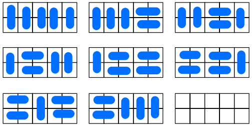
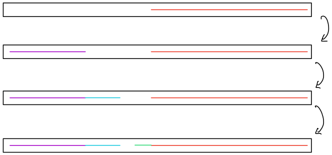
Last time...

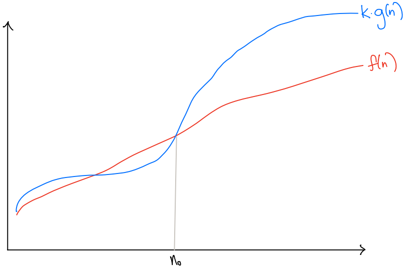
$$Fib(n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$





$$T_{BinSearch} = \Theta(\log(n))$$

Asymptotics review



Announcements

- HW2 solutions out.
- No class next week.
- Midterm the week after.

CSC 236 Lecture 6: Recurrences 2

Harry Sha

June 14, 2023

Today

Recurrences

Merge Sort

The Master Theorem

Recurrences

Last time we used induction to prove asymptotic bounds on recursive functions. For example, we showed that

$$\mathrm{Fib}(n) = \Theta(\varphi^n),$$

and

$$T_{\mathrm{BinSearch}}(n) = \Theta(\log(n)).$$

Last time's approach

Last time, the process looked like

- Guess an upper bound.
- Try to prove the upper bound.
- Try to prove a tighter upper bound or a matching lower bound.

Last time's approach

Last time, the process looked like

- Guess an upper bound.
- Try to prove the upper bound.
- Try to prove a tighter upper bound or a matching lower bound.

Here are two weaknesses to this approach.

- What if you get unlucky with your guess?
- The proofs were slow, technical and not incredibly intuitive.

Today's approach

Today we will see how to

1. Remove technical details.
2. Make better guesses.
3. Streamline the process for solving certain types of recurrences.

Technicalities

Technicalities - Base Cases

The base case typically involves calculating some values of the recursive function, and picking constants large enough so that things work out.

The base usually works out and is a little tedious to check, so for the rest of this class, you may skip this step - as long as you swear the following oath

> *I swear that I understand that a full proof by induction requires a base case*

Technicalities - Floors and Ceilings

In *divide and conquer* algorithms, we typically split up the problem in to subproblems of roughly even size. For example, we might split a problem of size n into 2 sub problems of size $n/2$. When n is not divisible by 2, this is really one subproblem of $\lfloor n/2 \rfloor$ and another of $\lceil n/2 \rceil$.

However, replacing $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ with $n/2$ has a negligible impact on the asymptotics so we can just ignore floors and ceilings. See *Introduction To Algorithms (CLRS)*, section 4.62 for a discussion on this.

The substitution method

The substitution method for solving recurrences is proof by (complete) induction with the simplifications applied.

The substitution method

1. Remove all the floors and ceilings from the recurrence T.
2. Make a guess for f such that $T(n) = O(f(n))$.
3. Write out the recurrence: $T(n) = \ldots.$
4. Whenever $T(k)$ appears on the RHS of the recurrence, *substitute* it with $cf(k)$.
5. Try to prove $T(n) \leq cf(n)$.
6. Pick c to make your analysis work!

The substitution method

- If you want to show $T = \Theta(f)$, you also need to show $T(n) = \Omega(f(n))$. This is the same as steps 3-6 where the $\leq$ in step 5 is replaced by a $\geq$.
- You can also add as many lower order terms as you want. I.e. you can show $T(n) = cf(n) + d$.
- The constant c that you pick when trying to show $T = \Omega(f)$ can be different to the constant that you picked when trying to show $T = O(f)$.

Today's approach

$$\boxed{T_{\mathrm{BinSearch}}(n) = T_{\mathrm{BinSearch}}(n/2) + 1}$$

Claim. $T_{\mathrm{BinSearch}}(n) = O(\log(n))$.

The Sorting Problem

Input. A list l.

Output. l, but sorted.

The Sorting Problem

Input. A list l.

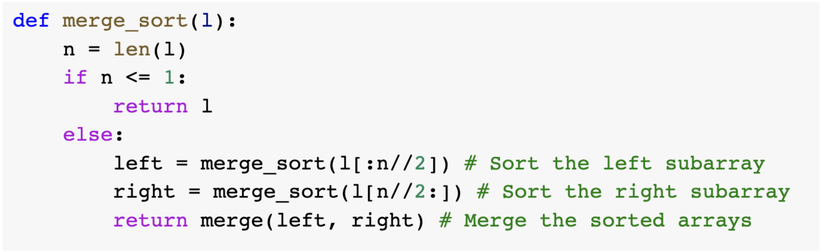
Output. l, but sorted.

Let's think of $l \in \mathrm{List}[\mathbb{N}]$, i.e. l is a list of natural numbers is sorted iff $i \leq j \implies l[i] \leq l[j]$.

In general, sorting makes sense for $l \in \mathrm{List}(A)$, as long as the elements of A can be ordered.

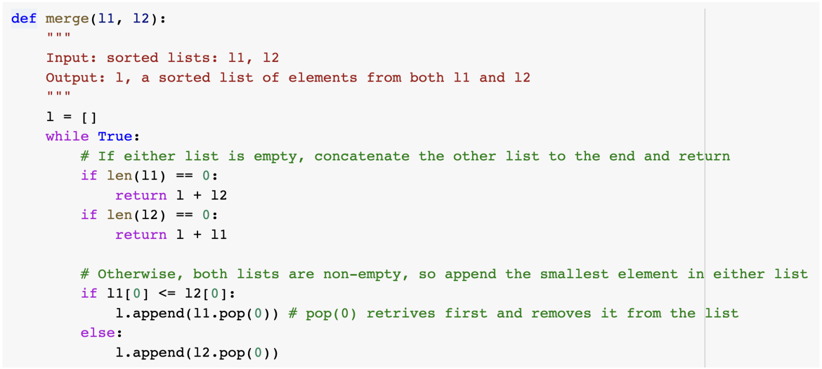
Merge Sort - Code

Screenshots of Code



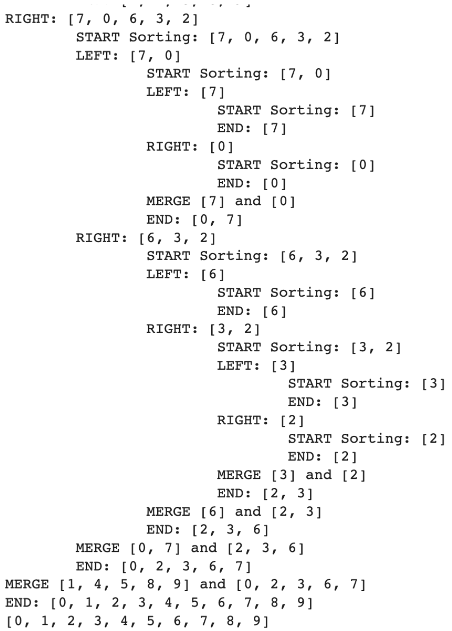
```
def merge_sort(l):
    n = len(l)
    if n <= 1:
        return l
    else:
        left = merge_sort(l[:n//2]) # Sort the left subarray
        right = merge_sort(l[n//2:]) # Sort the right subarray
        return merge(left, right) # Merge the sorted arrays
```

Screenshots of Code



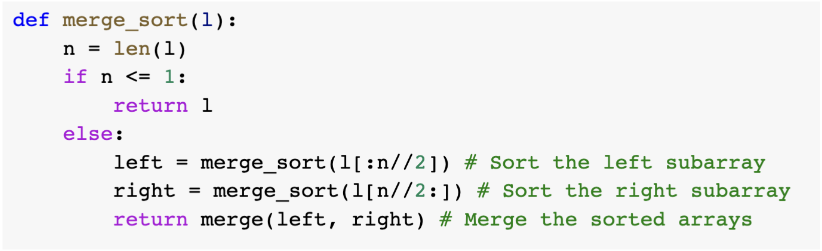
```python
def merge(l1, l2):
    """
    Input: sorted lists: l1, l2
    Output: l, a sorted list of elements from both l1 and l2
    """
    l = []
    while True:
        # If either list is empty, concatenate the other list to the end and return
        if len(l1) == 0:
            return l + l2
        if len(l2) == 0:
            return l + l1

        # Otherwise, both lists are non-empty, so append the smallest element in either list
        if l1[0] <= l2[0]:
            l.append(l1.pop(0)) # pop(0) retrives first and removes it from the list
        else:
            l.append(l2.pop(0))
```

Screenshots of Code



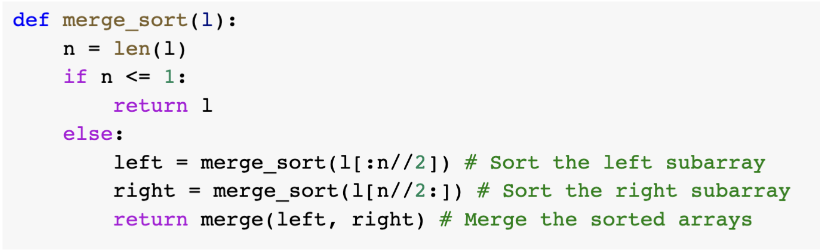
```
RIGHT: [7, 0, 6, 3, 2]
        START Sorting: [7, 0, 6, 3, 2]
        LEFT: [7, 0]
                START Sorting: [7, 0]
                LEFT: [7]
                        START Sorting: [7]
                        END: [7]
                RIGHT: [0]
                        START Sorting: [0]
                        END: [0]
                MERGE [7] and [0]
                END: [0, 7]
        RIGHT: [6, 3, 2]
                START Sorting: [6, 3, 2]
                LEFT: [6]
                        START Sorting: [6]
                        END: [6]
                RIGHT: [3, 2]
                        START Sorting: [3, 2]
                        LEFT: [3]
                                START Sorting: [3]
                                END: [3]
                        RIGHT: [2]
                                START Sorting: [2]
                                END: [2]
                        MERGE [3] and [2]
                        END: [2, 3]
                MERGE [6] and [2, 3]
                END: [2, 3, 6]
        MERGE [0, 7] and [2, 3, 6]
        END: [0, 2, 3, 6, 7]
MERGE [1, 4, 5, 8, 9] and [0, 2, 3, 6, 7]
END: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Merge Sort Complexity



```python
def merge_sort(l):
    n = len(l)
    if n <= 1:
        return l
    else:
        left = merge_sort(l[:n//2]) # Sort the left subarray
        right = merge_sort(l[n//2:]) # Sort the right subarray
        return merge(left, right) # Merge the sorted arrays
```

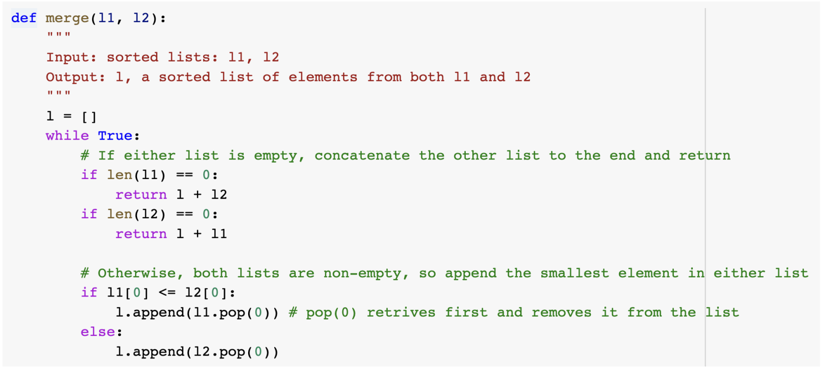
Merge Sort Complexity



```python
def merge_sort(l):
    n = len(l)
    if n <= 1:
        return l
    else:
        left = merge_sort(l[:n//2]) # Sort the left subarray
        right = merge_sort(l[n//2:]) # Sort the right subarray
        return merge(left, right) # Merge the sorted arrays
```

$$T_{MS}(n) = 2T_{MS}(n/2) + T_{\mathrm{Merge}}(n)$$

Merge Complexity



```python
def merge(l1, l2):
    """
    Input: sorted lists: l1, l2
    Output: l, a sorted list of elements from both l1 and l2
    """
    l = []
    while True:
        # If either list is empty, concatenate the other list to the end and return
        if len(l1) == 0:
            return l + l2
        if len(l2) == 0:
            return l + l1

        # Otherwise, both lists are non-empty, so append the smallest element in either list
        if l1[0] <= l2[0]:
            l.append(l1.pop(0)) # pop(0) retrives first and removes it from the list
        else:
            l.append(l2.pop(0))
```

Let n be the total number of elements in `l1` and `l2`, what is the complexity of `merge` in terms of n?

Merge Complexity

```python
def merge(l1, l2):
    """
    Input: sorted lists: l1, l2
    Output: l, a sorted list of elements from both l1 and l2
    """
    l = []
    while True:
        # If either list is empty, concatenate the other list to the end and return
        if len(l1) == 0:
            return l + l2
        if len(l2) == 0:
            return l + l1

        # Otherwise, both lists are non-empty, so append the smallest element in either list
        if l1[0] <= l2[0]:
            l.append(l1.pop(0)) # pop(0) retrives first and removes it from the list
        else:
            l.append(l2.pop(0))
```

Let n be the total number of elements in `l1` and `l2`, what is the complexity of `merge` in terms of n?

$\Theta(n)$. Explanation: each iteration of the while loop adds at least one element to the merged list.

Merge Sort Complexity

$$T_{MS}(n) = 2T_{MS}(n/2) + n$$

Merge Sort Complexity

$$T_{MS}(n) = 2T_{MS}(n/2) + n$$

Note that the $+n$ is really a $+\Theta(n)$, but since we care about asymptotics, this is another simplification that is ok!

Recurrences as Sums

$$\boxed{T_{MS}(n) = 2T_{MS}(n/2) + n}$$

$$\begin{aligned}
T_{MS}(n) &= 2T_{MS}(n/2) + n \\
&= 2(2T_{MS}(n/4) + n/2) + n \\
&= 4T_{MS}(n/4) + 2n \\
&= 4(2T_{MS}(n/8) + n/4) + 2n \\
&= 8T_{MS}(n/8) + 3n \\
&= 8(2T_{MS}(n/16) + n/8) + 3n \\
&= 16T_{MS}(n/16) + 4n \\
\ldots
\end{aligned}$$

Let's say $n = 2^k$ for some k. Then eventually, we get to...

$$T_{MS}(n) = 2^k T_{MS}(n/2^k) + kn = nT_{MS}(1) + kn = \Theta(n\log(n))$$

Recursion Trees

Recursion Trees are a great way to visualize the sum

Recursion Trees

$$\boxed{T_{MS}(n) = 2T_{MS}(n/2) + n}$$

Using recursion trees

Like in the previous example, we can sometimes use the recursion tree to compute the runtime directly.

Other times, we won't be able to compute the runtime directly, but we can still use recursion trees to make a good guess. We can then prove our guess was correct using the substitution method.

$$T(n) = T(n/3) + T(2n/3) + n$$

Standard Form Recurrences

A recurrence is in standard form if it is written as

$$T(n) = aT(n/b) + f(n)$$

For some constants $a \geq 1$, $b > 1$, and some function $f : \mathbb{N} \to \mathbb{R}$.

Most divide and conquer algorithms will have recurrences that look like this.

Thinking about the parameters

$$T(n) = aT(n/b) + f(n)$$

- a is the branching factor of the tree - how many children does each node have?
- b is the reduction factor - how much smaller is the subproblem in the next level of the tree compared to this level?
- $f(n)$ is the non-recursive work - how much work is done outside of the recursive call on inputs of size n? Again we make the assumption that f is positive and non-decreasing.

Recursion tree for standard form recurrences

Draw a recursion tree for the standard form recurrence. In terms of $a, b, f \ldots$

- What is the height of the tree?
- What is the number of vertices at height h?
- What is the subproblem size at height h?
- What is the total non-recursive work at level h?

Recursion tree for standard form recurrences

Recursion tree for standard form recurrences

Summary

- The height of the tree is $\log_b(n)$
- The number of vertices at level h is a^h
- The total non-recursive work done at level h is $a^h f(n/b^h)$. Of note are
  - Root work. $f(n)$
  - Leaf work. $a^{\log_b(n)} \cdot f(1) = \Theta(n^{\log_b(a)})$[1].
- Summing up the levels, the total amount of work done is

$$\sum_{h=0}^{\log_b(n)} a^h f(n/b^h).$$

[1] for calculation see slide 48

The Master Theorem

The Master Theorem is a way to solve most standard form recurrences quickly.

We get the Master Theorem by analyzing the recursion tree for a generic standard form recurrence.

The Master Theorem

Theorem (The Master Theorem)

Let $T(n) = aT(n/b) + f(n)$. Define the following cases based on how the root work compares with the leaf work.

1. *Leaf heavy. $f(n) = O(n^{\log_b(a)-\epsilon})$ for some constant $\epsilon > 0$.*
2. *Balanced. $f(n) = \Theta(n^{\log_b(a)})$*
3. *Root heavy. $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ for some constant $\epsilon > 0$, and $af(n/b) \leq cf(n)$ for some constant $c < 1$ for all sufficiently large n.*

Then,

$$T(n) = \begin{cases} \Theta(n^{\log_b(a)}) & \textit{Leaf heavy case} \\ \Theta(f(n)\log(n)) & \textit{Balanced case} \\ \Theta(f(n)) & \textit{Root heavy case} \end{cases}$$

ϵ

$f(n) = O(n^{\log_b(a)-\epsilon})$ for some $\epsilon > 0$ means that $f(n)$ is smaller than $n^{\log_b(a)}$ by a factor of at least n^{ϵ}. You might find it easier to think of ϵ as 0.0001, and $n^{\log_b(a)-\epsilon}$ as $\frac{n^{\log_b(a)}}{n^{\epsilon}}$.

For example,

$$n^{1.9} = O(n^{2-\epsilon})$$

for some $\epsilon > 0$ (e.g $\epsilon = 0.01$), but

$$n^2/\log(n) \neq O(n^{2-\epsilon})$$

for any $\epsilon > 0$ since $n^{2-\epsilon} = n^2/n^{\epsilon}$, and $\log(n) = O(n^{\epsilon})$ for any choice of $\epsilon > 0$.

Root heavy case additional regularity condition.

The condition in the root heavy case that $af(n/b) \leq cf(n)$ for some constant $c < 1$ for all sufficiently large n is called the regularity condition.

In the root heavy case, most of the work is done at the root. $af(n/b)$ is the total work done at level 1 of the tree. The regularity condition says if most of the work is done at the root, we better do more at the root than at level 1 of the tree!

What you need to know

I will now present the proof of the Master Theorem.

In this class, you only need to know how to apply the master theorem.

However, understanding the proof is incredibly helpful for getting an intuition for the case splits, remembering the conditions, and applying the theorem.

Here we go.

Proof Outline for Master Theorem

Analyze the recursion tree for the generic standard form recurrence. Apply the case splits to f.

Geometric Series

Before we prove the Master Theorem, let's remind ourselves about geometric series. A geometric series is a sum that looks like

$$S = a + ar + ar^2 + ...ar^{n-1} = \sum_{i=0}^{n-1} ar^i$$

I.e. each term in the sum is obtained by multiplying the previous term by r.

The closed-form solution for S is

$$S = a\left(\frac{r^n - 1}{r - 1}\right)$$

Proof

$$\boxed{S = a + ar + ar^2 + ...ar^{n-1}}$$

Balanced case

$$\boxed{f(n) = \Theta(n^{\log_b(a)})}$$

Leaf heavy case

$$\boxed{f(n) = O(n^{\log_b(a)-\epsilon})}$$

Root heavy case

The third case is similar to the previous cases. Check *CLRS* section 4.6.1 for the details.

Applying the Master Theorem

1. Write the recurrence in standard form to find the parameters a, b, f
2. Compare $n^{\log_b(a)}$ to f to determine the case split.
3. Read off the asymptotics from the relevant case.

Master Theorem applied to Merge Sort

$$T_{MS}(n) = 2\,T_{MS}(n/2) + n$$

Master Theorem applied to Merge Sort

$$T_{MS}(n) = 2\,T_{MS}(n/2) + n$$

Master Theorem applied to Binary Search

$$T_{\mathrm{BinSearch}}(n) = T_{\mathrm{BinSearch}}(n/2) + 1$$

Master Theorem applied to Binary Search

$$T_{\mathrm{BinSearch}}(n) = T_{\mathrm{BinSearch}}(n/2) + 1$$

Summary of Methods

Method	**Pros**	**Cons**
Induction	Always works, can get more precision	Requires a guess, can get technical, and proofs can get quite complex.
Substitution	Always works	Requires a guess and is slower than the below.
Recursion Tree	More Intuitive/Visual	Doesn't always work but is a good starting point and good for generating guesses.
Master Theorem	Proofs are super short	Restricted scope (recurrence must be in standard form and must fall into one of the cases).

Log calculation

$$\begin{aligned}
a^{\log_b(n)} &= a^{\frac{\log_a(n)}{\log_a(b)}} && \text{(Change of base)} \\
&= \left(a^{\log_a(n)}\right)^{1/\log_a(b)} \\
&= n^{1/\log_a(b)} \\
&= n^{\log_b(a)} && (1/\log_a(b) = \log_b(a))
\end{aligned}$$