## Longest Common Subsequence

Usage: DNA

Recursion

|  |
| --- |
| **def** **LCS**(n, m):  **if** 0 **in** [n,m]:  **return** 0 **elif** X[n] == Y[m]:  **return** 1+LCS(n-1, m-1) **else**:  **return** max( LCS(n-1, m), LCS(n, m-1) )` *#no need for LCS(n-1, m-1) because next level it will be called twice* *#memorization ⇒ 2d array since n and m are both changing* |

DP

|  |
| --- |
| **def** **LCS**(n,m):  dp = [0 **for** \_ **in** range(n+1)]\* (m+1) **for** i **in** range(1, n+1):  **for** j **in** range(1, m+1):  **if** X[i] == Y[j]:  dp[n][m] = 1 + dp[n-1][m-1]  **else**:  dp[n][m] = max(dp[n][m], dp[n-1][m], dp[n][m-1]) |

## Subset Sum

Recursion

|  |
| --- |
| SubsetSum(n, target):  **if** target == 0: **return** true **if** n == 0: **return** false **return** SubsetSum(n-1, target) **or**  (target - S[n] >= 0 **and** SubsetSum(n-1, target - S[n])) |

DP

|  |
| --- |
| SubsetSum(n, target):  **for** i **in** range(n+1): sub[i][0] = true **for** i **in** range(1, target): sub[0][i] = false **for** i **in** range(1, n+1):  **for** j **in** range(1, target):  sub[i][j] = sub[i-1][j] **or**  (target - S[i] >= 0 **and** sub[i-1][target-S[i]])  **return** sub[n][target] |

Ex. Knapsack hiker with the option of n items. Each item i has weight wi and benefit bi. Maximize total benefit while keeping total weight <= W.

//R → maximum benefit value that comes from the first n items with maximum weight W

|  |
| --- |
| **def** **R**(n, W):  **if** 0 **in** [n, W]: **return** 0 **if** w[n] <= W: **return** max(R(n-1, W), b[n] + R(n-1, W-w[i])) **else**: **return** R(n-1, W)  **def** **DP**(n, W, B):  **for** i **in** range(n + 1): ben[i][0] = 0 **for** i **in** range(W + 1): ben[0][i] = 0 **for** i **in** range(n +1):  **for** j **in** range(W + 1):  **if** w[i] <= W:  ben[i][j] = max(ben[i-1][j], ben[i-1][W-w[i]] + b[j])  **else**:  ben[i][j] = ben[i-1][j] |

Ex. N homework in total. Each homework has easy version (ei) and hard version(hi). Doing hi ⇒ 0 for i-1 homework. Maximize total score.

//R → maximum points to get from frist i homeworks

|  |
| --- |
| **def** **R**(i):  **if** i <= 0: **return** 0 **return** max( e[i] + R(i-1), h[i] + R(i-2) )  **def** **DP**(n):  score[0] = 0 score[1] = h[i] **if** allowed **else** e[i] **for** i **in** range(2, n + 1):  score[i] = max(e[i] + score[i-1], h[i] + score[i-2] )  *#fib*  **def** **DP**(n):  x, y = 0, h[i] **for** i **in** range(2, n):  x, y = y, max(e[i] + y, h[i] + x) |

Ex. Each person has vi of happiness. Parent of the person is not included if the person is present. Maximize happiness.

//R → maximum happiness value to get from the person and its sub

|  |
| --- |
| **def** **R**(node):  **if** node.children == null: **return** v[node] **return** max( sum(R(c) **for** c **in** node.children),  v[node] + sum(R(gc) **for** gc **in** node.grandchildren )  **def** **R**(person, canBeInvited):  **if** canBeInvited:  notInvited = (0, []) invited = (v[person], [person]) **for** c **in** person.children:  notInvited += R(c, canBeInvited) invited += R(c, **not** canBeInvited)  **if** notInvited > invited: **return** notInvited **else**: **return** invited  **else**:  notInvited = (0, [])  **for** c **in** person.children:  notInvited += R(c, **not** canBeInvited)  **return** notInvited  **def** **DP**(n):  hap[n]; **for** l **in** leafNodes.num:  hap[l] = v[l] **for** i **in** range(n-1, -1, -1):  **if** i **is** leafNode:  **continue**  hap[i] = max(  sum(hap[c] **for** c **in** node.children), v[node] + sum( hap[gc] **for** gc **in** node.grandchildren )  **return** hap[0] |

## Edit Distance

Problem: Given two strings X & Y (not necessarily of equal length), we want to convert the first string to the other by a sequence of insertions, deletions, and substitutions. The cost is the number of operations we perform.

Recursion - returns the minimal cost to convert the X[0…i] to Y[0…j]

|  |
| --- |
| **def** **R**(i, j):  **if** i == 0: **return** j **if** j == 0: **return** i **if** X[i] == Y[j]:  **return** R(i-1, j-1)  **else**:  **return** 1 + **min**(R(i-1, j-1), R(i-1, j), R(i,j-1)) |

DP = O(n\*m) = O(len(X)\*len(Y))

|  |
| --- |
| **def** **DP**():  **for** i **in** range(len(X)):  **for** j **in** range(len(Y)):  **if** X[i] == Y[j]:  dp[i][j] = dp[i-1][j-1]  **else**:  dp[i][j] = 1 + min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1])  **return** dp[-1][-1] |

## Shortest Paths with Negative Edge Weights

Problem: given directed graph with weights w\_e on edges, shortest path v to T. (NO negative circle)

What is the longest, in terms of the number of edges, that a shortest path could be?

What would it mean if a shorter path had more edges than that?

What are the shortest paths ending at T for the following graph?

Recursion - returns minimal cost of the path to v to t (limit i edges)

|  |
| --- |
| **def** **SDSP**(i, v):  **if** i == 0:  **if** (v == t) **return** 0 **else** **return** inf  **else**:  cost = SDSP(i-1, v)  **for** each w **in** adj[v]:  cost = min(cost, SDSP(i-1, w) + cost[w][v]) **return** cost |

DP = O(n\*m) = O(|V|\*|E|)

|  |
| --- |
| **def** **DP**():  fill in base cases?  **for** i **in** range(total\_number\_edges):  **for** each w **in** adj[v]:  dp[i][v] = min(dp[i-1][v], dp[i-1][w] + cost[w][v])  //better one  if newcost < dp[i-1][v]: |

sum of O((v)) = m(directed graph) = |E| for fill in all the dp[i][v]

# Discussion 1

## Complexity

NP-complete problems can be verified in polynomial time and polynomial space.

Yes answer and No answer can both be approved in polynomial time ⇒ faster

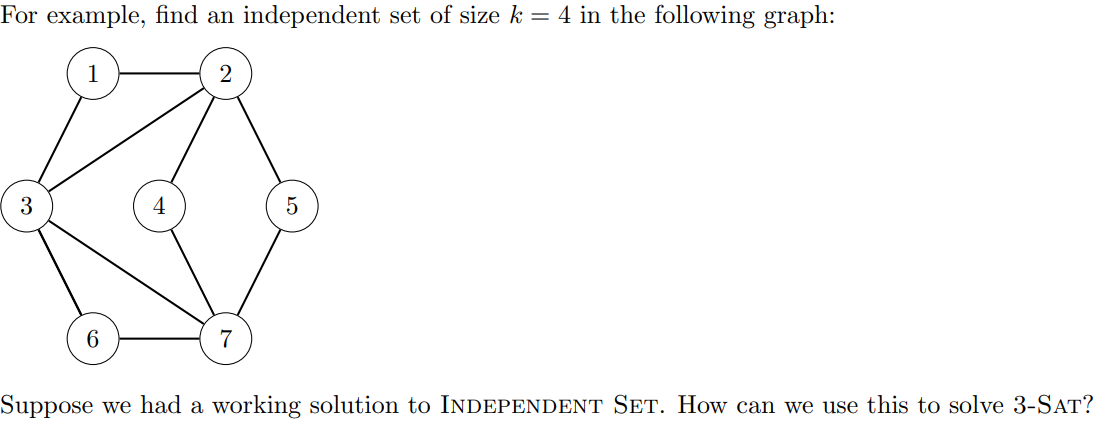
### 3-Sat

Given a set X of Boolean variables = {x1, . . . , xn}; We have k clauses, each of 3 terms, disjuncted ((x\_i1 or x\_j1 or x\_n1) and … and ((x\_ik or x\_jk or x\_nk)). Find the value of X set that k clauses are all true

### Independence Set

Given a graph G and an integer k, determine if there is some subset of the vertices V ′ , |V ′ | ≥ k such that no two vertices in V ′ share an edge

Ex.



Answer: 1, 4, 5, 6

Verifier

if then

return false

if then

return false

for all edges e = (u, v) in E:

if u in V’ and v in V’ then

return false

return true

Reduction from Independent-Set to 3-Sat

(x2 ∨ x3 ∨ x4)(x2 ∨ x3 ∨ x4)(x1 ∨ x3 ∨ x5)......

……

Connect (x1, x1)  
Connect all nodes in clauses

Independent set with k = # of clauses

|  |
| --- |
| **def** 3-**SAT**(n, k):  **for** each clauses A **and** B **and** C do  create 3 vertices A, B, C  add edge (A, B), (B, C), (A, C)  **for** each variable x\_i do  connect any x\_i node to all (**not** x\_i) nodes  **return** INDEPENDENT SET(G, k) |

false positive. Show G has Independent set of size k ⇒ 3-SAT satisfied

false negative. Show 3-SAT satisfied ⇒ G has independent set with size k

### 3-Color

Given an undirected graph G, determine whether there is a mapping of vertices in G to 3 distinct colors such that no edge is monochromatic

reduce to 3-Sat

fully connected graph with 3 vertices to represent the color (True, False, neutral)

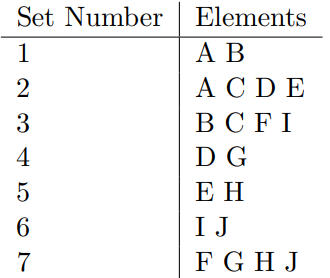
fully connected 3-vertices graph connected to neutral (x1, not x1, neutral)...

neutral

Ex. A or B or C for 3-color

…

### Set Cover problem

We are given a series of n sets and a value k, can you select no more than k sets such that every element in the n sets is in at least one element of the k chosen?

1 Select the one with maximum length. (2)

2 For every other sets M, make M = M - (2)

1 A

3 B F I

4 G

5 H

6 I J

7 F G H J

Repeat 1 and 2

Set Cover - Vertex Cover

Make a adjacent list for the vertex cover, and perform the exactly the same steps in set cover

For each vertex, define a set.

For each edge, define a distinct element and add it only to the sets that were created from its endpoints.

The resulting set of sets has a set cover of size k iff the graph had a vertex cover of size k.

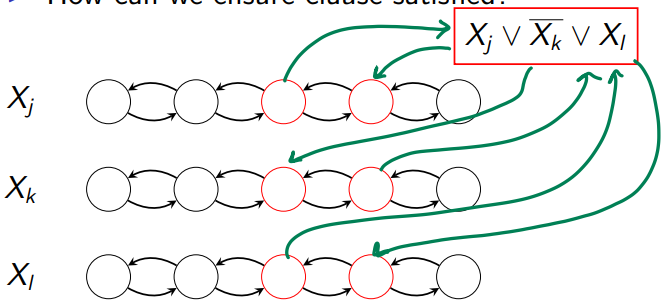
## Hamiltonian Path

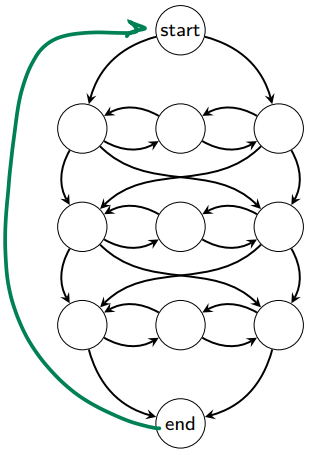
A simple **path/cycle** that includes every vertex in **directed/undirected** graph

Use Hamiltonian Path Solution to solve 3-SAT (Reduce 3-SAT to Hamiltonian Path)

For all the possible bool variables <x1, x2…xi>, have i rows just like the following.

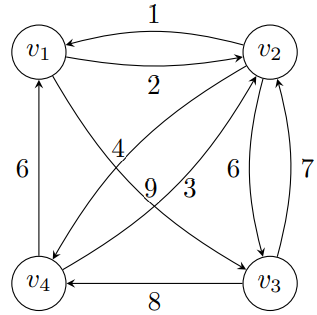
v

**



## TSP - Traveling Salesman Problem

Hamiltonian **Cycle** of lowest total weight



TSPJourney(i, A) = cost of the shortest path from vi to v1 passing through each vertex in A exactly once. (vi and v1 are not in A) ⇒ time complexity

O(n) for i range

O(2^n) for A range

O(n) to compute (i, A)

|  |
| --- |
| **def** **TSPJourney**(i, A):  **if** len(A) == 1:  **return** cost[A[0]][1]  curMin = inf  **for** a **in** A:  curMin = min(curMin, cost[i][a] + **TSPJourney**(a, A - {a}))  **return** curMin  totalMin = inf  **for** i **in range**(1, A.len):  totalMin = **min**(totalMin, cost[A[0]][A[i]] + **TSPJourney**(i, A)) |

# Discussion 2

Vertex Cover

Given a graph G and integer k, determine if there is some subset of the vertices V ′ , |V ′ | ≤ k such that each edge is incident to some vertex in V ′

with k = 3 ⇒ 3 2 7 (Select vertices that are not in vertex cover ⇒ complement of the vertex cover = independent set)

Ex. Given S students, B busses, F projects. For i project it takes s\_i and b\_i resource to get g\_i. Maximize sum(g\_i).

|  |
| --- |
| **def** **R**(i, s, b):  **if** i == 0:  **return** 0 **if** s < s[i] **or** b < b[i]:  **return** R(i-1, S, B)  **else**:  **return** max(R(i-1, S, B), R(i-1, S-s[i], B-b[i]) + g[i]) |

DP = O(FBS)

|  |
| --- |
| **declare** Goodwill[0..n, 0..B, 0..S] // to maintain our optimal values  **declare** DoProject[0..n, 0..B, 0..S] // to reconstruct whether we are doing the project or not  // fill base cases  **for** j = 0 -> B **do**  **for** k = 0 -> S **do**  Goodwill[0, j, k] = 0  DoProject[0, j, k] = false  // general cases  **for** i = 1 -> F **do**  **for** j = 0 -> B **do**  **for** k = 0 -> S **do**  // fill in Goodwill[i, j, k] with our defined recurrence  **if** Bi > j **or** Si > k **then** // we are not able to undertake this project  Goodwill[i, j, k] = Goodwill[i – 1, j, k]  **else**  Goodwill[i, j, k] = **max**(Goodwill[i – 1, j, k], Gi + Goodwill[i – 1, j - Bi, k - Si])  // if we undertake the project, we will have different goodwill  **if** Goodwill[i, j, k] **!=** Goodwill[i – 1, j, k] **then**  DoProject[i, j, k] = true  **else**  DoProject[i, j, k] = false |

# 

## Easy vs Hard Problem

|  |  |
| --- | --- |
| Easy - no answer is easy to prove | Hard - no answer is hard to prove |
| 2-color (find a cycle with odd vertices)  minimum spanning tree  euler tour (visit all edges exactly once)  ⇐ edge degree  shortest path  bipartite matching  a group matches to another group  pair matching without duplicate  linear programming | 3-color  TSP  Hamiltonian Tour  visit all vertices exactly once  longest path without re-visiting any vertex  3D-matching  3 groups wanna match without duplicate  integer programming |

## Graph Coloring

Given a (simple, undirected) graph G = (V, E)

assign each vertex a color

cannot assign the same color to the two endpoints of an edge.

Chromatic number of a graph G, written χ(G), is the minimum distinct colors you need to produce this.

## Spanning Tree Problem

Given a connected graph, a minimum spanning tree has input with weighted edges and keeps the one of lowest total weight. (minimal cost to connect all vertices in the graph)

Prim/Dijkstra’s = start with visited set, and get the adjacent node with minimal cost

Kruskal’s = order edge by cost and connect it if two endpoints are in different set

Cycle Property

C is any cycle

e is heaviest edge

any tree T that includes e, not MST

(Suppose T includes e and is MST…

)

Cut property

Cut the vertices into X and G – X

e is the lightest edge with endpoints in X and G – X respectively

Any tree that avoids e, not MST

(Suppose T avoids e and is MST.

in T’s edges s.t. one endpoint in X, and one in G – X.

cost(f) > cost(e) ⇒ T with f is not MST)

## Euler and Hamiltonian

Hamiltonian tour (NP complete)

visit every vertex exactly once

Euler tour

cross every edge exactly once

(impossible if len(edges with odd degree) > 2)

## Shortest/Longest Path Problems

Shortest path

unweighted ⇒ BFS

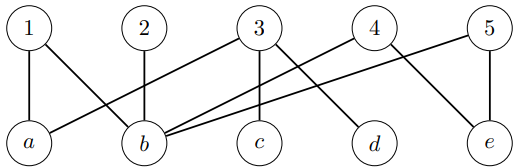
weighted, non-negative cost ⇒ Dijkstra’s

weighted, arbitrary cost ⇒ Bellman-Ford

## Matching Problem

Bipartite matching

given undirected bipartite graph G = (V, E), where V = V1 ∪ V2. M ⊆ E is a matching if each node appears at most once in M. Find the largest matching.



Hall’s Theorem …

3D Matching …

## Linear and Integer Problem

Problem: Web server company wants to buy new servers

Standard Model • $400 • 300W power • Two shelves of rack • Handles 1000 hits/min

Cutting-edge model • $1600 • 500W power • One shelf • 2000 hits/min

Budget: • $36,800 • 44 shelves of space • 12,200W power

Goal: maximize the number of hits we serve per minute (integer solution ⇒ NP Complete)

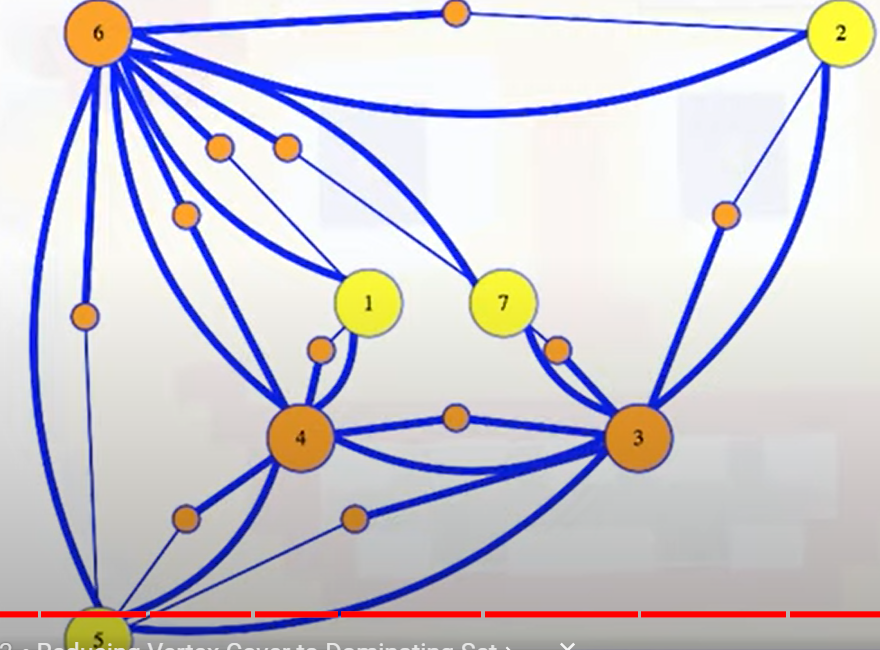
# Discussion 3

Dominating Set

Def

a subset V ′ ⊆ V of the vertices such that every vertex either is in V ′ or is adjacent to one that is. Prove that the problem of determining if a dominating set of size k is present in a general graph is NP-complete.

Answer



Dynamic Programming

1. The earliest legal step may not be in the optimal solution

1 2 3 4 5 6 7 8 9

1 -4 3 -1 7 -3 6 7 -2

The optimal solution is 1, 4, 6, 9

However the earliest legal step from 1 is 3

def MaxEvent(i, prev): # return max number of event at i position when previous event is at prev

if i == n: return 1;

if i - prev >= abs(d[i] - d[prev] and n - i >= abs(d[n] - d[i]):

return max(1 + R(i+1, i), R(i+1, prev))

else:

return R(i+1, prev)

dp = [0] \* n

dp[-1] = 1

for i in range(n-2, -1, -1):

for j in range(j):

if abs(d[i] - d[j]) <= abs(j - i) and abs(d[n] - d[i]) <= n – j:

dp[i][j] = max(1 + dp[i+1][i], dp[i+1][j])

else:

dp[i][j] = dp[i+1][j]

return dp[0][0]

## Dominating Set on a Undirected Tree

Each vertex has a cost

Select

or

Minimize the total cost of vertices in V’

DOM\_y(u): //min cost dom set rooted at u that includes u

cost(u) + {}

DOM\_n(u): //min cost dom set rooted at u that excludes u

min children c{DOM\_y(children) + }

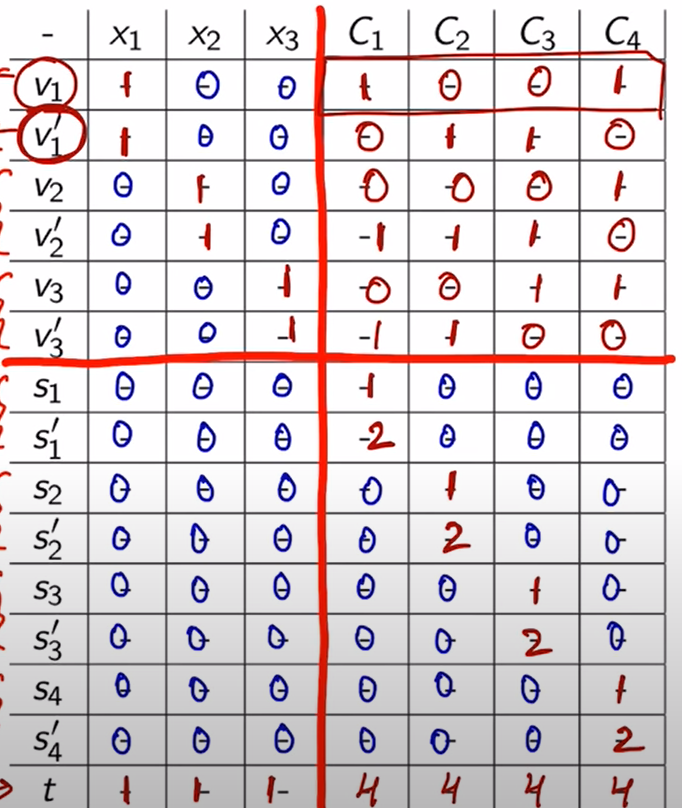
## Subset Sum to 3 SAT

for every variable x\_i, = 1 if , = 1 if

for every clause Ci column, put Si = 1 and Si’ = 2

for every clause Ci column, only put 1 to the v (v ← x in Ci)

The target value in the end = 111…4444…



## Choose Reduction

Constraint Satisfaction → 3 SAT

Def - categorize problem into an existing bucket

Clique Problem (CP) ← 3 SAT

Given an undirected graph G and an integer k, is there a complete subgraph of G with k nodes?

k clauses in SAT. Each x in SAT → vertex in CP

connect vertex in different clause if they are not negation (ex. x and not x)

solution(CP) ⇒ Solution(k-clause 3-SAT)

Maximum Cut Problem (MCP) ← 3 SAT (TBC)

Given a graph, the problem is to partition its vertices into two disjoint sets such that the number of edges between the two sets is maximized.

Each variable x\_i and not x\_i in SAT → vertex in MCP and add edge between

x\_i is in same clause with x\_j → edge between

Packing → Independent Set

Def. choose at least k objects (some pair cannot be choose at the same time)

Covering → Vertex/Set Cover, Dominating Set

Def. choose at most k objects to achieve certain goals

Permutation/Sequencing → Directed Hamiltonian Path/Cycle

Def. determine an order for n objects, or if one exists, subject to constraints

Scheduling Problem (SP) ← Subset Sum (SS)

Each integer x in SS → a job with processing time = x

Target integer t in SS → deadline of the job

Solution to SP ⇒ able to generate a solution for SS

SP with order ← directed Hamiltonian path/cycle G

Each vertex v with in G → job

Directed edge from i to j → job i must be completed before j starts

If no constraint between two jobs, double arrowed between vertices

Solution(G) ⇒ solution(SP) to schedule all jobs in the correct order

Longest Common Subsequence Problem (LCS): Given two sequences of characters, the problem is to find the longest subsequence that is common to both sequences.

Numerical → Subset Sum

Def. select object(s) subject to a totality constraint

Knapsack Problem (KP) ← Subset Sum (SS)

Given a set of items, each with a weight and a value, and a knapsack of a certain capacity, the problem is to determine the items to include in the knapsack such that the total weight does not exceed the capacity, and the total value is maximized.

SS target sum T → capacity in KP

SS integer s\_i in set → item with v\_i and w\_i both equal to s\_i

Solution(KP) to fill pack with capacity ⇒ solution(SS) to sum items to T

Bin Packing Problem (BPP) ← Subset Sum (SS)

Given a set of items, each with a size, and a set of bins with fixed capacity, the problem is to pack the items into the bins such that the total number of bins used is minimized.

same as KP ← SS process

Solution(BPP) means we find a way to pack all items to a single box with capacity = T(SS) ⇒ solution(SS) to sum items to T

Partitioning → 3 Color

Def. divide a collection into subsets (one object appears in exactly one bucket)

## Knapsack

Find the smallest knapsack weight W s.t. a subset of items {1…i} can be taken with value at least V

|  |
| --- |
| **def** **R**(i, V): #returns smallest weight at i item  **if** i <= 0 **and** V > 0: **return** inf #reach the end **and** V **is** **not** satisfied  //skip i item **or** take it  **if** prefixSum[i-1] < V:  **return** R(i-1, max(V - v[i], 0) ) + W[i]  **else:**  **return** min(R(i-1, V), R(i-1, max(V - v[i], 0) ) + W[i]) |

* Instead of directly min(R(), R()), we can use **prefix sum** to know if the ith value is mandatory to take

|  |
| --- |
| **def** **DP**():  //fill **in** dp[i][0] **with** 0  //fill **in** dp[0][V != 0] **with** inf  **for**(i = 1 to n):  **for**(j = 1 to V):  **if** prefixSum[i-1] < v:  dp[i][j] = dp[i-1][max(..)] + w[i]  **else**:  dp[i][j] = min(dp[i-1][j], dp[i-1][max(j-v[i], 0)] + w[i])  **return** the min(dp) |

Rounding Knap Problem

trade accuracy for efficiency

# Discussion 4

Sharing Problem ← Subset Sum (SS)

Assume we have SS with input set = S and sum(S) = M, target = T

create a new element x\_n+1 s.t. solution(SS) + x\_n+1 = sum(rest of the set) or solution(SS) = sum(rest of the set) + x\_n+1 (depending on which on is larger sum)

Assume solution(rest) = M - T is the larger sum. then x\_n+1 = M - 2T.

SP input = S + x\_n+1, and goal of SP is to find two sets that sum up to (M-T)

If there is a solution for SP, then it’s also the solution for SS

Algorithm design

(naive solution) For each connected subgraph, call A with k = size(sub graph). Decrease k until “yes” answer.

(bfs to find “yes”)

Inner Circle (IC) ← Vertex Cover (VC).

vertex in VC → professor in IC

edge E between i and j in VC → issue E agreed by professor i and j

solution(IC) means we have at least one professor, who correctly voted on each issue, in the solution ⇒ solution(IC) has at least one endpoint for each edge ⇒ this solution(IC) can be used for VC

Strongly independent set is in NP

add a vertex to the middle of every edge. SIS(new graph) = ID(previous graph)

connect every newly added vertex to every other newly added vertex (so that they are not selected)

Set Packing ← Independent set (ID)

connect two sets if they have any elements in common. ID(graph) = set packing

Stingy-SAT ← Vertex Cover (VC)

if a variable is in set, connect variable to the set. the covering vertices of size k = stingy sat of size k

Clustering ←3 Color

adjacent vertex in 3C ⇒ distance > T in C

solution(C) with k = 3, then we have a 3C solution

Tonian Path (TP) ← directed Hamiltonian path (HP)

edges and vertex in HP → edges and vertex in TP

solution(TP) with size = n ⇒ solution for HP

Min-Cost Fast Path (MCFP) ← Hamiltonian path (HP)

edge, vertex in HP → edge, vertex in MCFP

T = n - 1, C = n-1 and t(e) = c(e) = 1 for all edges

solution(MCFP) = solution(HP)

# Midterm Review

Poly-solvable and NP hard

poly-solvable can verify “NO” answer easily

NP hard can verify “NO” instance but not “NO” overall easily

To prove problem B is NP complete

Known NP complete problem A has its input I(A)

Translate I(A) to I(A’) so that

I(A’) can be inputted into B

output(A’) can be used to solve A

Example of 3-Color to Truck loading

vertex i in 3-Color → canister i in Truck

edge between i and j in 3 Color → truck(i) != truck(j) cannot be put together

no capacity in 3-Color → truck capacity = |V|

input = (canisters, 3 trucks, |V| capacity)

Example of 3-Color to Diameter

vertex i in 3-color → vertex i in diameter

edge between i and j → 100 distance from i and j

no edge between i and j → 1 distance from i and j

// j = 1st char of last word s[1…j-1] and word: s[j:i]

def segment(i):

if i == 0: return 0

max( segment(j-1) + quality(s,j,i) 1<= j <= i

Trucks

1. canister to truck

canister to track

each track has max carrying cap

1. canister to canister

truck has max carrying cap

load all trucks

Spacing: min distance between vertices in different sets → maximize the min dist

Diameter: max distance between vertices in same set → minimize the max dist

## Interval Problem

Unweighted interval scheduling (same credit + different length of courses)

Solution:

max # courses without overlapping → choose that ends earliest

(why class with the fewest conflicts doesn’t work?)

Claim: there is an optimal solution that includes the first-ending class (FEC)

Proof: suppose all optimal solutions do not. Select a random optimal solution OPT

OPT’ = OPT – FEC(OPT) + FEC(overall)

OPT’ size = OPT size

If FEC(overall) doesn’t overlap with FEC(OPT), then OPT could have added the FEC(overall), which means it’s not the optimal solution.

Since OPT is an optimal solution, FEC(overall) and FEC(OPT) overlap. If we remove FEC(OPT), we should be able to add FEC(overall) without overlapping the other courses in OPT.

Interval Coloring (n reservations. min # room without overlapping)

when a group arrives, give lowest numbered **free** room ⇒ max # of overlapping schedule

sort by end\_time of classes

Fractional Knapsack

Decide

Require

Goal:

Ex. suppose W = 10

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Item | 1 | 2 | 3 | 4 | 5 |
| Weight | 4 | 8 | 2 | 6 | 1 |
| Benefit | 12 | 32 | 40 | 30 | 50 |

Take the item with highest benefit/weight

Claim

sort by bi/wi and choose in order

Proof

Suppose there is an optimal solution, where only one item is different.

i < k and . That means we could have taken i instead of k to get better result (sum of benefit)

Scheduling with Deadlines

Def

Each homeword can be started at s\_i, and estimated work time is t\_i, the deadline is d\_i. The lateness of the homework = s\_i + t\_i - d\_i. Penalty = max(all lateness). We want to minimize penalty

sort by deadline

sort by d\_i - t\_i (latest start time) is not working

Proof

Any schedule that doesn’t agree with our algorithm has at least one pair of consecutive intervals i, i + 1 that are inverted relative to our order.

Any diff →

if j = i+1, done

else k=i+1, i,k inverted? if yes, done. Else A[i]<A[k], A[i] > A[j]

When i,j inverted, j = i+1 but **d\_i >= d\_j**

before swap i,j in alt. order

f\_i = s\_i + t\_i → f\_i’ = s\_i + t\_j + t\_i = f\_j

f\_j = s\_i + t\_i + t\_j → f\_j’ = s\_i + t\_j < f\_j

if f\_i’ is late, no worse if **d\_i >= d\_j**

## Approximate TSP

Assumption:

complete graph: exists an edge between all pairs of vertices

triangle inequality:

Goal:

fast algorithm in poly time and provide some correctness. For example,

Solution

Find MST **T** of G

Do depth-first walk **W** of T ⇒ 2\*cost(T) = cost(W)

Output vertices to **J** in order seen on W



Without triangle inequality:

HamiltonianCycle(G)

G’ = complete graph with G’s vertices

weight(edge e) = 1 if e in G, otherwise, =

## Approximation of Vertex Cover

Problem Def

V set so that all edges have at least one endpoint in V

Greedy

Load Balancing

Problem Def

Given m identical machines. n jobs each needs time t\_j. Give every job to some machine. Goal is to balance the machine loads; minimize

# Discussion 6

1. There are n houses located at various places on this road, and we want to make sure that each house is within 5 miles of an Algorithmic Pizza. Give an algorithm that will achieve this goal with the fewest possible pizza parlors placed

Algorithm

Put all the houses in the consider list

From the leftmost house, move 5 miles and place a pizza parlor and remove all houses along this way from the consider-list

Remove all the houses, which locates 5 miles to its right, from the consider list

Move to the right and find the next house in the consider list

Prove

our algorithm SOL place the leftmost pizza parlor at p

select an optimal solution OPT which places a different leftmost pizza parlor at q

since p is 5 miles to the right of the leftmost house, q is on the left side of p

every house covered by q is also covered by p (the leftmost house is already covered by both SOL and OPT)

OPT’ = OPT - q + p = SOL will still be an optimal solution since it still covers all houses and it has the same size as OPT

1. n pizza orders. Each takes pi time for preparation and bi time for baking. Only a single pizza can be in preparation. You can bake any or all of the pizzas simultaneously. We want the last pizza to come out of the oven as soon as possible.

Algorithm

Sort the pizza by baking time

Prepare the pizza with the greatest bi time

Bake pizza immediately when it’s prepared

Prove

total time = total p\_i + b\_i after p

Since only one pizza can be in preparation ⇒ total p\_i is the same

Suppose there is an optimal solution OPT’. When we take q, it takes k pizza to work on. b\_q > b\_k. q in OPT’ is prepared and baked later than k. Since b\_q > b\_k. If q and k are the last two pizzas, we have longer wait time. That means OPT’ is not actually optimal..

## Min and Max

2n – 2 information in total

C1 + 2 C2 >= 2n-2

at most n/2 comparisons give us two units → - C2 >= -n/2

⇒ C1 + C2 >= [(3/2)\*n - 2](https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_max_min_problem.htm#:~:text=is%20the%20technique.-,Divide%20and%20Conquer%20Approach,two%20minima%20of%20each%20half.)

|  |
| --- |
| def **maxmin**(x, y)  **if** y - x ≤ 1 then  **return** (max(numbers[x], numbers[y]), min((numbers[x], numbers[y]))  **else**   (max1, min1):= maxmin(x, ⌊((x + y)/2)⌋)  (max2, min2):= maxmin(⌊((x + y)/2) + 1)⌋,y)  **return** (max(max1, max2), min(min1, min2)) |

## Counting inversions

def: i,j are an inverted pair if i < j and A[i] > A[j]

increasing heap. If the new element is smaller, pop until it’s good. Number of pop = number of inverted pairs?

A[1…n/2] and A[n/2 + 1…n] are sorted

|  |
| --- |
| #merge-and-count  count = 0 i = 1, j = n/2 + 1  T[1..n], k = 1 **while** i <= n/2 and j <= n:  **if** A[i] > A[j]:  j ++  count++  T[k] = A[j]  **else**:  i++  T[k] = A[i]  k++  **while** i <= n/2:  T[k] = A[i]  i++  k++  **while** j <= n:  T[k] = A[j]  j++  k++  copy T to A  **return** count |

Running time T(n) = 2 \* T(n/2) + n = 4 \* T(n/4) + 2\*n/2 = …

* for divide & conquer has running time T(n) = a\*T(n/b) + f(n) where a >= 1 and b >= 1 and f(n) is asymptotically positive
  + If there is a small constant ε > 0 such that f(n) is , then T(n) is
  + If there is a constant k ≥ 0, such that f(n) is , then T(n) is
  + If there is a small constant ε > 0 such that f(n) is , then T(n) is

Ex. T(n) = 4T(n/2) + n → when ε = 0.1 in 1, f(n) = n is in → T(n) is in

Ex. T(n) = 2T(n/2) + n\*logn → when k = 1 in 2, f(n) is in n\*logn → T(n) is in n log n

Ex. T(n) = T(n/3) + n → when ε = 0.1 in 3, f(n) = → T(n) is in O(n)

Ex. T(n) = 9T(n/3) + n^2.5 → when ε = 0.1 in 3, f(n) is

## Selection Problem

Problem = given a list S and numeric k. What is k-th smallest number in S?

Randomized Selection

|  |
| --- |
| **def** **quickSelect**(S, k):  **if** n **is** small : brute force **and** **return**  x = random element **in** S //pivot  L = elements smaller than x  G = elements larger than x  **if** k <= |L|:  quickSelect(L, k)  **else** **if** k == |L| + 1: **return** x  **else**:  quickSelect(G, k - |L| - 1) |

worst-running time = O(n^2) with bad pivot (every pivot is next largest/smallest number)

average = O(n) if pivot is the median ??

Median-of-five (better quickselect?)

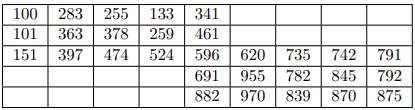
instead of randomly picking x

divide S into g groups (g = n/5)

find the median of each group (most groups contain 5 elements) ← O(1)

x = median of those medians ← T(n/g)

* n/5 groups has n/5 medians
  + → n/10 medians smaller than x
  + → 2n/10 smaller than x and in median groups (not median)
* → 3n/10 smaller elements in those groups (non-median + median)
* → 3n/10 larger elements
  + 4n/10 element unknown



linear running time?

Median-of-five()

Project 5 (better than brute force to get credit)

# Discussion 8

1. T(n) = T(n/2) + O(1) → O(log n)

find-local(A, l, h):

if A.size <= 2: return 0

mid = (l+h)//2

if A[mid] is local\_min: return A[mid]

if A[mid] > A[mid-1]: return find-local(A, l, mid)

else: return find-local(A, mid, h)

1. T(n) = 2 \* T(n/2) + O(n) → O(n log n)

sort-hat(i, H, M):

if H.size is small: brute force

M\_s, M\_l

exact = -1

for every m in M:

if H[i] is smaller for M[m]:

add to M\_s

else if H[i] is larger for M[m]:

add to M\_l

else:

exact = m

H\_s, H\_l

for h in H:

if H[h] is smaller for M[exact]:

add to H\_s

else if larger:

add to H\_l

sort-hat(0, H\_s, M\_s)

sort-har(0, H\_l, M\_l)

1. T(n) = 2 \* T(n/2) + O(n) → O(n log n)

findMajorityCard(keycards)

if n is “small”

Brute force and return the majority card (or n == 1: return keycards[0])

left\_subset = keycards[:n/2]

right\_subset = keycards[n/2:]

left\_majority\_card = findMajorityCard(left\_subset)

right\_majority\_card = findMajorityCard(right\_subset)

for card in keycards:

if left\_majority\_card == card:

left++

if right\_majority\_card == card:

right++

if left > n/2:

return left\_majority\_card

if right > n/2:

return right\_majority\_card

return None

## Integer Multiplication

Problem

Given two n-bit integers X and Y , compute X × Y . The algorithm you learned for this in grade school takes time O(n 2 ).

Al-Khwarizmi’s Algorithm

|  |
| --- |
| Mult(X, Y):  Create Xh, Xl, Yh, Yl  A = Mult(Xh, Yh)  B = Mult(Xh, Yl)  C = Mult(Xl, Yh)  D = Mult(Xl, Yl)  //E = Multi(Xh+Xl, Yh+Yl)  **return** A \* 2^n + (B+C) \* 2^(n/2) + D  //return A \* 2^n + (E-A-D)\*2^(n/2)... + D |

without E:

with E:

## Minima-Set Problem

Problem

Given a set S of n points in the plane, we want to find the set of minima points. That is, if we include (x, y) in our output, we want to ensure that there is no point (x ′ , y′ ) in the output such that x ≥ x ′ and y ≥ y ′

Find the A with minimal y and B with minimal x. Find all points x <= A’s x and y <= B’s y

Divide & Conquer

|  |
| --- |
| MinimaSet(S):  **if** n <= 1: **return** S  p = median point in S by x  L = points less than p  G = points greater than or equal to p  M1 = MinimaSet(L)  M2 = MinimaSet(G)  **return** M1 + {e in M2 whose y is smaller than min(M1.y)} |

# Discussion 9

1. T(n) = T(n/2) + O(1) → log(n)

//cycle graph

Pick opposite elements X and Y

find-local(X,Y):

if X and Y is local minimum: return

X = X.lesser

Y = Y.lesser

find-local(X, Y)

1. T(n) = T(n/2) + O(1) → O(log n)

find-midian(A,B):

if A and B are small: brute force

if middleElement(A)[0] > middleElement(B)[0]:

A = discardRight(A)

B = discardLeft(B)

else:

A = discardLeft(A)

B = discardRight(B)

find-median(A,B)

1. T(n) = 2 \* T(n/2) + O(1) → O(n)

maxOverlap(A):

if A is small: brute force and return

mid = A.size()//2

max1 = maxOverlap(A left half) //1…mid-1

max2 = maxOverlap(B right half) //mid…end

return max(max1, max2,

min(A[mid-1].graduation, A[mid].graduation) − max(A[mid-1].enrollment, A[mid].enrollment))

# Midterm

packing, covering, sequencing for greedy algorithm proofs

1. Suppose all optimal solutions don’t have B2PP(best 2 player paired)

randomly select an optimal solution OPT.

A, B are the best players and q\_a > q\_b in OPT.

since they cannot be paired together, they are paired with some other players C and D where q\_b > q\_c and q\_b > q\_d

A and C team quality = q\_c

B and D team quality = q\_d

total quality(OPT) = q\_c + q\_d

For the same four players A, B, C, D, the B2PP will have the following teams:

A and B with the team quality q\_b

C and D with the team quality q\_d

total quality(B2PP) = q\_b + q\_d > q\_c + q\_d since q\_b > q\_c

That means B2PP is better than the OPT, which conflicts with the assumption that OPT is one of the optimal solutions.

1. **T(n) = T(n/2) + O(n) → O(n)**

missing-bit(A, count\_bit):

if A.size is small: brute force and return

count = (A.size() + 1) // 2

for i in range(A.size()):

count-=bit[i, count\_bit]

if count == 1:

remove all elements in A whose count\_bit = 1

else:

remove all elements in A whose count\_bit = 0

missing-bit(A, count-bit + 1)

1. Suppose all optimal solutions don’t include the LFW (last-finished-worker)

Randomly choose an optimal solution OPT

Consider the following case:

1\_\_\_\_ 3 \_\_\_\_\_

2 \_\_\_\_\_\_\_

When we look at worker 1 with an overlapping area with worker 2, we either choose worker 1 or 2 as the supervisor.

LFW chooses worker 2 with maximum finish time. This will handle the worker 1 and 3. Number of supervisor(LFW) = 1

OPT will choose worker 1. Then when we see worker 3, OPT needs to make worker 2 or 3 a supervisor. Number of supervisor(OPT) = 2 > number of supervisor(LFW), which conflicts with the definition of optimal solution.

x = OPT’s first chosen

y = my first chosen

OPT’ = OPT - x + y

both cover 1st and finish\_time(x) <= finish\_time(y)

1. 2 T(n/2) + O(n) → O(n log n) if not sorted

2 T(n/2) + O(1) → O(n) if sorted

skyline(T, lo, hi):

if hi == low: return list[tuple( T[low][0], T[low][1] )]

mid = (lo+hi) // 2

res1 = skyline(T, lo, mid)

res 2 = skyline(T, mid + 1, hi)

//sorted

// return res1[:-1] + tuple(res1[-1][0], res1[-1][1] + res2[0][1] ) + res[1:]

//not sorted

int i, j = 0, 0

res = list()

cur\_x = 0

while i < res1.size() and j < res2.size:

if res1[i][0] < res2[j][0]:

res.append( tuple (res1[i][0], res1[i][1] - cur\_x)

cur\_x = res1[i]][1]

i ++

else:

res.append( tuple (res2[j][0], res1[j][1] - cur\_x)

cur\_x = res2[j]][1]

j ++

return res;