**Ruolin Li**
**31764160**

# Divide and Conquer Recurrences Completed

Here we'll wrap up divide and conquer recurrences, and cover the Master Theorem. But first some handy math.

**Geometric Sums.** Here are useful formulas, when summing up runtimes over levels of a recurrence tree. A *geometric sum* has the form $\sum_{i=0}^{n} x^i$, where $x > 0$. Note that when $x = 1$, $\sum_{i=0}^{n} x^i = n$. When $x \neq 1$ we have that:

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1} = \frac{1 - x^{n+1}}{1 - x}.$$

**Now, for $x < 1$, take the limit as $n \to \infty$ to get an expression for the infinite sum:**

$$\sum_{i=0}^{\infty} x^i =$$

1. Section 5.5 of the text describes an integer multiplication algorithm, where the inputs are $n$-bit numbers. The runtime of this algorithm is given by the recurrence:

$$T(n) = \begin{cases} c, & \text{if } n = 1, \\ 3T(n/2) + cn, & \text{otherwise.} \end{cases}$$

**Draw out and label enough of the recurrence tree, so that you can express $T(n)$ as a sum of the total work per level.**

2. **Apply the formula for geometric sums to simplify your expression for $T(n)$.**

3. More generally, suppose that the recursive part of a recurrence for some runtime $T(n)$ has the form:

$$T(n) = aT(n/b) + cn^k \text{ for } n \geq n_0, \text{ where } a > 0, b > 1, c > 0, k > 0 \text{ are constants.}$$

**What is the total work done at level 2 of the recurrence tree?**

**What is the total work done at level $i$?**

**Write a sum for the total runtime.**
Note that the tree has $\log_b n$ levels.

4. Looking back over the runtime recurrences for problems you've seen so far, **give examples where the recurrence has the form shown in part 3, and also satisfies the following constraints. Write down what $a, b$, and $k$ are for these examples.**

   - $a > b^k$:

   - $a = b^k$:

   - $a < b^k$:

5. Here is a statement of the **Master Theorem**. Suppose that $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$ satisfies

$$T(n) = \begin{cases} c, & \text{for } n < n_0, \\ aT(n/b) + cn^k, & \text{for } n \geq n_0, \end{cases}$$

   where $a > 0$, $b > 1$, $c > 0$, and $k > 0$ are constants.

   - If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$.
   - If $a = b^k$, then $T(n) = \Theta(n^k \log n)$.
   - If $a < b^k$, then $T(n) = \Theta(n^k)$.

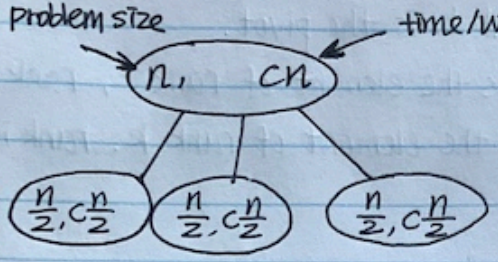
   **Go back and verify that the results we obtained using recurrence trees for the problems you list in part 4 match those provided by the Master Theorem.**

**The master theorem is used to analyze runtime of the algorithm, which has an instance of size n and can be solved by recursively break it into one or more subproblems of size n/b(b > 1). It is not useful when the subproblems have different sizes or the subproblems have size (n-b) rather than n/b.**

Note: The Master Theorem is handy for analyzing runtime of algorithms where an instance of size $n$ is solved by recursively solving one or more subproblems of size $n/b$ for some $b > 1$ (plus some extra work to break the problem down and/or piece the solutions of subproblems back together). It is not as useful when the subproblems have different sizes, or if subproblems have size $n - b$ rather than $n/b$.

# Divide and conquer (Cont)

1. level 0




the total work / level

$cn$

$\frac{3}{2}cn$

$\vdots$

$(\frac{3}{2})^i cn$

So the total work is $\sum_{i=0}^{\log n-1}(\frac{3}{2})^i cn$ we start at level 0.

$$T(n)=\begin{cases} c, & n=1 \\ 3T(n/2)+cn, & \text{otherwise} \end{cases}$$

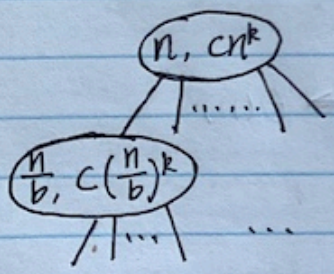
2. Geometric sum in question 1, $\sum_{i=0}^{\log_2 n-1}(\frac{3}{2})^i cn = \frac{(\frac{3}{2})^{\log_2 n}-1}{\frac{3}{2}-1} = \Theta((\frac{3}{2})^{\log_2 n})$

ignore constant.

transform $\frac{3}{2}$ to $2^{\log_2(\frac{3}{2})}$

$\Theta((\frac{3}{2})^{\log_2 n}) = \Theta((2^{\log_2(\frac{3}{2})})^{\log_2 n}) = \Theta((2^{\log_2 n})^{\log_2 3/2}) = \Theta(n^{\log_2 \frac{3}{2}})$

3.




a children

$cn^k$

$a\cdot(\frac{n}{b})^k\cdot c \quad (\frac{a}{b^k})cn^k$

level 2

$a^2(\frac{n}{b^2})^{2k}\cdot c \quad (\frac{a}{b^k})^2\cdot c\cdot n^k$

level i

$\left(\sum_{i=0}^{\log_b n-1}(\frac{a}{b^k})^i\right)cn^k \quad (\frac{a}{b^k})^i cn^k$

4. $\left(\sum_{i=0}^{\log_b n-1}(\frac{a}{b^k})^i\right)cn^k$

- $a>b^k$: work grows as levels increase $T(n)=3T(n/2)+cn$
- $a=b^k$: Quicksort (with perfect pivot), mergesort $T(n)=2T(n/2)+cn$
- $a<b^k$: Quickselect $T(n)=T(\frac{3n}{4})+cn$.

# CPSC 320: Memoization and Dynamic Programming I *

You want to make change in the world, but to get started, you're just ... making change. You have an unlimited supply of quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent, once upon a time). You want to make change for $n \geq 0$ cents using the minimum number of coins.

## 1 Build intuition through examples.

1. Here is an optimal greedy algorithm to make change. Try it on at least one instance.

   **function** GREEDY-CHANGE($n$)
   **while** $n > 0$ **do**
   **if** $n \geq 25$ **then** give a quarter and reduce $n$ by 25
   **else if** $n \geq 10$ **then** give a dime and reduce $n$ by 10
   **else if** $n \geq 5$ **then** give a nickel and reduce $n$ by 5
   **else** give a penny and reduce $n$ by 1

2. A few years back, the Canadian government eliminated the penny. Imagine the Canadian government accidentally eliminated the nickel rather than the penny. That is, assume you have an unlimited supply of quarters, dimes, and pennies, but no nickels. Adapt algorithm GREEDY-CHANGE for the case where the nickel is eliminated, by changing the code above. Then see if you can find a counterexample to its correctness.

**39**
**The algorithm will return 1 quarter and 5 pennies.**
**But 3 dimes would be better.**

## 2 Write down a formal problem specification.

We'll assume that a currency which includes the penny is fixed, with coins of value $1, v_1, \ldots v_k$ for some $k \geq 1$. We'll work with the currency $1, 10, 25$ in what follows, but want an algorithm that can easily be adapted to work more generally. What is an instance of the making change problem? This is an example of a *minimization problem*; what quantity are we trying to minimize?

**problem instance: nonnegative integer n**

**potential solution: number of coins needed to make changes for n**

**optimal solutions: minimum number of coins to make changes for n**

## 3 Evaluate brute force.

As is often the case, this approach will lead us to even better approaches later on. It will be helpful to write our brute force algorithm recursively. We'll build up to that in several steps.

1. To make the change, you must start by handing the customer some coin. What are your options?

   **3 options: give 1 penny, 1 dime, or 1 quarter**

2. Imagine that in order to make $n = 81$ cents of change using the minimum number of coins, you can start by handing the customer a quarter. Clearly describe the subproblem you are left with (but **don't** solve it). You can use the notation above in the formal problem specification.

   **we are left with 81-25=56**

3. Even if we're not sure that a quarter is an optimal move, we can still get an upper bound on the number of coins by considering the subproblem we are left with when we start with a quarter. What upper bound do we get on $C(81)$?

   **C(81) ≤ C(56) + 1**

4. What other upper bounds on $C(81)$ do we get if we consider each of the other "first coin" options (besides a quarter), and the corresponding subproblem?

   **C(81) ≤ C(71)+1**

   **C(81) ≤ C(80)+1**

5. There are three choices of coin to give first. Can you express $C(81)$ as the minimum of three options?

   **C(81) = min{C(71)+1, C(80)+1, C(56)+1}**

6. Now, consider the more general problem of making change when there are $k+1$ different coins available, with one being a penny, and the remaining $k$ coins having values $v_1, v_2, \ldots, v_k$, all of which are greater than 1. Let $C'(n)$ be the minimum number of coins needed in this case. For sufficiently large $n$, how can you express $C'(n)$ in terms of $C'()$ evaluated on amounts smaller than $n$?

**C′(n) = min{ C′(n-v(k))+1, C′(n-v(k-1))+1, C′(n-v(k-2))+1, …, C′(n-v(1))+1, C′(n-1)+1}**

7. Complete the following recursive brute force algorithm for making change:

**function** BRUTE-FORCE-CHANGE($n$)
  **if** $n < 0$ **then**
    **return** **∞**
  **if** $n = 0$ **then**
    **return** **0**
  **if** $n > 0$ **then**
    **return** the minimum of:
      **B_F_C(n-25)+1**,
      **B_F_C(n-10)+1**, and
      **B_F_C(n-1)+1**

8. Complete the following recurrence for the runtime of algorithm Brute-Force-Change:

$T(n) =$ **c** for $n < 0$

$T(n) =$ **T(n-25)+T(n-10)+T(n-1)+c** otherwise.

9. Give a disappointing $\Omega$-bound on the runtime of BRUTE-FORCE-CHANGE by following these steps:

   (a) $T(n)$ is hard to deal with because the three recursive terms in part (8) above are different. To lower bound $T(n)$, we make them all equal to the largest term. Complete the lower bound that we get for the recursive case when we do this:

   For the recursive case, $T(n) \geq$ **3T(n-25)+c** .

   (b) Now, draw a recurrence tree for $T(n)$ and figure out its number of levels, work per level, and total work.

10. Why is the performance so bad? Does this algorithm waste time trying to solve the same subproblem more than once? For $n = 81$, draw the first three levels (the root at level 0 plus two more levels) of the recursion tree for BRUTE-FORCE-CHANGE to assess this. Label each node by the size of its subproblem. Does any subproblem appear more than once?