Lecture5:
Sorting Algorithms

GE - Java sessions Fall 2015

Mohamed Wiem Mkaouer

Slides Reference : Prof. Skiena | Prof. Elenbogen | Prof. Welch

Data Structure Visualizations

https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

Mergesort

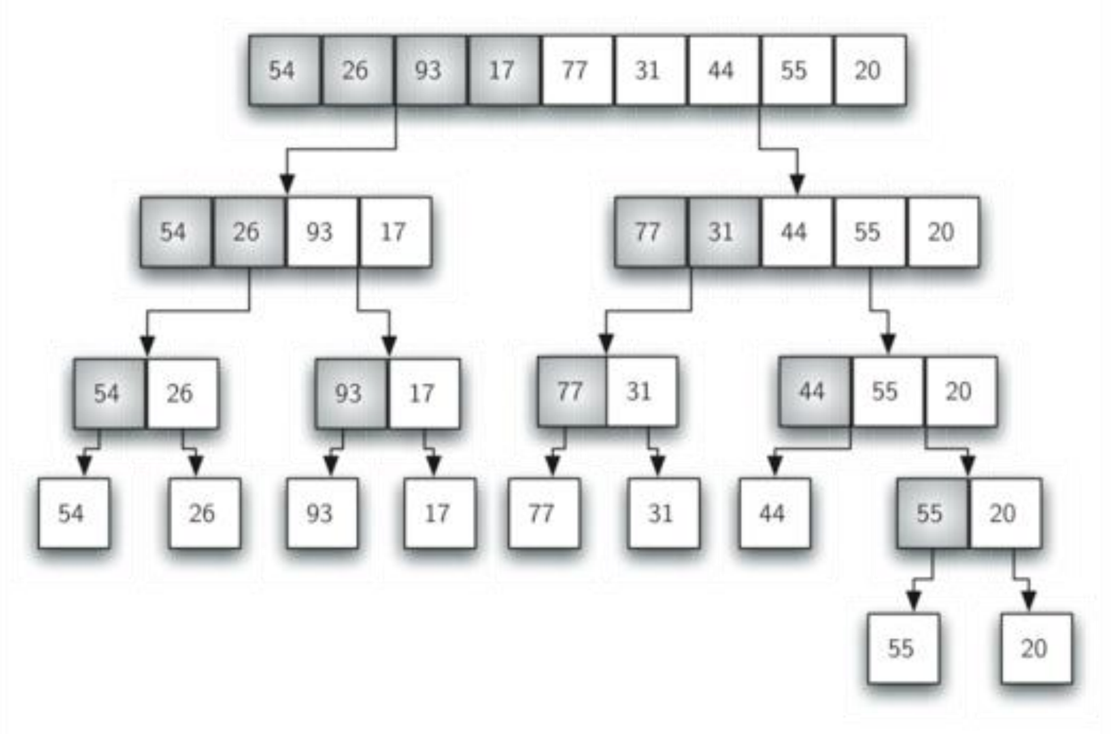
Recursive algorithms are based on reducing large problems into small ones.

A nice recursive approach to sorting involves partitioning the elements into two groups, sorting each of the smaller problems recursively, and then interleaving the two sorted lists to totally order the elements.

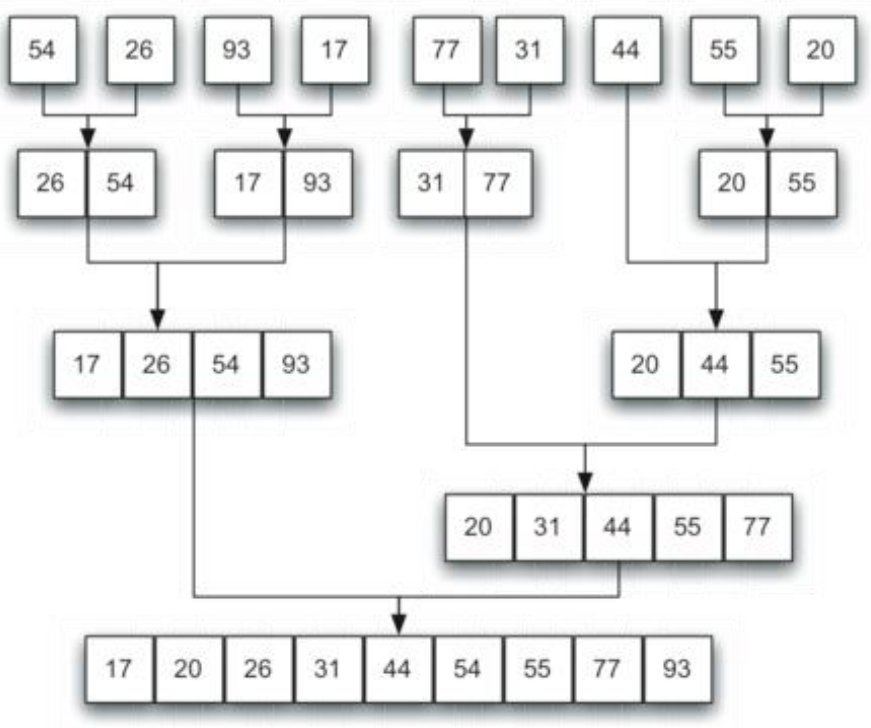
Mergesort Implementation

```
mergesort(item_type s[], int low, int high)
{
        int i; (* counter *)
        int middle; (* index of middle element *)

        if (low < high) {
                middle = (low+high)/2;
                mergesort(s,low,middle);
                mergesort(s,middle+1,high);

                merge(s, low, middle, high);
        }
}
```

Mergesort Animation



Mergesort Animation



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 slighly 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 $n - 1$ comparisons or $O(n)$ total work
Example: $A = \{5, 7, 12, 19\}$ and $B = \{4, 6, 13, 15\}$.

Buffering

Although mergesort is $O(n \lg n)$, it is inconvenient to implement with arrays, since we need extra space to merge. the lists.

Merging $(4, 5, 6)$ and $(1, 2, 3)$ would overwrite the first three elements if they were packed in an array.

Writing the merged list to a buffer and recopying it uses extra space but not extra time (in the big Oh sense).

External Sorting

Which $O(n \log n)$ algorithm you use for sorting doesn't 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.

Divide and Conquer

Divide and conquer is an important algorithm design technique used in mergesort, binary search the fast Fourier transform (FFT), and Strassen's matrix multiplication algorithm.

We divide the problem into two smaller subproblems, solve each recursively, and then meld the two partial solutions into one solution for the full problem.

When merging takes less time than solving the two subproblems, we get an efficient algorithm.

Quicksort

In practice, the fastest *internal* sorting algorithm is Quicksort, which uses *partitioning* as its main idea.

Example: pivot about 10.

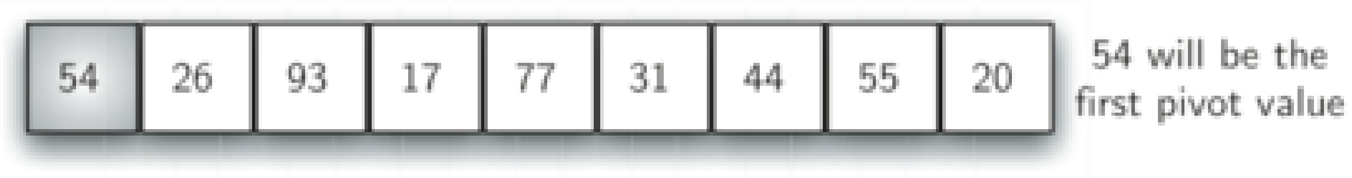
Before: 17 12 6 19 23 8 5 10

After:: 6 8 5 10 23 19 12 17

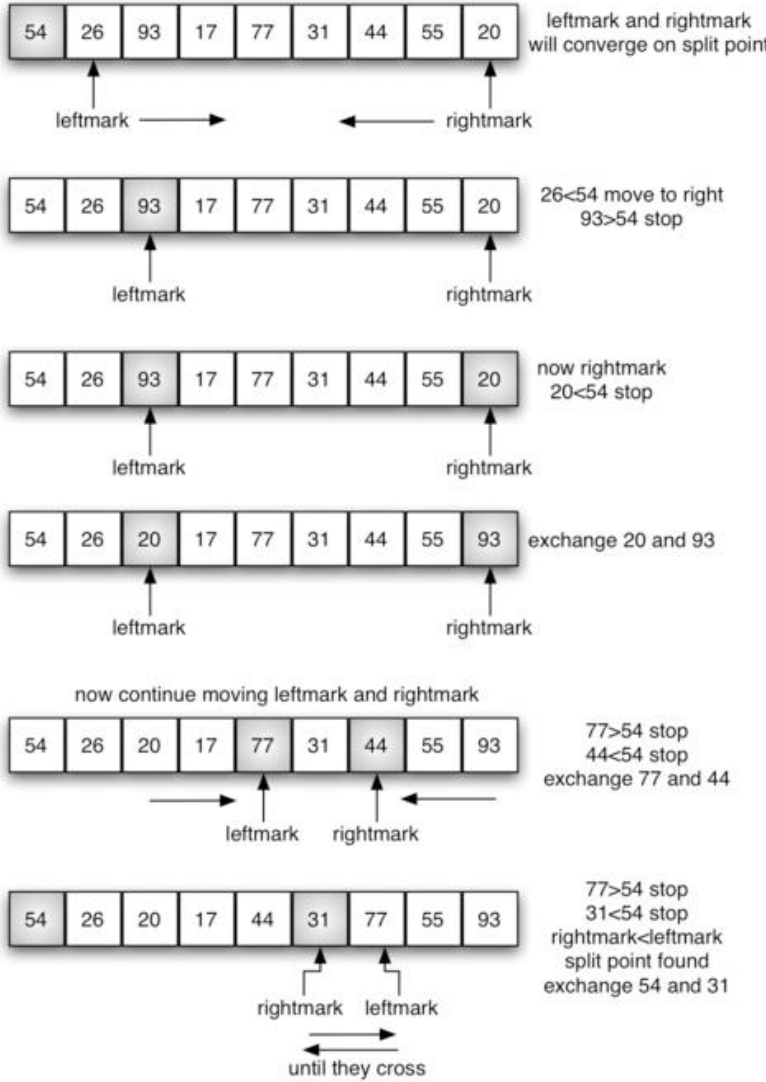
Partitioning places all the elements less than the pivot in the *left* part of the array, and all elements greater than the pivot in the *right* part of the array. The pivot fits in the slot between them.

Note that the pivot element ends up in the correct place in the total order!

Quicksort Animation

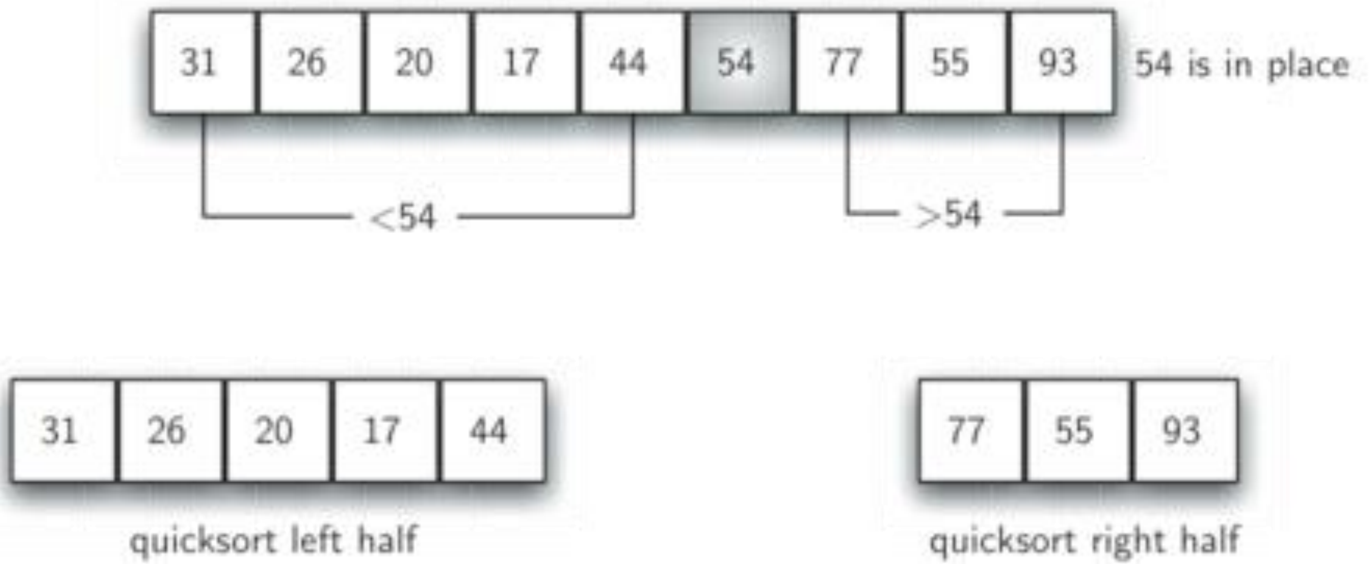


Quicksort



54 26 93 17 77 31 44 55 20
leftmark and rightmark
will converge on split point
leftmark
rightmark
54 26 93 17 77 31 44 55 20
26<54 move to right
93>54 stop
leftmark
rightmark
54 26 93 17 77 31 44 55 20
now rightmark
20<54 stop
leftmark
rightmark
54 26 20 17 77 31 44 55 93
exchange 20 and 93
leftmark
rightmark
now continue moving leftmark and rightmark
54 26 20 17 77 31 44 55 93
77>54 stop
44<54 stop
exchange 77 and 44
leftmark
rightmark
54 26 20 17 44 31 77 55 93
77>54 stop
31<54 stop
rightmark<leftmark
split point found
exchange 54 and 31
rightmark
leftmark
until they cross

Quicksort Animation



Partitioning the Elements

We can partition an array about the pivot in one linear scan, by maintaining three sections: < pivot, > pivot, and unexplored. 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.

Why Partition?

Since the partitioning step consists of at most n swaps, 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 Pseudocode

```
Sort(A)
        Quicksort(A,1,n)

Quicksort(A, low, high) if (low <
        high)
                pivot-location = Partition(A,low,high)
                Quicksort(A,low, pivot-location - 1)
                Quicksort(A, pivot-location+1, high)
```

Partition Implementation

```
Partition(A,low,high)
        pivot = A[low]
        leftwall = low
        for i = low+1 to high
                if (A[i] < pivot) then
                        leftwall = leftwall+1
                        swap(A[i],A[leftwall])
        swap(A[low],A[leftwall])
```

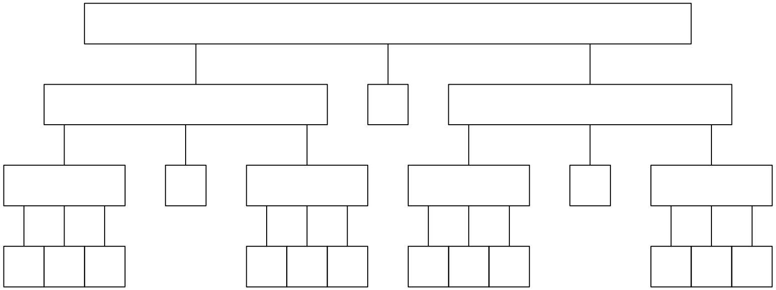
Best Case for Quicksort

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 $O(n)$.

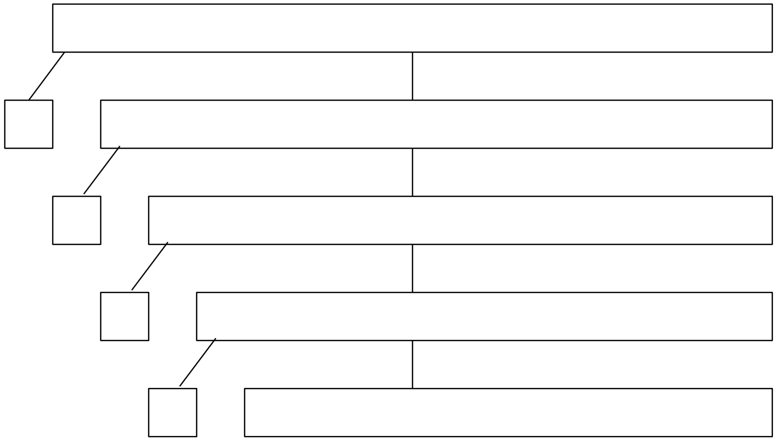
Best Case Recursion Tree



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.



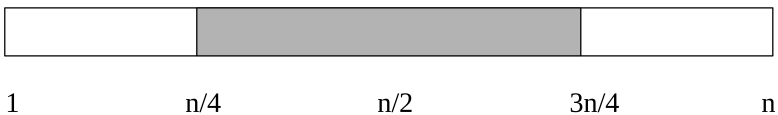
Now we have $n-1$ levels, instead of $\lg n$, for a worst case time of $\Theta(n^2)$, since the first $n/2$ levels each have $\geq n/2$ elements to partition.

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, what is the maximum number of partitions we need to get from n elements down to 1 element?

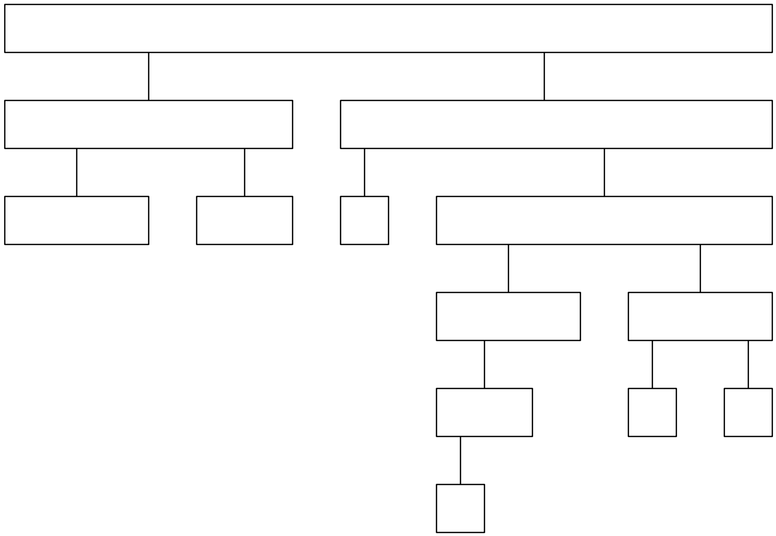
$$(3/4)^l \cdot n = 1 \longrightarrow n = (4/3)^l$$

$$\lg n = l \cdot \lg(4/3)$$

Therefore $l = \lg(4/3) \cdot \lg(n) < 2 \lg n$ good partitions suffice.

How Many Bad Partitions?

How often when we pick an arbitrary element as pivot will it generate a decent 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 $2\lg n$ levels of decent partitions to finish the job, and half of random partitions are decent, then on average the recursion tree to quicksort the array has $\approx 4\lg n$ levels.



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 $T(n)$:

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

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 summation. The rest is just messy algebra.

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

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

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

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

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

rearranging the terms give us:

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

substituting $a_n = A(n)/(n+1)$ 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

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

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 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 won't see it.

Non-Comparison-Based Sorting

All the sorting algorithms we have seen assume binary comparisons as the basic primative, 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

Can we beat O(n log n)? Yes but we have to abandon comparison. Say we are sorting integers and know the Maximum key, N. Also we do not care about how much space we will use. Then we could but integers of the unsorted sequence into the array at the rank of the integer value. Then we can unload the array back into the sequence. The time of the sort is O(n+N).

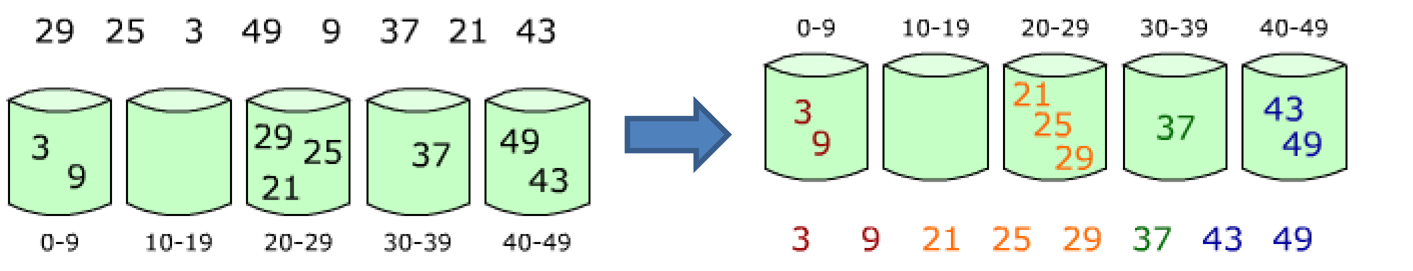
Bucketsort

Algorithm bucketSort(S)
Input: Sequence , S, of items with integer keys between 0 and N-1
Output: Sequence S sorted

Let B be and array size N initially empty

for each item *x* in S **do** **// n iterations of the loop size of S**
    let *k* be the key of *x*
    remove *x* form S and insert it into B[*k*]

for *i* = 0 to N-1 **do** **// N iterations the size of B**
    remove *x* from B[*i*] and insert at the end of S



Bucketsort Analysis

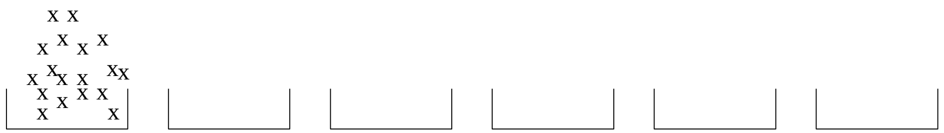
With uniformly distributed keys, the expected number of items per bucket is 1. Thus sorting each bucket takes $O(1)$ time!

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!

Real World 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!

The Shifflett’s of Charlottesville

For comparison, note that there are seven Shifflett’s (of various spellings) in the 1000 page Manhattan telephone directory.

Shifflett Debbie K Ruckersville 985-7957
Shifflett Debra S SR 617 Quinque 985-8813
Shifflett Delma SR609 985-3688
Shifflett Delmas Crozet 823-5901
Shifflett Dempsey & Marilynn
100 Greenbrier Ter 973-7195
Shifflett Denise Rt 627 Dyke 985-8097
Shifflett Dennis Stanardsville 985-4560
Shifflett Dennis H Stanardsville 985-2924
Shifflett Dewey E Rt667 985-6576
Shifflett Dewey O Dyke 985-7269
Shifflett Diana 508 Bainbridge Av 979-7035
Shifflett Doby & Patricia Rt6 286-4227
Shifflett Don&Ola Rt 621 974-7463

Shifflett James 2219 Williamsburg Rd
Shifflett James B 801 Stonehenge Av
Shifflett James C Stanardsville
Shifflett James E Earlysville
Shifflett James E Jr 552 Cleveland Av
Shifflett James F & Lois LongMeadow
Shifflett James F & Vernell Rt671
Shifflett James J 1430 Rugby Av
Shifflett James K St George Av
Shifflett James L SR33 Stanardsville
Shifflett James O Earlysville
Shifflett James O Stanardsville
Shifflett James R Old Lynchburg Rd
Shifflett James R Rt733 Esmont

Big O complexity

http://bigocheatsheet.com/