Analyzing Recursive Algorithms

→ with a recursive algorithm, we will account for each operation that is performed based upon the particular activation of the function that manages the flow of control at the time it is executed.

→ ie, for each invocation of the function we only account for the # of operations that are performed within body of that activation

→ in general, we may rely on the intuition afforded by a recursion trace in recognizing how many recursive activations occur

Recursion Run Amok:

→ element uniqueness problem (return true if no duplicate elements)

```
def unique3(S, start, stop):
    if stop - start <= 1: return True                        // at most one item
    elif not unique(S, start, stop-1): return False          // 1st part has duplicate
    elif not unique(S, start+1, stop): return False          // 2nd part has duplicate
    else: return S[start] != S[stop-1]                        // do first and last differ?
```

→ terribly inefficient: each nonrecursive call uses $O(1)$ time, so the overall running time is proportional to total # of recursive invocations

→ let n denote the # of entries in consideration (ie, $n = stop - start$)

- if $n=1$, the running time is $O(1)$.
- in general case, the important observation is that a single call for a problem of size n may result in four calls with a range of size $n-2$, and thus eight calls with size $n-3$ and so on. Thus, in worst case, the total # of function calls is given by the geometric summation $1+2+4+\ldots+2^{n-1}$ which is equal to 2^n-1.
- thus, the running time is $O(2^n)$