***1. Time complexity for recursion***

T(n) = T(n/2)+a -> O(logn)

T(n) = a+2T(n-1) -> O(2^n)

T(n) = n/2+T(n-2) = T(5m-b) -> O(n^2)

T(n) = n^4+2T(n/2) -> O(n^4)

T(n) = n^2+16T(n/4)=O(n^2logn)

T(n) = n^2+7T(n/2)=O(n^log7)

T(n) = sqrt(n)+2T(n/4)=O(sqrt(n)logn)

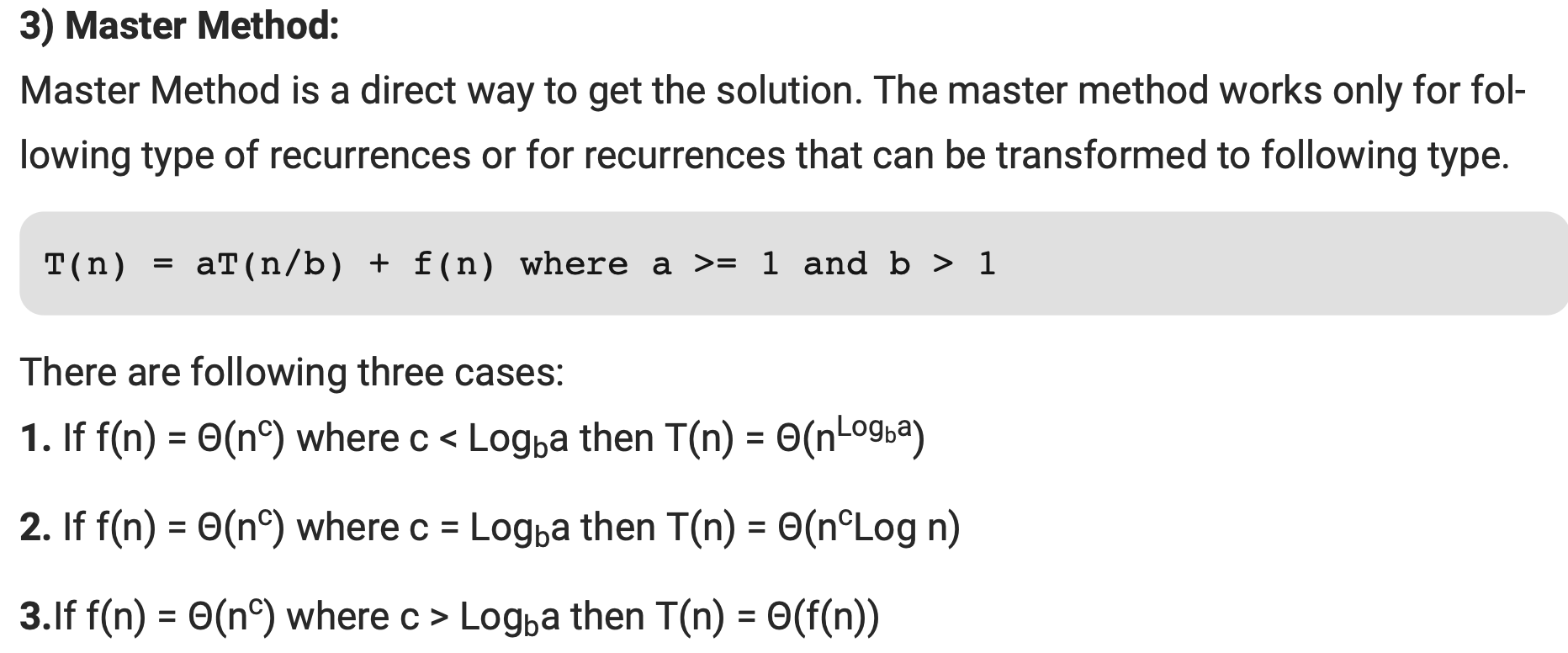
sum(i-logn)(i\*n/2^i) -> O(n)

T(n/3) <= d(n/3)log(n/3) <= dn(logn-log3)

T(n) = n+n/2+n/4+n/8+... <= O(n)

Divide-and-conquer -> O(nlogn)

Binary-search -> O(logn)

***2. Binary heap***

(1) Definition:

Parent(i): i/2

Left(i): 2i

Right(i): 2i+1

(2) Max-heapify(A, i) in O(logn)

l = left(i)

r = right(i)

if l <= heapsize && A[l] > A[i]

largest = l

else if r <= heapsize && A[r] > A[i]

largest = r

else

largest = i

if largest != i

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

Max-heapify(A,largest)

(3) Build-max-heap(A) in O(n)

for i = A.size/2 to 1

Max-heapify(A,i)

(4) Heap-sort(A) in O(nlogn)

Build-max-heap(A) --> O(nlogn)

// repeatedly swap, remove, and heapify

for i = A.size to 2 --> O(nlogn)

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

A.size--

Max-heapify(A,1)

(5) Implementing a max priority queue (all in O(logn))

Priority queue maintains the elements in the order of their keys

Heap-max(A)

return A[1]

Heap-extract-max(A) //remove the highest priority from queue

if size < 1 error()

max = A[1]

A[1] = A[A.size]

size--

Max-heapify(A,1)

return max

Heap-increase-key(A,i,key) // increase A[i] to key

if key < A[i] error()

A[i] = key

// compare and swap with smaller parents

while i > 1 and A[Parent(i)] < A[i]

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

i = Parent(i)

Heap-insert(A,key)

size++

A[size] = negINF

Heap-increase-key(A,size,key)

(6) Young tableaus

Each row and column is sorted

INF to represent non-existent element

Extract-min(A, i, j) in O(m\*n)

min = Y[i,j]

if A[i,j+1] = A[i+1, j] = INF

A[i,j]=INF // remove min

return min

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

return Extract-min(A, i+1, j)

Insert(key)

i=m, j=n

A[i,j] = key

while A[i-1,j] > A[i,j] or A[i,j-1] > A[i,j]

swap(A[i,j], min(A[i-1,j], A[i,j-1]))

decrement i or j, respectively

Sort(A)

for every key

insert(key)

for every element in A

sorted[i++] = Extract-min

Find(key,i,j)

if A[i,j] = key

return true

i = j = 1

while A[i,j] < key and i < m

Find(key,i,j)

i++

while i>1 and j<n

Find(key,i,j)

if A[i,j] < key

j++

else

i--

return false

***3. Dynamic programming***

- Identify optimal substructure

- Recursion

- Compute the value for each recursive call and save it to a lookup array (memorization)

- Construct optimal solution from computed information

**(1) Rod-cutting**

RECURSION:

cr-recur(p,n)

if n = 0

return 0

q = -∞

for 1 to n

q = max(q, p[i]+cr-recur(p,n-i))

return q

MEMOIZATION:

cut-rod-aux(p,n,r)

if r[n] >= 0 // solve this subproblem before

return r[n]

if n = 0

q = 0

else

q = -∞

for i = 1 to n

q = max(q, p[i]+cut-rod-aux(p,n-i,r))

r[n] = q // save the result to the look-up array

return q

Memoized-cut-rod(p,n)

r[0 to n] = -∞

return cut-rod-aux(p,n,r)

BOTTOM\_UP: (tabluation)

cr(p,n)

r[0] = 0

for j = 1 to n

q = -∞

for i = 1 to j

q = max(q, p[i]+r[j-i])

r[j] = q

return r[n]

cr2(p,n)

initialize r[0 to n] and s[0 to n]

r[0] = 0

for j = 1 to n

q = -∞

for i = 1 to j

if q < p[i]+r[j-i]

q = p[i]+r[j-i]

s[j] = i // store the first piece to be cut off for a subproblem of size j

r[j] = q

return r and s

print(p,n)

(r,s) = cr2(p,n)

while n>0

print s[n]

n -= s[n]

**(2) Matrix-chain multiplication**

BOTTOM UP O(n^3)

Matrix-chain-order(p) // p is the input sequence

n = p.length-1

initialize m[n,n] to all zeros and s[1 to n-1, 2 to n]

for l = 2 to n: // l is the chain length

for i = 1 to n-l+1

j = i+l-1

m[i,j] = ∞

for k = i to j-1

q = m[i,k]+m[k+1,j]+p\_i-1\*p\_k\*p\_j

MEMOIZATION

mem-matrix-chain(p)

n = p.length-1

intialize m[n,n] to ∞

return lookup-chain(m, p, 1, n)

lookup-chain(m,p,i,j)

if m[i,j] < ∞ return m[i,j]

if i == j

m[i,j] = 0

else

for k = i to j-1

q = lookup-chain(m,p,i,k)

+lookup-chain(m,p,k+1,j)

+p\_i-1\*p\_k\*p\_j

m[i,j] = min(m[i,j], q) // save result

return m[i,j]

**(3) Longest common sequence**

LCS(x,y)

m = len(x)

n = len(y)

b[1~m,1~n]=0, c[0~m,0~n]=0

for i 1 to m

for j 1 to n

if xi = yj

c[i,j]=c[i-1,j-1]+1

else

c[i,j]=max(c[i-1,j],c[i,j-1])

return c

MEMOIZATION

LCS(x,y,c,b)

m = len(x)

n = len(y)

if c[m,n] != 0 or m=0 or n=0 return

if xm = yn

b[m,n] = found

c[m,n] = LCS(x-last, y-last, c,b)+1

else

c[m,n] = max(LCS(x-last, y, c,b), LCS(x, y-last, c,b))

b[m,n] = found up/left

**(4) Longest increasing sequence**

FINDMAXSEQLEN(A[1..n], start, end)

endingSeqMaxLen = [1, 1, ..., 1]

for i ← 1...n

for j ← 1...i-1

max = 0

if A[j]<A[i] and max < endingSeqMaxLen[j]

max = endingSeqMaxLen[j]

endingSeqMaxLen[i] = max + 1

return max(endingSeqMaxLen)

LAS(A[1..n])

tails ← [0, 0, ..., 0] // length: n+1

tails[1] ← A[1]

l ← 1 // max subsequence length

for i ← 2...n

if A[i] < tails[1]

tails[1] ← A[i]// update the smallest value

elif A[i] > tails[n]

l += 1

tails[l] = A[i]

else

tails[BINARYSEARCH(tails, l, A[i])] ← A[i]

BINARYSEARCH(B[1..n + 1], r, v)

// Find the index of the ceil of the v by binary search

l ← 0 // search in B[l...r]

while r > l

m = l + (r − l)/2

if A[m] ≥ v

r←m

else

l←m

return r

**(5) Optimal BST**

OPT\_BST(p,q,n) ---O(n^3)

e[1~n+1, 0~n]

w[1~n+1, 0~n]

root[1~n,1~n]

for i = 1 to n+1

e[i,i-1] = q\_i-1

w[i,i-1] = q\_i-1

for l = 1 to n

for i = 1 to n-1+1

j = i+l-1

e[i,j] = ∞

w[i,j] = w[i,j-1]+pj+qj

for r = i to j

t = e[i,r-1]+e[r+1,j]+w[i,j]

if t < e[i,j]

e[i,j] = t

root[i,j] = r

return e and root

**(6) Longest path in DAG**

longest(G,s,t) = 1 + max{longest(G-s, s', t)}

**(7) Longest Palindrome**

def lps(seq, i, j):

if (i == j):

return 1

# Base Case 2: If there are only 2

# characters and both are same

if (seq[i] == seq[j] and i + 1 == j):

return 2

# If the first and last characters match

if (seq[i] == seq[j]):

return lps(seq, i + 1, j - 1) + 2

# If the first and last characters

# do not match

return max(lps(seq, i, j - 1),

lps(seq, i + 1, j))

**(8) Bitonic tour**

Algorithm:

1) Let 1 be the starting and ending point for salesman.

2) Construct MST from with 1 as root using Prim’s Algorithm.

3) List vertices visited in preorder walk of the constructed MST and add 1 at the end.

**(9) Printing neatly**

dp\_printing(words[]):

print[n]

cost[n,n]

/\* Assign costs for each i, j \*/

for i <-- 1 to n:

for j <-- i to n:

if j-i+∑\_(k=i)^n▒l\_k >M:

cost[i,j] <-- ∞

else

cost[i,j] <-- [M-(j-i+∑\_(k=i)^n▒l\_k )]^3

/\* Find the optimal cost \*/

for i <-- n to 1:

minCost[i] <-- cost[i,n]

print[i] <-- words[n]

for j <-- n to i:

minCost[i] <-- min(minCost[i], minCost[j]+cost[i,j])

if minCost[i] is changed:

print[i] <-- j

/\* Store words line by line \*/

lastword <-- print[1]

j <-- 1

for i <-- 1 to n:

if print[i] != lastword:

j < j+1

lastword <-- print[i]

append words[i] to line[j]

if length(line[j]) < M:

append space to line[j]

**(10) Edit distance**

Edit(x,y,i,j)

m = len(x)

n = len(y)

if i=m

return (n-j)cost(insert)

if j=n

return min{(m-i)cost(delete), cost(kill)}

initialize o1~o5 to ∞

if x[i] = y[j]

o1 = cost(copy)+edit(x,y,i+1,j+1)

o2 = cost(replace)+edit(x,y,i+1,j+1)

o3 = cost(delete)+edit(x,y,i+1,j)

o4 = cost(insert)+edit(x,y,i,j+1)

if i < m-1 and j < n-1

if x[i]=y[j+1] and x[i+1]=y[j]

o5=cost(twiddle)+edit(x,y,i+2,j+2)

return min(o1~o5)

**(11) Planning**

Find-Max-Conv(Tree t)

Let MC[ ] be an array of length n that contains max conviviality from this node down in the tree

for i = Node n downto 1

MC[i] = max(i.rating + Sum of all MC[i.grandchildren], Sum of all MC[i.children])

(If node i has no grandchildren or children, replace i.grandchildren and/or i.children with 0)

return MC[1]

**(12) Viterbi's algorithm**

Given a directed graph, each edge is labeled with a sound. Each path starts from a distinguished version v corresponds to a possible seuqnece of k sounds.

Given v0 and sequence of sound, returns the path in G.

Viterbi(G,s,v0) O(k^2|V|)

if len(s) = 0

return v0

for outgoing edges of v0:

for alls edge = s[1]

curr = max(prob[v1], curr)

res = viterbi(G, s-s[1], curr)

if res != no-such-path

return v0, res

return no-such-path

**(13) Image compression by seam carving**

Compress image with lowest disruption measure. Removing pixels in adjacent rows be in the same or adjacent columns.

Seam(A) ----O(m^3logn)

Intialize D[m,n] and S[m,n]

for 1 to n

D[1,i] = d1i

S[1,i] = (1,i)

for i = 2 to m

for j = 1 to n

if j == 1 //left edge

if D[i-1,j] < D[i-1,j+1]

D[i,j] = D[i-1,j]+dij // find the min disruption

S[i,j] = S[i-1,j].insert(i,j) // pixel to be removed

else

D[i,j] = D[i-1,j+1]

S[i,j] = S[i-1,j+1].insert(i,j)

else if j == n //right edge

if D[i-1,j-1] < D[i-1,j]

D[i,j] = D[i-1,j-1]

S[i,j] = S[i-1,j-1].insert(i,j)

else

D[i,j] = D[i-1,j]

S[i,j] = S[i-1,j].insert(i,j)

x <-- min(D[i-1,j-1], D[i-1,j], D[i-1,j+1])

D[i,j] = D[i-1,j+x]

S[i,j] = S[i-1,j+x].insert(i,j)

q=1

for j 1 to n

if D[m,j] < D[m,q]

q = j

Print the list S[m,q]

**(14) Break a string**

Cut(L,i,j,l,r)

if l=r

return 0,[]

if cut\_array[i][k][l][r] != null

return cut\_array[i][k][l][r]

mincost = ∞

for k from i to j

curr <-- l+r+cut(L,i,k,l,L[k]).cost+cut(L,i,j,L[k],j)

if curr < mincost

mincost <-- curr

minseq <-- L[k]+cut(L,i,k,l,L[k])+cut(L,i,j,L[k],j) // can do memoization

cut[i][j][l][r] = (mincost, minseq)

return mincost, minseq

**(15) Investment strategy**

Invest(d,n)

inv\_type[11] <-- 0

revenue[11] <-- 0

for k = 10 to 1

q = 1

for i = 1 to n

if rik > rqk

q = i // best inv for a given year

if R[k+1] + drI[k+1]k - f1 > R[k+1]+drqk-f2 // better is money is not moved

R[k] = R[k+1] + drI[k+1]k - f1 like the last year

I[k] = I[k+1]

else

R[k] = R[k+1]+drqk-f2

I[k] = q

return I and R[1]

**(16) Sign a player**

Baseball(N,X,P)

B[N+1,X+1] -> VORP of given number of players and cost

P[N] -> player at each position

for i : 1 to N

for j: 1 to X

if j < i.cost

B[i,j] = B[i-1,j]

q = B[i-1,j]

p = 0

for k = 1 to p

if B[i-1,j-i.cost] + i.value > q

q = B[i-1, j-i.cost]+i.value

p = k

B[i,j] = q

P[i] = p

**(17) Knapsack**

fractional Knapsack - greedy

int median\_ratio\_helper(ratios[n], int k)

l[] <-- 0

g[] <-- 0

p[] <-- 0

pivot <-- ratios[k] //arbitrarily choose a pivot

for i <-- 1 to n

if ratios[i] < pivot

add to l[]

else if ratios[i] = pivot

add to p[]

else

add to g[]

if l.length = g.length

return pivot

if l.length >= k

median\_ratio(l, k)

else

median\_ratio(g, k-l.length-p.length)

int median\_ratio(items[n], k)

for i <-- 1 to n:

ratios[i] <-- items[i].value/items[i].weight

return median\_ratio\_helper(ratios[n], k)

int f\_knapsack(items[], n, W)

if n = 0 or W = 0

return 0

if n = 1 and items[n].weight >= W

return items[n].weight/W\*items[n].value

median <-- median\_ratio(items, n/2)

w <-- 0

greater[] <-- 0

less[] <-- 0

for i <-- 1 to n:

if ratios[i] > median

add items[i] to greater[]

w <-- w+items[i].weight

else

add items[i] to less[]

if w < W

k <-- k + f\_knapsack(less[], less.length, W-w)

else

k <-- k + f\_knapsack(greater[], greater.length, W)

return k

0/1 knapsack w/ memoization:

int knapsack(W,n,wt[],val[])

int K[n+1][W+1]

for i: 0 to n

for w: 0 to W

if i=0 || w=0

K[i][w] = 0

else if wt[i-1] <= w // the last item can fit in knapsack

K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])

else // can't fit, keep the same result from the previous

K[i][w] = K[i-1][w];

return K[n][w]

**(17) Minpartition**

minPalPartion(str, i, j) = 0 if i == j.

minPalPartion(str, i, j) = 0 if str[i..j] is palindrome.

else

// calculated recursively using the following formula.

minPalPartion(str, i, j) =

min { minPalPartion(str, i, k) + 1 +

minPalPartion(str, k+1, j) }

where k varies from i to j-1

***4. Greedy Algorithm***

Greedy Problems Guideline

● Determine whether optimal substructure exists (We can solve for a smaller subset of S).

● Develop a recursive solution (Take one of the front intervals, solve for the Sj).

● Show that if we make a greedy choice, only 1 subproblem remains(Si is our subproblem), instead of two more. The choice is finite.

and that it’s always safe to make the greedy choice (monotonicity is your friend).

● Use the greedy solution, and make it iterative for brownie points.

**(1) Coin change O(nC)**

int[] coin\_change(C, n)

ncoins[C] <-- ∞

coin\_type[C] <-- ∞

for value from 1 to C:

curr\_coin <-- null

curr\_n <-- ∞

for coin in n:

if ncoins[value-coin]+1 < curr\_n:

curr\_n = ncoins[value-coin]

curr\_coin = coin

ncoins[value] <-- curr\_n

coin\_type[value] <-- curr\_coin

solution[ncoins[C]] <-- 0

value <-- 0

while value < C:

add coins[value] to solution[]

value <-- value + coins[value]

return solution[]

**(2) Scheduling**

1) Sort all jobs in decreasing order of profit/completion time

2) Initialize the result sequence as first job in sorted jobs.

3) Do following for remaining n-1 jobs

a) If the current job can fit in the current result sequence

without missing the deadline, add current job to the result.

b) Else ignore the current job.

Select the max cardinality subset of jobs S, S' such that the jobs do not overlap, or f\_i < s\_j. Finish time of the this job is earlier than the start time of the next job. Greedily pock the earliest end time intervals that do not overlap

Monotonicity: S\_i is always better S\_j for i < j

Proof of monotonicity

Induction: |S|=1 base case. Suppose we have a solution E. If in another better solution of theta, we do not choose the earliest end time interval, we will miss jobs that start between the optimal interval end time and the chosen interval.

void printJobScheduling(Job arr[], int n)

sort(arr, arr+n, comparison);

int result[n]

bool slot[n] <-- false

for every job

searching from the end of slots

if (slot[j]==false)

result[j] = i;

slot[j] = true;

break

**(5) Huffman coding**

Huffman(C)

n = len(C)

Queue = C

for i = 1 to n-1

node.left = x = extract-min(Q)

node.right = y = extract-min(Q)

node.freq = x.freq+y.freq

insert(Q, node)

return extract-min(Q) //root of the tree

HUFFMAN( f [1..n])

Qf ←QUEUE(f) //construct a queue from f

Qinternal ← QUEUE() //construct an empty queue

while there are two or more files in Qf and Qinternal

take two files a and b with the smallest frequency from Qf and Qinternal merge them into an internal file ab with f [ab] = f [a] + f [b]

construct the tree with a, b as the child and a b as the parent.

enqueue ab in Qinternal

return the last node in Qinternal as the rooted optimial binary tree

***5. Divide and conquer***

**(1) Merge sort (counting significant inversions)**

int merge(A, l, m, r)

i <-- 0

j <-- m+1

k <-- 0

arr[l-r] <-- 0

while i <= m and j < r

if A[i] <= A[j]

arr[k++] <-- A[i++]

else

arr[k++] <-- A[j++]

if A[i] > 2\*A[j]

count <-- count + (m – i)

while i <= m

arr[k++] <-- A[i++]

while j < r

arr[k++] <-- A[j++]

A <-- arr

return count

int merge\_sort(A, l, r)

if l < r

return 0

else

count <-- 0

mid <-- (l+r)/2

count <-- count

+ merge\_sort(A, l, m)

+ merge\_sort(A, m+1, r)

+ merge(l, m+1, r)

return count

**(2) Median finding algorithm O(n)**

int median\_ratio\_helper(ratios[n], int k)

l[] <-- 0

g[] <-- 0

p[] <-- 0

pivot <-- ratios[k] //arbitrarily choose a pivot

for i <-- 1 to n

if ratios[i] < pivot

add to l[]

else if ratios[i] = pivot

add to p[]

else

add to g[]

if l.length = g.length

return pivot

if l.length >= k

median\_ratio(l, k)

else

median\_ratio(g, k-l.length-p.length)

**(3) Closest pair of points O(n(logn)^2)**

sort by x-coordinates

split at the midpoint

d = min(min\_left, min\_right)

find min\_cross:

sort by y coordinate

check 7 squares around each node

return min dist

d = min(d, min\_cross)

float stripClosest(Point strip[], int size, float d)

float min = d;

qsort(strip, size, sizeof(Point), compareY);

// Pick all points one by one and try the next points till the difference

// between y coordinates is smaller than d.

// This is a proven fact that this loop runs at most 6 times

for (int i = 0; i < size; ++i)

for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)

if (dist(strip[i],strip[j]) < min)

min = dist(strip[i], strip[j]);

return min;

// A recursive function to find the smallest distance. The array P contains

// all points sorted according to x coordinate

float closestUtil(Point P[], int n)

if (n <= 3)

return bruteForce(P, n);

// Find the middle point

int mid = n/2;

Point midPoint = P[mid];

float dl = closestUtil(P, mid);

float dr = closestUtil(P + mid, n-mid);

float d = min(dl, dr);

// Build an array strip[] that contains points close (closer than d)

// to the line passing through the middle point

Point strip[n];

int j = 0;

for (int i = 0; i < n; i++)

if (abs(P[i].x - midPoint.x) < d)

strip[j] = P[i], j++;

// Find the closest points in strip. Return the minimum of d and closest

// distance is strip[]

return min(d, stripClosest(strip, j, d) );

**(4) Largest two elements n+logn**

int[] find\_max(A, start\_i, end\_i):

if start\_i = end\_i

candidates[0...n]

candidates[0] <-- 1

candidates[1] <-- A[start\_i]

candidates\_1[] <-- find\_max(A, 1, n/2-1)

candidates\_2[] <-- find\_max(A, n/2, n)

if candidates\_1[1] > candidates\_2[1]

candidates\_1[0] <-- candidates\_1[0] + 1

candidates\_1[candidates\_1[0]] <-- candidates\_2[1]

return candidates\_1[]

else

candidates\_2[0] <-- candidates\_2[0] + 1

candidates\_2[candidates\_2[0]] <-- candidates\_1[1]

return candidates\_2[]

int find\_second\_max(A)

cand <-- find\_max(A, 1, n)

second\_max <-- find\_max(cand+2, 2, cand[0])

return second\_max[1]

**(5) Majority (On previous midterm)**

Find if one key appear more than N/2 times in O(nlogn)

- Divide: into halves

- Conquer: In the right side: a key more than n/4 times. In the left side: a key more than n/4. Check if they are the same key -> return key

if not, check left key and check in right

check right key and check in left -> O(n)

***6. Topological sorting O(E)***

A DAG, if A has an edge to B, A precedes B in the ordering

If G has a topo ordering, then G is a DAG

We can prove by contradiction. If G has a topo ordering which has a cycle, let vi be the lowest node. j<i if there is an edge from i to j. But vi is the lowest node.

In every DAG, there is a node with no incoming edges

Proof by contradiction: Let G be a DAG where every node has at least one incoming edge. Pick any node v, and begin following edges backward from v: since v has at least one incoming edge we can always follow an edge backwards to some node u, and so on.We can do this indefinitely, since every node has an incoming edge. After doing this n + 1 times, by the Pigeonhole Principle we have visited some node w twice. We can then let C denote the nodes visited between visits of w, which is a cycle, a contradiction.

If a graph is DAG, then it has a topo ordering

Claim by induction that every DAG has a topological ordering. This is true for base cases of DAGS with one or two nodes. Now consider that it is true for DAGs with n nodes. Given a DAG G with n + 1 nodes, we can use Observation 2 to find a node v with no incoming edges, and place it first in our topological ordering since all of its edges point forward. G - {v} is a DAG, since deleting v can’t create any cycles. G - {v} has n nodes, so we can apply induction to get its topological ordering. We append that to v. This is an ordering of G where all nodes point forward, so it is indeed a topological ordering. Whew!

TS(v, visited, stack)

visited[v] = True

for each outgoing neighbor w for node v: --O(|E|)

if visited[w] == False:

TS(w, visited, stack)

stack.push(v)

TopologicalSort()

for i in 0 to n-1:

visited[i] = False

for i in 0 to n-1: -- O(|V|)

if visited[i] == False:

TS(i, visited, stack)

while stack is not empty:

print stack.pop()

***7. Finding minimal spanning tree***

**(1) Prim: start from a node, look for adjacent** edges with lightest weight(dijkstra)

visited array

start from a source node

while still unvisited nodes:

for every unvisited nodes adjacent to the current visited node

choose min edge

- a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a Spanning Tree. And they must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

for every node

// Pick the minimum key vertex from the

// set of vertices not yet included in MST

int u = minKey(key, visited);

visited[u] = true;

for adjacent vertices of u

// graph[u][v] is non zero only for adjacent vertices of m

// mstSet[v] is false for vertices not yet included in MST

// Update the key only if graph[u][v] is smaller than key[v]

if unvisited < than the current key

parent[v] = u, key[v] = graph[u][v];

**(2) Kruskal: continue look for lightest edge that** do not form a cycle

class UNIONQUERY

initialize(n)

// Every set has only one element at the beginning. Each node points to itself.

parent ← [1, 2, ..., n]

size ← [1, 1, ..., 1]

root(i) // find the root for ei

if parent[i] == i

return i

return root(parent[i])

Union(x, y)

i ← root(x)

j ← root(y)

if i ̸= j

if size[i] ≤ size[j]

parent[i] ← j // let the smaller tree root point to the larger tree root size[j] += size[i]

else

parent[j] ← i

size[i] += size[j]

Query(x, y) return root(x) == root(y)

KRUSKAL(G = (V, E))

MST ← []

E ← sort(E)

UQ ← UNIONQUERY(|V |)

for e = (u, v) in E

if UQ.Query(u, v) == FALSE

UQ.Union(u, v)

add e to MST

return MST

**(3) Distributed kruskal**

DISTRIBUTEDKRUSKA(E = (i, j))

for round ← 1...n − 1

if status(E) = unexplored

ri ← FIND(i)

rj ←FIND(j)

if ri = rj

status(E) ← false //E is not in MST

else

res← FindMin(w(E)) //find the min edge that connects two different components

if res = w(E) //E is the min edge connects two different components

UNION(i, j)

status(E) ← true //E is in MST

return status(E)

***8. Finding shortest path from source node to all*** nodes

**(1) Dijkstra (BFS) (nonnegative edges, weighted,** undirected)

dist[s] = 0 + visited array

for every unvisited node

set to visited

for every adjacent nodes

find min dist

DIJKSTRAMST(s)

put s in the priority queue

empty T[]

while the priority queue is not empty

extract node u from the priority queue with the minimum weight

remove u from priority queue

add u to the tree

for all edges from u to unvisted nodes v

if v is not in priority queue

P(v) ← {P(v), u − v}

put v in the priority queue

else if P' = {P(u), u − v} < P(v)

P(v) ← {P(v), u − v}

**Proof:**

Suppose the current distance from s to u is not optimal

From s to t, suppose algorithm picks u to t, let's say the minimal is y to t. (s,y) < (s,u), should have picked y instead.

We want to prove this inductively with the invariant properties:

- At every inductive step, any element in our finalized explored set S has the correct

distance.

Base case: Starting node s is in our set S. Distance to itself is 0.

Inductive: Suppose we take in a node u into our set S, and suppose to contradict that d(s,u) != du. Then this is not the shortest path. Consider the real shortest path from s → u then. On this path s → u, there’s a “crossing” from S to V\S. The first crossing between a node x in S to a node y in V\S gives us the path: s →x → y →u. By inductive hypothesis we already know dx = d(s,x). Look at dy - it is dy = dx + f(x,y) = d(s,y). Why is it not less than? Because if it was, then this path from s→x→y→u cannot be the shortest path! Then, we argue that if y != u, by positive weighted edges, d(s,y) <= d(s,u), then our algorithm would’ve chosen y as the next node. Contradiction.

In a directed graph, determine if there is a node that can reach every node?

Use dijkstra. Use super source node that is connected to every other nodes to determine if there is a node that can reach every node.

Assign n,2n,3n,4n weights for the edges leaving the super node.

Let d(s,x) = kn + (n-1) mod n -> d(s,x)=n-1 shows it has visited every node

**(2) Bellman-ford (works on negative weights) O(VE)**

dist(s->s) = 0

dist(s->others) = ∞

dist(s->v) = min(dist(s->v), dist(s->u)+weight(u,v))

void BellmanFord(struct Graph\* graph, int src)

dist[src] = 0;

dist[V] <-- ∞

for every node

for every edge e

if dist[e.src] != ∞ and

dist[e.src] + weight < dist[e.dst]

dist[e.src] = dist[e.dst] + weight

***9. Finding shortest path among all pairs***

Floyd Warshall (like bellman ford)

void floydWarshall (int graph[][V])

dist[ij] <-- graph[ij]

for (k = 0; k < V; k++)

for (i = 0; i < V; i++) //src

for (j = 0; j < V; j++) //dst

dist[i][j] = min(dist[ij], dist[ik] + dist[kj])

***10. Strongly connected graphs***

DFS: SCC graph, directed graph in which there is path between all pairs of vertices, maximal subgraph

Tarjan:

void Graph::SCCUtil(node u, int disc[], int low[], stack st, bool stackMember[])

static int time = 0;

disc[u] = low[u] = ++time;

st->push(u);

stackMember[u] = true;

for every adjacent node v of u

if unvisited

SCCUtil(v, disc, low, st, stackMember);

low[u] = min(low[u], low[v]);

// Update low value if v in stack

else if (stackMember[v] == true)

low[u] = min(low[u], disc[v])

// head node found, pop the stack and found an SCC

if (low[u] == disc[u])

while (st->top() != u)

w <-- top()

stackMember[w] = false;

low[w] = disc[u]

st->pop();

stackMember[u] = false;

st->pop();

sccCount++

void Graph::SCC()

disc[1~V] <-- NIL

low[1~V] <-- NIL // earliest visited vertex that can be reached from subtree with current vertex

stackMember[1~V] <-- false //whether a node is in stack

for every node

if (disc[i] == NIL) ->unvisited

SCCUtil(i, disc, low, st, stackMember)

return low[]

***11. Directed acyclic graphs***

isCyclic(i, visited, recStack)

if resStack[i]

return True

if visited[i]

return False

visited[i] = True

recStack[i] = True

for each neighbor j of i

if isCyclic(j, visited, recStack)

return True

recStack[i] = False

return False

main()

for i ← 1 to n

if isCyclic(i, visited, recStack) then

Return True

return False

***12. Tree traversals***

**(1) Diameter of a tree**

height(current\_node, diameter):

if current\_node is a terminal node

return 0

for every child node of current\_node:

child\_height <-- height(current\_node->child, diameter)

max1 <-- maximum child height

max2 <-- second maximum child height

diameter <-- maximum between the old diameter and (max1 + max2)

return 1 + max1

***13. Recursion***

**(1) Celebrity: knows nobody and everybody knows him**

- If A knows B, A is not a celebrity, B could be a celebrity

- If A doesn't know B: B is not a celebrity, A could be a celebrity

- You can eliminate a person in every iteration -->O(n)

FINDCELEBRITY(M)

c ← 1 # candidate

for i ← 2 to n

if M[c,i] = 1

c←i

if ISCELEBRITY(c)

return c

else

return None

ISCELEBRITY(c)

for i ← 1 to n

if i ̸= c and (M[i,c] = 0 or M[c,i] = 1) return FALSE

return TRUE

**(2) Water fill up**

4 1 2 5 3 4

-

- - -

- - - -

- - - - -

- - - - - -

4 4 4 5 5 5 -> left O(n)

5 5 5 5 4 0 -> right O(n)

4 4 4 0 4 0 -> find the loser O(n)

0 3 1 0 1 0 -> find the difference O(n)

Brute force: look at the tallest buildings on my left and right, the loser limits my capacity --O(n^2)

Dynamic programming: idea is we dont have to keep checking the right tallest building unless I am taller than the current tallest building. We dont have to keep checking the left highest unless I am now the tallest building.

water <- 3+2+1 = 6

***14. DFS***

void Graph::DFSUtil(int v, bool visited[])

visited[v] = true;

for every adjacent node of v

if (!visited[\*i])

DFSUtil(\*i, visited);

void Graph::DFS(int v)

new visited[] <-- false

for (int i = 0; i < V; i++)

if (visited[i] == false)

DFSUtil(i, visited)

**(1) Counting number of islands**

void DFS(int M[][COL], int row, int col, bool visited[][COL])

visited[row][col] = true;

for all 8 neighbors

if the neighbor is within the range and is '1'

DFS(M, neighbor\_i, neighbor\_j, visited);

int countIslands(int M[][COL])

bool visited[ROW][COL] <-- false

int count = 0;

for every grid

if is '1' and unvisted

DFS(M, i, j, visited);

++count;

return count;

***15. BFS***

visited array

visit source node

push source node to queue

while queue is not empty, visit every node’s adjacent node, push to queue if every adjacent is visited

while(!queue.empty()))

s = queue.front();

queue.pop\_front();

for adjacent nodes of s

if (!visited[\*i])

visited[\*i] = true

queue.push\_back(\*i)

**(1) Detect cycle**

for every visited node's adjacent nodes

if visited and not a parent

cycle

**(2) Detect Bipartite**

for every edge

assign two color flags for nodes being visited for the first time

if an edge has two nodes of the same color flag

return false

return true

***16. Functions***

repeatedly remove elements with no x mapping to it

Bijection(f)

counter[]<--0 // #elements mapping to y

for i : 1 to n

counter[f(i)]++ // f(i) is y

for i : 1 to n

if counter[i] = 0 // no element mapping to it

add i to Queue

while queue not empty

pop top

S <-- S - {top}

counter[f(top)]--

if counter[f(top)] = 0 //now there is no element mapping to it

add f[top] to Queue