The dictionary problem.

A dictionary can be seen as a database of records; in each record we distinguish the *key* part (the word) and the *data* part (its definition).

When sorting such a database, we sort according to the *key* part, and the rest of the record gets a free ride.

> *For example, we might sort an array of employee records into alphabetical order of employee name, or into numeric order of salary, or into date order of joining the company.*
>
> *Same records – different keys.*

The cost of ***comparison*** will depend on the size of the key; the cost of ***exchange*** or ***copy*** will depend on the size of the record.

None of the sorting algorithms so far needs to be changed to cope with the key / data distinction.

> *Though we might have to choose differently between algorithms depending on the relative cost of comparison and exchange.*

In Java, we can define an `interface` for items which can be sorted (we provide an ordering method) or be searched for (provide an equality method).

Weiss (pp 91-93) defines a `Comparable` interface. In principle we need (=) and (≤):

```
public interface RBComparable {
  int iseq(RBComparable b);
  int islesseq(RBComparable b);
}
```

> *Any class which* `implement`*s this interface must provide at least those two methods.*

An example implementation of this interface for a word→string list dictionary:

```
public class DictElem implements RBComparable
extends ... implements ... etcetera ... {
  private String word;
  private StringList definition;
  ... (loads of other stuff) ...
  int iseq(RBComparable b) {
    return b instanceof DictElem &&
           word==((DictElem)b).word;
  }
  int islesseq(RBComparable b) {
    return ((DictElem)b).comparesTo(word)<=0;
  }
  ... (more stuff) ...
}
```

Note a subtle distinction between iseq and islesseq: one always delivers a value, the other may throw an exception.

I shall continue, in presenting algorithms, to pretend that I can use operators like == and <= to compare keys of records; in practice you might have to use methods from an interface like `RBComparable`.

In the case of searching in arrays (binary chop and hash addressing), the key / data distinction isn't important. When it comes to searching in recursive data structures (binary trees, B-trees), it comes to the front.

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

Searching in arrays: binary chop.

We want to find a record in a sequence $A_{m..n-1}$ with key x.

We simplify this to the problem of detecting that there is a record identical to x: $\exists k : m \le k < n \wedge x = A_k$.

The obvious solution is $O(N)$-time.

We shall see later that we can search a sorted array in $O(\lg N)$ time.

Later we shall see that, given enough space, there's an $O(1)$-time solution to this problem!

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

An aside: solving "∃?" problems.

Repetition in programs is the analogue of quantification in predicate calculus.

To find *x* in an array by *sequential search*: look along the array and record success when you see an *x*:

i
```
for (k=m; k<n; k++)
  if (x==A[k]) found=true;
```

That isn't a correct solution, because it never records failure!

There is, of course*, no 'else' in this program.*

ii
```
found=false;
for (k=m; k<n; k++)
  if (x==A[k]) found=true;
```

The trivial case of ∃k... is false; we assume failure in case $A_{m..n-1}$ *is empty.*

There is no 'else' in this program either.

This program illustrates a general solution to "∃?" questions.

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

An aside: solving "∀?" problems.

There is a well-known equivalence in predicate calculus: $\forall x(P(x))$ is equivalent to $\neg(\exists x(\neg P(x)))$.

This means that to solve a "∀?" problem – are all the components of the array like this? – we *look for a counter-example*.

For example, is every element of the sequence (=*k*)?

iii
```
allsame=true;
for (k=m; k<n; k++)
  if (x!=A[k]) allsame=false;
```

The trivial case of ∀k... is true; we assume success in case $A_{m..n-1}$ *is empty.*

There is, once again, *no 'else' in this program.*

This program illustrates a general solution to "∀?" questions.

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

Solving "∃?" problems more quickly.

Suppose we write a method *find* to see if there is a value x in $A_{m..n-1}$:

iv
```
boolean find(type x, type[] A,
             int m, int n) {
  int found=false;
  for (int k=m; k<n; k++)
    if (x==A[k]) found=true;
  return found;
}
```

We might as well return *true* as soon as we find the first occurrence of x:

iv'
```
boolean find(type x, type[] A,
             int m, int n) {
  int found=false;
  for (int k=m; k<n; k++)
    if (x==A[k]) return true;
  return found;
}
```

Now we don't need the variable *found*, because it always contains *false*:

```
iv" boolean find(type x, type[] A,
                 int m, int n) {
      for (k=m; k<n; k++)
        if (x==A[k]) return true;
      return false;
    }
```

There is **still no 'else'** *in this program.*

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

Each of the examples i-iv'' implements what is called a *sequential search*; each is $O(N)$ in time and $O(1)$ in space. None of them takes any time to 'set up', or prepare the sequence for searching.

There are alternatives, even when using arrays.

Binary chop takes $O(N \lg N)$ time to setup (because the sequence must be sorted), then $O(\lg N)$ time for each subsequent search. It is $O(1)$ in space. It takes $O(N)$ time to add or delete an element from the sequence.

Hash addressing takes $O(N)$ time to setup (because it uses a table at least twice the size of the sequence you are searching), then $O(1)$ time for each subsequent search. But it's $O(N)$ in space. It takes $O(1)$ time, mostly, to add an element to the sequence, but sometimes that can be $O(N)$ – and similarly for deletion.

Binary search trees take no time to set up, and can be made to take $O(\lg N)$ time to search. But they use `new`, and so the space behaviour is unpredictable, as is insert / delete performance.

Engineering tradeoffs again: setup time vs search time, space vs both of them.

‘Binary chop’ search.

Look at the ***midpoint*** of a ***sorted*** sequence, and decide whether the sought-for key – if it’s present – must fall in the first or the second half of the sequence.

We keep on ‘probing’ until we have reached a sequence length 1; then we have a look to see if we have the key we are looking for.

> *Each ‘probe’ divides the problem in half, but does no more: that turns out to be important for reasons of efficiency.*

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

I assume that $m < n$ – that is, the sequence we are searching isn't empty:

```
boolean binarychop
      (type x, type[] A, int m, int n) {
  while (m+1!=n) {
    int k = (m+n)/2;
    if (A[k]<=x) m=k; // in top half?
    else n=k; // in bottom half?
  }
  return A[m]==x; // the answer!!!!
}
```

false assertion, often believed: "we use binary chop search when we look up a name in the telephone directory". We don't; we guess where the name might be and look there, not in the middle; from what we see we guess more accurately, and so on. It's a form of **interpolation search**.

Binary chop is what you do if you have no basis for interpolation.

How fast does binary chop search run?

Each probe (each execution of the *while* loop) divides the sequence almost exactly in half, so we make $\lceil \lg N \rceil$ probes in a sequence length N,

> $\lceil X \rceil$ *is 'the ceiling of X', the smallest integer which is not smaller than X.*

and we make one final comparison.

This is obviously $O(\lg N)$ in execution time.

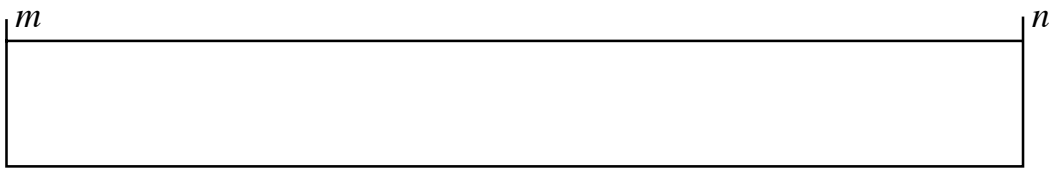
By contrast, sequential search is $O(N)$ and makes about $N/2$ comparisons on average.

If N is more than a very small number, binary chop is going to be faster than sequential search; if N is a large number, binary chop is going to be *very much faster* than sequential search.

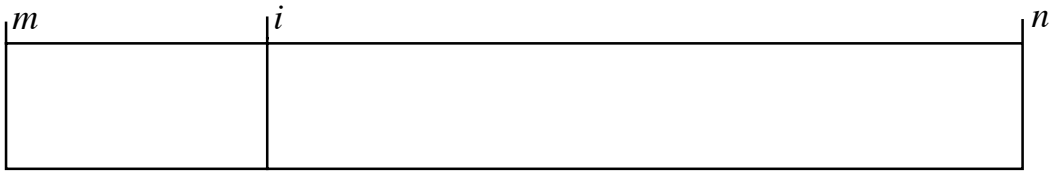
> *Don't forget the 'setup costs': the array must be sorted before the first search, which will take at least* $O(N \lg N)$ *time.*

How to make a catastrophic mistake.

It is no accident that I draw an array, indexed from m to $n - 1$, like this:



It is no accident that I write each index above the array and to the right of a vertical line:



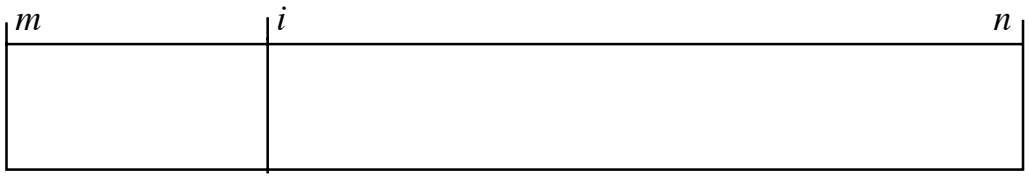
Drawing them like that makes arithmetic about the number of elements much easier.

The top picture shows an array with exactly $m - n$ elements. In the second picture the left-hand segment has $i - m$ elements, and the right-hand segment has $n - i$ elements.

It is particularly easy to find the middle element of the array. I simply write $k = (m + n) \div 2$, or as near as I can get to that in Java, to find the middle element of a sequence $A_{m..n-1}$.

> *This doesn't work exactly if the array has an even number of elements, but that isn't usually a problem.*

If you draw your arrays the ***wrong*** way, like this:



– indicating a sequence indexed from *m..n* – then it's fiddlier to find lengths, and it's much fiddlier to find mid-points.

If you index your arrays the ***wrong way***, then it's easy to write the binary chop procedure so that it loops:

vi
```
boolean badchop
        (type x, type[] A, int m, int n) {
        // find x in A[m..n]
  while (m!=n) {
    int k = (m+n)/2;
    if (A[k]<=x) m=k; // in top half?
    else n=k; // in bottom half?
  }
  return A[m]==x; // the answer!!!!
}
```

If $m + 1 = n$ then $(m + n) \div 2 = m$, and then if $A[m] \leq x$ the method will loop.

The correct formula for the midpoint of a sequence indexed $m..n$ is, of course, $(m + n + 1) \div 2$.

How to slow a binary chop search.

Suppose we look for *x* each time we make a probe:

vii
```
boolean ternarychop
          (type x, type A[], int m, int n) {
      while (m<n) {
        int k = (m+n)/2;
        if (A[k]==x) return true; // success!!
        else
        if (A[k]<x) m=k+1; // in top half?
        else n=k-1; // in bottom half?
      }
      return false; // failure!!
    }
```

This method makes *about twice as many comparisons* as binary chop.

> *It doesn't pay, sometimes, to follow your instincts as a programmer ...*

Each probe-and-test eliminates one element and reduces the problem to about half its original size.

One element in the array can be found in one probe-and-test (the middle one); two can be found on the second probe-and-test (the middle one of the top half, and the middle one of the bottom half); ... 2^i can be found on the *i*th probe-and test ...

Each probe-and-test covers 1+(the number of elements covered in all the previous probe-and-tests).

So *about half the time* you have to do $\lg N$ probe-and-tests, *about a quarter of the time* you have to do $\lg N - 1$ probe-and-tests, ...

Each probe-and-test does two comparisons; the number of comparisons on average is about

$$\frac{2\lg N}{2} + \frac{2(\lg N - 1)}{4} + \frac{2(\lg N - 2)}{8} + \ldots\ldots + \frac{1}{2^{\lg N}}$$

$$= \left\lceil \begin{array}{l} \left(1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{(N \div 2)}\right)\lg N \\ -\left(\frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{(N \div 2)}\right) + \frac{1}{N} \end{array} \right\rceil \approx 2\lg N - 1.$$

So we *don't* test on every probe; binary chop is about twice as fast as ternary chop.

Academic health warning: the argument above is valid only for a machine which takes one test to decide between two conditions ($A_k \leq x$ and $A_k > x$), but two tests to decide between three conditions ($A_k = x$, $A_k < x$ and $A_k > x$).

Insertion, deletion and binary chop.

Suppose that we want to insert x into the array, if it isn't already there.

We can use the *binarychop* technique to find the position m at which x would occur if it was in the array. Then there are three possibilities: either x comes before $A[m]$, it's equal to $A[m]$, or it comes after $A[m]$.

In the first case we must shift $A[m..n-1]$ up one position to make room; in the second we must replace the *data* part of $A[m]$; in the third we must shift $A[m+1..n-1]$ up one position.

Shifting a section of the array out of the way is $O(N)$ (slides 1); assigning the new *key* and *data* values is $O(1)$; the initial search is $O(\lg N)$; so the whole is $O(1+\lg N+N)$ which is $O(N)$.

viii
```
boolean binarychopinsert
          (type x, type[] A, int m, int n) {
      while (m+1!=n) {
        int k = (m+n)/2;
        if (A[k]<=x) m=k; // in top half?
        else n=k; // in bottom half?
      }
      if (x<A[m]) {
        for (int i=n; i>m; i--) A[i]=A[i-1];
        A[m]=x;
      }
      else
      if (A[m]<x) {
        for (int i=n; i>m+1; i--) A[i]=A[i-1];
        A[m+1]=x;
      }
      // replacement of data part not shown
    }
```

Deletion is slightly simpler: find the position m at which x would occur if it was in the array; if x does occur at that position then shift $A[m+1..n-1]$ down one position to obliterate it. That is worst-case $O(N)$ (because of the shifting) just like insertion.

Key points

Binary chop uses a sorted database.

Binary chop search is $O(\lg N)$ in time and $O(1)$ in space.

Insertion into the table is $O(N)$ in time and $O(1)$ in space, deletion likewise.

Setup costs are the cost of sorting – probably $O(N \lg N)$ in time, and either $O(N)$ space (if a database copy is required) or $O(1)$ otherwise.

Binary chop is faster than ternary chop, but it is *not* the fastest search mechanism.