Linear Search

Problem: Given a list of N values, determine whether a given value X occurs in the list.

For example, consider the problem of determining whether the value 55 occurs in:

1	2	3	4	5	6	7	8
17	31	9	73	55	12	19	7

There is an obvious, correct algorithm:

start at one end of the list,
if the current element doesn't equal the search target, move to the next element,
stopping when a match is found or the opposite end of the list is reached.

Basic principle: divide the list into the current element and everything before (or after) it; if current isn't a match, search the other case

Linear Search

```
algorithm LinearSearch takes number X, list number L, number Sz

# Determines whether the value X occurs within the list L.
# Pre:  L must be initialized to hold exactly Sz values
#

    # Walk from the upper end of the list toward the lower end,
    # looking for a match:

    while Sz > 0 AND L[Sz] != X
       Sz := Sz - 1
    endwhile

    if Sz > 0        # See if we walked off the front of the list
       display true  #  if so, no match
    else
       display false # if not, got a match

    halt
```


Binary Search

But, consider the problem of determining whether the value 31 occurs in:

1	2	3	4	5	6	7	8
7	9	12	17	19	31	55	73

Suppose we pick an arbitrary element of the list to consider first, say element #4.

Now, element #4 is 17, which is smaller than our search target (31).

But, the elements of this list are in ascending order, and that means we not only know that element #4 isn't a match, but we also know that no element that precedes it could be a match either.

This suggests that, if we have a list that is in ascending (or descending) order then there may be a more efficient approach than linear search.

The basic approach seems to be:

Pick an element in the list
While the current element doesn't match our search target,
    If the current element is larger than our search target
        pick a preceding element to consider next
    else
        pick a succeeding element to consider next

	1	2	3	4	5	6	7	8
Pick #3, too small:	7	9	12	17	19	31	55	73
Pick #7, too large:	7	9	12	17	19	31	55	73
Pick #5, too small:	7	9	12	17	19	31	55	73
Pick #6, done!:	7	9	12	17	19	31	55	73

Binary Search

This leaves some important questions:

How do we pick elements?
How do we know we are done if the search target is not in the list?

For lack of a more compelling strategy, it seems we might as well just pick an element that is in the middle of the part of the list that is still "in play"… and that suggests we give up when the part that's still "in play" becomes empty.

Binary Search: Design Details

Keeping track of what's "in play":

This seems simple enough; we keep track of the list positions that bound the part of the list that could still contain a match to our search target.

So, we need two variables, say Lo and Hi.

Initially, the whole list is in play, so we set them to 1 and N, respectively.

How do we update them?

If the current element (which will be at Mid, the average of Lo and Hi) is too large, we also just ruled out every element beyond Mid, so we should set Hi to be one less than Mid.

On the other hand, if the current element is too small, we also just ruled out every element before Mid, so we should set Lo to be one more than Mid.

Binary Search: Design Details

Choosing next element to consider:

We want to find an element that's (about) halfway between Lo and Hi.

This seems simple, we want to average Lo and Hi.

So, we need a variable, say Mid, to keep track of the value.

One issue:

The position of an element must be an integer, but the average of two integers is not necessarily an integer… is this a problem?

No. We just need to round/truncate the result to get an integer, and it doesn't seem it matters whether we round up or down; since the details of making this happen are language-dependent, we'll ignore the issue at this level.

Binary Search: Design Details

How do we know when to stop?

Well, obviously we quit if we find a match.

And, we know there is no match if the "in play" region becomes empty.

And, we will know that's happened if we ever reach the state that Lo is larger than Hi.

Binary Search

```
algorithm BinarySearch takes number X, list number L, number Sz

# Determines whether the value X occurs within the nondescending
# ordered list L.
# Pre:  L must hold exactly Sz values, in nondescending order
# Returns:
#       true if X occurs in L[1:Sz], false otherwise
#

    # Initially, the part of the list to search is from
    # index 1 to index Sz
    number Lo := 1
    number Hi := Sz
    . . .
```


Binary Search

```
   . . .
   while Lo <= Hi

      number Mid := (Lo + Hi) / 2  # find middle of in-play list

      if List[Mid] = X             # if we have a match, done
         display true
         halt
      endif

      if List[Mid] > X        # otherwise, eliminate about half
         Hi := Mid - 1
      else
         Lo := Mid + 1
      endif

   endwhile

   display false              # no match found

   halt
```


Integer Exponentiation: Naive

Problem: Given an integer X and a non-negative integer N, calculate X^N.

This has a simple solution:

```
algorithm XtoNv0 takes number X, number N

# Computes the value of X^N.
# Pre:  X and N are nonnegative integers, not both zero.
#
    XtoN := 1

    while N > 0           # iterate; on kth pass we have X^k
       XtoN := X * XtoN
       N := N - 1
    endwhile

    display XtoN
    halt
```

But this requires N multiply operations… seems expensive…

Divide and Conquer

One technique for deriving a solution to a problem involves dividing the problem into sub-parts which are easier to solve, and then deriving a solution to the original problem by somehow recombining the solutions to the sub-parts.

Commonly, a problem is broken into two nontrivial subparts and the resulting algorithm naturally involves recursion… a topic we will not explore at this time.

But in some cases, a problem is broken into a trivial part and a more complex part, and the resulting algorithm is naturally iterative.

Binary search can be viewed in the latter light, where the current element is the trivial case and the "in play" portion of the list constitutes the nontrivial part.

Exponentiation: Divide and Conquer

Consider that we can approximately cut the number of multiplications in half if we have an even exponent:

$$x^{2k} = \left(x^2\right)^k = x^2 \times x^2 \times x^2 \cdots \times x^2$$

That's k multiplications instead of $2k$… quite a savings.

And a similar "trick" works if we have an odd exponent:

$$x^{2k+1} = x\left(x^2\right)^k = x \times x^2 \times x^2 \times x^2 \cdots \times x^2$$

That's $k+1$ multiplications instead of $2k$.

But, it is possible to do even better; consider that:

$$x^{11} = x \times x^{10} = x \times x^2 \times x^8 = x \times x^2 \times \left(\left(x^2\right)^2\right)^2$$

That requires only 5 multiplications, and the advantage grows even larger if we have a larger exponent…

Exponentiation: Divide and Conquer

The key is to efficiently compute a factor whose exponent is a power of 2, since we can compute a factor of that form with a small number of multiplications:

$$x^{2^r} = \left(\left(\left(x^2 \right)^2 \right)^2 \cdots \right)^2$$

That requires only r multiplications!

But, of course, we not only have to do this, but we also have to keep track of the rest of the computation; say we want to compute x^{21} :

$$x^{21} = x \times x^4 \times x^{16}$$

So, we notice that the exponent is odd and we remember we need to throw in x^1 ;

now we need to deal with the even exponent 20, which is the 5th power of x^4 ;

but now we have an odd exponent, 5, so we remember to throw in an x^4

. . .

Exponentiation: Divide and Conquer

But, of course, we not only have to do this, but we also have to keep track of the rest of the computation; say we want to compute x^{21}:

Note	Current multiplier	Exp. remaining	Accumulated value
initial values	x	21 : odd	1
decrement exponent, apply multiplier to accumulated value	x	20 : even	x
divide exponent by 2, square multiplier	x^2	10 : even	x
divide exponent by 2, square multiplier	x^4	5 : odd	x
decrement exponent, apply multiplier to accumulated value	x^4	4 : even	x^5

Exponentiation: Divide and Conquer

But, of course, we not only have to do this, but we also have to keep track of the rest of the computation; say we want to compute x^{21}:

Note	Current multiplier	Exp. remaining	Accumulated value
prior state	x^4	4 : even	x^5
divide exponent by 2, square multiplier	x^8	2 : even	x^5
divide exponent by 2, square multiplier	x^{16}	1 : odd	x^5
decrement exponent, apply multiplier to accumulated value	x^{16}	0 stop!	x^{21}

That's 7 multiplications to compute x^{21}, not bad!

But we can eliminate the first multiplication (previous slide) since the exponent is not zero, so this can be reduced to 6 multiplications.

Integer Exponentiation: More Efficiently

```
algorithm XtoN takes number X, number N

# Computes the value of X^N.
# Pre:  X and N are nonnegative integers, not both zero.

   number XtoN := 1          # start with 1

   while  N > 0

      if N is odd            # if odd exponent
         XtoN := XtoN * X    #   accumulate value so far
         N := N – 1          #   decrement to even exponent
      endif
      if N > 0               # if exponent not zero (IS even)
         X := X * X          #   square the base value
         N := N / 2          #   cut exponent in half
      endif
   endwhile

   display XtoN
   halt
```

Polynomial Evaluation: Naive

Problem: Given a value *A* and a polynomial *P(X)*, calculate *P(A)*.

We will assume that the polynomial is represented as a list of coefficients, listed from low to high powers of the variable.

So, the polynomial $P(x) = 17x^5 - 8x^3 + x^2 + 6x - 10$

would be represented as:

1	2	3	4	5	6
-10	6	1	-8	0	17

Polynomial Evaluation: Naive


```
algorithm evalPoly takes list number P, number D, number X

# Computes the value of P(X), where P is a polynomial of degree D.
# Pre:  X, D are initialized, D is an integer, D >= 0,
#       P is a list of D + 1 numbers
#

   number value := P[1]      # initialize to constant term
   if X = 0                  # if X is zero, we're done
      display value

   number pos := 2           # otherwise, need to process rest

   . . .
```

Polynomial Evaluation: Naive

```
. . .

while pos <= D + 1                     # must use all coeff's

   if P[pos] != 0                      # but not zero coeff's

      number term := X, pow := 1

      while pow < pos – 1              # iterate to get X^k
         term := X * term
         pow := pow + 1
      endwhile

      value := value + term * P[pos]  # add C_k*x^k
   endif

   pos := pos + 1
endwhile

display value
halt
```

Polynomial Evaluation: Horner's Method

An alternative way of evaluating a polynomial is suggested by grouping and factoring:

$$\begin{aligned} P(x) &= -10 + 6x + x^2 - 8x^3 + 17x^5 \\ &= -10 + x\Big(6 + x\Big(1 + x\Big(-8 + x\big(0 + 17x\big)\Big)\Big)\Big) \end{aligned}$$

This requires 5 multiplication operations and 4 addition operations.

This is known as Horner's Method, for William George Horner (1786 – 1837).

How does that compare with the naïve version?

$$\begin{aligned} P(x) &= -10 + 6x + x^2 - 8x^3 + 17x^5 \\ &= -10 + 6x + xx - 8xxx + 17xxxxx \end{aligned}$$

This requires 11 multiplication operations and 4 addition operations.

Polynomial Evaluation: Horner's Method

```
algorithm HornersMethod takes list number P, number D, number X

# Computes the value of P(X), where P is a polynomial of degree N.
# Pre:  X, N are initialized, N is an integer, N >= 0,
#       P is a list of N + 1 numbers.
#

   number value := P[N+1]        # value = C_{N+1}

   while N >= 1                  # process remaining terms

      value := P[N] + value * X  # value = X*current + C_{N}
      N := N – 1

   endwhile

   display value
   halt
```

