CSE 6140 / CX 4140

Computational Science & Engineering Algorithms

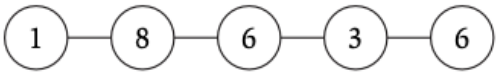
Homework 3

Please type in all answers.

1. (20 points) Let $G = (V, E)$ be an undirected graph with n nodes. Recall that a subset of the nodes is called an *independent set* if no two of them are joined by an edge. Finding large independent sets is difficult in general; but here we'll see that it can be done efficiently if the graph is simple enough.

Call a graph $G = (V, E)$ a *path* if its nodes can be written as $v_1, v_2, ..., v_n$, with an edge between v_i and v_j if and only if the numbers i and j differ by exactly 1. With each node v_i, we associate a positive integer weight w_i.

Consider, for example, the five-node path drawn below. The weights are the numbers drawn inside the nodes. The maximum weight of an independent set here is 14.



The goal in this question is to solve the following problem:

Find an independent set in a path G whose total weight is as large as possible.

a. Give an example to show that the following algorithm *does not* always find an independent set of maximum total weight.

```
The "heaviest-first" greedy algorithm
  Start with S equal to the empty set
  While some node remains in G
    Pick a node v_i of maximum weight
    Add v_i to S
    Delete v_i and its neighbors from G
  Endwhile
  Return S
```

Solution: Consider the graph in Figure 1: Note that the `heaviest-first` algorithm first picks node 2 with weight 10, removes nodes 1 and 3. Next, it picks node 5 which has weight 5, thus, giving the independent of weight 15, which is not the maximum weight independent set. The nodes 1, 3 and 5 form the maximum weight independent set of weight 21.

b. Given an example to show that the following algorithm also *does not* always find

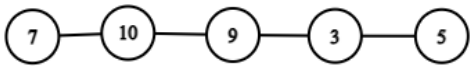


Figure 1: Counterexample for the `heaviest-first` greedy algorithm

an independent set of maximum total weight.

```
Let S_1 be the set of all v_i where i is an odd number
Let S_2 be the set of all v_i where i is an even number
(Note that S_1 and S_2 are both independent sets)
Determine which of S_1 or S_2 has greater total weight,
  and return this one
```

Solution: Consider the graph Figure 2:

Figure 2: Counterexample for the `odd-even` algorithm for maximum-independent set

Note that according to the `odd-even` algorithm, $S_1 = \{v_1, v_3, v_5\}$, which has a weight of 19, and $S_2 = \{v_2, v_4\}$, which has a weight of 15. So, the algorithm returns S_1 as the maximum weight independent set with weight of 19. However, $S = \{v_2, v_5\}$ forms the maximum weight independent set with a weight of 22.

c. Given an algorithm that takes an n-node path G with weights and returns an independent set of maximum total weight. The running time should be polynomial in n, independent of the values of the weights.

Solution: First, we discuss the idea behind the pseudocode:

- **Subproblems:** Let $OPT(j)$ be the maximum weight of the independent set for path graph with j nodes $(v_1, v_2, \ldots, v_j)$. The final node v_n could either be present in the $OPT(n)$ or absent.
  - If it is absent: $OPT(n) = OPT(n-1)$
  - If it is present: $OPT(n) = w_n + OPT(n-2)$
  - So, $OPT(n) = max\{OPT(n-1), w_n + OPT(n-2)\}$ (since v_n is present, v_{n-1} is omitted)
- Note, that this argument holds for any $j \geq 2$.
- **Relating subproblems:** So, $OPT(j) = max\{OPT(j-1), w_j + OPT(j-2)\}$ (since v_j is present, v_{j-1} is omitted)

- **Base cases:** We can define the base case as: $OPT(0) = 0$ and $OPT(1) = w_1$
- We note that we do not need to maintain another index to track the start of the max weight independent set since, even if it starts at some v_i, including v_k for $1 \leq k \leq i-2$ will only add to the weight of the independent set (provided the weights are positive). So, the independent set is going to start at v_1 or v_2 and hence we do not need a separate index to track start of the independent set
- **Topological order:** We solve $OPT(i)$ in increasing order of i ($i = 0, 1, \ldots, n$)
- **Original problem:** The solution to the original problem is given by $OPT[n]$ in the OPT array of size $(n+1)$
- **Time complexity:** We require to calculate $OPT(n)$ and we do so by solving n subproblems each taking a constant time to solve (taking maximum, two lookups and one addition). Thus, we solve this problem in linear time in number of nodes (i.e.,) **O(n)** for path graph $v_1, v_2, \ldots, v_n$

The Algorithm 1 returns the weight k of the maximum independent set of the path graph on n nodes. Calling Algorithm 2 with $i = n$ and $W = k$ gives the nodes in the maximum weight independent set:

Algorithm 1: MWIS(A)

Input: A: Path graph as an array of the node weights ($A[i] = w_i$, $i = 1, 2, \ldots, n$)

Output: opt: The weight of the maximum weight independent set of path graph

```
// Define the OPT array of size n+1 along with base case
OPT = array(n+1)
OPT[0] = 0
OPT[1] = A[1]

// Recursive step
for i=2,...,n
  if OPT[i-1] ≥ (A[i] + OPT[i-2])
    OPT[i] = OPT[i-1]
  else
    OPT[i] = (A[i] + OPT[i-2])

// Required optimal
opt = OPT[n]

return opt
```

Algorithm 2: FIND_SOL_MWIS(i, OPT,A)

Input: i: Index to track path
OPT: Array used to compute the optimal
A: The node weights of path graph on n vertices
Output: The set of nodes in the maximum weight independent set of path graph

```
// Base case
1 if i == 0
2   return []

// Recursive step
3 else
4   if A[i] + OPT[i-2] > OPT[i-1]
5     S = [i] +  FIND_SOL_MWIS(i-2,OPT,A)
6     return S
7   else
8     return FIND_SOL_MWIS(i-1,OPT,A)
```

2. (20 points) You are managing a consulting team. You need to plan their schedule for the year. For each week, you can either assign them a low-stress job or a high-stress job.

You are given arrays l and h. $l[i]$ is the revenue you make by assigning a low stress job in week i. $h[i]$ is the revenue you make by assigning a high stress job in week i.

In order for the team to take on a high-stress job in week i, it's required that they do no job (of either type) in week $i-1$; they need a full week of prep time for the high-stress job. On the other hand, it is okay for them to take a low-stress job in week i even if they have done a job (or either type) in week $i-1$.

Write pseudocode of an algorithm that finds the maximum revenue you can make for the year.

Hint: Use dynamic programming. For week i, assume that you know the maximum revenue achieved up to weeks $i-1$ and $i-2$. Let's say these are $OPT(i-1)$ and $OPT(i-2)$ respectively. In week i you can either choose a low stress job (revenue $l[i]$) or high-stress job (revenue $h[i]$). You want to find a recurrence that relates $OPT(i)$ to $OPT(i-1)$ and $OPT(i-2)$.

Solution: First, we discuss the approach:

- First, we consider the problem for general n weeks and then finally we can use $n = 52$ (since a year has 52 weeks).
- **Subproblems:** Let $OPT(i)$ be the maximum revenue for i weeks.
- The i^{th} week could be assigned a low-stress job or a high-stress job:

- If low-stress job: $OPT(i) = l[i] + OPT(i-1)$ (since low-stress job, no break in week $i-1$)
- If high-stress job: $OPT(i) = h[i] + OPT(i-2)$ (since high-stress job, break in week $i-1$)

- **Relate subproblems:** So, we relate the subproblems by the following recurrence relation:

$$OPT(i) = max\{OPT(i-1) + l[i], OPT(i-2) + h[i]\}$$

- **Base case:** The base case being: $OPT(0) = 0$ and $OPT(1) = max\{l[1], h[1]\}$ (since we have $OPT(i-2)$ in the recurrence, we need two base cases)
- **Topological order:** We fill the array OPT of length $n+1$ in increasing order of i
- **Original problem**: The optimal solution is $OPT[n]$
- **Time complexity:** We have $(n+1)$ subproblems, each solved in constant time (four array lookups, two additions, and one max operation). Thus, running time is **O(n)**

The pseudocode is given in Algorithm 3.

Algorithm 3: JOB_ASSIGN(l,h,n)

Input: l: Array containing revenue for low-stress jobs
h: Array containing revenue for high-stress jobs
n: Number of weeks
Output: Maximum revenue in n weeks

```
// Define the OPT array of size n + 1
OPT = array(n+1)

// Base case
OPT[0] = 0
OPT[1] = l[1] if l[1] > h[i] else h[1]

// Recursive step
for i=2,...,n
  if (OPT[i-1] + l[i]) ≥ (OPT[i-2] + h[i])
    OPT[i] = (OPT[i-1] + l[i])
  else
    OPT[i] = (OPT[i-2] + h[i])

// Required optimal
return OPT[n]
```

3. (20 points) You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Write pseudocode of a dynamic programming algorithm that returns the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

Solution: Suppose we have to form the value V with the given denominations in *coins* array. First, we discuss the idea of the approach:

- **Subproblems:** We define $OPT(i, v)$ as the minimum number of coins to add up to v with i denominations $coins[1....i]$
- The last denomination $i = n$ either is present in the optimal solution or not.
  - o If it is absent: $OPT(n, V) = OPT(n-1, V)$ (since we are not using the last denomination)
  - o If it is present: $OPT(n, V) = 1 + OPT(n, V - coins[n])$ (since we use 1 of last denomination and remaining could use it as well)
- **Relate subproblems:** Note that this argument holds for any of the n denominations. So, we have the recursive relation:

$$OPT(i, v) = min\{OPT(i-1, v), 1 + OPT(i, v - coins[i])\}$$

  - o However, note that in order to include the i^{th} denomination, the remaining value V should be $\geq coins[i]$. So, we modify the above relation as follows:

$$OPT(i, v) = \begin{cases} OPT(i-1, v) & v < coins[i] \\ min\{OPT(i-1, v), 1 + OPT(i, v - coins[i])\} & v \geq coins[i] \end{cases}$$

- **Base cases:** Now, to handle the base cases:
  - o If we have to pick minimum number of coins to add up to 0 it is 0: $OPT(i, 0) = 0$ $i = 0, 1, 2, \ldots, n$
  - o $i = 0$ is the case when there are no coins at all. So we fill as $OPT(0, k) = \infty$ $k = 1, 2, \ldots V$ (to indicate that we can't form the combination)
- **Original problem:** The required optimal is given by $OPT(n, V)$ (all $i = n$ denominations to form value $v = V$)
- **Topological order:** We fill the table in row-major order (row indicated by i and column indicated by v)
- **Running time:** Note that we have $(n+1)(V+1)$ subproblems each done in a constant time. So, running time is **O(nV)**

Refer to the pseudocode given by Algorithm 4.

Algorithm 4: MIN_COINS(coins,V)

Input: coins: The array of denominations
V: The value to form using the denominations
Output: Minimum number of coins to form V using denominations from coins

```
// Form the table to be filled
OPT = array(n+1, V+1)
// Initialize them with some large number (10^9)
for i=0 to n
  for v=0 to V
    OPT[i][v] = 1e9

// Base case
for i=0 to V
  OPT[i][0] = 0

// Table filling
for i=1 to n
  for v=1 to V
    if coins[i] > v // Exclude i^th denomination if coins[i] > v
      OPT[i][v] = OPT[i-1][v]
    else
      if OPT[i-1][v] ≤ 1 + OPT[i][v - coins[i]]
        OPT[i][v] = OPT[i-1][v]
      else
        OPT[i][v] = 1 + OPT[i][v - coins[i]]

// Get the final solution
if OPT[n][V] == 1e9 // V can't be formed with these denominations
  return -1
else
  return OPT[n][V]
```

4. (20 points) A number of languages (including Chinese and Japanese) are written without spaces between words. You are given text in such a language, and you are required to design an algorithm to infer likely boundaries between consecutive words in the text. If English were written without spaces, the analogous problem would consist of taking a string like "meetateight" and deciding that the best segmentation is "meet at eight" (and not "me et at eight", or "meet ate aight", or any of a huge number of possibilities).

To solve the problem, you are given a function Quality that takes any string of letters and returns a number that indicates the quality of the word formed by the string. A high number indicates that the string resembles a word in the language (e.g. 'meet'), whereas a low number means that the string does not resemble a word (e.g. 'eeta')

The total quality of a segmentation is determined by adding up the qualities of each of its words.

Write pseudocode of a dynamic programming algorithm that take a string y and computes a segmentation of maximum total quality. What is the running time of your algorithm? (You can treat the call to Quality as a single computational step, O(1)).

Solution: Suppose we have to find the optimal segmentation for the string $y[1 \dots n]$. First, we discuss the idea behind our approach:

- We know that the last letter of the string belongs to the last segment in the optimal solution. Suppose that it starts at some index $i \leq n$, then we could remove these letters and find optimal segmentation for the smaller string $y[1 \dots (i-1)]$
- **Subproblems:** Let $OPT(i)$ be the quality of the maximum quality segmentation of the string $y[1 \dots i]$
- **Relating subproblems:**
    - Based on the above argument, for the optimal segmentation, if we know the start point $j \leq i$ of the last segment, then we could relate the larger problem to the smaller one as follows:

$$OPT(i) = OPT(j-1) + Quality(y[j...i])$$

    - However, we don't know the start point j. So, we just check all the possible $j \leq i$ and choose the one that maximizes the quality of the last segment.
    - So, we relate the subproblems as follows:

$$OPT(i) = max_{1 \leq j \leq i}\{OPT(j-1) + Quality(y[j \dots i])\}$$

- **Base cases:** Here, our base cases will be $OPT(0) = 0$ and $OPT(1) = Quality(y[1])$. We require $OPT(0)$ since the recurrence relation involves $OPT(j-1)$ for $j \geq 1$.
- **Topological order:** Now, we just fill the OPT array of size $(n+1)$ in increasing order of i.
- **Original problem:** The solution to the original problem is given by $OPT[n]$ in the array
- **Time complexity:** We have $(n+1)$ subproblems and for each $i \geq 2$, we take maximum of i values each computed in constant time (one lookup and one call to $Quality()$ function, which runs in $O(1)$ time). So, each subproblem is solved in linear time. Hence, the total running time is $O(n^2)$ for a string y of length n

The Algorithm 5 returns the optimal segmentation of y and call to Algorithm 6 as FIND_SOL_OPT_SEG(i, OPT, y) returns the indices where each segment starts.

Algorithm 5: OPT_SEG(y)

Input: y: The string $y[1 \dots n]$ which has to be segmented to maximize the quality

Output: The quality of the maximum quality segmentation of the string

```
// Define the OPT array of size n+1 along with base case
OPT = array(n+1)
OPT[0] = 0
OPT[1] = Quality(y[1])

// Recursive step
for i=2,...,n
  sub = []
  for j=1,...,i
    sub.append(OPT[j-1] + Quality(y[j...i]))
  // Find the max of the values in sub
  OPT[i] = max(sub)

// Required optimal
opt = OPT[n]

return opt
```

Algorithm 6: FIND_SOL_OPT_SEG(i, OPT, y)

Input: i: Index to track segment
OPT: The dynamic programming array
y: The string to segment
Output: The set of indices that indicate the start of each segment

```
// Base case
if i == 0
  return []

// Recursive step
else
  j = argmax_{1≤j<i}(OPT[j-1] + Quality(y[j...i]))
  S = [j] + FIND_SOL(j-1,OPT,y)
  return S
```

5. (20 points) A thief robbing a store wants to take the most valuable load that can be carried in a knapsack capable of carrying at most W pounds of loot. The thief can choose to take any subset of n items in the store. The i-th item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. Which items should the thief take and what is their total value? Write pseudocode to solve the problem.

Solution: This problem is a typical case of the knapsack problem. We outline the algorithm as follows:

- **Subproblems:** Let $OPT(i, w)$ be the maximum value loot with i items and weight capacity w.
- Similar to previous cases, we have a binary choice of either to include the i^{th} item or to exclude it. So, when $w_i \leq w$ (i^{th} item can fit in the knapsack):

$$OPT(i, w) = max\{OPT(i-1, w), v_i + OPT(i-1, w-w_i)\}$$

- When $w_i > w$, we just exclude the i^{th} item: $OPT(i, w) = OPT(i-1, w)$
- **Relate subproblems:** So, the recurrence relation for the subproblems is:

$$OPT(i, w) = \begin{cases} OPT(i-1, w) & w < w_i \\ max\{OPT(i-1, w), v_i + OPT(i-1, w-w_i)\} & w \geq w_i \end{cases}$$

- **Base cases:** Here, we have the base cases:
  - $OPT(0, w) = 0$ $w = 0, 1, \ldots, W$ (since no items, maximum loot is 0)
  - $OPT(i, 0) = 0$ $i = 0, 1, \ldots, n$ (capacity is 0, so cannot take any items)
- **Topological order:** We again fill the table of size $(n+1) \times (W+1)$, and we fill it in row-major order.
- **Original problem:** We get the optimal solution for our original problem from $OPT(n, W)$.

- **Running time:** We have $(n+1)(W+1)$ subproblems each taking $O(1)$ time. Thus, running time is **O(nW)**

The Algorithm 7 gives maximum loot and Algorithm 8 gives which items to pick for the maximum loot:

Algorithm 7: MAX_LOOT(V,W)

Input: V: The array of value of each of the n items
W: The weight capacity of the knapsack
A: The array of weight of each of the n items
Output: Maximum loot

```
// Form the table to be filled
1  OPT = array(n+1, W+1)
   // Base cases
2  for i=0 to n
3    OPT[i][0] = 0
4  for w=0 to W
5    OPT[0][w] = 0
   // Table filling
6  for i=1 to n
7    for w=1 to W
8      if A[i] > w // Exclude item i
9        OPT[i][w] = OPT[i-1][w]
10     else
11       if OPT[i-1][w] ≥ V[i] + OPT[i][W - A[i]]
12         OPT[i][w] = OPT[i-1][w]
13       else
14         OPT[i][w] = V[i] + OPT[i][W - A[i]]
   // Get the final solution
15 return OPT[n][W]
```

Algorithm 8: FIND_SOL_KNAPSACK(i, W, OPT,A)

Input: i: Index to track path
W: The remaining weight capacity of the knapsack
OPT: Array used to compute the optimal
A: The array of weights of the n items

Output: The set of nodes in the maximum weight independent set of path graph

```
// Base case
1  if i == 0
2    return []

// Recursive step
3  else
4    if A[i] > W
5      return FIND_SOL_KNAPSACK(i-1, W, OPT,A)
6    else
7      if OPT[i-1][w] ≥ V[i] + OPT[i][W - A[i]]
8        S = [i] +  FIND_SOL_KNAPSACK(i-1, W - A[i], OPT,A)
9        return S
10     else
11       return FIND_SOL_KNAPSACK(i-1, W, OPT,A)
```