```
LCS-Length(X,Y)
m = length(X)
n = length(Y)
for I = 1 to m c[I,0] = 0
for j = 1 to n c[0,j] = 0
for I = 1 to m
  for j = I 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
O(mn)
```

```
def mergeSort(list):
  if len(list) > 1:
    mid = len(list)//2
    left = list[ :mid]
    right = list[mid: ]
    mergeSort(left)
    mergeSort(right)
    i = 0, j = 0, k = 0
    while I < len(left) and j <
len(right):
      if left[i] < right [j]:
        list[k] = left[i]
        i += 1
      else:
        list[k] = right[j]
        j += 1
      k += 1
    while i < len(left):
      list[k] = left[i]
      i += 1
      k += 1
    while j < len(right):
      list[k] = right
      j +=1
      k+=1
O(n log n)
```

```
def insertionsort(list):
  for index in range(1, len(list)):
    currVal = list[index]
    pos = index
    while pos >0 and list[pos-1] >
currVal:
      list[pos] = list[pos-1]
      pos = pos -1
    list[pos] = currVal
O(n^2)
```

```
STOOGESORT(A[0…n-1])
  if n =2 and A[0] < A[1]
    swap A[0] and A[1]
  else if n > 2
    k = ceiling(2n/3)
    STOOGESORT(A[0…k-1])
    STOOGESORT(A[n-k…n-1])
    STOOGESORT(A[0…k-1])
```

```
def linearFindSubMax(arr):
  n = array length
  left = 0, right =0, curr = 0
  maxSum = -1
  tempSum = 0
  for x in range(0,n)
    if arr[curr] >= tempSum + arr[x]:
      tempSum = arr[x]
    elif arr[curr] <= tempSum + arr[x]:
      tempSum = tempSum + arr[x]
    if tempSum > maxSum:
      right = x
      left = curr
      maxSum = tempSum
    elif tempSum = arr[x]
      curr = x
  return max, left, right+1
O(n)
```

```
def Fibonacci(n):
  if n == 1:
    return 0
  if n==2:
    return 1
  else:
    return Fibonacci(n-1) +
Fibbonacci(n-2)
T(n) = T(n-1) + T(n-2) exponential.
```

```
def enumeration(arr)
if len of arr is 0
  return 0
else
  maxSum = 0
  for I in range (0, len(arr))
    for j in range (I, len(arr))
      temp = 0
      for k in range (i, j+1)
        temp = arr[k]
        if temp > maxSum
          maxSum = temp
          max subarray = array from I to j+1
return subarray and maxSum
O(n^3)
```

```
def makeChange(V, A):
  numCoins = [infinity]*A+1
  index = [-1]*A+1
  numCoins[0] = 0
  index[0] = 0
  for j in range (0, V.length)
    for i in range (1, A+1)
      if i > V[j]
        if numCoins[i – V[j]] + 1 <
numCoins[i]
          numCoins[i] = numCoins[i-V[j]]
+1
          index[i] = j
  C = [0]*V.length
  while A != 0
    C[index[A+1]] += 1
    A = A-V[index[A]]
O(VA)
```

```
Fucntion Find-Array-Max(A,n)
  if(n=1) then
    return (A[1])
  else
    return[max(A[n]), Find-Array-
Max(A,n-1))
T(n) = T(n-1) + c  = Theta(n)
```

```
minMax(arr)
  minmax  = []
  if len(arr) = 1
    minmax[0] = arr[0]
    minmax[1] = arr[0]
  elif len(arr) = 2
    minmax[0] = min(arr[0],arr[1])
    minmax[1] =
max(arr[0],arr[1])
  else
    left = []
    right = []
    mid = len(arr)/2
    left = minMax(arr[0:mid])
    right =
minMax(arr[mid+1:len(arr)-1)
    minmax[0] = min(left[0],
right[0])
    minmax[1] =
max(left[1],right[1])
    return minmax
T(n) = 2T(n/2) + C = Theta(n)
```

```
def lis(arr): //longest inc subseq
  n = len(arr)
  lis = [1]*n
  for i in range (1, n):
    for j in range (0, i):
      if arr[i] > arr[j] and lis[i] <
lis[j] +1:
        lis[i] = lis[j]+1
  max = 0
  for I in range (n):
    max = max(max, lis[i])
  return max
O(nlogn)
```

```
Greedy-Activity-Selector(s,f)
n <- length(s)
A <- {a1}
i <- 1
for m <-2 to n
  do if sm >= fi
    then A <- A union {am}
      i <- m
return A
```

```
def dpKnapsack(W, wt, val, n):
  K = [[0 for x in range(W+1)] for x in range(n+1)]
  for i in range (n+1):
    for w in range(W+1):
      if i == 0 or w == 0
        K[i][w] = 0
      elif wt[i-1] <= w:
        K[i][w] = max(val[i+1] + K[i-1][w-wt[i-1]],
K[i-1][w])
      else:
        K[i][w] = K[i-1][w]
  return K[n][W]
O(nW)
```

$$\text{If } lim_{n\to\infty}\frac{f(n)}{g(n)} = \begin{cases} 0 & then\ f(n)\ is\ O(g(n)) \\ c>0 & then\ f(n)\ is\ \Theta(g(n)) \\ \infty & then\ f(n)\ is\ \Omega(g(n)) \end{cases}$$

```
def palindrome(seq):
  n = len(seq)
  L = [[0 for x in range(n)] for x in
range(n)]
  for i in range(n):
    L[i][i] = 1
  for cl in range(2, n+1):
    for i in range(n-cl+1):
      j = i + cl -1
      if seq[i] == seq[j] and cl ==2:
        L[i][j] = 2
      elif seq[i] == seq[j]:
        L[i][j] = L[i+1][j-1]+2
      else:
        L[i][j] = max(L[i][j-1],
L[i+1][j])
  return L[0][n-1]
O(n^2)
```

```
Krusal(){
  A= NULL;
  for each v in V
    MakeSet(v)
  sort E by invd edge weight w
  for each (u,v) in E (in ordered order)
    if FindSet(u) != FindSet(v)
      A=A union {{u,v}}
      Union(FindSet(u), FindSet(v))
}
O(ElgV)
```

```
fractionalKnapsack(S,W)
Input set S of items w/
benefit bi
and weight wi; Max
weight W
Output: amount xi of each
item i
to maximize benefit with
weight at most W
for each item i in S
  xi = 0
  vi = bi/wi {value}
w = 0
while w < W
  remove item I with
highest vi
  xi  = min(wi, W-w)
O(n log n)
```

```
Def schedule_jobs(jobArray):
  sJobs = sorted(jobsArray, key=lambda k:
k['penalty'], reverse – true)
  jobsSched = [0]*len*(sJobs)
  result =[0]*len(sJobs)
  for i in sJobs:
    for j in range(min(len(sJobs), i['deadline']), 0):
      if jobsSched[j] ==0:
        result[j] = i['id']
        jobsSched[j] =1
        break
  for I in range(0, len(jobsSched)):
    if jobsSched[i] !=0:
      print(result[i]
O(n^2 + n)
```

```
def roadTrip(hotels):
  curr = 0
  while distance <= farthestHotelDist
    prev = curr
    while curr <= hotels[len(hotels)-1] &
hotels[curr+1] – hotels[prev> <= d
      curr += 1
    add currr to solution.
```

Decrease and Conquer

Muster Method for *"decrease and conquer"* recurrences of the form

$$T(n) = a\,T(n\text{-}b) + f(n)$$

for some integer constants a, b > 0, d ≥ 0 .
If $f(n)$ is $O(n^d)$ then

$$T(n) = \begin{cases} O(n^d), & if\ a<1, \\ O(n^{d+1}), & if\ a=1 \\ O(n^d a^{n/b}), & if\ a>1. \end{cases}$$

Master Method

"Formula" for solving recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where, a ≥ 1, b > 1, and f(n) > 0

case 1: if $f(n) = O(n^{\log_b a - \varepsilon})$ for some ε > 0, then: $T(n) = \Theta(n^{\log_b a})$

case 2: if $f(n) = \Theta(n^{\log_b a})$, then: $T(n) = \Theta(n^{\log_b a} \lg n)$

case 3: if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some ε > 0, and if

af(n/b) ≤ cf(n) for some c < 1 and all sufficiently large n, then:

$T(n) = \Theta(f(n))$

regularity

You just started a consulting business where you collect a free for completing various types of projects (diff per proj). You can select in advance the projects you will work on during some finite time period. You work on only one project at a time and once you start a project it must be completed to get the fee. There is a set of n projects p1..p2 each with a duration d1…d2 and you receive a free f1..f2 associated with it. That is project p1 takes d1 days you collect f1 dollars when compelted.
Each of the n projects must be completed in the next D days or you lose its contract. Unfortunately, you do not have enough time to complete all the projects.Your goal is to select a subset S of the project to complete to max the frees.
What type of algorithm would you use? Why?
DP, similar to 0-1 knapsack
Describe, if using DP give formula:
Let OPT(i,d) be the max fee collected for considering projects 1….
The base cases are OPT(i,0) = 0 for I = 1…n and OPT(0,d) for d= 1… D

```
for i = 1 to n{
  for j = 1 to D{
    if(di > j){
      OPT(I,j) = OPT(i-1,j) //not enough to complete proj i}
    Else{
      OPT(I,j) = max(OPT(i-1, j), OPT(i-1, j-di)+fi))
Theta(nD)
```

Scheduling jobs with early complete dates and bonuses:

(a) What type of algorithm could you use to solve this?
Greedy

(b) Describe the algorithm verbally with enough detail to obtain the running time.
Note since there are n days it will take at most n days.
1 Sort jobs by bonuses in dec order. You will select the biggest bonus first.
2 For i=1 to n Select the next job ji to be scheduled according to the sorted bonus order. There are two outcomes
The job can be completed early and you get the bonus. Schedule the job ji as close to the early completetion date ei as possible without exceeding the date. You get the bonus bi
The job cannot be scheduled by the early completion date. If there are no days left on or before ei to schedule job ji then schedule the job at the first available day closest to day n. The job is put at the end of the queue so not to conflict with other jobs.

(c) What is the funning time? Explain
1 the sorting takes theta(nlgn)
2 To find the spot in the schedule can take in worse case 1+2….+ n = theta(n^2) depending on the data struct used.

(a)Describe in words an efficient algorithm that Benny could use to maximize his total points. Why does your Algorithm give the optimal. (Fractional knapsack)
Benny will first calculate the points per minute of each problem, that is the ration p/t, and then sort the p/t ratio into desc order.
Benny Starts with the problem that has the greatest ratio of points to time. He will then solve the problems in order until the final problem for which the remaining time is less than the time it would take him to solve it. For this last problem he would then solve as much as possible given the time left and earn partial credit on the final question.
This gives the correct answer because the problem exhibits the characterstics of a greedy algorithm problem, a locally optimal solution is globally optimal, and each choice splits the problem into one remaining subproblem, by always making the next greedy choice, the problem that will earn him points the fastest, he will maximize his points.
(b)Theta(nlgn), in other words the greatest factor on the running time is the sorting problem.

Acme Industries produces four types of men's ties using three types of material. Your job is to determine how many of each type of tie to make each month. The goal is to maximize profit, profit per time = selling price – labor cost – material cost. Labor cost is $.75 per tie for all four types of ties. The material requirements are below

Material	Cost Per Yard	Yards Available per Month
Silk	$20	1000
Polyester	$60	2000
Cotton	$9	1250

	Silk=s	Poly=p	Blend1=b	Blend2=c
Selling price per tie	$6.70	$3.55	$4.31	$4.81
Monthly Min units	6000	10000	13000	6000
Monthly max units	7000	14000	16000	8500

Yards	Silk	Poly	Blend1	Blend2
Silk	.125	0	0	0
Poly	0	.08	.05	.03
Cotton	0	0	.05	.07

type	Selling price	labor	material	Profit pet tie
Silk s	6.7	.75	2.5	3.45
Poly p	3.55	.75	.48	2.32
Blend1 b	4.31	.75	.75	2.81
Blend2 c	4.81	.75	.81	3.25

```
LP Code
max 3.45s + 2.32p + 2.81b _
3.25c
ST
  .125s <= 1000
  .08p + .05b + .03c <=2000
  .05b + .07c <= 1250
  s >= 6000
  s <= 7000
  p >= 10000
  p <= 14000
  b >= 13000
  b <= 16000
  c >= 6000
  c <= 8500
  non-negativity constraints
```

```
MST-Prim(G, w r)
  Q = V[G]
  for each u in Q
    key[u] = inf
  key[r] = 0
  p[r] = NULL;
  while (Q not empty)
    u = ExtractMin(Q);
    for each v in Adj[u]
      if (v in Q and w(u,v) < key[v])
        p[v] = u
        key[v] = w(u,v)
binary heap = O(E lg V)
fib heap = O(V log V +E)
```

```
BFS(G, s){
  Initialize vertices;
  Q = {s}; //queue containing s
  while(Q not empty){
    u = DEQUEUE(Q);
    for each v in G.adj[u]{
      if(v.color == WHITE)
        v.color == GREY;
        v.d = u.d+1;
        v.p = u
        ENQUEUE(Q, v);
    }
    u.color = BLACK;
  }
}
O(V+E)
```

```
babyfaceVHeel(graph, source)
  let Q be a new queue
  add source to Q
  while Q is not empty
    u = queue.pop
    indicate that u has been encountered
    if u does not have a team
      u.team = babyface
    for rival in u.rivals
      if rival doesn't have a team
        assign rival opposite team from u
      if rival does have team and it is same as u's
        return false
      if rival hasn't been encountered
        add to Q
```

Consider the 0-1 knapsack where all items have the same benefit of b. We have n items where wi is the weight of the ith item, every item has a benefit b, and the capacity of knapsack is W. Find subset of items whos total weight is at most W and whos total benefit Is maximized. (knapsack without repetition)

(a) What algorithm design paradigm is most appropriate for this problem?
Greedy

(b) Verbally Describe efficient algorithm.
Merge sort the input array in asc order based on weight. The items with lowest weight would eb first. This would take O(nlgn). Algorithm would step through array, placing items in knapsack as long as the item will fit. This takes O(n)

(c) Total runtime O(nlgn)

```
Dijkstra(G,w,s)
  InitializeSingleSource(G,s)
  S <- NULL
  Q <- V[G]
  while Q != 0 do
    u <- ExtractMin(Q)
    S <- S union {u}
    For v in Adj[u] do
      Relax(u,v,w)

InitializeSingleSource(G,s)
  for v in V[G] do
    d[v] <- inf
    p[v] <- 0
  d[s] <- 0

Relax(u,v,w)
  if d[v] > d[u] + w(u,v) then
    d[v] <- d[u] + w(u,v)
    p[v] <- u
Array O(v^2)
Binary heap O(ElgV)
Fib heap O(VlgV + E)
```

```
DFS(G){
  for each vertex u in G.V{
    u.color = WHITE;
  }
  time = 0;
  for each vertex u in G.V{
    if (u.color == WHITE)
      DFS_Visit(G,u);
  }
}
O(V^2)
DFS_Visit(G,u){
  u.color = GREY;
  time = time+1;
  u.d = time;
  for each v in
G.Adj[u]{
    if (v.color ==
WHITE)
      DFS_Visit(G,v);
  }
  u.color = BLACK;
  time = time+1;
  u.f = time;
}
```

```
BellmanFord()
  for each v in V
    d[v] = inf
  d[s] = 0
  for i=1 to |V|-1
    for each edge (u,v) in E
      Relax(u,v,w(u,v))
    for each edge (u,v) in E
      if(d[v] > d[u] + w(u,v))
        return "no solution"
Relax(u,v,w): if (d[v] > d[u]+w) then d[v] = d[u] +w
O(VE)
```

Reduce 3-SAT to Clique
Pick an instance of 3-SAT, phi, and transform into <G,k> an instance of Clique
If phi has m clauses, we create a graph with m clusters of 3 nodes each and set k=m
Each cluster corresponds to a caluse
Each node in a cluster is labeled with a literal from the clause
We do not connect any nodes in the same cluster
We connect nodes in different clusters whenever they are not contradictory
Any k-clique in this graph corresponds to a satisfying assignment.

Reduce 3-SAT to Subset Sum
Choosing the subset numbers from the set s corresponds to choosing the assignments of the variables in the 3-SAT formula.
The different digits of the sum correspond to the different clauses of the formula
If the target t is reached, a valid and satisfying assignment is found.
Let phi in 3-CNF with k caluses and l variables x1…xl
Create a Subset-Sum instance <Sphi, t> by:
2l + 2k elements of
Sphi = {y1z1…ylzl,g1,h1…gk,hk)
yj indicates positive xj literals in clauses
zj indicates negated xj literals in clauses
gj and hj are dummies
For every 3-CNF phi, take target t=1…. And the corresponding set Sphi
If phi in 3-SAT then the satisfying assignment defines a subset that reaches the target
Also, the target can only be ontained via a set that gives a satisfying assignment for phi

You are going on another long trip(this tiem your headlines are working). You start on the road at mile post 0. Along the way there are n hotels, at mile posts a1 < a2…<an, where each ai is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel(at distance an), which is your destination.
You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of hotels). If you travel x miles during a day, the penalty for that day is (200-x)^2. You want to plan your trip so as to minimize the total penalty – that is the sum, over all travel days, of daily penalties. Give an efficient algorithm that determines min penalty for the opt sequence of hotels.

Let S[j] be the min total penalty when you stop at hotel j.

```
Let S[0] = 0
for j >= 1, j<=n
  S[j] = inf
  for I = 0, i<j
    S[j] = min{S[i] + (200 –(aj-ai)(^2, S[j]}
return S[n]
```

OSU student, Benny, is taking his CS 340 algorithms exam which consists of n questions. He notices that the professor has assigned points p1..p2 to each problem according to the professor's opinion of the difficulty of the problem. Benny wants to max the total num of points he earns on the exam, but he is worried about running out of time since there is only T minutes for the exam. He estimate that the amount of time it will take him to solve each of the n question is t1..t2. You can assume Benny gets full credit for every question he answers complete. Develop an algoirhtm to help Benny select which questions to answer. No partial credit is assigned to partially completed problems.

(a) What time of algorithm?
Dynamic Programming. Similar to 0-1 knapsack

(b) Describe the algorithm verbally and give psudocode. If you select dp give the formula.
Let OPT(I,d) be the max point earned for considering problem 1,… i with t minutes available
Base cases are OPT(I,0) = 1 for i= 1… n OPT(0,t) = 0 for t= 1…T

```
For I = 1 to n{
  For t = 1 to T{
    If(ti>t){
      OPT(I,t) = OPT(i-1,t);
    }else{
      OPT(I,t) = max(OPT(i-1,t), OPT(i-1,t-ti) + p1)
```

(c) Running time is theta(nT) because this is the number of sub problems.

(d) Now suppose the professor gave partial credit for the problems that are partially completed. Would you use the algorithm you described in this scenario?
No, I would not

(e) What algorithm would you use?
Fractional greedy knapsack type of problem

```
For each problem i in P
  xi = 0
  vi = pi/ti
  t = 0
  while t< T
    remove item i with highest point value
    xi = min(ti, T-t)
    t = t+min(ti, T-t)
```

Reduce Ham-Cycle to TSP
Let G=(V,E) be an instance of Ham-cycle. We construct an instance of TSP as follows
Form the complete graph G' = (V,E') where E' = {(i,j) : i, j in V and i != j} and
Define the cost function c by c(i,j) = {0 if (i,j) in E, 1 if (i,j) not in E}
The instance of TSP is then <G', c, 0> which is easily formed in polynomial time.
Show graph G has a Ham-cycle if and only if graph G' has a tour of cost at most 0.
Suppost the graph G has a ham cycle h
Each edge in h belongs to E and thus has a cost 0 in G'.
Thus h is a tour in G' with cost 0.
Conversely suppose that graph G' has a tour h' of cost at most 0. Since the cost of edges in E' are 0 and 1, the cost of tour h' is exactly 0 and each edge on the tour must have a cost 0.
Thus h' contains only edges in E.
Hence we conclude that h' is a ham-cycle in graph G

<u>Reduce Subset-Sum to Knapsack</u>

Subset-Sum = {<S, t> | S={y1….yk} and for some subset T = {y1…yl} is a subset of S, sum yi = t
Set bi = yi and wi = yi
Set W = t and K= t
Then for any subset T in the set of S
Sum yi = t if and only if sum bi = sum yi >= t and sum of wi = sum yi <=t

A Hamiltonian path in a graph is a simple path that visits every vertex exactly once. Show that HAM-PATH = { (G, u, v): there is a Hamiltonian path from u to v in G} is NP-complete. You may use the fact that HAM-CYCLE is NP-complete

The first thing we need to do in order to prove that HAM-PATH is NP-Complete is to prove that it can be verified in polynomial time. We need input in the form of {G, u, v} as well as a certificate that is simply a sequence of vertices ranging from 1 to n. In order to verify that the solution in the certificate given is a HAM-PATH we simply need to traverse each vertex in the certificate sequence and ensure that each vertex is visited exactly once. This traversal can certainly be done in polynomial time, which would put HAM-PATH into the NP category. Now to prove the complete part of NP-Complete we must find a problem that can be reduced into a HAM-PATH. From the NP-Completeness slide in the lecture we can see that a HAM-CYCLE can be reduced into a HAM-PATH. A HAM-CYCLE is simply a HAM-PATH that begin and end at the same vertex and we know that HAM-CYCLE is NP-complete so it is a good candidate. To convert from HAMCYCLE we will create a graph, G’, which is a copy of the graph used in the HAM-CYCLE, G, and add a couple vertices. The vertices that need to be added are u’ and v’. These vertices will only be connected to the corresponding versions of themselves in the copy of the graph i.e. u’ connects with only u and v’ connects with only v. G’ will have a HAM-PATH if (and only if) G has a HAM-CYCLE with edge = {u, v}. The HAM-PATH algorithm will need to be ran on each G’ for all edges in G and if there is no HAM-PATH present then there is no HAM-CYLCE in the original graph.

<u>Circuit-SAT</u>- Given a bool combination circuit composted of AND, OR and NOT gates, is it satisfiable?
<u>3-CNF-SAT</u> – each clause contains three Boolean literals. Can we satisfy each clause?
<u>Clique</u> – Given G and int k, does G contain a clique(set of nodes such that all pairs in the set are neighbors) of size k
<u>Vertex-cover</u> – Given a graph G and int k, does G have a vertex cover (set of nodes such that every edge is incident to at least one of them) of size <=k
<u>TSP</u> – Given a graph and int k, is there a ham cycle with total weight at most k
<u>Subset-sum</u> – Given a set of numbers S and a target T, is there a subset S’ of S such that y = t
<u>Independent set</u> – is there a vertex cover of size <= n-k

<u>Reduce Ham-Cycle undirected to directed</u>

Transform an undirected graph G into a directed graph H by replacing each edge vw of G by edges in each direction vw and wv.
Then if there is a ham cycle in G, there is one in H. For each edge in cycle G, the cycle H uses the edge from vw and wv pair that corresponds to the direction the edge was traversed in G.
If there is a ham cycle in H then there is one in G’, simply replace either edge vw or wv in the cycle in H by corresponding undirected edge.

Order of Complexity by Speed
O(1), O(lglgn),O(lgn),O((lgn)^k),O(n),O(nlgn),O(n^2),O(n^k),O(c^n)

Graph-Coloring

Mapmakers try to use as few colors as possible when coloring countries on a map, as long as no two countries that share a border have the same color. We can model this problem with an undirected graph G=(V,E) in which each vertex represents a country and verticies whose respective countries share a border are adjacent. A k-coloring is a function c:V->{1,2…k}
Such that c(u) != c(v) for every edge (u,v) in E. In other words the number 1..2..k represent the k colors and adjacent verticies must have different colors. The graph-coloring problem is to determine the min num of colors needed to color a given graph.

(a) Give an efficient algo to determine a 2-coloring of a graph

```
Let G=(V,E) be a graph to which a 2-coloring is applied.
Let C be an array indexed as C[0]<- color1 and C[1]<- color2
2-COLOR(G,C)
  For v in V
    v.visited <-false
    v.color<-none
  for v in V
    if v.visted == false
      TWO-COLOR-VISIT(v,0,C)
TWO-COLOR-VISIT(v,i,C)
  v.visited <-true
  v.color<-C[i]
  let N be a set of verts adj to v
  for n in N
    if v.visted == true
      if v.color == n.color
        return false
      else
        TWO-COLOR0VISIT(v,1-I,C)
  Return true
O(V+E)
```

Consider the problem COMPOSITE: given an integer y, does y have any factors other than one and itself? For this exercise, you may assume that COMPOSITE is in NP, and you will be comparing it to the well-known NP-complete problem SUBSET-SUM: given a set S of n integers and an integer target t, is there a subset of S whose sum is exactly t? Clearly explain whether or not each of the following statements follows from that fact that COMPOSITE is in NP and SUBSET-SUM is NP-complete:

a. SUBSET-SUM ≤p COMPOSITE.
No it does not follow. We know that SUBSET-SUM is NP complete which means it can be reduced to any other NP complete (or harder) problem. We only know that COMPOSITE is NP (which is a large umbrella) so we cannot say with 100% certainty that SUBSET-SUM can be reduced to COMPOSITE.

b. If there is an O(n^3) algorithm for SUBSET-SUM, then there is a polynomial time algorithm for COMPOSITE.
Yes, this we can infer. It is said if we can solve 1 NP-Complete algorithm in polynomial time, we can solve all of them in polynomial time. If SUBSET-SUM can be solved in polynomial time we are basically saying that P=NP, which would indicate anything in NP could be solved in polynomial time.

c. If there is a polynomial algorithm for COMPOSITE, then P = NP
No this does not follow. In order for P=NP COMPOSITE would have to be confirmed as NPComplete. Since we have not confirmed COMPOSITE to be NP complete, having a polynomial time does not mean that P=NP. Since P is a subset of NP it is possible to have NP problems that are also P, but it does not mean all of them are.

d. If P ≠ NP, then no problem in NP can be solved in polynomial time.
No this definitely does not follow. P is a subset of NP which means that some problems can be solved in polynomial time that are NP problems but not all problems. NP-Complete problems cannot be solved in polynomial time.

LONG-PATH is the problem of, given (G, u, v, k) where G is a graph, u and v vertices and k an integer, determining if there is a simple path in G from u to v of length at least k. Prove that LONGPATH is NP-complete.

The first step in determining if this problem is NP-Complete is to make sure that you can verify a given certificate for the problem in polynomial time. The information we need to do this verification would be the certificate or solution itself and a set of edges for G. We would need to verify that the edges are in the graph as well as sum up the edges to see if they match the solution k. This can be done in O(E) time (meaning it is based on the number of edges) and therefor it is polynomial. This proves that the LONG PATH problem is NP The next step would be to find a problem that can be reduced into the LONG PATH problem to prove that the problem is NP-Complete. From the problem above we know that the HAM-PATH problem is NP-Complete and from the chart in the lecture we know that the HAM-PATH problem can be reduced into the LONG PATH problem. The HAM-PATH problem we are looking to find the longest path without repeating an edge. The difference between this and the LONG PATH problem is that the LONG PATH problem is looking to find a path of at least length k. If we assume a connected graph, we can set all the edge weights in the graph to be equal to 1. Since each edge will only contribute one, we need k to be set equal to the number of all the vertices- 1. Using the LONG PATH problem we can now see if there is a PATH of at least length k. If there is this would indicate that the solution is also a HAM-PATH and therefor shows that HAM-PATH can be reduced to LONG PATH and solved by LONG PATH. This proving that LONG PATH is NPComplete.

<u>Reduce K-Color to Course time</u>

Given graph G and positive int k, produce a course assignment problem whos courses are the verts of G and with K available time slots. Include, in the problem, a student for each edge of G. The student associated with edge (u,v) wants to take course u and v and nothing more.
Suppose that G can be colored with K colors. Get K-coloring of G. Assign time slots to the courses by following the coloring. If vertex v is colored by color m then use time slot m for course v. Since the coloring does not color any two adjacent verts the same color there can be no time conflicts
Suppose that courses can be assigned time slots so that there are no conflicts. Then assign colors to G in the same way, using the m-th color for vertex v if course v received the m-th time slot. Since there is a student for each edge, and none of the students have time slot conflicts, this coloring must avoid coloring two adj verts the same color.

Consider the following game. A “dealer” produces a sequence s1…sn of cards, face up, where each card si has a value vi. Then two players take turns picking a card from sequence, but can only pick the first or last card of the remaining sequence. The goal is to collect cards of the largest value. Assume n is even.

(a) Show a sequence of cards such that it is not optimal for the first player to start by picking up the available card of larger value. That is, the greedy strat is suboptimal.
(5,100,1,1)

(b) Give an O(n^2) algo to compute an optimal strat for the first player. Given the initial sequence, your algorithm should precompute in O(n^2) time some information and then the first player should be able to make each move optimally in O(1) time by looking up the precomputed information. Let OPT(i,j) be the difference between: the largest total the first player can obtain and the corresponding score of the second player when the playing on the sequence si….sj
At any stage of the game, there are 2 possible moves for the player. Either choose the first fard, in which case he will gain si and score -OPT(i+1, j) in the rest of the game or the last card, gaining sj and -OPT(i,j-1) from the remaining cards.
Because we are interested in the largest total the first player can gain over the second, we take the max of these two values
OPT(i,j) = max(st – OPT(i+1,j), sj-OPT(I,j-1)}
And the base cases are for all I in the set of {1…n}, OPT(i,i) = st.

```
for i=i…n:
  OPT[i,i] = s[i]
for j = 1…n:
  for i = j…1:
    OPT[i,j] = max(s[i] – OPT[i+1, j), s[j] – OPT[i,j-1])
Return OPT[i,n]
```

Recude 3-COLOR to 4-COLOR

Let G=(V,E) be an instance of 3-COLOR transform G to G’ by adding a new vertex w’ that is connect t every other vertex. That is G’=(V’,E’) where V’=V union{w’} and E’=E union {(w’, u) for all u in G}
If G has a -COLORing then G’ has a 4-COLORing. Assume G has a 3-COLORing then there exists a function c:V->{1,2,3} such that for all u, w in V if (u,w) in E then c(u) != c(w). Now define the 4-COLORing function c’ for G’
c’(u) = c(u) if u is in V
c’(u) = 4 if u not in V(u=w’)
Therefor if there is a 3-COLORING in G then there is a 4-COLORing in G’
IF G’ has a 4-COLORing then G has a 3-COLORing. Assume G’ has a 4-COLORing. Since w’ is adj to all other verts in G’ then w’ must be a different color. Let c’ be the coloring function for G’, without loss of generality we can say that c’(w’)=4 and c(u) != 4 for all u in (V’-{w’}). However, (V’={w’}) = V. So we have colored all of the original verts in V using only colors 1,2,3 proving that G is 3-COLORable. Thus the 4-COLOR problem is NP-hard

Two well-known NP-complete problems are 3-SAT and TSP, the traveling salesman problem. The 2-SAT problem is a SAT variant in which each clause contains at most two literals. 2-SAT is known to have a polynomial-time algorithm. Is each of the following statements true or false? Justify your answer.

a. 3-SAT ≤p TSP.
This one is true. They are both NP-complete problems so one can be reduced to the other. There may be a couple steps along the way but it can be done. Going based off of the lectures, 3-SAT can be reduced to DIR-HAM-CYCLE to HAM-CYCLE which can then be reduced to TSP. The chart in the lecture slides displays this information.

b. If P ≠ NP, then 3-SAT ≤p 2-SAT.
False. There is an existing polynomial time algorithm for 2-SAT, but 3-SAT is NP-Complete. If P != NP then there is no polynomial time algorithm that exists for NP-Complete problems such as 3-SAT. If 3-SAT was reducable to 2-SAT which we already know has a polynomial algorithm, then 3-SAT would also have a polynomial algorithm because the algorithm that 3-SAT is reducing to must be equal or harder than the 3-SAT. This is a contradiction that there is not a polynomial algorithm for 3-SAT.

c. If P ≠ NP, then no NP-complete problem can be solved in polynomial time.
True. If a single NP-Complete problem could be solved in Polynomial time then ALL NPComplete problems could be solved in Polynomial time. If all NP-Complete problems could be solved in polynomial time we would have, by definition, P=NP. So it follows that since we have P!=NP by definition there are no NP-Complete problems that can be solved in polynomial time.

<u>Reduce 3-SAT to 4-SAT</u>

To prove that 4-SAT is NP-hard, we reduce 3-SAT to 4-SAT as follows. Let phi denote an instance of 3-SAT. We convert phi to a 4-SAT instance phi’ by turning each clause (x V y V z) in phi to (x V y V z V h) and (x V y V z V ~h), where h is a new variable. Clearly this is polynormal-time doable.
If a given clause (x V y V z) is satisfied by a truth assignment then (x V y V z V h) and (x V y V z V ~h) is satisfied by the same truth assignment with h arbitrarily set. Thus if phi is satisfiable, phi’ is
Suppose phi’ is satisfied by truth assignment T. Then (x V y V z V h) and (x V y V z V ~h) must be true under T. As h and ~h assume different truth values, (x V y V z) must be true under T as well. This phi is satisfiable.

<u>Reduce 3-SAT to IP-D</u>

Any 3-SAT instance has Bool variables and clauses. Our int programming problem will have twice as many variables, one for each variable and its compliment as well as the following inequalities:
f(x1, ~x1…) =sum xi + sum ~xi
0 <= xi <=1 and 0<=~xi<=1 for i=1…n
xi + ~xi <=1 and xi + ~xi >=1
K=n and for each caluse C=(xs V xt V xi) and xs + xt + xi >=1
In any 3-SAT solution, a TRUE literal corresponds to a 1 in IP-D. If the express is SATISFIED in 3-SAT, at least one literal per clause is TRUE so inequality sum is >=1 in IP-D
Given a solution to this IP-D instance all variables will be 0 or 1. Set the literals corresponding to 1 as TRUE and 0 as FALSE. No bool variable and its complement will both be true, so it is a legal assignment and will satisfy the clauses

Consider a connected weighted directed graph G=(V,E,w). Define the fatness of a path P to be the maximum weight of any edge in P. Give an efficient algorithm that, given such a graph and two verticies u, v in V, find the minimum possible fatness of a path from u to v in G.

We can see that the fatness must be the weight of one of the edges, so we sort all edge weights and perform a binary search. To test whether or not a path with a fatness no more than x exists, we perform a breadth-first search that only walks edges with weight less than or equal to x. If we reach v, such a path exists. If such a path exists, we recurse on the lower part of the range we are searching, to see if a tighter fatness bound also applies. If such a path does not exist, we recurse on the supper part of the range we are searching to relax that bound. When we find two neighboring values one of which works and one of which doesn’t we have our answer. This takes O((V+E)lgE) time.

Another good solution is to modify Dijkstra’s algorithm. We use “fatness” instead of the sum of edge weights to score paths, and the only change to Dijkstra itself that is necessary is to change the relaxation operation so that it compares the destination node’s existing min-fatness with the max of the weight of the incoming edge(i,j) and the min-fatness of any path to i (the source of the incoming edge) The correctness argument is almost precisely the same as that for Dijkstra’s algorithm. A correct solution also had to note the negative-weight edges, which could be present here and normally break Dijksta, don’t do that here. Adding negative numbers produces ever-more-negative path weights, but taking their max doesn’t. This solution has the same time complexity as Dijkstra.