Divide-Conquer-Glue Algorithms

Quicksort, Quickselect and the Master Theorem

Tyler Moore

CS 2123, The University of Tulsa

Some slides created by or adapted from Dr. Kevin Wayne. For more information see http://www.cs.princeton.edu/~wayne/kleinberg-tardos. Some code reused from Python Algorithms by Magnus Lie Hetland.

Quickselect algorithm

- 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

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.

Partition and Select

```
1  def partition(seq):
2      pi, seq = seq[0], seq[1:]          # Pick and remove the pivo
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 plac
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
```

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
```


Seeing the Select in action

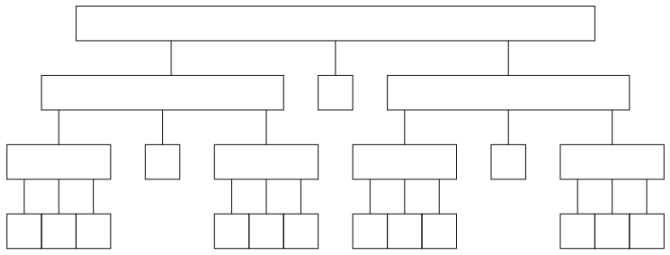
```
>>> select([3, 4, 1, 6, 3, 7, 9, 13, 93, 0, 100, 1, 2, 2, 3, 3, 2]
[1, 3, 0, 1, 2, 2, 3, 3, 2] 3 [4, 6, 7, 9, 13, 93, 100]
small partition length 9
small partition has 9 elements, so kth must be in left sequence
[0, 1] 1 [3, 2, 2, 3, 3, 2]
small partition length 2
small partition has 2 elements,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


From Quickselect to Quicksort

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

```
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
```

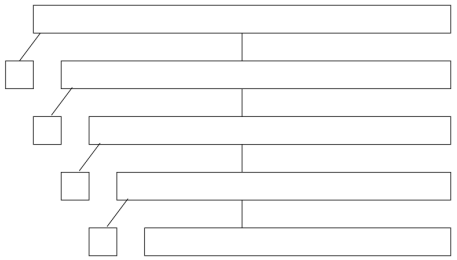

Best case for Quicksort



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 $O(n \lg n)$.

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.

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 $O(n^2)$.

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!

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?

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.

Recurrence Relations

- Recurrence relations specify the cost of executing recursive functions.
- Consider mergesort
  1. Linear-time cost to divide the lists
  2. Two recursive calls are made, each given half the original input
  3. Linear-time cost to merge the resulting lists together
- Recurrence: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$
- Great, but how does this help us estimate the running time?

Master Theorem Extracts Time Complexity from Some Recurrences

Definition

The Master Theorem For any recurrence relation of the form $T(n) = aT(n/b) + c \cdot n^k$, $T(1) = c$, the following relationships hold:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k. \end{cases}$$

- So what's the complexity of Mergesort?
- Mergesort recurrence: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$
- Since $a = 2, b = 2, k = 1$, $2 = 2^1$. Thus $T(n) = \Theta(n^k \log n)$

Apply the Master Theorem

Definition

The Master Theorem For any recurrence relation of the form $T(n) = aT(n/b) + c \cdot n^k$, $T(1) = c$, the following relationships hold:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k. \end{cases}$$

- Let's try another one: $T(n) = 3T(\frac{n}{5}) + 8n^2$
- Well $a = 3, b = 5, c = 8, k = 2$, and $3 < 5^2$. Thus $T(n) = \Theta(n^2)$

Apply the Master Theorem

Definition

The Master Theorem For any recurrence relation of the form $T(n) = aT(n/b) + c \cdot n^k$, $T(1) = c$, the following relationships hold:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k. \end{cases}$$

- Now it's your turn: $T(n) = 4T(\frac{n}{2}) + 5n$

What's going on in the three cases?

Definition

The Master Theorem For any recurrence relation of the form $T(n) = aT(n/b) + c \cdot n^k$, $T(1) = c$, the following relationships hold:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k. \end{cases}$$

1. **Too many leaves**: leaf nodes outweighs sum of divide and glue costs
2. **Equal work per level**: split between leaves matches divide and glue costs
3. **Too expensive a root**: divide and glue costs dominate

Quicksort Recurrence (Average Case)

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

$T(n) = 2T(\frac{n}{2}) + \Theta(n)$, so $T(n) = n \log n$

Selection (Average Case)

```
def select(seq, k):
    lo, pi, hi = partition(seq)      # [<= pi], pi, [> pi]
    m = len(lo)
    if m == k: return pi             # Found kth smallest
    elif m < k:                      # Too far to the left
        return select(hi, k-m-1)     # Remember to adjust k
    else:                            # Too far to the right
        return select(lo, k)         # Use original k here
```