# CS 330: Homework 4

Michael Lin

November 2, 2015

## Problem 1

(a) Algorithm that determines whether a point $q$ lies inside a convex polygon $P$.

**Algorithm 1** Algorithm that determines whether a point $q$ lies inside a convex polygon $P$

```
1:  function INSIDE(polygon P[1, ..., n], point q)
2:      if |P| = 3 then
3:          d_1 = DIRECTION(q, P[1], P[2])
4:          d_2 = DIRECTION(q, P[2], P[3])
5:          d_3 = DIRECTION(q, P[3], P[1])
6:          if max{d_1, d_2, d_3} ≤ 0 then
7:              return q inside P.
8:          else
9:              return q not inside P.
10:     if |P| ≥ 4 then
11:         y = line that connects P[1] and P[⌊n/2⌋]
12:         if q above the line y then
13:             return INSIDE(P[1, ..., ⌈n/2⌉], q)          ▷ recurse on top half of polygon
14:         else
15:             return INSIDE(P[1, ⌈n/2⌉, ..., n], q)       ▷ recurse on bottom half of the polygon
16:
17: function DIRECTION(a, b, c)
18:     d = det [1  a_x  a_y]
                [1  b_x  b_y]
                [1  c_x  c_y]
19:     return d
```

*Proof of correctness.* The algorithm recursively check which half of the polygon it could reside in. Certainly, if $q$ is above the line $y$, which roughly divides the polygon into halves, then $q$ cannot be inside the bottom half of the polygon. For cases where $q$ lies on the line $y$, we arbitrarily choose throw away the top half. Each time we throw away half of the points, the remaining polygon is still convex, which means the line segment that connects $P[1]$ and $P[\lceil n/2 \rceil]$ is entirely within $P$ (but rest of the line is outside $P$). Once we have only 3 points left, we check the signs of the cross products to determine whether $q, P[i \mod 3], P[i+1 \mod 3]$ are all right turns. Since the points are sorted radially, all points inside $P$, inclusive of the boundary, will have non-left turns (i.e. at least one of the cross products could be 0 if $q$ lies on the boundary). Points outside of $P$ will result in at least one left turn, which means a value of $d_i > 0$. Therefore, the algorithm correctly whether $q$ is inside $P$. □

*Proof of runtime.* The base case when $|P| = 3$ runs in constant time. For all other cases, we can write the following recurrence relation:

$$T(n) = T(n/2) + O(1)$$

since each time we compute the line $y$ takes only constant time, and we reduce the problem by half depending on whether $q$ lies above $y$ or not. By Master Theorem, $T(n) = \Theta(\log n)$. Thus, the algorithm runs in $O(\log n)$ time. □

(b) Algorithm that computes the tangents of a point $q$ to $P$. First, use algorithm in part (a) to determine if $q$ is inside $P$ or not. If $q$ is on or inside the boundary of $P$, then there are no tangents.

Otherwise, each point $q$ has a left tangent and a right tangent, which can be pictured by rotating the image so that the point $q$ is above the polygon. Let's first compute the left tangent. Choose an arbitrary vertex $P[i]$ and consider the helper function below. What we want to find is a tangent point $P[TL]$ such that

$$X_{TL} := (a_{TL}, b_{TL}) = \textsc{Helper}(P[TL], q) = (R, L)$$

which would tell us that this tangent point is the left tangent, since every left tangent is characterized by a right turn then left turn. For simplicity of notation, we will drop the $\mod n$, and any out of bound indices are implied as $\mod n$. First, compute

$$X_i := (a_i, b_i) = \textsc{Helper}(P[i], q)$$

then compute

$$X_{i+\lceil n/2 \rceil} := (a_{i+\lceil n/2 \rceil}, b_{i+\lceil n/2 \rceil}) = \textsc{Helper}(P[i + \lceil n/2 \rceil], q)$$

where the index of $P$ in calculating $X_{i+\lceil n/2 \rceil}$ is just $i$ shifted by $\lceil n/2 \rceil$, half the length of the list (if index out of bound, then loop back to beginning of the list, hence "$\mod n$"). Based on the values of $X_i$ and $X_{i+\lceil n/2 \rceil}$, we can make the following decisions:

(a) If $X_i = (R, L)$, then we are done, and we return $P[i]$ as the left tangent; similarly, if $X_{i+\lceil n/2 \rceil} = (R, L)$, then we are also done, and we return $P[i + \lceil n/2 \rceil]$ as the left tangent.

(b) If $X_i = (R, R)$ or $(L, R)$ and $X_{i+\lceil n/2 \rceil} = (L, L)$ or $(L, R)$, then we recurse on $P[i, \ldots, i + \lceil n/2 \rceil]$, where we can loop back to the beginning of list $P$ if $i > i + \lceil n/2 \rceil$.

(c) If $X_i = (L, L)$ or $(L, R)$ and $X_{i+\lceil n/2 \rceil} = (R, R)$ or $(L, R)$, then we recurse on $P[i + \lceil n/2 \rceil, \ldots, i]$, where we can loop back to the beginning of the list $P$ if $i < i + \lceil n/2 \rceil$.

(d) If $X_i = (R, R)$ and $X_{i+\lceil n/2 \rceil} = (R, R)$, then

   i. If $\textsc{Direction}(P[i], q, P[i + \lceil n/2 \rceil]) = L$, then recurse on $P[i + \lceil n/2 \rceil, \ldots, i]$
   ii. If $\textsc{Direction}(P[i], q, P[i + \lceil n/2 \rceil]) = R$, then recurse on $P[i, \ldots, i + \lceil n/2 \rceil]$

(e) If $X_i = (L, L)$ and $X_{i+\lceil n/2 \rceil} = (L, L)$, then

   i. If $\textsc{Direction}(P[i], q, P[i + \lceil n/2 \rceil]) = R$, then recurse on $P[i + \lceil n/2 \rceil, \ldots, i]$
   ii. If $\textsc{Direction}(P[i], q, P[i + \lceil n/2 \rceil]) = L$, then recurse on $P[i, \ldots, i + \lceil n/2 \rceil]$

To find the right tangent $TR$, we want

$$X_{TR} := (a_{TR}, b_{TR}) = \textsc{Helper}(P[TR], q) = (L, R)$$

since every right tangent is characterized by a left turn then right turn. The rest is similar:

(a) If $X_i = (L, R)$, then we are done, and we return $P[i]$ as the right tangent; similarly, if $X_{i+\lceil n/2 \rceil} = (L, R)$, then we are also done, and we return $P[i + \lceil n/2 \rceil]$ as the right tangent.

(b) If $X_i = (L, L)$ or $(R, L)$ and $X_{i+\lceil n/2 \rceil} = (R, R)$ or $(R, L)$, then we recurse on $P[i, \ldots, i + \lceil n/2 \rceil]$, where we can loop back to the beginning of list $P$ if $i > i + \lceil n/2 \rceil$.

(c) If $X_i = (R, R)$ or $(R, L)$ and $X_{i+\lceil n/2 \rceil} = (L, L)$ or $(R, L)$, then we recurse on $P[i + \lceil n/2 \rceil, \ldots, i]$, where we can loop back to the beginning of the list $P$ if $i < i + \lceil n/2 \rceil$.

(d) If $X_i = (R, R)$ and $X_{i+\lceil n/2 \rceil} = (R, R)$, then

   i. If $\textsc{Direction}(P[i], q, P[i + \lceil n/2 \rceil]) = L$, then recurse on $P[i + \lceil n/2 \rceil, \ldots, i]$
   ii. If $\textsc{Direction}(P[i], q, P[i + \lceil n/2 \rceil]) = R$, then recurse on $P[i, \ldots, i + \lceil n/2 \rceil]$

(e) If $X_i = (L, L)$ and $X_{i+\lceil n/2 \rceil} = (L, L)$, then

i. If DIRECTION$(P[i], q, P[i + \lceil n/2 \rceil]) = R$, then recurse on $P[i + \lceil n/2 \rceil, \ldots, i]$

ii. If DIRECTION$(P[i], q, P[i + \lceil n/2 \rceil]) = L$, then recurse on $P[i, \ldots, i + \lceil n/2 \rceil]$

**Algorithm 2** Functions for algorithm described above

```
1: function HELPER(vertex P[i], point q)
2:     a_i = DIRECTION(q, P[i − 1 mod n], P[i mod n])
3:     b_i = DIRECTION(q, P[i mod n], P[i + 1 mod n])
4:     return (a_i, b_i)
5:
6: function DIRECTION(a, b, c)
7:     d = det [1 a_x a_y; 1 b_x b_y; 1 c_x c_y]
8:     if d > 0 then return L                ▷ L for left, R for right, C for collinear
9:     if d < 0 then return R
10:    if d = 0 then return C
```

*Proof of correctness.* The algorithm relies on the fact that given a point $q$ outside of the polygon $P$, the angle between $\overrightarrow{qP[i]}$ and $\overrightarrow{P[i]P[i+1]}$ form a contiguous region of right and a contiguous region of left. The tangents are the boundaries of these regions; particularly, the left tangent as define above is where the region of "right turns" changes to region of "left turns", and the right tangent is where the region of "left turns" changes to region of "right turns". Given just these facts, we can almost always eliminate about half of the points in the polygon by inspecting two points (call these "test points")that are about $n/2$ points apart. The rest is logical: since we know points are ordered clockwise, if we want the the left tangent, i.e. find a point with $(R, L)$, then we determine which half of the list could contains the left tangent point. We can take a similar approach if we want the right tangent, i.e. find a point with $(L, R)$. The special case is where we can't deduce which half of the points to discard, namely when both "test points" are $(R, R)$ or $(L, L)$. Here, we must additionally use the information of the location of $q$. In addition, we observe that the region in between the tangent points on the polygon (i.e. starting from the left tangent point and moving clockwise until the right tangent point) will always yield $(L, L)$ with respect to $q$. Thus, if "test points" are both $(R, R)$, then the tangent points must be in half of the list that's on the same side as $q$ (e.g. connect and extend the line between the "test points"; we are interested in the side with point $q$ and the half of the polygon on the same side). Similar logic works if "test points" are both $(L, L)$. □

*Proof of runtime.* The algorithm first checks whether or not $q$ is on or within the boundary of $P$, which takes $O(\log n)$ time. After that, to find the left tangent, we can always reduce the problem to only half the original list. Each recursive step only takes a constant number of operations, namely to determine the direction of consecutive points with respect to $q$. Therefore, we can write the following recurrence relation:

$$T(n) = T(n/2) + O(1)$$

which is $O(\log n)$. The same recurrence relation holds for finding the right tangent. Since we can do these operations separately, we have a total of $2O(\log n)$ operations, which is still $O(\log n)$. □

# Problem 2

Algorithm to find a partition of a set of words $W$ into valid lines. Here, we use dynamic programming to solve the problem. We need the following variables:

- $S(i)$: the minimum sum of square of slack of having word $w_i$ as the last word on a line.
- $L(i, j)$: the slack on a line with words $w_j, w_{j+1}, \ldots, w_{i-1}, w_i$. In closed form, this corresponds to

$$L(i,j) = [L - (c_i + 1) - (c_{i-1} + 1) - \cdots - (c_{j+1} + 1) - c_j]^+$$

  where the $+$ denotes only non-negative values. $L(i, j)$ is undefined for values less than 0.
- $P(i)$: binary variable that keeps track of the first word of the line corresponding to $S(i)$.

In recursive notation, we have

$$S(i) = \min \left\{ L(i,k)^2 + S(k-1) \right\}_{k=1}^{i}$$

$$L(i,j) = L(i,j+1) - (c_j + 1)$$

where $j < i$ and $L(i, j + 1) > (c_j + 1)$.

$$P(i) = \begin{cases} x & \text{if } S(i) \equiv L(i,x)^2 + S(x-1) \\ 0 & \text{otherwise} \end{cases}$$

where the $\equiv$ implies that $S(i)$ is not just any value that equals $L(i, x)^2 + S(x - 1)$, but in fact the index of $L(i, x)$ that gives the minimum over all values $x$ from 1 to $i$. $P$ is defined for all values from 1 to $n$.

The base cases for $L$ are $L(i, i) = L - c_i$ for all $i$ if $L \geq c_i$. All values $L(i, j)$ where $j < i$ can be computed based on the recursive formula above. When $j > i$, $L(i, j)$ is undefined. The base case for $S$ is $S(0) = 0$ so that the recursive formula for $S$ is well-defined.

To save computation time, we memoize all values of $S$ in an array of length $n + 1$, all values of $L$ in a matrix of dimensions $n \times n$, and all values of $P$ in an array of length $n$. Compute all values of $L$ first, then compute $S$ bottom-up, i.e. in the order $S(1), S(2), \ldots, S(n)$.

To find the minimum sum of square slack, we return $S(n)$. Based on this value, we can partition the words from the end to the beginning based on values of $P$. Let $a := P(n)$, then last line of words consists of $\{w_a, w_{a+1}, \ldots, w_n\}$. Now, let $b := P(a-1)$, then second to last line of words consists of $\{w_b, w_{b+1}, \ldots, w_{a-1}\}$. Repeat this until no words are left.

*Proof of correctness.* The algorithm has two parts: the first part is computing the smallest sum of square slack, and the other part is partitioning the words based on the value from the first part. By way that $S$ and $L$ are defined, we are computing the sum of square slack of all possible combination of words on a line where the last word on the last line is $w_i$, and finding the minimum of them. Then, to find the minimum sum of square slack for all $n$ words, i.e. last word on the last line is $w_n$, we return $S(n)$.

While calculating $S(i)$ for each $i$, we also kept track of the first word $x$ in the line that ends with word $w_i$ that gave the smallest sum of square slack. Then, we just need to recursively search values in $P$ to the find the first word of each line, which then allows us to partition all words as desired. □

*Proof of runtime.* We need to compute values for three data structures:

1. $L(i, j)$ is an $n \times n$ matrix, and we need to compute all lower triangular values. Thus, we have about $n^2/2$, or $O(n^2)$, values to compute. By memoizing values of $L$, each new value depends only on the value to its right in the matrix, so each computation takes $O(1)$ time. Together, it takes $O(n^2)$ time to compute all of $L$.
2. $S(i)$ is an array with $n+1$ elements, so we have $O(n)$ elements that need to be computed. For each value of $S(i)$, we first need to compute the sequence $\left\{ L(i,k)^2 + S(k-1) \right\}_{k=1}^{i}$, which has an upper bound of $n$ values. Then, we need to find the minimum value in the sequence, which takes at most $O(n)$ time. Thus, for each value of $S$, we need only $O(n)$ time. To compute all of $S$ would take $O(n^2)$ time.

3. $P(i)$ is an array of $n$ values. Each value of $P$ can be computed simultaneous with each corresponding value of $S$. Thus, this will not contribute additional Big-O time.

Computing every value in the 3 data structure takes, in total, $O(n^2)$ time. Finally, to partition the words into the lines, we would need to access all values of $P$ at most once, which is a negligible $O(n)$ in additional computational time. From here, the computational time to write out the lines depend on the type of data structure. If we use $M$ linked-lists, one for each line of the partition, then we need only $O(n)$ time because we only have $n$ words. In total, the algorithm would run in $O(n^2)$ time. □

# Problem 3

**Algorithm 3** Algorithm to compute the set of maximal points of S

```
1: function MAXIMAL(set S = {p_1, ..., p_n})
2:     X = sort S by x coordinate
3:     S_x = ∅                                          ▷ S_x, S_y are stacks
4:     for i in 1 to n do
5:         PUSH(X[i], S_x)
6:     C_x = ∅                                          ▷ C_x is a set where each value appears only once
7:     x_a = POP(S_x), x_b = POP(S_x)
8:     while S_x ≠ ∅ do
9:         if y(x_a) > y(x_b) then                      ▷ y(·) is the y coordinate of the point
10:            x_b = POP(S_x)
11:        else
12:            ADD(x_a, C_x)
13:            x_a = POP(S_x)
14:    ADD(x_a, C_x)
15:    if y(x_a) ≤ y(x_b) then
16:        ADD(x_b, C_x)
17:
18:    Y = sort C_x by y coordinate
19:    S_y = ∅
20:    for i in 1 to n do
21:        PUSH(Y[i], S_y)
22:    C_y = ∅                                          ▷ C_y is a set where each value appears only once
23:    y_a = POP(S_y), y_b = POP(S_y)
24:    while S_y ≠ ∅ do
25:        if x(y_a) > x(y_b) then                      ▷ x(·) is the x coordinate of the point
26:            y_b = POP(S_y)
27:        else
28:            ADD(y_a, C_y)
29:            y_a = POP(S_y)
30:    ADD(y_a, C_y)
31:    if x(y_a) ≤ x(y_b) then
32:        ADD(y_b, C_y)
33:
34:    return C_y
```

*Proof of correctness.* The meaningful part of the algorithm begins on line 10. First, consider the points sorted by $x$ coordinates. In descending order, we check if one point dominates another point with smaller or the same $x$ coordinate. If so, the point with smaller or the same $x$ coordinate cannot be in the set of maximal points. If not, then both points may still be in the set of maximal points. Continuing in this manner until all points are exhausted will give all points where smaller $x$ coordinates corresponds to larger $y$ coordinates. We then consider remaining candidates $C_x$ sorted by $y$ coordinates. This takes into account the fact that some points may have either the same $x$ or $y$ coordinates. $C_y$ is a subset of the candidates $C_x$, which will give the set of maximal points. □

*Proof of runtime.* Sorting the set $S$ in $x$ and $y$ coordinates each takes $O(n \log n)$ time. Adding sorted values into a stack takes $O(n)$ time. The while-loop (for $x$ and $y$) makes comparisons whether a point dominates another point. There are at most $O(n)$ comparisons, and each comparison takes a constant amount of time. Thus, the 2 while-loops take $O(n)$ time. Thus, the algorithm overall takes $O(n \log n)$ time. □

# Problem 4

This problem can be solved with dynamic programming. Suppose the string is given as $S[1, \ldots, n]$. Define $M(i, j)$, where $i \leq j$, as the following:

$$M(i,j) = \begin{cases} 1 & \text{if substring } \ S[i] \ldots S[j] \ \text{ is a palindrome} \\ \infty & \text{otherwise} \end{cases}$$

To compute $M$, we use the following recursive formula:

$$M(i,j) = \begin{cases} 1 & \text{if } \ i = j \\ 1 & \text{if } \ i+1 = j, S[i] = S[j] \\ M(i+1, j-1) & \text{if } \ i+1 < j, S[i] = S[j] \\ \infty & \text{otherwise} \end{cases}$$

Define $P(i)$ as the minimum number of palindromes in the substring $S[1, \ldots, i]$. To compute $P$, we will need values of $M$. For simplicity, define $P(0) = 0$. The recursive formula for $i \geq 1$ is:

$$P(i) = \min_{1 \leq k \leq i} \{P(k-1) + M(k, i)\}$$

To find the answer to the problem, simply return $P(n)$.

For efficient computation, we can memoize values of $M$ and $P$ in an $n \times n$ matrices and an $n+1$-element array, respectively. To compute values of $M$, we first observe that the diagonal entries are 1 because each single letter is a palindrome on its own. For other values, we can start from the diagonal, then move our way along the corresponding off-diagonal by moving up and to the right (note that $M$ is only defined for its upper triangular region). To compute values of $P$, we take a bottom-up approach, calculating in the order of $P(1), P(2), \ldots, P(n)$.

*Proof of correctness.* The first part of the algorithm in finding the values of $M$ is straightforward. If $i = j$, then we have a substring of 1 letter, so it is indeed a palindrome. If $i + 1 = j$, i.e. we have a substring of 2 letters, and both letters are the same, then we have a palindrome. If the first and last letter of the substring are the same, then it's a palindrome only if the substring $S[i+1, \ldots, j-1]$ is as well. Since we defined the value of 1 as the substring being a palindrome and $\infty$ otherwise, the third base will give a correct answer. The last case takes care of all non-palindromic instances.

The second part of the algorithm in finding the values of $P$ is also clear. For a substring $S[1, \ldots, i]$, we are computing the minimum number of palindromes in the sub-substring $S[1, \ldots, k-1]$ plus 1 if the sub-substring $S[k, \ldots, i]$ is a palindrome or $\infty$ otherwise, and setting $P(i)$ as the minimum of these values for all possible $k$'s. Essentially, we consider all possible substrings for minimum number of palindromes, which yields the correct solution. □

*Proof of runtime.* Calculating each value of $M$ takes constant time. Since there are $O(n^2)$ entries of $M$ to calculate, this takes an overall time of $O(n^2)$. Calculating each value of $P$ requires comparing of $O(n)$ values because for each $i$, we need to look at all $P$ values with index less than $i$ as well as all $M(\cdot, i)$ values with row index less than $i$. Since there are $O(n)$ entries of $P$ to calculate, this takes an overall time of $O(n^2)$. Together, the total runtime is $O(n^2)$. □

# Problem 5

This problem can be solved with the Bellman-Ford algorithm, which was discussed in class. First, we need to modify the matrix $E$ by replacing $E(i,j)$ with $F(i,j) := -\log E(i,j)$ for all $i,j$, so the problem becomes finding a negative cycle. Consider the matrix $F$ as an adjacency matrix for a graph $G$. Since there are at most $|V|-1$ edges between any source node $v$ and any target node $t$ in $G$, we need to run Bellman-Ford algorithm for $k$ from 1 to $|V|-1$ starting at an arbitrary source vertex (all vertices are reachable from this arbitrary vertex since all values of $F(i,j)$ are defined). This will give us the shortest path from source $s$ to any vertex $v$ with up to $|V|-1$ edges. In addition to the algorithm given in class, we need to keep track of all the predecessor node of $v$ on the $k$th iteration, which we will call $p(v,k)$, as follows:

$$p(v,k) = u \text{ if } \pi(v,k) = \pi(u,k-1) + F(u,v)$$

This can be computed every time $\pi(v,k)$ is updated. To detect a negative cycle, we need to essentially run the algorithm one more time, i.e. for $k = |V|$. If for any $v$ we have $\pi(v,|V|) < \pi(v,|V|-1)$, then there is a negative cycle (i.e. there is an opportunity cycle). To identify this negative cycle, we simply check values of $p$. For example, we see that for vertex $w$, $\pi(w,|V|) < \pi(w,|V|-1)$, then we check $p(w,k)$, then $p(p(w,k),k-1)$, then $p(p(p(w,k),k-1),k-2)$,..., until we find $w$ again, or find any vertex in the sequence we have seen so far. The reverse of the sequence of vertices in this recursive search is the opportunity cycle.

*Proof of correctness.* We want to find a cycle through different currencies such that the product of exchange rates is greater than 1. Suppose we are given exchange rates $x_1, x_2, \ldots, x_n \in \mathbb{R}$. Then, we want:

$$x_1 x_2 \cdots x_n > 1$$

$$\log(x_1 x_2 \cdots x_n) > \log 1 = 0$$

$$\log(x_1) + \log(x_2) + \cdots + \log(x_n) > 0$$

$$-\log(x_1) - \log(x_2) - \cdots - \log(x_n) < 0$$

$$[-\log(x_1)] + [-\log(x_2)] + \cdots + [-\log(x_n)] < 0$$

which means equivalently, we want to the find a negative cycle through values $-\log(x_1), -\log(x_2), \ldots, -\log(x_n)$. Therefore, we transformed $E(i,j)$ to $F(i,j) := -\log Ei,j$.

Correctness of the Bellman-Ford algorithm was discussed in class. To see that recursively searching $p$ gives the sequence of currencies in the opportunity cycle, first recall that $p(i,|V|)$ was defined as the predecessor of currency $i$ on the $|V|$th iteration that gave the minimum distance from the source vertex. Thus, $p(p(i,|V|),|V|-1)$ was also the predecessor of currency $p(i,|V|)$ on the $|V|-1$th iteration that gave the minimum distance from the source vertex. Continuing in this manner, we are guaranteed to eventually return to currency $i$ or find a vertex that we have already encountered in this sequence. This is because number of edges between any vertices is at most $|V|-1$, so $|V|$ edges means we have a cycle. But we found the reverse of the sequence of opportunity cycle since we worked backwards in our recursive search in $p$. Therefore, we just return the reverse of the sequence, which is an opportunity cycle. □

*Proof of runtime.* From class, we know that Bellman-Ford runs in $O(|V||E|)$ time. To identify the opportunity cycle, we need to check values in $p$ at most $|V|$ times because the cycle can have at most $|V|$ elements. However, identifying the opportunity cycle does not contribute additional time to the overall algorithm, so the runtime remains $O(|V||E|)$. In terms of the problem, $|V| = n$ and $|E| = n^2$, so the algorithm runs in $O(n^3)$. □