MS-CS Course Note (Non-Credit Course 2)

Mark Zhou

March 2024

This is my course note on "Algorithms for Sorting, Searching and Indexing" provided by Colorado University of Boulder. This is a non-credit prep course for an MS-CS degree.

Contents

1 Asymptotic Notation: Big O

1.1 Time and Space Complexity

We have some questions regarding how to figure out which algo is faster.

Q1: what input should we choose.

Q2: how to evaluate time.

When considering implementation details, that is under the realm of performance analysis. We don't want to be bogged down to this level and only wish to focus on the algorithm.

The runtime is the number of basic operations. For each sorting algorithm, there will be a range of possible runtime, due to the differences between inputs, the best case, average case, and worst case respectfully.

The average time cost is hard to analyze while may be more desirable than the worst case cost, which is too pessimistic in some cases. Never use best case cost.

In most of cases we will use worst case cost to evaluate.

Let's assume the cost of Insertion Sort is $f(n)$.

Hypothetically (just for examples) speaking:

$f(n) = 0.05n^2 + 1.5n + 70$

We assume that the different algorithms such as addition or multiplication have the same unit cost, that is to say, they cost the same under one unit run. But is that the real case?

Of course not. Different operations have different costs for a unit run. To change that, we can change the coefficient of the original equations.

We care about which algorithm is faster asymptotically. This is why

this kind of analysis is called asymptotic analysis.

The naming is inspired by one idea, that is the comparison should be within the realm of actual usage, for instance, when sorting 10 numbers, algo1 may be slower than algo2, but while sorting 10000 numbers, algo1 may outfast the opposite, that's why we compare them asymptotically.

And to be aware, we focus more on cases with large numbers or unit cases than small samples.

1.2 Big O

$f = O(g(n))$

Meaning: $f()$ **is ASUMPTOTICALLY upper bounded by** $g(n)$

Here are some properties:

When $g()$ **overtakes** $f()$**, meaning in the chart, g is above f after some time, and then f will remain overtaken forever.**

The constant factor must be disregarded.

> Note: In mathematics (logic), the symbol $\exists$ is read as "there exists" and the symbol $\forall$ is read as for all.

$\exists k > 0, N_0, \forall n \geq N_0, f(n) \leq k * g(n)$

$f(n) \leq k * g(n)$ **only happens beyond some point in time. When the above condition happens, we say:**

$f(n) = O(g)$

Let's take some examples:

$f = (1/2)n^2$

$g = 0.1n^3$

Assuming k=10, and N_0=1

Using the equation above $f(n) \leq k * g(n)$, then we will have $f(n) \leq g(n)$, which has got rid of the constant k;

Then we can say,

$f(n) = O(g)$

a.k.a.

$0.5n^2 = O(0.1n^3)$

As long as $g(n)$ will overtake $f(n)$ after K, $g(n)$ will be the cost of $f(n)$, aka $f(n) = O(g)$.

Under such circumstances, adding some constants to the equation of f and g, won't change the fact that $f(n) = O(g)$.

The $g(n)$ must always be less than or equal to $f(n)$ to make $f(n) = O(g)$ valid: that is to say, after a certain point N_0, the output of the function $f(n)$ is less than or equal to the output of the function $g(n)$.

2 Big Omega, Big Theta, and Examples

2.1 $f(n) = \Omega(g)$

f is asymptotically lower bounded by g (n): that is to say, after a certain point N_0, the output of the function f (n) is bigger or equal to the output of the function g (n).Since O () and $\Omega()$ are the opposite, if f (n) = O (g), then $g(n) = \Omega(f)$.

2.2 $f(n) = \Theta(g)$

f is asymptotically equal to g (n). This looks like a Bollinger band in technical analysis in some ways.

$\exists k_1, k_2, \forall n \geq N_0, k_1(g) \geq g \geq k_2(g)$

If $f = \Theta(g)$, it means f is asymptotically equal to $\Theta(g)$, which we can say,

$f = O(g), f = \Omega(g)$

When dealing with functions, we will typically get rid of constant figures in the function, however,

we cannot ignore the exponent constant such as the 2 and 4 in 2 to the power of 4x.

2^{4x}

Here is an example of big theta.

$f(n) = 2n + 3\log_2^n + 5$

$g(n) = 15n + 35\log_2^n + \sqrt{n} + 17$

In the above equations, n is the only thing that matters. Therefore we can ignore the log and square root elements by considering them as constants.

2.3 Examples of Asymptotic Notations

Here are five functions;

$f(n) = 2n^2 + 3\log_2^n + 4\sqrt{n} + 15$

$g(n) = 200\sqrt{n} + 15\log_2^n + 14n\sqrt{n}$

$h(n) = n^2 + 2n + 3\log^{logn}$

$l(n) = n^3 + 15n^2 + 2.5n$

$m(n) = 4n^2 + 13n^2\log_2^n$

Let's find out which element with n is the BIGGER one.

$n^2 = n * n > nlogn > \sqrt{n}$

Before we can compare these functions, we should determine the dominating term of each one:

$f : n^2$

$g : n^{1.5}$

$h : n^2$

$l : n^3$

$m : n^2 logn$

In this case, we can place these functions in such order, $l > m > f > h > g$

The smaller one is the Omega of the bigger one, and vice versa. For example, $l(n) = \Omega(m)$**,** $m = O(l)$**, and so on.**

3 Binary Search

3.1 what do we need to do in a binary search?

We need to check if a given element, say 6, is located in the given list. The prerequisite is that the list must be sorted to do a binary search.

Let's assume the list is sorted in ascending order. First, we need to find the middle element of the list by calculating the index number of left and right border elements, a.k.a. the first and last one in the list.

Here is how we do it:

$mid = (left + right)//2$

In the equation, // means floor divided, which will return exact the middle element with odd number list and the smaller one of the middle two elements in even number list.

Now we will check if the middle one equals to the given element. If so, problem solved! If not, and the given one is smaller than the middle, we will focus on the left side of the list, a.k.a. the elements located on the left of the middle one. In this case, if the first middle has index n, then the new "right" border element will have the index n-1.

If the middle element is smaller, of course we will focus on the right portion of the original list since this is an ascendibg ordered list. And in this case, if the first middle element's index is n, the new "left" will have the index n+1. Then we will conduct the calculation by finding the new middle on the left/right part of the list (a.k.a. the new range with the updated left or right border). We will compare the given value with the new middle.

We will repeat the process untill either find the element in the list, returning True, or not, which is a little bit complecated:

By the end of the process, the updated left border index will be bigger than the right border index in an ascending ordered list, which is not possible to find a valid range. This means the given element is not in the list.

For example, there is a list [1,4,9,15,20], and the given element is 8.

First we will find out the mid one by calculating the indexes:

$(0 + 4)//2 = 2$

The value of index 2 is 9, which is bigger than the given value 8. Then we will focus on the left part.

The updated right border will be index 1, therefore the new mid will be:

$(0 + 1)//2 = 0$

Now the equation gives us element value 1. This is smaller than 8. The current right border index is 1, and the updated left border is 1 as well. So the new range will gives us;

$(1 + 1)//2 = 1$

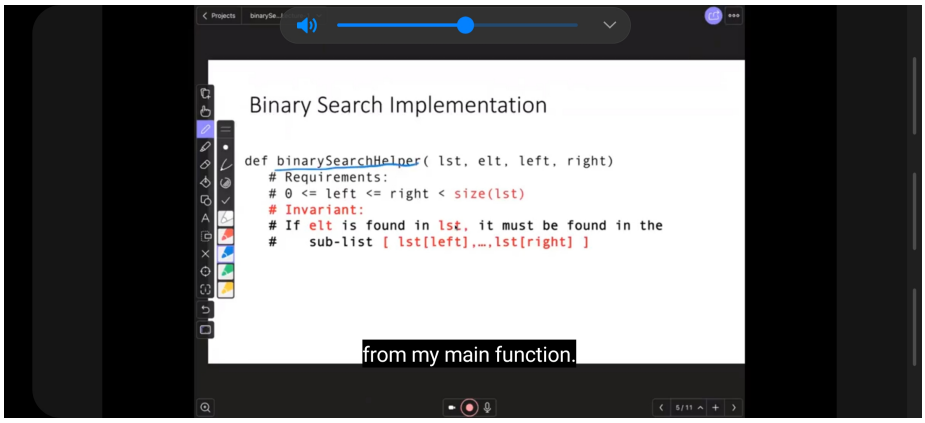
a.k.a. the value of index 1 is 4 which is smaller than 8.

And now we enter a crucial moment. The current right border index is 1, but since $4 > 8$, the updated left border index will be changed from 1 to 2.

Left index=2, right index =1, which is not valid.

Therefore, the given element 8 is not located in the given list. Problem solved.

3.2 Define a Func binarySearchHelper()



Here is the code implemented of the fuc binarySearchHelper().

This function will work as a recursion since it will continuously call a new binarySearchHelper() inside the current one until finished.

How to prove correctness?

The main property of binary search: if we can find the element in the list, it must be in the searching range.

3.3 Running Time Analysis of Binary Search Algo

For the running time analysis, we first assume this list has the length of:

$n = 2^k$

Each time when we check the middle element, we halve the list in half, then the updated list will have a new lengh of

$n/2 = 2^{k-1}$

By the end, the list will have a length of

$2^{k-k} = 1$

Now we narrow down the checking range to only one element and

check it with the given value. If they are not the same, we will try to find a new middle for more time, which will produce an invalid range as we mentioned before.

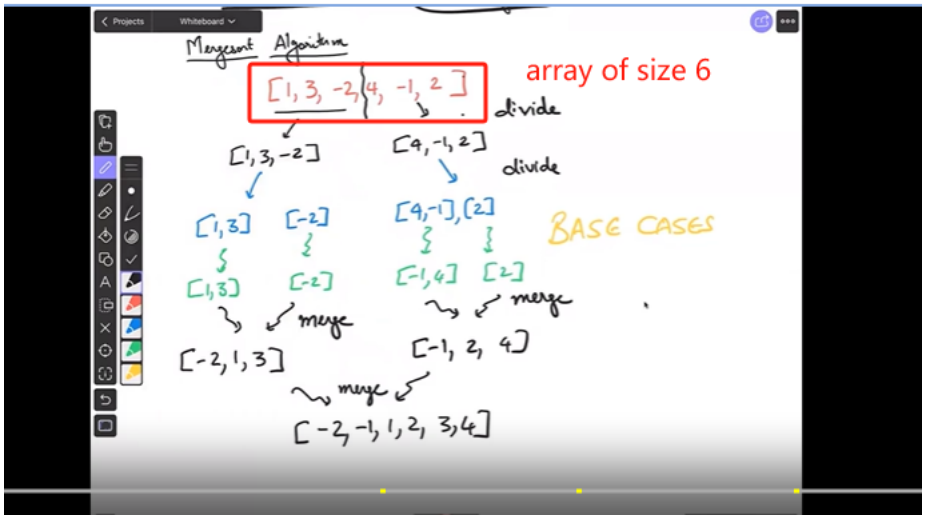
That is to say, we will at most run the program k+1 times until finished. So by the end, the largest running time will be:

$\log_2^n + 1$

Binary search is very efficient. If we have a list of 1 million values, we may only need to search for approximately 21 times.

4 Merge Sort

4.1 Mergesort: Correctness and Running Time



In order to do a mergesort, firstly we will divide the given array into two parts with the middle element as a separator.

For example, we have an array

$$1, 3, -3, 4, -1, 2$$

We will divide it into

$$1, 3, -2$$

and

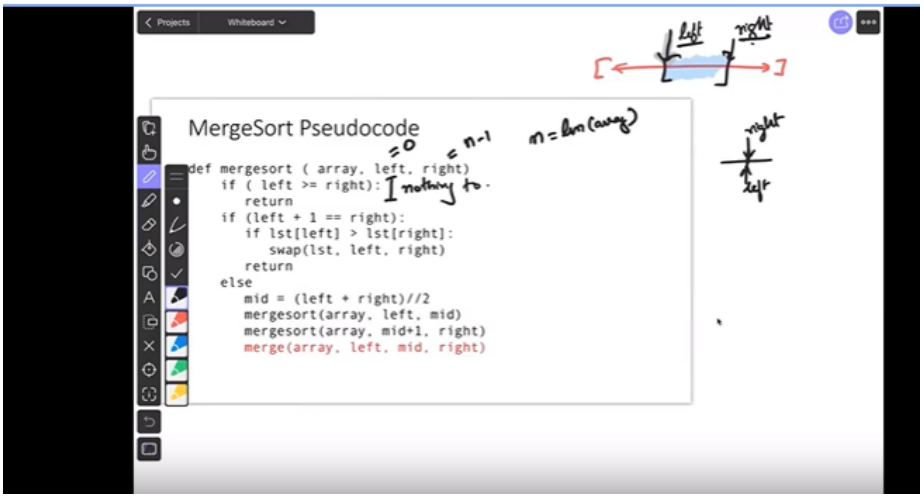
$$4, -1, 2$$

Then we will further divide these two into four parts. By now, we

have some array contain one one item while others contain two. For those with two items, we will do a swap if they are in the wrong order.

These kinds of very small divided array are called base cases, since they are somehow the base unit of this division process.

Then we will merge back all those sorted base case arrays, all the way to the top, aka one merged array.



Here is a pseudocode of mergesort. Let's get a closer look at how does it work.

At the very beginning, the algo will compare left and right element and check if they are in a valid form, that is to say, whether left (index of the value) is smaller than right (index of the value). If not, the range formed by the left and right index is invalid hence algo ended.

Second, we will determine whether the array formed by left and right contains only two elements. If $left + 1 = right$, that means these are adjacent indices. If this is the case, we will swap the elements if they have the wrong order.

At last, we enter a position in which the range is larger than two elements array. Now we will simply find the mid element by performing a floor division and do merge sort on left side of the array which is ended by mid and right side which begins with mid+1. This procedure will sort this two subarrays. After that, we will MERGE all the subarray in the order of left, mid and right.

The merge procedure is the next big thing. After this step, all the elements of the array will be sorted.
First thing in Merge function, we will assign an initial value to two looping variables, i and j, which are the index of each subarray, by assigning left to i and mid+1 to j. This is to make sure the two loops will begin from the first item of each sub array, aka the left array and right array.
We will introduce a temp storage variable `tem_store` to maintain the sorted result and then write back to the original array. Although there are in-place algos that will do the work, it is much easier to just include a third array for storage.
The following is straight forward. While i and j are still within the range of each subarray, aka $i \leq mid$ and $j \leq right$, we will compare each i and j pair and store the smaller one to the `tem_store`. Then we add one to the index of the subarray in which the value just compared is smaller, that is to say, move on to the next item. In the meantime, the other array remain the same index, waiting to compare in the next round.
After all these steps and iterations, we will find a well sorted `tem_store` array. Then we simply write it back to the orginal array and problem

solved.

4.2 Running Time of Merge Procedure

The running time of merge procedure is precisely the length of the two arrays since each element will be added to the temp store once. We assume each step cost 1, then the total time cost will be len(array A+array B).

It can be written as $right - left + 1$**.**

Now let's look at it at higher level.

Suppose the original array has lenth of n.

The first split will be two n/2 arrays, then the split goes on until it reaches the base unit.

For each step, the cost will be the array's count mutiply by the base unit, for example, for the 1st level, the cost will be $2 * (n/2)$**, for the 2nd, it will be** $4 * (n/4)$ **and so on.**

Now we can try to conclude a general equation for the time cost of merge sort.

Let us assume that $n = 2^k$**;**

Since for each level, the number of level is $n, n/2, n/4, \ldots, 1$**;**

We can write it as $2^{k-1}, 2^{k-2}, \ldots, 2^{k-k}$**.**

For each level, the time cost is the base unit of that level mutiply by that level's amount of subarrays in that level. For example, the base unit of the 1st level after n is n/2. Then the time cost will be $2 * (n/2)$**. Therefore, the generalized time cost will be** $n * No.(level)$**. Now let's find out what's the number of level.**

We have already seen this above: $2^{k-1}, 2^{k-2}, \ldots, 2^{k-k}$**, corresponding to level 1, level 2, . . . , last level. We can derive the number of level**

from this array, $1, 2, \ldots, k$**.**

That is to say, the number of level is k. Ergo $n * No.(level) = n * k$**.**

Since $n = 2^k, k = \log_2^n$**.**

The final equation will be:

$Cost = n * \log_2^n$

If we using the Big O quatation to address to this problem, we get

$Mergesort = \Theta(n * \log_2^n)$

And we can get rid of the constant 2 and get

$Mergesort = O(n * \log^n)$**.**

5 Heap, Min Heap and Max Heap

5.1 Basic Understanding of Heap

Heap is a type of array that has certain properties. There are the concepts of min heap and max heap and we will go through these soon.

First here's a concept called left/right child. Let's say we have a heap of 9 elements. For element number 1, the element number 2 will be called the left child of element 1, and element 3 will be called the right child.

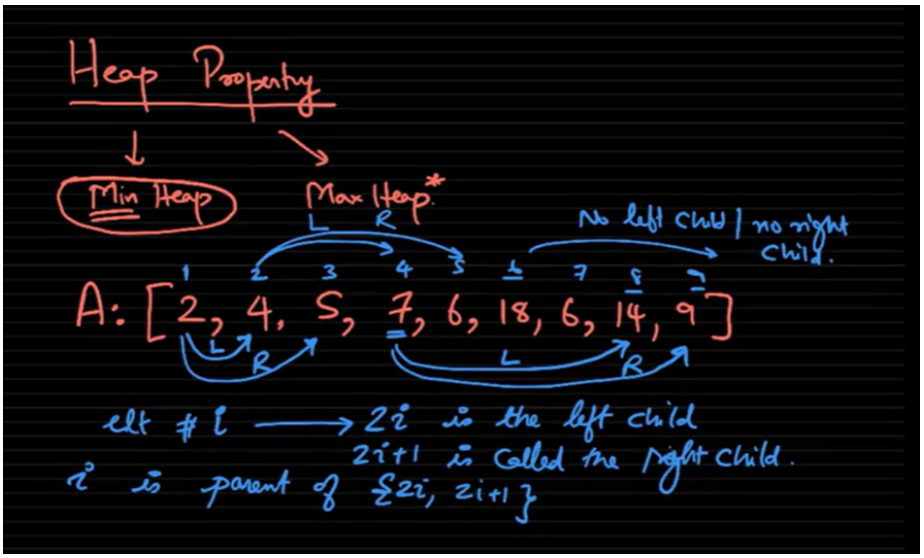
While for element 2, actually element 4 will be the left child and 5 the right child.

Therefore we have the equation of it:

$\forall i, leftchild = 2i, rightchild = 2i + 1$

For certain element in the array, If the left/right child element is not inside the current array, then the original one doesn't have left/right child.

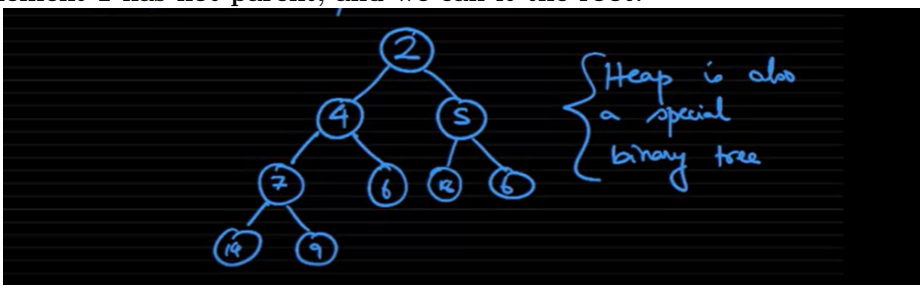
For example, element 6 has not such child since element 12 and 13 are not in the array of size 9.



We can say i element is the parent of 2i and 2i + 1.

for element j, j/2 is the parent. If the division gives a fraction of number then we just round it down (floor division).

Element 1 has not parent, and we call it the root.



If we write the heap array in a parent-child structure, it actually looks like a tree. Yes, heap is also a special binary tree (laid out as an array).

Here are some properties:

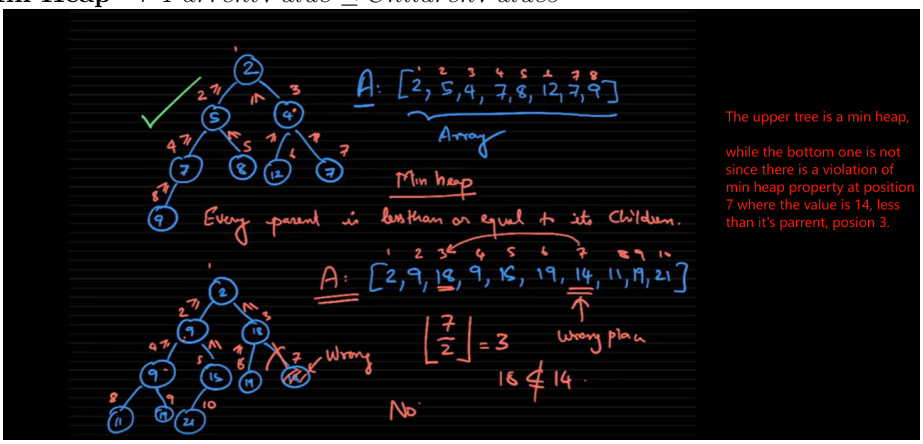
1. **If a node in a heap has a right child, it must have a left child.**
2. **Heap is an array of size n.** $\forall A[i], 2i \leq n, A[2i] = leftchild$

3. $\forall A[i], 2i+1 \leq n, A[2i+1] = rightchild$

And here is the Min Heap Property:

The value of the parent node must be less than or equal to the value of children nodes.

Min Heap $\rightarrow ParrentValue \leq ChildrenValues$



If we turn to the opposite side, we get the Max Heap Property: The value of the parent node must be larger than or equal to the value of children nodes.

Max Heap $\rightarrow ParentValue \geq ChildrenValues$

So for any array, we can simply check whether they meet min heap property or max heap property or not in order to determine their identity.

Why do we care about min heap or max heap? Here is a basic idea; For any min heap A, we can say A[1], which is the root, is the smallest element of the entire array.

We can solve this problem with induction.

The Base Case: The root element is the smallest one for Heap depth = 1.

And we have the induction hypothesis: Let the result from base case

hold for all $depth \leq d$, and then prove it for depth of d + 1.

Illustrated as a binary tree, we can see each triangle (consisting of three nodes) as a "sub-heap". The node in the upper level of the sub-heap is the root of it. For every level from 1 to d, the upper node is always the root of each sub-heap. Therefore the root of level 1 is the smallest in all level from 1 to d.

Then we can say that's the same case for level d+1.

That is to say, for any min heap A, we can say A[1], the root of it is the smallest element of the entire heap.

5.2 Heap Primatives: Bubble up

We are interested in the following operations on a heap:

1. Inserting elements into a heap

2. Deleting one from a heap

The main primitives of heap are called bubble up and bubble down. We will explain them later.

For now, lets assume there is a heap with only one point of failure. Here is an example of such min heap.

In the above array, the 6th element, which value is also 6, is smaller than its parrent element 3, with a value of 7. Besides that, the entire array fit the definition of min heap.

We can say in this case, the 6th element is in a wrong relation with its parent. It should be moved up in order to make this array a valid min heap.

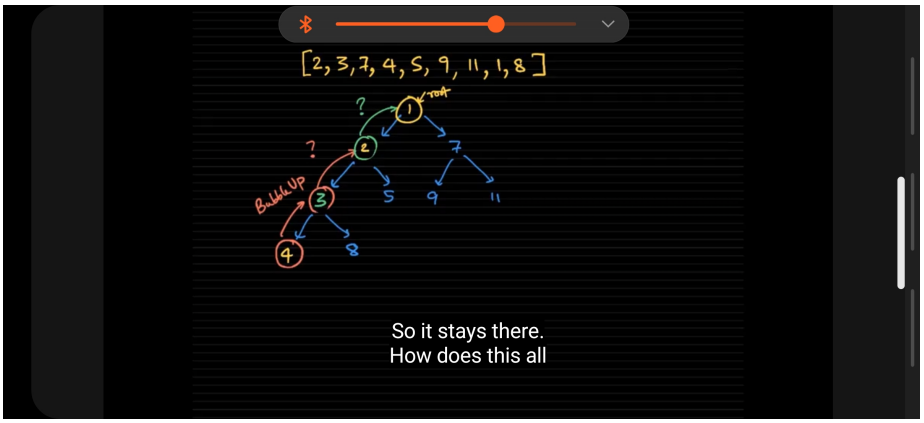
The operation that will acheive this is called bubble up.

Bubble Up: the swap of a value with its parrent value if they are in the wrong relation.

After we sway the misplaced value with its parent, now we have a new parrent of value 6 and its left child 7. Since the parrent is also less than or equal to its right child. The array is now a valid min heap.

Here is another example. We need to bubble up 3 times in order to make it a min heap.

$$2, 3, 7, 4, 5, 9, 11, 1, 8$$



In the above array, every elements except the last but one are in the right position. Element 8 with value of 1, is smaller than its parrent. Then we need to bubble up value 1 three times into position 4, 2, and 1 to make it work. Now value 1 become the root.

We can write a pseudocode for bubble up:

```
bubble up (A,j):
    if $j \leq 1$:
        return (Do nothing)
```

```
else if:
    $A[j] < A[j//2]$: (// means floor divided)
    swap (A[j], A[j//2])
    bubble up (A,j//2)
    #Yes, this line makes this function recursive.
return
```

This is a recursive program that will bubble up the misplaced element until it is in the right place.If we try to estimate the running time of bubble up, the worse case would be O (depth of heap), since the element will be a leaf (the one without child) and it lays on the bottom level of the heap.

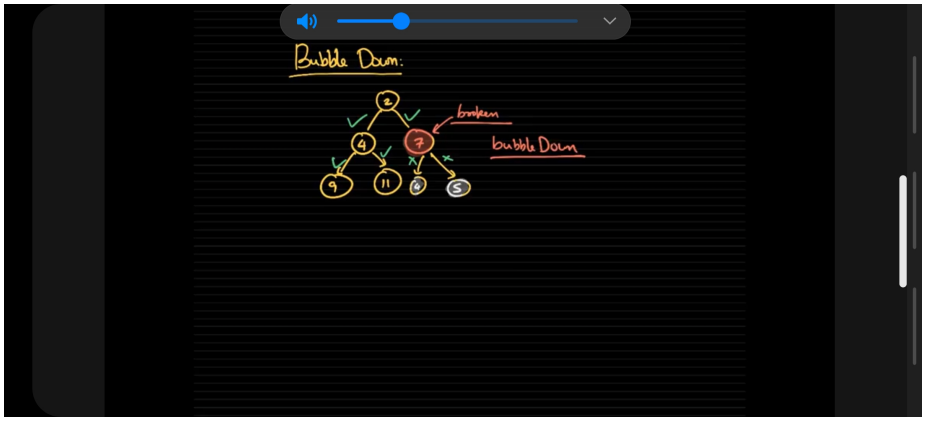
Then how about θ? Let's assume a heap has n element. If the leaf needed to be bubble up all the way to root position, the steps will be:

$n, n/2, n/4, \ldots, 1$

Then the steps will be $\log_2^n$, a.k.a. $\theta(\log_2^n)$.

If we have a heap of $n = 10^6$, which is 2^{19}, then the total operation required for bubble up is 19.

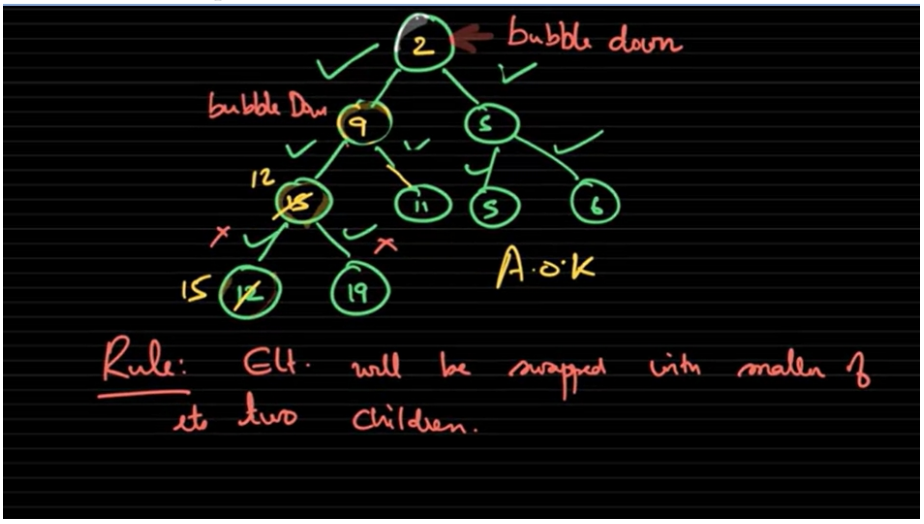
5.3 Bubble Down



In opposite to bubble up, bubble down is the operation that will move the root element to the right position. That is to say, if the root element is not the smallest one, we will swap it with the smallest child. So in the example above, the element 7, which is the parrent of 4 and 3, will be swapped with 3.

What if we cannot bubble down once and place the root element in the right position? Sometimes we need to do it multiple times.

Here is a min heap that needs to be bubbled down twice.



Now let's take a look at the pseudocode of bubble down.

```
bubble down (A,j):
    if $2j > n$: (2j is actually the left child, n is the length of A.)
        return (Do nothing since A has no children.)
    else if: $2j \leq n$ and $2j+1 \geq n$: (no right child)
        if $A[j] > A[2j]$:
            swap (A[j],A[2j])
            bubble down (A,2j) (recusive)
    else if: $2j \leq n$ and $2j+1 \leq n$: (has both children)
        if $A[2j] \leq A[2j+1]$ and $A[j] > A[2j]$:
        #Left child is smaller, which will be swapped with the current parrent.
            swap (A[j],A[2j])
            bubble down (A,2j) (recusive)
        else if $A[2j] > A[2j+1]$ and $A[j] > A[2j+1]$:
        #right child is smaller and will be swapped.
            swap (A[j],A[2j+1])
            bubble down (A,2j+1) (recusive)
    return
```

We will first compare 2j, which is the left child of j, with n, the length of A. If $2j > n$, j has no children whatsoever.
Then we go into the second condition, parrent j has only one child, the left one of course. If the parrent is bigger than its child, in this case $A[j] > A[2j]$, we need to swap them.
At this moment, we will need to run the recursive procedure, that is to call bubble down again.
The third condition is that the parrent has both children. We will

compare the left child with the right child. If the left child is smaller, we will compare it with the parrent. If the parrent is bigger, we will swap them. Of course, we will need to recursively run bubble down again.

By the end, if the right child is smaller, we will compare it with the parrent. If the parrent is bigger, we will swap them and call the main function recursively.

Problem solved.

How do we estimate the running time of bubble down?

The worst case will be the leaf node, which is the last level of the heap. That is to say, we need to bubble down the root element all the way to the bottom of the heap.

The time cost will be the depth of the heap, which is $\theta(\log_2^n)$.

How do we get this number? Let's do the math again.

In order to get the depth of the heap, we need to find out how many levels the heap has. The root element which needed to be bubbled down will move through the following steps;

$1, 2, 4, 8, 16, \ldots, n$

Which can be rewritten as $2^0, 2^1, 2^2, 2^3, 2^4, \ldots, 2^k$

k is the depth of the heap, and $k = \log_2^n$.

5.4 Reading Note on CLRS Chapter 6.3

Heapify is the process of reshaping a binary tree into a heap data structure.

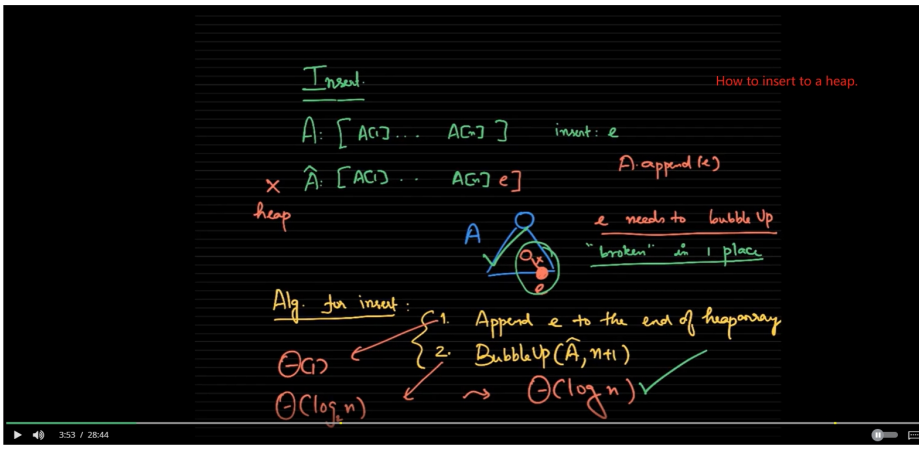
The main idea of min heapify is to make sure that the root of the

tree is the smallest element of the entire tree.

The min heapify procedure is a recursive one.

5.5 Priority Queue, Heapify and Heapsort

5.5.1 Inserting into a heap



Priority Queues, Heapify, and Heapsort

First we need to understand how to insert an element into a heap.

The main idea is to insert the element at the end of the array and then bubble up the element until it is in the right position.

That is to say, insertion = insert to the end + bubble up.

The time cost of inserting to the end of the heap is just as simple as $\theta(1)$**.**

Then we need to bubble up the element. The time cost is $\theta(\log_2^n)$**.**

So the total time cost is just $\theta(\log_2^n)$**.**

5.5.2 Deleting from a heap

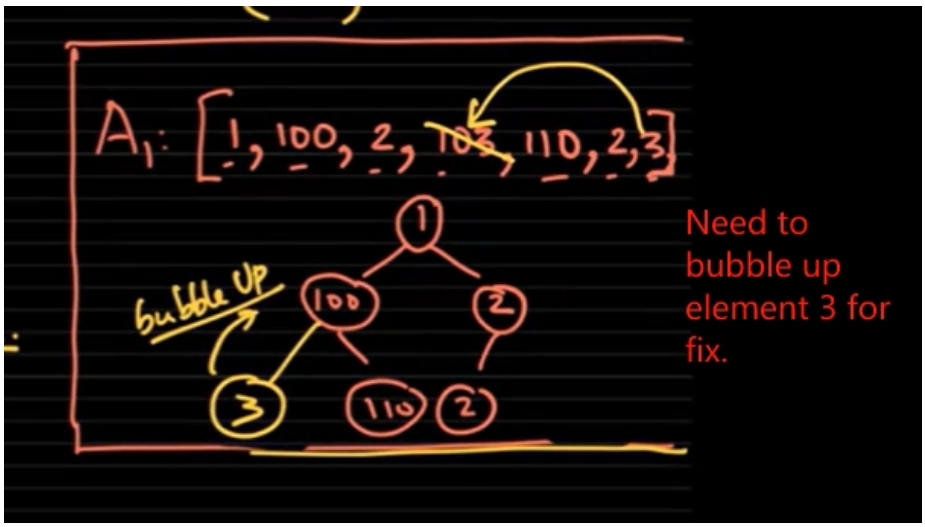


Figure 1: Bubble up for a fix.

Then we come to the concept of deletion of a heap element.

1. Replace the element that we want to delete with the last element of the heap.

2. Adjust the lenth of the heap to n-1, assuming the original length is n. Now we can simply delete the last element.

3. Bubble up or bubble down the element that we moved. The specific operation we use depends on the situation. We need only one operation for any cases.

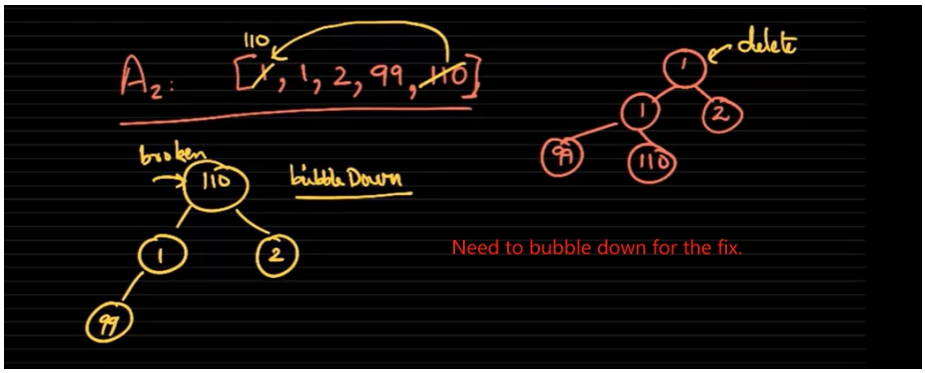


Figure 2: Bubble down for a fix.

Here is a question regarding the fix operation after deletion of a heap.

Question

Consider the heap $A = [a, b, c, d, e, f]$, and suppose we want to delete b from it. The first step is to move the last element, f, into b's position, giving $[a, f, c, d, e]$. The remaining step is to bubble f either up or down. How do we know that only one of these two directions is needed?

Try to answer in one or two sentences.

Because f is the only broken element. And it is either bigger or smaller than its parrent.

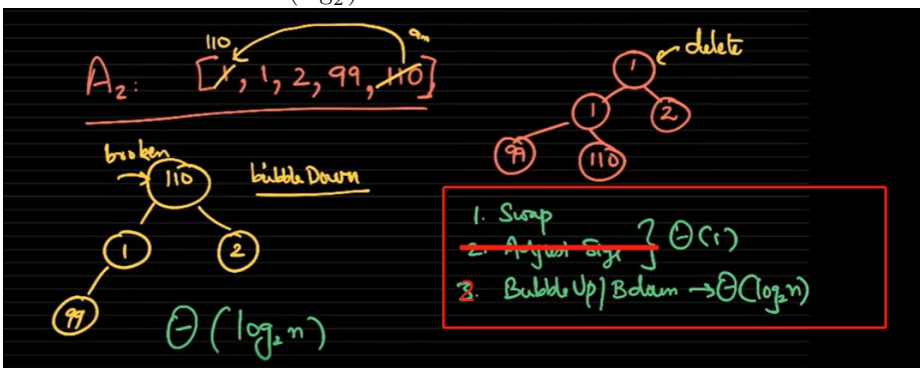
Correct

We only bubble f up if $a > f$, and we only bubble down if either $f > d$ or $f > e$ (or both). The min-heap property ensures that $a > d$ and $a > e$, and so f cannot be both less than a and greater than d or e.

The cost of time for deletion consists of two parts, 1 swap and then 2 bubble up/down.

The time of cost is $\theta(1)$ for step 1 and $\theta(\log_2^n)$ for step 2.

The total time cost is $\theta(\log_2^n)$.

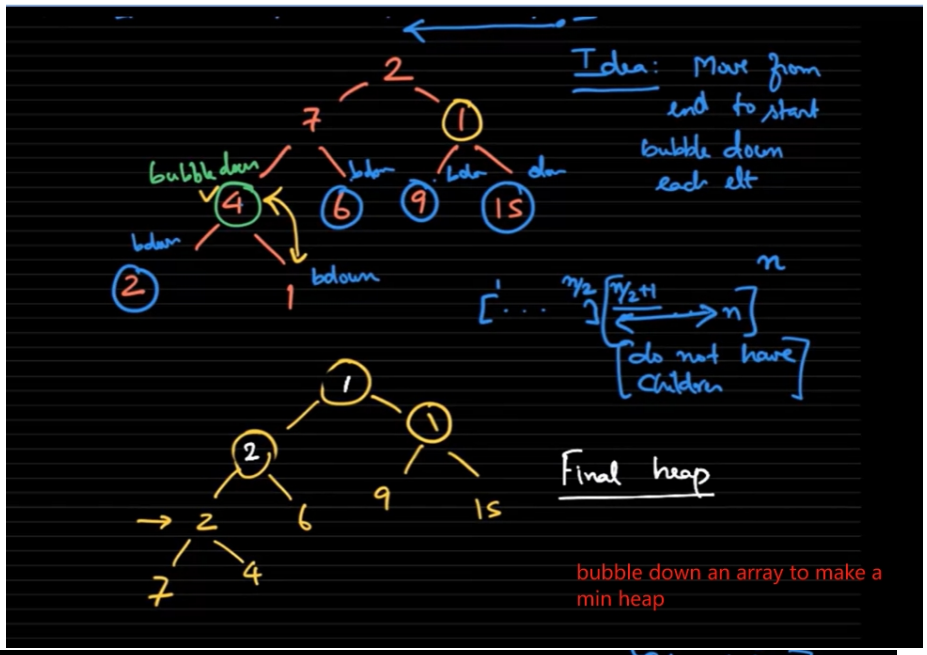


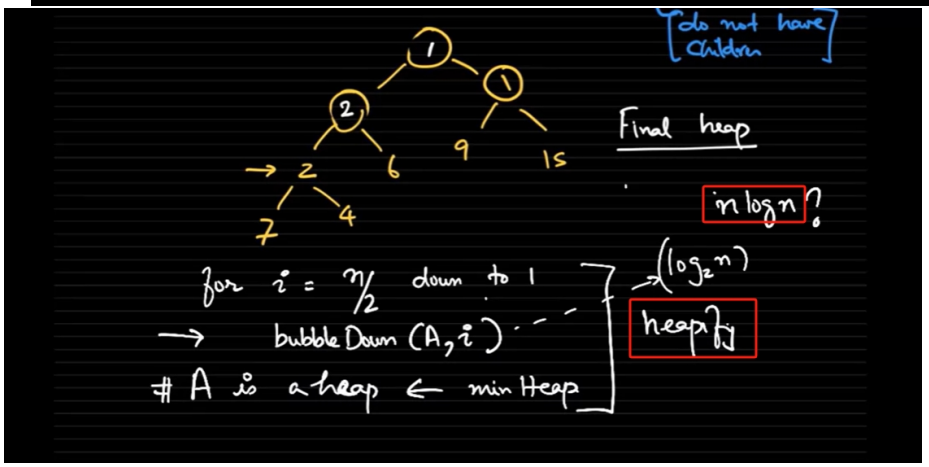
5.5.3 Finding the smallest element of a heap

The time cost of finding the smallest element of a min heap is $\theta(1)$. However, finding the largest element of a min heap is pretty hard and inefficient.

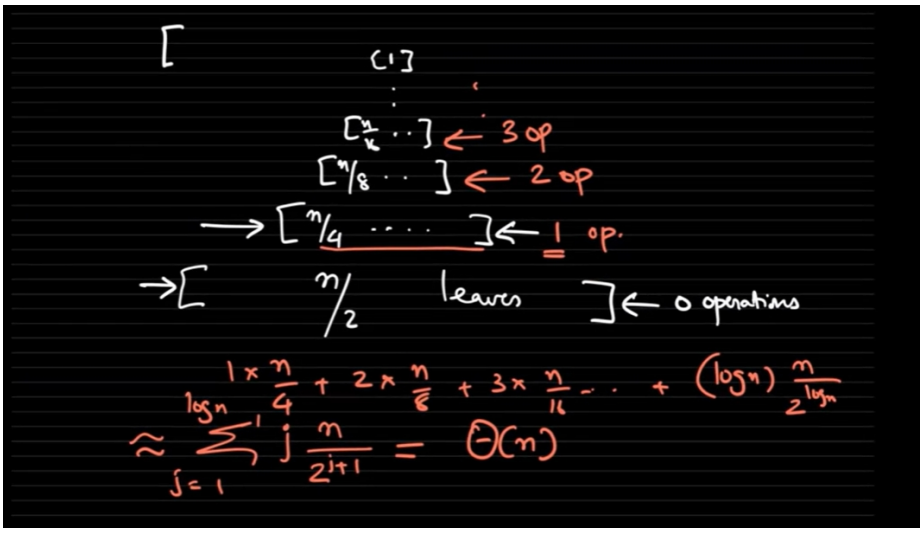
Now we are talking about priority queue.
Priority queue is a data structure that will allow us to insert elements and delete the smallest one.
The main idea of priority queue is to keep the smallest element at the root of the heap.
Priority queue has a priority setting.

We can bubble down elements that are in the wrong position and then get a min heap.
This is the main idea of heapify.
The running time of heapify is $O(n \log_2^n)$.

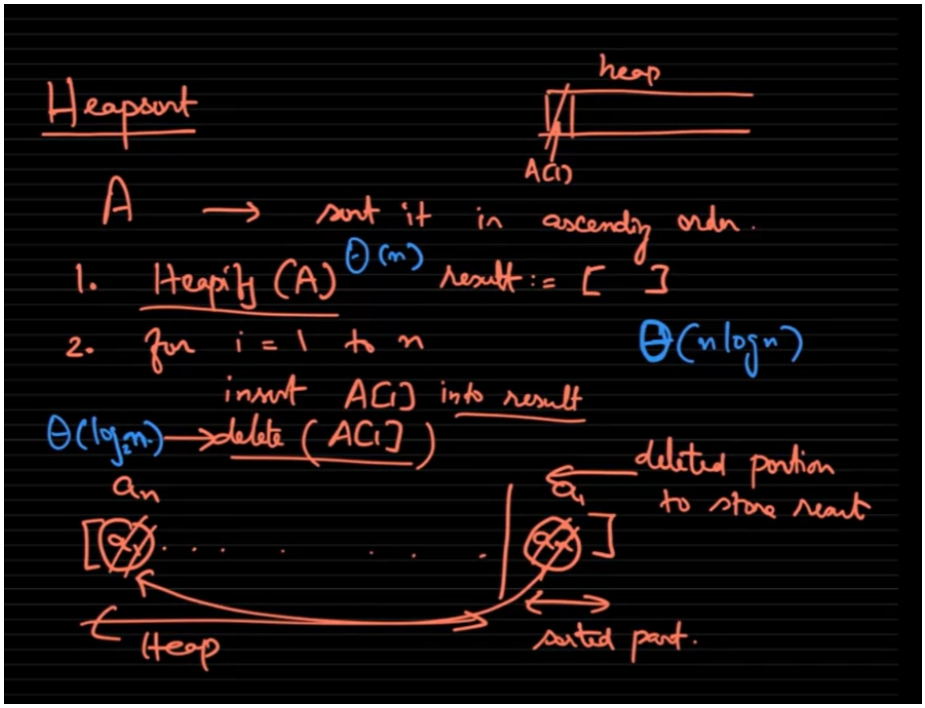




The overall running time of heapify is linear to the size of the array. That is to say, the running time is $\theta(n)$.


[
[1]
[n/k ..] ← 3 op
[n/8 ..] ← 2 op
→ [n/4 ....] ← 1 op.
→ [n/2 leaves] ← 0 operations
1 × n/4 + 2 × n/8 + 3 × n/16 ... + (log n) n/2^log n
≈ Σ_{j=1}^{log n} j n/2^{j+1} = Θ(n)

5.5.4 Heapsort



Question

Select all of the features below which accurately describe Heapsort.

- [x] $\Theta(n \log n)$ complexity.

  Correct

- [x] Easily parallelized.

  This should not be selected
  Unlike Mergesort, Heapsort's steps do not operate on totally separate sections of the array, and thus cannot be easily performed in parallel.

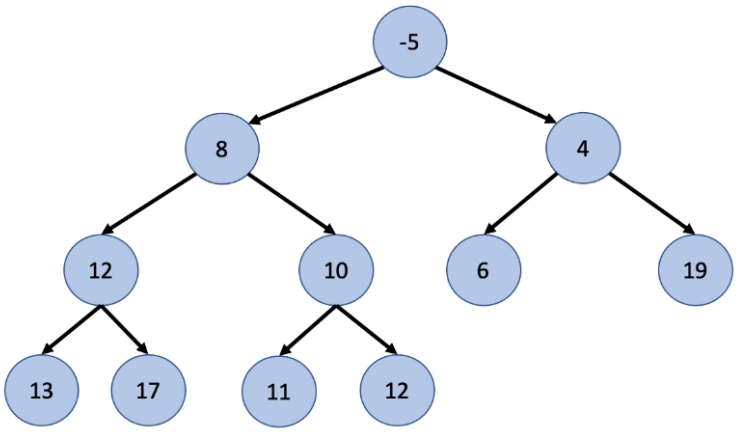
- [x] No need to allocate a new array to store the sorted elements ("in-place").

  Correct
  This is an advantage over Mergesort.

5.6 Quiz: Bubble-Up/Down, Insertion and Deletion

5.6.1 Question 1

1. Consider the following min-heap, in tree form: 6 / 6 points



Consider the following min-heap, in tree form:
Now suppose we perform a heap-insert to add a new element 1 to the heap.Select all the correct facts about this insertion process from the list below.

The first step is to place the new element at the beginning of the array, making it the root of the tree.

The first step is to place the new element at the end of the array, making it a child of the element.
Correct: After taking the first step, in which the new element is placed in the array, the heap property continues to hold.

In order to fix a heap property violation encountered during insertion, we perform a bubble-up operation, which swaps the new element and its parent

element.
Correct: In order to fix a heap property violation encountered during insertion, we perform a bubble-down operation which swaps the new element with one of its child elements.

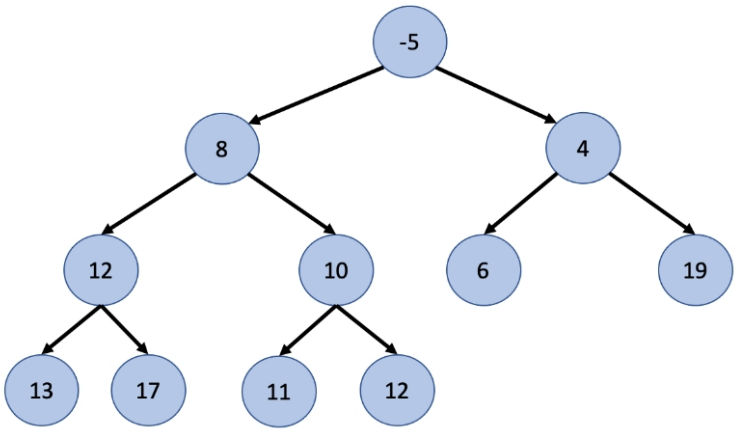
When we insert 1 into the above min-heap, the first bubble-up step swaps element 1 with its parent 6. This fixes all the min-heap property violations.

When we insert 1 into the above min-heap, a second bubble-up step occurs which swaps element 1 with 4. This fixes all the min-heap property violations.
Correct: This is correct.

In the worst case, bubbling up will require $\theta(\log_2^n)$ swaps, moving the inserted element all the way from a leaf position in the heap to the root.
Correct: This is correct: note that the length of the longest path from any leaf of a heap to the root is at most $\log_2^n$.

5.6.2 Question 2

1. Consider the following min-heap, in tree form: 6 / 6 points



Consider again the min-heap example:
Suppose we heap-delete the element 8.

Select all correct facts about this deletion process from the list below.

The first step replaces the element to be deleted (8) with the very last element in the heap's array (12).

The first step replaces the element to be deleted (8) with the very first element in the heap's array (-5).

After the first step, in which we replace the element we are deleting (8), we are guaranteed that we have created at most one min-heap property violation. This should be selected as well.

To fix the heap property violation caused by the first step, we perform a bubble-up or bubble-down operation depending on whether the swapped-in element is larger than its children or smaller than its parent.

Deletion of an element in a heap with n elements requires at $O(\log_2^n)$ comparison and swap operations.

5.7 Quiz: Heapify, Priority Queue and Heapsort

5.7.1 Question 1

Suppose we wished to heapify an array into a minheap using the following algorithm (assume $a[0]$ is not used and assume that bubble up routine is implemented).

```
def heapify (a):
    n = len (a)
    for i in range (1, n):
        bubble_up (a, i)
    return a
```

The worst case complexity of this procedure will be $\theta(n \log_2^n)$.

The procedure above is computationally less efficient than the one presented in the lecture because $n/2$ of the elements in the array may potentially need to bubble up all the way to the root.
Correct: Indeed: each of these $n/2$ elements may incur a cost proportional to $\log(\frac{n}{2})$ since they may each bubble up from the leaf to the root of he heap.

Sorting the array using insertion sort will achieve the same complexity as the heapify procedure above.

Sorting the array using mergesort will achieve the same complexity as the heapify procedure above.

The worst case of the procedure above is realized if we tried to heapify an array that is already sorted in descending order.

The worst case of the procedure above is realized if we tried to heapify an array that is already sorted in ascending order.

5.7.2 Question 2

Select all the facts that are true about the heapsort procedure in comparison with the insertion and mergesort procedures we have studied thus far. Assume that all heaps are minheaps unless otherwise mentioned in the option.

Heapsort requires extra storage to store the heap unlike insertion sort that can sort an array in place.

Heapsort is asymptotically faster than mergesort.

If an array is already sorted in ascending order and the heapify procedure is run on it, then it does not modify the array in any way.

If a min-heap is in fact already sorted in ascending order, deleting the minimum element operation runs in constant time.

6 Hashtables

The basic idea of hashtables is to implement a mapping from keys to values.

What's important of a hashtable is not key or value, but a special function called hash function.

With the hash function implemented, the key will be converted to a hash value.

The hash value will be used as an index to store the value. That is to say, the key will be converted to an index. However, since the hash value is based on the key itself, different keys may have the same hashed values, which means the same index may be mapped with several keys. How do we deal with this?

6.1 Chaining for Collision Resolution

The chaining of a hashtable is to store all the values that have the same index in a linked list. This list is empty initially.

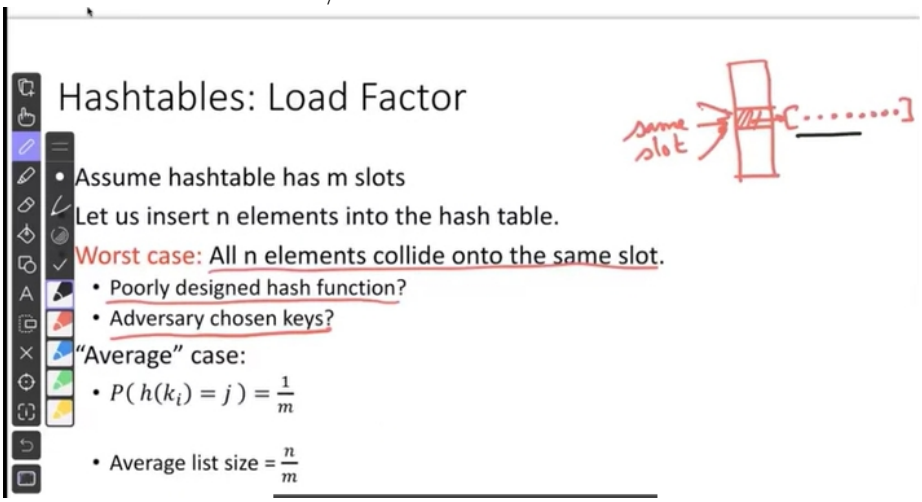
When we insert a new key-value pair, we will first calculate the hash value of the key. Then we will check if the index is empty. If it is, we will simply insert the key-value pair. If not, we will insert the new pair to the linked list.

If we want to delete, we will first calculate the hash value of the key. Then we will check if the index is empty. If it is, we will simply delete

the key-value pair. If not, we will delete the key-value pair from the linked list.

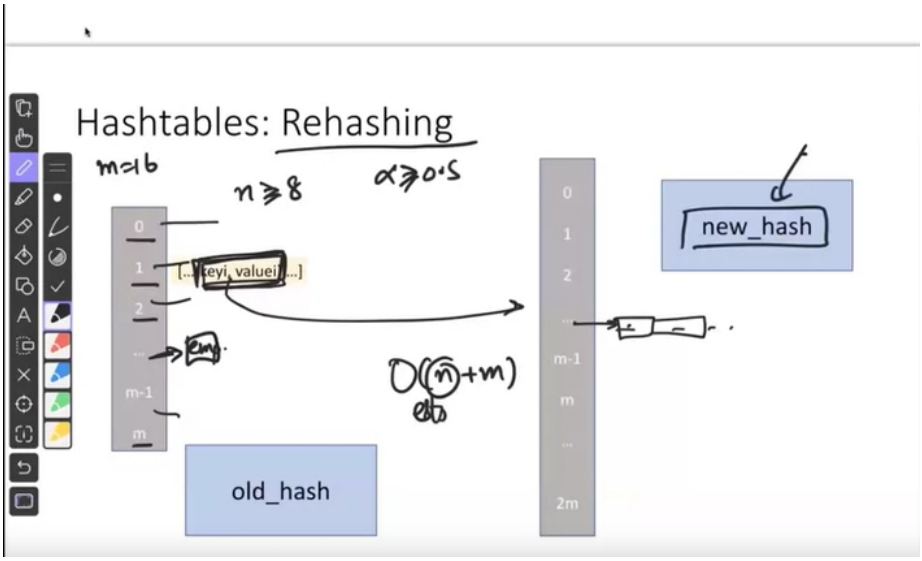
6.2 Load Factor

The load factor is the ratio of the number of key-value pairs to the number of indices. Suppose we have m slots(indices) and n key-value pairs. The load factor is n/m.



6.3 Rehashing

Rehashing is the process of creating a new hashtable with a larger number of slots. Since we need to go through all the key-value pairs (n pairs) and all the slots/indices (m slots) in the original hashtable in order to create the new one, The running time of rehashing is $O(n+m)$.



6.4 Quiz: Hashtables

6.4.1 Question 1

Select all true facts about Hashtables which use unordered-list chaining to resolve collisions.

Given a Hashtable with m slots and containing n elements, the average number of elements chained together on a single slot is $m * \log_2^n$.

The Find operation, which returns a value for a key if it is present in the Hashtable, has a worst-case complexity of $O(n)$ for a table containing n entries.

The cost of the hash function depends on the number of elements currently stored in the table.

Expanding the number of slots in a Hashtable requires re-applying a hash

function to every element already stored.

Given a Hashtable with m slots and containing n elements, the average number of elements chained together on a single slot is n/m.

6.4.2 Question 2

2. 1 / 1 point

Slot #			
0	Apples → 51	Plums → 24	Avocados → 29
1	Grapes → 35		
2			
3			
4	Yams → 120	Oranges → 50	

Consider the above Hashtable, which stores numbers for different produce items (maybe the amount in stock at a store) according to a hash function called HASH.

For the entry Apples →51, the key is Apples and the value is 51. The entry for Apples is at the front of a chained list including Plums and Avocados, which all share Slot 0.

The HASH function picks a slot for the entry by taking the ASCII value of the first letter of the key, mod 5. So for Apples, it is 65 mod 5 = 0 You won't need to calculate these for this problem.

Suppose it costs 1 time unit to run HASH, and 1 time unit to check whether an entry is already present in each place in the chained lists. It also takes 1 time unit to write a new value to a particular place in a chained list. Assume

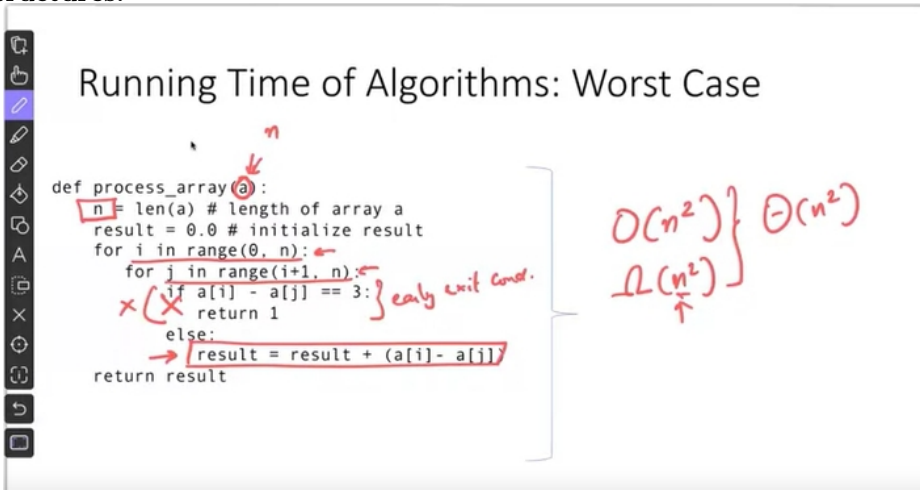
that newly inserted items must go at the end of the chained list in their slot.

How many time units does it cost to insert a new entry Limes $\rightarrow$ 40 into the Hashtable? When HASH is called on Limes, it gives 76 mod 5 = 1.

Answer:3

7 Introduction to Randomization, Average Case Complexity Analysis and Recurrences

The topic of this section is Randomization in Algorithm and Data Structures.



7.1 Algorithm that Use Randomness

The main idea of randomization is to use randomness in the algorithm.

The main advantage of randomization is that it can reduce the run-

ning time of the algorithm, so that we will focus on the average case of the running time rather than the worst case.

When looking at the running time of an algorithm, we typically consider the worst case scenario.

While in the case of randomization, the probability of worst case is relatively low because Of the nature of randomness. Therefore, it is not efficient to always consider the worst case, since it may not happen, and the average case of running time may be much lower and more realistic.

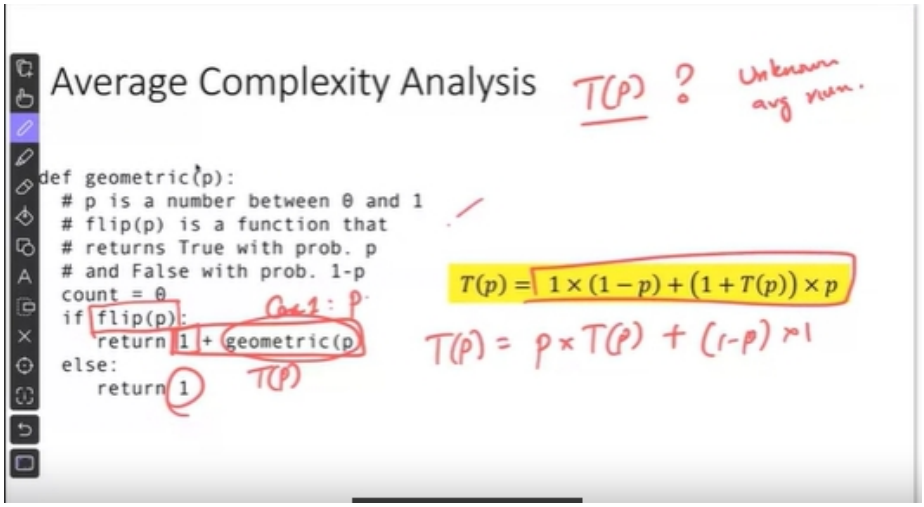


Figure 3: When running into case 1, a.k.a. $flip(p) == True$, the running time will be $p*(1+T(p))$, and in the other case, when $flip(p) == False$, the running time will be $1 * (1 - p)$.

Let's take a look at this $geometric(p)$ **function.**

We will introduce a new concept $T(p)$**, which is the expected value of**

the geometric function, a unknown average case running time.

The unknown running time $T(p) = p * (1 + T(p)) + (1 - p) * 1$, which can be simplified to $T(p) = 1/(1 - p)$.

7.2 Analysis of Algorithms: Recurrences

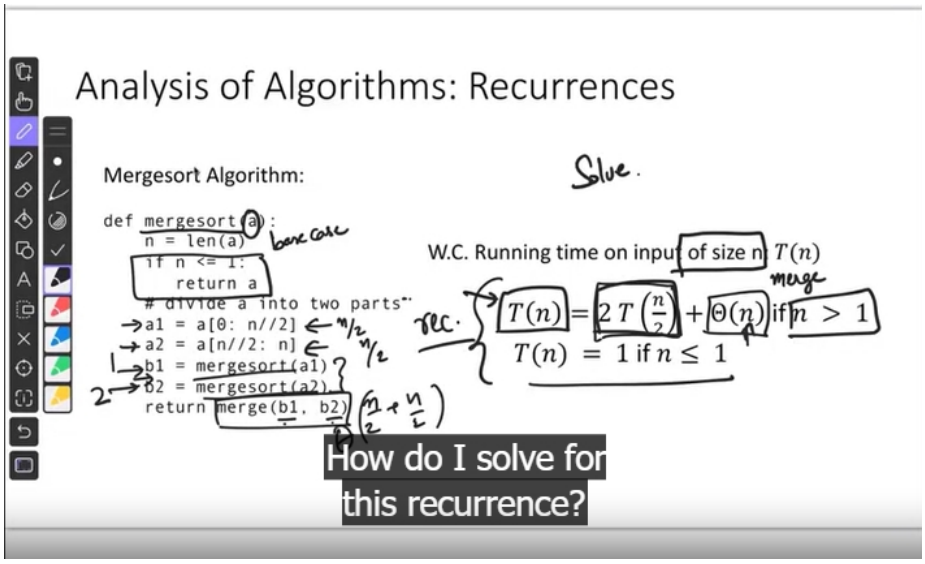


Figure 4: The $T(n)$ equation is a recursive function in this Mergesort function. If $n <= 1$, the function returns 1, where running time is 1. Otherwise, the function will be $2 * T(n/2) + n$ based on the function details.

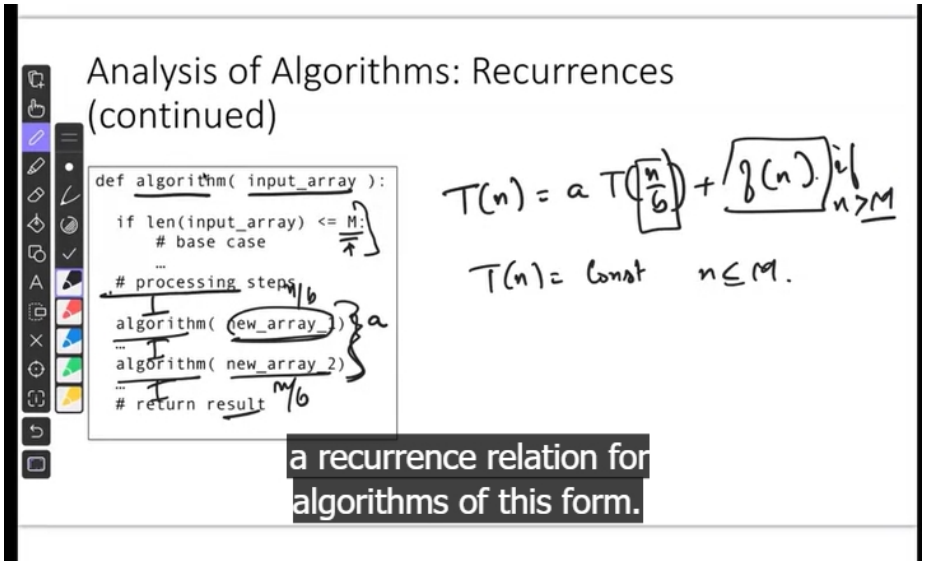


Figure 5: Here is a general form of a recursive function.

$T(n) = a * T(n/b) + \theta(n), if n > m$

$T(n) = Constant, if n <= m$

Here is a general understanding of recurrences.

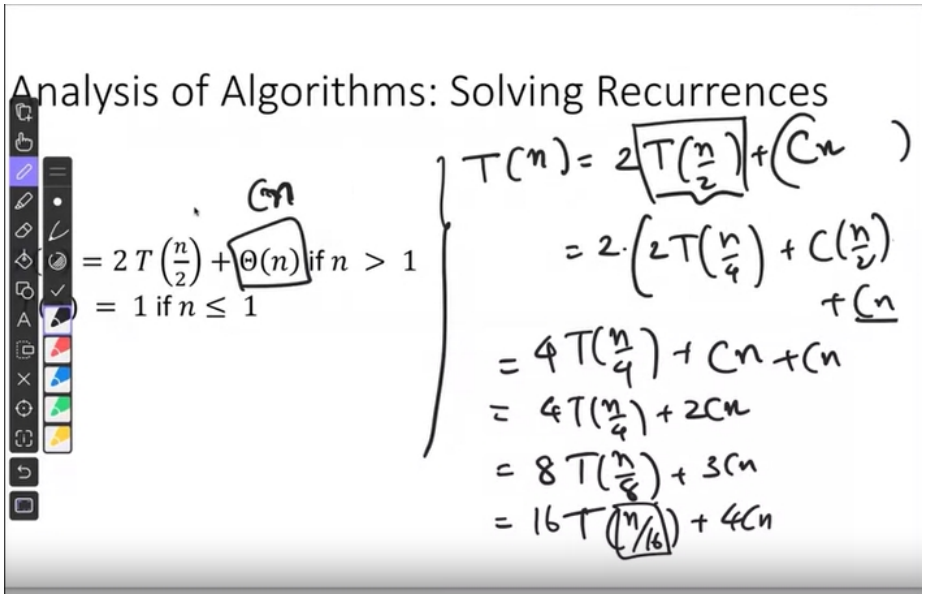


Figure 6: We can assign $C(n) == \theta(n)$ at first.

Then, we assign $T(n)$ to the original function with an expanded form.

For example, by multiplying 2, $T(n) = 2 * T(n/2) + C(n) + D$ will be converted to $T(n) = 2(2T(n/4) + C(n/2) + D) + C(n) + D$.

This expansion the function will keep going on by 4, 8 ,16 etc., until it reaches the base case, as displayed in the firgure.

The running time of this function is $\theta(n^{\log_b^a})$.

If we want to solve for the recurrence relations, we can use the expansion method.

The main idea is to expand the function until we reach the base case.

8 Partition and Quicksort Algorithm

8.1 Basic Idea and Pseudocode

Quicksort is an efficient in-place sorting algorithm that uses the divide and conquer strategy.
Basically it divide the array into two parts, one with the elements smaller than the pivot and the other bigger.
At first glance, quicksort sounds a lot like mergesort, but the main difference is that in mergesort, the major time costing part is at the end of the function when merging is taking place; while in quicksort, we will conduct a lot of work by partitioning the array first.

What is the partition function?
First we are going to choose a pivot element from within the array and it doesn't matter which one we choose. Then we will compare all the rest of the elements with the pivot and divide them into smaller-than-pivot and larger-than-pivot part. At this stage, we are not necessarily sorting what is being compared.
Now we get a updated array where the pivot is rested somewhere in the middle range of the array, and it is the final position for the pivot in the fully sorted array later. All we need to do now is to sort the left part and the right part.

Here is the pesudocode for quicksort. $left$ means the index of the first element in array A and $right$ means the index of the last.

```
quicksort(A,left,right):
    #Base case
    if right-left <= 1:
        swap(A[left],A[right])
        return
    # First just choose a element as pivot.
    x == A[right]
    p = partition(A,x)
    # Then we call quicksort recursively.
    quicksort(A,left,p-1)
    quicksort(A,p+1,right)
```

8.2 Quiz on Quicksort

1. Consider the array : [1, 2, 7, 5, -4, 3, 4] being partitioned into two parts. Select all the correct answers from the list below. **5 / 5 points**

- [x] By running partition once using 4 as the pivot, we conclude that there are exactly 4 elements in the array that are less than or equal to the pivot (excluding the pivot itself).

  Correct
  Correct.

- [x] The quicksort algorithm will recursively be called on both partitions with the pivot element itself in its correct position in the sorted array.

  Correct
  Correct: that is the idea.

- [ ] If an array is partitioned with the least element as the pivot then both partitions will be equal in size.
- [x] In order to get two partitions whose sizes differ by at most one element, one must choose the median of the array as the pivot element.

  Correct
  Correct: since roughly half the array elements will be larger and half will be smaller.

- [ ] Partitioning an array based on a pivot element requires time that is quadratic in the size of the array, i.e. $\Theta(n^2)$ time.
- [x] If 4 is used as the pivot, the two partitions will have the elements { -4, 1,2,3} (in some order depending on exact partition scheme) and {5,7}.

  Correct
  Correct. The pivot element itself will be in the middle of the two partitions.

- [ ] Once a partition is performed with a pivot element, the array is split into two parts that are equal in size.

Figure 7: Quiz 1

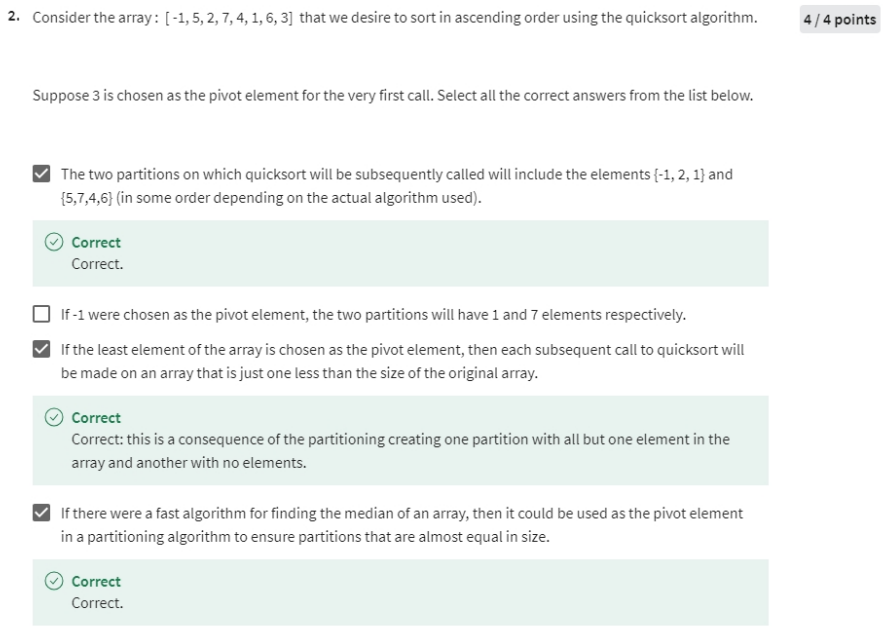


Figure 8: Quiz 2

9 Designing Partition Scheme and Correctness

First we will go through the Lomuto Partition Scheme.

We will introduce two intermediate variables, i and j, which are the index of the first element and the last element of the array.

Then we will choose the last element as the pivot.

The original array can be divided into four parts, $A[1, \ldots, i-1], A[i, \ldots, j-1], A[j, \ldots, n-1], A[n]$.

The end of the first region can be either i or $i-1$ as noted, which is not a big deal.

The main idea of the Lomuto Partition Scheme is to compare the

element at index j with the pivot.

If the element is smaller than the pivot, we will swap it with the element at index i and then increase i by 1.

That is to say, everything after j is categorized as "unprocessed".

Here is a step-by-step analysis of the partition scheme.

Let $i = 0, j = 1$, and the pivot is $x = A[n]$.

Under this situation, everything besides the pivot is unprocessed, located in Region 3.

$\forall Region1 < x$, $\forall Region2 >= x$,

if $A[j] < x$, we will swap $A[i]$ and $A[j]$, and then increase i as well as j by 1.

If $A[j] >= x$, we will do nothing but increasing j by 1.

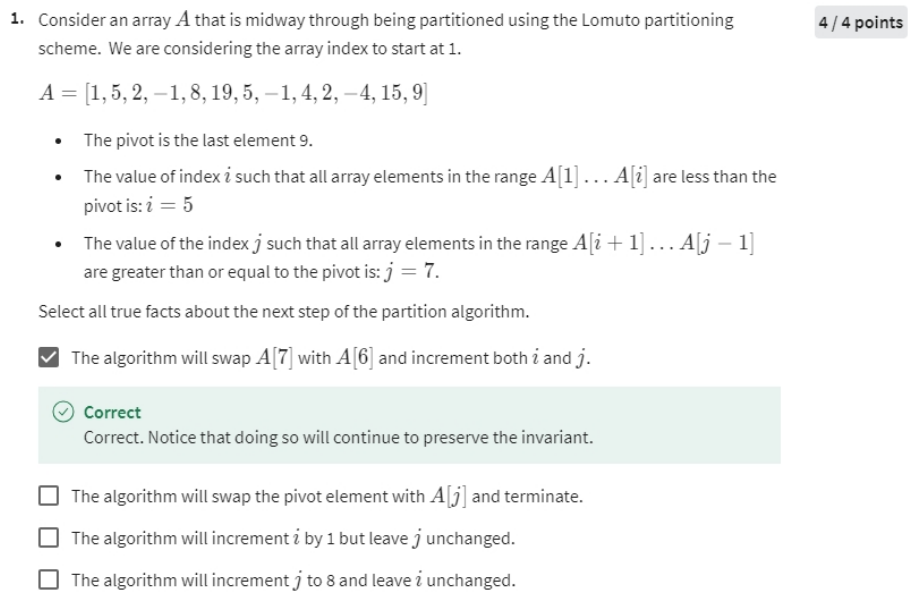
Here is the pseudocode for the Lomuto Partition Scheme.

```
partition(A,x):
    # Four regions: A[1,...,i-1], A[i,...,j-1], A[j,...,n-1], A[n]
    i = 0
    j = 1
    # Perform partitioning until the end of the array.
    while j <= n-1:
        if A[j] < x:
            swap(A[i],A[j])
            i += 1
            j += 1
        else:
            j += 1
    # Swap and place the pivot to its right position.
```

```
swap(A[i],A[n])
return i
```

The running time of Lomuto Partition Scheme is just n.

9.1 Lomutio Partition Algorithm Quiz



1. Consider an array A that is midway through being partitioned using the Lomuto partitioning scheme. We are considering the array index to start at 1. **4 / 4 points**

$A = [1, 5, 2, -1, 8, 19, 5, -1, 4, 2, -4, 15, 9]$

- The pivot is the last element 9.
- The value of index i such that all array elements in the range $A[1] \ldots A[i]$ are less than the pivot is: $i = 5$
- The value of the index j such that all array elements in the range $A[i+1] \ldots A[j-1]$ are greater than or equal to the pivot is: $j = 7$.

Select all true facts about the next step of the partition algorithm.

☑ The algorithm will swap $A[7]$ with $A[6]$ and increment both i and j.

Correct
Correct. Notice that doing so will continue to preserve the invariant.

☐ The algorithm will swap the pivot element with $A[j]$ and terminate.

☐ The algorithm will increment i by 1 but leave j unchanged.

☐ The algorithm will increment j to 8 and leave i unchanged.

Figure 9: Quiz 1

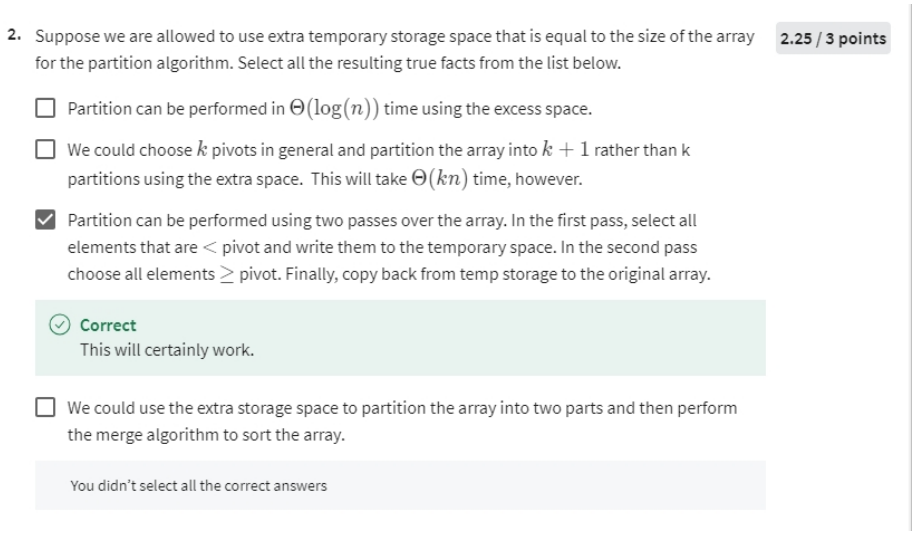
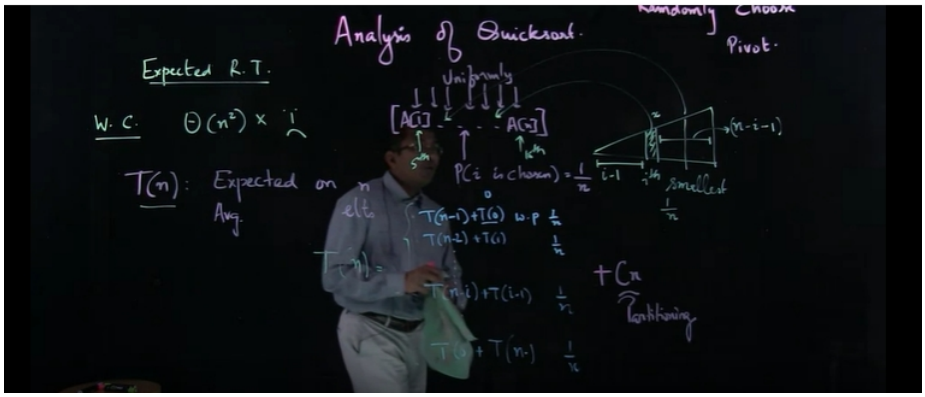


Figure 10: Quiz 2

10 Analysis of Quicksort Alogorithm

The best case analysis of quicksort is that what we call a balanced partition.

Although it rarely happens, the running time when occuring is $\theta(nlog^n)$.



Now Let's take a look at the worst case analysis of quicksort algorithm.

The worst case of quicksort is when the pivot is the smallest or the largest element of the array. Therefore, either side of the pivot will be empty.

The running time of the worst case is $\theta(n^2)$, which equals to the running time of insertion sort.

So what we should do is to avoid the worst case by choosing a pivot randomly.

The expected running time of quicksort is $\theta(nlog^n)$.

Let's assume $T(n)$ is the expected (or average) running time of quicksort.

Here is the equation of $T(n)$.

$T(n) = 1/n[\sum_{i=1}^{n-1} T(i) + \sum_{i=1}^{n-1} T(n-i)] + n$

And it equals to

$T(n) = 2/n[\sum_{i=1}^{n-1} T(i)] + n$

The above is the compressed form of the equation. We can expand

it like the following;

$T(n) = 2/n[T(1) + T(2) + \cdots + T(n-1)] + n$**.**

If we multiply n **on both sides, we get**

$n * T(n) = 2[T(1) + T(2) + \cdots + T(n-1)] + n^2$**.**

If we replace n **with** $n-1$**, we will get**

$(n-1) * T(n-1) = 2[T(1) + T(2) + \cdots + T(n-2)] + (n-1)^2$**.**

Then we can subtract $nT(n-1)$ **from** $nT(n)$ **and get**

$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n - n^2 + n - 1$**.**

$T(n) = T(n-1) + 1/nT(n-1) + 2 - 1/n$**.**

Now once again, we will try to expand the above equation.

For convenience, we will assume $2 - 1/n = \theta(1)$**.**

Then we will get $T(n) = T(n-1) + \theta(1)$

$T(n) = T(n-1) + \theta(1)$

$T(n) = T(n-2) + 2\theta(1)$

$\cdots T(n) = (n+1)/(n-j+1)T(n-j) + \sum_{i=1}^{j-1}(n+1)/(n-i) * 2\theta(1) + \theta(1)$**.**

When $j == n$**, the equation will be**

$T(n) = \sum_{i=1}^{j-1}(n+1)/(n-i) * 2\theta(1) + \theta(1)$**.**

$T(n) = [(n+1)/n + (n+1)/(n-1) + \cdots + (n+1)/1] * 2\theta(1) + \theta(1)$

$T(n) = 2\theta(1)[n+1][1 + 1/2 + 1/3 + \cdots + 1/n] + \theta(1)$

The above equation is the harmonic series, in which the $1 + 1/2 + 1/3 + \cdots + 1/n$ **part is called the nth harmoic number, and it equals to** $\theta(\log^n)$**.**

Therefore, $T(n) = 2\theta(1)\theta(\log^n)$**,**

Then we can get the final result of $T(n) = nlog^n$**, a.k.a** $\theta(n \log^n)$**. Now we can say, if we choose a pivot randomly in Quicksort, the average running time will be** $\theta(n \log^n)$**.**

10.1 Quiz on Quicksort Analysis

1. Consider a run of Quicksort over array A of size 8: **3 / 3 points**

$A = [4, 5, 7, 1, 3, 2, 6, 8]$

Suppose we choose a pivot element uniformly at random from the elements of A. Select all the correct answers from the list below.

- [ ] With probability $1/8$, the resulting two partitions will have 3 and 4 elements.
- [x] The probability that the element 8 or 1 is chosen is $1/4$.

Correct
Correct.

- [x] With probability $1/4$, the resulting two partitions will have 3 and 4 elements.

Correct
This is correrct since the choices that will ensure this happens are to choose either 4 or 5 as the pivot.

Figure 11: Quiz 1

2. Consider an array of size n with distinct elements to be sorted using Quicksort with randomized pivoting. **3 / 3 points**

Let us say that a partition is completely unbalanced if the two partitions have 0 and $n-1$ elements each (note that the pivot is placed in its final sorted location).

- [x] The probability of a completely unbalanced partition is $2/n$.

  Correct
  Correct: since choosing the least *or* largest element can cause this to happen.

- [ ] The probability of a completely unbalanced partition is $1/n$.

- [x] The probability that all pivots chosen by the Quicksort run will lead to completely unbalanced partitions is $\frac{2^{n-1}}{n!}$. Assume base case is encountered for arrays of size 1.

  Correct
  This is correct. After j partitions, the array size is $n-j$ and the probability of an unbalanced partition will be $\frac{2}{n-j}$. This goes on until $n-j=1$ when the base case is reached. Since each of these are independent, the expression above is obtained.

Figure 12: Quiz 2

3. Select all true facts about Quicksort. **5 / 5 points**

- [x] If it were possible to find the median element of an array in $O(n)$ time worst case, then quicksort will run in $\Theta(n \log(n))$ time in the worst case.

  Correct
  Correct, since it is possible to find the median and partition with the median as the pivot in $O(n)$ time. This will cause roughly balanced partitions and the running time will become $\Theta(n \log(n))$ even in the worst case.

- [x] Suppose we chose the pivot by randomly choosing three elements and taking the median of the chosen three as the pivot, the running time of quicksort is now $\Theta(n \log(n))$ in the **expected case.**

  Correct
  True, since this is more or less equivalent to a random choice of pivot and the analysis carries over with a few modifications.

- [ ] Choosing the mean of the smallest and largest element of an array of n elements will always lead to perfectly balanced paritition with at least $$\lfloor n/2 \rfloor$ elements in each partition.

- [ ] Suppose we chose the pivot by randomly choosing three elements and taking the median of the chosen three as the pivot, the running time of quicksort is now $\Theta(n \log(n))$ in the **worst case.**

Figure 13: Quiz 3

11 Quickselect Algorithm

Suppose you are running **Quickselect**$(12, A)$, meaning that you are seeking the 12 th-smallest element from array A of size 20.

As the first step, you choose a pivot p from A and partition, which produces left-hand $A_{<p}$ of size 4 and right-hand $A_{\geq p}$ of size 15.

What is the next step?

Figure 14: The next step will be to perform a *quickselect*$(12-5, A >= p)$ operation, in order to find the $7th$ smallest element on the right side of the pivot, since the $12th$ smallest element is already larger than the 4 elements in the left side of the pivot.

The main idea of Quickselect is to find the kth smallest element in an array.
We can use Quicksort to select better than the brute force method, which is just trying to sort the array first and then find the target by comparing every element.
The running time of the brute force method is $\theta(n \log^n)$, while the running time of Quickselect is $\theta(n)$.
Here is the logic of using quicksort for quckselect function.
At first, we will choose a pivot and partition the array.
Then we will compare the pivot with the target. Let's say the pivot is j in array A with n elements in total.
If $j == k$, we will return the pivot.

If $j > k$, we will recursively call quickselect on the left part of the pivot, a.k.a. $quickselect(A[1], \ldots, A[j-1])$.

If $j < k$, we will recursively call quickselect on the right part of the pivot, a.k.a. $quickselect(A[j+1], \ldots, A[n])$.

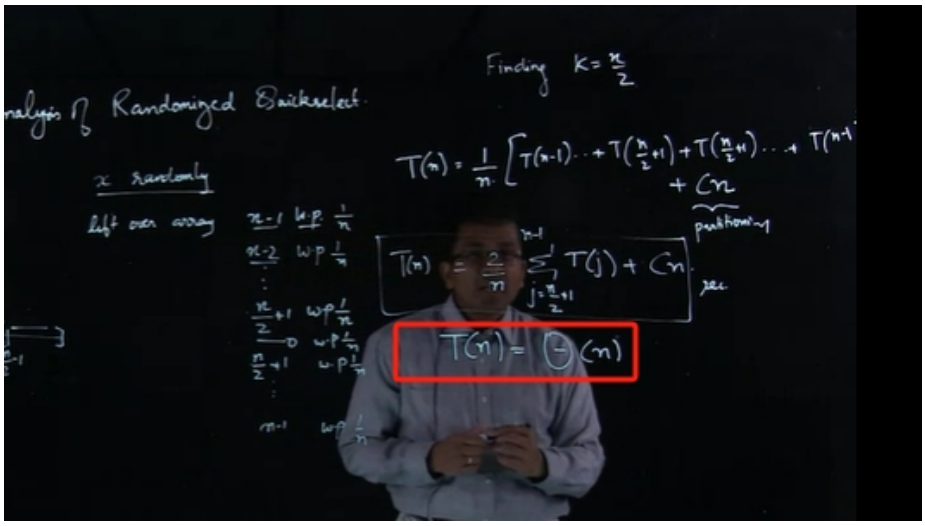


Figure 15: The running time of $T(n)$ is actually $\theta(n)$.

Now let's go through a real example of quickselect.

Let's say we have an array $A = [3, 2, 1, 5, 4, 6, 7, 8, 9]$.

We want to find the 4th smallest element in the array.

Let's say we choose the pivot randomly and it's 5.

After partitioning, the array will be $A = [3, 2, 1, 4, 5, 6, 7, 8, 9]$.

The pivot is 5, which is the 5th smallest element. Since $pivot > k$, we need to focus on the left side of the pivot, which is $[3, 2, 1, 4]$.

Now we need to find the 4th smallest element in the left side of the

pivot.

Let's say we choose the pivot randomly and it's 2.

After partitioning, the array will be $A = [1, 2, 3, 4]$**.**

The pivot is 2, which is the 2nd smallest element. Since $pivot < k$**, we need to focus on the right side of the pivot, which is** $[3, 4]$**.**

Since the 4th smallest element is already larger than the 2 elements in the left side of the pivot, we can just find the $4 - 2 = 2th$ **smallest elment in** $[3, 4]$**, which is 4.**

11.1 Quiz on Quickselect

1. Consider the input array A that has just been partitioned around the pivot element 6:

$A = [-7, 3, 5, -4, 6, 8, 12, 9, 15, 7]$

We wish to run Quickselect to find the 4th-smallest element of A. Select all the correct answers from the list below.

- [x] Quickselect will recursively find the 4th-smallest element on the left-hand side of the partition.

  ✓ **Correct**
  Correct since the pivot element 6 is the 5th smallest element of the array

- [ ] Quickselect will terminate at this step and return the pivot (6) as the 4th-smallest element.
- [x] The partition has revealed that pivot element 6 is the 5th-smallest element of A.

  ✓ **Correct**

- [ ] Quickselect will recursively find the smallest element on the right-hand side of the partition.

Figure 16: Quiz 1

2. Consider the input array A that has just been partitioned around the pivot element 6:

$A = [-7, 3, 5, -4, 6, 8, 12, 9, 15, 7]$

We wish to run Quickselect to find the 6th-smallest element of this array. Select all the correct answers from the list below.

- [x] Quickselect will eventually conclude that the element 7 is the 6th-smallest.

  ✓ **Correct**
  Correct.

- [x] Quickselect will now recursively seek the smallest element on the right-hand side of the partition.

  ✓ **Correct**
  Correct since the pivot is revealed as the 5th smallest element.

- [ ] Quickselect will now recursively seek the smallest element on the left-hand side of the partition.
- [ ] Quickselect will return the pivot (6) as the 6th-smallest element.

Figure 17: Quiz 2

3. Select all the correct answers about the running time complexity of Quickselect from the list below.

- [x] In the worst case, Quickselect will run in $\Theta(n^2)$ time, which happens when we choose the smallest or largest element as the pivot.

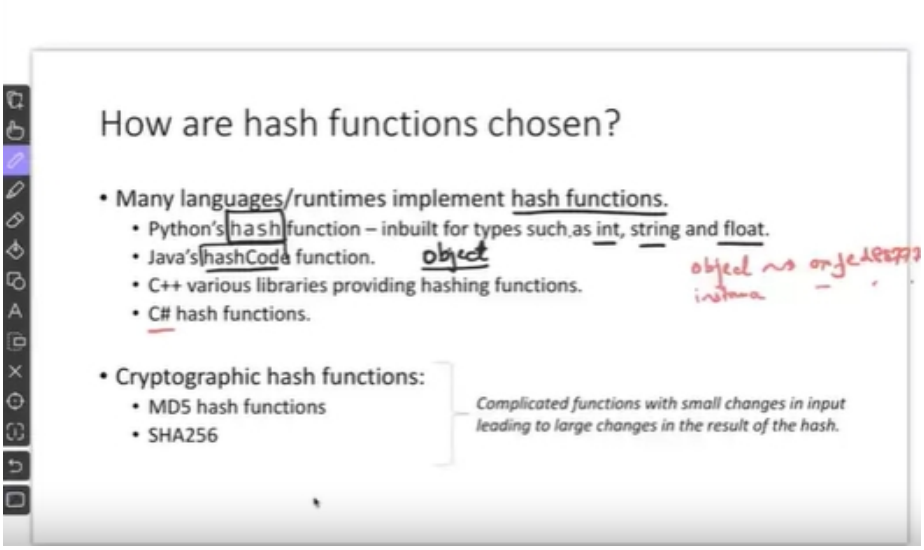
  ✓ **Correct**
  Correct: this is the worst possible case for quick select.

- [x] In the average case where the pivot is chosen at random, Quickselect is expected to run in $\Theta(n)$ time.

  ✓ **Correct**

- [ ] Even though Quickselect runs much faster on average than its worst case time, the worst case complexity is quite frequently encountered when we choose the pivot at random.

12 Hash Functions and Universal Hashing



The main idea of a hash function is to convert a key to an index. It decides where the key-value pair will be stored in the hashtable. Ideally a hash function should be a one-to-one mapping, which means no collisions.

Many programming languages have built-in hash functions, selected based on run time. While cryptographic hash functions are complicated for more advanced use case. Some popular use case includes Merkle Tree in the Blockchain technology.

12.1 How to Design Hash Functions?

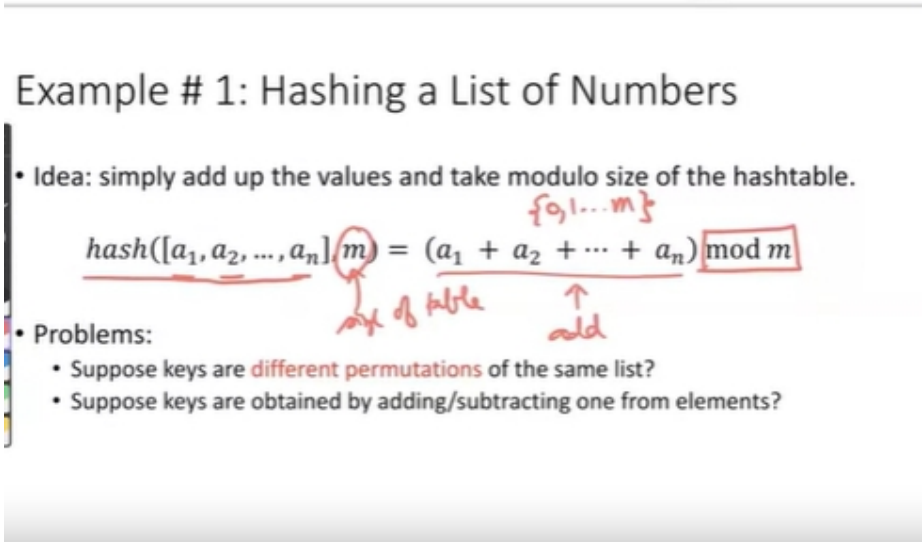


Figure 20: Example 1 is a hash function that is designed for hasing a list of numbers. the lecturer is using 5 as the modulo number and the result turns out to be pretty bad, since the hashed values are easily hashed into the same table position.

The main idea of designing a hash function is to make sure that the hash value is uniformly distributed.
The hash value should be as random as possible.

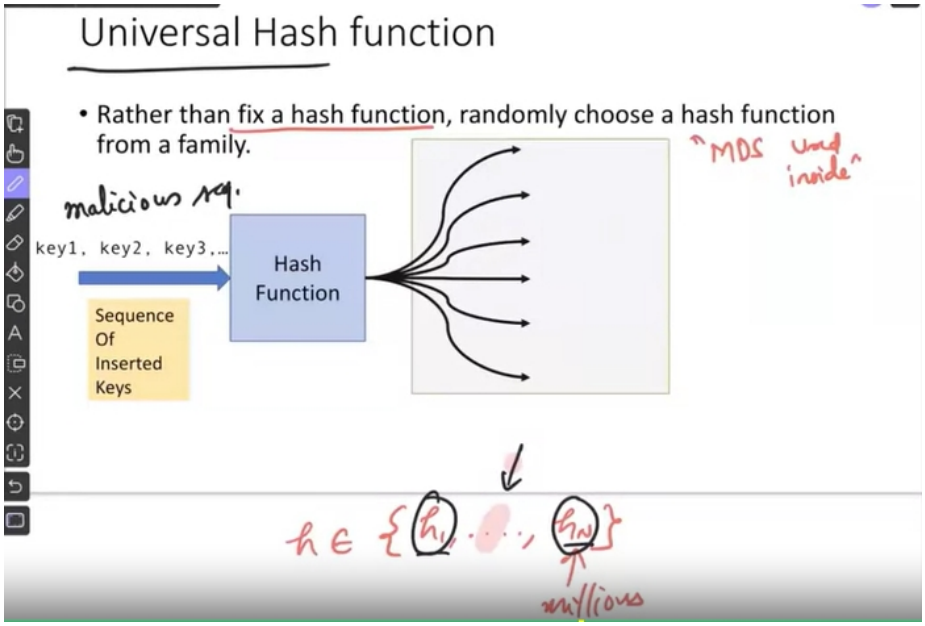
For some hash functions like the one in Example 1, the hash value is not uniformly distributed. To conquer this problem, one simple solution is to use a polynominally rolling hash function.

The basic idea is to multiply the hash value by a prime number and then add the new element.

Here is a simple equation for that.

$\sum_{i=0}^{k} w_i * p^i$**, where** w **is a 64 bit number and** p **is a prime number.**

12.2 Problems with Fixed Hash Functions



The main problem with fixed hash functions is that they are predictable.

If the attacker knows the hash function, they can easily predict the hash value.

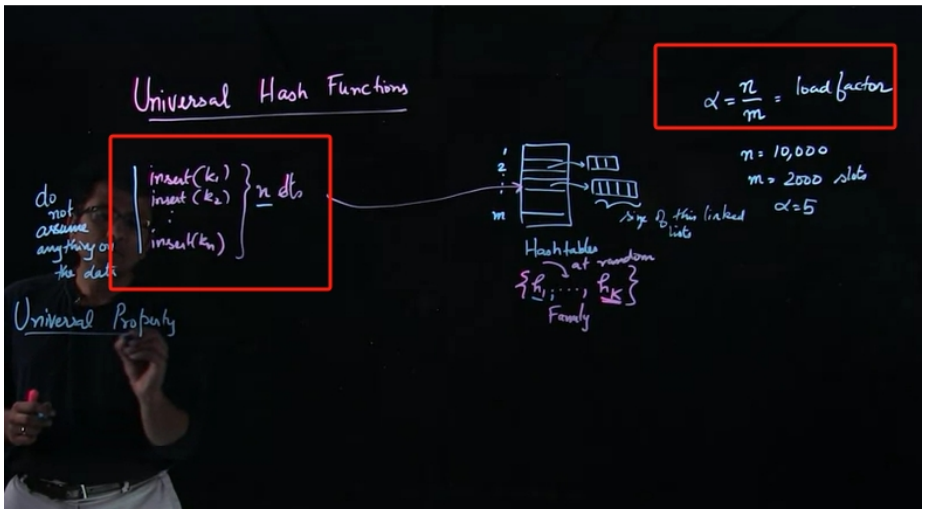
Therefore, we need to design a hash function that is unpredictable.

The main idea of universal hashing is to randomly choose a hash func-

tion from a set of hash functions.

The set of hash functions should be large enough to make it hard for the attacker to predict.

12.3 Universal Hash Functions and Analysis



In the universal hashing scheme, we will randomly choose a hash function from a set of hash functions.

Let's assume we have a set of n^{th} elements k waiting to be inserted into the hashtable.

The number of hashtable slots is m. $\alpha = n/m$ is called the load factor.

These hash functions in the set must obey the following rule.

$\forall k_1, k_2 \in U, x \neq y, Pr[h(k_1) == h(k_2)] \leq 1/m$**,**

So that the probability of collision is $1/m$.

First, we need to choose a prime number p that is larger than m.
Then we need to choose a random number a from 1 to $p-1$.
Assuming we have a set of hash functions $h_1, h_2, \ldots, h_k$ and keys $\in$ $1, 2, \ldots, n$.
Now we will consider the hash functions $h_1, h_2, \ldots, h_{p-1}$, and there will be $[(a_j) \pmod p] \pmod m$, where $j \in 1, 2, \ldots, p-1$.
p is the prime number, a is the random number, and m is the number of slots in the hashtable.

Now let's consider a specific example.
Assuming $n = 10, m = 7, p = 13$, then we have $p - 1 = 12$ as the total number of hash functions.
Then $h_5(j) = (5_j \pmod 1)3) \pmod 7$. Since $k_1 \neq k_2$, the probability of collision is $1/7$.
In order to make these hashing function universal, we need the hash of same elements to be different.
But what is the chance of collision? Or in other word, what is the probability that the hashed value of Different keys are the same?
We can test it with the following equation.
$Pr[h(k_1) == h(k_2)] = Pr[(a(k_1) \pmod p) \pmod m == (a(k_2) \pmod p) \pmod m]$**. Therefore;**
$[(ak_1 - ak_2) \pmod p] \pmod m = 0$**.**
That is to say, the value of $(ak_1 - ak_2) \pmod p$ should be integer multiple of m, such as $m, 2m, \ldots, km$.

Now we will use the concept of Modular Inverse in number theory to solve the equation.

```
Here is the theory:
If p is prime number,
and ax = b (mod p),
then x = a^-1 * b (mod p).

Therefore,
The modular inverse of a is a^{-1}.
```

Therefore $a \in ((k_1 - k_2)^{-1} * 0, (k_1 - k_2)^{-1} * m, \ldots, (k_1 - k_2)^{-1} * km)$**.**
a should not be 0, since $a = 0$ **will make the hash function useless. So there are** k^{th} **possibilities of a.**
The probability of collision is $k/(p-1) \approx (p/m) * (1/(p-1)) \approx 1/m$**.**

What will happen if you use a near universal hash function?
Assuming $x, y \in Keys$**.**
$\sum_{y \in hashtable} Pr(x, ycolliding) \leq 2/m \leq 2n/m$**, which is just** 2α**.** $(\alpha = n/m = loadfactor)$
That is to say, if we want to have a universal hash function, the average size of the list of the functions will be 2α**.**

13 Open Address Hashing

The first solution to resolve hash collisions is 'Chaining' as we discussed before.

Instead of storing single value, we store a list of key-value pairs, so that we can resolve collisions in the process.

The disadvantage of chaining lies in two points. First, it requires extra space to store the linked list, a.k.a there's memory concern; second, it produces inefficient use of hashtable(some lists are empty while others pretty big).

Now we will introduce another solution called 'Open Address Hashing'.

The main idea of Open Address Hashing is to store the key-value pairs in the hashtable itself.

If there is a collision, that is to say the original slot is occupied, we will find another empty slot to store the key-value pair.

13.1 Lookup and Insertion in Open Address Hashing

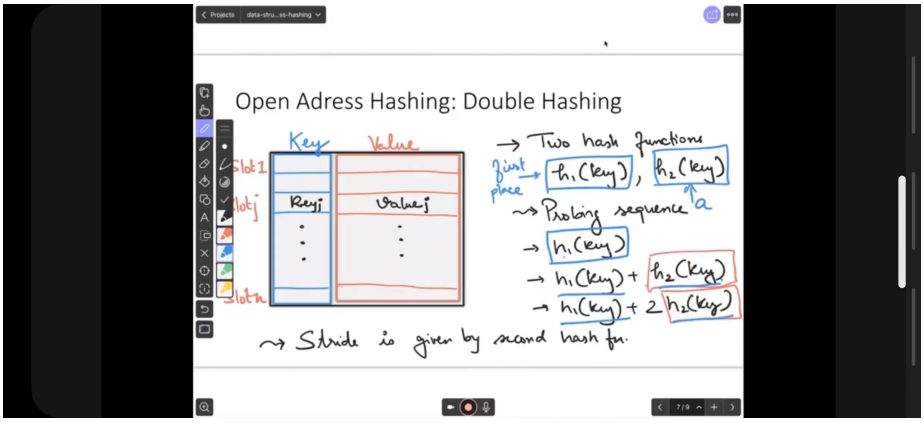
During lookup, we will first calculate the hash value of the key.

Then we will check if the slot, j for example, is empty. If not, we will need to check the alternative slot such as j+1.

If both j and j+1 is empty, that means the key we are looking for is not in the hashtable yet.

This process is called linear probing. Next we will look at quadratic probing, such as $j, j^2 + a, (j^2 + a)^2 + a$, etc.

13.2 Double Hashing



The main idea of double hashing is to use two hash functions.

The first hash function is used to calculate the slot, while the second hash function is used to calculate the step.

The main advantage of double hashing is that it can avoid clustering.

For instance, we have $h_1(key) and h_2(key)$ **from two differenct hash functions.**

The probing sequence will then be like $h_1(key), h_1(key)+h_2(key), h_1(key)+2h_2(key), \ldots$.

13.3 Deletion

The main idea of deletion is to mark the slot as deleted.

When we are looking for a key, we will treat the deleted slot as empty.

However since the slot is marked deleted, we will prevent from encountering a problem of early stop, that is to say, when looking up

with linear probing, we will not stop at the deleted slot.

14 Perfect Hashing and Cuckoo Hashing

14.1 Perfect Hashing

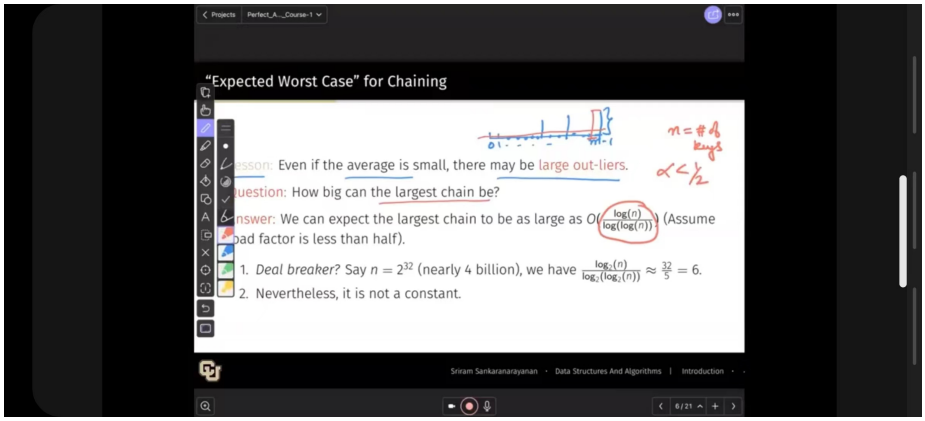
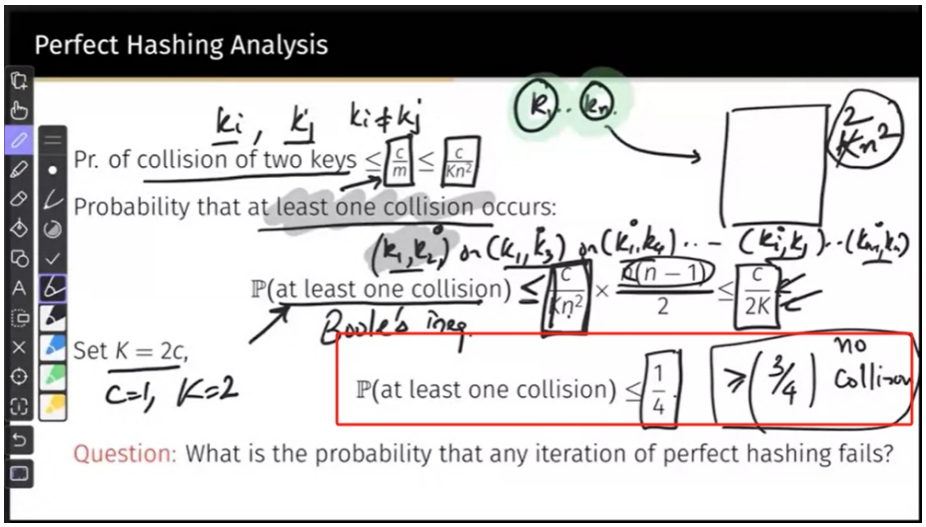


Figure 21: Expected worst case for Chaining.

In a chaining solution, even if the average time cost is small, there may be still large outliers. We can expect the largest chain is as large as $O(\log(n)/\log(\log(n)))$**.**

That is to say, we may find it reach the $\log(n)/\log(\log(n))$ **size of the chain alternatives since the original slot is occupied. This is actually pretty good algorithm already since the number is relatively small.**



The main idea of perfect hashing is to have no collision at all.

For an array that contains n keys, we randomly choose a hash function from a set of hash functions H, hence $h \in H$.

The hashtable will be of size $k * n^2$, in which k is a parameter to be determined.

Since the probability of collision of two keys $\leq c/m$, and in this case, $m = k * n^2$.

Therefore, in a array of n elements, the probablity of at least one collision happening is $(c/k * n^2) * (n * (n-1)/2) \leq c/2k$.

The main recipe of the above hashing approach is to use a huge hashtable, and the universal hashing scheme.

But can we get rid of the relies on the hashtable size?

The answer is yes, we can use a two-level hashing scheme.

The first level is to use a universal hash function to hash the keys into n slots.

The second level is to use a universal hash function to hash the keys in the n slots into $k * n^2$ slots.

The total number of slots will be $\leq \sum_{j=1}^{n} n_j^2$, n_j is the number of elements that collide with key k_j.

This sumation is actually $\leq$ some constant for each j. Therefore, the total size of the second level hashtable is $\leq c * n$, c as the constant for each j.

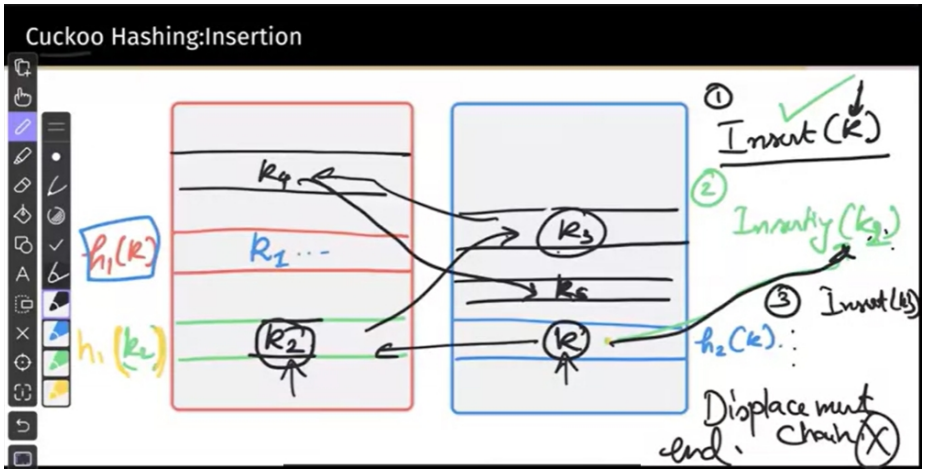
The main advantage of perfect hashing is that it can achieve $O(1)$ time complexity.

Along with all the benefits from perfect hashing, there are still some disadvantages.

First, all keys need to be known in advance.

Second, you cannot insert or delete keys.

14.2 Cuckoo Hashing



The main idea of Cuckoo Hashing is to use two hash functions in two hashtables.

The first hash function is used to calculate the slot, while the second hash function is used to calculate the alternative slot.

If both the first and the alternative slots are occupied, we will insert the key-value pair to the alternative slot anyway and kick out the one that's already been there. Then recursively insert the kicked-out key-value pair to another alternative slot.

If the alternative slot is empty, we will insert the key-value pair to that slot.

This is called a displacement chain. Once an empty slot is found, the chain will end.

There are two possible scenarios here. One is that the replacement chain just carries on and on; the other is that the chain will get into

a loop, a.k.a the new alternative slot is occupied by some key we just encountered, then the whole kicking out situation starts again.
How do we resolve this?
If there is too many displacement, say in an array of $n = 2^{10}$, we encounter more than 10 replacement, which is rare, we simply create a new hashtable with a larger size and rehash all the key-value pairs.

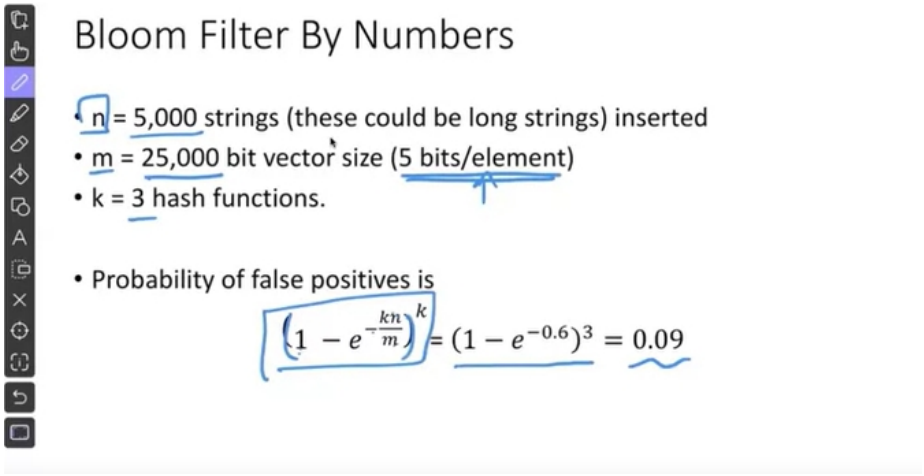
15 Bloom Filters and Analysis

15.1 The Probability of False Positive

This is a fast set data structure based on hashing. The main idea of Bloom Filters is to use multiple hash functions to check if a key is in the hashtable.
The main advantage of Bloom Filters is that it can save a lot of space, while the main disadvantage is that it may have false positives, due to the approximaty nature. How? Let's find out later.
Here is the basic idea of Bloom Filters. We will use k randomly choosen hash functions to hash the keys into m slots.
In each slot, we only has ones and zeros, one means the slot is set while zero means not.
Now we are going to hash the keys. For key x, we have hashed value $h_1(x)$, which is a number ranging from 0 to $m-1$. Then we will go ahead and using all the hash functions choosen and calculate the hashed values of x, and correspondingly set slots.
So, if we want to check whether x is in the hashtable, we will simply check every hashed value of x, a.k.a. $h_1(x), h_2(x), \ldots, h_k(x)$, and see if

all the slots are set to 1.

Now the ambiguity comes in. Even if all the slots are set to 1, we are still prone to the possibility of false positive, because other hashed values of certain keys may set the slots into 1, which has nothing to do with x.



What is the probability of false positive?

Let's say we have m slots and n keys.

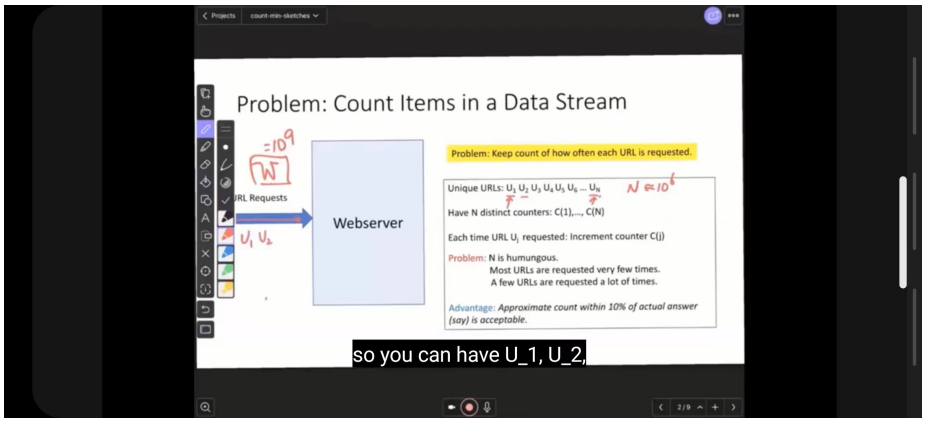
The probability of a slot being set to 1 is $1/m$. And the probability of a slot being set to 0 is $1 - 1/m$.

The probability of a slot being set to 0 after hashing n keys with k hash functions is $(1 - 1/m)^{kn} \approx e^{-kn/m}$.

Then the probability of a slot being set to 1 with the same condition is $1 - e^{-kn/m}$.

Last, the probability of every slots being set to 1 with the same conditions is $(1 - e^{-kn/m})^k$. This is the probability of false positive.

16 Count-Min Sketching Using Hashing



The main idea of Count-Min Sketching is to use multiple hash functions to count the frequency of keys.

We have a large stream of keys and we will keep a counter in order to record the apprearance of each key.

There are two counts involved in this concept: the real count and the appoximate count.

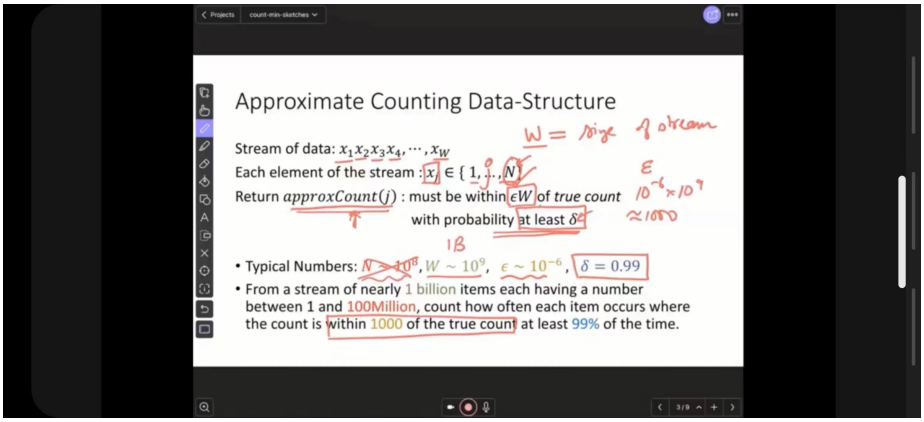
Let's take a look at a simple example.

We would like to keep count of how ofter each URL in a given web server is requested.

For each time we get a request, we will hash the URL and increase the counter by 1.

Let's say URL U_j is hashed into slot $h_i(U_j)$, and we will increase the counter $C(j)$ by 1.

16.1 Approximate Counting Data Structure



Let's take a look at a stream of elements $x_1, x_2, \ldots, x_w$ in which w is the size of the stream.

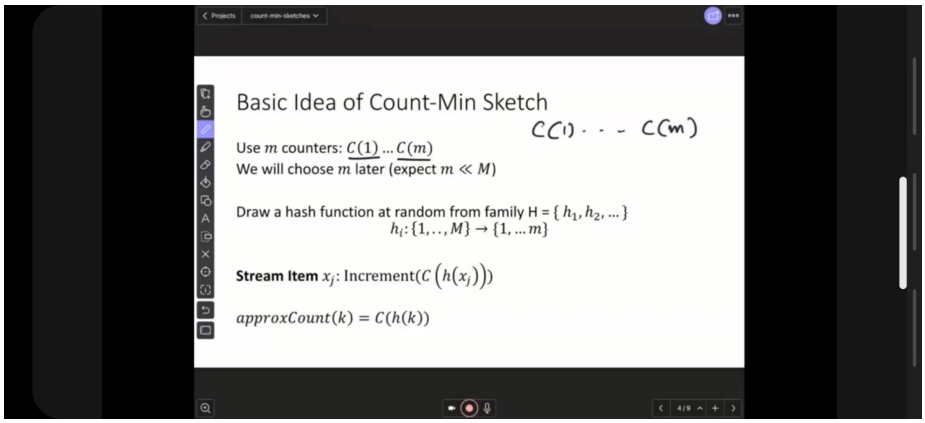
Each element $x_j \in 1, 2, \ldots, n$ (The number type is just an example, it can be any type), therefore there'll be n possibilities of x_j.

In order to get an appoximate count, the returned value must be within the range of $\epsilon * w$, where ϵ is a relatively small number.

And the probability of approximate count must larger than δ.

Here are the typical number choice for above variables:

$w \approx 10^9, n \approx 10^8, \epsilon \approx 10^{-6}, \delta \approx 0.99$.



Let's dive into the basic idea of Count-Min Sketching.

We will use m counters: $C(1), \ldots, C(m)$.

Once we encounter an element $X(j)$, we will hash it into m slots, a.k.a. $h_1(X(j)), \ldots, h_m(X(j))$.

Then we will increase the counters in the corresponding slots, such as $C(j)$, by 1.

As this mechanism goes on, we will have multiple hashes that are the same, and the counter will be increased by 1 multiple times (that is to say, multiple hashes may point to the same $C(j)$);

```
Xj -> Cj
Xm -> Cj
Xn -> Cj
```

16.2 Count-Min Sketch Error Analysis

We know that the approximate count generally is larger than the real count. But how do they relate to each other?

The probability that $h(j) = h(i)$ under certain hash function is as following;

$Pr(h(i) = h(j)) = c/m, h \in H, i \neq j$

Typically, $c = 1$.

Then the expected value of appoximatecount(j) is as following;

$E[appoximatecount(j)] = \sum_{i=1}^{n} Pr(h(i) = h(j)) * realcount(i) = \sum_{i=1}^{n} c/m * realcount(i)$**;**

And it's $\leq realcount(j) + c * w/m$.

Therefore, the 'error' in the analysis is denoted as;

$error = appoximatecount(j) - realcount(j) \leq c * w/m$**.**

16.3 Chosing m

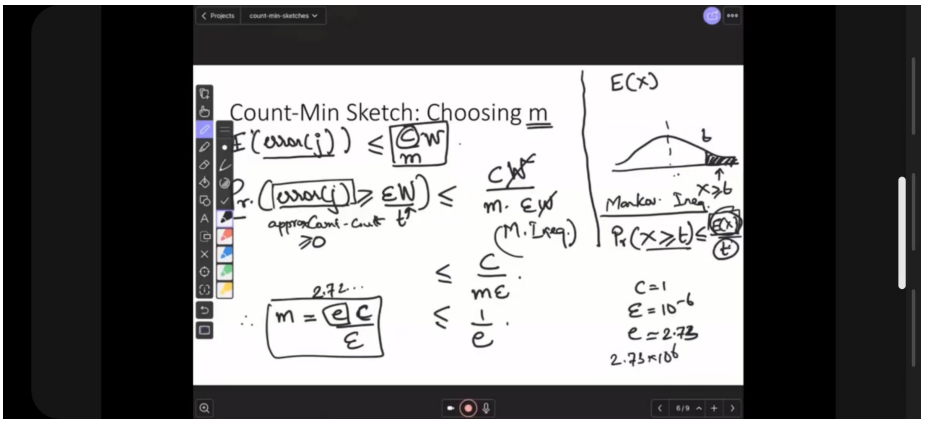
The main idea of choosing m is to make sure the error is within the range of $\epsilon * w$.

Therefore, we need to make sure $c * w/m \leq \epsilon * w$, which is $m \geq c/\epsilon$.

Using Maklov's inequality, we can get the probability of the error being larger than $\epsilon * w$ is as following;

$Pr[error(j) \geq \epsilon * w] \leq Expectation(error(j))/\epsilon * w$**, which is $\approx c/m * \epsilon$.**

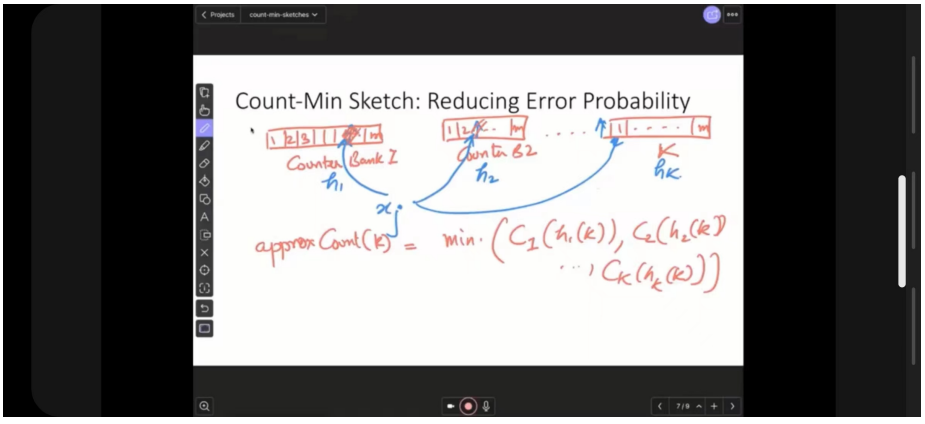
```
Here is the theory of Markov's inequality:
If X is a non-negative random variable, then for any t > 0,
Pr[X >= t] <= E[X] / t.
```

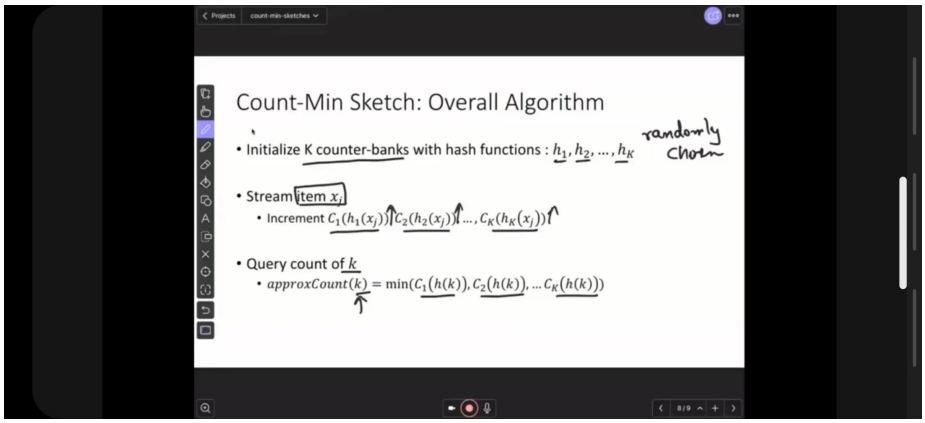


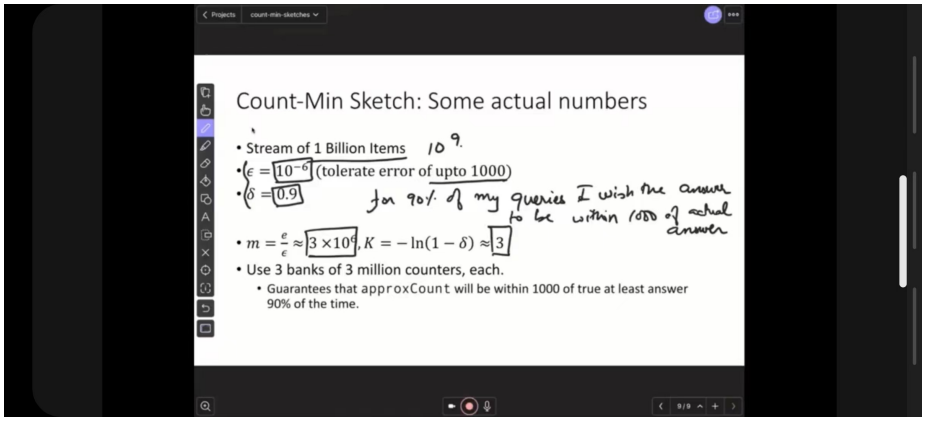
So, the probability of $Error(j) \geq \epsilon * w$ is upper bounded by $c/m * \epsilon$, a.k.a. $E(x)/t$.

That gives us $m = e * c/\epsilon$, e is the natural number.

With the given value of these variables, $c = 1, \epsilon = 10^{-6}, e = 2.73$, then $m = 2.73 * 10^6$.







Under such circumstances, we still need to reduce the error probability.

In order to do that, we need to assimilate k different hash functions. Then we will find the min count, a.k.a. the min $Pr(k)$.

We can tell that $Pr(allkgoeswrong) = (1/e)^k$, therefore when k goes up, $(1/e)^k$ goes down exponentially.

17 String Matching Using Hashing and Rabin-Karp Algorithm

The main idea of Rabin-Karp Algorithm is to use hashing to find the pattern in a string.

The main advantage of Rabin-Karp Algorithm is that it can find the pattern in $O(n+m)$ time complexity.

The main disadvantage of Rabin-Karp Algorithm is that it may have false positives.

Here is the basic idea of Rabin-Karp Algorithm.

We will hash the pattern and the string, and then compare the hashed values.

If the hashed values are the same, we will compare the pattern and the string.

If the hashed values are not the same, we will move the window and rehash the string.

The main idea of the hashing function is to use the polynomial rolling hash function.

The main advantage of the polynomial rolling hash function is that it can hash the string in $O(1)$ time complexity.

The main disadvantage of the polynomial rolling hash function is that it may have collisions.