Algorithms in Systems Engineering
ISE 172

Lecture 12

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - Chapter 6
- References
  - CLRS Chapter 7
  - D.E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching* (Third Edition), 1998.
  - R. Sedgewick, *Algorithms in C++* (Third Edition), 1998.

Optimal Algorithms

- In Lecture 7, we saw *merge sort*.
  - Merge sort is asymptotically optimal and stable.
  - However, it cannot be performed in place.
- Later in this lecture, we'll introduce a more sophisticated recursive algorithm called *quick sort*, which is based on partitioning.
  - Quick sort is also $\Theta(n^2)$ in the worst case, but is $\Theta(n \lg n)$ on average.
  - However, it is unstable and can result in a large call stack and poor performance in common special cases if not implemented carefully.
- Another alternative, which is optimal and stable is *heap sort*, which sorts using a priority queue data structure.

Priority Queues and Sorting

- To understand heap sort, we must introduce a new data structure called a *priority queue*.
  - A priority queue is a data structure for maintaining a list of items that have associated *priorities*.
  - It is like a queue, but items might have their priorities changed so we need to be able to shuffle items around efficiently.
  - The usual operations are
    * construct a queue from a list of items.
    * find the item with the highest priority.
    * insert an item.
    * delete an item.
    * change the priority of an item.
- Note that any implementation of a priority queue can be used to sort a list of items.
  - Put the items in a priority queue.
  - Delete the maximum item n times.

Heap Sort

- We will see later an implementation of priority queues for which each of the major operations has a running time of $O(\log n)$.
- This immediately yields an algorithm that runs in $O(n \log n)$.
- Neverthless, we will see it is not very competitive in practice.

Quicksort

- We now discuss a sorting algorithm called *quicksort* that is a recursive algorithm like mergesort.
- The basic quicksort algorithm is as follows.
  - Choose a partition element.
  - Partition the input array around that element to obtain two subarrays.
  - Recursively call quick sort on the two subarrrays.
- Here is pseudo-code for the algorithm.

```
def quicksort(data, beg, end):
    if end <= beg: return
    i = partition(data, beg, end)
    quicksort(data, beg, i-1)
    quicksort(data, i+1, end)
```

Partitioning

- One big advantage of quicksort is that the partitioning (and hence the entire algorithm) can be performed in place.
- Here is an in place implementation of the partitioning function.

```
def partition(data, beg, end):
    i = beg
    j = end - 1
    v = data[end]
    while True:
        while data[i] < v: i += 1
        while v < data[j]:
            if j == beg: break
            j -= 1
        if i >= j: break
        exchange(i, j)
    exchange(i, end)
    return i
```

Analyzing Quicksort

- Questions to be answered
  - How do we prove the correctness of quick sort?
  - Does quicksort always terminate?
  - Can we do some simple optimization to improve performance?
  - What are the best case, worst case, and average case running times?
  - How does quicksort perform on special files, such as those that are almost sorted?

Importance of the Partitioning Element

- Note that the performance of the algorithm depends primarily on the chosen partition element.
- Some questions
  - What is the "best" partition element to select?
  - What is the running time if we always select the "best" partition element?
  - What is the "worst" partition element to select?
  - What is the running time in the worst case?
  - What is the running time in the average case?

Choosing the Partitioning Element

- We would like the partition element to be as close to the middle of the array as possible.
- However, we have no way to ensure this in general.
- If the array is randomly ordered, any element will do, so choose the last element (this was our original implementation).
- If the array is almost sorted, this will be disastrous!
- To even the playing field, we can simply choose the partition element randomly.
- How can we improve on this?

More Simple Optimization

- Note that the check `if (j == l)` in the partition function can be a significant portion of the running time.
- This check is only there in case the partition element is the smallest element in the array.
- Here again, we can use the concept of a sentinel.
- If we place a sentinel at the beginning of the array, we avoid this check.
- Another approach is to ensure that the pivot element is never the smallest element of the array.
- If we use median-of-three partitioning, then the partition element can never be the smallest element in the array.

Average Case Analysis

- Assuming the partition element is chosen randomly, we can perform average case analysis.
- The average case running time is the solution to the following recurrence.

$$T(n) = n + 1 + \frac{1}{n} \sum_{1 \le k \le n} T(k-1) + T(n-k)$$

along with $T(0) = T(1) = 1$.

- Although this recurrence looks complicated, it's not too hard to solve.
- First, we simplify as follows.

$$T(n) = n + 1 + \frac{2}{n} \sum_{1 \le k \le n} T(k-1)$$

Average Case Analysis (cont.)

- We can eliminate the sum by multiplying both sides and subtracting the formula for T(n-1).

$$nT(n) - (n-1)T(n-1) = n(n+1) - (n-1)n + 2T(n-1)$$

- This results in the recurrence

$$nT(n) = (n+1)T(n-1) + 2n$$

- The solution to this is in $\Theta(n \lg n)$.
- In fact, the exact solution is more like $2n \ln n \approx 1.39 n \lg n$.
- This means that the average case is only about 40% slower than the best case!

Duplicate Keys

- Quicksort can be inefficient in the case that the file contains many *duplicate keys*.
- In fact, if the file consists entirely of records with identical keys, our implementation so far will still perform the same amount of work.
- The easiest way to handle this is to do *three-way partitioning*.
- Instead of splitting the file into only two pieces, we have a third piece consisting of the elements equal to the partition element.
- Implementing this idea requires a little creativity.
- How would you do it?

Small Subarrays

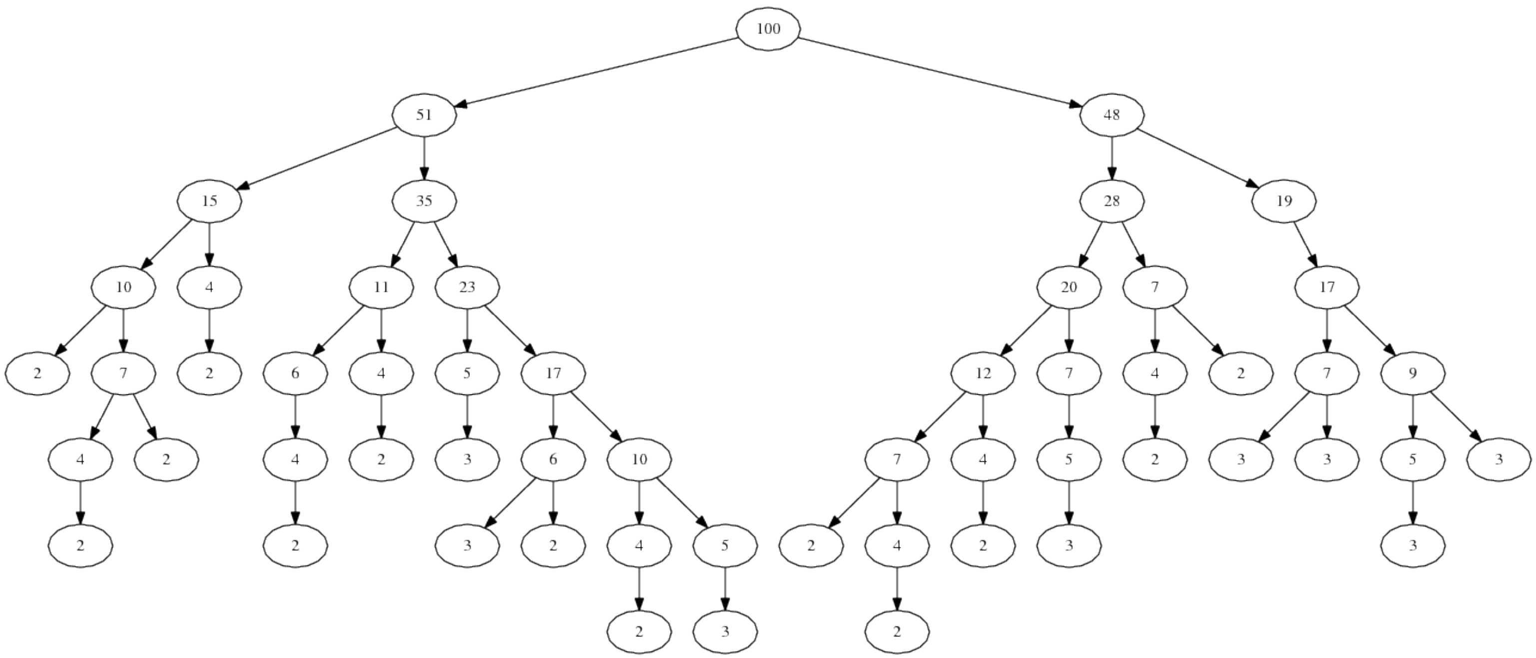
- Another way in which quicksort, as well as other recursive algorithms can be optimized is by sorting small subarrays directly using insertion sort.
- Empirically, subarrays of approximately 10 elements or smaller should be sorted directly.
- An even better approach is to simply ignore the small subarrays and then insertion sort the entire array once quick sort has finished.

Stack Depth

- An important consideration with any recursive algorithm is the depth of the call stack.
- Each recursive call means additional memory devoted to storing the values of local variables and other information.
- In the worst case, quicksort can have a stack as deep as the number of elements in the array.
- One way to deal with this is to ensure that the smaller of the two subarrays is processed first.
- This does not affect the correctness.
- Even this idea will not work in a truly recursive implementation without compiler optimization.
- The most memory-efficient implementation is a nonrecursive one that explicitly maintains the stack of subarrays to be sorted.

Picturing the Stack for Quick Sort

Here is a visualization of the call tree for a run of quicksort. The numbers in each circle represent the size of the sublist. This tree shows good balance.



Picturing the Stack for Quick Sort (Second Run)

Here is another visualization. Note that the tree is not so well banaced this time.

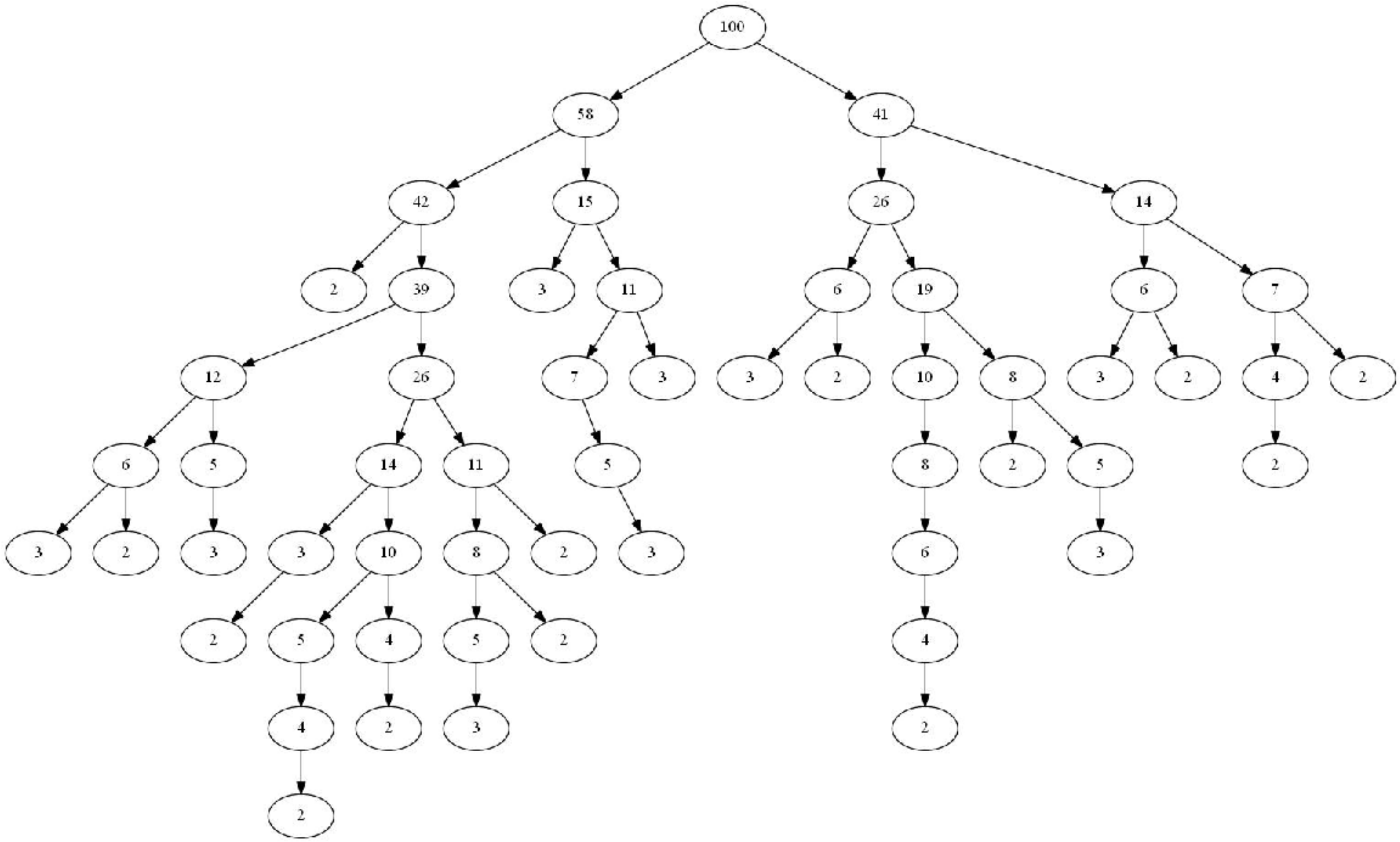


Figure 1: The call stack for sorting an array of size 100 by quick sort

A Nonrecursive Quicksort

```
def quicksort(data, beg, end)
   s = Stack()
   s.push((beg, end))
   while s.isEmpty():
      begin, end = s.pop()
      if end <= beg: continue
      i = partition(data, beg, end)
      if i - beg > end - i:
         s.push((beg, i-1))
         s.push((i+1, end))
      else:
         s.push((i+1, end))
         s.push((beg, i-1))
```

Quick Sort Visualization

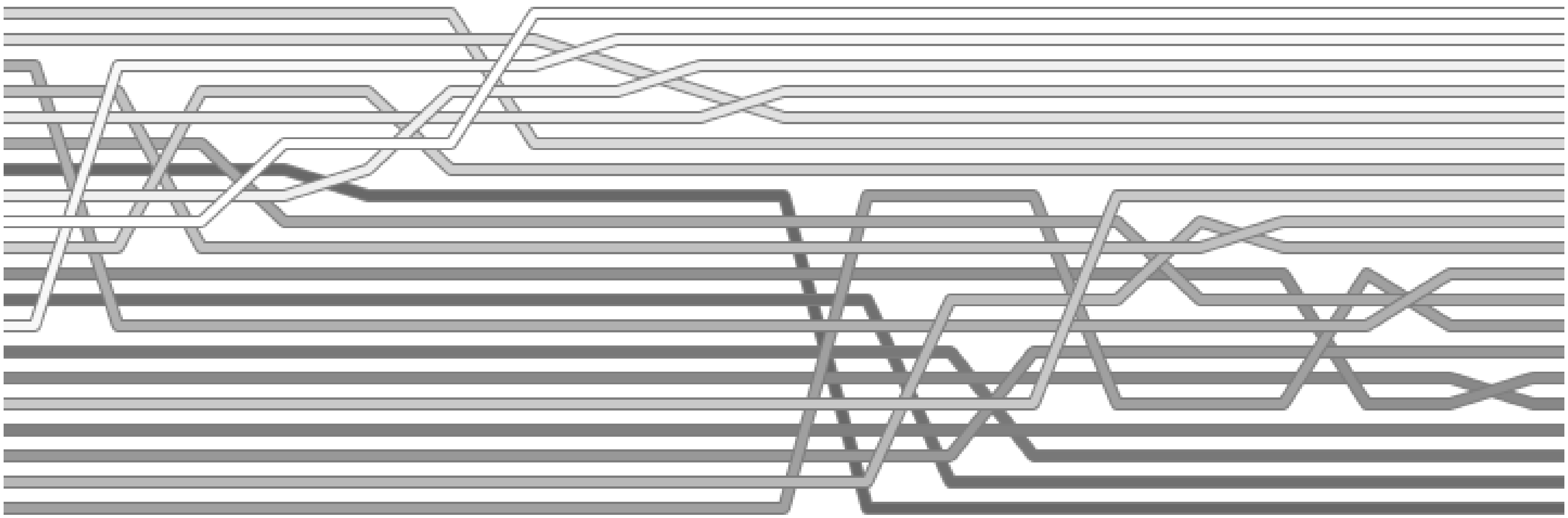


Figure 2: Quicksort visualization

Merge Sort Visualization

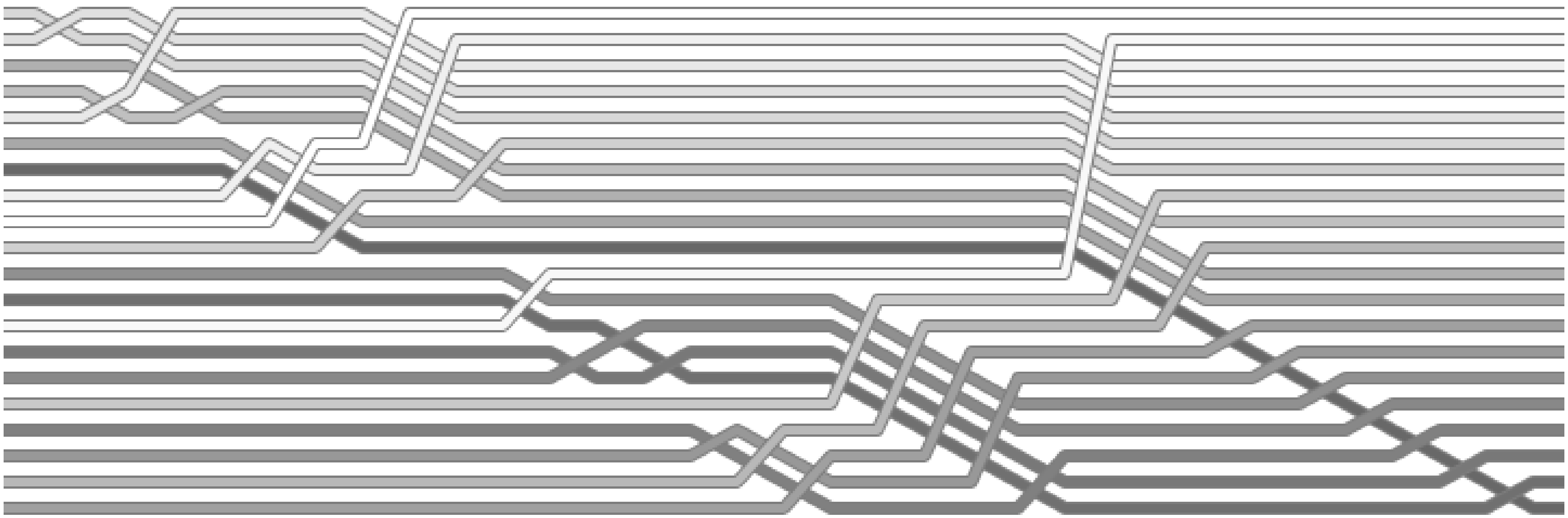


Figure 3: Merge sort visualization

Heap Sort

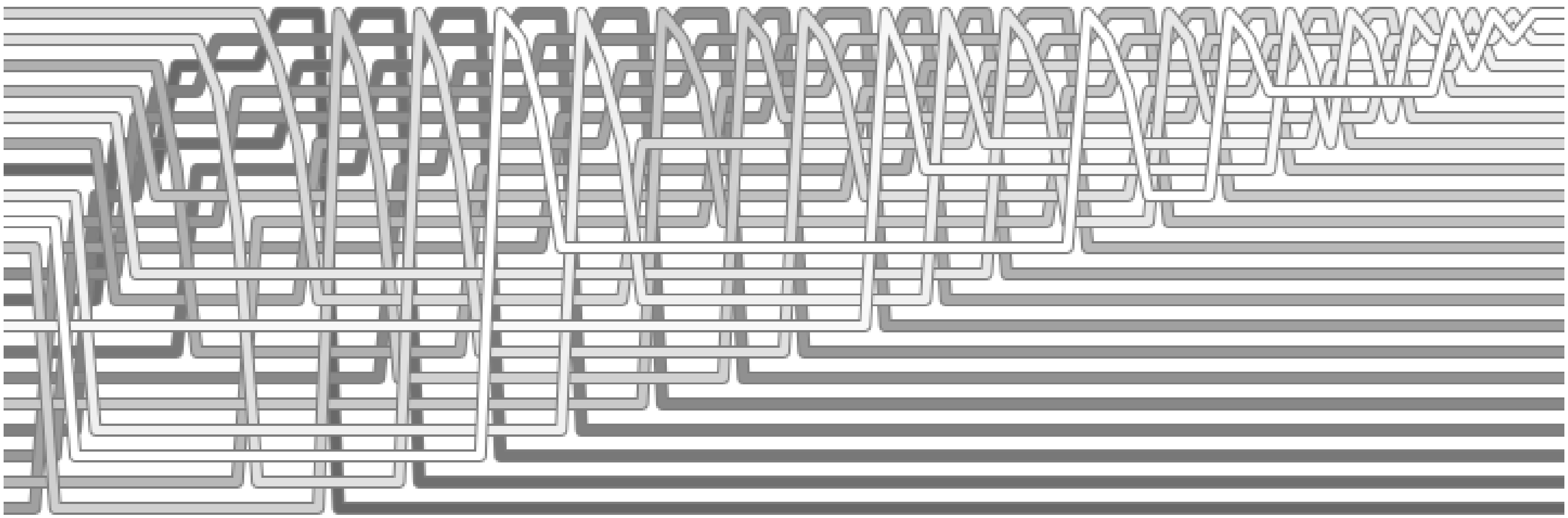


Figure 4: Heap sort visualization

Comparing Running Times

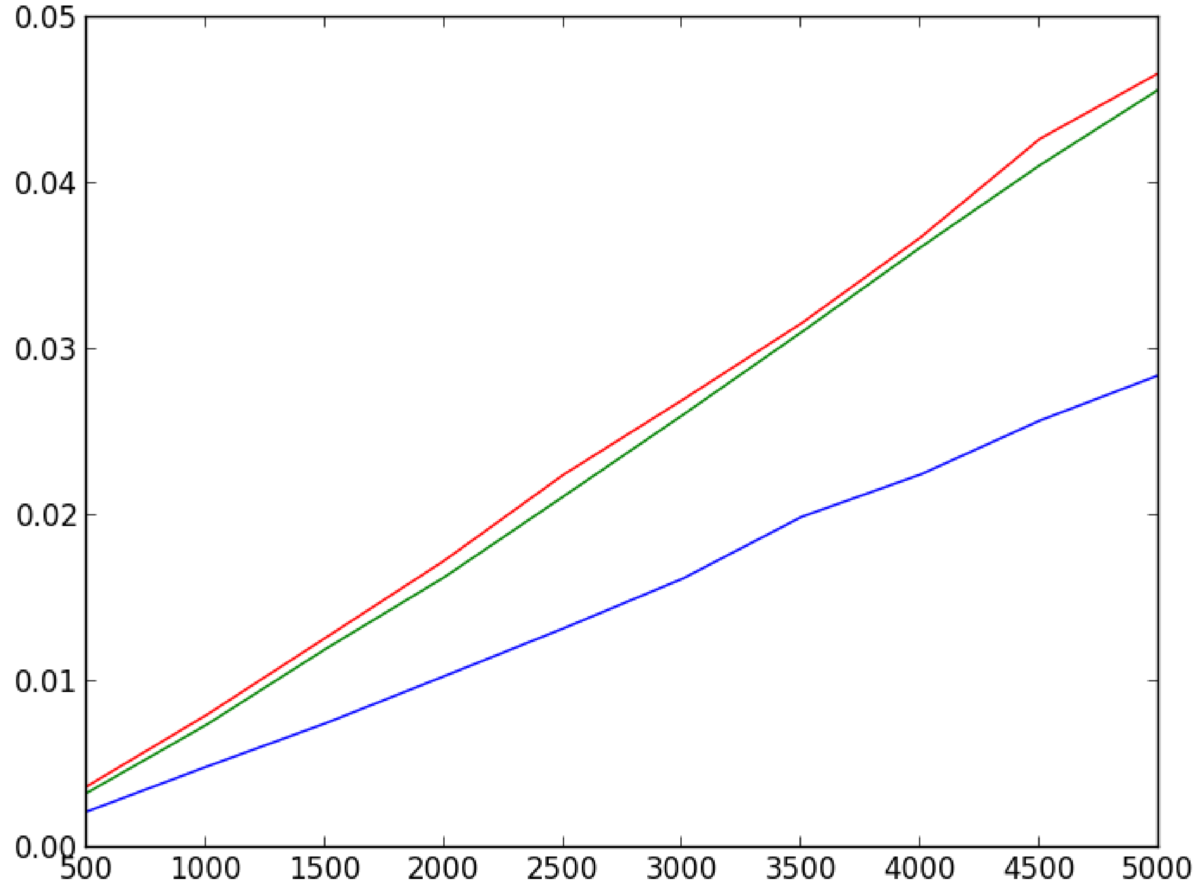


Figure 5: Comparing naive sorting algorithms by running time (pure Python implementation) Green = Heap, Red = Merge, Blue = Quick

Comparison Analysis for Times

- Notice that the running times all look linear.
- This is an “average case” analysis, so this is possible.
  - Quick sort is fastest
  - Heap and merge sort are each very similar.

Comparing Number of Operations

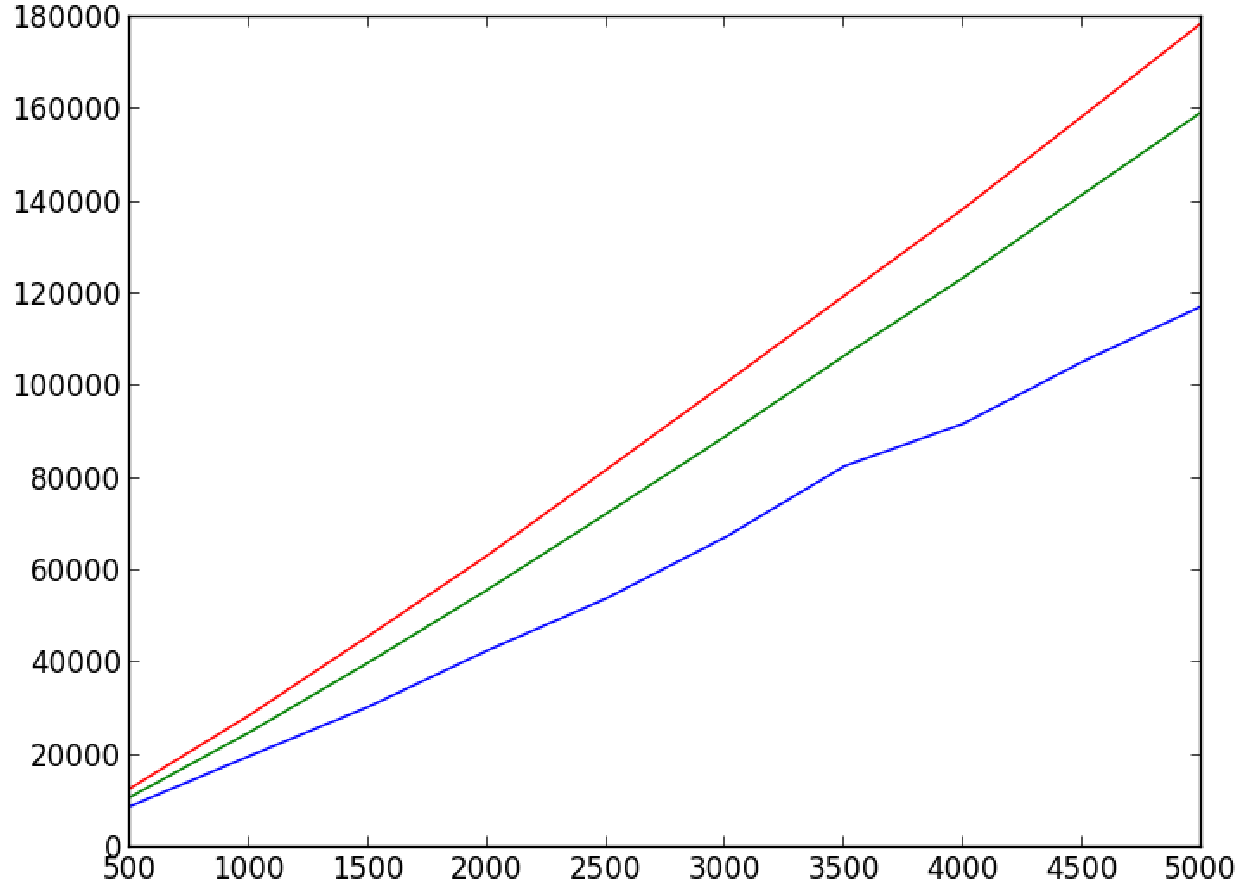


Figure 6: Comparing naive sorting algorithms by number of operations (pure Python implementation) Green = Selection, Red = Insertion, Blue = Bubble

Comparison Analysis for Operations

- The ordering of the algorithms by number of operation is the same as by time.
- Heap sort now has a bigger advantage.