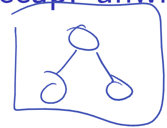
# CSC236 winter 2020, week 5: The Master Theorem

Recommended supplementary reading: David Liu 236 course notes pp 27-41, Ch. 5 "Algorithm Design" by Kleinberg & Tardos, Ch. 3 Vassos course notes

Colin Morris
colin@cs.toronto.edu
http://www.cs.toronto.edu/~colin/236/W20/

February 5, 2020

# Recap: unwinding



$T(n) = (\log n \times 2n + 5n$

$T(n) = 2n + 2n + \dots nT(n/n)$

$$T(n) = \begin{cases} \cancel{2}\ 5 & \text{if } n = 1 \\ 2n + 2T(n/2) & \text{if } n > 1 \end{cases}$$

Convention used in slides:

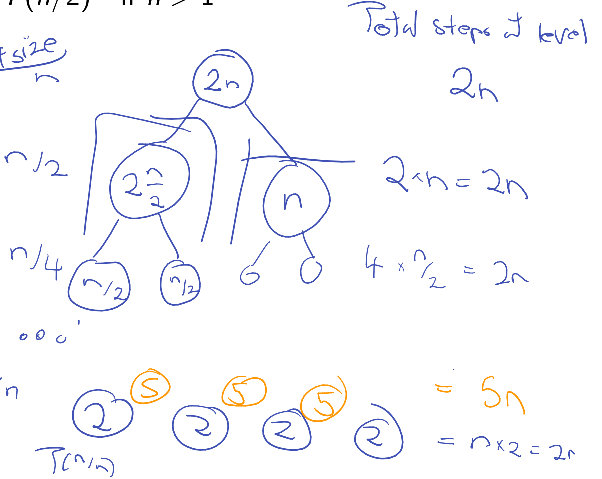
- label nodes with number of non-recursive steps
- label *levels* with problem size and total steps

Also note:

- **height** $\neq$ num 'levels'
- Usually good to draw the final (leaf) level, especially if seeking an exact closed form

max distance from root to leaf

Input size
n



# From last week: closest_pair

$T(n) = aT(\frac{n}{b}) + f(n)$. What are $a$, $b$, and $f(n)$?

```
def closest_distance(A):
  if len(A) == 2:
    return abs(A[0] - A[1])
  mid = len(A)//2
  L = A[:mid]
  R = A[mid:]
  # Find the closest distance between pairs that straddle L and R
  closest_LR = infinity
  for l in L:
    for r in R:
      closest_LR = min(closest_LR, abs(l-r))
  # Closest pair is either within L, within R, or between L and R
  return min(closest_LR, closest_distance(L), closest_distance(R))
```

$n$

$n$

$n \times n = n^2$

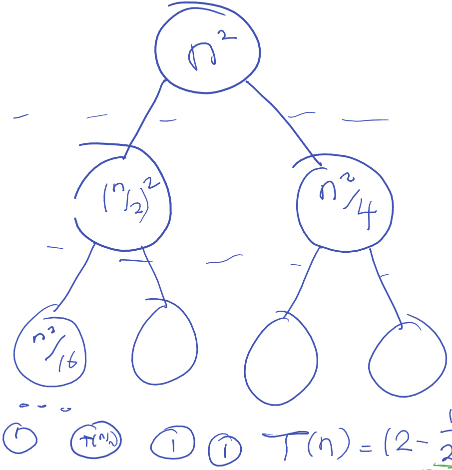
$T(n) =$

# Closed form when cost per recursive call is quadratic?

$T(n) = 2T(n/2) + f(n)$, where $f(n) \in \Theta(n^2)$

$f(n) = n^2/4 \approx n^2$

| Size of problem | Total steps |
|---|---|
| $n$ | $n^2$ |
| $n/2$ | $n^2/4 \times 2 = n^2/2$ |
| $n/4$ | $n^2/4$ |
| | $n^2/8$ |
| | ... |




$n/k \Rightarrow$ total steps $= \frac{n^2}{k}$

$T(n) = (2 - \frac{1}{2^n})n^2$

Oops, This should have been $2^{\log n}$

$T(n) \approx 2n^2$

# Closed form when cost per recursive call is quadratic?

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ n^2 + 2T(n/2) & \text{if } n > 1 \end{cases}$$

Smallest input size = 1

$$T(n) = n^2 + n^2/2 + n^2/4 \dots n^2/n = n$$

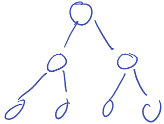
$$= \sum_{i=0}^{\log n} \frac{n^2}{2^i}$$

$$= n^2 \sum_{i=0}^{\log n} \frac{1}{2^i}$$

$$= n^2 \left( 2 - \frac{1}{2^{\log n}} \right) = 2n^2 - n$$

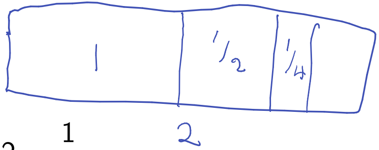
# Useful geometric series to recognize

## Powers of 2 come up a lot in computer science!



$$\sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \ldots + 2^n = 2^{n+1} - 1$$

(Number of nodes in a binary tree of height $n$)




$$\sum_{i=0}^{n} 2^{-i} = 1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^n} = 2 - \frac{1}{2^n}$$

But you don't *need* to memorize these. For tests, we'll either provide you with the formula, or allow you to leave these as un-reduced $\Sigma$ sums.

# Finding the maximum by divide-and-conquer

$f(n) \in \Theta(1)$

$T(n) = aT(\frac{n}{b}) + f(n)$. What are $a$, $b$, and $f(n)$?

$a = 2$ $b = 2$ $f(n) = 1$

```
def maximum(A):
  if len(A) == 1:
    return A[0]
  mid = len(A) // 2
  L_max = maximum(A[:mid])
  R_max = maximum(A[mid:])
  if L_max > R_max:
    return L_max
  else:
    return R_max
```

(Brainteaser: can you prove that *any* algorithm solving this problem must be in $O(n)$?)

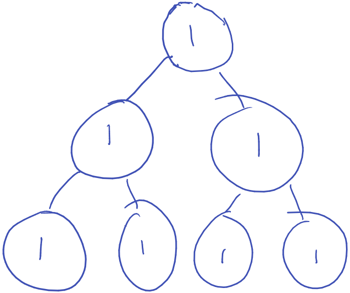
# Asymptotic runtime of `maximum`?

$T(n) = 2T(n/2) + f(n)$, where $f(n) \in \Theta(1)$

$T(1) = 1$

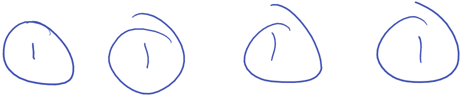
$1 + 2 + 4 + \dots 2^k = 2^{k+1} - 1$

| Input size | Total steps at level |
| --- | --- |
| $n$ | $1$ |
| $n/2$ | $1+1 = 2$ |
| $n/4$ | $1+1+1+1 = 4$ |



$n/k \Rightarrow k$ steps

$n/n$



$= n$

$T(n) = 1 + 2 + 4 \dots n = 2^k$

$= 2^{k+1} - 1$

$= 2^{\log n + 1} - 1$

$= 2 \times 2^{\log n} - 1$

$T(n) = 2n - 1$

# Bifurcate? No, trifurcate!

$T(n) = aT(\frac{n}{b}) + f(n)$. What are $a$, $b$, and $f(n)$?

$a = 3 \quad b = 3 \quad f(n) = 1 \quad f(n) \in \Theta(1)$

```python
def max_tri(A):
  if len(A) == 1:
    return A[0]
  m1 = len(A) // 3
  m2 = (2*len(A)) // 3
  L_max = max_tri(A[:m11])
  Centre_max = max_tri(A[m_1:m_2])
  R_max = max_tri(A[m_2:])
  if L_max > Centre_max and L_max > R_max:
    return L_max
  elif Centre_max > R_max:
    return Centre_max
  else:
    return R_max
```

$T(n) \approx 3n$ ?

# Asymptotic runtime of `max_tri`?

$T(n) = 3T(n/3) + f(n)$, where $f(n) \in \Theta(1)$

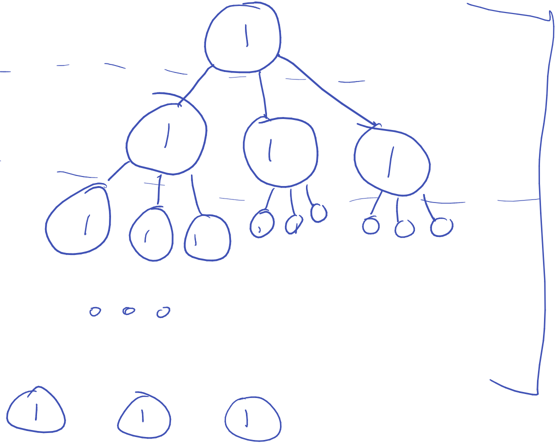
Input size

$n$

$n/3$

$n/9$

$n/n$



$1$

$3$

$9$

$\dots$

$3^{\log_3 n} = n$

$$T(n) = 1 + 3 + 9 + \ldots n$$

$$= \sum_{i=0}^{\log_3 n} 3^i$$

$$= \frac{3^{(\log_3 n)+1} - 1}{2}$$

$$= \frac{3n - 1}{2}$$

$$S = 1 + 3 + 9 + \ldots 3^k$$

$$3S = 3 + 9 + 27 + \ldots 3^{k+1}$$

$$3S = S - 1 + 3^{k+1}$$

$$2S = 3^{k+1} - 1$$

$$S = \frac{3^{k+1} - 1}{2}$$

$n = 1$, $T(1) = 1$, $\frac{3-1}{2} = 1$

$n = 3$, $T(3) = 3 \cdot T(\frac{3}{3}) + 1$ $\quad \frac{3 \times 3 - 1}{2} = \frac{8}{2} = 4$

$= 3 + 1 = 4$

# What if $a > b$?

i.e. number of recursive calls is greater than shrinkage factor. Overlapping subproblems?

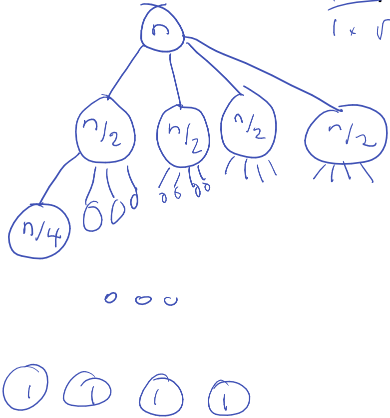
Input size

$n$

$n/2$

$n/4$

$n/n$




Total $T(n) = 4T(\frac{n}{2}) + n$

$1 \times n = n$

$4 \times \frac{n}{2} = 2n$

$16 \times \frac{n}{4} = 4n$

$64 \times \frac{n}{8} = 8n$

$2^{\log_2 n}$

$n^2$

Pattern: input size $= n/k$

work at this level $k \cdot n$

Split $\log_2 n$ times

each split: nodes $\times 4$

$4^{\log_2 n}$ nodes

level 2: $4^{\log_2 2}$

level 3: $4^{\log_2 4}$

level 4: $4^{\log_2 8}$

$$T(n) = n + 2n + 4n + \dots n^2$$

$$= \sum_{i=0}^{\log n} 2^i n$$

$$= n \times \sum_{i=0}^{\log n} 2^i$$

$$= n \times \left(2^{\log n + 1} - 1\right)$$

$$= n \times \left(2 \times 2^{\log n} - 1\right) = n(2n - 1) = 2n^2 - n$$

# The Master Theorem

## Now that we're thoroughly tired of unwinding...

A handy-dandy recipe for finding the asymptotic complexity of divide-and-conquer algorithms. Given $T(n)$ of the form

$$T(n) = aT(\frac{n}{b}) + f(n)$$

$a \in \mathbb{N}^+$

$b \in \mathbb{N}^+ - \{1\}$

$d \in \mathbb{N}$

The Master Theorem says that, if $f \in \Theta(n^d)$, then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_b n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

# Looking back

$a \_ b^d$

| Algo | $a$ | $b$ | $f(n) \in \Theta(n^d)$ | $b^d$ | $T(n) \in \Theta(\_)$ | |
|---|---|---|---|---|---|---|
| `mergesort` | 2 | 2 | $n^1$ | 2 | $n \log n$ | $=$ |
| `closest_distance` | 2 | 2 | $n^2$ | 4 | $n^d = n^2$ | $<$ |
| `binsearch` | 1 | 2 | $1 = n^0$ | | $\log_2 n$ | $=$ |
| `maximum` | 2 | 2 | $1 = n^0$ | 1 | $n^{\log_2 a} = n^{\log_2 2} = n$ | $>$ |
| `max_tri` | 3 | 3 | $1 = n^0$ | 1 | $n^{\log_3 3} = n$ | $>$ |
| (anon) | 4 | 2 | $n^1$ | 2 | $n^{\log_2 4} = n^2$ | $>$ |

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^d) \xrightarrow{\text{Master Theorem}} T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_b n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

# Looking back even further

$$T(n) = aT(\frac{n}{b}) + \Theta(n^d) \xrightarrow{\text{Master Theorem}} T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_b n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

What about `fact`, which had recurrence

$$T(n) = 1 + T(n-1)$$

Or `subset_sum`?

$$T(n) = 1 + 2T(n-1)$$

Master Theorem can't replace unwinding for *all* recurrences. (It also doesn't give an exact closed form.)

# Appendix: Slices and step counting

What is the cost of running the following code?

```
# Sublist with the left half of A
L = A[:len(A)//2]
```

Reality of Python's implementation $= \Omega(n)$
In this course, we'll count it as $\Theta(1)$. Justification:

- We can generally rewrite our algorithms to avoid slicing by passing additional arguments, representing start and end indices into the original list (see next slide)
- We could also imagine our algorithms are taking `numpy` arrays instead of lists
- We don't want to tie ourselves to the implementation details of any particular language.

Except where we explicitly state otherwise, we will treat *all* built-in functions and operators as constant time.

# Appendix: `maximum` without slices

**Original**

```python
def maximum(A):
  if len(A) == 1:
    return A[0]
  mid = len(A) // 2
  L_max = maximum(A[:mid])
  R_max = maximum(A[mid:])
  if L_max > R_max:
    return L_max
  else:
    return R_max
```

**Transformed**

```python
def maximum(A, start, end):
  if end - start == 1:
    return A[start]
  mid = (start + end) // 2
  L_max = maximum(A, start, mid)
  R_max = maximum(A, mid, end)
  if L_max > R_max:
    return L_max
  else:
    return R_max
```

Exercise: Use the Master Theorem to devise a recurrence T(n) having big-Theta complexity...

$T(n) \in \Theta(n^2 \log_5 n)$

$$T(n) \in \begin{cases} \Theta(n^d) & , a < b^d \\ \Theta(n^d \log_b n) & , a = b^d \\ \Theta(n^{\log_b a}) & , a > b^d \end{cases}$$

$T(n) = 5^2 T(n/5) + n^2$

$a = 5^2 = 25$

$b = 5 \quad f(n) = n^2 \Rightarrow d = 2$

$b^d = 25 = a$

$\Theta(n^d \log_b n) = \Theta(n^2 \log_5 n)$

Exercise: What are the possible big-Theta complexities of a divide-and-conquer recurrence where f(n) is constant? i.e. T(n) of the form

$$T(n) = aT(n/b) + 1$$

$f(n) \in \Theta(1)$

$\Theta(n^d \log n)$, $a = b^d = 1$

$\Theta(n^{\log_b a})$, $a > b^d$, $a > 1$

$T(n) \in \Theta(\sqrt{n})$ ?

$b = 4$

$a = 2$

$T(n) \in \Theta(1)$ ?

1. Consider the following sketch of a divide-and-conquer algorithm $r(s)$ for reversing a string:

   (a) $s$ is a string.

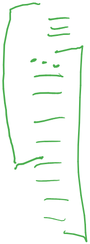
   (b) If $\text{len}(s) < 2$, return $s$

   (c) Else, partition $s$ into three roughly equal parts: prefix $s_1$, suffix $s_3$, and mid-section $s_2$, and return $r(s_3) + r(s_2) + r(s_1)$.

   (d) You may assume that the time complexity of string concatenation of $s_3 + s_2 + s_1$ is proportional to $\text{len}(s_3) + \text{len}(s_2) + \text{len}(s_1)$

   Use the Master Theorem to find the asymptotic time complexity of function $r$ in terms of $\text{len}(s)$. Be sure to show all the components of your analysis, including the values of $a, b$, and $d$. How does this compare to the complexity of simply copying the string elements in reverse order, using a loop?

2. Describe a ternary version of MergeSort where the list segment to be sorted is divided into three (roughly) equal sub-lists, rather than two. Use the Master Theorem to find the asymptotic time complexity of your ternary MergeSort in terms of the length of the list segment being sorted, and compare/contrast it with the version we analyzed in class. Be sure to show all the components of your analysis, including the values of $a, b$, and $d$.



$4j+3$ $P(n-4)$