CSE 373 Analysis of Algorithms
Fall 2016
Instructor: Prof. Sael Lee

LEC09: SORTING IV
QUICK SORT AND RANDOMIZED ALGORITHM

Lecture slide courtesy of Prof. Steven Skiena

PROBLEM OF THE DAY

Problem: You have a computer with only 2Mb of main memory. How do you use it to sort a large file of 500 Mb that is on disk?

QUICKSORT

- In practice, the fastest ***internal* sorting algorithm** is Quicksort, which uses *partitioning* as its main idea.
  - Example: pivot about 10.
  - Before: 17 12 6 19 23 8 5 10
  - After: 6 8 5 10 23 19 12 17
- Partitioning places all the elements less than the pivot in the *left* part of the array, and all elements greater than the pivot in the *right* part of the array. The pivot fits in the slot between them.
- Note that the pivot element ends up in the correct place in the total order!

QUICKSORT PSEUDOCODE

Sort(A)

quicksort(A,1,n)

```
quicksort(item_type s[], int l, int h)
{
        int p;                          /* index of partition */

        if ((h-l)>0) {
              p = partition(s,l,h);
              quicksort(s,l,p-1);
              quicksort(s,p+1,h);
        }
}
```

PARTITIONING THE ELEMENTS

- We can partition an array about the pivot in one linear scan, by maintaining three sections:
  - < pivot (to the left of *firsthigh*),
  - >= pivot (between *firsthigh* and *i*), and
  - unexplored (to the right of *i*).
- As we scan from left to right,
  - we move the left bound to the right when the element is less than the pivot,
  - otherwise we swap it with the *rightmost unexplored* element and move the right bound one step closer to the left.

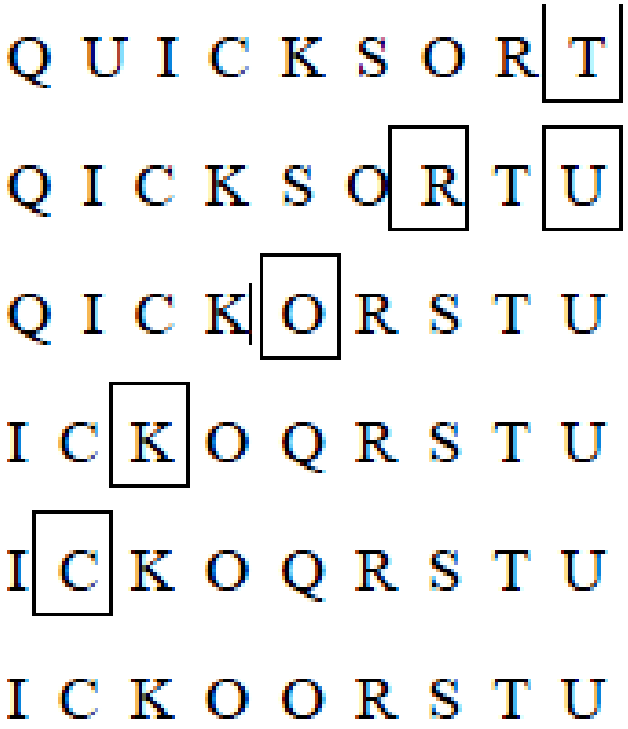
PARTITION IMPLEMENTATION

```
int partition(item_type s[], int l, int h)
{
        int i;                  /* counter */
        int p;                  /* pivot element index */
        int firsthigh;          /* divider position for pivot element */

        p = h;
        firsthigh = l;
        for (i=l; i<h; i++)
                if (s[i] < s[p]) {
                        swap(&s[i],&s[firsthigh]);
                        firsthigh ++;
                }
        swap(&s[p],&s[firsthigh]);

        return(firsthigh);
}
```

WHY PARTITION?

- Since the partitioning step consists of at most n swaps, it takes time linear in the number of keys. But what does it buy us?
  - 1. The pivot element ends up in the position it retains in the final sorted order.
  - 2. After a partitioning, no element flops to the other side of the pivot in the final sorted order.
- *Thus we can sort the elements to the left of the pivot and the right of the pivot independently, giving us a recursive sorting algorithm!*

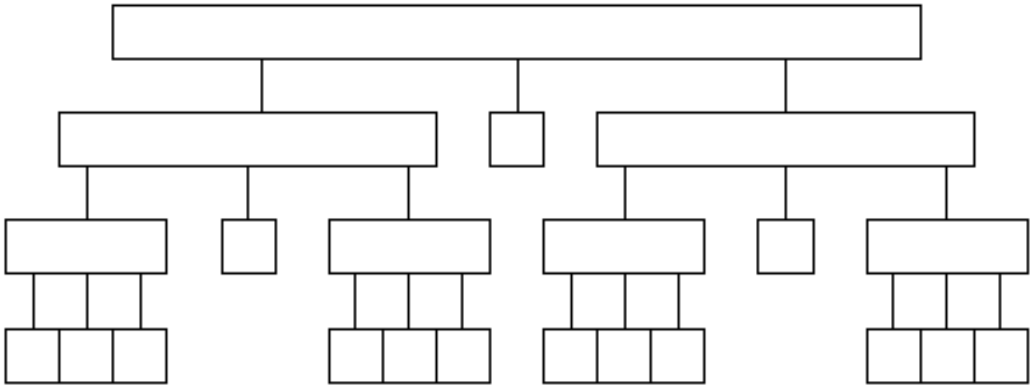
QUICKSORT ANIMATION



BEST CASE FOR QUICKSORT

- Since each element ultimately ends up in the correct position, the algorithm correctly sorts. But how long does it take?
- The best case for *divide-and-conquer* algorithms comes when we split the input as evenly as possible. Thus in the best case, each subproblem is of size n/2.
- The partition step on each subproblem is linear in its size.
- Thus the total effort in partitioning the 2^k problems of size $n/2^k$ is O(n).

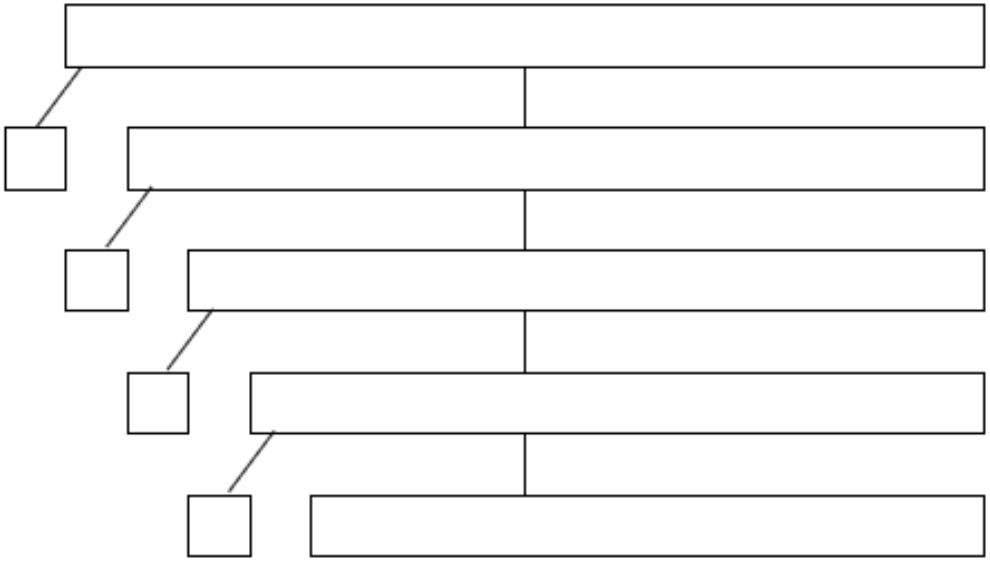
BEST CASE RECURSION TREE



- 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

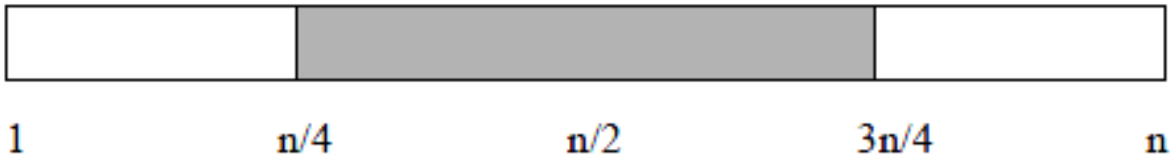
- Suppose instead our pivot element splits the array as unequally as possible. Thus instead of n/2 elements in the smaller half, we get zero, meaning that the pivot element is the biggest or smallest element in the array.



- Now we have n-1 levels, instead of lgn, for a worst case time of $\theta(n^2)$, since the first n/2 levels each have $\geq n/2$ elements to partition.
- To justify its name, Quicksort had better be good in the average case. Showing this requires some intricate analysis.
- The divide and conquer principle applies to real life. If you break a job into pieces, make the pieces of equal size!

INTUITION: THE AVERAGE CASE FOR QUICKSORT

- Suppose we pick the pivot element at random in an array of n keys.



- Half the time, the pivot element will be from the center half of the sorted array.
- Whenever the pivot element is from positions n/4 to 3n/4, the larger remaining subarray contains at most 3n/4 elements.

HOW MANY GOOD PARTITIONS

- If we assume that the pivot element is always in this range (n/4 ~ 3n/4), what is the maximum number of partitions we need to get from n elements down to 1 element?

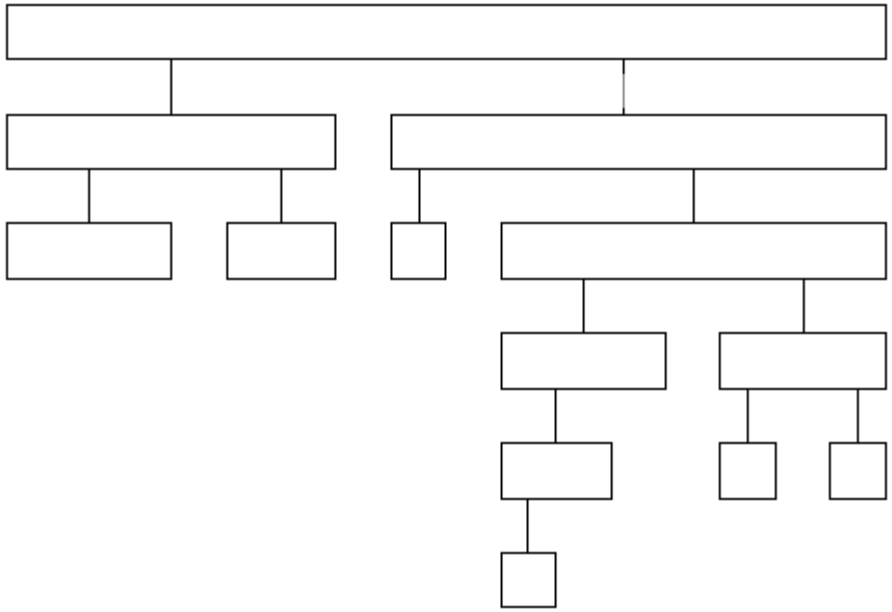
$$\left(\frac{3}{4}\right)^h * n = 1 \rightarrow n = \left(\frac{4}{3}\right)^h$$

$$\lg n = h * \lg(\frac{4}{3})$$

- Therefore $h = \lg(n) / \lg(\frac{4}{3})$ or h = O(lgn) good partitions suffice.

HOW MANY BAD PARTITIONS?

- How often when we pick an arbitrary element as pivot will it generate a good partition?
- Since any number ranked between n/4 and 3n/4 would make a decent pivot, we get one half the time on average.
- If we need *O(log n)* levels of decent partitions to finish the job, and since the expected number of good splits and bad splits is the same, the bad splits can only double the height of the tree, expected height of the tree is $\theta(\log n)$



Since O(n) work is done partitioning on each level, the average time is O(n lg n).

AVERAGE-CASE ANALYSIS OF QUICKSORT (*)

- To do a precise average-case analysis of quicksort, we formulate a recurrence given the exact expected time T(n):

$$T(n) = \sum_{p=1}^{n} \frac{1}{n}\left(T(p-1) + T(n-p)\right) + n - 1$$

- Each possible pivot p is selected with equal probability.
- The number of comparisons needed to do the partition is n - 1.
- We will need one useful fact about the Harmonic numbers H_n, namely $H_n = \sum_{i=1}^{n} 1/i \approx \ln n$
- It is important to understand (1) where the recurrence relation

- comes from and (2) how the log comes out from the summatio n. The rest is just messy algebra.

$$T(n) = \sum_{p=1}^{n} \frac{1}{n}\big(T(p-1) + T(n-p)\big) + n - 1$$

$$T(n) = \frac{2}{n}\sum_{p=1}^{n} T(p-1) + n - 1$$

$nT(n) = 2\sum_{p=1}^{n} T(p-1) + n(n-1)$ multiply by n

$(n-1)T(n-1) = 2\sum_{p=1}^{n-1} T(p-1) + (n-1)(n-2)$ apply to n-1

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2(n-1)$$

- Rearranging the terms give us:

$$\frac{\mathrm{T(n)}}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

- Substituting $a_n = \frac{A(n)}{(n+1)}$ gives

$$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)}$$

$$a_n \approx 2\sum_{i=1}^{n} \frac{1}{(i+1)} \approx 2\ln n$$

- We are really interested in A(n), so

$$A(n) = (n+1)a_n \approx 2(n+1)\ln n \approx 1.38n\lg n$$

PICK A BETTER PIVOT

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 $O(n^2)$.

IS QUICKSORT REALLY FASTER THAN HEAPSORT?

- Since Heapsort is $\Theta(n \lg n)$ and selection sort is $\Theta(n^2)$, there is no debate about which will be better for decent-sized files.
- When Quicksort is implemented well, it is typically 2-3 times faster than mergesort or heapsort.
- The primary reason is that the operations in the innermost loop are simpler.
- Since the difference between the two programs will be limited to a multiplicative constant factor, the details of how you program each algorithm will make a big difference.

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:

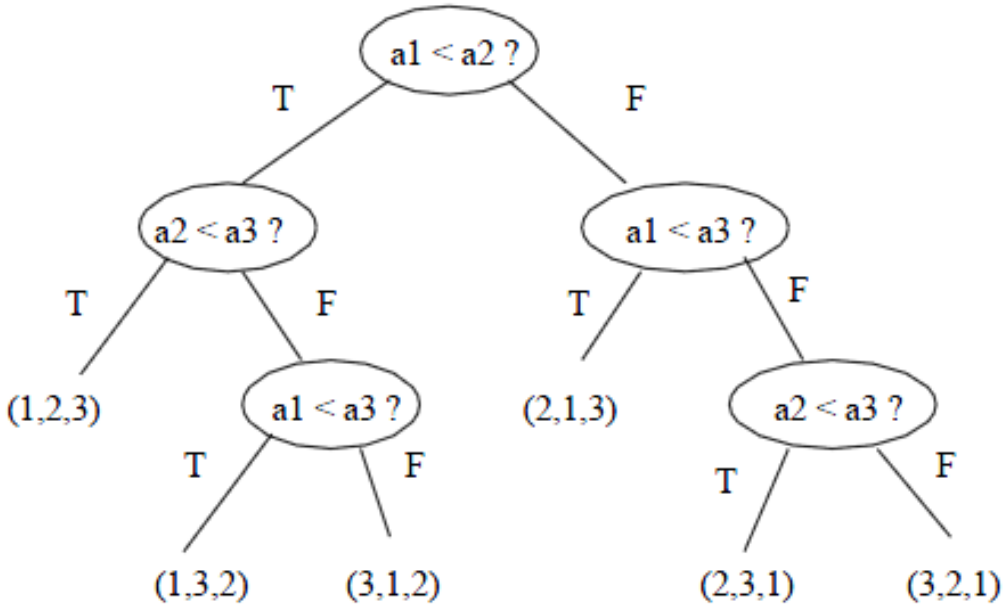
  "If you give me random input data, quicksort runs in expected $\Theta(n \lg n)$ time."

IMPORTANCE OF RANDOMIZATION

- Since the time bound now does not depend upon your input distribution, this means that unless we are *extremely* unlucky 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 won't see it.

CAN WE SORT $o(nlgn)$

- Any **comparison-based** sorting program can be thought of as defining a decision tree of possible executions.
- Running the same program twice on the same permutation causes it to do exactly the same thing, but running it on different permutations of the same data causes a different sequence of comparisons to be made on each.



Claim: the height of this decision tree is the worst-case complexity of sorting.

LOWER BOUND ANALYSIS

Since any two different permutations of n elements requires a different sequence of steps to sort, there must be at least n! different paths from the root to leaves in the decision tree.

Thus there must be at least n! different leaves in this binary tree.

Since a binary tree of height h has at most 2^h leaves, we know $n! \leq 2^h$ or $h \geq \lg(n!)$.

By inspection $n! > (\frac{n}{2})^{\frac{n}{2}}$, since the last n/2 terms of the product are each greater than n/2. Thus

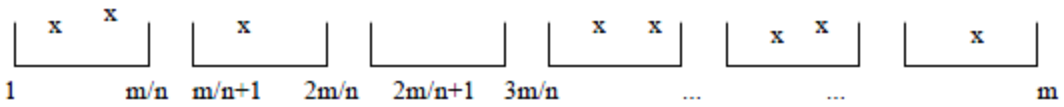
$$\log(n!) > \log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) = \frac{n}{2}\log\left(\frac{n}{2}\right) \rightarrow \Theta(n\log n)$$

NON-COMPARISON-BASED SORTING

- All the sorting algorithms we have seen assume binary comparisons as the basic primitive, questions of the form “is x before y?”.
- But how would you sort a deck of playing cards?
- Most likely you would set up 13 piles and put all cards with the same number in one pile. With only a constant number of cards left in each pile, you can use insertion sort to order by suite and concatenate everything together.
- If we could find the correct pile for each card in constant time, and each pile gets O(1) cards, this algorithm takes O(n) time.

BUCKETSORT

- Suppose we are sorting n numbers from 1 to m, where we know the numbers are approximately uniformly distributed.
- We can set up n buckets, each responsible for an interval of m/n numbers from 1 to m



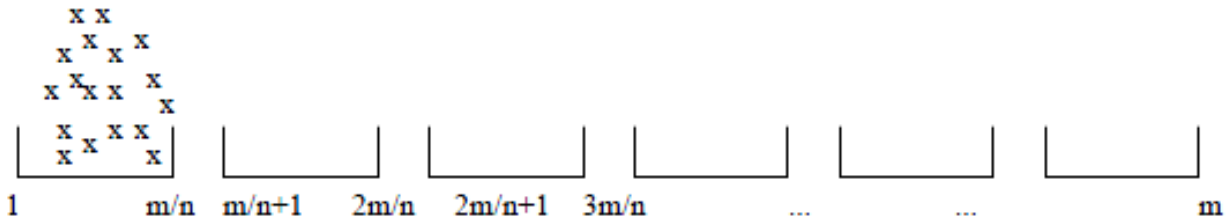
- Given an input number x, it belongs in bucket number $\lceil xn/m \rceil$.
- If we use an array of buckets, each item gets mapped to the right bucket in O(1) time.

BUCKETSORT ANALYSIS

- With uniformly distributed keys, the expected number of items per bucket is 1. Thus sorting each bucket takes O(1) time!
- The total effort of bucketing, sorting buckets, and concatenating the sorted buckets together is O(n)
- What happened to our $\Omega(n \lg n)$ lower bound!

WORST-CASE VS. ASSUMED-CASE

- Bad things happen to bucketsort when we assume the wrong distribution



- We might spend linear time distributing our items into buckets and learn *nothing*.
- Problems like this are why we worry about the worst-case performance of algorithms!

REALWORLD DISTRIBUTIONS

- The worst case “shouldn’t” happen if we understand the distribution of our data.
- Consider the distribution of names in a telephone book.
  - Will there be a lot of Skiena’s?
  - Will there be a lot of Smith’s?
  - Will there be a lot of Shifflett’s?
- Either make *sure* you understand your data, or use a good worst-case or randomized algorithm!