Java Notes Big O, Recursion, Priority Queues and Heaps

Joe Moore, 07508617059, joe.moore@warwick.ac.uk

Big O

The big O notation gives an upper bound on the growth rate of a function. The statement $f(n)$ is $O(g(n))$ means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$.

Given function $f(n)$, if $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$.

Asymptotic analysis

The asymptotic analysis of an algorithm determines the running time in big O notation.

To perform the asymptotic analysis:

- We find the worst-case number of primitive operations executed as a function of the input size
- We express this function with big O notation

For an example with array of elements n the big O of a function to find the largest element is $O(n)$, since each element must be visited once in a worst case scenario.

Whilst big O is a measure of the longest amount of time it could possibly take for the algorithm to complete. big Omega describes the best that can happen for a given data set. Big Theta is essentially saying that the function, $f(n)$ is bounded both from the top and bottom by the same function, $g(n)$.

Big O:
$f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically less than or equal to $g(n)$

Big-Omega:
$f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically greater than or equal to $g(n)$

Big Theta:
$f(n)$ is $\Theta(g(n))$ if $f(n)$ is asymptotically equal to $g(n)$

Recursion

Recursion: when a method calls itself.

A classic example of this is the factorial function,

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n$$

Recursive definition:

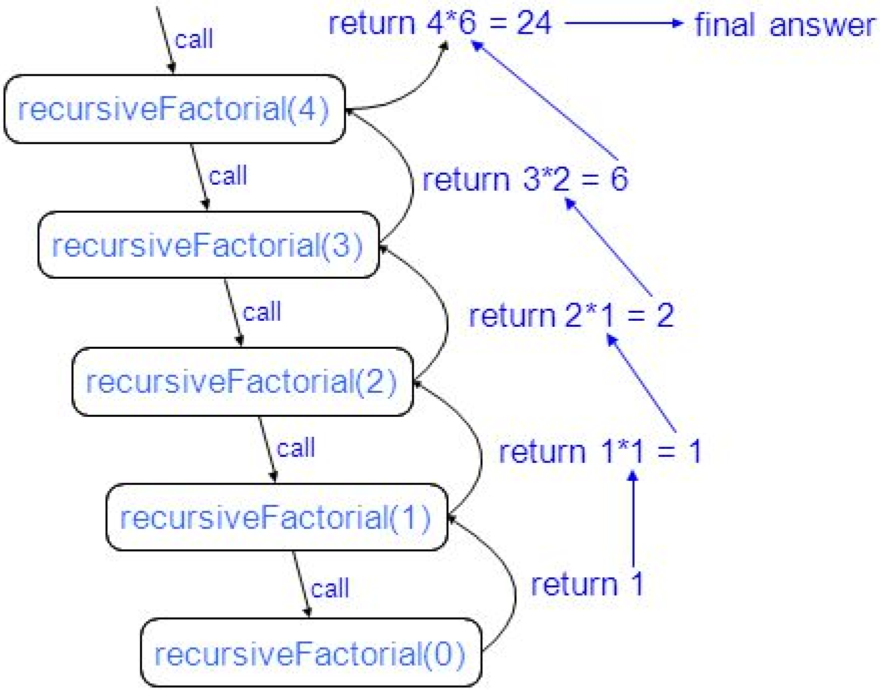
$$f(x) = \begin{cases} 1, & \text{if } n = 0 \\ n \cdot f(n-1), & \text{else} \end{cases}$$

This is seen in the Java program below:

```
public static int factorial(int n) throws IllegalArgumentException{
    if (n < 0)
        throw new IllegalArgumentException();
    else if (n == 0)
        return 1
    else
        return n * factorial(n-1);
}
```

Recursive procedures have 2 constituent parts, base cases and recursive calls. A base case is when the value of the inputs for which we perform no recursive calls. Every possible chain of recursive calls must eventually reach a base case. A recursive call is a call to the current method.

A recursive trace is a good way to visually represent a recursive call. In a recursive trace a box is used to represent each recursive call, with an arrow running from each caller to callee. And a separate arrow from each to caller showing return value. Example below:



Below is a binary search algorithm in Java that utilises recursion:

```
/**
* Returns true if the target value is found in the indicated
* portion of the data array. This search only considers the
* array portion from data[low] to data[high] inclusive.
*/
public static boolean binarySearch(int[ data, int target, int low, int high
    if (low > high)
        return false; // interval empty; no match
    else int mid = (low + high) / 2;
    if (target == data[mid])
        return true; // found a match
    else if (target < data[mid])
        return binarySearch(data, target, low, mid - 1);
        // recur left of the middle
    else
        return binarySearch(data, target, mid + 1, high);
        // recur right of the middle
```

After each iteration the remaining portion of the list is of size high - low + 1. After one comparison, this becomes one of the following:

$$(\text{mid} - 1) - \text{low} + 1 = \lfloor \frac{\text{low} + \text{high}}{2} \rfloor - \text{low} \leq \frac{\text{high} - \text{low} + 1}{2}$$

$$\text{high} - (\text{mid} + 1) + 1 = \text{high} - \lfloor \frac{\text{low} + \text{high}}{2} \rfloor \leq \frac{\text{high} - \text{low} + 1}{2}$$

Thus, each recursive call divides the search region in half; hence, there can be at most log n levels. So the algorithm runs in $O(log(n))$.

Reversing an array

```
if i < j then
    swap A[i] and A[j]
    reverseArray(A, i+1, j-1)
return
```

Computing Powers

The power function, p(x,n)=x^n, can be defined recursively:

$$p(x,n) = \begin{cases} 1, & \text{if } n = 0 \\ x \cdot p(x, n-1), & \text{else} \end{cases}$$

Giving us a power function that runs in $O(n)$ time (for we make n recursive calls) However, it is possible to do better...

Through effective use of repeated squaring we can derive a more efficient linearly recursive algorithm.

$$p(x,n) = \begin{cases} 1, & \text{if } n = 0 \\ x \cdot p(x, (n-1)/2)^2, & \text{else} \\ p(x, n/2)^2 & \text{if n} > 0 \text{ is even} \end{cases}$$

Heaps

A heap is a binary tree storing keys at its nodes and satisfying the following properties:
Heap-Order: for every internal node v other than the root, $key(v) \geq key(parent(v))$
Complete Binary Tree: let h be the height of the heap,

- for $i = 0, \ldots, h-1$ there are 2^i nodes of depth i

- at depth $h-1$, the internal nodes are to the left of the external nodes
- The last node of a heap the the rightmost node of maximum depth

Theorem: A heap storing n keys has height $O(\log n)$
Proof: (we apply the complete binary tree property

- Let h be the height of a heap storing n keys
- Since there are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \cdots + 2^{h-1} + 1$
- Thus, $n \geq 2^h$ i.e., $h \leq \log n$

We can use a heap to implement a priority queue. We store a (key, element) item at each internal node and keep track of the position of the last node.

Insertion into a heap

Method `insertItem` of the priority queue ADT corresponds to the insertion of a key `k` to the heap.

The algorithm consists of three steps.

- Find the insertion node `z` (the last new node)
- Store `k` at `z`
- Restore the heap order property

Upheap is what restores the heap order property. After insertion of a new key this property may be violated. upheap restores the heap order property by swapping k along an upward path from the insertion node. Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k.

Removal from a heap

Method `removeMin` of the priority queue ADT corresponds to the removal of the root key from the heap

The removal algorithm consists of three steps.

- Replace the root key with the key of the last node w
- Remove w

- Restore the heap order property.

Dowheap After replacing the root key with the key k of the last node, the heap-order property may be violated. Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root.

Heap-sort

Consider a priority queue with n items implemented by means of a heap.

- The space used in $O(n)$
- methods `insert` and `removeMin` take $O(\log n)$ time
- methods `size` , `isEmpty` , and `min` take time $O(1)$ time.

Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time The resulting algorithm is called heap-sort Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort.

Merging

We are given 2 heaps and a key *k* with the intention of merging the heaps and adding *k*. We create a new heap and store *k* at the root node. We then perform downheap to restore the heap-order property.