Sorting Algorithms

Correctness, Complexity and Other Properties

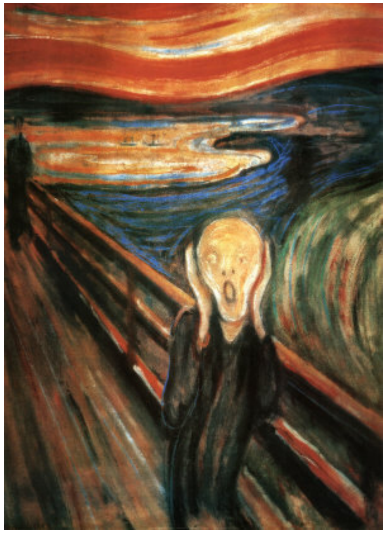
Joshua Knowles

School of Computer Science

The University of Manchester

COMP26120 - Week 9 LT1.1, 22nd November 2013

Mystery Pictures



TASKS:

1. Write down the name of a sorting algorithm.
2. What do the above pictures depict?

Mystery Pictures

TASKS:

1. Write down the name of a sorting algorithm.
2. What do the above pictures depict?

Sorting — A Lot to Learn

In this lecture, we'll learn

- What to learn
- ***HOW to learn it*** (important)
- ... and **a few** of the basics

After the lecture, you need to be active to learn the rest!

Motivation: The Importance of Sorting

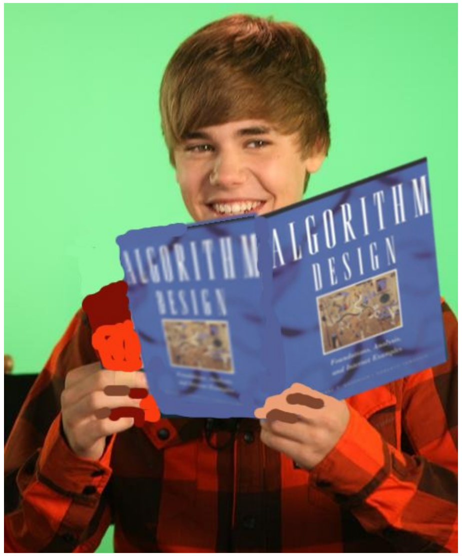
Important because

- Fundamental to organizing data

- Provides excellent practice of ***Complexity and Correctness*** thinking
- Employers love to ask about it
- On the other hand...
  - Progress in computer speed and memory has reduced the practical importance of further developments in sorting
  - ***quicksort() is often an adequate answer in many applications***

Sorting in Your Textbook

Every algorithms textbook devotes many pages to Sorting, and yours is no exception...



The COMP26120 Sorting Syllabus

What you should learn about sorting (what is examinable)

- Definition of sorting. Meaning of correctness of sorting algorithms
- Complexity bounds on the task of sorting
- How the following work: **Bubble sort**, ***Insertion sort***, **Selection sort**, ***Quicksort***, **Merge sort**, ***Heap sort***, **Bucket sort**, ***Radix sort***
- Main properties of those algorithms
  - Space and Time Complexity (basic)
  - Stability
- How to reason about complexity — worst case and special cases

Covered in: **the course book**; labs; this lecture; online resources including wikipedia; sorting animations; **wider reading**

Relevant Pages of the Course Text Book

Selection sort: 97 (very short description only)
Insertion sort: 98 (very short)
Merge sort: 219–224 (pages on multi-way merge not needed)
Heap sort: 100–106 and 107–111
Quicksort: 234–238
Bucket sort: 241–242
Radix sort: 242–243
Lower bound on sorting 239–240
Practical issues, 244

Some of the exercise on pp. 251–252. (More advice to follow)

Definition of Sorting

What is a sorting algorithm?

Input:
an (ordered) array of ***keys***, $array$

All keys must be ***comparable***. The keys must form ***a total order***:
If X is totally ordered under $\leq$, then the following statements hold for all a, b and c in X:

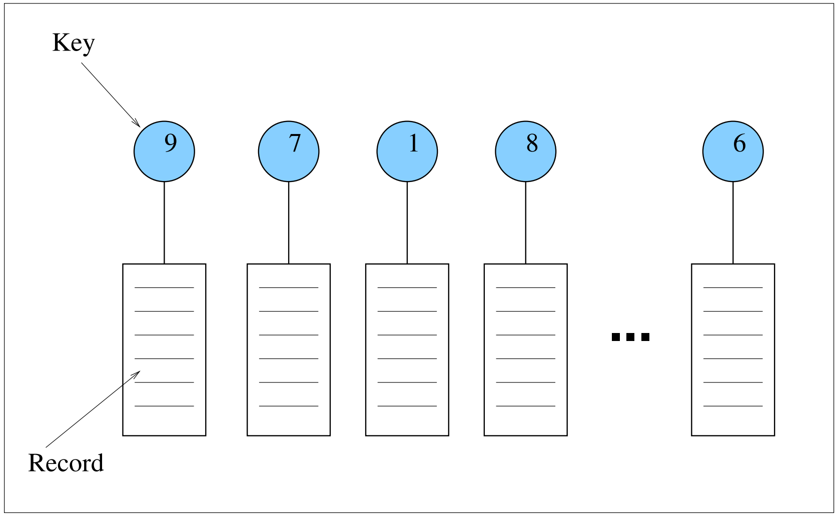
If $a \leq b$ and $b \leq a$ then $a = b$ (antisymmetry);
If $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity);
$a \leq b$ or $b \leq a$ (totality).

(Note: an array of items all with the same value is still a total order).

Output:
a sorted array $sortedArray$ such that

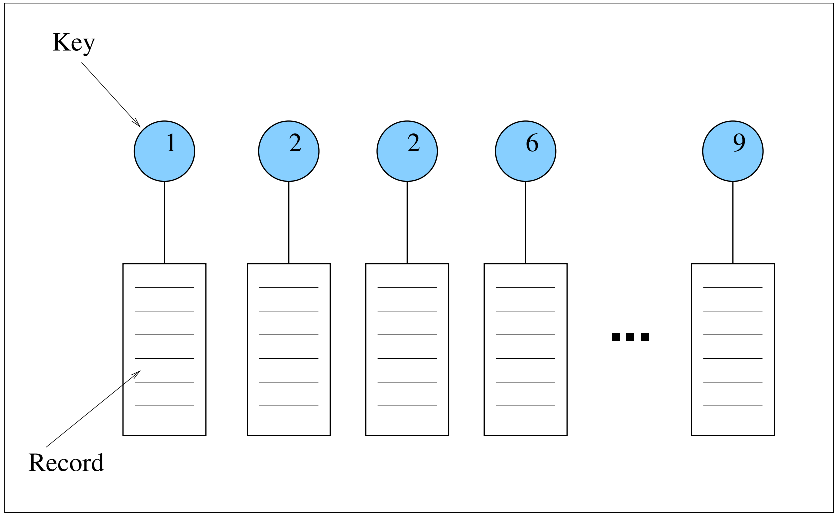
1. $sortedArray$ is a permutation of $array$ and
2. keys are in ascending order: $sortedArray[i] <= sortedArray[j]$ for all i, j where $i < j$.

Keys and *Records* View of Sorting



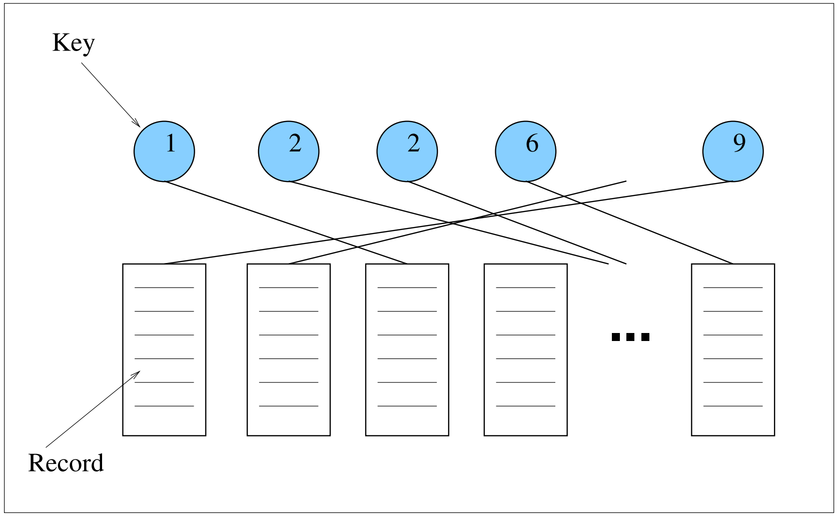
We sort ***keys***. Keys may be part of larger *Records*. We could move the whole records about, or use pointers to them. (Moving whole records may be costly).

Keys and *Records* View of Sorting



We sort ***keys***. Keys may be part of larger *Records*. We could move the whole records about, or use pointers to them. (Moving whole records may be costly).

Keys and *Records* View of Sorting



We sort ***keys***. Keys may be part of larger *Records*. We could move the whole records about, or use pointers to them. (Moving whole records may be costly).

Not Only Numbers

Much of the time, we want to sort numbers into numerical order. Comparison is then straightforward.

But we also sort:

- playing cards (ace high or not)
- names and words (proper nouns first or not)
- dates, written in different forms
- complex, multi-field data
- books, by e.g. Dewey-Decimal System

What are the rules for comparison? Do they vary? **Homework:** Write the compare(a,b) function for the date written in three parts, as in January 24 1989. Note: the compare(a,b) function should implement $a \leq b$?

But all of these lists are ***totally ordered***. So it is possible to sort them correctly, provided we get the comparison rules right.

[Demo - See demo1.c]

For an example where the compare (i.e., $\leq$) function does not properly define a total order of the keys, see the demo:

[Demo - See demo2.c]

The sorting algorithm FAILS to behave properly, even though it is correct. Why?

Non-transitivity: Rock - Paper - Scissors

In the game, ***Rock - Paper - Scissors*** we have
Rock $>$ Scissors
Scissors $>$ Paper
Paper $>$ Rock
It is impossible to determine what comes first. This is not a strictly ordered set of things (not a total order).

[DEMO - See demo3.c]

The problem is that we are not using the $>$ sign in the correct way. It must be a ***transitive*** relation. In Rock-Paper-Scissors it is non-transitive.

Sorting is Easy (Bounds on Complexity)

Where does sorting appear in a list of complexities?

$O(1), O(\log n), O(\sqrt{n})$, $\boxed{O(n), O(n \log n), O(n^2)}$, $O(n^3), ..., O(2^n), O(n!)$

Common sorting methods are upper-bounded by a polynomial — they are all $O(n^2)^\dagger$.

But in practice, since n can be quite large (thousands, millions, billions), $O(n \log n)$ methods are valuably quicker than $\Omega(n^2)$ methods. Hence the research effort in devising efficient sorting methods.

† Remember, an algorithm with $O(n \log n)$ complexity is also in $O(n^2)$.

Time Complexity Bounds for Sorting

Upper Bound Any existing sorting algorithm provides an ***upper bound*** for the task of sorting.

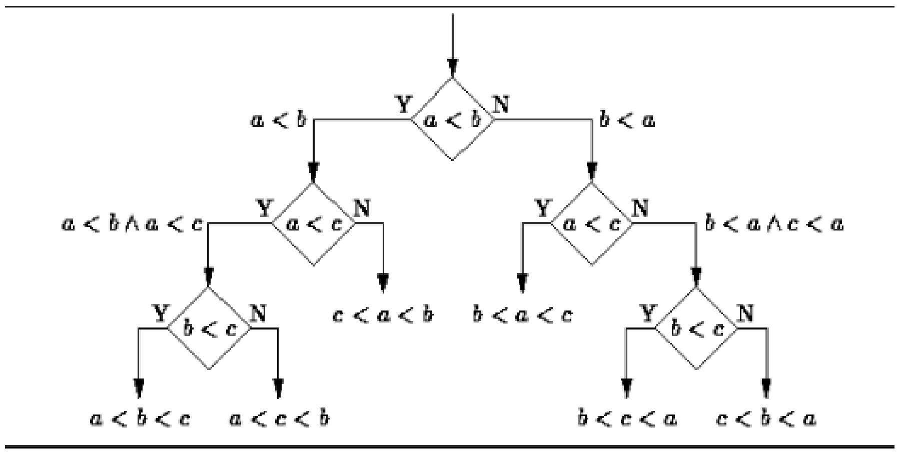
E.g. Mergesort has worst-case time complexity of $O(n \log n)$. In other words, from the existence of Mergesort, we know it *is possible* to sort n elements in at most $c.n \log n$ comparisons for some sufficiently large constant c and all $n > n_o$, where n_o is a constant. (In practice, $c = 2$ and $n_o = 10$ would be sufficient)

Lower Bound What is the ***minimum*** amount of work needed to sort an unsorted array? We must at least have to read each value in the array, so ***a lower bound is*** $O(n)$.

Bucket sort (also Counting sort) and Radix sort achieve this lower bound, but only on *restricted* types of inputs.

Lower Bound for Comparison-Based Sorting

No algorithm based on ***comparison*** of keys can sort a worst-case input sequence in fewer than $\lceil \log_2(n!) \rceil$ comparisons. This is lower bounded by $\Omega(n \log_2 n)$.



The input sequence is a, b, c, and $a \neq b$, $b \neq c$, $a \neq c$. We can see there are $n!$ leaves in the decision tree, one for each possible permutation of the input. In order to be able to locate the correct order (for the particular values of a,b,c), the decision tree must have a depth of $\lceil \log_2(n!) \rceil$.

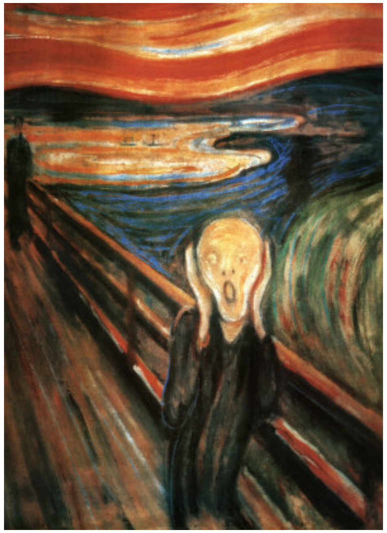
And

$$\begin{aligned}
\lceil \log_2 n! \rceil &\geq \log_2 n! && (1)\\
&\geq \sum_{i=1}^{n} \log_2 i && (2)\\
&\geq \sum_{i=1}^{n/2} \log_2 n/2 && (3)\\
&\geq n/2 \log_2 n/2 && (4)\\
&\text{is } \Omega(n \log n). && (5)
\end{aligned}$$

Mergesort, Heapsort are ***asymptotically optimal*** sorting algorithms. Quicksort is asymptotically optimal for ***most*** inputs.

The Broad View of Sorting Algorithms

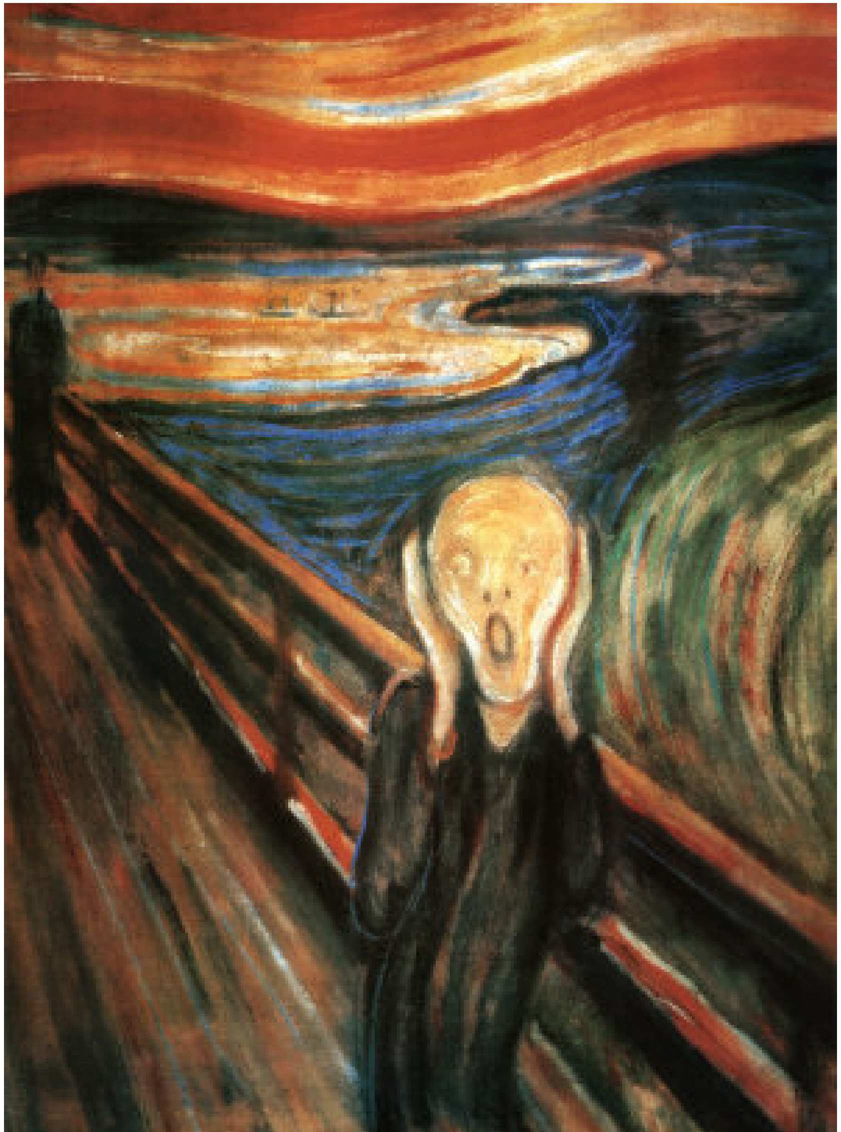
Bubblesort (terrible):



Quicksort (lovely):

Who wrote down bubblesort at the beginning of lecture?

The Broad View of Sorting Algorithms



The Broad View of Sorting Algorithms

$$\boxed{\begin{array}{c}\text{Bubblesort is terrible}\\ \\ \text{Quicksort is good}\end{array}}$$

More precisely, since Bubblesort is $O(n^2)$ and Quicksort is $O(n \log n)$ (for most inputs), Quicksort will ***massively*** outperform Bubblesort for larger input sizes.

Remember, for $n = 10^6$, n^2 is 10^{12}, but $n \log n$ is only about 2.10^7.

Note also: Bubblesort is poor in practice even compared with other $O(n^2)$ algorithms like insertion sort.

More Detailed Summary of Main Sorting Algorithms

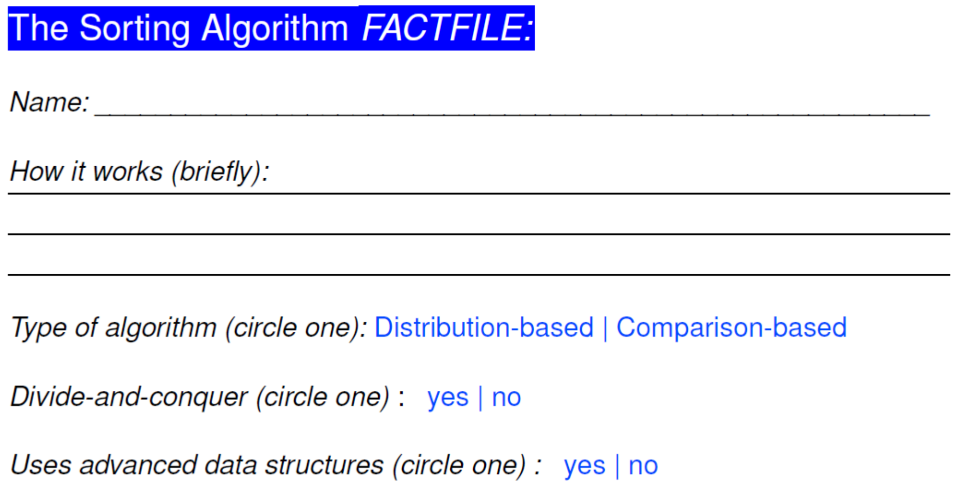
Time complexities of most important sorting algorithms (worst case, except for Quicksort):

Bucket sort	
Counting sort	$O(n)$
Radix sort	
Mergesort	
Heapsort	$O(n \log n)$
Quicksort	
Tree sort	
Insertion sort	
Selection sort	$O(n^2)$
Bubble sort	

The first three can only be used on ***special cases*** though, as we shall see.

How to Learn Sorting (Actively)

Try to fill out your own Sorting Algorithm *Factfiles*.



The Sorting Algorithm *FACTFILE:*

Name: ____

How it works (briefly): ____

____

____

Type of algorithm (circle one): Distribution-based | Comparison-based

Divide-and-conquer (circle one) : yes | no

Uses advanced data structures (circle one) : yes | no

Use this to direct your learning / investigation...

How to Learn Sorting (Actively)

... Then visit websites e.g.:
`http://www.sorting-algorithms.com/`

and read your text book to fill out things you didn’t know.

You could also:

- Try to fill out blank Factfiles again from memory
- Play “Top Trumps” with your Factfiles
- Identify which Factfile is missing (have a friend remove one) and recall what you know about it

A Basic Sorting Algorithm: Insertion Sort

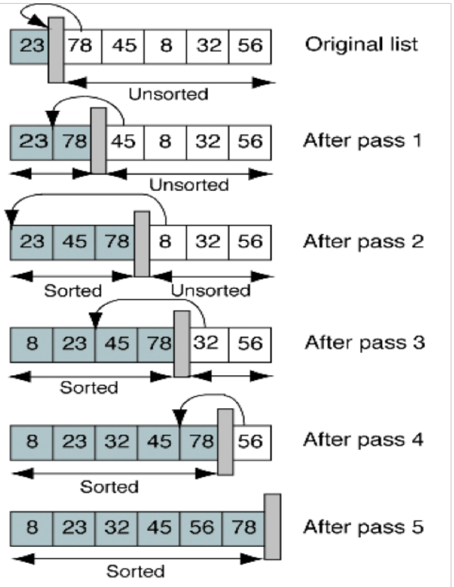
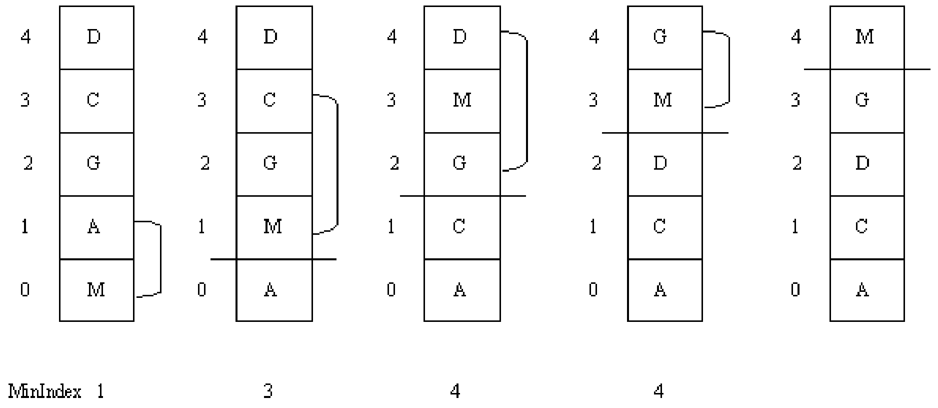


FIGURE 12-8 Insertion Sort Example

Why does insertion sort have time complexity in $O(n^2)$?
What is its space complexity? How much *additional* space does it need?

Insertion sort is in $O(n)$ for already sorted input. Why ?

A Basic Sorting Algorithm: Selection Sort



Selection sort uses the same number of comparisons, independently of whether the input is sorted, reverse-sorted, or unsorted. Why?

Selection sort is the basis of Heap Sort.

Question: What are the other important properties of Selection Sort and Insertion Sort? (See next slides for explanation of properties)

Some Properties of Sorting Algorithms

(other than time complexity)

Space Complexity

Some sorting algorithms require that the data are copied to a new list during the sort. This gives a space complexity of $O(n)$.

Some other sorting algorithms require only a constant amount of additional space, often just a single variable. These sorting algorithms are said to be ***in-place***. They require $O(1)$ extra space.

Stability

If items with equal keys are guaranteed to remain in the same order (not position!) they were in the input, then the sort is ***stable***.

(You met stability in the Lab this week on Lexicographic Sorting.)

Some Properties of Sorting Algorithms

General Purpose

If an algorithm can sort any list of items belonging to a total ordering, without restrictions, then it is **general purpose**.

If, on the other hand, an algorithm requires the items to be restricted to a finite number of keys, then it is not general purpose.

Comparison-based sorts are general purpose.
Distribution sorting algorithms (e.g. bucket sort) are not general purpose. They rely on knowing *a priori* something useful (limiting) about the universal set from which the elements to be sorted are drawn. E.g. they range from 200–1000.

Time Complexity: Different Cases

Counting the Sorting Operations

The ***basic operations*** of sorting algorithms are ***comparison***, and ***copy*** or ***swap***.

When the complexity of an algorithm is calculated, usually only one of the basic operations is counted. It can be important to state which one.

For some applications, comparison is expensive. For others, copying is expensive. (For example, if the items to be sorted are large database records, it can be expensive to copy them). ***Selection sort*** uses only $O(n)$ swaps, whereas Bubblesort uses $O(n^2)$.

If records are large, however, it can be better to apply a sorting algorithm to pointers to the records. After sorting, the complete records can be ordered in $O(n)$ copy operations.

Time Complexity: Different Cases

Unsorted/sorted Inputs

Often the complexity on **worst-case inputs** is given.

But some algorithms have poor worst-case performance, yet have good performance on ***most*** inputs, e.g. quicksort. For quicksort, the complexity usually quoted $O(n \log n)$ is for a typical (unsorted) input.

Quicksort has poor peformance $O(n^2)$ on sorted or reverse-sorted lists. Even <u>nearly</u> sorted lists cause inefficiency. [**Homework:** Why?]

Insertion sort is linear $O(n)$ on ‘nearly’ sorted or sorted lists. [**Homework:** Why?]

Time Complexity: Different Cases

Small input size

Some sorting algorithms have very low overhead, and so they tend to be very efficient on small input sizes, even if their asymptotic complexity is poor. ***Selection sort*** is like this. It is often used to sort lists shorter than 16 within a quicksort routine.

Many values the same (duplicates)

Some sorting algorithms perform poorly when many keys in the input have the same value. Which algorithms? Why? On the other hand, ***Bingo sort*** is a variant of selection sort that is efficient if many values are duplicates. (see wikipedia)

Performance Profiling

If the speed of sorting on a particular platform/input data is very important, then the best way to select an algorithm is by experimental performance profiling.

	128	256	512	1024	O1024	R1024	2048
Bubblesort	54	221	881	3621	1285	5627	14497
Insertion Sort	15	69	276	1137	6	2200	4536
Selection Sort	12	45	164	634	643	833	2497
Quicksort	12	27	55	112	1131	1200	230
Quicksort1	6	12	24	57	1115	1191	134
Mergesort	18	36	88	188	166	170	409
Mergesort1	6	22	48	112	94	93	254

Table 1: Time in microseconds of some sorting algorithms. O1024 means the input of size 1024 was in sorted order. R1024 means it was in reverse sorted order. Quicksort1 and Mergesort1 use Selection Sort to sort small subarrays (≤ 16 elements)

NB: These results are reproduced from IT Adamson (1996), page 155.

$O(n \log n)$ Sorting

- Merge sort
- Quicksort
- Heap sort

Merge Sort

Principle

MergeSort is a ***Divide-and-Conquer*** approach. The input array is divided recursively until there are n lists of length 1. All the sorting 'work' is done in the merge step.

Observation: given two sorted lists of total length n, it is possible to merge them into a single sorted list in $O(n)$ time.

Properties

- MergeSort has worst-case time complexity of $O(n \log n)$ in the number of comparisons.
- There is ***no*** implementation as an in-place sort.
- It is ***stable***. *Depends on implementation

Merge Sort: Merging Operation

```
           ↓
  List1:  12  16  19
           ↓
  List2:  11  12  12  35
--------------------------------
           ↓
 Output:  11
```

Compare the two pointed-at values. Copy the smaller one into the pointed-at place in the output. If the two values compared are equal, copy the one from List1. Move the pointer of the output, and the input list we copied from, one place to the right.

MergeSort: Merging Operation

$$\begin{array}{rccccccc}
 & & & \downarrow & & & & \\
\text{List1:} & 12 & 16 & 19 & & & & \\
 & & & & \downarrow & & & \\
\text{List2:} & 11 & 12 & 12 & 35 & & & \\
\hline
 & & & & & & & \downarrow \\
\text{Output:} & 11 & 12 & 12 & 12 & 16 & 19 & 35
\end{array}$$

Further Merge Sort Properties

Merge sort's running time is almost unaffected by the ordering of the input (see the table of running times on a previous slide). Why?

How can the merge operation be optimized slightly?

How is Merge sort implemented when input size n is not a power of 2?

What steps in Merge sort ensure it is stable?

Heap sort

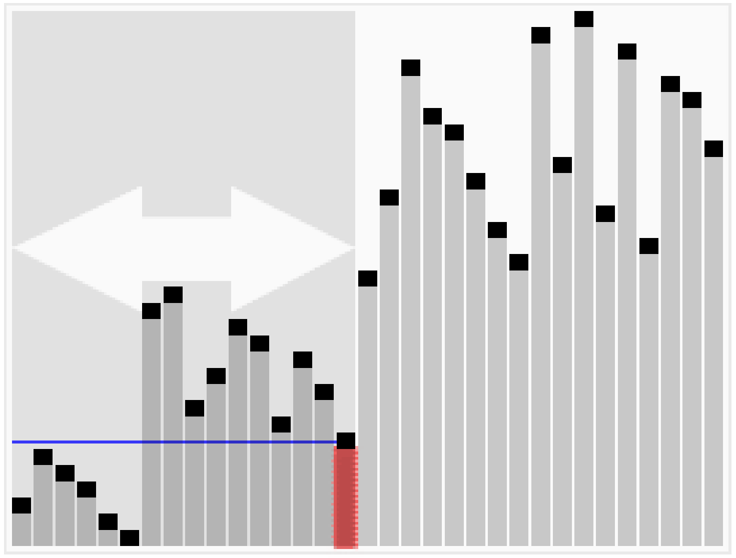
See: pp. 99–111 of the course textbook for full details.

Principle: Store the items one by one in a heap - ***a priority queue*** data structure. Then remove the items again in sorted order.

You will use (and possibly implement) a priority queue in the lab on knapsack problems.
You have already stored dictionary words in a tree to sort them. The principle is the same.

Properties:
Time complexity is $O(n \log n)$ on worst-case inputs
Heap sort can be implemented ***in-place*** by implementing the heap within a portion of the input array
Heap sort is ***not*** stable. Why not?

QuickSort



Invented by ‘Tony’ Hoare in 1960.

QuickSort

Principle: Quicksort is a divide-and-conquer algorithm. All the sorting work is done in the divide (partitioning) step. Merging back the sublists is trivial.

Partitioning Step: A ‘pivot’ element of the input array is chosen. After the partitioning,

- The pivot will be in the correct place in the array
- All items to pivot’s left are less than or equal to it
- All items to pivot’s right are greater than or equal to it

After the partitioning step, partitioning is applied recursively to the left and right ‘half’ of the array.

QuickSort

```
qs(int *a, int l, int r)
{
  int v, i, j, t;
  if(r>l)
    {
      v=a[r]; i=l-1; j=r; // set v (pivot) to rightmost element
      for(;;)
       {
         while(a[++i]<v);   // move left pointer
         while(a[--j]>v);   // move right pointer
         if(i>=j)break;   // stop if pointers meet or cross
         t=a[i]; a[i]=a[j]; a[j]=t; // swap elements
       }
      t=a[i]; a[i]=a[r]; a[r]=t; // swap elements
      qs(a, l, i-1);  // recursive call on left half
      qs(a, i+1, r);  // recursive call on right half
}
```

from R. Sedgewick “Algorithms in C”. Comments added.

QuickSort - Random Input

The above code has been modified to print out the sub-array it is sorting and the pivot value. Every swap operation also causes 'swap' to be printed. To fully understand quicksort, try this yourself!

```
 0  7  1  8  2  9  3 10  4 11
piv=11
swap
 0  7  1  8  2  9  3 10  4 11
piv=4
swap swap swap
 0  3  1  2  4  9  7 10  8 __
piv=2
swap swap
 0  1  2  3 __ __ __ __ __ __
piv=1
swap
 0  1 __ __ __ __ __ __ __ __
```

```
piv=8
swap swap
__ __ __ __ __  7  8 10  9 __
piv=9
swap
__ __ __ __ __ __ __  9 10 __
 0  1  2  3  4  7  8  9 10 11
```

QuickSort - Sorted Input

```
 1   2   3   4   5   6   7   8   9  10
piv=10
 1   2   3   4   5   6   7   8   9  __
piv=9
 1   2   3   4   5   6   7   8  __  __
piv=8
 1   2   3   4   5   6   7  __  __  __
piv=7
 1   2   3   4   5   6  __  __  __  __
piv=6
 1   2   3   4   5  __  __  __  __  __
piv=5
 1   2   3   4  __  __  __  __  __  __
piv=4
 1   2   3  __  __  __  __  __  __  __
piv=3
 1   2  __  __  __  __  __  __  __  __
piv=2
```

```
1 ___ ___ ___ ___ ___ ___ ___ ___ ___
1  2   3   4   5   6   7   8   9  10
```

$O(n)$ sorting

$O(n)$ sorting is only possible on special inputs.

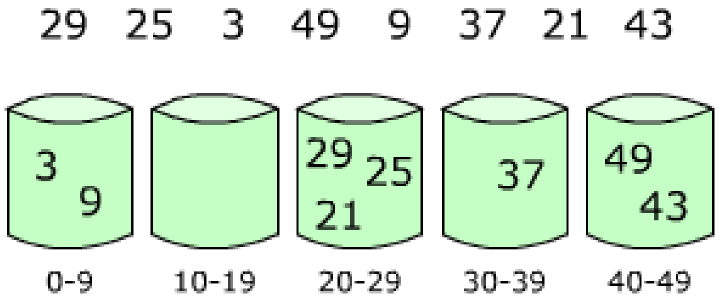
You met ***bucket sort*** (or counting sort) in the lab on complexity.

Bucket Sort

Principle: Given keys are in some finite range, j to $j + k$, the sort proceeds as follows:

- initialize a bucket array $b[0] = 0$ to $b[k - 1] = 0$
- increment $b[v - j]$ for each key v in the input
- write out $i + j$, $b[j]$ times, for each value i in 0 to k

The algorithm, as stated, is in $O(n + k)$. Why?



Buckets need not be of size 1. If larger buckets are used (as above), then an extra sorting procedure can be used to sort the contents of each bucket.

Bucket sort can be implemented to be ***stable***.

Radix Sort

(See pp. 242–243 of Goodrich & Tamassia)

Principle: To sort keys consisting of a sequence of symbols (e.g. words, n-digit numbers), we can apply a bucket sort to each symbol in turn, i.e. do multiple passes of the bucket sort.

Time Complexity: Radix sort is an $O(n)$ sorting algorithm, if the number of symbols in each sorting key is considered to be a constant. Why?

Advantages: This extends the range of applications for which a bucket sort is suitable. It would be impractical to sort (dictionary) words by a bucket sort because it would be difficult to index words into buckets. Using radix sort, we just need 26 buckets (1 per letter).

To sort a sequence of integers of arbitrary length, first we left-fill each integer with zeros so that all of them have the same length. E.g.

1, 100, 33 becomes 001, 100, 033

Then we use bucket sort to sort by the least significant digit.
10**0**, 00**1**, 03**3**

Next, we sort by the next more significant digit.
1**0**0, 0**0**1, 0**3**3

And finally by the most significant digit.
001, **0**33, **1**00.

The bucket sort must be stable. Why?

Conclusion

Quicksort is good.

Bubble sort is evil

Try to understand how the sorting algorithms work by looking at demos and animations - start with wikipedia

Know the space and time complexity of them

Appreciate other properties such as **stability**, ***comparison- or distribution-based***, **good-at-nearly sorted-lists** etc

Happy sorting !