[with some extra annotations added outside of lecture – in green]

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

W4

W5

[I cut out the slides we didn't reach. We'll use them next week.]

Cardinal sins of induction

You will always lose marks for these

$P(0) = P(1) = P(2)$

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'` ed
   - 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.

   # of IH

   - 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

$P(n)$:

Basis: $P(0)$

Let $n \in \mathbb{N}$

Assume $P(n)$

$\ldots$

thus $P(n+1)$

- 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)
```

basis: $0! = 1$

$n! = n \times (n-1)!$

$$T(n) = \begin{cases} 1, & n = 0 \\ 1 + T(n-1), & n > 0 \end{cases}$$

steps taken on input n

could have counted these as 2 steps or 3 or "c" to represent a fixed constant. Asymptotic behaviour will be the same regardless.

$T(1000)$

$O(\quad)$

$\Theta(\quad)$

Closed form for $T(n)$?

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

$$T(n) = \begin{cases} 1, & n = 0 \\ 1 + T(n-1), & n > 0 \end{cases}$$

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

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

$$= 1 + (1 + (1 + T(n-3)))$$

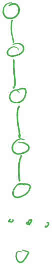
... [intuition happens]

$$= \underbrace{1 + 1 + 1 + 1 \ldots}_{n} T(n-n)$$

$$= n + 1$$

Visual unwinding

Input size	Steps taken (not counting recursive call)
n	1
$n-1$	1
$n-2$	1
$n-3$	1
$n-4$	1
...	
1	



(vine?)

Height of "tree" $= n+1$

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

What is the runtime of subset_sum?

[1, 3, 7, 9, 14] 16

[3, 7, 9, 14], 16   [3, 7, 9, 14] 15

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

```python
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])
```

$$T(n) = \begin{cases} 1 & , n = 0 \\ 1 + 2T(n-1) & , n > 0 \end{cases}$$

steps on input of size n in worst case

$n = \text{len}(A)$

Guess: $\Theta(2^n)$

Closed form for runtime of subset_sum?

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

$$T(n) = \begin{cases} 1 & , n = 0 \\ 1 + 2T(n-1) & , n > 0 \end{cases}$$

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

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

$$= 1 + 2(1 + 2(1 + 2T(n-3)))$$

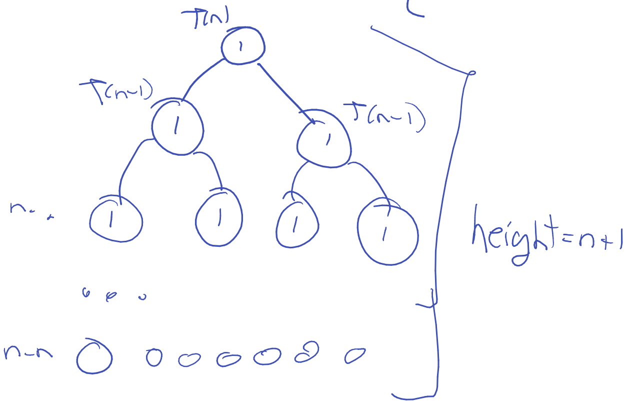
$$= 1 + 2 + 4 + 8T(n-3)$$

$$\ldots$$

$$= 1 + 2 + 4 + 8 + \ldots 2^{n} T(n-n)$$

n-1

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



number of nodes = sum of value of nodes $= T(n)$

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

Proving our closed form is correct

$$T(n) = \begin{cases} 1 & , n = 0 \\ 1 + 2T(n-1) & , n > 0 \end{cases}$$

$P(n): T(n) = 2^{n+1} - 1$

Basis, $n = 0$

$T(0) = 1 = 2^{0+1} - 1$, so $P(0)$

IS Let $n \in \mathbb{N}$, assume $P(n)$

$T(n+1) = 1 + 2T(n)$ # $n+1 > 0$

$= 1 + 2(2^{n+1} - 1)$

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

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

thus $P(n+1)$

Runtime of merge sort

$[3, 7, 1 \,|\, 8, 4] \Rightarrow [1, 3, 7 \,|\, 4, 8]$

```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:]
```

$$T(n) = \begin{cases} 1, & n \le 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n, & n > 1 \end{cases}$$

ceiling

floor

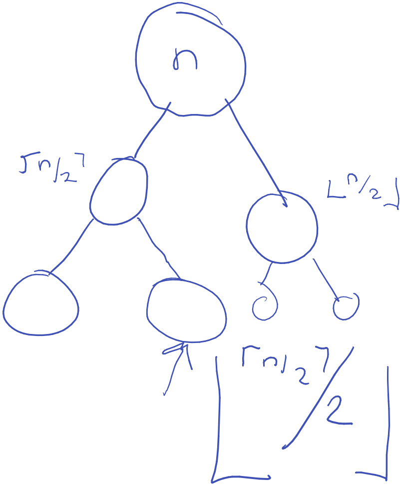
n = len(A)

n iterations × constant work, so n

Finding a closed form for $T(n)$

Via unwinding

$$T(n) = \begin{cases} 1 & , n \le 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n & n > 1 \end{cases}$$



gross...

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

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

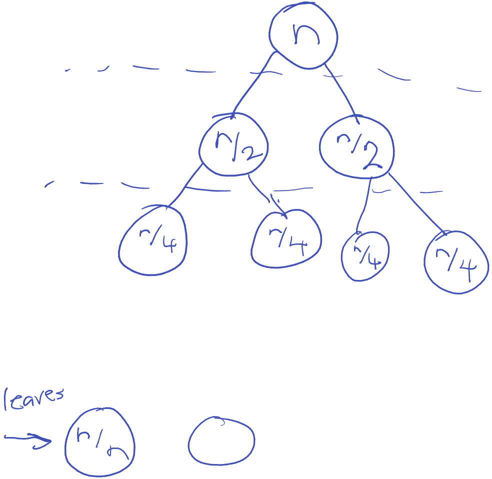
i.e. $n = 2^k$ for some $k \in \mathbb{N}$

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

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

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



n

$2 \times n/2 = n$

$4 \times n/4 = n$

...

$n \times 1 = n$

height $= \log_2 n + 1$

At each level, we do n steps

$T(n) = n(\log n + 1)$

$= n \log n + n$

NB: Test closed form on a small value of n to check for off-by-one errors.

Trying again

$P(k)$: $n = 2^k$

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

Prove

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

$P(n)$: (n is a power of 2) $\Rightarrow T(n) = n \log n + n$

Let $n \in \mathbb{N}$, assume $P(k)$ holds for all $k < n$

Case: n is a power of 2, $n > 1$

$T(n) = 2T(n/2) + n$ # $n > 1$

$= 2\left(\frac{n}{2} \cdot \log(n/2) + n/2\right) + n$ # $P(n/2)$ by IH

$=$ [algebra]

$= n \log n + n$, so $P(n)$

Case 2: $n = 1$

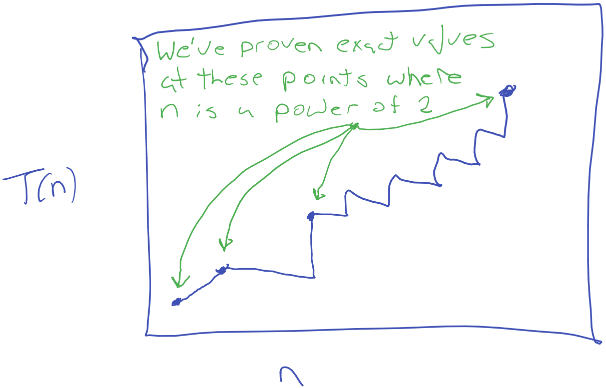
by defn $T(1) = 1 = 1 \log 1 + 1$, so $P(n)$

Case 3: n not a power of 2, $P(n)$ is vacuosly true

Trying again

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

Proof sketch: $T(n) \in \Theta(n \log n)$ for *all* n (not ignoring floor and ceiling)

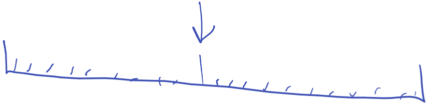


1. prove for 2^k

2, prove $T(n)$ is nondecreasing

3. trickery big-Oh
(To show that behaviour in between powers of 2 "doesn't matter" in terms of big-Oh)

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:

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

mergesort values

Where

- a is the number of recursive calls $= 2$
- b is the 'shrinkage factor' of the subproblems $= 2$
- $f(n)$ is the cost of the non-recursive part (splitting and recombining) $f(n) = n$

Bin search: $a = 1$ $b = 2$ $f(n) = 1$

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 $f(n)$?

[1, 2 | 11, 5, 16]

"obvious" implementation $\in \Theta(n^2)$

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

$n/2$

$n/2$

$\times$ $n^2/4$

$a = 2$ $b = 2$ $f(n) = n^2/4$

dominated by nested loop on lines 9-11. Each iterates $n/2$ times. $n/2 \times n/2 = n^2/4$

Closed form when cost per recursive call is quadratic?

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

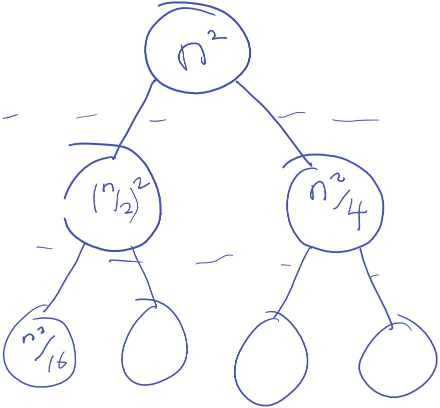
$f(n) = n^2/4 \approx n^2$

Size of problem

n

$n/2$

$n/4$



Total steps

n^2

$n^2/4 \times 2 = n^2/2$

$n^2/4$

$n^2/8$

...

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

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

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