j<array.length; ++j) {
      if ((array[j] & mask) != 0) ++sum;
    }
    result |= ((sum%3) << i);
  }

  return result;
}
```

This algorithm runs in time O(kn) = O(n), where n is the size of the given array and k is the total number of digits in an integer.

Note: We check `(array[j] & mask) != 0)` so that it could work for both positives and negatives. `-1 & (1 << 31) = -2147483648`

Note: Here [http://www.geeksforgeeks.org/find-the-element-that-appears-once/] Provides an alternative solution but I like this simple solution.

Posted 22nd June 2013 by Sophie

Labels: Array, BitManipulation, Java, Maths

6 View comments

19th June 2013

Summary: Tree and Graph

Tree

There are three type of Trees:

- **Binary Tree**: [Problems [http://n00tc0d3r.blogspot.com/search/label/BinaryTree] on Binary Tree]
  Each node has at most 2 subtrees, left and right subtrees.
- **Binary Search Tree [http://en.wikipedia.org/wiki/Binary_search_tree] (BST)**: [Problems [http://n00tc0d3r.blogspot.com/search/label/BinarySearchTree] on Binary Tree]
  BST is a binary tree where for any node x, values of all children nodes in left subtree are < x.value and values of all children in right subtree are > x.value. Each left and right subtree must also be a BST.
  An In-order traversal [http://en.wikipedia.org/wiki/Inorder_traversal#In-order] of a BST produces a sorted list of values.
  Looking up a value in a BST takes O(logn) time on average, where n is the total number of the nodes in the tree.
- **Heap [http://en.wikipedia.org/wiki/Heap_(data_structure)]**
  Heap is a special tree-based data structure. The root has the minimum value of all node values in a Min-heap, whereas in a Max-heap, the root has the maximum value. There is no implied orderings between siblings and children.
  Fing the min/max takes O(1) time in a min/max heap.
  Heap is an efficient implementation of a priority queue [http://en.wikipedia.org/wiki/Priority_queue] .

Recursion is good to solve tree-related problems. Even if an iterative solution is required, starting from a recursive solution may give you some hints about how to proceed.

Traversals

Three ways to traverse a binary tree: In-order [http://en.wikipedia.org/wiki/Inorder_traversal#In-order] , Pre-order [http://en.wikipedia.org/wiki/Inorder_traversal#Pre-order] and Post-order [http://en.wikipedia.org/wiki/Inorder_traversal#Post-order] . All of them are deep-first traversals.

Recursive implementations of three types of traversals is straightforward. Be familiar with both recursive and iterative implementations. See Binary Tree Traversals (I) [http://n00tc0d3r.blogspot.com/2013/01/binary-tree-traversals-i.html] for more discussions.

Transform

Other possible questions include transform a binary tree to a heap, balance a binary tree, and balance a BST.

Graph

Graph [http://en.wikipedia.org/wiki/Graph_(mathematics)] is a data structure that consists of a set of nodes/vertex and a set of edges connecting one node to another. In an undirected graph [http://en.wikipedia.org/wiki/Graph_(mathematics)#Undirected_graph] , edges have no orientations, i.e. edge (a, b) is equivalent to edge (b, a); whereas in a directed graph [http://en.wikipedia.org/wiki/Graph_(mathematics)#Directed_graph] , edge (a, b) is an arc starting from node a and directed to node b. Tree can be considered as a special type of undirected graph.

Traversals (Search)

Breath-First Search [http://en.wikipedia.org/wiki/Breath-first_search] (BFS)

- Surrounded Regions [http://n00tc0d3r.blogspot.com/2013/06/surrounded-regions.html]
- Maximum Depth of Binary Tree [http://n00tc0d3r.blogspot.com/2013/04/maximum-depth-of-binary-tree.html]
- Tree Comparison [http://n00tc0d3r.blogspot.com/2013/05/same-tree.html]

Depth-First Search [http://en.wikipedia.org/wiki/Depth-first_search] (DFS)

- Surrounded Regions [http://n00tc0d3r.blogspot.com/2013/06/surrounded-regions.html]
- Maximum Depth of Binary Tree [http://n00tc0d3r.blogspot.com/2013/04/maximum-depth-of-binary-tree.html]
- Sum Root to Leaf Numbers [http://n00tc0d3r.blogspot.com/2013/06/sum-root-to-leaf-numbers.html]
- Palindrome Partitioning [http://n00tc0d3r.blogspot.com/2013/05/palindrome-partitioning.html]
- N-Queens [http://n00tc0d3r.blogspot.com/2013/04/n-queens.html]

Shortest Path Algorithms

Single source

- Dijkstra's algorithm [http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm] : Given a weighted undirected graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between a given node and every other nodes. It can also be used for finding costs of shortest paths from one given node to one goal node by stopping the algorithm once the shortest path to the destination node has been determined.
  Dijkstra's algorithm picks the unvisited vertex with the lowest-distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. Mark visited (set to red) when done with neighbors.
  An implementation of Dijkstra's algorithm using Fibonacci heap [http://en.wikipedia.org/wiki/Fibonacci_heap] gives running time O(|E|+|V|log|V|).
- A* Search algorithm [http://en.wikipedia.org/wiki/A*_search_algorithm] : Given a weighted undirected graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between a given node and one goal node.
  A* uses BFS to traverses the graph, it follows a path of the lowest expected total cost or distance, keeping a sorted priority queue of alternate path segments along the way. It uses a cost function on node to determine the order of visiting nodes in the tree. The cost function is a sum of the past path-cost and a heuristic estimated future path-cost to the goal node.

All pairs

- Floyd-Warshall algorithm [http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm] : Given a weighted undirected graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between all pairs of nodes.
  This algorithm is also a good example of Dynamic Programming. The core formula is
  ```
  shortestPath(i, j, 0) = w(i, j);
  shortestPath(i, j, k+1) = Min(shortestPath(i, j, k), shortestPath(i, k+1, k) +
  ```

`shortestPath(k+1, j, k)`
where `shortestPath(i, j, k)` is shortest possible path from i to j using intermedia nodes only from the set {1, 2, ..., k}.

Ref: <Programming Interviews Exposed [http://www.amazon.com/gp/product/1118261364/ref=as_li_ss_il?ie=UTF8&camp=1789&creative=390957&creativeASIN=1118261364&linkCode=as2&tag=n00tc0d3r-20] > Chapter 5.

Posted 19th June 2013 by Sophie

Labels: BinarySearchTree, BinaryTree, DP, Graph, Recursion, Summary, Tree

1 View comments

15th June 2013

Surrounded Regions

Surrounded Regions [http://leetcode.com/onlinejudge#question_130]

Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'.
A region is captured by flipping all 'O's into 'X's in that surrounded region .

For example,

```
X X X X
X O O X
X X O X
X O X X
```

After running your function, the board should be:

```
X X X X
X X X X
X X X X
X O X X
```

Solution

First question to yourself is which 'O' cells would or would not be flipped to 'X'.

According to the definition, if 'O' cell is surrounded by 'X' cells, i.e. up/down/left/right cells are 'X'.
The first thought could be for each 'O' cell, add it to an array, check its up/down/left/right and if it is another 'O' cell, continue until, it hits all 'X' cells or it hits boundary. In the former case, cells in the array can all be flipped to'X'; while in the latter case, cells in the array cannot be flipped.

But it is not easy to check cells in the middle of board.
Notice that any 'O' cells that connected (directly or indirectly) to a boundary 'O' cell can not be flipped. So, we don't need to keep track of both flip-able and non-flip-able 'O' cells. We can start from boundary cells and use BFS or DFS to find out all its neighbor 'O' cells -- those are the non-flip-able 'O' cells!

The algorithm will be something like:

- Start from those boundary cells, use BFS or DFS to traverse all non-flipable 'O' cells, and mark them with a special sign, say 'N'.
- Revisit the board, flip all remaining 'O' cells to 'X' and also flip back 'N' cells to 'O'.

Notice that for each boundary cell, no matter what value it has, it will **never** be flipped. Also, for the four corner cells, their value will **not** impact any other cells.
So, we can improve the above algorithm a little bit: only mark non-boundary cells and then only flip non-boundary 'O' and 'N' cells.

```
private void mark(char[][] board, int row, int col, Queue<Integer> que) {
  if (board[row][col] != 'O') {
    return;
  }
  board[row][col] = 'N';
  int columns = board[0].length;
  que.offer(row * columns + col);
}

private void markBFS(char[][] board, int row, int col) {
  Queue<Integer> que = new LinkedList<Integer>();
  mark(board, row, col, que);
  int rows = board.length, columns = board[0].length;
  while (!que.isEmpty()) {
    int cell = que.poll();
    int x = cell / columns, y = cell % columns;
    // push its neighbors to stack if needed
    if (x+1 < rows-1)  mark(board, x+1, y, que);
    if (x-1 > 0)  mark(board, x-1, y, que);
    if (y+1 < columns-1)  mark(board, x, y+1, que);
    if (y-1 > 0)  mark(board, x, y-1, que);
  }
}

private void markDFS(char[][] board, int x, int y) {
  if (board[x][y] != 'O') {
    return;
  }
  // mark the current node
  board[x][y] = 'N';
  // mark its neighbors if needed
  int rows = board.length, columns = board[0].length;
  if (x+1 < rows-1)  markDFS(board, x+1, y);
  if (x-1 > 0)  markDFS(board, x-1, y);
  if (y+1 < columns-1)  markDFS(board, x, y+1);
  if (y-1 > 0)  markDFS(board, x, y-1);
}

public void solve(char[][] board) {
  if (board.length <= 0 || board[0].length <= 0) return;
  int rows = board.length, columns = board[0].length;

  // Start from 'O's on the edge and mark connected ones as non-flipable.
  // first and last columns
  for (int i=1; columns>2 && i<rows-1; ++i) {
    if (board[i][0] == 'O')  markBFS(board, i, 1);
    if (board[i][columns-1] == 'O')  markBFS(board, i, columns-2);
  }
  // first and last rows
  for (int j=1; rows>2 && j<columns-1; ++j) {
    if (board[0][j] == 'O')  markBFS(board, 1, j);
    if (board[rows-1][j] == 'O')  markBFS(board, rows-2, j);
  }

  // flip
  for (int i=1; i<rows-1; ++i) {
    for (int j=1; j<columns-1; ++j) {
      if (board[i][j] == 'O') {
        board[i][j] = 'X';
      } else if (board[i][j] == 'N') {
        board[i][j] = 'O';
      }
    }
  }
}
```

Posted 15th June 2013 by Sophie

Labels: Array, BFS, DFS, Graph, Java, Stack

3 View comments

15th June 2013

Swap Nodes in Pairs

Swap Nodes in Pairs [http://leetcode.com/onlinejudge#question_24]

Given a linked list, swap every two adjacent nodes and return its head.

For example,
Given 1->2->3->4, you should return the list as 2->1->4->3.

Your algorithm should use only constant space. You may **not** modify the values in the list, only nodes itself can be changed.

Solution

(This problem is same as the second problem in this post [http://n00tc0d3r.blogspot.com/2013/05/reverse-linked-list.html] , Reverse Linked List in k-Groups when k = 2.)

For nodes in the middle of the list, swap two nodes, say node A->B, is simple:

- Find the node, say node N, ahead of the two nodes (i.e. ahead of node A) and store its next node (i.e. node A) to a temp node;
- Set node B as the next node of N;
- Set node B's next as the next node of A;
- Set node A as the next node of B;
- Forward N to B and repeat above steps if it has two next nodes.

How about the last node?
The last node doesn't require any special process in this problem. The only difference of the last node comparing with a middle node is that its next node is null.

How about the head node?
Swapping the first two nodes requires to update the head node. We can add a dummy node to the list and then handle the head node as other middle nodes.

The following algorithm runs in time O(n) and it is optimal since it touch each node O(1) times and only use O(1) extra space.

```
public ListNode swapPairs(ListNode head) {
  ListNode dummy = new ListNode(0);
  dummy.next = head;
  ListNode cur = dummy;

  while (cur.next != null && cur.next.next != null) {
    ListNode temp = cur.next;
    cur.next = cur.next.next;
    cur = cur.next;
    temp.next = cur.next;
    cur.next = temp;
    cur = cur.next;
  }

  return dummy.next;
}
```

Posted 15th June 2013 by Sophie

Labels: Java, LinkedList

0 Add a comment

15th June 2013

Sum Root to Leaf Numbers

Sum Root to Leaf Numbers [http://leetcode.com/onlinejudge#question_129]

Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.
An example is the root-to-leaf path 1->2->3 which represents the number 123.
Find the total sum of all root-to-leaf numbers.

For example,

```
    1
   / \
  2   3
```

The root-to-leaf path 1->2 represents the number 12; The root-to-leaf path 1->3 represents the number 13.
Return the sum = 12 + 13 = 25.

Solution

This is another variant of Tree Path Sum [http://n00tc0d3r.blogspot.com/2013/01/tree-path-sum.html] problem.

It is easy to solve it with a recursive DFS solution.

- If it is a leaf, add the current path sum to the sum and return the sum;
- If it has left subtree, add the left child node value to the path sum and recursively traverse the left subtree;
- Similarly, if it has right subtree, add the right child node value to the path sum and continue on the right subtree.

```
// path is the sum from root to the current node
// sum is the sum of all root-to-leaf numbers up to the current node
private int sumNumbersHelper(TreeNode root, int path, int sum) {
  if (root.left == null && root.right == null) { // reach a leaf
    return sum+path;
  }

  if (root.left != null) { // go to left subtree
    sum = sumNumbersHelper(root.left, path*10+root.left.val, sum);
  }
  if (root.right != null) { // go to right subtree
    sum = sumNumbersHelper(root.right, path*10+root.right.val, sum);
  }

  return sum;
}

public int sumNumbers(TreeNode root) {
  if (root == null) return 0;
  return sumNumbersHelper(root, root.val, 0);
}
```

We touch each node once in this algorithm and thus the total running time is O(n). Since we use recursion, it takes O(logn) space for stacks.

Posted 15th June 2013 by Sophie

Labels: BinaryTree, DFS, Java, Recursion, Tree

0 Add a comment

14th June 2013

Sudoku Solver

Sudoku Solver [http://leetcode.com/onlinejudge#question_37]

Write a program to solve a Sudoku puzzle by filling the empty cells. Empty cells are indicated by the character '.'.
Rules of Sudoku:

- Each row must have the numbers 1-9 occuring just once.

- Each column must have the numbers 1-9 occuring just once.
- And the numbers 1-9 must occur just once in each of the 9 sub-boxes of the grid.

You may assume that there will be only one unique solution.

[http://2.bp.blogspot.com/-33tAlJ3AQrE/UbjK2G54BxI/AAAAAAAAEf0/xyDHHjl3YFo/s1600/square.jpg]

Solution

How to solve a Sudoku?
Here is how I solve it:

1. For each empty cell, find out candidate values by checking its row/column/box.
   - Visit the grid once to collect numbers in each row/column/box
   - For each empty cell, initialize with all 9 numbers and then remove conflict numbers in the same row/column/box.
2. Then pick up a cell with the least number of candidates,
   - Try to fill in a candidate value, and remove the number from candidate sets of the same row/column/box;
   - Repeat step 2 until either the Sudoku is successfully solved or a conflict is detected. For the latter case, recover the touched candidate sets and try to fill in another value.

Step 1 takes time O(n^2) to visit the grid twice, where n is the size of the grid (i.e. n=9). While the running time of Step 2 depends on the total number of empty cells and number of candidates. Suppose there are m empty cells and the number of candidates for each empty cell must be less than n. Updating all related candidate sets takes time O(m). So, the total running time is Ω(n*m^2) (see wiki for Big Omega notation [http://en.wikipedia.org/wiki/Big_O_notation#Big_Omega_notation]).

Each time we start from an empty cell with most constraints (i.e. with least number of candidates), which can improve the overall performance. The actual running time is much lower than (n*m^2). If we use another data structure to track the candidate sets for each row/column/box, then the time can be reduce to Ω(m*n^2). But this optimization increases the difficulty of implementation.

```
private class Pair<A, B> {
  public A first;
  public B second;
  public Pair(A a, B b) {
    first = a; second = b;
  }
  public String toString() {
    return "(" + first + ", " + second + ")";
  }
  public boolean equals(Pair pp) {
    return (pp != null && first==pp.first && second==pp.second);
  }
}

// For each cell, populate an array of candidate numbers.
// Return the cell with least number of candidates.
private Pair<Integer, Integer> init(char[][] board, HashMap<Pair<Integer, Integer>, HashSet<Integer>> 
  // collect existing numbers
  ArrayList<ArrayList<Integer>> numbersInRow = new ArrayList<ArrayList<Integer>>();
  ArrayList<ArrayList<Integer>> numbersInCol = new ArrayList<ArrayList<Integer>>();
  ArrayList<ArrayList<Integer>> numbersInBox = new ArrayList<ArrayList<Integer>>();
  for (int i=0; i<9; ++i) {
    ArrayList<Integer> row = new ArrayList<Integer>(); // row i
    ArrayList<Integer> col = new ArrayList<Integer>(); // column i
    for (int j=0; j<9; ++j) {
      if (board[i][j] != '.') row.add(board[i][j]-'0');
      if (board[j][i] != '.') col.add(board[j][i]-'0');
    }
    numbersInRow.add(row);
    numbersInCol.add(col);
  }
  for (int i=0; i<9; i+=3) {
    for (int j=0; j<9; j+=3) {
      ArrayList<Integer> box = new ArrayList<Integer>(); // box i
      for (int r=i; r<i+3; ++r) {
        for (int c=j; c<j+3; ++c) {
          if (board[r][c] != '.') box.add(board[r][c]-'0');
        }
      }
      numbersInBox.add(box);
    }
  }
  // initialize with all empty cells
  int minSet = 10;
  Pair<Integer, Integer> minCell = null;
  for (int i=0; i<9; ++i) {
    int boxR = (i/3) * 3;
    for (int j=0; j<9; ++j) {
      if (board[i][j] == '.') {
        Pair<Integer, Integer> cell = new Pair<Integer, Integer>(i, j);
        HashSet<Integer> set = new HashSet<Integer>(Arrays.asList(1,2,3,4,5,6,7,8,9));
        set.removeAll(numbersInRow.get(i));
        set.removeAll(numbersInCol.get(j));
        set.removeAll(numbersInBox.get(boxR + j/3));
        cand.put(cell, set);
        if (set.size() < minSet) {
          minCell = cell; minSet = set.size();
        }
      }
    }
  }
  return minCell;
}

// Fill in the given cell with the given value and update candidate sets of same row/column/box.
// Return the cell with least number of candidates. If a conflict is detected, return the given cell.
private Pair<Integer, Integer> fillOneCell(char[][] board,
    Pair<Integer, Integer> cell, int num,
    ArrayList<Pair<Integer, Integer>> touched,
    HashMap<Pair<Integer, Integer>, HashSet<Integer>> cand) {
  board[cell.first][cell.second] = (char)('0' + num);
  int minSet = 10;
  Pair<Integer, Integer> minCell = null;
  // resolve conflicts
  for (Pair<Integer, Integer> c : cand.keySet()) {
    if (c.first == cell.first || c.second == cell.second
        || ((c.first/3) == (cell.first/3) && (c.second/3) == (cell.second/3))) {
      if (cand.get(c).remove(num)) {
        touched.add(c);
        if (cand.get(c).isEmpty()) return cell;
      }
    }
    if (minCell == null || cand.get(c).size() < minSet) {
      minCell = c; minSet = cand.get(c).size();
    }
  }
  return minCell;
}

// Recursively solve a Sudoku.
// Return true if the current board is a valid Sudoku (could be partially filled).
private boolean solveSudokuHelper(char[][] board, Pair<Integer, Integer> cell,
    HashMap<Pair<Integer, Integer>, HashSet<Integer>> cand) {
  if (cand.isEmpty()) return true;
  HashSet<Integer> set = cand.get(cell);
  cand.remove(cell);
  for (Integer num : set) { // try each candidate value
    ArrayList<Pair<Integer, Integer>> touched = new ArrayList<Pair<Integer, Integer>>();
    Pair<Integer, Integer> minCell = fillOneCell(board, cell, num, touched, cand);
    // try next cell
    if (!cell.equals(minCell) && solveSudokuHelper(board, minCell, cand)) return true;
    // recover
    for (Pair<Integer, Integer> cc : touched) {
      cand.get(cc).add(num);
    }
  }
  cand.put(cell, set);
  return false;
}

public void solveSudoku(char[][] board) {
  HashMap<Pair<Integer, Integer>, HashSet<Integer>> cand =
      new HashMap<Pair<Integer, Integer>, HashSet<Integer>>();
  Pair<Integer, Integer> cell = init(board, cand);
  solveSudokuHelper(board, cell, cand);
}
```

We can also solve the problem without the optimization, which makes the code much simpler.

The basic idea is as follows:

1. Starting from (0,0), find the next empty cell;
2. Try every possible number (1, ..., 9) to fill in the empty cell and check whether it produces a valid Sudoku;
   Note that we only need to check the current row/column/box now.
3. If the resulted Sudoku is valid, go back to Step 1; Otherwise, repeat Step 2.

Finding empty cells one by one takes time O(n^2) in total, where n is the size of grid (i.e. n=9), since each time we start from the previous cell and thus only visit the grid once.
Each call of isValidSudoku takes time O(n) but it has been called at most O(n*m), where m is the total number of empty cells.
So, the total time of this algorithm is O(m*n^2).

```
private class Pair<A, B> {
  public A first;
  public B second;
  public Pair(A a, B b) {
    first = a; second = b;
  }
  public String toString() {
    return "(" + first + ", " + second + ")";
  }
  public boolean equals(Pair pp) {
    return (pp != null && first==pp.first && second==pp.second);
  }
}

private Pair<Integer, Integer> nullCell = new Pair<Integer, Integer>(-1, -1);

// Find the next empty cell.
// Return true if found one. Otherwise, return false.
private Pair<Integer, Integer> findNextEmptyCell(char[][] board, Pair<Integer, Integer> cell) {
  for (int i=cell.first; i<9; ++i) {
    for (int j=(i==cell.first)?cell.second:0; j<9; ++j) {
      if (board[i][j] == '.') {
        return new Pair<Integer, Integer>(i, j);
      }
    }
  }
  return nullCell;
}

// Check whether the board is still valid after fill in a cell.
// Only need to check the current row/column/box
private boolean isValidSudoku(char[][] board, Pair<Integer, Integer> cell) {
  // check row and column
  for (int i=0; i<9; ++i) {
    if (i != cell.second && board[cell.first][i] == board[cell.first][cell.second]) {
      return false;
    }
    if (i != cell.first && board[i][cell.second] == board[cell.first][cell.second]) {
      return false;
    }
  }
  // check box
  for (int row=(cell.first/3)*3, i=row; i<row+3; ++i) {
    for (int col=(cell.second)/3*3, j=col; j<col+3; ++j) {
      if (i != cell.first && j != cell.second && board[i][j] == board[cell.first][cell.second]) {
        return false;
      }
    }
  }
  return true;
}

private boolean solveSudokuHelper(char[][] board, Pair<Integer, Integer> c) {
  Pair<Integer, Integer> cell = findNextEmptyCell(board, c);
  if (cell.equals(nullCell)) return true;
  // try every possible number
  for (int i=1; i<10; ++i) {
    board[cell.first][cell.second] = (char)('0' + i);
    if (isValidSudoku(board, cell) && solveSudokuHelper(board, cell)) {
      return true;
    }
    // recover
    board[cell.first][cell.second] = '.';
  }
  return false;
}

public void solveSudoku(char[][] board) {
  solveSudokuHelper(board, new Pair<Integer, Integer>(0, 0));
}
```

Posted 14th June 2013 by Sophie

Labels: Array, Java, Recursion

1 View comments

13th June 2013

Valid Sudoku

Valid Sudoku [http://leetcode.com/onlinejudge#question_36]

Determine if a Sudoku is valid.
Rules of Sudoku:

- Each row must have the numbers 1-9 occuring just once.
- Each column must have the numbers 1-9 occuring just once.
- And the numbers 1-9 must occur just once in each of the 9 sub-boxes of the grid.

[http://4.bp.blogspot.com/-33tAlJ3AQrE/UbjK2G54BxI/AAAAAAAAEfw/y4k21KtBYFg/s1600/square.jpg]
The Sudoku board could be partially filled, where empty cells are filled with the character '.'.

Solution

The question itself is easy. A straightforward solution is as follows:

- Verify each row;
- Verify each column;
- Verify each 3*3 block.

This algorithm runs in time O(n^2), where n is the size of the square (n=9 in this case). The order of the running time is optimal since we have to check each spot at least once. And we use O(n) space to keep track of numbers that have seen in the current block (row/column/box). But we use extra O(n) time and spaces to copy the numbers to a block.

```
private boolean isValid(char[] block) {
  boolean[] numbers = new boolean[9];
  for (Character c : block) {
    if (c == '.') continue;
    if (!(c >= '1' || c <= '9') || numbers[c-'1']) return false;
    numbers[c - '1'] = true;
  }
  return true;
}

public boolean isValidSudoku(char[][] board) {
  if (board.length != 9) return false;
  if (board[0].length != 9) return false;

  // check rows
  for (int i = 0; i < 9; ++i) {
    char[] row = Arrays.copyOf(board[i], 9);
    if (!isValid(row)) return false;
  }
  // check columns
  for (int i = 0; i < 9; ++i) {
    char[] column = new char[9];
    for (int j=0; j < 9; ++j) column[j] = board[j][i];
    if (!isValid(column)) return false;
  }
  // check box
  for (int i = 0; i < 3; ++i) {
    for (int k=0; k < 9; k+=3) {
      char[] box = new char[9];
      for (int j=0; j < 3; ++j) System.arraycopy(board[i*3+j], k, box, j*3, 3);
      if (!isValid(box)) return false;
    }
  }

  return true;
}
```

We can revise the auxiliary function by passing in the boolean array and then we don't need to perform the copy any more.

```
private boolean isValid(char c, boolean[] numbers) {
  if (c == '.') return true;
  if (!(c >= '1' || c <= '9') || numbers[c-'1']) return false;
  numbers[c - '1'] = true;
  return true;
}

public boolean isValidSudoku(char[][] board) {
  if (board.length != 9) return false;
  if (board[0].length != 9) return false;

  for (int i = 0; i < 9; ++i) {
    boolean[] block = new boolean[9];
    for (int j=0; j < 9; ++j) { // row i
      if (!isValid(board[i][j], block)) return false;
    }

    Arrays.fill(block, false);
    for (int j=0; j < 9; ++j) { // column i
      if (!isValid(board[j][i], block)) return false;
    }

    Arrays.fill(block, false);
    int row = (i/3)*3, col = (i%3)*3;
    for (int j=0; j < 9; ++j) { // box i
      if (!isValid(board[row+j/3][col+(j%3)], block)) return false;
    }
  }

  return true;
}
```

Posted 13th June 2013 by Sophie

Labels: Array, Java

0 Add a comment

12th June 2013

Substring with Concatenation of All Words

Substring with Concatenation of All Words [http://leetcode.com/onlinejudge#question_30]

You are given a string, S, and a list of words, L, that are all of the same length. Find all starting indices of substring(s) in S that is a concatenation of each word in L exactly once and without any intervening characters.

For example, given:
S: "barfoothefoobarman"
L: ["foo", "bar"]

You should return the indices: [0,9].
(order does not matter).

Solution

The basic idea is to iterate through the string S and check whether the current index is a starting point of a concatenation of all words in the list L.

How to check whether the current index is a starting point of a concatenation?
Since we know that all of the words in the given list has the same length, starting from index i, we can cut the string into m substrings, where m is the total number of words in list L, and then check whether each substring is a word in L and is not duplicate.

How to check whether each substring is a word in L?
A straightforward way is to compare the substring with every word in L. But for each substring, it takes time O(m) which is quite expensive, especially when we have a lot of substrings or words. Well, it actually takes time O(k*m) = O(m), where k is the length of each word and is a constant in this case. That said, if words in list L are not very short, the actual running time of this operation gets more expensive.
We can add words of list L into a hash table and searching in hash only takes O(1) time.

What kind of hash table? How to detect duplicate substrings? What if L contains duplicate words?
To check a concatenation, we create a hash that contains all words in L. Each time when a substring matches a word, remove the word from the hash.
Instead of using HashMap<String, Boolean>, use HashMap<String, Integer> to record the frequencies of each word. And when we hit a word in the hash, reduce its frequency until it gets down to zero (in that case, the key would have been removed from the hash.).

Now we have a draft of the algorithm:

- Create a hash containing all words in list L and also their frequencies.
- For each character of string S, check whether it is a concatenation. If it satisfies all of the following conditions, add the index to the result list.
    - Each following substring of the word length should be a word in list L.
    - Each word in list L should occur once and only once.

What is the complexity of this algorithm?
The time of checking a concatenation is O(k*m) = O(m) and we check it (n - k*m) times, so the total running time is O((n - k*m)*m). The space complexity is the size of list L, which is O(k*m) = O(m), since we create a hash table of list L.

```java
public ArrayList<Integer> findSubstring(String S, String[] L) {
  ArrayList<Integer> indices = new ArrayList<Integer>();
  if (L.length == 0) return indices;

  int total = L.length, wordLen = L[0].length();

  // store the words and frequencies in a hash table
  HashMap<String, Integer> words = new HashMap<String, Integer>();
  for (String s : L) {
    if (words.containsKey(s)) {
      words.put(s, words.get(s)+1);
    } else {
      words.put(s, 1);
    }
  }

  // find concatenations
  for (int i=0; i <= S.length() - total*wordLen; ++i) {
    // check if it is a concatenation
    HashMap<String, Integer> target = new HashMap<String, Integer>(words);
    for (int j = i; j <= S.length() - wordLen && !target.isEmpty(); j+=wordLen) {
      String sub = S.substring(j, j+wordLen);
      if (!target.containsKey(sub)) break;
      if (target.get(sub) > 1) {  // reduce the frequency
        target.put(sub, target.get(sub)-1);
      } else {  // remove the word if only one left
        target.remove(sub);
      }
    }
    if (target.isEmpty()) {
      indices.add(i);
    }
  }

  return indices;
}
```

Notice that in the algorithm above, we check each substrings multiple times. Do we really need to recheck all substrings followed by each character? The answer is "No".

We can use a similar strategy of KMP string matching algorithm [http://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm] :
Instead of restart from beginning, forward the begin pointer to next possible place.

Recall that we are looking for continuous concatenations of words in list L and all words in L are of the same length. To cover all possibilities, we can start from a letter of S[0..k-1] and search for all potential concatenation from that position. That is, we check concatenation at 0, k, 2k, ..., n-1; then 1, k+1, 2k+1, ..., n-1; then 2, k+2, 2k+2, ..., n-1, etc.; until k-1, 2k-1, 3k-1, ..., n-1.
Start from the first character of S (begin = 0), cut a substring of length k,

- If it is not a valid word in list L, restart from next substring (i.e. begin += k);
- If it is a valid word but has seen before (i.e. a duplicate), forward begin pointer until previous one is removed from the current window;
- Otherwise, add the substring to the current collection. If we get all words (i.e. hit a concatenation), forward begin pointer by one and check the next substring.

What is the complexity now?
We hit each substring at most twice, one to add into the collection and one to remove from the collection. There are O(n) substrings in total, where n is the total number of characters in string S, and for each substring it takes O(k) = O(1) to check whether it is a valid word or not.
So, in total, this algorithm runs in time O(n*k) = O(n).

```
private void addWord(String w, HashMap<String, Integer> words) {
  if (words.containsKey(w)) {
    words.put(w, words.get(w)+1);
  } else {
    words.put(w, 1);
  }
}

private void removeWord(String w, HashMap<String, Integer> words) {
  if (!words.containsKey(w)) return;
  if (words.get(w) > 1) {
    words.put(w, words.get(w)-1);
  } else {
    words.remove(w);
  }
}

private int slideWindow(String S, int begin, int wordLen, HashMap<String, Integer> words) {
  String old = S.substring(begin, begin+wordLen);
  addWord(old, words);
  return begin+wordLen;
}

public ArrayList<Integer> findSubstring(String S, String[] L) {
  ArrayList<Integer> indices = new ArrayList<Integer>();
  if (L.length == 0) return indices;

  int total = L.length, wordLen = L[0].length();

  // store the words and frequencies in a hash table
  HashMap<String, Integer> expectWords = new HashMap<String, Integer>();
  for (String w : L) {
    addWord(w, expectWords);
  }

  // find concatenations
  for (int i=0; i < wordLen; ++i) {
    // check if there are any concatenations
    int count = 0;
    HashMap<String, Integer> collectWords = new HashMap<String, Integer>(expectWords);
    for (int j = i, begin = i; j <= S.length() - (total-count)*wordLen && begin <= S.length() - total*
      String sub = S.substring(j, j+wordLen);
      if (!expectWords.containsKey(sub)) { // if not an expect word, reset
        begin = j + wordLen;
        j = begin;
        count = 0;
        collectWords.putAll(expectWords);
      } else if (!collectWords.containsKey(sub)) { // if duplicate, forward begin by 1
        begin = slideWindow(S, begin, wordLen, collectWords);
      } else {
        removeWord(sub, collectWords);
        j += wordLen;
        ++count;
        if (collectWords.isEmpty()) {
          indices.add(begin);
          begin = slideWindow(S, begin, wordLen, collectWords);
          --count;
        }
      }
    }
  }
  return indices;
}
```

Posted 12th June 2013 by Sophie

Labels: Array, Hash, Java, String

4 View comments

10th June 2013

Summary: Linked List

There are three types of linked list:

- **Singly linked list**
  Each list node has one and only one pointer pointing to the next node. Typically a head pointer to the list is provided to traverse the list and the traversal can only go forward.
- **Doubly linked list**
  Each list node has two pointers, one to next and the other to previous node. A head pointer, and sometimes a tail pointer, are going to be provides to traverse the list and the list can be traversed forwards and backwards.
- **Circular linked list**
  No head or tail pointers exist in such a list since "tail" is connected to "head".

Singly linked list is preferable in technical interviews since questions on such lists are non-trivial, while most of them turns to be trivial when using doubly linked list.

Algorithms/Tricks

Two pointers

Two pointers is to use two pointers moving with different speed.
For problems on singly linked list, since such type of linked list can only be traversed forward, sometimes two pointers may provide a decent O(n) solution, instead of repeatedly restart from head again and again (which could be O(n^2)).

Here are some example problems that are solved with two pointers:

- Rotate List [http://n00tc0d3r.blogspot.com/2013/05/rotate-list.html]
- Remove n-th-to-end Element From List [http://n00tc0d3r.blogspot.com/2013/05/remove-n-th-to-end-element-from-list.html]

Dummy head

Adding dummy head in front of the give head can reduce the complexity of the algorithm: Don't need to consider the special cases for head node since it becomes a middle node.

Here are some example problems that are solved with dummy head:

- Reverse Linked List [http://n00tc0d3r.blogspot.com/2013/05/reverse-linked-list.html]
- Partition List [http://n00tc0d3r.blogspot.com/2013/05/partition-list.html]
- Merge Sorted Lists/Arrays [http://n00tc0d3r.blogspot.com/2013/04/merge-sorted-listsarrays-i.html]

Edge Cases for Tests

When checking an algorithm for a linked list, ask yourself the follow questions.

Have you maintained the head(and tail) pointer?

If you lost the head pointer, that means you lost the list.

Do not modify the head pointer if it is not supposed to be changed.
For example, when traversing a list, you need to use another pointer to iterate through the list, instead of moving the original head.
The following code won't work:

```
public void printEvenAndOdd(ListNode head) {
  int count = 0;
  // print even nodes first
  System.out.println("Even nodes:");
  while (head != null) {
    if ((count & 1) == 0)
      System.out.print(head.value + "->");
    head = head.next;
    ++count;
  }
  System.out.println("End");
  //// You cannot print odd nodes any more... (X)
}
```

Using a current pointer fixes the issue:

```
public void printEvenAndOdd(ListNode head) {
  int count = 0;
  ListNode cur = head;
  // print even nodes first
  System.out.println("Even nodes:");
  while (head != null) {
    if ((count & 1) == 0)
      System.out.print(head.value + "->");
    head = head.next;
    ++count;
  }
  System.out.println("End");
  // print odd nodes first
  System.out.println("Odd nodes:");
  cur = head; // reset to the head of the list
  while (head != null) {
    if ((count & 1) == 1)
      System.out.print(head.value + "->");
    head = head.next;
    ++count;
  }
  System.out.println("End");
}
```

On the other hand, do update the head pointer if it is expected.
For example, when inserting or deleting the head node. In such cases, in Java, since input arguments are passed in by value (see here [http://sophie-notes.blogspot.com/2012/12/java-passing-arguments-into-methods.html] for more discussions), a return value for the new header is expected; while in C++, either return the new head as in Java, or pass in a pointer to the head pointer so as to modify the value of the head pointer.
In the following code, the actual head pointer never gets updated.

```
public void insertToHead(ListNode head, int val) {
  ListNode node = new ListNode(val);
  node.next = head;
  head = node; // This only updates a local copy of the head pointer.
}
```

To update the head pointer, in Java, we have to return the new head and the caller should be something like `head = insertToHead(head, 5);`.

```
public ListNode insertToHead(ListNode head, int val) {
  ListNode node = new ListNode(val);
  node.next = head;
  return node; // Return the new head.
}
```

In C++, it will look like this:

```
void insertToHead(ListNode** head, int val) { // Pass in a pointer to the head.
  ListNode* node = new ListNode(val);
  node->next = *head;
  *head = node;
}
```

Have you considered edge cases?

To design an algorithm for a linked list, it is okay to start with the case where the desired node is in the middle of the list. But after you finish working on the basic case, you need to check your code with special cases, such as:

- Does it works for Empty list (where `head == null` and `tail == null`)?
- Does it works for Single-node list (where `head == tail`)?
- Does it works for Two-node list (where no "middle" node between head and tail)?
- What if pass-in the head ponter itself is null?
- Have we reached the end of list during a loop?

Usually, Interview questions on linked list are not difficult to design, but is not easy to get it right at the first time with those special cases. Be careful and make sure you test all edge cases when you've done.

Ref: <Programming Interviews Exposed [http://www.amazon.com/gp/product/1118261364/ref=as_li_ss_il?ie=UTF8&camp=1789&creative=390957&creativeASIN=1118261364&linkCode=as2&tag=n00tc0d3r-20] > Chapter 4.

Posted 10th June 2013 by Sophie

Labels: LinkedList, Summary

0 Add a comment

6th June 2013

Subsets

Subsets - No Duplicates [http://leetcode.com/onlinejudge#question_78]

Given a set of distinct integers, S, return all possible subsets.

Note:

- Elements in a subset must be in non-descending order.
- The solution set must **not** contain duplicate subsets.

For example,
If S = [1,2,3], a solution is: [[3],[1], [2], [1,2,3], [1,3], [2,3], [1,2], []]

Solution - Recursion

How to compute subsets?

Given S = [1, 2, 3], subsets started with 1 are [1], [1, 2], [1, 3], [1, 2, 3]. That is, add 1 to all subsets made up with elements 2 and 3, i.e. elements after 1 in S.
Similarly, subsets started with 2 are [2], [2, 3], subset started with 3 is [3], and empty subset [] (which can be considered as subset started with element after 3 in S :).

This gives us a recursive solution:
For each element in S,

- First populate subsets made of elements after the current one;
- Then for each subsets, generate a new subset which starts with current element and append elements in the subset.

```
private ArrayList<ArrayList<Integer>> subsetsHelper(int[] S,
    int cur, ArrayList<ArrayList<Integer>> results) {
  // reach end of set -> add [] as a subset
  if (cur >= S.length) {
    results.add(new ArrayList<Integer>());
    return results;
  }
  // generate subsets with remaining elements
  results = subsetsHelper(S, cur+1, results);
  // append the current one to all subsets made up with remaining elements
  int curSize = results.size();
  while (curSize-- > 0) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    res.add(S[cur]);
    res.addAll(results.get(curSize));
    results.add(res);
  }
  return results;
}

public ArrayList<ArrayList<Integer>> subsets(int[] S) {
  // sort the given set
  Arrays.sort(S);
  // generate subsets
  return subsetsHelper(S, 0, new ArrayList<ArrayList<Integer>>());
}
```

This algorithm touches each subset once. It runs in time O(2^n) and use O(2^n) spaces, which is the same as the result size. So, although it runs in exponential time, it is optimal.
Why 2^n? C(n, 0) + C(n, 1) + C(n, 2) + ... + C(n, n) = 2^n.

Solution - Iteration

This can also be done with iterations with the same idea.

A small change is that now we solve it in another direction such that we consider subsets ending with an element and subsets made of elements up to the current one.

```
public ArrayList<ArrayList<Integer>> subsets(int[] S) {
  // sort the given set
  Arrays.sort(S);
  // generate subsets
  ArrayList<ArrayList<Integer>> results = new ArrayList<ArrayList<Integer>>();
  results.add(new ArrayList<Integer>());
  for (int i=0; i<S.length; ++i) {
    int curSize = results.size();
    while (curSize-- > 0) {
      ArrayList<Integer> res = new ArrayList<Integer>(results.get(curSize));
      res.add(S[i]);
      results.add(res);
    }
  }
  return results;
}
```

Subsets - With Duplicates [http://leetcode.com/onlinejudge#question_90]

Given a collection of integers that might contain duplicates, S, return all possible subsets.
For example,
If S = [1,2,2,2], a solution is: [[],[1], [2], [1,2], [2,2], [1,2,2], [2,2,2], [1,2,2,2]]

Solution - Iteration

We can still use a similar idea from the problem above: append elements to previous subsets. The question is when to append and when not to.

Let's analysis the given example.
For the second 2, it will not be appended to [] or [1], but it will be appended to [2] and [1,2], which give us [2,2] and [1,2,2];
For the third 2, it will only be appended to [2,2] and [1,2,2].

What's the difference between those subsets?
When we hit a duplicate element, we only append it to subsets that contains all previous duplicates.

Therefore, we revise the previous iterative solution as follows:

```
public ArrayList<ArrayList<Integer>> subsetsWithDup(int[] num) {
  // sort the given set
  Arrays.sort(num);
  // generate subsets
  ArrayList<ArrayList<Integer>> results = new ArrayList<ArrayList<Integer>>();
  results.add(new ArrayList<Integer>());
  int count = 0;
  for (int i=0; i<num.length; ++i) {
    int curSize = results.size();
    // count duplicates
    if (i > 0 && num[i] == num[i-1]) {
      ++count;
    } else {
      count = 0;
    }
    // append to previous subsets
    while (curSize-- > 0) {
      ArrayList<Integer> res = new ArrayList<Integer>();
      ArrayList<Integer> pre = results.get(curSize);
      // for duplicates, only append to subsets containing all previous duplicates
      if (count > 0 && (pre.size() < count || pre.get(pre.size()-count) != num[i]))
        continue;
      res.addAll(pre);
      res.add(num[i]);
      results.add(res);
    }
  }
  return results;
}
```

Solution - Recursion

Recursive solution is more straightforward:

- Add [] to results
- Add each single distinct element to results and recursively append the rest of elements to each subset.

Note that when we add an ArrayList to ArrayList<ArrayList>, it only add the reference of the ArrayList. Thus, we need to make a copy of each pass-in subset. Otherwise, the final results will end up with [[], [], ..., []].

```
private ArrayList<ArrayList<Integer>> subsetsHelper(int[] S, int cur,
    ArrayList<Integer> path, ArrayList<ArrayList<Integer>> results) {
  results.add(path);
  for (int i=cur; i<S.length; ++i) {
    // skip duplicates
    if (i > cur && S[i] == S[i-1])
      continue;
    ArrayList<Integer> result = new ArrayList<Integer>(path);
    result.add(S[i]);
    results = subsetsHelper(S, i+1, result, results);
  }
  return results;
}

public ArrayList<ArrayList<Integer>> subsetsWithDup(int[] num) {
  // sort the given set
  Arrays.sort(num);
  // generate subsets
  return subsetsHelper(num, 0, new ArrayList<Integer>(), new ArrayList<ArrayList<Integer>>());
}
```

Posted 6th June 2013 by Sophie

Labels: Java, Recursion, Set

0 Add a comment

5th June 2013

ZigZag Conversion

ZigZag Conversion [http://leetcode.com/onlinejudge#question_6]

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

```
P   A   H   N
A P L S I I G
Y   I   R
```

And then read line by line: "PAHNAPLSIIGYIR"
Write the code that will take a string and make this conversion given a number of rows:

`string convert(string text, int nRows);`
`convert("PAYPALISHIRING", 3)` should return "PAHNAPLSIIGYIR".

Solution

A straightforward solution is to actually build up the zigzag character matrix and read the matrix into a string. But that will take extra O(nRows*Max_column_in_a_row) = O(n) spaces.

Another way is to calculate the corresponding index on the fly and append the character into a string.

Take `convert("PAYPALISHIRING", 4)` as an example.
The matrix will look as follows.

```
P   I   N
A L S I G
Y A H R
P   I
```

which can be expended as below (You will see why we rewrite like this soon.).

```
P   I   N
A   S   G
Y   H
P   I
A   R
L   I
```

Some useful properties:

- For any full columns, the odd ones have nRows characters, even ones have (nRows - 2) characters.
- For the first and last rows, we read one single character from each expended column;
- For the rest of rows, we read two characters, one from top part and one from bottom part, from each expended column.

One edge case is that nRows = 1, where (nRows*2 - 2) becomes 0. In this case, we should simply return the original string.

```
public String convert(String s, int nRows) {
  if (nRows == 1) return s;

  StringBuilder ss = new StringBuilder();
  int n = nRows + nRows - 2;
  // rest rows
  for (int i = 0; i < nRows; ++i) {
    int cur = i;
    while (cur < s.length()) {
      ss.append(s.charAt(cur));
      cur += n;
      if (i > 0 && i < nRows - 1 && (cur - i - i) < s.length()) {
        ss.append(s.charAt(cur - i - i));
      }
    }
  }
  return ss.toString();
}
```

Posted 5th June 2013 by Sophie

Labels: Array, Java, String, StringBuilder

0 Add a comment

4th June 2013

String to Integer (atoi)

String to Integer (atoi) [http://leetcode.com/onlinejudge#question_8]

Implement atoi to convert a string to an integer.

Solution

Before jumping into coding, we need to clarify the requirements of the problem.

- Does input string have whitespace?
  Possibly. It may start from whitespace characters and those characters can be ignored.
- What if the input string has other invalid characters?
  Valid characters for an integer could be '+', '-', '0' - '9', and whitespace according to previous question.
  Here you can assume the first non-whitespace character in the input string must be plus, minus, or a numeric digit, which is followed by as many numeric digits as possible. The input string may have invalid characters after those digits and the invalid characters can be ignored.
- What if the input string is not valid? What would be the expected behavior, throwing out an exception or return some specific value?
  If the input string is invalid, return zero.
- What if the resulting value is out of the range of Integer?
  Return INT_MAX (2147483647) or INT_MIN (-2147483648) as needed.

Okay. So now we know the input string is in the following pattern:

> ^\\s*[+-]?[0-9]+.*$

which starts with zero or more whitespace, follows by an optional plus/minus sign, several numeric digits, and zero or more other characters.

If we are allowed to use Java regex [http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html] (FYI, here is a tutorial of Java Regex [http://www.vogella.com/articles/JavaRegularExpressions/article.html]), then we can remove leading whitespace and trailing invalid characters using `replaceAll` method of String class.

```
public int atoi(String str) {
  // use regex to validate the input
  String pattern = "(^\\s*)([+-]?[0-9]+)(.*$)";
  if (!str.matches(pattern)) return 0;
  // trim the string to [+-]?[0-9]+
  String numstr = str.replaceAll(pattern, "$2");
  // convert string to integer
  int sign = (numstr.charAt(0) == '-') ? -1 : 1;
  long res = 0;
  for (int i = (numstr.charAt(0) == '+' || numstr.charAt(0) == '-') ? 1 : 0; i<numstr.length(); ++i) {
    res = res * 10 + (numstr.charAt(i) - '0');
    if (sign > 0 && res > Integer.MAX_VALUE) return Integer.MAX_VALUE;
    else if (sign < 0 && res * (-1) < Integer.MIN_VALUE) return Integer.MIN_VALUE;
  }
  return (int)res*sign;
}
```

This algorithm runs in time O(n) where n is the length of the input string.

If we are not allowed to use `replaceAll` method, we need to iterate the input string character by character.

```
public int atoi(String str) {
  long res = 0; int index = 0;
  // trim leading whitespace
  while (index < str.length() && str.charAt(index) == ' ') ++index;
  if (index == str.length()) return (int)res;
  // get sign
  int sign = 1;
  if (str.charAt(index) == '-') {
    ++index;
    sign = -1;
  } else if (str.charAt(index) == '+') {
    ++index;
  }

  // convert digits
  while (index < str.length() && Character.isDigit(str.charAt(index))) {
    res = res * 10 + (str.charAt(index++) - '0');
    if (sign > 0 && res > Integer.MAX_VALUE) return Integer.MAX_VALUE;
    else if (sign < 0 && res * (-1) < Integer.MIN_VALUE) return Integer.MIN_VALUE;
  }
  return (int)res*sign;
}
```

Posted 4th June 2013 by Sophie

Labels: Java, Regex, String

1 View comments

3rd June 2013

sqrt(x)

sqrt(x) [http://leetcode.com/onlinejudge#question_69]

Implement int sqrt(int x).
Compute and return the square root of x.

Solution - Binary Search

If you don't remember Newton's method [http://en.wikipedia.org/wiki/Newton%27s_method] , then this problem can be solved using Binary Search.
Note that the goal is not to find the exact square root r but to find the floor(r). So we terminate the loop when narrowing down the range to 1.

Notice that we calculate x/mid rather than mid*mid to avoid data overflow.

```
public int sqrt(int x) {
  if (x < 2) return x;

  int low = 0, high = x;
  while (high - low > 1) {
    int mid = low + (high - low) / 2;
    int div = x / mid;
    if (mid == div) return mid;
    if (mid < div) low = mid;
    else high = mid;
  }

  return low;
}
```

This algorithm runs in time O(logx).

Solution - Babylonian method [http://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Babylonian_method]

The key idea of Babylonian method is that if root is an underestimate of x, then x/root must be an overestimate and the average of the two may reasonably provide a better approximation.

Babylonian method is derived from Newton's method [http://en.wikipedia.org/wiki/Newton%27s_method] as shown below.

To find the square root of S, it is equivalent to find an x such that f(x) = x^2 - S = 0. And thus, Newton's method uses f(x) and its derivative [http://en.wikipedia.org/wiki/Derivative] to approach a better approximation iteratively.

[http://upload.wikimedia.org/math/5/0/4/504e3fd1368a623d05c22142471eb697.png]

The process is as follows:

- Let r_0 be an initial guess.
- r_(i+1) = (r_i + x / r_i) / 2 for abs(r_(i+1) - r_i) >= e), where e is desired approximation.

To reduce the converge iterations, we use a rough estimation [http://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Rough_estimation] of 2^(floor(d/2)) as the initial estimate, where d is the number of binary digits of x.

```
public int sqrt(int x) {
  // initial guess, x_0 = 2^(D/2)
  int root = 1;
  for (int digit = 0, origin = x; origin > 1; origin >>= 1, ++digit, root <<= (digit & 1));

  // calculate root
  double r0 = 0, r1 = 1.0*root;
  while (Math.abs(r1 - r0) > 0.01) {
    r0 = r1;
    r1 = (r0 + x/r0) / 2;
  }
  return (int)r1;
}
```

If the initial guess can be found in constant time, this algorithm runs in time O(loglogn).

There are several root-finding algorithms. Here [http://www.codeproject.com/Articles/69941/Best-Square-Root-Method-Algorithm-Function-Precisi] are comparisons and contrasts of 14 algorithms. Most of them are variants of Newton's method or Babylonian's method. Some of them either use magic numbers (e.g. the famous Quake3 magic number [http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf] 0x5f3759df) or depend on IEEE representation [http://en.wikipedia.org/wiki/IEEE_floating-point_standard] , which are unlikely to come up with during an interview...

Posted 3rd June 2013 by Sophie

Labels: BinarySearch, Java, Maths

0 Add a comment

3rd June 2013

Sort Colors

Sort Colors [http://leetcode.com/onlinejudge#question_75]

Given an array with n objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

Note:
You are not suppose to use the library's sort function for this problem.

Solution

A rather straight forward solution is a two-pass algorithm using counting sort.

- Iterate the array counting number of 0's, 1's, and 2's
- Overwrite array with total number of 0's, then 1's and followed by 2's.

```
private static final int RED = 0;
private static final int WHITE = 1;
private static final int BLUE = 2;

public void sortColors(int[] colors) {
  // one-pass for counting
  int[] count = new int[3];
  for (int color : colors) ++count[color];
  count[WHITE] += count[RED];

  // fill up the array
  int i = 0;
  for (; i < count[RED]; ++i) colors[i] = RED;
  for (; i < count[WHITE]; ++i) colors[i] = WHITE;
  for (; i < colors.length; ++i) colors[i] = BLUE;
}
```

This algorithm runs in time O(n) and uses O(1) spaces.

Can we improve it a little bit? Can we solve it with one-pass? Although it won't change the order of time complexity, it will practically reduce the running time by half.

If there are only two colors, say red and blue, we can use two pointers iterating the array from the two ends.

- Forward the left pointer if it points to a red, and forward the right pointer if it points to a blue;
- Swap the two colors if left points to a blue and right points to a red;
- Repeat until left > right.

Now, we have three colors. How can we handle them? We still need a current pointer to iterate the array. We also need two pointers pointing to the next position for a red and a blue, respectively.

- If the cur pointer points to a red, swap the values of cur and red, and forward cur and red pointers;
- If the cur pointer points to a blue, swap the values of cur and blue, and forward blue pointers (Do not forward cur pointer yet since it might be a red now.);
- If the cur pointer points to a white, keep going;
- Repeat until cur > blue.

Note that a hidden characteristic is that whatever are left between red and cur when the loop finishes must be white.

```
private static final int RED = 0;
private static final int WHITE = 1;
private static final int BLUE = 2;

private void swap(int[] A, int l, int r) {
  int temp = A[l];
  A[l] = A[r];
  A[r] = temp;
}

public void sortColors(int[] colors) {
  // initialize cur pointers
  int cur = 0;
  // red is the next position for reds; blue is the next position for blues
  int red = 0, blue = colors.length - 1;
  //
  while (cur <= blue) {
    if (colors[cur] == RED) {
      swap(colors, cur++, red++);
    } else if (colors[cur] == BLUE) {
      swap(colors, cur, blue--);
    } else {
      ++cur;
    }
  }
}
```

Posted 3rd June 2013 by Sophie

Labels: Array, Java, Sort

1 View comments

2nd June 2013

Simplify Path

Simplify Path [http://leetcode.com/onlinejudge#question_71]

Given an absolute path for a file (Unix-style), simplify it.

For example,
path = "/home/", => "/home"
path = "/a/./b/../../c/", => "/c"

Solution - Stack

The rule of Unix-style [http://en.wikipedia.org/wiki/Path_(computing)#Unix_style] path includes:

- An absolute path always starts from the root directory, i.e. "/".
- "." represents the current directory itself.
- ".." goes upwards to the parent directory if exists. E.g. "/a/b/.." == "/a", while "/../" == "/".
- Redundant slashes can be ignored. E.g. "/a///b" == "/a/b"

A straightforward way to solve this problem is

- Split the path into an array of substrings, each representing one level of the hierarchy.
- Go through the splits and push the "valid" substring into a stack: skip "." and redundant slashes. If hits a "..", pop out the top from the stack.
- Assemble the substrings in the stack to a path and that is the result.

```
public String simplifyPath(String path) {
  if (path == null || path.charAt(0) != '/') return null;

  // split by "/"
  String[] splits = path.split("/");

  // simplify
  Stack stack = new Stack();
  for (String split : splits) {
    if (split.equals("..") && !stack.isEmpty()) stack.pop();
    else if (!split.equals(".") && !split.equals("..") && !split.isEmpty()) {
      stack.push(split);
    }
  }

  // print new path
  if (stack.isEmpty()) return "/";
  StringBuilder sb = new StringBuilder();
  while (!stack.isEmpty()) {
    sb.insert(0, "/" + stack.pop());
  }
  return sb.toString();
}
```

We go through the path and its splits once, which takes time O(n). We store the splits in an array and then copy them over to a stack, which takes O(n) spaces. So, in total, this algorithm runs in time O(n) and uses O(n) space.

Solution - No Stack

Alternatively, we can get there without the stack by editing the array on the fly.
To do that, we need to maintain a pointer pointing to the last which is the spot after the last valid substring. As iterating through the array, if the current string is "valid", copy over it to the last and forward the last pointer. If current string is "..", move back last pointer by 1.

```
public String simplifyPath(String path) {
  if (path == null || path.charAt(0) != '/') return null;

  // split by "/"
  String[] splits = path.split("/");

  // simplify
  int last = 0;
  for (String split : splits) {
    if (split.equals("..") && last > 0) --last;
    else if (!split.equals(".") && !split.equals("..") && !split.isEmpty()) {
      splits[last++] = split;
    }
  }

  // print new path
  if (last == 0) return "/";
  StringBuilder sb = new StringBuilder();
  for (int i=0; i<last; ++i) {
    sb.append("/" + splits[i]);
  }
  return sb.toString();
}
```

The running time of this algorithm is the same as that of above, O(n).Although we removed the stack, we still store the splits in an array, the space complexity is still O(n).

Solution - In Place

We can also solve the problem in place, i.e. without extra spaces.
Note that the input has to be a char array since String in Java is immutable. That says, if the given input is a String, we have to create a char array for it anyway so as to modify the values.

The basic idea is the same except that we cannot use the String class method split.

- Go through path char by char. If hitting a delimiter or get to the end of array,
  - If the current split is "/..", back up one level.
  - If the current split is neither "/." nor "/", copy it to the end of previous valid split.

```
public String simplifyPath(char[] path) {
  if (path == null || path[0] != '/') return null;

  // last points to the end of previous valid split
  // start points to "/..." of current split
  char[] path = path.toCharArray();
  int last = 0, start = 0, cur = 0;
  for (; cur<path.length; ++cur) {
    if (path[cur] == '/') { // hit a delimiter
      if (cur-start == 3 && path[cur-1] == '.' && path[cur-2] == '.') { // back up
        while (last > 0 && path[--last] != '/');
      } else if (cur - start > 2 || (cur - start == 2 && path[cur-1] != '.')) {
        while (start < cur)  path[last++] = path[start++];
      }
      // reset start
      start = cur;
    }
  }
  // check the last split
  if (cur > start) {
    if (cur-start == 3 && path[cur-1] == '.' && path[cur-2] == '.') { // back up
      while (last > 0 && path[--last] != '/');
    } else if (cur - start > 2 || (cur - start == 2 && path[cur-1] != '.')) {
      while (start < cur) path[last++] = path[start++];
    }
  }

  if (last == 0) path[last++] = '/';
  return new String(path, 0, last);
}
```

Posted 2nd June 2013 by Sophie

Labels: Array, Stack, String, StringBuilder

0 Add a comment

28th May 2013

Search in Rotated Sorted Array

Search in Rotated Sorted Array - No Duplicates [http://leetcode.com/onlinejudge#question_33]

Suppose a sorted array is rotated at some pivot unknown to you beforehand.
(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Given a target value to search, return its index if found in the array, otherwise return -1.

You may assume no duplicate exists in the array.

Solution

We still use binary search to find the target. There are three cases:

- [1 2 3 4 5]
  That says, the rotate point is out of the range [low..high].
  In this case, we still use the classic binary search algorithm to move low or high pointers.
- [2 3 4 5 1]
  That says, the rotate point is in the range [low..mid).
  In this case, target < A[mid] is not sufficient to tell us to move the left half. Instead, only if A[low] <= target < A[mid], move to the left half; otherwise, move to the right half.
- [4 5 1 2 3]
  That says, the rotate point is in the rage (mid..high]. Similarly, only if A[mid] < target <= A[high], move to the right half; otherwise, move to the left half.

```
1:  public int search(int[] A, int target) {
2:    int low = 0, high = A.length - 1;
3:    while (low <= high) {
4:      int mid = low + (high - low) / 2;
5:      if (A[mid] == target) return mid;
6:
7:      // rotate point is out of [low..high]
8:      if ((A[low] < A[high] && A[mid] < target)
9:          // rotate point is in [low..mid)
10:          || (A[mid] < A[high] && target > A[mid] && target <= A[high])
11:          // rotate point is in (mid..high]
12:          || (A[mid] > A[high] && (target > A[mid] || target <= A[high]))) {
13:        low = mid + 1;
14:      } else {
15:        high = mid - 1;
16:      }
17:    }
18:
19:    // not found
20:    return -1;
21:  }
```

We can make the code simpler.

There are actually two cases:

- Rotate point is not in the right half.
- Rotate point is in the right half.

Then line 7-17 can be replaced with

```
     // rotate point is NOT in [low..mid)
     if ((A[mid] < A[high] && target > A[mid] && target <= A[high])
         // rotate point is in (mid..high]
         || (A[mid] > A[high] && (target > A[mid] || target <= A[high]))) {
       low = mid + 1;
     } else {
       high = mid - 1;
     }
   }
```

These two algorithms run in O(logn) time and use O(1) space.

Search in Rotated Sorted Array - Has Duplicates [http://leetcode.com/onlinejudge#question_81]

Follow up for "Search in Rotated Sorted Array":
What if duplicates are allowed? Would this affect the run-time complexity? How and why?

Write a function to determine if a given target is in the array.

Solution

When the array has duplicates, comparing A[low], A[mid] and A[high] cannot tell the difference between [1 1 1 1 2 1] and [1 2 1 1 1 1].

We have to filter out the duplicates and then apply the binary search on the remaining elements of the array.

To filter out the duplicates, either we check duplicates from one end every time when we move low or high pointer and remove them, or we check duplicates from both ends and cleanup the array until at least one end has no duplicates (as shown in algorithm below). Unfortunately, the worst case running time of both two methods is O(n). Thus the total running time becomes O(n).

```
public boolean search(int[] A, int target) {
  int low = 0, high = A.length - 1;

  // filter the duplicates
  while (low < high && A[low] == A[high]) {
    if (A[low] == target) return true;
    ++low; --high;
  }

  // find the target in non-duplicates
  while (low <= high) {
    int mid = (low + high) / 2;
    if (A[mid] == target) return true;

    // rotate point is NOT in [low..mid]
    if ((A[mid] < A[high] && target > A[mid] && target <= A[high])
        // rotate point is in [mid..high]
        || (A[mid] > A[high] && (target > A[mid] || target <= A[high]))) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }

  // not found
  return false;
}
```

Posted 28th May 2013 by Sophie

Labels: Array, BinarySearch, Java

2 View comments

26th May 2013

Search In Sorted Array

Search Insert Position [http://leetcode.com/onlinejudge#question_35]

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You may assume no duplicates in the array.

Here are few examples.
[1,3,5,6], 5 → 2
[1,3,5,6], 2 → 1
[1,3,5,6], 7 → 4
[1,3,5,6], 0 → 0

Solution

This is a typical Binary Search problem.

The only tricky part is if the target is not found, which value shall we return, low or high?
In binary search, if the middle is smaller than the target, we forward low to be mid+1. That tells us, when the loop terminates, either we found the target or the resulted low would be greater than the target.

```
public int searchInsert(int[] A, int target) {
  int low = 0, high = A.length - 1;
  while (low <= high) {
    int mid = (low + high) / 2;
    if (A[mid] == target) return mid;
    if (A[mid] < target) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }
  return low;
}
```

This algorithm runs in time O(logn) and takes O(1) space.

Search for a Range [http://leetcode.com/onlinejudge#question_34]

Given a sorted array of integers, find the starting and ending position of a given target value.
If the target is not found in the array, return [-1, -1].

Your algorithm's runtime complexity must be in the order of O(log n).

For example,
Given [5, 7, 7, 8, 8, 10] and target value 8,
return [3, 4].

Solution

This is a variant of the binary search.

Basically, what we need to do here is to apply binary search on the entire array to find the beginning of the range, and perform another binary search on the remaining of the array to find the end of the range.

We can put the two searches in the same while loop by reset the low and high pointer when we hits one end of the range.

```
public int[] searchRange(int[] A, int target) {
  int[] range = {-1, -1};

  int low = 0, high = A.length - 1;
  while (low <= high) {
    int mid = (low + high) / 2;
    if (A[mid] == target) {
      if (mid == 0 || A[mid-1] < target) {
        // find the beginning
        range[0] = mid;
        // reset
        low = mid; high = A.length - 1;
      }
      if (mid == A.length-1 || A[mid+1] > target) {
        // find the end
        range[1] = mid;
        // reset
        low = 0; high = mid;
      }
      // find the range
      if (range[0] > -1 && range[1] > -1) return range;

      if (range[0] < 0) high = mid - 1;  // continue searching for beginning in (.., mid)
      else low = mid + 1;  // continue searching for end in (mid, ..)
    } else if (A[mid] < target) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }

  return range;
}
```

This algorithm runs in time O(logn) and takes O(1) space.

Posted 26th May 2013 by Sophie

Labels: Array, BinarySearch, Java

0 Add a comment

26th May 2013

Scramble String

Scramble String [http://leetcode.com/onlinejudge#question_87]

Given two strings s1 and s2 of the same length, determine if s2 is a scrambled string of s1.

"A scrambled string"

Given a string s1, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.

Below is one possible representation of s1 = "great":

```
    great
   /    \
  gr    eat
 / \    /  \
g   r  e   at
           / \
          a   t
```

To scramble the string, we may choose any non-leaf node and swap its two children.

For example, if we choose the node "gr" and swap its two children, it produces a scrambled string "rgeat".

```
    rgeat
   /    \
  rg    eat
 / \    /  \
r   g  e   at
           / \
          a   t
```

We say that "rgeat" is a scrambled string of "great".

Similarly, if we continue to swap the children of nodes "eat" and "at", it produces a scrambled string "rgtae".

```
    rgtae
   /    \
  rg    tae
 / \    /  \
r   g  ta  e
       / \
      t   a
```

We say that "rgtae" is a scrambled string of "great".

Solution - Recursion

Intuitively, we can solve it with recursion.

Given two strings, try every possible split of s1 and check whether the resulted substrings of s1 equal or "swap-equal" of the corresponding substrings of s2.

For example, given "great" and "rgate", the splits are

- ("great") vs. ("rgate")
- ("g", "reat") vs. ("r", "gate") or ("rgat", "e")
- ("gr", "eat") vs. ("rg", "ate") or ("te", "rga")
- ("gre", "at") vs. ("rga", "te") or ("ate", "rg")
- ("grea", "t") vs. ("rgat", "e") or ("gate", "r")

```
public boolean isScramble(String s1, String s2) {
  int len = s1.length();
  if (len != s2.length()) return false;
  if (s1.equals(s2)) return true;

  for (int i=1; i < len; ++i) {
    String s1l = s1.substring(0, i), s1r = s1.substring(i, len);
    // w/o swap
    String s2l = s2.substring(0, i), s2r = s2.substring(i, len);
    if (isScramble(s1l, s2l) && isScramble(s1r, s2r)) return true;
    // w/ swap
    s2l = s2.substring(0, len-i); s2r = s2.substring(len-i, len);
    if (isScramble(s1l, s2r) && isScramble(s1r, s2l)) return true;
  }

  return false;
}
```

There are O(n) possible split points. At each point, there are two possibilities: with swap and without swap. At each point, we recursively check the two substrings until both of them are single character, which takes time O(2^k+2^(n-k)), where k and n-k are the lengths of the two substrings. So, this algorithm runs in exponential time, O(2^n), and space complexity is also exponential (recursion stack).

Solution - Recursion w/ Backtracking

We have discussed several algorithms whose performance can be improved using Backtracking.
Unfortunately, for this problem, backtracking can improve the time but the order of the time is still exponential.

```
private boolean isScrambleHelper(String s1, String s2, HashMap<String, String> map) {
  int len = s1.length();
  if (len != s2.length()) return false;
  if (s1.equals(s2) || s2.equals(map.get(s1))) return true;

  for (int i=1; i < len; ++i) {
    String s1l = s1.substring(0, i), s1r = s1.substring(i, len);
    // w/o swap
    String s2l = s2.substring(0, i), s2r = s2.substring(i, len);
    if (isScramble(s1l, s2l) && isScramble(s1r, s2r)) {
      map.put(s1l, s2l); map.put(s1r, s2r);
      return true;
    }
    // w/ swap
    s2l = s2.substring(0, len-i); s2r = s2.substring(len-i, len);
    if (isScramble(s1l, s2r) && isScramble(s1r, s2l)) {
      map.put(s1l, s2r); map.put(s1r, s2l);
      return true;
    }
  }

  return false;
}

public boolean isScramble(String s1, String s2) {
  return isScrambleHelper(s1, s2, new HashMap<String, String>());
}
```

Solution - DP

In the recursion solution, we split the strings and compare recursively. It ends up with comparing every possible pair of equal-length substrings in the two strings. So, suppose the length of the two strings is n, the subproblems are

- For each pair of (n-1)-char-long substrings of the two strings, are they scramble to each other?
- For each pair of (n-2)-char-long substrings, are they scramble?
- ... ...
- For each pair of 2-char-long substrings, are they scramble?
- For each pair of char in the two strings, are they scramble (i.e. do they equal)?

That is saying, we can build up a table and solve the problem in a bottom-up fashion.

Then to verify whether two k-char-long strings are scramble, we still split the two strings at k-1 position. For each pair of split, we don't need to take O(n) time to recursively compare the two of them. Instead, simply take O(1) time to look at the table.

This gives us a O(n^4)-time solution which uses O(n^4) spaces.

```
public boolean isScramble(String s1, String s2) {
  int len = s1.length();
  if (len != s2.length()) return false;
  if (s1.equals(s2)) return true;

  // a table of matches
  // T[i][j][k] = true iff s2.substring(j,j+k+1) is a scambled string of s1.substring(i,i+k+1)
  boolean[][][] scrambled = new boolean[len][len][len];
  for (int i=0; i < len; ++i) {
    for (int j=0; j < len; ++j) {
      scrambled[i][j][0] = (s1.charAt(i) == s2.charAt(j));
    }
  }

  // dynamically fill up the table
  for (int k=1; k < len; ++k) { // k: length
    for (int i=0; i < len - k; ++i) { // i: index in s1
      for (int j=0; j < len - k; ++j) { // j: index in s2
        scrambled[i][j][k] = false;
        for (int p=0; p < k; ++p) { // p: split into [0..p] and [p+1..k]
          if ((scrambled[i][j][p] && scrambled[i+p+1][j+p+1][k-p-1])
              || (scrambled[i][j+k-p][p] && scrambled[i+p+1][j][k-p-1])) {
            scrambled[i][j][k] = true;
            break;
          }
        }

      }
    }
  }

  return scrambled[0][0][len-1];
}
```

Posted 26th May 2013 by Sophie

Labels: Backtracking, DP, Java, Recursion

4 View comments

22nd May 2013

Search a 2D Matrix

Search a 2D Matrix [http://leetcode.com/onlinejudge#question_74]

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

- Integers in each row are sorted from left to right.
- The first integer of each row is greater than the last integer of the previous row.

For example, consider the following matrix:
[
[1, 3, 5, 7],
[10, 11, 16, 20],
[23, 30, 34, 50]
]
Given target = 3, return true.

Solution

If we start with 1D sorted array, how to find a target efficiently?
I bet your answer would be Binary Search [http://en.wikipedia.org/wiki/Binary_search] . (If not, you need to spend some time to review your algorithm textbook, like CLRS <Introduction to Algorithms [http://www.amazon.com/gp/product/0262033844/ref=as_li_ss_il?ie=UTF8&camp=1789&creative=390957&creativeASIN=0262033844&linkCode=as2&tag=n00tc0d3r-20] >. :)

Now, given a sorted 2D array, how to find a target?
Yes, we still want to use binary search, with some modifications of course.

One way to solve it is to use two binary searches:

- One searches for the row that may contain the target.
- The other searches for the target in that row.

The latter one is a typical binary search while the former one for the row needs some small revises.
The general idea is the same:

- If the mid number (/row) is the target, return there;
- If target is less than mid, move to [low, mid);
- If target is greater than mid, move to (mid, high).
- Repeat until find the target or fail (where low meets high).

When the target becomes a row, we also need to compare to the first element of next row (row mid+1), or the last element of current row (row mid), to determine whether the current row is what we are looking for.

Note: If you are not familiar with Binary Search, I highly recommend you to try to write down the code by yourself. The algorithm is not easy to write correctly although it looks simple. After finishing coding, you can use an array with 3 elements to test your code, e.g. [x, target, y], [target, x, y], [x, y, target].

```
public boolean searchMatrix(int[][] matrix, int target) {
  // binary search to find the row
  int low = 0, high = matrix.length - 1;
  while (low < high) {
    int mid = (low + high) / 2;
    if (target == matrix[mid][0]) return true;
    else if (target < matrix[mid][0]) high = mid - 1;
    else if (target < matrix[mid+1][0]) { low = mid; break; }
    else low = mid + 1;
  }

  // binary search to find the target
  int row = low;
  low = 0; high = matrix[row].length - 1;
  while (low <= high) {
    int mid = (low + high) / 2;
    if (target == matrix[row][mid]) return true;
    else if (target < matrix[row][mid]) high = mid - 1;
    else low = mid + 1;
  }

  return false;
}
```

Another way to solve the problem is to think of the 2D matrix as a sorted 1D array and then apply the regular binary search on the 1D array. After computing the mid value, we calculate the corresponding row and column numbers based on mid.

```
public boolean searchMatrix(int[][] matrix, int target) {
  // treat the 2D matrix as a sorted 1D array and use binary search to find the target
  int low = 0, high = matrix.length * matrix[0].length - 1;
  while (low <= high) {
    int mid = (low + high) / 2;
    int row = mid / matrix[0].length, col = mid % matrix[0].length;
    if (target == matrix[row][col]) return true;
    else if (target < matrix[row][col]) high = mid - 1;
    else low = mid + 1;
  }

  return false;
}
```

Posted 22nd May 2013 by Sophie

Labels: BinarySearch, Java, Matrix

1 View comments

21st May 2013

Tree Comparison

Same Tree [http://leetcode.com/onlinejudge#question_100]

Given two binary trees, write a function to check if they are equal or not.

Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

Solution - Recursion

A recursive solution is quite straightforward.

```
public boolean isSameTree(TreeNode p, TreeNode q) {
  if (p == null && q == null) return true;
  if (p == null || q == null || p.val != q.val) return false;
  return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
}
```

This runs in time O(n) and on average use O(logn) space for recursion stacks. The worst case space complexity is O(n) where the height of binary tree is n.

Solution - Iteration

We can use BFS to traverse the two trees and perform comparisons.

Here we use ArrayDeque [http://docs.oracle.com/javase/6/docs/api/java/util/ArrayDeque.html] as the implementation of Queue, which is faster than LinkedList [http://docs.oracle.com/javase/6/docs/api/java/util/LinkedList.html] implementation.
ArrayDeque is a resizable array and most of its operations (such as `add(E e)`, `offer(E e)`, `remove()`, `poll()`, `element()`, `getFirst()`, `getLast()`, `peek()`, `peekFirst()`, `peekLast()`, etc.) run in amortized constant time.
c.f. For LinkedList, remove() and poll() takes O(1) time but add() and oꞔer() takes O(n) time.

Another option is to use two ArrayList [http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html] s and maintain pointers moving from the beginning to the end. Add() operation in ArrayList takes O(1) time. Maintaining a pointer is to avoid unnecessary remove()'s, each of which takes time O(n).

```
public boolean isSameTree(TreeNode p, TreeNode q) {
  if (p == null && q == null) return true;
  if (p == null || q == null || p.val != q.val) return false;

  Queue<TreeNode> pque = new ArrayDeque<TreeNode>();
  Queue<TreeNode> qque = new ArrayDeque<TreeNode>();
  pque.add(p); qque.add(q);
  while (!pque.isEmpty() && !qque.isEmpty()) {
    TreeNode pp = pque.remove();
    TreeNode qq = qque.remove();

    if (pp.left != null && qq.left != null && pp.left.val == qq.left.val) {
      pque.add(pp.left); qque.add(qq.left);
    } else if (!(pp.left == null && qq.left == null)) {
      return false;
    }

    if (pp.right != null && qq.right != null && pp.right.val == qq.right.val) {
      pque.add(pp.right); qque.add(qq.right);
    } else if (!(pp.right == null && qq.right == null)) {
      return false;
    }
  }

  return (pque.isEmpty() && qque.isEmpty());
}
```

This algorithm runs in time O(n) since we visit each node once. It takes O(n) space if the binary tree is almost-full since it stores nodes level by level.

Symmetric Tree [http://leetcode.com/onlinejudge#question_101]

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).
For example, this binary tree is symmetric:

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

But the following is not:

```
    1
   / \
  2   2
   \   \
   3    3
```

Solution - Recursion

This problem is similar to the above problem. The difference is that rather than comparing two trees, here we compare the left and right child subtrees and compare them in a symmetric way (i.e. compare left with right and vice versa).

```
private boolean isSymmetric(TreeNode left, TreeNode right) {
  if (left == null && right == null) return true;
  if (left == null || right == null || left.val != right.val) return false;
  return isSymmetric(left.left, right.right) && isSymmetric(left.right, right.left);
}

public boolean isSymmetric(TreeNode root) {
  return (root == null) || isSymmetric(root.left, root.right);
}
```

Solution - Iteration

Similarly, we can derive an iterative solution based on the iterative solution for the Same Tree problem.

```
private boolean isSymmetric(TreeNode left, TreeNode right) {
  if (left == null && right == null) return true;
  if (left == null || right == null || left.val != right.val) return false;

  Queue<TreeNode> lque = new ArrayDeque<TreeNode>();
  Queue<TreeNode> rque = new ArrayDeque<TreeNode>();
  lque.add(left); rque.add(right);
  while (!lque.isEmpty() && !rque.isEmpty()) {
    TreeNode l = lque.remove();
    TreeNode r = rque.remove();

    if (l.left != null && r.right != null && l.left.val == r.right.val) {
      lque.add(l.left); rque.add(r.right);
    } else if (!(l.left == null && r.right == null)) {
      return false;
    }

    if (l.right != null && r.left != null && l.right.val == r.left.val) {
      lque.add(l.right); rque.add(r.left);
    } else if (!(l.right == null && r.left == null)) {
      return false;
    }
  }

  return (lque.isEmpty() && rque.isEmpty());
}

public boolean isSymmetric(TreeNode root) {
  return (root == null ) || isSymmetric(root.left, root.right);
}
```

Posted 21st May 2013 by Sophie

Labels: BFS, BinaryTree, Java, Recursion, Tree

2 View comments

20th May 2013

Set Matrix to Zeroes

Set Matrix to Zeroes [http://leetcode.com/onlinejudge#question_73]

Given a m x n matrix, if an element is 0, set its entire row and column to 0. Do it in place.

Solution

As we go through the matrix, if we find a zero, we cannot set its row and column to zeroes yet. If we do that, we would not know whether or not to set the next row and next column to zeroes.

So, we iterate through the matrix twice.

- In the first iteration, we find out all of the zeroes and store their row and column numbers.
- In the next iteration, we set a cell to zero if it is on a row or a column that we stored.

```
public void setZeroes(int[][] matrix) {
  Set<Integer> rows = new HashSet<Integer>();
  Set<Integer> columns = new HashSet<Integer>();

  // find zeroes
  for (int i = 0; i < matrix.length; ++i) {
    for (int j = 0; j < matrix[0].length; ++j) {
      if (matrix[i][j] == 0) {
        rows.add(i);
        columns.add(j);
      }
    }
  }

  // set zeroes
  for (int i = 0; i < matrix.length; ++i) {
    for (int j = 0; j < matrix[0].length; ++j) {
      if (rows.contains(i) || columns.contains(j)) matrix[i][j] = 0;
    }
  }
}
```

Obviously, this algorithm runs in time O(m*n) and use O(m+n) spaces, where m and n are the number of rows and columns, respectively.

Can we do it in-place?

We have an entire matrix and thus we can use one row and one column of the matrix to store the zero information. Before we revise the values in that row and column, we need to know whether the original row/column contain zero. If so, we also need to set the row/column to zeros; If not, leave other values as they are.

```
public void setZeroes(int[][] matrix) {
  int rownum = matrix.length;
  if (rownum == 0) return;
  int colnum = matrix[0].length;
  if (colnum == 0) return;

  boolean hasZeroFirstRow = false, hasZeroFirstColumn = false;

  // Does first row have zero?
```

```
    for (int j = 0; j < colnum; ++j) {
      if (matrix[0][j] == 0) {
        hasZeroFirstRow = true;
        break;
      }
    }

    // Does first column have zero?
    for (int i = 0; i < rownum; ++i) {
      if (matrix[i][0] == 0) {
        hasZeroFirstColumn = true;
        break;
      }
    }

    // find zeroes and store the info in first row and column
    for (int i = 1; i < matrix.length; ++i) {
      for (int j = 1; j < matrix[0].length; ++j) {
        if (matrix[i][j] == 0) {
          matrix[i][0] = 0;
          matrix[0][j] = 0;
        }
      }
    }

    // set zeroes except the first row and column
    for (int i = 1; i < matrix.length; ++i) {
      for (int j = 1; j < matrix[0].length; ++j) {
        if (matrix[i][0] == 0 || matrix[0][j] == 0) matrix[i][j] = 0;
      }
    }

    // set zeroes for first row and column if needed
    if (hasZeroFirstRow) {
      for (int j = 0; j < colnum; ++j) {
        matrix[0][j] = 0;
      }
    }
    if (hasZeroFirstColumn) {
      for (int i = 0; i < rownum; ++i) {
        matrix[i][0] = 0;
      }
    }
  }
```

Although the code looks much longer than the previous one, the running time are the same: we iterate through the matrix twice which gives us O(m*n) time. And now the space complexity is O(1)!

Posted 20th May 2013 by Sophie

Labels: Array, Java, Matrix

0 Add a comment

20th May 2013

Rotate List

Rotate List [http://leetcode.com/onlinejudge#question_61]

Given a list, rotate the list to the right by k places, where k is non-negative.

For example:
Given 1->2->3->4->5->NULL and k = 2, return 4->5->1->2->3->NULL.
Given 1->2->3->4->5->NULL and k = 6, return 5->1->2->3->4->NULL.

Solution

There are two cases: k < length-of-list and k >= length-of-list.
In the case of k < length-of-list,

- use two pointers to find the k-to-the-end node,
- link the original tail to the original head,
- then cut the list before k-to-the-end node and make it the new head.

In the other case, again, we use two pointers to iterate through the list. When we hit the tail for the first time, link it to the original head to make the list into a circular linked list and then keep forwarding the pilot pointer until we find the k-to-the-end node.
Note that now the tail node is the one whose next is head (not null any more).

Put them together,

- Move the pilot pointer forward until it is k-node away from slow pointer
- Link the tail to the head if pilot hits the tail
- Move pilot and slow pointers together until pilot stops at the tail
- Cut the list at the link between slow pointer and its next and return the new head

```
public ListNode rotateRight(ListNode head, int k) {
  if (head == null || k <= 0 || head.next == null) return head;

  ListNode pre = head, end = head;
  // find the k-to-the-end
  while (end.next != head || k > 0) {
    if (k > 0) { // only forward end pointer
      --k;
    } else { // forward both pointers
      pre = pre.next;
    }
    end = end.next;
    // make it a circular linked list
    if (end.next == null) end.next = head;
  }

  ListNode newHead = pre.next;
  pre.next = null;

  return newHead;
}
```

In this algorithm, the pilot pointer has been forwarded at least k times and the two pointers were moved together to get to the k-to-the-end node. The worst case running time is O(k+n) where n is the length of the list.

If k >> n, we can improve the algorithm by first compute the length of the list. By doing this, the running time becomes O(2n) = O(n) since we iterate through the list twice: one for the length and the other for the k-to-the-end node.

Both algorithm use O(1) space.

```
public ListNode rotateRight(ListNode head, int k) {
  // find the length of the list
  int len = 0; ListNode cur = head;
  while (cur != null) {
    ++len;
    cur = cur.next;
  }

  if (len == 0 || k % len == 0) return head;
  k = k % len;

  ListNode pre = head; cur = head;
  // find the n-to-the-end
  while (cur.next != head) {
    if (k > 0) { // only forward end pointer
      --k;
    } else { // forward both pointers
      pre = pre.next;
    }
    cur = cur.next;
    // make it a circular linked list
    if (cur.next == null) cur.next = head;
  }

  ListNode newHead = pre.next;
  pre.next = null;

  return newHead;
}
```

Posted 20th May 2013 by Sophie

Labels: Java, LinkedList

0 Add a comment

20th May 2013

Rotate/Spiral Matrix and Variants

Rotate Square Matrix [http://leetcode.com/onlinejudge#question_48]

You are given an n x n 2D matrix representing an image.
Rotate the image by 90 degrees (clockwise) in-place.

Solution

[http://4.bp.blogspot.com/-DR5Jem2JseA/UZlEzsy_i0I/AAAAAAAAEfg/79uY-YRUjvY/s1600/Screen+Shot+2013-05-19+at+2.05.45+PM.png]

As shown in the picture, we need to move the value from lighter color to darker one, level by level and one by one. It is not difficult to get to the algorithm if you can draw this picture.

```
public void rotate(int[][] matrix) {
  for (int level = 0, len = matrix.length; level < len; ++level, --len) {
    int end = len - 1;
    for (int pos = level; pos < end; ++pos) {
      int tail = matrix.length - pos - 1;

      int tmp = matrix[level][pos];
      // left -> top
      matrix[level][pos] = matrix[tail][level];
      // bottom -> left
      matrix[tail][level] = matrix[end][tail];
      // right -> bottom
      matrix[end][tail] = matrix[pos][end];
      // top -> right
      matrix[pos][end] = tmp;
    }
  }
}
```

This algorithm runs in time O(n^2) and it is optimal since we have to visit each node at least once.

Spiral Matrix to Array [http://leetcode.com/onlinejudge#question_54]

Given a matrix of m x n elements (m rows, n columns), return all elements of the matrix in spiral order.

For example,
Given the following matrix:
[
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]
],
return [1,2,3,6,9,8,7,4,5].

Solution

This problem looks similar to the above one. But the difference is now the matrix may not be a square matrix.
Also, in the rotate problem, we don't need to rotate the center node but now we need to add te center node/row/column to the result.

A slightly modification on the above algorithm can make it work for this problem.

```
public ArrayList<Integer> spiralOrder(int[][] matrix) {
  ArrayList<Integer> spiral = new ArrayList<Integer>();
  if (matrix.length == 0 || matrix[0].length == 0) return spiral;

  for (int level = 0, m = matrix.length, n = matrix[0].length;
      m > level && n > level; ++level, --m, --n) {
    int right = n - 1;
    int bottom = m - 1;

    // top row
    for (int i = level; i <= right; ++i) spiral.add(matrix[level][i]);
    if (bottom == level) return spiral;
    // right column (from second to end)
    for (int i = level+1; i <= bottom; ++i) spiral.add(matrix[i][right]);
    if (right == level) return spiral;
    // bottom row
    for (int i = right-1; i >= level; --i) spiral.add(matrix[bottom][i]);
    // left column
    for (int i = bottom-1; i > level; --i) spiral.add(matrix[i][level]);
  }

  return spiral;
}
```

Generate Square Spiral Matrix [http://leetcode.com/onlinejudge#question_59]

Given an integer n, generate a square matrix filled with elements from 1 to n2 in spiral order.

For example,
Given n = 3, return the following matrix:

[
[1, 2, 3],
[8, 9, 4],
[7, 6, 5]
]

Solution

Same idea: use level and n to maintain lower and upper boundary and fill up the matrix level by level (top - right - bottom - left).

```
public int[][] generateMatrix(int n) {
  int val = 1;
  int[][] matrix = new int[n][n];

  for (int level = 0; level < n; ++level, --n) {
    // top
    for (int i=level; i < n; ++i) matrix[level][i] = val++;
    // right
    for (int i=level+1; i < n; ++i) matrix[i][n-1] = val++;
    // bottom
    for (int i=n-2; i >= level; --i) matrix[n-1][i] = val++;
    // left
    for (int i=n-2; i > level; --i) matrix[i][level] = val++;
  }

  return matrix;
}
```

Posted 20th May 2013 by Sophie

Labels: Array, Java, Matrix

2 View comments

19th May 2013

Reverse Linked List

Reverse Linked List in Range [http://leetcode.com/onlinejudge#question_92]

Reverse a linked list from position m to n. Do it in-place and in one-pass.

For example:
Given 1 -> 2 -> 3 -> 4 -> 5 -> |, m = 2 and n = 4, return 1 -> 4 -> 3 -> 2 -> 5 -> |.

Note: m and n are both 1-based and assume 1 <= m <= n <= list length.

Solution

This is a basic linked list question.

Add a dummy node to handle m = 1 case. After we hit the m-th node, we need to reverse the link between the current two nodes and also update the boundary nodes.

For example, given 1 -> 2 -> 3 -> 4 -> 5 -> |, m = 2 and n = 5, we let pre and cur always point to node 1 and node 2, but keep moving their next points and also update link between the two next nodes. It becomes

- pos = 2, nt = 4, pre.next = 2, cur.next = 3, and the list 1 -> 2 -> 3 -> 4 -> 5 -> |,
  3 -> 2,
  1 -> 3,
  2 -> 4,
  => 1 -> 3 -> 2 -> 4 -> 5 -> |
- pos = 3, nt = 5, pre.next = 3, cur.next = 4,
  4 -> 3,
  1 -> 4,
  2 -> 5,
  => 1 -> 4 -> 3 -> 2 -> 5 -> |
- pos = 4, nt = |, pre.next = 4, cur.next = 5,
  5 -> 4,
  1 -> 5,
  2 -> |,
  => 1 -> 5 -> 4 -> 3 -> 2 -> |

```
public ListNode reverseBetween(ListNode head, int m, int n) {
  ListNode dummy = new ListNode(0);
  dummy.next = head;

  // first if the first position, begin is the node before first.
  ListNode pre = dummy, cur=head;
  int pos = 1;

  // find the first
  while (pos < m && cur != null) {
    pre = cur;
    cur = cur.next;
    ++pos;
  }

  // reverse the list
  while (pos < n && cur != null) {
    ListNode nt = cur.next.next;
    cur.next.next = pre.next;
    pre.next = cur.next;
    cur.next = nt;
    ++pos;
  }

  return dummy.next;
}
```

This algorithm walks through the list in one-pass and thus its running time is O(n). It uses constant space.

Reverse Linked List in k-Groups [http://leetcode.com/onlinejudge#question_25]

Given a linked list, reverse every k nodes of a linked list and return its modified list.
If the number of nodes is not a multiple of k then left-out nodes in the end should remain as it is.

Note: You may not alter the values in the nodes, only nodes itself may be changed.
Only constant memory is allowed.

For example:
Given this linked list: 1->2->3->4->5,
For k = 2, you should return: 2->1->4->3->5
For k = 3, you should return: 3->2->1->4->5

Solution

This time we reverse the linked list in groups. Since we will not reverse the nodes unless they are in a k-group, we need to go through the list twice: one to determine groups and the other is to reverse each group when found such one. The above algorithm can be used as a subroutine to reverse one group.

The total running time here is still O(n) with O(1) space. But the actual running time is about twice of above one.

```
// reverse the linked list between pre.next and end, inclusively
// e.g. 0 -> 1 -> 2 -> 3 -> 4 -> 5, pre = 0, end = 5
//  => 0 -> 5 -> 4 -> 3 -> 2 -> 1
private ListNode reverse(ListNode pre, ListNode end) {
  if (pre == null || end == null) return end;
  ListNode cur = pre.next;
  while (pre.next != end) {
    ListNode nt = cur.next.next;
    cur.next.next = pre.next;
    pre.next = cur.next;
    cur.next = nt;
  }
  return cur;
}

public ListNode reverseKGroup(ListNode head, int k) {
  ListNode dummy = new ListNode(0);
  dummy.next = head;

  ListNode pre = dummy, cur = head;
  int pos = 1;
  while (cur != null) {
    if (pos == k) {
      pos = 0;
      pre = reverse(pre, cur);
      cur = pre.next;
    } else {
      cur = cur.next;
    }
    ++pos;
  }

  return dummy.next;
}
```

Posted 19th May 2013 by Sophie

Labels: Java, LinkedList

0 Add a comment

19th May 2013

Reverse Integer / Binary Bits

Reverse Integer [http://leetcode.com/onlinejudge#question_7]

Reverse digits of an integer. Example1: x = 123, return 321

Solution

The problem description is vague and desired to be like that. During an interview, you need to ask questions to clarify the problem and also show that you are detain-oriented.

What questions would you ask? Here are mine:

- Can it be negative?
  - Of course. Example 2: x = -123, return -321.
  (Well, it turns out we don't need to explicitly address the sign in the solution unless you want to use a string. Mod, %, will preserve the sign of the input number.)
- What if it is ended with 0's?
  - The output should be a valid number. That said, if x = 100, return 1.
- What if reversed numer is greater than Integer.MAX_VALUE?
  - What will you do to handle it?
  - Throw an overflow exception or in C++, change the method signature to be something like `bool reverse(int x, int res);`

The problem itself is easy and takes O(n) time to solve it.

```
public int reverse(int x) {
  long res = 0;

  while (x != 0) {
    res = (res * 10) + x % 10;
    x /= 10;
  }

  return (int)res;
}
```

Reverse Binary [http://discuss.leetcode.com/questions/169/reverse-bits]

Reverse bits of an unsigned integer.

Solution

There is a constant solution for reversing 4-byte-long binary by using bit manipulations [http://sophie-notes.blogspot.com/2013/02/bit-manipulation-tutorial-i.html] .

The basic idea is to use Divide and Conquer algorithm [http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm] .

- Switch the neighboring bits, i.e. 1001 -> 0110
- Switch the neighboring two bits, i.e. 1011 -> 1110
- ...
- Switch the neighboring two bytes.

```
unsigned int reverseBinary(unsigned int x) {
  x = ((x & 0x55555555) << 1) | ((x & 0xAAAAAAAA) >> 1);
  x = ((x & 0x33333333) << 2) | ((x & 0xCCCCCCCC) >> 2);
  x = ((x & 0x0F0F0F0F) << 4) | ((x & 0xF0F0F0F0) >> 4);
  x = ((x & 0x00FF00FF) << 8) | ((x & 0xFF00FF00) >> 8);
  x = ((x & 0x0000FFFF) << 16) | ((x & 0xFFFF0000) >> 16);
  return x;
}
```

Posted 19th May 2013 by Sophie

Labels: BitManipulation, Java, Maths

2 View comments

17th May 2013

Restore IP Addresses

Restore IP Addresses [http://leetcode.com/onlinejudge#question_93]

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

For example:
Given "25525511135", return ["255.255.11.135", "255.255.111.35"]. (Order does not matter)

Solution

The basic problem is to partition a string into exactly four substrings and the length of each substring is between 1 and 3. We can use recursion to solve it.

- Partition the string into two, one is 1-3 char long and the other is the rest.
- Recursively partition the rest part until we get all four substrings or it is not valid.

A valid field of an IP address is

- 1-3 digits (as stated above)
- between 0 and 255, inclusively
- not prefixed with "00"

```
// a valid field value is 1-3 digits, "0", "1**", ~ "255", and not 0??
private boolean isValidField(String s) {
  if (s.length() == 1 || (!s.isEmpty() && s.length() < 4 && !s.startsWith("0"))) {
    int num = Integer.parseInt(s);
    if (num >= 0 && num <= 255) return true;
  }
  return false;
}

private ArrayList<String> restoreIPAddrHelper(String s, int fields) {
  ArrayList<String> results = new ArrayList<String>();

  // last field
  if (fields == 1) {
    if (isValidField(s)) results.add(s);
    return results;
  }

  // 1-3 digits for a field
  for (int i=1; i<=3 && i<=s.length(); ++i) {
    String num = s.substring(0,i);
    if (!isValidField(num)) return results;

    ArrayList<String> substrings = restoreIPAddrHelper(s.substring(i), fields-1);
    for (String substr : substrings)  results.add(num + "." + substr);
  }

  return results;
}

public ArrayList<String> restoreIpAddresses(String s) {
  return restoreIPAddrHelper(s, 4);
}
```

In the above algorithm, we created an ArrayList in each recursion which wastes a lot of time and spaces since it causes inexplicit copies.

A better practise is to pass in an ArrayList to recursions and reuse it repeatedly.

```
  private void getAllIps(String s, int n, StringBuilder path, ArrayList<String> res) {
    if (n == 1) {
      if (isValidIp(s)) res.add(path.toString() + s);
      return;
    }

    for (int i=1, len=path.length(); i<=3 && i<s.length(); ++i) {
      String field = s.substring(0, i);
      if (isValidIp(field)) getAllIps(s.substring(i), n-1, path.append(field).append('.'), res);
      path.delete(len, path.length());
    }
  }

  public ArrayList<String> restoreIpAddresses(String s) {
    ArrayList<String> res = new ArrayList<String>();
    getAllIps(s, 4, new StringBuilder(), res);
    return res;
  }
```

Posted 17th May 2013 by Sophie

Labels: Java, Recursion, String, StringBuilder

0 Add a comment

16th May 2013

Wildcard Matching

Wildcard Matching [http://leetcode.com/onlinejudge#question_44]

Implement wildcard pattern matching with support for '?' and '*'.

'?' Matches any single character.
'*' Matches any sequence of characters (including the empty sequence).

The matching should cover the entire input string (not partial).

The function prototype should be:
bool isMatch(const char *s, const char *p)

Some examples:
isMatch("aa","a") ? false
isMatch("aa","aa") ? true
isMatch("aaa","aa") ? false
isMatch("aa", "*") ? true
isMatch("aa", "a*") ? true
isMatch("ab", "?*") ? true
isMatch("aab", "c*a*b") ? false

Solution - Recursion/Backtracking

This problem is an extended version of previous regex post [http://n00tc0d3r.blogspot.com/2013/05/regular-expression-matching.html] . The difference is that '*' now becomes a wildcard which can match any sequences of char (zero or more).

We can use similar ideas to solve this problem.

- If current of p is not '*', try to match single char. If succeed, move to next; if fail, fail.
- If current of p is '*', firstly, skip duplicate '*'s; then use backtracking to test all subsequences of s.

```
public boolean isMatch(String s, String p) {
  if (s == null || p == null) return false;
  if (p.isEmpty()) return s.isEmpty();

  // '*'
  if (p.charAt(0) == '*') {
    // skip duplicate '*'
    int nonstar = 0;
    while (nonstar < p.length() && p.charAt(nonstar) == '*') ++nonstar;
    int index = 0;
    while (index <= s.length()) {
      if (isMatch(s.substring(index), p.substring(nonstar)))
        return true;
      ++index;
    }
  } else {
    // single char match
    if (!s.isEmpty() &&
        (s.charAt(0) == p.charAt(0) || p.charAt(0) == '?')) {
      return isMatch(s.substring(1), p.substring(1));
    }
  }

  // no match
  return false;
}
```

The worst case running time for this algorithm is exponential. Consider a case where `isMatch("aaa...aaa", "*a*a*a...a*a*a*")`. :-[

Solution - DP

Use DP to optimize performance.

Suppose we have a m*n table T of boolean values, where m is the length of pattern p, n is the length of source s, and T[i][j] == true means p[0..i] matches s[0..j]. After fill up the table, T[m-1][n-1] will be the result for the problem.

How to fill up T[i] row? Given results from previous rows and a single char p[i] in pattern p,

- If p[i] is not '*', if p[0..i-1] matches s[0..k-1] and p[i] matches s[k], then we can say p[0..i] matches s[0..k].
  That said, T[i][k] == true iff p[i] matches s[k] and the diagonal value is previous row, T[i-1][k-1] also match.
  Note that there may exist multiple match points in previous row and thus we need to go through the previous row to get all of them.
- If p[i] is '*', if p[0..i-1] matches s[0..k-1], then we know for 0 < k < n, p[0..i] matches s[0..k] since '*' can match any length of char sequences, including 0.
  That said, if T[i-1][k-1] is true, **starting from** k-1, the rest of row T[i] are true.
  Note that we can not fill up the entire row with true, otherwise, we lost track of which part has been matched in source.

Here is an example for `isMatch("mississippi", "m*iss*p")`.

[http://2.bp.blogspot.com/-slb6dDq3HkI/UZW3bhlg4oI/AAAAAAAAEfQ/do3_hztppdI/s1600/Screen+Shot+2013-05-16+at+9.50.40+PM.png]

Notice that we only need a few pieces of information from previous row: the diagonal value in previous row for non-'*', and the first matching point in previous row for '*'. That means, we only need one row R, instead of a 2D table! Now it becomes

- For non-'*', walk backwards to update values by verifying matching and check the value in R[i-1]. Also update the first matching point for future usage.
- For '*', starting for the first matching point and set the rest of row to true.

The average running time is still O(m*n) since for each row, we need to update n - firstTrue values and there are m rows at most. Obviously, we only use O(n) space now.

Several small optimizations:

- Add a dummy node the matches array.

This can handle empty strings cases and also no need to worry about first row case.

- Add a preprocess to calculate the count of non-'*' char in p, if that count is still greater than the length of s, then fail right away.

Counting takes an extra O(n) time. But for some extreme cases like `isMatch("aaa...aaa", "*aaa...aaa*")`, which requires a full table filling, this extra O(n) may save the unnecessary O(m*n) time. In other cases, this extra time will not increase the order of total running time. Note that this will not improve the average running time of this algorithm.

- When iterating through p to fill up the table (I mean, to update the matches value for current row.), skip duplicate '*'s.

When hitting a duplicate '*', it will "copy" over the match values from previous row (i.e. set them to true again although they already are), which is waste of time. Again, this may not improve the average running time of this algorithm. But for cases like `isMatch("abscwet", "***...***")`, it does reduce the running time from O(m*n) to O(n).

```
public boolean isMatch(String s, String p) {
  if (s == null || p == null) return false;

  // calculate count for non-wildcard char
  int count = 0;
  for (Character c : p.toCharArray()) {
    if (c != '*') ++count;
  }
  // the count should not be larger than that of s
  if (count > s.length()) return false;

  boolean[] matches = new boolean[s.length()+1];
  matches[0] = true;
  int pid = 0, firstMatch = 0;
  while (pid < p.length()) {
    // skip duplicate '*'
    if (pid > 0 && p.charAt(pid) == '*' && p.charAt(pid-1) == '*') {
      ++pid;
      continue;
    }

    // if '*', fill up the rest of row
    if (p.charAt(pid) == '*') {
      // fill up the rest of row with true, up to the first match in previous row
      for (int i = firstMatch+1; i <= s.length(); ++i) matches[i] = true;
    } else {
      // fill up backwards:
      // - set to true if match current char and previous diagnal also match
      // - otherwise, set to false
      int match = -1;
      for (int i=s.length(); i>firstMatch; --i) {
        matches[i] = (p.charAt(pid) == s.charAt(i-1) || p.charAt(pid) == '?')
              && matches[i-1];
        if (matches[i]) match = i;
      }
      if (match < 0) return false;
      firstMatch = match;
    }

    ++pid;
  }

  return matches[s.length()];
}
```

Posted 16th May 2013 by Sophie

Labels: Backtracking, DP, Java, Recursion

0 Add a comment

13th May 2013

Regular Expression Matching

Regular Expression Matching [http://leetcode.com/onlinejudge#question_10]

Implement regular expression matching with support for '.' and '*'.

'.' Matches any single character. '*' Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

Some examples:

isMatch("aa","a") ? false
isMatch("aa","aa") ? true
isMatch("aaa","aa") ? false
isMatch("aa", "a*") ? true
isMatch("aa", ".*") ? true
isMatch("ab", ".*") ? true
isMatch("aab", "c*a*b") ? true

Solution - Recursion

This is a simplified Regex [http://en.wikipedia.org/wiki/Regex] problem (see this post [http://swtch.com/~rsc/regexp/regexp1.html] for more implementation details). Notice that we only consider '.' and '*' and '*' is not a wildcard in this problem.

'.' is easy to implement: if the current char of p is '.', ignore the corresponding char of s.

For '*', since it could be zero instance of its preceding char, even the current char are not matching or s is empty, the entire strings may still match. E.g. "" and ".*" or "a*"; "b" and ".*b" or "a*b".

That said, in the cases p has '(something)*',

- If s is empty or current char do not match, skip the '(something)*' part of p and continue to compare the rest of strings;
- If current char match, we need to consider three subcases
    - zero instance of (something) in p
    - only one instance of (something) in p
    - multiple instances of (something) in p

```
public boolean isMatch(String s, String p) {
  if (s.isEmpty()
      && (p.isEmpty() || p.length() == 2 && p.charAt(1) == '*'))
    return true;
  if ((p.isEmpty() && !s.isEmpty()) || (!p.isEmpty() && s.isEmpty())) return false;

  if (p.charAt(0) == '.' || s.charAt(0) == p.charAt(0)) {
    if (p.length() >= 2 && p.charAt(1) == '*') {
      return isMatch(s.substring(1), p) // more
        || isMatch(s.substring(1), p.substring(2)) // once
        || isMatch(s, p.substring(2)); // zero
    }
    return isMatch(s.substring(1), p.substring(1));
  } else if (p.length() >= 2 && p.charAt(1) == '*') {
    return isMatch(s, p.substring(2));
  }

  return false;
}
```

Solution - Recursion/Backtracking

Notice that the above algorithm may have redundant computations.
For example, given "ab" and "a*a*a*b".

- In the first iteration, it populates three subcases: "b" and "a*a*a*b", "b" and "a*a*b", "ab" and "a*a*b".
- In the next iteration for "b" and "a*a*a*b", it populates "" and "a*a*a*b", "" and "a*a*b", "b" and "a*a*b".
- In the next iteration for "b" and "a*a*b", it populates "" and "a*a*b", "" and "a*b", "b" and "a*b".
- In the next iteration for "ab" and "a*a*b", it populates "b" and "a*a*b", "b" and "a*b", "ab" and "a*b".
- ... ...

The performance can be improved if we use Backtracking [http://en.wikipedia.org/wiki/Backtracking] .
When we hit a character followed by a '*', we use a loop to match repeat characters and fast succeed if a match is found.

The result of the example showed above becomes:

- "ab" and "a*a*b"
- "ab" and "a*b"
- "ab" and "b" (fail to match)
- "b" and "b" (match)

Another example is "aab" and "a*a*a*b".

- "aab" and "a*a*b"
- "aab" and "a*b"
- "aab" and "b" (fail to match)
- "ab" and "b" (fail to match)
- "b" and "b" (match)

You can see that there is no redundant computations any more.

```
public boolean isMatch(String s, String p) {
  if (s == null) return (p == null);
  if (p.isEmpty()) return (s.isEmpty());

  // next char is not '*': do current char match?
  if (p.length() == 1 || p.charAt(1) != '*') {
    if (s.isEmpty()) return false;
    return (p.charAt(0) == '.' || s.charAt(0) == p.charAt(0))
          && isMatch(s.substring(1), p.substring(1));
  }

  // next char is '*'
  // current char match, zero or more repeats
  int i = 0;
  while (i < s.length() && (p.charAt(0) == '.' || s.charAt(i) == p.charAt(0))) {
    if (isMatch(s.substring(i), p.substring(2))) return true;
    i++;
  }

  // zero
  return isMatch(s.substring(i), p.substring(2));
}
```

Note that in Java, `substring` will make a copy of the string. That said, if we have k lever of recursions, copy-string could cost O(n^2) in total. So, the performance can be further improved by using char arrays instead of strings.

```
public boolean isMatch(String s, String p) {
  return isMatchHelper(s, 0, p, 0);
}

private boolean isMatchHelper(String s, int i, String p, int j) {
  if (p.length() == j) return (s.length() == i);

  // next char is not '*': do current char match?
  if (j == p.length() - 1 || p.charAt(j+1) != '*') {
    if (s.length() == i) return false;
    return (p.charAt(j) == '.' || s.charAt(i) == p.charAt(j))
          && isMatchHelper(s, i+1, p, j+1);
  }

  // next char is '*'
  // one or more
  while (i < s.length() && (p.charAt(j) == '.' || s.charAt(i) == p.charAt(j))) {
    if (isMatchHelper(s, i, p, j+2)) return true;
    i++;
  }

  // zero
  return isMatchHelper(s, i, p, j+2);
}
```

Posted 13th May 2013 by Sophie

Labels: Backtracking, DP, Java, Recursion

0 Add a comment

13th May 2013

Remove Duplicates from Sorted List

Remove Duplicates from Sorted List [http://leetcode.com/onlinejudge#question_83]

Given a sorted linked list, delete all duplicates such that each element appear only once.

For example,
Given 1->1->2, return 1->2.
Given 1->1->2->3->3, return 1->2->3.

Solution

This is a simple problem.

As we discussed in previous post [http://n00tc0d3r.blogspot.com/2013/05/remove-n-th-to-end-element-from-list.html] , to remove node A, we need to know the node ahead of A.
So, we start from the head, and check whether the nodes after current has the same value. If so, skip to next; if not, move on.

```
public ListNode deleteDuplicates(ListNode head) {
  ListNode cur = head;
  while (cur != null && cur.next != null) {
    if (cur.val == cur.next.val) { // remove duplicates
      cur.next = cur.next.next;
    } else {
      cur = cur.next;
    }
  }
  return head;
}
```

This algorithm runs in time O(n) and uses O(1) space.

Remove Duplicates from Sorted List [http://leetcode.com/onlinejudge#question_81]

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list.

For example,
Given 1->2->3->3->4->4->5, return 1->2->5.
Given 1->1->1->2->3, return 2->3.

Solution

Now the problem gets to be interesting.

Since we need to remove not only the duplicates but also the one has duplicates (i.e. the first one), we need two pointers: pre and cur. If cur has duplicates, remove all its duplicates and then change pre.next to cur.next so as to skip cur itself.

The special case is that the head node has duplicates (and of course head node has no previous node).

```
public ListNode deleteDuplicates(ListNode head) {
  ListNode pre = null, cur = head;
  while (cur != null && cur.next != null) {
    if (cur.val != cur.next.val) { // non-duplicate, move on
      pre = cur;
    } else { // has duplicates
      while (cur.next != null && cur.val == cur.next.val) { // skip duplicates
        cur.next = cur.next.next;
      }
      // skip the one has duplicates
      if (pre == null)
        head = cur.next;
      else
        pre.next = cur.next;
    }
    cur = cur.next;
  }
  return head;
}
```

Another way to solve this problem is to add a dummy node ahead of the head node and assign it a value that is different from head's. Then every node in the original list has a previous node. We can apply same strategy for all of them.

```
public ListNode deleteDuplicates(ListNode head) {
  ListNode dummy = new ListNode(0);
  dummy.next = head;

  ListNode pre = dummy, cur = head;
  while (cur != null && cur.next != null) {
    if (cur.next.val != cur.val) {
      pre = cur;
    } else {
      while (cur.next != null && cur.next.val == pre.next.val) {
        cur = cur.next;
      }
      pre.next = cur.next;
    }
    cur = cur.next;
  }

  return dummy.next;
}
```

Both algorithm run in time O(n) and use O(1) space.

Posted 13th May 2013 by Sophie

Labels: Java, LinkedList

0 Add a comment

13th May 2013

Remove n-th-to-end Element from List

Remove the n-th Element From End of List [http://leetcode.com/onlinejudge#question_19]

Given a singly linked list, remove the n-th node from the end of list and return its head.

For example, given linked list: 1->2->3->4->5, and n = 2, after removing the second node from the end, the linked list becomes 1->2->3->5.

Note: Try to do this in one pass.

Solution

As a singly linked list, we have to iterate through the list to find out the length of the list and the n-th-to-end node. Also, to remove a node A, we need to know the node before A so as to move the next pointer of the previous node from A to A.next (In C++, you also need to delete A to release the memory.).

To do these in one pass, we need two pointers and keep them n+1 nodes away, i.e. having n nodes between the two pointers. We also need to consider special case where n is not valid (i.e. n > list length) or n-th-to-end is the head node.

```
 public ListNode removeNthFromEnd(ListNode head, int n) {
  // pilot is supposed to be n+1 ahead of pre
  // so when pilot reaches the end, pre points to the node right before n-th
  // that said, there has to be n nodes between the two pointers
  ListNode pilot = head, pre = head;
  while (pilot != null) {
    if (n >= 0) { // forward pilot pointer
      --n;
    } else { // forward pre pointer
      pre = pre.next;
    }
    pilot = pilot.next;
  }
  if (n > 0) { // nothing to remove
  } else if (n == 0) { // remove head
    head = head.next;
  } else { // remove n-th
    pre.next = pre.next.next;
  }
  return head;
}
```

Posted 13th May 2013 by Sophie

Labels: Java, LinkedList

0 Add a comment

13th May 2013

Remove Elements from Array

Remove Elements from Array [http://leetcode.com/onlinejudge#question_27]

Given an array and a value, remove all instances of that value in place and return the new length.
It doesn't matter what you leave beyond the new length.

Solution

This is an simple question.

One thing is that we don't want to move all of the elements every time when we find an occurrence since that could be expensive and worst case running time would be O(n^2).

Instead, if we count the number of known occurrence, we can move the next not-to-remove element directly to the right spot. By doing this, we only need to iterate through all elements once, which gives us an O(n) solution.

```
public int removeElement(int[] A, int elem) {
  int count = 0;
  for (int i=0; i<A.length; ++i) {
    if (A[i] == elem) { // find one, skip
      ++count;
    } else if (count > 0) { // copy over
      A[i-count] = A[i];
    }
  }
  return A.length - count;
}
```

Remove Duplicates from Sorted Array [http://leetcode.com/onlinejudge#question_26]

Given a sorted array, remove the duplicates in place such that each element appear only once and return the new length.
Do not allocate extra space for another array, you must do this in place with constant memory.

For example, given input array A = [1,1,2], your function should return length = 2, and A is now [1,2].

Solution

The basic idea is same as above. Maintaining count of duplicates and copy over non-duplicates directly to the right spot.

```
public int removeDuplicates(int[] A) {
  int count = 0;
  for (int i=1; i<A.length; ++i) {
    if (A[i] == A[i-1]) { // skip duplicates
      ++count;
    } else if (count > 0) { // copy over non-duplicates
      A[i-count] = A[i];
    }
  }
  return A.length - count;
}
```

Remove Duplicates from Sorted Array II [http://leetcode.com/onlinejudge#question_80]

Follow up for "Remove Duplicates":
What if duplicates are allowed at most twice?

For example,
Given sorted array A = [1,1,1,2,2,3],

Your function should return length = 5, and A is now [1,1,2,2,3].

Solution

So, if you simple change the condition to `if (A[i] == A[i-1] && A[i] == A[i-2])`, you are trapped!

Why?

In the above problem, we only need to test the one right before the current one which is not possible to have been overwritten. But now, we also need to test the one that is two nodes away which is possible to have been overwritten. That said, we should test the two nodes that are supposed to be ahead of the current node in the new array.

```
public int removeDuplicates(int[] A) {
  int count = 0;
  for (int i=2; i<A.length; ++i) {
    if (A[i] == A[i-count-1] && A[i] == A[i-count-2]) { // skip duplicates
      ++count;
    } else if (count > 0) { // copy over non-duplicates
      A[i-count] = A[i];
    }
  }
  return A.length - count;
}
```

Posted 13th May 2013 by Sophie

Labels: Array, Java

4 View comments

10th May 2013

Recover Binary Search Tree

Recover Binary Search Tree [http://leetcode.com/onlinejudge#question_99]

Two elements of a binary search tree (BST) are swapped by mistake.
Recover the tree without changing its structure.

Solution

A solution using O(n) space is pretty straight forward.

First, recall that an important property of a BST is that in-order traversal will give us a sorted array of numbers. Now the question becomes to recover a sorted array where two elements were swapped wrongly.

E.g. Given [1, **4**, 3, **2**, 5], which two shall we swap to fix the array?
One way is starting from the second element in the list and comparing it with its previous one. If it is smaller than its previous one, then we know at least one of the two is in wrong spot. So, the larger one in the first pair and the smaller one in the second pair are the ones we are looking for.

But, how about [1, **3**, **2**, 4, 5]?
This tells us that if there is only one such pair where the latter one is greater than the previous one. That pair is actually the two element we are looking for.

Since we only need to compare with previous node, we don't need to store all nodes during a traversal. We only need to keep track of the previously visited node. The algorithm is quite similar to the ones in the post [http://n00tc0d3r.blogspot.com/2013/04/validate-binary-search-tree.html] of validating BST.

Here is a solution using stack.

```
private void swap(TreeNode a, TreeNode b) {
  if (n1 == null || n2 == null)  return;
  int tmp = a.val;
  a.val = b.val;
  b.val = tmp;
}

public void recoverTree(TreeNode root) {
  TreeNode cur = root, pre = null, first = null, second = null;
  // in order travesal should return a sorted list
  Stack<TreeNode> stack = new Stack<TreeNode>();
  while (cur != null) { // find the left most child
    stack.push(cur);
    cur = cur.left;
  }
  while (!stack.isEmpty()) {
    cur = stack.pop();

    // is it wrong?
    if (pre != null && cur.val < pre.val) {
      if (first == null) {
        // the first wrong item should be the bigger one
        first = pre;
        second = cur; // there is a chance that the two were swapped
      } else {
        // the second wrong item should be the smaller one
        second = cur;
        break;
      }
    }

    // go to right child and repeat
    pre = cur;
    cur = cur.right;
    while (cur != null) {
      stack.push(cur);
      cur = cur.left;
    }
  }

  swap(first, second);
}
```

This algorithm runs in time O(n) and uses O(h) spaces, where n is the number of nodes in the tree and h is the height of the tree. So, in worst cases, the space complexity can be O(n).

Here is a solution using recursion.

Notice that instead of passing in two TreeNode, first and second, we passed in an array of two nodes. The reason of doing that is for Java we cannot change the value of pass-in object (see this post [http://sophie-notes.blogspot.com/2012/12/java-passing-arguments-into-methods.html]).

```
public void recoverTree(TreeNode root) {
  TreeNode[] nodes = new TreeNode[2];
  inorder(root, nodes, null);
  swap(nodes[0], nodes[1]);
}

// in-order traversal and return the last visited node in the traversal
private TreeNode inorder(TreeNode root, TreeNode[] nodes, TreeNode pre) {
  if (root == null) return pre;

  // left subtree
  TreeNode last = inorder(root.left, nodes, pre);
  // visit
  if (last != null && root.val < last.val) {
    nodes[1] = root;
    if (nodes[0] == null) { // found first node
      nodes[0] = last;
    } else {
      return root;
    }
  }
  // right subtree
  return inorder(root.right, nodes, root);
}
```

Again, this algorithm runs in time O(n) and uses O(h) spaces since it uses implicit stack for recursions.

Can we do better?
To find out the bad nodes, we have to perform an inorder traversal. So the question becomes: Can we conduct an inorder traversal with constant space usage?
The answer is Yes! See this post [http://n00tc0d3r.blogspot.com/2013/09/inorder-binary-tree-traversal-with.html] for an algorithm, Morris Traversal, that implement inorder traversal without recursion or stack.

We need to modify that algorithm so that we can store the previously visited node and compare it with the current node. Notice that only when we move the current to its right child, we store the current as previously visited.

```
private void recoverMorris(TreeNode root) {
  TreeNode pre = null, cur = root, n1 = null, n2 = null;

  while (cur != null) {
    if (cur.left != null) {
      TreeNode p = cur.left;
      while (p.right != null && p.right != cur) {
        p = p.right;
      }
      if (p.right == null) { // set right to successor
        p.right = cur;
        cur = cur.left;
      } else { // visit and revert the change
        p.right = null;
        if (pre.val > cur.val) {
          n2 = cur;
          if (n1 == null) n1 = pre;
        }
        pre = cur;
        cur = cur.right;
      }
    } else { // visit
      if (pre != null && pre.val > cur.val) {
        n2 = cur;
        if (n1 == null) n1 = pre;
      }
      pre = cur;
      cur = cur.right;
    }
  }
  swap(n1, n2);
}
```

This algorithm runs in time O(n) and uses O(1) space! :-]

Posted 10th May 2013 by Sophie

Labels: BinarySearchTree, BinaryTree, Java, Recursion, Stack, Tree

0 Add a comment

7th May 2013

Pow(x, n)

Pow(x, n) [http://leetcode.com/onlinejudge#question_50]

Implement pow(x, n).

Solution

The straight forward way is to use a while loop and increase/decrease the exponent by one in each iteration.

- return 1, if n == 0;
- return x*x*...*x (n times), if n > 0;
- return 1/x/x.../x (-n times), if n < 0.

```
public double pow(double x, int n) {
  double result = 1;
  if (n == 0) return result;
  while (n > 0) {
    result *= x;
    --n;
  }
  while (n < 0) {
    result /= x;
    ++n;
  }
  return result;
}
```

This works and will return the right result. But the running time is O(n). What if n is a large number like 1152489739?

Rather than changing the exponent linearly, we can fold it each time. By doing that, the running time can be reduced to O(logn).

Now the formula is

- return 1, if n == 0;
- return 1/pow(x, -n), if n < 0;
- return x^(n/2) * x^(n/2), if n is even;
- return x * x^(n/2) * x^(n/2), if n is odd.

Notice that in the algorithm showing below, we did not use pow(x, -n). The reason is that -n may cause overflow sometimes (remember the range of integers is [-2^32, 2^32-1]).

Also notice that we use ">>1" and "&1" to replace "/2" and "%2" since bit operations are way faster than divide and mod.

```
public double pow(double x, int n) {
  if (n == 0) return 1;
  int exp = Math.abs(n);
  double half = pow(x, exp>>1);
  double res = half * half;
  if ((exp & 1) == 1) res *= x;
  return (n > 0) ? res : 1.0/res;
}
```

We can also implement with iteration. Enjoy!

```
public double pow(double x, int n) {
  double result = 1;
  for (int m = Math.abs(n); m > 0; x *= x, m >>= 1) {
    if ((m & 1) == 1) result *= x;
  }
  return (n >= 0) ? result : 1.0 / (result);
}
```

Posted 7th May 2013 by Sophie

Labels: Java, Maths, Recursion

3 View comments

6th May 2013

Populating Next Right Pointers for Each Tree Node

Populating Next Right Pointers for Each Tree Node in a full binary tree [http://leetcode.com/onlinejudge#question_116]

Given a binary tree

```
struct TreeLinkNode {
      TreeLinkNode *left;
      TreeLinkNode *right;
      TreeLinkNode *next;
    }
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

Note:

- You may only use **constant extra space**.
- You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).

For example,
Given the following perfect binary tree,

```
        1
       /  \
      2    3
     / \  / \
    4  5  6  7
```

After calling your function, the tree should look like:

```
        1 -> NULL
       /  \
      2 -> 3 -> NULL
     / \  / \
    4->5->6->7 -> NULL
```

Solution

In previous post [http://n00tc0d3r.blogspot.com/2013/01/binary-tree-traversals-ii.html] , we discussed how to get a level-order traversal. If we traverse the tree level by level, simply set the next pointer to the next node in the same level and set it to null if reaching the end of a level. But that requires a queue to store the node in the current level, which could be O(n) space.
Here, it requires constant extra space. So, level-order traversal cannot be used for this problem.

The constant-extra-space also tells us no recursion since an n-level recursion usually takes at least O(n) space for stacks.

Given a node with left and right children, it is easy to set the next pointer of its left child but for the right child, how do we know whether this node has or has not siblings?
The answer is: Use the next pointer. :D

Since we assume the given tree is a full binary tree, to get to next level, just move the first node of this level to its left child.

```
public void connect(TreeLinkNode root) {
  TreeLinkNode first = root;
  while (first != null) {
    TreeLinkNode cur = first;
    // set up next pointers for the next level
    while (cur != null) {
      if (cur.left != null)  cur.left.next = cur.right;
      if (cur.right != null && cur.next != null)  cur.right.next = cur.next.left;
      cur = cur.next;
    }
    // move to next level
    first = first.left;
  }
}
```

We can also implement it with one single while-loop.

```
public void connect(TreeLinkNode root) {
  TreeLinkNode first = root, cur = root;
  while (cur != null) {
    if (cur.left == null && cur.right == null) break;

    if (cur.left != null)  cur.left.next = cur.right;
    if (cur.next != null) {
      if (cur.right != null)  cur.right.next = cur.next.left;
      cur = cur.next;
    } else {
      cur = first.left;
      first = cur;
    }
  }
}
```

Populating Next Right Pointers for Each Tree Node in any binary tree [http://leetcode.com/onlinejudge#question_116]

What if the given tree could be any binary tree? Would your previous solution still work?

Again, you may only use **constant extra space**.

Solution

If the given tree may not necessarily be a full binary tree, two things need to change:

- To find the next child, we may need to go through the current one's siblings to find the one that has at least one child;
- To move to next level, similarly, we need to find the first node in the current level that has at least one child.

```
public void connect(TreeLinkNode root) {
  TreeLinkNode first = root;
  while (first != null) {
    TreeLinkNode cur = first;
    // set up next pointers for the next level
    while (cur != null) {
      // find the next node that has at least one child
      TreeLinkNode next = cur.next;
      while (next != null && next.left == null && next.right == null) next = next.next;
      // the next child
      TreeLinkNode nextChild = null;
      if (next != null) nextChild = (next.left != null) ? next.left : next.right;
      // set up next pointers for the children of current node
      if (cur.left != null && cur.right != null) {
        cur.left.next = cur.right;
        cur.right.next = nextChild;
      } else if (cur.left != null) {
        cur.left.next = nextChild;
      } else if (cur.right != null) {
        cur.right.next = nextChild;
      }
      // move to next has-child node
      cur = next;
    }
    // move to next level
    while (first != null && first.left == null && first.right == null) first = first.next;
    if (first != null) first = (first.left != null) ? first.left : first.right;
  }
}
```

By wrapping up a helper method, the code can be more clear.

```
private void fillRow(TreeLinkNode cur) {
  while (cur != null) {
    if (cur.left != null && cur.right != null) {
      cur.left.next = cur.right;
    }

    // find next has-child node
    TreeLinkNode next = cur.next;
    while (next != null && next.left == null && next.right == null) {
      next = next.next;
    }
    if (next == null) break;
    // find next child and link it
    TreeLinkNode nextChild = (next.left != null) ? next.left : next.right;
    if (cur.right != null) {
      cur.right.next = nextChild;

    } else if (cur.left != null) {
      cur.left.next = nextChild;
    }

    // move to next has-child node directly
    cur = next;
  }
}

public void connect(TreeLinkNode root) {
  TreeLinkNode first = root;
  while (first != null) {
    fillRow(first);
    // move to next level
    while (first != null && first.left == null && first.right == null) first = first.next;
    if (first != null) first = (first.left != null) ? first.left : first.right;
  }
}
```

Posted 6th May 2013 by Sophie

Labels: BinaryTree, Java, Tree

1 View comments

6th May 2013

Plus One

Plus One [http://leetcode.com/onlinejudge#question_66]

Given a number represented as an array of digits, plus one to the number.

Solution

This problem is easy.

We don't need to pass carry to prior digit unless the current digit is 9.
So, start from the last digit, if it is less than 9, increase it by 1 and return; if it is 9, set it to zero and pass on 1 to prior digit.

One thing that needs to pay attention is that if digits are all 9's, the result will be one digit longer than original array and it will be 1 followed by n 0's where n is the length of original array. In this case, we need to create a new array but we don't really need to copy anything to the new array since by default the array will be initialized with 0's. We only need to flip the first digit to 1.

```
public int[] plusOne(int[] digits) {
  int n = digits.length;
  for (int i=n-1; i>=0; --i) {
    if (digits[i] < 9) {
      ++digits[i];
      return digits;
    } else {
      digits[i] = 0;
    }
  }
  // if we are here, digits are 99...99 and the result should be 100...00.
  int[] result = new int[n+1]; // initialized to all 0's by default
  result[0] = 1;
  return result;
}
```

This problem can be extended to "plus an integer".

Now we need to calculate sum on each digit and pass carry to higher digits until carry is zero.
Again, if carry is greater than zero after iterating through all digits, we need to construct a new integer array of len+k, where k is the length of carry. This time we need copy over all digits from the prior results.

```
public int[] plusOne(int[] digits, int x) {
  int n = digits.length;
  for (int i=n-1; i>=0; --i) {
    if (x == 0) return digits;
    int d = x % 10; // get last digit of x
    x /= 10; // remove last digit from x
    digits[i] += d;
    if (digits[i] > 10) { // has carry
      digits -= 10;
      ++x; // pass on carry
    }
  }
  // calculate length of x
  int m = 0, y = x;
  while (y > 0) {
    ++m;  y /= 10;
  }
  // create a longer array
  int[] result = new int[n+m];
  System.arraycopy(digits, 0, result, m, n);
  for (int i=m-1; i>=0; --i) {
    result[i] = x % 10;
    x /= 10;
  }
  return result;
}
```

Other variants of adding numbers can be found in my previous post [http://n00tc0d3r.blogspot.com/2013/01/add-two-binary-numbers.html] .

Posted 6th May 2013 by Sophie

Labels: Array, Java

3 View comments

6th May 2013

Permutation Sequence

Permutation Sequence [http://leetcode.com/onlinejudge#question_60]

The set [1,2,3,…,n] contains a total of n! unique permutations.
By listing and labeling all of the permutations in order, we get the following sequence (ie, for n = 3):

"123"
"132"
"213"
"231"
"312"
"321"

Given n and k, return the k-th permutation sequence.
Note: Given n will be between 1 and 9 inclusive.

Solution

If this is a follow-up question of "next permutation" (see previous post [http://n00tc0d3r.blogspot.com/2013/04/next-permutation.html]), the first thought will be calling nextPermutation k times. That's expensive: we have to compute all previous k-1 permutations even though we never gonna use them!

Take a look at the entire sequence.
The first number will not change to next until all permutations of the rest of n-1 numbers have shown up. That said, given n = 4, k = 15 (= 2 * 3! + 1 * 2! + 1 * 1! + 0), the first number will be 3 (the 3-rd of [1,2,3,4]), the second number will be 2 (the 2-nd of [1,2,4]), and the last two numbers will be 1 and 4 (the 1-st permutation of [1,4]).

By doing these math, we only need O(n) time to get an index for the current spot (O(n) time to compute n!).
Notice that the calculated index is a relative index and thus we need to shift the numbers after we find out the one for the current spot (or erase the current one if we use separate strings for original and k-th permutation).

So, the total running time is O(n^2) since we may need to shift O(n) times. Also, it takes extra O(n) space for the factorials to reduce redundant calculations.

```
public String getPermutation(int n, int k) {
  if (n <= 0 || k <= 0) return "";

  // factorials of n
  int[] fact = new int[n];
  // an array of n numbers
  StringBuilder nums = new StringBuilder();
  // pre-compute factorials
  for (int i=1; i<=n; ++i) {
    nums.append(i);
    if (i == 1)
      fact[i-1] = 1;
    else
      fact[i-1] = fact[i-2] * i;
  }

  // normalize k so that it is within range [0 .. n!]
  while (k > fact[n-1]) k -= fact[n-1];
  k -= 1; // convert to 0-based

  // compute the permutation
  for (int i=0; i<n-1; ++i) {
    int factorial = fact[n-2-i]; // (n-1-i)!
    int id = k / factorial + i;
    // shift the numbers
    char num = nums.charAt(id);
    for (int j=id; j>i; --j) nums.setCharAt(j, nums.charAt(j-1));
    nums.setCharAt(i, num);
    while (k >= factorial) k -= factorial;
  }

  // convert to string
  return nums.toString();
}
```

Posted 6th May 2013 by Sophie

Labels: Array, Java, Maths, String, StringBuilder

2 View comments

6th May 2013 Permutations

Permutations (no duplicates) [http://leetcode.com/onlinejudge#question_46]

Given a collection of numbers, return all possible permutations.

For example,
[1,2,3] have the following permutations:
[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], and [3,2,1].

Solution

Permutations of n numbers can be computed based on permutations of n-1 numbers in the following way:

- Select one number, i, from n of them.
- Compute permutations of the rest n-1 numbers.
- Append i to the end of each permutation computed in above step. Note that adding a number to the end of array takes time O(1) while adding to the beginning takes time O(n).

This algorithm runs in time O(n!) which is the size of the results.

```
public ArrayList<ArrayList<Integer>> permute(int[] num) {
  ArrayList<ArrayList<Integer>> results = new ArrayList<ArrayList<Integer>>();
  if (num.length < 1) return results;
  if (num.length == 1) {
    ArrayList<Integer> perm = new ArrayList<Integer>();
    perm.add(num[0]);
    results.add(perm);
    return results;
  }

  for (int i=0; i<num.length; ++i) {
    // copy a new array of n-1 numbers
    int[] subset = new int[num.length-1];
    for (int j=0; j<i; ++j) subset[j] = num[j];
    for (int j=i+1; j<num.length; ++j) subset[j-1] = num[j];
    // append the current number to the end of pernutations of n-1 subset
    for (ArrayList<Integer> perm : permute(subset)) {
      perm.add(num[i]); // append to the end, O(1)
      results.add(perm);
    }
  }
  return results;
}
```

Permutations (with duplicates) [http://leetcode.com/onlinejudge#question_47]

Given a collection of numbers that might contain duplicates, return all possible **unique** permutations.

For example,
[1,1,2] have the following unique permutations:
[1,1,2], [1,2,1], and [2,1,1].

Solution

In previous post [http://n00tc0d3r.blogspot.com/2013/04/next-permutation.html] , we discussed "next permutation" problem. We can use the algorithm for that problem as a subroutine for this problem (with slight modification).

- Let nextPermutation return null if no more next, i.e. go back to the original array.
- Call nextPermutation until it returns null.

The downside of this solution is that each call of nextPermutation costs O(n) time. That said, to get n! results, it requires n*n! time.

Alternatively, we can modify the algorithm above so that it can be tolerant to numbers with duplicates.
To do that, we can add a hash table which stores the numbers having been visited. If we hit a number that has been visited, simple skip it. That only takes O(1) time.

This algorithm runs in time of the result size.

```
public ArrayList<ArrayList<Integer>> permuteUnique(int[] num) {
  ArrayList<ArrayList<Integer>> results = new ArrayList<ArrayList<Integer>>();
  if (num.length < 1) return results;
  if (num.length == 1) {
    ArrayList<Integer> perm = new ArrayList<Integer>();
    perm.add(num[0]);
    results.add(perm);
    return results;
  }
  HashMap<Integer, Boolean> visited = new HashMap<Integer, Boolean>();
  for (int i=0; i<num.length; ++i) {
    if (visited.get(num[i]) != null) continue;
    visited.put(num[i], true);
    // copy a new array of n-1 numbers
    int[] subset = new int[num.length-1];
    for (int j=0; j<i; ++j) subset[j] = num[j];
    for (int j=i+1; j<num.length; ++j) subset[j-1] = num[j];
    // append the current number to the end of pernutations of n-1 subset
    for (ArrayList<Integer> perm : permuteUnique(subset)) {
      perm.add(num[i]);
      results.add(perm);
    }
  }
  return results;
}
```

Posted 6th May 2013 by Sophie

Labels: Array, Java, Recursion

0 Add a comment

6th May 2013

Pascal's Triangle

Pascal's Triangle [http://leetcode.com/onlinejudge#question_118]

Given numRows, generate the first numRows of Pascal's triangle.

For example, given numRows = 5, return

```
[
     [1],
    [1,1],
   [1,2,1],
  [1,3,3,1],
 [1,4,6,4,1]
]
```

Solution

Pascal's Triangle [http://en.wikipedia.org/wiki/Pascal_triangle] is a triangle made of binomial coefficients.

Suppose the top row is numbered with n = 0 and the entries in each row are numbered from left with k = 0. Then a six-row Pascal's triangle is

[http://1.bp.blogspot.com/-8kkYOp7Khb8/UYbJ-VJ3mcI/AAAAAAAAEc4/HOejJgHHYfM/s1600/200px-Pascal's_triangle_5.png]

which represents the coefficients of

[http://1.bp.blogspot.com/-w7UGJUdQqSY/UYbKFFFM3oI/AAAAAAAAEdA/HzateW52O2o/s1600/PascalsTriangleCoefficient.jpg]

The beauty of the triangle is that entries of each row can be constructed from the ones from previous row. That is, for each entry, add the two numbers above it: one to the left and the other to the right. If either of them is not present, substitute with 0.

That's pretty much how the algorithm work.

```
public ArrayList<ArrayList<Integer>> generate(int numRows) {
  ArrayList<ArrayList<Integer>> triangle = new ArrayList<ArrayList<Integer>>();
  for (int i=0; i<numRows; ++i) {
    ArrayList<Integer> row = new ArrayList<Integer>();
    if (i==0) { // first row
      row.add(1);
    } else {
      ArrayList<Integer> preRow = triangle.get(i-1);
      for (int j=0; j<=i; ++j) {
        int val = ((j > 0) ? preRow.get(j-1) : 0)
             + ((j < preRow.size()) ? preRow.get(j) : 0);
        row.add(val);
      }
    }
    triangle.add(row);
  }
  return triangle;
}
```

This algorithm runs in time O(n^2) and use O(n^2) spaces which is the size of the result space.

Row of Pascal's Triangle [http://leetcode.com/onlinejudge#question_119]

Given an index k, return the kth row of the Pascal's triangle. Note that the top row of the triangle starting with 0 not 1.

For example, given k = 3, return [1,3,3,1].

Could you optimize your algorithm to use only O(k) extra space?

Solution

Since each row of the Pascal's triangle is constructed based on previous row. We need to keep tracking of previous row. But we only need keep one row rather all of them.

```
public ArrayList<Integer> getRow(int rowIndex) {
  ArrayList<Integer> row = null;
  for (int i=0; i<=rowIndex; ++i) {
    ArrayList<Integer> preRow = row;
    row = new ArrayList<Integer>();
    if (i==0) { // first row
      row.add(1);
    } else {
      for (int j=0; j<=i; ++j) {
        int val = ((j > 0) ? preRow.get(j-1) : 0)
             + ((j < preRow.size()) ? preRow.get(j) : 0);
        row.add(val);
      }
    }
  }
  return row;
}
```

This algorithm runs in time O(k^2) and use O(k) spaces.

Actually, if we do the calculation backwards for each row, we can do it in place without any extra spaces.

Note: The reason why we wrap the `asList` within an ArrayList constructor is that `asList` returns a fix-sized list per its documentation [http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html#asList%28T...%29] . The required signature returns an ArrayList which is a dynamic array and people who use this method will expect the results to be a dynamic array.

```
public ArrayList<Integer> getRow(int rowIndex) {
  Integer[] row = new Integer[rowIndex+1];
  for (int i=0; i<=rowIndex; ++i) {
    row[0] = 1;
    for (int j=i; j>0; --j) {
      int num = row[j-1];
      if (j<i) num += row[j];
      row[j] = num;
    }
  }
  return new ArrayList<Integer>(Arrays.asList(row));
}
```

Posted 6th May 2013 by Sophie

Labels: Array, Java

1 View comments

5th May 2013

Partition List

Partition List [http://leetcode.com/onlinejudge#question_86]

Given a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.
You should preserve the original relative order of the nodes in each of the two partitions.

Note that this is different from "moving all <x nodes before x and all >=x nodes after x".

For example,

- given 1 -> 4 -> 3 -> 2 -> 5 -> 2 and x = 3, return 1 -> 2 -> 2 -> 4 -> 3 -> 5.
- given 1 -> 3 -> -1 -> 5 -> 2 -> 1 and x = 3, return 1 -> -1 -> 2 -> 1 -> 3 -> 5.

Solution

Usually, to move a node A to the front of B in a linked list, we need to know the nodes that are ahead of A and B. Thus, we create a dummy node in front of the given head node.

> Well, there does exist a hacky way to move A to the front of B without knowing the nodes ahead of them.
> But it will not work if A is the tail of the list.
> To add a new node A in front of B without knowing the one ahead of B:
>
> 1. Create a new node with value of B and let its next point to B.next;
> 2. Change the value of B to that of A and let B.next point to the new node.
>
> Similarly, to remove a node A (if A is not the tail of the list) without knowing the one ahead of it:
>
> 1. Change the value of A to that of A.next;
> 2. Set A.next to A.next.next.

The algorithm given below runs in time O(n) since both pointers visit each node once.

The basic idea is to keep one pointer to the last continuous node from head that has value of less than x, then when we loop through the list and find any node that has value of less than x, simply move it after the last pointer and forward last pointer by one.

```
public ListNode partition(ListNode head, int x) {
  ListNode dummy = new ListNode(0);
  dummy.next = head;

  // find the last node that < x
  ListNode last = dummy;
  while (last.next != null && last.next.val < x) last = last.next;

  // start from the >=x node
  ListNode nt = last;
  while (nt.next != null) {
    if (nt.next.val < x) {
      ListNode tmp = nt.next;
      nt.next = tmp.next;
      tmp.next = last.next;
      last.next = tmp;
      last = last.next;
    } else {
      nt = nt.next;
    }
  }

  return dummy.next;
}
```

Posted 5th May 2013 by Sophie

Labels: Java, LinkedList

1 View comments

5th May 2013

Palindrome Partitioning II

Palindrome Partitioning II [http://leetcode.com/onlinejudge#question_132]

Given a string s, partition s such that every substring of the partition is a palindrome.
Return the minimum cuts needed for a palindrome partitioning of s.

For example, given s = "aab", return 1 since the palindrome partitioning ["aa","b"] could be produced using 1 cut.

Solution

This is a variant of the problem in previous post [http://n00tc0d3r.blogspot.com/2013/05/palindrome-partitioning.html] .

In this case, it only asks for the minimum number of cuts. Thus, we don't need to keep track of all of the partitions. Instead, we only maintain an one-dimension array of minimum cuts in previous substrings.

> Cuts[i] = k, if the minimum number of cuts needed for s[i..len-1] is k.

```
private boolean[][] buildPalindromeTable(String s) {
  int len = s.length();
  // T[i][j] == true iff s[i..j] is a palindrome
  boolean[][] T = new boolean[len][len];
  for (int i=0; i<len; ++i) {
    T[i][i] = true;
    // odd case
    int l = i-1, r = i+1;
    while (l>=0 && r<len && s.charAt(l)==s.charAt(r)) {
      T[l--][r++] = true;
    }
    // even case
    l = i; r = i+1;
    while (l>=0 && r<len && s.charAt(l)==s.charAt(r)) {
      T[l--][r++] = true;
    }
  }
  return T;
}

public int minCut(String s) {
  int len = s.length();
  if (len <= 1) return 0;

  // build up a table of palindrome substrings
  // T[i][j] == true iff s[i..j] is a palindrome
  boolean[][] palindrome = buildPalindromeTable(s);

  // build up a table for minimum cuts of substrings
  // Cuts[i] = k means the minimum cuts needed for s[i..len-1] is k
  int[] Cuts = new int[len];
  Cuts[len-1] = 0;
  for (int i=len-2; i>=0; --i) {
    if (palindrome[i][len-1]) {
      Cuts[i] = 0;
    } else {
      Cuts[i] = len - i;
      for (int j=i; j<len-1; ++j) {
        if (palindrome[i][j]) {
          Cuts[i] = Math.min(Cuts[i], 1+Cuts[j+1]);
        }
      }
    }
  }
  return Cuts[0];
}
```

Here we use DP twice:

- One for valid palindrome substrings
- and one for minimum number of cuts in previous substrings

We can actually perform the two DPs in the same loop which can simplify the code.

```
public int minCut(String s) {
  int len = s.length();

  // a table of palindrome substrings
  // T[i][j] == true iff s[i..j] is a palindrome
  boolean[][] palindrome = new boolean[len][len];

  // a table for minimum cuts of substrings
  // Cuts[i] = k means the minimum cuts needed for s[i..len-1] is k
  int[] Cuts = new int[len+1];

  Cuts[len] = -1;
  for (int i=len-1; i>=0; --i) {
    Cuts[i] = len - i;
    for (int j=i; j<len; ++j) {
      if (s.charAt(i)==s.charAt(j) && (j-i < 2 || palindrome[i+1][j-1])) {
        palindrome[i][j] = true;
        Cuts[i] = Math.min(Cuts[i], 1+Cuts[j+1]);
      }
    }
  }

  return Cuts[0];
}
```

Posted 5th May 2013 by Sophie

Labels: Array, DP, Java, String

1 View comments

1st May 2013

Palindrome Partitioning

Palindrome Partitioning [http://leetcode.com/onlinejudge#question_131]

Given a string s, partition s such that every substring of the partition is a palindrome.
Return all possible palindrome partitioning of s.

For example, given s = "aab", return [["aa","b"], ["a","a","b"]]

Solution - DP+Recursion

First, use DP to build up a table for all palindrome substrings such that T[i][j] == true if s[i..j] is a palindrome.

Then generate all partitions using recursion.
To avoid duplicate results, only generate a new partition if hitting a palindrome.
Suppose s[0..i] is a palindrome. All partitions containing (starting with) this substring are

s[0..i] + all partions of s[i+1..s.len-1].

```
/* build up a table
 * T[i][j] == true if s[i..j] is a palindrome
 */
private boolean[][] palindromeTable(String s) {
  boolean[][] T = new boolean[s.length()][s.length()];
  // single-char word is a palindrome
  for (int i=0; i<s.length(); ++i) T[i][i] = true;
  // multi-char words
  for (int i=1; i<s.length(); ++i) {
    // even
    int l = i-1, r = i;
    while (l >= 0 && r < s.length() && s.charAt(l) == s.charAt(r)) {
      T[l--][r++] = true;
    }
    // odd
    l = i-1; r = i+1;
    while (l >= 0 && r < s.length() && s.charAt(l) == s.charAt(r)) {
      T[l--][r++] = true;
    }
  }
  return T;
}

/* return all partitions of substring s[l..(s.len-1)] */
private void partitionHelper(String s, int l, boolean[][] T, ArrayList<ArrayList<String>> results) {
  if (l == s.length()) {
    results.add(new ArrayList<String>());
  }
  for (int i=l; i<s.length(); ++i) {
    if (T[l][i]) {
      ArrayList<ArrayList<String>> res = new ArrayList<ArrayList<String>>();
      partitionHelper(s, i+1, T, res);
      for (ArrayList<String> partition : res) {
        partition.add(0, s.substring(l, i+1));
      }
      results.addAll(res);
    }
  }
}

/* return all partitions of string s */
public ArrayList<ArrayList<String>> partition(String s) {
  ArrayList<ArrayList<String>> results = new ArrayList<ArrayList<String>>();
  partitionHelper(s, 0, palindromeTable(s), results);
  return results;
}
```

Solution - DP+DP

The algorithm above involves redundant calculations in the recursions. Instead of using a recursion, we can calculate the partitions backwards and store all previous results.

So, we create a table for previous results such that preRes[i] contains all of the partitions of s[i..len-1].

```
/* return all partitions of string s */
public ArrayList<ArrayList<String>> partition(String s) {
  // build up a table for palindrome substrings
  boolean[][] T = palindromeTable(s);

  // this map is used to store previous results
  // preRes.get(i) is all partitions of s[i..len-1]
  HashMap<Integer, ArrayList<ArrayList<String>>> preRes =
      new HashMap<Integer, ArrayList<ArrayList<String>>>();

  for (int i=s.length()-1; i>=0; --i) {
    ArrayList<ArrayList<String>> partitions = new ArrayList<ArrayList<String>>();
    for (int j=i; j<s.length(); ++j) {
      if (T[i][j]) {
        if (j+1 == n) {
          ArrayList<String> pp = new ArrayList<String>();
          pp.add(s.substring(i, j+1));
          partitions.add(pp);
        } else {
          for (ArrayList<String> p : preRes.get(j+1)) {
            ArrayList<String> pp = new ArrayList<String>();
            pp.add(s.substring(i, j+1));
            pp.addAll(p);
            partitions.add(pp);
          }
        }
      }
    }
    preRes.put(i, partitions);
  }

  return preRes.get(0);
}
```

Posted 1st May 2013 by Sophie

Labels: Array, DFS, DP, Java, Recursion, String

4 View comments

30th April 2013

Palindrome Number

Palindrome Number [http://leetcode.com/onlinejudge#question_9]

Determine whether an integer is a palindrome. Do this without extra space.

Solution

Start with special cases first:

- Negative numbers? No. They have a negative sign.
- Single digit numbers? Yes, of course.

The special property of a palindrome number is that its reverse equals to itself.

```
public boolean isPalindrome(int x) {
  if (x < 0) return false;
  // make a reverse
  long rev = 0, origin = x;
  while (x > 0) {
    rev = x%10 + rev*10;
    x /= 10;
  }
  return (origin == rev);
}
```

But we have to use a larger type to avoid overflow if we simply reverse a number.
E.g. 2147483647, which is the max value of int, and its reverse is obviously greater than itself.

Alternatively, we compare digit by digit from the two ends, without saving digits into a string. ("Do not use extra space"!)

```
public boolean isPalindrome(int x) {
  if (x < 0) return false;

  // what is the highest digit?
  int div = 10; // start from 10 since single digit number is palindrome
  while (x / div >= 10) div *= 10;

  // is it palindrome?
  while (x >= 10) {
    int lDigit = x / div;
    int rDigit = x % 10;
    if (lDigit != rDigit) return false;
    x = (x % div) / 10;
    // lower the highest digit
    div /= 100;
  }

  return true;
}
```

Posted 30th April 2013 by Sophie

Labels: Array, Java

2 View comments

30th April 2013

Next Permutation

Next Permutation [http://leetcode.com/onlinejudge#question_31]

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.
If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

The replacement must be in-place, do not allocate extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.
1,2,3 → 1,3,2 → 2,1,3 → 2,3,1 → 3,1,2 → 3,2,1 → 1,2,3
1,1,5 → 1,5,1 → 5,1,1 → 1,1,5
2,2,7,5,4,3,2,2,1 → 2,3,1,2,2,2,4,5,7

Solution

First thing first, we need to know WHAT is supposed to be the "next" permutation.

Let's take a closer look at the examples.

```
2,2,     7,5,4,3,2,2,1
  |   This part was in descending order and got rearranged to ascending order.
  |
 Why 3? 3 is the next greater number than 2 from the rest of the array.
```

So, given an array of numbers, to find the next lexicographical permutation,

- Find the longest descending tail of the array and reverse them into ascending order.
- If there is any number ahead of the descending tail, it should be replace with the first greater number in the tail.

The algorithm below runs in time O(n) and takes O(1) space. We reverse the tail rather than sort them, since we already know they are in descending order. Some sorting algorithms may take O(nlogn) or O(n^2) time to sort a reverse ordered array rather than O(n). See wiki [http://en.wikipedia.org/wiki/Sorting_algorithm#Summaries_of_popular_sorting_algorithms] for more fun about sorting algorithms.

```
/* O(1) -- swap the two elements in the array, given the indexes */
private void swap(int[] num, int a, int b) {
  int temp = num[a];
  num[a] = num[b];
  num[b] = temp;
}

/* O(n) -- reverse the array, given a range */
private void reverse(int[] num, int l, int r) {
  while (l < r)  swap(num, l++, r--);
}
```

```
/* O(n) -- find the next permutation */
public void nextPermutation(int[] num) {
  // find longest descending tail
  int cur = num.length - 1;
  while (cur > 0 && num[cur-1] >= num[cur]) --cur;
  reverse(num, cur, num.length-1);
  if (cur > 0) {
    // insert cur-1 to the right spot
    int next = cur;
    cur -= 1;
    while (num[next] <= num[cur]) ++next;
    swap(num, next, cur);
  }
}
```

Posted 30th April 2013 by Sophie

Labels: Array, Java, Sort

0 Add a comment

29th April 2013

N-Queens

N-Queens [http://leetcode.com/onlinejudge#question_51]

The n-queens puzzle is the problem of placing n queens on an n�n chessboard such that no two queens attack each other.

[http://4.bp.blogspot.com/-Kx3qkpYb6nk/UX4T_ygBT8I/AAAAAAAAEcg/ngtIbaU7WXQ/s1600/8-queens.png]

Given an integer n, return all distinct solutions to the n-queens puzzle.
Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space respectively.

For example, there exist two distinct solutions to the 4-queens puzzle:

[
[".Q..", // Solution 1
"...Q",
"Q...",
"..Q."],

["..Q.", // Solution 2
"Q...",
"...Q",
".Q.."]
]

Solution

Bit Manipulations give us an elegant solution. Idea is borrowed from Matrix67 [http://www.matrix67.com/blog/archives/266] (if you read Chinese :). More Bit manipulation funs can be found in Tutorial I [http://sophie-notes.blogspot.com/2013/02/bit-manipulation-tutorial-i.html] and Tutorial II [http://sophie-notes.blogspot.com/2013/02/bit-manipulation-tutorial-i.html] .

This can dramatically reduce time complexity since in each recursion, it only takes O(1) time to find out a candidate position. Thus, the total time complexity is the total number of solutions times the size of the board (since we need to convert binary numbers to strings). The worst case running time is O(n^(n+2)) where we might test most of potential positions. In most cases, this algorithm fast fails on ineligible positions.

Key ideas are:

- In a N-Queen solution, no two queens lie on the same row/column. That said, if we count XOR of all rows, it must give us (1 1 1 ... 1), n of 1's.
  So, we can use O(1) time to test whether a given set of rows is a valid solution.
- In k-th recursion, we cannot place on the rows of existing queens, nor does left/right diagonals of existing queens (OR current row and shift one left/right).
  - OR all previous rows gives us all existing queens;
  - Shift left/right current row by 1 gives us its diagonal positions on the next row;
  - OR current row with known left/right diagonal positions on the current line and then shift left/right gives us all diagonal positions of existing queens.

```
private ArrayList<String[]> convertSolution(long[] rows, ArrayList<String[]> results) {
  String[] res = new String[rows.length];
  for (int i=0; i<rows.length; ++i) {
    res[i] = Long.toBinaryString(rows[i])
          .replace('0', '.')
          .replace('1', 'Q');
    while (res[i].length() < rows.length) res[i] = '.' + res[i];
  }
  results.add(res);
  return results;
}

private ArrayList<String[]> solveNQueensHelper(long[] rows, int cur,
    long row, long lDiagonal, long rDiagonal,
    ArrayList<String[]> results) {
  long validator = (1 << rows.length) - 1; // (1 1 1 ... 1), n of 1's
  if (row == validator) { // find a solution
    convertSolution(rows, results);
  } else {
    long candidates = ((~(row | lDiagonal | rDiagonal)) & validator);
    while (candidates > 0) { // find potential positions
      // pick up lowest bit
      long pos = (candidates & (0 - candidates));
      // remove it from candidates
      candidates -= pos;
      // add to result row array
      rows[cur] = pos;
      solveNQueensHelper(rows, cur+1, (row | pos),
        ((lDiagonal | pos) << 1), ((rDiagonal | pos) >> 1),
        results);
    }
  }
  return results;
}

public ArrayList<String[]> solveNQueens(int n) {
  return solveNQueensHelper(new long[n], 0, 0, 0, 0,
        new ArrayList<String[]>());
}
```

N-Queens Solution Size [http://leetcode.com/onlinejudge#question_52]

Follow up for above N-Queens problem.
Now, instead outputting board configurations, return the total number of distinct solutions.

Solution

Since we don't need to keep the solutions, the algorithm can be simplified a lot.

The total number of solutions to N-Queens problem is O(n^n). So this algorithms has similar time complexity as the above one. The only difference is that there is no need to convert binary numbers to strings which takes O(n^2) time.

```
private int solveNQueensHelper(int n, long row, long lDiagonal, long rDiagonal) {
  int solutions = 0;
  long validator = (1 << n) - 1; // (1 1 1 ... 1), n of 1's
  if (row == validator) { // find a solution
    solutions = 1;
  } else {
    long candidates = ((~(row | lDiagonal | rDiagonal)) & validator);
    while (candidates > 0) { // find potential positions
      // pick up lowest bit
      long pos = (candidates & (0 - candidates));
      // remove from candidates
      candidates -= pos;
      solutions += solveNQueensHelper(n, (row | pos),
              ((lDiagonal | pos) << 1), ((rDiagonal | pos) >> 1));
    }
  }
  return solutions;
}

public int totalNQueens(int n) {
  return solveNQueensHelper(n, 0, 0, 0);
}
```

Posted 29th April 2013 by Sophie

Labels: BitManipulation, DFS, Java, Matrix, Recursion

0 Add a comment

22nd April 2013

Multiply Strings

Multiply Strings [http://leetcode.com/onlinejudge#question_43]

Given two numbers represented as strings, return multiplication of the numbers as a string.
Note: The numbers can be arbitrarily large and are non-negative.

Solution

Since the numbers can be arbitrarily large, we cannot convert a string number into an integer or a long. We need to convert them to an integer array where each element is one digit of the original string number. E.g. "12345" -> {1, 2, 3, 4, 5}.

We also need such an integer array for the result.
How long could it be?
Think about it. We know 999 * 999 < 999 * 1000 = 999000. That said, the length of the result array is the sum of the lengths of the two integer arrays.

At the end, we also need to trim prefix 0's, i.e. "090" -> "90".

Note that it is not necessary to have two integer arrays for the input string number. We can do the conversion on the fly (i.e. when performing multiplication for the two digits). Here we use space to save some redundant conversion time.

```
public String multiply(String num1, String num2) {
  if (num1.equals("0") || num2.equals("0")) return "0";
  int l1 = num1.length(), l2 = num2.length();
  int[] n1 = new int[l1];
  int[] n2 = new int[l2];
  int[] res = new int[l1+l2];
  // convert num1 to number array
  for (int i=0; i<l1; ++i) {
    n1[i] = num1.charAt(i) - '0';
  }
  // convert num2 to number array
  for (int i=0; i<l2; ++i) {
    n2[i] = num2.charAt(i) - '0';
  }
  // multiply into number array
  for (int i=0; i<l1; ++i) {
    for (int j=0; j<l2; ++j) {
      res[i+j+1] += n1[i]*n2[j];
    }
  }
  // convert back to string
  StringBuilder ss = new StringBuilder();
  for (int k=l1+l2-1; k>=0; --k) {
    ss.append((char)(res[k] % 10 + '0'));
    if (k>0) res[k-1] += res[k] / 10;
  }
  // trim 0's
  int count = ss.charAt(ss.length()-1)=='0' ? 1 : 0;
  String s = ss.reverse().substring(count, ss.length());
  return (s.isEmpty()) ? "0" : s;
}
```

Better Solution

In the above algorithm, we use the "natural" way to compute the multiplication.

E.g. given 11111 and 234,

```
    1 1 1 1 1
*       2 3 4
    4 4 4 4 4
  3 3 3 3 3
2 2 2 2 2
2 5 9 9 9 7 4
```

That said, we first compute n2[0] * n1[0 .. 4], and then compute n2[1] * n1[0 .. 4], ..., and then perform sum from digit 0 to digit l1+l2.

What if we do it vertically? Think out of box!

Notice that

- digit 0 of result is n1[0]*n2[0]
- digit 1 is n1[0]*n2[1] + n1[1]*n2[0]
- digit i is sum_(k=0~i) { n1[k]*n2[i-k] }

With these information in hand, we don't need to maintain an integer array for the result. We can convert the result digit into char directly.

```
public String multiply(String num1, String num2) {
  if (num1.equals("0") || num2.equals("0")) return "0";
  int l1 = num1.length(), l2 = num2.length();
  int[] n1 = new int[l1];
  int[] n2 = new int[l2];
  // convert num1 to number array reversely
  for (int i=0; i<l1; ++i) {
    n1[l1-i-1] = num1.charAt(i) - '0';
  }
  // convert num2 to number array reversely
  for (int i=0; i<l2; ++i) {
    n2[l2-i-1] = num2.charAt(i) - '0';
  }
  // multiply into digit by digit
  int sum = 0;
  StringBuilder ss = new StringBuilder();
  for (int i=0; i<l1+l2-1; ++i) {
    for (int j=0; j<=i; ++j) {
      if (j < l1 && i-j < l2)
        sum += n1[j] * n2[i-j];
    }
    ss.append((char)(sum % 10 + '0'));
    sum /= 10;
  }
  if (sum > 0) ss.append((char)(sum + '0'));
  String s = ss.reverse().toString();
  return (s.isEmpty()) ? "0" : s;
}
```

Both of the two algorithm runs in time O(m*n) and uses O(m+n) space, where m and n are the lengths of the the two input strings. But the second one is simpler. I like the second one!

Posted 22nd April 2013 by Sophie

Labels: Array, Java, String, StringBuilder

0 Add a comment

22nd April 2013 Minimum Window Substring

Minimum Window Substring [http://leetcode.com/onlinejudge#question_76]

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).

For example, S = "ADOBECODEBANC", T = "ABC",
Minimum window is "BANC".

Note:

- If there is no such window in S that covers all characters in T, return the emtpy string "".
- If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in S.

Solution

Since it requires a O(n) solution, first we need to be able to determine whether a char is in T or not in O(1) time. Obviously, hash map can do that. We can build up a hash map for chars in T and also use it to keep track of the occurrence we've found in S.

Second, we need to let the window sliding towards the right end, instead of computing all possible windows in S, which would be at least O(n^2). We can use two pointers for the window, left and right.

- At the beginning, left and right are both at index 0.
- Move right to next char.
    - If it hits a char in T and we have not visited before, increase the occurrence in the hash map and increase a counter.
    - If this is the first hit, we should move left to here.
    - If we have multiple occurrences of the left char in the current window, we should advance left towards right.
- If counter equals to the size of T, we found a window that contains all chars in T. Update minimum window if this one is smaller.

So far we use one hash map and increase the occurrence if it was 0. What if T has duplicate characters?
We need two hash maps, one contains the number of occurrences of each char in T; and the other is for the number we have found.

Now, every time when we found a char in T, we increase the occurrences of hasFound hash map. Also, if hasFound(left) > needFind(left), we need to advance left.

In this algorithm, each of the two pointers iterates through S once and performs O(1) operations in each iteration. The total running time is O(n), where n is the length of S. And it takes O(m) spaces for the two hash maps, where m is the length of T.

Noticing that when advancing the right pointer, we already know which char of S is in T and which is not. So, if we store the indices of those char of S that are in T, then we can directly advance left pointer to the next spot. This will not change the big-o time though.

```
/*
 * Two pointers: one for window_left and one for window_right
 * As moving to the right, we know which char is in T,
 * store the index into an array so that left point can directly
 * jump to next spot.
 */
public String minWindow(String S, String T) {
  int ss = S.length(), tt = T.length();
  // build up hashmap for T and also keep track of occurrence
  HashMap<Character, Integer> needFind = new HashMap<Character, Integer>();
  HashMap<Character, Integer> hasFound = new HashMap<Character, Integer>();
  for (int i=0; i<tt; ++i) {
    hasFound.put(T.charAt(i), 0);
    if (needFind.containsKey(T.charAt(i))) {
      needFind.put(T.charAt(i), needFind.get(T.charAt(i))+1);
    } else {
      needFind.put(T.charAt(i), 1);
    }
  }

  // keep found T-letters in an array to avoid revisit non-T-letters when forwarding left
  ArrayList<Integer> nexts = new ArrayList<Integer>();
  // window = S[nexts.get(left), S[right]]
  int right = 0, found = 0, left = 0;
  // sliding window as needed
  String window = "";
  int winSize = Integer.MAX_VALUE;
  while (right < S.length()) {
    char c = S.charAt(right);
    if (!needFind.containsKey(c)) { // not a match
      ++right;
      continue;
    }

    nexts.add(right);
    ++right;
    hasFound.put(c, hasFound.get(c)+1);
    if (hasFound.get(c) <= needFind.get(c)) ++found;

    if (found >= tt) { // got a window
      // forward left?
      char ll = S.charAt(nexts.get(left));
      while (hasFound.get(ll) > needFind.get(ll)) {
        hasFound.put(ll, hasFound.get(ll)-1);
        ++left;
        ll = S.charAt(nexts.get(left));
      }
      // smaller window?
      if (right - nexts.get(left) < winSize) {
        winSize = right - nexts.get(left);
        window = S.substring(nexts.get(left), right);
      }
    }
  }
  return window;
}
```

Posted 22nd April 2013 by Sophie

Labels: Array, Hash, Java, String

1 View comments

21st April 2013

Sliding Window Maximum

Sliding Window Maximum [http://discuss.leetcode.com/questions/133/sliding-window-maximum]

A long array A[] is given to you. There is a sliding window of size w which is moving from the very left of the array to the very right. You can only see the w numbers in the window. Each time the sliding window moves rightwards by one position. Following is an example: The array is [1 3 -1 -3 5 3 6 7], and w is 3.

```
Window position                Max
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

Input: A long array A[], and a window width w
Output: An array B[], B[i] is the maximum value of from A[i] to A[i+w-1]
Requirement: Find a good optimal way to get B[]

Solution

To get maximal of current window, it can take O(w) or O(logw) time. But we also need to keep track of the maximals as window sliding from left to right.

The first thought might be heap.
By maintaining a heap for all numbers in the window can give us a O(nlogw)-time solution, where

- building up a heap for initial window takes time O(wlogw)
- when window moves to the next number, each insertion and deletion take time O(logw) and there are n-w moves in total.
- after updating the heap, findMax only takes time O(1) since we know the top of heap is the largest.

So, if w << n, the performance of this solution is good, close to O(n); but if w is not that small, say w = n/3 or n/4, the running time goes up to O(nlogn).

All we need is to keep track of the maximals. Do we really need a heap that performs unnecessary sorting for all numbers in the window?
The answer is NO.

We can use a Deque [http://docs.oracle.com/javase/6/docs/api/java/util/Deque.html] which allow insertions/deletions on both ends. For a Deque implemented by Circular Array/Buffer [http://en.wikipedia.org/wiki/Circular_buffer] or Double Linked List [http://en.wikipedia.org/wiki/Double_linked_list] , the basic insert/delete operations run in constant time.

Back to the problem.

What shall we keep in the deque?
If we keep all numbers in the window into the queue, each time when we insert a new number and remove a old one, we need to figure out where to insert so as to maintain the numbers in order. That's not easy...

Actually, we do not need to keep all numbers. For example, suppose numbers in a window of size 4 are

> [1, 3, -1, 2], ...

Obviously, no matter what next numbers are, 1 and -1 are never going to be a maximal as the window moving. The queue should look like [3, 2] in this case.

So, to maintain the queue in order,

- When moves to a new number, iterate through back of the queue, removes all numbers that are not greater than the new one, and then insert the new one to the back.
- findMax only need to take the first one of the queue.
- To remove a number outside the window, only compare whether the current index is greater than the front of queue. If so, remove it.

By doing these, we guarantee that the first element of the queue is the maximal and all numbers following it are smaller and came later then it. That said, findMax runs in constant time. Noticing that we only insert and delete each number once. Assuming the insert/delete operations of the Deque run in O(1) time, the total running time for this algorithm is O(n) and it takes at most O(w) space!

Isn't it simple and beautiful?!

```
/*
 * Given an array of numbers and a sliding window, find out the maximal
 * number within the window as its moving.
 * @param nums the array of numbers.
 * @param window the size of the sliding window.
 * @return an array of window maximals, i.e. B[i] is the maximal of A[i, i+w).
 */
public int[] windowMax(int[] nums, int window) {
  int w = (nums.length < window) ? nums.length : window;
  // A deque allows insertion/deletion on both ends.
  // Maintain the first as the index of maximal of the window
  // and elements after it are all smaller and came later than the first.
  Deque<Integer> que = new ArrayDeque<Integer>();

  // initialize window
  int i=0;
  while (i<w) {
    while (!que.isEmpty() && nums[que.getLast()] <= nums[i]) {
      que.removeLast();
    }
    que.addLast(i++);
  }

  // sliding window
  int[] max = new int[nums.length - w + 1];
  max[i-w] = num[que.getFirst()];
  while (i<nums.length) {
    // add new element
    while (!que.isEmpty() && nums[que.getLast()] <= nums[i]) {
      que.removeLast();
    }
    que.addLast(i);
    // remove old element if still in que
    if (!que.isEmpty() && i-w >= que.getFirst()) {
      que.removeFirst();
    }
    // get maximal
    ++i;
    max[i-w] = num[que.getFirst()];
  }

  return max;
}
```

Posted 21st April 2013 by Sophie

Labels: Deque, Heap, Java

8 View comments

20th April 2013

Validate Binary Search Tree

Validate Binary Search Tree [http://leetcode.com/onlinejudge#question_98]

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

Solution - Stack

At the first glance, you may think

> For each node, verify that it is greater than its left child and smaller than its right child.

But it may not be true.
For example,

```
      5
     / \
    3   6
       / \
      4   7
```

In this tree, each node itself satisfy the condition. But the tree is not a BST.

So, instead of simply checking its parent, we need to compare the current node with previous visited one.

A property of BST is that in-order traverse of BST returns a sorted array. That said, if we perform an in-order traverse of the given binary tree, the value of the previous visited node must be less than the current visited one.

```
public boolean isValidBST(TreeNode root) {
  // in-order traverse returns a sorted array
  TreeNode cur = root;
  Stack<TreeNode> stack = new Stack<TreeNode>();
  while (cur != null) {
    stack.push(cur);
    cur = cur.left;
  }
  TreeNode pre = null;
  while (!stack.empty()) {
    cur = stack.pop();
    if (pre != null && pre.val >= cur.val) return false;
    pre = cur;
    cur = cur.right;
    while (cur != null) {
      stack.push(cur);
      cur = cur.left;
    }
  }
  return true;
}
```

Solution - Recursion

We can also do in recursively by passing previous node all around.

Notice that in Java, this is kind of a dirty way:
since we cannot change the value of the passed in object (which is a "const pointer", see this post [http://sophie-notes.blogspot.com/2012/12/java-passing-arguments-into-methods.html]), we change the content of the object...

```
private boolean isValidBSTHelper(TreeNode root, TreeNode pre) {
  if (root == null) return true;
  if (!isValidBSTHelper(root.left, pre) || !(pre.val < root.val))  return false;
  pre.val = root.val;
  return isValidBSTHelper(root.right, pre);
}
public boolean isValidBST(TreeNode root) {
  // an in-order traverse should be a sorted array
  return isValidBSTHelper(root, new TreeNode(Integer.MIN_VALUE));
}
```

But this algorithm cannot handle the case where the left-mode child is Integer.MIN_VALUE.

We can also pass in pre as an array (with one single element) so as to modify its value.

```
private boolean isValidBSTHelper(TreeNode root, ArrayList<Integer> pre) {
  if (root == null) return true;

  if (!isValidBSTHelper(root.left, pre) || (!pre.isEmpty() && pre.get(0) >= root.val))
    return false;

  if (pre.isEmpty()) {
    pre.add(root.val);
  } else {
    pre.set(0, root.val);
  }
  return isValidBSTHelper(root.right, pre);
}
public boolean isValidBST(TreeNode root) {
  // an in-order traverse should be a sorted array
  return isValidBSTHelper(root, new ArrayList<Integer>());
}
```

Alternatively, we can pass in a valid range for the subtree. This way we can handle boundary cases and code looks more clean.

```
    private boolean validBSTRecur(TreeNode root, int low, int high) {
   if (root == null)  return true;
   if (root.val < low || root.val > high)  return false;
     return (validBSTRecur(root.left, low, root.val-1) && validBSTRecur(root.right, root.val+1, high))
}

public boolean isValidBST(TreeNode root) {
  return validBSTRecur(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
}
```

Posted 20th April 2013 by Sophie

Labels: BinarySearchTree, BinaryTree, Java, Tree

0 Add a comment

20th April 2013

Minimum Path Sum

Minimum Path Sum [http://leetcode.com/onlinejudge#question_64]

Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path.
Note: You can only move either down or right at any point in time.

Solution

This is a classic DP problem.

Suppose we have a m by n table for the sum results.

sum[i][j] is the minimum sum of all numbers from grid[0][0] to grid[i][j], inclusively.

Then the DP formular becomes

sum[i][j] = grid[i][j] + min(sum[i-1][j], sum[i][j-1])

Furthermore, noticing that we only need the neighbor to the left and the one above the current one. We can use just two rows and switch back and forth.

```
public int minPathSum(int[][] grid) {
  if (grid.length == 0 || grid[0].length == 0) return 0;

  int[][] sum = new int[grid.length][2];
  sum[0][0] = grid[0][0];
  // initialized the first row
  int row = 0;
  for (int j=1; j<grid.length; ++j) {
    sum[j][row] = grid[j][0] + sum[j-1][row];
  }
  // fill up the rest of table
  for (int i=1; i<grid[0].length; ++i) {
    // switch the two rows
    row = 1 - row;
    // initialize the first element in the row
    sum[0][row] = grid[0][i] + sum[0][1-row];
    for (int j=1; j<grid.length; ++j) {
      sum[j][row] = Math.min(sum[j-1][row], sum[j][1-row]);
      sum[j][row] += grid[j][i];
    }
  }

  return sum[grid.length-1][row];
}
```

Posted 20th April 2013 by Sophie

Labels: Array, DP, Java

0 Add a comment

18th April 2013

Minimum Depth of Binary Tree

Minimum Depth of Binary Tree [http://leetcode.com/onlinejudge#question_111]

Given a binary tree, find its minimum depth.
The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node. That said, it should return 0 for empty tree and return 1 for a tree with root node only.

Solutions

We have discussed how to find out maximum depth of binary tree [http://n00tc0d3r.blogspot.com/2013/04/maximum-depth-of-binary-tree.html] where both BFS and DFS work and perform similarly.

But for this problem, on average, BFS will perform better:

> BFS can stop as soon as it hits the first leaf and return the depth there.

The worst case running time of BFS is still O(n) time and O(n) space. But on average, it will be much faster.
Suppose the minimum depth is h*, the running time of BFS is O(2^h*-1) and using space O(2^h*-1).
For example, consider a tree whose max depth is 5 but min depth is 2. The total number of nodes in the tree is at least 6. But BFS only needs to iterate through the first two levels, i.e. at most 3 nodes.

The downside of DFS in this case is that it still needs to touch every node to find out the minimum depth. That means it will have the same time complexity, O(n), and the same space complexity, O(h), as the one for finding maximum depth.

We can improve the DFS algorithm by passing in current depth and current minimum depth. With the information in hand, we can prune subtrees if they are deeper than the current minimum depth.
The space complexity gets down to O(h*) and the time complexity goes down to O(2^h-1).
It might be a little bit slower than BFS but it takes less space.

```
/* BFS can stop when hitting the first leaf */
private int minDepthBFS(TreeNode root) {
  if (root == null) return 0;
  int depth = 0;
  Queue<TreeNode> que = new LinkedList<TreeNode>();
  que.offer(root);
  que.offer(null);
  while (!que.isEmpty()) {
    TreeNode cur = que.poll();
    if (cur == null) { // end of a level
      ++depth;
      que.offer(null);
    } else {
      if (cur.left == null && cur.right == null) return depth+1;
      if (cur.left != null) que.offer(cur.left);
      if (cur.right != null) que.offer(cur.right);
    }
  }
  return depth;
}

/* DFS still need to touch each node to find the min depth,
 * same performance as find-max-depth.
 */
private int minDepthDFS1(TreeNode root) {
  if (root == null) return 0;

  int left = minDepthDFS1(root.left);
  int right = minDepthDFS1(root.right);

  if (right == 0) return left+1;
  if (left == 0) return right+1;
  return Math.min(left, right)+1;
}

/* DFS can be improved a little bit by passing in the current min depth
 * and then it could prune the subtress deeper than minDep.
 */
private int minDepthDFS(TreeNode root, int depth, int minDep) {
  if (root == null || depth >= minDep) return depth;

  if (root.left == null && root.right == null) return depth+1;
  if (root.left != null) minDep = Math.min(minDep, minDepthDFS(root.left, depth+1, minDep));
  if (root.right != null) minDep = Math.min(minDep, minDepthDFS(root.right, depth+1, minDep));
  return minDep;
}

public int minDepth(TreeNode root) {
  return minDepthBFS(root);
  return minDepthDFS1(root);


  return minDepthDFS(root, 0, Integer.MAX_VALUE);
}
```

Posted 18th April 2013 by Sophie

Labels: BinaryTree, Java, Queue, Recursion, Tree

0 Add a comment

12th April 2013 Merge k Sorted Lists

Merge k Sorted Lists [http://leetcode.com/onlinejudge#question_23]

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

Solution - MergeSort

In previous post (Merge Sorted Lists/Arrays I [http://n00tc0d3r.blogspot.com/2013/04/merge-sorted-listsarrays-i.html]), we've discussed how to merge two sorted linked list into one with O(1) extra space. Here, we can use that as a subroutine to solve this problem:

> Merge lists 2 by 2.

E.g. Give 4 lists, l1, l2, l3, l4,

- merge l1 and l4 to l1
- merge l2 and l3 to l2
- merge l1 and l2 to l1

E.g. Give 5 lists, l1, l2, l3, l4, l5,

- merge l1 and l5 to l1
- merge l2 and l4 to l2
- merge l1 and l3 to l1
- merge l1 and l2 to l1

```
public ListNode mergeKLists(ArrayList<ListNode> lists) {
  int last = lists.size() - 1;
  if (last < 0) return null;

  while (last > 0) {
    int cur = 0;
    while (cur < last) {
      lists.set(cur, mergeTwoLists(lists.get(cur++), lists.get(last--)));
    }
  }

  return lists.get(0);
}
```

Solution - HeapSort

Another way to solve the problem is to use heapsort.

Recall that in Merge Sorted Lists/Arrays I [http://n00tc0d3r.blogspot.com/2013/04/merge-sorted-listsarrays-i.html] we compare the two head nodes of the two lists to find a smaller one. Now we have k lists and thus we need to compare k head nodes to find the smallest one.

Since we need sort numbers dynamically, heapsort becomes a perfect fit.

- Build up a min heap of the k head nodes of given lists.
- Take the smallest one (i.e. the head of min heap) and add its next to heap.
- Repeat until all nodes in lists have been merged into one list.

Note:

- PriorityQueue [http://docs.oracle.com/javase/6/docs/api/java/util/PriorityQueue.html] is an unbounded queue based on heap.
- Since ListNode doesn't have its compare method, we need to provide our own comparator and pass it in during construction time.
  (return -1 for less-than; return 1 for greater-than; return 0 for equal.)
- PriorityQueue's initial capacity cannot be zero.
- If we can build up the priority queue over the given arraylist, it can reduce the space complexity from O(k) to O(1).

```
public ListNode mergeKLists(ArrayList<ListNode> lists) {
  if (lists == null || lists.isEmpty()) return null;

  Comparator<ListNode> comparator = new Comparator<ListNode>() {
    @Override
    public int compare(ListNode n1, ListNode n2) {
      if (n1.val < n2.val) return -1;
      if (n1.val > n2.val) return 1;
      return 0;
    }
  };

  PriorityQueue<ListNode> heap = new PriorityQueue<ListNode>(lists.size(), comparator);
  for (ListNode node : lists) {
    if (node != null) heap.add(node);
  }

  ListNode head=null, cur=null;
  while (!heap.isEmpty()) {
    if (head == null) {
      head = heap.poll();
      cur = head;
    } else {
      cur.next = heap.poll();
      cur = cur.next;
    }
    if (cur.next != null) heap.add(cur.next);
  }

  return head;
}
```

Analysis and Comparison

In mergesort solution, each original list has been touched O(logk) times, where k is the number of lists. That said, each node has been touched O(logk) times. So the total running time is O(nlogk), where n is the total number of nodes.
Note that, for odd cases, as shown above, some list(s) may be touched more than logk times, but still O(logk) times.

In heapsort solution, only head nodes of lists are kept in the min-heap.
Building up such a heap takes time O(klogk), each following insertion requires O(logk) time and there are at most n insertions in total. So the total running time is also O(nlogk).

Typically, n >> k, which means both algorithms performs better than O(nlogn).

In the case of n <= k, there must exist empty lists.
Knowing that:

- Merging one empty list to another one takes time O(1);
- Heap insertion for null values also takes O(1) time (no real insertion but a null check).

Suppose there are m **nonempty** lists. The total running time becomes

> (k - m)*O(1) + O(nlogm) = O(nlogm).

Therefore, the time complexity of both algorithms is O(nlogm), where n is the total number of nodes in lists and m is the number of nonempty lists.

But the space complexity is quite different:
In the former solution, we need to keep previous merged lists; while in the latter solution, we only need to keep a k-heap.
Suppose you are given k infinite streams instead of k lists, the heap solution will be a better choice.

Posted 12th April 2013 by Sophie

Labels: Java, LinkedList, Sort

10 View comments

11th April 2013

Merge Sorted Lists/Arrays

Merge Two Sorted Lists [http://leetcode.com/onlinejudge#question_21]

Merge two sorted linked lists and return it as a new list.
The new list should be made by splicing together the nodes of the first two lists.

Solution

Merging two linked list is not difficult. Basically, we need two extra pointers:

- root of the resulting list that will never be moved
- current pointer that move along the two list and link nodes to its end

Here is how they work:

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
  if (l1 == null) return l2;
  if (l2 == null) return l1;

  ListNode root;
  if (l1.val <= l2.val) {
    root = l1;
    l1 = l1.next;
  } else {
    root = l2;
    l2 = l2.next;
  }

  ListNode cur = root;
  while (l1 != null || l2 != null) {
    if (l2 == null || (l1 != null && l1.val <= l2.val)) {
      cur.next = l1;
      l1 = l1.next;
    } else {
      cur.next = l2;
      l2 = l2.next;
    }
    cur = cur.next;
  }

  return root;
}
```

Since this is an in-place merging, it takes O(m+n) time and O(1) space.

Adding a dummy head node can make the code look simpler.

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
  ListNode head = new ListNode(0);
  ListNode cur = head;

  while (l1 != null || l2 != null) {
    if (l2 == null || (l1 != null && l1.val <= l2.val)) {
      cur.next = l1;
      l1 = l1.next;
    } else {
      cur.next = l2;
      l2 = l2.next;
    }
    cur = cur.next;
  }

  return head.next;
}
```

We can also implement it using recursion. Note that recursion takes O(n) space for stacks.

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
  if (l1 == null) return l2;
  if (l2 == null) return l1;
  if (l1.val <= l2.val) {
    l1.next = mergeTwoLists(l1.next, l2);
    return l1;
  } else {
    l2.next = mergeTwoLists(l1, l2.next);
    return l2;
  }
}
```

Merge Sorted Array [http://leetcode.com/onlinejudge#question_88]

Given two sorted integer arrays A and B, merge B into A as one sorted array.

Note: You may assume that A has enough space to hold additional elements from B. The number of elements initialized in A and B are m and n respectively.

Solution

You may have saw this problem as a subroutine of merge sort in your algorithm textbook.

The key thing here is that inserting a number to an array could take O(n) since it requires to move all elements after that one. That said, if we iterate the two array from the beginnings, the total running time would be O(n^2).

To get it done in O(n) time, as we already know how many numbers there, we iterate from the ends and put each number to its final spot directly.

```
public void merge(int A[], int m, int B[], int n) {
  int tail = m + n - 1;
  --n; --m;
  while (n >= 0) {
    if (m <0 || A[m] <= B[n]) {
      A[tail--] = B[n--];
    } else {
      A[tail--] = A[m--];
    }
  }
}
```

Posted 11th April 2013 by Sophie

Labels: Array, Java, LinkedList, Sort

1 View comments

10th April 2013

Merge Intervals

Merge Intervals [http://leetcode.com/onlinejudge#question_56]

Given a collection of intervals, merge all overlapping intervals.

For example,
Given [1,3],[8,10],[2,6],[15,18],
return [1,6],[8,10],[15,18].

Solution

This problem looks like a subproblem in the insert interval [http://n00tc0d3r.blogspot.com/2013/03/insert-interval.html] problem we discussed before. BUT, the difference is that the intervals here may not be sorted.
(The problem description is vague and thus you need to ask your interviewer to clarify it.)

First, we need to sort the intervals based on their start points. And then we can apply similar merging strategies in insert

interval [http://n00tc0d3r.blogspot.com/2013/03/insert-interval.html] .
We provide two merging algorithms as follows.

- mergeInPlace uses O(1) space but runs in time O(n^2) in worse cases if we may remove intervals O(n) times. Using range remove (subList clear) may improve the performance but it could not change the Big-O time in worst case.
- mergeWithArray runs in time O(n) and also requires O(n) spaces.

```
/**
 * Definition for an interval.
 * public class Interval {
 *   int start;
 *   int end;
 *   Interval() { start = 0; end = 0; }
 *   Interval(int s, int e) { start = s; end = e; }
 * }
 */

private ArrayList<Interval> mergeInPlace(ArrayList<Interval> intervals) {
  int cur = 0, next = 1;
  while (next < intervals.size()) {
    if (intervals.get(cur).end >= intervals.get(next).start) {
      intervals.get(cur).end = Math.max(intervals.get(cur).end,
                     intervals.get(next).end);
      ++next;
    } else {
      if (next > cur) {
        intervals.subList(cur+1, next).clear();
      }
      ++cur;
      next = cur + 1; // after removal, index has changed
    }
  }
  if (cur < intervals.size()-1) {
    intervals.subList(cur+1, intervals.size()).clear();
  }
  return intervals;
}

private ArrayList<Interval> mergeWithArray(ArrayList<Interval> intervals) {
  ArrayList<Interval> results = new ArrayList<Interval>();
  Interval cur = null;
  for (int i=0; i<intervals.size(); ++i) {
    if (cur == null) {
      cur = intervals.get(i);
    } else if (cur.end >= intervals.get(i).start) { // overlapping
      cur.end = Math.max(cur.end, intervals.get(i).end);
    } else { // gap
      results.add(cur);
      cur = intervals.get(i);
    }
  }
  if (cur != null) results.add(cur);
  return results;
}

public ArrayList<Interval> merge(ArrayList<Interval> intervals) {
  // sort the intervals based on start points
  Comparator<Interval> comparator = new Comparator<Interval>() {
    @Override
    public int compare(Interval i1, Interval i2) {
      if (i1.start < i2.start) {
        return -1;
      } else {
        return 1;
      }
    }
  };
  Collections.sort(intervals, comparator);

  //return mergeWithArray(intervals);
  return mergeInPlace(intervals);
}
```

Posted 10th April 2013 by Sophie

Labels: Array, Java

0 Add a comment

9th April 2013

Median of Two Sorted Arrays

Median of Two Sorted Arrays [http://leetcode.com/onlinejudge#question_4]

There are two sorted arrays A and B of size m and n respectively. Find the median of the two sorted arrays.
The overall run time complexity should be O(log (m+n)).

Solution

The definition of median [http://en.wikipedia.org/wiki/Median] is the middle one from a sorted array. Note that if the length of the array is even, median is the mean of the two middle values.
E.g. The median of {3, 5, 9} is 5; while the median of {3, 5, 6, 9} is (5+6)/2 = 5.5.

Given two sorted arrays, we can do a merge sort in O(m+n) time and find out the median of the final array. But, here, it requires O(log (m+n)).

The log factor seems like a hint of binary search...

One way to do the binary search is to search over two arrays with two pointers. Suppose we start from medians of A and B, say A[i] and B[j]. If A[i] <= B[j], we know that

- There are i-1 elements smaller than A[i].
- There are n-j elements greater than B[j].
- The median is between A[i] and B[j], inclusively.

So, we can remove A[0 .. i-1] and B[j+1 .. n-1] and move the two pointers accordingly.
Since we run binary search over two arrays separately, the running time is O(logm + logn).
You can find a detailed description about this algorithm here [http://leetcode.com/2011/03/median-of-two-sorted-arrays.html] .

Now let's talk about the O(log (m+n)) solution.

The key thing is given A[i], we can test whether it is the median in O(1) time! (Check out this MIT handout [http://www2.myoops.org/course_material/mit/NR/rdonlyres/Electrical-Engineering-and-Computer-Science/6-046JFall-2005/30C68118-E436-4FE3-8C79-6BAFBB07D935/0/ps9sol.pdf] .)

> Let mid = (m+n)/2.
> A[i] is the median **if and only if** B[mid-i-1] < A[i] < B[mid-i].

Given this in hand, we only need one pointer and do binary search over the two arraies.
Be very careful about boundary check and special cases...

(Notice that in the following algorithm, we use & 0x1 to replace mod since mod operation is much more expensive then bit-operations.)

```
private double findMedian(int A[], int B[], int left, int right) {
  int m = A.length, n = B.length, mid = (m+n)/2;
  if (left > right) {
    return findMedian(B, A, Math.max(0, mid-m), Math.min(n-1, mid));
  }

  int i = (left+right) / 2, j = mid - i - 1;
  if (j >= 0 && A[i] < B[j]) // A[i] < median
    return findMedian(A, B, i+1, right);
  if (j < n-1 && A[i] > B[j+1]) // A[i] > median
    return findMedian(A, B, left, i-1);
  // found median (j<0 => mid-i-1 < 0 => i=mid; j>=n-1 => mid-i-1>=n-1 => i=mid-n)
  // m+n is odd
  if ( ((m+n) & 0x1) > 0 || (i <= 0 && (j < 0 || j >= n)))
    return A[i];
  // m+n is even
  if (j < 0 || j >= n)
    return (A[i] + A[i-1]) / 2.0;
  if (i <= 0)
    return (A[i] + B[j]) / 2.0;
  return (A[i] + Math.max(B[j], A[i-1])) / 2.0;
}

public double findMedianSortedArrays(int A[], int B[]) {
  int m = A.length, n = B.length, mid = (m+n)/2;
  if (m<n)
    return findMedian(A, B, Math.max(0, mid-n), Math.min(m-1, mid));
  else
    return findMedian(B, A, Math.max(0, mid-m), Math.min(n-1, mid));
}
```

Posted 9th April 2013 by Sophie

Labels: Array, Java, Sort

3 View comments

8th April 2013

Maximum Subarray

Maximum Subarray I [http://leetcode.com/onlinejudge#question_53]

Find the contiguous subarray within an array (containing at least one number) which has the largest sum. Return the sum.

For example, given the array [−2,1,−3,4,−1,2,1,−5,4], return 6 since the contiguous subarray [4,−1,2,1] has the largest sum = 6.

Solution

This problem is easy.

Keep tracking the sum of previous numbers. If it's getting lower after adding the current one, it tells us that this number should not be included in the largest-sum-subarray, so, reset the sum to 0.

```
public int maxSubArray(int[] A) {
  int maxSum = Integer.MIN_VALUE, sum = 0;
  for (int n : A) {
    sum += n;
    maxSum = Math.max(maxSum, sum);
    if (sum < 0) sum = 0; // reset the current sum
  }
  return maxSum;
}
```

Maximum Subarray II

Find the contiguous subarray within an array (containing at least one number) which has the largest sum. Return the subarray.

For example, given the array [−2,1,−3,4,−1,2,1,−5,4], return [4,−1,2,1] since it has the largest sum = 6.

Solution

Now besides the current sum, we also need to keep tracking of the corresponding start index for the sum. This start index need to be reset every time when we reset the sum.
If the sum is greater than known max sum, we need to update the max and also the range.

```
public int[] maxSubArray(int[] A) {
  int max = Integer.MIN_VALUE, sum = 0;
  int start = 0, end = -1, curS = 0;

  for (int i=0; i<A.length; ++i) {
    if (sum < 0) {
      sum = 0;
      curS = i; // reset the current start index
    }
    sum += A[i];
    if (sum > max) {
      max = sum;
      start = curS;
      end = i;
    }
  }

  if (start<=end) return Arrays.copyOfRange(A, start, end+1);
  else return A; // the result subarray cannot be empty unless A is empty
}
```

Posted 8th April 2013 by Sophie

Labels: Array, Java

1 View comments

8th April 2013

Maximum Depth of Binary Tree

Maximum Depth of Binary Tree [http://leetcode.com/onlinejudge#question_104]

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Solution - Recursion

The recursion algorithm is easy to come up:

- Perform a DFS (depth-first search [http://en.wikipedia.org/wiki/Depth-first_search]) on the tree and return the max depth.

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *   int val;
 *   TreeNode left;
 *   TreeNode right;
 *   TreeNode(int x) { val = x; }
 * }
 */
public int maxDepth(TreeNode root) {
  if (root == null) return 0;
  return 1 + Math.max(maxDepth(root.left), maxDepth(root.right));
}
```

This algorithm touch each node once and thus runs in time O(n), where n is the total number of nodes in the tree. It requires O(h) space for recursion, where h is the height of the tree.

Solution - Iteration

This problem can also be solved iteratively.

Here is a solution using BFS (breadth-first search [http://en.wikipedia.org/wiki/Breadth-first_search]).

```
public int maxDepth(TreeNode root) {
  int len = 0;
  LinkedList<TreeNode> que = new LinkedList<TreeNode>();
  if (root != null) {
    que.addLast(root);
    que.addLast(null); // add a special node for level breaker
  }

  while (!que.isEmpty()) {
    TreeNode cur = que.removeFirst();
    if (cur == null) { // finish one level
      ++len;
      if (!que.isEmpty()) que.addLast(null);
    } else {
      if (cur.left != null) que.addLast(cur.left);
      if (cur.right != null) que.addLast(cur.right);
    }
  }

  return len;
}
```

Similarly, this algorithm runs in time O(n). Since we keep all nodes in a level in the queue, in worst case, it requires O(n) spaces. (Think about a full binary tree with n/2 nodes at bottom level.)

Posted 8th April 2013 by Sophie

Labels: BFS, BinaryTree, DFS, Java, Recursion, Tree

2 View comments

8th April 2013

Summary: Counting and Probability

Counting

Permutation [http://en.wikipedia.org/wiki/Permutations]

A permutation, on the contrary, is an arrangement of several items selected from a given set.

P(n, k) = n! / (n-k)!

Combination [http://en.wikipedia.org/wiki/Combination]

Mathematically, a combination is to select several items from a given set where orders are not matter.

[http://upload.wikimedia.org/math/a/a/2/aa2dd943c643e3045db02f71a126f0fc.png]

or

[http://upload.wikimedia.org/math/1/9/2/1928f752016eeb2c94f27269a14f7f47.png]

Useful properties:

- C(n, k) = C(n, n-k)
- C(n, 0) = C(n, n) = 1
- C(n, k) = (n/k) * C(n-1, k-1) = (n/(n-k)) * C(n-1, k)
- C(n, k) = C(n-1, k-1) + C(n-1, k)

Binomial Coefficients

In binomial expansion, we have:

(x+y)^n = SUM_$_{\{k=0\}}$^n (C(n, k) * x^k * y^(n-k))

Let x = y = 1.

2^n = SUM_$_{\{k=0\}}$^n C(n, k)

Statistics

Probability

Given a sample space S, which containing events A1, A2, ..., An.

- P(A) >= 0 for any event A
- P(S) = 1

For any two sets A and B, the union of the two sets is a set of points that in A or in B or in both, whereas the intersection of the two sets is a set of points that in both A and B.

The probability of A or B is

[http://upload.wikimedia.org/math/8/4/5/845ddb87f1592b81a16fed56c78323bf.png]

The probability of A and B is

[http://upload.wikimedia.org/math/b/0/5/b055f58770da7bed882240ee4823db35.png]

The conditional probability of A given B is

[http://upload.wikimedia.org/math/0/2/2/0227e1b934aae89737b49187778bd8a8.png]

Bayes' Theorem [http://en.wikipedia.org/wiki/Bayes%27_theorem]

Bayes' theorem connects conditional probabilities to their inverses.

[http://upload.wikimedia.org/math/3/e/6/3e643c8f00f16a170f3be87f0116b9cb.png]

It can be extended as follows, using the law of total probability.

[http://upload.wikimedia.org/math/2/6/5/265a8f07efbaa9bac9f7462d45f9de42.png]

Discrete Random Variables [http://en.wikipedia.org/wiki/Discrete_random_variable]

A discrete random variable X is a variable whose value is from a countable sample space S, each with an associated probability.

A probability mess function is a function that gives the probability of X being exactly equal to some specified value x.

[http://upload.wikimedia.org/math/d/9/7/d9702ef9212728e363cdadbc2ae6b0ca.png]

Note that random variables can also be defined on continuous sample spaces which is characterized with a probability density function.

The expected value of a discrete random variable is

E[X] = SUM_x (x * P(X = x))

Properties of expected value

- E[X + Y] = E[X] + E[Y]
- E[X + a] = E[X] + a, where a is a constant.
- E[aX + Y] = aE[X] + E[Y], where a is a constant.
- E[XY] = E[X]E[Y], if X and Y are independent and each has a defined expectation.
- E[X | Y = y] = SUM_x (x * P(X = x | Y = y))
- E[X] = E[E[X|Y]]

The variance of a discrete random variable is

Var[X] = E[X^2] - E^2[X]

Properties of variance

- Var[aX] = a^2 * Var[X]
- Var[X + Y] = Var[X] + Var[Y], if X and Y are independent.

Distributions

Bernoulli trial

An experiment with a probability p of success and a probability of 1-p of failure.

Geometric distribution

A probability distribution satisfying P(X = k) = p * (1 - p)^(k-1).
The expectation of a geometric distribution is E[X] = 1/p.
The variance of a geometric distribution is Var[X] = (1 - p) / p^2.

Binomial distribution

A probability distribution satisfying P(X = k) = C(n, k) * p^k * (1 - p)^(n-k).
The expectation of a binomial distribution is E[X] = np.
The variance of a binomial distribution is Var[X] = np(1 - p).

Ref: Introduction to Algorithms

2nd edition (2003) [http://www.amazon.com/gp/product/0072970545/ref=as_li_ss_il?ie=UTF8&camp=1789&creative=390957&creativeASIN=0072970545&linkCode=as2&tag=n00tc0d3r-20] 3rd edition (2009) [http://www.amazon.com/gp/product/0262033844/ref=as_li_ss_il?ie=UTF8&camp=1789&creative=390957&creativeASIN=0262033844&linkCode=as2&tag=n00tc0d3r-20]

Posted 8th April 2013 by Sophie

Labels: Maths, Summary

0 Add a comment

7th April 2013

Maximum Rectangle

Maximum Rectangle [http://leetcode.com/onlinejudge#question_85]

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing all ones and return its area.

For example, given a rectangle as shown below, return 6.

```
0 0 0 1 1 0 1
0 1 0 1 0 1 1
```

```
1 0 1 1 1 0 0
0 1 1 1 1 0 1
```

Solution

Recall that we discussed Largest Rectangle in Histogram [http://n00tc0d3r.blogspot.com/2013/03/largest-rectangle-in-histogram.html] problem a few weeks ago. We can apply that idea to solve this problem, this is in 2D though.

First, we need to populate a table A for max length of vertical 1's. For each A[i][j], it is the maximum length of vertically continuous 1's ended at matrix[i][j], i.e. A[i][j] = k, meaning that matrix[i][j-k+1 .. j] are all 1's.

E.g. A such table for the given example in problem description is:

```
0 0 0 1 1 0 1
0 1 0 2 0 1 2
1 0 1 3 1 0 0
0 1 2 4 2 0 1
```

Then we can apply histogram algorithm on each row to find the maximum rectangle in the row.

```
private int maxRectangle(int[] histogram) {
  Stack<Integer> ss = new Stack<Integer>();
  int maxArea = 0, i = 0;
  while (i < histogram.length) {
    if (ss.isEmpty() || histogram[i] >= histogram[ss.peek()]) {
      ss.push(i++);
    } else {
      maxArea = Math.max(maxArea,
          histogram[ss.pop()]*(ss.isEmpty() ? i : (i - ss.peek() - 1)));
    }
  }
  while (!ss.isEmpty()) {
    maxArea = Math.max(maxArea,
        histogram[ss.pop()]*(ss.isEmpty() ? i : (i - ss.peek() - 1)));
  }
  return maxArea;
}

public int maximalRectangle(char[][] matrix) {
  int rows = matrix.length;
  if (rows == 0) return 0;
  int columns = matrix[0].length;
  // lenTable[i][j] = k means matrix[i][j] - matrix[i][j+k-1] are all '1's.
  int[][] lenTable = new int[rows][columns];
  // initialize the table
  for (int h=0; h<columns; ++h) {
    if (matrix[0][h] == '1') lenTable[0][h] = 1;
  }
  for (int w=1; w<rows; ++w) {
    for (int h=0; h<columns; ++h) {
      if (matrix[w][h] == '0') continue;
      lenTable[w][h] = lenTable[w-1][h] + 1;
    }
  }
  // find the max rect area
  int maxArea = 0;
  for (int i=0; i<rows; ++i) {
    maxArea = Math.max(maxArea, maxRectangle(lenTable[i]));
  }
  return maxArea;
}
```

This algorithm runs in time O(n*m), where n and m are the number of rows and columns in the matrix, respectively. It takes O(n*m) space for the table and O(m) space for stack.

Posted 7th April 2013 by Sophie

Labels: Array, DP, Java, Matrix, Stack

2 View comments

5th April 2013

Valid Parentheses

Valid Parentheses [http://leetcode.com/onlinejudge#question_20]

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

The brackets must close in the correct order, "()" and "()[]{}" are all valid but "(]" and "([)]" are not.
Note that no need to consider the priority of parentheses, brackets and curly parentheses here. That said, "([{}])" is valid.

Solution

Similar to the first solution in Longest Valid Parentheses [http://n00tc0d3r.blogspot.com/2013/04/longest-valid-parentheses.html] problem, we use a stack to keep track of open parentheses.
Each time when we hit a close parenthesis, the top of the stack has to be the matching parenthesis. Otherwise, this string is not a valid parentheses.

```
public boolean isValid(String s) {
  HashMap<Character, Character> map = new HashMap<Character, Character>();
  map.put(')', '(');
  map.put(']', '[');
  map.put('}', '{');

  Stack<Character> stack = new Stack<Character>();
  for (int i=0; i<s.length(); ++i) {
    char c = s.charAt(i);
    if (!map.containsKey(c)) {
      stack.push(c);
    } else if (stack.isEmpty() || stack.pop() != map.get(c)) {
      return false;
    }
  }

  return (stack.isEmpty());
}
```

This algorithm runs in time O(n) and takes O(n) spaces for the stack.

This algorithm can also be modified to use recursion to replace the for-loop. But we still need a stack to keep track of previous left parentheses.
Although recursion has its own stack but we can take advantage of it here. We have multiple types of parentheses and need to handle left and right ones differently: when hit right ones, pop out, which means we need to jump back and forth in the stack. So, we need to maintain a stack for left parentheses.

```
private boolean isValidHelper(String s, int cur, Stack<Character> stack) {
  if (cur == s.length()) return stack.isEmpty();
  char c = s.charAt(cur);
  switch (c) {
  case '(':
  case '[':
  case '{':
    stack.push(c);
    break;
  case ')':
    if (stack.isEmpty() || stack.pop() != '(') return false;
    break;
  case ']':
    if (stack.isEmpty() || stack.pop() != '[') return false;
    break;
  case '}':
    if (stack.isEmpty() || stack.pop() != '{') return false;
    break;
  }
  return isValidHelper(s, cur+1, stack);
}

public boolean isValid(String s) {
  Stack<Character> stack = new Stack<Character>();
  return isValidHelper(s, 0, stack);
}
```

The time and space complexities of this problem is the same as the above one.

Posted 5th April 2013 by Sophie

Labels: Java, Recursion, Stack

0 Add a comment

5th April 2013

Longest Valid Parentheses

Longest Valid Parentheses [http://leetcode.com/onlinejudge#question_32]

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.

For "(()", the longest valid parentheses substring is "()", which has length = 2.
Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.

Solution - O(n) space

To detect a valid parentheses substring, we use a stack to maintain left parentheses:

- Each time when hitting a '(', push its index to the stack.
- Each time when hitting a ')',
  - If the stack is not empty, i.e. there is a matching '(' for it, pop the match out of the stack. At this time, we know it is an end of a valid parentheses substring. Calculate the length of the current substring and compare it with the

    longest we've got.
- If the stack is empty, there is no matching '(' for it. Move on to next character.

The tricky part is how to calculate the length of the "current substring".
E.g. to differentiate "()(()" and "(()()", or "())()" and ")()()".

We need to track the position where the current substring started.
For case one, "()(()" and "(()()", the extra '(' must be in stack. So, after pop up a matching '(', the next top of the stack is right before the start position of the current substring.
For case two, "())()" and ")()()", we need to keep track of the last invalid ')'.

```
public int longestValidParentheses(String s) {
  int maxLen = 0, last = -1;
  Stack<Integer> lefts = new Stack<Integer>();
  for (int i=0; i<s.length(); ++i) {
    if (s.charAt(i)=='(') {
      lefts.push(i);
    } else {
      if (lefts.isEmpty()) {
        // no matching left
        last = i;
      } else {
        // find a matching pair
        lefts.pop();
        if (lefts.isEmpty()) {
          maxLen = Math.max(maxLen, i-last);
        } else {
          maxLen = Math.max(maxLen, i-lefts.peek());
        }
      }
    }
  }
  return maxLen;
}
```

This algorithm takes time O(n) and O(n) space for the stack.

A Wrong Solution...

Now let's try to get rid of stack.

We use an integer lefts to keep track of open '('s.

- Each time when we hit a close ')', decrease lefts by 1 and increase the length of current substring.
- Each time when we hit an invalide ')', reset length.

It looks good. But it cannot differentiate "(()()" and "()(()" since there is no way to keep track of positions of invalid '('!

Similarly, if we iterate the string from right to left and let ')' as an open parenthesis, it cannot differentiate "()())" and "())()".

What if we iterate the string twice, one going forwards and the other going backwards, and then pick up the smaller one? It seems to be able to fix the problem.

Unfortunately, it will return 6 for "()))()()(()"...

```
public int longestValidParentheses(String s) {
  int maxLen = 0, lefts = 0, len = 0;
  for (int i=0; i!=s.length(); ++i) {
    if (s.charAt(i)=='(') {
      ++lefts;
    } else {
      if (lefts>0) {
        // exist a matching
        --lefts;
        len += 2;
        maxLen = Math.max(maxLen, len);
      } else {
        // no matching
        len = 0;
      }
    }
  }
  return maxLen;
}
```

Solution - O(1) space

Double think about the iterate-twice solution.

When going from left to right,

- First, skip invalid ')'s
- Then, find the longest valid parentheses that ended with an invalid '(' or ')'

When going backwards,

- Skip invalid '('s
- Find the longest valid parentheses that ended with an invalid parenthesis

And then select the longer one from the two results.

By doing this, we can find the longest one without using extra spaces! (Well, we iterate string twice but it is still O(n) time. ;)

```
private int findValidParentheses(String s, int start, int end, int step, char cc) {
  int maxLen = 0, count = 0, len = 0;
  for (int i=start; i!=end; i+=step) {
    if (s.charAt(i)==cc) {
      ++count;
    } else {
      if (count>0) {
        // exist a matching
        --count;
        len += 2;
        if (count==0) maxLen = Math.max(maxLen, len);
      } else {
        // no matching
        len = 0;
      }
    }
  }
  return maxLen;
}

public int longestValidParentheses(String s) {
  return Math.max(findValidParentheses(s, 0, s.length(), 1, '('),
          findValidParentheses(s, s.length()-1, -1, -1, ')'));
}
```

Posted 5th April 2013 by Sophie

Labels: Java, Stack, String

0 Add a comment

3rd April 2013

Longest Substring Without Repeating Characters

Longest Substring Without Repeating Characters [http://leetcode.com/onlinejudge#question_3]

Given a string, find the length of the longest substring without repeating characters. For example, the longest substring without repeating letters for "abcabcbb" is "abc", which the length is 3. For "bbbbb" the longest substring is "b", with the length of 1.

Solution

To check whether there exists a prior occurrence of a character, we can setup a hash table which provide O(1) lookup time.

If we hit a recurrence of a character, it means that we find a substring without repeating characters (ended before this character). Suppose we start from s[i] and find out the s[j] is same as s[k], where k<j. That said, s[i..j-1] is a substring without repeats. That also tells us that there is no repeating characters between k+1 and j.

So, next step, we can start from k+1.

Another thing is that we may need to remove characters between i and k from the hash table since they are no longer in the current substring. Alternatively, if we keep the index to the latest occurrence of a character, we don't need to cleanup the hash table, instead, we compare the index with start pointer to see whether it is in the current substring.

E.g. given "abcabcbb", let start be 0 and move end from left to right.

```
a b c b a c b b
0 1 2 3
```

When end = 3, we know it is a repeating character. The current no-repeats substring is "abc". We then set start to be the one next to prior 'b', which is 'c', and move forward.

```
a b c b a c b b
0 1 2 3 4
```

When end = 4, although 'a' is already in the hash table, but its index was 0 which is not in the current substring. So, 'a' is not a repeating character in the current substring.

Here is the algorithm.

```
public int lengthOfLongestSubstring(String s) {
  int maxLen = 0;
  int start = 0, end = 0;
  // a map mapping a char to its prior index in s
  HashMap<Character, Integer> map = new HashMap<Character, Integer>();
  while (end < s.length()) {
    char cur = s.charAt(end);
    if (map.containsKey(cur) && map.get(cur) >= start) {
      // hit a recurrence
      maxLen = Math.max(maxLen, end-start);
      start = map.get(cur) + 1;
    }
    map.put(cur, end++);
  }
  maxLen = Math.max(maxLen, end-start);
  return maxLen;
}
```

Posted 3rd April 2013 by Sophie

Labels: Hash, Java, String

0 Add a comment

3rd April 2013

Longest Palindromic Substring

Longest Palindromic Substring [http://leetcode.com/onlinejudge#question_5]

Given a string S, find the longest palindromic substring in S. You may assume that the maximum length of S is 1000, and there exists one unique longest palindromic substring.

Solution

In previous post, valid palindrome [http://n00tc0d3r.blogspot.com/2013/04/valid-palindrome.html] , we use two pointers moving towards the middle. But here, we use two pointers to towards ends.

Why?
If we move from ends towards middle, we need to determine the two ends. There are C(n, 2) choices and for each choice, it takes O(n) time to check whether it is a palindrome. So, the total running time is O(n^3)!

While, if we move from middle towards ends, we only need to iterate through the list and expand from each character. That gives us O(n^2) total running time and only O(1) space.

```
private String findPalindrome(String s, int left, int right) {
  while (left>=0 && right<s.length() && s.charAt(left)==s.charAt(right)) {
    left--; right++;
  }
  return s.substring(left+1, right);
}

public String longestPalindrome(String s) {
  String longest = "";
  for (int i=0; i<s.length(); ++i) {
    String palindrome = findPalindrome(s, i, i);
    if (palindrome.length() > longest.length()) {
      longest = palindrome;
    }
  }
  for (int i=1; i<s.length(); ++i) {
    String palindrome = findPalindrome(s, i-1, i);
    if (palindrome.length() > longest.length()) {
      longest = palindrome;
    }
  }
  return longest;
}
```

We can optimize the solution a little bit: If the longest is already longer than the remaining part of the string, we can stop there.
This optimization will not change the order of the running time but it improves the actual running time.

```
public String longestPalindrome(String s) {
  int n = s.length();
  String longest = "";
  for (int i=0; i < s.length() - longest.length()/2; ++i) {
    int left = i, right = i;
    for (; left >= 0 && right < n && s.charAt(left) == s.charAt(right);
        --left, ++right);
    // If the odd case produces the longest and reaches an end,
    // we don't need to check the even case since it must be shorter.
    if (right-left-1 > longest.length()) {
      longest = s.substring(left+1, right);
      if (left < 0 || right == n)  continue;
    }

    left = i-1;  right = i;
    for (; left >= 0 && right < n && s.charAt(left) == s.charAt(right);
        --left, ++right);
    if (right-left-1 > longest.length()) {
      longest = s.substring(left+1, right);
    }
  }
  return longest;
}
```

Posted 3rd April 2013 by Sophie

Labels: Array, Java, String

2 View comments

2nd April 2013

Valid Palindrome

Valid Palindrome [http://leetcode.com/onlinejudge#question_125]

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

For example,
"A man, a plan, a canal: Panama" is a palindrome.
"race a car" is not a palindrome.

Note:
Have you consider that the string might be empty? This is a good question to ask during an interview.
For the purpose of this problem, we define empty string as valid palindrome.

Solution

We can use two pointers moving towards middle of string to perform the check.

Before we do that, two things need to be take care:

- remove non-alphanumeric characters (alphanumeric characters includes a-z, A-Z, and 0-9)
- change string to lower cases

```
/** Remove non-alphanumeric char and change to lowercase
 * ss = normalize(s) is equivalent to
 *  ss = s.toLowerCase().replaceAll("[^a-z0-9]", "");
 */
private String normalize(String s) {
  StringBuilder ss = new StringBuilder();
  for (int i=0; i<s.length(); ++i) {
    char c = s.charAt(i);
    // skip non-alphanumeric
    if ((c < 'a' || c > 'z') && (c < 'A' || c > 'Z') && (c < '0' || c > '9'))
      continue;
    // lowercase
    if (c >= 'A' && c <= 'Z') c = (char)(c - ('A' - 'a'));
    ss.append(c);
  }
  return ss.toString();
}

public boolean isPalindrome(String s) {
  String ss = normalize(s);
  int left=0, right=ss.length()-1;
  while (left < right) {
    if (ss.charAt(left++) != ss.charAt(right--)) return false;
  }
  return true;
}
```

This algorithm runs in time O(n) and also requires O(n) spaces to normalize the string.

We can also do it in place: skip the invalid characters on the fly. By doing that, the space complexity gets down to O(1) while the time complexity is the same O(n).

```
/** Determine if c is alphanumeric.
 * If unicode is allowed, we can simply use Character.isLetterOrDigit(c).
 */
private boolean isLetter(char c) {
  return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9');
}

public boolean isPalindrome(String s) {
  int left=0, right=s.length()-1;
  while (left < right) {
    char c1 = s.charAt(left);
    if (!isLetter(c1)) {
      ++left; continue;
    }

    char c2 = s.charAt(right);
    if (!isLetter(c2)) {
      --right; continue;
    }

    if (Character.toLowerCase(c1) != Character.toLowerCase(c2)) return false;
    ++left;
    --right;
  }
  return true;
}
```

Posted 2nd April 2013 by Sophie

Labels: Java, String, StringBuilder

0 Add a comment

1st April 2013

Longest Consecutive Sequence

Longest Consecutive Sequence [http://leetcode.com/onlinejudge#question_128]

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

For example, given [100, 4, 200, 1, 3, 2], where the longest consecutive elements sequence is [1, 2, 3, 4], return its length: 4.

Your algorithm should run in O(n) complexity.

Solution I - Union/Find

Since there is a constraint on the running time, we can't use any O(nlogn) sorting algorithms.

Bucket sort seems not helpful here: We can target at bucket(s) that have largest number of elements. It's possible the consecutive sequence across buckets, which implies that we still need to sort numbers in a bucket and its left and right neighbors. The worst case running time would be O(nlogn).

Ideally, if we could build up a hash map that maps each number to a range that contains the number, then keep track the max length of existing ranges would solve the problem. But this requires that each time when we merge two ranges, we have to update the associated range information of all numbers in the range. That maintenance work costs too much and give us a O(n^2) algorithm.

Do we really need to update range information of ALL numbers?
Notice that each time we merge two ranges, there must be NO overlapping between the two and they are supposed to be consecutive. So, we only need to maintain range information of boundary elements!
Or, we can say, each time we "union" two ranges together. We then use hash-map to perform the "find" operation.

This algorithm is different from merge-sort in that it doesn't need to compare elements of range A with elements of range B as we already know each range contains consecutive numbers.

```
/* Per definition of the map,
 * - l maps to the len of a left range where l itself is the rightmost element;
 * - r maps to the len of a right range where r is the leftmost element.
 * After merging left and right ranges into one,
 * - the original leftmost of left range becomes the new leftmost;
 * - the original rightmost of right range becomes the new rightmost.
 */
public int merge(HashMap<Integer, Integer> map, int l, int r) {
  int left = l - map.get(l) + 1;
  int right = r + map.get(r) - 1;
  int range = right - left + 1;
  map.put(left, range);
  map.put(right, range);
  return range;
}

public int longestConsecutive(int[] nums) {
  // Map each number to the maxLen of the range that contains the number.
  // Only need to maintain length of range boundaries
  // since each time we either merge from left or merge from right.
  HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
  int maxLen = (nums.length==0) ? 0 : 1;
  for (int num : nums) {
    if (!map.containsKey(num)) {
      map.put(num, 1);
      if (map.containsKey(num-1)) {
        maxLen = Math.max(maxLen, merge(map, num-1, num));
      }
      if (map.containsKey(num+1)) {
        maxLen = Math.max(maxLen, merge(map, num, num+1));
      }
    }
  }
  return maxLen;
}
```

This algorithm runs in time O(n) and takes O(n) spaces.

Solution II - Visit Flag

Another way to solve this problem is:

- Put all numbers into a hash table.

- Start from one of them (arbitrarily, not-visited), find the longest consecutive sequence containing this number and during the process, mask each one as visited.

By doing this, each number will be visited at most twice and thus the total running time is O(n).
This algorithm performs slightly better (constant factor) than the above one since the calculation steps on each number is simpler.

An implementation with HashSet:

```
private int findConsecutive(HashSet<Integer> set, int num, int step) {
  int len = 0;
  while (set.contains(num)) {
    ++len;
    set.remove(num);
    num += step;
  }
  return len;
}

public int longestConsecutive(int[] nums) {
  HashSet<Integer> set = new HashSet<Integer>();
  // find all
  for (int num : nums) {
    set.add(num);
  }
  // find longest seq
  int maxLen = 0;
  for (int num : nums) {
    if (set.contains(num)) {
      set.remove(num);
      int len = 1 + findConsecutive(set, num-1, -1);
      len += findConsecutive(set, num+1, 1);
      maxLen = Math.max(maxLen, len);
    }
  }
  return maxLen;
}
```

An implementation with HashMap:

```
private int findConsecutive(HashMap<Integer, Boolean> map, int num, int step) {
  int len = 0;
  while (map.containsKey(num)) {
    ++len;
    map.put(num, true);
    num += step;
  }
  return len;
}

public int longestConsecutive(int[] nums) {
  HashMap<Integer, Boolean> map = new HashMap<Integer, Boolean>();
  // find all
  for (int num : nums) {
    map.put(num, false);
  }
  // find longest seq
  int maxLen = 0;
  for (int num : nums) {
    if (!map.get(num)) {
      map.put(num, true);
      int len = 1 + findConsecutive(map, num-1, -1);
      len += findConsecutive(map, num+1, 1);
      maxLen = Math.max(maxLen, len);
    }
  }
  return maxLen;
}
```

Posted 1st April 2013 by Sophie

Labels: Array, Hash, Java, Sort

1 View comments

1st April 2013

Letter Combinations of a Phone Number

Letter Combinations of a Phone Number [http://leetcode.com/onlinejudge#question_17]

Given a digit string, return all possible letter combinations that the number could represent.
A mapping of digit to letters is just like the mapping on the telephone buttons.

For example, given digit string "23", return ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

Note:
Although the above answer is in lexicographical order, your answer could be in any order you want.

Solution

We can use similar ideas used to solve the Combinations [http://n00tc0d3r.blogspot.com/2013/01/combinations.html] problem. But this time, let's try to play it with iterations.

One thing need to notice is that digits 7 and 9 are different from others: it may represent four characters instead of three!

```
public ArrayList<String> letterCombinations(String digits) {
  // initialize an array of starting char for each digit
  char[] letters = new char[8];
  for (int i=0; i<6; ++i) {
    letters[i] = (char)('a'+i*3);
  }
  letters[6] = 't'; letters[7] = 'w';
  ArrayList<StringBuilder> builders = new ArrayList<StringBuilder>((int)Math.pow(3, digits.length()));
  for (int i=0; i<digits.length(); ++i) {
    int digit = (int)(digits.charAt(i) - '2');
    // if hit invalid input, return null
    if (digit < 0 || digit > 7) return null;
    if (builders.size() == 0) {
      for (int j=0; j<3; ++j) {
        StringBuilder ss = new StringBuilder();
        ss.append((char)(letters[digit]+j));
        builders.add(ss);
      }
      // digit '7' and '9' has 4 chars
      if (digit == 5 || digit == 7) {
        StringBuilder ss = new StringBuilder();
        ss.append((char)(letters[digit]+3));
        builders.add(ss);
      }
    } else {
      int size = builders.size();
      for (int j=0; j<size; ++j) {
        StringBuilder ss = builders.get(j);
        StringBuilder ss1 = new StringBuilder(ss.toString());
        StringBuilder ss2 = new StringBuilder(ss.toString());
        builders.add(ss1);
        builders.add(ss2);
        if (digit == 5 || digit == 7) {
          StringBuilder ss3 = new StringBuilder(ss.toString());
          ss3.append((char)(letters[digit]+3));
          builders.add(ss3);
        }
        ss.append((char)(letters[digit]));
        ss1.append((char)(letters[digit]+1));
        ss2.append((char)(letters[digit]+2));
      }
    }
  }
  // convert stringbuilders to strings
  ArrayList<String> results = new ArrayList<String>(builders.size());
  for (StringBuilder ss : builders) {
    results.add(ss.toString());
  }
  if (results.isEmpty()) results.add("");
  return results;
}
```

Update: Use an array of Strings to improve readability.

```
public ArrayList<String> letterCombinations(String digits) {
  String[] letters = {"abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};
  ArrayList<StringBuilder> builders = new ArrayList<StringBuilder>();
  ArrayList<String> results = new ArrayList<String>();
  for (int i=0; i<digits.length(); ++i) {
    int d = digits.charAt(i) - '2';
    // validate input
    if (d < 0 || d > 7) return results;
    // generate strings
    if (builders.isEmpty()) {
      for (int k=0; k<letters[d].length(); ++k) { // letters
        StringBuilder sb = new StringBuilder();
        sb.append(letters[d].charAt(k));
        builders.add(sb);
      }
      continue;
    }
    int curSize = builders.size();
    for (int j=0; j<curSize; ++j) { // existing prefixes
      StringBuilder sb = builders.get(j);
      for (int k=0; k<letters[d].length(); ++k) { // letters
        if (k==letters[d].length()-1) {
          sb.append(letters[d].charAt(k));
        } else {
          StringBuilder ss = new StringBuilder(sb.toString());
          ss.append(letters[d].charAt(k));
          builders.add(ss);
        }
      }
    }
  }
  // convert builders to strings
  for (StringBuilder sb : builders) results.add(sb.toString());
  // append empty string if no input
  if (results.size() == 0) results.add("");
  return results;
}
```

Update 2: If the digits are not very long, we don't need to use StringBuilder to reduce the extra time from append a string.

```
public ArrayList<String> letterCombinations(String digits) {
  String[] letters = {"abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};
  ArrayList<String> results = new ArrayList<String>();
  for (int i=0; i<digits.length(); ++i) {
    int d = digits.charAt(i) - '2';
    // validate input
    if (d < 0 || d > 7) return results;
    // generate strings
    int curSize = results.size();
    if (curSize == 0) {
      for (int k=0; k<letters[d].length(); ++k) { // letters
        results.add(new String(""+letters[d].charAt(k)));
      }
    }
    for (int j=0; j<curSize; ++j) { // existing prefixes
      String s = results.get(j);
      for (int k=0; k<letters[d].length(); ++k) { // letters
        if (k==letters[d].length()-1) {
          results.set(j, s+letters[d].charAt(k));
        } else {
          results.add(new String(s+letters[d].charAt(k)));
        }
      }
    }
  }
  // append empty string if no input
  if (results.size() == 0) results.add("");
  return results;
}
```

Posted 1st April 2013 by Sophie

Labels: Array, DP, Java, StringBuilder

0 Add a comment

31st March 2013

Summary: Dynamic Programming

Reference: by CLRS, Introduction to Algorithms (second edition) [http://www.amazon.com/gp/product/0072970545/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=0072970545&linkCode=as2&tag=n00tc0d3r-20]

When to use DP?

- An optimal solution to the problem can be built on optimal solutions to subproblems.

That said, DP might apply. It also might mean that Divide-and-Conquer (Recursion) or Greedy Algorithm applies, too.

- Subproblems are independent and have overlapping.

`independent`: The solution to one subproblem does NOT affect the solution to another subproblem of the same problem.
`overlapping`: If a recursive algorithm for the problem revisit the same subproblems over and over, we say that the problem has overlapping subproblems. In such cases, we may apply DP so as to solve each subproblem only once and then store it in a table where it can be looked up in O(1) time as needed.

Bottom-up or Top-down memorized?

Bottom-up DP algorithm

To solve a problem of A[1 .. n],

1. starts from the smallest subproblems such as A[1], A[2], ..., A[n], and store results into a table
2. then build up next "level" of solutions based on previous ones, A[1..2], A[2..3], ..., A[n-1..n]
3. A[1..3], A[2..4], ..., A[n-2..n]
4. ... ...
5. A[1..n]

Top-down memorized DP algorithm

In most cases, a recursive algorithm might be more straightforward and thus easier to come up with. But we need to "memorize" intermediate results so as to reduce the running time from exponential to polynomial. To do that,

1. maintain a table to store previous results as we do in bottom-up
2. in each recursion,
    1. check the table for previous results and return directly if it has been computed before
    2. if no existing result, do the computation and store it in the table

Bottom-up vs. Top-down memorized

Generally, if ALL subproblems must be solved at least once, a bottom-up DP algorithm usually performs slightly better (by a constant factor) than a top-down memorized algorithm since a bottom-up one avoids overhead for recursion and is easier to maintain the table.

On the other hand, if NOT all subproblems in the subproblem space are needed to be solved, the top-down one works better since it only solves the subproblems that are required.

Comparisons with Other Strategies

DP vs. Divide-and-Conquer

Both of the two can be implemented by recursions. The major difference is whether there are any overlapping subproblems. If subproblem recurrences exist, then DP will be a better choice, which sacrifice polynomial space while reduce running time dramatically.

For instance,
We use divide-and-conquer to solve MergeSort problem by recursively merge sub-arraies together, as shown below.

```
2  1  4  3  8  7  5  6
\  /  \  /  \  /  \  /
 12    34    78    56
  \    /      \    /
   1234        5678
      \        /
       12345678
```

While we use DP to solve Count the Number of Parenthesizations by parenthesizing sub-arraies and then build up results based on previous ones. You can see overlapping as shown below.

```
(A1 A2 A3 A4)

/           |            \

/             |              \
         ((A1 A2 A3) A4) or ((A1 A2) (A3 A4)) or (A1 (A2 A3 A4))

/      \                                   /      \
((A1 A2) A3) or (A1 (A2 A3))                ((A2 A3) A4) or (A2 (A3 A4))
```

DP vs. Greedy Algorithm

DP first computes all possible subproblems and then "make a decision" to find the optimal one.
Whereas, Greedy algorithm "make a decision" first, the looks-to-be-best decision at the time, and then compute the resulting subproblem.

Posted 31st March 2013 by Sophie

Labels: DP, GreedyAlgorithm, Recursion, Summary

0 Add a comment

28th March 2013

Word Segment

Word Segment [http://discuss.leetcode.com/questions/765/word-break] (Word Break [http://www.geeksforgeeks.org/dynamic-programming-set-32-word-break-problem/])

Given an input string and a dictionary, determine if the string can be segmented into a space-separated sequence of dictionary words.

For example, given a standard English dictionary and the input string "codereaddiscuss", return true because the string can be segmented into "code read discuss".

Solution

We can solve the problem recursively:

```
for (int i=0; i<s.length(); ++i) {
  if (dict.contains(s.substring(0, i+1) && wordSegment(s.substring(i+1), dict))
    return true;
}
```

But there will be a lot of redundant computations.
For example, given "abcde", when we compute "a" and "bcde", we then compute suffix of "bcde" such as "cde", "de", "e"; but then we compute "ab" and "cde", we recompute "cde" and its suffix "de" and "e".

If we solve the problem in a bottom-up fashion, we first compute single-letter substrings and reuse the results to concatenate them into longer substrings, until ending up with the original string. This gives us a DP solution for this problem.

Let seg[i][j] represent a substring of s such that it starts at s[i] and ends at s[j].
seg[i][j] can be segmented iff

- seg[i][j] is a word in the given dictionary
- or seg[i][k] and seg[k+1][j] can both be segmented, where i<k<j

So, we can use DP to solve this problem.

We starts from checking substrings of length 1, i.e. s[i..i] for all i;
then length 2, i.e. s[i..i+1] (and of course we can use previous results for s[i..i]+s[i+1..i+1]),
then length 3, i.e. s[i..i+2] (test s[i..i]+s[i+1..i+2], s[i..i+1]+s[i+2..i+2] using previous results),
and so on.
The final result is the original string s, i.e. seg[0][s.length()-1].

```
public static boolean isSegmented(String s, HashSet<String> dict) {
  int n = s.length();
  if (n < 1) return false;

  // T[i][j] == true iff s[i..j] is segmentable
  boolean[][] seg = new boolean[n][n];
  for (int l=0; l<n; ++l) { // segment length, seg[i,i] has length of 0.
    for (int i=0; i<n-l; ++i) { // start letter
      int j = i + l;
      if (dict.contains(s.substring(i, j+1))) {
        seg[i][j] = true;
        continue;
      }
      for (int k=i; k<j; ++k) { // intermediate letter
        if (seg[i][k] && seg[k+1][j]) {
          seg[i][j] = true;
          break;
        }
      }
    }
  }

  return seg[0][n-1];
}
```

This algorithm runs in time O(n^3) and uses O(n^2) spaces.

Unfortunately, there still exist unnecessary computations (not redundant but unnecessary). For example, given "seashell" and ["sea", "shell"], we never need to know whether "a" is a valid word since "se" is not a valid prefix!

Let's revisit the previous recursive solution where we verify each possible prefix and if it is valid, we then verify whether the rest of string can be segmented. So, we have:

s[0..i) is segmentable iff

- s[0..i) is a valid word in the dictionary
- or s[0..k) is a valid prefix and s[k..i) is a valid suffix, where k is some value between 0 and i

We don't need to precompute all substrings. Instead, only when we find a valid prefix, we update the rest of substrings if there are valid suffixes. This is similar to Dijkstra's algorithm [http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm] where we select a node and update distances of its neighbors correspondingly.

```
public static boolean isSegmented(String s, HashSet<String> dict) {
  int n = s.length();
  if (n < 1) return false;

  // T[i] == true iff s[0..i) is segmentable
  boolean[] seg = new boolean[n];
  for (int i=0; i<n; ++i) {
    seg[i] = (seg[i] || dict.contains(s.substring(0, i+1)));
    // if seg[i] is false, check the current prefix
    if (!seg[i])  continue;

    // now seg[i] is true, update remaining elements
    for (int j=i+1; j<n; ++j) {
      seg[j] = (seg[j] || dict.contains(s.substring(i+1, j+1)));
    }

    // fast return
    if (seg[n-1]) return true;
  }

  return false;
}
```

This algorithm runs in time O(n^2) and uses O(n) spaces.

Posted 28th March 2013 by Sophie

Labels: Array, DP, Graph, Java, Recursion, String

0 Add a comment

28th March 2013

Longest Common Prefix

Longest Common Prefix [http://leetcode.com/onlinejudge#question_14]

Write a function to find the longest common prefix string amongst an array of strings.

Solution

There are two ways to get the longest common prefix:

- (DFS-like) Find a common prefix of the first two strings, compare that common prefix with the third one to find a common prefix of the first three, and so on, until find the common prefix of all strings.
- (BFS-like) Start from the first character. Compare whether the first character of all strings are the same, if so, move to next; otherwise, return the longest common prefix.

In worst cases, both methods need to go through all strings. But on average, the second one would be slightly faster.

```
public String longestCommonPrefix(String[] strs) {
  if (strs.length == 0) return "";
  int index = 0;
  while (index < strs[0].length()) {
    char c = strs[0].charAt(index);
    for (int i=1; i<strs.length; ++i) {
      if (index >= strs[i].length() || strs[i].charAt(index) != c) {
        return strs[0].substring(0, index);
      }
    }
    index++;
  }
  return strs[0];
}
```

Posted 28th March 2013 by Sophie

Labels: Java, String

0 Add a comment

21st March 2013

Length of Last Word

Length of Last Word [http://leetcode.com/onlinejudge#question_58]

Given a string s consists of upper/lower-case alphabets and empty space characters ' ', return the length of last word in the string.
If the last word does not exist, return 0.

Note: A word is defined as a character sequence consists of non-space characters only.

For example, given s = "Hello World ", return 5.

Solution

We first trim the while spaces from the end of the string and then go backwards to count the length of the last word (if exists).

```
public int lengthOfLastWord(String s) {
  int len = 0, last = s.length()-1;
  while (last>=0 && s.charAt(last) == ' ') last--;
  while (last>=0 && s.charAt(last--) != ' ') ++len;
  return len;
}
```

Posted 21st March 2013 by Sophie

Labels: Java, String

1 View comments

20th March 2013

Largest Rectangle in Histogram

Largest Rectangle in Histogram [http://leetcode.com/onlinejudge#question_84]

Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.

[http://1.bp.blogspot.com/-Tm7_0Lqpv-c/UUgAKqSnpil/AAAAAAAAEao/xSWYA3AyPcU/s1600/histogram.png]
Above is a histogram where width of each bar is 1, given height = [2,1,5,6,2,3].

[http://4.bp.blogspot.com/-EKJWTW9077U/UUgAKv5NiOI/AAAAAAAAEa0/WGI9HYIxUcY/s1600/histogram_area.png]
The largest rectangle is shown in the shaded area, which has area = 10 unit.

For example, given height = [2,1,5,6,2,3], return 10.

Solution - Recursion (bad solution...)

This problem seems like container-with-most-water [http://n00tc0d3r.blogspot.com/2013/02/container-with-most-water.html] problem we discussed before. But the difference is that we are not using two "wood boards" to make a bucket, but using several boards with different heights.

Given a bucket made of a set of boards with different heights, the volume of the bucket depends on the lowest board. That is, suppose a bucket is made of board [0 .. n), and suppose the lowest one has to be included and it has height of k, the maximal volume is k*n.

Then we can solve the problem recursively:

- Find the minimum height in [l, r), and compute an area as min*(l-r);
- Use the minimum to split the array into two parts and recursively find the max area in each of them;
- The final result is the maximum the above three results.

```
private int largestRect(int[] height, int l, int r) {
  if (l>=r) return 0;
  int min = height[l], minId = l;
  // find the min and its index
  for (int i=l+1; i<r; ++i) {
    if (height[i] < min) {
      min = height[i];
      minId = i;
    }
  }
  return Math.max(min*(r-l),
      Math.max(largestRect(height, l, minId), largestRect(height, minId+1, r)));
}

public int largestRectangleArea(int[] height) {
  return largestRect(height, 0, height.length);
}
```

This algorithm runs in time O(n^2). But it requires O(n^2) spaces since it is using recursion. So, if you run it on a memory-limited machine with a large array, it would return "Stack Overflow" error!

Solution - Stack

In most cases, we can use Stack to replace recursions. So is here.

For each height, we need to keep track the left boundary of the corresponding rectangle (that contains this height). We know that the left boundary is the left-most height that is greater or equal to this height.
We need to calculate the area when reaching the right boundary, where the right boundary is the right-most height that is greater or equal to this height.

For example, given height = [2,1,5,6,2,3]. We have

cur	left	index	area
0	~~0~~	~~0(2)~~	0
1	0	1(1)	2*1
2	~~2~~	~~2(5)~~	2*1
3	~~3~~	~~3(6)~~	2*1
4	2	4(2)	~~6*1~~ 5*2
5	5	5(3)	
			5*2 ~~3*1~~ ~~2*4~~ ~~1*6~~

```
public int largestRectangleArea(int[] height) {
  int area = 0;
  // stack to store the indices of left boundary
  // left boundary is the last height that is not lower than the current one
  Stack<Integer> left = new Stack<Integer>();
  Stack<Integer> index = new Stack<Integer>();
  int cur = 0;
  while (cur < height.length) {
    if (cur == 0 || height[cur] > height[index.peek()]) {
      left.push(cur);
      index.push(cur);
    } else if (height[cur] < height[index.peek()]) {
      int last;
      do {
        last = left.pop();
        area = Math.max(area, height[index.pop()] * (cur - last));
      } while (!left.isEmpty() && height[cur] < height[index.peek()]);
      left.push(last);
      index.push(cur);
    }
    cur++;
  }
  // pop out values in index and left and calculate their areas
  while (!index.isEmpty() && !left.isEmpty()) {
    area = Math.max(area, height[index.pop()] * (height.length - left.pop()));
  }

  return area;
}
```

Although it has embedded loops, it runs in time O(n) since it only access each height O(1) times. It requires O(n) spaces for the two stacks.

Improved Solution - Stack

Notice that we only push a new item into stack when the new item is no less than the top one in the stack. That said, when we pop up an item, the new top in the stack is the first one (go backwards from the popped one) that is smaller than the popped one, which implies that the left boundary of the current rectangle is top+1, inclusively.
Also, when we pop it up, it means that we hit a new height that is smaller than it, which implies that the right boundary is cur-1, inclusively.

With these information in hand, we only need to maintain one stack!

```
public int largestRectangleArea(int[] height) {
  Stack<Integer> left = new Stack<Integer>();
  int cur = 0, area = 0;
  while (cur < height.length) {
    if (left.isEmpty() || height[cur] >= height[left.peek()]) {
      // push to stack if we hit a greater or equal height
      left.push(cur++);
    } else {
      int top = left.pop();
      // the height at left.peek() must be smaller than the current one
      // so, the width of the rectangle is [left.peek()+1, cur)
      area = Math.max(area, height[top]*(left.isEmpty() ? cur : (cur-left.peek()-1)));
    }
  }
  while (!left.isEmpty()) {
      int top = left.pop();
      area = Math.max(area, height[top]*(left.isEmpty() ? cur : (cur-left.peek()-1)));
  }
  return area;
}
```

Posted 20th March 2013 by Sophie

Labels: Array, Java, Recursion, Stack

3 View comments

15th March 2013

Jump Game

Jump Game [http://leetcode.com/onlinejudge#question_55]

Given an array of non-negative integers, you are initially positioned at the first index of the array.
Each element in the array represents your maximum jump length at that position.
Determine if you are able to reach the last index.

For example:
A = [2,3,1,1,4], return true.
A = [3,2,1,0,4], return false.

Solution

It is much easier to tell whether the second to last element can reach the last one -- if the value is no less than 1. So, let's go through the array backwards.

Notice that the value of element i represents the **maximum** jump length.
Suppose we are at A[i].

- If the value is no less than the distance to the last element, then yes;
- Or if any of the elements within the jump range can reach the last one, then yes;
- Otherwise, no.

Don't forget the special case where A is a single element array. Then no matter what value it has, we already at the last element. :-]

```
public boolean canJump(int[] A) {
  if (A.length <= 1) return true;
  boolean[] canReach = new boolean[A.length];
  for (int i=A.length-2, dist=1; i>=0; --i, ++dist) {
    if (A[i] >= dist) {
      canReach[i] = true;
    } else {
      int j=1;
      while (j<=A[i] && !canReach[i+j]) ++j;
      if (j<=A[i]) canReach[i] = true;
    }
  }
  return canReach[0];
}
```

The downside of this algorithm is that it requires O(n^2) time since in worst case we need to check each element within the range. Also, it requires O(n) space.

Do we really need to check each of them? Actually, all we want to know whether the next can-reach-end element is within my range. So, during the iteration, we can store the position of the next can-reach-end element and then for each successor element, we only need to check whether that node is reachable.

```
public boolean canJump(int[] A) {
  int next = A.length - 1;
  for (int i=A.length-2; i>=0; --i) {
    if (A[i] >= (next - i)) {
      next = i;
    }
  }
  return (next == 0);
}
```

This algorithm runs in time O(n) and uses O(1) space.

Jump Game II [http://leetcode.com/onlinejudge#question_45]

Given an array of non-negative integers, you are initially positioned at the first index of the array.
Each element in the array represents your maximum jump length at that position.
Your goal is to reach the last index in the minimum number of jumps.

For example:
Given array A = [2,3,1,1,4], the minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

Solution

Again, we start from an O(n^2) algorithm where we go backwards, compute the minimum steps for the current element by checking each element within its range to find the next one with smallest number of steps.

```
public int jump(int[] A) {
  if (A.length <= 1) return 0;
  int[] steps = new int[A.length-1];
  for (int i=A.length-2; i>=0; --i) {
    if (A[i] >= (A.length - 1 - i)) {
      steps[i] = 1;
    } else {
      int min = A.length;
      for (int j=1; j<=A[i]; ++j) {
        min = Math.min(min, steps[i+j]);
      }
      steps[i] = min + 1;
    }
  }
  return steps[0];
}
```

This requires O(n^2) time and O(n) space.

We can use Greedy algorithm [http://en.wikipedia.org/wiki/Greedy_algorithm] here. The basic idea is each time when we jump, we jump to the farthest slot.
Start from slot i,

- next is the next position where we will jump, so it starts from 0.
- max is the farthest position we can reach so far, so every time when we jump, we set next to current max.

Note: It is possible that the endpoint is unreachable (e.g. 3 2 1 0 0 0 0 0 0 1). In that case, at some jump point, we will find out that we are jumping to the same point. Then, game over.

```
public int jump(int[] A) {
  int steps = 0;
  for (int i=0, max=0, next=0; i<A.length-1 && next<A.length-1; ++i) {
    max = Math.max(max, i+A[i]);
    if (i == next) {  // ready to jump
      if (max == next) return -1; // unreachable
      next = max;
      ++steps;
    }
  }
  return steps;
}
```

This algorithm runs in time O(n) and uses O(1) space.

Posted 15th March 2013 by Sophie

Labels: Array, DP, GreedyAlgorithm, Java

0 Add a comment

14th March 2013

Interleaving String (Gene Matching)

Interleaving String [http://leetcode.com/onlinejudge#question_97]

Given s1, s2, s3, find whether s3 is formed by the interleaving of s1 and s2.

For example, given: s1 = "aabcc", s2 = "dbbca",

When s3 = "aadbbcbcac", return true.
When s3 = "aadbbbaccc", return false.

Solution - Recursion

It is not too difficult to come up with a recursive solution:

- Test a character in s1, s1[i]==s3[k]?, if succeeded, move s1 and s3 to next character, and then test the rest of s3 with the rest of s1 and s2;
- Or test a character in s2, s2[j]==s3[k]?, if succeeded, move s2 and s3 to next character, and then test the rest of s3 with the rest of s1 and s2.

```
private boolean helper(char[] s1, int n1, char[] s2, int n2, char[] s3, int n3) {
  if (n1==s1.length && n2==s2.length && n3==s3.length) return true;
  if (n1==s1.length) return (s2[n2]==s3[n3] && helper(s1, n1, s2, n2+1, s3, n3+1));
  if (n2==s2.length) return (s1[n1]==s3[n3] && helper(s1, n1+1, s2, n2, s3, n3+1));
  return (s1[n1]==s3[n3] && helper(s1, n1+1, s2, n2, s3, n3+1))
      || (s2[n2]==s3[n3] && helper(s1, n1, s2, n2+1, s3, n3+1));
}

public boolean isInterleave(String s1, String s2, String s3) {
  if (s1.length() + s2.length() != s3.length()) return false;
  return helper(s1.toCharArray(), 0, s2.toCharArray(), 0, s3.toCharArray(), 0);
}
```

This algorithm is simple and straightforward but expensive...

Solution - DP

Construct a (n1+1)*(n2+1) table, A[0 .. n1][0 .. n2], where n1 and n2 is the length of s1 and s2, respectively.

> A[i][j] = true, i.e. s3[0 .. i+j-1] is an interleaving of s1[0 .. i-1] and s2[0 .. j-1].
> Thus, A[n1][n2] represents whether s3 is a interleaving of s1 and s2.

To build up such a table,

1. Set A[0][0] = true. Period.
2. Initialize the first row. The first row means whether s1[0 .. (n1-1)] matches s3[0 .. (n1-1)], correspondingly.
3. Similarly, initialize the first column which means whether s2[0 .. (n2-1)] matches s3[0 .. (n2-1)], correspondingly.
4. Now we fill up the table. For A[i][j], s3[0 .. (i+j-1)] match? s1[0 .. i-1] weaving s2[0 .. j-1]
    - Note that A[i-1][j] comes from s1[0 .. i-2] weaving s2[0 .. j-1]. Thus, if it is true, we need to compare s1[i-1] with s3[i+j-1].
    - Similarly, A[i][j-1] comes from s1[0 .. i-1] weaving s2[0 .. j-2]. Thus, if it is true, compare s2[j-1] with s3[i+j-1].
5. Return A[n1][n2]

You may need to draw a table with an example in your mind or on paper or on whiteboard or on a wall... :)

```
public boolean isInterleave(String s1, String s2, String s3) {
  int n1=s1.length(), n2=s2.length(), n3=s3.length();
  if (n1 + n2 != n3) return false;
  boolean[][] match = new boolean[n1+1][n2+1];
  // initialize the first row and first column
  match[0][0] = true;
  for (int i=0; i<n1; ++i) {
    match[i+1][0] = (s1.charAt(i) == s3.charAt(i));
  }
  for (int i=0; i<n2; ++i) {
    match[0][i+1] = (s2.charAt(i) == s3.charAt(i));
  }
  // fill up the table
  for (int i=0; i<n1; ++i) {
    for (int j=0; j<n2; ++j) {
      if ((match[i][j+1] && s3.charAt(i+j+1)==s1.charAt(i))
          || (match[i+1][j] && s3.charAt(i+j+1)==s2.charAt(j))) {
        match[i+1][j+1] = true;
      }
    }
  }
  return match[n1][n2];
}
```

In this algorithm, since we build up a 2D table, it takes time O(n1*n2) and space O(n1*n2).

Notice that for each A[i][j], we only need A[i-1][j] and A[i][j-1]. So, we can improve the above algorithm with an 1D array. By doing that, the space of the new algorithm get down to O(min(n1, n2)).

```
public boolean isInterleave(String s1, String s2, String s3) {
  int n1=s1.length(), n2=s2.length(), n3=s3.length();
  if (n1 + n2 != n3) return false;
  // switch to save space if needed
  if (n1<n2) {
    String tmp = s1; s1 = s2; s2 = tmp;
    n1 = s1.length(); n2 = s2.length();
  }
  boolean[] match = new boolean[n2+1];
  // initialize the first row and first column
  match[0] = true;
  for (int j=0; j<n2; ++j) {
    match[j+1] = (match[j] && s2.charAt(j)==s3.charAt(j));
    if (!match[j+1]) break;
  }
  // fill up the table
  for (int i=1; i<=n1; ++i) {
    match[0] = (match[0] && s1.charAt(i-1)==s3.charAt(i-1));
    for (int j=1; j<=n2; ++j) {
      match[j] = (match[j] && s3.charAt(i+j-1)==s1.charAt(i-1))
            || (match[j-1] && s3.charAt(i+j-1)==s2.charAt(j-1));
    }
  }
  return match[n2];
}
```

Posted 14th March 2013 by Sophie

Labels: DP, Java, Recursion, String, StringSearching

0 Add a comment

13th March 2013

Integer and Roman Numeral

Integer to Roman [http://leetcode.com/onlinejudge#question_12]

Given an integer, convert it to a roman numeral.
Input is guaranteed to be within the range from 1 to 3999.

Solution

Here is the wiki [http://en.wikipedia.org/wiki/Roman_numerals] to introduce roman numeral symbols and their corresponding values.
Basic symbols are M, D, C, L, X, V, and I, representing 1000, 500, 100, 50, 10, 5 and 1.
But there are several special rules for 900, 400, 90, 40, 9 and 4 (see wiki for details).

```
1:  public String intToRoman(int num) {
2:    int[] nums = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};
3:    String[] symbols = {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"};
4:    StringBuilder res = new StringBuilder();
5:    int i=0;
6:    while (num>0) {
7:      int times = num / nums[i];
8:      num -= nums[i]*times;
9:      for (; times>0; times--) {
10:        res.append(symbols[i]);
11:      }
12:      ++i;
13:    }
14:    return res.toString();
15:  }
```

Notice that line 8 can be replaced by `num %= nums[i]` but '%' is an expensive operation. So I use minus and multiply to replace it (not sure how much better between multiply and mod though).

Roman to Integer [http://leetcode.com/onlinejudge#question_13]

Given a roman numeral, convert it to an integer.
Input is guaranteed to be within the range from 1 to 3999.

Solution

This is the reverse problem of the above one. And we need to consider the special cases when mapping roman numerals back to integer. So, in those cases, we need to check the successor char.

```
/* int:  1000,  900, 500, 400,  100,  90,  50,   40,   10,   9,   5,    4,   1
* roman: "M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"
*/
public int romanToInt(String s) {
  HashMap<Character, Integer> map = new HashMap<Character, Integer>();
  map.put('M', 1000);
  map.put('D', 500);
  map.put('C', 100);
  map.put('L', 50);
  map.put('X', 10);
  map.put('V', 5);
  map.put('I', 1);

  int res = 0, i = 0;
  while (i<s.length()) {
    switch(s.charAt(i)) {
    case 'C':
    case 'X':
    case 'I':
      if (i+1 < s.length()
          && map.get(s.charAt(i+1)) > map.get(s.charAt(i))) {
        res += (map.get(s.charAt(i+1)) - map.get(s.charAt(i)));
        i += 2;
      } else {
        res += map.get(s.charAt(i));
        ++i;
      }
      break;
    default:
      res += map.get(s.charAt(i));
      ++i;
    }
  }
  return res;
}
```

Or without using `switch`

```
/* int: 1000, 900, 500, 400, 100, 90, 50,  40,  10,  9,  5,  4,  1
 * roman: "M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"
 */
public int romanToInt(String s) {
  HashMap<Character, Integer> map = new HashMap<Character, Integer>();
  map.put('M', 1000);
  map.put('D', 500);
  map.put('C', 100);
  map.put('L', 50);
  map.put('X', 10);
  map.put('V', 5);
  map.put('I', 1);

  int res = 0;
  for (int i=0; i<s.length(); ++i) {
    char c = s.charAt(i);
    if ((c == 'C' || c == 'X' || c == 'I')
        && i+1 < s.length() && map.get(s.charAt(i+1)) > map.get(c)) {
      res += map.get(s.charAt(i+1)) - map.get(c);
      ++i;
    } else {
      res += map.get(c);
    }
  }
  return res;
}
```

Posted 13th March 2013 by Sophie

Labels: Array, Hash, Java, Maths

1 View comments

12th March 2013

Insert Interval

Insert Interval [http://leetcode.com/onlinejudge#question_57]

Given a set of non-overlapping intervals, insert a new interval into the intervals (merge if necessary).
You may assume that the intervals were initially sorted according to their start times.

Example 1:
Given intervals [1,3],[6,9], insert and merge [2,5] in as [1,5],[6,9].

Example 2:
Given [1,2],[3,5],[6,7],[8,10],[12,16], insert and merge [4,9] in as [1,2],[3,10],[12,16].
This is because the new interval [4,9] overlaps with [3,5],[6,7],[8,10].

Solution

It is not difficult to figure out an algorithm to solve the problem. But to give a bug-free code and sufficiently test all edge cases make it not a easy game.

An intuitive solution is

- Iterate through the interval array, find out the possible place to insert the new interval (i.e. the first interval that all previous ones are ended before the new interval's start).
  Copy them into the new result array if present.
- To insert the new interval, there are several cases:
  - No more intervals
    : append it as the last one in the result
    e.g. [[1,3]], [5, 7] -> [[1,3], [5,7]] or [[]], [5,7] -> [[5,7]]
  - Next interval are far away, i.e. starts after new one's end
    : append the new one to the result and copy over the rest of intervals (no merge involved)
    e.g. [[1,2], [6,8]], [4,5] -> [[1,2], [4,5], [6,8]]
  - Merge is needed
    : merge and extend the new interval as needed, append it to the result, and copy over the rest
    e.g. [[1,2], [4,6], [8,10]], [2,5] -> [[1,6], [8,10]]

```
public ArrayList<Interval> insert(ArrayList<Interval> intervals, Interval newInterval) {
  ArrayList<Interval> results = new ArrayList<Interval>();
  // find the place to insert the new interval
  int pos = 0;
  while (pos < intervals.size() && intervals.get(pos).end < newInterval.start) {
    results.add(intervals.get(pos++));
  }
  // insert (merge as needed)
  while (pos < intervals.size() && intervals.get(pos).start <= newInterval.end) {
    newInterval.start = Math.min(newInterval.start, intervals.get(pos).start);
    newInterval.end = Math.max(newInterval.end, intervals.get(pos).end);
    ++pos;
  }
  results.add(newInterval);
  // add the rest
  while (pos < intervals.size()) {
    results.add(intervals.get(pos++));
  }
  return results;
}
```

This algorithm runs in time O(n) since we touch each interval O(1) times. But we use extra O(n) space to save the new result array.

There are some spaces to improve:

- To find out the insert place, instead of iterate linearly, we can use binary search.
  Be careful about edge cases! It is not easy to implement binary search correctly (even the first implementation in the original paper was not totally correct :).
- We can insert and update the given array, no need to copy to a new one.
  Notice that removing an element from an array requires O(n) time since it needs to update all of the rest elements. So, we should remove a range of elements once, rather than do it one by one.

```
// binary search to find the first place that end>=target
private int findPos(ArrayList<Interval> src, int target) {
  if (src.isEmpty()) return 0;
  int low = 0, high = src.size() - 1;
  while (low <= high) {
    int mid = (low + high) / 2;
    if (src.get(mid).end < target) {
      low = mid + 1;
    } else if (src.get(mid).end == target) {
      return mid;
    } else {
      high = mid - 1;
    }
  }
  return low;
}

public ArrayList<Interval> insert(ArrayList<Interval> intervals, Interval newInterval) {
  int pos = findPos(intervals, newInterval.start);
  if (pos == intervals.size()) {
    // append (no merge)
    intervals.add(newInterval);
  } else if (intervals.get(pos).start > newInterval.end) {
    // insert (no merge)
    intervals.add(pos, newInterval);
  } else {
    // update current interval (merge as needed)
    int start = pos;
    while (pos < intervals.size() && intervals.get(pos).start <= newInterval.end) {
      newInterval.start = Math.min(newInterval.start, intervals.get(pos).start);
      newInterval.end = Math.max(newInterval.end, intervals.get(pos).end);
      ++pos;
    }
    intervals.set(start, newInterval);
    intervals.subList(Math.min(start+1, intervals.size()),
             Math.min(pos, intervals.size()))
             .clear();
  }
  return intervals;
}
```

The worst case running time of this algorithm could be O(n^2) if we remove intervals O(n) times. But the space is O(1) now.

Posted 12th March 2013 by Sophie

Labels: Array, BinarySearch, Java

0 Add a comment

11th March 2013

Implement strStr()

Implement strStr() [http://leetcode.com/onlinejudge#question_28]

The strstr function locates a specified substring within a source string. Strstr returns a pointer to the first occurrence of the substring in the source. If the substring is not found, strstr returns a null pointer.

```
public String strStr(String haystack, String needle)
```

Solution

There are several well-known algorithms for string matching, such as Rabin-Karp algorithm [http://en.wikipedia.org/wiki/Rabin-Karp_algorithm] , KMP algorithm [http://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm] , Boyer-Moore algorithm [http://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string_search_algorithm] , etc., and of course, brute force string matching. Most of these algorithm have been taught in advanced algorithm class. Here is a very nice tutorial [http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=stringSearching] from TopCoder about Rabin-Karp and KMP.
Note: It can be solved in linear time, O(n+m), by using SuÕx tree [http://en.wikipedia.org/wiki/SuÕx_tree] . To implement a suÕx tree is complicated, you may not have enough time to finish it during an interview. Thus, we are not going to discuss it here.

Let's start from the brute force which is intuitive but a little expensive. :)

Basically, what it does is to iterate through the source string, compare whether there is a match, and if not, move on.

```
public String strStr(String haystack, String needle) {
  int n = haystack.length(), m = needle.length();
  if (m==0) return haystack;
  if (n<m) return null;
  for (int i=0; i<n-m+1; ++i) {
    int start = 0;
    for (int offset=i; start<m && haystack.charAt(offset)==needle.charAt(start); ++offset, ++start);
    if (start==m) return new String(haystack.toCharArray(), i, n-i);
  }
  return null;
}
```

This algorithm runs in time O((n-m+1)m)=O(nm) since it repeatedly compare the entire substring and each comparison could take O(m) time in worse case.

KMP-Algorithm

We can use a pre-process function that store some useful information about the pattern into an overlapping table. What is an overlapping table?
An overlapping table is to find the last position in [0..i-1] that have an overlapping prefix.
For example, suppose pattern is "ABABAC".

T[0] = 0 // ""
T[1] = 0 // ""
T[2] = 0 // ""
T[3] = 1 // "A" <- If fails at pattern[3], we already saw "A". So next comparison start from pattern[1].
T[4] = 2 // "AB"
T[5] = 3 // "ABA"
T[6] = 0 // ""

With such a table in hand, upon each failure of a comparison between source and pattern strings, we can skip overlapping part of the pattern.

```
/* A B A B A C
 -1 0 0 1 2 3 0 */
private void buildNextTable(String pattern, int[] next) {
  next[0] = -1;
  for (int i=2; i<=pattern.length(); ++i) {
    int j = next[i-1];
    while (j>-1 && pattern.charAt(i-1)!=pattern.charAt(j)) j = next[j];
    if (j>-1) next[i] = j+1;
  }
}

public String strStr(String haystack, String needle) {
  int n = haystack.length(), m = needle.length();
  if (m==0) return haystack;
  if (n<m) return null;
  int[] next = new int[m+1];
  buildNextTable(needle, next);
  int offset = 0, start = 0;
  while (offset < n) {
    if (haystack.charAt(offset) == needle.charAt(start)) {
      ++offset;
      if (++start == m) {
        return new StringBuilder()
              .append(needle)
              .append(haystack.toCharArray(), offset, n-offset)
              .toString();
      }
    } else if (start > 0) {
      start = next[start];
    } else {
      ++offset;
    }
  }
  return null;
}
```

This pre-process function runs in time O(m) and the algorithm runs in time O(n).

Although we still have embedded loops but this time, if there is a match, we will not test it repeatedly; plus, there is at most one mismatch for each character in source string. So the total running time of strStr is at most 2n = O(n).

Posted 11th March 2013 by Sophie

Labels: Java, String, StringSearching

0 Add a comment

8th March 2013

Gray Code and Variants

Gray Code [http://leetcode.com/onlinejudge#question_89]

The gray code is a binary numeral system where two successive values differ in only one bit.

Given a non-negative integer n representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0.

For example, given n = 2, return [0,1,3,2]. Its gray code sequence is:

```
00 - 0
01 - 1
11 - 3
10 - 2
```

Note: For a given n, a gray code sequence is not uniquely defined. For example, [0,2,3,1] is also a valid gray code sequence according to the above definition.

Solution

(To get some basic ideas of bit manipulation, check out Bit Manipulation tutorial I [http://sophie-notes.blogspot.com/2013/02/bit-manipulation-tutorial-i.html] and II [http://sophie-notes.blogspot.com/2013/02/bit-manipulation-tutorial-ii.html] .)

To calculate gray code sequence of 1 bit, only need to flip the last bit from 0 to 1.

```
0
1
```

Move to 2 bits, as shown above, first generate a gray code sequence of 1 bit, then go backwards and flip the second bit from 0 to 1.

```
00
01 ↑
---
11 ↓
10
```

Move to 3 bits, first generate a gray code sequence of 2 bits, then go backwards and flip the third bit from 0 to 1.

```
000
001
011
010 ↑
-----
110 ↓
111
101
100
```

The number of resulted sequence will be 8=2^3 since we iterate through the 2-bit results twice. The second half must satisfy the requirement of gray code since it is mirror of the first half except that the highest bit is 1 now.

Generalize to n bits, we generate a sequence of n-1 bits, then fold back and flip the n-th bit from 0 to 1.

```
public ArrayList<Integer> grayCode(int n) {
  ArrayList<Integer> results = new ArrayList<Integer>(1<<n);
  results.add(0);
  for (int i=0; i < n; ++i) {
    int flipper = 1<<i;
    for (int j=results.size()-1; j>=0; --j) {
      results.add(results.get(j) | flipper);
    }
  }
  return results;
}
```

Obviously, this algorithm runs in time O(2^n).

k-th Gray Code [http://www.matrix67.com/blog/archives/266]

Given a non-negative integer n, representing the total number of bits in the gray code, and a non-negative integer k, print the k-th code in the gray code sequence.

A gray code sequence must begin with 0.

Note: As mentioned in prior problem, for a given n, a gray code sequence is not unique. Here we assume the sequence is generated in the way we discussed above.

Solution

Why we interest in k-th code?

Notice that Gray Code can be used to solve Tower of Hanoi [http://en.wikipedia.org/wiki/Tower_of_Hanoi#Gray_code_solution] problem in that the bit sequence we flips in each step is same as the ordinal of the disk to be moved in each step.
E.g. suppose n=3, to generate a gray code sequence, we flips bits of previous number in the following order:

> 1 - 2 - 1 - 3 - 1 - 2 - 1

which is exactly the same sequence of disk moves, assuming we mark each disk from 1 to n.
So, if we can figure out the k-th code in the gray code sequence, we know the k-th move of the Hanoi solution.

Back to the problem.

Since we only want to know one code, it's not necessary/worthy to generate the entire sequence which takes exponential time. Let's compare the ordinal and the code.

Again, suppose n = 3.

code	ordial	
000	000	<- same
001	001	<- same
011	010	<- flip bit 0
010	011	<- flip bit 0
110	100	<- flip bit 1
111	101	<- flip bit 1
101	110	<- flip bits 11
100	111	<- flip bits 11

Therefore,

```
public int kthGrayCode(int k) {
  return k ^ (k>>1);
}
```

Gray Code Matrix [http://acm.sgu.ru/problem.php?contest=0&problem=249]

It is necessary to arrange numbers from 0 to 2^(N+M)-1 in the matrix with 2^N rows and 2^M columns. Moreover, numbers occupying two adjacent cells must differ only in single bit in binary notation. Cells are adjacent if they have common side. Matrix is cyclic, i.e. for each row the leftmost and rightmost matrix cells are considered to be adjacent (the topmost and the bottommost matrix cells are also adjacent).

Given two non-negative integers, n and m, n+m<=20, print a matrix in a form of 2^N lines of 2^M integers each.

For example, given n=m=1,

```
0 2
1 3
```

Solution

This is to produce a two dimension gray code matrix.

We can patch n-bit gray code as a prefix and append with m-bit gray code.
By doing this, adjacent numbers in the same row have one bit difference since the higher n bits are the same while the lower m bits are m-bit gray code;
similarly, adjacent numbers in the same column have one bit difference since the lower m bits are the same while the higher n bits are n-bit gray code.

```
public static int[][] grayMatrix(int n, int m) {
  int nn = 1<<n, mm = 1<<m;
  int[][] matrix = new int[nn][mm];
  for (int i=0; i<nn; ++i) {
    int high = i ^ (i>>1);
    for (int j=0; j<mm; ++j) {
      matrix[i][j] = (high<<m) + (j ^ (j>>1));
    }
  }
  return matrix;
}
```

Posted 8th March 2013 by Sophie

Labels: BitManipulation, Java, Matrix

0 Add a comment

7th March 2013 Probability Of Alive On Island

Probability Of Alive On Island [http://www.careercup.com/question?id=15556758]

There is an island which is represented by square matrix NxN, represented as (0,0) to (N-1,N-1).
A person on the island is standing at given coordinates (x,y). He can move in any direction one step, right, left, up, down on the island. If he steps outside the island, he dies.

Now He is allowed to move n steps on the island (along the matrix). What is the probability that he is alive after he walks n steps on the island?

Write an efficient full code and tests for function `double probabilityOfAlive(int x,int y, int n)`.

Solution - Recursion

This problem is similar to the Edit Distance [http://n00tc0d3r.blogspot.com/2013/03/edit-distance.html] problem we discussed before.

Let's start from a island of size 1x1. Since one step towards any direction will get out of the island, the probability to alive is 0.

Given a island of 3x3 and let's stand at (1,1) which is the center of the island. There are four possibilities and assume the probability of each is the same, 0.25. Walking one step towards any direction will end at a spot on the island. So the probability of alive is (1+1+1+1)/4 = 1.

Now let's stand at (0,1). Walking up one step will get out of the island! So the probability of alive is (0+1+1+1)/4 = 0.75.

Generalizing the formula, we have:

Prob(x, y, step) = 0.25*Prob(x-1, y, step-1) + 0.25*Prob(x, y-1, step-1)
+ 0.25*Prob(x+1, y, step-1) + 0.25*Prob(x, y+1, step-1)

Intuitively, we can solve it using recursion.

```
import java.util.*;
import java.lang.*;

class WaysToAlive
{
  public static double probabilityOfAlive(int x,int y, int step, int n) {
    if (x<0 || y<0 || x>=n || y>=n) return 0;
    if (step == 0) return 1;
    return 0.25*probabilityOfAlive(x-1, y, step-1, n)
        + 0.25*probabilityOfAlive(x, y-1, step-1, n)
        + 0.25*probabilityOfAlive(x+1, y, step-1, n)
        + 0.25*probabilityOfAlive(x, y+1, step-1, n);
  }

  public static void main (String[] args) throws java.lang.Exception {
    int n = 1;
    System.out.println("n=" + n + "(0,0,1) => " + probabilityOfAlive(0,0,1,n));
    n = 2;
      System.out.println("n=" + n + "(0,0,1) => " + probabilityOfAlive(0,0,1,n));
    System.out.println("n=" + n + "(0,0,2) => " + probabilityOfAlive(0,0,2,n));
    n = 3;
      System.out.println("n=" + n + "(0,1,1) => " + probabilityOfAlive(0,1,1,n));
    System.out.println("n=" + n + "(0,1,2) => " + probabilityOfAlive(0,1,2,n));
    System.out.println("n=" + n + "(1,1,1) => " + probabilityOfAlive(1,1,1,n));
      System.out.println("n=" + n + "(1,1,2) => " + probabilityOfAlive(1,1,2,n));
  }
}
```

The code looks simple but it takes exponential time since it doesn't reuse any previous results.

Solution - DP

To save us some time, we can use DP, which sacrifice spaces for time.

We build up a table of NxN and store the probability of walking 1 step from each spot in the table.
Then we build up another table and calculate the probability of walking 2 steps based on the results in 1-step table.
Since we only need results from previous step, we switch between two tables.

```
import java.util.*;
import java.lang.*;

class WaysToAlive
{
  public static double probabilityOfAlive(int x,int y, int step, int n) {
    if (x<0 || y<0 || x>=n || y>=n || (n==1 && step==1)) return 0;
    if (step == 0) return 1;
    double[][][] probTable = new double[n][n][2];
    int sp = 0;
    // init for step=1
    for (int i=0; i<n; ++i) {
      for (int j=0; j<n; ++j) {
        probTable[i][j][sp] = 0;
        if (i>0) probTable[i][j][sp] += 0.25;
        if (j>0) probTable[i][j][sp] += 0.25;
        if (i+1<n) probTable[i][j][sp] += 0.25;
        if (j+1<n) probTable[i][j][sp] += 0.25;
      }
    }
    // calculate with DP
    while (sp < (step-1)) {
      int pre = sp++;
      for (int i=0; i<n; ++i) {
        for (int j=0; j<n; ++j) {
          probTable[i][j][sp&1] = 0;
          if (i>0) probTable[i][j][sp&1] += 0.25*probTable[i-1][j][pre&1];
          if (j>0) probTable[i][j][sp&1] += 0.25*probTable[i][j-1][pre&1];
          if (i+1<n) probTable[i][j][sp&1] += 0.25*probTable[i+1][j][pre&1];
          if (j+1<n) probTable[i][j][sp&1] += 0.25*probTable[i][j+1][pre&1];
          if (sp==step-1 && i==x && j==y) return probTable[x][y][sp&1];
        }
      }
    }
    return probTable[x][y][sp&1];
  }
}
```

Recall that in Edit Distance [http://n00tc0d3r.blogspot.com/2013/03/edit-distance.html] problem, we only use a table of two rows to store the values.
Here we can reduce some spaces too.
Since the table is symmetric, we only need 1/4 of table and a lookup method that can map (i, j) to correct spot in the shrinked table.

Leave it for readers to finish. :-]

Posted 7th March 2013 by Sophie

Labels: DP, Java, Recursion

0 Add a comment

6th March 2013

Generate Parentheses

Generate Parentheses [http://leetcode.com/onlinejudge#question_22]

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.
For example, given n = 3, a solution set is:

"((()))", "(()())", "(())()", "()(())", "()()()"

Solution - Recursion

Intuitively, we can use recursion to solve this problem. Note that at any point, the number of right parentheses must be no more than the number of left ones.

Base case:
If l==r==n, return the current string, where l and r is the number of left and right parentheses, respectively. Actually, if l==n, meaning that we have used up all left parentheses, we can simply append all of the rest of right parentheses and then return the string.

Recursion part:
If l==r<n, we can only append a left parenthesis;
Otherwise, we can append either one.

The algorithm is shown below.

```
private void generator(int n, int l, int r, StringBuilder ss, ArrayList<String> resSet) {
  if (l==n) {
    while (r++ < n) ss = ss.append(")");
    resSet.add(ss.toString());
  } else if (l==r) {
    generator(n, l+1, r, ss.append('('), resSet);
  } else {
    int oldlen = ss.length();
    generator(n, l+1, r, ss.append('('), resSet);
    ss = ss.delete(oldlen, ss.length());
    generator(n, l, r+1, ss.append(')'), resSet);
  }
}

public ArrayList<String> generateParenthesis(int n) {
  ArrayList<String> resSet = new ArrayList<String>();
  generator(n, 0, 0, new StringBuilder(), resSet);
  return resSet;
}
```

Notice that we use StringBuilder rather than String to improve the performance (see discussion here [http://sophie-notes.blogspot.com/2013/02/java-stringbuilder-vs-string.html]).
Also, since Java methods pass pointers to objects, the content of object may be modified in the method (see discussion here [http://sophie-notes.blogspot.com/2012/12/java-passing-arguments-into-methods.html]). In the case here, we need to clean up the StringBuilder when we want to append other things.

Posted 6th March 2013 by Sophie

Labels: Java, Recursion, String, StringBuilder

0 Add a comment

4th March 2013

Flatten A Binary Tree to Linked List (In-place)

Flatten A Binary Tree to Linked List (In-place) [http://leetcode.com/onlinejudge#question_114]

Given a binary tree, flatten it to a linked list in-place.

For example,
Given

```
         1
        / \
       2   5
      / \   \
     3   4   6
```

The flattened tree should look like:

```
   1
    \
     2
      \
       3
        \
         4
          \
           5
            \
             6
```

Solution - Recursion

To flatten a binary tree, according to the given example, is to recursively insert the left subtree to the right subtree and append the original right subtree to the end of the left subtree, i.e.

```
     root                          root
     /  \            ->              \
  left  right                        left
                                       \
                                      right
```

Since we need to append the original right-tree to the end of the left subtree, we let the recursion function return the last node after flatten.

```
public TreeNode flatten(TreeNode root) {
  if (root == null) return root;
  TreeNode rtree = root.right;
  if (root.left != null) {
    root.right = root.left;
    root.left = null;
    root = flatten(root.right);
  }
  if (rtree != null) {
    root.right = rtree;
    root = flatten(root.right);
  }
  return root;
}
```

Solution - Non-Recursion, With Stack

We can also solve the problem without recursion.

To do that, we can use a stack to store all right subtrees when we prune them temporarily, and append each of them back after we go through the corresponding left subtree.

Basically, given a non-empty tree,

- if it has left subtree, store the right subtree (if not null) to stack, move the left subtree to right;
- if not, append back a subtree from stack to the current node's right;
- continue to the right node until finish.

```
public void flatten(TreeNode root) {
  TreeNode cur = root;
  Stack<TreeNode> rtrees = new Stack<TreeNode>();
  while (cur != null) {
    while (cur.left != null) {
      if (cur.right != null) rtrees.push(cur.right);
      cur.right = cur.left;
      cur.left = null;
      cur = cur.right;
    }
    if (cur != null && cur.right == null && !rtrees.isEmpty()) {
      cur.right = rtrees.pop();
    }
    cur = cur.right;
  }
}
```

Alternatively, we can append back a right subtree as early as possible (if the current left node has no right subtree).

```
public void flatten(TreeNode root) {
  TreeNode cur = root;
  Stack<TreeNode> rtrees = new Stack<TreeNode>();
  while (cur != null) {
    if (cur.left != null) {
      if (cur.right != null) rtrees.push(cur.right);
      cur.right = cur.left;
      cur.left = null;
    }
    if (cur.right == null && !rtrees.isEmpty()) {
      cur.right = rtrees.pop();
    }
    cur = cur.right;
  }
}
```

This algorithm runs in time O(n) and uses O(n) extra spaces for the stack.

This would make the code looks simpler but it may introduce some unnecessary extra work. For instance,

```
       1             1             1
      / \             \             \
     2   4    ->       2    ->       2
    /                 / \             \
   3                 3   4             3
                                        \
                                         4
```

Notice that 4 has been cut down and append back twice.

Solution - Non-Recursion, No Stack

We can also solve the problem even without a stack:
Each time when we prune a right subtree, we use while-loop to find the right-most leaf of the current left subtree, and append the subtree there.

```
public void flatten(TreeNode root) {
  TreeNode cur = root;
  while (cur != null) {
    if (cur.left != null) {
      if (cur.right != null) { // if we need to prune a right subtree
        TreeNode next = cur.left;
        while (next.right != null) next = next.right;
        next.right = cur.right;
      }
      cur.right = cur.left;
      cur.left = null;
    }
    cur = cur.right;
  }
}
```

We visit each node at most twice (one for flattening and maybe one for looking for rightmost leaf) and then for each node, cut the right tree and append it to its rightmost node. Overall, we access each node constant time. So the total running time is O(n) with O(1) space.

Similarly, since we append back right subtrees as early as possible (by finding the right-most leaf rather than the last pre-order leaf in a left subtree), it may introduce extra work as shown in previous section.

Posted 4th March 2013 by Sophie

Labels: BinaryTree, Java, LinkedList, Recursion, Stack, Tree

4 View comments

4th March 2013

Find First Missing Positive

Find First Missing Positive [http://leetcode.com/onlinejudge#question_41]

Given an unsorted integer array, find the first missing positive integer.

For example, given [1,2,0] return 3, and [3,4,-1,1] return 2.

<u>Your algorithm should run in O(n) time and uses constant space.</u>

Solution

A few quick thoughts:

- Sort all numbers and iterate through to find the first missing integer? No, most sorting algorithms take time at least O(nlogn).

- How about linear sorting algorithm? No, bucket sort requires O(n) space.
- Mapping all positive integers to a hash table and iterate from 1 to the length of the array to find out the first missing one? No, hash table requires O(n) space.

Then, how to solve this?

Let's take another look at the problem. It is asking for the first missing POSITIVE integer.
So, given a number in the array,

- if it is non-positive, ignore it;
- if it is positive, say we have A[i] = x, we know it should be in slot A[x-1]! That is to say, we can swap A[x-1] with A[i] so as to place x into the right place.

We need to keep swapping until all numbers are either non-positive or in the right places. The result array could be something like [1, 2, 3, 0, 5, 6, ...]. Then it's easy to tell that the first missing one is 4 by iterate through the array and compare each value with their index.

Here are two implementations.
I:

```
public int firstMissingPositive(int[] A) {
  // in-place swap (bucket sort)
  int i = 0;
  while (i < A.length) {
    if (A[i] > 0 && A[i] <= A.length && A[i] != i+1 && A[i] != A[A[i]-1]) {
      int temp = A[A[i]-1];
      A[A[i]-1] = A[i];
      A[i] = temp;
    } else {
      ++i;
    }
  }
  // find the first positive missing integer
  i = 0;
  while (i < A.length && A[i] == i+1) ++i;
  return i+1;
}
```

II:

```
public int firstMissingPositive(int[] A) {
  // in-place swap (bucket sort)
  for (int i=0; i<A.length; ++i) {
    while (A[i] > 0 && A[i] <= A.length && A[i] != i+1 && A[i] != A[A[i]-1]) {
      int temp = A[A[i]-1];
      A[A[i]-1] = A[i];
      A[i] = temp;
    }
  }
  // find the first positive missing integer
  int index = 0;
  while (index < A.length && A[index] == index+1) ++index;
  return index+1;
}
```

Since we only touch each value O(1) times, either swap it with another or check it with its index, this algorithm runs in time O(n) and use constant space.

Posted 4th March 2013 by Sophie

Labels: Array, Java, Sort

2 View comments

3rd March 2013

Edit Distance

Edit Distance [http://leetcode.com/onlinejudge#question_72]

Given two words word1 and word2, find the minimum number of steps required to convert word1 to word2. (Each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

Solution - Recursion

Edit Distance is to calculate Levenshtein distance [http://en.wikipedia.org/wiki/Levenshtein_distance] between two words.
Mathematically, the Levenshtein distance between two strings a and b is given by

[http://upload.wikimedia.org/math/2/b/7/2b7794edd1dd0c6b0c9feaf603b3f765.png]
where i and j are the lengths of the string a and b, respectively.
Intuitively, this can be solved by recursion.

```
public int minDistance(String word1, String word2) {
  return minDist(word1, word2, word1.length(), word2.length());
}

private int minDist(String w1, String w2, int l1, int l2) {
  if (l1 == 0)  return l2;
  if (l2 == 0)  return l1;
  return Math.min(
    Math.min(minDist(w1, w2, l1-1, l2) + 1,
             minDist(w1, w2, l1, l2-1) + 1),
    minDist(w1, w2, l1-1, l2-1) + (w1.charAt(l1-1)==w2.charAt(l2-1) ? 0 : 1));
}
```

But the downside of recursion in this case is that it has lots of redundant computations. If we can store the previous results and reuse them as needed...

Here comes DP solution!

Solution - DP

Dynamic Programming [http://en.wikipedia.org/wiki/Dynamic_programming] is to break down the original problem into smaller subproblems, each of them being solved once and reused for successor subproblems (c.f. Divide-and-Conquer [http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm]).

Back to this problem.

Notice that to calculate lev(i, j), we need lev(i-1, j), lev(i, j-1), and lev(i-1, j-1). So, we can build up a l1*l2 table, where l1 and l2 are the lengths of two strings, respectively, and then reuse the previous calculated values.

Furthermore, since we only need values of (i-1, j), (i, j-1), and (i-1, j-1), we don't need to maintain the entire table. We only need a table of two rows to maintain the results from last row. Plus, use bit manipulation [http://sophie-notes.blogspot.com/2013/02/bit-manipulation-tutorial-i.html] to select a row can be way faster than mod operation.

```
public int minDistance(String word1, String word2) {
  int l1=word1.length(), l2=word2.length();
  if (l1 == 0) return l2;
  if (l2 == 0) return l1;

  int[][] distTable = new int[l1][2];
  distTable[0][0] = (word1.charAt(0)==word2.charAt(0) ? 0 : 1);
  for (int i=1; i<l1; ++i) {
    distTable[i][0] = Math.min(distTable[i-1][0] + 1,
                  (word1.charAt(i)==word2.charAt(0) ? i : i+1));
  }
  for (int j=1; j<l2; ++j) {
    int last = (j-1)&1, cur = j&1;
    distTable[0][cur] = Math.min(distTable[0][last] + 1,
                  (word1.charAt(0)==word2.charAt(j) ? j : j+1));
    for (int i=1; i<l1; ++i) {
      distTable[i][cur] = Math.min(
        Math.min(distTable[i][last] + 1, distTable[i-1][cur] + 1),
        distTable[i-1][last] + (word1.charAt(i)==word2.charAt(j) ? 0 : 1));
    }
  }
  return distTable[l1-1][(l2-1)&1];
}
```

The above algorithm runs in time O(l1*l2) and uses O(l1) spaces.

A small optimization is to find the shorter string and reduce the space usage to O(min(l1, l2)).

```
public int minDistance(String word1, String word2) {
  String w1, w2;
  // find the shorter one
  if (word1.length() < word2.length()) {
    w1 = word1; w2 = word2;
  } else {
    w1 = word2; w2 = word1;
  }
  int l1 = w1.length(); l2 = w2.length();

  if (l1 == 0) return l2;

  int[][] dist = new int[l1][2];
  int row = 0;
  dist[0][row] = (w1.charAt(0) == w2.charAt(0)) ? 0 : 1;
  // initialize first row
  for (int j=1; j<l1; ++j) {
    dist[j][row] = Math.min(dist[j-1][row] + 1, (w1.charAt(j) == w2.charAt(0)) ? j : j+1);
  }
  // the rest rows
  for (int i=1; i<l2; ++i) {
    int last = row;
    row = 1 - row;
    dist[0][row] = Math.min(dist[0][last] + 1, (w1.charAt(0) == w2.charAt(i)) ? i : i+1);
    for (int j=1; j<l1; ++j) {
      dist[j][row] = dist[j-1][last] + (w1.charAt(j) == w2.charAt(i) ? 0 : 1);
      dist[j][row] = Math.min(dist[j][row], dist[j][last] + 1);
      dist[j][row] = Math.min(dist[j][row], dist[j-1][row] + 1);
    }
  }

  return dist[l1-1][row];
}
```

Posted 3rd March 2013 by Sophie

Labels: DP, Java, Recursion, String

0 Add a comment

22nd February 2013

Divide Two Integers

Divide Two Integers [http://leetcode.com/onlinejudge#question_29]

Divide two integers without using multiplication, division and mod operator.

Solution

First, how does a division work?

Dividend / Divisor = Quotient   =>   Dividend = Divisor * Quotient

Intuitively, if we subtract Dividend by divisor repeatedly and count, when the dividend is less than divisor, then the count would be the quotient. But in the case of dividend is >> divisor, e.g. 69165042 / 2, this method requires nearly O(n) time to do the subtractions.

Alternatively, we can increase divisor exponentially which gives us an O(logn) solution.
Each time we double the divisor by add it to itself or shift it by 1 (Yeah, it didn't say that we can't use bit manipulations [http://en.wikipedia.org/wiki/Bit_manipulation] !).
Bit manipulations can make the code looks much more concise (but may lost readability somehow :). Here is a tutorial [http://www.matrix67.com/blog/archives/263] (in Chinese...). With the help of bit manipulations, we don't need to keep track of the current multiple of divisor since we know it must be doubled in each iteration.

Okey, till now, everything seems fine and we seems to solve the problem. BE CAREFUL, here is a tricky part of this problem: overflow. Why?

Think about this number, -2147483648.

In our algorithm, we need to flip negatives to positives, which produces 2147483648, i.e. 2^32, i.e. -1 as an integer! So, we cast both dividend and divisor to long and then start the computations.

Here is the final algorithm:

```
public int divide(int dividend, int divisor) {
  if (dividend == 0 || divisor == 1) return dividend;
  if (divisor == -1) return 0-dividend;

  long divd = Math.abs((long)dividend), divs = Math.abs((long)divisor);

  ArrayList<Long> divisors = new ArrayList<Long>();
  while (divs <= divd) {
    divisors.add(divs);
    divs = divs << 1;
  }
  int result = 0, cur = divisors.size() - 1;
  while (divd > 0 && cur >= 0) {
    while (divd >= divisors.get(cur)) {
      divd -= divisors.get(cur);
      result += 1 << cur;
    }
    --cur;
  }

  return (dividend>0)^(divisor>0) ? (-result) : result;
}
```

Posted 22nd February 2013 by Sophie

Labels: BitManipulation, DataType, Java

1 View comments

19th February 2013

Find the Number of Distinct Subsequences

Distinct Subsequences [http://leetcode.com/onlinejudge#question_115]

Given a string S and a string T, count the number of distinct subsequences of T in S.

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not).

Here is an example:

Given S = "rabbbit", T = "rabbit", returns 3.

Solution - Recursion

This problem can be solved in a recursive way.
Suppose f(i, j) is the number of distinct recurrences of T in S, where i and j are the lengths of S and T, respectively. Then we have

f(i, j) = f(i-1, j) + (S[i]==T[j]) ? 0 : f(i-1, j-1), i >= j

For example, we have "rabbit" and "bit". The number of recurrences can be found as follows:

- find "bit" in "rabbi", find "bit" in "rabb", find "bit" in "rab", find "bit" in "ra" => 0
- find "bi" in "rabbi", find "bi" in "rabb", ... => 0
- find "b" in "rabb", find "b" in "rab", ... => 2

Here is the algorithm:

```
private int numSubseq(String S, String T, int si, int ti) {
  if (si<0 || ti<0 || si<ti) return 0;

  if (S.charAt(si)==T.charAt(ti)) {
    if (ti==0)
      return numSubseq(S, T, si-1, ti) + 1;
    else
      return numSubseq(S, T, si-1, ti) + numSubseq(S, T, si-1, ti-1);
  }

  return numSubseq(S, T, si-1, ti);
}

public int numDistinct(String S, String T) {
  return numSubseq(S, T, S.length()-1, T.length()-1);
}
```

Solution - DP

Notice that in the prior solution there are lots of duplicate computations. To optimize it, we can sacrifice space for time, using Dynamic Programming [http://en.wikipedia.org/wiki/Dynamic_programming] . (See Chapter 15 in Book Introduction to Algorithms [http://www.amazon.com/gp/product/0262033844/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=0262033844&linkCode=as2&tag=n00tc0d3r-20])

This time we store all previous computation results in a table of size i*j, where i and j are the lengths of S and T, respectively.

table[i][j] = the number of distinct subsequences of T[0,j] in S[0,i]

A later result can be calculated using pre-stored values. By doing this, we arrive at an algorithm with O(i*j) running time and O(i*j) spaces.

```
public int numDistinct(String S, String T) {
  int si = S.length(), ti = T.length();
  if (si<=0 || ti<=0 || si<ti) return 0;

  int[][] dptable = new int[si][ti];
  if (S.charAt(0) == T.charAt(0))
    dptable[0][0] = 1;
  else
    dptable[0][0] = 0;
  for (int j=0; j<ti; ++j) {
    for (int i=1; i<si; ++i) {
      dptable[i][j] = dptable[i-1][j];
      if (S.charAt(i) == T.charAt(j)) {
        if (j==0)
          dptable[i][j] += 1;
        else
          dptable[i][j] += dptable[i-1][j-1];
      }
    }
  }

  return dptable[si-1][ti-1];
}
```

Solution - DP2

In the above algorithm, in its i-th iteration, it only needs results in (i-1)-th iteration. More specifically, it only needs table[i-1][j] and table[i][j-1]. So, we can use one row, i.e. O(j) space, rather than a 2D table, i.e. O(i*j) space.

When we use an 1-dimensional array, at the beginning of i-th iteration, recurs[j] means the number of recurrences of T(0:j) in S(0:i-1) and it can be updated as recurs[j]+=recurs[j-1] if the current character matches.

> In i-th iteration,
> at the beginning, recurs[j] = the number of distinct subsequences of T[0,j] in S[0,i-1];
> after being updated, recurs[i] = the number of distinct subsequences of T[0,j] in S[0,i].

We have to run from T.length down to 0 since we don't want to overwrite recurs[j] which would be used for recurs[j+1] later.

```
public int numDistinct(String S, String T) {
  int si = S.length(), ti = T.length();
  if (si<=0 || ti<=0 || si<ti) return 0;

  int[] recurs = new int[ti];
  for (int i=0; i<si; ++i) {
    for (int j=Math.min(i, ti-1); j>=0; --j) {
      if (S.charAt(i)==T.charAt(j)) {
        recurs[j] += (j==0) ? 1 : recurs[j-1];
      }
    }
  }
  return recurs[ti-1];
}
```

Posted 19th February 2013 by Sophie

Labels: DP, Java, Recursion, String

5 View comments

5th February 2013

Decode a Message

Decode Ways [http://leetcode.com/onlinejudge#question_91]

A message containing letters from A-Z is being encoded to numbers using the following mapping:

'A' -> 1
'B' -> 2
...
'Z' -> 26

Given an encoded message containing digits, determine the total number of ways to decode it.

For example, given encoded message "1235", it could be decoded as "AB" (1 2 3 5) or "L" (12 3 5).
The number of ways decoding "1235" is 2.

Solution

Intuitively, this problem can be solved via recursion. For example, given "1235", the number of decoding ways can be calculated as (1)"235" + (12)"35" and then proceed recursively until the end of the string. But notice that there will be duplicate computations, e.g. "35" will be calculated twice in each branch.

This gives us a hint of using Dynamical Programming to solve this problem.

Before we starts, "one more thing": notice that coding started with 1 (not 0). It implies that a message like '021', '1001', etc. would be invalid!

Suppose we have an array of letters, A[1..n], and we already know the number of ways to decode A[1], A[1..2], A[1..i-1], how many ways to decode A[1..i]?

- If A[i] is '0' and A[i-1] is neither '1' nor '2', invalid message;
- If A[i] is '0' and A[i-1] is '1' or '2', the number of ways is the number of ways of A[1..i-2];
- If A[i] can be combined with prior letter, the number of ways is the number of ways of A[1..i-1] + A[1..i-2];

- Otherwise, it is equal to the number of ways of A[1..i-1].

Here is the algorithm.
Note: We add an extra count for A[2] so that we compute A[2] within the iteration and don't need to check whether i==2 repeatedly.

```
private boolean isTwoDigitCode(char a, char b) {
  return (a=='1' || (a=='2' && b>='0' && b<='6'));
}

public int numDecodings(String s) {
  int len = s.length();

  int[] count = new int[len+1];
  count[0] = (len <= 0) ? 0 : 1;
  for (int i=1; i<=len; ++i) {
    char c = s.charAt(i-1);
    if (c < '0' || c > '9')  return 0;  // invalid: non-digit
    if (c == '0') {
      // invalid: start with 0 or previous digit is not 1 nor 2
      if (i-1 == 0 || (s.charAt(i-2) != '1' && s.charAt(i-2) != '2'))  return 0;
        count[i] = count[i-2];
    } else if (i-1 > 0 && isTwoDigits(s.charAt(i-2), c)) {
        count[i] = count[i-2] + count[i-1];
    } else {
        count[i] = count[i-1];
    }
  }
  return count[len];
}
```

This algorithm can be refactored as follows.

```
private boolean isTwoDigitCode(char a, char b) {
  return (a=='1' || (a=='2' && b<='6'));
}

public int numDecodings(String s) {
  int len = s.length();

  int[] count = new int[len+1];
  count[0] = (len <= 0) ? 0 : 1;
  for (int i=1; i<=len; ++i) {
    char c = s.charAt(i-1);
    if ( c != '0' )  count[i] = count[i-1];
    if ( i-1 > 0 && isTwoDigitCode(s.charAt(i-2), c) ) {
      count[i] += count[i-2];
    }
    if ( count[i] == 0 ) return 0;  // invalid
  }
  return count[len];
}
```

It's easy to see that both algorithms run in time O(n) and use O(n) spaces.

Notice that in above algorithm, we only need previous two counts, count[i-1] and count[i-2]. So, we can use two integers instead of an array! This can reduce the space usage to O(1).

```
public int numDecodings(String s) {
  int len = s.length();

  int w1 = (len <= 0) ? 0 : 1, w2 = w1;
  for (int i=1; i<=len; ++i) {
    char c = s.charAt(i-1);
    int temp = w2; // w1 = count[i-2], temp = w2 = count[i-1]

    // update w2 to be count[i]
    if ( c == '0' )  w2 = 0;
    if ( i-1 > 0 && isTwoDigitCode(s.charAt(i-2), c) ) {
      w2 += w1;
    }
    if ( w2 == 0 ) return 0;  // invalid

    // set w1 = temp = count[i-1], i.e. A[(i+1)-2]
    w1 = temp;
  }
  return w2;
}
```

Posted 5th February 2013 by Sophie

Labels: DP, Java, String

0 Add a comment

4th February 2013

Count and Say

Count and Say [http://leetcode.com/onlinejudge#question_38]

The count-and-say sequence is the sequence of integers beginning as follows:
1, 11, 21, 1211, 111221, ...

1 is read off as "one 1" or 11.
11 is read off as "two 1s" or 21.
21 is read off as "one 2, then one 1" or 1211.

Given an integer n, generate the nth sequence.
Note: The sequence of integers will be represented as a string.

Solution

This question is to build a string in a "count-char-count-char" style. Since this requires a lot string append operations, we should use StringBuilder rather than String. (See here [http://sophie-notes.blogspot.com/2013/02/java-stringbuilder-vs-string.html] for more details.)

```
public String countAndSay(int n) {
  if (n<=0) return null;
  StringBuilder res = new StringBuilder("1");
  while (n-- > 1) {
    StringBuilder temp = new StringBuilder();
    int count = 1;
    for (int i=1; i<res.length(); ++i) {
      if (res.charAt(i) == res.charAt(i-1)) {
        count++;
      } else {
        temp.append(count);
        temp.append(res.charAt(i-1));
        count = 1; // reset
      }
    }
    temp.append(count);
    temp.append(res.charAt(res.length()-1));
    res = temp;
  }
  return res.toString();
}
```

We have to generate sequences one by one, and for each one, we have to iterate through the prior one. There is no further room to reduce the running time. The total running time would be the total length of all sequences.

Posted 4th February 2013 by Sophie

Labels: Java, String, StringBuilder

1 View comments

4th February 2013

Convert Sorted Array/List to BST

Convert Sorted Array to Binary Search Tree [http://leetcode.com/onlinejudge#question_108]

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

Solution

First, don't mess up Binary Search Tree [http://en.wikipedia.org/wiki/Binary_search_tree] with Binary Tree. A BST is a binary tree and **sorted**.

To search in a BST, we do it recursively and achieve O(logn) time. Similarly, we can build up a BST recursively. Given a sorted list, it is simple:

- Pick up the middle element of the array
- Set it as the root
- Recursively build up left and right subtrees with lower and higher halves of the array

```
public TreeNode sortedArrayToBST(int[] num) {
  return arrayToBST(num, 0, num.length-1);
}

private TreeNode arrayToBST(int[] data, int low, int high) {
  if (low > high) return null;
  int mid = low + (high - low) / 2;
  TreeNode root = new TreeNode(data[mid]);
  root.left = arrayToBST(data, low, mid-1);
  root.right = arrayToBST(data, mid+1, high);
  return root;
}
```

Running time: $T(n) = 2T(n/2) + O(1) = O(n)$.

Convert Sorted List to Binary Search Tree [http://leetcode.com/onlinejudge#question_109]

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

Naive Solution

We can use the same idea as above: look for the middle element in the list, set it as the root and then recursively build up left and right subtrees. But the thing here is that we have to iterate through half of list to find the middle element!

```
1:  public TreeNode sortedListToBST(ListNode head) {
2:    // calculate list length
3:    int len = 0; ListNode cur = head;
4:    while (cur!=null) {
5:      cur = cur.next;
6:      len++;
7:    }
8:    // build the BST
9:    return listToBST(head, 0, len-1);
10:  }
11:
12:  private TreeNode listToBST(ListNode head, int low, int high) {
13:    if (low > high) return null;
14:    int mid = (low + high) / 2;
15:    // find the middle listnode
16:    ListNode midNode = head;
17:    int cur = mid;
18:    while (cur > low) {
19:      midNode = midNode.next;
20:      cur--;
21:    }
22:    // build up tree recursively
23:    TreeNode root = new TreeNode(midNode.val);
24:    root.left = listToBST(head, low, mid-1);
25:    root.right = listToBST(midNode.next, mid+1, high);
26:    return root;
27:  }
```

What's the running time now?

To get the length of the list (Ln 3-7), O(n) time. To build up the BST, T(n) = 2T(n/2) + n/2 = O(nlogn). So, in total, O(nlogn).

Faster Solution

Let's take a closer look at what are doing in prior solution:

- Go through half of the list to find the root.
- Go through half of the two sublists to find the roots for left and right subtree.

In each recursion, we have to iteration through the list, although we have done that in subroutines. Note that one thing we can take advantage from a list is that we could move the head forward during recursions. That is,

- Go through the first half of the list to build left subtree and return the root. Also move the head forward during the process.
- Use the current head as the root and move the head to next.
- Go through the rest half of the list to build right subtree.

By doing this, we don't need to repeatedly iterate through the list. We only visit each list node ONCE.

Compared with the prior solution, which builds up the BST in a top-down fashion, now we do it in a bottom-up way: assuming we have the length of the list (which can be preprocessed in O(n) time), we know which part would be in left subtree and while part would be in right one. Here is the code as follows.

Note: In Ln 20-23, we didn't move the head with `head = head.next`. It is because that Java pass in Objects by reference and thus changing the reference have no impact on the original Object (i.e. the original head). But we can change the content of the reference! So, here we modify the head to be its successor.

```
1:  public TreeNode sortedListToBST(ListNode head) {
2:    // calculate list length
3:    int len = 0; ListNode cur = head;
4:    while (cur!=null) {
5:      cur = cur.next;
6:      len++;
7:    }
8:    // build the BST
9:    return listToBST(head, 0, len-1);
10:  }
11:
12:  private TreeNode listToBST(ListNode head, int low, int high) {
13:    if (low > high) return null;
14:    int mid = low + (high - low) / 2;
15:    // build up tree recursively
16:    TreeNode left = listToBST(head, low, mid-1);
17:    TreeNode root = new TreeNode(head.val);
18:    root.left = left;
19:    // Java pass in Object by reference, so we can't change head but we can change its content :)
20:    if (head.next != null) { // "move to next"
21:      head.val = head.next.val;
22:      head.next = head.next.next;
23:      root.right = listToBST(head, mid+1, high);
24:    }
25:    return root;
26:  }
```

Running time: T(n) = 2T(n) + O(1) = O(n)!

Posted 4th February 2013 by Sophie

Labels: Array, BinarySearchTree, BinaryTree, Java, LinkedList, Recursion, Tree

2 View comments

3rd February 2013

Container With Most Water

Container With Most Water [http://leetcode.com/onlinejudge#question_11]

Given n non-negative integers a1, a2, ..., an, where each represents a point at coordinate (i, ai). n vertical lines are drawn such that the two endpoints of line i is at (i, ai) and (i, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.
Note: You may not slant the container.

Naive Solution

A simple way to do it is to go through the height array, calculate possible volumes for the current height together with all other heights, and find the maximum.

```
public int maxArea(int[] height) {
  int maxArea = 0;
  for (int i=0; i<height.length-1; ++i) {
    for (int j=i+1; j<height.length; ++j) {
      int area = (j - i) * Math.min(height[i], height[j]);
      if (area > maxArea) maxArea = area;
    }
  }
  return maxArea;
}
```

It is easy to implement. But since it calculates all possible combinations, it takes time O(n^2).

Faster Solution

Take a closer look at the problem.

Given a container with boundaries of unequal heights, how much water can it accommodate?
- It would be width times the **smaller** height.

Suppose we have an array of five heights, {3, 5, 2, 7, 1}. We can sort the array in descending order and also store their original indice with them.

{[3] 7, [1] 5, [0] 3, [2] 2, [4] 1}

For a container with one boundary of 5 and the other >=5, how much water can it accommodate at most?
- 5 * (3-1)
What if a container with height >=3?
- 3 * (3-0)
What if a container with height >=2?
- 2 * (2-0)

So, the key here is that given height of one board, we need to find the furthest higher board to build a container. With a sorted array in hand, we can go through it one by one, keep track of the current index range of visited heights, and compute the volume by multiply the current height with the furthest distance to visited heights.

```
public int maxArea(int[] height) {
  int len = height.length;
  // map height to index
  int[] index = new int[len];
  for (int i=0; i<len; ++i) {
    index[i] = i;
  }
  // sort in descending order and update index array correspondingly
  mergeSortWithIndex(height, index, 0, len-1);
  // find the furthest higher boundary
  int left = Math.min(index[0], index[1]);
  int right = Math.max(index[0], index[1]);
  int maxArea = height[1] * (right - left);
  for (int i=2; i<len; ++i) {
    int id = index[i];
    int width = 0;
    if (id > right) {
      right = id;
      width = id - left;
    } else if (id < left) {
      left = id;
      width = right - id;
    } else {
      width = Math.max(id-left, right-id);
    }
    maxArea = Math.max(maxArea, height[i]*width);
  }
  return maxArea;
```

We use Merge sort here and thus sorting (Ln 31-54) takes time O(nlogn). We then go through the sorted array once to find the maximum volume (Ln 11-27), which takes time O(n).
Overall, this algorithm takes time O(nlogn).

Fastest Solution

There are two factors deciding the volume of a container: width and height. So, if we starts from the two ends of the array and moves towards middle, the only bottleneck would be the height.

We can apply greedy strategy as follows:
We always move the shorter boundary of the two. By moving the shorter one, we may arrive at a higher boundary so as to get a greater volume (although width decreased); it is not necessary to move the higher one since no matter if the next height is greater or smaller, it won't change the volume -- the shorter boundary is the limit for a container.

```
public int maxArea(int[] height) {
  int len = height.length, low = 0, high = len -1 ;
  int maxArea = 0;
  while (low < high) {
    maxArea = Math.max(maxArea, (high - low) * Math.min(height[low], height[high]));
    if (height[low] < height[high]) {
      low++;
    } else {
      high--;
    }
  }
  return maxArea;
}
```

Correctness

Prove by contradiction:
Suppose the returned result is not the optimal solution. Then there must exist an optimal solution, say a container with a_ol and a_or (left and right respectively), such that it has a greater volume than the one we got. Since our algorithm stops only if the two pointers meet. So, we must have visited one of them but not the other.
WLOG, let's say we visited a_ol but not a_or.
When a pointer stops at a_ol, it won't move until

- The other pointer also points to a_ol;
  In this case, iteration ends. But the other pointer must have visited a_or on its way from right end to a_ol. Contradiction to our assumption that we didn't visit a_or.
- The other pointer arrives at a value, say a_rr, that is greater than a_ol before it reaches a_or.
  In this case, we does move a_ol. But notice that the volume of a_ol and a_rr is already greater than a_ol and a_or (as it is wider and heigher), which means that a_ol and a_or is not the optimal solution -- Contradiction!

Both cases arrive at a contradiction.

Complexity

Obviously, this algorithm runs in time O(n) as iterating through the array only once.

Posted 3rd February 2013 by Sophie

Labels: Array, GreedyAlgorithm, Java, Sort

3 View comments

25th January 2013

Combination Sum

Combination Sum

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.
The same repeated number may be chosen from C **unlimited number of times**.
For example, given candidate set 2,3,6,7 and target 7, a solution set is:
[7]
[2, 2, 3]

Solution

Since a number can be reused, we can assume that the candidate set must not contain duplicates.

We also assume that all candidates and target are positive integers. Why? We need to iterate all possibilities, if negatives are allowed, it may fall into infinite loop. E.g. {2, -2, 3, 4}, 7.

Let's use the given example to show how to solve the problem.
We start with the target 7, a set {2, 3, 6, 7} and an empty list to keep track of the "path".

Path = {2}, new target 5=7-2, {2, 3, ~~6, 7~~} (skip the large numbers from our candidate set);
  Path = {2, 2}, new target 3, {2, 3};
    Path = {2, 2, 2}, new target 1, {};
    Path = {2, 2, 3}, hit target! Store the resulting path in our result set and continue with the next iteration.
  Path = {2, 3}, new target 2, {~~3, 6, 7~~}
Path = {3}, new target 4, {3, ~~6, 7~~}
  Path = {3, 3}, new target 1, {}
Path = {6}, new target 1, {}
Path = {7}, hit target! Store into results set.

As shown above, in each iteration, we remove the prior ones from candidate set so as to avoid duplicate results, but leave the current one in the set so as to reuse it when possible.

Also, notice that sorting is not necessary. We skip large candidates if they are bigger than target, but it's just an improvement of the running time performance and it won't even reduce big-O complexity.

```
public ArrayList<ArrayList<Integer>> combinationSum(int[] candidates, int target) {
  ArrayList<ArrayList<Integer>> results = new ArrayList<ArrayList<Integer>>();
  Arrays.sort(candidates);
  addUp(candidates, 0, target, new ArrayList<Integer>(), results);
  return results;
}

private void addUp(int[] candidates, int start, int target, ArrayList<Integer> path,
    ArrayList<ArrayList<Integer>> results) {
  if (start < 0 || target < 0) return;

  if (0 == target) {
    ArrayList<Integer> res = new ArrayList<Integer>(path);
    results.add(res);
  } else {
    for (int i=start; i<candidates.length && candidates[i] <= target; ++i) {
      // if (candidates[i] > target) continue; // if we don't sort the data at the beginning, we skip
      path.add(candidates[i]);
      addUp(candidates, i, target - candidates[i], path, results);
      path.remove(path.size() - 1);
    }
  }
}
```

Combination Sum II [http://leetcode.com/onlinejudge#question_40]

Given a collection of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.
Each number in C may only be used **once** in the combination.
For example, given candidate set 10,1,2,7,6,1,5 and target 8, a solution set is:
[1, 7]
[1, 2, 5]
[2, 6]
[1, 1, 6]

Solution

Notice that the differences between the two variaties are:

- there may exist duplicate numbers in candidate set;
- each candidate can by used at most once.

The former one tells us that we need to skip duplicates when we iterate through candidates;
The latter means we should start with the next candidate during iterations.

I've highlighted the code differences between the two of them.

```
public ArrayList<ArrayList<Integer>> combinationSum2(int[] candidates, int target) {
  ArrayList<ArrayList<Integer>> results = new ArrayList<ArrayList<Integer>>();
  Arrays.sort(candidates);
  addUp(candidates, 0, target, new ArrayList<Integer>(), results);
  return results;
}

private void addUp(int[] candidates, int start, int target, ArrayList<Integer> path,
    ArrayList<ArrayList<Integer>> results) {
  if (start < 0 || target < 0) return;

  if (0 == target) {
    ArrayList<Integer> res = new ArrayList<Integer>(path);
    results.add(res);
  } else {
    for (int i=start; i<candidates.length && candidates[i] <= target; ++i) {
      if (i>start && candidates[i] == candidates[i-1]) continue; // skip duplicates
      path.add(candidates[i]);
      addUp(candidates, i+1, target - candidates[i], path, results);
      path.remove(path.size() - 1);
    }
  }
}
```

Posted 25th January 2013 by Sophie

Labels: Java, Recursion

2 View comments

Combinations

21st January 2013

Combinations [http://leetcode.com/onlinejudge#question_77]

Given two integers n and k, return all possible combinations of k numbers out of 1 ... n.
For example, if n = 4 and k = 2, a solution is: [[1,2], [1,3], [1,4], [2,4], [2,3], [3,4]].

Solution I

Generalizing the given example to all n's and k's, then a possible combination contains k numbers, i.e. an array of size k such that

- A[0] may contain 1, 2, ..., n-k+1, i.e. any number ranging from 1 to n-k+1
- A[1] may contain 2, 3, ..., n-k+1, i.e. any number ranging from A[0]+1 to n-(k-1)+1
- Similarly, A[2] may contain any number ranging from A[1]+1 to n-(k-2)+1
- ... ...
- A[k-1] may contain any number from A[k-1]+1 to n-(k-(k-1))+1=n

We can solve it recursively.
In the i-th round, based on the partial array from (i-1)-th round, set A[i] to a number from A[i-1]+1 to n-(k-1)+1, and pass result to next rounds.
Recursive implementation:

```
public ArrayList<ArrayList<Integer>> combine(int n, int k) {
  ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
  combineHelper(1, n, k, new ArrayList<Integer>(), res);
  return res;
}

private void combineHelper(int s, int n, int k, ArrayList<Integer> path, ArrayList<ArrayList<Integer>>
  if (n == 0) return ;
  if (k == 0) {
    ArrayList<Integer> r = new ArrayList<Integer>(path);
    res.add(r);
    return ;
  }

  for (int i=s; i<=n-k+1; ++i) {
    path.add(i);
    combineHelper(i+1, n, k-1, path, res);
    path.remove(path.size()-1);
  }
}
```

Iterative implementation:

```
public ArrayList<ArrayList<Integer>> combine(int n, int k) {
  ArrayList<ArrayList<Integer>> results = new ArrayList<ArrayList<Integer>>();

  for (; k>0; --k) {
    int num = res.size();
    // add first element
    if (num == 0) {
      for (int i=1; i<=n-k+1; ++i) {
        ArrayList<Integer> res = new ArrayList<Integer>();
        res.add(i);
        results.add(res);
      }
      continue;
    }

    // append rest
    for (int j=0; j<num; ++j) { // loop through results from (i-1)-level
      ArrayList<Integer> cur = results.get(j);
      int last = cur.get(cur.size() - 1);
      for (last+=1; last<n-k+1; ++last) {
        ArrayList<Integer> res = new ArrayList<Integer>(cur);
        res.add(last);
        results.add(res);
      }
      cur.add(last);
    }
  }

  return results;
}
```

Solution II

Alternatively, we can think it in a "backward" way.
Given n=4 and k=2, a solution is [[4,1], [4,2], [4,3], [3,1], [3,2], [2,1]].
Generalizing to all n's and k's, we have a set of

- Append n to the result of combine(n-1, k-1);
- Append n-1 to the result of combine(n-2, k-2);
- ... ...
- Append k-1 to the result of combine(n-k+1, 1), i.e. (1, 2, ..., k).

```
public ArrayList<ArrayList<Integer>> combine(int n, int k) {
  ArrayList<ArrayList<Integer>> resSet = new ArrayList<ArrayList<Integer>>();
  if (k == 1) {
    for (int i=1; i<=n; ++i) {
      ArrayList<Integer> res = new ArrayList<Integer>();
      res.add(i);
      resSet.add(res);
    }
    return resSet;
  }
  for (int i=n; i>=k; --i) {
    ArrayList<ArrayList<Integer>> results = combine(i-1, k-1);
    for (ArrayList<Integer> res : results) {
      res.add(i);
      resSet.add(res);
    }
  }
  return resSet;
}
```

The code looks neat and much shorter than the prior one. But here, we've done some redundant computations for the subsets since we didn't store any results.

Posted 21st January 2013 by Sophie

Labels: Java, Recursion

2 View comments

21st January 2013

Climbing Stairs

Climbing Stairs [http://leetcode.com/onlinejudge#question_70]

You are climbing a stair case. It takes n steps to reach to the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Solution - Combination

Let's start from a solid example.
Suppose n = 5. How many distinct ways?

- We could climb 1-step all the time. Zero ②, Five ①s -- one way to do it, C(5, 0).
- One ②, Three(=5-2) ①s -- C(4, 1), put one ball in a total of four slots.
- Two ②s, One(=5-2*2) ① -- C(3, 2)

So, we have f(5) = C(5, 0) + C(4, 1) + C(3, 2). Generalizing to all n's,

f(n) = C(n, 0) + C(n-1, 1) + C(n-2, 2) + ... + C(ceiling(n/2), floor(n/2))

```
private int combination(int n, int k) {
  if (k < 0 || n < k) return 0;
  if (k*2 > n) k = n - k;
  double res = 1; int i=0;
  while (i<k) {
    res *= (n-i);
    i++;
    res /= i;
  }
  return (int)res;
}

public int climbStairs(int n) {
  if (n <= 0) return 0;
  int res = 0, last = n/2;
  for (int i=0; i<=last; ++i) {
    res += combination(n-i, i);
  }
  return res;
}
```

The method `combination` runs in linear time. So, the running time for `climbStairs` is O(n^2).

Solution - Recursion

Now, let's think about this problem in a different way.

Suppose f(n) is the number of ways to climb up n step. Since for each step, there are only two ways to do it: one step or two step, we have

f(n) = f(n-1) + f(n-2)

Does this formula looks familiar? Yes, this is the definition of Fibonacci sequence [http://en.wikipedia.org/wiki/Fibonacci_number] !
If you remember the mathematical result of Fib, then you can solve the problem with the formular directly.

[http://upload.wikimedia.org/math/0/9/e/09e7c11b3af63e2c3758b3b4d2ab5bf8.png]
If not, you will get to the following recursion algorithm based on the definition of Fib.

```
public int climbStairs(int n) {
  if (n <= 0) return 0;
  if (n == 1 || n == 2) return n;
  return climbStairs(n-1) + climbStairs(n-2);
}
```

Unfortunately, the running time of the recursion grows up exponentially...

Solution - Iteration

Notice that in the above recursion solution, we did lots of redundant calculations and didn't reuse any previous results. If we rewrite it in an iterative way by using an array to store previous results, we only need to go through the n numbers once. This is a classic Dynamic Programming strategy.

```
public int climbStairs(int n) {
  if (n <= 0) return 0;
  if (n == 1 || n == 2) return n;
  int[] ways = new int[n];
  ways[0] = 1; ways[1] = 2;
  for (int i=2; i<n; ++i) {
    ways[i] = ways[i-1] + ways[i-2];
  }
  return ways[n-1];
}
```

This time the running time is O(n) but it uses O(n) spaces to store previous results. We actually only need to the previous two results. So, we can improve it by using two integers for storage.

```
public int climbStairs(int n) {
  if (n <= 0) return 0;
  int p1 = 1, p2 = 1;
  for (int i=2; i<=n; ++i) {
    int temp = p2;
    p2 += p1;
    p1 = temp;
  }
  return p2;
}
```

Now the running time is same as before but it only uses O(1) space!

Some Maths

Combination [http://en.wikipedia.org/wiki/Combination]

Mathematically, a combination is to select several items from a given set where orders are not matter.

$$\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k!}.$$

[http://upload.wikimedia.org/math/a/a/2/aa2dd943c643e3045db02f71a126f0fc.png]

or

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$

[http://upload.wikimedia.org/math/1/9/2/1928f752016eeb2c94f27269a14f7f47.png]

Useful properties:

- C(n, k) = C(n, n-k)
- C(n, 0) = C(n, n) = 1
- C(n, k) = C(n-1, k-1) + C(n-1, k)
- Binomial Formula (1+X)^2 = SUM_{k>=0} (C(n, k)X^k)

Permutation [http://en.wikipedia.org/wiki/Permutations]

A permutation, on the contrary, is an arrangement of several items selected from a given set.

P(n, k) = n! / (n-k)!

Posted 21st January 2013 by Sophie

Labels: DP, Java, Maths, Recursion

2 View comments

20th January 2013

Tree Path Sum

Find A Path Sum [http://leetcode.com/onlinejudge#question_112]

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

For example: Given the below binary tree and sum = 22, return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.

```
              5
             / \
            4   8
           /   / \
          11  13  4
         /  \      \
        7    2      1
```

Solution

This can be easily solved in O(n) time with recursion. A trick part is that this is asking for a root-to-leaf path, not any path starting from root.

```
public boolean hasPathSum(TreeNode root, int sum) {
  if (root == null) return false;
  if (root.left == null && root.right == null) // get to a leaf
    return (sum == root.val);
  return hasPathSum(root.left, sum-root.val) || hasPathSum(root.right, sum-root.val);
}
```

Find ALL Path Sum [http://leetcode.com/onlinejudge#question_113]

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

For example: Given the below binary tree and sum = 22, return [[5,4,11,2], [5,8,4,5]].

```
              5
             / \
            4   8
           /   / \
          11  13  4
         /  \      \
        7    2      1
```

Solution

In this case, we need to keep track of the current path and also the result set. The basic idea is similar to previous version:

- Recursively visit left and right subtrees of the root and keep the current root updated (be careful about this part);
- If we hit a leaf and the pass-in sum is equal to the value of the node, then add the path to our result set.

Notice that we make a copy of the path before we added it to our result set. The reason is that when we add an Object (here, an ArrayList), Java add a copy of the pointer (i.e. a reference) of the Object, rather than a deep copy. So, any changes to **content** of the original Object will reflect into our final result set.

```
public ArrayList<ArrayList<Integer>> pathSum(TreeNode root, int sum) {
  ArrayList<ArrayList<Integer>> resSet = new ArrayList<ArrayList<Integer>>();
  findPathSum(root, sum, new ArrayList<Integer>(), resSet);
  return resSet;
}

private void findPathSum(TreeNode root, int sum,
                         ArrayList<Integer> path,
                         ArrayList<ArrayList<Integer>> resSet) {
  if (root == null) return;
  path.add(root.val);
  if (root.left == null && root.right == null) { // get to a leaf
    if (sum == root.val) { // found a path
      // has to make a copy, otherwise the content may be changed
      ArrayList<Integer> curPath = new ArrayList<Integer>(path);
      resSet.add(curPath);
    }
  }
  findPathSum(root.left, sum - root.val, path, resSet);
  findPathSum(root.right, sum - root.val, path, resSet);
  path.remove(path.size()-1);
}
```

During recursion, we

- visit each node constant times
- the amortized time of add/remove an item on an ArrayList is O(1)
- making a copy for each result costs at most O(logn).

So, the worst-case running time and space usage could be O(nlogn), where we have a full binary tree and each root-to-leaf path is eligible to be a valid result.

Binary Tree Maximum Path Sum [http://leetcode.com/onlinejudge#question_124]

Given a binary tree, find the maximum path sum. The path may start and end at any node in the tree.

For example: Given the below binary tree, return 6.

```
    -1
   /  \
  1    2
 / \
2   3
```

Solution

Let's start from a simpler version: suppose the path must contain the root. It can be solved in O(n) time as follows:

- Recursively find the max partial path sum in the left subtree (A partial path is either an empty path or a path starting from left child of the root but not necessarily ending at a leaf);
- Recursively find the max partial path sum in the right subtree;
- Return the maximum of (root.value, root.value+max-left-path-sum, root.value+max-right-path-sum, root.value+max-left-path-sum+max-right-path-sum).

Now, go back to the original problem. As said, "The path may start and end at any node", and thus, it may or may not contain the root. We can run the above algorithm on each node, but it requires O(n^2) time and obviously there are lots of redundancies of doing that. Another way is to keep track the current maximum path sum of each node and update it to a new higher value if exists.

So, in each recursion, we need to return the max partial path sum and the current max path sum.

Unfortunately, in Java, we can't pass in an int of current max to keep it updated as Java passes in primitive variables by value (see java-passing-arguments-into-methods.html [http://sophie-notes.blogspot.com/2012/12/java-passing-arguments-into-methods.html]). In this case, we either pass in an array of two item or construct an Object that contains the two. In the code given below, we use the latter way for better readability, it is easy to change it to the former way and we left it to readers.

```
public int maxPathSum(TreeNode root) {
  // pass in an Object that will be filled in the two values
  Data data = maxSubPath(root);
  return data.sum;
}

private class Data {
  int path = 0;
  int sum = Integer.MIN_VALUE;
}

private Data maxSubPath(TreeNode root) {
  Data data = new Data();
  if (root == null) return data;

  Data l = maxSubPath(root.left);
  Data r = maxSubPath(root.right);

  data.path = Math.max(0, Math.max(l.path, r.path) + root.val);
  data.sum = Math.max(Math.max(l.sum, r.sum), l.path+root.val+r.path);
  return data;
}
```

This algorithm runs in time O(n) by visiting each node constant times, and uses O(n) spaces with recursion.

Posted 20th January 2013 by Sophie

Labels: BinaryTree, Java, Recursion, Tree

2 View comments

20th January 2013

Construct Binary Tree from Traversal

Construct Binary Tree from Preorder and Inorder Traversal [http://leetcode.com/onlinejudge#question_105]

Given preorder and inorder traversal of a tree, construct the binary tree.
Note: You may assume that duplicates do not exist in the tree.

Discussion

- Why can we build a binary tree from preorder and inorder traversal?

  Preorder, as its name, always visits root first and then left and right subtrees. That is to say, walking through a preorder list, each node we hit would be a "root" of a subtree (treat a leaf as a tree with no subtrees).
  Inorder, on the other hand, always visits left subtree first and then root and then right subtree, which means that given a root, inorder list can tell us the sizes of its left and right subtrees.
  Think about it.
- Can we build a binary tree from any two traversal of the tree?

  In this section, we discuss how to construct a tree with preorder and inorder traversal. In the next section, we will discuss a similar how-to for postorder and inorder traversal. But we can't build a tree with preorder and postorder traversal. Why? As we can't separate left subtree and right subtree. A simple example is shown below.

```
      1           1
     /     or      \    ??
    2               2
```

Solution

To build a binary tree, intuitively, we can do it recursively. Basically,

- Pick up the first node in the preorder list and create a tree node with the value as root;
- Look it up in the inorder list to determine the sizes of left and right subtrees;
- Recursively build up the left and right subtrees.

Notice that we need to look up each value in the inorder list. If this step takes time O(n), the entire algorithm goes up to O(n^2)! So, we first create a hash table that maps a value to its index in the inorder list. By doing this, the look-up time reduced to O(1) with a O(n) one-time preparation computation.
Be careful about base case and boundary cases: empty tree, a (sub-)tree without left (or right) tree, etc., and about subtree size calculations.

Here is the algorithm.

```
public TreeNode buildTree(int[] preorder, int[] inorder) {
  HashMap<Integer, Integer> inorderMap = new HashMap<Integer, Integer>();
  int len = inorder.length;
  if (len < 1) return null;
  // map node value to its index in inorder results
  for (int i=0; i<inorder.length; ++i) {
    inorderMap.put(inorder[i], i);
  }
  return buildSubTree(preorder, 0, inorderMap, 0, len-1);
}

private TreeNode buildSubTree(int[] preorder, int cur,
               HashMap<Integer, Integer> inorder, int start, int end) {
  TreeNode root = new TreeNode(preorder[cur]);
  if (start < end) { // if more than one node left
    int mid = inorder.get(preorder[cur]);
    if (mid > start) {
      root.left = buildSubTree(preorder, cur+1, inorder, start, mid-1);
    }
    if (mid < end) {
      root.right = buildSubTree(preorder, cur+mid-start+1, inorder, mid+1, end);
    }
  }
  return root; // Base case: start==end, i.e. only one node, simply return it.
}
```

This algorithm runs in time O(n) and uses O(n) spaces (for hash and recursion).

Construct Binary Tree from Inorder and Postorder Traversal [http://leetcode.com/onlinejudge#question_106]

Given inorder and postorder traversal of a tree, construct the binary tree.
Note: You may assume that duplicates do not exist in the tree.

Solution

Notice that postorder visits nodes in a tree in a left->right->root order. If we read postorder list backwards, it gives us a root->right->left order, which is kind of a "mirror-like" way of preorder (root->left->right) traversal. Thus, several small modifications of previous algorithm can solve this problem, as shown below.

```
public TreeNode buildTree(int[] inorder, int[] postorder) {
  int len = inorder.length;
  if (len == 0 || len != postorder.length) return null;
  // map inorder values to their indices
  HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
  for (int i=0; i<len; ++i) {
    map.put(inorder[i], i);
  }
  // build the tree
  // read postorder values backwards
  return buildSubTree(postorder, 0, len-1, len-1, map);
}

private TreeNode buildSubTree(int[] postorder, int start, int end, int cur,
               HashMap<Integer, Integer> inorder) {
  if (start > end)  return null; // Base case: start > end, i.e. invalid index, return null

  int val = postorder[cur];
  TreeNode root = new TreeNode(val);
  int mid = inorder.get(val);
  // read postorder values backwards
  root.left = buildSubTree(postorder, start, mid-1, cur-(end-mid)-1, inorder);
  root.right = buildSubTree(postorder, mid+1, end, cur-1, inorder);

  return root;
}
```

Similarly, this algorithm runs in time O(n) and uses O(n) spaces (for hash and recursion).

Posted 20th January 2013 by Sophie

Labels: BinaryTree, Java, Recursion, Tree

0 Add a comment

14th January 2013

Binary Tree Traversals (II)

Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).

Level Order Traversal [http://leetcode.com/onlinejudge#question_102]

For example, given a tree as shown below, returns [[1], [2,3], [4,5,6], [7,8]].

```
      1
     / \
    2   3
   /\  /
  4  5 6
    /   \
   7     8
```

Solution

Notice that this problem requires to return separate lists for each level. If only one single list for all levels is requires, like [1,2,3,4,5,6,7,8], then we can use a FIFO queue and do a breadth-first tree search (BSF). But to return separate lists for each level, we can

- Either add a 'flag' or a special null node in the queue indicating the end of a level
- Or switch two queues for each level

Using one queue

The basic idea is to add nodes in one level to the queue, from left to right, and add an end-of-level node (a null node) when finishes one level.

1. Append root and a null node to the queue;
2. For each level in the queue, remove the first node from the queue, add its value to an array add its left and right children to the queue if exist;
3. If reach the end of a level, add the array to the result set and add an end-of-level node to the queue if queue is not empty yet.

```
public ArrayList<ArrayList<Integer>> levelOrder(TreeNode root) {
  ArrayList<ArrayList<Integer>> resSet = new ArrayList<ArrayList<Integer>>();
  if (root == null) return resSet;
  LinkedList<TreeNode> queue = new LinkedList<TreeNode>();
  queue.add(root);
  queue.add(null); // flag of end-of-level
  while (!queue.isEmpty()) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    TreeNode cur = queue.removeFirst();
    while (cur != null) { // go through one level
      res.add(cur.val);
      if (cur.left != null) queue.add(cur.left);
      if (cur.right != null) queue.add(cur.right);
      cur = queue.removeFirst();
    }
    // add result of one level
    resSet.add(res);
    // add a flag for end-of-level if needed
    if (!queue.isEmpty()) queue.add(null);
  }
  return resSet;
}
```

We touch each node once and thus the running time is O(n). For space usage, at any time, the queue contains nodes in one level, so in worst case, e.g. a full binary tree, we need O(n) space.

Using a temp queue

When we go through nodes in one level, we can first save their left/right children into a temp queue and switch back after finishing the level. Basically, starting for top level,

1. Add values of nodes in the level to an array;
2. Store left/right children of each node in the level to a temp queue;
3. After finishes the level, add the array to our result set and set the queue to the temp queue;
4. Repeat until no more levels left.

```
public ArrayList<ArrayList<Integer>> levelOrder(TreeNode root) {
  ArrayList<ArrayList<Integer>> resSet = new ArrayList<ArrayList<Integer>>();
  if (root == null) return resSet;
  LinkedList<TreeNode> queue = new LinkedList<TreeNode>();
  queue.add(root);
  while (!queue.isEmpty()) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    LinkedList<TreeNode> queueTmp = new LinkedList<TreeNode>();
    while (!queue.isEmpty()) { // go through one level
      TreeNode cur = queue.removeFirst();
      res.add(cur.val);
      if (cur.left != null) queueTmp.add(cur.left);
      if (cur.right != null) queueTmp.add(cur.right);
    }
    // add result of one level
    resSet.add(res);
    // switch the queue with the temp queue
    queue = queueTmp;
  }
  return resSet;
}
```

Similarly, this algorithm runs in time O(n) using O(n) spaces.

Bottom-up Level Order Traversal [http://leetcode.com/onlinejudge#question_107]

Bottom-up level order traversal: from left to right, level by level from leaf to root.
For example, given a tree as shown below, returns [[7,8], [4,5,6], [2,3], [1]].

```
     1
    / \
   2   3
  /\  /
 4  5 6
   /   \
  7     8
```

Solution

With solutions to level order traversal above, to solve this problem, we only need to reverse the result set at the very end.

```
    // reverse the result array
    Collections.reverse(resSet);
```

This algorithme runs in time O(n).
Note that if we add the value array of each level to the beginning of the result set, it will give us an O(n^2) algorithm as each time we need to move all existing elements in the result set.

Zigzag Level Order Traversal [http://leetcode.com/onlinejudge#question_103]

Zigzag level order traversal: from left to right, then right to left for the next level and alternate between.
For example, given a tree as shown below, returns [[1], [3,2], [4,5,6], [8,7]].

```
     1
    / \
   2   3
  /\  /
 4  5 6
   /   \
  7     8
```

Solution

In this case, we need a LIFO stack for each level so that it could return nodes in a reverse order as we push in. Also, we need to check whether the level is odd level or even and when we read current level from right to left, we need to push right children and then left child for each node.

```
public ArrayList<ArrayList<Integer>> zigzagLevelOrder(TreeNode root) {
  ArrayList<ArrayList<Integer>> resSet = new ArrayList<ArrayList<Integer>>();
  if (root == null) return resSet;
  Stack<TreeNode> stack = new Stack<TreeNode>();
  stack.push(root);
  boolean oddLevel = false; // false - 0,2,4,...,levels
  while (!stack.isEmpty()) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    Stack<TreeNode> stackTmp = new Stack<TreeNode>();
    while (!stack.isEmpty()) { // go through one level
      TreeNode cur = stack.pop();
      res.add(cur.val);
      if (oddLevel) {
        if (cur.right != null) stackTmp.push(cur.right);
        if (cur.left != null) stackTmp.push(cur.left);
      } else {
        if (cur.left != null) stackTmp.push(cur.left);
        if (cur.right != null) stackTmp.push(cur.right);
      }
    }
    // add result of one level
    resSet.add(res);
    // switch
    stack = stackTmp;
    oddLevel = !oddLevel;
  }
  return resSet;
}
```

Similarly, this algorithm runs in time O(n) using O(n) spaces.

Posted 14th January 2013 by Sophie

Labels: BinaryTree, Java, Recursion, Tree

13th January 2013

Binary Tree Traversals (I)

Given a binary tree, return a pre/in/post-order traversal of its nodes' values.
Note: Recursive solution is trivial, let's do these iteratively!

Binary Tree Inorder **Traversal [http://leetcode.com/onlinejudge#question_94]**

For example: Given binary tree as below, return [1,3,2].

```
   1
    \
     2
    /
   3
```

Solution

First, we need to figure out who is the succeed node to be visited. Given the root of a tree,

- If it has left child, visit the left-most leaf of the root;
- If it has no left child, visit the root itself and then go to its right subtree;
- repeat until all nodes have been visited.

To avoid recursion, we need a stack to keep track of the parent nodes along the way.

```
public ArrayList<Integer> inorderTraversal(TreeNode root) {
  ArrayList<Integer> res = new ArrayList<Integer>();
  if (root == null) return res;
  Stack<TreeNode> stack = new Stack<TreeNode>();
  TreeNode cur = root;
  // find the left-most node
  while (cur != null) {
    stack.push(cur);
    cur = cur.left;
  }
  while (!stack.empty()) {
    cur = stack.pop();
    // visit
    res.add(cur.val);
    // add the right child
    cur = cur.right;
    // find the left-most node
    while (cur != null) {
      stack.push(cur);
      cur = cur.left;
    }
  } // while-stack-empty
  return res;
}
```

Since we visit each node once, this algorithm runs in time O(n) using O(h) (worst-case O(n)) spaces, where h is the depth of the tree.

Binary Tree Preorder **Traversal**

For example: Given binary tree as below, return [1,4,3,2].

```
   1
  / \
 4   2
  \
   3
```

Solution

Preorder is simpler than inorder. Basically,

- Walk down the left tree to the bottom, visit each node and then push it into stack;
- Pop out a node from stack (the last visited);
- If it has right subtree, go to step 1 and repeat the same process on the subtree;
- If not, go to step 2 and repeat.

```
public ArrayList<Integer> preorderTraversal(TreeNode root) {
  ArrayList<Integer> res = new ArrayList<Integer>();
  if (root == null) return res;
  Stack<TreeNode> stack = new Stack<TreeNode>();
  TreeNode cur = root;
  // find the left-most node
  while (cur != null) {
    res.add(cur.val);
    stack.push(cur);
    cur = cur.left;
  }
  while (!stack.empty()) {
    cur = stack.pop();
    // go to the right subtree
    cur = cur.right;
    // find the left-most node
    while (cur != null) {
      res.add(cur.val);
      stack.push(cur);
      cur = cur.left;
    }
  } // while-stack-empty
  return res;
}
```

This runs in time O(n) and use O(h) spaces.

Binary Tree Postorder **Traversal**

For example: Given binary tree as below, return [3,4,2,1].

```
   1
  / \
 4   2
  \
   3
```

Solution

Postorder is to visit nodes in a left-right-root order, recursively. Recall that preorder is root-left-right. If we visit the nodes in a "mirrored preorder", that is, root-right-left, and store the values in a stack. After we finish the traversal, pop out values in the stack would give us left-right-root, which is exactly a postorder traversal!

```
public ArrayList<Integer> postorderTraversal(TreeNode root) {
  ArrayList<Integer> res = new ArrayList<Integer>();
  if (root == null) return res;
  Stack<TreeNode> stackBack = new Stack<TreeNode>();
  Stack<Integer> stack = new Stack<Integer>();
  TreeNode cur = root;
  // find the right-most node
  while (cur != null) {
    stack.push(cur.val);
    stackBack.push(cur);
    cur = cur.right;
  }
  while (!stackBack.empty()) {
    cur = stackBack.pop();
    // add the left child
    cur = cur.left;
    // find the right-most node
    while (cur != null) {
      stack.push(cur.val);
      stackBack.push(cur);
      cur = cur.right;
    }
  } // while-stackBack-empty
  // pop values to an array
  while (!stack.empty()) {
    res.add(stack.pop());
  }
  return res;
}
```

This runs in time O(n) and use O(h) spaces.

Posted 13th January 2013 by Sophie

Labels: BinaryTree, Java, Recursion, Tree

0 Add a comment

13th January 2013

Best Time to Buy and Sell Stock

Best Time to Buy and Sell Stock

Say you have an array for which the ith element is the price of a given stock on day i. Design an algorithm to find the maximum profit.

Basic version [http://leetcode.com/onlinejudge#question_121] : At Most One Transaction

You are only permitted to complete at most one transaction (ie, buy one and sell one share of the stock).

Solution

In this case, where we are allowed to make at most one transaction, we can find the max profit incrementally. As time moves forward, any new price we meet

- either it is lower than our previous lowest price (prices[buyDay]) and thus we may consider to buy the stock with this price;
- or it is higher than our previous highest price and thus we would like to sell the stock with this price.

We always store the max profit we got till today and keep it up-to-date.

```
public int maxProfit(int[] prices) {
  int profit = 0, buyDay = 0, sellDay = 1;
  while (sellDay < prices.length) {
    int curProfit = prices[sellDay] - prices[buyDay];
    if (curProfit <= 0)  buyDay = sellDay;
    profit = Math.max(curProfit, profit);
    sellDay++;
  }
  return profit;
}
```

Another implementation

```
public int maxProfit(int[] prices) {
  if (prices.length == 0) return 0;
  int profit = 0;
  int low=prices[0];
  for (int day = 1; day<prices.length; ++day) {
    if (prices[day] < low) {
      // new low price, try to buy
      low = prices[day];
    } else {
      // try to sell
      profit = Math.max(profit, prices[day] - low);
    }
  }
  return profit;
}
```

This can be done in time O(n) with O(1) spaces.

Version II [http://leetcode.com/onlinejudge#question_122] : Multiple Transaction

You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Solution

In this case, we are allowed to make multiple transactions. So, we can buy the stock at "local valley" (ie minimal) and sell it at "local peak" (ie. maximal) so as to maximize our profit.

```
public int maxProfit(int[] prices) {
  int totalProfit = 0, curProfit = 0, buyDay = 0, sellDay = 1;
  while (sellDay < prices.length) {
    if (prices[sellDay] <= prices[sellDay-1]) {
      totalProfit += curProfit;
      curProfit = 0;
      buyDay = sellDay;
      sellDay++;
    } else
      curProfit = Math.max(curProfit, (prices[sellDay++] - prices[buyDay]));
  } // end-while
  totalProfit += curProfit;
  return totalProfit;
}
```

Actually, we don't need to find local minimal and maximal. If today's price is higher than yesterday's, we know we should buy the stock yesterday and sell it today; If it goes down today, no transaction for today.
(Unfortunately, this could not happen in real world since we cannot travel back time. :)

```
public int maxProfit(int[] prices) {
  int totalProfit = 0;
  for (int day=1; day<prices.length; ++day) {
    int profit = prices[day] - prices[day-1];
    if (profit > 0)  totalProfit += profit;
  }
  return totalProfit;
}
```

This can be done in O(n) time with O(1) space.

Version III [http://leetcode.com/onlinejudge#question_123] : At Most Two Transactions

You may complete at most two transactions.
Note: You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Solution

This one is tricky. The first thought might be "ah, find out max and 2nd max in the given time span". An anti-example could be

> [8, 11, 10, 12, 10, 15], where 8~15 is max profit if only one transaction is allowed and then we could don't take the 2nd max profit since we "mush sell the stock before buy it again".

Think it in this way:
We want to split the whole time span into two sub-span such that the total max profit (sum of max profits in each sub-span) would be maximized. As shown above, given a list of prices in a time span, we can find the max profit in O(n) time. There are n possibilities to split. So, it looks like this would give us a O(n^2) solution.

Can we do better?

Yes, use DP.
Recall that we incrementally calculate max profit as we move forward which is done in O(n) time. Similarly, we can calculate max profit backward in O(n) time. If we store all results, backwardProfit[] and forwardProfit[], for each day, then to find the total max profit is simply to find the max sum of (forwardProfit[i] + backwardProfit[i]) which can also be done in O(n) time!
Actually, when we calculare profits backwards, we already have all forward profits, we can calculate the best total max profit at the same time and thus we don't need to store backward profits.

```
public int maxProfit(int[] prices) {
  int len = prices.length;
  if (len==0) return 0;
  int[] profitHist = new int[len];
  // forward calculate profit
  for (int buyDay=0, today=1; today<len; ++today) {
    int curP = prices[today] - prices[buyDay];
    if (curP <= 0) buyDay = today;
    else profitHist[today] = Math.max(curP, profitHist[today-1]);
  }
  // backward calculate profit
  int profit=profitHist[len-1], backProfit=0;
  for (int today=len-2, sellDay=len-1; today>0; --today) {
    int curP = prices[sellDay] - prices[today];
    if (curP <= 0) sellDay = today;
    else backProfit = Math.max(curP, backProfit);
    profit = Math.max(profit, profitHist[today]+backProfit);
  }
  return profit;
}
```

Another implementation

```
public int maxProfit(int[] prices) {
  int len = prices.length;
  if (len==0 || len==1) return 0;

  // calculate first max profit
  int[] ffProfits = new int[len];
  int low = prices[0]; ffProfits[0] = 0;
  for (int day=1; day<len; ++day) {
    if (prices[day] <= low) {
      low = prices[day];
      ffProfits[day] = ffProfits[day-1];
    } else {
      ffProfits[day] = Math.max(ffProfits[day-1], prices[day] - low);
    }
  }

  // calculate second max profit and total
  int totalProfit = ffProfits[len-1];
  int bbProfits = 0; int high = prices[len-1];
  for (int day=len-2; day>0; --day) {
    if (prices[day] >= high) {
      high = prices[day];
    } else {
      bbProfits = Math.max(bbProfits, high - prices[day]);
    }
    totalProfit = Math.max(totalProfit, bbProfits + ffProfits[day-1]);
  }

  return totalProfit;
}
```

This runs in time O(n) using O(n) spaces.

Posted 13th January 2013 by Sophie

Labels: Array, DP, Java

2 View comments

10th January 2013

Determine a Height-Balanced Binary Tree

Determine a (Height-) Balanced Binary Tree [http://leetcode.com/onlinejudge#question_110]

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of **every** node never differ by more than 1.

Solution

Since we need to check subtrees of each node, recursion looks like a good way to solve the problem.

Attempt 1: The basic logic of `isBalanced(root)` could be:

1. Find out the depths of left and right subtrees;
2. If it is not balanced (i.e. `abs(depLeft - depRight) > 1`), return `false`;
3. Otherwise, return `isBalanced(root.left) && isBalanced(root.right)`;

Of course, we need a base case so that recursion can terminate: return true if it is an empty tree (i.e. `root == null`).

This seems fine and it works. But, think about it more carefully: Starting from root, we trace down to its left and right subtrees to get the depth, do the calculation, then in the next recursion, we go to its left and right child, repeat the same process (and revisit all children nodes!). That is, we need to access each node multiple times. More specifically, suppose we have a full binary tree of height k,

- in top level, we visit each node once;
- in 2nd level, visit twice;
- ...
- in (k-1)-th level, visit (k-1) times;
- in k-th level, visit k time.

In total, we have T = k*(n/2), where n is the total number of nodes in the tree. We know k = log(n/2) = O(logn). So, this algorithm runs in time O(nlogn).

Can we do better? When we try to calculate the depth recursively, can we reuse any information without revisit?

Attempt 2: In the helper function, getDepth(root):

1. (base case) If it is an empty tree, return 0;
2. Find out the depth of left and right subtrees, recursively;
3. If it is not balanced or subtrees are not balanced, return a special value as a flag to pass back to upper levels;
4. Otherwise, return the depth of the tree;

Notice that the difference from attempt 1 is that this time we check the balance along the way to calculate the depth of the tree. So, we don't need to revisit the nodes.

Here is the code:

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *   int val;
 *   TreeNode left;
 *   TreeNode right;
 *   TreeNode(int x) { val = x; }
 * }
 */
  private int getHeight(TreeNode root) {
    if (root == null) return 0;
    int depL = getHeight(root.left);
    int depR = getHeight(root.right);
    if (depL < 0 || depR < 0 || Math.abs(depL - depR) > 1) return -1;
    else return Math.max(depL, depR) + 1;
  }
  public boolean isBalanced(TreeNode root) {
    return (getHeight(root) >= 0);
  }
```

Since we visit each node constant times, this algorithm runs in time O(n) and use O(n) space for recursion.

Posted 10th January 2013 by Sophie

Labels: Balanced-Binary-Tree, BinaryTree, Java, Recursion, Tree

1 View comments

8th January 2013

Find Anagrams

Find Anagrams [http://leetcode.com/onlinejudge#question_49]

Given an array of strings, return a list of all groups of strings that are anagrams.
Note: All inputs will be in lower-case.

For example, ["tea","and","ace","ad","eat","dan"] => ["and", "dan", "tea", "eat"]

Solution

Given a set of strings, if we could mark anagrams with a same "label", then a hash tabel can help us to find out all groups of anagrams.

So, how to mark anagrams?

Intuitively, 26 bits with '1' representing existence. We need check whether the length of the two strings are the same and do some bit-magics to generate the label. The bit-magic part is tedious and error prone, the comparison would be easy though. But it won't work for the case of "aae" and "aee" since we didn't track the occurrence during labeling.

As one bit is too short for occurrences, it looks like an integer for each character would do the job. But it will be a looong label even for a short word like "eat". We can trim it by skipping non-existing characters. E.g. "and" => "a1d1n1", "array" => "a2r2y1". Also notice that we need to keep character alphabetically in the label so as to ensure that anagrams are marked with a same label.

```
1:    public ArrayList<String> anagrams(String[] strs) {
2:      HashMap<String, ArrayList<String>> hash = new HashMap<String, ArrayList<String>>();
3:      for (String str : strs) {

4:        // create unique label for each string
5:        String key = generalLabel(str);
6:        // map the label to a list of anagrams
7:        ArrayList<String> res = hash.get(key);
8:        if (res==null) {
9:          res = new ArrayList<String>();
10:          hash.put(key, res);
11:        }
12:        res.add(str);
13:      }
14:      ArrayList<String> resSet = new ArrayList<String>();
15:      for (ArrayList<String> anagram : hash.values()) {
16:        // ignore strings without anagrams
17:        if (anagram.size()>1) resSet.addAll(anagram);
18:      }
19:      return resSet;
```

Analysis

This algorithm runs in time O(n) and uses O(n) space.

We need to general label for each string, so, that step (line 5 and 26-40) takes time O(kn) where k is the average length of strings and n is the number of strings. The mapping step takes time O(n), assuming the hash function is good (since our labels are designed to be unique for non-anagram strings, there is no collision in this case). In the last step, we go through the hash values, but each string will be touched once, and thus, the running time for the step is still O(n).

Posted 8th January 2013 by Sophie

Labels: Hash, Java, String

3 View comments

6th January 2013

Add Two (Binary) Numbers

The input of such a problem usually is a string or a list where each node contains one digit. One way to handle the carry is to reverse the string/list so that we can pipe the carry to next digit. Another way is to keep a pointer to the last zero and whenever need a pass a carry backwards, increase that digit by 1 and set digits in between to 0's.

The problem itself is not difficult. So the key thing here is to be carefully: check the end of string/list, move forward properly, calculate the carry bit correctly, etc.

Add Two Numbers in Linked List (Stored in Reverse Order) [http://leetcode.com/onlinejudge#question_2]

You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

For example: Input: (2 -> 4 -> 3) + (5 -> 6 -> 4) => Output: 7 -> 0 -> 8

Solution

Since the input lists are already in reverse order, we don't need to worry about carry here.

```
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
  if (l1==null) return l2;
  if (l2==null) return l1;
  int carry = 0;
  ListNode head=null, cur=null;
  while (l1!=null || l2!=null) {
    int sum = carry;
    if (l1!=null) {
      sum+=l1.val;
       l1 = l1.next;
     }
     if (l2!=null) {
       sum+=l2.val;
       l2 = l2.next;
     }
     if (cur==null) { // first node
       cur = new ListNode(sum%10);
       head = cur;
     } else {
       cur.next = new ListNode(sum%10);
       cur = cur.next;
     }
     carry = sum / 10;
   }// end-while
   if (carry>0) cur.next = new ListNode(carry);
   return head;
 }
```

Again, this algorithm runs in time O(n) and use O(n) space for the result list.

Add Two Numbers in Linked List [http://leetcode.com/onlinejudge#question_2]

This time you are given two linked lists representing two non-negative numbers. Each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

For example: Input: (3 -> 4 -> 2) + (4 -> 6 -> 5) => Output: 8 -> 0 -> 7

Solution

Now the digits are stored in plain order and thus we need to consider how to handle carry.

One way to do that is to use recursion:

- If this is the last digit, calculate the sum, create a new node with value = sum%10 and append it to previous node, return carry.
- If this is not the last digit, again, create a new node with value = sum%10 and append it to previous node, then recursively work on the remaining nodes. If a non-zero carry is returned, add it to the newly created node, then return carry.

```
private int length(ListNode l1) {
  int len = 0;
  while (l1 != null) {
    ++len;
    l1 = l1.next;
  }
  return len;
}

// The length of l1 >= the lenghth of l2 and n is the difference
private int addRecursive(ListNode l1, int n, ListNode l2, ListNode pre) {
  if (l1 == null && l2 == null) {
    return 0;
  }
  pre.next = new ListNode(0);
  int sum = l1.val;
  if (n <= 0) {
    sum += l2.val;
    sum += addRecursive(l1.next, 0, l2.next, pre.next);
  } else {
    sum += addRecursive(l1.next, n-1, l2, pre.next);
  }
  pre.val = sum % 10;
  return sum / 10;
}

// Digits in the two lists are in forward order.
// i.e. 345 = 3 -> 4 -> 5
private ListNode addTwoNumbersRecur(ListNode l1, ListNode l2) {
  int n1 = length(l1), n2 = length(l2);
  ListNode dummy = new ListNode(0);
  if (n1 >= n2) {
    dummy.val = addRecursive(l1, n1-n2, l2, dummy);
  } else {
    dummy.val = addRecursive(l2, n2-n1, l1, dummy);
  }
  return (dummy.val > 0) ? dummy : dummy.next;
}
```

This algorithm takes time O(n) and uses O(n) spaces (for the new list).

Can we do it iteratively?
Since each node contains a single digit, the largest two values could be 9+9 = 18. That said, carry could either be 0 or 1 and after passing a carry back to previous node once, the node itself has value <=8 and thus it will never pass any carry back no matter what digits are in remaining nodes.

So, if we keep track of the last less-than-9 node,

- Each time when there is non-zero carry, increase the value of last-less-then-9 node, move last towards current node (since the current node must have a value that is less than 9), and also set the digits between last and cur (exclusively) to 0 since their values were 9s.
- If there is no carry passing back, move last to current if current value is less than 9.

```
// Digits in the two lists are in forward order.
// i.e. 345 = 3 -> 4 -> 5
private ListNode addTwoNumbersIter(ListNode l1, ListNode l2) {
  ListNode dummy = new ListNode(0), last = dummy, cur = dummy;
  int n1 = length(l1), n2 = length(l2);
```

```
    if (n1 < n2) {
      ListNode tmp = l1; l1 = l2; l2 = tmp;
      int n = n1; n1 = n2; n2 = n;
    }

    while (n1 > n2) {
      cur.next = new ListNode(l1.val);
      cur = cur.next;
      if (cur.val < 9) last = cur;
      l1 = l1.next;
      --n1;
    }

    while (l1 != null) {
      int sum = l1.val + l2.val;
      l1 = l1.next;
      l2 = l2.next;
      cur.next = new ListNode(sum % 10);
      cur = cur.next;
      if (sum / 10 > 0) {
        while (last != cur) {
          last.val = (++last.val) % 10;
          last = last.next;
        }
      } else if (cur.val < 9) {
        last = cur;
      }
    }

    return (dummy.val > 0) ? dummy : dummy.next;
  }
```

This algorithm touch each node at most twice (one for addition and the other for updating carry). So the running time is O(n).

Add Two Binary Strings [http://leetcode.com/onlinejudge#question_67]

Given two binary strings, return their sum (also a binary string).

For example, a = "11", b = "1" => Return "100".

Solution

As we said above, we can reverse the strings first and also after calculation, we need to reverse the result and then return it.

Another thing is that (for Java) we should use `StringBuilder append()` method rather than `String "+" operator` to avoid cost. Note that what `"+" operator` does is:

1. create a new string;
2. make a copy of the left string;
3. append the right char/string to the end.

So, a series of "+" operation is O(n^2)-time, not O(n).
`StringBuilder` is smarter and you can think it as a dynamically resizing array of char. No copies for `append()` and it can output a `String` by calling `toString()` method.

Keep this in mind!

Back to the problem. To output the sum as a string, for each digit, we need to compute the carry bit and sum, and carry on the carry bit.

```
  public String addBinary(String a, String b) {
    StringBuilder aa = new StringBuilder(a);
    aa.reverse();
    StringBuilder bb = new StringBuilder(b);
    bb.reverse();
    /* 0 + 0 + 0/1 (cap) = 00/01
     * 1 + 1 + 0/1 = 10/11
     * 0 + 1 + 0/1 = 01/10
     */
    char carry = '0';
    StringBuilder res = new StringBuilder();
    int i=0;
    while (i<aa.length() && i<bb.length()) {
      if (aa.charAt(i) == bb.charAt(i)) { // 0+0 or 1+1
        res.append( carry );
        carry = aa.charAt(i);
      } else { // 0+1 or 1+0
        res.append( (carry=='0') ? '1' : '0' );
      }
      ++i;
    }// end-while-aa|bb
    /* 0 + 0/1 = 00/01, 1 + 0/1 = 01/10 */
    while (i<aa.length()) {
      if (aa.charAt(i) == '0') {
        res.append(carry);
        carry = '0';
      } else {
        res.append( (carry=='0') ? '1' : '0' );
      }
      ++i;
    }// end-while-aa
    while (i<bb.length()) {
      if (bb.charAt(i) == '0') {
        res.append(carry);
        carry = '0';
      } else {
        res.append( (carry=='0') ? '1' : '0' );
      }
      ++i;
    }// end-while-bb
    if (carry=='1') res.append(carry);
    return res.reverse().toString();
  }
```

Since we visit each letter in the two given strings three times (twice for reverse back and forth), this algorithm runs in time O(n) and take O(n) space (for the result :).

P.S.: If we are not allowed to use the build-in reverse() method, we need to make our own. It could be a in-place (take O(1) space) method by exchanging char's at first and last pointers and move the two pointers towards the middle; or it could be simply copy the char's into another string in reverse order and return the new string.

Alternatively, we keep a pointer to the last zero and flip all digits when a carry needs to be passed back. By doing this we visit each digit at most twice. So, the algorithm runs in time O(n) but slightly faster than the above one.

```
public String addBinary(String a, String b) {
  String longer, shorter;
  int n, m;
  if (a.length() > b.length()) {
    n = a.length(); m = b.length();
    longer = a; shorter = b;
  } else {
    n = b.length(); m = a.length();
    longer = b; shorter = a;
  }

  StringBuilder res = new StringBuilder();
  res.append('0'); // dummy head
  // find the last zero
  int lastZero = 0;
  for (int i=0; i<n-m; ++i) {
    res.append(longer.charAt(i));
    if (longer.charAt(i) == '0') lastZero = res.length() - 1;
  }
  // calculate
  for (int i=n-m, j=0; j<m; ++i, ++j) {
    int sum = (longer.charAt(i) == '1') ? 1 : 0;
    sum += (shorter.charAt(j) == '1') ? 1 : 0;
    res.append(sum & 1);
    int cur = res.length() - 1;
    if ((sum >> 1) > 0) {
      res.setCharAt(lastZero, '1');
      for (int k=lastZero+1; k<cur; ++k) {
        res.setCharAt(k, '0');
      }
    }
    if (res.charAt(cur) == '0') lastZero = cur;
  }

  // check head
  if (res.charAt(0) == '0') res.deleteCharAt(0);
  return res.toString();
}
```

Posted 6th January 2013 by Sophie

Labels: Java, LinkedList, Recursion, String, StringBuilder

4 View comments

4th January 2013

2Sum, 3Sum, 4Sum and Variances

2Sum [http://leetcode.com/onlinejudge#question_1]

Given an array of integers, find two numbers such that they add up to a specific target number.

The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.

You may assume that each input would have exactly one solution.

Input: numbers={2, 7, 11, 15}, target=9
Output: index1=1, index2=2

Solution

Intuitively, we can sort the array and use two pointers walking from two ends towards the middle of the array. But sorting takes O(nlogn) time.

An easy and efficient solution is to use a hash-table.

```
public int[] twoSum(int[] numbers, int target) {
  // a map that maps integer to its index
  HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
  int[] indexes = new int[2];
  for (int i=0; i<numbers.length; ++i) {
    int x = target - numbers[i];
    if (map.get(x) != null) {
      indexes[0] = map.get(x) + 1;
      indexes[1] = i + 1;
    } else {
      map.put(numbers[i], i);
    }
  }
  return indexes;
}
```

This solution runs in time O(n) but also require O(n) space.

3Sum [http://leetcode.com/onlinejudge#question_15]

Given an array S of n integers, are there elements a, b, c in S such that a + b + c = 0? Find all unique triplets in the array which gives the sum of zero.
Note:

- Elements in a triplet (a,b,c) must be in non-descending order. (ie, a ≤ b ≤ c)
- The solution set must not contain duplicate triplets.

For example, given array S = {-1 0 1 2 -1 -4}, A solution set is: (-1, 0, 1) (-1, -1, 2).

Solution

Hash table won't work here since it's not easy to mark a visited element and even if it works, it tends to give out a O(n^2) solution with O(n) space which is not the best option.

The problem gives us a hint: It asks for ordered triplets.
We can sort the array first, and then go through it to find b + c = -a in the rest of the array. To find b + c = -a, given a sorted array, we can have two pointers from the start and the end of the array. If start + end < -a, move start pointer forward; If start + end > -a, move end pointer backward; It equals, print out the result; Stop when start > end.

Here is the algorithm:

```
public ArrayList<ArrayList<Integer>> threeSum(int[] num) {
  ArrayList<ArrayList<Integer>> resSet = new ArrayList<ArrayList<Integer>>();
  if (num.length < 3) return resSet;
  Arrays.sort(num);
  for (int i=0; i<num.length-2 && num[i]<=0; ++i) {
    // remove duplicates
    if (i>0 && num[i]==num[i-1]) continue;
    // use two pointer to find b+c = -a
    int start=i+1, end=num.length-1;
     while (start<end) {
       int sum = num[i] + num[start] + num[end];
       if (sum < 0) {
         start++;
       } else if (sum > 0) {
         end--;
```

This solution runs in time O(n^2) and requires O(1) space.

3Sum Closest [http://leetcode.com/onlinejudge#question_16]

Given an array S of n integers, find three integers in S such that the sum is closest to a given number, target. Return the sum of the three integers. You may assume that each input would have exactly one solution.

For example, given array S = {-1 2 1 -4}, and target = 1. The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

Solution

Given a sorted list, how to find an element x of S such that x is the closest to -(a+b)? Yes, binary search. Here we go:

```
         int diff = sum - target;
         if ( Math.abs(diff) < Math.abs(res - target) ) {
           res = sum;
         }
         if (diff == 0) { // find the target
           return res;
         } else if (diff < 0) {
           start = mid + 1;
```

```
      } else {
        end = mid - 1;
      }
    }
  }
}
return res;
```

This solution runs in time O(n^2logn) and use O(1) space.

Better Solution

We can still use the same strategy for 3Sum here.

```
public int threeSumClosest(int[] num, int target) {
  if (num.length<3) { // if less than three items then return 0
    return Integer.MAX_VALUE;
  }
  Arrays.sort(num);
  int res = num[0]+num[1]+num[2];
  for (int i=0; i<num.length-2; ++i) {
    if (i>0 && num[i]==num[i-1]) continue;
    int start = i+1, end = num.length-1;
    while (start<end) {
      int sum = num[i] + num[start] + num[end];
      if (Math.abs(sum-target) < Math.abs(res-target)) {
        res = sum;
      }
      if (sum == target) {
        return res;
      } else if (sum < target) {
        start++;
      } else {
        end--;
      }
    }//end-while
  }
  return res;
}
```

This solution runs in time O(n^2) and requires O(1) space.

4Sum [http://leetcode.com/onlinejudge#question_18]

Given an array S of n integers, are there elements a, b, c, and d in S such that a + b + c + d = target? Find all unique quadruplets in the array which gives the sum of target.
Note:

- Elements in a quadruplet (a,b,c,d) must be in non-descending order. (ie, a ≤ b ≤ c ≤ d)
- The solution set must not contain duplicate quadruplets.

For example, given array S = {1 0 -1 0 -2 2}, and target = 0. A solution set is: (-1, 0, 0, 1) (-2, -1, 1, 2) (-2, 0, 0, 2).

Solution

Use the same strategy for 3Sum:

```
          ArrayList<Integer> res = new ArrayList<Integer>(4);
          res.add(num[i]);
          res.add(num[j]);
          res.add(num[ss]);
          res.add(num[ee]);
          resSet.add(res);
          // move forward and skip duplicates
          do { ss++; }while (ss<ee && num[ss]==num[ss-1]);
          do { ee--; }while (ss<ee && num[ee]==num[ee+1]);
        }
      }//end-while
    }
  }
  return resSet;
}
```

This solution runs in time O(n^3) and use O(1) space.

Posted 4th January 2013 by Sophie

Labels: Array, Hash, Java

2 View comments