Algorithms in Systems Engineering
ISE 172

Lecture 11

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - Chapter 6
- References
  - CLRS Chapter 6
  - D.E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching* (Third Edition), 1998.
  - R. Sedgewick, *Algorithms in C++* (Third Edition), 1998.

The Sorting Problem

- We will now undertake a more formal study of algorithms for the *sorting problem*.
- This problem is fundamental to the study of algorithms.
- Most often, the items to be sorted are individual *records*, usually consisting of a *key* and related *satellite data*.
- Recall our previous definition (slightly generalized here).
  Input: A sequence of n records $a_1, a_2, \ldots, a_n$.
  Output: A reordering $a'_1, a'_2, \ldots, a'_n$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.
- Note that the records can be anything for which a "$\leq$" operator can be defined (usually by comparing the specified key).
- We may be interested in sorting the same list in more that one way.
- What are some contexts in which sorting is important?

Sorting Algorithms

- It is safe to say that there are more algorithms for sorting than any other single problem.
- There are so many fundamentally different ways of solving this problem that entire books have been devoted to the topic.
- It is known that the running time of any comparison-based sorting algorithm is in $\Omega(n \lg n)$ (why?).
- Any algorithm whose worst-case running time matches this lower bound is said to be *asymptotically optimal* or just *optimal*.
- Many of the known algorithms, including merge sort, are optimal.
- However, this does not necessarily translate into good performance in practice.

Measuring Performance of Sorting Algorithms

- We have already seen that *comparisons* are the fundamental operation used in search algorithms.
- Counting the number of comparisons used for a search algorithm can tell us something about performance in practice.
- In sorting, we generally consider two fundamental operations.
  - *comparisons* and
  - *swaps*
- By counting these operations, we can usually get a pretty good idea of how a given algorithm will perform in practice.
- We will see later that in practice, the number of opeation types is really a bit higher than this.

Worst Case Versus Average Case

- There are many sorting algorithms that achieve the worst-case bound of $n \log n$ for comparison-based sorting.
- These algorithms can perform in wildly different ways in practice.
- It's important to select the right algorithm for a particular application.
- For this purpose, empirical testing is key.
- However, we must consider the right test set!

Dependence on Properties of the Input

- The practical behavior of sorting algorithms is highly dependent on certain properties of the input.
- These special kinds of inputs can produce very different behavior with different sorting algorithms.
  - Almost sorted list
  - Reverse sorted list
  - List with only a few unique values
- When testing sorting algorithms, we have to be careful not just to test with “random” inputs.
- Although random input would seem to be the worst case, some algorithms that perform well on random inputs, but not on the above types.

Properties of Sorting Algorithms

- In addition to worst-case running time, there are a few important properties of sorting algorithm that we may need to consider.
  - A *stable* sorting algorithm is one that leaves duplicate keys in the same relative order that they were in the original list.
  - This is an important property if you want to be able to sort on multiple keys.
  - Another important consideration is whether the algorithm sorts *in place*, i.e., does not have to allocate too much extra memory.
  - Finally, we might consider how well the algorithm performs on special types of arrays.
- The sorting algorithm you choose may depend on what you expect the data to look like, e.g., is it "almost sorted."
- The basic operations performed in sorting are comparison and exchange.
- The relative cost of these operations may also help determine the type of sort that is most appropriate.

Elementary Sorting Methods

- Most straightforward sorting algorithms have a running time in $O(n^2)$.
- *Selection sort* is perhaps the easiest to understand.
  - Selection sort consists of n passes through the list.
  - In pass i, the largest element in positions i through n is swapped with the element in position i.
- *Bubble sort* is another simple algorithm.
  - Bubble sort also consists of n passes through the list.
  - During each scan through the list, contiguous pairs of elements are compared and swapped if they are out of order.
- These simple algorithms are considered *nonadaptive* because the sequence of steps is not affected by the initial ordering of the list.
- The number of operations is $\Theta(n^2)$ regardless of the input.
- Naturally, there are ways to make these algorithms slightly adaptive (stop once you detect the list is sorted.

Bubble Sort Code

Here is a simple bubble sort code:

```
def bubble_sort(list):

    for i in range(0, len(list) - 1):
        swap_test = False
        for j in range(0, len(list) - i - 1):
            if list[j] > list[j + 1]:
                list[j], list[j + 1] = list[j + 1], list[j]  # swap
                swap_test = True
        if swap_test == False:
            break
```

What is role of the variable `swap_test`?

Selection Sort Code

Here is a simple selection sort code:

```
def selection_sort(list):
    for i in range(0, len (list)):
        minimum = i
        for j in range(i + 1, len(list)):
            if list[j] < list[minimum]:
                minimum = j
        list[i], list[minimum] = list[minimum], list[i]  # swap
```

How does it compare to bubble sort?

Adaptive Algorithms

- We can improve slightly on the performance of selection and bubble sort by using a closely related variant called *insertion sort*.
  - We maintain the invariant that at iteration k, the first k items are in sorted order.
  - At iteration $k+1$, we insert item $k+1$ into the sorted list by searching sequentially for the insertion point.
- Insertion sort is *adaptive*, since we stop when we reach the correct point of insertion.
  - It performs well on lists that are “almost sorted”.
  - It performs poorly in many cases and still takes $\Theta(n^2)$ in the worst case.
- Insertion sort can be performed in place and is stable.

Inserion Sort Code

Here is a simple selection sort code:

```
def insertion_sort(list):
    for i in range(1, len(list)):
        save = list[i]
        j = i
        while j > 0 and list[j - 1] > save:
            list[j] = list[j - 1]
            j -= 1
        list[j] = save
```

How does it differ from selection sort?

Visualizing sorting algorithms

- On the following slides, we'll look at some static visualization of sorting algorithms generated by the code available from

`http://sortvis.org`

- In the visualizations, the ordering is by shade of grey.
- Time goes from left to right.
- Reading vertically, we see the order of the items at a given point in time.
- We can also follow a particular items path.
- This gives a very dramatic view of the differences between algorithms.
- We can easily see their performance.

Bubble Sort

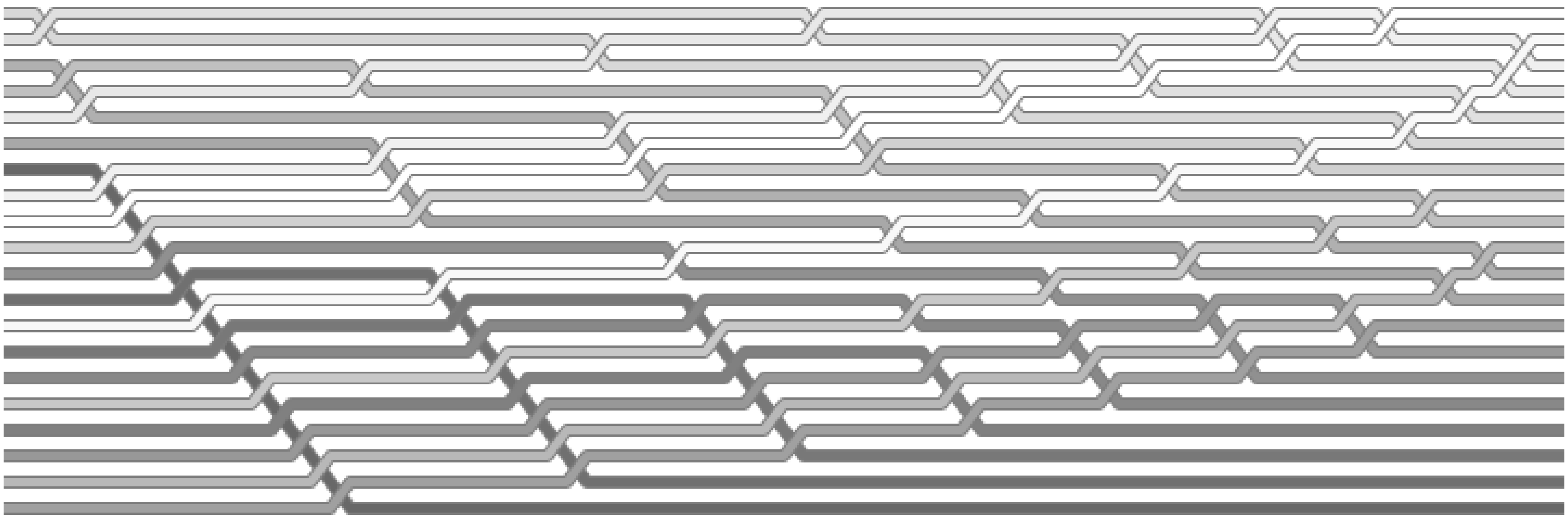

Figure 1: Bubble sort visualization

Selection Sort

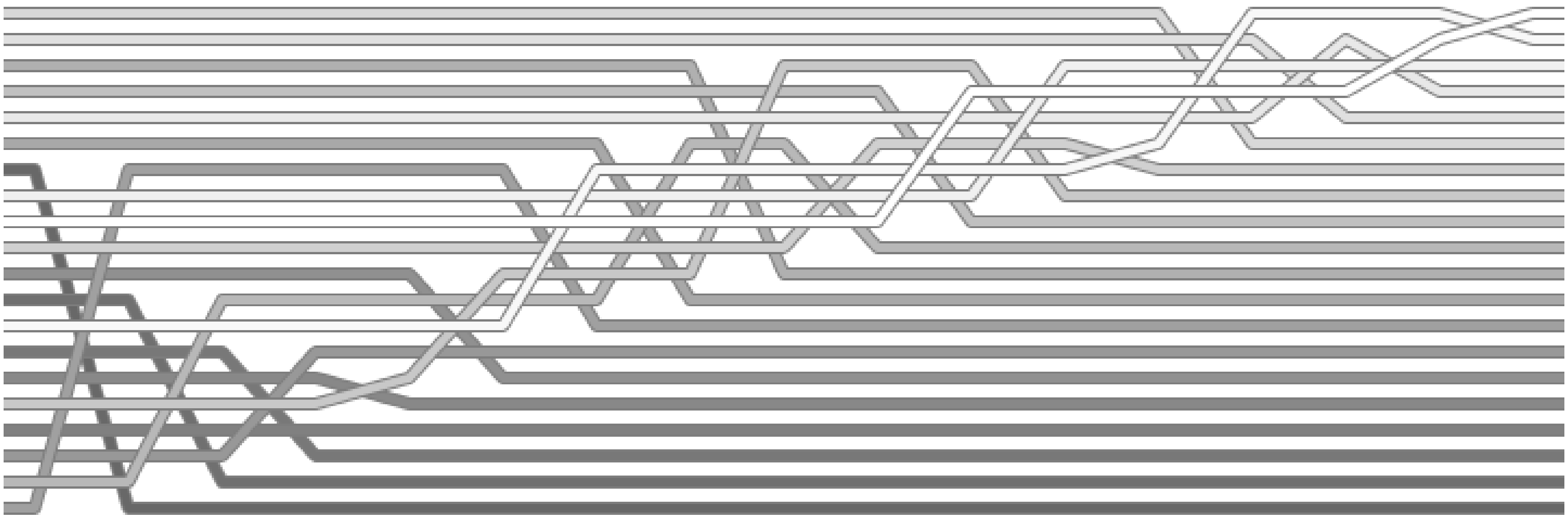

Figure 2: Selection sort visualization

Insertion Sort

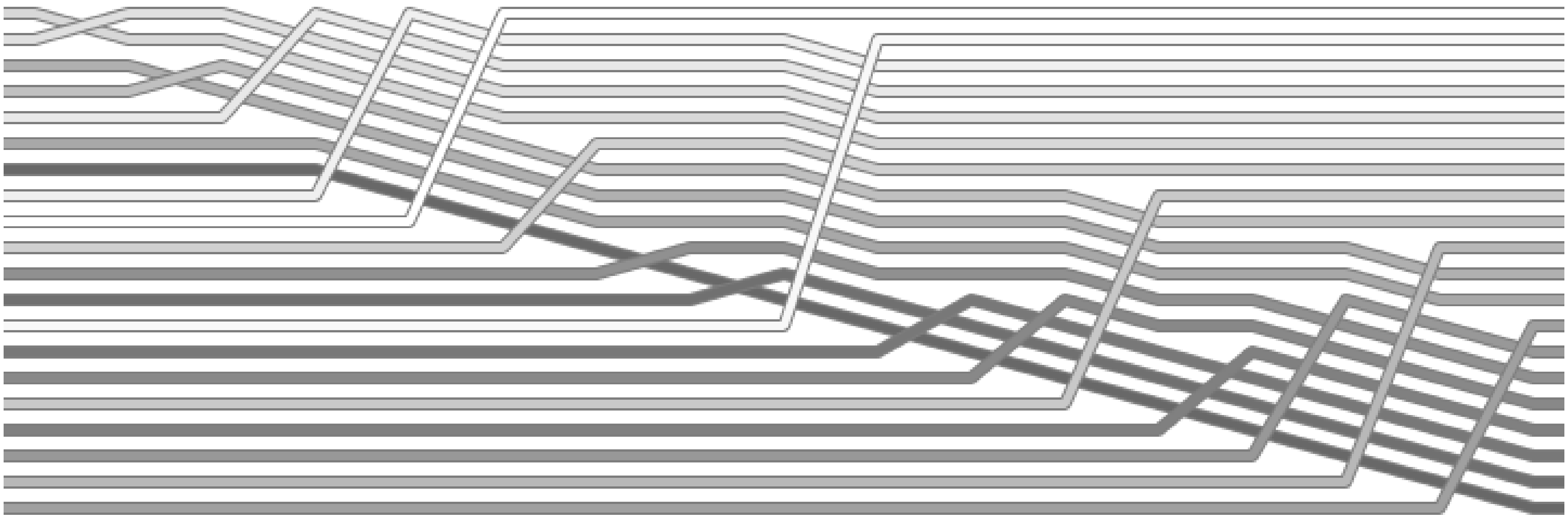

Figure 3: Insertion sort visualization

Performance of Elementary Sorting Algorithms

- Selection sort uses about $n^2/2$ comparisons and n exchanges.
- Insertion sort uses about $n^2/4$ comparisons and $n^2/4$ moves on average and twice that many at worst.
- Bubble sort uses about $n^2/2$ comparisons and $n^2/2$ exchanges on the average and in the worst case.
- Insertion and bubble sort use a linear number of comparisons and exchanges for files that have a constant number of *inversions* per element.
- Insertion sort uses a linear number of comparisons and exchanges for files having at most a constant number of elements having more than a constant number of inversions.

Comparing Running Times

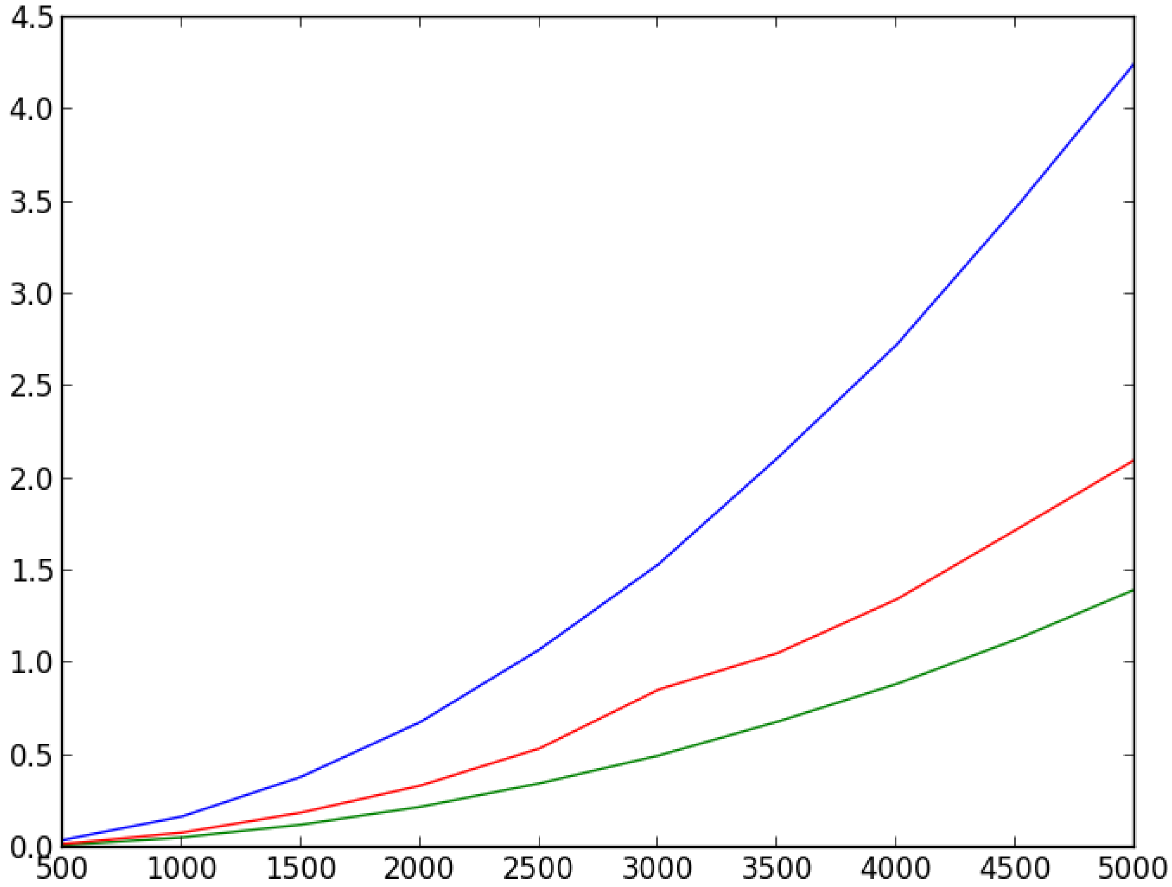

Figure 4: Comparing naive sorting algorithms by running time (pure Python implementation) Green = Selection, Red = Insertion, Blue = Bubble

Comparison Analysis for Times

- Notice that they all have the same running time function, but different consants.
  - Selection sort is fastest
  - Bubble sort is slowest
  - Insertion sort is in the middle
- Can you conjecture why the ordering is like this?

Comparing Number of Operations

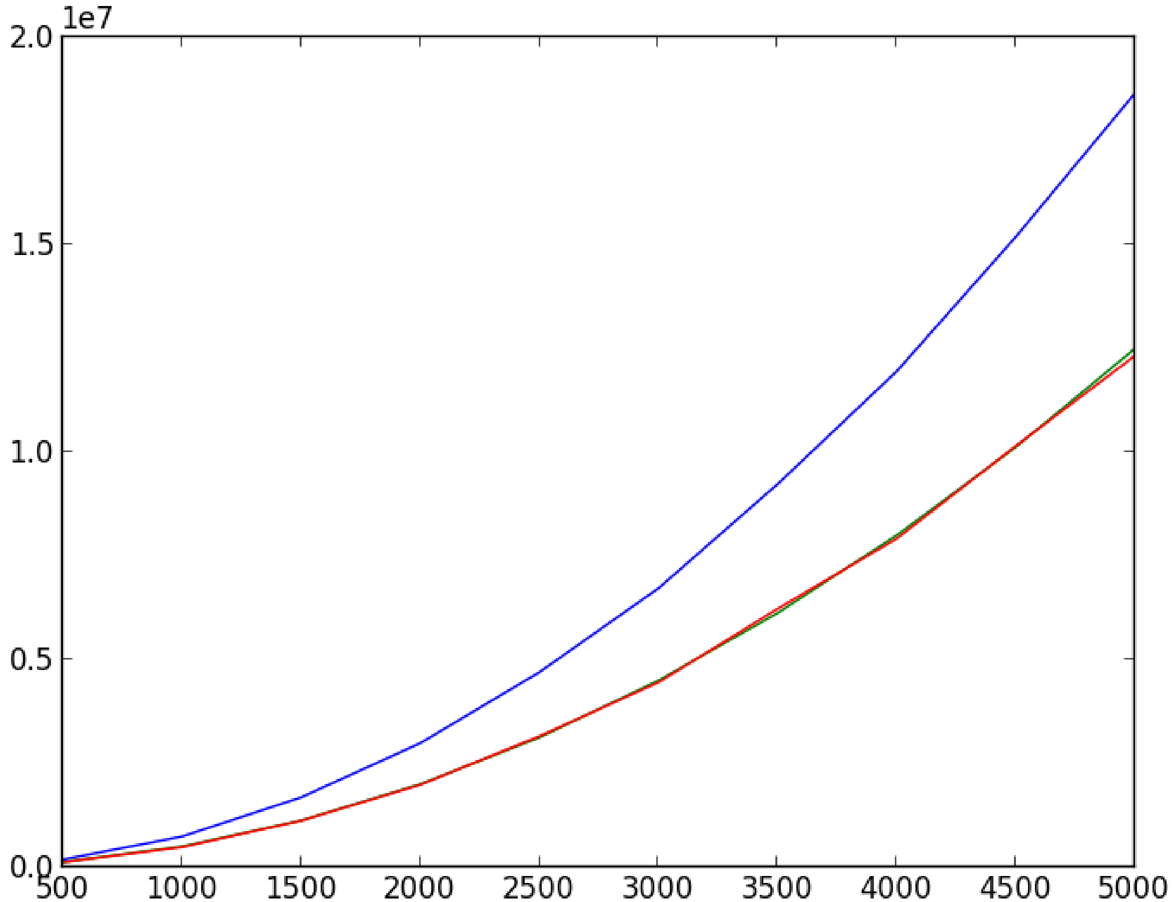

Figure 5: Comparing naive sorting algorithms by number of operations (pure Python implementation) Green = Selection, Red = Insertion, Blue = Bubble

Comparison Analysis for Operations

- Notice that the growth functions are the same as before, but the constants are different.
- Why are selection and insertion identical?

```
selection_sort took 10390.580ms
compare was called 12497500 times
swap was called 5000 times
Total number of operations was 12502500
insertion_sort took 12277.297ms
compare was called 6204693 times
shift_right was called 6199699 times
assign was called 4999 times
Total number of operations was 12409391
```