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

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

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

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))
```

Closed form when cost per recursive call is quadratic?

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

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 Σ sums.

Finding the maximum by divide-and-conquer

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

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

Bifurcate? No, trifurcate!

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

```
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
```

Asymptotic runtime of `max_tri`?

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

What if $a > b$?

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

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

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

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

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

Algo	a	b	$f(n) \in \Theta(n^d)$	b^d	$T(n) \in \Theta(\_)$
`mergesort`	2	2	n^1		$n \log n$
`closest_distance`	2	2	n^2		
`binsearch`	1	2	$1 = n^0$		
`maximum`	2	2	$1 = n^0$		
`max_tri`	3	3	$1 = n^0$		
(anon)	4	2	n^1		

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

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

```
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

```
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
```