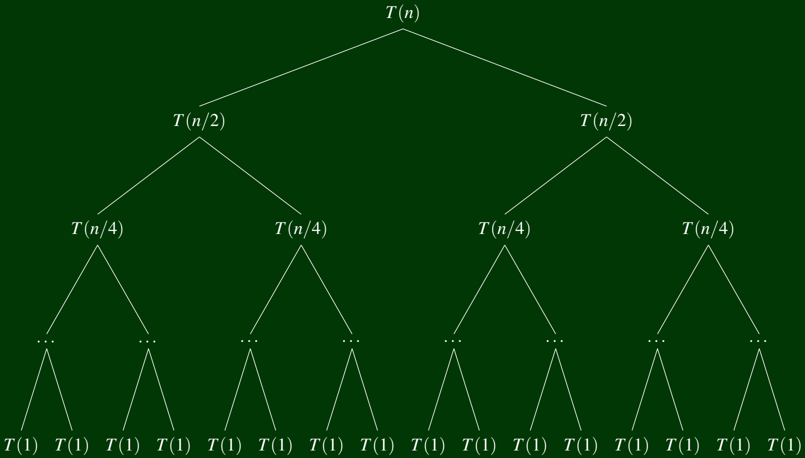
15
151

Mathematical Foundations for Computer Science

Algorithm Analysis



Outline

Merge

Merge

Purpose: Merge two sorted lists L_1 and L_2 into a single sorted list L.

```
@requires(is_sorted(L1))
@requires(is_sorted(L2))
@ensures(is_sorted(result))
def merge(L1, L2):
    L = []
    while len(L1) > 0 and len(L2) > 0:
        if L1[0] < L2[0]:
            L.append(L1[0])
            L1.remove(0)
        else:
            L.append(L2[0])
            L2.remove(0)
    return L + L1 + L2
```

How can we measure the performance of this code?

Measuring the Performance of Merge

```
def merge(L1, L2):
    L = []
    while len(L1) > 0 and len(L2) > 0:
        if L1[0] < L2[0]:
            L.append(L1[0])
            L1.remove(0)
        else:
            L.append(L2[0])
            L2.remove(0)
    return L + L1 + L2
```

Here's three potential measures of performance:

- number of comparisons
- number of array accesses
- amount of space used

Let's analyze each of these individually.

Measuring the Performance of Merge: Comparisons

```python
def merge(L1, L2):
    L = []
    # The loop runs len(L1) + len(L2) times at worst
    while len(L1) > 0 and len(L2) > 0:
        if L1[0] < L2[0]: # This is the only comparison
            L.append(L1[0])
            L1.remove(0)
        else:
            L.append(L2[0])
            L2.remove(0)
    return L + L1 + L2
```

A comparison happens any time we test the order of two elements in the input. How many comparisons are used in the above code?

How is this at all related to the counting we've been doing?

Let $C_{\text{merge}}(n,m)$ be the number of comparisons used in a call to `merge(L1, L2)` where $|L1| = n$ and $|L2| = m$.

Measuring the Performance of Merge: Comparisons

```
def merge(L1, L2):
    L = []
    # The loop runs len(L1) + len(L2) times at worst
    while len(L1) > 0 and len(L2) > 0:
        if L1[0] < L2[0]: # This is the only comparison
            L.append(L1[0])
            L1.remove(0)
        else:
            L.append(L2[0])
            L2.remove(0)
    return L + L1 + L2
```

Let $C_{\text{merge}}(n,m)$ be the number of comparisons used in a call to `merge(L1, L2)` where $|L1| = n$ and $|L2| = m$.

Another way of phrasing our above observation is that we're partitioning the comparisons based on which run of the loop they are in.

We know that there are at most $n+m$ runs of the loop, and during each one, we have exactly one comparison.

It follows that $C_{\text{merge}}(n,m) \leq n+m$.

Measuring the Performance of Merge: Array Accesses

```
def merge(L1, L2):
    L = []
    # The loop runs len(L1) + len(L2) times at worst
    while len(L1) > 0 and len(L2) > 0:
        if L1[0] < L2[0]:  # This is two array accesses
            L.append(L1[0]) # Here's another one
            L1.remove(0)
        else:
            L.append(L2[0]) # Here's another one
            L2.remove(0)
      return L + L1 + L2
```

Let $A_{\text{merge}}(n,m)$ be the number of array accesses used in a call to `merge(L1, L2)` where $|L1| = n$ and $|L2| = m$.

Again, we're partitioning based on the accesses in each iteration of the loop and after. We know that there are at most $n+m$ runs of the loop, and during each one, we have exactly two accesss.

It follows that $A_{\text{merge}}(n,m) \leq 3(n+m)$.

Measuring the Performance of Merge: Space

```
def merge(L1, L2):
    L = []
    # The loop runs len(L1) + len(L2) times at worst
    while len(L1) > 0 and len(L2) > 0:
        if L1[0] < L2[0]:
            L.append(L1[0])
            L1.remove(0)
        else:
            L.append(L2[0])
            L2.remove(0)
      return L + L1 + L2
```

Let $S_{\text{merge}}(n,m)$ be the number of auxiliary bytes used in a call to `merge(L1, L2)` where $|L1| = n$ and $|L2| = m$.
The only new space we use is for the copy of the list. This means it's exactly $n+m$ "elements" long. Supposing that each element is c bytes large, this means $S_{\text{merge}}(n,m) = c(n+m)$.

Putting It Together

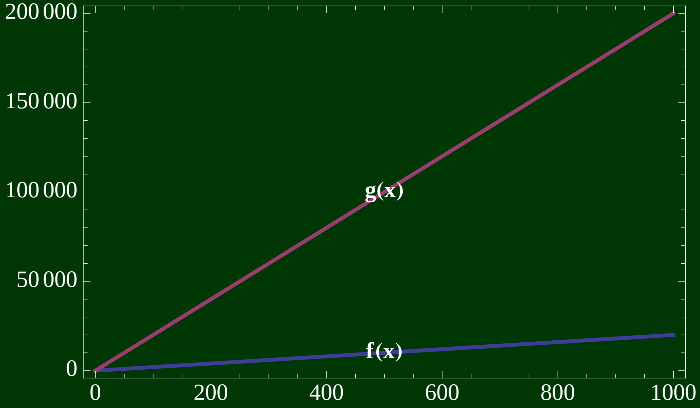
So, we've determined that the performance of `merge` based on these three factors is:

- $C_{\text{merge}}(n,m) \leq n+m$
- $A_{\text{merge}}(n,m) \leq 3(n+m)$
- $S_{\text{merge}}(n,m) = c(n+m)$

Somehow, these results are all "the same". As n and m grow large, the measures are all going to be "the same".
It turns out that the fact that these are all similar is a coincidence, but our definition of "similar" deserves more investigation.
What requirements do we need to consider two measures "similar"?

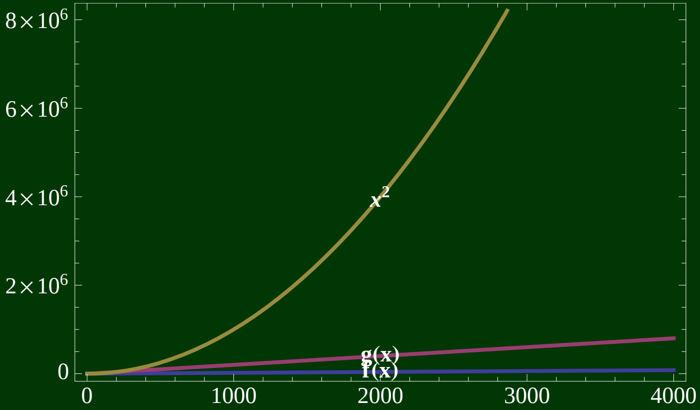
- For small inputs, we don't really care what happens.
- As the inputs get large, they shouldn't grow drastically apart.

Investigating with Pictures



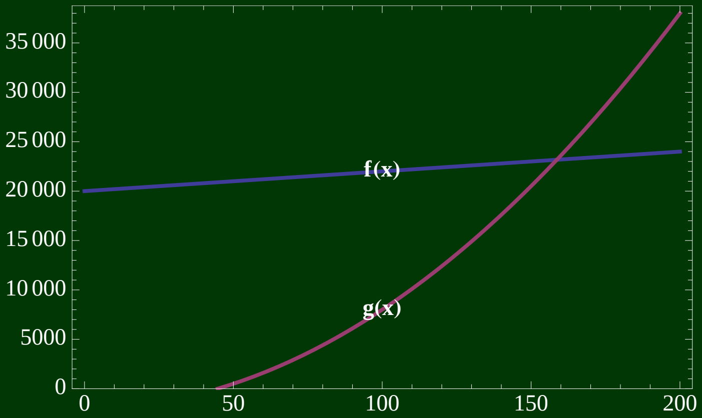
Should we consider these “the same”?

Investigating with Pictures



Probably a good idea, since they seem to be growing at the same rate. For reference, the function that dwarfs them both is x^2.

Investigating with Pictures 2



Here's two functions, $f(x)$ and $g(x)$. Ultimately, $g(x)$ will grow much faster than $f(x)$, but at the beginning, it is smaller.

Outline

Asymptotics

When we try to really get at when two functions have the same behavior, we're looking at asymptotics. Here's the formalizations of our intuitions:

Definition (Big-Oh)

We say a function $f : A \to B$ **is dominated by** a function $g : A \to B$ when:

$$\exists(c, n_0 > 0).\ \forall(n \geq n_0).\ f(n) \leq cg(n)$$

Formally, we write this as $f \in \mathscr{O}(g)$.

Again, back to our intuition: $\mathscr{O}$ notation strips away the small cases and constants. We can think of $\mathscr{O}$ as a sort of "upper bound".
There's a similar concept for "lower bound":

Definition (Big-Omega)

We say a function $f : A \to B$ **dominates** a function $g : A \to B$ when:

$$\exists(c, n_0 > 0).\ \forall(n \geq n_0).\ f(n) \geq cg(n)$$

Formally we write this as $f \in \Omega(g)$.

Asymptotics

Finally, we can construct our concept of "the same":

Definition (Big-Theta)

We say a function $f : A \to B$ **grows at the same rate as** a function $g : A \to B$ when:

$$f \in \mathscr{O}(g) \text{ and } f \in \Omega(g)$$

Formally we write this as $f \in \Theta(g)$.

Some "gotchas":

- $\mathscr{O}(f), \Omega(f)$, and $\Theta(f)$ are sets! This means we should treat them as such.
- If we know $f(n) \in \mathscr{O}(n)$, then it is also the case that $f(n) \in \mathscr{O}(n^2)$, and $f(n) \in \mathscr{O}(n^3)$, etc.
- Remember that small cases, really don't matter. As long as it's eventually an upper/lower bound, it fits the definition.
- The constants do not have to be the same for $\mathscr{O}$ and Ω to prove Θ. For instance, if we know for all $n \geq 1$ that: (1) $f \leq 2n$ and (2) $f \geq n$, then $f \in \Theta(n)$.

Moving Backwards

Here's the results of our analysis from before:

- $C_{\text{merge}}(n,m) \leq n+m$
- $A_{\text{merge}}(n,m) \leq 3(n+m)$
- $S_{\text{merge}}(n,m) = c(n+m)$

Rephrasing our results in terms of Asymptotics, we get:

- $C_{\text{merge}}(n,m) \in \mathscr{O}(n+m)$
- $A_{\text{merge}}(n,m) \in \mathscr{O}(n+m)$
- $S_{\text{merge}}(n,m) \in \Theta(n+m)$

Outline

1. Merging Sorted Lists
2. What is Asymptotics?
3. Merge Sort
4. Finding The Minimum

Merge Sort

Merge Sort

Purpose: Return the list L in sorted order.

```
def sort(L):
    if len(L) < 2:
        return L
    else:
        return merge(sort(FirstHalf(L)), sort(SecondHalf(L)))
```

Let's look at the same three metrics again:

- Let $C_{\text{mergesort}}(n)$ be the number of comparisons used in a call to `sort(L)` where $|L| = n$.
- Let $A_{\text{mergesort}}(n)$ be the number of array accesses used in a call to `sort(L)` where $|L| = n$.
- Let $S_{\text{mergesort}}(n)$ be the number of auxiliary bytes used in a call to `sort(L)` where $|L| = n$.

Merge Sort: Comparisons

```
def sort(L):
    if len(L) < 2:
        return L
    else:
        return merge(sort(FirstHalf(L)), sort(SecondHalf(L)))
```

Let $C_{\text{mergesort}}(n)$ be the number of comparisons used in a call to `sort(L)` where $|L| = n$.

If $n = 0$, $n = 1$, we have

$C_{\text{mergesort}}(n) = n$ (Note: We assume 1 comparison for $n = 1$ for convenience)

Otherwise, we have

$$C_{\text{mergesort}}(n) = C_{\text{mergesort}}\left(\frac{n}{2}\right) + C_{\text{mergesort}}\left(\frac{n}{2}\right) + C_{\text{merge}}\left(\frac{n}{2}, \frac{n}{2}\right)$$

We can justify this recurrence combinatorially: every time we call merge sort, we do the comparisons for the left half of the list, we do the comparisons for the right half, and we do the comparisons to merge.

Merge Sort: Dealing with A Recurrence

Let $C_{\text{mergesort}}(n)$ be the number of comparisons used in a call to `sort(L)` where $|L| = n$.

$$C_{\text{mergesort}}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ C_{\text{mergesort}}\left(\frac{n}{2}\right) + C_{\text{mergesort}}\left(\frac{n}{2}\right) + C_{\text{merge}}\left(\frac{n}{2}, \frac{n}{2}\right) & \text{otherwise} \end{cases}$$

Recall that, before, we proved that $C_{\text{merge}}(n,m) \le n + m$. So, we can simplify the recurrence:

$$C_{\text{mergesort}}(n) \le \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ 2C_{\text{mergesort}}\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}$$

Now, we'd like to solve this recurrence. One of the cleaner ways is to view the process as a tree.

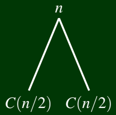
Merge Sort: Solving the Recurrence

$$C_{\mathsf{mergesort}}(n) \leq \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ 2C_{\mathsf{mergesort}}\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}$$

$C(n)$

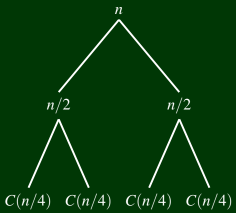
Merge Sort: Solving the Recurrence

$$C_{\text{mergesort}}(n) \le \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ 2C_{\text{mergesort}}\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}$$



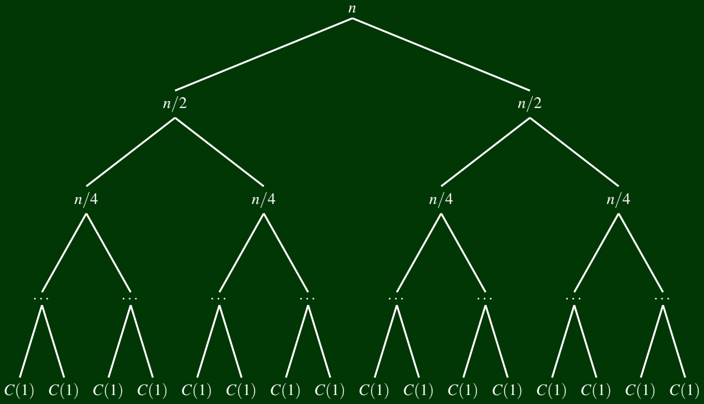
Merge Sort: Solving the Recurrence

$$C_{\text{mergesort}}(n) \le \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ 2C_{\text{mergesort}}\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}$$



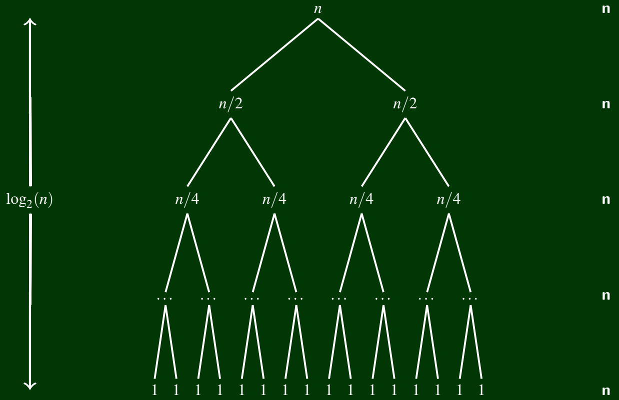
Merge Sort: Solving the Recurrence

$$C_{\text{mergesort}}(n) \leq \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ 2C_{\text{mergesort}}\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}$$



Merge Sort: Solving the Recurrence

$$C_{\text{mergesort}}(n) \leq \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ 2C_{\text{mergesort}}\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}$$



Since the recursion tree has height $\log_2(n)$ and each row does n work, it follows that $C_{\text{mergesort}} \in \mathcal{O}(n\log_2(n))$. But that's not a proof...

Merge Sort: Proving the Closed Form

To prove the closed form for the recurrence we found, we would do an induction proof.

It's straight-forward and boring; so, I'm going to skip it.

The same analysis we did for comparisons works for array accesses and auxiliary bytes as well.

Outline

1. Merging Sorted Lists
2. What is Asymptotics?
3. Merge Sort
4. Finding The Minimum

Finding The Minimum

Minimum Element

Purpose: Return the element of the list L that is smallest.

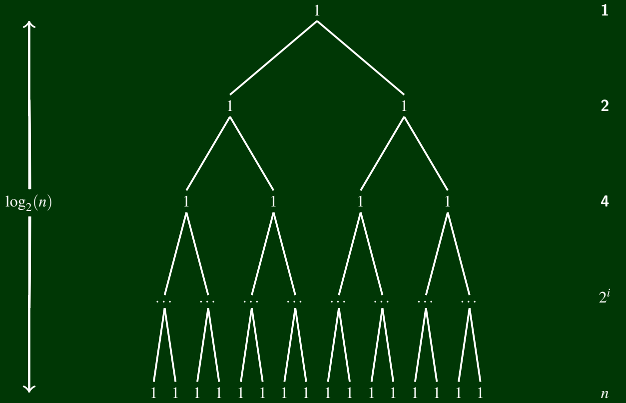
```
@requires(len(L) > 0)
def min(L):
    if len(L) == 1: return L[0]
    min1, min2 = min(FirstHalf(L)), min(SecondHalf(L))
    if min1 < min2: return min1
    else: return min2
```

Let $C_{\text{min}}(n)$ be the number of comparisons used in a call to `min(L)` where $|L| = n$. We note that if $n > 1$, then $C_{\text{min}}(n) = C_{\text{min}}\left(\frac{n}{2}\right) + C_{\text{min}}\left(\frac{n}{2}\right) + 1$. This is because the only three operations that use comparisons are calling min recursively on the left and right (each of size $n/2$), and doing the one comparison on their results. So, our recurrence is:

$$C_{\text{min}}(n) \leq \begin{cases} 1 & \text{if } n = 1 \\ 2C_{\text{min}}\left(\frac{n}{2}\right) + 1 & \text{otherwise} \end{cases}$$

Finding the Minimum: Solving the Recurrence

$$C_{\mathsf{min}}(n) \leq \begin{cases} 1 & \text{if } n = 1 \\ 2C_{\mathsf{min}}\left(\frac{n}{2}\right) + 1 & \text{otherwise} \end{cases}$$



That is, $\displaystyle\sum_{i=0}^{\log_2(n)} 2^i = \frac{1 - 2^{\log_2(n)+1}}{1-2} = 2(2^{\log_2(n)}) - 1 = 2n - 1 \in \mathcal{O}(n)$