CSC236 winter 2020, week 4: Runtime of recursive algorithms

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/

January 27, 2020

Outline

Induction roundup

Runtime analysis of recursive algorithms

Divide-and-conquer algorithms
- Example: merge sort
- Divide-and-conqer definitions
- Experimenting with different runtime characteristics

The Master Theorem

Appendix

Cardinal sins of induction

You will always lose marks for these

1. 'Overwriting' your predicate's argument. e.g. $P(n) : \forall n \in \mathbb{N}, 2^n \geq n$
   Should be $P(n) : 2^n \geq n$
   If you *really* want to be explicit about the domain of your predicate, these are also acceptable (but not necessary):
   - $P(n) : 2^n \geq n, n \in \mathbb{N}$
   - For $n \in \mathbb{N}$, define $P(n) : 2^n \geq n$
   - Define a predicate of the natural numbers $P(n) : 2^n \geq n, n \in \mathbb{N}$
2. Referring to a predicate without defining it.
3. Omitting the induction hypothesis.

Venial sins of induction

You *might* be able to get away with these, but try to avoid them.

1. Writing something like $P(0) = 2^0 \geq 0$
   - `TypeError: Incompatible types 'bool' and 'int'`
   - Instead "$2^0 \geq 0$, thus $P(0)$"
2. Using variables without introducing them. e.g. starting I.S. with "Assume $P(n)$"
   - `NameError: name 'n' is not defined`
   - Instead "Let $n \in \mathbb{N}$ and assume $P(n)$", or "Assume $P(n)$ for arbitrary n", etc.
   - Especially important if you need to put restrictions on n to make I.S. work. e.g. "Let $n \in \mathbb{N}, n \geq 20$"
3. Unnecessary base cases. *Usually* one is enough.
   - You'll never lose marks for this, but you will lose time!
   - If you need more than 1 base case, it should become apparent as you're writing your I.S.
4. Implicitly using I.H.
   - If you're using the I.H. to rewrite an expression, say so
   - In complete induction, if you invoke, say, $P(n-3)$, justify *why* you're able to do so. How do you know $n-3$ falls under the range of the I.H.?

Parting induction tips

- Not sure where to start? Looking at some small examples may help.
    - And, if applicable, scribbling some diagrams
- You can get most of the marks for having the right *structure*, even if you can't figure out the "trick" to complete the induction step.
- For complete induction, focus on **understanding** the induction hypothesis, rather than memorizing a formula
    - Remember, there are lots of acceptable ways to write the I.H.
    - In general, we're not picky about notation, as long as your reasoning is clearly expressed

Runtime analysis of recursive algorithms

What is the runtime of `fact` on input `n`?

```python
def fact(n):
    """Return n!
    """
    if n == 0:
        return 1
    return n * fact(n-1)
```

$T(n) =$

Closed form for $T(n)$?

Motivation: suppose we want to know $T(1000)$...

What is the runtime of `subset_sum`?

For more information see Wikipedia's article on the subset sum problem.

```
def subset_sum(A, target):
    """Return True iff there is a subset of items in A, a list
    of integers, which adds up to the given target sum.
    """
    if len(A) == 0:
        return target == 0
    # Try to make the sum either with the first number, or without
    return subset_sum(A[1:], target) or subset_sum(A[1:], target-A[0])
```

$T(n) =$

Closed form for runtime of `subset_sum`?

Use the technique of repeated substitution (AKA "unwinding")...

Proving our closed form is correct

Runtime of merge sort

```python
def mergesort(A):
  if len(A) <= 1:
    return A
  m = len(A) // 2
  L1 = mergesort(A[:m])
  L2 = mergesort(A[m:])
  return merge(L1, L2)

def merge(A, B):
  i = j = 0
  C = []
  while i < len(A) and j < len(B):
    if A[i] <= B[j]:
      C.append(A[i])
      i += 1
    else:
      C.append(B[j])
      j += 1
  return C + A[i:] + B[j:]
```

$T(n) =$

Finding a closed form for $T(n)$

Via unwinding

An officially sanctioned *deus ex machina*

For proofs in this course, you may assume that inputs are always "nice" sizes when analysing the runtime of recursive algorithms that partition their inputs, such as mergesort.
In this case, you may assume n is always such that

$$\lceil n/2 \rceil = \lfloor n/2 \rfloor = n/2$$

If you're skeptical, you may wish to look at chapter 3 of the course notes, which proves a closed form for mergesort without this hand-waving. The proof is *long*, but not difficult to understand.

Trying again

Find a closed form for $T(n)$ via unwinding, *assuming n is a power of 2*

Divide-and-conquer

Mergesort is an example of the general class of **divide and conquer algorithms**. These algorithms break their input into equally sized subproblems, solve them recursively, then combine the results. Their runtime can be written as:

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

Where

- a is the number of recursive calls
- b is the 'shrinkage factor' of the subproblems
- $f(n)$ is the cost of the non-recursive part (splitting and recombining)

Divide-and-conquer

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

We've seen $a = b = 2$ and $f(n) \in \Theta(n) \implies T(n) \in \Theta(n \log n)$

What happens when we change some of these parameters?

A silly algorithm

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

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

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
```

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]
  tertile_1 = len(A) // 3 # I like tertiles
  tertile_2 = (2*len(A)) // 3
  L_max = max_tri(A[:tertile_1])
  Centre_max = max_tri(A[tertile_1:tertile_2])
  R_max = max_tri(A[tertile_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 $f(n) \in \Theta(n)$

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	$f(n) \in \Theta(\_)$	b^d	$T(n) \in \Theta(\_)$
`mergesort`	2	2	n^1		$n \log n$
`closest_distance`	2	2	n^2		
`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`?

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

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
```