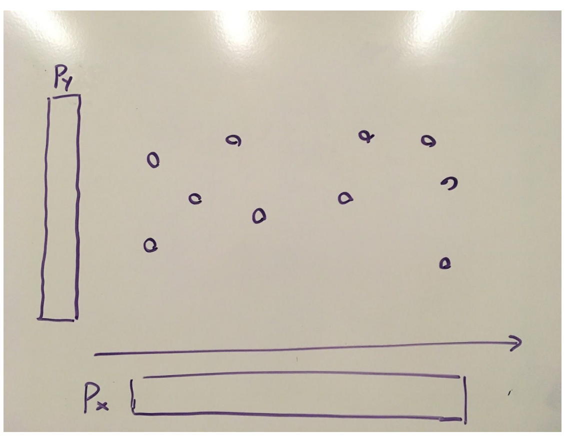
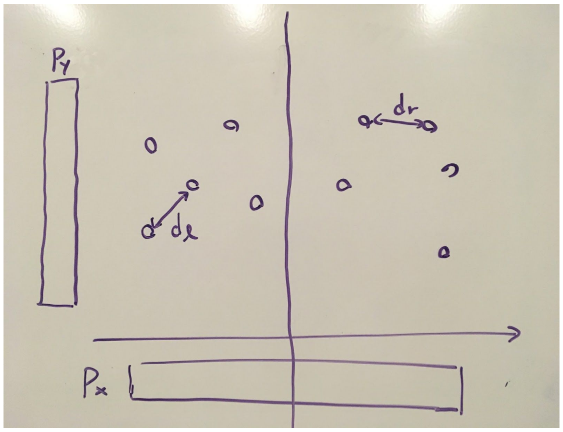
1. *Consider the divide and conquer algorithm for finding the closest pair of points. Analyze the time complexity of the algorithm . Include and discuss a detailed discussion of how to manage points in the x-dimension and how to manage (and search) points in the y-dimension. (You should do this without consulting the book or your notes)*

The divide and conquer algorithm for finding the closest pair of points is the following:

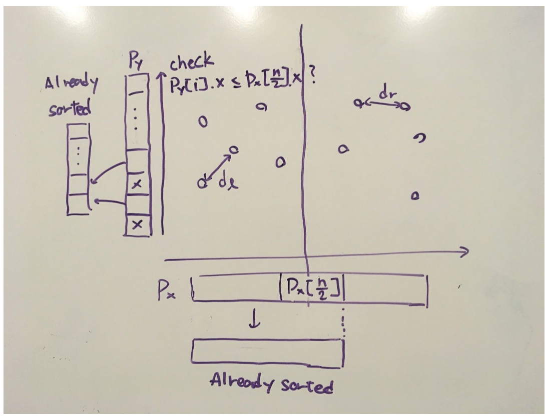
```
Let P be the points.
Let P_x be P sorted in x-axis
Let P_y be P sorted in y-axis

ClosestPair(P_x, P_y):
  If |P_x| is 2:
     Then return the distance between P_x[1] and P_x[2]
  (p_i, p_j) = ClosestPair(LeftHalf(P_x, P_y))
  (q_i, q_j) = ClosestPair(RightHalf(P_x, P_y))
  d = min(dist(p_i, p_j), dist(q_i, q_j))

  S_y = points in P_y whose x coord is within d of P_x[mid]
  For i = 1, ... , |S_y|
     For j = 1, ..., 6
        d = min(d, distance between S_y[i] and S_y[j])
        Keep track of pair of points (r_i, r_j) with distance d
  Return (r_i, r_j)
```



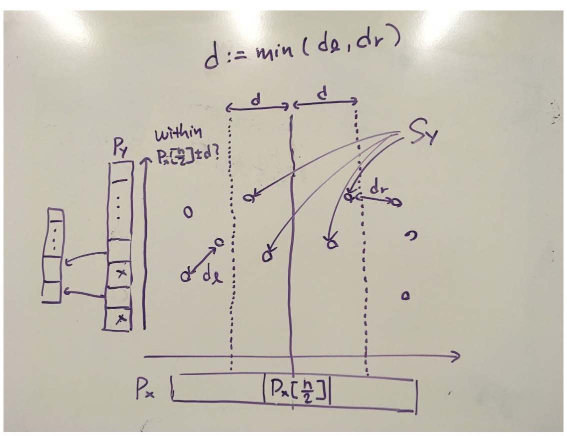
We want to show that the time complexity of this algorithm is $O(nlogn)$ where n is the number of points. Let P be the list of points given. First we sort P by x-axis and store it as P_x. Similarly, we sort P by y-axis and store it as P_y. Both operations take $O(nlogn)$



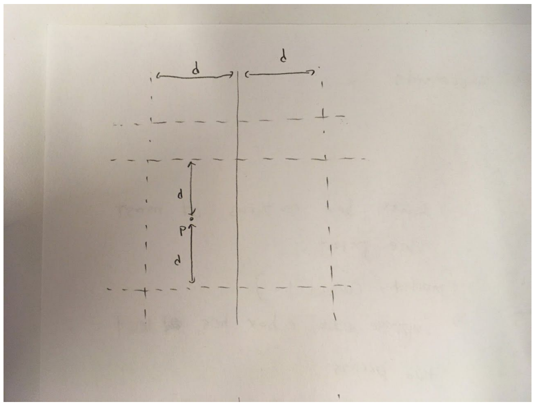
Then we divide the points into half by x-axis. Recursively find closest pair on the left half of the array (Let d_l be its distance). Recursively find closest pair on the right half of the array (Let d_r be its distance). Note during this recursion, we need to have the points sorted. How do we do this? Without loss of generality, we will focus on how to do this on left subarray. Sorting right half of the array follows the same idea.



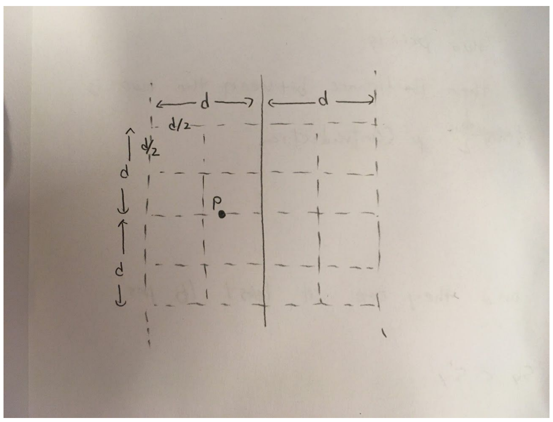
First, sorting the left subarray in x-axis is simple. Since the subarray is already sorted in x-axis, we can simply pass $P_x[0], \ldots, P_x[n/2]$ to the recursive function. Since the size of new P_x will be $n/2$, this operation takes $O(n)$. Note we want all the points whose x coord is smaller or equal to $P_x[n/2]$'s x coord. Thus, we can simply delete points from P_y whose x coord does not meet the requirement (i.e. greater than $P_x[n/2]$'s x coord). Since P_y was already sorted, the remaining array is also sorted, so we can pass this array to the recursive function as new P_y. Since size of P_y is n, this operations takes $O(n)$ time.



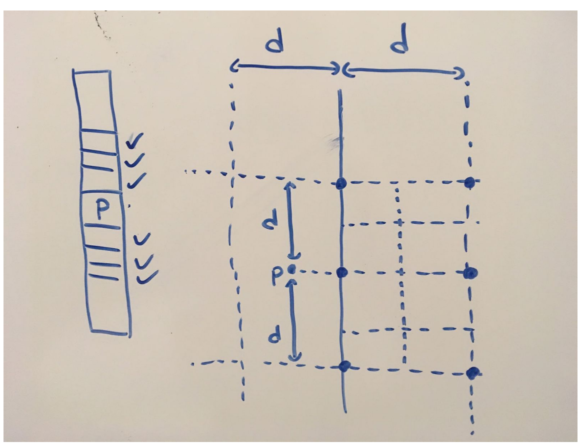
So now, the closest pair is either on the left half, right half, or it could be a pair such that one point is on the left and the other is on the right. To find such pair, we define d as $min(d_l, d_r)$. Then we only look at the region whose x coord is within d from $P_x[n/2]$. Note we don't need to worry about the points outside the dotted region because their distance will be greater than d anyways. So to get all the points in the dotted region (i.e. S_y), we can simply linear search P_y and delete points whose x coord is not within [$P_x[n/2] - d$, $P_x[n/2] + d$]. Since the size of P_y is n, this operations takes $O(n)$. Note S_y is sorted by y-axis since P_y was already sorted in y-axis.



Now let's consider a point p in S_y. We only need to worry about points whose y-coord are within $\pm d$ of p's y-coord because points outside of that region automatically have distances greater than d, which we can discard. Then we claim that there can be at most 15 points we need to check if its distance to p is smaller than d.



[Proof of Claim]
Suppose on the contrary, we can put at least two points in a box of size $d/2 \times d/2$. Then, the distance between two points in the box is smaller than d, which contradicts the fact that d is the min of closest pair on the left and closest pair on the right. Thus, there are at most 1 point in each box. In other words, there can't be too many points in the dotted region. Therefore, if $dist(p, p') < d$, then p' must exist within 15 points neighbors of p. (because there are 16 boxes)
(Q.E.D.)



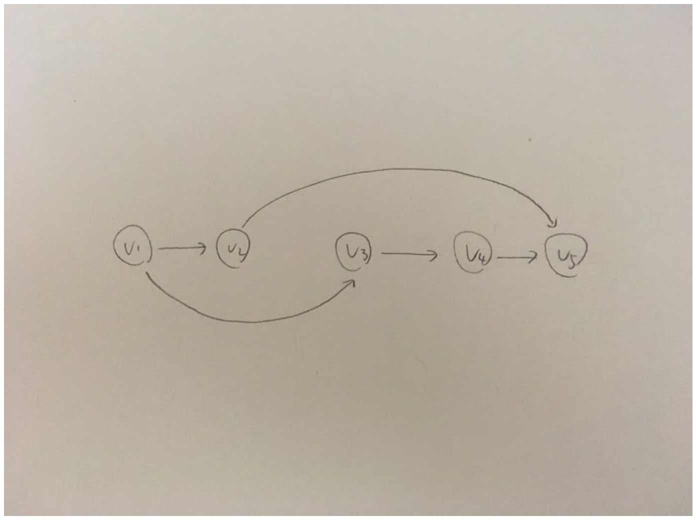
Thus for each point in S_y, we need to compare at most 15 points to determine the closest pair such that one point is on the left and the other is on the right. Since S_y is already sorted in y-axis, given point p, we can easily find such 15 points in linear time, and thus finding the closest point to p in linear time as well (i.e. O(1)). Since there are at most n points in S_y, it takes $O(n)$ to find the closest pair in S_y (i.e. Merging takes $O(n)$). Suppose the time complexity of this algorithm is $T(n)$. Then
$T(n) = T(n/2) + T(n/2) + cO(n)$. Thus
$T(n) = O(nlogn)$. (Q.E.D.)

[Proof of correctness]
We first divide the points in half by x-axis. Then recursively find the closest pair on the left and closest pair on the right. We also find a closest pair such that one point is on the left half and the other point is on the right half. Then the pair with minimum distance out of these 3 pairs is the pair we are looking for.

[Time complexity analysis]
Already analyzed above. $O(nlogn)$.

2. *Exercise 3 on page 314*

(a)



In the example above, there are edges $(v_1,\ v_2),\ (v_1,\ v_3),\ (v_2,\ v_5),\ (v_3,\ v_4), (v_4,\ v_5)$.
The proposed algorithm will do the following: First it visits v_1. There are two edges from v_1, which are $(v_1,\ v_2),\ (v_1,\ v_3)$. Since $2 < 3$, it will visit v_2. Then it will go to v_5. Thus it returns 2 as the length of the longest path. However, the actual longest path has length 3, which can be constructed by visiting $v_1 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5$.

(b)
The algorithm is the following.

Suppose we have n nodes.
Let dp be array of size n ($dp[1],\ ...,\ dp[n]$)
Here $dp[i]$ represents the length of the longest path from v_1 to v_i
Initialize dp such that $dp[1] = 0$ and $dp[i] = -\infty$ for all i greater than 1
For $i = 1, ..., n$
    For each edge $(v_i,\ v_j)$
        If $dp[i]$ is not $-\infty$:
            $dp[j] = max(dp[j],\ 1 + dp[i])$
If $dp[n] = -\infty$:
    There is no path from v_i to v_n
Else:
    Return $dp[n]$ as the length of the longest path from v_i to v_n

[Proof of correctness]
Our algorithm is correct because it performs exhaustive search.
First notice by the property of the problem, the graph is an ordered graph.
v_j does not contribute to the longest path from v_1 to v_i for all $j \geq i$.
Thus, we have the recurrence such that, to compute the longest path from v_1 to v_n,

we take the maximum of $1 + longest\ path\ from\ v_1\ to\ v_k$ for all $1 \le k \le n-1$.

[Time complexity analysis]
There are two nested loops in our algorithm. During each iteration, we perform line 7 and 8, which takes only constant time. Line 5 takes $O(n)$. Line 6 takes $O(n)$ because in the worst case, a node is connected to every other node. Thus, the time complexity of this algorithm is $O(n^2)$. But if we let e denote the number of edges, then line 6 takes $O(e)$ throughout the algorithm. Thus, the time complexity of this algorithm can also be written as $O(n+e)$.

3. *Exercise 5 on page 316*

Suppose a string $y = y_1 y_2 ... y_n$ is given where y_i is a character.
Let $dp[i]$ denotes the maximum total quality of segmentation of $y_1 y_2 ... y_i$.
Then we have the recurrence $dp[i] = max(dp[i-k] + quality(y_k ... y_i))$ $(1 \le k \le i)$.

Thus, the algorithm is the following.

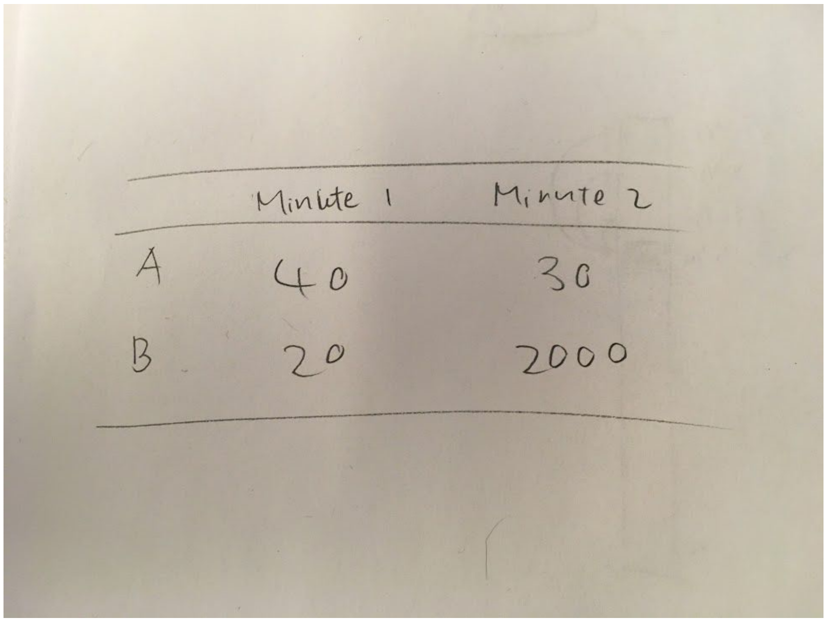
1 Let dp be an array of size $n+1$
2 Initialize dp such that $dp[0] = 0$ and $dp[i] = -\infty$ for any i greater than 0.
3 For i = 1 to n:
4   For $k = 1$ to i:
5         $dp[i] = max(dp[i],\ dp[i-k] + quality(y_k ... y_i))$
6 Return $dp[n]$

[Proof of correctness]
Our algorithm is correct because it performs exhaustive search.
We can prove this by induction.
(Base Case)
When there is no letters, then the maximum total quality is 0
Indeed, in line 2, we set $dp[0] = 0$
(Inductive Case)
Suppose we know the maximum total quality of $dp[i]$ $(1 \le i \le n-1)$
We want to show we can compute $dp[n]$ using $dp[i]$ $(1 \le i \le n-1)$
Notice we have the recurrence $dp[i] = max(dp[i],\ dp[i-k] + quality(y_k ... y_i))$
because we check all the possibilities of how y_i can be segmented.
Either y_i is one segment or $y_{i-1} y_i$ is a segment or $y_{i-2} y_{i-1} y_i$ is a segment and so on.
Thus, we can compute $dp[n]$ using $dp[i]$ $(1 \le i \le n-1)$ by taking the maximum of $dp[i-k] + quality(y_k ... y_i)$.

[Time complexity analysis]
We have two nested loops. Line 3 (outer loop) takes $O(n)$ and Line 4 (inner loop) takes $O(n)$ because i is n in the worst case. Then during each iteration we do line 5, which takes $O(1)$. So in total, it takes $O(n^2)$.

4. *Exercise 10 on page 321*

(a)



The proposed solution will first choose A for both minutes. However, the optimal solution will choose B for both minutes.

(b)

Let a_i denotes the value of machine A at minute i

Let b_i denotes the value of machine B at minute i

Let $dp(i,\ A)$ denotes the maximum value from minute 1 to i that ends on machine A.

Let $dp(i,\ B)$ denotes the maximum value from minute 1 to i that ends on machine B.

Then we have the recurrence

$$dp(i,\ A) = a_i + max(dp(i-1,\ A),\ dp(i-2,\ B))$$

$$dp(i,\ B) = b_i + max(dp(i-1,\ B),\ dp(i-2,\ A))$$

Then $max(dp(n,\ A),\ dp(n,\ B))$ is the optimal value

The algorithm is the following:

Let dp_A be an array of size $n+1$
Let dp_B be an array of size $n+1$
$dp_A[0] = dp_B[0] = 0$
$dp_A[1] = a_1$
$dp_B[1] = b_1$
For $i = 2\ to\ n$:
    $dp_A[i] = a_i + max(dp_A[i-1],\ dp_B[i-2])$
    $dp_B[i] = b_i + max(dp_B[i-1],\ dp_A[i-2])$
Return $max(dp_A[n],\ dp_B[n])$

[Proof of correctness]
Our algorithm works because it performs exhaustive search.
Thus it suffices to show why we have the recurrence above.
(Proof of $dp(i,\ A) = a_i + max(dp(i-1,\ A),\ dp(i-2,\ B))$)
When we reach machine A at time i, then we either came straight from machine A at time $i-1$, or we were at machine B at time $i-2$, and moving from machine B to machine A at time $i-1$.
Thus we can take the maximum of the two possibilities.
(Proof of $dp(i,\ B) = b_i + max(dp(i-1,\ B),\ dp(i-2,\ A))$)
Same as proof above with A and B flipped.

[Time complexity analysis]
Line 6 takes $O(n)$ time. During each iteration, we execute line 7 and 8, both of which are done in $O(1)$. Thus, the total is $O(n)$.

5. *Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces. For example, if length of the rod is 8 and the values of different pieces are given as follows, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6)*
length=[1,2,3,4,5,6,7,8]
price=[1,5,8,9,10,17,17,20]

Let $price[i]$ be the price of rod of length i
Let $dp[i]$ be the maximum value obtainable for rod of length i.
Then we have the recurrence $dp[i] = max(dp[i-k] + price[k])\ (1 \le k \le i-1)$

The algorithm is the following:

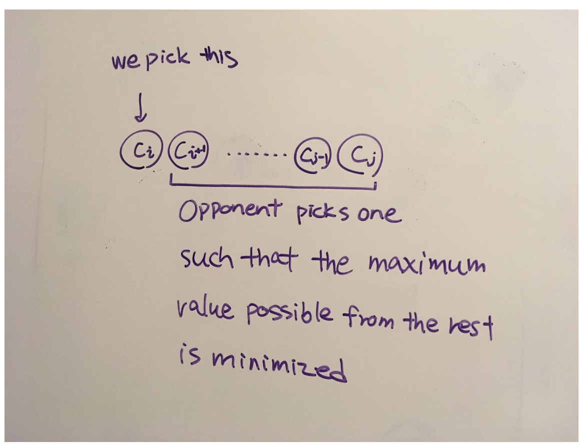
1 Let dp be an array of size n 2 Initialize dp such that $dp[1] = price[1]$ and $dp[i] = -\infty$ for all i greater than 1 3 For $i = 2\ to\ n$: 4   For $k = 1\ to\ i$: 5 $dp[i] = max(dp[i],\ dp[i-k] + price[k])$ 6 Return $dp[n]$

[Proof of correctness]
Our algorithm works because it performs exhaustive search using dynamic programming.
It is suffice to show why the recurrence $dp[i] = max(dp[i-k] + price[k])\ (1 \le k \le i-1)$ is correct.
(Base Case)
When we have a rod with length 1, clearly the maximum value we can get is $price[1]$.
(Inductive Case)
Suppose we know the maximum value for $dp[1],\ ...,\ dp[i-1]$. We want to show $dp[i] = max(dp[i-k] + price[k])\ (1 \le k \le i-1)$. Let's consider the optimal solution, and focus on the last piece of the rod after we cut. Then, the size of the last piece is either 1, 2, 3, …, or i. Thus, we just need to choose the maximum of these cases (i.e. $max(dp[i-k] + price[k])\ (1 \le k \le i-1)$).

[Time complexity analysis]
We have two nested for loop. Line 3 takes $O(n)$. Line 4 takes $O(n)$ because i is n in the worst case. During each iteration, line 5 takes $O(1)$. Thus, in total, this algorithm takes $O(n^2)$ time.

6. *Consider a row of n coins of values v1...vn, where n is even. We play a game against an opponent by alternating turns (you can both see all coins at all times). In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can win if we move first.*
Example 1: [5, 3, 7, 10] : The user collects maximum value of 15 (10 + 5) - Sometimes the greedy strategy works.
Example 2: [8, 15, 3, 7] : The user collects maximum value of 22 (7 + 15) – In general the greedy strategy does not work.

Suppose opponent is as smart as us
(i.e. They use the same algorithm we use to get maximum possible amount he/she can get)
Let c_i denotes i-th value of the coin
Let $dp(i, j)$ denotes the maximum possible amount of money we can win if we move first given coins from i-th coin to j-th coin. Since the opponent wants us to have minimum value, we have the recurrence
$dp(i, j) = max(c_i + min(dp(i+2, j),\ dp(i+1, j-1)),\ c_j + min(dp(i+1, j-1),\ dp(i, j-2)))$ by the picture below.



The algorithm is the following:

Let dp be a $n \times n$ table
Let c be the array of values of coins
For $i = n, \ldots, 1$:
    For $j = 1, \ldots, n$:
        $x = y = z = 0$
        $x = dp[i+2][j]$ if $i + 2 \le j$
        $y = dp[i+1][j-1]$ if $i + 1 \le j - 1$
        $z = dp[i][j-2]$ if $i \le j - 2$

9 $\quad dp[i][j] = max(c[i] + min(x,\ y),\ c[j] + min(y,\ z))$
10 Return $dp[0][n]$

[Proof of correctness]
Our algorithm works because it performs exhaustive search using dynamic programming. Thus, it suffices to show the recurrence
$dp(i,\ j) = max(c_i + min(dp(i+2,\ j),\ dp(i+1,\ j-1)),\ c_j + min(dp(i+1,\ j-1),\ dp(i,\ j-2)))$ is correct.
Given $c_i, ...,\ c_j$, we have two options. Either pick c_i or c_j.
If we pick c_i, then the opponent must pick a coin from $c_{i+1}, ...,\ c_j$.
The opponent will pick either c_{i+1} or c_j.
If the opponent pick c_{i+1}, we are left with $c_{i+2}, ...,\ c_j$
If the opponent pick c_j, we are left with $c_{i+1}, ..., c_{j-1}$
The opponent will pick one such that the maximum value we can get from leftover coins is minimized
Thus we will have $min(dp(i+2,\ j),\ dp(i+1,\ j-1)$.
Therefore, if we pick c_i, then we will end up with $c_i + min(dp(i+2,\ j),\ dp(i+1,\ j-1))$
Similarly, if we pick c_j, then we will end up with $c_j + min(dp(i+1,\ j-1),\ dp(i,\ j-2))$
Thus the maximum we can get is the maximum of these two values
(i.e. $dp(i,\ j) = max(c_i + min(dp(i+2,\ j),\ dp(i+1,\ j-1)),\ c_j + min(dp(i+1,\ j-1),\ dp(i,\ j-2)))$)

[Time complexity analysis]
We have two nested for loop. Line 3 takes $O(n)$, and line 4 takes $O(n)$ as well. During each iteration, we execute line 5 to 9, which can be done in $O(1)$. Thus, in total, this algorithm runs in $O(n^2)$.