CPSC 221

Basic Algorithms and Data Structures

May 25, 2015

Administrative stuff

Lab 4 posted.

Programming Assignment 1 to be posted soon.

Theory Assignment 1 had a typo and has been corrected.

Administrative stuff

How to turn in your homework:

Go to https://my.cs.ubc.ca/docs/hand-in

Follow instructions.

This assignment is ‘ta1’.

Try it today to make sure it works for you.

We don’t accept emailed submissions.

Why “eyeballing” is dangerous

From Koffman and Wolfgang book, pp. 178-179.

Determine how many times the output statement is displayed in the following fragment. Indicate whether the fragment execution time is O(n) or O(n^2).

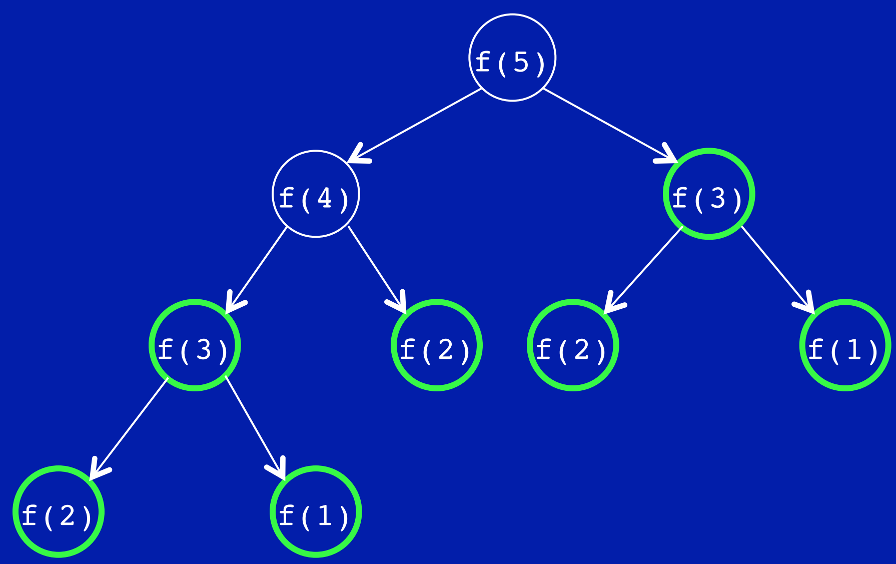
```
for (int i = 1; i < n; i++)
   for (int j = 0; j < i; j++)
     if (j % i == 0)
       cout << i << "   " << j << endl;
```

Just assuming that nested loops always give O(n^2) behaviour is assuming too much. And why are we counting output statements here? First, it’s what was asked for. Second, input/output takes way more time than any of the other operations going on here.

Visualizing recursion

How does it work? The recursion tree model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```



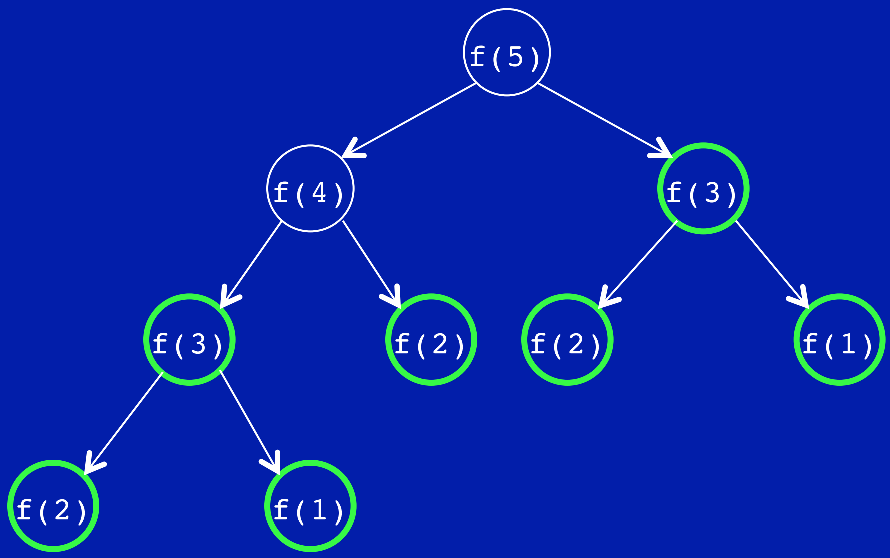
fib(3) is evaluated 2 times
fib(2) is evaluated 3 times
fib(1) is evaluated 2 times

How efficient is that?
Not very. Your book confirms that T(n) for fibonacci(n) increases exponentially with n, because of all the duplicated function calls.

Visualizing recursion

How does it work? The recursion tree model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```



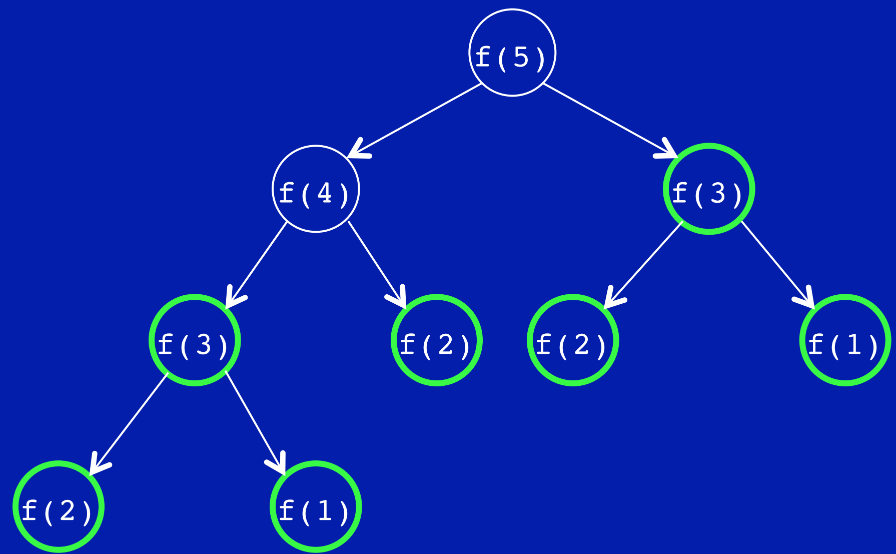
fib(3) is evaluated 2 times
fib(2) is evaluated 3 times
fib(1) is evaluated 2 times

How efficient is that?
Your book says that fib(100) requires about 2^{100} activation frames. If your computer can process 1,000,000 activation frames per second, it'll take 3×10^{16} years.

Visualizing recursion

How does it work? The recursion tree model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```



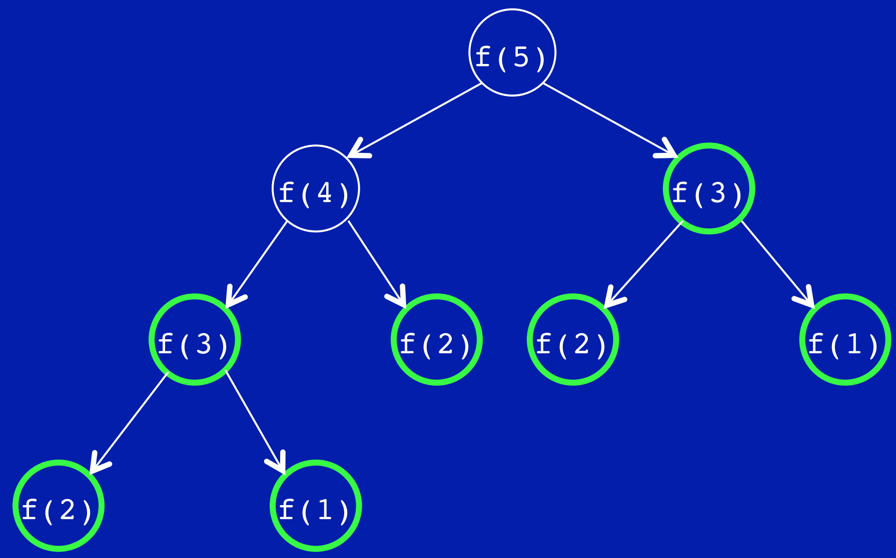
fib(3) is evaluated 2 times
fib(2) is evaluated 3 times
fib(1) is evaluated 2 times

How efficient is that?
But computers have become much faster since the book was written. With gigahertz speeds, we might process 1,000,000,000 frames/sec. Now we're down to a mere 3×10^{13} years. I feel much better now.

Visualizing recursion

How does it work? The recursion tree model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```



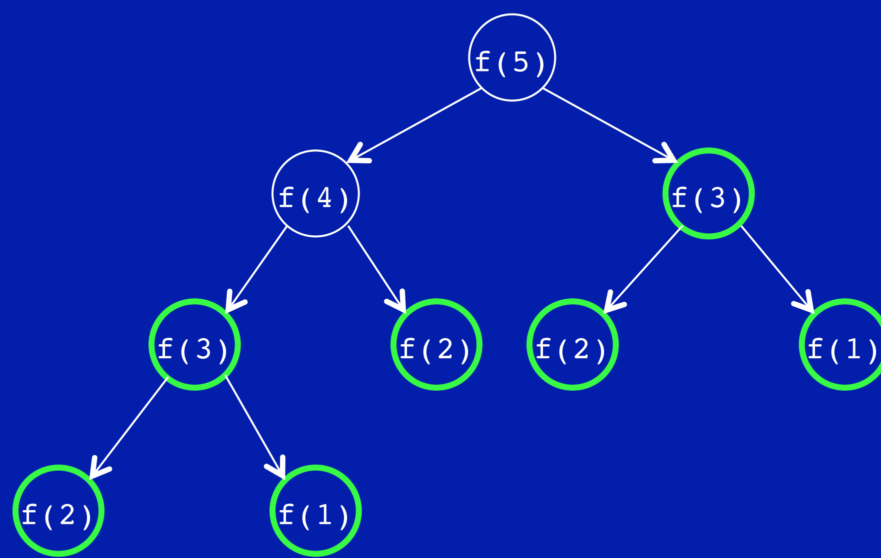
fib(3) is evaluated 2 times
fib(2) is evaluated 3 times
fib(1) is evaluated 2 times

How efficient is that?
We'll let you do the formal proof on your own.

Faster Fibonacci?

Can we make it go faster? Sure...

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

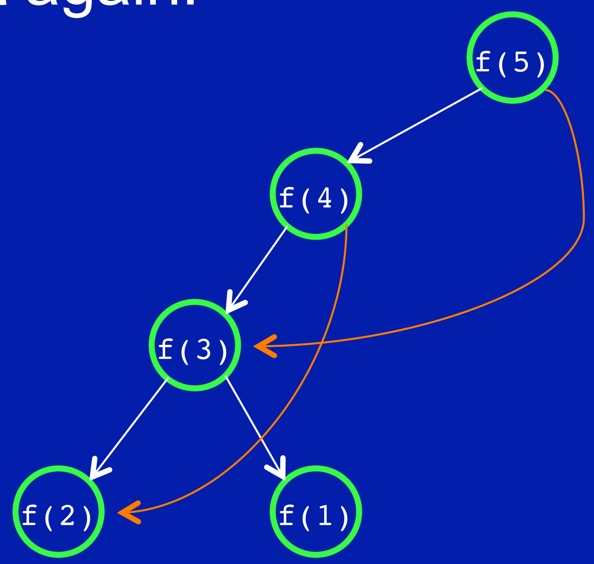


fib(3) is evaluated 2 times
fib(2) is evaluated 3 times
fib(1) is evaluated 2 times

How efficient is that?
We'll let you do the formal proof on your own.

Faster Fibonacci?

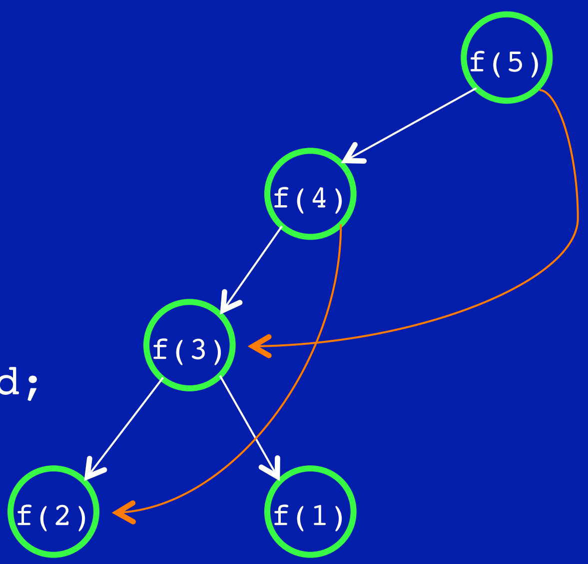
What we really want is to “share” nodes in the recursion tree so that for any n, once fib(n) is evaluated, we don’t have to evaluate it again.



Faster Fibonacci?

Here's one way. The program sort of "walks" up the left side of the tree.

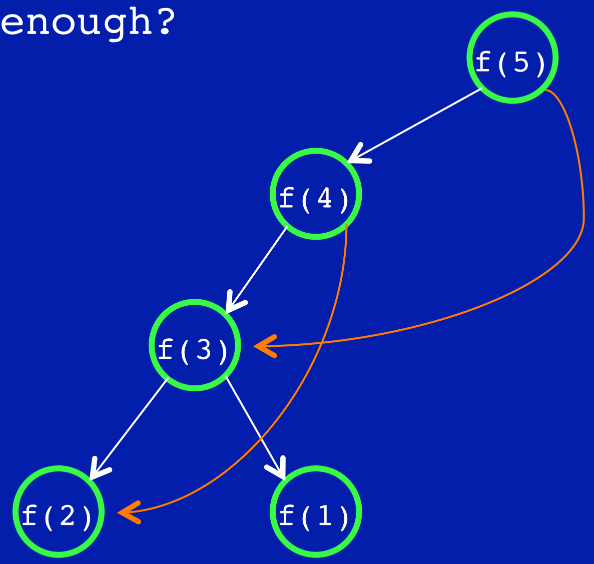
```
int fibit(int n)
{
  if (n == 1) return 1;
  int fib = 1, fib_old = 1;
  int i = 2;
  while (i < n)
  {
    int fib_new = fib + fib_old;
    fib_old = fib;
    fib = fib_new;
    i++;
  }
  return fib;
}
```



Faster Fibonacci?

Here's another way. This approach uses recursion and records problems as it solves them for later use:

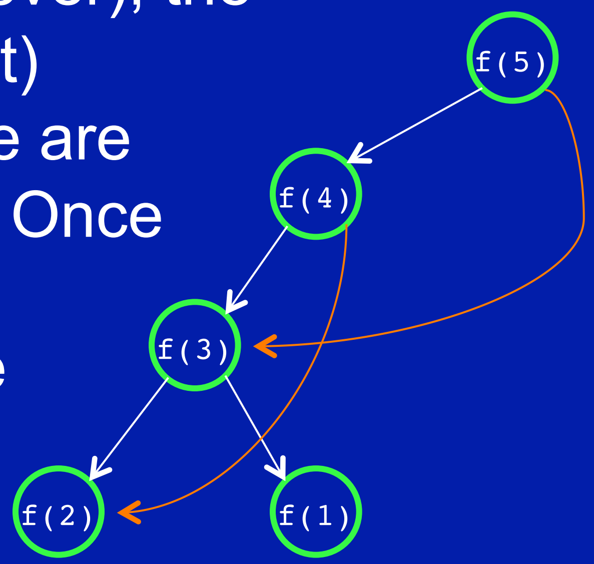
```
int fib_solns [200]; // large enough?
fib_solns[1] = 1;
fib_solns[2] = 1;

int fibm(int n, int solns [])
{
  if (solns[n] == 0)
      solns[n] =
        fibm(n - 1, solns) +
        fibm(n - 2, solns);
  return solns[n];
}
```



This technique is called "memoization"

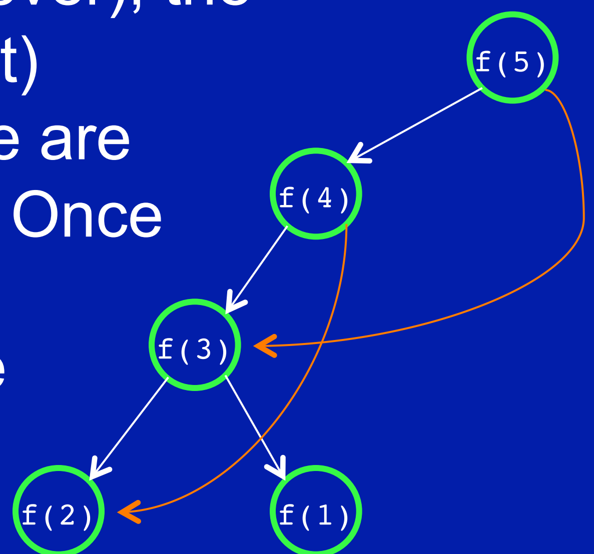
Faster Fibonacci?

Still another way is to let the programming language take care of it. In a “pure” functional programming language (i.e., one with no mutation ever), the interpreter can (but may not) notice that nodes in the tree are the same and share them. Once a function with a specific parameter is evaluated, the result is remembered and the function is never evaluated with that parameter again. How is this possible?



Faster Fibonacci?

Still another way is to let the programming language take care of it. In a “pure” functional programming language (i.e., one with no mutation ever), the interpreter can (but may not) notice that nodes in the tree are the same and share them. Once a function with a specific parameter is evaluated, the result is remembered and the function is never



evaluated with that parameter again. How is this possible? In a pure functional language, fib(n) for some specific n must always return the same result, so it need not be evaluated more than once.

Questions?

Recursion and arrays

72	3	19	57	8	21	44	68	99	80	33	6	15	51	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

CPSC 110 conditions us to think that we use recursion with linked list data structures, because linked lists are recursively-defined (i.e. self-referential). The recursive functions just “fall out” of the data definition.

Recursion and arrays

72	3	19	57	8	21	44	68	99	80	33	6	15	51	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

CPSC 110 conditions us to think that we use recursion with linked list data structures, because linked lists are recursively-defined (i.e. self-referential). The recursive functions just “fall out” of the data definition.

But there’s nothing to stop us from using recursion with arrays. Here are a couple of examples...

Linear search with recursion

72	3	19	57	8	21	44	68	99	80	33	6	15	51	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int linSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    if (array[left] == target)
        return left;
    else
        return linSearch(array, target, left + 1, right);
}
```

Linear search with recursion

72	3	19	57	8	21	44	68	99	80	33	6	15	51	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int linSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    if (array[left] == target)
        return left;
    else
        return linSearch(array, target, left + 1, right);
}

cout << linSearch(array, 57, 0, 14) << endl;

linSearch(array, 57, 0, 14)
```

Linear search with recursion

72	3	19	57	8	21	44	68	99	80	33	6	15	51	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int linSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    if (array[left] == target)
        return left;
    else
        return linSearch(array, target, left + 1, right);
}

cout << linSearch(array, 57, 0, 14) << endl;

linSearch(array, 57, 0, 14)
  linSearch(array, 57, 1, 14)
```

Linear search with recursion

72	3	19	57	8	21	44	68	99	80	33	6	15	51	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int linSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    if (array[left] == target)
        return left;
    else
        return linSearch(array, target, left + 1, right);
}

cout << linSearch(array, 57, 0, 14) << endl;

linSearch(array, 57, 0, 14)
  linSearch(array, 57, 1, 14)
    linSearch(array, 57, 2, 14)
```

Linear search with recursion

72	3	19	57	8	21	44	68	99	80	33	6	15	51	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int linSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    if (array[left] == target)
        return left;
    else
        return linSearch(array, target, left + 1, right);
}

cout << linSearch(array, 57, 0, 14) << endl;

linSearch(array, 57, 0, 14)
  linSearch(array, 57, 1, 14)
    linSearch(array, 57, 2, 14)
      linSearch(array, 57, 3, 14) // yippee!!
```

Linear search with recursion

72	3	19	57	8	21	44	68	99	80	33	6	15	51	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int linSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    if (array[left] == target)
        return left;
    else
        return linSearch(array, target, left + 1, right);
}

cout << linSearch(array, 57, 0, 14) << endl;

linSearch(array, 57, 0, 14)
  linSearch(array, 57, 1, 14)
    linSearch(array, 57, 2, 14)
      linSearch(array, 57, 3, 14) // yippee!!
3 // the returned value
```

Binary search with recursion

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

A linear search on an unsorted array is probably more simply implemented as a for loop instead of a series of recursive function calls. But there are other situations where a recursive search algorithm applied to an array is a very simple and elegant solution...

Binary search with recursion

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}
```

Binary search with recursion

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}

cout << binSearch(array, 55, 0, 14) << endl;

binSearch(array, 55, 0, 14)
```

Binary search with recursion

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}

cout << binSearch(array, 55, 0, 14) << endl;

binSearch(array, 55, 0, 14)
```

Binary search with recursion

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}

cout << binSearch(array, 55, 0, 14) << endl;

binSearch(array, 55, 0, 14)
  binSearch(array, 55, 8, 14)
```

Binary search with recursion

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}

cout << binSearch(array, 55, 0, 14) << endl;

binSearch(array, 55, 0, 14)
  binSearch(array, 55, 8, 14)
```

Binary search with recursion

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}

cout << binSearch(array, 55, 0, 14) << endl;

binSearch(array, 55, 0, 14)
  binSearch(array, 55, 8, 14)
    binSearch(array, 55, 8, 10)
```

Binary search with recursion

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}

cout << binSearch(array, 55, 0, 14) << endl;

binSearch(array, 55, 0, 14)
  binSearch(array, 55, 8, 14)
    binSearch(array, 55, 8, 10)
```

Binary search with recursion

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}

cout << binSearch(array, 55, 0, 14) << endl;

binSearch(array, 55, 0, 14)
  binSearch(array, 55, 8, 14)
    binSearch(array, 55, 8, 10)
      binSearch(array, 55, 10, 10)
```

Binary search with recursion

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}

cout << binSearch(array, 55, 0, 14) << endl;

binSearch(array, 55, 0, 14)
  binSearch(array, 55, 8, 14)
    binSearch(array, 55, 8, 10)
      binSearch(array, 55, 10, 10)
```

Binary search with recursion

5	10	15	20	25	30	35	40	45	50	$\boxed{55}$	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}

cout << binSearch(array, 55, 0, 14) << endl;

binSearch(array, 55, 0, 14)
  binSearch(array, 55, 8, 14)
    binSearch(array, 55, 8, 10)
      binSearch(array, 55, 10, 10)
10 // the returned value
```

Binary search with recursion

It was suggested that, in our phone book example, we didn't do binary search. We estimated where the name might be and started there, instead of in the middle.

This is called interpolation search.

Binary search with recursion

In the best of conditions, interpolation search can be faster than binary search: O(log log n) vs. O(log n).

But the data must be uniformly distributed.

And to justify the extra expense of making a guess as to where to search next, accessing the array should be very expensive compared to typical instructions (e.g. array stored on disk)

Binary search with recursion

However, if the data is not uniformly distributed, and/or if the calculation for where the item might be isn't accurate, the worst case run time could be O(n).

Because binary search is so good in most cases, interpolation search isn't used all that often.

Recursion vs. Iteration

Both involve the repetition of a sequence of statements.

An iterative solution exists for any problem solvable by recursion.

An iterative solution may be more efficient.

A recursive solution is often easier to understand.

Here's the iterative version. You decide...

Binary search with iteration

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearchIt(int array[], int target, int left, int right)
{
    int result = -1;
    while (! (right < left))
    {
        int mid = (left + right) / 2;
        if (array[mid] == target)
        {
            result = mid;
            right = left - 1;  // kill the loop
        }
        else if (target < array[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    return result;
}
```

Binary search with iteration

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearchIt(int array[], int target, int left, int right)
{
    int result = -1;
    while (! (right < left))
    {
        int mid = (left + right) / 2;
        if (array[mid] == target)
        {
            result = mid;
            right = left - 1;  // kill the loop
        }
        else if (target < array[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    return result;
}
cout << binSearch(array, 55, 0, 14) << endl;
```

Binary search with iteration

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearchIt(int array[], int target, int left, int right)
{
    int result = -1;
    while (! (right < left))
    {
        int mid = (left + right) / 2;
        if (array[mid] == target)
        {
            result = mid;
            right = left - 1;  // kill the loop
        }
        else if (target < array[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    return result;
}
cout << binSearch(array, 55, 0, 14) << endl;
```

Binary search with iteration

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearchIt(int array[], int target, int left, int right)
{
    int result = -1;
    while (! (right < left))
    {
        int mid = (left + right) / 2;
        if (array[mid] == target)
        {
            result = mid;
            right = left - 1;  // kill the loop
        }
        else if (target < array[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    return result;
}
cout << binSearch(array, 55, 0, 14) << endl;
```

Binary search with iteration

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearchIt(int array[], int target, int left, int right)
{
    int result = -1;
    while (! (right < left))
    {
        int mid = (left + right) / 2;
        if (array[mid] == target)
        {
            result = mid;
            right = left - 1;  // kill the loop
        }
        else if (target < array[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    return result;
}
cout << binSearch(array, 55, 0, 14) << endl;
```

Binary search with iteration

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearchIt(int array[], int target, int left, int right)
{
    int result = -1;
    while (! (right < left))
    {
        int mid = (left + right) / 2;
        if (array[mid] == target)
        {
            result = mid;
            right = left - 1;  // kill the loop
        }
        else if (target < array[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    return result;
}
cout << binSearch(array, 55, 0, 14) << endl;
```

Binary search with iteration

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearchIt(int array[], int target, int left, int right)
{
    int result = -1;
    while (! (right < left))
    {
        int mid = (left + right) / 2;
        if (array[mid] == target)
        {
            result = mid;
            right = left - 1;  // kill the loop
        }
        else if (target < array[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    return result;
}
cout << binSearch(array, 55, 0, 14) << endl;
```

Binary search with iteration

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearchIt(int array[], int target, int left, int right)
{
    int result = -1;
    while (! (right < left))
    {
        int mid = (left + right) / 2;
        if (array[mid] == target)
        {
            result = mid;
            right = left - 1;  // kill the loop
        }
        else if (target < array[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    return result;
}
cout << binSearch(array, 55, 0, 14) << endl;
```

Binary search with iteration

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearchIt(int array[], int target, int left, int right)
{
    int result = -1;
    while (! (right < left))
    {
        int mid = (left + right) / 2;
        if (array[mid] == target)
        {
            result = mid;
            right = left - 1;  // kill the loop
        }
        else if (target < array[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    return result;
}
cout << binSearch(array, 55, 0, 14) << endl;
```

Binary search with iteration

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearchIt(int array[], int target, int left, int right)
{
    int result = -1;
    while (! (right < left))
    {
        int mid = (left + right) / 2;
        if (array[mid] == target)
        {
            result = mid;
            right = left - 1;  // kill the loop
        }
        else if (target < array[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    return result;
}
cout << binSearch(array, 55, 0, 14) << endl;
```

Binary search with iteration

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearchIt(int array[], int target, int left, int right)
{
    int result = -1;
    while (! (right < left))
    {
        int mid = (left + right) / 2;
        if (array[mid] == target)
        {
            result = mid;
            right = left - 1;  // kill the loop
        }
        else if (target < array[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    return result;
}
cout << binSearch(array, 55, 0, 14) << endl;
```

Binary search with iteration

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearchIt(int array[], int target, int left, int right)
{
    int result = -1;
    while (! (right < left))
    {
        int mid = (left + right) / 2;
        if (array[mid] == target)
        {
            result = mid;
            right = left - 1;  // kill the loop
        }
        else if (target < array[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    return result;
}
cout << binSearch(array, 55, 0, 14) << endl;
```

Recursion vs. Iteration

Although the iterative and recursive solutions to binary search in a sorted array achieve the same result with roughly the same number of steps, there is technically more overhead for function calls and returns than for simple loop repetition.

This difference is small, however. To repeat what your textbook says, generally if it is easier to conceptualize a problem as recursive, it should be coded as such.

Recursion vs. Iteration

Although the iterative and recursive solutions to binary search in a sorted array achieve the same result with roughly the same number of steps, there is technically more overhead for function calls and returns than for simple loop repetition.

This difference is small, however. To repeat what your textbook says, generally if it is easier to conceptualize a problem as recursive, it should be coded as such. With problems like Fibonacci numbers being exceptions.

Recursion and Induction

Recursion and induction are quite similar.

Recursion

Base case
  Calculate for some small value(s)

Reduction step
  Break the problem down in terms of itself (smaller versions) and then call this function to solve the smaller versions, assuming it will work.

Induction

Base case
  Prove for some small values(s)

Induction step
  Break a larger case down into smaller ones that we assume work (the Induction hypothesis)

Induction for proving correctness

We can use induction to establish the truth of a given statement over an infinite range (usually natural numbers):

First, prove the base case (usually n = 0 or n = 1)

Then, prove that

if any one statement in the sequence of statements is true (the inductive hypothesis)

the very next one (n + 1) must be true (the inductive step)

Induction for proving correctness

So we have:

1. **Base Case (BC):** Prove the theorem for the simplest case
2. **Inductive Hypothesis (IH):** Assume the theorem holds for some arbitrary element, e
3. **Inductive Step (IS):** Show that the theorem holds for the successor of e
4. **Conclusion:** Together, 1 – 3 imply the theorem holds for all possible cases (the minimal case and all successors)

A valid induction proof clearly shows all these steps.

Induction for proving correctness

If the “statement” you want to prove correct is represented as a recursive function, the inductive proof is relatively easy...

Just follow the code’s lead and apply induction:

Your base case(s)? Your code’s base cases.

How do you break down the inductive step? However your code breaks the problem down into smaller cases.

What do you assume? That the recursive calls just work (for smaller input sizes as parameters, which better be how your recursive code works, no?).

Induction for proving correctness

To put it another way, to extend induction to the task of proving correctness of a recursive algorithm, you must show that your algorithm satisfies the following:

- Verify that the base case is recognized and solved correctly.
- Verify that each recursive case makes progress toward the base case. That is, any new problems generated are smaller versions of the original problem.
- Verify that if all smaller problems are solved correctly, then the original problem is also solved correctly.

Induction for proving correctness

Consider the definition for factorial:

$$n! = \begin{cases} 1, & n = 0 \\ n * (n - 1)!, & \text{otherwise} \end{cases}$$

Translated into code, it looks like this (again, for the zillionth time):

Induction for proving correctness

```
// Precondition: n is int >= 0
// Postcondition: returns n!

int factorial(int n)
{
  if (n == 0)
    return 1;









  else
    return n * factorial(n - 1);
}
```

Induction for proving correctness

```
// Precondition: n is int >= 0
// Postcondition: returns n!

int factorial(int n)
{
  if (n == 0)
    return 1;









  else
    return n * factorial(n – 1);
}
```

```
Prove: factorial(n) = n!
```

Induction for proving correctness

```
// Precondition: n is int >= 0
// Postcondition: returns n!

int factorial(int n)
{
  if (n == 0)
    return 1;







  else
    return n * factorial(n – 1);
}
```

```
Prove: factorial(n) = n!

Base case: n = 0
Our code returns 1 when
n = 0, and 0! = 1 by
definition. ✓
```

Induction for proving correctness

```
// Precondition: n is int >= 0
// Postcondition: returns n!

int factorial(int n)
{
  if (n == 0)
    return 1;





  else
    return n * factorial(n – 1);
}
```

Prove: factorial(n) = n!

Base case: n = 0
Our code returns 1 when n = 0, and 0! = 1 by definition. ✓

Inductive hypothesis: assume that factorial(k - 1) = (k - 1)!

Induction for proving correctness

```
// Precondition: n is int >= 0
// Postcondition: returns n!

int factorial(int n)
{
  if (n == 0)
    return 1;





  else
    return n * factorial(n – 1);
}
```

Prove: factorial(n) = n!

Base case: n = 0
Our code returns 1 when n = 0, and 0! = 1 by definition. ✓

Inductive hypothesis: assume that factorial(k - 1) = (k - 1)!

Inductive step: Show that factorial(k) = k!

Induction for proving correctness

```
// Precondition: n is int >= 0
// Postcondition: returns n!

int factorial(int n)
{
  if (n == 0)
    return 1;









  else
    return n * factorial(n – 1);
}
```

```
Prove: factorial(n) = n!

Base case: n = 0
Our code returns 1 when
n = 0, and 0! = 1 by
definition. ✓

Inductive hypothesis: assume
that factorial(k - 1) =
(k - 1)!

Inductive step:  Show that
factorial(k) = k!

factorial(k - 1) = (k - 1)!
  (by IH)
```

Induction for proving correctness

```
// Precondition: n is int >= 0
// Postcondition: returns n!

int factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n – 1);
}
```

```
Prove: factorial(n) = n!

Base case: n = 0
Our code returns 1 when
n = 0, and 0! = 1 by
definition. ✓

Inductive hypothesis: assume
that factorial(k - 1) =
(k - 1)!

Inductive step:  Show that
factorial(k) = k!

factorial(k - 1) = (k - 1)!
  (by IH)
k! = k * (k - 1)! (by defn)
```

Induction for proving correctness

```
// Precondition: n is int >= 0
// Postcondition: returns n!

int factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n – 1);
}
```

```
Prove: factorial(n) = n!

Base case: n = 0
Our code returns 1 when
n = 0, and 0! = 1 by
definition. ✓

Inductive hypothesis: assume
that factorial(k - 1) =
(k - 1)!

Inductive step:  Show that
factorial(k) = k!

factorial(k - 1) = (k - 1)!
  (by IH)
k! = k * (k - 1)! (by defn)
For any k > 0, our code
returns k * factorial(k - 1)
  (look at the last line of
   the code)
```

Induction for proving correctness

```
// Precondition: n is int >= 0
// Postcondition: returns n!

int factorial(int n)
{
  if (n == 0)
    return 1;









  else
    return n * factorial(n – 1);
}
```

Always connect what the code does with what you want to prove.

```
Prove: factorial(n) = n!

Base case: n = 0
Our code returns 1 when
n = 0, and 0! = 1 by
definition. ✓

Inductive hypothesis: assume
that factorial(k - 1) =
(k - 1)!

Inductive step:  Show that
factorial(k) = k!

factorial(k - 1) = (k - 1)!
  (by IH)
k! = k * (k - 1)! (by defn)
For any k > 0, our code
returns k * factorial(k - 1)
  (look at the last line of
   the code) QED!!!
```

Induction for proving correctness

Can we use the same techniques for proving correctness of loops that we used for proving correctness of recursion?

Yes, we do this by stating and proving "invariants" – properties that are always true (i.e., they don't vary) at particular points in the program.

Induction for proving correctness

A loop invariant is:

A statement that is true at the beginning of a loop or at a given point) and remains true throughout the execution of the loop

It allows us to relate the state of the program (i.e. the data values) to the current iteration *i*

We can then use mathematical induction...

Induction for proving correctness

Proving a loop invariant:

Induction variable: number of times through the loop

Base case: prove the invariant true before the first loop guard (i.e., conditional) test

Induction hypothesis: Assume the invariant holds just before some (unspecified) iteration's loop guard test

Inductive step: Prove the invariant holds at the end of that iteration (just before the next loop guard test)

Extra bit: Make sure the loop will eventually end

Induction for proving correctness

```
// Precondition: n >= 0
// Postcondition: returns n!

int factIter(int n)
{
    int f = 1;



    for (int i = 1; i <= n; i++)
        f = f * i;
    return f;
}
```

```
What's the loop invariant?
```

Induction for proving correctness

```
// Precondition: n >= 0
// Postcondition: returns n!

int factIter(int n)
{
    int f = 1;



    for (int i = 1; i <= n; i++)
        f = f * i;
    return f;
}
```

```
Loop invariant: at the end
of the ith iteration, f = i!
i is the induction variable
```

Induction for proving correctness

```
// Precondition: n >= 0
// Postcondition: returns n!

int factIter(int n)
{
    int f = 1;



    for (int i = 1; i <= n; i++)
        f = f * i;
    return f;
}
```

```
Loop invariant: at the end
of the ith iteration, f = i!
i is the induction variable

Base case: 0! and 1! are
both 1 by defn
f = 1 before first "i <= n"
```

Induction for proving correctness

```
// Precondition: n >= 0
// Postcondition: returns n!

int factIter(int n)
{
    int f = 1;



    for (int i = 1; i <= n; i++)
        f = f * i;
    return f;
}
```

Loop invariant: at the end of the ith iteration, f = i!
i is the induction variable

Base case: 0! and 1! are both 1 by defn
f = 1 before first "i <= n"

Induction hypothesis: assume that just before "k <= n", f = (k - 1)! (OR assume for 0, 1, ..., k - 1 that f = (k - 1)!)

Induction for proving correctness

```
// Precondition: n >= 0
// Postcondition: returns n!

int factIter(int n)
{
    int f = 1;



    for (int i = 1; i <= n; i++)
        f = f * i;
    return f;
}
```

Loop invariant: at the end of the ith iteration, f = i!
i is the induction variable

Base case: 0! and 1! are both 1 by defn
f = 1 before first “i <= n”

Induction hypothesis: assume that just before “k <= n”, f = (k - 1)! (OR assume for 0, 1, ..., k - 1 that f = (k - 1)!)

Inductive step: during the i = kth iteration, the statement “f = f * i” executes, giving $f_{new} = f_{old} * k$

Induction for proving correctness

```
// Precondition: n >= 0
// Postcondition: returns n!

int factIter(int n)
{
    int f = 1;



    for (int i = 1; i <= n; i++)
        f = f * i;
    return f;
}
```

Induction hypothesis: assume that just before "k <= n", f = (k - 1)! (OR assume for 0, 1, ..., k - 1 that f = (k - 1)!)

Inductive step: during the i = kth iteration, the statement "f = f * i" executes, giving f_{new} = f_{old} * k

By the induction hypothesis, f_{old} = (k - 1)!, so the value of f_{new} is k * (k - 1)!

Induction for proving correctness

```
// Precondition: n >= 0
// Postcondition: returns n!

int factIter(int n)
{
    int f = 1;



    for (int i = 1; i <= n; i++)
        f = f * i;
    return f;
}
```

Induction hypothesis: assume that just before "k <= n", f = (k - 1)! (OR assume for 0, 1, ..., k - 1 that f = (k - 1)!)

Inductive step: during the i = kth iteration, the statement "f = f * i" executes, giving f_{new} = f_{old} * k

By the induction hypothesis, f_{old} = (k - 1)!, so the value of f_{new} is k * (k - 1)!

By definition, k * (k - 1)! is k!, so f_{new} = k!

Induction for proving correctness

```
// Precondition: n >= 0
// Postcondition: returns n!

int factIter(int n)
{
    int f = 1;



    for (int i = 1; i <= n; i++)
        f = f * i;
    return f;
}
```

Induction hypothesis: assume that just before “k <= n”, f = (k - 1)! (OR assume for 0, 1, ..., k - 1 that f = (k - 1)!)

Inductive step: during the i = kth iteration, the statement “f = f * i” executes, giving f_{new} = f_{old} * k

By the induction hypothesis, f_{old} = (k - 1)!, so the value of f_{new} is k * (k - 1)!

By definition, k * (k - 1)! is k!, so f_{new} = k! QED again!