CPSC 413: Assignment 2 Solutions

1. *A biologist is looking at n samples of DNA. Finding the exact DNA code for each sample is quite a slow process. However, all the biologist wants to know is whether at least half the samples come from the same animal. He has a method to quickly test whether any pair of DNA samples comes from the same animal. Design a divide-and-conquer algorithm that takes as input a set of n DNA samples, and determines if at least half of the group come from the same animal by only testing $O(n \log n)$ pairs.*

   Let us begin by thinking about a solution that runs in $O(n^2)$ time. We could compare the first sample to every other sample: if it is the same as at least half of them, we are done. If not, we could then test the second element against every other sample but the 1st; if it is the same as at least half, we are done. Continuing on in this way guarantees that we either find a set of at least half being the same, or guarantee that none exists. Unfortunately, this solution will take $O(n^2)$ time.

   So, we need some way to consider fewer samples to test. Suppose we had a set of $n/2$ samples which all came from the same animal. Then we must have either $n/4$ of these elements in the first half of the list, or $n/4$ of these elements in the second half of the list (if neither of these are true, then we can't have $n/2$ elements!). So, we will use a divide and conquer approach. We recursively run our function on the first half of the list and the second half of the list, with the function returning 1 or 2 elements that are the same as half of that sublist. Thus, from our two recursive calls, we will get a maximum of 4 possible samples that could be the same as half the samples in the whole list. We then test each of these four (or fewer) samples against every other sample; if any of them are the same as as least half, we return that sample, if none exists, we return empty. Thus we will have a divide and conquer algorithm that splits the input into two pieces of size half, and with that data, does $O(n)$ further calculations. From class, we know such an algorithm runs in $O(n^2)$ time.

   Finally, if the input is of size 2, we simply return both of the elements in the array, as each one individually is half of the array. The algorithm in full:

```
FindHalfElement(input: array $A$ of samples)
  • initialize $R$ to be an empty set;
  • if $|A| = 2$, add $a[1]$ and $a[2]$ to $R$;
  • else;
      – let $C$ =FindHalfElement(first half of $A$);
      – let $D$ =FindHalfElement(second half of $A$);
      – for each item $A[i]$ in $C$ and $D$,
          ∗ test it against all elements in $A$, if it comes from the same
            animal as at least half, add $A[i]$ to R,
  • endif;
  • return R.
```

If R contains at least one element, then we return true, otherwise false.

Testing an element against every other element in A takes $O(n)$ time, and we do this at most four times, so total time after the recursive calls is $O(n)$. Thus, since have a divide and conquer algorithm that splits the input into two pieces of size $n/2$ each, and takes $O(n)$ outside the recursive calls, the total running time is $O(n \log n)$, as required.

For each of the next three problems, describe a polynomial-time dynamic programming solution to the problem, prove that it always returns the correct answer, and determine its running time.

2. *Each week, you have the choice between heating your house with solar power, or with gas. For each the of the next n weeks, you can estimate how much each heating type will cost you; say it will cost s_i dollars if you heat with solar power in month i, and g_i dollars if you heat with gas power in month i. Unfortunately, you can't simply switch back and forth between the two heating types easily; doing so costs p dollars each week you switch from solar to gas or vice versa. Give an algorithm that determines which weeks to switch from solar to gas and back so as to minimize your total costs (it should also determine which type of heating to start with). For example, if we had*

$$s_1 = \$10, s_2 = \$5, s_3 = \$7,$$

and

$$g_1 = \$5, g_2 = \$12, \$g_3 = \$4,$$

and $p = \$4$, *then the optimal solution would be to start with gas, then switch to solar for weeks two and three.*

We are going to use an array $A[i, g$ or $s]$ which contains the least cost of heating the first $i-1$ months, given that we will heat the ith month with gas (if the second variable is g), or solar (if the second variable is s). Then note that we can calculate $A[i, g]$ by the following formula:

$$A[i, g] = \text{Min}(A[i-1, g] + g_{i-1}, A[i-1, s] + s_{i-1} + p)$$

since we either choose to stay with gas, or switch to solar in the $(i-1)$st month, incurring cost p as we switch from month $i-1$ to month i. Similarly,

$$A[i, s] = \text{Min}(A[i-1, s] + s_{i-1}, A[i-1, g] + g_{i-1} + p).$$

Of course, we have to initally set $A[1, s] = A[1, g] = 0$. Thus, our algorithm to find the minimum cost for each month is

```
• initialize $A[1..n, g$ or $s]$;
• set $A[1, g] = 0$;
• set $A[1, s] = 0$;
• for each $2 \leq i \leq n$
    – set $A[i, g] = \text{Min}(A[i-1, g] + g_{i-1}, A[i-1, s] + s_{i-1} + p)$;
    – set $A[i, s] = \text{Min}(A[i-1, s] + s_{i-1}, A[i-1, g] + g_{i-1} + p)$;
• endfor
```

Once we have found these values, we then find

- $\text{Min}(A[n, g] + g_n, A[n, s] + s_n)$.

which gives the optimal value of heating all n months. To find the actual solution, we work backwards through this array of values. Define the following procedure:

FindOpt(i, $x = g$ or s)

```
• if $i = 1$ do nothing
• else if $x = g$,
    – if $A[i-1, g] + g_{i-1} \leq A[i-1, s] + s_{i-1} + p$ output "gas in month
      $i - 1$", run FindSolution($i - 1, g$);
    – else output "solar in month $i - 1$", run FindSolution($i - 1, s$);
• else if $x = s$,
    – if $A[i-1, s] + s_{i-1} \leq A[i-1, s] + g_{i-1} + p$, output "solar in
      month $i - 1$", run FindSolution($i - 1, s$);
    – else output "gas in month $i - 1$", run FindSolution($i - 1, g$);
• endif
```

We then run the following code, which sees what should happen in the last month, then checks previous months:

```
• if $A[n, g] + g_n \leq A[n, s] + s_n$
    – output "gas in month $n$";
    – run FindSolution($n, g$);
• else
    – output "solar in month $n$";
    – run FindSolution($n, s$);
• endif.
```

The total running time is $O(n)$ for the first procedure, and $O(n)$ for the second, for a total running time of $O(n)$. For the proof of correctness, we claim that $A[n, g]$ contains the minimal cost to heat months $0, 1 ... n-1$ if we must heat month n with gas, and $A[n, s]$ contains the minimal cost to heat months $0, 1 ... n-1$ if we must heat month n with solar.

We will prove this by induction on n. If $n = 1$, then there is no cost to heat the 0th month, and we set $A[1, s] = A[1, g] = 0$, so we are done. Now suppose the result is true for n, and consider $A[n+1, g]$. Let O be a solution to finding the minimal cost on the first n months if we must heat month $n+1$ with gas. If O heats month n with gas, then its value is given by

$$g_n + (\text{Optimal value on first } n-1 \text{ months where month } n \text{ is gas})$$

which is equal to

$$g_n + A[n-1, g]$$

by the induction assumption. Similarly, if O heats month n with solar, then its value is given by

$s_n + p +$ (Optimal value on first $n-1$ months where month n is solar)

which equals

$$s_n + p + A[n-1, s]$$

by the induction assumption. Since our algorithm takes the minimum of these two values, we have $A[n+1, g] \leq O$, as required. The proof is similar for $A[n+1, s]$. Thus, our array contains the optimal values, as required.

3. *At your company, you've just found out that a number of documents have become corrupted - somehow, the documents have lost all spacing between words. What you need to do is design a program that can turn such a document into the best possible readable file, by restoring the spaces in their most logical places. For example, the program should be able to take the sequence "Ihadanicedaytoday" and return "I had a nice day today", rather than "Ihad anice dayt oday". To help with this task, you've have a function "Quality" available to you that takes in a string of characters, and returns a real number (that could be positive or negative), representing how close that string is to a real word. For example, it might return a negative value for "Ihad" and a positive one for "had". Supposing that Quality runs in polynomial time, give an algorithm that takes a string and returns where to insert spaces so as to maximize the total quality of the strings seperated by the spaces.*

   Consider an optimal solution O. At some point j (j may be 0), O inserts its last space. Then all the letters from j to n are together, and so O would add the quality of the string from j to n. The rest of the value of O would be whatever the best value on the string $1 ... j-1$ is. That is, the total quality of the string that O has is

   Quality(j, n) + (whatever the optimal value on the string $1 ... j-1$ is).

Thus, in general, if we let $A[i]$ be the optimal total quality on the string $1...i$, then we have

$$A[i] = \max_{1 \leq j \leq i-1}(\text{Quality}(j, i) + A[j]).$$

So, we write the following procedure:

```
• initialize $A[0...n]$;
• set $A[0] = 0$;
• for each $1 \leq i \leq n$;
    – set $A[i] = \max_{1 \leq j \leq i-1}(\text{Quality}(j, i) + A[j])$;
• endfor.
```

This only gives the optimal total quality. To find where we insert the spaces, we work backwards through the array A. Define the following procedure:

```
FindSolution($i$)
• if $i = 0$ do nothing;
• else
    – set $j =$ to be the value between 0 and $i-1$ for which $(\text{Quality}(j, i) + A[j])$ is maximized;
    – output "Insert space between the $j$ and $j - 1$ characters"
    – run FindSolution($j - 1$)
• endif
```

We then simply run FindSolution(n) to find where to insert spaces.

Both the first and second procedure are polynomial in n, assuming that quality is polynomial. For the proof of correctness, we claim that $A[i]$ holds the optimal total quality on the first i characters. We prove this by induction on n. If $n = 0$, there is no string, so no quality, and we set $A[0] = 0$. Now assume the result is true for n, and consider $A[n + 1]$. Let O be an optimal solution to finding the maximum total quality on the first $n+1$ characters, and let j be the last spot where 0 inserts a space. Then the total quality of O is

Quality$(j, n+1)$+(whatever the optimal value on the string $1...j-1$ is).

By induction, this is equal to

$$\text{Quality}(j, n+1) + A[j-1].$$

But our algorithm takes the maximum over all possible j's, so $A[n+1]$ is greater than or equal to the value of O. Since O is optimal, our algorithm is optimal as well, as required.

4. *In the United States, a process called "gerrymandering" is used to change electoral disctricts in a candidates favour. Suppose we know, in each of n precincts, how many people voted for party A (a_i), and how many voted for party B (b_i). The candidate in power, who runs for party A, wants to seperate these precincts into two different districts (each of size $n/2$), so that his party has the majority in both of the districts. For example, suppose that we have four precincts, with*

$$a_1 = 55, a_2 = 43, a_3 = 60, a_4 = 47$$

and

$$b_1 = 45, b_2 = 57, b_3 = 40, b_4 = 40.$$

If he grouped precincts 1 and 4 together, and 2 and 3, together then he would have the majority in both of these new districts (102 to 95 in precincts 1 and 4, and 103 to 97 in precincts 2 and 3). Write an algorithm that, given the voting numbers for the n precincts, determines if it is possible to split the precincts into two districts, with A having the majority in both of the districts.

First of all, let us simplify things slightly by defining $f_i = a_i - b_i$, so that f_i is the difference between the A voters and the B voters in precinct i. Now, we are looking for a way to divide the precincts into two equal-sized districts so that the sum of the f_i in both districts is positive.

When we think about a dynamic programming solution, the following comes to mind: either we put the last precinct in district 1, or in district

2. Then consider the remaining $n-1$ precincts. No longer should they be split into two equal groups; instead, we must put $n/2-1$ into the first district, and the remainder in the 2nd. Moreover, we are no longer looking for a positive number in district 1. Instead, we are looking for a positive number, given that we have already added f_n to district 1.

Thus, we will need an array which holds the values of all these possible subproblems: we will try to assign $A[i,p,q,k]$ to be true if it is possible to split up the first i districts so that the total in district 1 is positive (given that we have already added p to district 1), the total in district 2 is positive (given that we have already added q to district 2), and we assign exactly k of the i precincts to district 1. Then since we can either assign precinct i to district 1 or district 2, $A[i,p,q,k]$ is true if and only if

$$[(A[i-1,p+f_i,q,k-1] \text{ and } (k \geq 1)] \text{ or } [(A[i-1,p,q+f_i,k] \text{ and } i \geq 1)]$$

is true (the condition $k \geq 1$ ensure that we still can assign a precinct to district 1, and the condition $i \geq 1$ ensures we still can assign a precinct to district 2). Finally, notice that the value $A[n,0,0,n/2]$ will hold the value of the problem itself: it asks whether it is possible to split up the n precincts so that there is a majority (minus 0) in district 1, a majority (minus 0) in district 2, and we assign $n/2$ precincts to district 1.

So, to solve the problem, we need to solve each of the subproblems $A[i,p,q,k]$. There is an important initial case to take care of. We should set $A[0,p,q,0]$ to be true if and only if p and q are positive. That is, once we have no more districts to assign, we must be left with a positive number in both districts 1 and 2. One final thing we need to take care of is to determine the bounds for each of the variables. i ranges from 0 to n, and k from 0 to i. For p and q, the most they could ever be is the sum of all positive f_i (call this value U), and the least p and q could ever be is the sum of all negative f_i (call this L). Thus, we have our algorithm:

- Initialize $A[i,p,q,k]$ for $0 \leq i \leq n$, $L \leq p,q \leq U$, $0 \leq k \leq i$;
- Set $A[0,p,q,0]$ to be true if and only if p and q are positive;

```
• for each $1 \leq i \leq n$;
• for each $L \leq p \leq U$;
• for each $L \leq q \leq U$;
• for each $0 \leq k \leq i$;
    – set $A[i,p,q,k] = [(A[i-1,p+f_i,q,k-1]$ and $(k \geq 1)]$ or $[(A[i-1,p,q+f_i,k]$ and $i \geq 1)]$;
• endfor
• endfor
• endfor
• endfor
```

Then, as mentioned above, it is possible to assign the n precincts to districts if and only if $A[n,0,0,n/2]$ is true. Since the question does not ask for how to split the precincts up, only whether it is possible to split them up, this is all we need.

If we let $S = U - L$, then the running time of the algorithm is $O(n^2S^2)$, as there are four for loops, one of size n, two of size $U - L$, and one of size less than or equal to n. To prove the correctness, we claim that $A[i,p,q,k]$ is true if and only if it is possible to split the first i precincts into districts 1 and 2, so that there is a majority minus p in district 1, a majority minus q in district 2, and exactly k of the i precincts are assigned to district 1. For $n = 0$, the only way this is possible is if $p, q > 0$, and we set these values in our algorithm. Now suppose the result is true for n. We claim that for any p, q, k, the result is true for $n + 1$. Indeed, there is a way to split the first $n+1$ precincts into two districts if and only if we can either assign the $n+1$st precinct into either district 1 or district 2, so that the remainder has a majority (minus f_{n+1}) in the first district if we assign it district 1, and a majority (minus f_{n+1}) in the second district if we assign it district 2. But, by induction assumption, whether this is possible is given by $A[n, p+f_{n+1}, q, k-1]$ in the first case, and $A[n, p, q+f_{n+1}, k]$ in the second case. Since we consider both these values when assigning $A[n+1,p,q,k]$ in our algorithm, $A[n+1,p,q,k]$ is true if and only if it possible to split the k i precincts into two groups so that we get a majority (minus p) in the first, a majority (minus q) in the second, and we assign exactly k to district 1.

5. *An ecologist has been observing the locations of where a number of animals have drunk water. She keeps this data in an array Water$[i, j]$, in which the (i, j)th entry of the array contains how many times the ith animal has drunk water at the jth location (suppose there are n animals and m locations). She would like to determine, given an integer $k \leq n$, if there is a set of k animals which have never drunk water from the same location. Show that this problem is NP-complete.*

Call this problem **Water**. We must first show that **Water** is in NP. Suppose we are given a set A of m animals. We first test whether $m >= k$: if it is not, we return false. If it is, we then check, for each j and pair of animals a, b in A, whether both Water$[a, j]$ and Water$[b, j]$ are greater than zero. If they are, we return false. Otherwise, we return true. This algorithm runs in polynomial time, and returns true if and only A is of size at least k, and no set of animals in A has drunk from the same location j.

To show that it is NP-complete, we will reduce **IndSet** to it. Suppose we have a graph $G = (V, E)$, and we want to see if there is an independent set of size k or greater. We define an instance of the problem **Water**. For each vertex v in the graph, make an animal v. For each edge e in the graph, make a water location e. Then, for each edge e, if e goes from v to w, set Water$[v, e] = 1$ and Water$[w, e] = 1$. Set all other values of the array to 0.

We claim that a set of vertices S in G is independent if and only the set of animals S solves the water problem. If S is independent, then no two vertices in S share an edge, so for any edge e, and v, w in S, Water$[v, e] = 0$ or Water$[w, e] = 0$, so the set of animals S solves the water problem. Conversely, if a set S of animals solves the water problem, then for any edge e, and vertices v, w in S, Water$[v, e] = 0$ or Water$[w, e] = 0$, so no vertices in S share an edge.

Thus, **IndSet** $\leq_p$ **Water**. Since **IndSet** is NP-complete, so is **Water**.