

Sorting

Or, adding a handy new sorting operation to many ADTs

Why Bother?

- Almost everything is sorted!
- We sort our lives
 - make ordered lists
 - make appointments (ordered by time)
 - etc.
- Computers waste large amounts of time sorting.

To Which Data Structures Does Sorting Apply

- Can just add a “sort” to many of the ADTs like the linked list, array list, etc.
- Simply re-orders the data and puts back in the data structure.
- But doesn't work for all data structs.
 - e.g., heap has no inherent order
 - BUT, can use the heap to help order other structs like the arraylist (heapSort – just like the heapSelect that we discussed)

Our Assumptions

1. We will assume that we are sorting an array of integers
 - Could really sort anything that is “comparable”
 - e.g., alphabetizing letters
 - big-O rankings
 - sort ski run names by their difficulty
 - can you think of another example?
 - almost always can convert objects to numbers and compare
2. Assume the sorts can be done in main memory
 - What’s the problem if have to sort stuff on a hard drive?
 - Run time is very slow w/ disk access operations.
 - Called external sorting.
 - Only necessary when have large amount of data.

Our Assumptions (cont.)

3. All functions/methods for our sorting algorithms will look the same
 - Pass in an array of numbers as an argument.
 - The array is assumed to be full.
 - Assume N is the number of elements in the array.

Preliminaries: New Big-O Notation

- $O(N^2)$
 - means that it runs in less than or equal to that time
- $o(N^2)$
 - means that it runs in less than that time.
 - So slightly better than $O(N^2)$
- $\Omega(N^2)$
 - means that it runs in greater than or equal to that time
 - often a bad thing!
- $\Theta(N^2)$
 - means that it runs in exactly equal to that time (rare)
 - I don't mean, the run time is exactly N^2 . I mean that every possible permutation of the code will give $O(N^2)$. For example, can't take a path that gives $O(N)$ for some input.
 - Often show Θ for a subset of the inputs – like “insertion sort is $\Theta(N^2)$ for all reverse sorted lists”.

Some Things to Keep In Mind

- Easy to come up with $O(N^2)$ or worse algorithms
 - Bubble sort
 - how's that work again?
 - Shellsort (will see soon)
 - actually $o(N^2)$... **this means always LESS time than N^2 .**
 - recall $O(N^2)$ means less than or equal to that time
- More sophisticated approaches will be $O(N \log N)$
 - Heapsort
 - QuickSort
 - etc.
- Any “all-purpose” sorting algorithm requires $\Omega(N \log N)$ comparisons – a mathematical reality.
 - Ouch! **That means GREATER than (or equal to) that time.**

Insertion Sort

- Requires $N-1$ passes through the array.
- For pass p , move the p th element of array to the left until it is in its correct position.

Example: {23, -4, 6, 3, 4, 21}

$p = 0$ {23, -4, 6, 3, 4, 21} (move 23)

$p = 1$ {-4, 23, 6, 3, 4, 21} (move -4)

$p = 2$ {-4, 6, 23, 3, 4, 21} (move 6)

$p = 3$ {-4, 3, 6, 23, 4, 21} (move 3)

$p = 4$ {-4, 3, 4, 6, 23, 21} (move 4)

$p = 5$ {-4, 3, 4, 6, 21, 23} (move 21)

Insertion Sort Growth Rate

- What's the worst case?
 - will have to move all the way to the left each time.
 - e.g., when the list is reverse sorted!
 - {6, 5, 4, 3, 2, 1}
 - So first compare and move 1 position
 - Then compare and move 2 positions
 - Then compare and move 3 positions

Cool!

$$\sum_{i=1}^{N-1} i = 1 + 2 + 3 + 4 + 5 + 6 + \dots + N - 1 = \frac{N(N-1)}{2} = \Theta(N^2)$$

Exactly N^2 when the list is reverse sorted.

Insertion Sort Code

```
int j;
for(p = 1; p<arraySize; p++)
{
    int tmp = a[p];
    for(j=p; j>0 && tmp < a[j-1]; j--)
    {
        a[j] = a[j-1];
    }
    a[j] = tmp;
}
```

So what is the big-O growth rate?

Good. Agrees with our math analysis – always nice to know!

More About Insertion Growth Rate: Inversions

- Inversions
 - An inversion is any pair (i, j) where $i < j$ and $a[i] > a[j]$.
- For example, the list $a = \{23, -4, 6, 3, 4, 21\}$ has some inversions
 - $i = 0, j = 1$ but $23 > -4$ so inversion
 - $i = 0, j = 2$ but $23 > 6$ so inversion
 - $i = 2, j = 4$ but $6 > 4$ so inversion
 - is $i = 1, j = 2$ an inversion?
- How many inversions are there? (7)

Inversions Give Growth Rate

- Suppose have a total of $I = 7$ inversions.
- Then each of these have to be swapped to put a list in order.
 - Why? Because a sorted list has no inversions.
 - Swapping two *adjacent* elements to put in the correct order will remove exactly one inversion.
 - That's what insertion sort does.
- S'pose the growth rate of everything except the swaps is $O(N)$.
- Then the total growth rate is $O(I+N)$.
 - I might be equal to N^2 . $O(N^2)$ total.
 - Or I might be equal to N . $O(N)$ total.

Inversion Growth Rate

- We already showed that $I = N^2$.
 - That is the worst case.
- What if the list is almost sorted to begin with?
 - Try sorting a list with only one element out of place. One pass will fix that. $O(N)$.
 - So $I \leq N$. **So $O(N)$ if list is almost sorted.**

So *simple* code for insertion sort is good in these situations.

Average Insertion Growth Rate

- But how about the average case? On *average*, how many inversions are there?
 1. Well, can count all the different ways that a list of numbers could be arranged.
 2. Then can count the number of inversions.

Permutations

- Let's consider all the permutations of $\{1, 2, 3\}$.

• $\{3, 1, 2\}$	2 inversions	←	} opposite order
• $\{3, 2, 1\}$	3 inversions	←	
• $\{2, 1, 3\}$	1 inversion	←	
• $\{2, 3, 1\}$	2 inversions	←	} opposite order
• $\{1, 2, 3\}$	0 inversions	←	
• $\{1, 3, 2\}$	1 inversion	←	} opposite order

- 6 total permutations (i.e., $3!$ permutations)
- 9 total inversions
- An average of $9/6 = 1.5$ inversions.

Consider “Opposites”: A Proof of the Average Run Time

1. Max number of possible inversions in an array of N numbers is

$$\sum_{i=1}^{N-1} i$$

2. The opposites are in the reverse order.
3. Suppose, there are m *inversions* for a particular permutation. When listed in opposite order there will be this many *inversions*.

$$\left(\sum_{i=1}^{N-1} i \right) - m = \left(\frac{N(N-1)}{2} \right) - m$$

Average Run Time Proof (cont.)

4. Taken together, this pair has $(m + (N(N-1)/2) - m)$ inversions. Or $N(N-1)/2$.

5. And how many pairs are there? $N!/2$.

6. So the *total number* of inversions is

$$(N(N-1)/2)(N!/2)$$

7. So the *average number* of inversions is

(total # of inversions) / (number of permutations)

$$= (N(N-1)/2)(N!/2) / (N!).$$

$$N(N-1)/4 = O(N^2)$$

done!

Theorem: Adjacent Pair Sorting

- **Theorem:** Any algorithm that sorts by exchanging adjacent pairs requires $\Omega(N^2)$ on average.
- **Proof:**
 1. The average number of inversions is $N(N-1)/4$.
 2. Each pair swap removes 1 inversion.
 3. So $N(N-1)/4$ swaps are required.
 4. Hence, $\Omega(N^2)$.

Bad News!

- Seems to imply that the best case behavior is $\Omega(N^2)$ or worse!
- If we want $o(N^2)$ run time (i.e., better than quadratic) then will need an improved strategy.
- Can't use adjacent pair swapping.

But What If Allow Long-Distance Swaps?

- $\{23, -4, 6, 3, 4, 21\}$
 - This has 7 inversions.
 - Now just swap the 23 and the 4. $\{4, -4, 6, 3, 23, 21\}$.
 - This leaves only 4 inversions. With one long-distance swap, we eliminated 3 inversions!
- This is the essence of faster sorting algorithms!

Shellsort

- Named after Donald Shell.
- First sub-quadratic sorting technique, 1959.
- Works by comparing distant pairs.
 - starts with long distance pairs.
 - progressively takes closer and closer pairs.
 - e.g., start with pairs that are 5 apart. Then 3 apart. Then 1 apart.
 - Always end with closest pairs.
 - Sometimes called “diminishing increment sort.”
- So by the time we get to the 1-apart-sort, we have an almost sorted list! So we use insertion sort, and by the end, it will run quickly, $O(N)$.

Shellsort Example

- Sort this array: {81, 94, 11, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15}
 - Start by doing insertion sort on the numbers separated by 5.
 - First:
 - {81, 94, 11, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15} becomes
 - {35, 94, 11, 96, 12, 41, 17, 95, 28, 58, 81, 75, 15}
 - Next:
 - {35, 94, 11, 96, 12, 41, 17, 95, 28, 58, 81, 75, 15} becomes
 - {35, 17, 11, 96, 12, 41, 75, 95, 28, 58, 81, 94, 15}
 - Next:
 - {35, 17, 11, 96, 12, 41, 75, 95, 28, 58, 81, 94, 15} becomes
 - {35, 17, 11, 96, 12, 41, 75, 15, 28, 58, 81, 94, 95}

Shellsort Example (cont.)

– Next:

- {35, 17, 11, 96, 12, 41, 75, 15, 28, 58, 81, 94, 95}

becomes

- {35, 17, 11, 28, 12, 41, 75, 15, 96, 58, 81, 94, 95}

– Next:

- {35, 17, 11, 28, 12, 41, 75, 15, 96, 58, 81, 94, 95}

becomes

- {35, 17, 11, 28, 12, 41, 75, 15, 96, 58, 81, 94, 95}

» no change

Shellsort Example (cont.)

- Now do insertion sort on the numbers separated by 3.
- First:
 - {35, 17, 11, 28, 12, 41, 75, 15, 96, 58, 81, 94, 95}
becomes
 - {28, 17, 11, 35, 12, 41, 58, 15, 96, 75, 81, 94, 95}
- Etc.:
 - {28, 17, 11, 35, 12, 41, 58, 15, 96, 75, 81, 94, 95}
becomes, when all done with remaining 3-sorts,
 - {28, 12, 11, 35, 15, 41, 58, 17, 94, 75, 81, 96, 95}

Shellsort Example (cont.)

- Now do insertion sort on the numbers separated by 1.
- First move $p = 1$:
 - {28, 12, 11, 35, 15, 41, 58, 17, 94, 75, 81, 96, 95} becomes
 - {12, 28, 11, 35, 15, 41, 58, 17, 94, 75, 81, 96, 95}
- Next move $p = 2$:
 - {12, 28, 11, 35, 15, 41, 58, 17, 94, 75, 81, 96, 95} becomes
 - {11, 12, 28, 35, 15, 41, 58, 17, 94, 75, 81, 96, 95}
- Etc. until have
 - {11, 12, 15, 17, 28, 35, 41, 58, 75, 81, 94, 95, 96}

So Why Is Shellsort Faster?

- Notice that each time we sort another “shell” it has been partially sorted already.
- Remember that insertion sort runs very quickly, $O(N)$, when the array has already been partially sorted.

What Increments?

- Previous example used 5, 3, 1.
- Turns out, can use *any* increments!
 - some are faster than others.
 - hard to prove.
- Popular (bad) choice:
 - Proposed by Shell himself.
 - Suppose sequence is $h_1, h_2, h_3, \dots, h_t$
 - Then let $h_t = N/2$ (integer arithmetic)
 - $h_{t-1} = h_t / 2$
 - Etc., until have $h_1 = 1$.
 - But is $\Theta(N^2)$ in *worst* case scenario.

Show Worst Case Bound

- I'll prove Shell's increments are $\Omega(N^2)$ for at least one example (greater than or equal to N^2).
- Proof by counter-example:
 - Suppose it runs $o(N^2)$. This means always faster than N^2 . Well, here's an example that takes longer.
 - start: {1, 9, 2, 10, 3, 11, 4, 12, 5, 13, 6, 14, 7, 15, 8, 16}
 - 8-sort: {1, 9, 2, 10, 3, 11, 4, 12, 5, 13, 6, 14, 7, 15, 8, 16}
 - 4-sort: {1, 9, 2, 10, 3, 11, 4, 12, 5, 13, 6, 14, 7, 15, 8, 16}
 - 2-sort: {1, 9, 2, 10, 3, 11, 4, 12, 5, 13, 6, 14, 7, 15, 8, 16}
 - 1-sort: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}
 - Everything happens in the last sort – it is just an insertion sort! And insertion sort is $O(N^2)$. So the last 1-sort alone is $O(N^2)$. And the other sorts add more (even though they didn't do anything). So it is $\Omega(N^2)$.

Better Increments

- **Problem:** The $h_k/2$ increments sort the same numbers over and over.
 - e.g., 4, 2, 1
 - {28, 17, 11, 35, 12, 41, 58, 15, 96, 75, 81, 94, 95}
- Wastes time.
- **Hibbard's increments:**
 - 1, 3, 7, ..., $2^k - 1$
 - Similar, but successive elements are “relatively prime” – no common factors.
 - Is $\Theta(N^{3/2})$ in *worst* case scenario.
 - but on average is closer to $O(N^{5/4})$

Best Increments

- Sedgewick's increments
 - 1, 5, 19, 41, 109...
 - notice that these are always presented in reverse order.
 - Alternate these formula
$$9 * 4^i - 9 * 2^i + 1$$
$$4^i - 3 * 2^i + 1$$
 - $O(N^{4/3})$ in worst case
 - $O(N^{7/6})$ on average (from simulations)
- Others options are sometimes used, and proofs show that some as yet undiscovered increments should be much faster.

Shellsort Code

- Using Shell's increments

```
int j;  
for(int gap = arraysize/2; gap > 0; gap/=2)  
{  
    for(int i = gap; i<arraysize; i++)  
    {  
        int tmp = a[i];  
        for(j = i; j >=gap && tmp < a[j-gap]; j -= gap)  
        {  
            a[j] = a[j-gap];  
        }  
        a[j] = tmp;  
    }  
}
```

The gap is just the spacing between the elements being sorted. The increment. Note it is being divided by two each time.

This is just the insertion sort routine, but performed on elements spaced apart by a distance of gap. Let gap = 1 and have exactly the insertion sort!

An Example

- **Suppose I told you to write a program that sorts an array of randomly generated numbers and prints it out. How would you do this?**
- **Steps:**
 1. Generate an array of random numbers.
 1. create array
 2. loop over each element in the array and put a random number in it
 2. Create a method that sorts the numbers.
 1. use insertion sort, or shellsort, or bubble sort, or...
 2. use the code from class.
 3. the method should take an array as an argument.
 3. Pass the array to the method.
 4. Print out the sorted numbers.
 1. loop over each element in the array and print it out.

Example Pseudo-code

```
public class MainClass
{
    main()
    {
        create array called "a";
        for(each element "i" in the array)
            a[i] = random number;

        a = SortingClass.sortingMethod(a);

        for(each element "i" in the array)
            print a[i];
    }
}
```

```
public class SortingClass
{
    public static int[] sortingMethod(int[] a)
    {
        code for insertion sort

        return the array
    }
}
```

Note: Java passes arrays by reference, so you don't have to return the array if you don't want to bother.