# Contents



# 1 Analyzing recursive algorithms

```
def sum(l:List):
    if len(l) == 1:
        return l[0]
    else:
        return l[0] + sum(l[1:])
```

I want to analyze this algo, what do?

- Count operations to reduce problem size
- Count operations to solve sub-problem

## 1.1 Recurrence Relation

$$
\begin{aligned}
&A(n) :: \text{Number of additions required to sum a list of length n} \\
&A(1) = 0 - - - \text{Base Case} \\
&A(n) = 1 + A(n-1) \\
&A(2) = 1 + A(1) = 1 \\
&A(3) = 1 + A(2) = 2 \\
&A(n) = n - 1
\end{aligned} \tag{1}
$$

$$
\begin{aligned}
A(n) &= 1 + A(n-1) \\
&= 1 + (1 + A(n-2)) = 2 + A(n-2) \\
&= 2 + (1 + A(n-3)) = 3 + A(n-3) \\
&... \\
&= n - 1 + A(1) :: A(1) = 0 \\
&= n - 1
\end{aligned} \tag{2}
$$

```
# For the sake of complexity, log(len(l), 2) is a whole number
def min(l:List):
    if len(l) == 1:
        return l[0]
    m1 = min(l[:n/2])
    m2 = min(l[n/2:])
    if m1 < m2:
        return m1
    return m2
```

C(n) = # of comparisons to find the min of a list of size n C(0) = 0 C(n) = 1 + C(n/2) + C(n/2) = 2C(n/2) + 1

## 2 Master Theorem

**Insert Master Theorem Here**

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&<= cn/2 \log(n/2) + n \\
&= cn - cn \log 2 + n \\
&= cn - cn + n \\
&<= cn
\end{aligned} \tag{3}
$$

Recurrence of the form: $ T(n)=aT(n/b)+O(f(n)) $

$$
\begin{aligned}
&\textsf{Case:} f(n) = O(...) T(n) = O(...) \\
&n \log_b an \log_b a \\
&n \log_b an \log_b a \\
&n \log_b a + andc, af(n/b)cf(n)f(n)
\end{aligned} \tag{4}
$$

March 1, 2019

# 3 Divide & Conquer

- Break the problem down and solve smaller problems
- Combine small solutions to form the bigger solution
- Ask yourself: Is solving problems and combining better?
  - Recursion is not free.
- Do keep in mind: Getting a solution is better than no solution. As long as it's "fast enough".

# 4 Greedy Solutions

- Dr. Eloe: "This is the YOLO approach to algorithms, don't look back."
- Short-term gain over long term benefit.
- Build the solution one step at a time.
- At each step, choose the most beneficial choice.

  **Locally optimal** For that specific step.

## 4.1 Example - Real life

You've got to head to GS after Colden Hall, but it's cold as **f u c k**. So let's building hop in order to get some warmth. With each step, let's choose the closest building.

1. Union
2. Admin Building
3. GS

### 4.2 Example - Algorithm

**Coin changing** Given a list of denominations, what is the smallest # of coins I can dispense for a given amount of change?

denoms = [50, 25, 10, 5, 1]
A = 70
Start with 50, then 2 dimes.

```
# Denominations is always sorted, and you always have a penny
def giveChange (A:int):
    d = 0 # index of our current coin.
    while A > 0:
        c = A // denoms[d] # integer division in python 3
        print(c, "@", denoms[d])
        A -= c*denoms[d]
        d += 1
```

- O(n) on average, since you may have to check every single denomination

A = 18 gives 10, 5, 3x1s

- However if you have
  denoms = [10, 6, 1] and
  A = 12, you get back 3 coins, the optimal answer is 2

Greedy Heuristic It's an algorithm applied to optimization where you may not get the most optimal answer, but you'll still get an *okay* answer.

- The question is, is it worth it to spend all this time to get the most optimal solution or something just "good enough". And in certain cases, the optimal solution is very impractical.