Recitation 8 (HW7)

Online: Xinyi Zhao xz2833@nyu.edu

GCASL 461: Yifan Jin yj2063@nyu.edu

New York University

Basic Algorithms (CSCI-UA.0310-005)

Problem 1

Problem 1 (10+10+5 points)

Recall that we designed a dynamic programming approach to solve the problem of making change for n cents using the least number of coins (refer to Problem 9 of ADDITIONAL PROBLEMS set). Now we want to solve this problem using a greedy approach.

(a) Design a greedy algorithm to make change for n cents using the least number of coins among quarters (25), dimes (10), nickels (5), and pennies (1). Fully explain your algorithm.
For example, if $n = 91$, your algorithm must return 6, since we can make change for 91 cents using 6 coins: Take 3 quarters, 1 dime, 1 nickel, and 1 penny. Also, we can show that it is not possible to make change for 91 cents using less than 6 coins.

(b) Show that your algorithm in part (a) outputs a correct result for all positive integers n.

(c) Provide an example of a set of coin denominations for which your greedy approach in part (a) does not output a correct result.
Note that your set of coin denominations must include a penny so that for every positive integer n, it is possible to make change for n cents using your set of coin denominations.

Problem 1

Recall how to solve this problem using DP:

Problem 9. *We want to make change for n cents using the least number of coins among 1, 10, 25 cents. Develop an $O(n)$-time dynamic programming algorithm to find the least number of coins needed. Compute the total running time of your algorithm.*

For $i = 0, \ldots, n$, let $\textsc{LeastCoins}(i)$ denote the least number of coins required to make change for i cents. We have [Why?]

$$LeastCoins(i) = \begin{cases} 0 & \text{if } i = 0 \\ LeastCoins(i-1)+1 & \text{if } 1 \le i \le 9 \\ \min(LeastCoins(i-1)+1,\ LeastCoins(i-10)+1) & \text{if } 10 \le i \le 24 \\ \min(LeastCoins(i-1)+1,\ LeastCoins(i-10)+1,\ LeastCoins(i-25)+1) & \text{if } i \ge 25 \end{cases}$$

We have $n+1$ subproblems to solve (i.e., $\textsc{LeastCoins}(0), \ldots, \textsc{LeastCoins}(n)$), and each takes a constant time to be solved. So the total running time is $\Theta(n)$.

Exercise: Try to simplify the recursion above.

Exercise: Write the pseudo-code for the bottom-up DP approach.

Note: We will develop a simpler greedy algorithm in HW7.

Problem 1

(a) Design a greedy algorithm to make change for n cents using the least number of coins among quarters (25), dimes (10), nickels (5), and pennies (1). Fully explain your algorithm.

For example, if $n = 91$, your algorithm must return 6, since we can make change for 91 cents using 6 coins: Take 3 quarters, 1 dime, 1 nickel, and 1 penny. Also, we can show that it is not possible to make change for 91 cents using less than 6 coins.

Problem 1

Example:

$n = 91$

⇒ Step 1: try to use quarters(25) as many as we can

$91 / 25 = 3$, $91 \% 25 = 16$

so we use $\boxed{\text{3 quarters}}$, 16 left

Step 2: try to use dimes(10) as many as we can

$16 / 10 = 1$, $16 \% 10 = 6$

so we use $\boxed{\text{1 dime}}$, 6 left

Step 3: try to use nickels(5) as many as we can

$6 / 5 = 1$, $6 \% 5 = 1$

so we use $\boxed{\text{1 nickel}}$, 1 left

step 4: try to use pennies(1) as many as we can

$1 / 1 = 1$, $1 \% 1 = 0$

so we use $\boxed{\text{1 dime}}$, 0 left

Problem 1

Pseudo-code

```
LeastCoins(n):
    coins = [25, 10, 5, 1]
    count = 0
    for i = 1 to 4:
        count = count + n / coins[i]
        n = n % coins[i]
    return count
```

init call: LeastCoins(n)

TC = O(1)

SC = O(1)

Problem 1

(b) Show that your algorithm in part (a) outputs a correct result for all positive integers n.

Lemma: At each iteration, exchanging for the coin that has the largest value as many as we can gives the least number of coins.

Problem 1

Lemma: At each iteration, exchanging for the coin that has the largest value as many as we can gives the least number of coins.

Proof: Suppose $a+b+c+d$ where $25a+10b+5c+d=n$ is the least number of coins changed for n cents, and we did Not use nickels(5) as many as we can, in another word, $d > 5$.

Then we substitute 5 pennies with one nickel. Then the newly formed combination is: $a+b+c+(d-5)+1 = a+b+c+d-4 < a+b+c+d$. Therefore, $a+b+c+d$ is not the least number of coins changed for n cents, which contradicts the assumption. The similar proof goes to quarters and dimes. This finishes the proof.

Problem 1

(c) Provide an example of a set of coin denominations for which your greedy approach in part (a) does not output a correct result.

Note that your set of coin denominations must include a penny so that for every positive integer n, it is possible to make change for n cents using your set of coin denominations.

Problem 1

(c) Example: 15, 10, 1

n = 20

With algorithm developed in (a), count = 1(15)+5(1)=6;

But the output should be 2. (2*10 = 20)

Problem 1

A follow-up question:

What feature of coin combination makes the greedy strategy work?

Problem 1

A follow-up question:

What feature of coin combination makes the greedy strategy work?

Each coin is at least twice as the next smaller one.

Assume C[1]>C[2]>...>C[n],

then there should be C[i]>=2*C[i+1] for 1<=i<n

Problem 2

Problem 2 (10+10+5 points)

Recall the frog problem discussed in Homework 5 Problem 3:
Consider the array $A[1 \dots n]$ consisting of n non-negative integers. There is a frog on the last index of the array, i.e. the nth index of the array. In each step, if the frog is positioned on the i^{th} index, then it can make a jump of size at most $A[i]$ towards the beginning of the array. In other words, it can hop to any of the indices $i, \dots, i - A[i]$.

(a) Develop a GREEDY algorithm to determine whether the frog can reach the first index of A.

(b) Show the correctness of your algorithm in part (a).

(c) Find and justify the time complexity of your algorithm.

Problem 2

(a) Develop a GREEDY algorithm to determine whether the frog can reach the first index of A.

Recall the Dynamic Programming way:

At the index **i**, try all options **i-1, i-2, …, i-A[i]**. If any work, it work.

How does Greedy Strategy work?

Problem 2

(a) Develop a GREEDY algorithm to determine whether the frog can reach the first index of A.

Recall the Dynamic Programming way:

At the index **i**, try all options **i-1, i-2, …, i-A[i]**. If any work, it work.

How does Greedy Strategy work?

Intuitive thought:

Since the frog jumps from right to left, each time it jumps to the leftmost **(i-A[i])**.

Problem 2

(a) Develop a GREEDY algorithm to determine whether the frog can reach the first index of A.

‘Since the frog jumps from right to left, each time it jumps to the leftmost **(i-A[i])**’.

This is indeed a Greedy Strategy.

But it may fail.

Problem 2

(a) Develop a GREEDY algorithm to determine whether the frog can reach the first index of A.

‘Since the frog jumps from right to left, each time it jumps to the leftmost **(i-A[i])**’.

This is indeed a Greedy Strategy.

But it may fail.

Example: 1 0 2 2 (assume 1-indexed)

The frog will stuck at index 2 if it jumps to index 2 at the first step.

But it can reach the index 1 if it jumps to index 3 at the first step.

Problem 2

(a) Develop a GREEDY algorithm to determine whether the frog can reach the first index of A.

<u>‘Since the frog jumps from right to left, each time it jumps to the leftmost **(i-A[i])**’.</u>

Need a little optimization

Problem 2

(a) Develop a GREEDY algorithm to determine whether the frog can reach the first index of A.

<u>‘Since the frog jumps from right to left, each time it jumps to the leftmost **(i-A[i])**’.</u>

Need a little optimization

Still take the strategy (jump to the leftmost), but consider all indexes that could be reached.

If index **i** could be reach, all indexes larger than **i** could must be reached.

Problem 2

(a) Develop a GREEDY algorithm to determine whether the frog can reach the first index of A.

‘Since the frog jumps from right to left, each time it jumps to the leftmost **(i-A[i])**’.

Need a little optimization

Still take the strategy (jump to the leftmost), but consider all indexes that could be reached.

If index **i** could be reach, all indexes larger than **i** could must be reached.

Solution: Find the current leftmost index that could be reached and update.

Meanwhile try all indexes not less than the current leftmost index.

Problem 2

(a) Develop a GREEDY algorithm to determine whether the frog can reach the first index of A.

```
CanReach(A):
    n = size of A (1-indexed)
    leftmost = n          Current leftmost index that could be reached
    for i = n downto 1:
        if(i < leftmost)       Means index i cannot be reached,
            return false
                                          Update the current leftmost index
        leftmost = min(leftmost, i-A[i])  could be reached
        if(leftmost <= 1)      Means index 1 could be reached,
            return true
```

Problem 2

(b) Show the correctness of your algorithm in part (a).

Problem 2

(b) Show the correctness of your algorithm in part (a).

Loop invariant:

Variable *leftmost* is the current leftmost index the frog could reach.

And all indexes in [*leftmost*, n] are reachable.

Problem 2

(b) Show the correctness of your algorithm in part (a).

Initialization:

Before the iteration, *leftmost* = n is the only reachable (also the leftmost) index.

Problem 2

(b) Show the correctness of your algorithm in part (a).

Maintenance:

Assume it holds for **l=p**:

Variable *leftmost* is the current leftmost index the frog could reach.

And all indexes in [*leftmost*, n] are reachable.

Problem 2

(b) Show the correctness of your algorithm in part (a).

Maintenance:

Then for **l=p-1**:

Since all indexes in [*leftmost*, n] are reachable, if *leftmost* <= **p-1**, then index **p-1** is also reachable.

So index **(p-1)-1, …., (p-1) - A[p-1]** will also be reachable.

Problem 2

(b) Show the correctness of your algorithm in part (a).

Maintenance:

Then for **l=p-1**:

If *leftmost is smaller, then* [*leftmost*, n] are already reachable from the assumption.

If **(p-1)-A[p-1]** is smaller, then [**(p-1)-A[p-1], p-1**] and [*leftmost*, n] are reachable.
Since *leftmost* <= **p-1**, we can infer that [**(p-1)-A[p-1],** n] is reachable.

So the updated *leftmost* is the smaller one between *leftmost* and (p-1)-A[p-1].

Problem 2

(b) Show the correctness of your algorithm in part (a).

Maintenance:

Then for **l=p-1**:

Otherwise, *leftmost* > **p-1,** so index **p-1** is unreachable, then all indexes smaller **p-1** are also unreachable. So the frog cannot jump to the first index.

Problem 2

(b) Show the correctness of your algorithm in part (a).

Termination:

After the iteration terminates, if the leftmost <= 1, the first index is reachable.

Also, all index in [1, n] are reachable.

Problem 2

(c) Find and justify the time complexity of your algorithm.

theta(n)

Problem 3

Problem 3 (13+12 points)

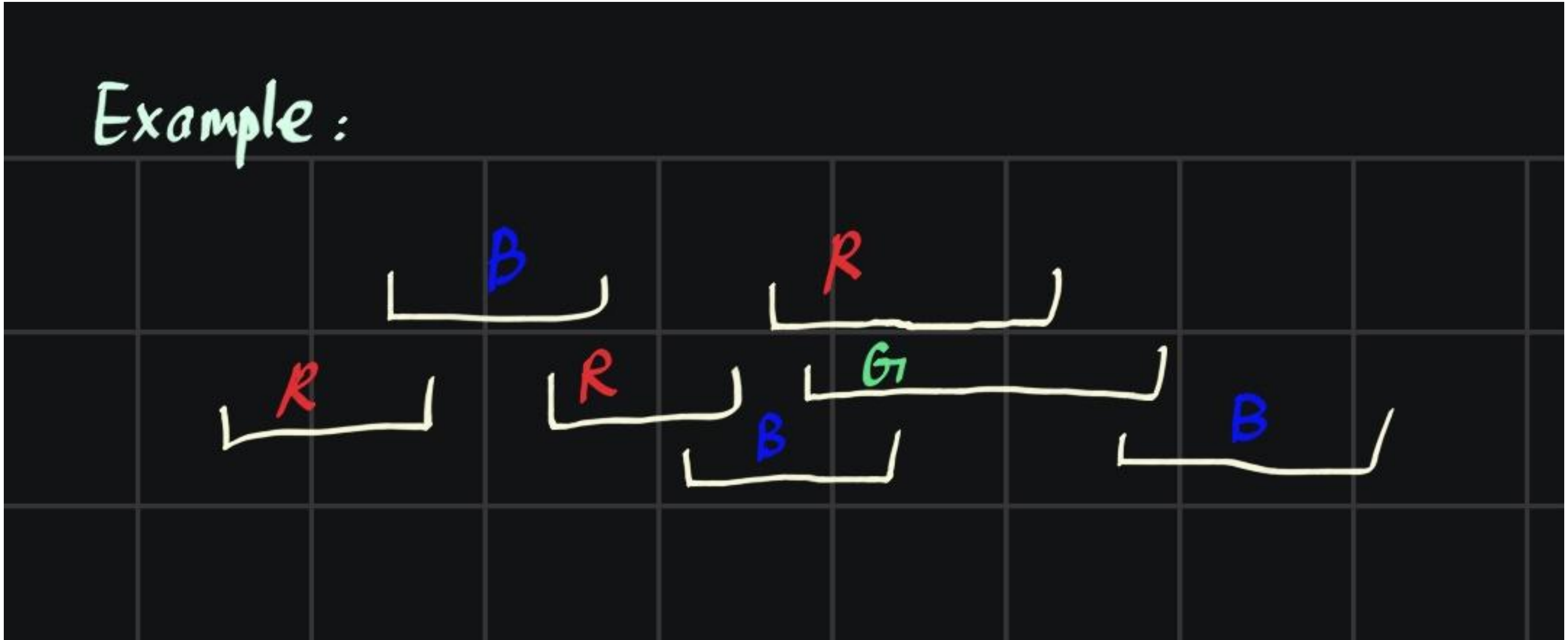
You are given a set $\mathcal{I}$ of n intervals on the real line. The starting and finishing times of these n intervals are given by the arrays $s[1 \dots n]$ and $f[1 \dots n]$, where $s[i]$ and $f[i]$ denote the starting time and the finishing time of the i^{th} interval in $\mathcal{I}$, respectively.
We want to color the intervals in $\mathcal{I}$ so that no two overlapping intervals are assigned the same color.

(a) Develop a GREEDY algorithm to compute the minimum number of colors needed to color $\mathcal{I}$ so that overlapping intervals are given different colors.

(b) Show the correctness of your algorithm in part (a) and find its running time.

Problem 3

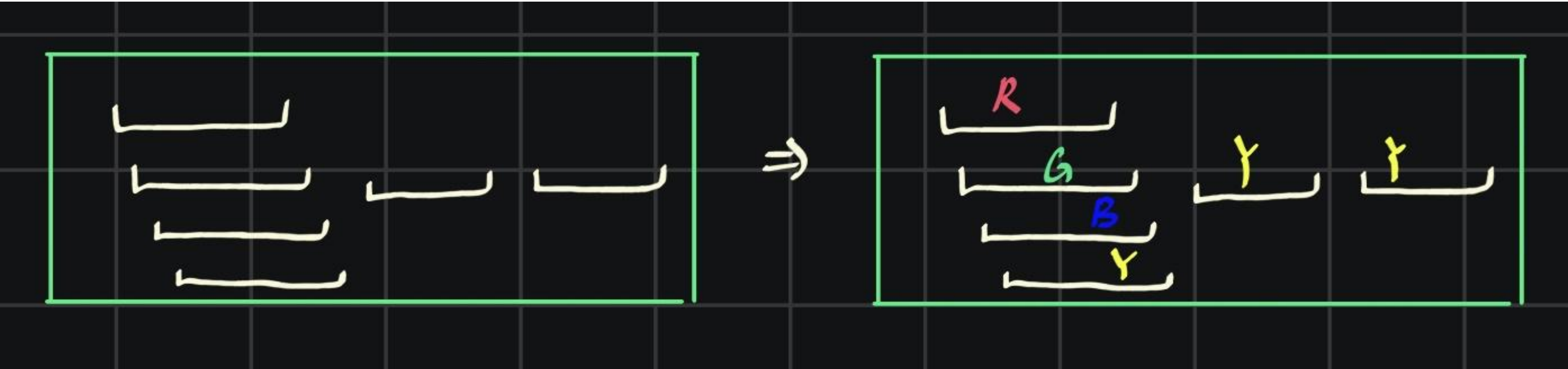
(a) Develop a Greedy algorithm to compute the minimum number of colors needed to color I so that overlapping intervals are given different colors.



Problem 3

How to deal with the intervals, overlaps?

What is the relationship between the colors and overlaps?

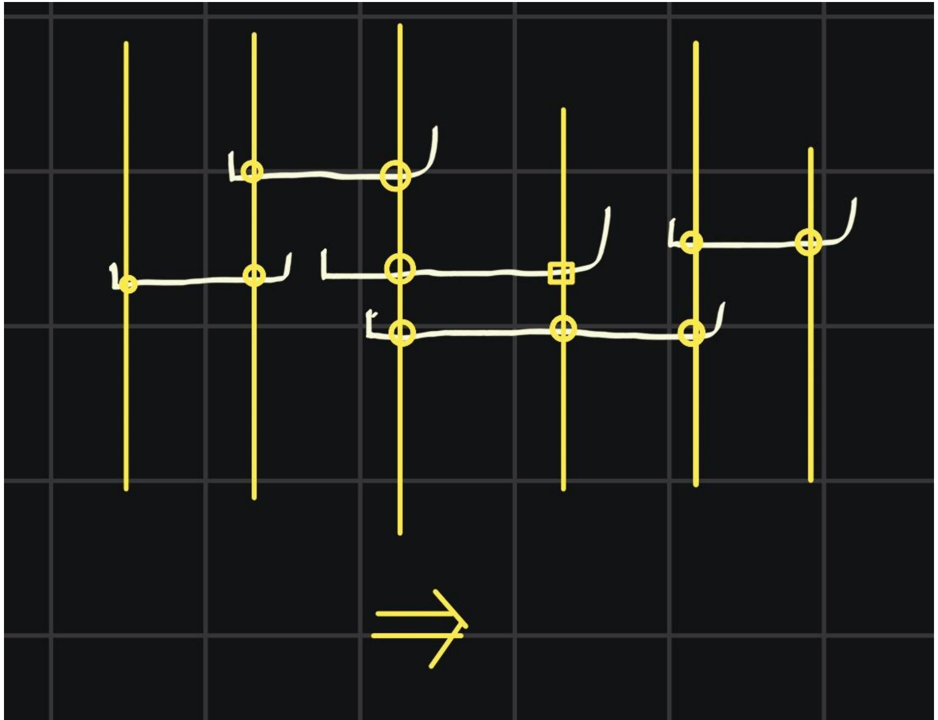


Problem 3

Imagine there is a sweep line, sweeping the intervals from left to right. And find the intersections between intervals with this sweep line.

Problem 3

The minimum number of colors needed = the maximum number of intersections



Problem 3

Then how to sweep??

How to move this line? How to implement this algorithm??

=> We only care about the start and end point for each interval.

Problem 3

Pseudo-code

```
MinColors( s[1...n], f[1...n] ):

    points[1...2n] = [(0, "")]
    for i = 1 to n:                                  (1)
        points[i] = (s[i], "s")
        points[i+n] = (f[i], "f")

    sort points[1...2n] increasingly, "f" goes       (2)
    to ahead if there is a tie.

    result = 0
    intersections = 0                                (3)
    for i = 1 to 2n :
        if (points[i][1] == "s"):
            intersections = intersections + 1
            result = max( result, intersections )
        else :
            intersections = intersections - 1

    return result
```

init call :

MinColor(s[1...n], f[1...n])

step (1) : make a new list of tuples, (num, "s" or "f")

step (2) : sort the list based on num, putting "f" elements ahead if there is a tie.

step (3) : iterate the list, simulate the sweep line.

case i : it's a "s" point

case ii : it's a "f" point

tes

Problem 3

(b) Show the correctness of your algorithm in part (a) and find its running time.

=> Use loop invariant to prove step (3)

Loop Invariant:

At the end of the for-loop iteration for index l, *intersections* represents the overlaps of intervals considering the first l points.

Problem 3

(Loop Invariant: At the end of the for-loop iteration for index l, *intersections* represents the overlaps of intervals considering the first l points.)

(I)Initialization: check that the loop invariant holds for l=1:

The first point is a “s” point. At the end of the for-loop iteration for index 1, *intersections* is equal to 1, representing only one interval (overlap) here.

Problem 3

(Loop Invariant: At the end of the for-loop iteration for index l, *intersections* represents the overlaps of intervals considering the first l points.)

(II)Maintenance:

Assumption: assume that the loop invariant holds for l = p: At the end of the for-loop iteration for index p, *intersections* represents the overlaps of intervals considering the first p points, say intersections = k.

Problem 3

(Loop Invariant: At the end of the for-loop iteration for index l, *intersections* represents the overlaps of intervals considering the first l points.)

(II)Maintenance:
Conclusion: prove that the loop invariant holds for l = p+1. If the (p+1)th point is a “s”, it means a new starting point of an interval presents, then one more overlap here, thus intersections = k+1;
If the (p+1)th point is a “f”, it means a finishing point of an interval presents, then overlap ends here, thus intersections = k-1. Hence proved.

Problem 3

(Loop Invariant: At the end of the for-loop iteration for index l, *intersections* represents the overlaps of intervals considering the first l points.)

(III)Termination: check that the loop invariant holds for l=2n:

The last point is a “f” point. At the end of the for-loop iteration for index l = 2n, there is no intersection point and no overlap. *Intersections* = 0.

Problem 3

Find its running time:

TC = O(nlogn + n)

= O(nlogn)

Pseudo-code

```
MinColors( s[1...n], f[1...n] ):

    points[1...2n] = [(0, "")]                               (1)
    for i = 1 to n:
        points[i] = (s[i], "s")
        points[i+n] = (f[i], "f")

    sort points[1...2n] increasingly, "f" goes               (2)
    to ahead if there is a tie.

    result = 0

    intersections = 0                                         (3)

    for i = 1 to 2n:
        if (points[i][1] == "s"):
            intersections = intersections + 1
            result = max(result, intersections)
        else:
            intersections = intersections - 1

    return result
```

$TC = O(n)$

$TC = O(2n\log 2n) = O(n\log n)$

$TC = O(2n) = O(n)$

notes

Problem 4

Problem 4 (10+15 points)

Let P be a set of n points in the plane. The points of P are given one point at a time. After receiving each point, we compute the convex hull of the points seen so far.

(a) As a naive approach, we could run Graham's scan once after receiving each point, with a total running time of $O(n^2 \log n)$. Write down the psuedo-code for this algorithm.

(b) Develop an $O(n^2)$-time algorithm to solve the problem. Write down the pseudo-code of your algorithm and justify that the run-time of your algorithm is $O(n^2)$.

Problem 4

(a) As a naive approach, we could run Graham's scan once after receiving each point, with a total running time of $O(n^2 \log n)$. Write down the psuedo-code for this algorithm.

Run Graham's Scan n times

Problem 4

(a) As a naive approach, we could run Graham's scan once after receiving each point, with a total running time of $O(n^2 \log n)$. Write down the psuedo-code for this algorithm.

For i = 1 to n:

Receive a new point

```
let points be the list of points
let stack = empty_stack()

find the lowest y-coordinate and leftmost point, called P0
sort points by polar angle with P0, if several points have the same polar angle then only keep the farthest

for point in points:
    # pop the last point from the stack if we turn clockwise to reach this point
    while count stack > 1 and ccw(next_to_top(stack), top(stack), point) <= 0:
        pop stack
    push point to stack
end
```

Also works if P0 is the leftmost one (and the bottom one if tied)

ccw > 0 if three points make a counter-clockwise turn, clockwise if ccw < 0, and collinear if ccw = 0.

Output stack

Again, determining whether three points constitute a "left turn" or a "right turn" does not require computing the actual angle between the two line segments, and can actually be achieved with simple arithmetic only. For three points $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$ and $P_3 = (x_3, y_3)$, compute the *z*-coordinate of the cross product of the two vectors $\overrightarrow{P_1P_2}$ and $\overrightarrow{P_1P_3}$, which is given by the expression $(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)$. If the result is 0, the points are collinear; if it is positive, the three points constitute a "left turn" or counter-clockwise orientation, otherwise a "right turn" or clockwise orientation (for counter-clockwise numbered points).

For i = 1 to n:

Receive a new point

```
let points be the list of points
let stack = empty_stack()

find the lowest y-coordinate and leftmost point, called P0
sort points by polar angle with P0, if several points have the same polar angle then only keep the farthest

for point in points:
    # pop the last point from the stack if we turn clockwise to reach this point
    while count stack > 1 and ccw(next_to_top(stack), top(stack), point) <= 0:
        pop stack
    push point to stack
end
```

Also works if P0 is the leftmost one (and the bottom one if tied)

ccw > 0 if three points make a counter-clockwise turn, clockwise if ccw < 0, and collinear if ccw = 0.

Output stack

Problem 4

(b) Develop an $O(n^2)$-time algorithm to solve the problem. Write down the pseudo-code of your algorithm and justify that the run-time of your algorithm is $O(n^2)$.

Problem 4

(b) Develop an $O(n^2)$-time algorithm to solve the problem. Write down the pseudo-code of your algorithm and justify that the run-time of your algorithm is $O(n^2)$.

Remark TC = O(n) if we can use counting sort for (2).

If all indexes are integers and are in limited range, counting sort will work.

Remaining part is same.

Space Complexity: O(range of X * range of Y)

Problem 4

(b) Develop an $O(n^2)$-time algorithm to solve the problem. Write down the pseudo-code of your algorithm and justify that the run-time of your algorithm is $O(n^2)$.

A solution to a more general case? (no restriction of index)

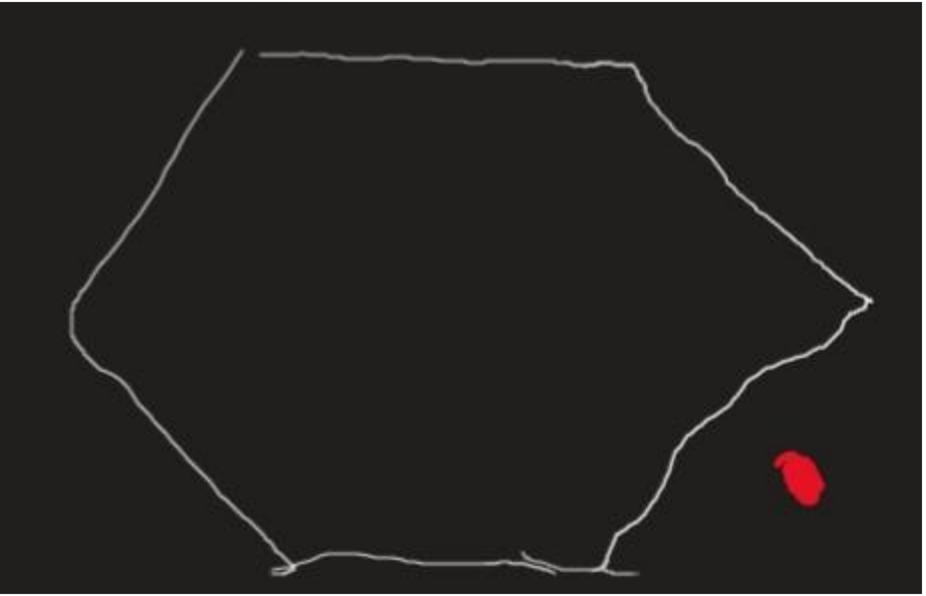
HARD!!!!!

Hint: Think in an inductive method

If we know the convex hull of **n** points, how to get that of **n+1** points?

Problem 4

(b) Develop an $O(n^2)$-time algorithm to solve the problem. Write down the pseudo-code of your algorithm and justify that the run-time of your algorithm is $O(n^2)$.



Problem 4

(b) Develop an $O(n^2)$-time algorithm to solve the problem. Write down the pseudo-code of your algorithm and justify that the run-time of your algorithm is $O(n^2)$.

Why do we delete that edge?

Problem 4

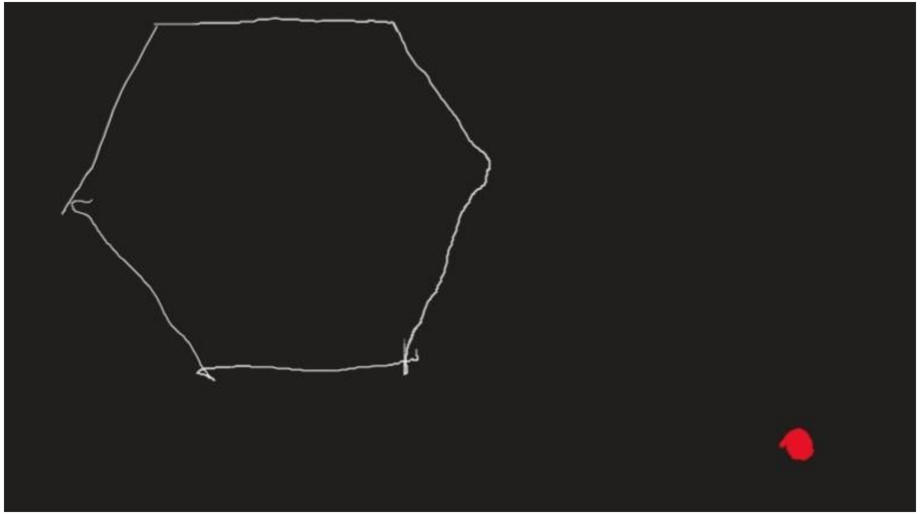
(b) Develop an $O(n^2)$-time algorithm to solve the problem. Write down the pseudo-code of your algorithm and justify that the run-time of your algorithm is $O(n^2)$.

Why do we delete that edge?

Since it does not satisfy the counter-clockwise direction with the new point.

Problem 4

(b) Develop an $O(n^2)$-time algorithm to solve the problem. Write down the pseudo-code of your algorithm and justify that the run-time of your algorithm is $O(n^2)$.



Problem 4

(b) Develop an $O(n^2)$-time algorithm to solve the problem. Write down the pseudo-code of your algorithm and justify that the run-time of your algorithm is $O(n^2)$.

Why do we delete those three edges?

Problem 4

(b) Develop an $O(n^2)$-time algorithm to solve the problem. Write down the pseudo-code of your algorithm and justify that the run-time of your algorithm is $O(n^2)$.

Why do we delete those three edges?

Since they don't satisfy the counter-clockwise direction with the new point.

Problem 4

(b) Develop an $O(n^2)$-time algorithm to solve the problem. Write down the pseudo-code of your algorithm and justify that the run-time of your algorithm is $O(n^2)$.

Also, we can infer that

edges that to be deleted must be consecutive in convex hull.

Problem 4

(b) Develop an $O(n^2)$-time algorithm to solve the problem. Write down the pseudo-code of your algorithm and justify that the run-time of your algorithm is $O(n^2)$.

Thus, the solution is:

For the convex hull of n points, delete all edges that don't satisfy the counter-clockwise direction with the new point.

Then add two edges (the red edges in the above graph).

Problem 4

(b) Develop an $O(n^2)$-time algorithm to solve the problem. Write down the pseudo-code of your algorithm and justify that the run-time of your algorithm is $O(n^2)$.

```
Convex_hull = [(P0,P1), (P1,P2), (P2,P0)]   // ordered

for i = 4 to n:
    Pi is the new point
    For edge in Convex_hull:
        if(edge is not counter-clockwise with Pi):
        delete edge from Convex_hull
                // The deleted edge must be consecutive
                // Assume the start of first deleted edge
                //      is Ps, the end of last deleted
                //      edge is Pe
    if(some edges were deleted)
        insert two edges (Ps, Pi),  (Pi, Pe) to Convex_hull in order
            //insert two edges into the position of deleted edges
```

Jarvis march algorithm

Reference: https://en.wikipedia.org/wiki/Gift_wrapping_algorithm

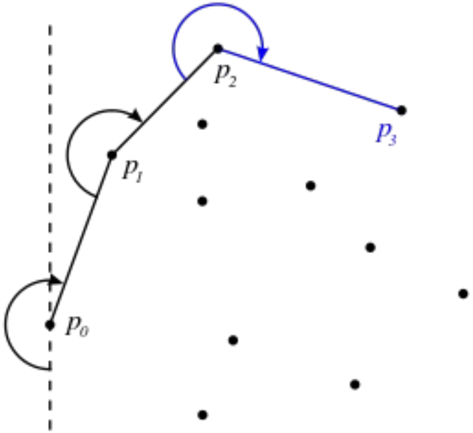
In the two-dimensional plane, **Jarvis march** is an algorithm for computing the convex hull of a given set of points.

It has O(nh) time complexity, where n is the number of points and h is the number of points on the convex hull.

Jarvis march algorithm

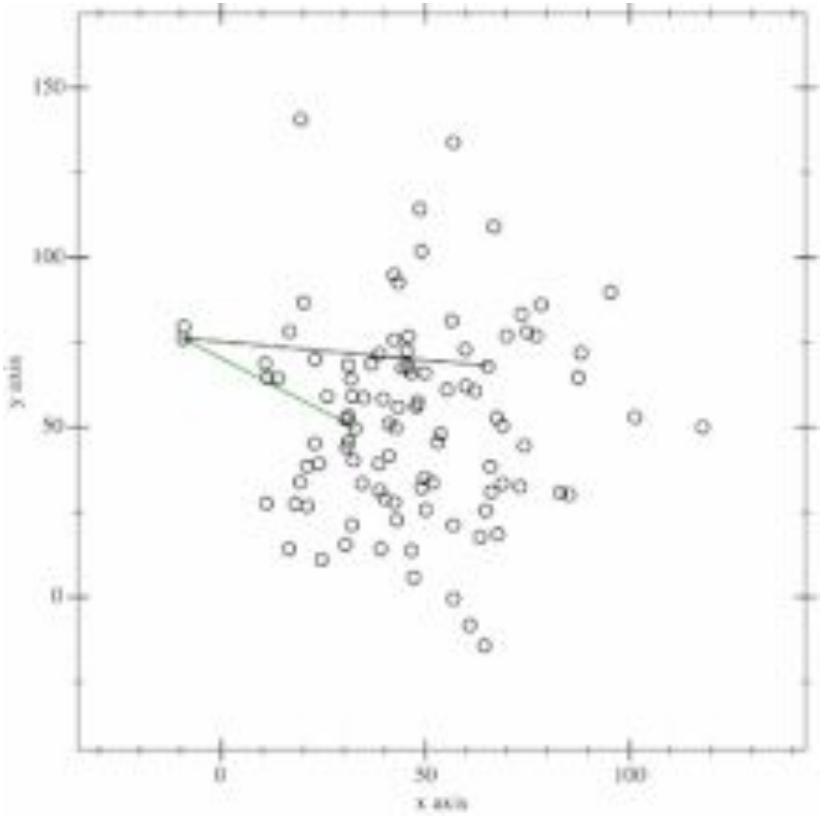
This algorithm begins with i=0 and a point p_0 known to be on the convex hull, e.g., the leftmost point, and selects the point p_{i+1} such that all points are to the right of the line p_i and p_{i+1}. This point may be found in O(n) time by comparing polar angles of all points with respect to point p_i taken for the center of polar coordinates.

Letting i=i+1, and repeating with until one reaches ph=p0 again yields the convex hull in h steps.



Jarvis march algorithm

Animation:



Jarvis march algorithm

Pseudocode [edit]

```
algorithm jarvis(S) is
    // S is the set of points
    // P will be the set of points which form the convex hull. Final set size is i.
    pointOnHull = leftmost point in S // which is guaranteed to be part of the CH(S)
    i := 0
    repeat
        P[i] := pointOnHull
        endpoint := S[0]      // initial endpoint for a candidate edge on the hull
        for j from 0 to |S| do
            // endpoint == pointOnHull is a rare case and can happen only when j == 1 and a better endpoint has not yet been set for
the loop
            if (endpoint == pointOnHull) or (S[j] is on left of line from P[i] to endpoint) then
                endpoint := S[j]   // found greater left turn, update endpoint
        i := i + 1
        pointOnHull = endpoint
    until endpoint = P[0]      // wrapped around to first hull point
```

Jarvis march algorithm

Complexity:

The inner loop checks every point in the set S, and the outer loop repeats for each point on the hull. Hence the total run time is O(nh).

The run time depends on the size of the output, so Jarvis's march is an output-sensitive algorithm.

However, because the running time depends linearly on the number of hull vertices, it is only faster than O(nlogn) algorithms such as Graham scan when the number h of hull vertices is smaller than log n.

Q & A

Thank you