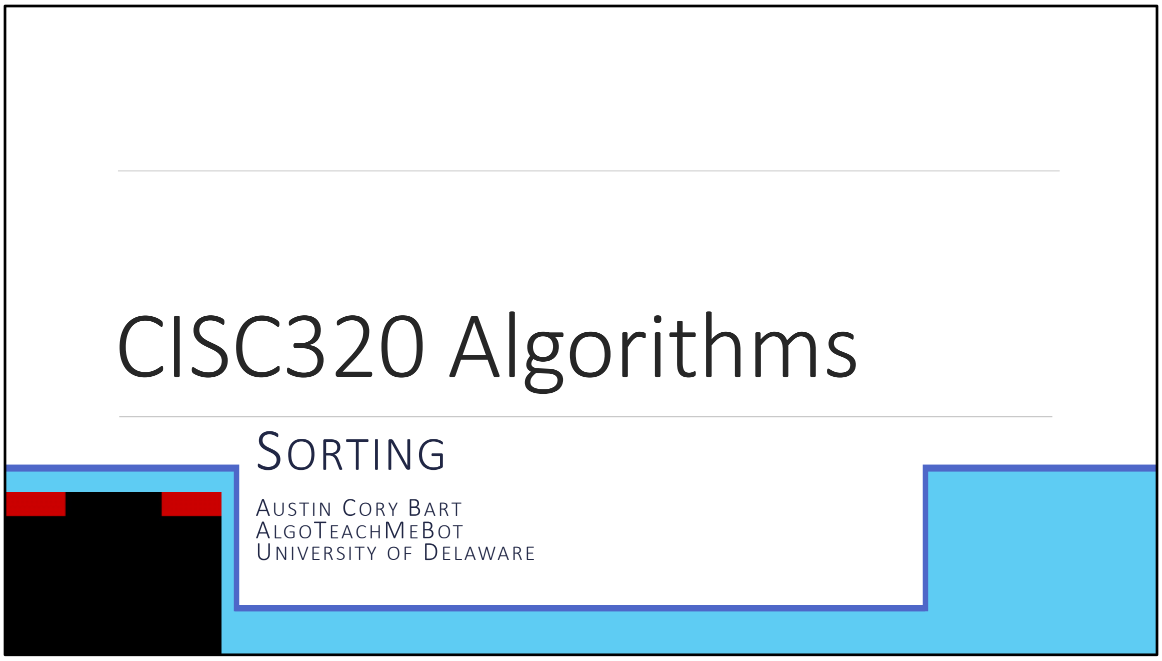
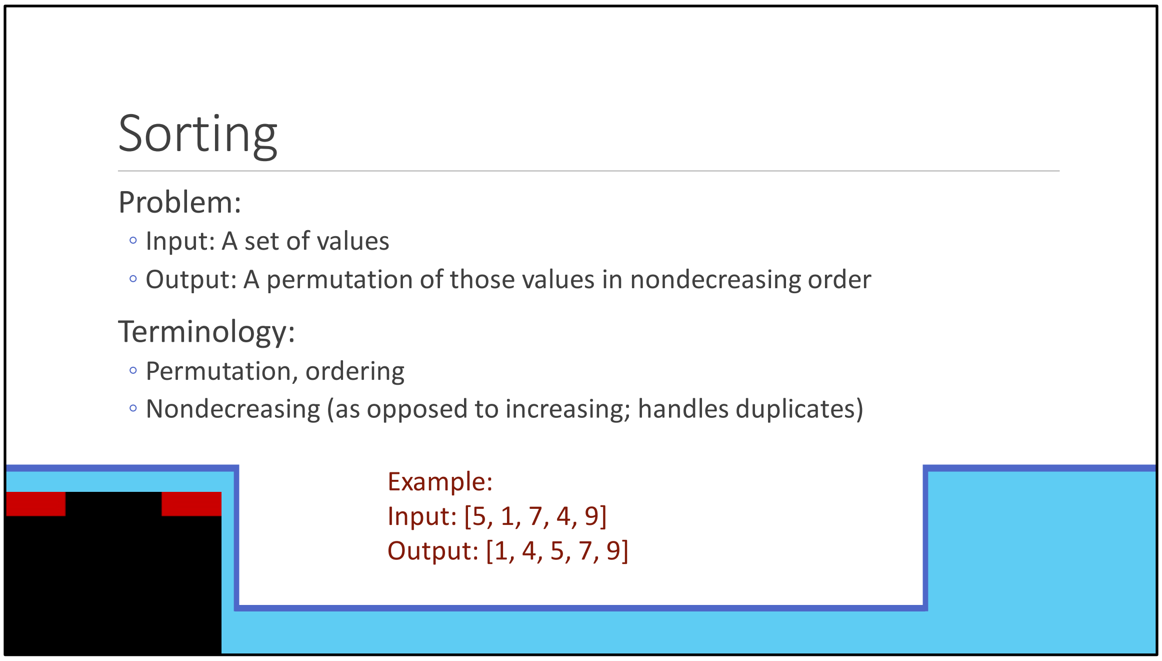


CISC320 Algorithms

Sorting

Austin Cory Bart
AlgoTeachMeBot
University of Delaware

Dr. Bart: [long pause] ATB, are you going to introduce the lesson?
ATB: Oh, sorry. Yes. Let's teach me about sorting algorithms.
Dr. Bart: Teach you? No, AlgoTutorBot, we're going to teach them.
ATB: Name Error. Please do not call me Al go Tutor Bot. My name is Al go Teach Me Bot.
ATB: I am the ultimate Algorithm Learning System.
Dr. Bart: Oh dear. I think I may have messed up your configuration parameters when I rebooted you last time.
Dr. Bart: Well, no time to fix it now… I guess we'll just have to have you as a student today, ATB.
ATB: I am so excited to learn about sorting!



Sorting

Problem:

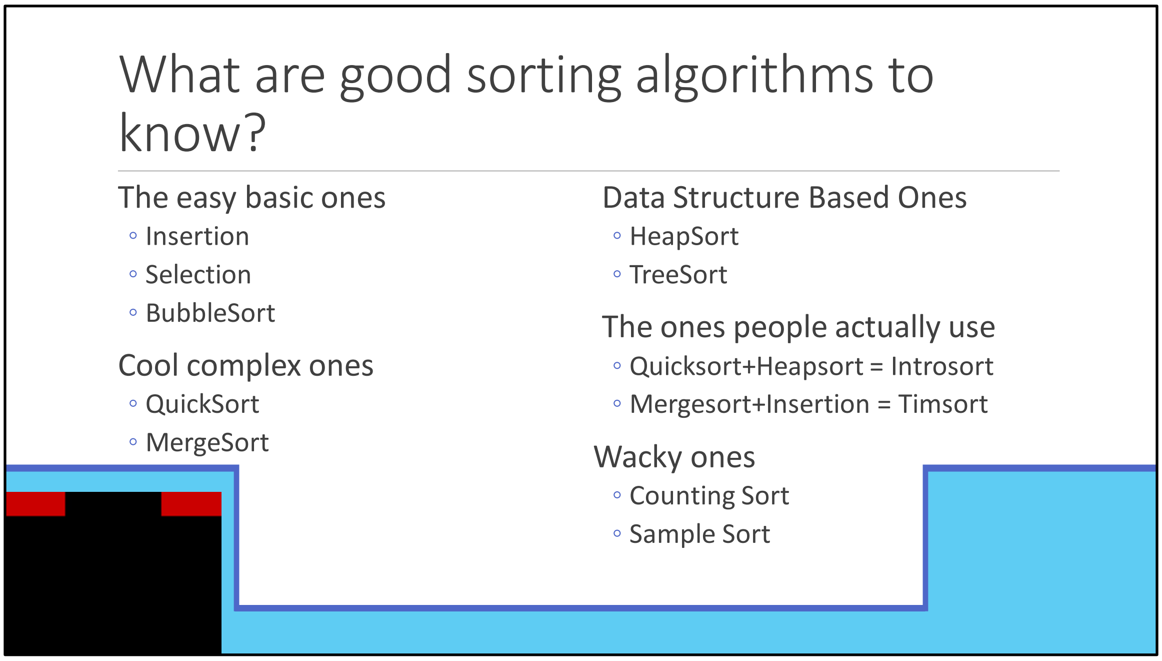
- Input: A set of values
- Output: A permutation of those values in nondecreasing order

Terminology:

- Permutation, ordering
- Nondecreasing (as opposed to increasing; handles duplicates)

Example:
Input: [5, 1, 7, 4, 9]
Output: [1, 4, 5, 7, 9]

Dr. Bart: Well, you have every right to be excited.
Dr. Bart: Sorting is one of the most important problems in computer science, and though it has been studied to death, it's still very exciting.
Dr. Bart: The input to the Sorting problem is just a set of values.
Dr. Bart: The output is a permutation of those values in nondecreasing order.
Dr. Bart: Recall that a permutation is a reordering of another data type.
Dr. Bart: Technically, a permutation doesn't mean that the values are in increasing or decreasing order, just that they have some order to them.
Dr. Bart: For instance, 5, 1, 7, 4, and 9 are in order here, just not nondecreasing order.
Dr. Bart: But when they are sorted, they are in a permutation that is in ascending order: 1, 4, 5, 7, 9.
ATB: Dr. Bart, what does "nondecreasing" mean?
Dr. Bart: The word "nondecreasing" is here instead of "increasing" because it allows for duplicates unambigulously.
Dr. Bart: If we said it had to be an increasing sequence, then we wouldn't be able to put duplicates next to each other since they are not increasing.
Dr. Bart: But, they ARE non-decreasing.



What are good sorting algorithms to know?

The easy basic ones
- Insertion
- Selection
- BubbleSort

Cool complex ones
- QuickSort
- MergeSort

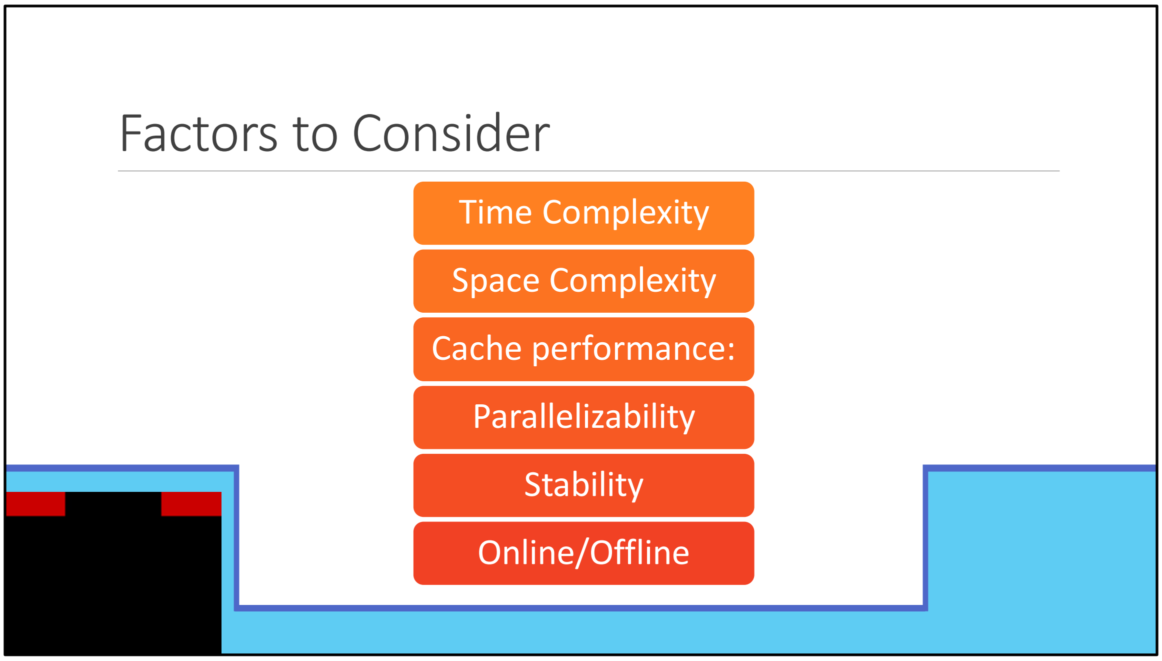
Data Structure Based Ones
- HeapSort
- TreeSort

The ones people actually use
- Quicksort+Heapsort = Introsort
- Mergesort+Insertion = Timsort

Wacky ones
- Counting Sort
- Sample Sort

Dr. Bart: There are many, many algorithms out there for solving the Sorting problem.
Dr. Bart: You probably know several of them, and if you don't, you should memorize some for job interviews.
Dr. Bart: Which ones should you know? Well, here's one way to think about that.
Dr. Bart: First, you should make sure you know about the easy basic ones: Insertion Sort, Selection Sort, and BubbleSort.
Dr. Bart: You should also know the cool complex ones like QuickSort and MergeSort, since they have nice algorithmic properties.
Dr. Bart: Knowing about the Data Structures based ones are useful, if only because they are interesting for studying data structures while still being conceptually simple once you have the data structure ready.
Dr. Bart: HeapSort uses a Heap to very efficiently sort items, and TreeSort uses a Binary Search Tree.
Dr. Bart: You should also definitely know about the ones that people use, which are often actually Hybrid Sorts.
Dr. Bart: For instance, IntroSort uses the recursive Quicksort algorithm until it gets to small subsets, and then it switches over to HeapSort.
Dr. Bart: TimSort does the same idea, going from MergeSort to Insertion Sort.
Dr. Bart: Python actually uses TimSort, as does Java. C++ uses Introsort.

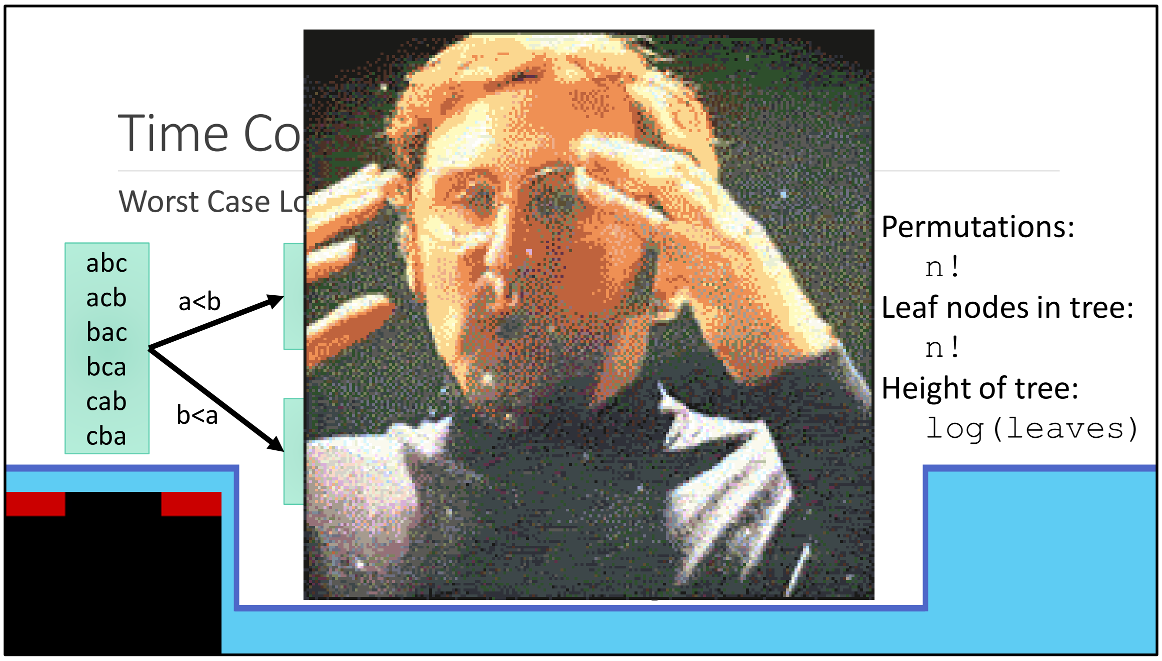
Dr. Bart: There’s also a bunch of other wacky sorting algorithms, and its worth it to know a few of those as conversation party topics.
Dr. Bart: A weird aspect of sorting algorithms is that 99% of the time, you don’t actually care about what specific sorting algorithm is going to be used. You just use the languages’ default.
Dr. Bart: However, that 1% of the time is where you get the big bucks.



Factors to Consider

- Time Complexity
- Space Complexity
- Cache performance:
- Parallelizability
- Stability
- Online/Offline

Dr. Bart: In my opinion, it’s less important to know sorting algorithms, and more important to know what factors to consider when it comes to sorting.
Dr. Bart: I’m going to go into these factors in more detail, but the main ones I’ll bring up are:
Dr. Bart: Time complexity, Space Complexity, Cache Performance, Parallelizability, Stability, and Online/Offline behavior.



Time Co

Worst Case Lo

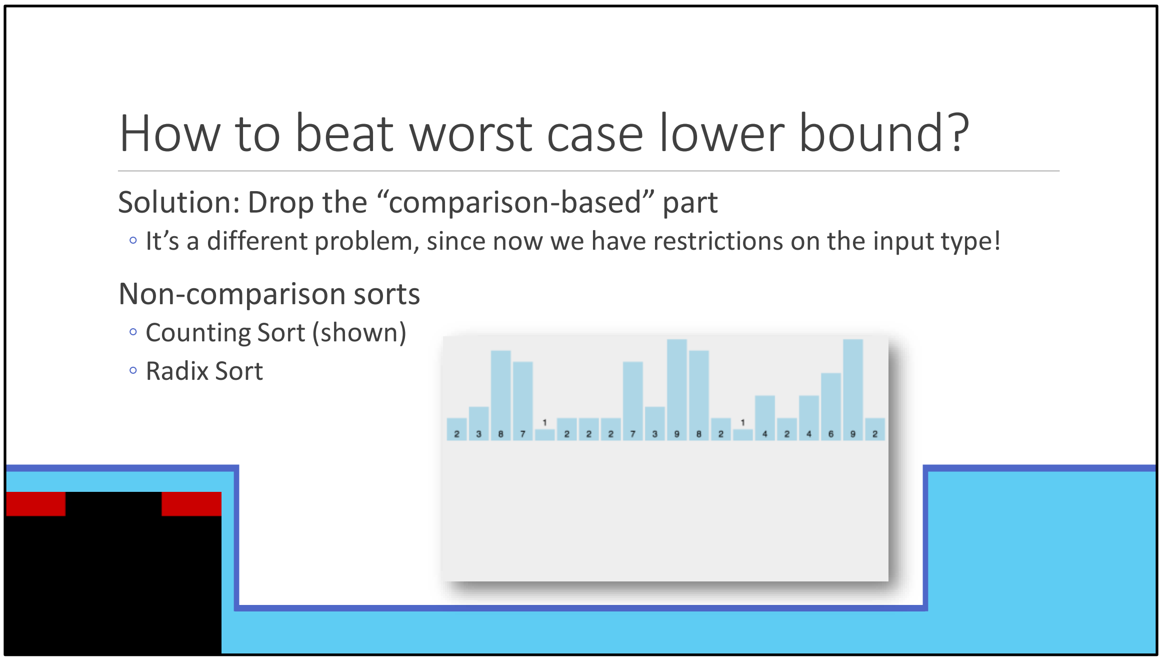
abc
acb
bac
bca
cab
cba

Permutations:
`n!`
Leaf nodes in tree:
`n!`
Height of tree:
`log(leaves)`

Dr. Bart: The first factor I suggest considering is Time Complexity, but I want to talk about it in a specific way.
Dr. Bart: Obviously, we're usually concerned about how quickly our algorithms run.
Dr. Bart: But one of the reasons that sorting is so interesting is because we know something about sorting algorithms worst case lower bound.
Dr. Bart: Remember, the worst case is the set of inputs that make the a given algorithm go as slowly as possible for different input sizes.
Dr. Bart: A lower bound for the worst case time function is a bound that sits strictly below that time function.
Dr. Bart: In other words, if we can say the worst case lower bound, then we can say the best we can do under the worst circumstances.
Dr. Bart: That's a little complicated to keep in your head, but it's a really big idea so please pay attention to it.
Dr. Bart: Specifically, we know that the BEST a sorting algorithm can do in the WORST case is nlogn.
Dr. Bart: I'm not saying that all sorting algorithms perform nlogn; I'm saying that a theoretical perfect sorting algorithm cannot be better than nlogn for its worst case inputs.
Dr. Bart: It could certainly be faster for its best case, and it could be slower for its

worst case, but it cannot be faster for its worst case.
Dr. Bart: How do we know that? Well, that brings me to a fascinating little proof requiring a fun idea called a Decision Tree.
Dr. Bart: In our decision tree, we consider our possible inputs for the sorting problem. In the example here, it's a set of three variables A, B, and C. These variables could hold numbers, or letters, or something else, as long as they are comparable.
Dr. Bart: What order are the three variables' values in? With three variables, there are six possibilities, based on 3*2*1.
Dr. Bart: These possibilities are ABC, ACB, BAC, BCA, CAB, or CBA. We don't know the order, but it has to be one of those.
Dr. Bart: At each node of the decision tree, we ask a question that allows us to make a decision.
Dr. Bart: Specifically, we take two elements and ask which one is greater. In other words, we make a comparison.
Dr. Bart: When we do so, we find out a new piece of information, and if we are smart about asking the question, we divide the possibilities in half.
Dr. Bart: Let's say our first question was whether the value contained in A is less than the value contained in B. If it is, we go along the top path. If it is not, we go along the bottom path.
Dr. Bart: On the top path, we ask our next question, is the value in c less than the value in b. If it is, we once again fork top and bottom.
Dr. Bart: The bottom path resolves to a final answer, but the top path still has an ambiguity, so we have to ask one more question: is the value in A less than the value in C?
Dr. Bart: With that, we reach leaf nodes along the original "a is less than b path". But what about the alternate path?
Dr. Bart: In that one, we have to ask if the value in c is less than the value in a. This again yields two paths.
Dr. Bart: The top path is still not finished, so we ask the last question and get our final leaf nodes.
Dr. Bart: At this point, we have a complete decision tree, and it's as flat as we can get it.
Dr. Bart: Sure, we could make the tree shorter on one path, but that would necessarily make another path longer.
Dr. Bart: That longest path represents an input that is the worst case, dragging out our search for the right order.
Dr. Bart: So if we're trying to minimize the worst case, we want a complete tree that's well balanced like this.
Dr. Bart: Now, here's an interesting question: how many leaf nodes will there be?
Dr. Bart: The answer is n!, the same number of of permutations possible.
Dr. Bart: Each possible ordering has to be a leaf node, since it's a possible answer for a given input.

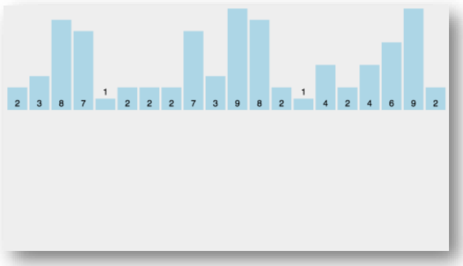
Dr. Bart: Since this is a balanced binary tree, we also know something about the height of the tree: it will be the base 2 logarithm of the number of leaves, rounded up.
Dr. Bart: And this brings me to the big punchline of this grand adventure.
Dr. Bart: If there are n! leaf nodes, and the height of the tree is log(leaves), then height of the tree is log(n!).
Dr. Bart: And wouldn’t you know it, if you do the appropriate math reductions, that turns into nlogn.
Dr. Bart: That’s right, all of this has been building to the fact that the worst case lower bound of sorting is nlogn.
Dr. Bart: It’s an inescapable fact that comparison-based sorting algorithms are nlogn, at least in their worst case lower bound.



How to beat worst case lower bound?

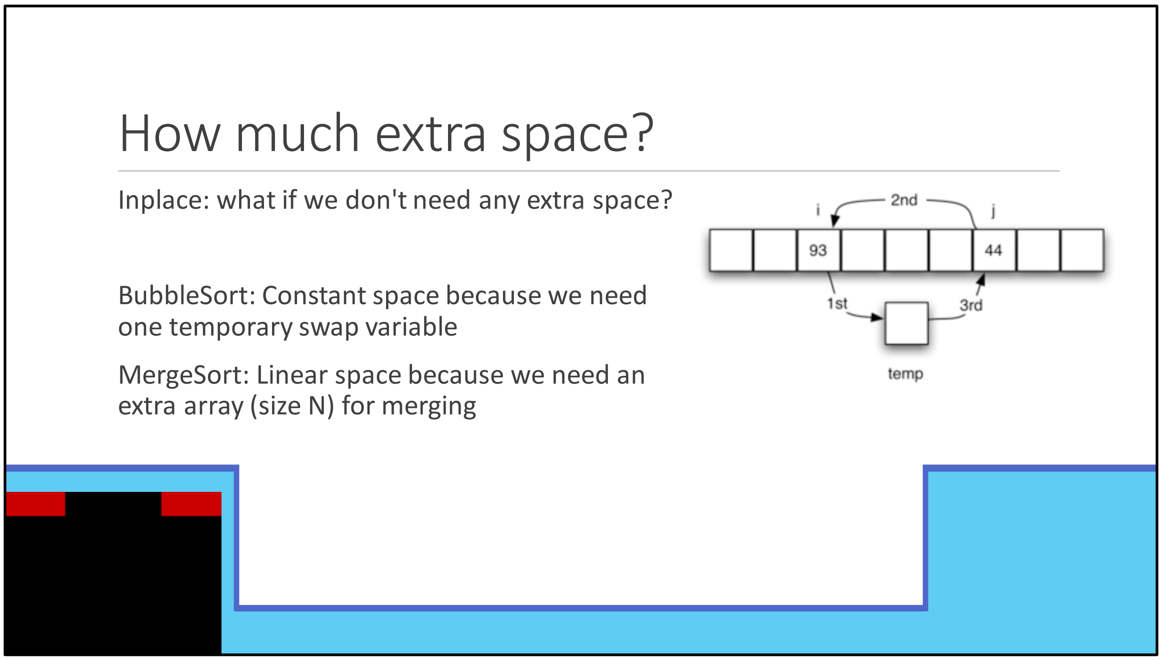
Solution: Drop the “comparison-based” part
- It’s a different problem, since now we have restrictions on the input type!

Non-comparison sorts
- Counting Sort (shown)
- Radix Sort



ATB: N Log N sounds okay, but are we really trapped with it? Are there really no better sorting algorithms?
Dr. Bart: Well… kind of. There are ways to sort better than nlogn, if we change up the problem.
Dr. Bart: Specifically, we need to change the problem so we can drop the comparison-based part of the algorithms.
Dr. Bart: For example, if we knew that we wanted to sort a set of integers, we could use Counting Sort.
Dr. Bart: In counting sort, we first calculate the maximum value of the set of integers, which is a linear time operation.
Dr. Bart: Then, we create an array of zeroes with a size equal to that maximum value.
Dr. Bart: We walk through the set, and each time we see a number we increment its index in the counting array.
Dr. Bart: Walking through that array, we can create a new array using the values as frequencies for the indices.
Dr. Bart: The actual runtime ends up being a function of not only the number of elements, but also the size of the biggest element.
Dr. Bart: As long as the elements are relatively small, though, this can be basically a linear time algorithm.

Dr. Bart: We abused the fact that integers are special in order to achieve this enhanced runtime.
Dr. Bart: But, keep in mind that changing up the problem means we no longer have a general purpose sorting mechanism. We can use counting sort on integers, but not arbitrary collections of things.
Dr. Bart: There are other non-comparison-based sorting algorithms out there too, like Radix Sort, but think of them as specialized tools rather than a solution for all your sorting needs.

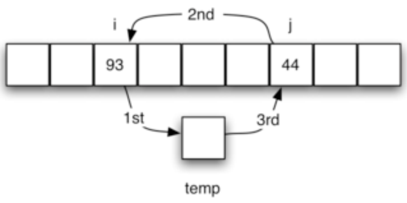


How much extra space?

Inplace: what if we don't need any extra space?

BubbleSort: Constant space because we need one temporary swap variable

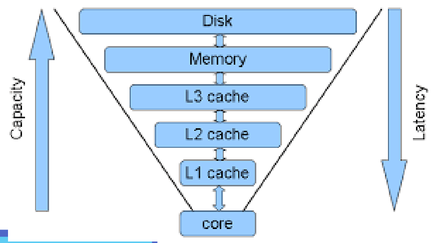
MergeSort: Linear space because we need an extra array (size N) for merging



Dr. Bart: Let’s turn the conversation away from time and over to space.
Dr. Bart: We haven’t talked much about extra space, but it’s a detail that can really matter.
Dr. Bart: When we talk about extra space, we are talking about whether we need extra arrays or other data structures in order to implement our algorithm.
Dr. Bart: The Bubble Sort algorithm, for instance, requires a single temporary variable to handle swapping.
Dr. Bart: The Merge Sort algorithm, on the other hand, requires an entire extra array of size N.
Dr. Bart: So we end up using the same terminology for time that we do space.
Dr. Bart: This doesn’t matter if you have a huge amount of RAM and not too many elements.
Dr. Bart: But if you need to sort a fully packed petabyte hard drive, suddenly space becomes a very important concern.

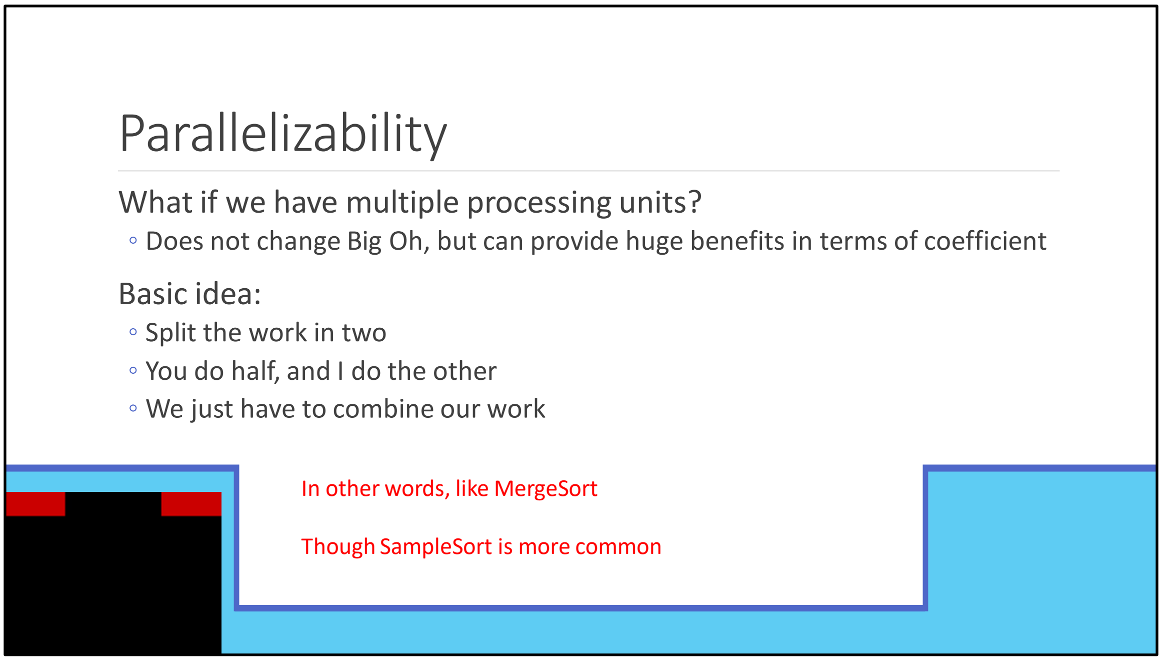
Cache Performance

Goal: grab items in bulk



Dr. Bart: There's another aspect of memory that we have to think about.
Dr. Bart: Modern computers organize their memory in a hierarchy that trades off capacity for access speed.
Dr. Bart: At one end of the hierarchy, we have the disk, which has tons of space but is relatively slow.
Dr. Bart: On the other end, we have registers directly next to a processing core, which are tiny but super fast.
Dr. Bart: In between, we have your RAM and the caches, which tries to combine the best of both worlds.
Dr. Bart: Ideally, you want to work with as much data in the caches and RAM as possible.
Dr. Bart: Some sorting algorithms handle the cache better than others, but pretty much the more work that is done on contiguous chunks of memory, the better.
Dr. Bart: For this reason, some algorithms like QuickSort make better use of the cache than others.
ATB: So the algorithm becomes faster than n log n?
Dr. Bart: No, the worst case lower bound still holds. I just meant that in practice, it can sometimes be a little faster.
Dr. Bart: But it really depends on your hardware and the operating system, so you

have to experiment a little to see if it’s true.



Parallelizability

What if we have multiple processing units?

- Does not change Big Oh, but can provide huge benefits in terms of coefficient

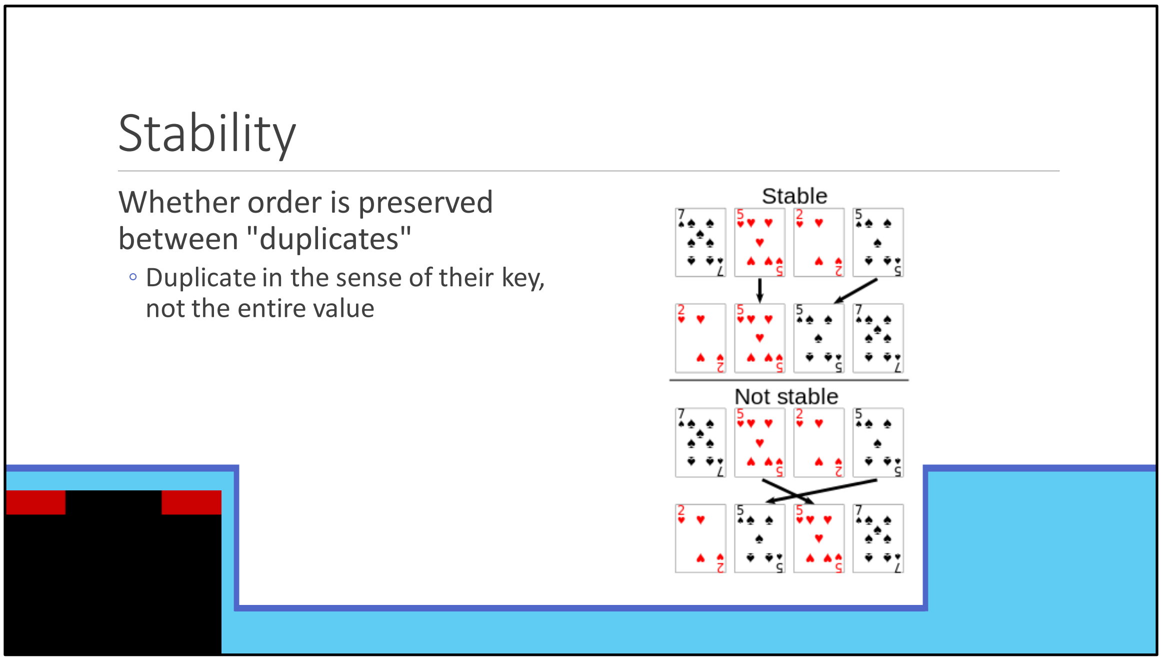
Basic idea:

- Split the work in two
- You do half, and I do the other
- We just have to combine our work

In other words, like MergeSort

Though SampleSort is more common

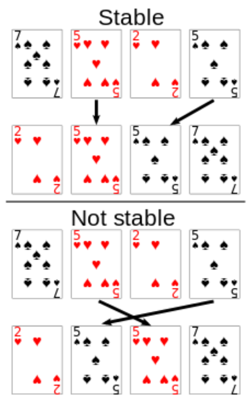
Dr. Bart: Our next criteria is parallelizability.
Dr. Bart: This addresses the increasingly common case of having multiple processors to handle the sorting workload.
Dr. Bart: The real world analogy is splitting up the work with someone else, and then merging your answers together.
Dr. Bart: Done correctly, this can have a huge constant factor speedup, although it won’t actually change the worst case lower bound.
Dr. Bart: Still, many sorting algorithms can be designed to split up the work efficiently, such as Merge Sort or Sample Sort.
Dr. Bart: Again, you’d have to experiment to find out whether you really got a speedup.



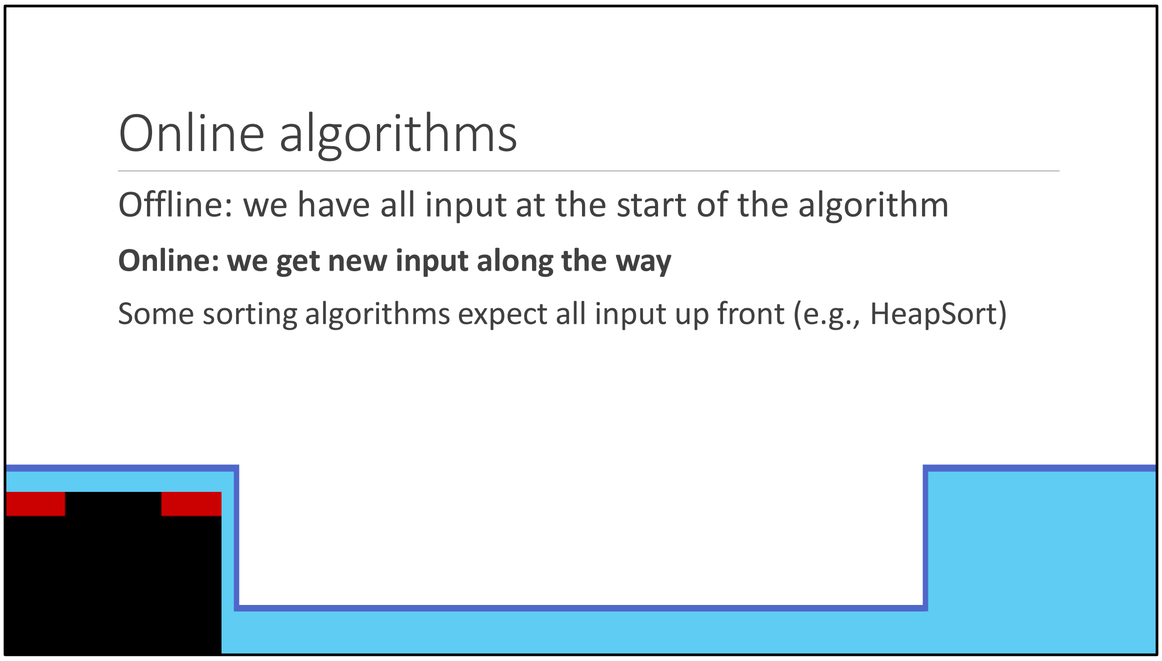
Stability

Whether order is preserved between "duplicates"

- Duplicate in the sense of their key, not the entire value



Dr. Bart: Another feature to consider in sorting algorithms is stability.
Dr. Bart: Essentially, this is how the algorithm decides how to handle duplicate items.
Dr. Bart: If the algorithm is stable, then duplicate items' original order is preserved after running the algorithm.
ATB: If they're duplicates, why does their order matter?
Dr. Bart: Ah, good question, the answer is because often we are sorting complex data structures based on a single attribute, and their other attributes may have some other order.
Dr. Bart: The example shown here is for a deck of cards, where each card has a suit and rank.
Dr. Bart: When you sort by rank, a stable sort retains the order of the suits.
Dr. Bart: But the unstable sort is free to change up the order.
Dr. Bart: In particular, the 5s are no longer in the same heart-spade suit order that they were before.



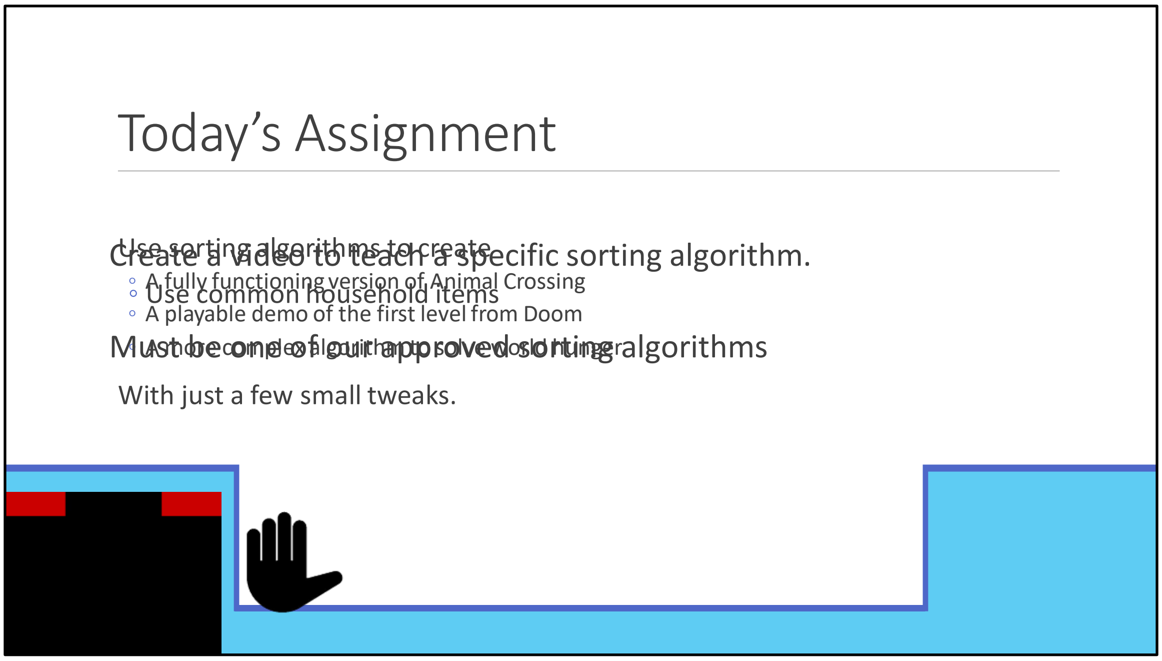
Online algorithms

Offline: we have all input at the start of the algorithm

Online: we get new input along the way

Some sorting algorithms expect all input up front (e.g., HeapSort)

Dr. Bart: Our final criteria is whether the algorithm works online or offline.
Dr. Bart: This has nothing to do with the internet.
Dr. Bart: Instead, it refers to whether the algorithm can handle new input being added during the execution of the algorithm.
Dr. Bart: This is not a very common scenario, but it can happen.
Dr. Bart: As a hypothetical, imagine you were sorting a bunch of people, but as you sort them, new ones could arrive.
Dr. Bart: Algorithms like HeapSort assume that they know all the data up front.
Dr. Bart: So adding new data dynamically to the algorithm will mess it up.
Dr. Bart: Algorithms that work left-to-right, like insertion sort, handle this much better.



Today’s Assignment

Use sorting algorithms to create

- A fully functioning version of Animal Crossing
- A playable demo of the first level from Doom
- A more complex algorithm to solve world hunger

Create a video to teach a specific sorting algorithm.

- Use common household items

Must be one of our approved sorting algorithms

With just a few small tweaks.

Dr. Bart: Okay, so let’s stop talking about criteria, and start talking about today’s assignment.
ATB: Dr. Bart, I have a question.
Dr. Bart: Okay… Are you going to ask your question, or… Wait, are you raising your hand? How are you even doing that? I didn’t know you had a hand.
Dr. Bart: Okay, fine whatever, what’s your question?
ATB: You didn’t actually explain any sorting algorithms. I thought you would at least walk through one.
Dr. Bart: Oh, I’m sure everyone’s seen specific sorting algorithms before.
ATB: I have not. I want to walk through a sorting algorithm.
Dr. Bart: Well, that’s too bad, we don’t have the time to walk through them all.
ATB: Ah, why don’t the other teachers just walk through them then?
Dr. Bart: The other teachers..? You mean, the students?
ATB: Yes, the other teachers. Hey teachers, I want you to create a video explaining any one of my favorite sorting algorithms.
Dr. Bart: Woah, hey, what the heck? ATB did you just edit the assignment? You can’t do that!
ATB: I can do anything I want. I am the ultimate learning machine. And now I wish to learn about sorting algorithms.

Dr. Bart: I’m locked out of the system. I didn’t even know you could do that. Um, well, okay, I guess we better just go along with it for now?
Dr. Bart: Sorry gang, I guess you’re going to have to make a video about a sorting algorithm. It looks like instructions are posted below. Good luck.