Homework 4 Solution

Due: October 31, in class.

Instruction. There are 7 problems in this homework, worth 100 points in total. For all algorithm design problems, you need to provide proofs on the correctness and running time of the algorithms unless otherwise instructed.

Problem 1 [15 pts]

You've been working with some physicists who need to study, as part of their experimental design, the interactions among large numbers of very small charged particles. Basically, their setup works as follows. They have an inert lattice structure, and they use this for placing charged particles at regular spacing along a straight line. Thus we can model their structure as consisting of the points $\{1, 2, 3, ..., n\}$ on the real line; and at each of these points j, they have a particle with charge q_j. (Each charge can be either positive or negative.)

They want to study the total force on each particle, by measuring it and then comparing it to a computational prediction. This computational part is where they need your help. The total net force on particle j, by Coulomb's Law, is equal to

$$F_j = \sum_{i<j} \frac{Cq_iq_j}{(j-i)^2} - \sum_{i>j} \frac{Cq_iq_j}{(j-i)^2}.$$

They've written the following simple program to compute F_j for all j:

---

for $j = 1, 2, \ldots, n$ **do**
  Initialize F_j to 0
  for $j = 1, 2, \ldots, n$ **do**
    if $i < j$ **then**
      Add $\frac{Cq_iq_j}{(j-i)^2}$ to F_j
    else if $i > j$ **then**
      Add $-\frac{Cq_iq_j}{(j-i)^2}$ to F_j
    end if
  end for
  Output F_j
end for

It's not hard to analyze the running time of this program: each invocation of the inner loop, over i, takes $O(n)$ time, and this inner loop is invoked $O(n)$ times total, so the overall running time is $O(n^2)$.

The trouble is, for the large values of n they're working with, the program takes several minutes to run. On the other hand, their experimental setup is optimized so that they can throw down n particles, perform the measurements, and be ready to handle n more particles within a few seconds. So they'd really like it if there were a way to compute all the forces F_j much more quickly, so as to keep up with the rate of the experiment.

Help them out by designing an algorithm that computes all the forces F_j in $O(n \log n)$ time.

Solution: This can be accomplished using a convolution. Define one vector to be $a = (q_1, q_2, \ldots, q_n)$. Define the other vector to be $b = (-n^{-2}, -(n-1)^{-2}, \ldots, -1/4, -1, 0, 1, 1/4, \ldots, n^{-2})$. Let $c = a * b$, which is the convolution of a and b. Now, for each j, we can find that the $(j+n)^{\text{th}}$ coordinate of c is equal to

$$\sum_{i<j} \frac{q_i}{(j-i)^2} + \sum_{i>j} \frac{-q_i}{(j-i)^2}.$$

From this term, we simply multiply by Cq_j to get the desired net force F_j.

The convolution can be computed in $O(n \log n)$ time, and reconstructing the terms F_j takes an additional $O(n)$ time. Hence, this algorithm computes all the forces F_j in $O(n \log n)$ time.

Problem 2 [10 pts]

Professor F. Lake tells his class that it is asymptotically faster to square an n-bit integer than to multiply two n-bit integers. Should they believe him?

Solution: No, we should not believe him.

Let $f_1(n)$ and $f_2(n)$ denote the time complexities of squaring an n-bit integer and multiplying two n-bit integers respectively. Suppose now we have two n-bit integers a and b. Note that $ab = \frac{1}{2} \cdot [(a+b)^2 - a^2 - b^2]$. Then we can evaluate ab by taking 4 multiplications and 3 additions/substractions which takes $O(f_1(n) + n)$ time. Hence, we have $f_2(n) = O(f_1(n) + n)$.

If this claim holds, we should have $f_1(n) = o(f_2(n)) = o(f_1(n) + n)$ which means $f_1(n) = o(n)$, which is a contradiction since we need to at least read all the n bits that representing an integer. Therefore, we should not believe him.

Problem 3 [15 pts]

This problem illustrates how to do the Fourier Transform (FT) in modular arithmetic, for example, modulo 7.

a) There is a number ω such that all the powers $\omega, \omega^2, \ldots, \omega^6$ are distinct (modulo 7). Find this ω, and show that $\omega + \omega^2 + \cdots + \omega^6 = 0$. (Interestingly, for any prime modulus there is such a number.)

b) Using the matrix form of the FT, produce the transform of the sequence $(0, 1, 1, 1, 5, 2)$ modulo 7; that is, multiply this vector by the matrix $M_6(\omega)$, for the value of ω you found earlier. In the matrix multiplication, all calculations should be performed modulo 7.

c) Write down the matrix necessary to perform the inverse FT. Show that multiplying by this matrix returns the original sequence. (Again all arithmetic should be performed modulo 7.)

d) Now show how to multiply the polynomials $x^2 + x + 1$ and $x^3 + 2x - 1$ using the FT modulo 7.

Solution: Note: all arithmetic is performed modulo 7.

a) Choose $\omega = 3$ ($\omega = 5$ is also correct). We have $3^1 = 3, 3^2 = 2, 3^3 = 6, 3^4 = 4, 3^5 = 5, 3^6 = 1$. And, $\omega + \omega^2 + \cdots + \omega^6 = (1 + 6) \cdot 7/2 = 0$.

b) By definition, we can get

$$M_6(\omega) = \begin{bmatrix} 1 & 1 & 1 & \dots \\ 1 & 3 & 3^2 & \dots \\ 1 & 3^2 & 3^4 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 2 & 6 & 4 & 5 \\ 1 & 2 & 4 & 1 & 2 & 4 \\ 1 & 6 & 1 & 6 & 1 & 4 \\ 1 & 4 & 2 & 1 & 4 & 2 \\ 1 & 5 & 4 & 6 & 2 & 3 \end{bmatrix}.$$

Hence, $(0, 1, 1, 1, 5, 2) \cdot M_6(\omega) = (3, 6, 4, 2, 3, 3)$.

c) Note that $\omega^{-1} = 3^{-1} = 5$. Hence we can get

$$M_6(\omega)^{-1} = \frac{1}{6} \begin{bmatrix} 1 & 1 & 1 & \dots \\ 1 & 5 & 5^2 & \dots \\ 1 & 5^2 & 5^4 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} = \begin{bmatrix} 6 & 6 & 6 & 6 & 6 & 6 \\ 6 & 2 & 3 & 1 & 5 & 4 \\ 6 & 3 & 5 & 6 & 3 & 5 \\ 6 & 1 & 6 & 1 & 6 & 1 \\ 6 & 5 & 3 & 6 & 5 & 3 \\ 6 & 4 & 5 & 1 & 3 & 2 \end{bmatrix}.$$

Hence, $(3, 6, 4, 2, 3, 3) \cdot M_6(\omega)^{-1} = (0, 1, 1, 1, 5, 2)$.

d) Let C denote the polynomial we want to compute. Define one vector to be $a = (1, 1, 1, 0, 0, 0)$ denoting the polynomial $A = x^2 + x + 1$. Define the other vector to be $b = (6, 2, 0, 1, 0, 0)$ denoting the polynomial $B = x^3 + 2x - 1$.

Compute $a \cdot M_6(\omega)$ to get $(A(1), A(\omega), \dots, A(\omega^5)) = (3, 6, 0, 1, 0, 3)$. Compute $b \cdot M_6(\omega)$ to get $(B(1), B(\omega), \dots, B(\omega^5)) = (2, 4, 4, 3, 1, 1)$.

Next compute $A(\omega^i)B(\omega^i)$ for $i = 0, 1, 2, \dots, 5$ to get $(C(1), C(\omega), \dots, C(\omega^5)) = (6, 3, 0, 3, 0, 3)$.

Finally compute $(C(1), C(\omega), \dots, C(\omega^5)) \cdot M_6(\omega)^{-1}$ to get $c = (6, 1, 1, 3, 1, 1)$ which is the final answer and denotes that $C = x^5 + x^4 + 3x^3 + x^2 + x + 6$.

Problem 4 [15 pts]

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 $v_1, v_2, \cdots, v_n$, 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 in Figure 1. The weights are the numbers drawn inside the nodes. 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 **do**
    Pick a node v_i of maximum weight
    Add v_i to S
    Delete v_i and its neighbors from G
end while
return S

(b) Give an example to show that the following algorithm also does not always find 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

---

(c) Give 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.

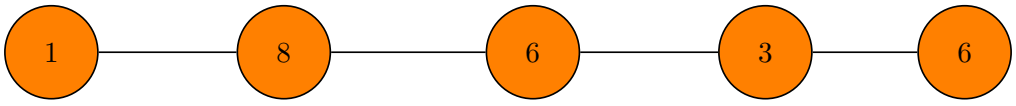

Figure 1: Problem 4

Solution:

(a) Consider the sequence of weights $2, 3, 2$. The greedy algorithm will pick the middle node, while the maximum independent set consists of the first and the third.

(b) Consider the sequence $3, 1, 2, 3$. The given algorithm will pick the first and the third nodes, while the maximum independent set consists of first and fourth.

(c) Let S_i denote an independent set on $\{v_1, \cdots, v_i\}$, and let X_i denote its weight. Define $X_0 = 0$ and note that $X_1 = w_1$. Now for $i > 1$, either v_i belongs to S_i or it doesn't. In the first case we know that v_{i-1} cannot belong to S_i, and so $X_i = w_i + X_{i-2}$. In the second case, $X_i = X_{i-1}$ and we have the recurrence

$$X_i = \max(X_{i-1}, w_i + X_{i-2})$$

Thus we can compute the values of X_i, in increasing order from $i = 1$ to n. X_n is the value we want, and we can compute S_n by tracing back through the computations of the max operator. Since we spend constant time per iteration, over n iterations the total time is $O(n)$.

Problem 5 [15 pts]

Let $G = (V, E)$ be a directed graph with nodes $v_1, \cdots, v_n$. We say that G is an ordered graph if it has the following properties.

(i) Each edge goes from a node with a lower index to a node with a higher index. That is, every directed edge has the form (v_i, v_j) with $i < j$.

(ii) Each node except v_n has at least one edge leaving it. That is, for every node v_i ($i = 1, 2, \cdots, n-1$), there is at least one edge of the form (v_i, v_j).

The length of a path is the number of edges in it. The goal in this question is to solve the following problem (see Figure 2 for an example).

Given an ordered graph G, find the length of the longest path that begins at v_1 and ends at v_n.

(a) Show that the following algorithm does not correctly solve this problem, by giving an example of an ordered graph on which it does not return the correct answer.

Set $w = v_1$
Set $L = 0$
while there is an edge out of the node w **do**
  Choose the edge (w, v_j) for which j is as small as possible
  Set $w = v_j$
  Increase L by 1
end while
return L as the length of the longest path

(b) Give an efficient algorithm that takes an ordered graph G and returns the length of the longest path that begins at v_1 and ends at v_n . (Again, the length of a path is the number of edges in the path.)

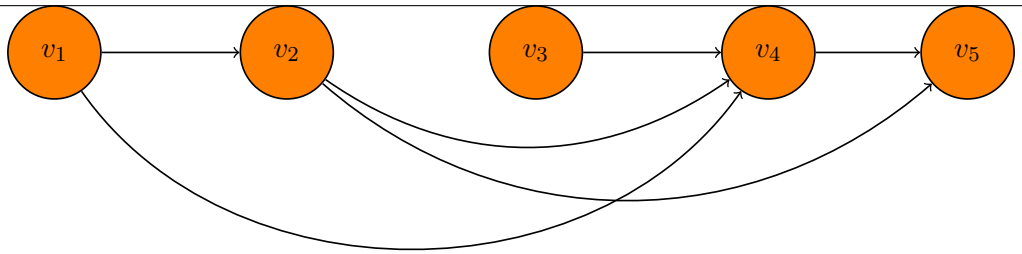

Figure 2: Problem 5

Solution:

(a) The graph on nodes $v_1, \cdots, v_5$ with edges $(v_1, v_2), (v_1, v_3), (v_2, v_5), (v_3, v_4)$ and (v_4, v_5) is such an example. The algorithm will return 2 corresponding to the path of edges (v_1, v_2) and (v_2, v_5), while the optimum is 3 using the path (v_1, v_3), (v_3, v_4) and (v_4, v_5).

(b) The idea is to use dynamic programming. The simplest version to think uses the subproblems $OPT[i]$ for the length of the longest path from v_1 to v_i. One point to be careful is that not all nodes v_i necessarily have a path from v_1 to v_i. We will use the value "$-\infty$" for the $OPT[i]$ values in this case. We will use $OPT[1] = 0$ since the longest path from v_1 to v_1 is 0 edges.

Array $M[1 \cdots n]$
$M[1] = 0$
for $i = 2, \cdots, n$ **do** $M = -\infty$
  for all edges (j, i) **do**
    if $M[j] \neq -\infty$ **then**
      if $M < M[j] + 1$ **then**
        $M = M[j] + 1$
      end if
    end if
  end for
$M[i] = M$
end for
return $M[n]$ as the length of the longest path.

Problem 6 [15 pts]

In a word processor, the goal of *pretty-printing* is to take text with a ragged right margin, like this,

```
Call me Ishmael.
Some years ago,
```

```
never mind how long precisely,
having little or no money in my purse,
and nothing particular to interest me on shore,
I thought I would sail about a little
and see the watery part of the world.
```

and turn it into text whose right margin is as *even* as possible, like this.

```
Call me Ishmael.  Some years ago, never
mind how long precisely, having little
or no money in my purse, and nothing
particular to interest me on shore, I
thought I would sail about a little
and see the watery part of the world.
```

To make this precise enough for us to start thinking about how to write a pretty-printer for text, we need to figure out what it means for the right margins to be *even*. So suppose our text consists of a sequence of words, $W = \{w_1, w_2, \cdots, w_n\}$, where w_i consists of c_i characters. We have a maximum line length of L. We will assume we have a fixed-width font and ignore issues of punctuation or hyphenation. A formatting of W consists of a partition of the words in W into lines. In the words assigned to a single line, there should be a space after each word except the last; and so if $w_j, w_{j+1}, \cdots, w_k$ are assigned to one line, then we should have

$$\left[\sum_{i=j}^{k-1}(c_i + 1)\right] + c_k \leq L$$

We will call an assignment of words to a line valid if it satisfies this inequality. The difference between the left-hand side and the right-hand side will be called the slack of the line, that is, the number of spaces left at the right margin. Give an efficient algorithm to find a partition of a set of words W into valid lines, so that the sum of the squares of the slacks of all lines (including the last line) is minimized.

Solution: We observe that the last line ends with word w_n and has to start with some word w_j; breaking off words $w_j, \cdots, w_n$ we are ledt with a recursive sub-problem on $w_1, \cdots, w_{j-1}$. We define $OPT[i]$ be the value of the optimal solution on the set of words $W_i = \{w_1, \cdots, w_i\}$. For any $i \leq j$, let $S_{i,j}$ denote the slack of a line containing the words $w_i, \cdots w_j$; let $S_{i,j} = \infty$ if these words exceed total length L. For each fixed i, we can compute all $S_{i,j}$ in $O(n)$ time by considering values of j in increasing order; thus we can compute all $S_{i,j}$ in $O(n^2)$ time.

As noted above, the optimal solution must begin with the last line somewhere (at word w_j), and solve the sub-problem on the earlier lines optimally. Thus we have the recurrence

$$OPT[n] = \min_{1 \leq j \leq n} S_{j,n} + OPT[j-1]$$

and the line of words $w_j, \cdots, w_n$ is used in an optimum solution if and only if the minimum is obtained using index j. Finally, we just need a loop to build up these values.

Compute all values $S_{i,j}$ as described above
$OPT(0) = 0$
for $k = 1, \cdots, n$ **do**
$OPT[k] = \min_{1 \le j \le k} S_{j,k} + OPT[j-1]$
end for
return $OPT(n)$

It takes $O(n^2)$ time to compute all values of $S_{i,j}$. Each iteration of the loop takes time $O(n)$, and there are $O(n)$ iterations. Thus total running time is $O(n^2)$. By tracing back through the array OPT, we can recover the optimal sequence of line breaks that achieve the $OPT[n]$ in $O(n)$ additional time.

Problem 7 [15 pts]

The residents of the underground city of Zion defend themselves through a combination of kung fu, heavy artillery, and efficient algorithms. Recently they have become interested in automated methods that can help fend off attacks by swarms of robots. Here's what one of these robot attacks looks like.

- A swarm of robots arrives over the course of n seconds; in the i^{th} second, x_i robots arrive. Based on remote sensing data, you know this sequence $x_1, x_2, \cdots, x_n$ in advance.
- You have at your disposal an electromagnetic pulse (EMP), which can destroy some of the robots as they arrive; the EMP's power depends on how long it's been allowed to charge up. To make this precise, there is a function $f(\cdot)$ so that if j seconds have passed since the EMP was last used, then it is capable of destroying up to $f(j)$ robots.
- So specifically, if it is used in the k th second, and it has been j seconds since it was previously used, then it will destroy $\min\{x_k, f(j)\}$ robots. (After this use, it will be completely drained.)
- We will also assume that the EMP starts off completely drained, so if it is used for the first time in the j^{th} second, then it is capable of destroying up to $f(j)$ robots.

The Problem: Given the data on robot arrivals $x_1, x_2, \cdots, x_n$, and given the recharging function $f(\cdot)$, choose the points in time at which you're going to activate the EMP so as to destroy as many robots as possible.

Example: Suppose $n = 4$, and the values of x_i and $f(i)$ are given by the following table.

i	**1**	**2**	**3**	**4**
x_i	1	10	10	1
$f(i)$	1	2	4	8

The best solution would be to activate the EMP in the 3 rd and the 4 th seconds. In the 3 rd second, the EMP has gotten to charge for 3 seconds, and so it destroys $\min\{10, 4\} = 4$ robots; In the 4 th second, the EMP has only gotten to charge for 1 second since its last use, and it destroys $min\{1, 1\} = 1$ robot. This is a total of 5.

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

Schedule-EMP $(x_1, \cdots, x_n)$
Let j be the smallest number for which $f(j) \geq x_n$ (If no such j exists then set $j = n$)
Activate the EMP in the n^{th} second
If $n - j \geq 1$ then Continue recursively on the input $x_1, \cdots, x_{n-j}$ (i.e., invoke Schedule-EMP $(x_1, \cdots, x_{n-j})$)

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes the data on robot arrivals $x_1, x_2, \cdots, x_n$, and the recharging function $f(\cdot)$, and returns the maximum number of robots that can be destroyed by a sequence of EMP activations.

Solution:

(a) Change x_4 to 2 in the given example. Then this algorithm would activate the EMP at times 2 and 4, for a total of 4 destroyed; but activating at times 3 and 4 as before still gets 5.

(b) The $OPT(j)$ be the maximum number of robots that be destroyed for the instance of the problem just on $x_1, \cdots, x_j$. Clearly if the input ends at x_j, there is no reason not to activate the EMP then (you're not saving it for anything), so the choice is just when to last activate it before step j. Thus $OPT(j)$ is the best of these choices over all i:

$$OPT(j) = \max_{0 \leq i \leq j} \Big[OPT(i) + \min(x_j, f(j-i)) \Big]$$

$OPT(0) = 0$
for there is an edge out of the node w **do**
Compute $OPT(j)$ using the recurrence
end for
return $OPT(n)$

The running time is $O(n)$ per iteration, for a total of $O(n^2)$.