Sorting

Sorting

Sorting is a very common operation

Suppose you have an array to sort into ascending order:

```
int ns[] = { 8, 2, 6, 5, 9, 1, 7 };
```

Of course you can sort ***this*** array by hand

But suppose it isn't constant, and needs to be sorted at run-time

Algorithms

There are lots of algorithms for sorting

We will look at a few of the well designed ones (that means ***excluding*** the terrible bubble sort algorithm!)

Swapping

To make code clearer, let's assume we have a swap function:

```
void swap(int array[], int i, int j) {
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}
```

(It can be made efficient by inlining)

Selection and insertion

The selection and insertion sort algorithms are the simplest well-designed algorithms

They use simple recursion (either do some work then sort `n-1` items, or sort `n-1` items then do some work)

Selection: select the smallest/largest then sort the rest

Insertion: sort all but one then insert it

Recursive selection sort

The recursive version makes the design clearer

It is slightly more convenient to select the largest and put it in the last place `array[n-1]`

```
// Sort the first n items of an array
void sort(int n, int array[]) {
    if (n <= 1) return;
    for (int i = 0; i < n-1; i++) {
        if (array[i] > array[n-1]) swap(array, i, n-1);
    }
    sort(n-1, array);
}
```

Iterative selection sort

Here's an iterative version, as a double loop:

```
void sort(int n, int array[n]) {
    for (int m = n-1; m > 0; m--) {
        for (int i = 0; i < m; i++) {
            if (array[i] > array[m]) swap(array, i, m);
        }
    }
}
```

This is, of course, O(n^2)

It is useful as a specification for other sorting algorithms

Recursive insertion sort

The recursive version makes the design clearer

```
// Sort the first n items of an array
void sort(int n, int array[]) {
    if (n <= 1) return;
    sort(n-1, array);
    int x = array[n-1];
    int i = n-1;
    while (i > 0 && array[i-1] > x) {
        array[i] = array[i-1];
        i--;
    }
    array[i] = x;
}
```

Iterative insertion sort

The iterative version is a double loop:

```
void sort(int n, int array[n]) {
    for (int m = 1; m < n; m++) {
        int x = array[m];
        int i = m;
        while (i > 0 && array[i-1] > x) {
            array[i] = array[i-1];
            i--;
        }
        array[i] = x;
    }
}
```

This is, of course, O(n^2)

Importance

Insertion sort is almost unbeatable for small arrays

And almost unbeatable for arrays which are almost in order already

So is often used in libraries as a last fine-grained pass to speed up more complex algorithms

Divide and conquer

A general recursive design technique is to split a problem into two half-sized problems

That leads to two divide-and-conquer algorithms, sort two smaller arrays, and either do some work first, or do some work afterwards:

Quicksort: divide the items into small and large, then sort the two divisions

Mergesort: sort the first half and second half, then merge the two results together

Pivot

Partitioning means dividing into small and large sections, i.e. `<=p` and `>p` for a ***pivot*** `p`

There is a chicken and egg problem: you can only find the perfect pivot after you have done the partitioning

Choosing the first or last item as the pivot is very poor on arrays that are almost in order already

The next simplest choice is the middle item

Partitioning

Here's a function that partitions an array between ***positions*** `lo` and `hi`, and returns the dividing index:

```
int partition(int array[], int lo, int hi) {
    int p = array[lo + (hi-lo)/2];
    while (true) {
        while (array[lo] <= p) lo++;
        while (array[hi-1] > p) hi--;
        if (lo == hi) return lo;
        swap(array, lo, hi-1);
    }
}
```

The version in Wikipedia when I wrote this slide was incorrect!

Recursive quicksort

Here's a recursive quicksort

```
void sort(int array[], int lo, int hi) {
    if (lo >= hi-1) return;
    int mid = partition(array, lo, hi);
    sort(array, lo, mid);
    sort(array, mid, hi);
}
```

Iterative quicksort

You can make quicksort iterative, but you need a stack of ranges which have yet to be sorted

The algorithm pulls a range off the stack, partitions, and pushes the two smaller ranges on the stack, if they have at least two elements

It is not worth showing you the code - ask google

Quicksort problems

Quicksort is one of the fastest known algorithms, very difficult to beat

It has best case and average case O(n*log(n)) performance

But it is O(n^2) in the worst case, because partitioning may split `n` items into unequal halves, e.g. `1` and `n-1`

When does speed ***really*** matter? In real time systems where you ***can't*** be late, so ironically quicksort is not used when time is critical!

And is often not used in modern language libraries

Mergesort problem

Mergesort needs a second array of the same size (well, actually it doesn't, but without it, it becomes too inefficient)

Merge

Here's a merge function:

```
void merge(int array[], int lo, int mid, int hi, int other[]) {
    int i = lo, j = mid;
    for (int k = lo; k < hi; k++) {
        if (i < mid && (j >= hi || array[i] <= array[j])) {
            other[k] = array[i++];
        } else {
            other[k] = array[j++];
        }
    }
}
```

`array[lo..mid]` and `array[mid..hi]` are already sorted and are merged into `other[lo..hi]`

It's O(n)

Recursive mergesort

Here's a recursive mergesort, which starts with the two arrays being copies of each other:

```
void sort(int array[], int lo, int hi, int other[]) {
    if (hi - lo <= 1) return;
    int mid = lo + (hi-lo)/2;
    sort(other, lo, mid, array);
    sort(other, mid, hi, array);
    merge(array, lo, mid, hi, other);
}
```

A lot of copying is avoided by alternating the arrays with the level of recursion

Iterative mergesort

Making mergesort iterative is easier than making quicksort iterative

That's because the divisions between sections are completely predictable

What you do is merge each pair of items from `array` into `other`, then merge each pair of runs of length 2 from `other` into `array`, then merge each pair of runs of length 4 from `array` into `other`, and so on

Importance of mergesort

Mergesort is ***guaranteed*** to be O(n*log(n)) under all circumstances

The extra memory required during the sorting is no longer regarded as a big problem

So, it is suitable for real time programming

And also for interactive software, because people prefer slightly slower but predictable-time algorithms

Some language libraries use mergesort for the long runs, and insertion sort for the short ones, or equivalent

Other algorithms

The outcome of these algorithms is that you should almost always use one of these four well-designed sorting algorithms

There is perhaps one exception, which is when you want an algorithm which is faster than O(n*log(n))

This is tough, because all algorithms ***based on comparisons*** are at least O(n*log(n))

Radix sort

Suppose you have a million numbers to sort, and they are all percentages `0..100`

You create an array of length `101` to count how many times each percentage appears, and run through the numbers once, incrementing the counts, then regenerate the list from the counts

That's an O(n) algorithm

It can be generalized to radix sort, which becomes O(n*log(n)) if the number of 'digits' or 'characters' in the numbers becomes O(log(n))

Strategy 1

What should you ***actually*** do if you want to do some sorting? (Or almost anything else)

You know an algorithm for sorting, or you can work one out, so you write a sorting function

This is a reasonable strategy for a lot of problems

But for sorting, it is time-consuming and error-prone, and there is a better way

Strategy 2

You look up sorting in Google and copy some code, e.g. you copy code from a wikipedia entry on sorting

You are much less likely to make logic errors this way, and this is often a reasonable strategy

But it can be error-prone to translate the code into the right form for your purposes, especially if it is written in another language or in pseudo-code

For sorting, there is a better way

Strategy 3

You find a library or module which someone has written, which is sufficiently generic to be adapted to your purposes without changing it in any way, and you download it and include it in your project

This is a good strategy, provided you check that the author of the library is reliable, not just some show-off

But for sorting, we can do a bit better

Strategy 4

You find a ***standard*** library module which can be used unchanged for your purposes

This is the best, when it works, because it is guaranteed to be readily available, in an identical form, on every platform

For sorting, there is a standard function in the `stdlib` library module (specified in the C11 standard)

The `qsort` function

The function is called `qsort`

This is a poor name because `qsort` is an unnecessary abbreviation for `quicksort`, left over from the days of 6-character variable names

It is a poor name anyway, because it is supposed to use "the best general purpose sorting algorithm", which could change, so it should have been called `sort`

The documentation

If you look up the documentation (by typing `C qsort` into Google) you find this declaration:

```
void qsort(
    void *base,
    size_t nitems,
    size_t size,
    int (*compar)(const void *, const void*)
);
```

It is poorly written - I've neatened the layout

It is not easy to understand - it uses features we haven't fully covered

The first argument

The first argument is:

```
void *base,
```

This is the array, passed by pointer, i.e. passed as a pointer to the first element

The type of the pointer is `void *`, which means the function is generic and will accept any type of array

The next two arguments

The next two arguments are:

```
size_t nitems,
size_t size,
```

The first is the length of the array, and the second is the size of each item in the array in bytes

The `size_t` type means "the most efficient type for representing sizes on your computer"

There shouldn't be any problem in passing `int`s

The last argument

The last argument is:

```
int (*compar)(const void *, const void*)
```

This says `compar` instead of `compare` because C variable names used to be limited to 6 characters

The type `void *` is written inconsistently, and the argument names have been left out

This uses `const` and function pointers, and means that we have to write a compare function and pass a pointer to it to `qsort`

Writing a compare function

Here's a suitable compare function:

```
int compare(const void *p, const void *q) {
    const int *pi = p, *qi = q;
    return *pi - *qi;
}
```

It has to have exactly the signature specified

It compares two `int`s, which are passed by pointer (because `qsort` didn't know their size when it was compiled)

`const` arguments

The arguments are declared as `const` which means the function has to promise not to change the integers pointed to

The statement `const int *pi = p` just copies the void pointer into an int pointer (probably without generating any code)

The variable `pi` needs to be declared as `const` in order to continue to promise not to change the ints

The return value

The function returns `*pi - *qi` which just subtracts the two ints

If you ***read the documentation*** you discover that the function only has to return negative, zero or positive (like `strcmp` for example) and not `-1 0 1`

It is normal to just subtract

The call

The call to `qsort` is like this:

sort.c

```
int ns[] = { 8, 2, 6, 5, 9, 1, 7 };
int itemSize = sizeof(int);
int length = sizeof(ns) / itemSize;
qsort(ns, length, itemSize, compare);
```

As with arrays, if you pass a function as an argument, it is automatically converted into a pointer, e.g. `compare` is treated as if you had written `&compare`

Function pointers

You can use function pointers in your own programs in a much more readable way than most tutorial writers seem to think

Imagine that you have a calculator, and an enumerated type `{Plus, Minus, Times, Over}` represent the four basic operators `+`, `-`, `*`, `/`

You can use an operator constant to index an array of four function pointers

How readable can you make this?

The function type

All the functions must have the same type, which you can describe using a typedef:

```
typedef int op(int x, int y);
```

This looks just the same as a normal function declaration, but it defines a type `op`, instead of declaring a function

The type describes functions, not function pointers

The array

Then you can define an array of function pointers:

```
op *ops[] = { add, sub, mul, div };          calc.c

int main() {
    op *f = ops[Times];
    int n = f(6, 7);
    printf("Answer = %d\n", n);
}
```

When you make a call on a function pointer, C automatically dereferences it

Overall, the result is quite readable