Quicksort Revisited

Corky Cartwright
Vivek Sarkar
Department of Computer Science
Rice University

Acknowledgments

- David Matuszek, UPenn CIT 594 (Programming Languages & Techniques II), Lecture 34, Spring 2003
  - www.cis.upenn.edu/~matuszek/cit594-2003/.../34-quicksort.ppt

The Sorting Problem

- Given a array of n objects (records) R, construct an array R' containing the same set of records as R but in ascending (non-descending) order according to a specified comparison function
- Example of comparison function: compareTo() method for Java objects that implement the Comparable interface

Quick Sort (Lecture 11)

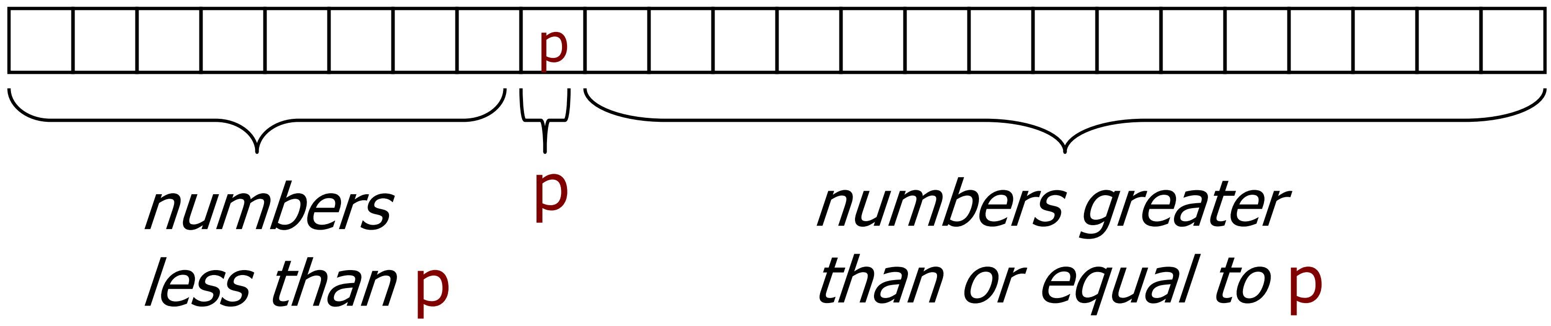
- Invented by C.A.R. ("Tony") Hoare
- Functional version is derived from the imperative (destructive) algorithm; less efficient but still works very well
- Idea:
  - Base case: list of length 0 or 1
  - Inductive case:
    - partition the list into the singleton list containing first, the list of all items <= first, and the list of all items > first
    - sort the the lists of lesser and greater items
    - return (sorted lesser) || (first) || (sorted greater) where || means list concatenation (append)

Quicksort in Scheme (Lecture 11)

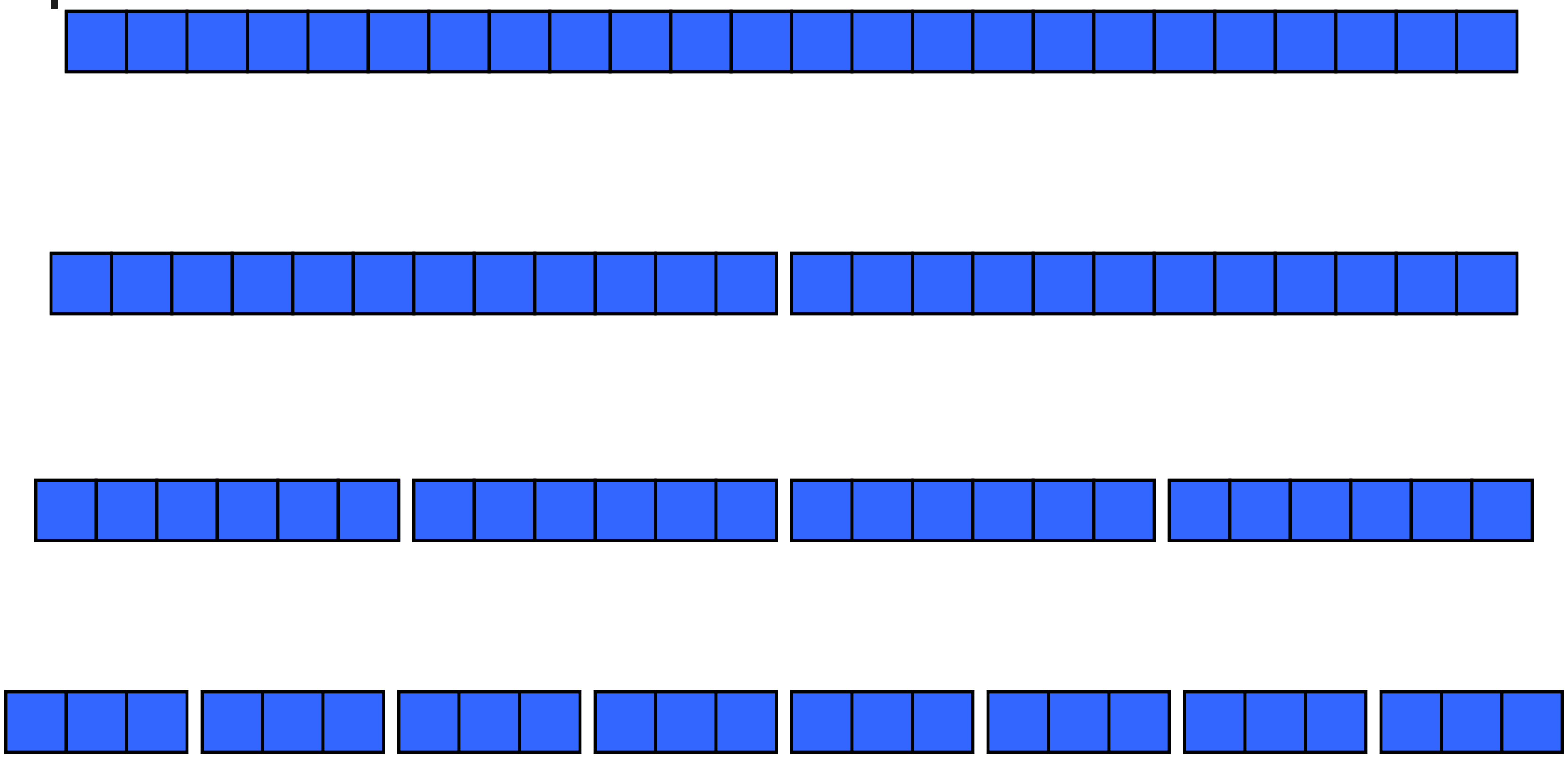
```
(define (qsort alon)
  (cond
    [(empty? alon) empty]
    [else
      (local ((define pivot (first alon))
              (define other (rest alon)))
        (append
          (qsort [filter (lambda (x) (<= x pivot)) other])
          (list pivot)
          (qsort [filter (lambda (x) (> x pivot)) other])))]))
```

Partitioning

- A key step in the Quicksort algorithm is partitioning the array
  - We choose some (any) number p in the array to use as a pivot
  - We partition the array into three parts:



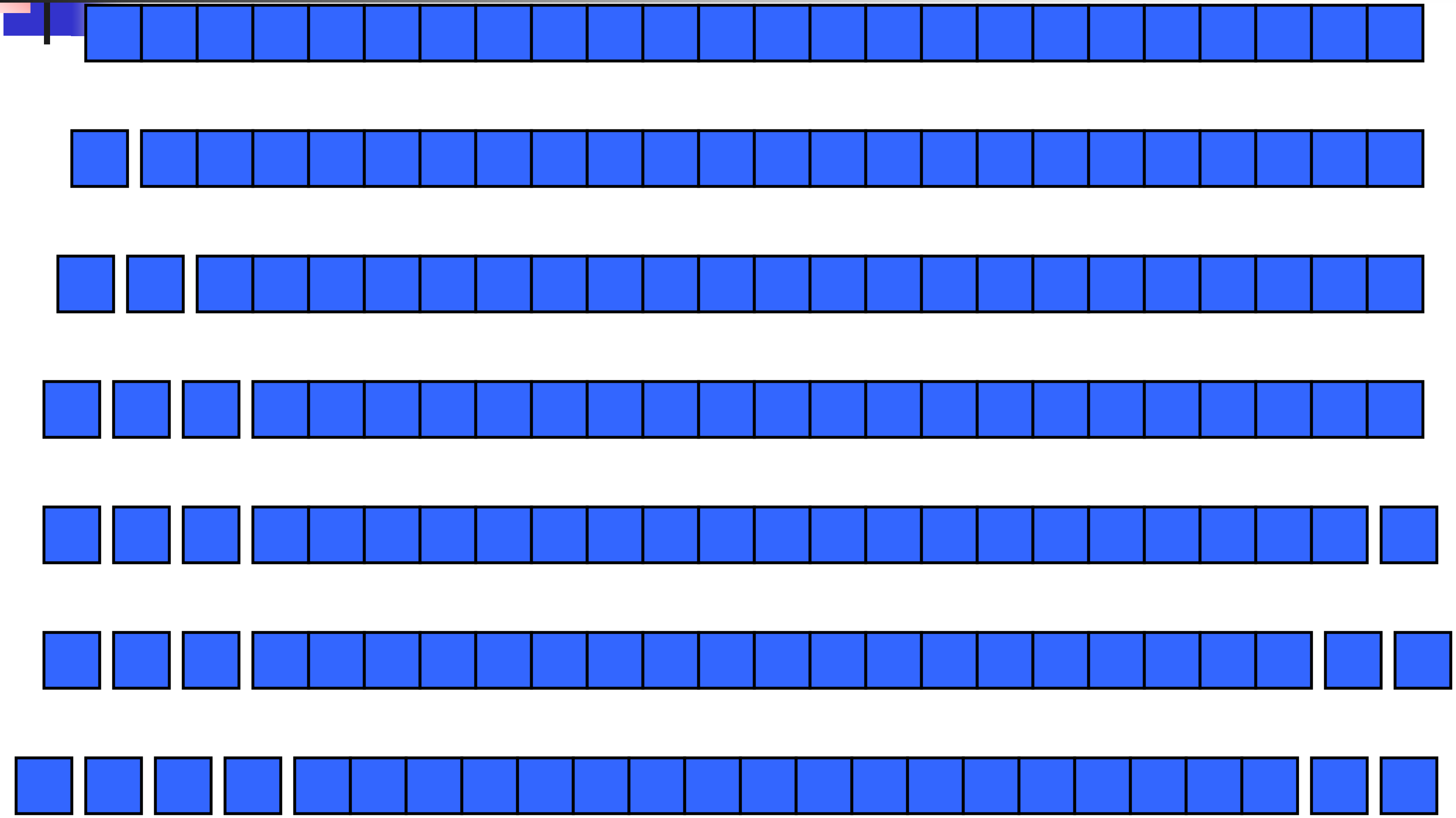
Partitioning at various levels (Best case)



Best case Partitioning

- We cut the array size in half each time
- So the depth of the recursion in $\log_2 n$
- At each level of the recursion, all the partitions at that level do work that is linear in n
- $O(\log_2 n) * O(n) = O(n \log_2 n)$
- Hence in the best case, quicksort has time complexity $O(n \log_2 n)$
- What about the worst case?

Worst case partitioning



Worst case for quicksort

- In the worst case, recursion may be n levels deep (for an array of size n)
- But the partitioning work done at each level is still n
- $O(n) * O(n) = O(n^2)$
- So worst case for Quicksort is $O(n^2)$
- When can this happen?
  - e.g., when the array is sorted to begin with!

Typical case for quicksort

- If the array is sorted to begin with, Quicksort is terrible: $O(n^2)$
- It is possible to construct other bad cases
- However, Quicksort is *usually* $O(n \log_2 n)$
- The constants are so good that Quicksort is generally the fastest algorithm known
- A lot of real-world sorting is done by Quicksort

Picking a better pivot

- Before, we picked the *first* element of the subarray to use as a pivot
  - If the array is already sorted, this results in $O(n^2)$ behavior
  - It's no better if we pick the *last* element
- We could do an *optimal* quicksort (guaranteed O(n log n)) if we always picked a pivot value that exactly cuts the array in half
  - Such a value is called a median: half of the values in the array are larger, half are smaller
  - The easiest way to find the median is to *sort* the array and pick the value in the middle (!)

Median of three

- Obviously, it doesn't make sense to sort the array in order to find the median to use as a pivot
  - There are faster more advanced algorithms to find the median that we'll ignore for today
- Instead, compare just *three* elements of our (sub)array—the first, the last, and the middle
  - Take the *median* (middle value) of these three as pivot
  - It's possible (but less likely) to construct cases which will make this technique $O(n^2)$

- For simplicity, we will continue with first element as pivot in the rest of this lecture.

Functional version of Quicksort in Java

```
public static ArrayList<Integer> quickSort(ArrayList<Integer> a) {
    if (a.isEmpty()) return new ArrayList<Integer>();
    ArrayList<Integer> left = new ArrayList<Integer>();
    ArrayList<Integer> mid = new ArrayList<Integer>();
    ArrayList<Integer> right = new ArrayList<Integer>();
    for (Integer i : a)
        if ( i < a.get(0) ) left.add(i); // Use element 0 as pivot
        else if ( i > a.get(0)) right.add(i);
        else mid.add(i)
    ArrayList<Integer> left_s = quickSort(left);
    ArrayList<Integer> right_s = quickSort(right);
    return left_s.addAll(mid).addAll(right_s);
}
```

Reprise: Task Decomposition

```
public static ArrayList<Integer> quickSort(ArrayList<Integer> a) {
    if (a.isEmpty()) return new ArrayList<Integer>();
    ArrayList<Integer> left = new ArrayList<Integer>();
    ArrayList<Integer> mid = new ArrayList<Integer>();
    ArrayList<Integer> right = new ArrayList<Integer>();
    for (Integer i : a)
        if ( i < a.get(0) ) left.add(i); // Use element 0 as pivot
        else if ( i > a.get(0)) right.add(i);
        else mid.add(i)
    final ArrayList<Integer> left_f = left, right_f = right; // QUESTION: why do we need these?
    Callable<ArrayList<Integer>> left_c = new Callable<ArrayList<Integer>>() {
        public ArrayList<Integer> call() { return quickSort(left_f); } } ;
    Callable<ArrayList<Integer>> right_c = new Callable<ArrayList<Integer>>() {
        public ArrayList<Integer> call() { return quickSort(right_f); } } ;
   // QUESTION: where can we place left_c.call() and right_c.call()?
  . . .
}
```

Original Task Order

```
public static ArrayList<Integer> quickSort(ArrayList<Integer> a) {
    if (a.isEmpty()) return new ArrayList<Integer>();
    ArrayList<Integer> left = new ArrayList<Integer>();
    ArrayList<Integer> mid = new ArrayList<Integer>();
    ArrayList<Integer> right = new ArrayList<Integer>();
    for (Integer i : a)
        if ( i < a.get(0) ) left.add(i); // Use element 0 as pivot
        else if ( i > a.get(0)) right.add(i);
        else mid.add(i)
    final ArrayList<Integer> left_f = left, right_f = right;
    Callable<ArrayList<Integer>> left_c = new Callable<ArrayList<Integer>>() {
        public ArrayList<Integer> call() { return quickSort(left_f); } } ;
    Callable<ArrayList<Integer>> right_c = new Callable<ArrayList<Integer>>() {
        public ArrayList<Integer> call() { return quickSort(right_f); } } ;
    ArrayList<Integer> left_s = left_s.call(); ArrayList<Integer> right_s = right_c.call();
    return left_s.addAll(mid).addAll(right_s);
```

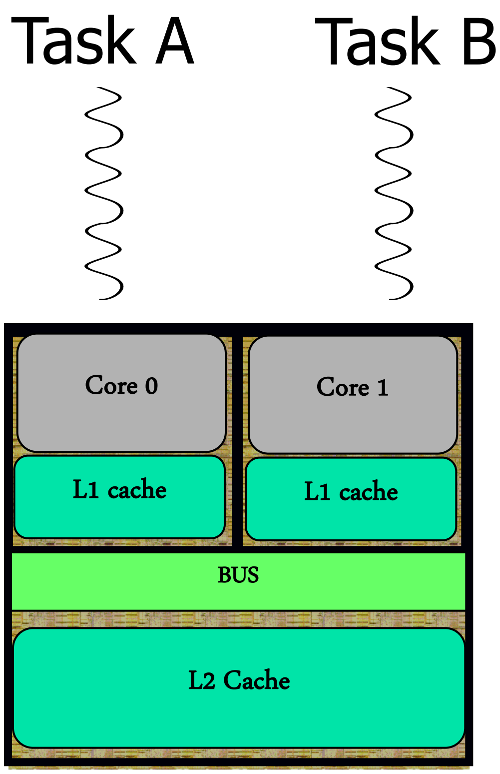
Alternate Task Order

```
public static ArrayList<Integer> quickSort(ArrayList<Integer> a) {
    if (a.isEmpty()) return new ArrayList<Integer>();
    ArrayList<Integer> left = new ArrayList<Integer>();
    ArrayList<Integer> mid = new ArrayList<Integer>();
    ArrayList<Integer> right = new ArrayList<Integer>();
    for (Integer i : a)
       if ( i < a.get(0) ) left.add(i); // Use element 0 as pivot
       else if ( i > a.get(0)) right.add(i);
       else mid.add(i)
    final ArrayList<Integer> left_f = left, right_f = right;
    Callable<ArrayList<Integer>> left_c = new Callable<ArrayList<Integer>>() {
        public ArrayList<Integer> call() { return quickSort(left_f); } } ;
    Callable<ArrayList<Integer>> right_c = new Callable<ArrayList<Integer>>() {
        public ArrayList<Integer> call() { return quickSort(right_f); } } ;
    ArrayList<Integer> right_s = right_c.call(); ArrayList<Integer> left_s = left_s.call();
    return left_s.addAll(mid).addAll(right_s);
}
```

From Sequential to Parallel Task Decomposition

<u>Key Observation:</u>

If two *functional* tasks can be executed in any order, they can also be executed *in parallel*



Schematic of a Dual-core Processor

How can we express Task Parallelism in Java?

- Answer: there are many ways, but they all ultimately involve execution on Java *threads*
- The Java main program starts as a single thread
- The code executed by the main thread can create other threads
  - Either explicitly (as in the following slides); or
  - Implicitly via library use:
    - AWT/Swing, Applets, RMI, image loading, Servlets, web services, `Executor` usage (thread pools), …

Executing a Callable task in a parallel Java Thread

```
// 1. Create a callable closure (lambda)
Callable<ArrayList<Integer>> left_c = …
// 2. Package the closure as a task
final FutureTask<ArrayList<Integer>> task_A =
    new FutureTask<ArrayList<Integer>>(left_c);
// 3. Start executing the task in a parallel thread
new Thread(task_A).start();
// 4. Wait for task to complete, and get its result
left_s = task_A.get();
```

Quicksort with Parallel Tasks

```
public static ArrayList<Integer> quickSort(ArrayList<Integer> a) {
     if (a.isEmpty()) return new ArrayList<Integer>();
    ArrayList<Integer> left = new ArrayList<Integer>();
    ArrayList<Integer> mid = new ArrayList<Integer>();
    ArrayList<Integer> right = new ArrayList<Integer>();
    for (Integer i : a)
       if ( i < a.get(0) ) left.add(i); // Use element 0 as pivot
       else if ( i > a.get(0)) right.add(i);
       else mid.add(i)
    final ArrayList<Integer> left_f = left, right_f = right;
    FutureTask<ArrayList<Integer>> left_t = new FutureTask<ArrayList<Integer>>(
       new Callable<ArrayList<Integer>>() {
          public ArrayList<Integer> call() { return quickSort(left_f); } } );
    FutureTask<ArrayList<Integer>> right_t = new FutureTask<ArrayList<Integer>>(
       new Callable<ArrayList<Integer>>() {
           public ArrayList<Integer> call() { return quickSort(right_f); } } );
   new Thread(left_t).start(); new Thread(right_t).start();
   ArrayList<Integer> left_s = left_t.get(); ArrayList<Integer> right_s = right_t.call();
    return left_s.addAll(mid).addAll(right_s);
}
```

Discussion

- Why must the tasks be functional? What would happen if two parallel tasks attempted to mutate the same object?
- It is strongly recommended that each FutureTask declaration be final. Why? Can you create a cyclic wait structure with blocking get() operations?
- Sometimes, a parallel program may run slower than a sequential program. Why? Note that it can take a large number (> 10^4) of machine instructions just to create a thread.