Limits to Comparison Sorting
Counting 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 of elements

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

How many comparisons do we really need to do (for an algorithm that is *guaranteed* to work on any input list)?

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 [x, y, z, t].
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 [x, y, z].
At the start, we know nothing about their relative ordering.
With each node, write down everything we have learned in the current path about the relative ordering of the elements.
The leaf nodes determine the final order of the sorted list.

There will be 6 leaf nodes.

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 by contradiction ...

Suppose two different input sequences end up at the same leaf.
The leaf determines the total order of the input elements by initial position.
But in the inputs, there must have been some case where L[i] < L[j] in one input, and L[j] < L[i] 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 $n! = n*(n-1)*(n-2)*...*2*1$

$= n*(n-1)*(n-2)*...*(n-\lfloor n/2 \rfloor)*(n-\lfloor n/2 \rfloor-1)*(n-\lfloor n/2 \rfloor-2)* ... * 1$

$> n*(n-1)*(n-2)*...*(n-\lfloor n/2 \rfloor)$

$> (n/2) * (n/2) * ... * (n/2)$

$= (n/2)^{n/2}$

So $\log(n!) \geq \log((n/2)^{n/2})$

$\geq (n/2) \log(n/2)$

$= (1/2)\, n\, (\log n - \log 2)$

$\geq (1/2)n((\log n)-1)$

$= (1/2)n \log n - (1/2)n = \Omega(n \log n)$

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

$\log a^b = b \log a$

$\log (a/b) = \log a - \log b$

$\log_2 2 = 1$

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

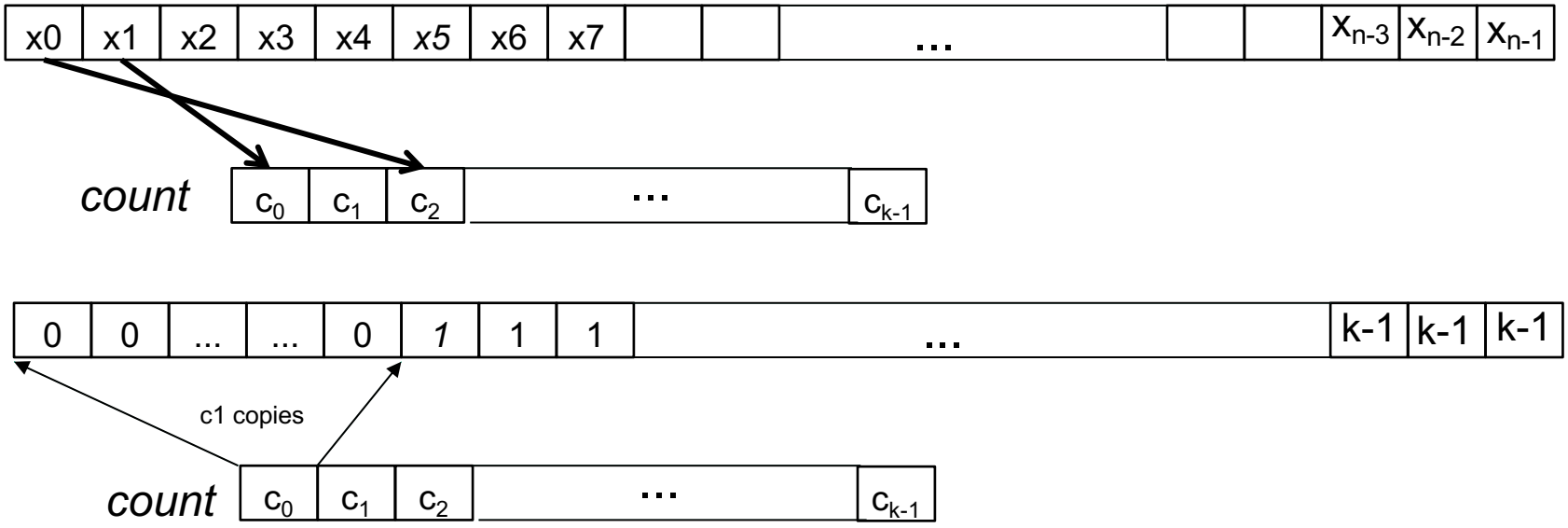
(ii) find some way of sorting that is not based on comparing pairs of input elements.

Counting Sort

Suppose we knew that all the elements in our input list had values within a limited range – say {0,1,...,k-1} – and where two elements of the same value are indistinguishable.

Create a new list of 0s, called *count*, with a cell for each value in the range. Iterate through the input, and for each value, increment its cell in *count.*

When the input has been processed, take each cell i of *count* in turn, and insert *count*[i] instances of i into the input list, overwriting the contents.



```
pseudocode countingsort(list, k)
    # list of length n contains integers in {0,... k-1}
    create list count of length k filled with 0s
    for each x in list
        increment count[x]
    i = 0  # index in input list
    j = 0  # index in count
    while j < k
        num = count[j]
        while num > 0
            list[i] = j
            i++
            num--
        j++
```

E.g. sort
4 2 3 4 0 2 1 3 4 1 2 0 3 4 3 1

What is the worst case complexity?

O(n + k)

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

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

Space requirement is also O(n + k)

Simple counting sort does not work if the individual items matter (e.g. if we were sorting a list of object references, where one field of the object was a sort key).

Counting sort just counts the number of times the key appears, and then creates the right number of entries. The original object data is lost.

Instead, we will transform *count* to record, for each cell i, the number of entries that are ≤ i

iterate though *count* and sum entries so far

Then step backwards through input list, and for each entry, lookup *count* to see how many items are ≤, decrement by 1, and copy the entry into the cell index indicated by count.

```
E.g.   sort
2a 0b 2c 1d 2e 0f

0 1 2
2 1 3

0 1 2
2 3 6
1 2 5
0   4
    3

0  1  2  3  4  5
0b 0f 1d 2a 2c 2e
```

```
pseudocode countingsort2(list, k)
    # list of length n contains integers in {0,... k-1}
    create list count of length k
    initialise each cell of count to 0
    for each x in list
        increment count[x]
    for each index j from 1 to k-1
        count[j] = count[j-1] + count[j]
    create outlist of length n
    i = n-1  # index in input list
    while i ≥ 0
        count[j] = count[j]-1
        outlist[count[j]] = list[i]
        i--
```

O(n + k)

E.g. sort

4	2	3	4	0	2	1	3	4	1	2	0	3	4	3	1
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Next lecture

Stable sorting

Radix Sort
Python's built-in sort