Algorithms Pseudocode Summary

Authors:

Dov Kruger, Ashish Garg

This document is a summary of all the algorithms covered in CPE-593. The intent here is to define in a single, clean document all these algorithms in as clean a way as possible.

This started as a project with Ashish Garg, and I have streamlined the notation, and cleaned up many of the algorithms. Still, this work is far from complete. All algorithms should use a uniform notation, there should be less c-like details such as declaration of data types. The focus here should be on algorithms. If you are collaborating on this project, the most important thing right now is to come up with a single clean representation (so let’s discuss) but also to keep writing up more of the algorithms. For algorithms such as quicksort where there are many variants, I would not mind writing up more of them. In other words, there does not have to be a single variant of an algorithm when there are multiple equally good ones, particularly if writing up a few teaches something.

# Number Theoretic Algorithms

## Factor(n)

Generate list of all factors of n

factors = {1,n}

for i← 2 to sqrt(n)

if n mod i = 0

factors.add(i)

factors.add(n/i)  
 end

end

return factors

## PrimeFactor(n)

Generate list of all factors of n which are prime

## Abundant(n)

Return true if the sum of all factors of n (not including n) > n

## 

## gcd(m,n)

Compute the greatest common denominator of two positive integers m,n

if n = 0  
 return m  
end  
return gcd(n, m mod n)

## iterative solution: gcd(m,n)

Compute the greatest common denominator of two positive integers m,n

while n ≠ 0  
 temp ← m  
 m ← n  
 n = m mod n  
end  
return m

## lcm(m,n)

return m \* n / gcd(m,n)

## Compute an

## Power(a,n)

prod ← 1

while n > 0

if n mod 2 = 1

prod←prod \* a

end

a←a\*a;

n←n/2;

end

return prod

## //Compute an mod m

## Powermod(a,n,m)

prod ← 1

while n > 0

if n mod 2 = 1

prod←prod \* a mod m

end

a←a\*a mod m;

n←n/2;

end

return prod

## 

## BruteForceIsPrime(n) //O(n)

for i ← 2 to n-1

if n mod i == 0

return false

end

end

return true

## BruteForceIsPrime(n) //O(√n)

for i ← 2 to sqrt(n)

if n mod i == 0

return false

end

end

return true

**EratosthenesSieve(n)**

isPrime ← new boolean[n]  
 for i ← 2 to n  
 isPrime[i] ← true  
 end

count ← 0  
 for i ← 2 to sqrt(n)  
 if isPrime[i]  
 count ← count + 1 // or print it out, whatever you want to do to each prime  
 for j ← 2\*i to n step i  
 isPrime[j] ← false  
 end  
 end  
 end  
  
  
**ImprovedEratosthenesSieve(n)**

isPrime ← new boolean[n]  
 for i ← 3 to n step 2  
 isPrime[i] ← true  
 end

count ← 1 // special case to account for the only even prime, 2  
 for i ← 3 to sqrt(n) step 2  
 if isPrime[i]  
 count ← count + 1 // or print it out, whatever you want to do to each prime  
 for j ← i\*i to n step 2\*i  
 isPrime[j] ← false  
 end  
 end  
 end

## Probabilistic Primality Tests

## FermatIsPrime(n, k)

for i ← 1 to k // k trials

a ← random(2, n-1)

if powermod(a, n-1, n) != 1

return false (notprime)

end

return true (probablyprime)  
end

## MillerRabinIsPrime(n,k)

Determine whether n is prime using Miller-Rabin probabilistic test. Unlike Fermat, this one is not fooled by Carmichael numbers.

for i ← 1 to k  
 a ← random(2, n-2) // Miller-Rabin needs a number at most n-2 LESS THAN Fermat!  
 d ← leading bits of n-1  
 s ← number of trailing 0 bits in n-1  
 if powermod(d, n-1, n) == 1  
 continue // this one worked, try all k to be fairly sure  
 end  
 do  
 d← d2 mod n  
 if d mod n == 1  
 return false (definitely not prime)  
 end  
 while d mod n != n-1  
end  
return true (probably prime)

## AKSIsPrime(n)

## 

## 

# Encryption Algorithms

## [e,d,n] = RSAgenkeys()

p← random prime

q← random prime

n← pq

do

e← random(2, n-2)

while e = p or e = q

d ← integer such that de mod = 1

## c ← RSAEncrypt(m, e, n)

c ← m.e mod n

## m ← RSADecrypt(c, d, n)

m = cd mod n

# 

# 

# Sorting

## Note: swap assumes passing a and b by reference

## Swap(ref a, ref b)

temp← a  
 a ← b  
 b ← temp

## It is possible to swap by using inverse operations. This one uses XOR. This is not important in data structures, but is a classic (though stupid) interview question.

## SwapWithoutTemp(ref a, ref b)

a ← a XOR b  
 b ← a XOR b  
 a ← a XOR b

## 

## 

## BubbleSort(A[], int n)

for j ← 0 to n-2

for( i ← 0 to n-2) {

if( A[i] < A[i + 1]) {

swap(A[i], A[i+1])

end

end

end

### Complexity

Worst Case: O(n2)

Best Case: O(n2)

Average Case: O(n2)

## BubbleSort(int A[], int n){

boolean swapped ← 0;

for j ← 0 to n-2 && swapped){

swapped ← true

for( i <- 0 to i <- n-2) {

if( A[i] < A[i + 1]) {

temp ←A[i];

A[i] ← A[i + 1];

A[i + 1] ←temp;

swapped ←1;

}

}

}

Complexity

Worst Case: O(n2)

Best Case: O(n)

Average Case: O(n2)

## SelectionSort (int A[] , int n) {

for i ← 0 to n-2 {

imin ← i; // set the initial minimum value as i

for j ← i+1 to n -1 {

if(A[j] < A[imin])

imin ← j ;

}

swap(A[i], A[imin])

}

}

Complexity

Worst Case: O(n2)

Best Case: O(n2)

Average Case: O(n2)

## InsertionSort ( int A[] , int n) {

for i ← 2 to n-1 {

v ← A [i] ;

j ← i;

while(A[j-1] > v j >= 0){

A[j] ← A[j+1];

j--;

}

A[j] ← v;

}

Complexity

Worst Case: O(n2)

Best Case: O(n)

Average Case: O(n2)

## quickSort (A[] , start, end) {

if (start < end) {

pindex ←partition (A[] , start, end);

quicksort (A[], start, pindex -1);

quicksort ( A[], pindex + 1, end);

}

## int partition ( int A[], start, end ) {

pivot ← A[random(start,end)];

pindex ← start;

//TODO: This is wrong! Fix...

for i ← start to end - 1 {

if(A[i] <= pivot) {

swap (A[i], A[pindex]);

pindex ← pindex + 1;

}

}

swap(A[pindex], A[end]);

}

Complexity

Worst Case: O(n2)

Best Case: O(nlog2n)

Average Case: O(nlog2n)

## 

## 

## HeapSort(A[], n){

BuildHeap(A[], n)

do

swap(A[0], A[n-1])

n ← n - 1

i ← 0

while i < n - 1

if A[(i+1)\*2-1] > A[(i+1)\*2] and A[i] < A[(i+1)\*2-1]

j ← 1

else if A[(i+1)\*2-1] <= A[(i+1)\*2] and A[i] < A[(i+1)\*2]

j ← 2

else

j ← 3

end

if j = 1

swap(A[i], A[(i+1)\*2-1]

i ←( i + 1) \* 2 - 1

else if j = 2

swap(A[i], A[(i+1)\*2])

i ←( i + 1) \* 2

else

break

end

end

while end > start

}

Complexity

Worst Case:O(nlog2n)

Best Case: O(nlog2n)

Average Case: O(nlog2n)

BuildHeap(A[], n){

for i ← n /2 -1 to start

if A[i] < A[(i+1)\*2-1]

swap(A[i], A[(i+1)\*2-1]

end

if A[i] < A[(i+1)\*2]

swap(A[i], A[(i+1)\*2])

end

end

}

## MergeSort(A[], B[], C[])

// assumes A and B are each ½ size of C

i← 0, j ← 0, k ← 0

if a[i] < b[j]

c[k++] ← a[i++]

if i > A.length

while j < B.length

c[k++] = B[j++]

end

break

end

else

c[k++] ← b[i++]

if j > B.length

while i < A.length

c[k++] = A[i++]

end

break

end

end

## RadixSort(A[], start, end)

# Shuffling

## randomSelectShuffle(sortedList)

Complexity: O(n2)

## randomlist ← FischerYates(sortedlist)

for i← n-1 to 1

r← random(0, i)  
 swap(sortedList[i], sortedList[r])

end

Complexity: O(n)

# 

# 

# 

# Searching

## int LinearSearch(A[ ], int n, int data ){

for i ← 0 to n-1 {

if(A[i] == data)

return i;

}

}

## int SortedLinearSearch(A[ ], target)

for i ← 0 to A.length-1

if A[i] >= target

if(A[i] > target)

return ***notfound***

return i;

end

end

end

Complexity

Worst Case: O(n)

Best Case: O(1)

Average Case: O(n)

## position ← binarySearch(A[ ], target )

low ← 0, high ← A.length -1

while low < high

mid ← (low + high) / 2

if A[mid] == target

return mid;

else if A[mid] < target

low ← mid + 1;

else

high ← mid – 1;

end

return ***notfound***

end

## position ← BinarySearchRecursive(A[ ], low, high, target)

mid ← (low + high) / 2

if(A[mid] == target)

return mid;

else if (A[mid] < target)

return BinarySearchRecursive(A, mid + 1, high, target);

else

return BinarySearchRecursive(A, low, mid – 1, target);

return ***notfound***

}

Complexity : O (log2n)

Worst Case: O(n)

Best Case: O(1)

Average Case: O(log2n)

## goldenMeanSearch(A[ ])

Data Structures

LinkedList (head only)

LinkedList (head and tail)

LinkedList (doubly linked list)

First, draw what the list looks like: Here is an example of an empty list and a list with one element

 

The add method adds a new value into the list. You are responsible to create the new node, not the person calling add.

Instead of this:

      public void addLast(SinglyLinkedListNode newNode) {

it should be

      public void addLast(int val) {

temp ← new node(val)

Do not check whether the new node is null. if temp is null, that is a problem with running out of memory. It is important in a real list class, but not in the algorithm theory of how to build a list.

Now that you have created the node, you have to find where it can be placed within your list.

Singly Linked List –

**Insertion with head and tail -**

public class SinglyLinkedList {

…

      public void addLast(SinglyLinkedListNode newNode) {

            if (newNode == null)

                  return;

            else {

                  newNode.next = null;

                  if (head == null) {

                        head = newNode;

                        tail = newNode;

                  } else {

                        tail.next = newNode;

                        tail = newNode;

                  }

            }

      }

      public void addFirst(SinglyLinkedListNode newNode) {

            if (newNode == null) Get rid of these tests

                  return;

            else {

                  if (head == null) { the logic here is good!

                        newNode.next = null;

                        head = newNode;

                        tail = newNode;

                  } else {

                        newNode.next = head;

                        head = newNode;

                  }

            }

      }

      public void insertAfter(SinglyLinkedListNode previous,

                  SinglyLinkedListNode newNode) {

            if (newNode == null)

                  return;

            else {

                  if (previous == null)

                        addFirst(newNode);

                  else if (previous == tail)

                        addLast(newNode);

                  else {

                        SinglyLinkedListNode next = previous.next;

                        previous.next = newNode;

                        newNode.next = next;

                  }

            }

      }

}

**Deletion with head and tail -**

public class SinglyLinkedList {

      …

      public void removeFirst() {

            if (head == null)

                  return;

            else {

                  if (head == tail) {

                        head = null;

                        tail = null;

                  } else {

                        head = head.next;

                  }

            }

      }

      public void removeLast() {

            if (tail == null)

                  return;

            else {

                  if (head == tail) {

                        head = null;

                        tail = null;

                  } else {

                        SinglyLinkedListNode previousToTail = head;

                        while (previousToTail.next != tail)

                             previousToTail = previousToTail.next;

                        tail = previousToTail;

                        tail.next = null;

                  }

            }

      }

      public void removeNext(SinglyLinkedListNode previous) {

            if (previous == null)

                  removeFirst();

            else if (previous.next == tail) {

                  tail = previous;

                  tail.next = null;

            } else if (previous == tail)

                  return;

            else {

                  previous.next = previous.next.next;

            }

      }

}

Linked list Array Dynamicarray

list

Indexing Θ(n) Θ(1) Θ(1)

Insert/delete at beginningΘ(1) N/A Θ(n)

Insert/delete at end Θ(n)

if last element is unknown

Θ(1)

if last element is known N/A Θ(1) amortized

Wasted space (average) Θ(n) 0 Θ(n)

# Hashing

**Chaining**

Please write the same method style. Do not suddenly go to a different style 3.1, 3.2, ...

hashmap.insert(key)

index ← hash(key) mod tablesize

1. Get the key k

2. Compute hash function h[k] = k % SIZE please use mod rather than % in the pseudocode

3. If hashtable[h[k]] is empty

(3.1) Insert key k at head of list in hashtable[h[k]]

(3.2) Stop

else

(3.3) Insert key at head of list while mainting the previous nodes.

4. Stop.

## LinearProbingInsert(key)

index ← hash(key) mod size

while table[index] ≠ key and table[index] ≠ empty

index ← (index + 1) mod size

end

table[index] ← key

end

## LinearProbingGet(key)

index ← hash(key) mod size

while table[index] ≠ empty

if table[index] = key

return value[index]

end

index ← (index + 1) mod size

end

return***null***

end

## QuadraticProbingInsert(key)

index ← hash(key) mod size

shift ← 1 or some other starting jump

while table[index] != key && table[index] != empty

index ← (index + shift) mod size

shift ← shift \* 2

end

table[index] ← key

end

**PerfectHash.insert(key)**

i← hash(key)

if table[i] = null

table[i] ← new Hash2;

end

do

table[i].insert(key)

while table[i].hasCollisions, table[i].resize() (and do again)

Example: insert 5, 13, 12, 19, 21 into table n=8, hash function is the same for both levels h(x) = x mod n

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | 19 | 12 | 5,13,21 |  |  |

Now we need to fix the collisions in bin 5. Bin 5 points to another table

Pick n2 to be relatively prime to n. n2 = 5.

| n=5 | 0 | 1 | 2 | 3 | 4 |
| --- | --- | --- | --- | --- | --- |
|  | 5 | 21 |  | 13 |  |

So in this case, every bin contains a second hash table with n=1, but one bin has n=5.

Example: perfect hash 10, 15, 2, 7, 23, into table n=8, hash function is the same for both levels h(x) = x mod n

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 10,2 |  |  |  |  | 15,7,23 |

try bin 2 with a hash table of 3 elements:

t1

| 0 | 1 | 2 |
| --- | --- | --- |
|  | 10 | 2 |

bin7 can have 5

t2

| n=5 | 0 | 1 | 2 | 3 | 4 |
| --- | --- | --- | --- | --- | --- |
|  | 15 |  | 7 | 23 |  |

So the top level hash table contains pointers to all the secondary tables. Bin 2 has a pointer to t1, and bin

# 

# 

# Matrix Algorithms

## Representation

Matrix: index(i,j) = i \* cols + j

TriangularMatrix: index(i,j) =   
 TridiagonalMatrix: index(i,j) = 3\*i + j - i + 1 = 2 \* i + j (neglect corner)

## C ← multiply(A,B)

## x← GaussJordanPartialPivot(A,B)

## x← GaussJordanFullPivot(A,B)

## x← NewtonIterativeImprovement(A,x, B)

## B← Power(A,n)

## C ← StrassenMultiply(A,B)

## B← GramSchmidt(A)

# 

# Graph Algorithms

Recursive Depth First Search starting with vertex v

## Graph.DFS(v, visited)

visited[v] ← true  
for Vnext← edges[v]

if not visited[v]  
 DFS(v, visited)

end

end

## Graph.DFS(v)

stack.push(v)

visited←-boolean(|V|)

while stack not empty

v ← stack.pop()

if visited[v]

continue

end

if (visited[v]label v as discovered

3 for all edges from v to w in G.adjacentEdges(v) do

4 if vertex w is not labeled as discovered then

5 recursively call DFS(G,w)

## G.BFS(v)

q← new Queue()

V← new Vector()

V.add(v)  
 create a vector set V  
 enqueue v onto Q  
 add v to V  
 while Q is not empty loop

7 t ← Q.dequeue()

8 if t is what we are looking for then

9 return t

10 end if

11 for all edges e in G.adjacentEdges(t) loop

12 u ← G.adjacentVertex(t,e)

13 if u is not in V then

14 add u to V

15 enqueue u onto Q

16 end if

17 end loop

18 end loop

19 return none

end

## 

## 

## BellmanFord(list vertices, list edges, vertex source)

::weight[],predecessor[]

// This implementation takes in a graph, represented as lists of vertices and edges,

// and fills two arrays (weight and predecessor) with shortest-path (less cost/weight/metric) information

// Step 1: initialize graph

for each vertex v in vertices:

if v is source then weight[v] := 0

else weight[v] := infinity

predecessor[v] := null

// Step 2: relax edges repeatedly

for i from 1 to size(vertices)-1:

for each edge (u, v) with weight w in edges:

if weight[u] + w < weight[v]:

weight[v] ← weight[u] + w

predecessor[v] ←u

// Step 3: check for negative-weight cycles

for each edge (u, v) with weight w in edges:

if weight[u] + w < weight[v]:

error "Graph contains a negative-weight cycle"

return weight[],predecessor[]

## 

## A\*Search(start, goal)

visited ← {}

toVisit ← {start}  
 cameFrom ← {}

gScore ← [∞, ∞, …]

fScore ← [∞, ∞, …]  
 gScore[start] ← 0

fScore[start] ← g\_score[start] + heuristic\_cost\_estimate(start, goal)  
 while toVisit not empty  
 current ← the node in openset having the lowest f\_score[] value  
 if current = goal  
 return reconstruct\_path(came\_from, goal)  
   
 remove current from openset  
 add current to closedset  
 for each neighbor in neighbor\_nodes(current)  
 if neighbor in closedset  
 continue  
 tentative\_g\_score := g\_score[current] + dist\_between(current,neighbor)  
   
 if neighbor not in openset or tentative\_g\_score < g\_score[neighbor]   
 came\_from[neighbor] := current  
 g\_score[neighbor] := tentative\_g\_score  
 f\_score[neighbor] := g\_score[neighbor] + heuristic\_cost\_estimate(neighbor, goal)  
 if neighbor not in openset  
 add neighbor to openset  
 return failure

from Wikipedia....

**function** A\*(start, goal)  
 *// The set of nodes already evaluated*  
 closedSet := *{}*  
  
 *// The set of currently discovered nodes that are not evaluated yet.*  
 *// Initially, only the start node is known.*  
 openSet := *{start}*  
  
 *// For each node, which node it can most efficiently be reached from.*  
 *// If a node can be reached from many nodes, cameFrom will eventually contain the*  
 *// most efficient previous step.*  
 cameFrom := an empty map  
  
 *// For each node, the cost of getting from the start node to that node.*  
 gScore := map **with** default value **of** Infinity  
  
 *// The cost of going from start to start is zero.*  
 gScore[start] := 0  
  
 *// For each node, the total cost of getting from the start node to the goal*  
 *// by passing by that node. That value is partly known, partly heuristic.*  
 fScore := map **with** default value **of** Infinity  
  
 *// For the first node, that value is completely heuristic.*  
 fScore[start] := heuristic\_cost\_estimate(start, goal)  
  
 **while** openSet **is** **not** empty  
 current := the node **in** openSet having the lowest fScore[] value  
 **if** current = goal  
 return reconstruct\_path(cameFrom, current)  
  
 openSet.Remove(current)  
 closedSet.Add(current)  
  
 **for** each neighbor **of** current  
 **if** neighbor **in** closedSet  
 **continue** *// Ignore the neighbor which is already evaluated.*  
  
 **if** neighbor **not** **in** openSet *// Discover a new node*  
 openSet.Add(neighbor)  
   
 *// The distance from start to a neighbor*  
 *//the "dist\_between" function may vary as per the solution requirements.*  
 tentative\_gScore := gScore[current] + dist\_between(current, neighbor)  
 **if** tentative\_gScore >= gScore[neighbor]  
 **continue** *// This is not a better path.*  
  
 *// This path is the best until now. Record it!*  
 cameFrom[neighbor] := current  
 gScore[neighbor] := tentative\_gScore  
 fScore[neighbor] := gScore[neighbor] + heuristic\_cost\_estimate(neighbor, goal)   
  
 return failure  
  
**function** reconstruct\_path(cameFrom, current)  
 total\_path := [current]  
 **while** current **in** cameFrom.Keys:  
 current := cameFrom[current]  
 total\_path.append(current)  
 return total\_path



# Numerical Methods

## Root Finding

## Bisection(f, a, b, eps)

if f(a) \* f(b) > 0

fail

do

x = (a + b) / 2

if f(x) > 0

while abs(b - a) < eps

## xn+1=NewtonRaphson(f, f’, x, eps)

## Ridders(f, f’, x, eps)

# 

# 

# Numerical Integration

## Trapezoidal(f, a, b)

sum ← h(½ f0 + ½ f1)

eps is proportional to h2

## Romberg(T1, T2)

sum ← (4T2 - T1) / 3

eps is proportional to h4

## Gauss2d(f, a, b)

sum ← h( f(x0) + f(x1))

eps is proportional to h4

## Gauss3d(f, a, b)

sum ← h( f(x0) + f(x1)+f(x2))

eps is proportional to h6

## AdaptiveQuadrature(f, a, b, eps)

mid ← (a + b)/2  
 sum ← integrationmethod(f, a, b)  
 leftsum ← AdaptiveQuadrature(f, a, mid, eps/2)

rightsum ← AdaptiveQuadrature(f, mid, b,eps/2)

if (abs(leftsum + rightsum - sum) < eps)

return leftsum + rightsum

[this is not expressed well!]

# double[] freq ← FastFourierTransform(double [] signal)

# 

# String

## pos← NaiveSearch(s, target)

**for i← 0 to s.length  
 if target[0] = s[i]**

**for j← 1 to target.length**

**if (target[j] != s[i]   
 next i  
 end  
end**

**return i**

**end**

**end**

**return notfound**

**end NaiveSearch**

## pos← KnuthMorrisPratt(s, target)

## pos← RegexSearch(s, target)

# 

# 

# 

# Finite State Machine

# 

# Compression

## out[] ← RunLengthEncoding(b[])

## out[] ← LempelZivWelch(b[])

## out[] ← BurrowsWheeler(b[])

Review Questions

Q: How  can  one  determine  whether  a  singly  linked  list  has  a  cycle?

Good answer:  Keep track of two pointers in the linked list, and start them at the

beginning of the linked list.  At each iteration of the algorithm, advance the first

pointer by one node and the second pointer by two nodes.  If the two pointers are

ever the same (other than at the beginning of the algorithm), then there is a cycle.  If

a pointer ever reaches the end of the linked list before the pointers are the same,

then there is no cycle.  Actually, the pointers need not move one and two nodes at a

time; it is only necessary that the pointers move at different rates.  This takes O(n)

time.

Q: Given a sorted array of integers, how can you find the location of a particular integer

x?

Good answer: Use binary search.  Compare the number in the middle of the array

with x.  If it is equal, we are done.  If the number is greater, we know to look in the

second half of the array.  If it is smaller, we know to look in the first half.  We can

repeat the search on the appropriate half of the array by comparing the middle

element of that array with x, once again narrowing our search by a factor of 2.  We

repeat this process until we find x.  This algorithm takes O(log n) time.

Not‐so‐good answer: Go through each number in order and compare it to x.  This

algorithm takes O(n) time.

How to find middle element of linked list in one pass?

In order to find length of linked list we need to first traverse through linkedlist till we find last node, which is pointing to null, and then in second pass we can find middle element by traversing only half of length. They get confused when interviewer ask him to do same job in one pass. In order to find middle element of linked list in one pass you need to maintain two pointer, one increment at each node while other increments after two nodes at a time, by having this arrangement, when first pointer reaches end, second pointer will point to middle element of linked list. See this trick to [find middle element of linked list in single pass](http://javarevisited.blogspot.com/2012/12/how-to-find-middle-element-of-linked-list-one-pass.html) for more details.

Which is faster: finding an item in a hashtable or in a sorted list?

Item retrieval is basically O(1) in a hash table, while O(log n) in a sorted list, so the hash table is faster on average.

What data structure should be used in order to…

find a range of values between m and n?