***1. Time complexity for algorithms***

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)

Heap: build in O(n),search,insert in O(logn),extract min/max in O(1)

Union-find: one round O(logn)

DFS/BFS:O(|V|+|E|)

Dynamic programming: often O(m\*n)

***2. 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) 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

(2) 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

(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

OPT[j] = min{OPT[i-1]+function[i,j]}

1<=i<=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]

(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

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]

midterm knapsack w/ replacement: OPT=min{OPT[i-1,w], OPT[i-1,w-w\_i]+weight[i]}

(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

(18) Billboard building

no two of the billboards be within less than or equal to 5 miles of each other. You’d like to place billboards at a subset of the sites so as to maximize your total revenue, subject to this restriction.

OPT[i] = max{OPT[i-1], OPT[f(i)]+rev[i]}

*f*(*i*) denote the easternmost site *xj* that is more than 5 miles from *xi*. Since sites are numbered west to east, this means that the sites *x*1, *x*2, . . . , *xf*(*i*) are still valid options once we’ve chosen to place a billboard at *xi*, but the sites *xf*(*i*)+1, . . . , *xi*−1 are not.

(19) weighted interval scheduling O(n)

Compute-Opt(j)

If j=0 then Return 0

Else if M[j] is not empty then Return M[j]

Else

M[j]=max(vj+Compute-Opt(p(j)), Compute-Opt(j − 1))

Return M[j]

We define p(j) = 0 if no request i < j is disjoint from j. p[j]=#requests completely ahead of j.

(20) Segmented least square O(n^2) if e found

If the last segment of the optimal partition is pi, . . . , pn, then the value of the optimal solution is OPT(n) = e\_i,n + C + OPT(i − 1). e\_i,n = min error from these points. C=multiplier.

For the subproblem on the points p1, . . . , pj, OPT(j) = min(ei,j + C + OPT(i − 1)),

1≤i≤j

and the segment pi , . . . , pj is used in an optimum solution for the subproblem if and only if the minimum is obtained using index i.

(21) sequence alignment O(mn)

Minimize sum of gap and mismatch

OPT(i,j)=min[α\_xiyj +OPT(i−1,j−1),δ+OPT(i−1,j),δ+OPT(i,j−1)].

δ:gap penalty, α:mismatch cost

(22) Shorted path O(n^3)

If G has no negative cycles, then there is a shortest path from s to t that is simple (i.e., does not repeat nodes), and hence has at most n − 1 edges.

OPT(i, v) = min(OPT(i − 1, v), min(OPT(i − 1, w) + cost\_vw) w∈V).

Shortest-Path(G, s, t)

n= number of nodes in G

Array M[0...n−1,V]

Define M[0,t]=0 and M[0,v]=∞ for all other v∈V

For i=1,...,n−1

For v∈V in any order

Compute M[i, v] using above eqn

Return M[n − 1, s]

***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)

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

(1) 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

***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])

***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 O(logn)***

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

***17. Max flow***

augment(f , P)

Let b=bottleneck(P,f) For each edge (u,v)∈P

If e=(u,v) is a forward edge then increase f(e) in G by b

Else ((u, v) is a backward edge, and let e = (v, u)) decrease f(e) in G by b

Endif Endfor

Return(f)

(1) fordfulkerson O(|E|\*f)

set up residual graph

while BFS augmenting path exists (visited all vertices or saturated residual):

find bottleneck capacity

update max\_flow

update residual graph

return max\_flow

(2) circulation

Suppose we have multiple sources and multiple sinks. Each sink wants to get a certain amount of flow

(its demand). Each source has a certain amount of flow to give (its supply). We can represent supply as negative demand. Goal: find a flow f that satisfies demand and capacity constraints. S=negative demand, T=positive demands. Convert to maxflow: all nodes in S connect to s\*, all nodes in T connect to t\*. Do FF.

(3) bipartite matching

i. Build residual graph with capacity = 1. Add source and sink. We use Ford-Fulkerson algorithm to find the maximum flow in the flow network built in step 1. The maximum flow is actually the MBP we are looking for.

ii. Hall’s theorem reduction: systems of distinct representatives exist iff union of all m sets have >= distinct m elements.

e.g. team/days: For any set of k days, there are at least k distinct winners. If not, there is a team that lost every game. By contradiction, there is k winners that won this team.

(4) max flow min cut application

i. edge disjoint paths: construct G’ replace every edge with parallel edges with cap=1. Max flow value=edge disjoint paths between s and t(1 cap limits one path). By maxflow mincut, a cut has capacity of maxflow. A cut in G’ is a cut in G. Since the cut in G’ is minimal, cut in G is also minimal and equal to max flow. Menger’s theorem: in a finite graph, the size of a minimum cut set is equal to the maximum number of disjoint paths that can be found between any pair of vertices.

(6) increase max flow in O(m+n): construct residual graph G’=add 1 to an edge in G. Do one iteration of FF, if there is an augmenting path, maxflow++

(7) decrease max flow in O(m+n): compose the residual graph on the original flow. If the decreased edge was not at capacity (that is, it still has positive residual capacity), then we can decrease the edge capacity by one without affecting the maximum flow. If not, then we add one to the negative capacity on the edge, and look for an augmenting path in reverse (going from t to s instead of from s to t) which includes the decreased edge.

(8) Exam scheduling max flow

Solution: let undirected unweighted bipartite graph G = (LeftV,RightV,E). For any classes, rooms and times combination, let each node si ∈ LeftV represent a class Ei, and each node tjk ∈ RightV represent a room Sj at time Tk. Add an edge between a node si ∈ LeftV and a node tjk ∈ RightV only if Ei < Sj (class size smaller than room size).

The maximum number of exams that can be scheduled is the max matching (of edges that share no vertices) in bipartite graph G. To solve the max matching in bipartite graph4, we add a node a that connects to all si ∈ LeftV, and a node b that connects to all tjk ∈ RightV, set the capacity of the newly added edges to 1, the capacity of the already existing edges to ∞, and do MaxFlow from a to b.

A schedule exists iff the MaxFlow equals with the number of classes; For each edge (si,tjk) in the max matching, we assign the class Ei to room Sj at time Tk.

(9) Deal cards and select

Solution: let undirected unweighted bipartite graph G = (LeftV,RightV,E). For any dealing of cards, let each node in LeftV represent a pile of cards, and each node in RightV represent a rank. Add anedgebetweenanodes∈LeftV andanodet∈RightV foreachcardofranktinpiles.

In the bipartite graph G, each node s ∈ LeftV has degree 4, and each node t ∈ RightV has degree 4. So for each subset of nodes S ⊆ LeftV, |N(S)| ≥ |S| (Since otherwise there exists an S′, such that the degree of S′ = 4 · |S′| > 4 · |N(S′)|, which contradicts with the definition of N(S′) since the total degree on S′’s side is larger than that on N(S′)’s side). Thus by Frobenius Hall theorem3, there exists a perfect matching between LeftV and RightV. And selecting a card of rank t in pile s only if the edge (s, t) is in the perfect matching shows the conclusion.

(10) Scheduling planes

Each vertex represents a source and a destination. Each edge has a capacity of exactly 1 meaning they must be served with one plane. If a flight i is reachable from flight j, we have d\_i->s\_j. Add s and t. For s->s\_i it can be 0 or 1 meaning the flight begins the day with the city s\_i. For d\_j->t, 0 or 1. edge(s,t)=0 to k. Finally, the node s will have a demand of −k, and the node t will have a demand of k. All other nodes will have a demand of 0.

(11) Survey design

customer\_i->product\_j if purchased (0,1). s->customer\_i = #questions. product\_j->t = #customers asked. edge(t,s)=overall #questions.

(12) Ideas

Build bipartite graphs. One unit flow. Max flow min cut.

***18. NP completeness***

**steps:**

1. prove NP (check solution in poly time)

2. prove Given NPC <p target NPC:

(a) construct target data structure, from given data structure

(b) claim given structure has the given property iff target structure has the target property

(c) if there exists given, implies in target... prove target is satisfied

(d) if there exists target, implies in target... prove given is satisfied

(e) construction of target structure is done in poly time

(1) H cycle <p traveling salesman

given G a H cycle, construct G’ with same edges=0 and vertices. Add more edges=1. If found TSP tour total length<=0, then there is H cycle.

(2) clique <p independent set: construct complement

(3) hitting set <p dominating set: construct set nodes and element nodes. Make clique of set nodes.

(4) dominating set <p hitting set: for each node create a set, add adjacent nodes to the set.

(5) set cover <p dominating set: construct set nodes and element nodes, connect set node with the elements in the set. Make clique for all set nodes.

(6) dominating set <p set cover: for each node create a set, add adjacent nodes to the set.

(7) set cover <p hitting set: given a set cover, if a set covers an element, then that set hits the element. there is a hitting set of size at most k if and only if there is a set cover of size at most k.

(8) hitting set <p dominating set: construct graph for each node create a set, add adjacent nodes to the set. hitting every set=hitting every vertex

(9) dominating set <p set cover: for each node create a set, add adjacent nodes to the set. each element node in the DS covers at least a set. all nodes in ds covers all sets by our construction.

(10) subset sum <p partition: We will construct our new set S' such that

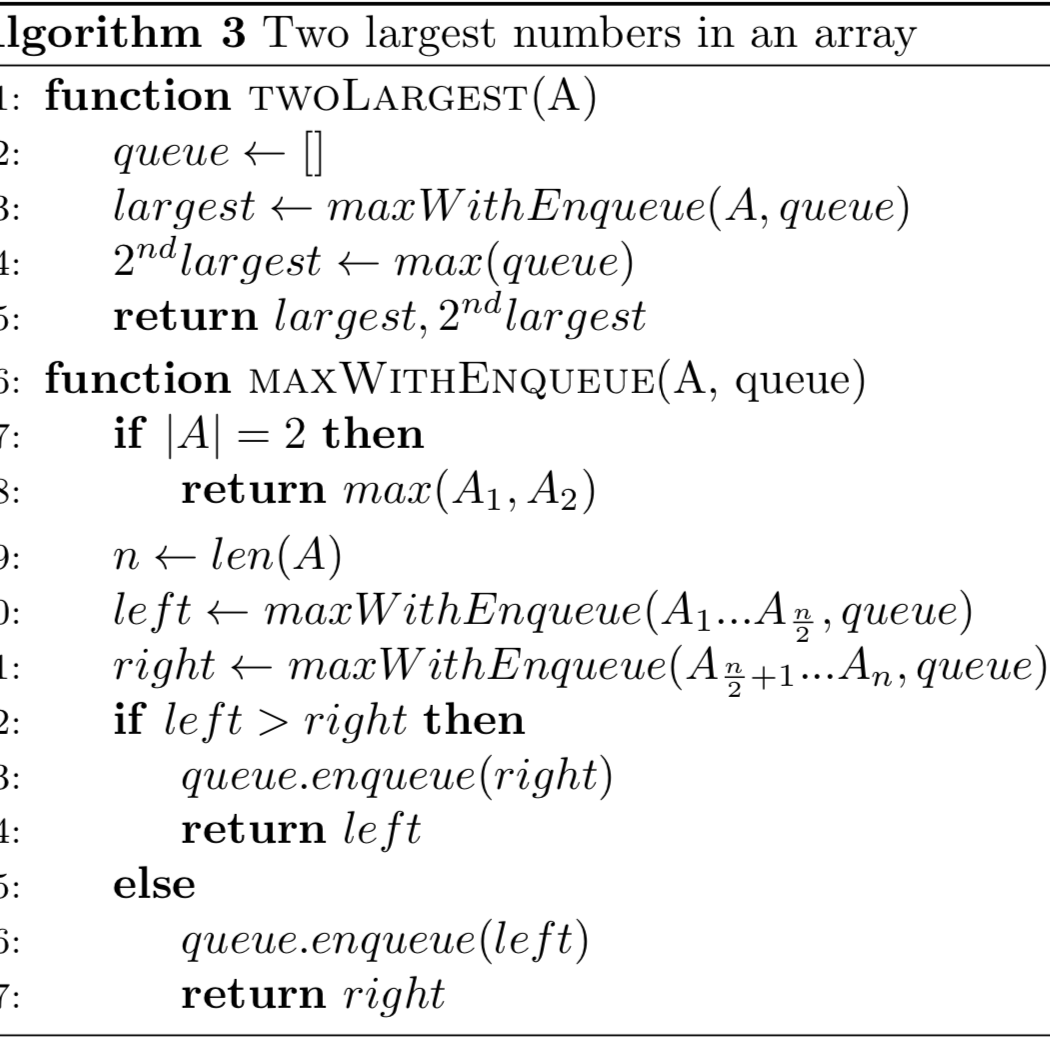
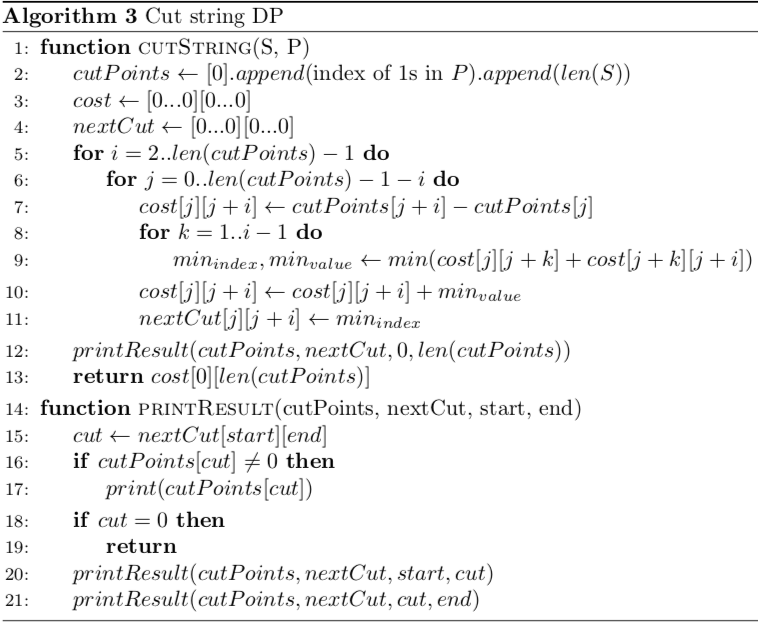
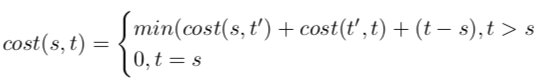
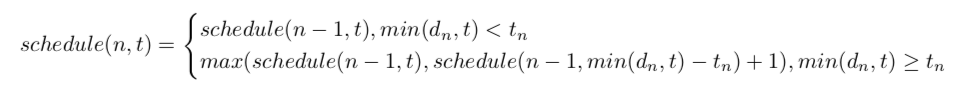
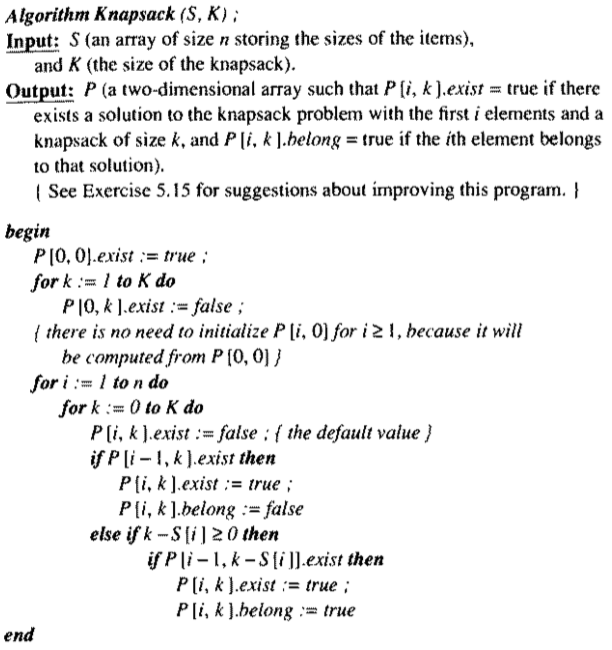
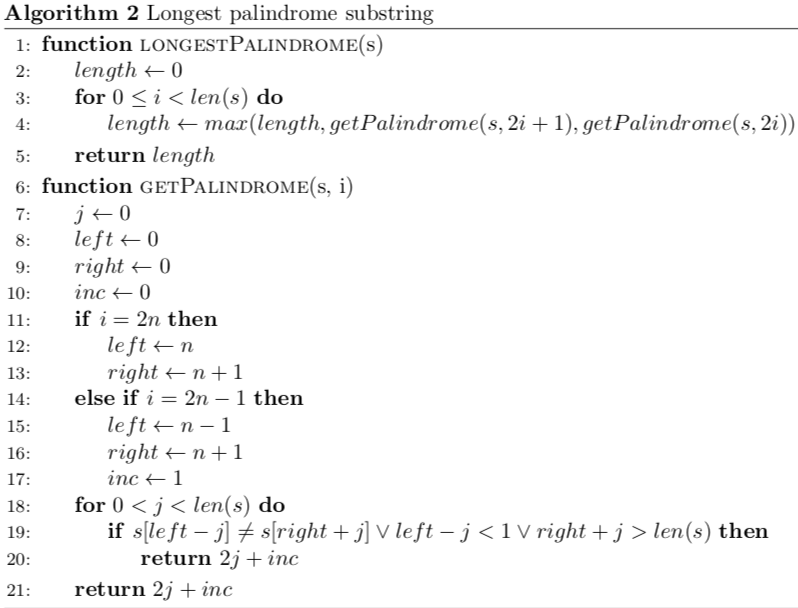
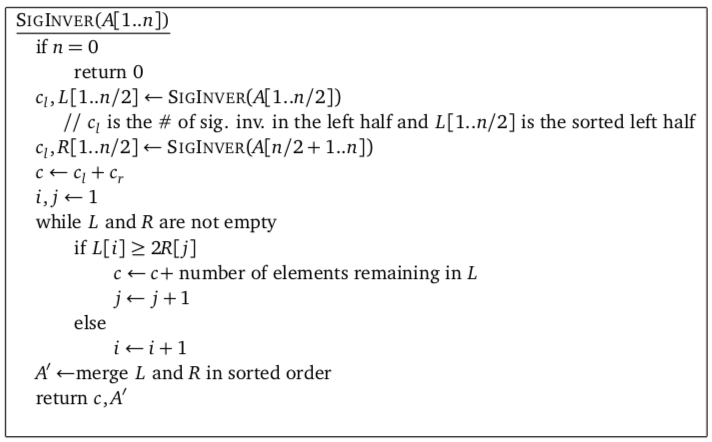
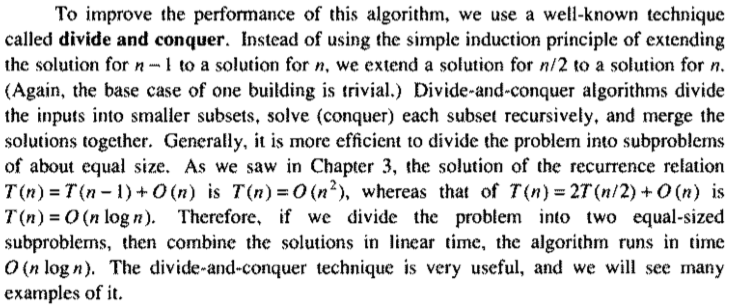
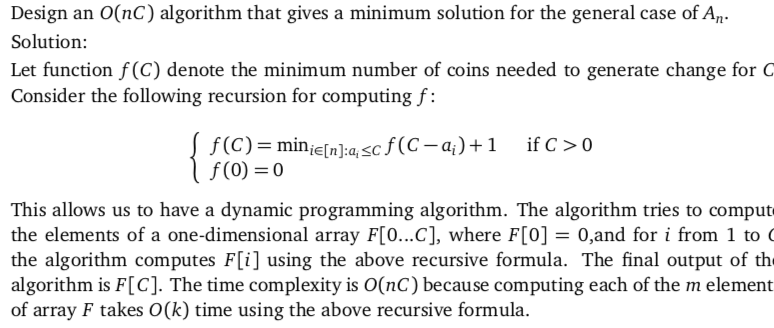
If a subset of S has total k, we can add in a new element to make up the difference to half the total sum. Add in a new element to the set such that a subset with the appropriate sum also forms a partition.

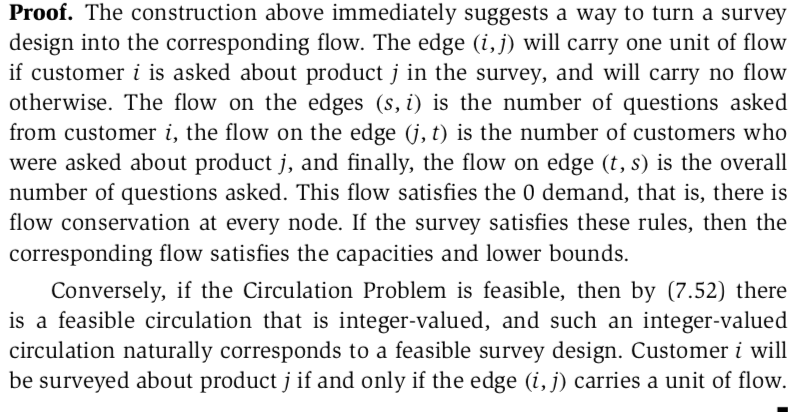
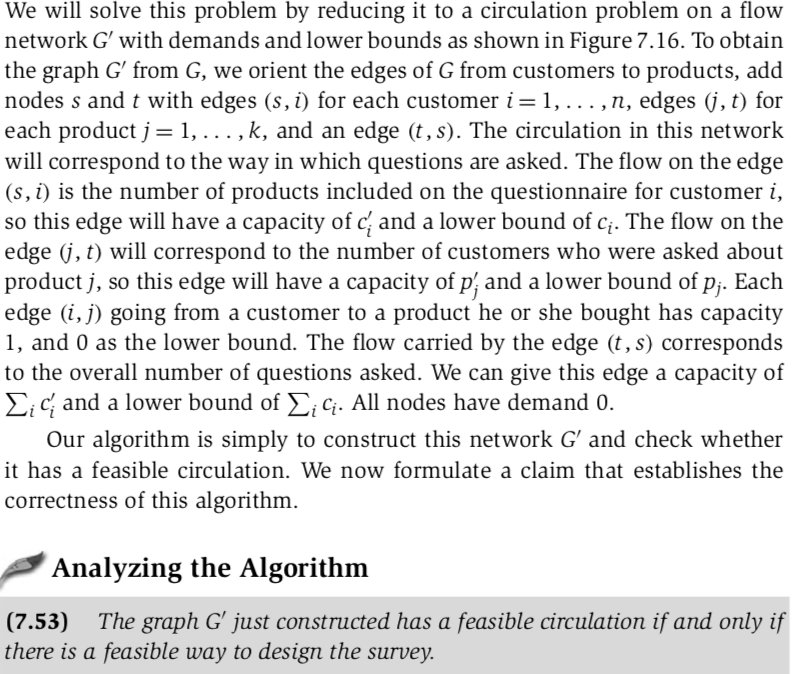
The new element added in might need to go in the subset that originally added to k, or it might have to go in the complement of that set.

(11) exact cover <p knapsack(subset sum): We will construct our set S and number k such that Each number corresponds to a set of elements. Suppose there are n elements in the universe and k different sets. Replace each set S with a number that is 1 in its ith position if i ∈ S and has a 0 in its ith position otherwise. Set k to a number that is n copies of the number 1. and k corresponds to the universe U.

(12) bin packing

(13) Hamiltonian path <-> Hamiltonian cycle: split a vertex or add a vertex connecting source and destination





initialize skyline or result as empty, then one by one add buildings to skyline. A building is added by first finding the overlapping strip(s). If there are no overlapping strips, the new building adds new strip(s). If overlapping strip is found, then height of the existing strip may increase. Time complexity of this solution is O(n2)

We can find Skyline in Θ(nLogn) time using [Divide and Conquer](https://www.geeksforgeeks.org/divide-and-conquer-set-1-find-closest-pair-of-points/). The idea is similar to [Merge Sort](http://geeksquiz.com/merge-sort/), divide the given set of buildings in two subsets. Recursively construct skyline for two halves and finally merge the two skylines.

How to Merge two Skylines?  
The idea is similar to merge of merge sort, start from first strips of two skylines, compare x coordinates. Pick the strip with smaller x coordinate and add it to result. The height of added strip is considered as maximum of current heights from skyline1 and skyline2.

