Algorithms: Design and Analysis, Part 1 Review

1. Integer Multiplication

Input: 2 n-digit numbers x and y

Output: product x \* y

The Grade-School Algorithm:

Karatsuba Multiplication:

Pseudocode:

1. Recursively compute ac
2. Recursively compute bd
3. Recursively compute (a+b)(c+d)=ac+bc+ad+bc

Gauss’s Trick: (3)-(1)-(2)=ad+bc

1. Merge Sort

Input: array of n numbers, unsorted. (Assume Distinct)

Output: Same numbers, sorted in increasing order

Pseudocode:

1. recursively sort 1st half of the input array
2. recursively sort 2nd half of the input array
3. merge two sorted sublists into one

Pseudocode for Merge:

C = output [length = n]

A = 1st sorted array [n/2]

B = 2nd sorted array [n/2]

i = 1

j = 1

for k = 1 to n

if A(i) < B(j)

C(k) = A(i)

i++

else [B(j) < A(i)]

C(k) = B(j)

j++

end

Running time:

1. Big-Oh notation: if and only if there exist constants , such that

For all

Omege Notation: if and only if there exist constants , such that

Theta Notation: if and only if and

Equivalent: there exists constants , , such that

Little-Oh notation: if and only if for all constants , there exists a constants such that

1. The Closest Pair Problem

Input: a set of n points in the plane

Notation: Euclidean distance

So if and

Output: a pair of distinct points that minimize over , in the set

Brute-force search:

1-D Version of Closest Pair:

Pseudocode:

1. Sort points (O(nlog(n)) time)
2. Return closest pair of adjacent points (O(n) time)

Pseudocode of ClosestPair(Px,Py)

1. Let Q = left half of P, R = right half of P. Form Qx,Qy,Rx,Ry [O(n)]
2. (p1,q1) = ClosestPair(Qx,Qy)
3. (p2,q2) = ClosestPair(Rx,Ry)
4. Let = min{d(p1,q1), d(p2,q2)}
5. (p3,q3) = ClosestSplitPair(Px,Py,)
6. Return best of (p1,q1), (p2,q2), (p3,q3)

Pseudocode of ClosestSplitPair(Px,Py,)

Let = biggest x-coordinate in left of P. (O(1) time)

Let Sy = points of P with x-coordinate in

Sorted by y-coordinate (O(n) time)

Initialize best = , best pair = NULL

for i = 1 to |Sy|-7 (O(n) time)

for j = 1 to 7 (O(1) time)

Let p,q=ith, (i+j)th points of Sy

if d(p,q) < best

best pair = (p,q), best = d(p,q)

1. Counting Inversions

Input: array A containing the numbers 1,2,3,…,n in some arbitrary order

Output: number of inversions = numbers of pairs (i,j) of array indices with i < j and A[i] > A[j]

Brute-force: time

Pseudocode of Sort-andCount(array A, length n)

if n = 1, return 0

else

(B,X) = Sort-and-Count(1st half of A, n/2)

(C,Y) = Sort-and-Count(2nd half of A, n/2)

(D,Z) = CountSplitInv(A,n) (O(n) time just like Merge Sort)

return x+y+z

Claim the split inversions involving an element y of the 2nd array C are precisely the numbers left in 1st array B when y is copied to the output D.

Pseudocode of Merge\_and\_CountSplitInv

-- while merging the two sorted subarrays, keep running total of number of split inversions

-- when element of 2nd array C gets copied to output D, increment total by number of elements remaining in 1st array B

Run time of subroutine: O(n) + O(n) = O(n)

Sort\_and\_Count runs in O(nlog(n)) time

1. Matrix Multiplication

Write and

Then

Recursive Algorithm #1

Step 1: recursively compute the 8 necessary products.

Step 2: do the necessary additions ( time)

Runtime is

Strassen’s Algorithm

Step 1: recursively compute only 7 (cleverly chosen) products

Step 2: do the necessary (clever) additions + subtractions (still time)

The Seven Products:

Claim:

1. The Master Method
2. Quick sort

Pseudocode of QuickSort(array A, length n)

* if n = 1 return
* p = ChoosePivot(A,n)
* Partition A around p
* Recursively sort 1st part
* Recursively sort 2nd part

Pseudocode for Partition

Partition(A,l,r)

- p:= A[l]

- i:= l+1

- for j=l+1 to r

- if A[j] < p

- swap A[j] and A[i]

- i:= i+1

- swap A[l] and A[i-1]

How to choose pivots? BIG IDEA: RANDOM PIVOTS!

Average Running time:

1. Linear-Time Selection

Input: array A with n distinct numbers and a number

Output: ith order statistic (i.e., ith smallest element of input array)

Reduction to Sorting

1) Apply MergeSort 2) return ith element of sorted array

Randomized Selection

Rselect(array A, length n, order statistic i)

1. if n = 1 return A[1]
2. Choose a pivot p from A uniformly at random
3. Partition A around p

let j = new index of p

1. if j = i, return p
2. if j > i, return Rselect(1st part of A, j-1, i)
3. [if j < i] return Rselect(2nd part of A, n-j, i-j)

Best pivot: the median!

Deterministic ChoosePivot

ChoosePivot(A,n)

-- logically break A into n/5 groups of size 5 each

-- sort each group (e.g., using Merge Sort)

-- copy n/5 medians (i.e., middle element of each sorted group) into new array C

-- recursively compute median of C (!)

-- return this as pivot

The DSelect Algorithm

DSelect(array A, length n, order statistic i)

1. Break A into groups of 5, sort each group
2. C = the n/5 “middle elements”
3. p = Dselect(C, n/5, n/10) [recursively computes median of C]
4. Partition A around p
5. if j = i return p
6. if j < i return DSelect(1st part of A, j-1, i)
7. [else if j > i] return DSelect(2nd part of A, n-j, i-j)

Runtime O(n)

1. Comparison-based sorting lower bound

n! configurations of total possibilities of a array of n elements

assume we make k comparisons

altogether 2k resolution of the comparisons

By Pigeonhole Principle:

1. The Minimum Cut Problem

Input: An undirected graph G=(V,E)

[Parallel edges allowed]

Output: Compute a cut with fewest number of crossing edges. (a min cut)

Random Contraction Algorithm

While there are more than 2 vertices:

* pick a remaining edge (u,v) uniformly at random
* merge (or “contract”) u and v into a single vertex
* remove self-loops

return cut represented by final 2 vertices.

Running time: polynomial in n and m but slow ()

1. Graph Search algorithms

Generic Algorthm (given graph G, vertex s)

-- initially s explored, all other vertices unexplored

-- while possible:

-- choose an edge (u,v) with u explored and v unexplored

-- mark v explored

Running time: O(m+n) (linear time)

1. BFS(graph G, start vertex s)

[all nodes initially unexplored]

-- mark s as explored

-- let Q = queue data structure (FIFO), initialized with s

-- while :

-- remove the first node of Q, call it v

-- for each edge(v, w):

-- if w unexplored

-- mark w as explored

-- add w to Q (at the end)

Shortest Paths: compute dist(v), the fewest # of edges on path from s to v

Extra code: initialize

- When considering edge(v, w):

- if w unexplored, then set dist(w) = dist(v) + 1

Undirected Connectivity

-- initialize all nodes as unexplored O(n)

[assume labelled 1 to n]

-- for i = 1 to n O(n)

-- if i not yet explored O(n)

-- BFS(G, i)

Running time: O(m+n)

1. DFS

Run Time: O(m+n)

The Code: mimic BFS code, use a stack instead of a queue [ + some other minor modifications ]

Recursive version:

DFS(graph G, start vertex s)

-- mark s as explored

-- for every edge (s,v):

-- if v unexplored

-- DFS(G,v)

Topological Sort

-- let v be a sink vertex of G

-- set f(v) = n

-- recurse on G-{v}

Topological Sort via DFS

DFS-Loop(graph G)

-- mark all nodes unexplored

-- current-label = n [to keep track of ordering]

-- for each vertex

-- if v not yet explored [in previous DFS call]

-- DFS(G,v)

DFS(graph G, start vertex s)

-- for every edge (s,v)

-- if v not yet explored

-- mark v explored

-- DFS(G,v)

-- set f(s) = current\_label

-- current\_label = current\_label – 1

Running Time: O(m+n)

1. Strongly Connected Components

Kosaraju’s Two-Pass Algorithm

Run time: O(m+n)

Algorithm: (given directed graph G)

1. Let Grev = G with all arcs reversed
2. Run DFS-Loop on Grev

Let f(v) = “finishing time” of each v in V

1. Run DFS-Loop on G

processing nodes in decreasing order of finishing times

[SCCs = nodes in decreasing order of finishing times]

DFS-Loop

DFS-Loop(Graph G)

Global variable t = 0

[# of nodes processed so far]

Global variable s = NULL

[current source vertex]

Assume nodes labeled 1 to n

For i = n down to 1

if i not yet explored

s := i

DFS(G,i)

DFS(graph G, node i)

-- mark i as explored

-- set leader(i) := node s

-- for each arc(i, j) in G:

-- if j not yet explored

-- DFS(G, j)

-- t++

-- set f(i) := t

1. Dijkstra’s Algorithm

Single-Source Shorted Paths

Input: directed graph G=(V, E). (m=|E|, n=|V|)

* each edge has non negative length le
* source vertex s

Output: for each , compute

L(v) := length of a shortest s-v path in G

Dijkstra’s Algorithm

Initialize:

* X = [s] [vertices processed so far]
* A[s] = 0 [computed shortest path distances]
* B[s] = empty path [computed shortest paths]

Main Loop

* while xV: - need to grow x by one node
* among all edges with , pick the one that minimizes

[call it (v\*, w\*)]

* add w\* to X
* set
* set

Run time: naïve implementation

Using heap

To maintain Invariant #2: [i.e., that Key[v] = smallest Dijkstra greedy score of edge (u, v) with u in X]

When w extracted from heap(i.e., added to X)

* for each edge (w,v) in E:
* if v in V-X (i.e., in heap)
* delete v from heap
* recompute key[v] = min{key[v], A[w]+lwv}
* re-Insert v into heap

Running Time Analysis

# of heap operations in O(m+n)=O(m)

running time = O(m log(n)) (like sorting)

1. Heap: Supported Operation

Insert: Running time: O(log(n))

Extract-Min: Running time: O(log n)

Heapify (n batched Inserts in O(n) time)

Delete (O(log n) time)

Sorting: fast way to do repeated minimum computations.

Heap Sort: 1.) insert all n array elements into a heap

2.) Extract-Min to pluck out elements in sorted order

Running Time = 2n heap operations = O(nlog(n)) time

Median Maintenance

Input: a sequence x1,…,xn of numbers, one-by-one

Output: at each time step i, the median of {x1,…,xi}.

Solution: maintain heaps HLow: supports Extract Max

HHigh: supports Extract Min

Key Idea: maintain invariant that ~ i/2 smallest (largest) elements in HLow (HHigh)

The Heap Property: at every node x, Key[x] <= all keys of x’s children

Insert (given key k)

Step 1: stick k at end of last level

Step 2: Bubble-Up k until heap property is restored (i.e., key of k’s parent is <= k)

runtime = O(log(n))

Extract-Min

1. Delete root
2. Move last leaf to be new root
3. Iteratively Bubble-Down until heap property has been restored

[always swap with smaller child!]

run time = O(log(n))

1. Balanced Search Trees:

Supported Operations

OPERATIONS RUNNING TIME

SEARCH

SELECT

MIN/MAX

PRED/SUCC

RANK

OUTPUT IN SORTED ORDER

INSERT

DELETE

To Search for key k in tree T

-- start at the root

-- traverse left / right child pointers as needed

-- return node with key k or NULL, as appropriate

To Insert a new key into tree T

-- search for k (unsuccessfull)

-- rewire final NULL ptr to point to new node with key k

To compute the minimum (maximum) key of a tree

- Start at root

- Follow left child pointers (right ptrs, for maximum) until you cant anymore

(return last key found)

To compute the predecessor of key k

- Easy case: If k’s left subtree nonempty, return max key in left subtree

- Otherwise: follow parent pointers until you get to a key less than k.

TO PRINT OUT KEYS IN INCREASING ORDER

- Let r = root of search tree, with subtrees TL and TR

- recurse on TL

[by recursion (induction) prints out keys of TL in increasing order]

- Print out r’s key

- Recurse on TR

[prints out keys of TR in increasing order]

RUNNING TIME: O(1) time, n recursive calls => O(n) total

TO DELETE A KEY FROM A SEARCH TREE

- SEARCH for k

EASY CASE (k’s node has no children)

- Just delete k’s node from tree, done

MEDIUM CASE (k’s node has one child)

(unique child assumes position previously held by k’s node)

DIFFICULT CASE (k’s node has 2 children)

- Compute k’s predecessor l

[i.e., traverse k’s (non-NULL) left child ptr, then right child ptrs until no longer possible]

- SWAP k and l

NOTE: in it’s new position, k has no right child! => easy to delete or splice out k’s new node

HOW TO SELECT Ith ORDER STATISTIC FROM AUGMENTED SERACH TREE (with subtree sizes)

- start at root x, with children y and z

- let a = size(y) [a = 0 if x has no left child]

- if a = i – 1, return x’s key

- if a >= i, recursively compute ith order statistic of search tree rooted at y

- if a < i-1 recursively compute (i-a-1)th order statistic of search tree rooted at z

RUNNING TIME =

1. Red-Black Trees

Red-Black invariant

1. Each node red or black
2. Root is black
3. No 2 reds in a row

[red node => only black children]

1. Every root-NULL path has same number of black nodes

Height Guarantee: every red-black tree with n nodes has height

Rotations: locally rebalance subtrees at a node in O(1) time

Insert(x):

(1) insert x has usual (makes x a leaf)

(2) try coloring x red

(3) if x’s parent is black, done.

(4) else y is red => y has a black parent w

Case 1: the other child z of x’s grandparent w is also red.

=> recolor y,z black and w red

[key point: doew not break invariant 4]

=> eitherd restores invariant 3 or propagates

Case 2: let x,y be the current double-red, x the deeper node.

let w = x’s grandparent. Suppose w’s other child (y) is NULL or is a black node z.

can elimintae double-red [=> all invariants satisfied] in O(1) time via 2-3 rotations + recolorings.

1. Hash Tables

Supported Operations

Insert, Delete, Lookup

AMAZING GUARANTEE: All operations in O(1) time!\* 1. properly implemented 2. non-pathological data

De-Duplication

Input: a “stream” of objects

Output: remove duplicates (i.e., keep track of unique objects)

Solution: when new object x arrives

- lookup x in hash table H

- if not found, Insert x into H

The 2-SUM Problem

Input: unsorted array A of n integers. Target sum t.

Output: determine whether or not there are two numbers x,y in A with x + y = t

Naïve Solution: time via exhaustive search

Better: 1.) sort A ( time) 2.) for each x in A, look for t-x in A via binary search

Amazing: 1.) insert elements of A into hash table H 2.) for each x in A, Lookup t-x

High-Level Idea

Setup: universe U [generally, REALLY BIG]

Goal: want to maintain evolving set [generally, of reasonable size]

Naïve Solutions:

1. Array-based solution [indexed by u]

- O(1) operations but space

1. List-based solution

- space but Lookup

Solution: 1.) pick n = # of “buckets” with (for simplicity assume |S| doesn’t vary much)

2.) choose a hash function

3.) use array A of length n, store x in A[h(x)]

Resolving Collisions

Solution #1: (separate) chaining

- keep linked list in each bucket

- given a key/object x, perform Insert/Delete/Lookup in the list in A[h(x)]

Solution #2: open addressing. (only one object per bucket)

- Hash function now specifies probe sequences h1(x), h2(x),…

(keep trying till find open slot)

- Example: linear probing (look consecutively), double hashing

Quick-and-Dirty Hash Functions

How to choose n = # of buckets

1. Choose n to be a prime (within constant factor of # of objects in table)
2. Not too close to a power of 2
3. Not too close to a power of 10

Unavoidable Pathological Data Sets

Solutions:

1. Use a cryptographic hash function (e.g., SHA-2)

-- infeasible to reverse engineer a pathological data set

1. Use randomization

-- design a family H of hash functions such that for all data sets S, “almost all” functions spread S out “pretty evenly”.

1. Universal Hashing

A Universal Hash Function

Define:

1. Bloom Filters

Supported Operations: fast Inserts and Lookups.

Comparison to Hash Tables:

Pros: more space efficient

Cons:

1. can’t store an associated object
2. No deletions
3. Small false positive probability

Ingredients: 1) array of n bits ()

2) k hash functions h1,…, hk (k = small constant)

Insert(x): for i = 1,2,…,k (whether or not bit already set to 1)

set A[hi(x)] = 1

Lookup(x): return TRUE ⬄ A[hu(x)] = 1 for every i = 1,2,…,k