Analysing Sorting Algorithms

Lecture 19

Menu

- Analysing Fast Sorting Algorithms
  - MergeSort
  - QuickSort

Sorting Algorithm costs:

- Insertion sort, Selection Sort, Bubble Sort:
  - All slow (except Insertion sort on almost sorted lists)
  - **O(n^2)**

- Merge Sort?
- Quick Sort?
  - There's no inner loop!
  - How do you analyse recursive algorithms?

MergeSort

```
private static <E> void mergeSort(E[] data, E[] temp, int low, int high,
                                   Comparator<E> comp){
   if (high > low+1){
        int mid = (low+high)/2;
        mergeSort(temp, data, low, mid, comp);
        mergeSort(temp, data, mid, high, comp);
        merge(temp, data, low, mid, high, comp);
   }
}
```

- Cost of mergeSort:
  - Three steps:
    - first recursive call: cost?
    - second recursive call: cost?
    - merge: has to copy over (high-low) items

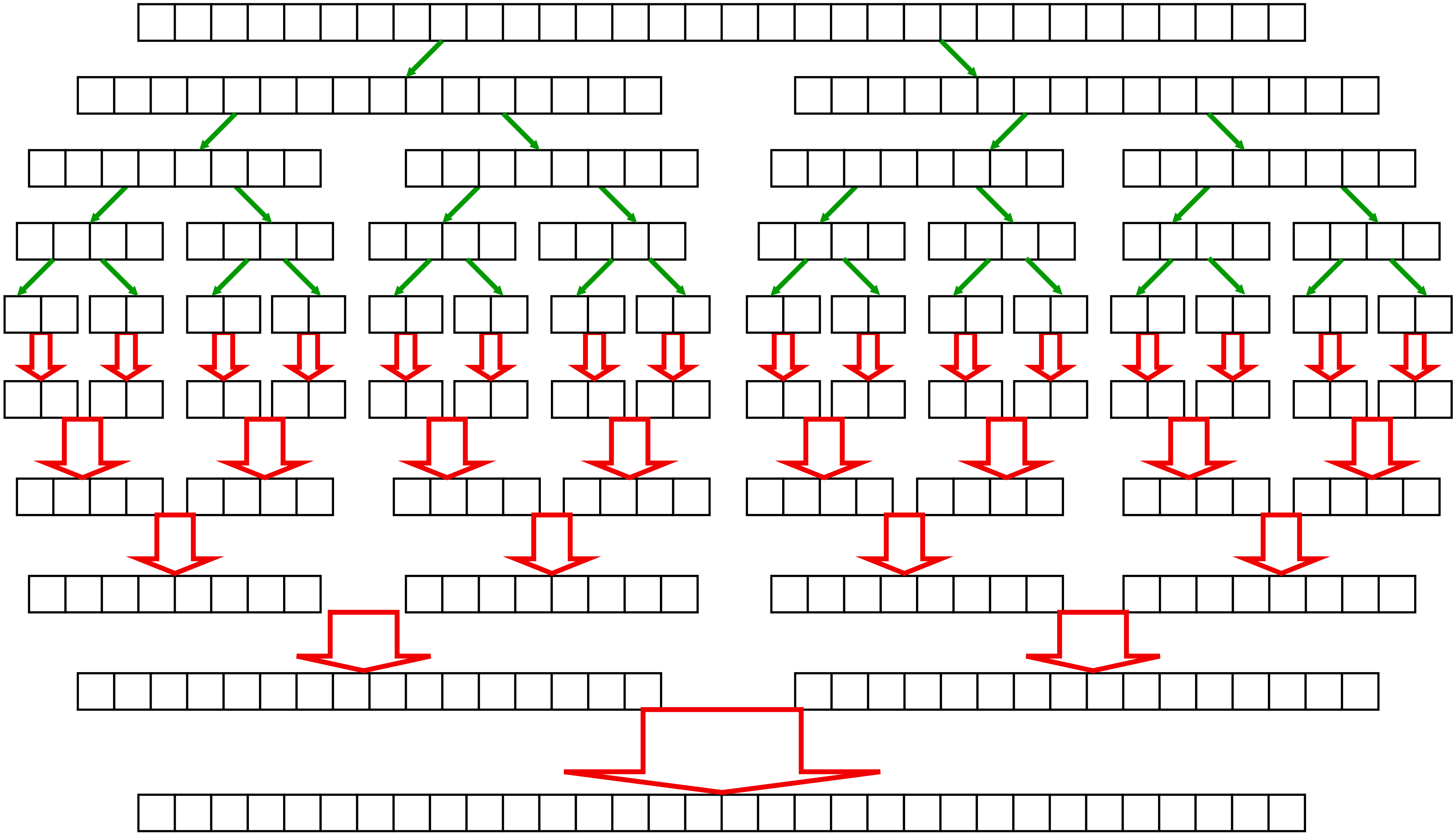
QuickSort

```
public static <E> void quickSort(E[] data, int low, int high,
                                  Comparator<E> comp){
    if (high > low +2){
        int mid = partition(data, low, high, comp);
        quickSort(data, low, mid, comp);
        quickSort(data, mid, high, comp);
    }
}
```

Cost of Quick Sort:

- three steps:
  - partition: has to compare (high-low) items
  - first recursive call
  - second recursive call

MergeSort Cost (real order)



MergeSort Cost

- Level 1: 2 * n/2 = n
- Level 2: 4 * n/4 = n
- Level 3: 8 * n/8 = n
- Level 4: 16 * n/16 = n
- Level k: n * 1 = n
- How many levels?
- Total cost? = O( )
- n = 1,000:
  n = 1,000,000
  n = 1,000,000,000

Analysing with Recurrence Relations

```
private static <E> void mergeSort(E[] data, E[] temp, int low, int high,
                                  Comparator<E> comp){
  if (high > low+1){
      int mid = (low+high)/2;
      mergeSort(temp, data, low, mid, comp);
      mergeSort(temp, data, mid, high, comp);
      merge(temp, data, low, mid, high, comp);
  }
}
```

- Cost of mergeSort = C(n)
  - $C(n) = C(n/2) + C(n/2) + n$
    $= 2\,C(n/2) + n$
- Recurrence Relation:
  - Solve by repeated substitution & find pattern
  - Solve by general method

Solving Recurrence Relations

$$
\begin{aligned}
C(n) &= 2\,C(n/2) + n \\
&= 2\,[\,\underline{2\,C(n/4) + n/2}\,] + n \\
&= 4\,C(n/4) + 2 * n \\
&= 4\,[\,\underline{2\,(C(n/8) + n/4}\,] + 2 * n \\
&= 8\,C(n/8) + 3 * n \\
&= 16\,C(n/16) + 4 * n \\
&\;\vdots \\
&= 2^k\,C(n/2^k) + k * n
\end{aligned}
$$

when $n = 2^k$, $k = \log(n)$

$$= n\,C(1) + \log(n) * n$$

since $C(1)$ = fixed cost

$$C(n) = \log(n) * n$$

QuickSort Cost

- If Quicksort divides the array exactly in half, then:
  - $C(n) = n + 2\,C(n/2)$

    $= n\log(n)$ comparisons $= O(n\log(n))$

  (best case)

- If Quicksort divides the array into 1 and n-1:
  - $C(n) = n + (n-1) + (n-2) + (n-3) + \ldots + 2 + 1$

    $= n(n-1)/2$ comparisons $= O(n^2)$

  (worst case)

- Average case?
  - Very hard to analyse.
  - Still O(n log(n)), and very good.

- Quicksort is "in place", MergeSort is not

Where have we been?

Implementing Collections:

- ArrayList: O(n) to add/remove, except at end
- Stack: O(1)
- ArraySet: O(n) (cost of searching)
- SortedArraySet O(log(n)) to search (with binary search)
  O(n) to add/remove (cost of inserting)
  $O(n^2)$ to add n items
  O(n log(n)) to initialise with n items.
  (with fast sorting)

Summary

- Analysing Fast Sorting Algorithms
  - MergeSort
  - QuickSort

Readings

- [Mar07] Read 2.3
- [Mar13] Read 2.3