Divide-Conquer-Glue

Tyler Moore

CSE 3353, SMU, Dallas, TX

February 19, 2013

Portions of these slides have been adapted from the slides written by Prof. Steven Skiena at SUNY Stony Brook, author of Algorithm Design Manual. For more information see http://www.cs.sunysb.edu/~skiena/

Notes

Divide-Conquer-Glue Algorithm Strategy

- The main programming paradigm you've learned *iterates* through problems: given a problem of size n, split it up into subproblems of size 1 and $n-1$
- How did you do this in Q1 of HW1 (say for exhaustive search job selection)?
- Rather than bite off one very small piece at a time for processing, with divide-and-conquer, you repeatedly divide the problem in half until it is manageable
- You've already encountered this paradigm in Mergesort
- By dividing tasks evenly, we can often solve tasks in logarithmic time, rather than linear (or log-linear instead of quadratic)

Notes

Skyline Problem as an Example of Divide-Conquer-Glue

- We can incrementally add buildings to a skyline in linear time
- Thus, to build a complete skyline, we can do so in quadratic time
- But is there a better way? Can't we also merge two existing skylines into a combined skyline for the same cost of adding one building to a skyline?

Notes

Canonical Divide-Conquer-Glue Algorithm

```
def divide_and_conquer(S, divide, glue):
    if len(S) == 1: return S
    L, R = divide(S)
    A = divide_and_conquer(L, divide, glue)
    B = divide_and_conquer(R, divide, glue)
    return glue(A, B)
```


Notes

Before we get to Mergesort

- Let's talk through a simpler algorithm that employs divide-conquer-glue
- *Selection Problem:* find the kth smallest number of an unsorted sequence.
- What is the name for selection when $k = \frac{n}{2}$?
- $\Theta(n \lg n)$ solution is easy. How?

- There is a linear-time solution available in the average case

Notes

Partitioning with pivots

- To use a divide-conquer-glue strategy, we need a way to split the problem in half
- Furthermore, to make the running time linear, we need to always identify the half of the problem where the kth element is
- Key insight: split the sequence by a random *pivot*. If the subset of smaller items happens to be of size $k - 1$, then you have found the pivot. Otherwise, pivot on the half known to have k.

Notes

Partition and Select

```
1  def partition(seq):
2      pi, seq = seq[0], seq[1:]         # Pick and remove the pivot
3      lo = [x for x in seq if x <= pi] # All the small elements
4      hi = [x for x in seq if x > pi]  # All the large ones
5      return lo, pi, hi                 # pi is "in the right place"
6
7  def select(seq, k):
8      lo, pi, hi = partition(seq)       # [<= pi], pi, [> pi]
9      m = len(lo)
10     if m == k: return pi              # Found kth smallest
11     elif m < k:                       # Too far to the left
12         return select(hi, k-m-1)      # Remember to adjust k
13     else:                             # Too far to the right
14         return select(lo, k)          # Use original k here
```


Notes

A verbose Select function

```
def select(seq, k):
    lo, pi, hi = partition(seq)       # [<= pi], pi, [> pi]
    print lo, pi, hi
    m = len(lo)
    print 'small partition length %i' %(m)
    if m == k:
        print 'found kth smallest %s' % pi
        return pi              # Found kth smallest
    elif m < k:                # Too far to the left
        print 'small partition has %i elements, so kth must be in right sequence' % m
        return select(hi, k-m-1)      # Remember to adjust k
    else:                      # Too far to the right
        print 'small partition has %i elements, so kth must be in left sequence' % m
        return select(lo, k)          # Use original k here
```


Notes

Seeing the Select in action

```
>>> select([3, 4, 1, 6, 3, 7, 9, 13, 93, 0, 100, 1, 2, 2, 3, 3, 2],4)
[1, 3, 0, 1, 2, 2, 3, 3, 2] 3 [4, 6, 7, 9, 13, 93, 100]
small partition length 9
small partition has 9 elements, so kth must be in left sequence
[0, 1] 1 [3, 2, 2, 3, 3, 2]
small partition length 2
small partition has 2 elements, so kth must be in right sequence
[2, 2, 3, 3, 2] 3 []
small partition length 5
small partition has 5 elements, so kth must be in left sequence
[2, 2] 2 [3, 3]
small partition length 2
small partition has 2 elements, so kth must be in left sequence
[2] 2 []
small partition length 1
found kth smallest 2
2
```


Notes

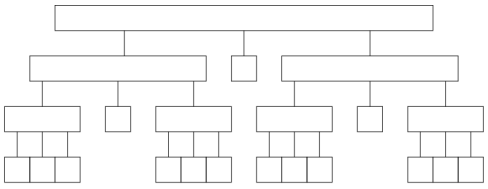
From Quickselect to Quicksort

Question: what if we wanted to know all k-smallest items (for $k = 1 \rightarrow n$)?

```
1 def quicksort(seq):
2     if len(seq) <= 1: return seq                  # Base case
3     lo, pi, hi = partition(seq)                   # pi is in its place
4     return quicksort(lo) + [pi] + quicksort(hi) # Sort lo and hi separately
```


Notes

Best case for Quicksort

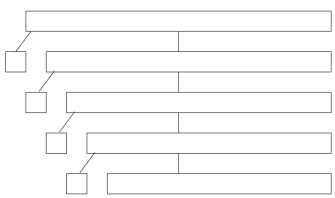


The total partitioning on each level is $O(n)$, and it take $\lg n$ levels of perfect partitions to get to single element subproblems. When we are down to single elements, the problems are sorted. Thus the total time in the best case is $O(n \lg n)$.

Notes

Worst case for Quicksort

Suppose instead our pivot element splits the array as unequally as possible. Thus instead of $n/2$ elements in the smaller half, we get zero, meaning that the pivot element is the biggest or smallest element in the array.



Now we have $n1$ levels, instead of $\lg n$, for a worst case time of $\Theta(n^2)$, since the first $n/2$ levels each have $\geq n/2$ elements to partition.

Notes

Picking Better Pivots

- Having the worst case occur when they are sorted or almost sorted is very bad, since that is likely to be the case in certain applications. To eliminate this problem, pick a better pivot:
  1. Use the middle element of the subarray as pivot.
  2. Use a random element of the array as the pivot.
  3. Perhaps best of all, take the median of three elements (first, last, middle) as the pivot. Why should we use median instead of the mean?
- Whichever of these three rules we use, the worst case remains $O(n^2)$.

Notes

Randomized Quicksort

- Suppose you are writing a sorting program, to run on data given to you by your worst enemy. Quicksort is good on average, but bad on certain worst-case instances.
- If you used Quicksort, what kind of data would your enemy give you to run it on? Exactly the worst-case instance, to make you look bad.
- But instead of picking the median of three or the first element as pivot, suppose you picked the pivot element at random.
- Now your enemy cannot design a worst-case instance to give to you, because no matter which data they give you, you would have the same probability of picking a good pivot!

Notes

Randomized Guarantees

- Randomization is a very important and useful idea. By either picking a random pivot or scrambling the permutation before sorting it, we can say: "With high probability, randomized quicksort runs in $\Theta(n \lg n)$ time"
- Where before, all we could say is: "If you give me random input data, quicksort runs in expected $\Theta(n \lg n)$ time."
- See the difference?

Notes

Importance of Randomization

- Since the time bound how does not depend upon your input distribution, this means that unless we are extremely unlucky (as opposed to ill prepared or unpopular) we will certainly get good performance.
- Randomization is a general tool to improve algorithms with bad worst-case but good average-case complexity.
- The worst-case is still there, but we almost certainly wont see it.

Notes

Canonical Divide-Conquer-Glue Algorithm

```
def divide_and_conquer(S, divide, glue):
    if len(S) == 1: return S
    L, R = divide(S)
    A = divide_and_conquer(L, divide, glue)
    B = divide_and_conquer(R, divide, glue)
    return glue(A, B)
```

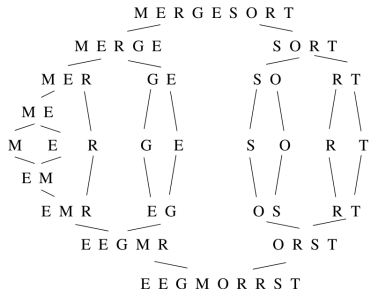

Notes

Mergesort

```
1  def mergesort(seq):
2      mid = len(seq)/2                    #Midpoint for division
3      lft, rgt = seq[:mid], seq[mid:]
4      if len(lft) > 1: lft = mergesort(lft)#Sort by halves
5      if len(rgt) > 1: rgt = mergesort(rgt)
6      res = []                            #Merge sorted halves
7      while lft and rgt:                  #Neither half is empty
8          if lft[-1] >= rgt[-1]:          #lft has greatest last value
9              res.append(lft.pop())       #Append it
10         else:                           #rgt has greatest last value
11             res.append(rgt.pop())       #Append it
12     res.reverse()                       #Result is backward
13     return (lft or rgt) + res           #Also add the remainder
```


Notes

Mergesort



Notes

Merging Sorted Lists

- The efficiency of mergesort depends upon how efficiently we combine the two sorted halves into a single sorted list.
- This smallest element can be removed, leaving two sorted lists behind, one slightly shorter than before.
- Repeating this operation until both lists are empty merges two sorted lists (with a total of n elements between them) into one, using at most $n1$ comparisons or $O(n)$ total work
- Example: $A = 5, 7, 12, 19$ and $B = 4, 6, 13, 15$.

Notes

External Sorting

- Which $O(n \log n)$ algorithm you use for sorting doesnt matter much until n is so big the data does not fit in memory.
- Mergesort proves to be the basis for the most efficient external sorting programs.
- Disks are much slower than main memory, and benefit from algorithms that read and write data in long streams – not random access.

Notes

Notes

Notes

Notes