15. Binary search and sorting arrays

Herman Kamper

The last lecture ended with linear search: walk through an array from the front, compare every element to the key, stop when you find it. It works on any array in any order, and it costs you one comparison per element. In this lecture we do two things with that.

First, a better search algorithm. If the array happens to be sorted, you can find the key without looking at every element, and in fact without looking at very many at all. Second, sorting itself: given an array in no particular order, how do you put it in order? Those two are connected, because the fast search only works on a sorted array, so the moment you want it you also want a way to produce one.

One thing is different about this lecture. There is much less code in it than usual. That is deliberate. You now know enough C syntax that the translation step, going from a solution to working code, is something you can practise on your own. The harder part, and the part that is worth spending a lecture on, is working out what the algorithm should be in the first place. So most of what follows is figuring out the algorithm, and the code comes at the end and is shorter than you would guess.

Searching a sorted array

Start with the problem. Here is a small array of marks, and I want to know where the value 9 sits in it:

```
int marks[6] = {99, 75, 9, 65, 70, 55};
```

The answer is index 2. Linear search finds that by checking index 0, then index 1, then index 2. Fine for six elements. Now imagine the array has millions on entries (something that does happen in real life). In the worst case, the element I am searching for might be at the end – I don't want to have to check everything! This is where more advanced search algorithms comes in.

Keeping with our example, suppose now the array is sorted:

```
int marks[6] = {9, 55, 65, 70, 75, 99};
```

and I am looking for 65. Do I still have to check every element? No, and here is the trick. Jump straight to the middle of the array and look at what is there. If the middle element is bigger than the key, then the key can only be in the bottom half, because everything above the middle is

bigger still. If the middle element is smaller, the key can only be in the top half. Either way, one comparison has thrown away half of the array.

And then you do not stop there. Take the half you have left, jump to its middle, and throw away half of that. Repeat. Each comparison halves what remains, and it does not matter whether you started with six elements or a million. That is the intuition behind binary search.

Binary search step by step

Before the walkthrough, one piece of terminology, because the whole algorithm depends on keeping these two straight. Given an array `c`, the number you write inside the square brackets is the *index*, and the number sitting in the box is the *value*. So `c[6]` is a particular box, and inside that box is the value 13. Binary search juggles both at once, and it is very easy to confuse an index with a value while you are doing it.

The algorithm keeps three indices. `low` is the bottom of the region still worth searching, `high` is the top of it, and `middle` is the element halfway between them. At the start, `low` is 0 and `high` is the last index, because the whole array is still in play. Then `middle` is worked out as `(low + high) / 2`, and because both are integers this is integer division: the fractional part is thrown away.

Here is a sorted 16-element array of values, we are searching for the key 11, and the four panels below are the four comparisons the algorithm makes. Greyed-out boxes are the ones that have already been ruled out.

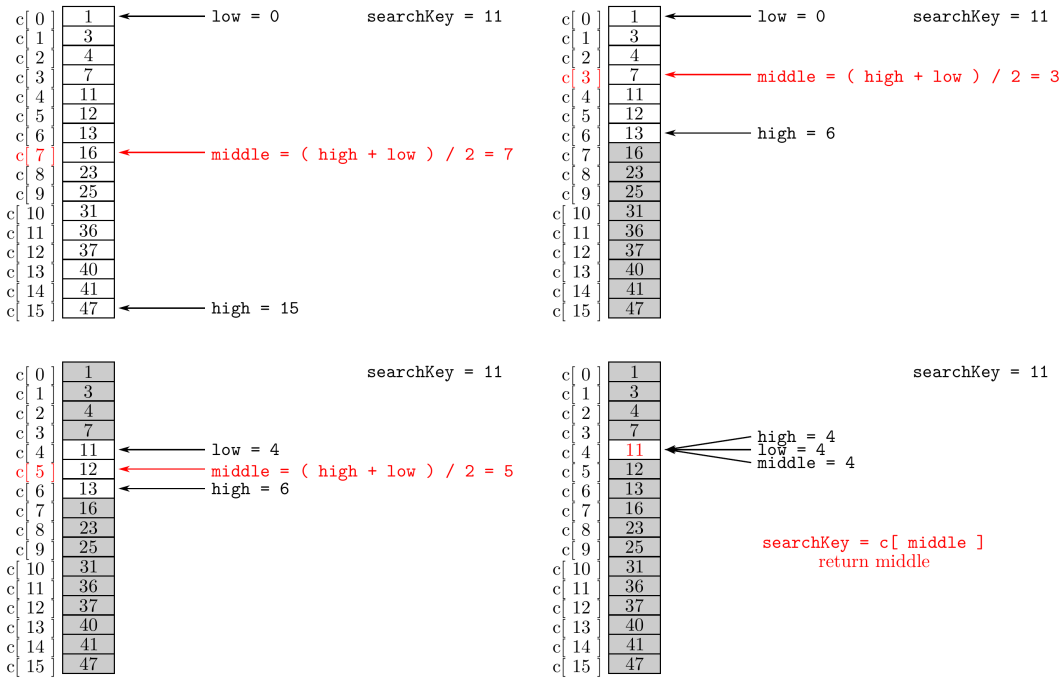

Figure 1: Binary search for the key 11, reading left to right and then top to bottom. Each panel shows where `low`, `high` and `middle` sit at that comparison, and the greyed boxes are the part of the array that has already been eliminated.

Take them one at a time. At the start `low` is 0 and `high` is 15, so `middle` is `(0 + 15) / 2`, which would be 7.5 in real arithmetic but is 7 in integer division. The value sitting at index 7 is 16, and the key is 11, so the key is smaller. That rules out everything from index 7 upwards in a single comparison, and we say so by moving `high` down to `middle - 1`, which is 6.

Second comparison. The region left is indices 0 to 6, so `middle` is `(0 + 6) / 2`, which is 3, and the value there is 7. This time the middle is smaller than the key, so the key must be above it, and it is `low` that moves, up to `middle + 1`, which is 4.

Third comparison. The region is indices 4 to 6, `middle` is 5, and the value there is 12. That is bigger than the key, so `high` comes down to `middle - 1`, which is 4. Now `low` and `high` are both 4: exactly one element is left, and the middle of it is itself.

Fourth comparison. `low`, `high` and `middle` have all converged on index 4, the value there is 11, and that is the key. The function returns 4 and we are done.

Four comparisons on a sixteen-element array. Notice what the algorithm never did: it never looked at index 0, or 1, or 2, or any of indices 8 to 15. It only ever looked at the middle of whatever was left, and it kept moving the goalposts inwards until they met.

How many steps does it take?

Binary search looks fast, but let us be precise about it, because "looks fast" is not an argument.

Let's compare linear search (last lecture) to binary search (here). In the sixteen-element array above, linear search finds the value 1 in a single step, because 1 happens to sit at index 0. That makes linear search look wonderful. But it is luck. Ask for the value 47 instead, which sits at the very end, and linear search needs all sixteen comparisons to reach it. The honest way to compare two algorithms is to ask about the worst case, not the best case. Linear search on n elements costs n comparisons in the worst case, and that is the number to beat.

Binary search is the interesting one. Each comparison looks at the middle element, rules that element out, and then throws away everything on one side of it. So a region of n elements becomes a region of at most half of $n - 1$, rounded up, and the search only stops once there is nothing left at all. The question is therefore how many times you can halve n before you run out.

Count it down from 16 and the answer falls out. Sixteen elements become at most 8, then at most 4, then 2, then 1, then 0. That is five halvings, each one costing a comparison, so five comparisons:

$$\begin{aligned} \text{steps} &= \text{the smallest } k \text{ for which } 2^k > \text{elements} \\ &= \lfloor \log_2 \text{elements} \rfloor + 1 \end{aligned}$$

The first line is the rule in plain terms: keep doubling 2 until you overtake the number of elements, and count the doublings. The second line says the same thing with a logarithm, where the half-brackets mean round down. For 16 elements, 2^4 is 16, which does not overtake it, and 2^5 is 32, which does, so the answer is five.

Note that this is one more than you might have guessed. Our walkthrough found the key 11 in four comparisons, but that was not the worst case: search the same array for 47 and it takes five. The culprit is the middle element. Splitting sixteen boxes does not give you two halves of eight – it gives a middle, a lower part of seven and an upper part of eight – and the worst case follows the bigger of the two.

That detail aside, the shape of the thing is what matters, and the shape is a logarithm. Compare the two algorithms in their worst cases:

Elements	Linear search	Binary search
16	16	5
32	32	6
1,000	1,000	10
1,000 000	1,000,000	20

Linear search grows in a straight line and binary search barely grows at all. Going from a thousand elements to a million multiplies linear search's work by a thousand; it adds ten comparisons to binary search's.

Binary search in code

Now the code, briefly. The function takes the array, the key, and the `low` and `high` indices to search between:

```
int binarySearch(const int b[], int searchKey, int low, int high)
{
    int middle;

    while (low <= high)
    {
        middle = (low + high) / 2;

        if (searchKey == b[middle])
        {
            return middle;
        }
        else if (searchKey < b[middle])
        {
            high = middle - 1;
        }
        else
        {
            low = middle + 1;
        }
    }
```

```
    return -1;
}
```

Read it against the walkthrough and every line should be recognisable. The `const` says the search will not modify the array, exactly as with linear search. The array parameter is written `int b[]` with no size, because as we know an array parameter never carries its size, and here we do not need one: `low` and `high` between them say which part of the array to look at. To search the whole thing you would call it as `binarySearch(c, 11, 0, 15)`.

The loop condition is `low <= high`, which reads as "there is still at least one element left to check". Each pass computes the middle and then does one of three things. If the middle element is the key, we are finished and return the index. If the key is smaller, the answer can only be below, so `high` drops to `middle - 1`. If the key is bigger, the answer can only be above, so `low` rises to `middle + 1`.

The two `- 1` and `+ 1` are doing real work. We have just compared against `middle` and it was not the key, so `middle` itself is now excluded. Leave it in, by writing `high = middle` instead, and on a search that is going to fail the region will stop shrinking and the loop will spin forever.

And if the key is not there at all? Then `low` and `high` close in on each other until `low` finally overtakes `high`, the `while` condition fails, and the function falls through to `return -1` – the same not-found convention we used for linear search.

Sorting

Binary search has one catch: the array has to be sorted. So what do you do when it is not?

You sort it. Which raises the obvious question of how, and that is the second half of this lecture.

Sorting is something we do all the time without thinking about it: numbers from lowest to highest, names into alphabetical order, results by score. We will use numeric examples here, but nothing about the ideas is specific to numbers. The algorithm we are going to look at is called bubble sort. It is not efficient, and it is very rarely the right choice in practice. It is here because it is the simplest sorting algorithm there is, which makes it the best one to think through from scratch, and the habits of thought you pick up doing so transfer to the ones you would actually use.

Working out bubble sort

Before reading on, it is worth trying this yourself. Here is an array:

```
int c[7] = {13, 4, 1, 12, 3, 7, 11};
```

How would you get it into order, from low to high? Not "how would you write the C", but what would the procedure be?

One idea that comes up immediately is to find the smallest element in the array, put it in position 0, then find the smallest of what is left, put it in position 1, and so on. That is a real sorting

algorithm and it works, and it is worth writing yourself. But it needs a function that finds a minimum, and it needs to keep track of which part of the array is already finished.

Here is a smaller idea, and the one bubble sort is built on. Forget the whole array. Just look at the first two elements. `c[0]` is 13 and `c[1]` is 4, and they are in the wrong order relative to each other, so swap them.

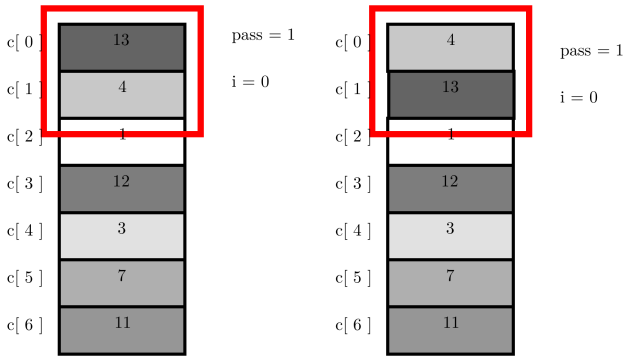

Figure 2: Comparing the first two elements and swapping them, because 13 is bigger than 4. On the right is the array afterwards.

The array is now `4, 13, 1, 12, 3, 7, 11`. It is not sorted. But it is a little more sorted than it was, and that turns out to be enough to build on. Move down one and do exactly the same thing to the next pair, `c[1]` and `c[2]`. 13 is bigger than 1, so swap. Then `c[2]` and `c[3]`, then `c[3]` and `c[4]`, all the way to the end of the array, comparing each adjacent pair and swapping when they are the wrong way round.

Notice that we never look at the array as a whole. Each step involves exactly two boxes. That matters, because it means the procedure is something C can actually do, because C does not get to see the shape of your data, it only ever looks at particular boxes.

One trip from the top of the array to the bottom is called a pass. Here is where our array stands at the end of the first pass, and at the end of the second and third:

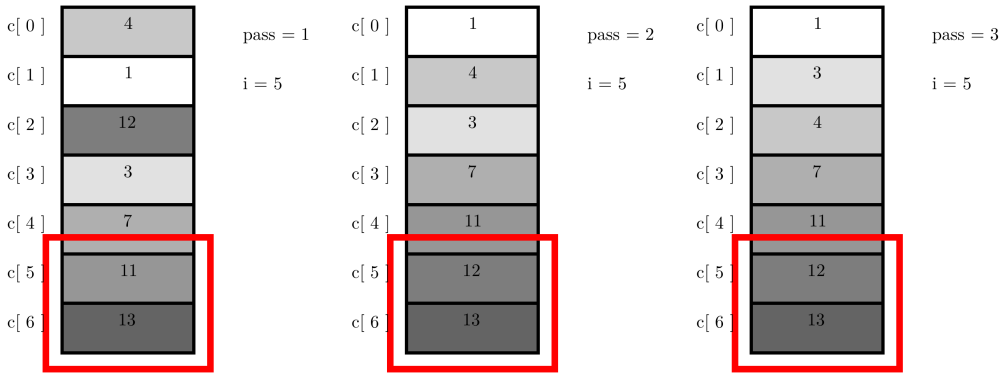

Figure 3: The array at the end of pass 1, pass 2 and pass 3. Each pass compares every adjacent pair from top to bottom, swapping wherever they are out of order. After the third pass the array is sorted.

Follow what happens to the 13. In the first pass, the moment we reach it we find that everything below it is smaller, so it gets swapped down, and down, and down, until it ends up at the bottom. The largest element always sinks all the way to the end on the very first pass. On the next pass, the second largest does the same, and so on. That behaviour is where the name comes from, with the small values bubbling up to the top.

That also tells us how many passes we need. Each pass is guaranteed to put at least one more element in its final position, so `size - 1` passes are always enough. Our seven-element array happened to come out sorted after three, but that was luck, and the algorithm has no way of knowing it. Unless you add a check for it, bubble sort simply does its full quota of passes.

At this point somebody usually asks the obvious question: if we are going to walk the array six times over, what have we gained on just walking it once? The answer is that this is not a search, it is a sort, and a sort is a genuinely harder job. But the objection is a fair one – bubble sort really is slow, and there are much better sorting algorithms. What is worth taking away is the thinking and principles here.

Bubble sort in code

Here is the frame of the program, with the sorting function still empty:

```
#include <stdio.h>

#define SIZE 10

void bubbleSort(int b[], int size);

int main()
{
    int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
    int j;

    printf("Data items in original order\n");
    for (j = 0; j < SIZE; j++)
    {
        printf("%d ", a[j]);
    }

    bubbleSort(a, SIZE);

    printf("\nData items after sort\n");
    for (j = 0; j < SIZE; j++)
    {
        printf("%d ", a[j]);
    }
```

```
    return 0;
}

void bubbleSort(int b[], int size)
{
    int i;
    int pass;
    int tmp;

    for (pass = 1; pass < size; pass++)
    {
        for (i = 0; i < size - 1; i++)
        {
            // compare b[i] and b[i + 1] here
        }
    }
}
```

Note that `bubbleSort` returns nothing and yet the program expects `a` to come back sorted. That is not an oversight: an array parameter is an address, so the function is working on the caller's array directly, which is exactly what we want here.

The two loops of the function itself follow straight from the walkthrough. The outer loop counts passes, and the inner loop walks the pairs.

The inner loop stops at `size - 1` rather than `size`, and the reason is worth being clear about. The body compares `b[i]` against `b[i + 1]`, so if `i` were allowed to reach the last index, `b[i + 1]` would be reading one element past the end of the array – which, as we saw last lecture, C will happily let you do and will not warn you about.

Now the body inside the two `for` loops of the `bubbleSort` function. The comparison is the easy half:

```
if (b[i] > b[i + 1])
{
    // swap
}
```

The swap is the half that catches people out. The tempting thing to write is:

```
b[i + 1] = b[i];
```

and if you are looking at 13 and 4, that does put 13 where the 4 was, which is half of what you wanted. But the 4 is now gone. You have overwritten the only copy of it, and there is nothing left to put back into `b[i]`.

So you need somewhere to keep it. Before overwriting anything, copy the value you are about to lose into a temporary variable, and copy it back out afterwards:

```
if (b[i] > b[i + 1])
{
    tmp = b[i + 1];
    b[i + 1] = b[i];
    b[i] = tmp;
}
```

Three lines: park the 4 in `tmp`, move the 13 down over it, then take the 4 out of `tmp` and put it where the 13 used to be. With that filled in, the program prints:

```
Data items in original order
2 6 4 8 10 12 89 68 45 37
Data items after sort
2 4 6 8 10 12 37 45 68 89
```

And that is the whole algorithm: a comparison and a three-line swap, wrapped in two loops. Once you have worked out what the algorithm should be, turning it into C is the small part of the job. Which is the point of doing it in this order.

Sorting in practice

Two honest remarks before we move on.

The first is that you will almost certainly never write a sorting function in real life. Nor a search function, for that matter. Every language ships with them, they are written by people who have thought very hard about it, and they will beat anything you write in an afternoon. What you are practising here is not the deliverable. It is the thinking: look at a problem, work out a procedure, convince yourself the procedure is correct, and only then write the code.

The second is that bubble sort is genuinely one of the slower options. If you look up animations of sorting algorithms running side by side – there are lovely ones online – you can watch bubble sort make its slow full sweep of the array over and over while insertion sort, merge sort and quicksort are long finished.

A last example: mean, median and mode

To finish, a problem that uses everything in this lecture at once.

> Ninety-nine responses were collected in a survey, and each response is a number between 1 and 9. Place the responses in an array and summarise the results by giving the mean, median and mode of the survey data.

This is worth solving before reading further. Work out the strategy for each of the three, then write it. What follows is the strategy, not the finished program.

The mean is the average: add all the values up and divide by how many there are. A running total in a loop over the array, and then one division by 99 at the end. Nothing here you have not done before; just be careful to divide as a `double` of `float` rather than as an integer, or you will lose the fractional part.

The median is the middle value. That means two steps: sort the array from low to high, which is exactly what `bubbleSort` is for, and then look at whatever sits in the middle. The second step sounds trivial until you actually try to say which index the middle is. Is it 49? 50? 48?

Here is how to answer a question like that when you cannot do it in your head. Shrink the problem until you can see it. Instead of 99 boxes, draw 9:

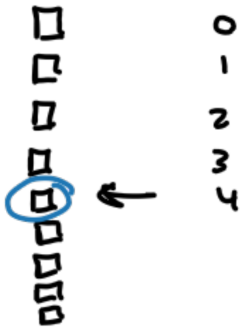

Figure 4: Nine boxes instead of 99. The circled box has four boxes above it and four below, so it is the middle one, and counting down from the top its index is 4.

Nine boxes, indices 0 to 8. The middle box has four above it and four below, so it is the one at index 4. And 4 is `9 / 2` in integer division. So for 99 elements the median sits at index `99 / 2`, which is 49, with 49 elements on either side of it. In code that is just `answer[size / 2]`.

The mode is the value that occurs most often. Of the ratings 1 to 9, which one comes up the most? The strategy people usually reach for first is to have a counter for each possible rating: a variable counting the ones, another counting the twos, and so on up to nine. Loop through the responses, and each time you see a 5 you add one to the five-counter. At the end, whichever counter is largest tells you the mode.

That works, and nine separate variables is exactly the situation arrays exist to fix. So make it an array of counts instead, indexed by the rating itself. Then "add one to the counter for rating 5" becomes `freq[5]++`, and since the rating is sitting right there in the response array, that is `freq[answer[j]]++`. Here is the function:

```c
void mode(int freq[], const int answer[], int size)
{
    int j;
    int rating;
    int largest = 0;
    int modeValue = 0;

    for (j = 0; j < size; j++)
    {
        freq[answer[j]]++;
    }

    for (rating = 1; rating <= 9; rating++)
    {
        if (freq[rating] > largest)
        {
            largest = freq[rating];
            modeValue = rating;
        }
    }

    printf("The mode is %d, which occurred %d times\n", modeValue, largest);
}
```

Two loops. The first builds the frequency array by using each response as an index. The second is a plain search for the largest value, keeping track not of the largest count itself but of which rating produced it, because it is the rating we were asked for.

One detail in the caller: `freq` must be declared with ten elements, not nine, and initialised to zeros: `int frequency[10] = {0};` Ten because the ratings run from 1 to 9 and we are indexing by the rating directly, so `freq[9]` has to exist, which means index 0 is there too and simply goes unused. Zeros because the counts have to start from somewhere, and an uninitialised array holds junk.

Note also that `freq` is not `const` while `answer` is. The function writes into the frequency array, so the caller sees the counts afterwards; it only reads the responses, and says so.

Looking forward

This lecture had less C in it than any so far, and more thinking. Binary search is four lines of logic that turn a million comparisons into twenty, and it works only because somebody noticed that a sorted array tells you something about where a value cannot be. Bubble sort is a comparison and a three-line swap, and the whole difficulty was in seeing that repeatedly fixing adjacent pairs eventually fixes everything.

In both cases the code was the easy part. That will keep being true, and increasingly so.

Exercises

1. Trace binary search by hand on the array `{1, 3, 4, 7, 11, 12, 13, 16, 23, 25, 31, 36, 37, 40, 41, 47}` searching for the key 47. Write down `low`, `middle` and `high` at every step. How many comparisons does it take, and how many would linear search take?

2. Trace the same array searching for 8, which is not in it. Follow `low` and `high` until the loop ends, and satisfy yourself that the function returns -1 rather than looping forever.

3. Implement `binarySearch` and add a `printf` inside the loop that prints `low`, `middle` and `high` on every pass. Run it on the array above for a few different keys and check it against what you traced by hand.

4. What happens if you run `binarySearch` on an array that is not sorted? Try it on `{99, 75, 9, 65, 70, 55}` searching for 75. Explain the result.

5. Change `bubbleSort` so that it sorts from high to low instead of low to high.

6. As written, `bubbleSort` always does `size - 1` passes even when the array came out sorted after two. Add a variable that records whether any swap happened during a pass, and stop early if none did. Check that it still sorts correctly, then check that it now finishes quickly on an array that is already in order.

7. Write the full survey program. Store these 99 responses in an array and report the mean, median and mode:

   ```
   int response[99] =
       {6, 7, 8, 9, 8, 7, 8, 9, 8, 9, 7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
        6, 7, 8, 9, 3, 9, 8, 7, 8, 7, 7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
        6, 7, 8, 7, 8, 7, 9, 8, 9, 2, 7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
        5, 6, 7, 2, 5, 3, 9, 4, 6, 4, 7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
        7, 4, 4, 2, 5, 3, 8, 7, 5, 6, 4, 5, 6, 1, 6, 5, 7, 8, 7};
   ```

   You should get a mean of 6.8788, a median of 7 and a mode of 8, which occurs 27 times.

8. The `bubbleSort` function in the lecture doesn't return anything. Is this not a mistake?