# Homework 5

Simon Zheng
260744353

December 3rd, 2017

## 1 Closest pair of points

The algorithm should mostly stay the same except at the part where we must search for a possible closer pair of points crossing the middle line.

For this formula, we must check the next 12 closest points.

When create the "boxes", where we the two sides of the dividing line into 6 boxes of equal size each of $\frac{1}{2}\alpha$

## 2 Cell-hopping frog

We can make some assumptions from the statement "*At every step, the frog can jump from its current cell* $(i, j)$ *to any cell* $(i\prime, j\prime)$ *that satisfies* $i\prime + j\prime = i + j + 1$":

The frog can only ever move up and right. When the frog is at $(1, 1)$, it can only move to $(1, 2)$ or $(2, 1)$. These two cells form a sort of diagonal perpendicular to $(1, 1)$.

It is the same for the next hop: The frog can only move to $(1, 3), (2, 2), (3, 1)$ which form a sort of diagonal perpendicular to the movement.

This is always the case because this is how we can satisfy the equation for allowed cell movements.

But the more interesting thing is that, when at any diagonal (or "level"), which represents the number of hops the frog has done, the next hop always has $i + j$ possible cells. E.g. $(1, 1)$ has $1 + 1 = 2$ hop options to $(1, 2), (2, 1)$, which has $1 + 2 = 2 + 1 = 3$ hop options to $(1, 3), (2, 2), (3, 1)$, etc.

Now, if the table isn't already created, create it (as a double array matrix or graph, it does not matter) in $O(n^2)$ in such a way that we can store a value inside.

Starting from $(1, 1)$, store a value of 1 in it. This is the base case.

Next, the algorithm stores the same value of 1 in the next "level" (cells in the next diagonal) $(1, 2), (2, 1)$.

Same for the next one $((1, 3), (2, 2), (3, 1))$, except that you add up all the values from the cells of the previous diagonal.

The idea is that every cell stores this computed value from their "parents" or previous cells (from the previous diagonal). Thus, we will "push" the added up values of every possible path to the destination and then we just get the final value from the destination cell.

If a cell is marked as forbidden, just put 0, and as such, we stop it from adding to future paths. If an entire diagonal is full of forbidden cells, then everything will add up to 0, indicating there are 0 possible paths.

To make things slightly more efficient, we can store 0 when the matrix or graph is created.

This still requires visiting every cell, but only once, until we reach the level with the destination in it, and we only need to add up values from the "parents"/previous diagonal cells, which would be visiting them a second time only. Thus, this is $O(2n^2) = O(n^2)$ (if we must create the cells it is just one other $n^2$ operation so it stays the same).

## 3 Largest subsequence

I will use the example sequence given in the problem $1, 3, 9, 8, 5, 7, 6$ where the answer is $1, 9, 5, 7, 6$ to help explain the algorithm.

The way we can do this is by first creating a table of size $n \times n$. The columns will be the numbers of the sequence in order (each $i$-th column is for the $i$-th number, so it will look like $1, 3, 9, ...$) and the rows are the number $i$, representing the longest subsequence before the $i$-th term (so going from 1 to $n$). First row is for subsequences of length 1, second for those of length 2, etc. This will be explained further next.

For the first two rows, they are special cases (base ones) as the problem requires a sequence of at least length 3.

In the first row, for each column that represents the $i$-th term, we store every $j$-th number where $j \leq i$. So as an example for the column for 8, the first row will contain $1, 3, 9, 8$. This is done as it will be convenient later.

In the second row, we store a sort of power set of the subsequence of the first row for each $i$-th column but only with sets of length 2 and sorted. We do this to get all the pairs representing a possible range from all the $j$-th terms before the $i$-th term for all the $i$-th column. So for the column for the the number 8 (fourth term), its second row would contain $(1, 3), (1, 8), (1, 9), (3, 8), (3, 9), (8, 9)$. The first column would just be empty since this is not possible for a single term (subsequence of length 1).

For the next $k$-th row (labelled as the number $k$), what we put in the $k$-th row for the $i$-th column will be the set of all the possible subsequences in which the $i$-th term can fit in with the $i$-th term concatenated to it.

Explanation:

For the third row, for each $i$-th column, look at every pair representing a range in the second row (in the same column, as it will contain all the previous pairs too the way we created it).

For each pair, if the current $i$-th term can fit in its range, concatenate the pair with the current $i$-th term being considered, and put this subsequence in the the same column in the third row (since we are doing the third row now). So if we take the term 8 as an example, in its third row, it would contain $[1, 9, 8], [3, 9, 8]$ since 8 fits between 1 and 9 or 3 and 9.

If there are no possible subsequences, just leave the "cell" empty. We also do not need to consider the second column since we cannot have subsequences of length 2 or smaller (so $i$ must be greater than 2).

For the fourth row, instead of looking at the pair of ranges from the second row, we look

at the third row. The third row now contains the previously constructed subsequences. We do the same thing. But now we also do not need to consider the third column since we cannot have subsequences of length 3 or less.

But now where are the pairs of ranges to consider from the third row if there only are subsequences of length 3? Well, in accordance with the problem, we only consider the two last terms of each of the subsequences of the third row as pairs of ranges.

So for the fourth row, in the column for the term 8, we'd look at the third row and consider the pair of ranges $(8, 9), (8, 9)$ taken from the two last terms of $[1, 9, 8], [3, 9, 8]$.

Continue doing this for each $k$-th row, and at each row there is an extra column we do not need to consider because of the minimum possible length of subsequence at the $i$-th term (and $k$ is the length of the subsequences being considered).

Thus, for the first three rows and columns we would get:

| Length \ Term | 1 | 3 | 9 | 8 |
|---|---|---|---|---|
| 1 | 1 | 1,3 | 1,3,9 | 1,3,9,8 |
| 2 | X | (1,3) | (1,3), (1,9), (3,9) | (1,3); (1,8); (1,9); (3,8); (3,9) |
| 3 | X | X | none | [1,9,8]; [3,9,8] |
| 4 | X | X | X | none |

So in the fourth row, fourth column (not counting labels), the ranges to consider are found by looking at the last two terms of $[1, 9, 8]; [3, 9, 8]$ so $(8, 9)$.

When this is done for the whole table (the whole subsequence), we just find the bottom-rightmost non-empty cell and pick any of the subsequences (as they will all be of equal length).

So, say, the first one, just so the algorithm can choose.

Creating the table is $O(n^2)$, but having to process and go through the previous sequences every time makes it $O(n^3)$. To clarify, creating the first row is $O(n)$ since we just concatenate the current term to the previous subsequence. The second row takes $O(n^2)$, same for every successive row, so $O(n^3)$.

We can run an $O(n)$ algorithm that checks if it is a monotonically increasing sequence (which means there are no solutions) or the main algorithm will just return empty.

## 4 Job processing with weights and deadlines

First, we sort the jobs by deadline (in a list or an array) and by processing time (this can be done in the same sort operation, so $O(n \log n)$).

Then, starting from the beginning (earliest deadline), and going job by job (so deadline by deadline), add up the profits and processing times, so every job gets added to one big "cumulative" job and keep track of it (its cumulative processing time, deadline, and profit).

In more details, at every $i$-th job when considering it, choose the cumulative job to be the max between the previous cumulative job ($i - 1$-ith) plus the current $i$-th job, or

## 5 Currency exchange

So we know that an $\alpha$ value between 0 and 1 is a disadvantageous exchange, and anything greater than 1 is advantageous.

We can use this fact and the fact that the logarithm of a number between 0 and 1 is negative to find an "advantageous cycle".

First, we convert the weight of every edge to its negative log, which is $O(n+m)$ where there are $n$ nodes and $m$ edges.

After this, we can use the algorithm seen in class to find a negative cycle, which is what we want. This is the reason we want the negative of the logarithm, as without this, disadvantageous $\alpha$ weights would be negative and the opposite for advantageous ones, yet the algorithm searches for negative cycles.

Algorithm to find the shortest path with negative weights:

Input: A node $u$ in a graph $G$
Output: The nodes of the shortest path

```
1     D[s] = 0 and D[v] = ∞ where v ≠ s
2
3     For i = 1 to n − 1
4       ForAll v
5         x = min_{wv∈E} c_wv + D[w]
6         If x < D[v]
7           D[v] = x
8         EndIf
9       EndFor
10    EndFor
```

As seen in class, the minimum shortest path has at most $n-1$ length where $n$ is the number of nodes. If values are still being updated after visiting more, we know we have a negative cycle.

But in this case we *want* find any negative cycle. As such, when we detect a negative cycle, we just take the first point that gets repeatedly updated as the starting point and the following nodes are part of the path needed to get a negative cycle.

Thus, we get the answer in $O(nm)$ since it is the same as the normal algorithm.

The running time is the same, and added with converting the weights to the negative logarithm, this gives us $O(n+m)+O(nm)=O(nm)$.

## 6 Number of shortest paths

We can use the same algorithm then for question 2 (frog hopping on cells):

Run $BFS$ on the graph from $s$ to $t$, so stop at $t$. At each node or vertex, add up the values stored in all of its "parents".

We initially put a value of 1 in $s$. A parent is defined as every neighbouring vertices of a vertex $u$ which has already been visited, and as such its children are defined as its neighbours that weren't visited.

Thus, each visited vertex will store the values of all the vertices that can reach it, and since we do it in a $BFS$ order, it is all the possible paths. Stopping at vertex $t$ in $BFS$ guarantees shortest path, and we just take the final value that will be stored in $t$.

The operation of adding can be done at the same time as when we get all the neighbours in a standard $BFS$ algorithm and is $O(1)$ for each.

Thus, the running time stays $O(n+m)$ where $n, m$ are respectively the number of vertices and edges.

If $BFS$ never reaches $t$, then there is no solution.