Statement of Integrity: I, Derek Zhu, state that I completed this assignment with integrity and by myself.

Homework 4 is here, Due: Jul 22, 2024, to be submitted at here

Problem 1:

If there is only **one** well, then the optimal position of the main pipe line is that well, because the distance from the well to the pipe line is zero, which is the shortest length of the spurs.

If there are only **two** wells, then the optimal position will be any point between the two wells, including the location of any one of the two wells, because the total distance from the two wells to the pipeline is the same, which is the shortest length of the spurs.

If there are only **three** wells, then the optimal position will be at the median well of the three, because the total distance from the other two wells to the pipeline is minimized, which results in the shortest possible length for the spurs.

In **general**, the optimal position will be at the median well of all the wells. There will be one median well if the total number of wells is odd, or two median wells if the total number of wells is even. When there are two medians, choosing either one will not change the total length of the spurs.

The problem becomes how to find the median well in linear time. In other words, we need to find the well which has the n/2 th smallest (in terms of y-coordinates) well of all n wells, or the median well.

The CLRS textbook chapter 9 Medians and Order Statistics provided an algorithm Select(A, p, r, i) at page 237 (9.3 Selection in worst-case linear time). This algorithm returns the i-th order statistic element of n elements in a subarray A[p:r] in worst-case linear time: $\Theta(n)$.

So, the professor just needs to call: **Select(A, 1, n, n/2)** if the array start at index=1, or:
Select(A, 0, n-1, n/2) if the array starts at index=0. To compare A[i] with A[j], **only the y-coordinates need to be compared .**

After finding the median well, pleace the main pipeline at this y-coordinate. This solution efficiently finds the optimal location in linear time $\Theta(n)$ as proved in CLRS chapter 9.3.

Problem 2:

To solve this problem, we'll create an algorithm **calculateDegrees (adj_list)** that calculates the in-degree and out-degree of each vertex in a directed graph represented using adjacency lists "adj_list" . The algorithm will have a running time of $\Theta(m + n)$, where m is the number of edges and the n is the number of vertices.

Per the text book at page 550, the adjacency-list representation of a Graph $G = (V, E)$ consists of an array Adj of |v| list, one for each vertex in V. For each $v \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$. That is, $Adj[u]$ consists of all the vertices adjacency to u in G.

Here is my algorithm:

calculateDegrees (adj_list)

```
//      initialize the in_degree and the out_degree:
1       for each v in adj_list:
2               in_degree[v] =0
3               out_degree[v]=0

//      populate the out_degree:
4       for each v in adj_list:
5               out_degree[v] = length of ad_list[v]

//              populate the int_degree:
6               for each u in ad_list[v]:
7                       in_degree[u] += 1
8       return in_degree, out_degree
```

Runtime analysis:

Line#1~3: 2*n
Line#4~7: 2*(n + m), because we visit each of the vertices and each of the edges twice.
Line#8~9: 1

Total runtime: $T(n, m) = 4n + 2m + 1 = 2(n + m) + 2n + 1 = 4(n + m) - 2n + 1$
Since $2(n + m) \leq T(n, m) \leq 4(n + m)$,
Therefore: $T(n, m) = O(n + m)$, and: $T(n, m) = \Omega(n + m)$
Hence: $T(n, m) = \Theta(n + m)$

Problem 3:

(a) To bring the skip-ahead brute force algorithm closer to its expected n^2 bound, we need an input where the distance check optimization (using the x-coordinate difference) is ineffective. This happens when points are densely packed within a small region in the x-direction but spread out in the y-direction. **Example input:** Imagine a set of points where all x-coordinates are very close to each other but y-coordinates vary significantly. For example, points like (0,1), (0.01,2), (0.02,3), … , (0.99, 100), (0, 1), (0.01, 2), (0.02, 3). **General properties:** Points have very close x-coordinates (so that the difference in x-coordinates between any two points is always smaller than the minimum distance found so far). Points have significantly varying y-coordinates. In this scenario, the skip-ahead optimization won't be effective because the x-difference condition rarely holds true, leading the algorithm to continue comparing almost every pair of points, making it closer to the n^2 bound.

(b) The CLRS divide-and-conquer approach involves three steps: divide, conquer, and combine. Let's focus on the combined step, which is crucial for understanding why the algorithm performs better even with the problematic input described in part (a). In the combined step, the algorithm forms a vertical strip (or rectangle) around the dividing line with a width of 2δ , where δ is the minimum distance found in the two halves. Within this strip, it only needs to check pairs of points that are within δ distance vertically. The key insight from CLRS is that each point in the strip needs to be compared only to the next 7 points (as proved by geometric arguments and properties of the plane). Even when the x-coordinates of points are very close to each other (thus making the skip-ahead optimization ineffective), the CLRS algorithm maintains its efficiency because the recursive nature ensures that each half contains fewer points, maintaining the $O(nlogn)$ complexity. During the combine step, each point in the strip is compared to at most 7 other points, not all pairs. This significantly reduces the number of comparisons. Thus, the bad input scenario described in part (a) does not negatively impact the CLRS divide-and-conquer algorithm because of its strategic use of recursive halving and the efficient combination step involving a limited number of comparisons within the defined strip.

(c) Reason 1: when the number of points is equal to or less than 3, use brute-force as the base case to stop the recursive call in the divide-and-conquer algorithm.

Reason 2: another reason to use the skip-ahead brute force algorithm instead of the CLRS divide-and-conquer algorithm is **simplicity of implementation**. The skip-ahead brute force method, while potentially slower in the worst-case scenario, is generally simpler to code and understand: It does not require recursive function calls. There is no need for complex logic to handle dividing and merging points. The algorithm's logic is straightforward, making it easier to debug and maintain.

Problem 4:

(a) MultiPopA(k) pops k elements from stack A if k is equal to or less than n, or pops n elements from stack A if k is greater than n. In another word, it pops $min\ (k,\ n)$ elements from stack A. Since each single pop can be performed in $O(1)$ time worst-case as given, hence the worst-case running time complexity for MultiPopA(k) is $min(k, n)\ *\ O(1)\ =\ O(min(k, n))$.

Same reason, the worst-case running time complexity for MultiPopB(k) is $O(min(k, m))$.

In the worst-case, the worst-case running for Transfer(k) is the time for MultiPopA(k) plus the time for PushB(x): $O(min(k, n)) + min(k, n)\ *\ O(1)\ =\ O(min(k, n))$

(b) To prove that each operation has an amortized running time of $O(1)$ using the given potential function $\Phi(n, m)\ =\ 3n\ +\ m$, we need to analyze the amortized cost of each operation.

The amortized cost A_i of an operation is given by:

$A_i = C_i + \Phi(D_{i+1}) - \Phi(D_i) = C_i + \Delta\Phi_i$

$\Delta\Phi_i = \Phi(D_{i+1}) - \Phi(D_i)$

Where C_i is the actual cost of the i-th operation, $\Phi(D_i)$ is the potential before the i-th operation, $\Phi(D_{i+1})$ is the potential after the i-th operation, $\Delta\Phi_i$ is the potential change between before and after the i-th operation.

$PushA(x)$:

- Actual cost: $C_i = O(1)\ as\ given$
- Potential change (note: the size of stack A increase by one for each operation): $\Delta\Phi_i = \{3(n + 1) + m\} - \{3n + m\} = 3$
- Amortized cost: $A_i = C_i + \Delta\Phi_i = O(1) + 3 = O(1)$

$PushB(x)$:

- Actual cost: $C_i = O(1)\ as\ given$
- Potential change(note: the size of stack B increase by one for each operation): $\Delta\Phi_i = \{3n + (m + 1)\} - \{3n + m\} = 1$
- Amortized cost: $A_i = C_i + \Delta\Phi_i = O(1) + 1 = O(1)$

$MultipopA(k)$:

- Actual cost: $C_i = O(min(k, n))\ as\ given\ in\ (a)\ above$
- Potential change(note: the size of stack A decrease by $min(k, n)$ for each operation): $\Delta\Phi_i = \{3(n - min(k, n)) + m\} - \{3n + m\} = -3 * min(k, n)$
- Amortized cost:
  $A_i = C_i + \Delta\Phi_i = O(min(k, n)) - 3 * min(k, n)$
  $= O(min(k, n)) - O(min(k, n)) = O(1)$

$MultipopB(k)$:

- Actual cost: $C_i = O(min(k,m))\ as\ given\ in\ (a)\ above$
- Potential change(note: the size of stack A decrease by $min(k,m)$ for each operation):
  $\Delta\Phi_i = \{3n + m - min(k,n)\} - \{3n + m\} = -min(k,n)$
- Amortized cost:
  $A_i = C_i + \Delta\Phi_i = O(min(k,n)) - min(k,n)$
  $= O(min(k,n)) - O(min(k,n)) = O(1)$

$Transfer(k)$:

- Actual cost: $C_i = O(min(k,n))\ as\ given\ in\ (a)\ above$
- Potential change(note: the size of stack A decrease by $min(k,n)$ for each operation, the size of stack B increased by $min(k,n)$ for each operation):
  $\Delta\Phi_i = \{3(n - min(k,n)) + (m + min(k,n))\} - \{3n + m\} = -2 * min(k,n)$
- Amortized cost:
  $A_i = C_i + \Delta\Phi_i = O(min(k,n)) - 2 * min(k,n)$
  $= O(min(k,n)) - O(min(k,n)) = O(1)$

Thus, use the potential function $\Phi(n,m) = 3n + m$, we have shown that the amortized running time for each operation is $O(1)$.

Now, let's define a different potential function $\Psi(n,m) = an + bm$, where a and b are constants.

$PushA(x)$:

- Actual cost: $C_i = O(1)\ as\ given$
- Potential change (note: the size of stack A increase by one for each operation):
  $\Delta\Phi_i = \{a(n + 1) + bm\} - \{an + bm\} = a$
- Amortized cost: $A_i = C_i + \Delta\Phi_i = O(1) + a = a$

$PushB(x)$:

- Actual cost: $C_i = O(1)\ as\ given$
- Potential change (note: the size of stack A increase by one for each operation):
  $\Delta\Phi_i = \{an + b(m + 1)\} - \{an + bm\} = b$
- Amortized cost: $A_i = C_i + \Delta\Phi_i = O(1) + b = b$

The coefficients a and b will determine the change in potential, and thus the amortized cost of each operation.