Limits to Comparison Sorting, and Bucket Sort

Algorithm	Worst Case Complexity
Bubblesort	$O(n^2)$
Selectionsort	$O(n^2)$
Insertionsort	$O(n^2)$
Heapsort	O(n log n)
Mergesort	O(n log n)
Quicksort	$O(n^2)$

Can we do any better?

All of the algorithms we have looked at have been based on repeated comparisons between pairs of input elements.

- they do other work as well – dividing, copying, etc
- but the comparisons determine the order

The comparisons seem to provide the biggest chunk of the complexity.

How many comparisons do we really need to do?

Consider an arbitrary sorting algorithm that uses *only* comparisons to decide on the relative placement of elements.

Consider an arbitrary input list consisting of some permutation of n distinct elements.

Build a binary tree to represent all possible sequences of comparisons that our algorithm does to the input list.

At each node in the tree, if the test is true, branch left; if it is false, branch right.

We can only stop a branch once we have gathered enough knowledge to determine the final sorted order.

Bubble sort a list of four elements V1, V2, V3, V4.
At the start, we know nothing about their relative ordering.
Build the tree of all possible sequences of comparisons:

Consider any comparison sort of a list of three elements V1, V2 and V3.
At the start, we know nothing about their relative ordering.
With each node, write down everything we have learned in the current path.

Suppose we now do the same for input lists of *n* distinct items.

Every different input sequence must end up at a different leaf node in the tree.

Proof?

Suppose two different input sequences end up at the same leaf.
The leaf determines the final order of the input elements by initial position.
But in the inputs, there must have been some case where Vi < Vj in one input, and Vj < Vi in the other.
For those two positions, the leaf will definitely choose one of the two orderings.
So one of the two input lists must end up with an incorrectly sorted output.
But each leaf in our tree correctly sorts the input.
Contradiction
So two different inputs must end up at different leaves.

This means there must be one leaf for each possible permutation of the n distinct items.

We know there are $n!$ different permutations. So there are $n!$ leaf nodes.

There must therefore be at least $n!$ nodes in the tree.
So the tree must have depth at least $\log(n!)$.

But $n! = n*(n-1)*(n-2)*...*2*1$
and there are at least $n/2$ factors $\geq n/2$ in that product.

So $n! \geq (n/2)^{n/2}$, and so $\log(n!) \geq \log((n/2)^{n/2})$
But we know that $\log a^b = b \log a$
So $\log(n!) \geq (n/2) \log(n/2) = (1/2)\, n \log (n/2) = (1/2)n((\log n)-1) = \Omega(n \log n)$

So the depth of the tree is $\Omega(n \log n)$

So there is at least one path to a leaf of length $\Omega(n \log n)$, which means at least one input sequence requires $\Omega(n \log n)$ comparisons.

Our binary decision tree model can represent *any* sorting algorithm that makes its decisions based only on comparisons.

Whichever sorting algorithm we represent, if it is to correctly sort any permutation of the n items, at least one of these permutations will require $\Omega(n \log n)$ comparisons.

So this is a lower bound for all comparison-based sorting.

We cannot get a comparison-based sorting algorithm which has its worst case complexity significantly better than heapsort or mergesort.

Note: heapsort and mergesort are $\Theta(n \log n)$ in worst case.

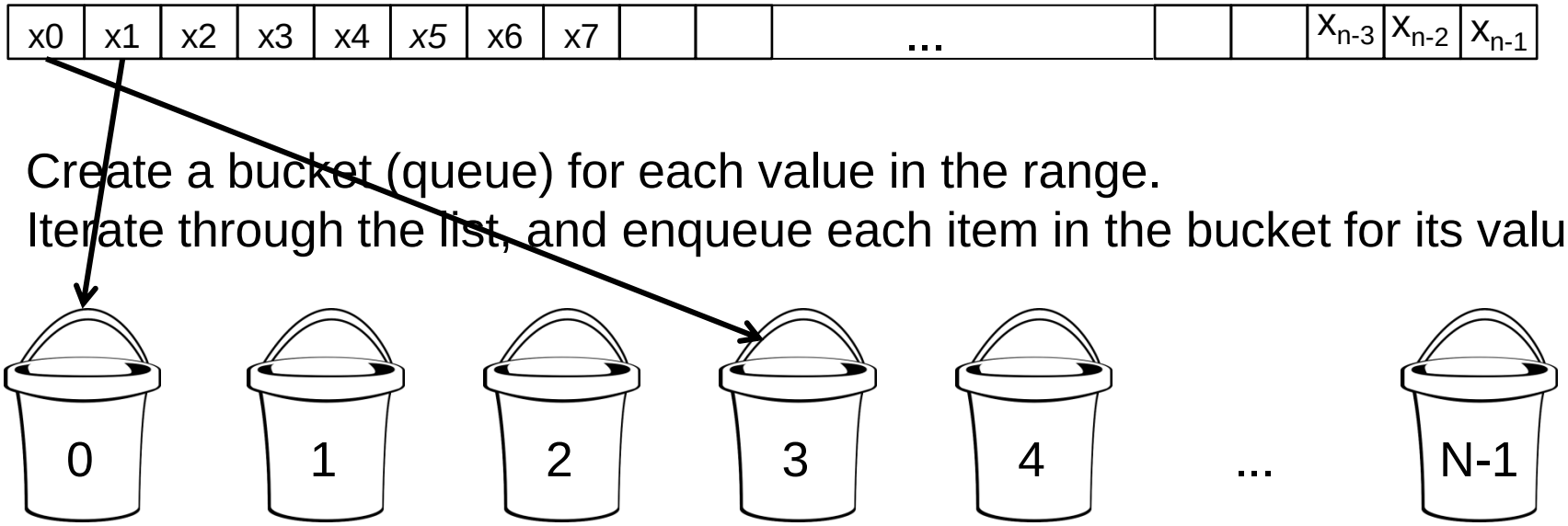
If we want to find a better sorting algorithm, then there are only two options:

(i) accept that we are not going to get anything *significantly* better, and just look at improving the expected case, or making relatively small improvements to the worst case

(ii) find some way of sorting that is not based on comparing pairs of input elements.

Suppose we knew that all the elements in our input list had values within a limited range – say [0, N-1].

(Or the elements have keys whose values are in [0,N-1])



When the list has been processed, take each bucket in turn, and dequeue its items into the list in sequence.

```
pseudocode bucketsort(list, N):
    for each i from 0 to N-1
        create a queue bucket[i]
    for each element in list
        v = element.key
        bucket[v].enqueue(element)
    j = 0
    for each i from 0 to N-1
        while bucket[i] is not empty
            list[j] = bucket[i].dequeue()
            j = j + 1
```

If N is not big compared to n (i.e. if N is O(n)), then the complexity is O(n).

If N is $O(n^2)$ or worse then the complexity is $O(n^2)$ or worse

Analysis: N steps to create each bucket, 2n steps to read each element and add to correct bucket, N steps to process each bucket, and a total of n assignments of elements back into list.
This is O(N+n).

Space requirement is also O(N+n)

Next lecture

Stable sorting

Radix Sort
Python's built-in sort