Announcements

- Programming Project 2 is ongoing, due March 6th
- Programming Project 3 has been released, it is due March 26th

Topic 4: Internal Sorting

By: Professor Hudson Lynam

Sorting Definition

- Definition: *Sorting Algorithm*
- *A sorting algorithm, given a sequence of records* $<r_1, r_2, \ldots, r_n>$ *produces a permutation of those* $<r_1', r_2', \ldots, r_n'>$ *such that* $r_1' \leq r_2' \leq \cdots \leq r'_n$ *(ascending order)*
- Note that achieving descending order trivial
- An “internal” sort is a sort in which all records fit into RAM; that’s what we’ll be focusing on

Why bother studying sorting?

- Java can sort for you, plenty of libraries with sorting algorithms, why bother with this?
- A few reasons:

1. Sorting is a frequently-performed task
2. Sorting algorithms provides good examples of:
   - Analysis of algorithms
   - Algorithm design principles
3. Studying several distinct approaches to one problem
4. Your language's API algorithm might not work well on your data!

Sort Algorithm Considerations

- Best vs. Worse vs. Average cases
- Key values vs. associated data (is the algorithm stable? Does it matter for your data?)
- Key comparisons vs. data movements
- Algorithm categories: Simple ($O(n^2)$) vs. Complex ($O(n \log_2 n)$)
- Quantity of data to be sorted (internal vs. external)

Classifying Sorting Algorithms

Complexity → Category ↓	Simple ($O(n^2)$)	Complex ($O(n \log_2 n)$)
Exchange	Bubble	Quick
Insertion	Insertion	Shell
Selection	Selection	Heap
Merging	---	Two-way merge
Distribution	---	Radix

Permutation Sort

- Review: In how many ways can n items be ordered?
- Answer: n!
- So, let's check all of them!

Permutation Sort:
While the list is not ordered:
    Permute the list

- Efficiency? O(n * n!) (it's O(n) to permute/check if sorted)
- Also known as Bogo Sort, this is far from the most efficient sorting technique…

The Simple Sorts

- We'll cover the traditional, common three: Bubble, Insertion, and Selection sort
- But, if they're not as efficient as the complex sorts, why cover them at all?
- They're easy to implement and test
- They have little overhead when sorting relatively small lists
- Simple sorts are often used by complex sorts when sublists become small

Bubble Sort

- The idea is that on each pass through the data, we compare overlapping pairs of elements, and exchange those not in the correct relative order.
- Bubble sort is one of the most commonly-known sorting algorithms… and also among the least efficient! It does have a cute name though.

Bubble Sort

- Let's do two passes on this data: 33 41 7 38 16

1st Pass	[33 41] 7 38 16	*No exchange*
	33 [41 7] 38 16	*Exchange 7 and 41*
	33 7 [41 38] 16	*Exchange 38 and 41*
End 1st Pass	33 7 38 [41 16]	*Exchange 16 and 41*
2nd Pass	[33 7] 38 16 **41**	*Last index (41) is **set**, ex. 33/7*
	7 [33 38]16 **41**	*No exchange*
	7 33 [38 16] **41**	*Exchange 16 and 38*
End 2nd Pass	7 33 16 **38 41**	*38 is **set.***

Bubble Sort

- For key comparisons, what would the best, worst, and average cases be?
- For the basic algorithm, $O(n^2)$ best case. With a per-pass sortedness test, best case is O(n)
- For the worst case, let's consider how many comparisons are done per pass.
- 1st Pass does n-1 comparisons. 2nd pass does n-2 comparisons… all the way to the n-1 pass, which does 1 comparison. n-1 comparisons + n-2 comparisons + … + 1 comparisons… what would be another way of expressing this?

Bubble Sort

- $\sum_1^{n-1} i$ key comparisons for our worst case!
- Review: $\sum_1^n k = \frac{n(n+1)}{2}$, so what would our polynomial be for the # of key comparisons?
- $\sum_1^{n-1} i = \frac{n(n+1)}{2} - n = \frac{n^2+n-2n}{2} = \frac{n(n-1)}{2}$
- Which is Big-O of…?
- $O(n^2)$
- Now let's do the average case!

Bubble Sort

- Imagine doing half of the possible n – 1 passes. Our # of key comparisons would equal the total # of comparisons minus the comparisons from the passes *not* performed
- $\sum_{1}^{n-1} i - \sum_{1}^{\frac{n-1}{2}} i = \frac{n(n-1)}{2} - ((\frac{n-1}{2})(\frac{n-1}{2} + 1))/2$
- $= \frac{(n^2-n)}{2} - \frac{n^2-1}{8} = \frac{3}{8} * n^2 - \frac{1}{2} * n + \frac{1}{8}$, which is Big-O of…?
- $O(n^2)$

Bubble Sort

- That's key comparisons, what about data movements?
- Best Case: O(1), since nothing is moved!
- Worst Case: $O(n^2)$, if the data is in reverse order (same sum as key comparison's worst case!)
- Average Case: $O(n^2)$, again, same reasoning as for the key comparison's average case
- Finally, what about space complexity?
- Just O(n). We need to store n data values plus a few other variables
- So, when to use bubble sort? When the list is small, and you've forgotten how the other simple sorts work!

(Cocktail) Shaker Sort: Bubble Sort Variant

- The idea is to alternate the pass directions in bubble sort. Why? Because we can move small values large distances in a single pass, not just large values.
- Quick demo: sort 5 2 3 4 1
- Pass #1 (left to right): 2 3 4 1 5
- Pass #2 (right to left): 1 2 3 4 5
- Shaker Sort is also $O(n^2)$

Insertion Sort

- The idea is to build a sorted list one item at a time, by inserting the next item into the growing list

1st Pass	**33** [41] 7 38 16	*Assume 33 is sorted, insert 41*
End 1st Pass	**33 41** 7 38 16	*33 and 41 are sorted*
2nd Pass	**33 41** [7] 38 16	*Insert 7*
	33 41 **41** 38 16	*Remember 7, slide 41 down*
	33 **33 41** 38 16	*Slide 33 down*
End 2nd Pass	**7 33 41** 38 16	*Place 7*

Insertion Sort

- Let's start with a key comparison analysis:
- Best case: O(n), if the data is already sorted, so insertions aren't necessary, just checks at the edge of the growing list
- Worst case: $O(n^2)$, if the data is in reverse order. Same summation as the worst case for bubble sort
- Average case: $O(n^2)$, same reasoning and summation as bubble sort average case

Insertion Sort

- On to data movements analysis:
- Best case: O(1), if the data is already sorted, nothing's moved!
- Worst case: if the data is in reverse order. Each pass p has p+2 moves (save, slides(s), place).
- $\sum_{3}^{n+1} i = \sum_{1}^{n-1} i + 2(n-1)$
- $\sum_{1}^{n-1} i + 2(n-1) = \frac{n(n-1)}{2} + 2(n-1)$, which is big-O of…?
- $O(n^2)$. However! If we used a linked list, the worst case is just O(n)
- Average case: $O(n^2)$.

Insertion Sort

- Finally, space complexity analysis:
- O(n), similar to bubble sort, need store n data values, plus a few other variables. If we use a linked list? Still O(n), but a larger coefficient
- When to use insertion sort?
- When the list is small… or close to being sorted!

Selection Sort

- The idea is to select the smallest unsorted item, and move it the front of those items.

1st Pass	33 41 7 38 16	*Assume 33 is smallest*
	33 [41] 7 38 16	*Scan 41, no change*
	33 41 [7] 38 16	*Scan 7, new smallest*
	33 41 7 [38]16	*Scan 38, no change*
	33 41 [7]38 [16]	*Scan 16, no change*
End 1st Pass	**7** 41 33 38 16	*Exchange 7 and 33*
2nd Pass	**7** 41 33 38 16	*Assume 41 is smallest, scan all*
End 2nd Pass	**7 16** 33 38 41	*Exchange 16 and 41*

Selection Sort

- Let's start with a key comparison analysis:
- Best case: $O(n^2)$, if the data is already sorted, we still have to check the unsorted sublist completely, every single pass. Same summation as the other two simple sort worst cases…
- We can get a Best case of O(n) with a per-pass sortedness test
- Worst case: $O(n^2)$, if the data is in reverse order. Same summation as the best case. Which means…
- Average case: $O(n^2)$

Selection Sort

- On to a data movements analysis:
- Best case: $O(1)$, if the data is already sorted, nothing will be moved!
- Worst case: $O(n)$, if the data is in reverse order, we have to swap on every pass
- Average case: $O(n)$, if we have swap n/2 times, that's still $O(n)$
- And space complexity?
- O(n), n data values plus a few other variables
- When to use selection sort?
- When the list is small, or the data is expensive to copy

Simple Sort Summary

	Bubble Sort K.C.	Bubble Sort D.M	Insertion Sort K.C.	Insertion Sort D.M	Selection Sort K.C.	Selection Sort D.M.
Best	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n^2)$	$O(1)$
Worst	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$ or $O(n)$ if LL	$O(n^2)$	$O(n)$
Average	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$ or $O(n)$ if LL	$O(n^2)$	$O(n)$

Simple Sort Practice

- Let's sort the array: 8 5 6 1 2, using each of the three simple sorts
- First: Bubble Sort. Two passes.

1st Pass	*[8 5] 6 1 2*	*Exchange 8 and 5*
	5 [8 6] 1 2	*Exchange 8 and 6*
	5 6 [8 1] 2	*Exchange 8 and 1*
	5 6 1 [8 2]	*Exchange 8 and 2*
End 1st Pass	*5 6 1 2* ***8***	*Last index* ***sorted***
2nd Pass	*[5 6] 1 2* ***8***	*No exchange*
	5 [6 1] 2 ***8***	*Exchange 6 and 1*
	5 1 [6 2] ***8***	*Exchange 6 and 2*
End 2nd Pass	*5 1 2* ***6 8***	*Second-to-last index* ***sorted***

Simple Sort Practice

- Let's sort the array: 8 5 6 1 2, using each of the three simple sorts
- Second: Insertion Sort. Two passes.

1st Pass	***8*** *[5] 6 1 2*	*Assume 8 is sorted, insert 5.*
	8 ***8*** *6 1 2*	*Save 5, slide 8 down*
End 1st Pass	***5 8*** *6 1 2*	*Place 5*
2nd Pass	***5 8*** *[6] 1 2*	*Insert 6*
	5 *8* ***8*** *1 2*	*Save 6, slide 8 down.*
End 2nd Pass	***5 6 8*** *1 2*	*Insert 6.*

Simple Sort Practice

- Let's sort the array: 8 5 6 1 2, using each of the three simple sorts
- Third: Selection Sort. Two passes.

1st Pass	*8 5 6 1 2*	*Assume 8 is smallest, scan list.*
	8 5 6 1 2	*Found that 1 is smallest*
End 1st Pass	***1 5 6 8 2***	*Swap 1 and 8.*
2nd Pass	***1 5 6 8 2***	*Assume 5 is smallest, scan list.*
	1 5 6 8 2	*Found that 2 is smallest.*
End 2nd Pass	***1 2 6 8 5***	*Swap 5 and 2.*

Sort Stability

- Definition: Stable *Sorting*
- *A sorting algorithm is stable if it retains the relative ordering of identical data items*
- Quick example: $5_1\ 3\ 5_2\ 2_1\ 8\ 5_3\ 1\ 2_2$
- A stable sort of this data would result in:
- $1\ 2_1\ 2_2\ 3\ 5_1\ 5_2\ 5_3\ 8$
- Why does it matter? For cases where you want to retain a secondary ordering of the data. Ex: for a telephone book, first sort by first names, then sort by last names. Or in financial system, keeping the transactions by order they came in *even if they have the same timestamp*

Sort Stability

- Definition: Stable *Sorting*
- *A sorting algorithm is stable if it retains the relative ordering of identical data items*
- With that being said… How would we implement a stable version the simple sorting algorithms?
- Bubble sort: Just don’t exchange equal key values
- Insertion sort: Search from high to low, and insert to the right of the first equal key found
- Selection sort: Search low to high, ties do not replace the current smallest item

The Complex Sorts

- Two opening facts:
- If you sort by comparing key values, $\Omega(\mathrm{n} * \log_2 n)$ key comparisons are needed in the worst and average cases. (The reason for this is related to information theory… we'll get into it, a little bit, later).
- Such an algorithm must use at least $\lceil \log_2 n! \rceil$ or $\mathrm{n} * \log_2 n - 1.44n$ comparisons for input sequences
- So, the complex sorts are as good it as gets… for sorting by key comparisons. But how else would we do it? Stay tuned!

Quicksort

- The idea behind quicksort is: to partition the data using a pivot selected from the keys, and then using quicksort to recursively sort the partitions
- Notes about quicksort:
- It's a naturally recursive algorithm
- Despite the name, quicksort's worst case is $O(n^2)$
- We'll start with an example array partitioning…

Quicksort

81	94	11	96	12	35	17	95	28	57	41	75	15
	L											*R*
81	15	11	96	12	35	17	95	28	57	41	75	94
		L									*R*	
81	15	11	96	12	35	17	95	28	57	41	75	94
			L								*R*	
81	15	11	75	12	35	17	95	28	57	41	96	94
							L			*R*		
81	15	11	75	12	35	17	41	28	57	95	96	94
									R	*L*		
57	15	11	75	12	35	17	41	28	81	95	96	94

Select 81 as pivot. L and R start at second index and final index.

Loop:

Move L right to find a value > the pivot. Move R left to find a value < the pivot.

If L > R, break out of Loop.

Swap the values at the L and R locations

Swap the pivot with the value at R

Quicksort

- If we have the partition subprogram available, the rest of quicksort is short and easy:

Quicksort(list[], low, high):
If low < high:
    pivot ← partition(list, low, high)
    Quicksort(list, low, pivot-1)
    Quicksort(list, pivot+1, high)

Quicksort

- Quicksort is recursive, which means our efficiency analysis will need recurrence relations…
- Best Case: We always get the median as the pivot for each sublist
  - Two partitions per pivot, with about n/2 values for each
- $Q(n) = O(n) + Q\left(\frac{n}{2}\right) + Q\left(\frac{n}{2}\right) = 2Q\left(\frac{n}{2}\right) + O(n)$
- Does this fit the Master Theorem?
- Yes! What are a, b, c, and d (and what would our big-O be)?
- a=2, b=2, c=some constant, d=1.
- Which means our big-O is: $O(n * \log_2 n)$

Quicksort

- Worst Case: We always get an “extremal” (the largest or smallest number in the list) as the pivot
  - One partition per pivot, with n – 1 values
- $Q(n) = O(n) + Q(n-1) = O(n^2)$
- Average Case: Assume a ¼ and ¾ split each time. How large would the call trees for both splits be (what would the “height” of the two trees be?)
- $\log_4 n$ and $\log_{\frac{4}{3}} n$, the larger call tree being the latter. What is $\log_{\frac{4}{3}} n$?
- $\log_{\frac{4}{3}} n = (\log_2 n)/\,(\log_2 \frac{4}{3}) = 2.41 \log_2 n$; this is the depth of the recursion, how much work is being done at each level of recursion?
- O(n), linear work. So final analysis of → O($n * \log_2 n$)

Quicksort

- There are lots of ways to improve Quicksort; we'll go over a few of them in this class

1. Shuffle the data before choosing the 1st key as the pivot. This makes it more likely to get a better pivot on almost-sorted data.
2. "Median of Three" (first, middle, last indices) pivot choice. Choosing a pivot out of three options means no extremal pivot! If there's no duplicate keys, at least…
   - John Tukey's median of medians is an extension of this.
3. Call insertion sort on smallish (~30-40 keys) sub-lists. This avoids excessive recursion overhead.
4. Dual-Pivot Quicksort partitions the data into three groups using two pivot values. This is what Java's Arrays.sort() uses, claiming $O(n * \log_2 n)$ in the worst-case

Shellsort

- The Idea: Use insertion sort on every d^{th} item, and do that d times per pass to include all items. Reduce d on each pass, finishing with d=1 (one call to regular insertion sort).
- But, what's the initial value of d, and by how much do we reduce d per pass?
- If we choose well, can Shellsort reach $O(n * \log_2 n)$?

Shellsort

81 94 11 96 12 35 17 95 28 57 41 75 15

Pass #1: Let d = 5:

81 94 11 96 12 35 17 95 28 57 41 75 15

35 17 11 28 12 41 75 15 96 57 81 94 95

Pass #2: Reduce d to 3:

35 17 11 28 12 41 75 15 96 57 81 94 95

28 12 11 35 15 41 57 17 94 75 81 96 95

Pass #3: Reduce d to 1:

28 12 11 35 15 41 58 17 94 75 81 96 95

-- 1 2 0 2 0 0 4 0 1 1 0 1

11 12 15 17 28 35 41 58 75 81 94 95 96

Shellsort

- Shellsort is very complicated to analyze—it depends on the gap sequence
- Shell: Starts with $\mathrm{d} = \frac{\mathrm{n}}{2}$, each pass $\mathrm{d} = \left\lfloor \frac{d}{2} \right\rfloor$, end at 1. Worst case: $O(n^2)$, Average case: $O(n^{\frac{3}{2}})$ (which is > than $O(n * \log_2 n)$)
- Pratt: $h_1 = 1, h_{k+1} = 3h_k + 1$. This has achieved (experimentally) $O(n^{\frac{5}{4}})$
- Gonnet: $h_0 = n, h_k = \max(\frac{5h_{k-1}}{11}, 1)$. Experimentally, $O(n^{\frac{7}{6}})$
- Ciura: 1, 4, 10, 23, 57, 132, 301, 701. Need to go bigger? Multiply by 2.25 (Tokuda's sequence)

Merge Sort

- Like quicksort, merge sort is naturally recursive.
- Unlike quicksort, no $O(n^2)$ worst case to avoid!
- The idea is to divide the list in half, recursively merge sort the halves, then merge the sorted halves into a sorted whole.

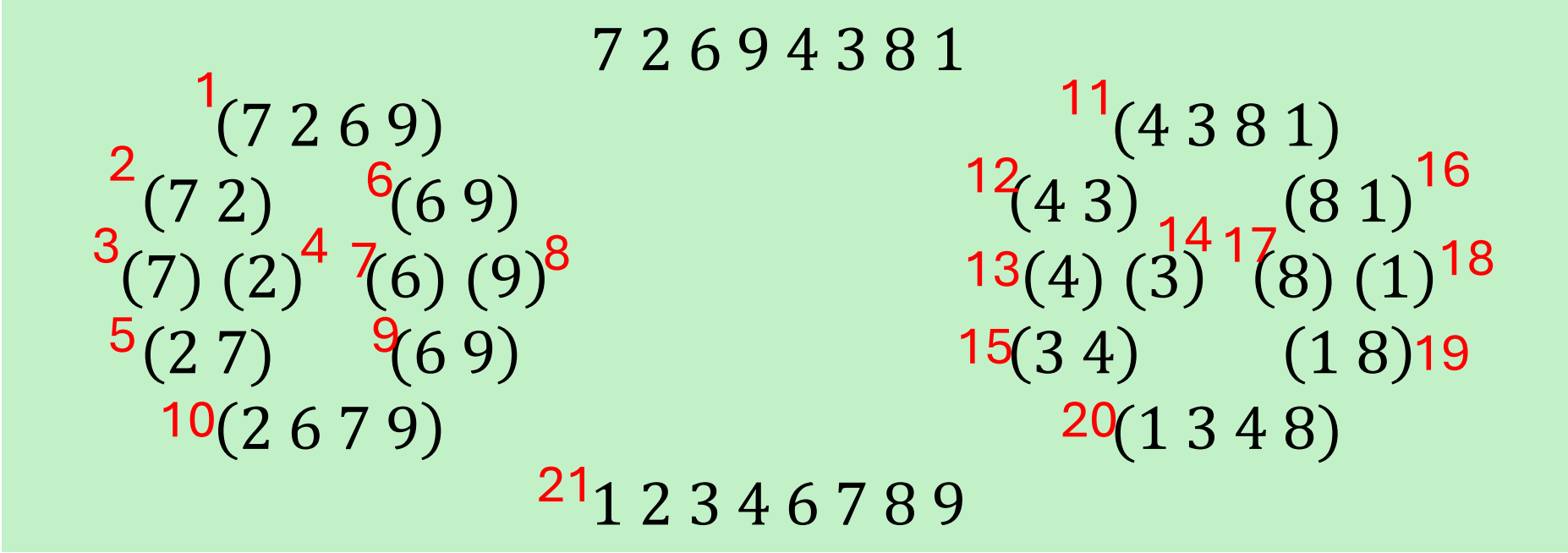
Merge Sort

- Assuming we have the merge function…

Mergesort(list[], low, high):
If low < high:
    mid ← (low + high) / 2
    Mergesort(list, low, mid)
    Mergesort(list, mid+1, high)
    merge(list, low, mid, high)

- Quicksort partitions then recurses twice, while merge sort recurses twice then merges.
- Even list splitting on each half, so no worst case!
- Merging needs O(n) additional storage for execution efficiency

Merge Sort


7 2 6 9 4 3 8 1
1 (7 2 6 9)
11 (4 3 8 1)
2 (7 2)
6 (6 9)
12 (4 3)
(8 1) 16
3 (7) (2) 4
7 (6) (9) 8
13 (4) (3)
14 17 (8) (1) 18
5 (2 7)
9 (6 9)
15 (3 4)
(1 8) 19
10 (2 6 7 9)
20 (1 3 4 8)
21 1 2 3 4 6 7 8 9

Merge Sort

- For our efficiency analysis, we need to know how efficient it is to merge two sorted sub-lists:
- O(n) comparisons and data movements, if we have access to O(n) additional space. So…
- Our best case is the same recurrence as quicksort: $M(n) = 2M\left(\frac{n}{2}\right) + O(n) \rightarrow O(n \log_2 n)$
- But the worst case is the same recurrence as the best case! So also $O(n \log_2 n)$… which means that average case should be $O(n \log_2 n)$

Merge Sort

- Two potential/common improvements for merge sort:
- Use a simple sort on smallish sub-lists (same as Quicksort, avoids recursion overhead)
- Natural Merge Sort exploits existing data orderings

$$7\ 2\ 6\ 9\ 4\ 3\ 8\ 1$$
$$(7)\ (2\ 6\ 9)\ (4)\ (3\ 8)\ (1)$$
$$(2\ 6\ 7\ 9)\ (3\ 4\ 8)\ (1)$$
$$(2\ 3\ 4\ 6\ 7\ 8\ 9)\ (1)$$
$$1\ 2\ 3\ 4\ 6\ 7\ 8\ 9$$

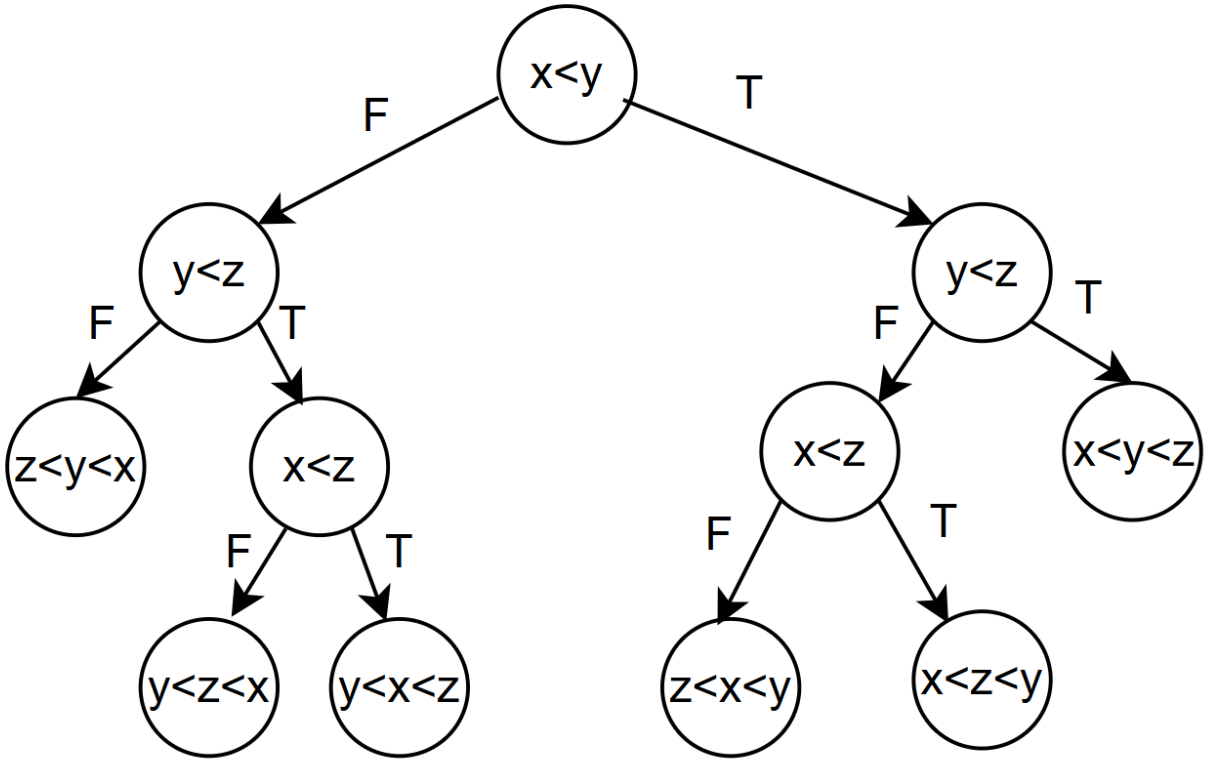
Timsort: A Modified Merge Sort

- Created to be Python's library sort, used by Java to sort objects. There are a *lot* of modifications to standard merge sort, including:

1. Exploiting existing ascending and descending runs by flipping the latter end-for-end
2. Only merge sufficiently-long contiguous runs, which are built by extending existing runs and sorting with insertion sort
3. Use "galloping" to speed up copying while merging runs
   - For example (1 2 3 4 5 6 7 8 9 16) (10 11 12 13 14 15), go from 1 -> 2 -> 4 -> 8 to find merge location faster

The Best Worst Case for Comparison Sorting

- The best case is O(n) comparisons and O(1) data movements.
- But how good can the worst case be?
- Let's try to count the number of comparisons in a decision tree



The Best Worst Case for Comparison Sorting

- We had six leaves, but a three-value decision tree can have up to eight leaves… why the discrepancy?
- For height = h, the max number of leaves is 2^h
- For n items, n! permutations exist
- Since the number of orderings $\leq$ the number of leaves ($n! \leq 2^h$), just take the log of both sides, to get: $\log_2(n!) \leq h$
- Stirling's Approximation for $\log_2(n!) = n\log_2(n) - n + O(\log_2 n)$
- So… $\mathrm{h} \geq n\log_2(n) - n + O(\log_2 n)$
- What is h representing here?
- The number of comparisons we have to do to find a given ordering of n
- So, the number of comparisons (h) is no better than $O(n\log_2 n)$

Sorting Without Key Comparisons?

- Can we do sort without key comparisons (thus avoiding the best worst case of $O(n \log_2 n)$)? Yes!
- … However, algorithms like these aren't general-purpose. We'll go over two such algorithms (briefly):
- Counting Sort
- Radix Sort

Counting Sort

- The idea is to count the occurrences of the data's values
- Example: Sort 7 2 1 4 1 2 10 4 2
- First, tabulate the quantities of occurrences:

1	2	3	4	5	6	7	8	9	10
//	///		//			/			/

- Second, create a list of the values as counted:
- 1 1 2 2 2 4 4 7 10

Counting Sort

- This looks great! Why did we bother with any of the other sorting algorithms?!
- Well… There are some significant disadvantages

1. We need additional storage for the tabulations.
   - How much? Depends on the min and max key values!
2. Reproducing the keys from the counts disconnects the keys from the data associated with the keys
3. Finally, while Counting Sort *looks* like a linear-time sort, it isn't really…

Counting Sort

- Analysis of Counting Sort:
- Step 1: Tallying the key values
- O(n)
- Step 2: Creating the ordered list of key values
- O(k), where k is the length of the tally array!
- Total: O(n+k), which could still be good… except for the storage and data dissociation issues!

Radix Sort

- The idea is to examine values by the positions of their glyphs (digits). ‘Radix’ means ‘base’ in number systems.
- The algorithm (assuming 0 is smallest digit & we have ‘base’ queues):

Radix Sort:
Enqueue all data into Queue 0
For each radix position (least—significant first):
    Dequeue values from the radix value queues, starting with Q
        Queue 0
    Enqueue each of those values into the appropriate queue
        for that value’s digit in the current radix position
Dequeue all values from the radix value queues to produce the sorted key list

Radix Sort

- Example: Sort 426 4 219 97 134 6 494 17 101
- 1's position:

0	1	2	3	4	5	6	7	8	9
	101			4 134 494		426 6	97 17		219

- 10's position

0	1	2	3	4	5	6	7	8	9
101 4 6	17 219	426	134						494 97

Radix Sort

- Example: Sort 426 4 219 97 134 6 494 17 101
- 10's position:

0	1	2	3	4	5	6	7	8	9
101 4 6	17 219	426	134						494 97

- 100's position

0	1	2	3	4	5	6	7	8	9
4 6 17 97	101 134	219		426 494					

- Dequeue L → R to get keys in sorted order

Radix Sort

- Can we keep keys and associated data together for Radix Sort?
- Yes! We can just queue up objects instead of only key values
- We must start with the least-significant radix position
- We can use any radix that we want (Base 95 for printable ASCII strings, for example)
- We can reduce passes (but add queues) by using pairs of glyphs
- For example: Co mp ut er

Radix Sort

- A brief analysis:
- The sort requires p passes, where p = the number of glyphs in the longest key
- Each pass works with r queues and n key values, where r = the radix
- All together: O(p(r+n))
- …Which seems pretty good! But it has significant overhead (queues, ‘radixizing’ the keys)

Quicksort Partitioning Practice!

16 4 6 8 12 23 13 5 14 18 22 33 0

Select 13 as the pivot, and swap with 16

13 4 6 8 12 23 16 5 14 18 22 33 0

L *R*

13 4 6 8 12 23 16 5 14 18 22 33 0

L *R*

13 4 6 8 12 0 16 5 14 18 22 33 23

L R

13 4 6 8 12 0 5 16 14 18 22 33 23

R L

5 4 6 8 12 0 13 16 14 18 22 33 23

Except… you must use the "Median of Three" approach.

Start with L and R at the second and final indices

Move them until L's element is > pivot, and R's element is < pivot

Swap R and L's elements

Identify the next swapping point.

Swap again

Identify the next swapping point…

but R < L, so…

Swap the pivot with R