Prof. Darrell Long

CSE 13S

# Sorting

# What is sorting?

- Sorting is the act of putting things into a defined order.
- Dictionaries are sorted in what is called *lexicographical* order.
    - Only fancy people call it that, most people say *alphabetical* order.
- Numbers can be sorted in their natural order, or reverse order.
- There are *total* and *partial* orderings, but we will only concern ourselves with total orderings.

# Why do we sort?

- Sorting adds information to our data.
- For example, we now can make assertions about before, after, lesser, greater, and so forth.
- Here are a few examples:
  - We can search most efficiently in sorted data.
  - We can marge sorted lists efficiently.
  - We can detect duplicates efficiently.


https://xkcd.com/1386/

# First idea…

- Enumerate all of the possible orderings of *n* objects.
  - There are $n!$ such orderings
- Pick the one that is in order.

- On second thought, that was a *bad* idea.
- So what should we do? Find a *better* idea.

| $\boldsymbol{n}$ | $\boldsymbol{n}!$ |
|---:|---:|
| 1 | 1 |
| 10 | 3628800 |
| 20 | 2432902008176640000 |
| 30 | $2.652\times10^{32}$ |
| 40 | $8.158\times10^{47}$ |
| 50 | $3.041\times10^{64}$ |
| 60 | $8.320\times10^{81}$ |

This number exceeds the number of protons in the known universe.

# Second idea…

$$n + (n-1) + (n-2) + \ldots + 1 = \frac{n(n+1)}{2}$$

Carl Friedrich Gauß

- Find the smallest item, to do that we must look at all $n$ of them.
  - Put it in the first slot.
- What do I do with what was already there?
  - Let’s swap them.
- Once the smallest item is found, we never look at it again. In effect, we have a smaller, by 1, unsorted array.
- We repeat the same process for smaller and smaller subarrays: $n-1, n-2, \ldots, 1$.
- This is an instance of *Selection Sort*.

# MinSort

```c
#include "minsort.h"
#include <stdint.h>

// minIndex: find the index of the least element.

int minIndex(uint32_t a[], int first, int last) {
  int min = first;
  for (int i = first; i < last; i += 1) {
    min = a[i] < a[min] ? i : min;
  }
  return min;
}

// minSort: sort by repeatedly finding the least element.

void minSort(uint32_t a[], int length) {
  for (int i = 0; i < length - 1; i += 1) {
    int min = minIndex(a, i, length);
    if (min != i) {
      SWAP(a[min], a[i]);
    }
  }
  return;
}
```

Good use of the ternary operator

# Third idea…

$$1 + 2 + 3 + \ \ldots + (n-1) = \frac{n(n-1)}{2}$$

- Suppose we have an array with one element, is it sorted?
  - Yes, by definition we say it is sorted.
- Given a second element, it must go either before or after the first element.
  - Now the two element subarray is sorted.
- Given a third element, it must before the first element, between the two elements, or after the second.
  - Now the three element subarray is sorted.
- Proceed to 4, 5, … elements. So, $1, 2, 3, 4, \ldots, (n-1)$ steps.
- Hold on, if the element before the one we are considering is less then we can stop!
  - That means we only have to consider all of the preceding elements in the *worst case*.

InsertionSort

```c
#include "insertionsort.h"
#include <stdint.h>

void insertionSort(uint32_t a[], int length) {
  for (int i = 1; i < length; i += 1) {
    int j = i;
    uint32_t tmp = a[i];
    while (j > 0 && a[j - 1] > tmp) {
      a[j] = a[j - 1];
      j -= 1;
    }
    a[j] = tmp;
  }
  return;
}
```

# Fourth idea…

- Suppose we have an unsorted array.
- Look at the first two elements, if the first is greater than the second then *swap them*.
- Move on to the second and third elements, swap if they are out of order.
  - Continue on with this procedure up to the last pair of elements in the array.
- What do we know?
  - The last element of the array is the largest! So we can now ignore it.
  - Repeat this procedure on the first $n - 1$ elements, then on $n - 2$, …
- But wait, what happens if you look at a subarray and we make *no swaps*?
  - We know that the array is sorted!
- In the best case, we only examine $n - 1$ pairs.

```python
def bubble(a):
    for i in range(len(a) - 1):
        j = len(a) - 1
        while j > i:
            if a[j] < a[j - 1]:
                a[j], a[j - 1] = a[j - 1], a[j]
            j -= 1
    return a
```

Without the hack!

# BubbleSort

# We can immediately do better!

- We observe that swapping means that at least two elements were *out of order*.
- This implies that if we examine every pair, and none of them are inverted then the array is sorted.
- The best case is thus $O(n)$.

```c
#include "bubblesort.h"
#include <stdbool.h>
#include <stdint.h>

void bubbleSort(uint32_t a[], int length) {
  bool swapped;
  do {
    swapped = false;
    for (int i = 1; i < length; i += 1) {
      if (a[i - 1] > a[i]) {
        SWAP(a[i - 1], a[i]);
        swapped = true;
      }
    }
    length -= 1;
  } while (swapped);
  return;
}
```

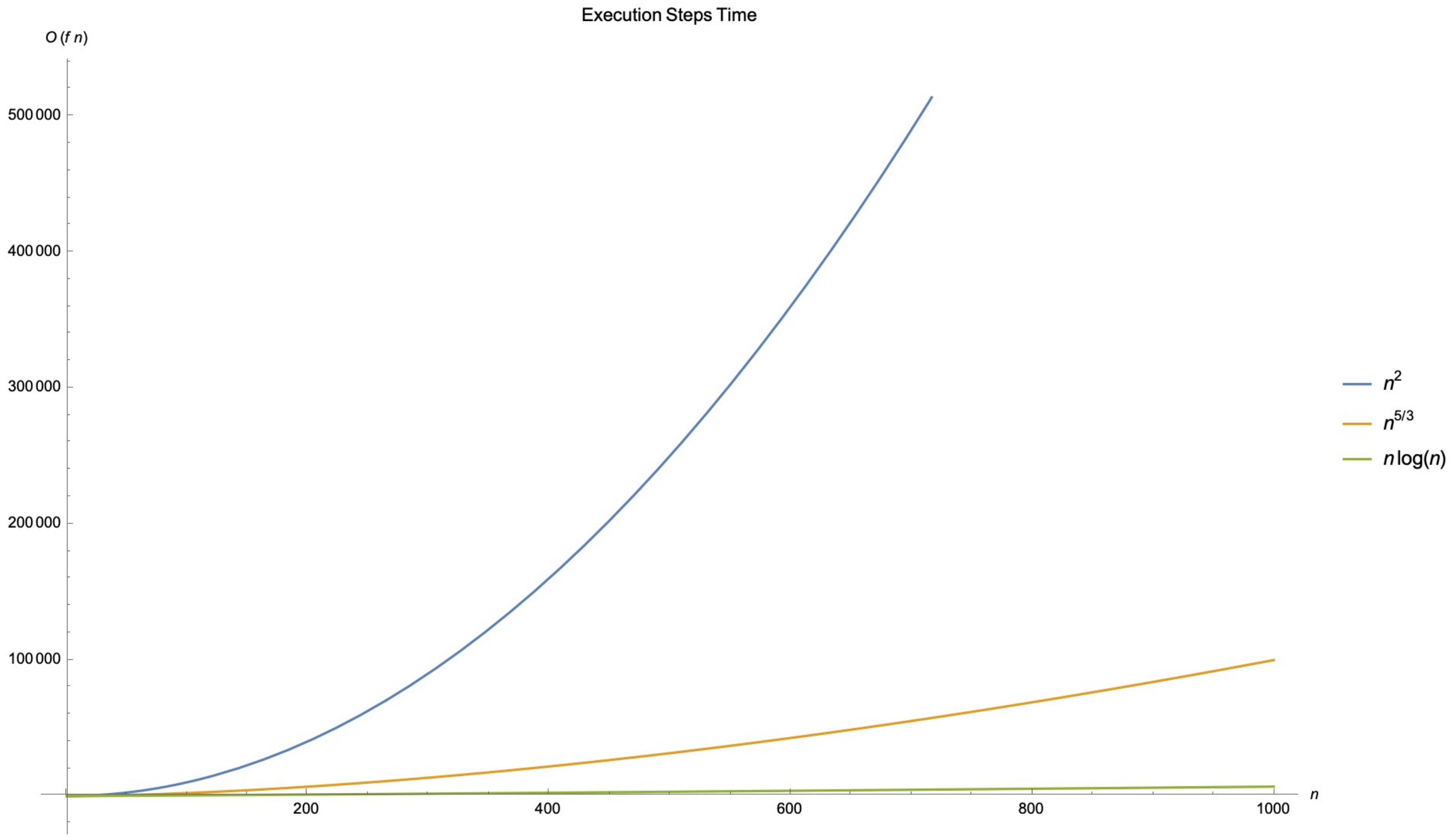
# Is that the best we can do?

- No, as you will learn in later classes we can prove that we can sort in time proportional to $n \log n$.
  - At the risk of annoying my colleagues by jumping ahead, let's do a little thought experiment.
- Suppose I examine my array in disjoint pairs: `a[0]`, `a[1]`, then `a[2]`, `a[3]`, and so forth.
  - We have $\frac{n}{2}$ such pairs, but we have to look at both so let's call that $n$.
  - We put the elements of every pair in order (length 2), and call that a *run*.
- Now let's take a pair of runs, each of length 2, and merge them into runs of length 4.
  - Again, this will take us time proportional to $n$.
- How many times can we double 1 before we exceed $n$? That's easy, $\log n$.
- Thus, our sort finishes in time proportional to $n \log n$. Our sort has a name: *Merge Sort*.

# Comparative Sorting Algorithms

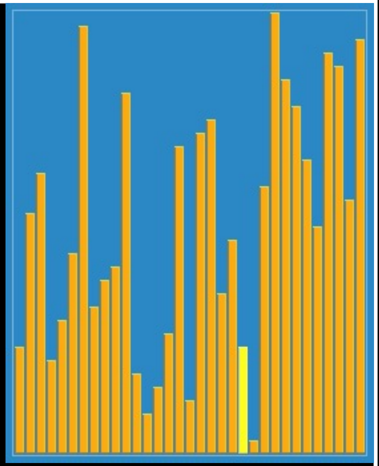
- Here are some $O(n^2)$ sorting algorithms:
  - Bubble Sort
  - Insertion Sort
  - Selection Sort
  - Quick Sort (worst case)
- Shell sort is an $O(n^{\frac{5}{3}})$ sorting algorithm.
  - It is surprisingly good!
- Here are some $O(n \log n)$ sorting algorithms:
  - Merge Sort
  - Heap Sort
  - Quick Sort (average case)

# Comparison of Execution Times



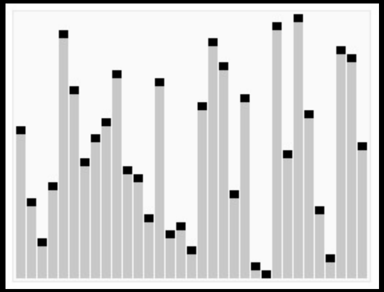
```python
def gap(n):
    while n > 1:
        n = 1 if n <= 2 else 5 * n // 11
        yield(n)

def shellSort(s):
    for step in gap(len(s)):
        for i in range(step, len(s)):
            for j in range(i, step - 1, -step):
                if s[j] < s[j - step]:
                    s[j], s[j - step] = s[j - step], s[j]
    return s
```



# Shell Sort

```python
def QSort(a):
    if len(a) < SMALL:
        return shellSort(a)
    else:
        pivot = (a[0] + a[len(a) / 2] + a[len(a) -1]) / 3
        left  = [ _ for _ in a if  _ <  pivot ]
        mid   = [ _ for _ in a if  _ == pivot ]
        right = [ _ for _ in a if  _ >  pivot ]
        return QSort(left) + mid + QSort(right)
```



QuickSort

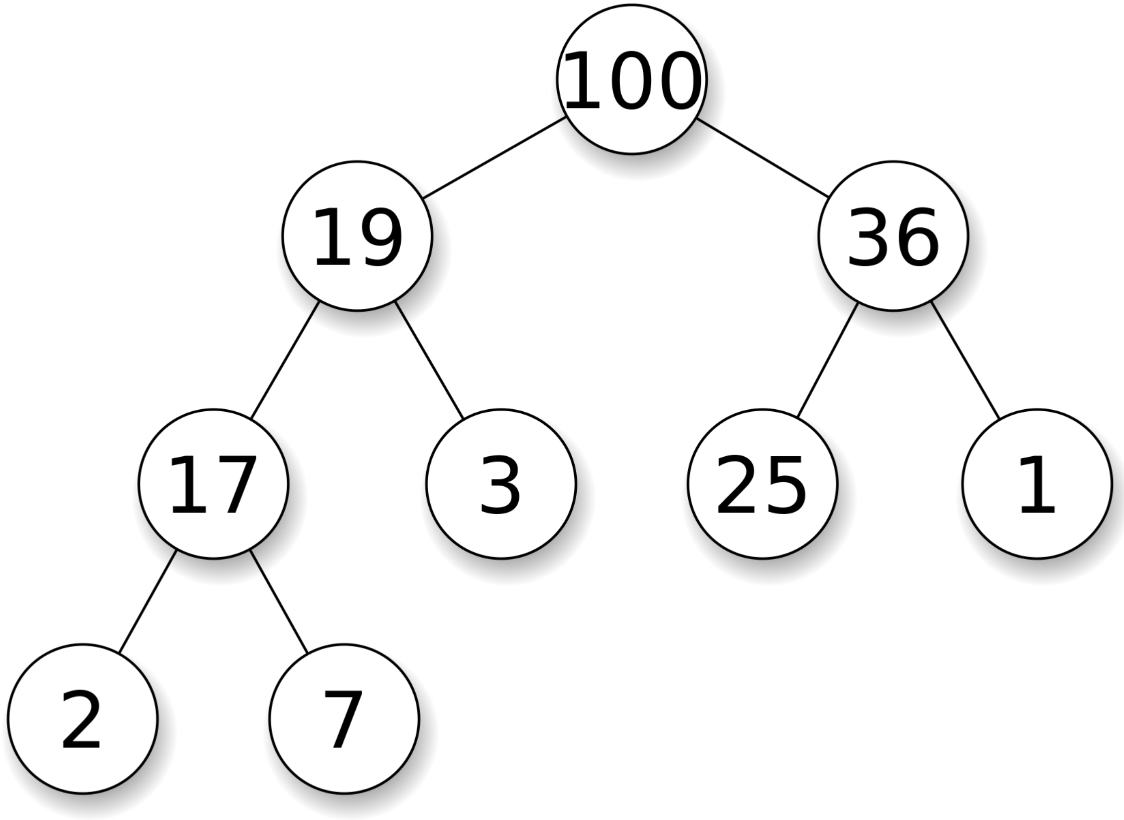
# Heaps

There are many types of heaps:

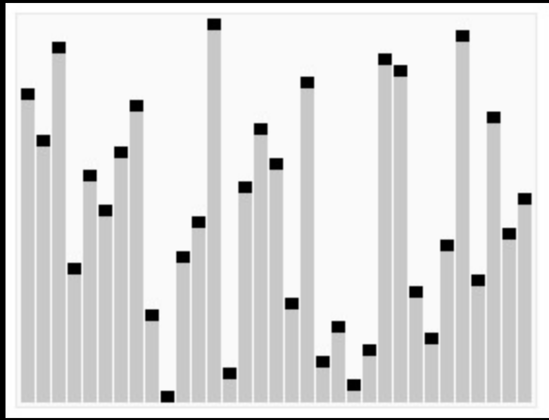
- Minimum/Maximum Heaps
- Leftist Heaps
- Binomial Heaps
- Fibonacci Heaps
- Brodal Heaps
- Radix Heaps, …
- Uriah Heeps

# A (Max) Heap

- A single node is a heap.
- It is a heap if the parent is a heap and the trees rooted at both children are heaps.
- A parent’s value (key) is greater than that of either child.
- There is no order among the children.



```
      subroutine HEAP_SORT( SEQUENCE, FIRST, LAST )
C Heap_Sort - Sort an array of items using a generalized heap-sort
C             technique.
      integer    SEQUENCE(*), FIRST, LAST
      integer    LEAF
      call BUILD_HEAP(SEQUENCE, FIRST, LAST)
      do LEAF = LAST, FIRST + 1, -1.
         call SWAP(SEQUENCE(FIRST), SEQUENCE(LEAF))
         call FIX_HEAP(SEQUENCE, FIRST, LEAF - 1)
      end do
      end
```



# Heap Sort

```fortran
      subroutine BUILD_HEAP( SEQUENCE, FIRST, LAST )
C
C Build_Heap - Construct a heap from an array of items.
C
      integer   SEQUENCE(*), FIRST, LAST
      integer   FATHER
      do FATHER = (LAST / 2), FIRST, -1)
         call FIX_HEAP(SEQUENCE, FATHER, LAST)
      end do
      end
```

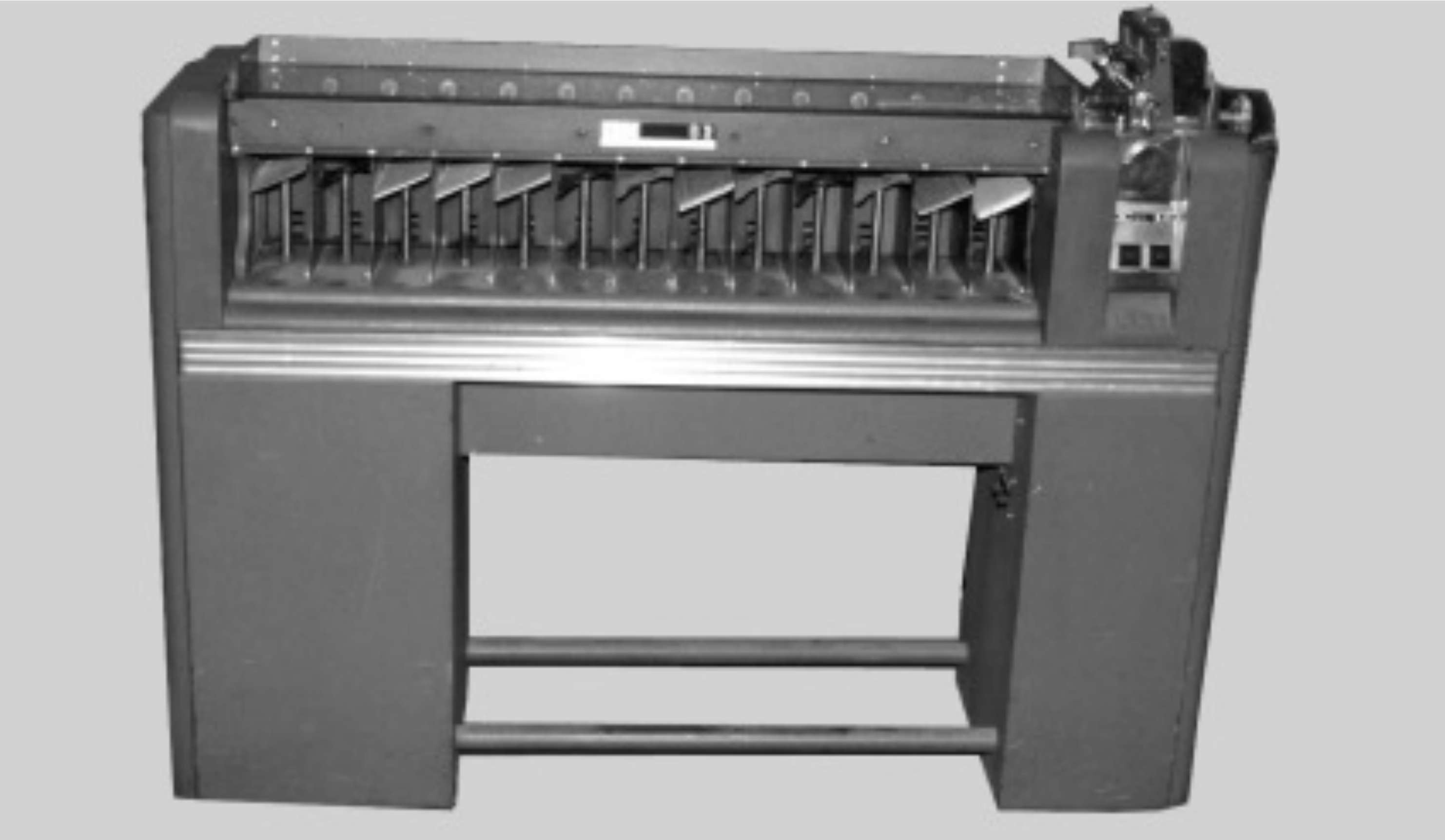
Build Heap

```fortran
      subroutine FIX_HEAP( SEQUENCE, FIRST, LAST )
C
C Fix_Heap - Repair a subheap.
C
      integer    SEQUENCE(*), FIRST, LAST
      integer    FATHER, GREAT
      integer    MAX_CHILD
      logical    FOUND
      FOUND  = .false.
      FATHER = FIRST
      GREAT  = MAX_CHILD(SEQUENCE, FATHER, LAST)
      do while ((FATHER .le. (LAST / 2)) .and. .not. FOUND)
         if (SEQUENCE(FATHER) .lt. SEQUENCE(GREAT)) then
            call SWAP(SEQUENCE(FATHER), SEQUENCE(GREAT))
            FATHER = GREAT
            GREAT  = MAX_CHILD(SEQUENCE, FATHER, LAST)
         else
            FOUND = .true.
         end if
      end do
      end
```

# Fix Heap

```fortran
      integer function MAX_CHILD( SEQUENCE, FIRST, LAST )
C
C Max_Child - Find the largest of the two children of a node rooted at
C             FIRST.
C
      integer SEQUENCE(*), FIRST, LAST
      integer LEFT, RIGHT
      LEFT  = FIRST * 2
      RIGHT = LEFT  + 1
      MAX_CHILD = LEFT
      if (RIGHT .le. LAST) then
         if (SEQUENCE(RIGHT) .gt. SEQUENCE(LEFT)) then
            MAX_CHILD = RIGHT
         end if
      end if
      end
```

# Max Child



# Can I do even better?

- Using *comparisons*, the answer is *no*.
- But, if we make some assumptions about the encoding then you can use a *Radix Sort*.
    - It runs in time proportional to the number of digits in the key times the number of records.
    - It was invented for the mechanical sorting of punched cards.

# Summary

- Sorting is a *fundamental* operation, so doing efficiently has a huge impact on computing.
- Analysis of algorithm complexity is an important topic, and you will be a lot of it in your advanced classes.
  - A better algorithm makes a lot more difference than a faster computer.

- You will encounter many instances in your career where you need to employ a sorting algorithm, and the best one will depend on the circumstances and the data structures employed.