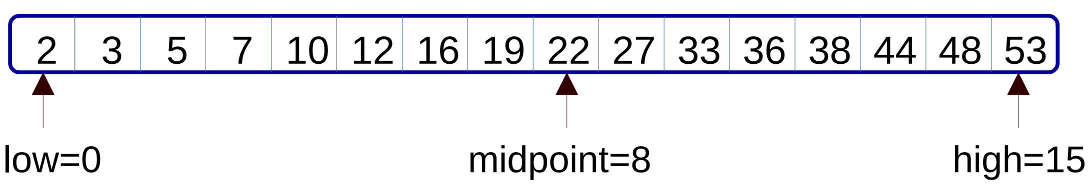
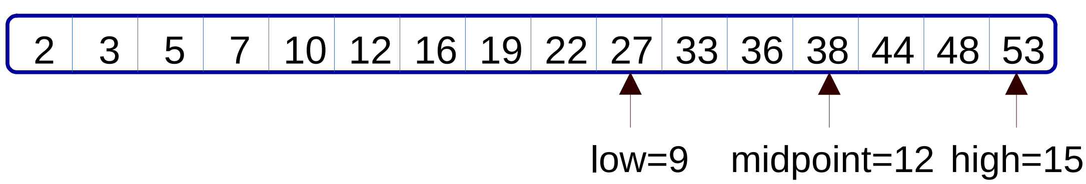
# Lab: Prove Traditional Binary Search is Problematic

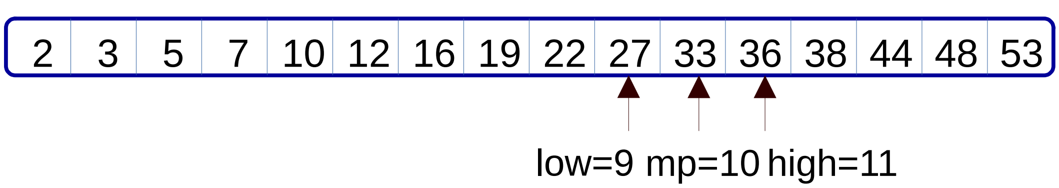
The problem of finding an element `key` in an array of elements stored in non-decreasing order is best solved using binary search. Assume the array is of size *n* and element positions are signified by the number of elements they are removed from the first element: the first element is at position 0, the element next to it is in position 1 and so on. Assume a pointer `low` references the position of the first element in the array and a pointer `high` references the position of the last element. The first move of binary search is to find the element, of value `midValue`, at the `midpoint` in the array which is a position approximately equal to *n*/2. If `midValue` is less than `key` then `key` must be in the upper half of the array due to the ordering of elements in the array. This means a binary search to find `key` may continue in an array half the size of the original so change `low` to `midpoint`+1 and search again. If `midValue` is less than `key` then `key` must be in the lower half of the array so change `high` to `midpoint`-1 and search again. If `key` is equal to `midValue` then the position of `key` in the array is found and the search is finished. Repeat the changing of `low` and `high` and comparing until either `key` is found or the search terminates without finding `key`. The following sequence shows an example. Let `key` be 33 in the array shown below. Set `low`, `high` and `midpoint` as shown.




Compare `key` with 22. Since 33 is greater than 22 change `low` and `midpoint` as follows.




Compare `key` with 38. Since 33 is less than 38 change `high` and `midpoint` as follows.




Since `key` is at the `midpoint`, it has been found in the array at position 10.

## The Problem: how to calculate the midpoint

For decades, students have been taught that the midpoint should be calculated as `(low+high)/2`. For decades, no one suspected that this might cause a problem, not only because the output position might be wrong, but also because a malicious agent might exploit this problem. The reason that this was considered safe all those years was because no one set up a array that was long enough for the problem to be discovered. That is, not until Jon Bentley publicized results of others. The problem is the sum `low+high` can overflow the memory in which the result is contained. For example, suppose integers are stored in cells that are 8 bits wide. Then, (122+174)/2 turns out to be 40/2 = 20 instead of 148 which is the expected value.

To see this examine `mid.cry`. To do this click the 'Edit' button and choose (double click) `mid.cry`. Notice `midA` at the top of the file. This is a function of two arguments, `low` and `high`. The signature of the function is `[8]->[8]->[8]` which means both inputs are 8 bit numbers and the output is an 8 bit number. One could use 32 or 64 instead of 8 but 8 is used here to illustrate the problem. The output of `midA` is `(low+high)/2`. Open the Cryptol window by clicking the Cryptol button, if Cryptol is not running. At the `Cryptol>` prompt type `:l mid.cry` and hit return to load the `mid.cry` file. At the `Main>` prompt type `:s base=10` and return (`:s base=10` presents all numbers in the console as decimal rather than hexadecimal which is the default) then `midA 122 174` and return to find the value of (122+174)/2. The result is 20. Run `midA 42 66` at the `Main>` prompt to get 54 which is the expected value and illustrates that the computed midpoint is correct if there is no overflow.

An alternative, and safe, computation of `midpoint` is given in `mid.cry` as the function `midS`. This is `low+(high-low)/2`. Observe the property `midS_is_safe`. If this property is true, it says that as long as both `low` and `high` are less than 255 (the highest 8 bit number) and `low` is no greater than `high`, then `high-low` is less than 256 (does not overflow) and `(high-low)/2` is less than `256-low` (so the midpoint computation, `low+(high-low)/2`, is less than 256 and therefore does not overflow 8 bits). Observe the property assumes 9 bit numbers to prevent overflow. To show this property is true, at the `Main>` prompt type `:prove midS_is_safe` and hit return. The response will be `Q.E.D`.

A second property in `mid.cry`, named `midpointsAreClose` says if `high` is less then 255, `low` is no greater than `high`, and `low` is at least 0, then the value computed by `midA` is at most 1 different from the value computed by `midS` and if `low` is no greater than `high` but at least 0, and `high` is `255` then the value computed by `midS` is equal to the value computed by `midA`. Thus both `midA` and `midS` compute nearly the same, if not the same, midpoints. To prove `midpointsAreClose` is true type `:prove midpointsAreClose` at the `Main>` prompt and hit return. The result will be `Q.E.D`. Again the property is written for 9 bit numbers to prevent overflow.

## Binary Search

The file `csearch_bug.cry` contains the implementation of binary search using the traditional calculation of midpoint (`(low+high)/2`). Open the file in the text editor (click the Edit button then choose `csearch_bug.cry`). Observe the function `computeMid` which computes the midpoint using `(low+high)/2`. Lines 22 to 24 choose which half of the remaining input array to search for the `key`, as above. Observe array `lst3` which will be used for testing. Load `csearch_bug.cry` in Cryptol (`:l csearch_bug.cry`) and run `:s base=10`, if necessary, then `bsearch(45,lst3)` at the `Main>` prompt. The result is 27. Check that 45 is at position 27 in `lst3`. Run `bsearch(239,lst3)`. The result is 255. Check that 239 is NOT at position 255 in `lst3`.

The file `csearch_nobug.cry` contains the implementation of binary search using the safe calculation of midpoint. Open the file in the text editor and observe function computeMid which computes the midpoint using `low+(high-low)/2`. Load `csearch_nobug.cry` in Cryptol (`:l csearch_nobug.cry`) and run `:s base=10`, if necessary. Observe that all aspects of `bsearch`, and `ff` are identical to those functions in `csearch_bug.cry`: only `computeMid` is different. Run `bsearch(45,lst3)` at the `Main>` prompt. The result is 27 as before. Run `bsearch(239,lst3)`. The result is 195 which corrects the error of `bsearch` in `csearch_bug.cry`. Now it is appropriate to <u>prove</u> that bsearch with the safe computation of midpoint will always return the correct position of a given `key` in a given array, in this case for 8 bit numbers and 8 bit arithmetic. Please note, 8 bits is used merely to demonstrate the concept of ‘proof’. This could be tried with 16 bits, 32 bits, or 64 bits but would take a lot longer, may not even finish due to excessive time or memory requirements, and in any case, the objective is to show that the traditional midpoint calculation runs the risk of error unnecessarily.

## The Specification

The proof of correctness is based on a comparison of `bsearch` outputs with outputs of a second function that also searches a given array for the location of a given `key` but which is designed so simply that there can only be high confidence that the function always correctly finds the position of the `key` if it is in the given array or returns -1 if the `key` is not in the array. The second function will be called the <u>specification.</u> If the specification is functionally equivalent to `bsearch` then it can be said that the correctness of `bsearch` is highly assured. In the case of search, it is easy to construct such a specification: just walk through the array, comparing each element to the `key`, stop when found (if found), and report the position, or else (if not found) report -1. Of course, if duplicate elements are allowed in the array, the position returned by the specification may be different from the position returned by `bsearch`. Thus, for the proof to succeed, it is required that input arrays have no duplicates. An exercise below asks to remove that restriction and redesign `bsearch` to return true if and only if the given `key` is in the given array.

The file `csearch_nobug.cry` contains the specification that will be used in the proof. It is called `refSearchIdx`. Here it is:

```
refSearchIdx (key, lst) = s!0
  where
    s = [-1]#[ if (key == x /\ h == -1) then c else h | x <- lst |  h <- s | c <- [0...]]
```

This function will be easy to understand after the lesson that acquaints readers with the cryptol language but, for now, the following paragraph attempts to convince the reader that the output, given inputs `key` and array `lst`, is the position of key in `lst`, if it exists in `lst`, and -1 if not. The reader may safely skip to the next section if not willing to struggle with understanding how the syntax of this expression supports the desired semantics.

The variable `s` is an array that is initialized to `[-1]` (that is, an array of a single element, namely -1) due to `[-1]#[...]`. Elements are added to the array s, one at a time, one for each element `x` of array `lst`, in order, due to `x <- lst`. For every element that is added to `s`, `h` gets the previous value of the last element of `s` due to `h <- s`. The condition of the if statement is `key == x` and `h == -1` (symbol `/\` represents logical 'and'). If the key is not equal to any of the elements `x` taken from `lst` so far, then the condition of the `if` has always been false so far and `h` is the next element of `s` due to `else h`. Thus, as long as the `key` is not equal to the first so many elements of input array `lst`, -1 is appended to s (note `s = [-1,-1,-1,...-1]` after `lst` is exhausted if the `key` is not an element of `lst`). While this is happening, due to `c <- [0...]`, variable `c` gets numbers 0 then 1 then 2 and so on, while `x` is pulling numbers from `lst`, until `x` can no longer pull elements from `lst` because it has taken them all out. Thus `c` is the position in `lst` of the current `x` that is being considered in the `if` condition. Now suppose the `key` is equal to the `x` and `h` is -1. Then `s` is `[-1,-1,...-1]` and, due to `key == x` and `h == -1`, the next element of `s` is `c`, which is the position of the `x` that equals the key. All further elements x taken from `lst` result in a false condition due to h not equal to -1 (this requires an assumption that all elements are positive numbers). Thus, when all elements of `lst` have been pulled, and building s is complete, s is similar to `[-1,-1,-1,...,-1,P,P,P,P,P]` where `P` is the position of the first element of `lst` that is equal to the `key`. But `s` does not get returned. Instead, `s!0` gets returned: this is the last element of `s` which is `P`. If the `key` is not an element of `lst` then -1 is returned. The function `refSearchIdx` is a simple, straightforward, implementation of a search through an array of numbers looking for a `key`, and reporting the `key`'s position in the array if it is there. The writer of refSearchIdx has complete confidence in this (note: the possibility that the specification can be wrong means we can't say for sure whether correctness is inferred but we should be able to say we have high assurance of correctness if the correctness theorem is proved). The correctness theorem for `bsearch` in `csearch_nobug.cry` is next.

## Correctness

It is desired to show `refSearchIdx(key,lst)` is functionally equivalent to `bsearch(key,lst)`. In Cryptol this is expressed like this:

```
bsearch (key, lst) == refSearch (key, lst)
```

But this is not the whole story because there are some assumptions that must be made in order for this to hold. These have been mentioned earlier. Since -1 is used to indicate the key is not found, one assumption is `0 <= key`. Since refSearchIdx is likely to return positions different from those returned by bsearch when input arrays contain duplicates, we need to assume the input arrays are strictly increasing. To satisfy this assumption a function needs to be constructed as follows:

```
onlyIncreasing lst = pairComps == ~zero
   where pairComps = [ x < x' | x  <- [0] # lst | x' <- lst ]
```

All this function does is compare every two consecutive elements in the given `lst`. The array `pairComps` is produced as a sequence of Bits where each bit is `True` if a corresponding comparison resulted in the leftmost element of a pair strictly less than the rightmost element and `False` otherwise. If all bits of `pairComps` are `True`, the comparison with `~zero` is `True` and if any one bit is `False`, the comparison with `~zero` is `False`. So `onlyIncreasing` returns `True` if and only if the input array is strictly increasing. See the exercise to remove the requirement of strictly increasing arrays.

It is desired to prove

`onlyIncreasing(lst)` and `0 <= key` implies `bsearch(key,lst) == refSearchIdx(key,lst)`

If `P` and `Q` are propositions, `P` implies `Q` can be written in Cryptol as the following:

```
if P then Q else True
```

In this case, we can write

```
property bsearchOK key lst =
  if (onlyIncreasing(lst) /\ 0 <= key)
  then bsearch(key, lst) == refSearchIdx(key, lst)
  else True
```

There is one last problem. Cryptol requires properties to be written as monomophic types before they can be proved. But Cryptol tries to infer the most general type for every function and property unless the user overrides that type with something else. One can find the inferred type of `bsearchOK` using `:t bsearchOK`. The result is:

```
{n, m} (8 >= width n, fin m, fin n) => [m] -> [n][m] -> Bit
```

which is polymorphic (polymophism is usually good except not when proving something). The following would be a monomorphic type suitable for `bsearchOK`:

```
bsearchOK : [8] -> [100][8] -> Bit
```

Placing this line above the definition for `bsearchOK` given above allows a proof to be managed. This has been done in `csearch_nobug.cry`. This signature says, the `key` is a number of 8 bits, the array `lst` contains 100 8 bit numbers, and the output is `True` or `False`. To demonstrate the proof, type `:l csearch_nobug.cry` and hit return at the `Cryptol>` or `Main>` prompt. Next, select the prover to use. Cryptol allows the use of five SMT solvers for proving properties. These are: `z3`, `cvc4`, `boolector`, `yices`, and `abc`. For this demonstration `cvc4` is best and should prove in about 4 seconds. To select `cvc4` type `:s prover=cvc4` at the `Main>` prompt. Then type `:prove bsearchOK` and hit return at the `Main>` prompt. The result is similar to this:

```
Q.E.D.
(Total Elapsed Time: 4.221s, using "CVC4")
```

The `refSearchIdx` and `bsearch` functions are equivalent.

## Exercise

Make small changes to the functions and properties in `csearch_nobug.cry` so that `bsearch` returns `True` or `False` if `key` is in `lst` or is not in `lst`, respectively, and similarly for `refSearchIdx` (which now should be renamed to `refSearch`). Relax the precondition of `bsearchOK` to `nonDecreasing` from `onlyIncreasing` (you have to write `nonDecreasing(lst)`). Prove `bsearchOK` correct. What SMT solver is best for this and how much time did it take?