Addis Ababa University, Amist Kilo July 22, 2016

Algorithms and Programming for High Schoolers

Lecture 5

Recursion: worked examples

Ternary Search. The last example we're covering today is ternary search. Binary search is useful for the following problem: we are given a sorted list L[0] $\leq$ L[1] $\leq \ldots \leq$ L[n-1], and we would like to find out which i has L[i] == x for some input x. One way is with a `for` loop:

```
# L is sorted
def findX(L, x):
    for i in xrange(len(L)):
        if L[i] == x:
            return i
    return -1
```

However, in the worst case the above code takes $\Theta(n)$ time. It is much better to find x by binary search as in Lecture 3, where we check the middle element to see whether it is too small or too big, then recursively check the half where x might possibly lie.

What if L is not increasing, but rather is decreasing until some unknown point, then increasing? To find an x in such a list we could first find the position j where L switches from being decreasing to increasing, then binary search in L[:j] and L[j:] separately to try to find x. So, how do we find this position j where L switches behavior? To do this, we could use a an algorithm known as *ternary search*.

The idea behind ternary search is as follows. First, we have to assume that L never has the same value twice at two different positions for this algorithm to work. Now, for a `list` L of length n set `posA` $= n/3$ and `posB` $= 2n/3$. We look at L[posA] and L[posB]. There are two cases: either L[posA] ¡ L[posB], or the other way around. In the first case, it can be that both posA and posB are to the right of the switching point j, or posA is to the left of it and posB is to the right of it. However, if L[posA] ¡ L[posB], it *cannot* be the case that both are to the left of the switching point. Thus, we can eliminate L[posB:] from consideration. Similarly in the case L[posA] ¿ L[posB], it cannot be the case that both posA and posB are to the right of the switching point, so we can eliminate L[:posA+1]. In either case we eliminate 1/3rd of the possible entries and are thus left with only $2n/3$ possibilities. The running time is thus the smallest k such that $(2/3)^k \cdot n \leq 1$, so it is $\Theta(\log_{3/2} n) = \Theta(\log_2 n)$ (recall that $\log_a n = (1/\log_b a) \cdot \log_b n$).

```
# find the switching point from decreasing to increasing
# in the list L[from:to+1]
def recurse(L, from, to):
    if from == to:
        return from
    # n items remaining
    n = to - from + 1
    posA = from + n/3
    posB = from + 2*n/3
```

```
    if L[posA] < L[posB]:
        return recurse(L, from, posB - 1)
    else:
        return recurse(L, posA + 1, to)

# find the switching point from decreasing to increasing
# in the list L
def ternarySearch(L, x):
    return recurse(L, 0, len(L) - 1)
```

Nested brackets. Consider the following lists: [], [[]], [[[]]], [[[[]]]], etc. The first is the empty list, the second is a list containing the empty list, the third is a list containing a list that contains the empty list, etc. We say that the first list in this sequence has nesting level zero, and the second has nesting level 1, etc. Write a function which takes in the nesting level n and outputs the appropriate list.

Example solutions: With recursion:

```
def nest(n):
    if n == 0:
        return []
    return [nest(n-1)]
```

Iteratively:

```
def nest(n):
    ans = []
    for i in xrange(n):
        ans = [ans]
    return ans
```