A Familiar Search Algorithm

Binary search works if the array is sorted
1. Look for the target in the middle.
2. If you don’t find it, you can ignore half of the array, and repeat the process with the other half.
Example: Find first page of pizza listings in the yellow pages
Let’s solve this again, recursively

Recursive Binary Search

Key idea – do a little bit of work, and make recursive call to do the rest
Binary search has value restricted to a range
Look at midpoint, and decide which half of the range is of interest
Use binary search to find value in reduced range. Recursion.

Binary Search Strategy

0 \(\leq\) L \(\leq\) mid \(\leq\) R \(\leq\) n

Values in b[0..L] \(\leq\) x
Values in b[R..n-1] > x
Values in b[L+1..R-1] are unknown

\[\text{mid} = \frac{(L + R)}{2}\]
Compare b[mid] and x
Replace either L or R by mid

Base Case

No remaining unknown area:

0 \(\leq\) L \(\leq\) R \(\leq\) n

We recognize the base case when L+1 == R
Recursive Case

Situation while searching

\[
\begin{array}{cccccc}
0 & L & ? & R & n \\
\text{L & R}
\end{array}
\]

Step: Look at \( b[(L+R)/2] \). Move L or R to the middle depending on test
Each recursive call is given L and R as parameters

The Search Function

The original search problem called for a function with 3 parameters:
\[
\text{int bsearch (int b[ ], int n, int x);} \\
\]

Our recursive approach requires L and R as parameters
Let’s call this function by a different name:
\[
\text{int bsearchHelper (int b[ ], int L, int R, int x)} \\
\]

Recursive Search Function

\[
\text{int bsearchHelper (int a[ ], int L, int R, int x) \{} \\
\text{int mid;} \\
\text{if (L+1 == R) /*base case*/} \\
\text{\quad return L;} \\
\text{mid = (L+R)/2; /*recursive case*/} \\
\text{if (a[mid] <= x)} \\
\text{\quad L = mid;} \\
\text{\quad else} \\
\text{\quad R = mid;} \\
\text{\quad return bsearchHelper(a, L, R, x);} \\
\text{\}} \\
\]

Initialization Dilemma

The proper initial values for L and R are:
\[
\text{L = -1;} \\
\text{R = n;} \\
\]
These initializations cannot be inside the bsearchHelper function, since L and R are parameters!

Termination Dilemma

After the base case is reached, we must make the final decision about what value to return: -1 if not found, L if found
This decision cannot be placed inside bsearchHelper!

Solution: A "Wrapper" Function

1. It sets the recursion in motion
   Calls the recursive function with the correct initial parameters
2. After the recursion completes, determines the correct final action
Non-Recursive Wrapper

```c
int bsearch (int a[], int asize, int x) {
    int L = -1;
    int R = asize;
    L = bsearchHelper (a, L, R, x); /*kickoff*/
    if (a[L] == x) /* final */
        return L;
    else
        return -1;
}
```

Trace

```
-17  -5   3   6   12  21  45  142
bSearch(a, 8, 5)          -1
  bsearchHelper(a, -1, 8, 5)   2
    bsearchHelper(a, -1, 3, 5)  2
      bsearchHelper(a, 1, 3, 5)  2
```

Iteration vs. Recursion

Turns out any iterative algorithm can be reworked to use recursion instead (and vice versa).
There are programming languages where recursion is the only choice!
Some algorithms are more naturally written with recursion
But naïve applications of recursion can be inefficient

When to Use Recursion?

Problem has one or more simple cases
These have a straightforward nonrecursive solution, and:
Other cases can be redefined in terms of problems that are closer to simple cases
By repeating this redefinition process one gets to one of the simple cases

Recursion Wrap-up

Recursion is a programming technique
   It works because of the way function calls and local variables work
Recursion is more than a programming technique