Sort: Divide and Conquer

Contents

Divide and Conquer

- Divide the problem into sub-problems (recursively).
- Conquer sub-problems, which effectively solves the super problem.

	Merge	Tim	Quick	Intro
Best	$O(n \log n)$	$O(n)$	$O(n \log n)$	$O(n \log n)$
Worst	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Average	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n \log n)$

- Why do people ever want to use `QuickSort` then?

Merge Sort

- Divide a list into two sub-lists.
- Merge sub-lists into a super list in which all keys are sorted.

Source: `MergeSort.java`

```java
public class MergeSort<T extends Comparable<T>> extends AbstractSort<T> {
    private T[] temp;

    @Override
    public void sort(T[] array, int beginIndex, int endIndex) {
        if (beginIndex + 1 >= endIndex) return;
        int middleIndex = Utils.getMiddleIndex(beginIndex, endIndex);

        // sort left partition
        sort(array, beginIndex, middleIndex);
        // sort Right partition
        sort(array, middleIndex, endIndex);
        // merge partitions
        merge(array, beginIndex, middleIndex, endIndex);
    }
```

- Backup array: `T[] temp` .
- Static method: `Utils.getMiddleIndex()` .

```
private void copy(T[] array, int beginIndex, int endIndex) {
    int N = array.length;

    if (temp == null || temp.length < N)
        temp = Arrays.copyOf(array, N);
    else {
        N = endIndex - beginIndex;
        System.arraycopy(array, beginIndex, temp, beginIndex, N);
    }

    assignments += N;
}
```

- Base API: `Arrays.copyOf()` vs. `System.arraycopy()`.
- How often does the backup array `temp` get created?
- How many values are assigned to `temp`?

```java
/**
 * @param beginIndex  the beginning index of the 1st half (inclusive).
 * @param middleIndex the ending index of the 1st half (exclusive).
 * @param endIndex    the ending index of the 2nd half (exclusive).
 */
 private void merge(T[] array, int beginIndex, int middleIndex, int endIndex) {
     int fst = beginIndex, snd = middleIndex;
     copy(array, beginIndex, endIndex);

     for (int k = beginIndex; k < endIndex; k++) {
         if (fst >= middleIndex)                    // no key left in the 1st half
             assign(array, k, temp[snd++]);
         else if (snd >= endIndex)                  // no key left in the 2nd half
             assign(array, k, temp[fst++]);
         else if (compareTo(temp, fst, snd) < 0)    // 1st key < 2nd key
             assign(array, k, temp[fst++]);
         else
             assign(array, k, temp[snd++]);
     }
}
```

- How many comparisons and assignments?
- Can we reduce the number of assignments?

Quick Sort

- Pick a pivot key in a list.
- Exchange keys between left and right partitions such that all keys in the left and right partitions are smaller or bigger than the pivot key, respectively.
- Repeat this procedure in each partition, recursively.

Source: `QuickSort.java`

```
@Override
public void sort(T[] array, int beginIndex, int endIndex) {
    // at least one key in the range
    if (beginIndex >= endIndex) return;

    int pivotIndex = partition(array, beginIndex, endIndex);
    // sort left partition
    sort(array, beginIndex, pivotIndex);
    // sort right partition
    sort(array, pivotIndex + 1, endIndex);
}
```

```
protected int partition(T[] array, int beginIndex, int endIndex) {
    int fst = beginIndex, snd = endIndex;

    while (true) {
        // Find where endIndex > fst > pivot
        while (++fst < endIndex && compareTo(array, beginIndex, fst) >= 0);
        // Find where beginIndex < snd < pivot
        while (--snd > beginIndex && compareTo(array, beginIndex, snd) <= 0);
        // pointers crossed
        if (fst >= snd) break;
        // exchange
        swap(array, fst, snd);
    }

    // set pivot
    swap(array, beginIndex, snd);
    return snd;
}
```

Intro Sort

- Quicksort is the fastest on average.
- The worse-case complexity of Quicksort is $O(n^2)$.
- $\exists$ sorting algorithms with faster worst-case complexities than Quicksort.
- Quicksort for random cases and a different algorithm for the worst case.
- How to determine if Quicksort is meeting the worst-case?

Source: `IntroSort.java`

```java
public class IntroSort<T extends Comparable<T>> extends QuickSort<T> {
    private AbstractSort<T> engine;

    public IntroSort(AbstractSort<T> engine, Comparator<T> comparator) {
        super(comparator);
        this.engine = engine;
    }

    @Override
    public void resetCounts() {
        super.resetCounts();
        if (engine != null) engine.resetCounts();
    }
```

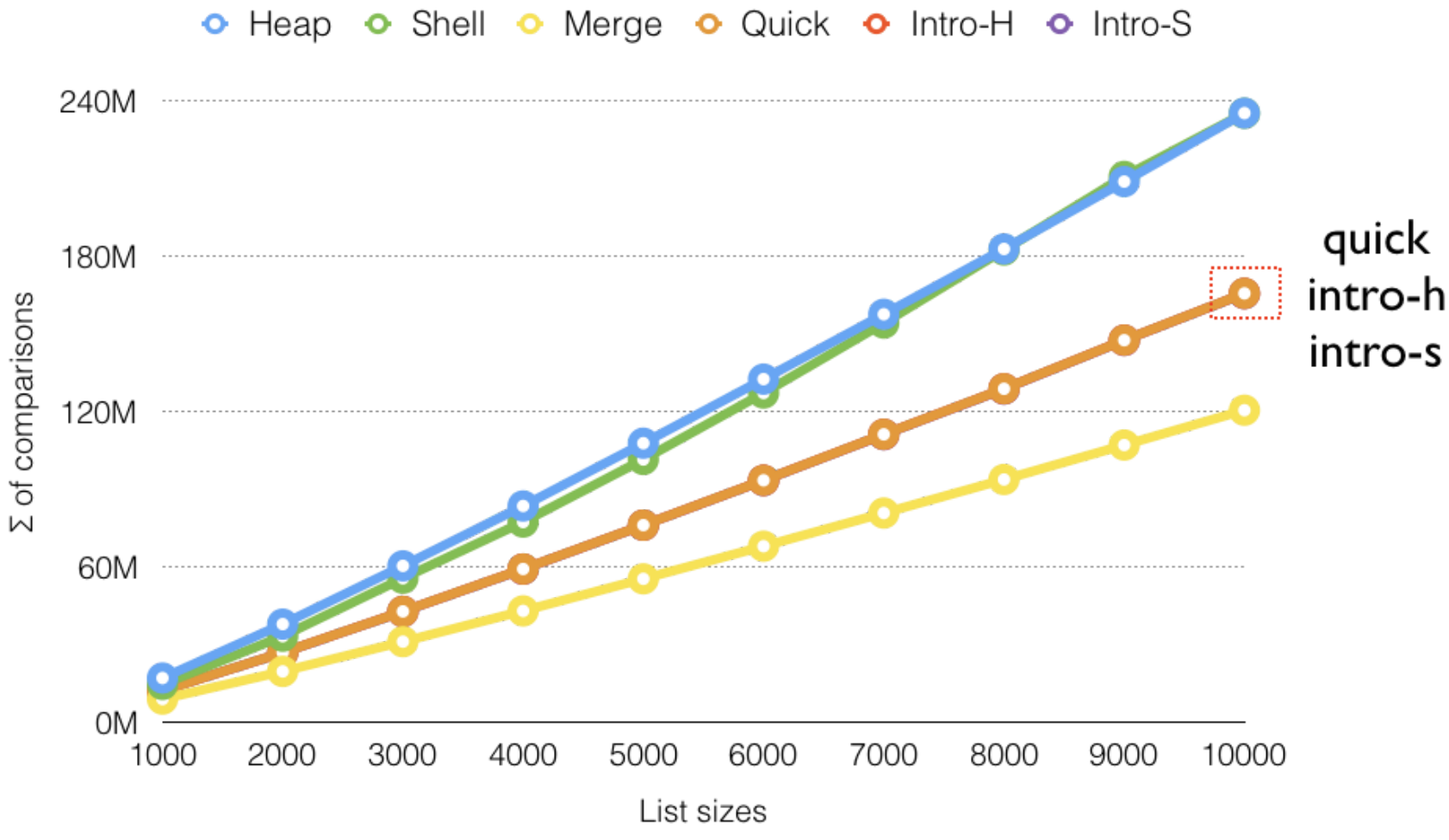
- `engine` must guarantee $O(n \log n)$.
- Benchmark: `resetCount()` recursive call?

```java
@Override
public void sort(T[] array, int beginIndex, int endIndex) {
    final int maxdepth = getMaxDepth(beginIndex, endIndex);
    introsort(array, beginIndex, endIndex, maxdepth);
    comparisons += engine.getComparisonCount();
    assignments += engine.getAssignmentCount();
}

protected int getMaxDepth(int beginIndex, int endIndex) {
    return 2 * (int) Utils.log2(endIndex - beginIndex);
}

private void introsort(T[] array, int beginIndex, int endIndex, int maxdepth) {
    if (beginIndex >= endIndex) return;

    if (maxdepth == 0)    // encounter the worst case
        engine.sort(array, beginIndex, endIndex);
    else {
        int pivotIndex = partition(array, beginIndex, endIndex);
        introsort(array, beginIndex, pivotIndex, maxdepth - 1);
        introsort(array, pivotIndex + 1, endIndex, maxdepth - 1);
    }
}
```

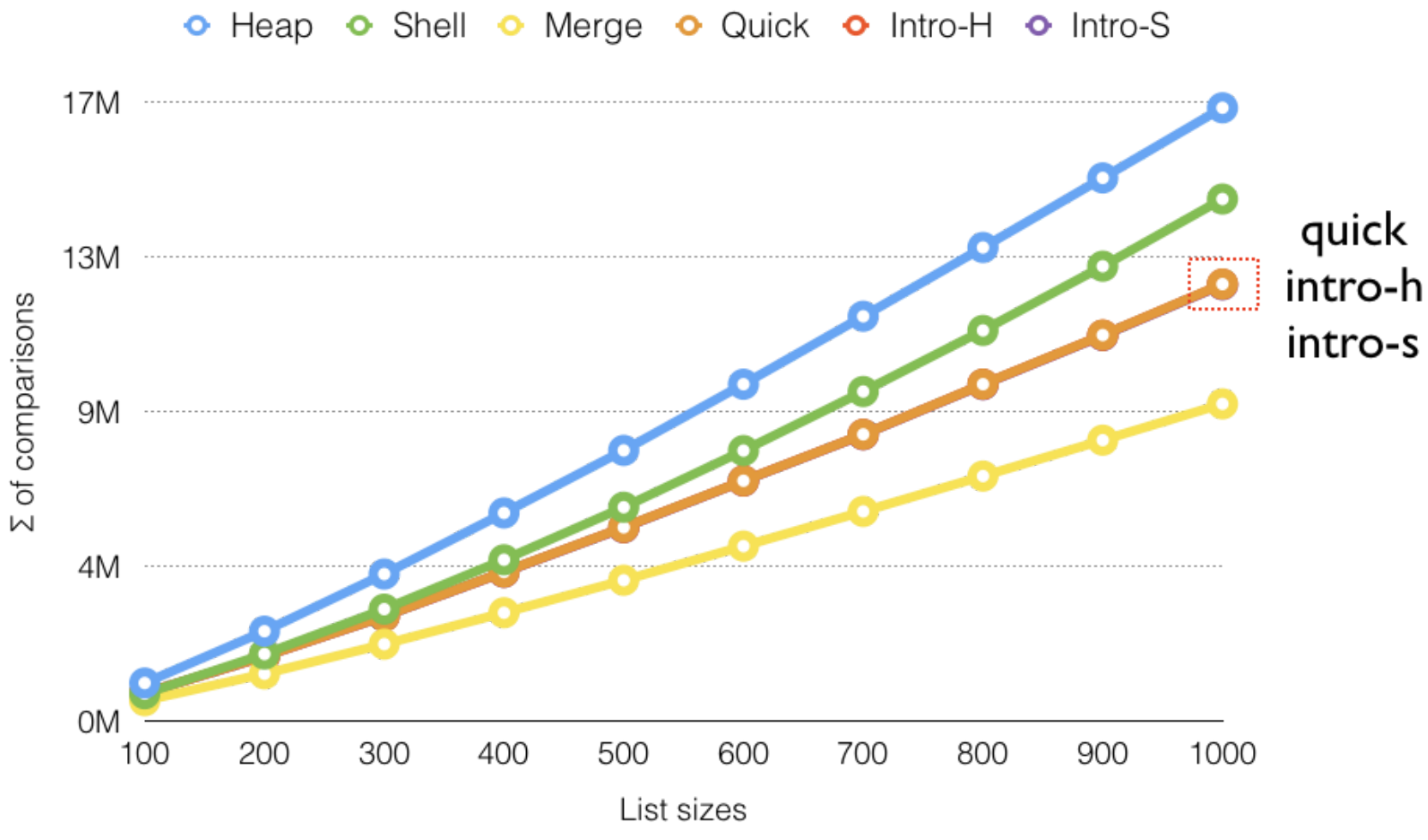
- Estimate the partition depth that can lead to the worst case.
- Switch to another algorithm when it encounters the worst case.

Benchmarks

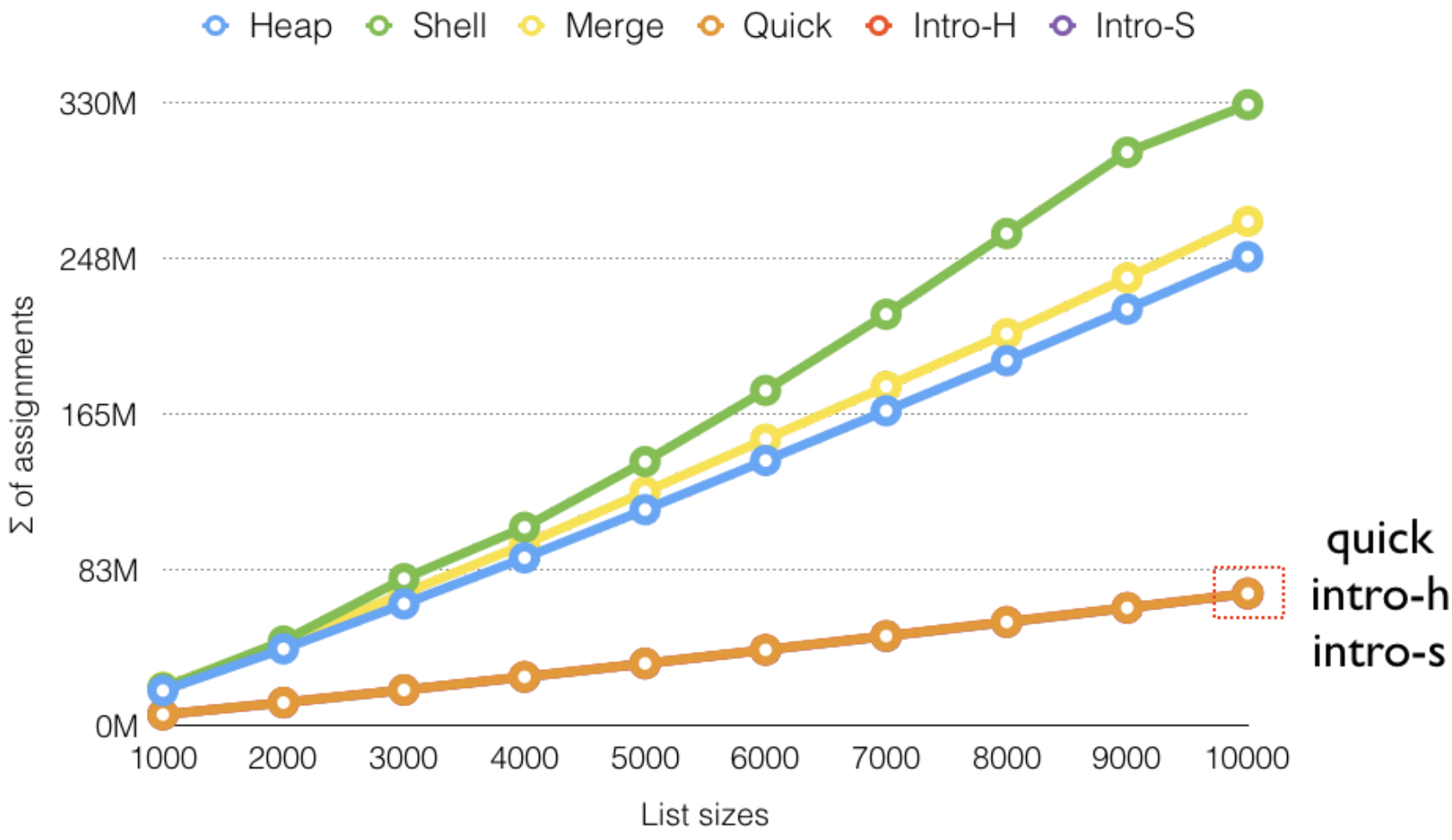
Number of Comparisons (Random)



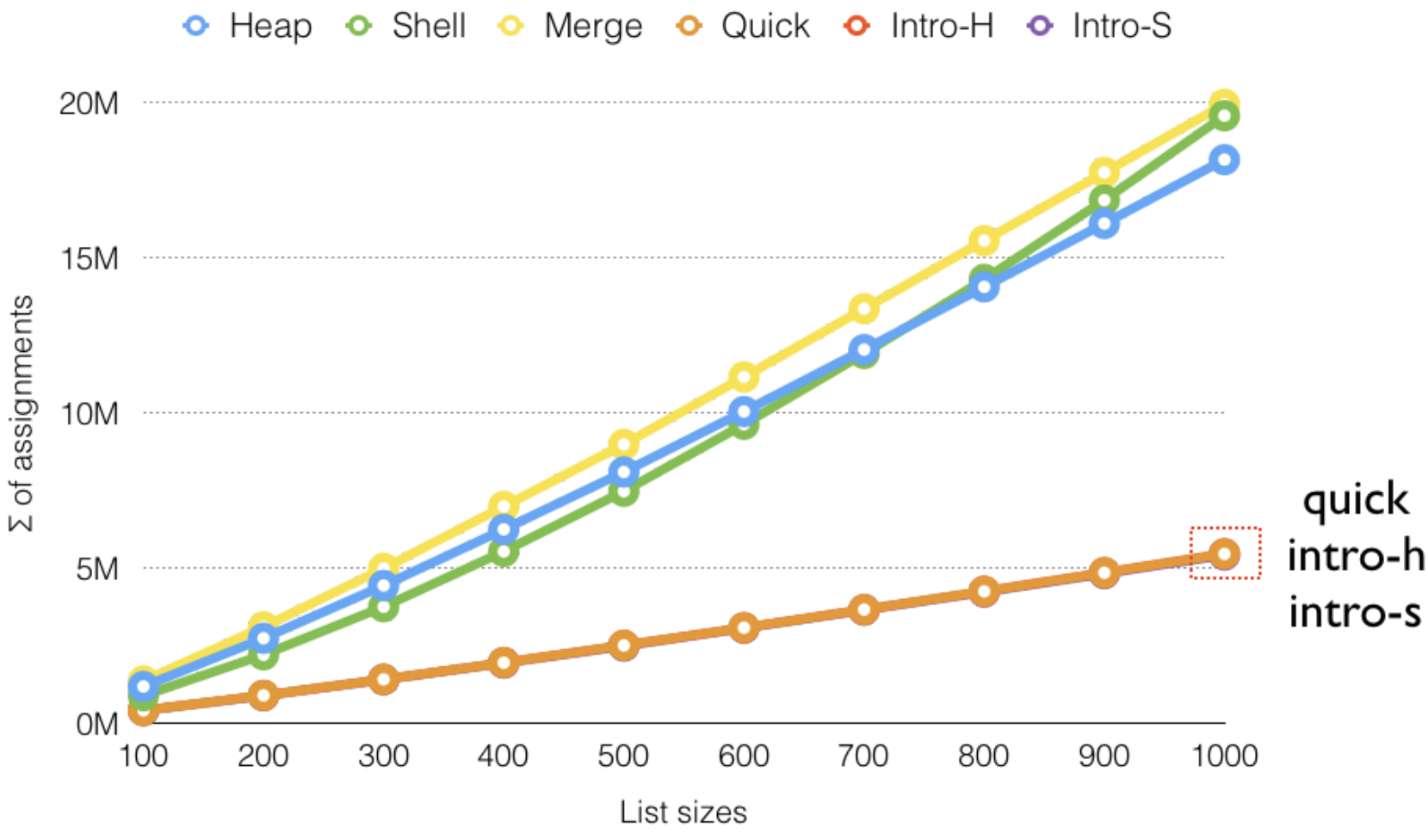
Number of Comparisons (Random)



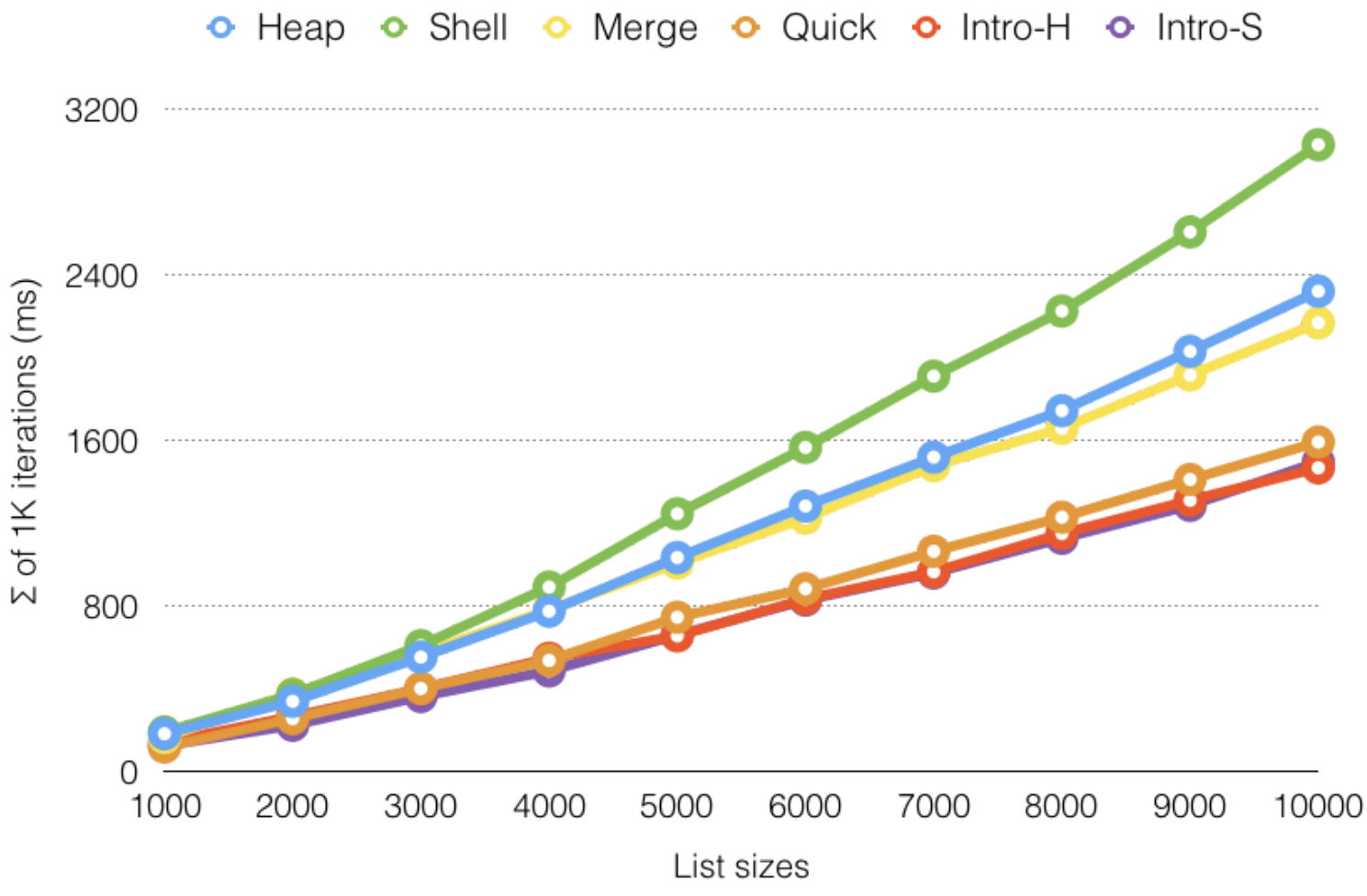
Number of Assignments (Random)



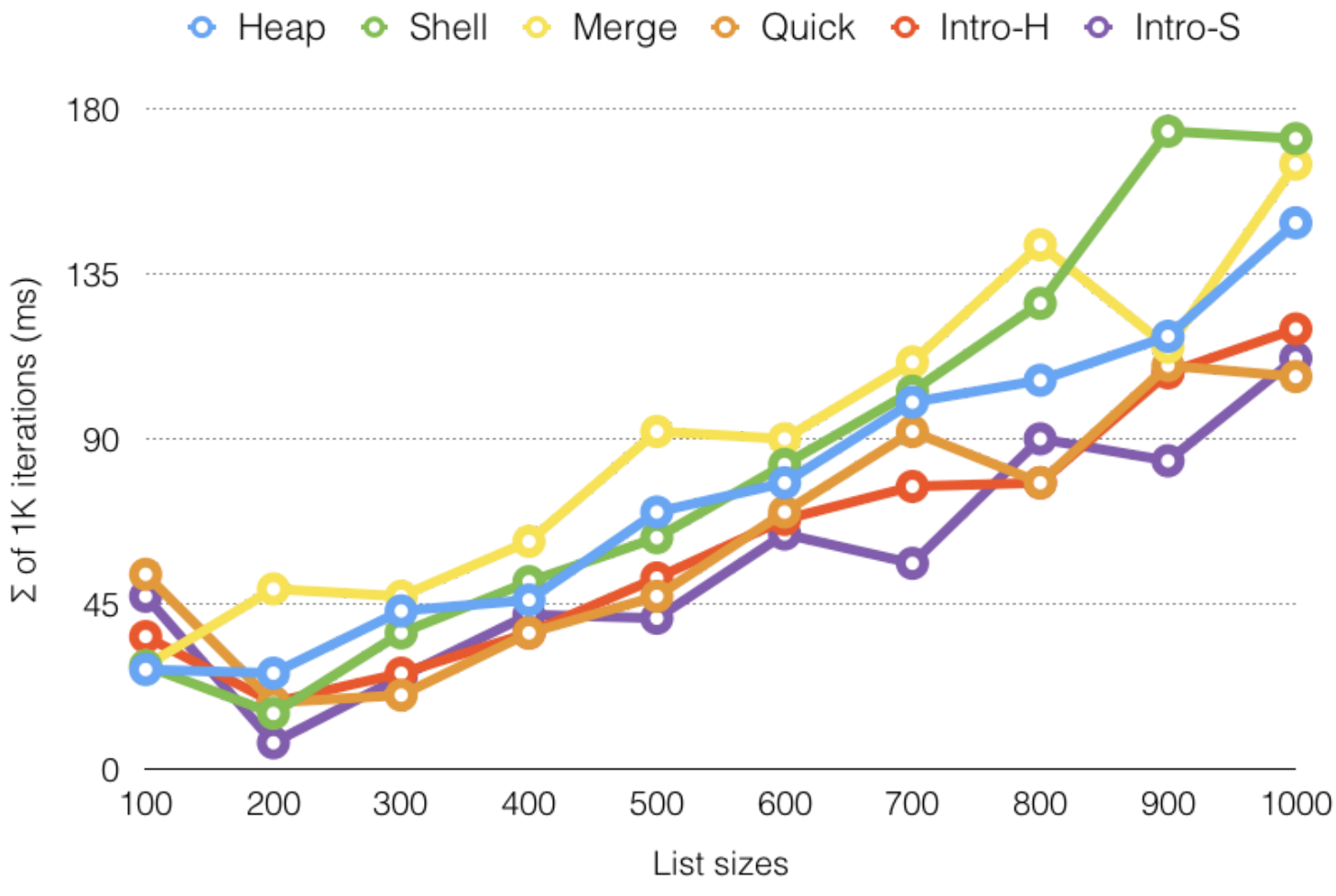
Number of Assignments (Random)



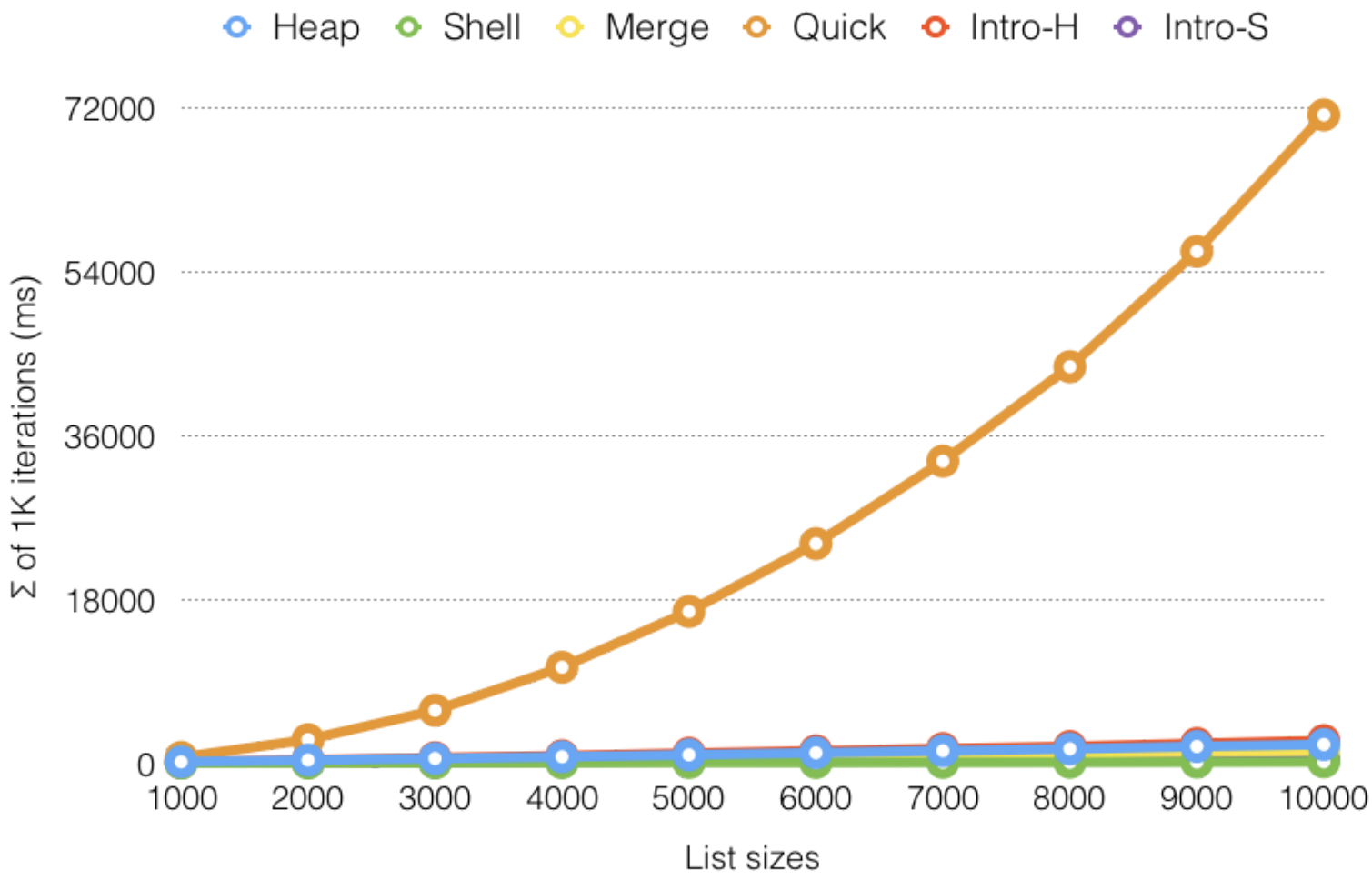
Speed Comparison (Random)



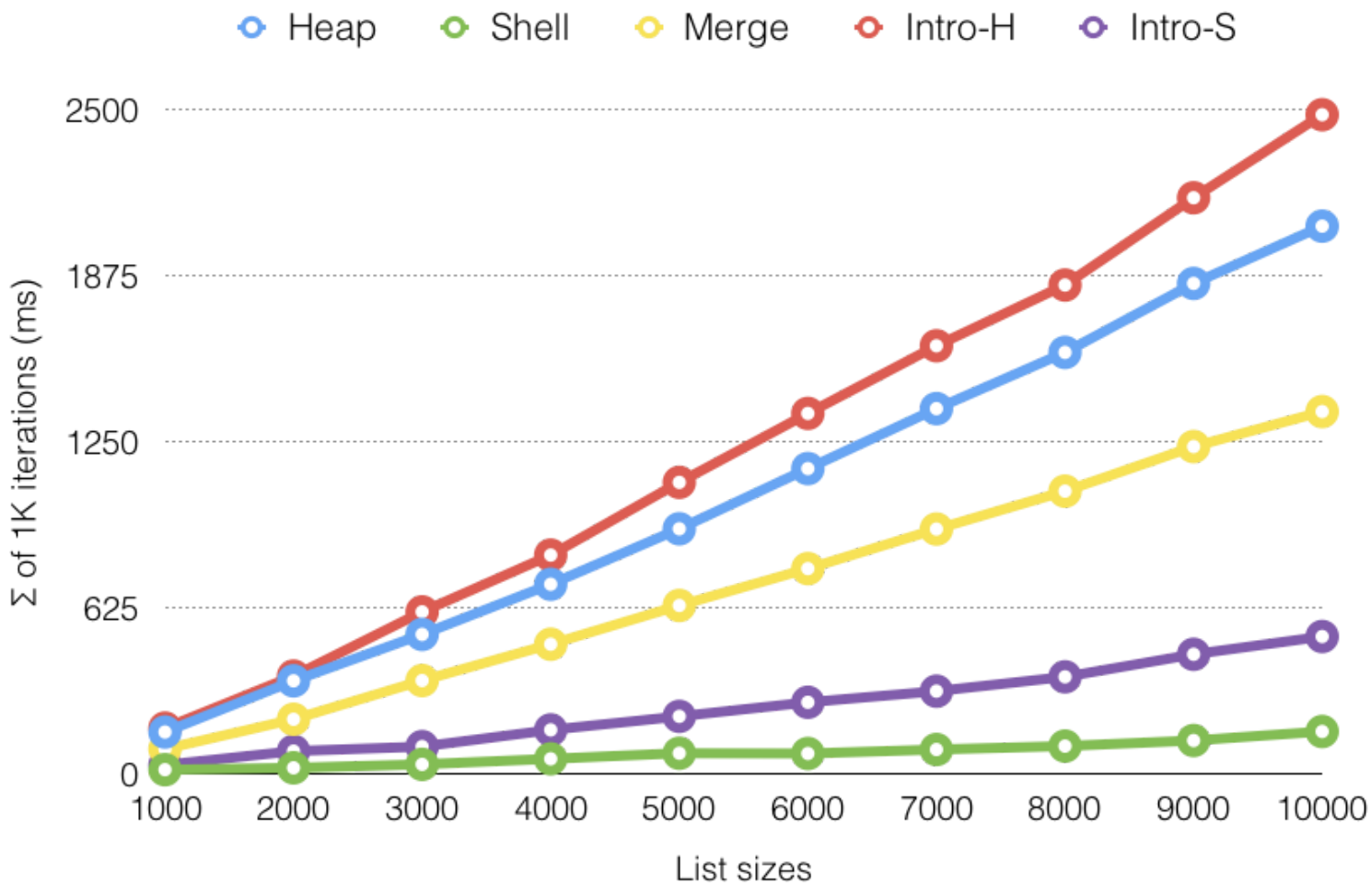
Speed Comparison (Random)



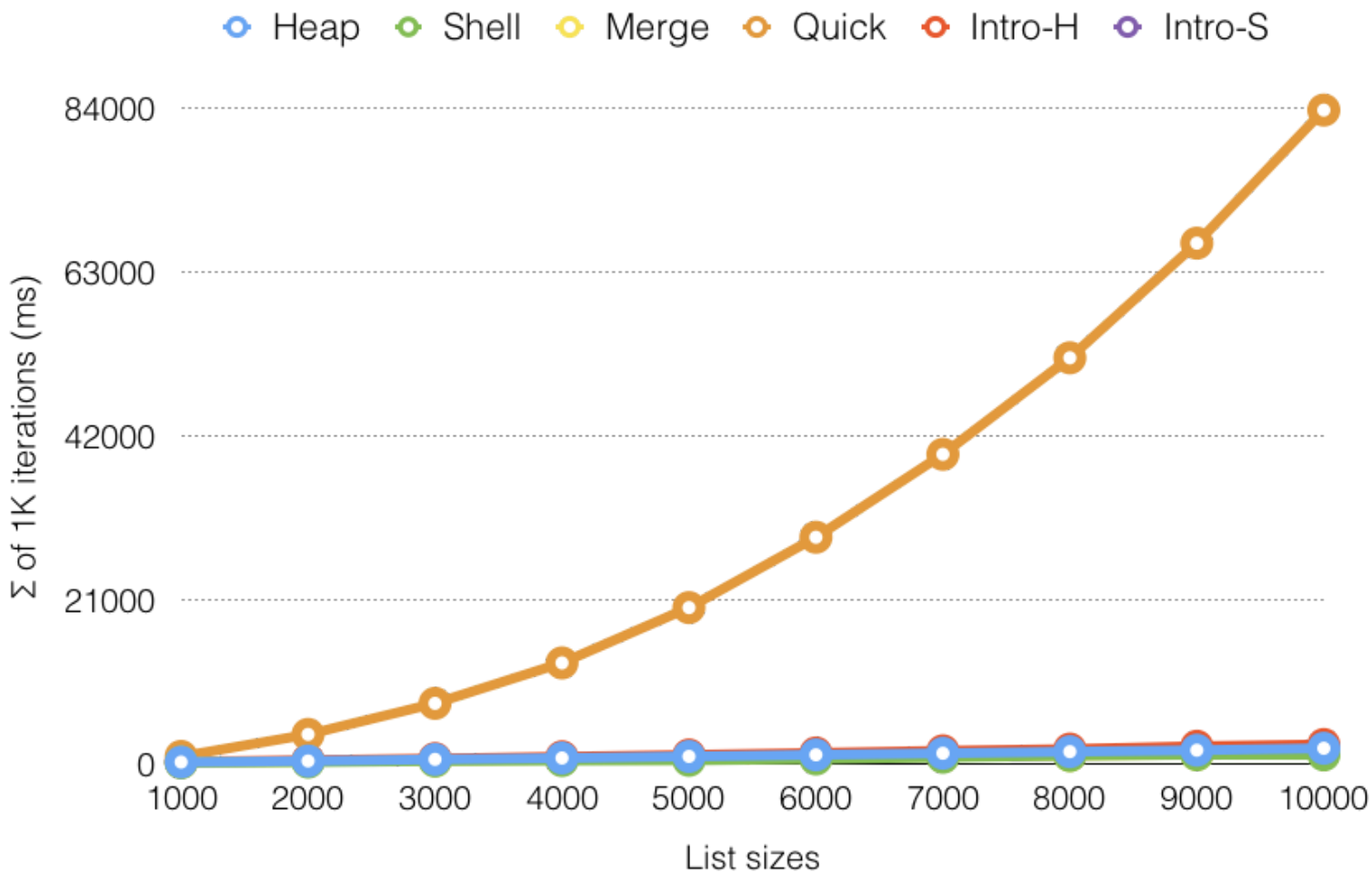
Speed Comparison (Ascending)



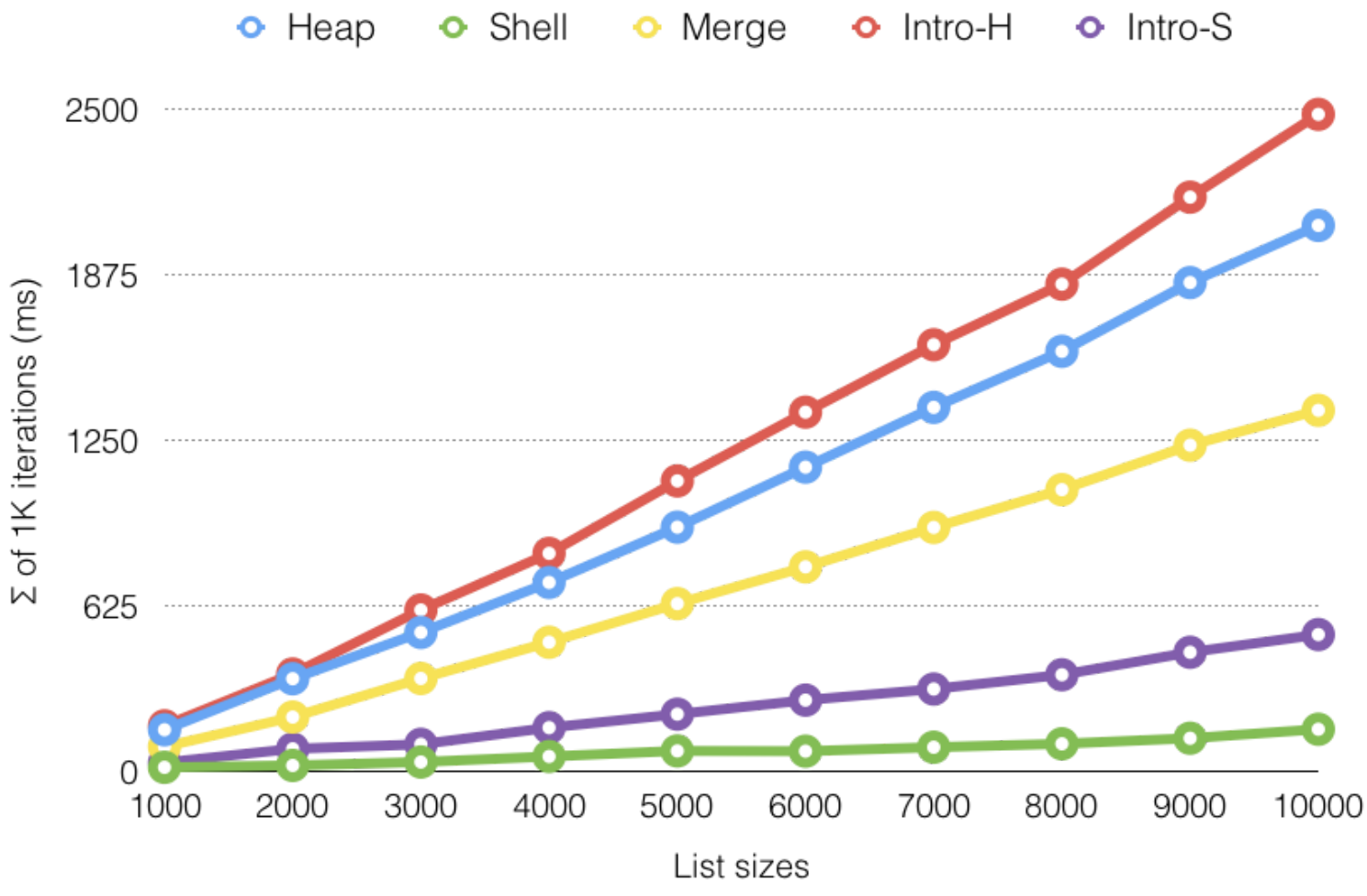
Speed Comparison (Ascending)



Speed Comparison (Descending)



Speed Comparison (Descending)



References

- Mergesort.
- Quicksort.
- Introsort.
- Timsort.