# Problem Set 6

**All parts are due Friday, May 9 at 11:59PM**. Please download the .zip archive for this problem set, and refer to the `README.txt` file for instructions on preparing your solutions. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

## Part A

**Problem 6-1.** [20 points] MIT student Lance is planning a recreational bicycle trip from his home in Bismarck, North Dakota, to Wood Buffalo, Alberta. He prefers to bike approximately 30 miles per day, rain or shine, and spend the rest of the time eating, sleeping, and thinking about algorithms. The entire trip will take exactly 40 days. Sounds like a nice vacation.

To plan the trip, Lance maps out a directed graph in the plane, with nodes corresponding to the starting point, the destination, and locations with affordable hotels. The directed edges correspond to roads between the hotels. Each edge has a weight corresponding to its length in miles. Lance would like a route that consists of exactly $40$ edges, each with a weight close to $30$. In fact, he is so determined that the distance each day should be close to $30$ that he would like to minimize the sum, over all the $40$ edges in the path, of $(weight(e) - 30)^2$.

Lance has not taken 6.006 yet. So you must help Lance to determine a good itinerary for his trip.

**(a)** [5 points] Give pseudocode for a recursive algorithm to solve a generalized variant of the problem, in which the inputs are:

- A weighted directed graph with designated start node $s$ and target (destination) node $t \neq s$. The weights are positive integers.
- A positive integer number $k$ of edges.
- A positive integer edge cost $m$.

The output should be the smallest cost for a route from $s$ to $t$ consisting of exactly $k$ edges. The cost of a route is the sum, over all edges $e$ of the route, of $(weight(e) - m)^2$. If no $k$-edge route exists, your algorithm should output $\infty$. Analyze the time complexity of your algorithm.

**(b)** [4 points] Use Dynamic Programming with memoization to make your recursive algorithm from Part (a) more efficient. Analyze the time complexity of the resulting improved algorithm.

**(c)** [4 points] Now use Dynamic Programming with bottom-up calculation to make your recursive algorithm from Part (a) more efficient. Analyze the time complexity of the resulting improved algorithm.

**(d)** [3 points] Describe a situation in which the memoized version of the algorithm is more efficient than the bottom-up version.

**(e)** [4 points] Modify one of your efficient algorithms to produce an actual best route. Analyze the time complexity of the path-finding algorithm.

**Problem 6-2.** [20 points] Suppose we are given a set $S$ of $n$ positive integers, in the form of an unsorted list without repeats.

**(a)** [4 points] Describe a simple, straightforward algorithm that, given $S$ as above, returns the length of a longest possible geometric progression that can be formed from the numbers in $S$. For example, if $S = \{27, 18, 1, 4, 3, 6, 96, 16, 9, 14\}$, then your answer should be $4$ (corresponding to the sequence $1, 3, 9, 27$). Your algorithm should run in time $O(n^3)$. Include an analysis.

**(b)** [6 points] We are going to try to improve on this algorithm using dynamic programming. But first, a warmup:

Describe an algorithm that, given a sorted list $L$ with $n$ distinct elements, and an integer $j$, $2 \leq j \leq n-1$, returns the list of all pairs $(i, k)$ such that the triple $L(i), L(j), L(k)$ forms a geometric progression. Your algorithm should run in time $O(n)$. Include an analysis.

**(c)** [10 points] Now describe another algorithm for the same problem as in Part (a), but with time complexity $O(n^2)$. Include an analysis.

**Problem 6-3.** [20 points] When a new gene is discovered, a standard approach to understanding its function is to look through a database of known genes and find close matches. The closeness of two genes is measured by the extent to which they are aligned. To formalize this, think of a gene as being a long string over an alphabet $\Sigma = \{A, C, G, T\}$. Consider two genes (strings) $x = ATGCC$ and $y = TACGCA$. An alignment of $x$ and $y$ is a way of matching up these two strings by writing them in columns, for instance:

```
-  A  T  -  G  C  C
T  A  -  C  G  C  A
```

Here the "-" indicates a "gap". The characters of each string must appear in order, and each column must contain a character from at least one of the strings, that is, it can't contain two gaps. The score of an alignment is determined using a scoring matrix $\delta$ of size $(|\Sigma|+1)$ by $(|\Sigma|+1)$, where the extra row and column are included to accommodate gaps. The entries in the matrix are integers, which may be positive or negative; positive is considered "good". For instance, the alignment above has the score

$$\delta(-, T) + \delta(A, A) + \delta(T, -) + \delta(-, C) + \delta(G, G) + \delta(C, C) + \delta(C, A).$$

**(a)** [10 points] Give a Dynamic Programming algorithm that takes as input two strings $x[1, \ldots, n]$ and $y[1, \ldots, m]$ and a scoring matrix $\delta$ and returns the highest-scoring alignment. Your algorithm should run in time $O(mn)$.

**(b)** [10 points] Often two DNA sequences are significantly different, but contain regions that are very similar. Design an algorithm that takes as input two strings $x[1, \ldots, n]$ and $y[1, \ldots, m]$ and a scoring matrix $\delta$ (as defined in Part (a)), and that outputs (contiguous) substrings $x'$ and $y'$ of $x$ and $y$, respectively, that have the highest-scoring alignment over all pairs of such (contiguous) substrings. Your algorithm should take time $O(mn)$.

## Part B

**Problem 6-4.** [40 points] **Supercolliding Super Conductor**

The Supercolliding Super Conductor (SSC) Project has finally been completed, somewhere in the vicinity of Waxahachie, Texas, using private investments. The owners of the Conductor have been overrun with requests from physicists all around the world for time slots on the Conductor. The owners can charge a lot of money for these time slots, and the machine was very expensive, so it is important to them to maximize the amount of time the Conductor is actually in use. They have asked MIT for help in optimizing the allocation of time slots to maximize utilization of the SSC.

**(a)** [40 points] Requests for use of the Conductor during a month-long period arrive before the month begins. Each request is for a particular time slot, consisting of an interval of consecutive minutes (in UTC time, meaning no time zones) during the month, e.g., "request ABC, minute 1413 through minute 1452" (a day consists of 1440 minutes, meaning a month consists of a large number of time slots, between 40320 (for a 28-day month) and 44640 (31 days). Regard the minutes as discrete time slots to be allocated, so the above request "ABC" is asking for 40 time slots.) A given request `ABC` is given as a `Request` object, which can be accessed via the following methods:

- `len(ABC)` returns the number of time slots requested, in this case `40`.
- `start(ABC)` or `ABC.start` returns the first time slot of the request, in this case `1413`.
- `end(ABC)` or `ABC.end` returns the last time slot of the request, in this case `1452`.
- `name(ABC)` or `ABC.name` returns the string name associated with the request, in this case "`ABC`".
- `str(ABC)` or `print ABC` returns a string representation of `ABC`, in this case "`ABC: [1413, 1452]`"

Implement an algorithm that takes as input `requests`, an unordered list of `Request` objects, and outputs an unordered list `schedule`, which enumerates the subset of `requests` to be granted (the remaining requests are denied). All requests are guaranteed to be "valid", meaning any non-overlapping subset of `requests` may be scheduled onto the Conductor.

Implement your solution in the `create_schedule` function in `PSET6.py`. The goal of your algorithm is to maximize profit, that is, to utilize the Conductor for the most possible time slots, regardless of how many requests are granted or denied to make that happen. The returned `schedule` should be a subset of `requests` but consist only of non-overlapping requests such that the total number of time slots in `schedule` is maximized. If multiple maximal schedules of equal length exist, any one of them is a valid choice.

For your convenience, a some infrastructure code is given to you:

- `is_overlapping(a, b)` takes two `Request` objects, and returns `true` if and only if request `a` includes one or more time slots also in request `b`.
- `sort_requests(requests, sort_by)` takes a list of `Request` objects and a "key function" `sort_by`, and returns a sorted list of the same requests ordered by a criteria specified via the key function. `sort_by` can be one of {`name`, `len`, `start`, `end`}.