Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```python
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

def fib_tree(n):
    if n == 1:
        return Tree(0)
    if n == 2:
        return Tree(1)
    left = fib_tree(n-2)
    right = fib_tree(n-1)
    return Tree(left.entry + right.entry, left, right)
```

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.

```python
def count_factors(n):
    factors = 0
    for k in range(1, n+1):
        if n % k == 0:
            factors += 1
    sqrt_n = sqrt(n)
    k, factors = 1, 0
    while k < sqrt_n:
        if n % k == 0:
            factors *= 2
            k += 1
        if k * k == n:
            factors += 1
    return factors
```

The Consumption of Space

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of active environments.

Values and frames in active environments consume memory.

Memory used for other values and frames can be reclaimed.

Active environments:

- Environments for any statements currently being executed
- Parent environments of functions named in active environments

Fibonacci Memory Consumption

Assume we have reached this step
**Fibonacci Memory Consumption**

- `fib(6)`
- `fib(4)`
- `fib(2)`
- `fib(1)`
- `fib(0)`

- Has an active environment
- Can be reclaimed
- Hasn’t yet been created

---

**Order of Growth**

A method for bounding the resources used by a function as the “size” of a problem increases.

- **$n$: size of the problem**
- **$R(n)$**: Measurement of some resource used (time or space)

$$R(n) = \Theta(f(n))$$

means that there are positive constants $k_1$ and $k_2$ such that

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

for sufficiently large values of $n$.

---

**Iteration vs Memoized Tree Recursion**

Iterative and memoized implementations are not the same.

**Implementation of the same functional abstraction can require different amounts of time.**

- **Memoized Implementation**
  ```python
  @memo
def fib(n):
      if n == 0:
          return 0
      if n == 1:
          return 1
      return fib(n-2) + fib(n-1)
  ```

---

**The Consumption of Time**

- **Implementation of Time**
  ```python
  def count_factors(n):
      factors = 0
      for k in range(1, n+1):
          if n % k == 0:
              factors += 1
      return factors
  ```

---

**Exponentiation**

**Goal**: one more multiplication lets us double the problem size.

- **Memoized Implementation**
  ```python
  def exp(b, n):
      if n == 0:
          return 1
      if n % 2 == 0:
          return b * exp(b, n-1)
      return b * exp(b, n-1)
  ```

- **Square Implementation**
  ```python
  def square(x):
      return x*x
  ```

- **Fast Exponentiation**
  ```python
  def fast_exp(b, n):
      if n == 0:
          return 1
      if n % 2 == 0:
          return square(fast_exp(b, n//2))
      else:
          return b * fast_exp(b, n-1)
  ```
Comparing orders of growth (n is the problem size)

| $\Theta(n^n)$ | Exponential growth! Recursive fib takes $\Theta(n^n)$ steps, where $\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828$ |
| $\Theta(n^6)$ | Incrementing the problem scales $R(n)$ by a factor. |
| $\Theta(n^2)$ | Quadratic growth. E.g., operations on all pairs. Incrementing $n$ increases $R(n)$ by the problem size $n$. |
| $\Theta(n)$ | Linear growth. Resources scale with the problem. |
| $\Theta(\sqrt{n})$ | |
| $\Theta(\log n)$ | Logarithmic growth. These processes scale well. Doubling the problem only increments $R(n)$. |
| $\Theta(1)$ | Constant. The problem size doesn't matter. |