

Sorting

Sorting

Sorting is a very common operation

Suppose you have an array to sort into ascending order:

```
char *xs[] = { "h", "b", "f", "e", "i", "a", "g" };
```

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 simplify code, let's assume we have a swap function:

```
typedef ... item;  
  
void swap(int i, int j, item xs[]) {  
    item temp = xs[i];  
    xs[i] = xs[j];  
    xs[j] = temp;  
}
```

We'll use `item` for the type of things to be sorted

Array `xs[]` must have length greater than `i` or `j`

The function can be made efficient (free) by inlining

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

They use simple recursion: either deal with one item, leaving $n-1$ items left to sort, or sort $n-1$ items then deal with the other item

Selection sort: select one item then sort the rest

Insertion sort: sort all but one then insert the last item

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 `xs[n-1]`

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

Iterative selection sort

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

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

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, item xs[n]) {
    if (n <= 1) return;
    sort(n-1, xs);
    item x = xs[n-1];
    int i = n-1;
    while (i > 0 && xs[i-1] > x) {
        xs[i] = xs[i-1];
        i--;
    }
    xs[i] = x;
}
```

Iterative insertion sort

The iterative version is a double loop:

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

Partitioning means dividing into small and large sections, i.e. $\leq p$ and $\geq p$ 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 lo, int hi, item xs[hi]) {
    swap(lo + (hi-lo)/2, lo, xs);
    item p = xs[lo];
    while (true) {
        while (xs[lo] < p) lo++;
        while (xs[hi-1] > p) hi--;
        if (hi - lo <= 1) return hi;
        swap(lo, hi-1, xs);
        lo++;
        hi--;
    }
}
```

The partitioning function (Hoare's algorithm) is very delicate, so needs to be adapted with testing in place

Although it uses the position convention (measuring) and `while` instead of `do..while` and the middle item instead of the first, it is the same as the original

All the details (swapping the mid item with the first, using `<` and not `<=`, returning `hi` and not `lo`) matter, to avoid an infinite loop

Recursive quicksort

Here's a recursive quicksort

```
void sort(int lo, int hi, item xs[hi]) {  
    if (hi - lo <= 1) return;  
    int split = partition(lo, hi, xs);  
    sort(lo, split, xs);  
    sort(split, hi, xs);  
}
```

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 $O(n \cdot \log(n))$ 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 lo, int mid, int hi, item xs[hi], item other[hi]) {
    int i = lo, j = mid;
    for (int k = lo; k < hi; k++) {
        if (i < mid && (j >= hi || xs[i] <= xs[j])) {
            other[k] = xs[i++];
        } else {
            other[k] = xs[j++];
        }
    }
}
```

$xs[lo..mid]$ and $xs[mid..hi]$ 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 lo, int hi, item xs[hi], item other[hi]) {  
    if (hi - lo <= 1) return;  
    int mid = lo + (hi-lo)/2;  
    sort(lo, mid, other, xs);  
    sort(mid, hi, other, xs);  
    merge(lo, mid, hi, xs, 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 `xs` into `other`, then merge each pair of runs of length 2 from `other` into `xs`, then merge each pair of runs of length 4 from `xs` into `other`, and so on

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

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

Choosing

There are other algorithms, but it is usually enough to choose between quicksort and mergesort

They each need to be sped up by using insertion sort on small subarrays

The best way to choose is to differentiate between 'background' and 'foreground' sorting

Back- and fore-ground

Background sorting is when it doesn't matter how long the sorting itself takes, but overall efficiency matters, e.g. in a server

Foreground sorting is where sorting time really matters, in real time software with hard deadlines, or interactive software that needs to be predictably responsive

Importance of mergesort

Quicksort is best for background sorting, because of the average speed

Mergesort is best for foreground sorting, because of the predictability

So a general purpose library should offer both (but they rarely do)

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 $O(n \cdot \log(n))$

This is tough, because no algorithm ***based on comparisons*** can be better than $O(n \cdot \log(n))$

Suppose you have a million numbers to sort, and they are all percentages $0 \dots 100$

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 $O(n \cdot \log(n))$ if the number of 'digits' or 'characters' in the items becomes $O(\log(n))$

What should you *actually* do if you want to do some sorting? (The same applies to many problems)

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

You look up sorting in Google and copy some code, e.g. 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

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

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

This is often 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 C 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`

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 `ints`

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 for `ints`:

```
int compare(const void *p, const void *q) {
    const int *pi = p, *qi = q;
    int x = *pi, y = *qi;
    if (x < y) return -1;
    else if (x == y) return 0;
    else return 1;
}
```

It has to have exactly the signature specified

It compares two `ints`, 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 return value is calculated by:

```
if (x < y) return -1;  
else if (x == y) return 0;  
else return 1;
```

It used to be calculated as:

```
return x - y;
```

But that doesn't work across the whole range of int, and sometimes people recommend:

```
return (x > y) - (x < y);
```

Assembly language

40

```
if (x < y) return -1;           return (x > y) - (x < y);
else if (x == y) return 0;
else return 1;
```

```
movl    -4(%rbp), %eax
cmpl    -8(%rbp), %eax
jge     .L2
movl    $-1, %eax
jmp     .L3
.L2:
movl    -4(%rbp), %eax
cmpl    -8(%rbp), %eax
jne     .L4
movl    $0, %eax
jmp     .L3
.L4:
movl    $1, %eax
.L3:
```

```
movl    -4(%rbp), %eax
cmpl    -8(%rbp), %eax
setl    %al
movzbl  %al, %edx
movl    -4(%rbp), %eax
cmpl    -8(%rbp), %eax
setg    %al
movzbl  %al, %eax
subl    %eax, %edx
movl    %edx, %eax
```

The trickery on the right is better, with no jumps.

Optimisation

41

```
if (x < y) return -1;           return (x > y) - (x < y);
else if (x == y) return 0;
else return 1;
```

```
xorl    %eax, %eax
cmpl    %esi, %edi
movl    $-1, %edx
setne   %al
cmovl   %edx, %eax
```

```
xorl    %eax, %eax
cmpl    %esi, %edi
setg    %dl
setl    %al
movzbl  %dl, %edx
subl    %edx, %eax
```

With `-O2`, the code on the left gives the optimizer more scope, so all the trickery does is to reduce readability

In C++, there is an operator `return x <==> y;`

A call to `qsort` is like this:

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

sort.c

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 program, and an enumerated type `{Plus, Minus, Times, Over}` to 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