Recursion

210CT 2015/16

Week C, Block 5

1 Introduction

- PDF notes for this block: recursion.pdf
- Nice video about algorithms: `https://www.youtube.com/watch?v=6hfOvs8pY1k`

1.1 Recap

- A recursive function calls itself
- It must have a condition to stop this process
  - This is the **base case**
  - The simple answer
  - Example: $n^0 = 1$ is always true, so no thinking needed
- Every other case should take us toward the base case
  - This is the **recursive case**
  - Example: $n^x = n^{x-1} \times n$ for all $x > 1$

1.2 Simple Example: Factorial

Remember: $5! = factorial(5) = 5 \times 4 \times 3 \times 2 \times 1$

```
FACTORIAL(n)
if(n < 2)
  return 1
else
  return FACTORIAL(n-1) * n
```

- When n is 1, the recursion stops

1.3 Iteration, Recursion and counting

- With iterative loops, we can easily 'see' how many times it will repeat
- How about with recursion?

```
FACTORIAL(n)
( 1)  if(n < 2)
( 1)    return 1
(  )  else
( 1)    return FACTORIAL(n-1) * n
```

- But we know this isn't $O(1)\ldots$
- Recursive algorithms repeat as a function of their input argument

2 Binary Search

2.1 Binary Search (interactive demo)

- `http://www.cs.armstrong.edu/liang/animation/web/BinarySearch.html`

2.2 Binary Search (Recursive)

```
BINARY_SEARCH(A, v, first, last)
if last < first                      //Base case 1
    return false
i ← first + (last - first)/2
if A[i] == v                         //Base case 2
    return true
if A[i] > v
    BINARY_SEARCH(A,v,first,i-1)     //Recursion case 1
else
    BINARY_SEARCH(A,v,i+1,last)      //Recursion case 2
```

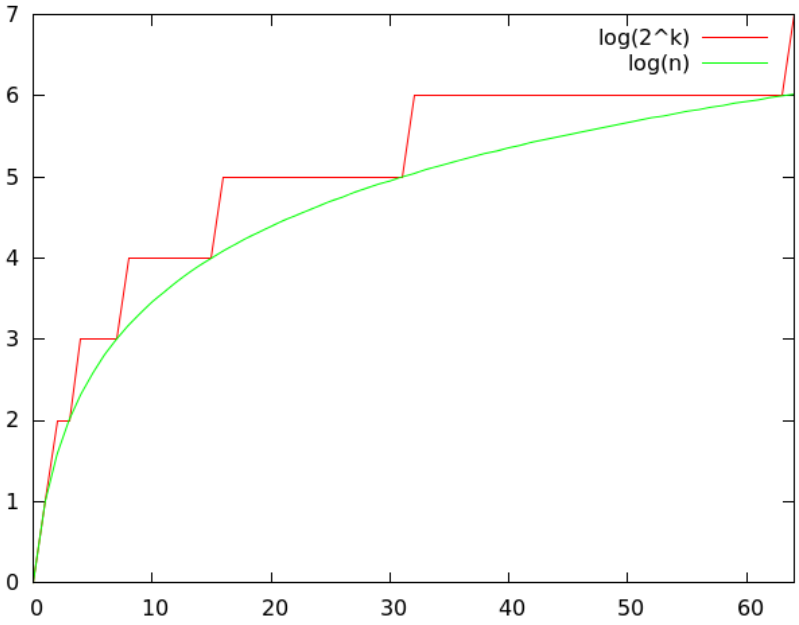
2.3 Binary Search (Iterative)

```
BINARY_SEARCH(A, v) //Find v in A
first ← 0
last ← length[A]-1
while last > first
```

```
    mid ← first + (last - first)/2
    if A[mid] = v
        return TRUE
    else if A[mid] < v
        first = mid + 1
    else
        last = mid - 1
return FALSE
```

2.4 Binary Search - run-time

- Recursive or iterative, the number of steps is the same.
- How do we calculate the run-time in big-O?
  - We can't just count because even in the iterative version, the loop doesn't run a fixed amount of time
- There are different ways to work this out
- This is possibly the simplest
- First let us work on only values of n that are powers of 2.
  - Is this OK?
  - Yes, because we are interested in the **upper bound** or **worst case**
  - So, if we have a value of $n = 44$, for example, we could use the next highest power of 2, $n = 64$ in our example, and know we will only ever be over the curve.
  - The plot below shows that the average shape is the same



- If we assume n is a power of 2, we can say $n = 2^k$ and know that there is always a value of k for all values of n
- Remember that 2^k means $2 \times 2 \times \ldots \times 2$, k times
- Or put another way, it's a number that you can divide by 2, k times and get 1.
- So, k is the magic number we need to find in terms of n because it's the number of steps in our binary search (which divides by two until there is one answer). We can't just say $O(k)$ without relating it to the input size, n.
- Another truth: we can divide n by 2, k times to get 1, because $n = 2^k$. So, $\frac{n}{2 \times 2 \times \ldots \times 2} = 1$ and the bottom of that is 2^k, so $\frac{n}{2^k} = 1$. Which we could get to by rearrangement from $n = 2^K$.
  - Proving this bit isn't necessary, I just wanted to point out that the two things are definitely the same
- So...

$$n = 2^k \tag{1}$$
$$\log_2(n) = \log_2(2^k) \tag{2}$$
$$\log_2(n) = k \log_2(2) \tag{3}$$
$$\log_2(n) = k \tag{4}$$

3 Merge Sort

- `http://www.sorting-algorithms.com/merge-sort`
- Animation on Wikipedia: `http://en.wikipedia.org/wiki/Merge_sort`

3.1 Pseudocode (Merge sort)

```
MERGE-SORT(l):
    if length(l) <= 1
        return l
    a←l[0 .. length(l)/2]
    b←l[length(l)/2 .. length(l)]
    a←MERGE-SORT(a)
    b←MERGE-SORT(b)
    return MERGE(a,b)
```

3.2 Pseudocode (Merge)

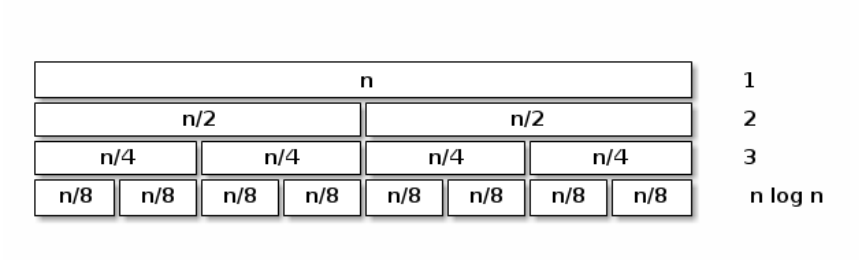
```
MERGE(a,b)
out ← array of length length(a)+length(b)
i←0
j←0
for x ← 0 length(out)
    if a[i] < b[j]
        out[x] = a[i]
        i++
    else
        out[x] = b[j]
        j++
```

3.3 Analysis

- The merge operation is clearly $O(n)$ (where n is the combined length of the two inputs) because it simply iterates once over them.
- But how many times is it called?
- The proof we used before to show the binary search was $O(\log n)$ still applies, but now it's used to show that we apply the merge $\log n$ times and so we have $O(n \log n)$

3.4 Thinking another way

- Draw the divisions in a tree (or imagine doing so)
- Each layer has n elements that must be processed
- There are $\log n$ layers to process
- So, process n items $\log n$ times



4 Towers of Hanoi

- `http://www.dynamicdrive.com/dynamicindex12/towerhanoi.htm`
- Fancy moving 65 disks?

4.1 Towers in pseudocode

```
MOVETOWER(disks, source, dest, spare)
if disk == 1
    move last from source to dest
else
    MoveTower(disks - 1, source, spare, dest)
```

```
move last disk from source to dest
MoveTower(disks - 1, spare, dest, source)
```

4.2 Towers algorithm

- Do we just have one item to move?
  - If so, move it
- If not
  - move all the disks except one to the spare
    - use the destination as a spare while doing so
  - move the remaining disk to the destination
  - move the stack from the spare to the destination
    - use the source as a space while doing so

4.3 Towers in code (C++)

- `http://ix.io/eW6`

```
[ 3 2 1 ]    [ ]    [ ]
[ 3 2 ]    [ ]    [ 1 ]
[ 3 ]    [ 2 ]    [ 1 ]
[ 3 ]    [ 2 1 ]    [ ]
[ ]    [ 2 1 ]    [ 3 ]
[ 1 ]    [ 2 ]    [ 3 ]
[ 1 ]    [ ]    [ 3 2 ]
[ ]    [ ]    [ 3 2 1 ]
```

4.4 Towers in code (python)

```
a=[3,2,1]
b=[]
c=[]

def display():
    print "%10s %10s %10s"%(a,b,c)
```

```
def solveHanoi(n, src, dst, spr):
    """Move n discs from src to dst, using the spare peg spr"""

    if n==1:
        dst.append(src[-1])
        del src[-1]
        display()

    else:
        solveHanoi(n-1,src,spr,dst)

        dst.append(src[-1])
        del src[-1]
        display()

        solveHanoi(n-1,spr,dst,src)

display()
solveHanoi(3,a,c,b)
```

```
[3, 2, 1]         []         []
   [3, 2]         []        [1]
      [3]        [2]        [1]
      [3]     [2, 1]         []
       []     [2, 1]        [3]
      [1]        [2]        [3]
      [1]         []     [3, 2]
       []         []  [3, 2, 1]
```

4.5 Run-time

- Moving 1 disk: 1 step
- $T(1) = 1$
  - Simply means time for base case of 1 disk, is a single unit of time, or single operation
- Moving 2:
  - $T(2) = \ldots$

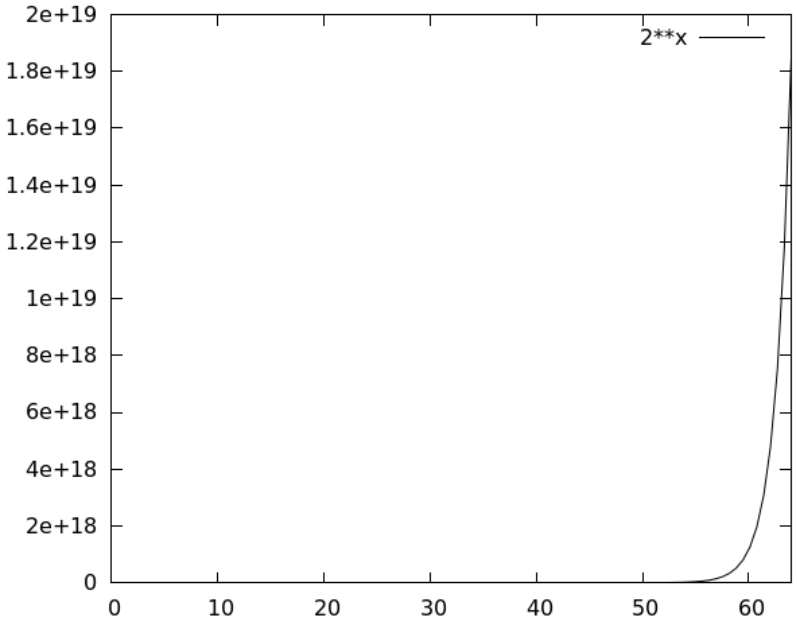
    * Looking back at the algorithm, to move more than 1, we recurse, move 1 and recurse again, so
    * $T(2) = T(1) + 1 + T(1)$
    * $T(2) = 1 + 1 + 1$

- Moving 3:
  - $T(3) = T(2) + 1 + T(2)$
  - $T(3) = T(1) + 1 + T(1) + 1 + T(1) + 1 + T(1)$

- Generally:
  - $T(1) = 1$
  - $T(n) = 2T(n-1) + 1$
  - $T(n-1) = 2T(n-2) + 1$
  - $T(n-2) = 2T(n-3) + 1$
  - So $T(n) = 2 \times 2 \times 2 \ldots T(1) + 1 + 1 + \ldots + 1$
    * Each layer adds another multiplication by two, constants can be ignored to give:
    * $O(2^n)$

Disks	Instructions
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
16	65536
32	4294967296
64	18446744073709551616

- If 2^{64} represents 1 millisecond for the operation, then it will take just over 2 million years to move 64 discs



5 Homework

5.1 Pre-homework

- Write a function that takes 4 numbers, called x1, y1, x2 and y2. These represent the x,y coordinates of two points. The function should return the distance between them using Pythagoras' theorem

5.2 Actual Homework

1. Use pseudocode to describe an algorithm which searches for values in a given range. It should fit the following description:
   - INPUT - An unsorted array A and two integers l and u
   - OUTPUT - TRUE if A contains an element which is both greater than l and less than u, FALSE otherwise
2. make a note of the algorithm's time complexity using O-notation
3. implement the function in the programming language of your choice

4. Now consider the problem when the array is sorted. You should be able to improve on the efficiency of the algorithm by using a divide and conquer approach. Write pseudocode and give the time complexity of the algorithm using Big O notation. If there is time available you should implement the algorithm with code, although this is not essential.

Emacs 23.3.1 (Org mode 8.0.3)