All Dynamic Programming

4. Consider the SIGN CHANGE problem. We own a business with a big sign above it, and we need to change the name. It costs $10 to change a letter, and $25 to delete a letter - importantly, it is not possible to insert a new letter. We want to find the minimum cost to do this.

   For example, a sign saying "HELLO WORLD" can be changed to "YELLOWORC" for $70: replace "H" with "Y", delete the space, replace "L" with "C", and then delete the "D".

   Suppose $X = x_1 x_2 \ldots x_n$ is the old sign, and we want to change it to $Y = y_1 y_2 \ldots y_m$. Note that $0 \leq m \leq n$. Define a subproblem $SC(i,j)$ as the minimum cost to convert $x_1 \ldots x_i$ to $y_1 \ldots y_j$. Give a recurrence relation for SC. Include the base cases.

$$SC(i,j) = \begin{cases} 25\,i & j = 0 \\ \infty & i < j \\ SC(i-1, j-1) & x_i = y_j \\ \min(10 + SC(i-1, j-1),\ 25 + SC(i-1, j)) & x_i \neq y_j \end{cases}$$

Subproblem: the cost to change / delete a letter on the sign

```
# Dynamic Programming implementation of LCS problem

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

# Driver program to test the above function
X = "AGGTAB"
Y = "GXTXAYB"
print "Length of LCS is ", lcs(X, Y)
```

LCS Algorithm

7. You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts $a_1 < a_2 < \ldots < a_n$, 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 the daily penalties. Give an efficient algorithm that determines the optimal sequence of hotels at which to stop.

Solution: Suproblems definition
Let $OPT(i)$ be the minimum total penalty to get to hotel i.

Recursive formulation
To get $OPT(i)$, we consider all possible hotels j we can stay at the night before reaching hotel i. For each of these possibilities, the minimum penalty to reach i is the sum of:

- the minimum penalty $OPT(j)$ to reach j,
- and the cost $(200 - (a_j - a_i))^2$ of a one-day trip from j to i.

Because we are interested in the minimum penalty to reach i, we take the minimum of these values over all the j:

$$OPT(i) = \min_{0 \leq j < i} \{OPT(j) + (200 - (a_j - a_i))^2\}$$

And the base case is $OPT(0) = 0$.

Pseudo-code

```
// base case
OPT[0] = 0
// main loop
for i = 1...n:
  OPT[i] = min([OPT[j] +(200 − (a_j − a_i))^2 for j=0...i-1])
// final result
return OPT[n]
```

Complexity
- We have n subproblems,
- each subproblems i takes time O(i)
The overall complexity is

$$\sum_{i=1}^{n} O(i) = O(\sum_{i=1}^{n} i = O\frac{n(n-1)}{2} = O(n^2)$$

```
# Return the maximum size of
# substring of X which is
# substring in Y.
def maxSubsequenceSubstring(x, y, n, m):
    dp = [[0 for i in range(MAX)]
             for i in range(MAX)]

    # Initialize the dp[][] to 0.

    # Calculating value for each element.
    for i in range(1, m + 1):
        for j in range(1, n + 1):

            # If alphabet of string
            # X and Y are equal make
            # dp[i][j] = 1 + dp[i-1][j-1]
            if(x[j - 1] == y[i - 1]):
                dp[i][j] = 1 + dp[i - 1][j - 1]

            # Else copy the previous value
            # in the row i.e dp[i-1][j-1]
            else:
                dp[i][j] = dp[i][j - 1]

    # Finding the maximum length
    ans = 0
    for i in range(1, m + 1):
        ans = max(ans, dp[i][n])
    return ans

# Driver Code
x = "ABCD"
y = "BACDBDCD"
n = len(x)
m = len(y)
```

```
# Python 3 program to check if
# a string is substring of other.

# Returns true if s1 is substring of s2
def isSubstring(s1, s2):
    M = len(s1)
    N = len(s2)

    # A loop to slide pat[] one by one
    for i in range(N - M + 1):

        # For current index i,
        # check for pattern match
        for j in range(M):
            if (s2[i + j] != s1[j]):
                break

        if j + 1 == M :
            return i

    return -1

# Driver Code
if __name__ == "__main__":
    s1 = "for"
    s2 = "geeksforgeeks"
    res = isSubstring(s1, s2)
    if res == -1 :
        print("Not present")
    else:
        print("Present at index " + str(res))
```

```
import sys
INF = 1000000000

"""
I worked on this assignment with Anne Glickenhaus, Miguel N. No code was copied, but ideas were shared.
I also used Geeks for Geeks a lot on this assignment to understand the concepts more and be able
to create the code.
"""

def main():
    input_file = sys.stdin

    num_of_items = int(input_file.readline())
    spend_exactly_amt = int(input_file.readline())
    sys.setrecursionlimit(spend_exactly_amt)

    items = []
    all_costs = []
    all_calories = []
    for line in input_file:
        cost, calories, food = line.strip().split(" ")
        amount = 0
        items.append([int(cost), int(calories), str(food), int(amount)])
        all_costs.append(int(cost))
        all_calories.append(int(calories))

    min_cal, min_cal_list, can_spend_exact = chipotle(items, spend_exactly_amt, num_of_items)


    if can_spend_exact == True:
        curr_cost = 0
        num = mem_chipotle(spend_exactly_amt, all_costs, all_calories, num_of_items, curr_cost)

        reconstruction = reconstruction_fun(min_cal_list, items, spend_exactly_amt, num_of_items)

        print('Possibly to spend exactly: {0}'.format(spend_exactly_amt))
        print('Minimum calories (iteratively): {0}'.format(min_cal))
        print("Minimim calories (memoization): {}".format(num))

        dict_food = {}
        for cost, calories, food, amount in reconstruction:
            if food not in dict_food:
                dict_food[food] = 1
            else:
                dict_food[food] += 1
        # print (dict_food)
        for food, value in dict_food.items():
            print ("{} {}".format(food, value))
```

```
def reconstruction_fun(min_cal, items, spend_exactly_amt, num_of_items):
    """
    This is a function to go through the matrix and see what food items were purchases in order
    to spend that amount and get that calorie number. This returns a list of all the items used.
    """
    bought_food = []

    i = num_of_items
    j = spend_exactly_amt
    while i > 0 and j != 0:

        if min_cal[i][j] != min_cal[i-1][j]:
            bought_food.append(items[i - 1])
            j -= items[i - 1][ 0]
            items[i-1][3] += 1

        else:
            i -= 1

    return bought_food
```

```
def maxRevenue(m, x, revenue, n, t) :

    # Array to store maximum revenue
    # at each miles.
    maxRev = [0] * (m + 1)

    # actual minimum distance between
    # 2 billboards.
    nxtbb = 0;
    for i in range(1, m + 1) :

        # check if all billboards are
        # already placed.
        if (nxtbb < n) :

            # check if we have billboard for
            # that particular mile. If not,
            # copy the previous maximum revenue.
            if (x[nxtbb] != i) :
                maxRev[i] = maxRev[i - 1]

            # we do have billboard for this mile.
            else :

                # We have 2 options, we either take
                # current or we ignore current billboard.

                # If current position is less than or
                # equal to t, then we can have only
                # one billboard.
                if (i <= t) :
                    maxRev[i] = max(maxRev[i - 1],
                                    revenue[nxtbb])

                # Else we may have to remove
                # previously placed billboard
                else :
                    maxRev[i] = max(maxRev[i - t - 1] +
                                    revenue[nxtbb],
                                    maxRev[i - 1]);

                nxtbb += 1

        else :

            maxRev[i] = maxRev[i - 1]

    return maxRev[m]
```

```
def mem_chipotle(spend_exactly_amt, all_costs, all_calories, num_of_items, curr_cost):
    """
    This is the memoization version to find the min calories while spending that amount.
    """
    if num_of_items == 0 or spend_exactly_amt == 0:
        return 0

    if (all_costs[num_of_items-1] > spend_exactly_amt):
        # curr_cost = add_amount(curr_cost, all_costs[num_of_items-1])
        cal_amt = mem_chipotle(spend_exactly_amt , all_costs, all_calories[num_of_items-1] , num_of_items-1, curr_cost)

    else:
        cal_amt =  max(all_calories[num_of_items-1] + mem_chipotle(spend_exactly_amt - all_costs[num_of_items-1],
                        all_costs, all_calories , num_of_items-1, curr_cost),
                        mem_chipotle(spend_exactly_amt , all_costs, all_calories,
                        num_of_items-1, curr_cost))

    return cal_amt
```

```
def chipotle(items, spend_exactly_amt, num_of_items):
    """
    This is the iterative method to find the minimum number of calories
    while spending that exact limit. To keep track of if this was done
    correctly or not, I have a bool can_spend_exact which is return and
    used in main to see if it is possible or not.

    It also returns the minimum calories, and returns the matrix.
    https://www.geeksforgeeks.org/minimum-cost-to-fill-given-weight-in-a-bag/
    """
    can_spend_exact = False # change this bool if can spend exactly money
    food_bought = []

    min_cal = [[0 for i in range(spend_exactly_amt + 1)] for j in range (num_of_items + 1)]

    for i in range(spend_exactly_amt + 1):
        min_cal[0][i] = INF

    for i in range(1, num_of_items+1):
        min_cal[i][0] = 0


    for i in range(1, num_of_items + 1):
        for j in range(1, spend_exactly_amt+1):
            cost = items[i-1][0]
            cal = items[i-1][1]

            if (cost > j):
                min_cal[i][j] = min_cal[i-1][j]
            else:
                min_cal[i][j] = min(min_cal[i-1][j], min_cal[i][j-cost] + cal)

    if(min_cal[num_of_items][spend_exactly_amt] == INF):
        can_spend_exact = False
    else:
        can_spend_exact = True
        min_cal_amt = min_cal[num_of_items][spend_exactly_amt]


    min_calories = min_cal[len(items)][spend_exactly_amt]

    return min_calories, min_cal, can_spend_exact
```

Unbounded Knapsack

You are given a string of *n* characters *s*[1... *n*], which you believe to be a corrupted text document in which all punctuation has vanished (so that it looks something like "itwasthebestoftimes..."). You wish to reconstruct the document using a dictionary, which is available in the form of a Boolean function dict(.): for any string *w*,

$$dict(w) = \begin{cases} true & \text{if } w \text{ is a valid word} \\ false & \text{otherwise.} \end{cases}$$

(a) Give a dynamic programming algorithm that determines whether the string *s*[.] can be reconstituted as a sequence of valid words. The running time should be at most $O(n2)$, assuming calls to dict take unit time.

(b) In the event that the string is valid, make your algorithm output the corresponding sequence of words.

a)

In order to check whether the given string *s* is a sequence of valid words or not using dynamic programming algorithm, first define sub problems as follows:

Assume *S*(*i*) is a sub problem and it is 1 if and only if the string *s*[1…*i*] is a sequence of valid words. Where $0 \leq i \leq n$.

The above sub problem can be solved using the following relation:

$$S(i) = \max_{0 \leq j \leq i} \{S(j): dict(s[j+1...i]) = true\}$$

Algorithm:

function valid-str(*s*)

Input: a string of *n* characters, *s*[1…*n*]

Output: returns 1, if s is a sequence of valid words. Returns 0 otherwise.

If n=0:

return 1

for i=0 to n:

$$S(i) = \max_{0 \leq j \leq i} \{valid - str(s[1...j]): dict(s[j+1...i]) = true\}$$

return S(n)

Proof of correctness:

• Obviously, the empty string is valid, thus when n=0, the algorithm returns 1 and sets *S*(0)=1.

• If *s* is not empty, then *S*(*i*) is set to 1 if and only if *S*(*j*) is a sequence of valid words and for the last word $dict(s[j+1...i])$ returns true. Otherwise, *S*(*i*) will be 0.

• The given string *s* can be reconstructed as a sequence of valid words if and only if *S*(*n*)=1. Thus, the above algorithm finally returns *S*(*n*).

Running time complexity:

• There are O(n) sub problems, as the for loop in the above runs O(n) times. Each sub problem takes at most O(n) time.

Therefore, the running time of the above algorithm is $O(n \times n) = O(n^2)$.

b)

keep track the index of starting of the previous valid word so that the string can be recovered as the partition in words by tracing back these indexes.

Updated algorithm:

function valid-str(s[1…*n*])

Input: a string of *n* characters, s[1…*n*]

Output: returns 1, if s is a sequence of valid words. Returns 0 otherwise.

If n=0:

return 1

for i=0 to n:

$$S(i) = \max_{0 \leq j \leq i} \{valid - str(s[1...j]): dict(s[j+1...i]) = true\}$$

If S(i)==1:

prev(i)=j

return S(n)

If the above algorithm returns 1, then the valid words can be recovered using the global array prev as follows:

function print-valid-words(s[1…n],prev)

Input: a string of *n* characters, s[1…*n*]

And array of indexes of valid words.

Output: prints sequence of valid words

p=prev(n)

if p!=0:

print-valid-words(s[1…p],prev)

print s[p+1…n]

keep tracking the index of starting of the previous valid word takes only constant time. But recovering partitions of the string takes $O(n)$ time.

Thus, the running time of the above algorithm remains $O(n^2)$.

3. Consider the following saved array s computed during an instance of the MATRIX-CHAIN PRODUCT for $n = 7$ arrays. Recall that if $s[i, j] = k$ then the final multiplication for $A_i \cdots A_j$ will be between A_k and A_{k+1}.

the s table →

	2	3	4	5	6	7	s
1	1	1	1	1	4	4	1
	2	3	3	3	4	4	2
		3	3	5	5	5	3
			4	5	4	4	4
				5	6	5	5
					6	6	6

MCP parentheses

Use this table to show the optimal order in which to multiply the matrices

$$A_1 \cdot A_2 \cdot A_3 \cdot A_4 \cdot A_5 \cdot A_6 \cdot A_7.$$

Use parentheses to show the order.

the optimal solution can be constructed from the split table s. each entry s[i,j] = k shows where to split the product $A_i \dots A_j$

$$((A_1)((A_2 \cdot A_3)(A_4))((A_5 \cdot A_6)(A_7)))$$

	2	3	4	5	6	7	s
1	1	1	1	4	4	1	
	2	3	3	3	4	2	
		3	3	5	5	3	
			4	5	4	4	
				5	6	5	
					6	6	

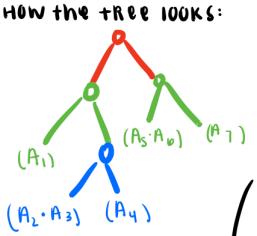
4 = splits it at position 4 then we look at $A_1 - A_4$ and $A_5 - A_7$.

then go to the index s[1,4] and s[5,7] to see where the sub parenthesis split

A_1-A_4 splits @ 1 & A_5-A_7 splits @ 6 and these splits are within the outer brackets

A_2-A_4 = s[2,4] = 3 and then when there are only 1 or 2 within a parenthesis, you are done.

HOW the tree looks:



$$(((A_1)((A_2 \cdot A_3)(A_4)))((A_5 \cdot A_6)(A_7)))$$

HW) switch cities (C & D) based on which is cheaper FOR that month

moving-cost (M) = 10 C = (1, 3, 20, 30) D = (50, 20, 3, 4)

subproblem: [$(c_1, d_1) \dots (c_i, d_i)$] array for the cost of living in and C(i) = min cost ending in city C and D(i) = min cost ending in city D

Recurrance:

$$C(i) = \begin{cases} c_1 & i = 1 \\ \min[C(i-1), D(i-1) + M] + c_i & i > 1 \end{cases}$$

$$D(i) = \begin{cases} D_1 & i = 1 \\ \min[D(i-1), C(i-1) + M] + d_i & i > 1 \end{cases}$$

so: cost of best plan = min[C(n), D(n)]

class 3) given 3 strings. Obtain str.T from reading str.R and str.S

Recurrance: "interleaving"

$$\text{IL}(i,j) = \begin{cases} \text{true} & i=0 \ \& \ j=0 \\ [t_{i+j} = r_i \text{ and IL}(i-1, j)] \text{ or } [t_{i+j} = s_j \text{ and IL}(i, j-1)] & i>0 \text{ and } j>0 \end{cases}$$

Returns T or F

> ***What is difference between memoization and dynamic programming?***

Memoization is a term describing an optimization technique where you cache previously computed results, and return the cached result when the same computation is needed again.

Dynamic programming is a technique for solving problems of recursive nature, iteratively and is applicable when the computations of the subproblems overlap.

Dynamic programming is *typically* implemented using tabulation, but can also be implemented using memoization. So as you can see, neither one is a "subset" of the other.

A reasonable follow-up question is: **What is the difference between tabulation (the typical dynamic programming technique) and memoization?**

When you solve a dynamic programming problem using tabulation you solve the problem **"bottom up"**, i.e., by solving all related sub-problems first, typically by filling up an *n*-dimensional table. Based on the results in the table, the solution to the "top" / original problem is then computed.

If you use memoization to solve the problem you do it by maintaining a map of already solved sub problems. You do it **"top down"** in the sense that you solve the "top" problem first (which typically recurses down to solve the sub-problems).

6. The *longest common subsequence* problem is to find the longest (non-contiguous) sequence of characters shared by two strings $X = x_1 x_2 \dots x_n$ and $Y = y_1 y_2 \dots y_m$. For example, if $X = CAB$ and $Y = BACB$, the answer is 2 (they both share length 2 subsequences AB and CB). Here, we define a sub-problem $C(i, j)$ to be the longest subsequence of $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_j$. The recurrence we have seen for C is

LCS CHART

$$C(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + C(i-1, j-1) & \text{if } i > 0 \text{ and } j > 0 \text{ and } x_i = y_j \\ \max[C(i-1,j), C(i,j-1)] & \text{if } i > 0 \text{ and } j > 0 \text{ and } x_i \neq y_j \end{cases}$$

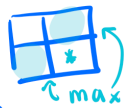
Use the recurrence to fill in a table for C for the strings $X = roror$ and $Y = orro$. You may use (or not) the sketch below.

		0	r	r	0	
	C	0	1	2	3	4
	0	0	0	0	0	0
R	1	0	0	1	1	1
0	2	0	1	1	1	2
R	3	0	1	2	2	2
0	4	0	1	2	2	3
R	5	0	1	2	3	3

i

j

Different:



R vs. 0

i = 1 j = 1

max[C(0,1), C(1,0)]

0,0

Same:

R vs. r

i = 2 j = 1

1 + C(1,0)

1 + 0 = 1

then ... at the bottom and track where it can from to find the longest seqvence. only keep track when its the sam

		0	r	r	0	
	C	0	1	2	3	4
	0	0	0	0	0	0
R	1	0	0	1	1	1
0	2	0	1	1	1	2
R	3	0	1	2	2	2
0	4	0	1	2	2	3
R	5	0	1	2	3	3

i

j

longest seqvence: 0, r, r

HW) Place mile post 100m apart if it is not exactly 100m apart → penalty of $(100 - x)^2$ so avg post spots to minimize penalties:

subproblem: S(i) = min possible total penalty when placing sign at m_i all the way up to m_i answer = S(n)

Recurrance:

$$S(i) = \begin{cases} 0 & i = 1 \\ \min_{1 \le j < i} \{S(j) + 100 - (m_i - m_j)^2\} & i > 1 \end{cases}$$

Now S(i) = total possible penalty btwn $m_1 \to m_n$

class1) having low/high stress job. High stress must have week prior = no work.

subproblem: J(i) = value of best job choices ending in week$_i$ at finishing either H/L stress

Recurrance:

$$J(i) = \begin{cases} 0 & i = 0 \\ \max(h_1, l_1) & i = 1 \\ \max[l_i + J(i-1), h_i + J(i-2)] & i > 1 \end{cases}$$

Now J(i) = best job choice

class 2) max continuous sum

subprocess: MCS(i) = value of max cont. subseq. sum ending at position i

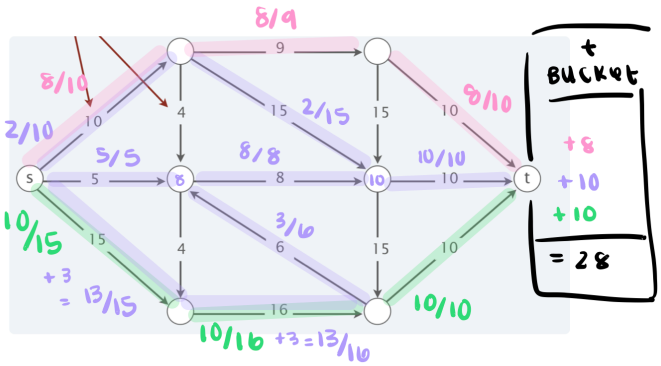
Recurrance:

$$MCS(i) = \begin{cases} 0 & i = 0 \\ a_1 & i = 1 \\ \max[a_i, a_i + MCS(i-1)] & i > 1 \end{cases}$$

start new seq ↑ ↑ attach to tail of prev. best

NETWORK FLOW

Time complexity: $O(V \cdot E \cdot C)$ or $O(V \cdot E^2)$



Ford-Fulkerson Method

input: graph G, source s, terminus t

1. **initialize** flow f of each edge to 0
2. **while** there exists an augmenting path p in the residual network G_f
3. augment flow along p
4. **return** f

- flow increases by at least 1 with each iteration
- total time is O(V•E•C) where C is network capacity
- could be slow if C is very large

- many improvements
- Edmonds-Karp is $O(V \cdot E^2)$
- idea is to use BFS to find s-t path in G_f

max-flow min-cut theorem

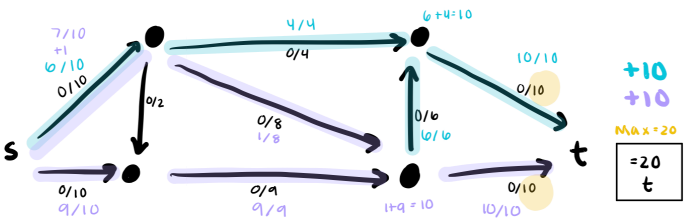
The size of the maximum flow from s to t is the capacity of the minimum (s,t)-cut

note

- An (s,t)-cut is a partition of the nodes S and V-S with s in S and t in V-S
- the capacity of that cut is the sum of the flow leaving S
- algorithm correctness follows (with a lot of work)

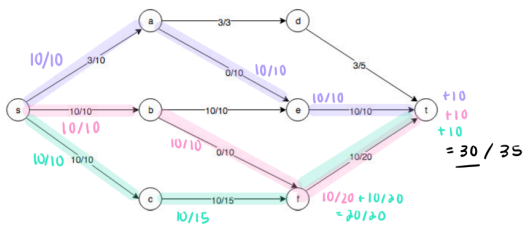
example

/
flow / capacity



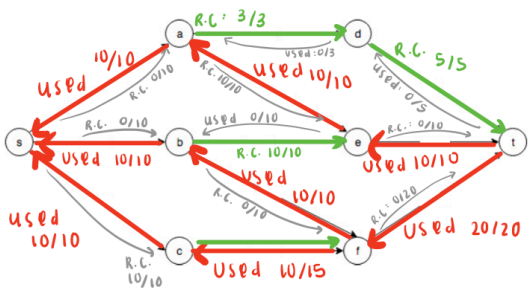
5. The following flow graph G has some flow assignments already made. Use the Ford-Fulkerson method to continue the process of deriving the maximum flow from s to t. Show the residual graph G_f, find an augmenting path in G_f, and then update the flow in G. One step should be enough to find the max-flow.

G_f:

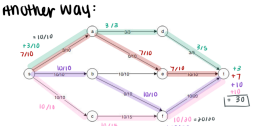


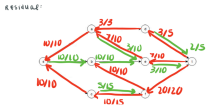
Residual graph:

used = residual capacity



Another way:





NP

NP = non-deterministic polynomial

Time complexity = polynomial

n = input size
k = constant (like 2)

$O(n^2)$ / $O(n^k)$

- it allows a choice of next-step
- uses yes/no problems
  ↓
  accepts

NP-complete: if one NP-complete problem can be solved in poly-time, then all of NP can be solved in poly-time

↑ no one knows if this is possible but if you prove one, you prove it all.

1) If a single NP-complete problem is shown to have a polynomial time (deterministic) algo, then it can be concluded that P = NP

TRUE. If you prove one, you prove all.

2) Whether any NP-complete problem is P OPEN.

3) If a single NP-complete problem is shown to require exponential time, then it can be concluded that P = NP.

FALSE

4) Every problem that can be solved in polynomial time non-deterministically can be solved by a polynomial time deterministic algo. OPEN million $?

5) There is an NP-complete problem with a poly-time algo. OPEN

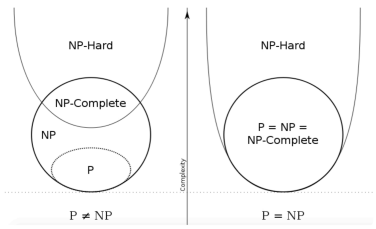
6) If one NP-complete problem is proved to require exponential time, then P = NP. FALSE

7) If one NP-complete problem is proved to require exponential time, then P ≠ NP. TRUE

if F is satisfiable, there exists a way for this algorithm to reach an accepting state

```
nondeterministic algorithm
 for i = 1 to n
    set x[i] = 0 or 1 (nondet step)

if F[x[1],..,x[n] is true
    then ACCEPT
    else REJECT
```



GREEDY

Every Greedy algorithm NEEDS:
- subproblems
- optimal substructure (like dynamic programming)
- greedy choice algorithm ⭐

RECURSIVE ALGO

RAS on inputs s (start times), f (finish times), K (subproblem), and n

```
RAS(s,f,k,n)
  m=k+1
  while m≤n and s[m]<f[k]
     m=m+1
  if m≤n
    return {a_m}URAS(s,f,m,n)
  else return ∅
```

find first a_m in S_k to finish

finish times already sorted, time now looks like O(n)

ITERATIVE ALGO

```
A={a_1}
k=1
for m=2 to n
   if s[m]≥f[k]
      A=AU{a_m}
      k=m
return A
```

time is pretty clearly O(n), but don't forget the O(n/gn) time used to sort f

GREEDY STRATEGY

1) determine optimal substructure of the problem
2) develop a recursive solution
3) show that if greedy choice made, only one subproblem remains
4) prove it is always safe to make greedy choice
5) develop recursive algorithm implementing greedy
6) convert recursive version to iterative

1. Derive a greedy strategy solving the following problem: We have a company with n workers. Worker w_i works a shift (s_i, f_i), where s_i is that worker's start time and f_i the finish time. (We will also refer to a shift as an *interval*, and you can assume $0 \le s_i < f_i$ are integers.)

   We want to form a small committee $C \subseteq \{w_1, \ldots, w_n\}$ which is *complete*, in other words for every worker not on the committee, that worker's shift overlaps (at least partially) with the shift of some other worker who is on the committee. Or formally, for every worker w_i there exists a worker $w_c \in C$ such that the shift of w_i overlaps with the shift of w_c (the intervals (s_i, f_i) and (s_c, f_c) must non-trivially intersect).

   The problem here is to find the *smallest* possible complete set C of workers. A greedy strategy will work. Here are two possible (different) strategies. One works and one does not.

   - choose the worker whose interval intersects the the largest number other intervals, select that worker to add to C, and eliminate all intersecting intervals
   - look at the earliest finishing uncovered interval w, and from among all intervals that intersect w add to the committee C the worker whose interval finishes the latest

   (a) Choose one of the greedy choices above and prove that it yields the smallest possible C. ⟨*write your choice and justification on the next page*⟩

#2 because interval I with the earliest finish time must be covered anyway. And if you select the shift that finishes last, then you will have a constant slight overlap so no gaps, and you can continue this for all the shifts next (hence the greedy approach. You are just looking at the current overlapping shifts, not the entire day schedules.

(b) In a few sentences, describe an algorithm that implements your greedy choice and give its run-time bound

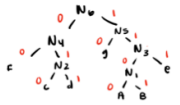
min-cover[n]
sort intervals based on finishing times. then if you have an overlap, continue checking, if you don't, add the last (j-1) shift to the solution set, after you complete this, you want to then start looking at the overlapping shifts of this new shift (j-1) and continue the same looping process. and then the solution set contains the min. interval cover.
Time complexity = O(nlogn) because we loop through o(n) and then sort n intervals O(nlogn) so total = O(nlogn).

Huffman Code

Time complexity: O(nlogn) b/c store heap = O(logn) and O(n) iterations.

1) order in min-heap
2) combine two smallest frequencies
3) repeat step 2

then number each branch 0 or 1.
0 = left branch
1 = right branch



A	1100
B	1101
C	010
D	011
E	111
F	00
G	10

CIS 315, Intermediate Algorithms
Spring 2019

Sample Solution to Something Like Problem 1 of the Sample Final Exam from the Previous Term

Derive a greedy strategy solving the following problem:

We have a company with n workers. Worker w_i works a shift (s_i, f_i), where s_i is that worker's start time and f_i the finish time.

We want to form a small committee $C \subseteq \{w_1, \ldots, w_n\}$ with the following property: for every worker w_i there exists a worker $w_c \in C$ such that the shift of w_i overlaps with the shift of w_c. That is, the intervals (s_i, f_i) and (s_c, f_c) must intersect.

So the problem here is to find the smallest possible set C of workers whose shifts overlap with all workers.

1. Describe your greedy choice solving it. ("Choose the first worker with property $\mathcal{P}$.")
2. Show that if there is an optimal solution for which the greedy choice was not made, then an exchange can be made to conform with your greedy choice. ("Let schedule $\mathcal{S}$ use worker w_j who does not satisfy property $\mathcal{P}$, and let w_k be the worker that does. Here I show that the schedule $\mathcal{S}'$, which is obtained by exchanging worker w_j for w_k, is just as good as $\mathcal{S}$ ...")
3. Describe, in English, how to implement a greedy algorithm.
4. How long would your algorithm take?

solution:

1. Note: during the description of the greedy strategy we distinguish between the set of workers who need their shift covered V and those workers W who are eligible to serve on the committee. Initially both V and W are the set of all workers. The distinction is that V will shrink as the committee C increases, whereas W will always contain all workers (two committee members may need to overlap).

   Consider the first worker $w_i \in V$ to finish - this is the critical shift to cover. Consider the set of workers in W that cover that w_i's shift: $S = \{w_k \in W | s_k \le f_i\}$. The **greedy choice** is that we add to C the $w_j \in S$ with the latest finish time. Then remove from V all workers whose shift intersects with w_j.
2. Suppose an optimal solution C' is given which differs from the greedy solution above. Look at the first stage of our greedy approach, consider the earliest finisher w_i at that stage as

above, and take the $w \in C'$ which covers w_i's shift. We can replace w with our w_j, getting committee $C' - \{w\} \cup \{w_j\}$, and get a solution that is just as good since w_j has a later finish time than w. Also note that no workers' shifts get uncovered, since w_i is the first to finish.

3. Perhaps it would be convenient to sort the workers by finish time, and perhaps other optimizations may help the constants. But basically:
   - scan the V to find w_i
   - scan the W to find w_j
   - add w_j to C, and scan the V removing all workers whose shift intersects with w_j
   - repeat until V is empty
4. Each scan is $O(n)$ and removes at least one worker. Total is $O(n^2)$.

MIDTERM TOPICS

	TIME COMPLEXITY:
DIJKSTRAS	O(elogv)
PRIMS	O(elogv)
KRUSKALS	O(elogv)
FLOYD-W	O(n³)
BFS	O(n+m)
DFS	O(v+e)
TOPOLOGICAL	O(v+e)