Master Theorem

A >= 1, b > 1, k >= 0

(1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.

(2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.

(3) If $k > \log_b a$, then $T(n) = O(n^k)$.

Standard Divide and Conquer Form

$$T(n) = aT(n/b) + \Theta(n^k)$$

- ***a*** is the number of recursive calls
- ***b*** is the rate at which subproblem size decreases
- ***k*** represents the runtime of the non-recursive part of the algorithm
  - like max_crossing in max_seg_sum, k=1
  - merge in MergeSort, k=1

Correctness of Recursive Programs

Each program path is a path from the first line to the first return statement. Show that the precondition holds, and the recursive call runs on smaller data then the original call. Show the postcondition passes after the recursive call is made.

Some terminology

```
while E:
    S
```

```
while i < len(A):
    sum += A[i]
    i += 1
```

- **E** is called the **loop guard** (e.g., `i < len(A)`)
- **S** is called the **loop body** (one or more statements)
- A **loop invariant** gives a relationship between variables
  - it's a **predicate** with the variables being the parameters.
    - e.g., **Inv(i, sum): sum = \sum from A[1] to A[i]**
  - **Requirements** on loop invariant (otherwise it's not an invariant)
    - The invariant must hold prior to the first iteration (i.e., before entering the loop)
    - Assuming that the **invariant** and the **guard** are both true, the invariant must remain true after one **arbitrary** loop iteration

If we pick the right invariant, it will help proving the correctness of the program with the loop.

6

- **Base case**: Argue that the loop invariant is true when the loop is reached
- **Induction Step**:
  - assume that the invariant and guard are true at the end of an arbitrary iteration (**induction hypothesis P(n-1)**)
  - show that the invariant remains true after one iteration (**P(n)**)
- **Check postcondition**: Argue that the **invariant** and the **negation of the loop guard** together let us conclude the program's postcondition.

A loop variant must be reduced each iteration of the loop. The variant and the loop guard must show that the variant is always >= 0.

$Inv(x, y, total): 0 \le x \land total + x * y = a * b$

2. Induction Step

Case 1: x_0 is even

$x_1 = x_0/2$

$y_1 = y_0 * 2$

$total_1 = total_0$

$$\begin{aligned} & total_1 + x_1 * y_1 \\ = {} & total_0 + (x_0/2) * (y_0 * 2) \\ = {} & total_0 + x_0 * y_0 \quad = a * b \end{aligned}$$

```
def mult(a,b):
    '''
    Pre: a and b are natural numbers
    Post: returns a * b
    '''
    x = a
    y = b
    total = 0
    while x > 0:
        if x % 2 == 1:
            total = total + y
        x = x // 2
        y = y * 2
    return total
```

27

$Inv(x, y, total): 0 \le x \land total + x * y = a * b$

2. Induction Step

Case 2: x_0 is odd

$x_1 = (x_0 - 1)/2 \qquad y_1 = y_0 * 2$

$total_1 = total_0 + y_0$

$$\begin{aligned} & total_1 + x_1 * y_1 \\ = {} & (total_0 + y_0) + (x_0 - 1)/2 * y_0 * 2 \\ = {} & (total_0 + y_0) + (x_0 - 1) * y_0 \\ = {} & total_0 + y_0 + x_0 * y_0 - y_0 \\ = {} & total_0 + x_0 * y_0 \quad = a * b \end{aligned}$$

```
def mult(a,b):
    '''
    Pre: a and b are natural numbers
    Post: returns a * b
    '''
    x = a
    y = b
    total = 0
    while x > 0:
        if x % 2 == 1:
            total = total + y
        x = x // 2
        y = y * 2
    return total
```

28

Finding the variant:

- You usually want to identify which variable is decreasing each time.
- Make sure you account for all routes a loop iteration can take.
- Think of addition/subtraction/multiplicationof two terms, or len(list) kind of variants like those done in the lectures/tests/course_notes.
- Make sure by the end of the loop, your variant should be non-negative.
- The decrease amount can be 5,4,3,2,1,0 or like 10,8,6,2,0

Termination of Multiplication Algorithm

```
def mult(a,b):
    '''
    Pre: a and b are natural numbers
    Post: returns a * b
    '''
    x = a
    y = b
    total = 0
    while x > 0:
        if x % 2 == 1:
            total = total + y
        x = x // 2
        y = y * 2
    return total
```

The variant can be **x**:

- x decreases on each iteration
- by the loop guard implies that x > 0

So `mult` terminates.

Structural Induction

A single node is in T. If t_1 is in T, then the bigger tree with root r connected to the root of t_1 is in T. If t_1 and t_2 and t_3 are in T, then the bigger tree with root r connected to the roots of t_1 t_2 t_3 is in T. Height is H(t), nodes are V(t). Use structural to prove:
Predicate P(t): $V(t) <= (3^{H(t)+1} - 1) / 2$.
<u>Base Case:</u> Tree of a single node, height 0, nodes 1.
$1 <= (3^{0+1} - 1) / 2$: $1 <= (3 - 1) / 2$: $1 <= 2/2$: $1 <= 1$: HOLDS TRUE
<u>Induction Step:</u> 1. Assume $P(t_1)$: $V(t_1) <= (3^{H(t_1)+1} - 1) / 2$: Use: $V(t) = V(t_1) + 1$ and $H(t_1) + 1$ to get the following:
$V(t) <= [(3^{H(t_1)+1} - 1) / 2] + 1$: $V(t) <= [(3^{H(t)} - 1 + 2) / 2]$: Since $2 <= 3^{H(t)}$: $V(t) <= [(3^{H(t)+1} - 1) / 2]$ TRUE
2. Assume $P(t_1)$: $V(t_1)$, $P(t_2)$: $V(t_2)$, $P(t_3)$: $V(t_3)$, "Bigger Tree" n = max[$H(t_1)$, $H(t_2)$, $H(t_3)$], use H(t) and V(t) to get:
$V(t) <= V(t_1) + V(t_2) + V(t_3)$ sub the formula for V(): $V(t) <= 3 * [(3^{H(t)} - 1)/2] + 1$: $V(t) <= 3 * [(3^{n+1} - 1)/2] + 1$:
$(3^{n+1+1} - 3 + 2)/2$: $(3^{h(t)+1} - 1)/2$: Therefore holds true.

Big O (Upper Bound) $n >= N_0$, $f(n) <= cg(n)$ implies f(n) is in O(g(n)) Raise all the terms to the highest order
Big Omega (Lower Bound) $n >= N_0$, $f(n) >= cg(n)$ implies f(n) is in Omega(g(n)) It's always in Omega(1) if you're desperate
Big Theta (Tight Bound) Prove the function is in both O and Omega of g(n)

Finding Closed Form:
- Substitute multiple times (k)
- Find a formula in terms of k and n
- Solve k in the recursive call for the base case T(...) <= for n = 1 if 1 is base case
- Put this k value into formula to get closed form
- Prove with induction (if required)

Languages and DFAs and NFAs:
How to determine if a language is regular? If you can express if you can make a DFA or NFA from the language. Meaning it has a finite amount of states.
DFAs have Starting States, Accepting States, and Stranded States. Each state must have all possible inputs of a language leading to another state (be it stranded or not).
NFAs have Starting States, and Accepting States, and NFAs have epsilon transitions, and the possibility that one value can take you from one state to multiple different states.
Proving Minimal Number of States:
1. Write out all the possible strings for the language (with the original number of state).
2. Assume that you have one less state than the amount in the original diagram.
3. Compare all your strings with each other (compare 2 at a time), and see if they are accepted by the language, (you can add items to each string in order to make it match).
4. For each comparison, only 1 of two things should be accepted.
5. For each string that is NEVER accepted, write down that it needs its own state.
6. By the end you should have one extra string that doesn't have a state, in which case you can say that the assumed number of state (original – 1) is not enough, and you need the original amount

$Q_0 : \{q_0\} \xrightarrow{\epsilon} \{q_0, q_3\}$ (initial state of the DFA)
$\{q_0, q_3\} \xrightarrow{0} \{q_1\}$ (new state!)
$\{q_0, q_3\} \xrightarrow{1} \{q_0\} \xrightarrow{\epsilon} \{q_0, q_3\}$
$\{q_1\} \xrightarrow{0} \{q_2\} \xrightarrow{\epsilon} \{q_1, q_2\}$ (new state!)
$\{q_1\} \xrightarrow{1} \emptyset$ (new state!)
$\{q_1, q_2\} \xrightarrow{0} \{q_2\} \xrightarrow{\epsilon} \{q_1, q_2\}$
$\{q_1, q_2\} \xrightarrow{1} \{q_3\}$ (new state!)
$\{q_3\} \xrightarrow{0} \emptyset$
$\{q_3\} \xrightarrow{1} \emptyset$
No more new states. Done.

old state	symbol	new state
q0	0	q1
q0	1	q0
q0	ε	q3
q1	0	q2
q2	ε	q1
q2	1	q3

How many states in the DFA?
- 5
- {q0, q3}, {q1}, {q1, q2}, {q3}, ∅

Which state is the initial state?
- Q0: {q0, q3}

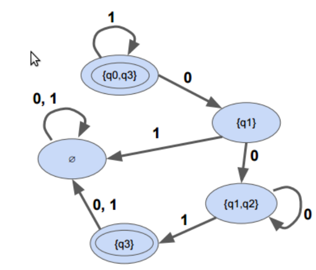
Which states are accepting states?
- any state that contains q3
- {q0, q3} and {q3}.

2

The resulting DFA from subset construction

old state	symbol	new state
{q0, q3}	0	{q1}
{q0, q3}	1	{q0, q3}
{q1}	0	{q1, q2}
{q1}	1	∅
{q1, q2}	0	{q1, q2}
{q1, q2}	1	{q3}
{q3}	0	∅
{q3}	1	∅
∅	0	∅
∅	1	∅

Initial state: {q0, q3}
Accepting states: {q0, q3}, {q3}



Inspiration: