Comp 160: Algorithms **Recitation 7**

1. Work in pairs and write bullet points explaining how the accountant technique can bound the amortized runtime of an algorithm (try to be as general as possible. Do not limit the explanation to a single example)

   - Rather than tracking runtime in a usual fashion we tie runtime to number of coins withdrawn.
   - Each coin withdrawn accounts for $\Theta(1)$ runtime. Thus, Final runtime is Θ(# coins withdrawn)
   - We deposit a fixed amount of virtual coins (say c) each time an operation is invoked.
   - In total we invoke k operations. Thus, overall we deposit ck coins.
   - We withdraw the coins when we execute the operation.
   - We never run out of coins, which means we withdraw at most ck coins.
   - At most ck coins and each is $\Theta(1)$ runtime $\Rightarrow$ total runtime of n operations is $O(ck)$ (or $O(\frac{ck}{n})$ amortized)

   Summary: we deposit at most ck coins, each corresponding to $O(1)$ runtime. In total, we cannot withdraw more than $O(ck)$ time.

   In most cases c is some fixed constant and $k = n$. Thus, we will get $O(n)$ total time needed to do all n operations. But we could have $c = \log n$ or any other number.

2. Do the the same process with the potential technique

   - Let c_i be the *real cost* of doing the i-th operation. The runtime is $\sum_i c_i$
   - It is difficult to give a bound for c_i (sometimes it is low, sometimes it is high). We want to somehow flatten the runtimes
   - In order to make all runtimes as similar as possible we introduce the *amortized cost* $\hat{c}_i = c_i + \phi_i - \phi_{i-1}$
   - ϕ_i is the potential of the data structure after the i-th operation
   - In short, when c_i is small, the potential should grow ($\phi_i - \phi_{i-1}$ must be positive). The reverse when c_i is big
   - With some minor requirements on ϕ we can show $\sum c_i \leq \sum \hat{c}_i$
   - Use some magic to show that $\hat{c}_i$ is small (say, $\hat{c}_i \leq O(1)$ or $\hat{c}_i \leq O(\log n)$)
   - Total time spent by the algorithm is equal to $\sum c_i \leq \sum \hat{c}_i \leq n \cdot$the bound obtained in previous step

   Summary: cheap operations increase the potential. Expensive operations must use enough potential to make sure $\hat{c}_i$ remains low.

3. Lets refresh the dynamic array data structure and a discussion on amortized cost.

(a) Quickly review the accounting method for proving that Inserting n elements in a dynamic array needs $O(n)$ time in total

We have two types of operations:

(vanilla) Insertion An insertion that does not expand the array. We charge 3 coins per such operation

Insertion + expand An insertion that expands the array. We do not charge extra for this operation (conceptually speaking, we charge 3 for the insertion, 0 for the expand)

Note: to be clear, for the user there is a single operation (knowing when we expand or not is an abstract data violation, so we must charge the same number of coins to both operations). In this case, we charge 3 coins per operation. We just visualize it as charging 3 coins to the insertion and zero to the expansion.

Lemma 1. *With the coin cost described above, we never run out of coins*

Proof. (sketch. Full proof was shown in class)

- Each vanilla insertion nets us a gain of 2 coins. We assign them to the just inserted coin
- If we need to expand, the array is full. More importantly, the later half of the array still has 2 unspent coins
- We have at least $2 \cdot n_i/2$ coins, where n_i is the current size of the array
- We use them all to copy the array. The result is an array that the first half is occupied and second part is empty
- Expand operation used most (or all) of the previously stored coins, we still have 3 coins for the follow-up insertion
- Next operations will occupy empty spots, and starts saving two coins each as before

□

(b) What has changed from the Dynamic Array that was described in Comp 15 (or any previous data structure course you studied)?

Nothing! We just gave an argument to better bound the cost of doing n operations.

(c) Assume that we initialize the Dynamic array with an array of size 2 BUT we start with 1 element already inserted. What would change in the runtime?

Nothing! In our coin strategy we only use the coins in the second half of the array. The element inserted at the beginning would not have coins, but those are never used.

(d) In a normal dynamic array when we expand we double the array size. Say that we change it and instead grows the size of the array by only 10%. Can we still have $O(n)$ total time? If so, what would change in the previous proof argument?

Yes. The trick is to now change the constants: specifically, we deposit 12 coins per insertion (and still 0 for expansion). Each insertion now gains a net benefit of 11 coins). Now, the invariant is that when we we expand into an array of size n_i, the last $\frac{n_i}{11}$ positions of the array are currently empty (note: we increased the array by 10% which means a 1/11-th fraction of the new size of the array is empty) but when the array is eventually filled, each element in the last portion of the array will save 11 coins each. This gives a total of n_i coins that can be withdrawn when copying the n_i elements in the expansion. After the next expansion, the last $n/11$ positions are again empty, so we can start saving coins again.

Fun question: how many coins do we need to charge if we expand the array by x% each time? Does your solution match the answers we computed for 10% and 100%? Why?

(e) What if we now grows the size of the array by 10 (a constant each time)? Can we still have $O(n)$ total time? If so, what would change in the previous proof argument?

Unfortunately the cost would grow to $\Theta(n^2)$. Every 10 insertions we will need an expand. Thus, we will do $n/10$ expands, each needing $\Omega(n)$ time. In total, such a dynamic array would use $\Omega(n^2)$ time to do the n insertions.

Additional practice questions:

4. For simplicity, in our discussion of hash tables we always assumed that n (the number of elements that we insert into the table) is known in advance. If this is not the case, we need to introduce some modifications to the hash data structure.

In our structure now we must keep track of two new values: for any instant of time i, let n_i and m_i represent the number of elements and buckets in the hash table, respectively. When table is created (i.e., $t = 0$, we start with $n_0 = 0$ (since the table does not contain any elements) and $m_0 = 2$ (arbitrary choice). We must also fix a desired upper bound on the load factor α (for this exercise we will set $\alpha = 1$, Java uses 0.75 as the default. In practice a number like 2 or 3 would be better).

Before inserting element e_i in the table, you check if after insertion, the load factor would exceed α. If so, we create a new array of double the size of the current array, rehash all elements into the new table, insert e_i (and update values n_i and m_i). Otherwise, no resizing is needed so we proceed with the usual insertion (and update n_i). For simplicity, we will assume that deletions are not allowed, and we are handling collisions with chaining.

(a) Explain in 1-2 sentences why now the worst case runtime of insertion under this modification is $\Theta(n)$

Any time we insert and need to expand we must pay $\Theta(n_i)$. So any expansion where $n_i > n/2$ will have this situation.

(b) The remainder of the exercise will show that insertion has an amortized cost of $O(1)$. Assume this is true. What does it mean to say *insertion has an amortized cost of $O(1)$*? Explain it in your own words

One single operation may need linear time, but n executions will need in total $O(n)$ time.

(c) For a hash table H_i we use the potential function $\Phi(H_i) = 2n_i - m_i + 2$.

i. What is the potential of an empty hash table? Remember that we do not allow deletions and initial table size is 2
$2 \cdot 0 - 2 + 2 = 0$

ii. What is the potential of a hash table when $n_i = 5$ and $m_i = 8$?
$2 \cdot 5 - 8 + 2 = 4$

iii. Can the potential be ever negative? Why? Remember that we rehash when $\alpha > 1$
no, because as soon as we have inserted one element in the table, we will always satisfy $n_i \leq m_i \leq 2n_i$ (and thus $2n_i - m_i \geq 0$).

(d) Say that we execute an insertion that does not need to rehash all elements. Show that the potential grows by 2 (that is $\Phi(H_i) - \Phi(H_{i-1}) = 2$)

If we do not rehash elements, then we only increase n by one and do not modify m. In other words: $m_i = m_{i-1}$ and $n_i = n_{i-1} + 1$. Thus $\Phi(H_i) - \Phi(H_{i-1}) = 2(n_{i-1} + 1) - m_{i-1} + 2 - (2n_{i-1} - m_{i-1} + 2) = 2$

(e) Say that we do an insertion that causes a rehash. Show that the potential after the insertion is 4 (that is $\Phi(H_i) = 4$). What is the value of $\Phi(H_i) - \Phi(H_{i-1})$?

Now we have $m_i = 2m_{i-1}$ and $n_i = n_{i-1} + 1$. More before we were full, so $n_{i-1} = m_{i-1}$ and $2n_{i-1} = m_i$.

So, $\Phi(H_i) = 2n_i - m_i + 2 = 2(n_{i-1} + 1) - 2n_{i-1} + 2 = 4$.

Similarly:

$$\begin{aligned}
\Phi(H_i) - \Phi(H_{i-1}) &= 2n_i - m_i + 2 - (2n_{i-1} - m_{i-1} + 2) && (1)\\
&= 2n_{i-1} + 2 - 2n_{i-1} + 2 - (2n_{i-1} - n_{i-1} + 2) && (2)\\
&= 4 - (n_{i-1} + 2) = 2 - n_{i-1} && (3)
\end{aligned}$$

(f) With the above results, show that the amortized cost $\hat{c}_i$ of an insertion satisfies $\hat{c}_i \leq 3$, regardless of whether or not it caused a rehash. For simplicity, you may assume that it takes 1 unit of time to insert an element in a hash table (in normal insertion or when rehashing) and that the load factor α is 1. You can ignore times needed to check if pointers are null, time needed to deallocate memory, and other similar computations.

In an insertion without expansion we have $c_i = 1$ but if we need to expand cost is $c_i = n_i$ (this makes sense with previous intuition, in the former case we insert one element, whereas in the latter case we need to rehash all elements of the table). So, now we have:

Without expansion: $\hat{c}_i = c_i + \phi_i - \phi_{i-1} = 1 + 2 = 3$.

With expansion: $\hat{c}_i = c_i + \phi_i - \phi_{i-1} = n_i + (2 - n_{i-1}) = (n_i - n_{i-1}) + 2 = 3$.

(g) Use your previous answers to justify (with 4-6 bullet points) why amortized cost of insertion is $O(1)$.

Pasted from recitation:

- Let c_i be the *real cost* of doing the i-th operation. The runtime is $\sum_i c_i$
- It is difficult to give a bound for c_i (sometimes it is low, sometimes it is high). We want to somehow flatten the runtimes
- In order to make all runtimes as similar as possible we introduce the *amortized cost* $\hat{c}_i = c_i + \phi_i - \phi_{i-1}$
- ϕ_i is the potential of the data structure after the i-th operation
- In short, when c_i is small, the potential should grow ($\phi_i - \phi_{i-1}$ must be positive). The reverse when c_i is big
- With some minor requirements on ϕ we can show $\sum c_i \leq \sum \hat{c}_i$
- Use some magic to show that $\hat{c}_i$ is small (say, $\hat{c}_i \leq O(1)$ or $\hat{c}_i \leq O(\log n)$)
- Total time spent by the algorithm is equal to $\sum c_i \leq \sum \hat{c}_i \leq n\cdot$the bound obtained in previous step

Summary: cheap operations increase the potential whereas expensive operations use the accumulated potential to make sure $\hat{c}_i$ remains low.

5. We consider several variations of the bit counter problem described in class. Consider each variation independent from the previous ones

(a) Suppose we wish not only to increment a counter, but also reset it to zero. Show how to implement the counter as an array of bits so that any sequence of n INCREMENT and RESET operations takes $O(n)$ time on an initially zero counter. Justify the runtime using the potential method. Make your proof as structured as possible

Note: This is a great practice of not only the concept of amortization, but also how to structure a relatively long argument. We encourage you to break the whole argument into pieces, make a few helpful lemmas, and finally glue all pieces to make one big statement.

(solution draft. Would not be given full credit in an exam)

There are many possible solutions, but the key point is that we need to modify our data structure. As before, we need to store a value per bit, but in addition we must also have some way of knowing when we are done reseting. Say that the current number is 000000111. Then, we want a RESET operation that starts from the right, changes 1s to 0s and (very important) stops as soon as it has changed the last 1 (in the example above, it should never look past the third digit). The simplest way to do so, is to keep track of the number of 1s in the vector. All operations need to. keep track and maintain this attribute, but it is fairly easy to do so in constant time.

A good potential function would be $\Phi(\text{bit vector}) = \#$ if 1s in vector + position of most significative 1 (starting from 1, counting from right to left).

So, for example $\Phi(00100000000) = 1 + 9 = 10$, $\Phi(001011010110) = 6 + 9 = 15$ and $\Phi(00000000000) = 0 + 0 = 0$.

In this case we can show that amortized cost $\tilde{c}_i$ of an INCREMENT is $\tilde{c}_i \leq 3$: the bit manipulation has a cost of 2 as shown in lecture. In position of the most significative bit increases. This would happen if the "Rest of the vector" were all zeroes (see lecture slides).

A RESET would have a negative amortized cost. This is because (i) the potential after a reset is zero, (ii) we look at bits from the right up to the most significative 1, and (iii) the potential before the reset was bigger than the number of bits from the right up to the most significative 1.

Naturally, the fact that the (amortized) cost of an operation is negative does not mean that it will take negative time to run an operation (insert back to the future music). What it means is that somehow we overcharged previous INCREMENT operations and now we are taking into account that overcharge.

(b) Using your answer to the previous question, fill in the following table:

Counter	Operation	c_i	Φ	$\hat{c}_i$
010011111110	INCREMENT	N/A		N/A
010011111111	INCREMENT			
010100000000	RESET			
000000000000				

Note that positions with N/A cannot be filled in (we would need to know the previous or following operation).

Again, many solutions depending on the exact potential. In our case we have (note how potential is expressed as a sum of the two terms for ease):

Counter	Operation	c_i	Φ	$\hat{c}_i$
010011111110	INCREMENT	N/A	8+11	N/A
010011111111	INCREMENT	1	9+11	2
010100000000	RESET	9	2+11	2
000000000000		11	0+0	-2

Note that positions with N/A cannot be filled in (we would need to know the previous or following operation).

(c) Show that if a DECREMENT operation were included in the k-bit counterexample, n operations would count as much as $\Theta(nk)$ time

Note: This is Problem 17.1-2 from CLRS.

We need to examine the worst case for each operation. For INCREMENT, this is when all but the leftmost bit are 1's, which means we need $\Theta(k)$ time to flip all k

bits. DECREMENT is analogous. Thus, a series of n alternating INCREMENT and DECREMENT operations will result in $n \cdot \Theta(k) = \Theta(nk)$ time, as shown:

$$[0,1,\ldots,1] \xrightarrow{\text{INCREMENT}} [1,0,\ldots,0] \xrightarrow{\text{DECREMENT}} [0,1,\ldots,1] \xrightarrow{\text{INCREMENT}} \ldots$$

(d) What if the counter did not start at zero? Say that the counter begins with a number that has b many 1's in binary representation. Then, we execute $n = \Omega(b)$ INCREMENT operations. Show that the total cost is $O(n)$.

This is Problem 17.3-5 from CLRS. At first sight, it looks like a tricky question. We could use exactly the same potential as explained in class all seems to work. The only change would be that instead of starting with zero potential, we now have $\Phi(D_0) = b$. Is there anything wrong with that?

Exactly like in lecture, we can derive that the amortized cost $\hat{c}_i$ of the i-th INCREMENT is 2 (none of the calculations used the initial value in the argumentation). This implies that $\sum_{i=1}^{n} \hat{c}_i = 2n = \Theta(n)$, so what is the problem?

Keep in mind that our goal is not to show that some abstract amortized time is linear. We want to show that the real cost ($\sum_{i=1}^{n} c_i$) takes linear time. And in this case, it is no longer true that $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$, which is a key step for correctness of our analysis.

More formally:

$$\sum_{i=1}^{n} \hat{c} \;=\; \sum_{i=1}^{n} c_i + (\Phi(D_n) - \Phi(D_0))$$

But if $\Phi(D_n) < b$ we cannot upper bound $\sum_{i=1}^{n} c_i$ by $\sum_{i=1}^{n} c_i$ as we normally do. Indeed, imagine that initially, we set the counter to 01111111 and the super extreme case in which $n = 1$: we do a single operation, lose a lot of potential and end. The computer spent a lot of time updating but we only spent 2 amortized cost? That does not make sense!

The above example was extreme (since we are interested in large values of n), but it shows well the point. What we need to do is modify our potential function to guarantee that $\Phi(D_n) \geq b$ regardless of the number of 1s in D_n or in the initial bit vector.

There are many ways to achieve this, one of them being $\Phi(D_i) =$ number of 1s in $D_i + i \cdot \frac{b}{n}$. Note how the potential is exactly like before, but we now add a small cost $\frac{b}{n}$ that grows the more operations we do. This allows us to slowly increase the potential.

With this new potential function we can show that the amortized cost of textscIncrement is $2 + \frac{b}{n}$ (2 is shown as before, the $\frac{b}{n}$ is because i grows by one). This is still a constant (that will tend to 2 as n grows).

Note: As you can see, we skipped several details of the proof. We thought than rather than giving the solution directly, we could show you how we deduced the modifications to the potential function instead. Why don't you try finishing this to a full proof? Do the math and indeed verify that with the new potential function everything works.

Counter	Operation	c_i	Φ	$\hat{c}_i$
010011111110	Increment	N/A		Starting condition
010011111111	Increment			
010100000000	N/A			

(e) Fill in the same table with the new potential function:

(f) What if we had variations described in a) and c) at the same time? Can you guarantee still linear time? What would be the new potential function?

(g) Assuming that the answer to the previous question was $O(n)$, fill in the table for a third (and final) time.

Counter	Operation	c_i	Φ	$\hat{c}_i$
010011111110	Increment	N/A		Starting condition
010011111111	Increment			
010100000000	Reset			
000000000000	N/A			

(h) Imagine we implement the answer to the previous questions. What differences would we notice in the code between each of the implementations?

(i) For all of the previous questions that had $\Theta(n)$ total time, repeat the proofs using the accountant method instead

6. A **disjoint-set data structure** (also called a **union-find data structure**) is a structure of disjoint sets that lends itself to various operations including Union and Find-Set, hence the two different names for the structure. In this problem we will focus on the amortized cost of Union in a context that will be relevant later for Kruskal's MST algorithm.

   In short, we need a data structure to answer the following operations:

   - Make-Set(x), which creates a new set containing only the element x.
   - Union(x, y), which creates a new set that is the union of the two sets containing the elements x and y (call them S_x and S_y). We assume S_x and S_y are disjoint.
   - Query(x, y) which returns true if x and y are in the same set (false otherwise)

   We are interested in the following complete merge process: given a list of n elements, we first execute the Make-Set operation for every element, so that we now have n sets with one element each. This is followed by a sequence of Union operations until we obtain a single set with n elements. (In Kruskal's algorithm the will be additional Query(x, y) operations, but here we are focusing mainly in the other two operations).

   (a) What data structure would you use to handle these operations?

- First, we describe the data structure to store a single set. The elements of the set are kept in a linked list. The first element of the list is the *representative* of the set. For each list we also keep a counter of how many elements are in the list.
- We will have several sets. We store them in a dynamic array of sets. Note that a static array also works if we know an upper bound on the number of sets that will be created.
- In addition to having the dynamic array of sets we need to efficiently find the set containing a given a query element x. For this we use a hash table. The key are the elements, and for each element we store in which set they are currently in (say, they store the index within the dynamic array of sets that contains it).

(b) How many MAKE-SET operations were required?

n.

(c) What is the cost of a single MAKE-SET operation?

Given x we must create a new set containing a single element (x), and insert this set in the dynamic array. Additionally, we must insert x into the hash table (paired with its position within the dynamic array). Everything can be done in worst case constant time, except possibly the insertion in the dynamic array that could create an expand. Runtime is amortized $O(1)$ (as usual in dynamic array insertion). From now on we assume that n is known in advance. If so, we can instead use a static array and thus runtime becomes $O(1)$ worst case. Note that it just simplifies the description below and everything would work if n is not known.

(d) How many UNION operations are required until we only have one set?

$n-1$, because we start with n sets and each time we perform UNION we have one fewer set, until we end up with 1 set.

(e) Describe the code of the QUERY operation. What is the runtime?

Search for y and x in the hash tables. Return `true` if both elements point to the same set, `false` otherwise. Since we use hash tables runtime is expected $O(1)$.

(f) Describe the code of the UNION operation

Search for y and x in the hash table. Merging the two sets can be done in $O(1)$ time by playing with pointers (and updating the count of elements in the new set). However, we need to go through all elements in one of the two sets and update in their hash tables that the representative has changed.

(g) What is the worst-case cost of a single UNION operation?

$\Theta(n)$. This happens, for example when we merge two sets, each containing $n/2$ elements. We must update every a pointer per element in one of the tables, leading to the $\Theta(n)$ runtime.

(h) (The key result) Show that the amortized cost of the operations in the complete merge process is $O(\log n)$ per operation.

Hint: There is a choice of which set to absorb into the other during UNION; *the* *key result depends on making this choice strategically.*

The key trick is that when we need to do the union of two sets, we update the pointers of the *smaller* set. Although it seems a minor change that has no impact in worst-case runtime, it actually has a very important impact in amortized runtime.

Lemma 2. *The total runtime of executing n times* Make-Set *and $n-1$ times* Union *is $O(n \log n)$.*

Proof. We will prove the statement using the accountant method
Each time we invoke Make-Set(x) for some x we deposit $\log n + 1$ coins. We withdraw 1 to pay for the runtime of Make-Set and there are $\log n$ coins remaining which can be associated to the element x. Each time we invoke Union we do not deposit any coins, but we must withdraw a coin for each element of the set which is absorbed, and each coin corresponds to $O(1)$ runtime. We strategically always choose to absorb the smaller set into the larger one when we invoke Union, which minimizes the number of coins withdrawn.

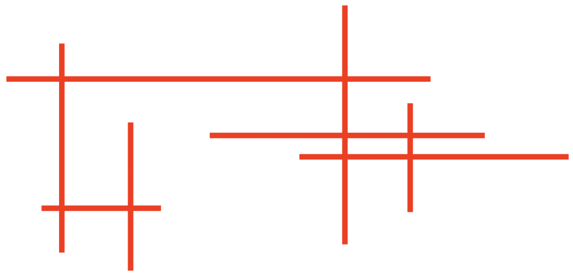
Let's look at a simple example: when we execute Make-Set(x), x will have $\log n$ coins left. Say we invoke Union with a larger set. Then, we will need to update the representative of x, which means we need to withdraw a coin ($\log n - 1$ left). Then x we invoke Union with a smaller set. In this case, we will need to do some updating, but none associated to x. This means that no coins are withdrawn (for x) and we still have the $\log n - 1$.
The key consequence of always absorbing the smaller set is that each time an element x is absorbed into a new set (and causes an associated coin to be withdrawn), the size of the new set containing x will be at least **twice** the size of the old set. Since the final set is size n, this absorbing will happen to x at most $\log n$ times and thus we have enough coins.

Overall we deposited $n(\log n+1) = O(n \log n)$ coins and we never ran out of coins. We executed $2n - 1 = O(n)$ operations. Thus our amortized cost per operation is $O(\log n)$. □

Note: Wow! Things are getting interesting. For this question we had to use expected (for the hash table) and amortized runtime. And all of this will be used in a graph algorithm in the future! You have to admit that algorithms are getting more and more interesting now!
Note 2: If we merge the small element into the large one the analysis would fail. Can you point out where exactly?

7. **Challenge**: Consider a set S of $2n$ line segments in 2D: n horizontal and n vertical. For example:

   Assume that no two segments have endpoints at the same x-coordinate or y-coordinate.



It is easy to determine if two given segments intersect, in constant time, by comparing the coordinates of their 4 endpoints, so don't worry about that detail.

Your job is to provide a sub-quadratic-time algorithm (i.e., $\Theta(n^2)$ doesn't cut it) to **count how many intersections there are among the segments in S**. Note that you do not need to list the intersections, just report how many there are.

Hints:

(a) Visualize a vertical line sweeping over S, starting from the far left, ending to the far right.

(b) Consider only the horizontal segments in S. How can we track the horizontal segments encountered by the sweeping line? What data structure is appropriate, and what data is important?

(c) Now consider all segments (including vertical segments) in S. How can we detect when the sweeping line has encountered a vertical segment?

(d) Using our data structure from (ii.), how do we determine the number of intersections crossing any given vertical line?

Given the assumption stated, no horizontal segments overlap each other, and no vertical segments overlap each other. So we are only looking for intersections of vertical vs horizontal segments.

Sort all endpoints by x-coordinate, along with their "type" (belonging to a horizontal or vertical edge, and in the former case, whether they are the left or right end of the edge).

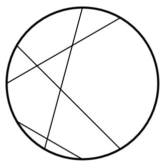
Then scan through this sorted list. Every time we find a left end of a horizontal segment, we insert the segment into a balanced BST, using its y-coordinate as a key. Every time we find the right endpoint of a horizontal segment, we delete its entry from the BST. This can be visualized as scanning through the 2D data from left to right with a vertical line. At any point in time, the BST will represent exactly the set of horizontal segments intersected by this vertical sweeping line. In fact the BST will contain the y-coordinates of all the horizontal segments that the sweep-line intersects, naturally in sorted order.

So far, we have ignored elements in the sorted list that correspond to vertical segments. As we scan through the sorted list, every time we find such an element, we will perform a range counting query in the BST, using the corresponding vertical segment as the query range. So, whenever our sweep line overlaps a vertical segment, v, we find out the number of overlaps that is has, in logarithmic time.

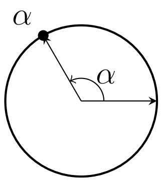
This algorithm takes $O(n \log n)$ time to sort all the endpoints. Then we set up an empty BST and we have a mix of n insertions, n deletions, and n range counting queries. We have seen how to maintain a balanced augmented BST that supports these operations and allows range counting, all in $O(\log n)$ time per operation, so the total time is $O(n \log n)$.

8. **Note**: this question was an exam question a couple of years ago. However, since that time we have removed the plane sweep algorithms from lectures (it is only mentioned, but not explained in detail). As such, this is question now lies outside the scope of the course.

   Consider a set S of n chords on a circle, each defined by its endpoints. We wish to design an $O(n \lg n)$-time algorithm to determine the number of pairs of chords that intersect inside the circle. (For example, if the n chords are all diameters that meet at the center, then the correct answer is $\binom{n}{2}$.) Assume that no two chords share an endpoint.

   (a) We can assume that the circle is centered at the origin and that the endpoints are described by the angle between the positive x-axis and the vector from the origin to the point.

   For example, a chord that connects the top to the bottom of the circle is described by $(90°, 270°)$. Given two chords (a, b) and (c, d), give a $O(1)$-time algorithm to determine whether the two chords intersect.

   Without loss of generality, assume $a < b$, c, d, and $a < c$ (we could rename the points so that this is true). Two chords intersect if and only if $a < c < b < d$. Such condition can be checked in constant time.

   (b) For a particular angle α, let $C_\alpha \subseteq S$ denote the set of chords such that their first endpoint is smaller, and their second endpoint is larger than α. Describe a data structure that, given a chord (α, β), $\alpha < \beta$, can return the number of chords in C_α that intersect with (α, β) in $O(\log n)$ time. (You want this data structure to be dynamic, i.e., support fast insertions and deletions, so that you can use your answer in (c))

If $(a, b) \in C_\alpha$, then $a < \alpha < b$. Therefore (a, b) intersects (α, β) if and only if $b < \beta$. Then, in order to count the number of intersections, we have to count the number of "second" endpoints smaller than β. This is equivalent to the rank of β in the set of "second" endpoints. We can achieve $O(\log n)$ query time by using a dynamically balanced BST (AVL or Red-Black tree, for example), augmented with subtree sizes as seen in class. The elements in the tree represent the chords in C_α, and their key is the angle value of their second endpoint.

(c) Using your answer for (b), describe an $O(n \lg n)$-time algorithm to determine the number of pairs of chords that intersect inside the circle.

We can use a technique called *sweeping* seen in recitation. Sweep from 0° until 360°. In practice, thin involves sorting the endpoints by angle, which takes $O(n \log n)$ and scanning the list in sorted order. When the first endpoint of a chord is encountered, add the chord to the tree and count the number of intersections with this chord and the chords already in the tree. When the second endpoint of a chord c is reached, remove c from the tree. Notice that no subsequent chord that gets inserted in the tree can intersect c, or else they would have an endpoint in between the angles that define c, which is a contradiction because they would have been inserted before the deletion of c.