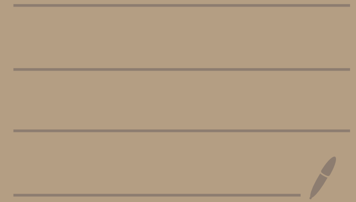
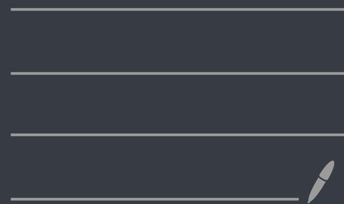
CISC 365



Unit 1



Lecture 2 - Complexity

Comparing Algorithms

- Optimality
- Elegance / Explainability
- Time complexity
- Space complexity

Order of Growth

- Problems grow in decideability
- rate of growth relates to constant, linear, polynomial, exponential
- Beyond n^n Problems are undecideable

- Big O (O): upper bound → worst case
- Big Omega (Ω) Lower bound → Best case
- Big Theta (Θ) Tight bound → average case

For any 2 functions $f(n)$ and $g(n)$

$f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Lecture 3 - Recursive complexity

Reccurrence Relation - math equation that defines a value using earlier values of itself.

Recursion - functions calling themselves into smaller versions of same problem.

General Form: $a_n = f(a_{n-1}, a_{n-2} \dots)$

Ex: - Recursive Algorithms
- Growth processes
- Time complexity

Base case: Most simple version of problem. (Termination condition)

Complexity and implementation matter, iterative solutions are harder to understand.

Recurrence → Time complexity
Sometimes recursive has worse time complexity

T(n) - Actual running time, run algorithm

Recursive function complexity

How do we find $O(T(n))$?
Substitution/Expansion

$$T(n) = O(1) + T(n-1)$$
$$T(n) = O(1) + O(1) + T(n-2)$$
$$T(n) = O(1) + O(1) + O(1) + T(n-3)$$
...
$$T(n) = O(1) + O(1) + O(1) + \dots + O(1) + T(n-n)$$

Process

- Write clean reccurrence relation
- unwind
- Add base case

- Unwind and do it n times.

Recursive call is not $T(n-1)$

- constant reduction:
  $T(n) = O(1) + T(n-c)$
- unwind it $T(n) = O(1) + T(n-c)$

Do many Examples

Lecture 4 - P and NP

- An algorithm is good if it scales well as input size grows

$O(1) \to O(\log n) \to O(n) \to O(n \log n) \to O(n^2) \to O(2^n) \to O(n!)$

- n is like the number of bxes you must check

- Easy problems run in polynomial time
  → up to $O(n^k)$
- Hard problems have no polynomial time algorithm

- Split at polynomial time as real world problems should be "easy"
  ↳ closed under composition (theory)

Cook-Levin Theorem

- Let x be any problem in NP, then $x \leq SAT$. Boolean Satisfiability.
- Every problem in NP, can be transformed into SAT in polynomial time.

Class P: Polynomial Time - problems we can verify a solution fast
Class NP: Non deterministic polynomial time. (Hard to find solution, easy to check.

Decision problems - P and NP are defined for only yes/No questions

- Relationship: $P \subseteq NP$, if you can solve a problem fast. You can verify it fast. (Every P problem is in NP)
- Is $P = NP$ (if we can verify a solution quickley, can we always find one quickley.

SAT is the hardest problem in NP, every NP problem can be translated to polynomial time. $x \leq SAT$. Can be reduced to SAT in polynomial time.

NP-Complete: hardest problems in NP.

NP-hard: if a problem A can be reduced to B. Where B is in all of NP.

Reduction

- way of solving problem x, using a solver for problem y. If you can turn x into y, solving y also solves x, so x is no harder than y
- Form of Transformation.
- Every instance of x should be transformed to an instance of Y.

Conditions:

1) Fast transformation; (takes polynomial time)
2) Answer is Preserved; (Answer is same in both x/y)

$X \leq_p Y$

x is no harder than y.

Reduction is how we prove problems are NP-hard.

- Transitive Property: $X \leq Z$ and $Z \leq Y$
  Then: $X \leq Y$
- SAT is like a bench mark
  (any NP Problem can be translated to SAT).

X is np-complete if: x is in NP and every NP problem reduced to x.

A is np-hard if for all $B \in NP$, $B \leq_p A$

A is NP-complete if A is NP-hard and $A \in NP$

If A is NP-Complete and $A \in P$, $P = NP$. (if it can be solved in polynomial time)

Mental Recipe

1) write down "Yes" versions of each problem
2) what do these problems have in common (proof of objects)
3) Force one to look like the other (choose parameters, add numbers, nodes ...)
4) Prove the two directions preserve correctness

Quiz: complexity and NP-Completeness

1) 1st for loop - $O(n)$
Then 2 cases inside if statement are: $O(\log n)$
$O(n)$

would think its $O(n^2)$
but this case is negligible due to the exponential increase of 2^i in if statement. $\therefore O(n \log n)$

2) first loop is $O(n)$
then second loop is $O(n)$
↳ but actually $i \times 2$ gets done faster. So $\log n$

$\therefore O(n \log n)$

3.) $T(1) = O(1)$

Base case: $O(1)$

$T(n) = O(n) + T(n/2)$ → $\log n$

Each iteration time ← $= O(n) + O(n) + T(n/4)$

$T(n) = O(n)\ O(\log n)$ → how many times n is called back

$= O(n \log n)$

4) $O(b.c) + O(c) \cdot O\left(\frac{n}{\log n} \rightarrow \text{return}\right)^{n\text{-}c}$

$O(1) + O(\log n) \cdot O(n)$

$O = (n \log n)$

5.) $E = \{e_1, e_2, e_3 \ldots\}$
$S = \{s_1, s_2, s_3 \ldots\}$

Student mapped to exam
$K \geq 3$
K = time slots

1 timeslot → 1 exam

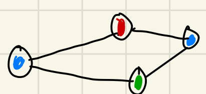
If we can prove K-coloring can be reduced to ETS, we can say ETS is np-complete.

K-colouring

$K = 3$

K colours min

K-Coloring $\propto$ ETS
↗ np complete

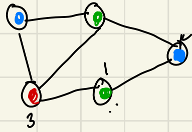


1.) Proof in NP

go through every exam (n exams)
ensure only 1 per timeslot. Verifiable in $O(n)$

Check each student, exams against every other exam.
Do this for m student. So $O(m) \cdot O(n^2)$
This is polynomial time.

ETS



Timeslot → colours (K)
↳ adjacency, no two exams that 1 student has can be same timeslot

Nodes → Exams

Students: Cant have two exams in same timeslot. They are reperesented by edges. Because no two nodes (no two exams) can be same colour.

set of color is at same time

At most a student will have 2 exams

when converting from Kcolouring to ETS

created relational mapping from input to EST

3) show reduction is polynomial time

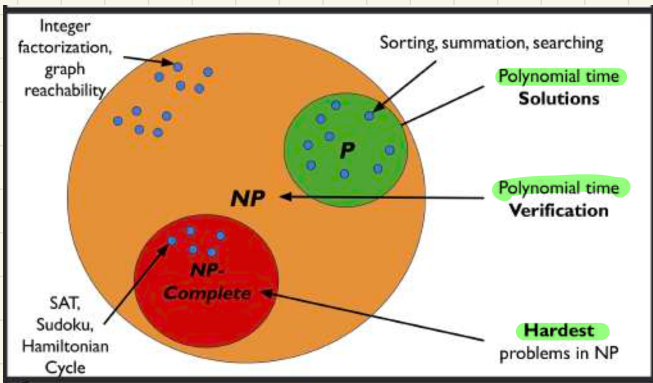
- 1) show K=K in both problems (O(1))
- 2) made graph with nodes → exams O(n)
- 3) turned edges to Students → O(n) and made relational mapping with 2 exams
- 4) ∴ O(n) is polynomial time

4) Answer preserving

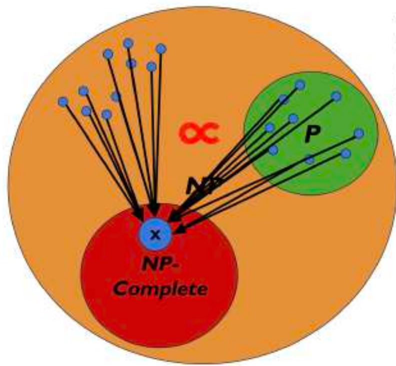
- Assuming K colouring is "yes", we know there is distinct colours such that adjacent colours are different and each edge connects two nodes (2 exams)
  ↳ this implies the students have 2 exams at different distinct timeslots / colours.

- Assuming ETS "yes" theres a number of students and exams, s.t there are no two overlapping exam timeslots which implies no two same colors is tocching.

Lecture 5 - NP and Reduction



A problem x in NP, is NP-complete if all problems in NP reduce to X.



If any NP-Complete has a polynomial time solution P = NP

Once a problem is NP-complete Stop looking for Solution

To prove NP-completeness

1) Show Problem in NP
2) Show every Problem in NP reduces to your problem (NP-hard)

Tutorial 2 - Partition Reduction to Subset Sum

DEFINITION: The PARTITION problem is defined as follows: Given a set S of n positive integers, can S be partitioned (ie. divided) into two subsets S_1 and S_2 with the sum of the elements of S_1 equal to the sum of the elements of S_2?

DEFINITION: The SUBSET SUM problem is defined as follows: Given a set S of n positive integers and a positive integer k, does S contain a subset that sums to k?

$S: \{x1, x2, x3, x4\}$ $\quad sum(S1) = sum(S2)$

$S1: \{x1, x3\}$ $\quad S2: \{x2, x4\}$

$S: \{\{x1, x2\} x3 \ldots\}$

$K = 5$ $\quad SS(sum) = K?$

1) Showing Partition ≤ Subset Sum

Partition: $\Sigma(S_1) = \Sigma(S_2)$

Subset sum: $S_1 \subseteq S$, s.t $\Sigma(S1) = K$

2) Key Insight

$\Sigma(S_1) = \Sigma(S_2) = \frac{\Sigma(S)}{2}$

"Is there a subset whose sum is half the total"

3) Define reduction

given an instance of partition:

$K = \frac{\Sigma(S)}{2}$

Output to subset sum instance (S, K)

Computing sum is $O(n)$

∴ The reduction is in polynomial time

4) Correctness proof

if you can partition subsets to sum equally that must be half of total sum.

$\Sigma(S_1) = \Sigma(S)/2 = K$

if subset sum is true that mean sum must be positive even.

$\Sigma(S_2) = \Sigma(S) - \Sigma(S_1) = \frac{\Sigma(S)}{2}$

So S_1, S_2 form valid partition

5) NP-Completeness conclusion

subset sum is $\in NP$ $(O(n))$ time

Partition ≤ subset sum

Partition is NP-complete

∴ Subset sum is NP-complete

Tutorial 2 - Subset Sum Reduction to Partition

DEFINITION: The PARTITION problem is defined as follows: Given a set S of n positive integers, can S be partitioned (ie. divided) into two subsets S_1 and S_2 with the sum of the elements of S_1 equal to the sum of the elements of S_2?

S $\quad S_1 \; S_2 \quad S_1 \cup S_2 = S$

$Sum(S_1) = Sum(S_2)$

DEFINITION: The SUBSET SUM problem is defined as follows: Given a set S of n positive integers and a positive integer k, does S contain a subset that sums to k?

$S, k \quad S_1 \quad S_1 \subseteq S$

$Sum(S_1) = k$

assumed to be NP-complete

Subset sum $\propto$ Partition

If subset sum reduces to Partition, Partition is NP-hard.

Since Partion is in NP, Partition is NP-complete.

Why obvious Idea fails

$t = \Sigma(S)$

if subset sum has solution S_1: $S_2 = S / S_1$

and $\Sigma(S_2) = t - K$, where $t = \Sigma(S)$

But, we need $\Sigma(S_1) = \Sigma(S_2)$

only works if $K = t - K \Rightarrow K = \frac{t}{2}$

Key Idea

add an extra number x, so balance total

$K + x = t - K$

$x = t - 2K$

Define reduction

$t = \Sigma(S)$

$x = t - 2K$

New set: $S' = S \cup \{x\}$

S' is the new input to partition

computing sums is $O(n)$ polynomial time

Correctness Proof

Subset $\Rightarrow$ Partition

Assume Subset Sum (S, K) = Yes

- $\exists S_1 \subseteq S$ with $\Sigma(S_1) = K$
- Let $S_2 = S / S_1$ so $\Sigma(S_2) = t - K$

Now if $S' = S \cup \{x\}$

put $S_1' = S_1 \cup \{x\}$

$S_2' = S_2$

$\Sigma(S_1') = K + (t - 2K) = t - K = \Sigma(S_2')$

Partition $\Rightarrow$ Subset Sum

- Then S' splits into A, B

Assuming Partion exists

split S' into A, B with equal sums

$\Sigma(S') = t + (t - 2K) = 2(t - K)$

$\Sigma(A) = \Sigma(B) = t - K$

Added element x must be in 1/2 sets

Then $\Sigma(A / \{x\}) = (t - K) - (t - 2K) = K$

$A \setminus \{x\} \subseteq S$

sum is K

$\therefore$ SubsetSum $\leq$ Partition

Subset sum is NP-Complete

Partion is NP-complete.

Unit 2

Lecture 6 - Divide and Concquer

Unit 2

- follows recursive structure based on n
- if n is small, solve problem directly

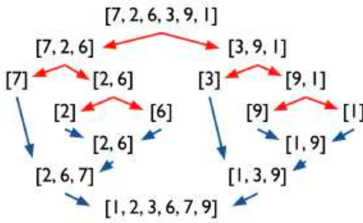
Large N

1) Divide: Break into subproblems
2) Conquer: Solve subproblem recursively
3) Combine: merge subproblem

Merge Sort

- divides list based on position
- splits list into two halves, recursively sorts them, uses merge function to combine sorted halves
- Complexity $O(n\log n)$ → for all cases
  - ↳ space complexity: $O(n)$

Splitting is: $O(\log n)$
↳
Merge is: $O(n)$



Quicksort

- divides based on value rather than position
- Pivot element, partitions list into 3 sublists. Less than, equal to and greater than.
- worst case t.c is $O(n^2)$
- Best/Average case is $O(n\log n)$
- Advantage: space complexity $O(1)$
- Pivot strategy stays same

Binary Search

- divides search space in half, picks subproblem that contains target solutions. Searches based on 2^x rounds

Lecture 7 - Divide and Conquer 2

- Maximum Stock Profit: buy low and sell high
- Brute Force Approach: enumerate all buy and sell days: t.c of $O(n^2)$

Divide and Conquer Approach:

- Divide: Split list into two halves
- recursively find max profit
- combine; the best profit. May be in both seperate groups. Calculated as max (p_r) - min (p_l).
- Result: returns maximum of left profit, right profit and crossing profit.

Conquer: Best Profit fully inside left half
Best Profit fully in right half
Best crossing split, buy in P_l, Sell in P_r

if best buy is in left, best sell in right

$\max(P_r) - \min(P_l)$

- Complexity: reccurence relation $T(n) = 2T(n/2) + O(n)$.
- Problem transformation: Change[i] = Price[i] - price[i-1]

Often checking 3 cases.

The maximum Subarray problem

- find largest subset of contigous array
- Stock profit transforms this by calculating daily change, change[i] = Price[i] - Price[i-1]
- Sum of the change array is best stock profit (Positive change is good days)

Divide and Conquer Algorithm

- Split array at midpoint
- recursively find max subarray in left and right halves
- combine: find maximum crossing subarray by iterating from midpoint outwards to find bet sum that spans both halves
  - Complexity is O(nlogn)

Real World

- Bioinformatics, sports analytics, image processing

Divide and conquer - 3

- How the game determines when 2 players are close enough to interact.

Brute Forces

- Check all possible pair of players. But is $O(n^2)$

Divide and Concquer

- preprocessing: Sort points based on x, y coordinates. $O(n \log n)$
- Split into 2 halves: P_L and P_R
- Recursively find closest pair in left δ_L and right half δ_R.
- Combine: must check if absolute closest pair consists of one point from P_L to P_R.

- get x, y vals, split down middle (Left and Right)
- δ is the minimum of δ_L, δ_R
  - ↳ need 2 distances ⟶ $\min(\delta_L, \delta_R)$ → one distance
- only check points δ within middle line
  - they cant beat our current best $\geq \delta$
- green rectangle of 2δ width
- only points within this distance can be smaller
- only check a pink rectangular box of $\delta \times 2\delta$ region around each point
- Sorted by y, have to check next 5 points, guarantee to catch all possible pairs

T.C: $O(n \log n)$ → better than brute force

Divide and conquer 4

Pair Sub Problem

- if set S contains, two elements that sum to target K.
- Brute force $O(n^2)$

↓

Optimized Solution

- Set S: gets sorted $O(n \log n)$
- Use two pointers l at the start and counter at the end r

$S = \{n_1, n_2, n_3 \ldots n_n\}$

Pointer l (at n_1), Pointer r (at n_n)

- if the current sum $t < K$, move left pointer up $l{+}{+}$. if $t > K$, move the r pointer $(r{-}{-})$.
- This improves complexity to $O(n \log n)$

Two-Set Pair Sum

- This variant takes a number from each set (S_1, S_2) which sums up to K.
- Same approach: $S_1 = \{ \quad \}$, $S_2 = \{ \quad \}$

Pointer l (at start of S_1), pointer n (at end of S_2)

Subset Sum

- Find subset of elements that add to K.
- Standard Brute force is 2^n possible subsets
- Optimized version
- Divide into 2 sets equally
- generate all possible sums for each half, creating sets A_1, A_2. These new sets have $2^{n/2}$ elements.
- Combine: Use two set Pair Sum on A_1, A_2 to see if any sum from left + right is equal to K. Total t.c is $O(2^{n/2} \cdot n)$

↑ Exponentially Better

Pair Sum - Main logic → Two Set Pair Sum (S,K)

```
def  pairsum(s ,k)
     n = |s. length |
     L = 1
     r = n
     t = s[L] + s[n]

     if  t = = k:
          return (s[L], s[n])
     elif  t > k
          s[n] = s[n-1]
     elif  t < k
          s[L] = s[L+1]

   return  none
```

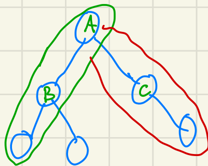
```
• s1. sort()
  s2. sort()

while  l <= n  and  r >= 1:
     t = s1[l] + s2[r]
       if  t = = k:
            return (s1[l], s2[r])

       elif  t < k: l += 1

       else     r -= 1
   return none
```

Longest Tree Path - Tutorial

Initial Ideas:

- Split subtree based on nodes, split
  - ↳ or subtrees with x amt of edges
- From each subtree we need to calculate longest edge, then add them
- recursive call should be reducing subtree space while counting edges
- about 2 furthest apart nodes

- Key Idea is 2 cases: max height is either through root node or child path



length = best child + second best child + 2

Quiz 2

1) - Pile of gems
- Can only compare pairs:
- only care about min/max

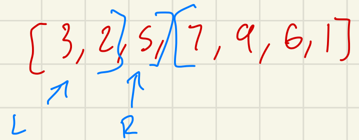
Pseudocode:

```
function   minmaxdnc(A, L, R):
    # Base case
    if L == R:
        return (A[L], A[R])

    if A[n] == 2:
        if A[l] < A[r]:
            return (A[L], A[R])
        else
            return A[r], A[l]

    mid = n//2
    (minL, maxL) = min_max_dnc(A, left, mid)
    (minR, maxR) = min_max_dnc(A, mid, right,)

    overall_min = min(minL, minR)
    overall_max = max(maxL, maxR)

    return (overall_min, overall_max)
```

no min/max

Divide

Key Ideas: compare 2 at a time
Keep reducing sample size
and compare overall at end



Q2) list with both pos and neg
Keep reducing subspace and comparing
Base case if list is 1 element
strictly greater

[3, 7, 6, -2, -5, 3, 2, 4]

[3, 7]

cur_count: 1

```
Solve (A, L, R)
    if L == R:
        return(best=1, pref=1, suff=1, first, last)

mid = (l+r)//2

size L = mid-L+1
size R = R-mid

if lastL < firstR:
    cross = suffL + prefR
else:
    cross = 0
```

Q1)

```
def   min_max_dnc( A, l, r ):
        if len(A) == 1:
            return ( A[1], A[1] )
        if len(A) == 2:
            return (min(A[1], A[2]), max(A[1], A[2]))

    mid == (len(A)//2 )

    (minL, maxL) = min_max_dnc(A, L, mid)
    (minR, maxR) = min_max_dnc(A, mid, r)

    global_max = max(maxL, maxR)
    global_min = min(minL, minR)

    return (global_min, global_max)
```

- where l and r are indexes

Q2)
- recursively call current elem, + next elem to be checked, current longest streak
- Strictly greater base case
- Base Case: if len(A)==1:

```
def  max_increase_dnc( A, cur_elem, next_elem, longest_streak)
    if len(A) ==1:
        return 1
    longest_Streak =0

     if cur_elem < next_elem:
         longest streak +=1
         return  max_increase_dnc( A, cur_elem +1, next_elem +1, longest_streak)
     else:
         return  max_increase_dnc(A, cur_elem +1, next_elem+1, longest streak)

     if  Leftlist [r] < Rightlist [L]:
         longest_Streak = longest_streak + len(rightlist)

  return   longest_Streak
```

Q3)

```
def name_dupe(A, l, r)
    if len(A) <= 1:
        return False
    mid = n//2:
    (dupeL, SetL) = nape-dupe(A, l, mid)
    if dupL: return True

    (dupeR, SetR) = nape-dupe(A, mid+1, r)
    if dupR: return False

    if SetL intersects SetR
        return True:

    return (False, {A})
```

Key Ideas:

Check left side and right side for dupes then crossover between the 2.

Q4) Key Ideas: Split left and right then add up Even and odd globally then Final compare

-recursively Call smaller subset by reducing r / increase mid

```
def majority_even_odd(A, l, r)
    Majority = none
    if len(A) == 1:
        is_even = A[l] % 2
        if is_even == True:
            return (1, 0)
        else:
            return (0, 1)
    mid = ((l+r) //2)
    (even_r, evenL) = majority_even-odd(A, l, mid)
    (odd_r, odd_L) = majority_even_odd(A, mid+1, r)

    total_even = Even_r + Even_l
    total_odd = odd_r + odd_l
    if total_even > total_odd:
        return "Even"
    elif total_even < total_odd:
        return "ODD"
    else return "Tie".
```

Unit 3

Greedy Search - Lecture 1

Unit 3

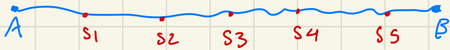
- greedy problems are optimization problems defined by set of potential solutions, feasibility constraint, target function and goal of finding min/max value.
  - ↳ how good proposed solution is

Roadtrip Problem

- roadtrip electrical vehicle, minimizing number of charging stops
- greedy strategy: Sort charging stations based on distance from start.

From current location: repeatedly drive to furthest reachable station before stopping

- Complexity is: $O(n \log n)$ for initial sorting, $O(n)$ for selecting



R is distance most distance you can travel: S_j - current_position $\leq R$

- t is next gas station not reachable, $t-1$, most reachable station.

Proving optimality using Induction

- greedy algos are easy to design but hard to prove it's optimal solution
- to prove optimality: argue locally optimal choice is part of globally optimal solution, when greedy choice is made, remaining problem is optimal

Proof by Induction

- Base case: destination is reachable in single charge. (zero stops)
- Inductive Hyp: Assume it works for size K
- Inductive Step: Show it works for K+1

Optimal Solution Proof

- n is number of stations

- Start at x (thunder bay) to d (montreal). Drive to furthest station you can reach, then repeat.

1) Base case: $n=0$, best case no stops needed, straight drive is enough 0 stops or no charging stations
2) Inductive Hyp: Assume: for any trip with k or less stations between start and destination
3) Inductive Step: if theres $K+1$ stations, greedy picks furthest reachable S_1 not S_{1+1} given $O = \{o_1, o_2 \dots d\}$, the most optimal route o_1 cant be past the furthest station. $o_1 \leq S_1$

Key Idea: Stopping at S_1 gives at least as much as o_1, you can create another plan with same same amount of stops $\{S_1 - d\}$ plus optional remainder

Exchange Argument

- Replace o_1 with s_1

  Define : $o' = (s_1, o_2 \ldots d)$

If o_2 reachable from o_1,
$s_1 \geq o_1$, then s_1 gives
atleast as much reach :
$|o'| = |o|$

General Greedy Paradigm

- Sort objects based on criteria (specific)
- Repeatedly Select next item in Sorted list,

if it does not violate Constraint. Continuing until no more Possible Selections.

If there exists feasible solutions, there is atleast 1 optimal

Matroids Theory?

```
def   roadtrip( i, Stations[])
        n = len(stations) - 1
        t = i + 1

        while  t < n  and  reachable( t, Stations[t+1])
                t += 1

                if t == n
                    return  n

                else
                  return  [t] +  roadtrip( t, Stations)
```

Greedy Algos 2

Optimization Problems

- 4 main parts
  → set of potential solutions
  → feasibility constraint
  → target function (to maximize or minimize)
  → goal to find the best value

Feasibility Constraint?

- No two activities can overlap ($start_i \leq start_j < finish_i$)
- Can't choose x_1 and x_2
- Can choose x_4 and x_6

	x_1	x_2	x_3	x_4	x_5	x_6
Start	8:00	8:30	9:50	13:00	13:20	14:30
Finish	10:00	9:20	10:40	14:30	14:10	16:00

To find the options that satisfy the feasibility constraint by sorting, then iterate and check if satisfied.

Activity Selection Problem

- maximize total number of activities selected from set A, each activity has a start time s_i and end time c_i.

- Feasibility Constraint: - No 2 activities can overlap, start of new activity must be ≥ (later/after first activity finishes) ($start_j \geq finish_i$)
- Sorting Strategies: earliest start time, shortest length, longest length, optimal is sort by earliest end time.
  ↳ Finishing early leaves max room for other activities.

Greedy Algorithm Pseudocode

- Sort activities A by end times
- Add first activity a1 to solution set S. current_end = a1 finish time
- For each next activity, if start ≥ last finish, choose it.

- Iterate through other activities if its start time is ≥ current_end add it to S and update current end.

Optimal Proof

- Base case $n=1$, only time for 1 activity
- inductive hyp: assume $k+1$ activities
- inductive step: greedy chooses earliest finish time

  $F(o_1) \geq F_1$, because a_1 finishes earliest

  replacing o_1 with a_1,

  a_1 finishes no later than o_1, so there must be anything compatible with o_1 is compatible with a_1.

  removing all overlapping activities, less than $< k$ activities remaining

  $F_1 \leq F(o_1)$

  The finish time can only be as quick as optimal finish time.

Note: if activities have same duration, choosing earliest finish time = choosing earliest start time. Because finish time = start time + constant, so EFS, is dependent on start time

```
def activity Selection (activities):
    sort activities by ascending finish times
    queue = [ ]
    last finish = -∞

    for every activities:
        if Finish_i ≤ Start_j
            append to queue
            last_finsh = finish_j
        else
            break
    return queue
```

Least overlaps, does not guarantee optimality because it may block shorter activities.

If activities carry different weight, can't use greedy must use dynamic programming.

Greedy Algos 3

Coin Change Problem

- Provide exact change using fewest possible coins
- Strategy: repeatedly select max coin value that is less than or equal to remaining amount.

→ Feasibility Constraint: coin value ≤ remaining balance

or current coin value ≤ remaining balance

- pseudocode: while remaining amount (r) greater than zero, find largest coin value ($v \leq r$) add it to solution and subtract from remaining balance.

- limitations: greedy approach works with Standard Canadian money but may fail with "weird" coin values. Or if we don't have unlimited coins

→ Take largest coin ≤ m

→ subtract it

→ Repeat untill remainder = 0

Pseudocode:

```
change = [    ]
remaining = m
while remaining_balance > 0:
    V = max coin value
    change += V
    remaining_balance = remaining_balance - V
return change
```

Optimality Proof induction

1) Base Case:

↳ check small amounts manually $m < 5$, best pennies

...... if $5 < m < 10$, use pennies + nickel.

.... if $m = 200$ → 1 toonie

2) Inductive Hyp:

↳ greedy gives optimal solution for every value ≤ K

where $K > 200$. assume works up to k

3) Inductive Step:

↳ Since $K+1 > 200$: greedy must choose a toonie first, $S = \{200, rest\}$, after removing 200,

$K+1-200 \leq K$, remaining part, next page

Compare with any optimal solution

let O be optimal for k+1,

any optimal solutio must contain toonie due to highest value,

if k+1 − 200 ≤ k, remaining part must be optimal for smaller value.

Platform Assignment

of overlaps is # of platforms needed

n (at, dt)

k platforms, always enough for n trains

- minimum k platforms, means as many trains n assigned as possible

1) Sort trains by arrival time
2) keep track of platform availability,
   ↳ min heap to queue up, which platforms free up fastest

Algorithm

1) platform that frees up earliest
2) if that departure < current arival:
   → reuse platform
3) else make new platform, add this depart time to heap

Greedy Algos 4

Knapsack problem

- A bag with capacity C
- Items each with a value and weight, pick items to maximize total value, without exceeding weight capacity C.

- 0/1 Knapsack Problem: you must take a full item or leave behind. no greedy algorithm will be optimal for this, implication of no Polynomial time algorithm.

↳ Hard to: balance high value items, low weight items, capacity → solved by dynamic programming

- Fractional Knapsack: if you can split items greed works, pick best value/weight ratio first.

Greedy Idea:

- Calculate unit value $= \frac{value}{mass}$, sort these items
- Keep choosing best unit valued item until $\leq$ Capacity of bag
- Take full if possible, if not, take as much as possible
- Stop when capacity = full

Optimality proof

- if item does not take as much as possible of highest value item, if there is anything of lesser value, it can't be optimal
- Base case: $|A| = 0$, no objects
- Inductive Hyp: assume FKS finds optimal solution when $\leq K$ objects where $0 \leq K$.
- After reducing problem, it works locally with smaller subset.

$S = \{p_1, \dots p_{K+1}\}$

$O = \{o_1, \dots o_{K+1}\}$

First choice has to be same, if $o_1 < p_1$, then there exists $o_j > p_j$, any weight from item j would not change best value increase

Quiz 3 Questions

T/F Questions

1) True, would mean you can't finish the trip ✓
2) False, Finishing early still matters, to start next activity
3) False, always take highest ratio possible ✓
4) True, coin system not canocical, would mess up how cents are calculated ✓

Pseudo codes

1) mission — deadline (days)
   - → amt of gems
   - → 1 day to complete
   - → 1 mission at a time

Maximize target function for gems.

Constraint: Choose next soonest deadline, if it can be completed before it expires, if tie then choose higher gem mission.

Pseudocode

Sort missions by gems (descending)
Let m = max deadline
Create scheduling queue
  For every mission in sorted list
    For day from d to 1
      if queue[day] empty:
        highest gem mission there
        add g total
        break

Ideas

- Sort gems,
- schedule "best" missions on last possible days
- if 2+ missions have same deadline, choose higher gem one

Every mission = same duration
latest available day ≤ deadline.
work backword from deadline.
Leaves more time for earlier deadline missions
Protects tighter deadlines

2) ## Rough Ideas

$A = [3, 5, 1, 4, 2]$

- diff size eggs
- increasing order
- Only reverse K eggs
- K seavence
- Swaps K number of elements,

Find largest elem in list

if already in last index, skip
else flip to front,
then flip to final position

at most 2 flips per elem

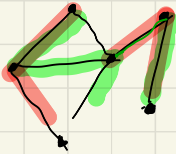
Greedy nature: keeps finding largest remaining.

Pseudo Ideas

- reduce number of K needed to sort list
- Swap front of list in alternations
- Choose largest K value, then keep reversing

function takes in (list)
n = len(list)
Sorted = false
while list is unsorted
for elem in list swap
with K=2
if unsorted
with K=5
Sorted = true
return K seavence

Spanning Tree



Unit 4

Unit 4 - Dynamic Programming Lecture 1

- Premises: way to solve big problem by breaking it down to sub problems and reusing their answers

Steps:

1) Break problem into sub problems
2) Solve each subproblem
3) Store answers
4) Reuse them instead of calculating

* saves you time, from repeated calculations
* reuses calculations

Describe State

- use indices to describe which sub problem we are solving.
- use recurrence relation to describe how to compute bigger state using smaller state

Coin Change Problem - revisited

- give exact change with fewest coins
- greedy strat: keep choosing next highest valued coin.
  - ↳ not always optimal and has limitations (type of currency, number of coins)

Dynamic programming Solution

- Problem is broken into finite decision tree, each node is subproblem.
- Identify identical subproblems solve them to find optimal path
- reccurrence relation: defines optimal solution based on sub problems
  - → $Min_Coins(n) = 1 + \min\{Min_Coins(n - c_1), min_Coins(n - c_2), ... Min_Coins(n - c_k)\}$
  - where $c_1, c_2 ... c_k$ are values of the coins.
- Logic: at min you need 1 coin, then add coins needed for remaining value; 1 + min Coins (v - c)
  - MinCoins(v) = min number of coins needed to make value v,
  - MinCoins(12) → tries all 3 coins 1) 1 + MinCoins(11), 2) 1 + MinCoins(8) 3) 1 + MinCoins(3)
- Base cases: $min_Coins(0) = 0$ and $Min_Coins(x) = \infty$, if $x < 0$
  - ↳ 0 money needs 0 coins
  - ↳ invalid solution, they owe you money

Two table types

- Bottom-up, used when order of subproblems is known, store smaller problems at top
- Top-down, subproblems arent obvious, start from goal recursively explore
- Current state = Function(smaller states)

Time complexity is $O(K^n)$, recursion is slow, because it builds a huge tree. But this gives an optimal solution always.

- Store answers once in table, $O(nK)$ · where K is number of coin types, n = change

Dynamic Programming Solution

```
def Min_Coins(n):
    Set A[0] = 0
    for i = 1 to n:
        A[i] = 1 + min(getval(A,i-1),getval(A,i-4), getval(A,i-9))
    return A[n]
```

A	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
val	0	1	2	3	1	2	3	4	2	1	2	3	3

Actual T.C using DP is $O(n)$

Traceback: $A[12] = 1 + A[8]$, what coins created the answer
$12 \to 8 \to 4$

5 Steps using Dynamic Programming

1) Find a recurrence relation and base cases
2) Determine parameters that define each subproblem
3) Define table → store optimal solutions
4) Determine order → fill table ensure subproblems solved before larger ones
5) Extract solution → working backwords or storing extra information

- Time complexity: $O(Kn)$, K is number of coin types
- Traceback: extracting coins from table is $O(n)$.

- Once m gets big enough, answer will always use 9. greedy works after 9.
- DP solves small cases, greedy works for big.

Dynamic Programming - 2

Problem Board Cutting

- Cut a board of length n into smaller pieces to achieve the maximum possible profit, based on provided price table

- Brute force by checking all n-1 possible cut points, resulting in exponential time comple
  - ↳ time complexity: $O(2^{n-1})$

- greedy Strategy: not always optimal, getting most valuable piece may not leave room for smaller pieces with better selling prices.

To solve any length n, try every first cut. Pick max value.

1) Recurrence Relation: $dp[n] = max(price[i] + dp[n-i])$, first piece value + best value of remaining
   → Start with some first cut, look at all possible options, pick max of the remaining board.
   - Base cases: max_value (0): no wood left = no money   dp[0]=0
     max_value (x) = $-\infty$ if $x < 0$, invalid board cut

Pseudocode:

```
cut-wood(P, n)
  r = [ ]
  add 0
  K = len(P)
  For every move
      cur_max = 0
      for j from 1 to K
          cur_max = max(current_max, p[j], +r[i-j])
      add to r
return r
```

create list
loop through board size
best ever first cut
compute value
keep best
store answer

2) Define Parameters: Subproblems are defined by single parameter: the board size

3) Define Table: only one parameter: 1D table used to store optimal profit for every length 1 to n.

4) Order of Filling: table is filled in increasing order of board length, ensuring solutions for smaller boards are available, when calculating max value for larger boards.

5) Extract Solution: determine specific cuts made, algorithm stores length of first cut, for each optimal subproblem, then easy reconstruction.

2 Methods

- $Price[i] + dp[n-i]$
- $dp[i] + dp[n-i]$

Defining a table based on the board size

- What is the optimal cutting for a board of length 12?

Length	1	2	3	4	5	6	7	8	9	10
Selling Price	1	5	8	9	10	17	17	20	24	30

A	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
val	1	5	8	10	13	17	18	22	25	30	31	35

Simplified Approach

- double for loop, first through each length 1-n to Compute best first cut, then second time with n+1, to find next highest value.

Value is best price for that specific board length which uses best first cut + best remaining

If finding val for certain board length for 6. 1-5 are filled, try all first cut + best remaining and take max

Pseudo Code: max_val(n) = max { selling price( ) + max_val(n-1) [best first cut + best remaining]
Selling price( ) + max_val(n-2)
⋮
}

```
r[0] = 0
for i in range(1, n+1):
    best = 0
    for j in range(1, i+1):
        best = max( best_first_cut, price[j] + r[i-j])
    r[i] = best
return r[n]
```

Dynamic Programming 3

Grid Traversal problem

- given grid intersections, each road (edge) has cost, only right and down movements are allowed
- Start top-left (0,0). go to bottom right (m,n). Find minimum total cost.
- only can choose paths with down/right.
- Naive way: trying every possible path. Slides grow at $O(2^{m+n})$

Greedy Approach

- treat grid as graph, each intersection = node, each road = edge with cost
  - ↳ expand node with smallest current distance from start
  - → works for only positive edges, uses priority queue $O[E \log V]$
  - → dist[neighbor] = min(dist[neighbor], dist[current] + edge_cost) → Cheapest place we can reach next

DP Approach

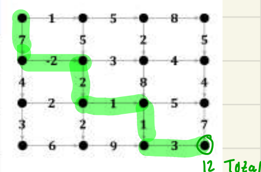
Base case: min(0,0)

- dp[i][j] = min cost to reach cell (i,j)
  - → must come from (i-1, j) → above
  - (i, j-1) → left
- each cell is best path to neighbour + edge cost
- O(mn) time (grid size)

```
dp[i][j] =
min(
    dp[i-1][j] + cost_down
    dp[i][j-1] + cost_right
)
```

Building DP Table

Row/Column	0	1	2	3
0	0	1	6	14
1	7	5	8	12
2	11	7	8	13
3	14	9	9	12



Complexity: O(mn)

Table only tells min cost to reach that cell, need to trace backwards to find the path. (Follow neighbour that recreates val)

Dynamic Programming - 4

The Knapsack Problem

- Each item has weight and value. The capacity is limited by backpack
- Choose items s.t total weight ≤ capacity while maximizing value
- 0/1 problem only chooses full items. Total 2^n possibilities. 20 items = 1 million combinations
- dynamic programming uses smaller decisions, by taking or not taking items, becoming smaller problem.

Dynamic Programming Solution

→ 2 main cases

Case 1: include item n

- value gained = value[n]
- remaining capacity = c - weight[n]

Then solve smaller backpack
items: 1 ... n-1
capacity: c - weight[n]

value[n] + best_value(n-1, c-weight[n])
(best_value(n-1, c-weight[n]) = best remaining)

Case 2 - Dont include

- best_value(n-1, c)

Combine and take max

- Take larger of 2:

Maxval(n, c) =
max(
value[n] + best_val(n-1, c-weight[n])
best_val(n-1, c)
)

- Every subproblem is defined by **number of items considered and remaining capacity.**

Base cases: if either n = 0 (no more items) or capacity = 0 (full backpack)

item	1	2	3	4	5	6
mass	7	4	6	4	3	5
value	100	70	40	80	25	80

Row/Column	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	100	100	100	100
2	0	0	0	70	70	70	100	100	100	100
3	0	0	0	70	70	70	100	100	100	110
4	0	0	0	80	80	80	100	150	150	150
5	0	0	25	80	80	80	105	150	150	150
6	0	0	25	80	80	80	105	150	160	160

→ mass / capacity

↑ items

Max_Val(n,c) = max{
v_n + Max_Val(n − 1, c − m_n)
Max_Val(n − 1, c)
}

(between rows 5 and 6, last column:) diff, means take item 6

14

Complexity: O(n × c)

T.B: O(n)

Still NP-Complete, depends on c

$O(n \times 2^n)$

Path [ ]
cur best path
while cur =! 1
cur = prev[cur]
reverse(path)

Traceback: take last column find difference in entry, means you took it.

Tutorial - Longest Common Subsequence

- find longest sequence of characters that appear in both strings and in the same relative order
- order matters, skipping is allowed

Dynamic Programming

- DP[i][j] = length using first i chars of s_1 and j chars of s_2
- Base cases: no sequence: DP[0][j] = 0, DP[i][0] = 0
- at each match take it or leave it.

Ex Seq 1: x = ABCEGSAB
y = BEAGSCD

Answer = BEGS → Length = 4

- Time complexity: O(n · m)
- Space complexity: O(n · m)

Ex x = ABC
y = AC

	""	A	C
""	0	0	0
A	0	1	1
B	0	1	1
C	0	1	2

Compare $s_1[i-1] = s_2[j-1]$

if equal add it

DP[i][j] = 1 + DP[i-1][j-1]

else:

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

Diagonal match

Otherwise move max(up, left)

(ignoring char from s1) or (ignoring char for s2)

Dynamic Programming Quiz 4

True / False

1) True - it remains valid because it calculates cost to reach that point, accounting negative weights
2) False - if you take first match, you may block future longer subsequence
3) True - if K is polynomial, the DP solution runs in O(n·K) times

Pseudocode — Q1

Ideas: cell (mass, radius)

Consumable if $mass_1 > m_2$, $r_1 > r_2$

return max number of cells eaten

```
Sort by mass and radius ascending order
DP = 1·n  → each cell on its own, chain=1
Parent cell = [-1]·n

for i in range(n)
    for j in range(i)
        if j.mass < i.mass and j.rad < i.rad
        dp[i] = max(dp[i], dp[j]+1) →

        if DP[j]+1 > DP[i]  → chain can be extended
            DP[i] = DP[j]+1
            Parent[i] = j
max_len = max(DP)
```

dp[i] i≥0: 1 + max(dp[j] for all j < i)
if j.mass < i.mass
and j.rad < i.rad

dp[j] + 1 > dp[i]

min (dp[i] , dp[i-1] + cost)

Traceback

```
Chain = [ ]
best-end = max_index(dp)
cur-end = best-end
while cur-end != 1

    chain.append(cur-end)
    cur-end = prev[cur-end]
return (max(dp), chain)
```

Pseudo code #2

Key Ideas: allowed to cut words into smaller words but must be palindrome
- want to use as less cuts as possible

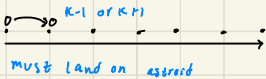
At every index i
- assume worst (cut everything)
- loop through all j
- if j==0: no cuts
  else DP[j-1] + 1
  take minimum

```
n = len(s)
DP = [0]·n   → min cuts for s[0→i]

for i in range(n):
    DP[i] = i   → worst case make i cuts

    for j in range(i+1):   → where last piece starts
        if is palindrome(s, j, i)   → check substring
            if j==0:
                DP[i] = 0   → whole substring s[0→i] is palindrome
            else:
                DP[i] = min (DP[i], DP[j-1] + 1)
                    ↳ take best cuts before j, add 1 for this split
return DP[n-1]
```

Traceback:

```
i = n-1
cuts = [ ]

While i>=0,
    for j in range(i+1)
        if is palindrome(s, j, i)
            if j==0 or DP[i] == DP[j-1] + 1:
                cuts.append(s[j:i+1])
                i = j-1
                break
    cuts.reverse.
```

Debug



Bug: for k in range assumes distance between astroids are dependent on n (length of astroid list), which is false

Fix: use proper distance asteroids[-1]

2) elif num[i] <= nums[i-1]:
this is a bug, because if equal it should not be allowed in wiggle sequence.

Bug 2: up = down +1, down
assigns a tuple instead of number

Should just be up = down +1
down = up+1

Unit 5

Branch and Bound Unit-5

- Paradigm for optimization problems. Try all possibilities but skip clearly bad ones.
- Branch = split problem into choices
- Bound = estimate how good a choice would be. If worse than found solution, stop exploring path.
- Lower bound: min possible cost of any solution fully extends partial solutions. LB = current cost + guaranteed future
- Upper bound: estimate of best possible extending P, often found using greedy / good guess
- global upper bound: Cost of best solution discovered so far
- goal: minimize search space to optimal solution, move lower bound up by finding better solution, and upper bound tells us if this branch is worth exploring. (Locally prune branches)

Algorithm Steps

1) Initialize set of partial solutions (Lib) with first solution
2) Each iteration choose P from Lib that is the best current solution
3) If P is full solution, return it
4) Extend P by one direction
   - ↳ if lower bound ≤ GU, add it (promising path)
   - ↳ if upper bound ≥ GU, update GU (better complete solution found)

Shortest Trail

- goal: minimize numeric target function of distance
- Feasibility constraint: total time ≤ 9 hours
- Labels [a, b], where a is lower bound, b is upper bound
- Computing bounds on blue being optimistic, orange being full possible time
- Path is pruned if optimistic time exceeds 9 hours, or worse than current best

Template

```
def branch_bound(S):
    Lib = { }
    Fill lib with partial solutions          } Keep track of partial solutions

    while lib not empty
        find P s.t P minimizes lower bound   ] if full solution found
        if P is full solution return
        else
            Lib.Pop                                      ] Keep looking
            for all P that exted P      ---> if better partial solution found
                if lower bound ≤ Global upper bound
                   add P to lib
                if upper(P) < GU  ->  better solution   ---> found better complete solution
                   GU = upper(P)
```

Branch and Bound 2

Knapsack Problem - 0/1

- Description: given items with weights and values, trying to maximize value, while staying under capacity
- empty set, then branch decision trees taking A or not etc
- lower and upper is each node having its own bound
- Prune if upper(P) ≤ GU
- Representation: Partial solution represented as sequence of 0's and 1's.
- Lower bound: cost so far (total value of items decided not to take) + guaranteed future costs (value of items left out due to capacity constraint)
- Upper bound: use greedy heuristic, based on unit value (value/weight)

Walkthrough maximize version

- lower bound = current value taken
- upper bound = fill remaining capacity with best remaining items
- Take item 1: LB = 5 (val of item), say leaves 7 weight, fill best remaining: upper bound = 22.
  Then compare current best with global best

Minimize problem

- lower(P) = cost left out + value forced to lose
- upper(P) = cost left out + what you could lose

Improved lower bounds

- techniques to improve pruning by grouping into disjointed sets, individually exceed remaining capacity, forcing at least one item to be dropped.
- Causing more pruning, by finding conflicts between items

Branch and Bound 3

Job Assignment

- assigning n jobs to n people to minimize total time. Each person does exactly one job, intelligently prunes space from $O(n!) \rightarrow$
- greedy is not optimal

Key Ideas

- Build decision tree to assign next job, upper bound: assign each person lowest number job not assigned then greedily choose next best remaining. Lower bound = min possible time for every unassigned job
- For table choosing each job best person, could mean 1 person gets assigned twice, assigning by person, may have same job
- Calculate GFC per job/per person. minimum cost you guaranteed to pay for remaining assignments
- these give tighter baseline and upperbound uses greedy approach
- Any partial solution whose LB ≥ GU, pruned

CSF: what youve already added
GFC: best case future (everything remaining is as cheap as possible) - lowest possible cost, if we ignore constraints
FFC: actual complete solution, ussually greedy
LB: CSF + GFC
UP: CSF + FFC

Branching choose next item, check if LB, UB still possible
If LB = UB, you can prune

Quiz 5: Branch and Bound

1) True, if bounds are correct and search is exhaustive BnB is optimal
2) True, Feasible future cost uses greedy to estimate upper cost
3) False, BnB can be as slow as brute force

Bound Computation Questions

Objective: Cheapest possible path from Base to 3 different planets, back to base.

Initial Partial Solution: (1,,, 1)

a) for initial partial solution. Cost so far = 0.

GFC: Choose next smallest each time 8, 10, 12 → 1. Then back home → move logic to FFC

Nova, Vega, Echo

take min of each column because, we will have to move from each planet atleast once.

min (8, 12, 10) = 8
min (7, 9, 14) = 7
min (11, 13, 6) = 6
min (9, 12, 5) = 5

GCF = 0 + 26
= 26

LB = 0+26 = 26

FFC: use greedy solution, choose next smallest move every time. Base → Nova → Echo → Vega → Base
8 + 9 + 6 + 9
= 32

UP = 0 + 32
= 32

b) All possible solutions that extend one step:
(1, 2, , 1)
(1, 3, , 1)
(1, 4, , 1)

- Can visit any of the 3 planets first

c) (1, 2, _ _, 1)

LB = CSF + GFC
LB = 8 + 20 min of remaining (9, 6, 5) = 20
= 28

UP = CSF + FFC
UP = 8 + (9+6+9)
= 32

FFC: best move is to echo (9), then echo to vega (6) then vega back to base (9)

Q2) Maximization of value, with C = 7 at most

- Partial Solution: (–, –, – – –),

a) LB = CSF + GFC

CSF = 0 (best of remaining possible) (use fractional knapsack, best unit value item)

GCF = 23 take all of item 1 (highest unit val), take all of 2, take half of item 3

Value: group x1, x2, x3, omit minimum = 14 + 9 = 23

LB = 23

UP = CSF + FCC 0/1 Problem, take item 1, 2, thats it
= 0

FCC = 69 − 34 (how to differentiate?)
= 35

b) (1 - - - -)
(0 _ - - - -)

if we take item 1,
LB = 12 + 23

→ CSF = 12, because we can't take item 5
LB = CSF + GFC
GFC = 9 x2, x3 (18, 14) x3, x4 (14, 9) x2 x4 (18, 9)
(18, 14, 9)

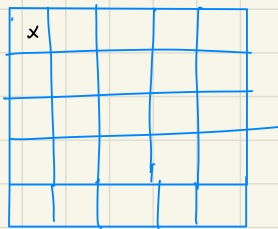
UP = CSF + FCC
FCC = x2
= 35 (69 − 16 + 18)

UP = 47

Branching and Pruning

Q3) n queens on n x n chessboard, S.t no 2 queens cant share row, column or diagonal

5 by 5 instance



a) reperesentation: column of queen in row i (c1, c2, c3, c4, c5)

"_" not assigned yet.

root node "_ _ _ _ _"

b) First 2 iterations would be: c_1, meaning you cant place queen in second col/row, Pos (1,2), (2,1) or (2,2).

So first iteration is anything in column 1

Second iteration was (1,1,,,,)
(1,2,,,,)
(1,3,,,,)
(1,4,,,,)
(1,5,,,,)

Branching Tree (_ _ _ _ _)

iteration 1 → (1,,,,) (2,,,,) (3,,,,) (4,,,,) (5,,,,)

iteration2 → (1,1,,,) (1,2---) (1,3 ---) (1,4,,) (1,5,,,,)

3c) Bound rules are valid so far: LB = UB = 5
if LB = UB = ∞, prune

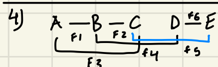
For level 1 all positions are valid

For level 2: if we use (1,....) then Positions (1,3....), (1,4 ....), (1,5....) are valid and bounds are [5,5]

3d) Pruned solutions would be (1,1,,,,), (2,1,...) or (1,2,,,)

---

4) A — B — C D — E



ABC, triangle ∴ min 3 freq

edges = constraints

Multiple vertices can share same frequency as long as they are not connected.

minimizing number of distinct frequencies used

a) (fa, fd, fB, fc, fe)

Each entry is assigned number frequency

"_" unassigned → root (_, _, _, _, _)

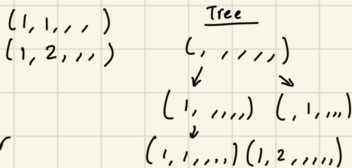
b) Root: (_, _, _, _, _)

Iteration 1: assign colour 1 to any of each vertex:
(1,,,,) (_, 1, ---) ....

Iteration 2: Assign D. D is not connected to A, so we can reuse colour 1 or introduce color 2.

(1,1,,,)
(1,2,,,)

Tree



4c) For our root node, we need atleast 3 colours ABC, triangle

Greedy Solution would use (1, 1, 2, 3, 2)

Bound: [3,3]

For all partial solutions bound is [3,3]

4d) All solutions optimal, proved you cant do better no benefit of expanding.