Week 10: Lab

Module 4: Techniques

Collaboration level 0 (no restrictions). Open notes.

1 This week's practice problems

1. **Longest increasing subsequence:** A *subsequence* of a sequence (for example, an array, linked list or string), is obtained by removing zero or more elements and keeping the rest in the same order. A subsequence is called a *substring* if its elements are contiguous in the original sequence. For example:

   (a) gorit, algo, rithms are all substrings of algorithms

   (b) grm, lori, lot, agis,gorims are all subsequences of Algorithms

   (c) algomi, og are *not* subsequences of Algorithms

   The problem: Given an array $A[1..n]$ of integers, compute the length of a *longest increasing subsequence* (A sequence $B[1..k]$ is *increasing* if $B[i] < B[i+1]$ for all $i = 1..k-1$). For example, given the array

   $$\{5, 3, 6, 2, 1, 5, 3, 1, 2, 5, 1, 7, 2, 8\}$$

   your algorihm should return 5 (for e.g. correponding to the subsequence $\{1, 3, 5, 7, 8\}$; there are other subsequences of length 5).

   Describe an algorithm which, given an array $A[]$ of n integers, computes the LIS of A.

```
EXAMPLES:

Input: arr[] = {3, 10, 2, 1, 20}
Output: Length of LIS = 3
The longest increasing subsequence is 3, 10, 20

Input: arr[] = {3, 2}
Output: Length of LIS = 1
The longest increasing subsequences are {3} and {2}

Input: arr[] = {50, 3, 10, 7, 40, 80}
Output: Length of LIS = 4
The longest increasing subsequence is {3, 7, 40, 80}

Input: A[]={0,8,4,12,2,10,6,14,1,9,5, 13,3,11,7,15}
Output: Length of LIS = 6
Explanation:Longest increasing subsequence 0 2 6 9 13 15, which has length 6
```

```
Input: A[] = {5,8,3,7,9,1}
Output: Length of LIS = 3
Explanation:Longest increasing subsequence 5 7 9, with length 3
```

If you need a hint, turn to Appendix A.

2. **Unbounded knapsack:** We have n items, each with a value and a positive weight; Item i has value v_i and weight w_i. We have a knapsack that holds maximum weight W. The problem is which items to put in the backpak to maximize its value.

   In this problem you'll consider a variation of the $0-1$ Knapsack problem, where we have *infinite copies* of each item. Describe an algorithm that, given $W, \{w_1, ..., w_n\}$ and $\{v_1, ..., v_n\}$, finds the maximal value that can be loaded into the knapsack.

   The sequence of steps below will guide you towards the solution

   (a) A friend proposes the following greedy algorithm: start with the item with the largest cost-per-pound, and pick as many copies as fit in the backpack. Repeat.

   Show your friend this is not correct by coming up with a counter-example.

   (b) Now let's define a useful sub-problem. With the classical knapsack, once you take an item, you cannot take it again, so you have fewer items to choose from. So the subproblem we use has to keep track of both the knapsack size as well as which items are allowed in the knapsack. We used $knapsack(w, i)$ to be the maximum value we can get for a knapsack of weight w considering items 1 through i.

   For the unbounded knapsack, we have infinite supplies of each item. So we don't need to keep track of which items we already included, only of the size of the knapsack. Let $K(w)$ represent the optimal value for a knapsack of capacity w.

   Argue optimal sub-structure:

   - Suppose $K(w)$ contains item i.
   - Then the remainig items in $K(w)$ must be an optimal solution for .....

   (c) Come up with a recursive definition of $K(w)$. (Hint: Consider all the choices, and pick the best.)

(d) Develop a DP solution based on this recurrence — either top-down or bottom up. What is the running time of your algorithm ?

2 Additional problems

1. **Matching points on a line:**[1] You are given two arrays of n points in one dimension: red points $r_1, r_2, ..., r_n$ and blue points $b_1, b_2, ..., b_n$. You may assume that all red points are distinct and all blue points are distinct. We want to pair up red and blue points, so that each red point is associated uniquely with a blue point and vice versa. Given a pairing, we assign it a score which is the sum of distances between each pair of matched points.

   Find the pairing (matching) of minimum score. Hint: Aim for an $O(n \lg n)$ algorithm. Draw examples for small valus of n to get intuition.

   Example: Consider the input where the red points are $r_1 = 8, r_2 = 1$ and the blue points are $b_1 = 3$ and $b_2 = 9$. There are two possible matchings:

   - match r_1 to b_1 and r_2 to b_2: the score of this matching is $|r_1 - b_1| + |r_2 - b_2| = |8 - 3| + |1 - 9| = 13$
   - match r_1 to b_2 and r_2 to b_1: the score of this matching is $|r_1 - b_2| + |r_2 - b_1| = |8 - 9| + |1 - 3| = 3$

   The algorithm shoud return the cost of the optimal matching, which is 3.

2. **The skis and skiers problem:**[2] You've decided to become a ski bum, and hooked up with Sugarloaf Ski Resort. They'll let you ski for free all winter, in exchange for helping their ski rental shop with an algorithm to assign skis to skiers.

   Ideally, each skier should obtain a pair of skis whose heigth matches his or her own height exactly. Unfortunately, this is generally not possible. We define the disparity between a skier and his/her skis as the absolute value of the difference between the height of the skier and the height of the skis. The objective is to find an assignment of skis to skiers that minimizes the sum of the disparities.

   The questions below will guide you towards a solution.

   (a) First, let's assume that there are n skiers and n skis. Consider the following algorithm: consider all possible assignments of skis to skiers; for each one, compute the sum of the disparities; select the one that minimizes the total disparity. How much time would this algorithm take on a 1GHz computer, if there were 50 skiers and 50 skis?

   (b) One day, while waiting for the lift, you make an interesting discovery: if we have a short person and a tall person, it would never be better to give to the shorter person a taller pair of skis than were given to the taller person. Prove that this is always true. (For a hint, see appendix A).

   (c) Describe a greedy algorithm to assign the skis to skiers, and argue why this algorithm is correct. What is the time complexity of your algorithm?

   (d) Your next task is to design an efficient dynamic programming algorithm for the more general case where there are m skiers and n pairs of skis ($n \geq m$).

   Here is some notation that may help you. Sort the skiers and skis by increasing height. Let $h[i]$ denote the height of the ith skier in sorted order, and $s[j]$ denote the height of the jth pair of skis in sorted order. Let $\text{OPT}(i, j)$ be the optimal cost (disparity) for matching the

[1] Credit: reproduced from Stanford University, cs161, fall 2016-2017

[2] Credit: adapted from B. Thom, Harvey Mudd College

first i skiers with skis from the set $\{1, 2, ..., j\}$. The solution we seek is simply OPT(m, n). Define OPT(i, j) recursively.

(e) Now describe a dynamic programming algorithm for the problem. What is the running time of your algorithm? Explain.

(f) Briefly describe how your algorithm can be modified to allow you to find an actual optimal assignment (rather than just the cost) of skis to skiers. How does this affect the running time of your algorithm?

3 Appendix A: Hints

- **Longest increasing subsequence:**

  Use the following subproblem: Let $L(i)$ be the length of a LIS starting with $A[i]$.

  Express $L(i)$ rercursively.

  Assume you computed $L(i)$ for all $i = 1, 2, ..., n$. How do you find the LIS of A?

- **Matching points on a line:**

  Think greedily, and argue with an exchange argument.

- **Skis and skiers**

  Let P1, P2 be the length of two skiers, and S1, S2 the lengths of the skis. We assume $P1 < P2$ and $S1 < S2$. Try proving that pairing P1 with S1 and P2 with S2 is better than pairing P1 with S2 and P2 with S1.

  Basically we'd like to show that $|P1 - S1| + |P2 - S2| < |P1 - S2| + |P2 - S1|$. To do so, consider all possible cases:

  - case 1: $P1 < S1 < P2 < S2$: we need to show that $(S1 - P1) + (S2 - P2) < (S2 - P1) + (P2 - S1)$. Draw a picture and go from there.
  - case 2: ...

4 Appendix B: Solutions

1. **Finding the LIS (Longest Increasing Subsequence):**

   Notation and choice of subproblem: Let $L(i)$ be the length of a LIS starting with $A[i]$.

   To find the LIS(A), we need to return $\max\{L(i), 1 \leq i \leq n\}$ Why? Because LIS(A) must start at some index i in A.

   Optimal substructure of $L(i)$: Consider a LIS starting with $A[i]$, and denote it $LIS(i)$. Let $A[j]$, with $j > i$, be the element after $A[i]$ in $LIS(i)$. Then $LIS(i)$ must consist of $A[j] + LIS(j)$. [insert here the usual justification by contradiction]. This leads us to a recursive definition of $L(i)$.

   Recursive definition of $L(i)$:

   Base case: if $i == n$ then there is no element after $A[n]$ so

   $$L(n) = 1$$

   Otherwise if $i < n$:

   $L(i) = 1 + \max\{L(j)$, where $i + 1 < j \leq n$ and $A[i] < A[j]\}$, or $L(i) = 1$ if no such j exists

   Explanation: We know that $L(i)$ includes $A[i]$. The second element in the $L(i)$ must be an index j such that $j > i$ and $A[j] > A[i]$. By optimal substructure, it must be that the subsequence starting at j is $L(j)$ (if this was not the case, ...[insert here the usual justification by contradiction]). Since we do not know what j is, we try all indices j and pick the largest.

   Running time of $L(i)$: Without DP, exponential.

   Computing $L(i)$ with dynamic programming:

   The table: We use a table $table[1..n]$ such that table[i] will store the solution returned by $L(i)$.

   Initialize: set $table[n] = 1$, and $table[i] = 0$ for $i = 1, ..., n - 1$.

```
L(i)

//if this was computed before, retrieve it from the table
if table[i] !=0: return table[i]

//otherwise, compute it and store it in the table
max = 0
for j=i+1 to n do:
   if A[j] > A[i]:
           thisj =  1 + L(j)
           if thisj > max:  max = thisj

//max is the solution to L(i)
table[i] = max
return max
```

Analysis: computing L(1) takes $O(n)$ ignoring teh cost of recursion, and assuming all values of $L(j)$ that it needs are already computed. In the worst case it needs $L(j)$ for all $j = 2, ..., n$. Overall there are $O(n)$ subproblems each of which takes $O(n)$ time, so $L(1)$ runs in $O(n^2)$.

To find the length of a LIS in A we initialize the table as above, and then call $L(i)$ for $i = 1, 2, ..., n$ (note that it is not sufficient to call just L(1) because it will only recurse on L(j) whith $A[j] < A[i]$, so it won't fill the whole table). Then we traverse the table and find that largest value of $L(i)$. That is the $L(A)$. The logic is that LIS(A) must start at some element i, and $LIS(A) = L(i)$.

Analysis: The table has size n, and each $table[i]$ takes $O(n)$ to compute ignoring the cost of the recursion. Because of the table, each $L(i)$ is computed precisely once, and the total cost of the recursion is $\Theta(n)$. So overall $\Theta(n) + \Theta(n) \cdot O(n) = \Theta(n^2)$.

2. **Unbounded knapsack**

The sub-problem: Let $K(x)$ represent the optimal value for a knapsack of capacity x. To pick the next item to add to the knapsack, we have the choice of any of the n items; specifically we'll choose the item that maximizes the value in the knapsack.

$K(x) = \max\{v_i + K(x - w_i),$ for all $1 \leq i \leq n$ and $w_i < x\}$ and basecase $K(0) = 0$

Developing a DP solution around this recurrence is straightforward. First the top-down recursive solution:

```
create a table[0..W]
table[0] = 0, and table[i] = -1 for all i = 1, 2, ..., W
return K_with_DP(W, table) // this is the solution we were looking for

K_with_DP(x, table)
    if x==0: return 0
    //else
    if table[x] != -1:  return table[x]
    //else
    table[x] = 0
    for i=1 to n
          if w[i] < x:
             table[x] = max { table[x], v[i] + K_with_DP(x-w[i], table) }

    return table[x]
```

or bottom-up DP:

```
create a table K[0..W]
K[0] = 0  //basecase
for x = 1 to W do:
    K[x] = 0
    for i = 1 to n do:
         if w[i] < x:   K[x] = max { K[x], v[i] + K[x - w[i]] }

return K[W]  // this is the solution we were looking for
```

The overall running time for both solutions is $O(n \cdot W)$.

3. **Matching points on a line:**

   Algorithm: Sort the red and blue points separately, and match them in order, that is, r_1 with b_1, r_2 with b_2, and so on (here I assumed that they are numbered in sorted order). This is a greedy algorithm.

   Analysis: $O(n \lg n)$ to sort plus $O(n)$ to match.

   Correctness: The hardest part is to actually prove that this is always correct.To do so, it is sufficient to show that there is always a solution that contains the first greedy choice, namely matching r_1 to b_1.

   Proof: Let O be an optimal matching O and assume that it does not match r_1 to b_1. Suppose that r_1 is matched with b_i and b_1 is matched with r_j. We modify O so that r_1 is matched with b_1 and r_j is matched with b_i. Denote by O' the resulting matching. We'll show that the cost of O' is $\leq O$, therefore O' must be an optimal matching. Therefore there exists an optimal matching that matches r_1 to b_1.

   The cost of O' is $r_1 - b_1| + |r_j - b_i| + ...$

   The cost of O is $r_1 - b_i| + |r_j - b_1| + ...$

   To show that the cost of O' is $\leq O$, we need to show that $|r_1 - b_1| + |r_j - b_i| \leq r_1 - b_i| + |r_j - b_1|$

   Assume wlog that $r_1 < b_1$. Then

   - case 1: $r_1 < r_j \leq b_1 < b_i$ ...
   - case 2: $r_1 < b_1 \leq r_j < b_i$ ...
   - case 3: $r_1 < b_1 < b_i \leq r_j$ ...

   In each case, draw the points in order to visualize, and ...

4. **Skis and skiers:**

   (a) **Solution:** Let's keep the set of skiers fixed. There are $n!$ possible orderings of skis to skiers. For each order, it takes us $O(n)$ to compute the sum of the disparities. Overall this is $O(n! \cdot n)$. For $n = 50$, this is $50 \cdot 50!$. A 1GHz computer executes 10^9 operations per second, so our algorithm would run in $50 \cdot 50! \cdot 10^{-9}$. To compute $n!$ we can use Stirling's formula $n! \sim (\frac{n}{e})^n \cdot \sqrt{2 \cdot pi \cdot n}$. We have $50 \cdot 50! \cdot 10^{-9} > 10^{50} \cdot 10^{-9} = 10^{41}$ seconds. This is more than 10^{30} years.

   (b) **Solution:** Let P1, P2 be the length of two skiers, and S1, S2 the lengths of the skis. We assume $P1 < P2$ and $S1 < S2$. We'll prove that pairing P1 with S1 and P2 with S2 is better than pairing P1 with S2 and P2 with S1.

   Basically we'd like to show that $|P1 - S1| + |P2 - S2| < |P1 - S2| + |P2 - S1|$. To do so we'll consider all possible cases:

- $P1 < S1 < P2 < S2$: we need to show that $(S1-P1)+(S2-P2) < (S2-P1)+(P2-S1)$ Note that $S2-P1 = S2-P2+P2-S1+S1-P1$, which means that $S1-P1+S2-P2 < S2-P1$.

  All the other cases can be shown in a similar way:
- $P1 < S1 < S2 < P2$: we can see that $(S1-P1)+(P2-S2) < (S2-P1)+(P2-S1)$
- $P1 < P2 < S1 < S2$: we can see that $(S1-P1)+(S2-P2) < (S1-P2)+(S2-P1)$
- $S1 < P1 < P2 < S2$: we can see that $(P1-S1)+(S2-P2) < (P2-S1)+(S2-P2)$
- $S1 < P1 < S2 < P2$: we can see that $(P1-S1)+(P2-S2) < (P2-S1)+(S2-P1)$
- $S1 < S2 < P1 < P2$: we can see that $(P1-S1)+(P2-S2) < (P1-S2)+(P2-S1)$

(c) **Solution:** Sort the skis and skiers, and pair them up, in order. This is correct because of (b) above. That is, assume that the optimal solution does not match the skis and skiers in order. This means there exists a pair of ski, skiers assignments (P1, S1) and (P2, S2) with $P1 < P2$ and $S1 > S2$. According to part (b), we can give S1 to P2 and S2 to P1 and this results in a better solution — contradiction.

Analysis: Use one of the $O(n \lg n)$ sorts (mergesort, quicksort, heapsort) and this runs in $O(n \lg n)$.

(d) **Solution:** It's interesting to see that if the number of skis and skiers is the same, the problem can be solved greedily in $O(n \lg n)$ time, while if $n > m$, then we need dynamic programming.

To recurse, we'll want to decrement i and j. Basically this tells us we want to try to find skis for skier i, and then recurse on the remaining skiers.

To assign skis to skier i, we have two options:

- pair skier i with skis j, and recurse on what's left
- do not pair skier i with skis j, and recurse on what's left

We'll try both and chose the best one.

$OPT(i,j) = min\{|h_i - s_j| + OPT(i-1, j-1), OPT(i, j-1)\}$

```
OPT(i, j)
    if (i==0) return 0
    if (j==0) return $\infty$
    if (j<i)  return $\infty$

    x = |h[i] - s[j]| + OPT(i-1, j-1)
    y = OPT(i, j-1)
    return min{x, y}
```

(e) **Solution:** We'll use a table $T[1..m][1..n]$ to store solutions to subproblems. Entry $T[i][j]$ will store the solution of $OPT(i,j)$.

We initialize the table with some initial value than can be distinguished from a result.

We implement the recursive function OPT(i,j) so that it first checks the table to see if the result is there. If yes, it returns it, otherwise it computes and stores it in the table.

```
OPT(i, j)
    if (T[i][j] != initial value) return T[i][j]

    if (i==0) return 0
    if (j==0) return $\infty$
    if (j<i)  return $\infty$

    x = |h[i] - s[j]| + OPT(i-1, j-1)
    y = OPT(i, j-1)

    T[i][j] = min{x, y}
    return min{x,y}
```

(f) **Solution:** We can figure out the assignment by examining the table. We start at $T[m][n]$. This value is either $T[m][n-1]$ or $|h_m - s_m| + T[m-1][n-1]$. If the value at $T[m][n]$ is equal to $T[m][n-1]$, then we know that h_m was NOT paired with s_n, so we jump to $T[m][n-1]$ and continue. If $T[m][n]$ is equal to $|h_m - s_m| + T[m-1][n-1]$ then we know that h_m was paired with s_n, so we jump to $T[m][n-1]$ and continue.