Common Shapes of Call Trees

A review of recursive function complexity analysis

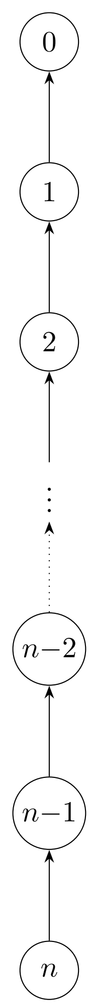
Here are the five common shapes of call trees. Of course, there are other kinds as well.

Example 1: Factorial

Pattern: Only one recursive call from each call; the runtime does not depend on n; n decreases by a constant amount (i.e., 1) with each call.

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n-1)
```

Analysis: $n+1$ calls in total. **Complexity:** $O(n)$.
Call Tree:

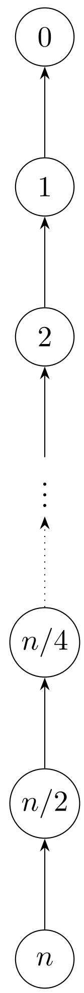


Example 2: Fast Exponentiation

Pattern: Only one recursive call from each call; the runtime does not depend on n; n decreases by a constant factor (e.g., a factor of 2) with each call.

```
def power(x, n):
    if n == 0:
        return 1
    if n == 1:
        return x
    half_power = power(x, n // 2)
    return half_power * half_power
```

Analysis: $\log_2(n) + 1$ calls in total. **Complexity:** $O(\log n)$.
Call Tree:



Example 3: Slow Power of 2

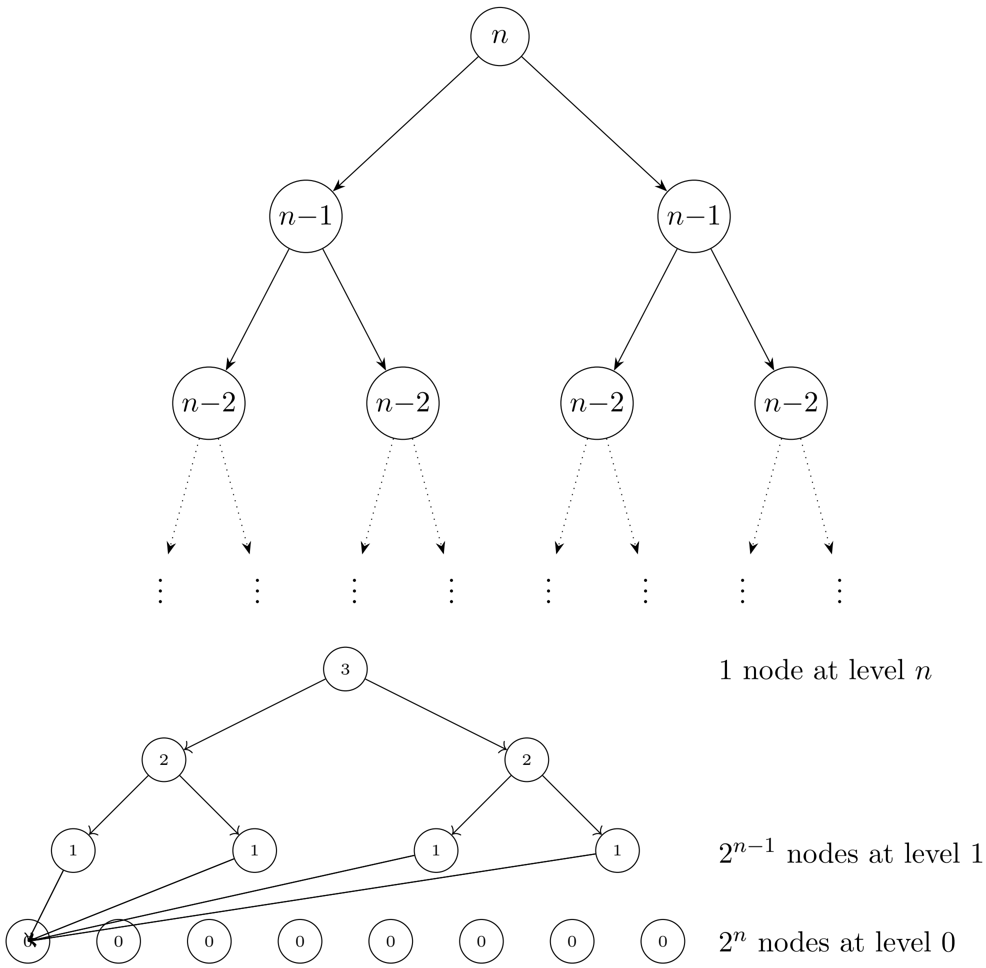
Pattern: Two recursive calls from each call; the runtime for each call does not depend on n; n decreases by 1 every time.

```
def slow_power_2(n):
```

```
    if n == 0:
        return 1
    return slow_power_2(n-1) + slow_power_2(n-1)
```

Analysis: $1 + 2 + 4 + \cdots + 2^n = \dfrac{1 - 2^{n+1}}{1 - 2} = 2^{n+1} - 1$ calls in total, i.e., $O(2^n)$ calls.

Call Tree:



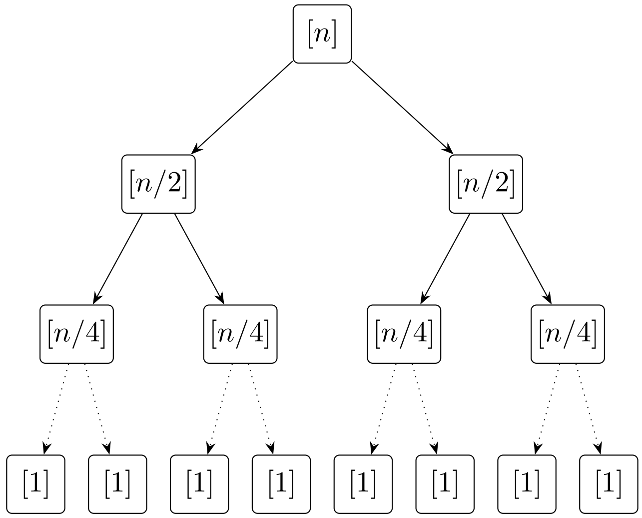
Example 4: sum_list2 (Divide and Conquer)

Pattern: Two recursive calls from each call; the runtime does not depend on n; n is smaller by a factor of 2 every time.

```
def sum_list2(L):
    '''Return the sum of the list of ints L'''
    if len(L) == 0:
        return 0
    if len(L) == 1:
        return L[0]  # the sum of the list is L[0] if L[0] is the only element

    mid = len(L) // 2  # the index of the approximate midpoint
    return sum_list2(L[:mid]) + sum_list2(L[mid:])
```

Analysis: Total number of calls: $1 + 2 + 4 + \cdots + 2^{\log_2 n} = \dfrac{1 - 2^{\log_2 n + 1}}{1 - 2} = 2n - 1 = O(n)$.

Call Tree:



Now, each call `sum_list2(L)` will take a time proportional to the length of L, so that, like in the analysis of MergeSort, the runtime here will be $O(n \log n)$. However, if we imagine that we get away with doing slicing for free, the runtime will be $O(n)$, since the number of calls is $O(n)$.

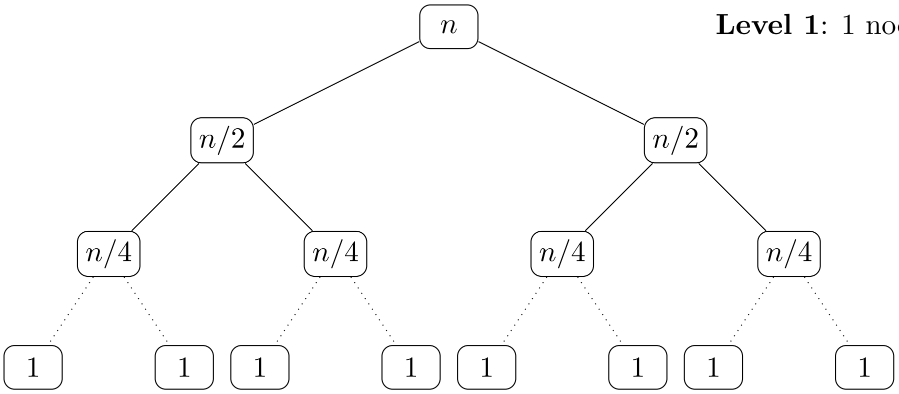
Here is how we could get away with not using slicing:

```python
def sum_list2_fast(L, start, end):
    if end == start:
        return 0
    if end == start + 1:
        return L[start]

    mid = (start + end) // 2
    return sum_list2_fast(L, start, mid) + sum_list2_fast(L, mid, end)

def sum_list2_fast_noargs(L):
    return sum_list2_fast(L, 0, len(L))
```

Example 5: MergeSort

Pattern: Two recursive calls from each call; each call does $O(n)$ work for the merge step; n is smaller by a factor of 2 every time.

Call Tree with Level Annotations:



Level 1: 1 node, work = $O(n)$

Level 2: 2 nodes, work = $O(n)$

Level 3: 4 nodes, work = $O(n)$

Level $\log_2 n + 1$: n nodes, w

Analysis:

- Number of levels: $\log_2 n + 1$
- Work at each level: $O(n)$ (summed across all nodes at that level)
- **Total Runtime:** $O(n \log n)$

Example	Recursive Calls	Size Reduction	Work/Call	Complexity
Factorial	1	$n-1$	$O(1)$	$O(n)$
Fast Exponentiation	1	$n/2$	$O(1)$	$O(\log n)$
Slow Power of 2	2	$n-1$	$O(1)$	$O(2^n)$
sum_list2	2	$n/2$	$O(1)^*$	$O(n)^*$
MergeSort	2	$n/2$	$O(n)$	$O(n \log n)$

*Assuming slicing is free; otherwise $O(n)$ per call leading to $O(n \log n)$ total.

Practice Problem: Complexity Analysis (2015)

Source: CSC 180 H1F Final Examination, December 2015, Question 6 [6 marks]

The left-hand column in the table below contains different pieces of code that work with list `L`, string `s` and integer `n`. In the right-hand column, give the asymptotic tight upper bound on the worst-case runtime complexity of each piece of code, using Big O notation.

Code	**Complexity**

<pre># L is a list of floats with n = len(L) 	
total = 0.0
for i in range(len(L)):
    if L[i] > 0.0:
        total += L[i]</pre>	

<pre># L is a list with n = len(L) 	
a = 5.0
for i in range(n):
    for j in range(i % 2):
        a += 1</pre>	

<pre># L is a list with n = len(L) 	
def f(L, i, j):
    if j-i <= 1:
        return L[i]
    if L[i] == 0:
        return f(L, i, i + (j-i)//2)
    else:
        return f(L, i + (j-i)//2, j)
 	
f(L, len(L)//5, len(L)//4)</pre>	

Practice Problem: Complexity Analysis (2016)

Source: CSC 180 H1F Final Examination, December 2016, Question 7 [8 marks]

The left-hand column in the table below contains different pieces of code that work with integer `n`. In the right-hand column, give the asymptotic tight upper bound on the worst-case runtime complexity of each piece of code, using Big O notation. Assume that arithmetic operations such as `+` and `**` take constant time.

<table>
<tr><th>Code</th><th>Complexity</th></tr>
<tr><td><pre>
total, i = 0.0, 0
for i in range(n):
    for j in range(i//2):
        total += i
</pre></td><td></td></tr>
<tr><td><pre>
i, j, sum = 1, 1, 0
while i < n**3:
    while j < n:
        sum = sum + i
        j += 1
    i += n
</pre></td><td></td></tr>
<tr><td><pre>
def f(n):
    if n == 0:
        return 1
    return f(n//2) + f(n//2)
if __name__ == "__main__":
    f(n)
</pre></td><td></td></tr>
<tr><td><pre>
def f(n):
    i, total = 0, 0.0
    while (i < n) and ((i % 10000) != 0):
        total += i
        i += 1
if __name__ == "__main__":
    f(n)
</pre></td><td></td></tr>
</table>

Practice Problem: Recursion Complexity

For each recursive function below, give the asymptotic tight upper bound on the worst-case runtime complexity using Big O notation.

Code	Complexity

<pre>def mystery(n):
    if n <= 1:
        return 1
    return mystery(n-1) + mystery(n-2)</pre>	

<pre>def f(n):
    if n <= 0:
        return 1
    return f(n-1) + f(n-1) + f(n-1)</pre>	

<pre>def g(n):
    if n == 0:
        return 0
    total = 0
    for i in range(n):
        total += i
    return total + g(n-1)</pre>	