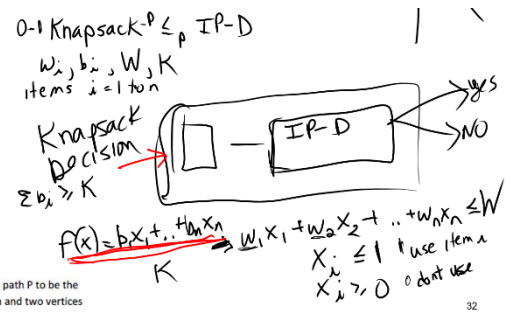
We must show that:

a. Any 3-SAT solution gives a solution in IP-D.

In any 3-SAT solution, a TRUE literal corresponds to a 1 in IP-D. If the expression is SATISFIED in 3-SAT, at least one literal per clause is TRUE, so the inequality sum is ≥ 1 in IP-D.

b. Any IP-D solution gives a 3-SAT solution.

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 Boolean variable and its complement will both be true, so it is a legal assignment and will satisfy the clauses.



32

a) Greedy because in this case we have a knapsack problem, but a fractional knapsack where the number of points/time needed. So to maximize this, greedy would be best, in order to achieve the most points. A locally optimal solutional is also globally optimal and by always picking the problem that will earn him the most points the fastest, greedy is best.

b)

pseudocode

calculate the points / time per problem

sort the points/time by descending order, most points per less time to least points for most time

Iterate through sorted list

subtract the time of the problem from the total tiem T and add the points for hte problem

if there is any remaining time left and you can't get full credit on the problem

then schedule the the highest points/time left to get partial credit.

c) The running time of this algorithm is O(nlgn). This is because the for loop goes through n iterations thus having O(n). But sorting takes O(nlgn). nlgn is of higher order so the overall runnign time would be O(n lg n)

6. 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 vertices u,v ∈V, finds the minimum possible fatness of a path from u to v in G.

Solution: There are two good solutions to this problem.

We can see that that 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 upper 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 $O((V + E) \lg E)$ 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 that negative-weight edges, which could be present here and normally break Dijkstra, 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's algorithm.

Two less-efficient solutions were to use Bellman-Ford instead of Dijkstra (with the same modified relaxation step and an analagous correctnes argument) and to perform the iterative search linearly instead of in a binary fashion. These recieved partial credit.

```
# A Dynamic Programming based Python Program for 0-1 Knapsack problem
# Returns the maximum value that can be put in a knapsack of capacity W
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W+1)] for x in range(n+1)]

    # Build table K[][] in bottom up manner
    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]

# Driver program to test above function
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print(knapSack(W, wt, val, n))

# This code is contributed by Bhavya Jain
```

modified mergesort question
about complexcity theta(n^2)

(a) $P = NP$. **U**

(b) If $P \neq NP$, then the satisfiability problem (SAT) cannot be solved in polynomial time. **T**

(c) If a problem is NP-complete then it is NP-hard. **T**

(d) If there is a polynomial time reduction from the satisfiability problem (SAT) to problem X, then X is NP-hard. **T**

(e) If there is a polynomial time reduction from problem X to the satisfiability problem (SAT), then X is NP-hard. **F**

(f) No problem in NP can be solved in polynomial time. **F**

(g) If there is a polynomial time algorithm to solve problem A then A is in P. **T**

(h) If there is a polynomial time algorithm to solve problem A then A is in NP. **T**

1. Let X and Y be two decision problems. Suppose we know that X reduces to Y in polynomial time. Which of the following can we infer? Explain

a. If Y is NP-complete then so is X.
Can't infer this because X can just be NP.

b. If X is NP-complete then so is Y.
Can't infer this because Y might be just NP.

c. If Y is NP-complete and X is in NP then X is NP-complete.
Can't infer because X can be NP.

d. If X is NP-complete and Y is in NP then Y is NP-complete.
You can infer this because X reduces to Y. X is known to be complete. And Y is in NP so it proves Y is NP complete.

e. X and Y can't both be NP-complete.
Can't infer this.

f. If X is in P, then Y is in P.
Can't infer this because X reduces to Y and if X is in P, this means that Y is at least as hard as X so you can't say that Y is in P.

g. If Y is in P, then X is in P.
You can infer this because since X reduces to Y, and Y is in P, then we know that X is at least as hard as Y, so X is in P.

put:

```
20
```

ıe Complexity: O(nW) where n is the number of items and W is the capacity of knapsack.

CS 325 - Homework 5
Philip Chang

No because while COMPOSITE IS NP, we don't know if it is also NP complete, so we don't know for sure that P = NP.

d. If P ≠ NP, then ***no*** problem in NP can be solved in polynomial time.
No, because if P≠ NP, only proves that NP Complete problems can't be solved in polynomial time.

2. 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, because SUMSET-SUM is NP complete, meaning that it can be reduced to any other NP complete problem. However, we only know that COMPOSITE so we cannot definitively say 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, because SUBSET SUM is NP complete, this implies P = NP, so that since COMPOSITE is NP then there should be a polynomial time algorithm for COMPOSITE too.

c. If there is a polynomial algorithm for COMPOSITE, then P = NP.

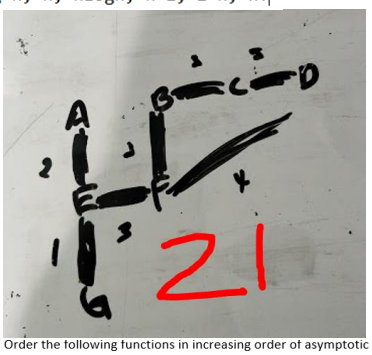
3. Two well-known NP-complete problems are 3-SAT and TSP, the traveling salesman problem. The 2-SAT problem is a SAT variant in which each clause contains at most two literals. 2-SAT is known to have a polynomial-time algorithm. Is each of the following statements true or false? Justify your answer.

a. 3-SAT ≤p TSP.
True because they are both NP complete problems. 3SAT can be reduced from Circuit Sat to 3 Sat. While TSP can be reduced from 3 sat to DIR-HAM-CYCLE to HAMCYCLE to TSP. They can be reduced each other according to the chart on SLIDE 42 so it's true. In addition, if they are both NP complete then they are NP.

b. If P ≠ NP, then 3-SAT ≤p 2-SAT.
False because P!= NP means there is no polynomial time algorithm for 3SAT. But 2SAT is in P, so if you could reduce 3SAT to 2SAT 3sat would also be it would also be in P, so it contradicts this, so it's false.

c. If P ≠ NP, then no NP-complete problem can be solved in polynomial time.
True, because if NP complete means that if an NP complete problem can be solved in polynomial time, then all NP complete problems can be solved in polynomial time and by definition P=NP is NP complete, so if P != NP, then by definition no NP complete problems can be solved in polynomial time.

1,log n, n, nlogn, n^2, 2^n, n!



Order the following functions in increasing order of asymptotic (big-O) complexity.

$$f(n) = 2^{2^{1000}}, \quad g(n) = \sum_{i=1}^{n}(i+1), \quad h(n) = 2^n, \quad p(n) = 10^{10}n, \quad q(x) = n2^{n/2}$$

lution:

f(n), p(n), g(n) q(n), h(n)

2. Product Mix: **(10 points)** 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, where profit per tie = selling price - labor cost – material cost. Labor cost is $0.75 per tie for all four types of ties. The material requirements and costs are given below.

Material	Cost per yard	Yards available per month
Silk	$20	1,000
Polyester	$6	2,000
Cotton	$9	1,250

	Type of Tie			
Product Information	**Silk**	**Polyester**	**Blend 1**	**Blend 2**
Selling Price per tie	$6.70	$3.55	$4.31	$4.81
Monthly Minimum units	6,000	10,000	13,000	6,000
Monthly Maximum units	7,000	14,000	16,000	8,500

Material Information in yards	Type of Tie			
	Silk	**Polyester**	**Blend 1 (50/50)**	**Blend 2 (30/70)**
Silk	0.125	0	0	0
Polyester	0	0.08	0.05	0.03
Cotton	0	0	0.05	0.07

Formulate the problem as a linear program with an objective function and all constraints.

Profit per tie = selling price - labor cost – material cost.

Profit for 1 Silk Tie = 6.70 – (.125 * 20) - .75 = 3.45

Profit for 1 Polyester Tie = 3.55 – (.08 * 6) - .75 = 2.32

Profit for 1 Blend 1 Tie = 4.31 – (.05 * 6 + .05 *9) - .75 = 2.81

Profit for 1 Blend 2 Tie = 4.81 – (.03 * 6 + .07 * 9) - .75 = 3.25

x1 = silk, x2 = Polyester, x3 = Blend 1, x4 = Blend 2

Objective Function and Constraints

Maximize profit = 3.45x1 + 2.32x2 + 2.81x3 + 3.25x4

Subject To

0.125x1 <= 1000

.08x2 + .05x3 + .03x4 <= 2000

.05x3 + .07x4 <= 1250

x1 >= 6000

x1 <= 7000

x2 >= 10000

x2 <= 14000

x3 >= 13000

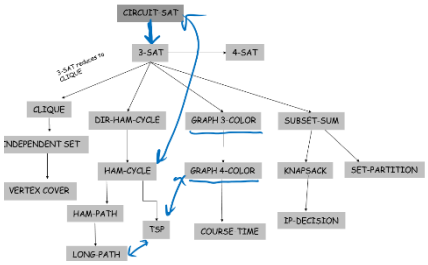
x3 <= 16000

x4 >= 6000

x4 <= 8500

LPCODE

```
max 3.45x1 + 2.32x2 + 2.81x3 + 3.25x4
ST
    .125x1 <= 1000
    .08x2 + .05x3 + .03x4 <= 2000
    .05x3 + .07x4 <= 1250
    x1 >= 6000
    x1 <= 7000
    x2 >= 10000
    x2 <= 14000
    x3 >= 13000
    x3 <= 16000
    x4 >= 6000
    x4 <= 8500
    x1 >= 0
    x2 >= 0
    x3 >= 0
    x4 >= 0
```



O: less than (worst case) - asymptotic upper bound
$f(n) = O(g(n))$ is like $a \le b$

Ω: greater than (best case) - asymptotic lower bound
$f(n) = \Omega(g(n))$ is like $a \ge b$

Θ: equal to (tight bound) - asymptotic upper and lower bounds
$f(n) = \Theta(g(n))$ is like $a = b$

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

12. Prove or disprove the following $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.

***DISPROVE:** Many counter examples. Let $f(n) = n^3$ and $g(n) = n$. $\min(f(n), g(n)) = n$, n^3+n is not $\Theta(n)$.*

Prove or disprove $f(n)+g(n) = O(\max(f(n), g(n)))$
$0 \le f(n)+g(n) \le c \max(f(n), g(n))$
$f(n)+g(n) \le 2\max(f(n), g(n))$
since $f(n) \le \max$ and $g(n) \le \max$
let $c = 2$ let $n_0 = 1$
$0 \le f(n)+g(n) \le c\max(f(n), g(n))$ for $n \ge n_0$

Exs. Taking to the limit of f(n)/(g(n).
f(n)=$n^{0.25}$; g(n)=$n^{0.5}$;O(g(n))
f(n)=n; g(n) = $\log^2 n$; Ω(g(n));
f(n)= log n; g(n)= ln n; Θ(g(n)).
f(n) = $1000n^2$; g(n) = $0.0002n^2 - 1000n$
lower terms not significant so **theta**
f(n) = nlog n; g(n) =n O(g(n))
f(n) = e^n; g(n) = 3^n O(g(n))
f(n) = 2^n; g(n) = 2^{n+1} ; theta, logs have same growth
f(n) = 2^n; g(n) =2^{2n} O(g(n))
f(n) = 2^n; g(n) = n! O(g(n))
f(n) = lgn;g(n) = √n O(g(n))

Spiral Recurrence
M(n) = M(n-2) + n + n + n + n-1 + 1
= M(n-2) + 4n | a = 1 b = 2 d = 1
f(n)=4n = theta(n) Case 2
theta(n^2)

merge(nlogn) BI TER Quat log n

```
def binary_search(item_list,item):
    first = 0
    last = len(item_list)-1
    found = False
    while( first<=last and not found):
        mid = (first + last)//2
        if item_list[mid] == item :
            found = True
        else:
            if item < item_list[mid]:
                last = mid - 1
            else:
                first = mid + 1
    return found
```

```
def ternary_search(int l,int r, int x):
    if(r>=l):
        int mid1 = l + (r-l)/3;
        int mid2 = r -  (r-l)/3;
        if(ar[mid1] == x)
            return mid1;
        if(ar[mid2] == x)
            return mid2;
        if(x<ar[mid1])
            return ternary_search(l,mid1-1,x);
        else if(x>ar[mid2])
            return ternary_search(mid2+1,r,x);
        else
            return ternary_search(mid1+1,mid2-1,x)
    }
    return -1;
```

Divide+conqer

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])

$$T(n) = T(2n/3) + 1 = \Theta(\log n)$$

$j \leftarrow 0$ while ($j < n$) do $j \leftarrow j+2$; $z \leftarrow z+1$	for $k \leftarrow 0$ to n do for $j \leftarrow 0$ to k do $z \leftarrow z+1$	$i \leftarrow n$ while ($i > 1$) do $i \leftarrow \lfloor i/2 \rfloor$; $z \leftarrow z+1$

Left: Θ(n), Middle: Θ(n²), Right: Θ(log₂n)

Decrease and Conquer

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

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

for some integer constants a, b > 0, d ≥ 0 .

If $f(n)$ is $O(n^d)$ then

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

Also theta

$T(n) = T(n-2) + n$
Using muster method
T(n) = a T(n-b) + f(n)
Let a = 1 b = 2, and d = 1 because f(n) = Θ(n^d) = n
If a = 1, then the bound for T(n) = Θ(n^(d+1)
By substitution, Θ(n^(1+1) = Θ(n^2)

Proof
If f1(n) = O(g(n)) and f2(n) = O(g(n)) then f1(n)= Θ (f2(n)).
Counterexample: f1(n) = n, f2(n) = n^2 and g(n) = n^3 then f1(n)≠Θ (f2(n))

Proof
If f1(n) = O(g1(n)) and f2(n) = O(g2(n)) then f1(n)+ f2(n)= O(max{g1(n), g2(n)}) By definition there exists a c1, c2, n1, n2 > 0 such that f1(n) ≤ c1g1(n)) for n ≥ n1 and f2(n) ≤ c2g2(n) for n ≥ n2 Since the functions are asymptotically positive f1(n)+ f2(n) ≤ c1g1(n) + c2g2(n) ≤ c1 max{g1(n),g2(n)} + c2 max{g1(n),g2(n)} ≤ (c1 +c2)max{g1(n),g2(n)} Let k = (c1 + c2) and n0= max (n1, n2) then f1(n)- f2(n) ≤ k max{g1(n),g2(n)} for n ≥ n0; k, n0 > 0 and by definition f1(n)+ f2(n) = O(max{g1(n),g2(n)})

Quatsearch(arr, value)
If arr size = 0
Return -1 #white line after
Lo = 0
P1 = arraysize /4 – 1
P2 = array size / 2 – 1
P3 = array size * 3 / 4 – 1
Hi = arraysize – 1#whitelineafter
List = [p1, p2, p3]
Try
Arr[x] for x in list.index(value)
Return 1
Except
Pass
If value < arr[p1]
Return quatsearch(arr[lo:p1], value]
Else if value < arr[p2]
Return quatsearch(arr[p1:p2], value]
Else if value<arr[p3]
Return quatsearch(arr[p2:p3],value]
Else if value < arr[hi]
Return quatsearch(arr[p3:hi], value]
Else
Return - 1

T(n) = Θ(lg n)
T(n) = T(n/4) + C

Sort in python o(n)

```
def mergeSort(alist):
    print("Splitting ",alist)
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:] #space after
        mergeSort(lefthalf)
        mergeSort(righthalf) #space after
        i=0 j=0k=0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1
        while i < len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1
        while j < len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
```

$T(n) = 2T(n/2) + \Theta(n)$
$\Theta(nLogn)$

Function FIND-ARRAY-MAX(A, n)
1: **if** (n = 1) **then**
2: **return**(A[1])
3: **else**
4: **return**(max(A[n], FIND-ARRAY-MAX (A, n − 1)))
5: **end if**

T(n) = T(n-1) + c Θ(n)

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:

regularity

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

$$T(n) = 2T\left(\frac{n}{8}\right) + 4n^2$$

using master method
T(n) = a * T(n/b) + f(n)
Let a = 2, b = 8, which is equal to n^(log 2 base 8)
or (n^1/3), and f(n) = 4n^2
F(n) = Ω(n^(1/3)
Therefore T(n) = Θ(n^2)

Ex. $T(n) = 4T(n/2) + n^2$
$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^2.$
CASE 2: $f(n) = \Theta(n^2)$
$\therefore T(n) = \Theta(n^2 \lg n)$.

$(0.5)^n$, $\log(n^3)$, log(n!), nlog(n), $10000n^2$, 2^n

Foo(n) {
total = 0
if n = 1 return 2
else {
total = Foo(n/4) + Foo(n/4)
for i = 1 to n do
for k = 1 to n do
total = total + k
return total }}

T(n) = 2 T(n/4) + c n2
Case 3 of the master method
Check Regularity
2f(n/4) = 2(n/4)2 = 1/8n2 ≤ cn2 if c = 0.5
regularity condition holds, so T(n) = Θ(n2)

Algorithm *prefixAverages1(X, n)*
Input array X of n integers
Output array A of prefix averages of X
A ← new array of n integers
for i ← 0 to n − 1 do
s ← X[0]
for j ← 1 to i do
s ← s + X[j]
A[i] ← s / (i + 1)
return A

Θ(n²)

Algorithm *prefixAverages2(X, n)*
Input array X of n integers
Output array A of prefix averages of X
A ← new array of n integers
s ← 0
for i ← 0 to n − 1 do
s ← s + X[i]
A[i] ← s / (i + 1)
return A

Algorithm prefixAverages2 runs in Θ(n) time

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 = 1 to n
if (X i == Y j)
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 m*n runtime

Memoized DP Algorithm
Memo = {}
Put it into a dictionary
Fib(n):
If n in memo: return memo[n]
If n <= 2: f = 1
Else: f = fib(n1) + f(n-2)
Memo[n] = f
Return f

O(n)

DP minCoins(int A, V_n)
numCoins = A // V_n
remainder = A % V_n
if remainder == 0
return numCoins
else
numCoins += minCoins(remainder, V_{n-1});
return numCoins

O(n)

GREEDY-ACTIVITY-SELECTOR(s, f)
1 n ← length[s]
2 A ← {a₁}
3 i ← 1
4 for m ← 2 to n
5 do if $s_m \ge f_i$
6 then A ← A ∪ {a_m}
7 i ← m
8 return A

Majority(array)
If array size == 1
Return array[0]
Mid = floor(array size / 2)
Left = majority(arr[0:k])
Right = majority(k:)

If left == right
Return left
Left count = freq(array, left)
Right Count = freq(array, right)

If left count > floor(arraysize/2)
Return left
Else if right > floor(arraysize/2)
Return right
Else
Print "no majority"
Return 0
T(n) = 2*T(n/2) + 2n Θ((n)

Def enumeration(arr)
If length of array is 0
Return 0
Else
Set the max sum to zero
For i in range of 0 to array length
For j in range of i to array length
Set a temp tohold current sum and initialize to 0
For k in range of i to j + 1
Set temp to equal array value at location k
If the temporary variable is bigger than the max sum
Max sum is temporary variable
Max subarray is the array from i to j+1
Return subarray and max sum O(n^3)

Def better_enumeration()
If array size is greater than 0, set max sum and subarray to first element
For i in range 0 to array length
For j in range i to array length
If i = j add sum and subarray to dictionary
Else add arr[j] to stored total sum of arr[i,j-1], add to dictionary
If current sum > max sum, store new max sum and array
Return max subarray and sum O(n^2)

10. You are going on another long trip (this time your headlights are working). You start on the road at mile post 0. Along the way there are n hotels, at mile posts $a_1 < a_2 < ... < a_n$, where each a_i 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 a_n), 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 the 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 the minimum penalty for the optimal sequence of hotels at which to stop.

Let S[j] be the minimum 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- $(a_j - a_i)$)² , S[j])
Return: S[n]

DP

14. You just started a consulting business where you collect a fee for completing various types of projects (the fee is different for each project). 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 receive your fee. There is a set of n projects $p_1, p_2, ..., p_n$ each with a duration $d_1, d_2, ..., d_n$ (in days) and you receive the fee $f_1, f_2, ..., f_n$ (in dollars) associated with it. That is project p_i takes d_i days and you collect f_i dollars after it is completed.

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 projects to complete that will maximize the total fees you earn in D days.

a) What type of algorithm would you use to solve this problem? Divide and Conquer, Greedy or **Dynamic Programming. Why? It is similar to the 0-1 knapsack.**

b) Describe the algorithm verbally. If you select a DP algorithm give the formula used to fill the table or array.

Let OPT(i,d) be the maximum fee collected for considering projects 1,…,i with d days available.
The base cases are OPT(i, 0) = 0 for i = 1, …, n and OPT(0, d) = 0 for d = 1, …, D.
For i = 1 to n {
For j = 1 to D {
If (di > j) {
OPT(i,j) = OPT(i-1,j) //updated from d to j
Else {
OPT(i,j) = max (OPT(i-1,j), // not enough time to complete project i
OPT(i-1, j-di) + fi) // Don't complete project i / Complete project i and earn fee fi
}

O(nD) or Θ(nD)

A is the array you're trying to fill ordered by finish time
a1 is the starting value since you already have the first value
inside you do a for loop from the 2nd activity onwards the activities are inside s, start time, f is the finish time you look at the start times if they're larger than the last finish time then you set it inside the set A then you set the new finishing time to m

Def linearFindSubMax(arr):
Set the array length to n
Initialize the leftIndex, rightIndex, and currentIndex to 0
Set the maxSum to lowest possible integer
Initialize the tempSum to 0
Iterate through the array from 0 to n
If the array index >= the sum of tempSum + arr[i]
tempSum = arr[i]
Else if arr index <= the sum of tempSum + arr[i]
tempSum = tempSum+arr[i]
If tempSum > maxSum
Set the rightIndex to i
Set the leftIndex to currentIndex
maxSum = tempSum
Else if tempSum equals the arr[i]
currentIndex = i
Return max,left, right+1 O(n)

Def maxArraySub(arr,left,right)
If leftIndex is equal to rightIndex
Then set the array to the left index
middle index = (left+right)/2
Get the left max
Get the right max
Return the max of the max subarrays or total array
Def maxSubArrayHelper(arr,left,middle,right)
Set the left subArray max as the lowest possible integer
Set the leftIndex = middle
Initialize tempSum, runningSum to 0
For i in range decreasing from range middle to the beginning of the array
Sum the array
If tempSum > leftSum
Set the leftSum to tempSum
Set the left index to i
Set the right subArray max as the lowest possible integer
Set the rightIndex = middle+1
Reset the tempSum, runningSum to 0
For i in range iterate through the array from middle+1 to the end of the array
Sum the array
If tempSum > rightSum
Set the rightSumto tempSum
Set the right index to i
Set the total = leftSum+rightSum
return the total, leftindex, rightindex T(n) = 2T(n/2) + n. Θ(n lg n)

Longest Palindromic Subsequence (LPS)
Similar to LCS, just work in from either end of 1 set

Knapsack problem
Given a set of items, each with a weight and benefit, pack a knapsack with a subset of items to achieve maximum total benefit
-0-1 problem, use DP, items are indivisible; "fractional knapsack": use greedy, items are divisible and can take any portion O(nlgn) runtime
-Subproblem is to computer B[k,w] k = set of items , w = exact weight of set
-O(nW) runtime, pseudo-polynomial

Greedy Algorithms
-Greedy-choice property: global optimal solution can be arrived at by making locally optimal (greedy) choices
-Optimal substructure: an optimal solution to the problem contains within it optimal solutions to subproblems
-At each phase you take the best you can get right now, hope that by choosing a local optimum at each step you arrive at a global optimum. Uses a variety of possible selection criteria

Scheduling Problems
-Many variations: activity (maximize number of activities), machine/task (minimize number of machines needed to complete all tasks with start/finish constraints), job (minimize the total time it takes to complete all jobs on a set of machines, etc.
-Activity scheduling: earliest finish greedy strategy to maximize number of activities

DP Strategy
1.Characterize structure of an optimal solution
2. Define value of optimal solution recursively
3. Compute optimal solution values bottom-up in a table (or memorize)
4. Construct an optimal solution from computed values
Used for **optimization problems, minimum/maximum value**
-Optimal substructure: an optimal solution contains optimal solutions to subproblems, optimal solution to the entire problem is build in a bottom up manner from optimal solutions to subproblems
-Overlapping subproblems: if recursive algo revisits same subproblems -> overlapping subproblems

Longest Common Subsequence (LCS)
Have 2 sets, compare using prefixes. (e.g. X_i = <x_1...x_i>)
Make a table c, c[i,j] = the length of an LCS of the sequences
1. Case 1, x_i = y_j, c[i,j] = c[i-1,j-1] + 1. Append x_i = y_j to the LCS of X_{i-1} and Y_{j-1}
2. Case 2, $x_i \ne y_j$, solve 2 problems and take max.
c[i,j] = max {c[i-1,j], c[i,j-1]}
3. Base case, 0 if i = 0 or j = 0
First fill out table, i=j=0, if either i or j is 0 then the table value for that is 0 (empty strings have no LCS)
O(ij) run time, each c[i,j] is calculated in constant time and there are i*j elements in the array

Algorithm FKnapsack(S, W)
Input: set S of items w/ benefit bi and weight w i ; max. weight W
Output: amount x i of each item i to maximize benefit with weight at most W
for each item i in S
x i =0
v i = b i / w i {value}
w = 0 {current total weight}
while w < W
remove item i with highest v i
x i = min{w i , W − w}
w = w + min{w i , W - w}
nlogn runtime

10. (8 points) Scheduling jobs with early complete dates and bonuses:

(a) (2 point) What type of algorithm would you use to solve this problem? Greedy

(b)) (4 points) Describe the algorithm verbally with enough detail to obtain the running time. If you select a DP algorithm give the formula used to fill the table or array.

Note since there are n days it will take at most n days

1. Sort jobs by bonuses in decreasing order. You will select the biggest bonus first
2. For i = 1 to n. Select the next job j_i to be scheduled according to the sorted bonus order. There are two outcomes
a) The job can be completed early and you get the bonus Schedule the job j_i as close to the early completion date e_i as possible without exceeding the date. You get the bonus b_i
b) The job cannot be scheduled by the early completion date. If there are no days left on or before e_i to schedule job j, 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)) (2 points) What is the running time of your algorithm? Explain.
1. The sorting takes Θ(nlgn)
2. To find the spot in the schedule can take in worst case 1+2+…+ n = Θ(n²) depending on the data structure used.

(a) (4 points.) Describe in words an efficient algorithm that Benny could use to maximize his total points. Why does your algorithm gives the optimal answer. (if DP -2)

Note: This is similar to the fractional knapsack problem and a greedy method will be optimal.

Benny will first calculate the points per minute of each problem, that is the ratio p/t, and then sort the p/t ratio in descending order.

Benny starts with the problem that has the greatest ratio of points to time. He will then solve the problems in this 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 that final question.

This gives the correct answer because the problem exhibits the characteristics 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 total points.

b) (2 points) Θ(nlgn) or Θ(nlogn), in other words the greatest factor on the running time is sorting the problems, after that it is a linear pass through to then see how many can be completed in time.

c) (2 points) No it would not, in this case the greedy choice is not always optimal. This would then be more similar to the Knapsack 0-1 problem. The why can be a counter example or the comparison to the 0-1 knapsack

For example that the student only has 15 minutes, and there are 3 problems as shown below

10 minutes - 10points
9 minutes - 6 points
6 minutes - 5 points

In this case the greedy strategy holds as he would get partial credit on the 6 minute problem still, but if there is no partial credit we see that by completing the 10 minute problem with the greatest ration of points per minute he earns only 10 points, but with a dynamic programming algorithm we could find that solving the 9 minute and 6 minute problems yields 11 points.

```
# A Dynamic Programming based Python Program for 0-1 Knapsack problem
# Returns the maximum value that can be put in a knapsack of capacity W
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W+1)] for x in range(n+1)]

    # Build table K[][] in bottom up manner
    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)

Scheduling jobs intervals with penalties:
For each 1 ≤ i ≤ n job j_i is given by two numbers d_i and p_i, where d_i is the deadline and p_i is the penalty. The length of each job is equal to 1 minute. We want to schedule all jobs, but only one job can run at any given time. If job i does not complete on or before its deadline d_i, we should pay its penalty p_i. Design a greedy algorithm to find a schedule which minimizes the sum of penalties.

Job 1 $d_1=1$ $p_1 = 2$
Job 2 $d_2=1$ $p_2 = 5$
Job 3 $d_3=2$ $p_3 = 2$
sort by penalty θ(nlgn)
n Jobs n minutes O(n²) look for a position for job with next highest penalty

```
# lis returns length of the longest increasing subsequence
# in arr of size n
def lis(arr):
    n = len(arr)

    # Declare the list (array) for LIS and initialize LIS
    # values for all indexes
    lis = [1]*n

    # Compute optimized LIS values in bottom up manner
    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

    # Initialize maximum to 0 to get the maximum of all
    # LIS
    maximum = 0

    # Pick maximum of all LIS values
    for i in range(n):
        maximum = max(maximum , lis[i])

    return maximum
# end of lis function
```

nlogn

Pseudocode - Algo #3 - Dynamic Programming
Def changedp(denominations, change)
If no denominations or amount is zero
Return empty array, 0
Create 3 empty arrays, min coins, current, and final
Initialize min coins to hold 0
Initialize current to hold 0
Populate the final array with 0 for range to # of denominations
Create sub problems with for loop in range of 1 to change + 1
Current total is the current iteration
Current coin is the last denomination
For loop for each denomination < current total
If mincoins[iteration - denomination] + 1< the total for current
Set current total to mincoins[iteration - denomination] + 1
Set current coin to current denomination
Add the current total to min coins, and the current coins to current
Set new variable to equal change
While variable > 0
Get denomination used to make change at the wanted change value
Count how many coins were used at this denomination in final array
Subtract the value of the denomination from the new variable
Return final array, min coins O(n^A).

Pseudocode - Algo #1 - Brute Force
A = change needed
V = coin list
def changeslow(A,V,total = 1):
Create 2 empty lists arr, temp
Populate arr with 0 for every item in V
If A is 0, return the arr and 0 as minimum coins needed
If A is one of the coin list items, increment that coin index in arr, and return the arr and total
Populate temp arr with the numbers in V that are lower than or equal to A
Set the arr[0] to A (used for comparison of the cum)
Iterate through temp
recursively get the minimum amount of coins per item in temp
Increase the item that is being iterated
If the current Arr total is greater than the temp arr (more coins)
Set the current arr to the temp arr
Set the current total to the new arr
Return the arr and total. Therefore O(V * temp) * O(A) = O(n*n) * O(A) = O(n^2 * A).

Pseudocode - Algo #2 - Greedy
Use for loop to iterate through denominations array in descending order
While current denomination is less than or equal to change goal
Change goal -= denomination
Coin result array[current denomination] += 1
Min coins += 1 O(n).

```
# Dynamic Programming Python implementation of Coin Change problem
def count(S, m, n):
    # We need n+1 rows as the table is constructed in bottom up
    # manner using the base case 0 value case (n = 0)
    table = [[0 for x in range(m)] for x in range(n+1)]

    # Fill the entries for 0 value case (n = 0)
    for i in range(m):
        table[0][i] = 1

    # Fill rest of the table entries in bottom up manner
    for i in range(1, n+1):
        for j in range(m):
            # Count of solutions including S[j]
            x = table[i - S[j]][j] if i-S[j] >= 0 else 0

            # Count of solutions excluding S[j]
            y = table[i][j-1] if j >= 1 else 0

            # total count
            table[i][j] = x + y

    return table[n][m-1]
```

O(mn)

```
def DPmake_change(coin_val_list, change, known_results):
    min_coins = change
    if change in coin_val_list:
        known_results[change] = 1
        return 1
    elif known_results[change] > 0:
        return known_results[change]
    else:
        for i in [c for c in coin_val_list if c <= change]:
            num_coins = 1 + make_change(coin_val_list, change - i, known_results)
            if num_coins < min_coins:
                min_coins = num_coins
                known_results[change] = min_coins
    return min_coins
```

```
def panlindrome(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]
```

n^2

```
# jobsArray contains jobs objects that have the form:
# sampleJob = { id: str, deadline: int, penalty: int }
def schedule_jobs(jobsArray):
    # Sorting the jobsArray so penalties are in descending order
    sortedJobs = sorted(jobsArray, key=lambda k: k['penalty'], reverse=True)
    # Creating the jobsSchedule array with which to assign jobs within
    jobsSchedule = [0] * len(sortedJobs)
    result = [0] * len(sortedJobs)
    for i in sortedJobs:
        for j in range(0, min(len(sortedJobs), i['deadline'])):
            if jobsSchedule[j] == 0:
                result[j] = i['id']
                jobsSchedule[j] = 1
                break
    #Print the result
    for i in range(0, len(jobsSchedule)):
        if jobsSchedule[i] != 0:
            print(result[i])
```

n²+n

```
def lcs(X , Y):
    # find the length of the strings
    m = len(X)
    n = len(Y)

    # declaring the array for storing the dp values
    L = [[None]*(n+1) for i in xrange(m+1)]

    """Following steps build L[m+1][n+1] in bottom up fashion
    Note: L[i][j] contains length of LCS of X[0..i-1]
    and Y[0..j-1]"""
    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0 :
                L[i][j] = 0
            elif X[i-1] == Y[j-1]:
                L[i][j] = L[i-1][j-1]+1
            else:
                L[i][j] = max(L[i-1][j] , L[i][j-1])

    # L[m][n] contains the length of LCS of X[0..n-1] & Y[0..m-1]
    return L[m][n]
#end of function lcs
```

O(mn)

(a) $P = NP$. **U**
(b) If $P \ne NP$, then the satisfiability problem (SAT) cannot be solved in polynomial time. **T**
(c) If a problem is NP-complete then it is NP-hard. **T**
(d) If there is a polynomial time reduction from the satisfiability problem (SAT) to problem X, then X is NP-hard. **T**
(e) If there is a polynomial time reduction from problem X to the satisfiability problem (SAT), then X is NP-hard. **F**
(f) No problem in NP can be solved in polynomial time. **F**
(g) If there is a polynomial time algorithm to solve problem A then A is in P. **T**
(h) If there is a polynomial time algorithm to solve problem A then A is in NP. **T**

```
#A naive recursive implementation of 0-1 Knapsack Problem

# Returns the maximum value that can be put in a knapsack of
# capacity W
def knapSack(W , wt , val , n):

    # Base Case
    if n == 0 or W == 0 :
        return 0

    # If weight of the nth item is more than Knapsack of capacity
    # W, then this item cannot be included in the optimal solution
    if (wt[n-1] > W):
        return knapSack(W , wt , val , n-1)

    # return the maximum of two cases:
    # (1) nth item included
    # (2) not included
    else:
        return max(val[n-1] + knapSack(W-wt[n-1] , wt , val , n-1),
                   knapSack(W , wt , val , n-1))
```

```
# hotels is an array containing hotel objects with the following form:
# { 'distance': int, 'id': str }
def road_trip(hotels, daily_dist):
    hotels_slept_in = []
    dist_done = 0
    i = 0
    cur_hotel = hotels[i]
    while cur_hotel['id'] != hotels[len(hotels) - 1]['id']:
        dist_diff = dist_done + daily_dist
        while dist_diff >= 0 and i + 1 < len(hotels):
            cur_hotel = hotels[i]
            dist_done += cur_hotel['distance'] - dist_done
            dist_diff -= dist_done
            i += 1
        hotels_slept_in.append(cur_hotel)
    return hotels_slept_in
```

n^3

In the single machine scheduling problem, you are given a single machine and a collection of tasks $T = \{T_1, T_2, \ldots, T_n\}$ with respective start times $S = \{s_1, s_2, \ldots, s_n\}$ and respective finish times $F = \{f_1, f_2, \ldots, f_n\}$. Each task T_i starts at time s_i and finishes at time f_i. The machine can execute precisely one task at a time and tasks scheduled on the machine must be non-conflicting. The goal is to maximize the number of tasks scheduled on the machine. Design a greedy algorithm for this problem and argue its correctness.

Function SCHEDULE-TASK(T)
1: Order the tasks in T by their finish times.
Without loss of generality, assume that $f_1 \le f_2 \le \ldots f_n$.
3: $X_G = T_1$
4: **for** (i = 2 to n) **do**
5: **if** (T_i does not create a conflict with the tasks in X_G) **then**
6: $X_G = X_G \cup T_i$
7: **end if**
8: **end for**

X_G set of tasks to be scheduled
Sort by finishing times select earliest finishing time
Running time Sort Θ(nlgn) + selection Θ(n)
Overall Θ(nlgn)

Decision problems
– Given an input and a question regarding a problem, determine if the answer is yes or no

Optimization problems
– Find a solution with the "best" value
– Optimization problems can be casted as decision problems that are easier to study

Class P consists of (decision) problems that are solvable in polynomial time

Polynomial-time algorithms
– Worst-case running time is $O(n^k)$, for some constant k

Examples of polynomial time:
– $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
– Searching and Sorting

Sorting Θ(nlgn) - O(n²)

Class NP consists of problems that could be solved by Nondetermistic Polynomial algorithms

Or verifiable in polynomial time

To show that B is NP-Complete:
1. Show that B is in NP.
Give a polynomial time algorithm for verifying a solution.
2. Show that A ≤ₚ B for some A∈NP-Complete
Pick an instance, A, of your favorite NP-Complete problem. Show a polynomial algorithm to transform A into an instance of B
Step 2 alone shows that a problem is NP-Hard

B - decision problems

Is Integer Programming Decision (IP-D) NP-Complete?

Instance: A set *x* of integer variables, a set of inequalities over these variables, a function *f*(*x*) to maximize, and integer *K*.

Question: Does there exist an assignment of integers to X such that all inequalities are true and $f(x) \ge K$?

Example: Decision
$f(x) = 2x_2$; $K = 3$ $2x_2 \ge 3$
$x_1 + x_2 \le 3$
$x_1 \ge 1$, $x_2 \ge 0$

Optimization
max $2x_2$
ST $x_1 + x_2 \le 3$
$x_1 \ge 1$
$x_2 \ge 0$
x_1, x_2 integers

NP-Completeness Proof Method

To show that IP-D is NP-Complete:
1. Show that IP-D is in NP.
Give a polynomial time algorithm for verifying a solution.
2. Show that 3-SAT ≤ₚ IP-D or
Show that KNAPSACK ≤ₚ IP-D

Step 2 alone shows that a problem is NP-Hard

Prove IP-Dec NP-Complete?

1. IP-D is in NP – verify a candidate solution in polynomial time

Example:
$f(X) = 2x_2$; $K = 3$
$x_1 \ge 1$, $x_2 \ge 0$
$x_1 + x_2 \le 3$

To verify that x_1=1 , x_2 = 2 is a solution requires checking
$f(X) = 2(2) = 4 > 3$; $K = 3$
$1 \ge 1$, $2 \ge 0$
$1 + 2 \le 3$

To check all inequalities and objective function f(X) ≥ K is polynomial in the input size (number of variables and number of inequalities).

2. IP-D is NP-Hard

3-SAT ≤ₚ IP-D

$(x_1 \vee x_2 \vee x_3) \rightarrow x_1 + x_2 + x_3 \ge 1$
True

Any 3-SAT instance has Boolean variables and clauses. Our Integer programming problem will have twice as many variables, one for each variable and its compliment, as well as the following inequalities:

$f(x_1, \neg x_1, \ldots) = \sum x_i + \sum \neg x_i = n$

$x_1 = 1 \Rightarrow \neg x_1 = 0$
$x_1 = 0 \Rightarrow \neg x_1 = 1$

$0 \le x_i \le 1$ and $0 \le \neg x_i \le 1$ for i = 1..n

$x_i + \neg x_i \le 1$ and $x_i + \neg x_i \ge 1$ $\Rightarrow x_1 + \neg x_1 = 1$

K = n and for each clause $C = (x_s \vee x_t \vee x_r)$ add $x_s + x_t + x_r \ge 1$

This transformation is polynomial in the size of the original 3-SAT problem