

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-conquer definitions

Experimenting with different runtime characteristics

The Master Theorem

Appendix



Cardinal sins of induction

You will always lose marks for these

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

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.

$$P(0): 2^0 \geq 0 \quad \checkmark$$

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.?

by IH

Parting induction tips

$P(n)$:
Basis. ...thru $P(0)$
Let $n \in \mathbb{N}$
Assume $P(n)$
...
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?

```
1 def fact(n):  
2     """Return n!  
3     """  
4     if n == 0:  
5         return 1  
6     return n * fact(n-1)
```

$0! = 1$
 $n! = n \times (n-1)!$

$T(n) =$
↑
steps taken
on input n

$\left. \begin{array}{l} 1 \\ 1 + T(n-1) \end{array} \right\} \begin{array}{l} n=0 \\ n>0 \end{array}$

$T(1000)$

$\cdot \Theta(\)$

$\hat{\Theta}(\)$

Closed form for $T(n)$?

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

$$T(n) =$$

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

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

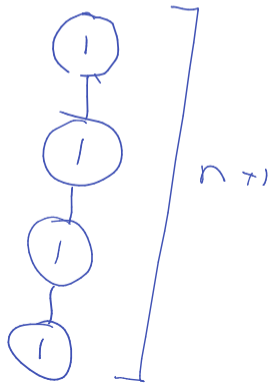
$$= 1 + (1 + T(n-2))$$

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

... [intuition happens here]

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

$$= n + 1$$



What is the runtime of subset_sum? $[1, 3, 7, 9, 13], 16$

$[3, 7, 9, 13], 16$ $[3, 7, 9, 13]$

For more information see [Wikipedia's article on the subset sum problem](#).

```
1 def subset_sum(A, target):
2     """Return True iff there is a subset of items in A, a list
3     of integers, which adds up to the given target sum.
4     """
5     if len(A) == 0:
6         return target == 0
7     # Try to make the sum either with the first number, or without
8     return subset_sum(A[1:], target) or subset_sum(A[1:], target - A[0])
```

Worst-case # of steps on input n

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

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

Closed form for runtime of subset_sum?

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

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

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

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

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

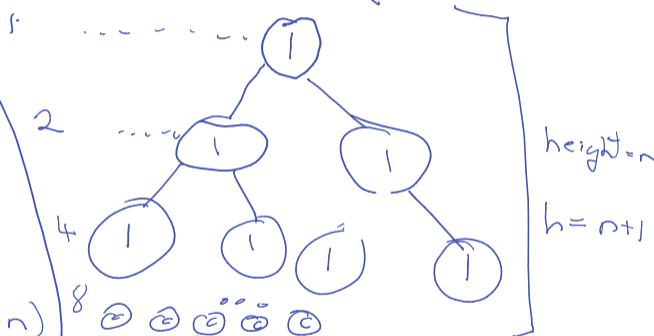
$$= 8T(n-3) + 4 + 2 + 1$$

...

$$= 1 + 2 + 4 + 8 + \dots + 2^n T(n-n)$$

$$= \sum_{k=0}^n 2^k = 2^{n+1} - 1$$

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



Sum of all nodes = $2^n - 1$

Prove: $T(n) = 2^n - 1 + 2^n c$

Proving our closed form is correct

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

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

P(n): $T(n) = 2^n + 2^0 c - 1$

Basis

$$T(0) = c = 2^0 + 2^0 c - 1 \\ = 1 + c - 1 = c$$

, so P(0)

is

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

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

$$= 1 + 2(2^n + 2^0 c - 1)$$

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

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

So P(n+1)



Runtime of merge sort $[3, 1, 7, 11, 5] \Rightarrow [1, 3, 5, 7, 11]$

```

1 def mergesort(A):
2     if len(A) <= 1:
3         return A
4     m = len(A) // 2
5     L1 = mergesort(A[:m])
6     L2 = mergesort(A[m:])
7     return merge(L1, L2)
8
9 def merge(A, B):
10    i = j = 0
11    C = []
12    while i < len(A) and j < len(B):
13        if A[i] <= B[j]:
14            C.append(A[i])
15            i += 1
16        else:
17            C.append(B[j])
18            j += 1
19    return C + A[i:] + B[j:]

```

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

$\left. \begin{array}{l} 1 \\ n \leq 1 \end{array} \right\}$

~~$2T(n/2) + n$~~

$T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n$

$n = \text{len}(A) + \text{len}(B) = \text{len}(\text{original list})$

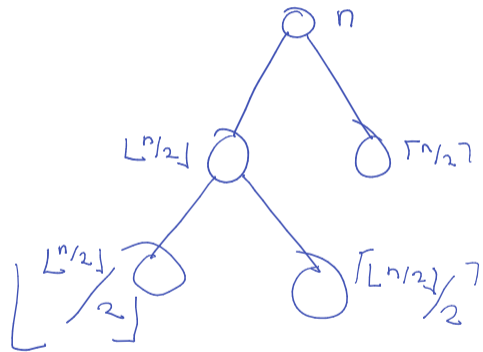
$T(n/2) + T(n/2)$

$[1, 2, 3, 4]$ $[1, 2, 3, 4, 5]$

Finding a closed form for $T(n) =$

Via unwinding

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



n

n

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

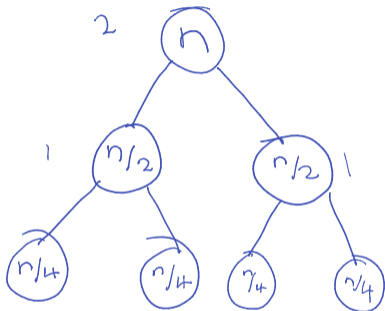
$$\lceil n/2 \rceil = \lfloor n/2 \rfloor = n/2$$

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

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

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



height $\approx \log_2 n$
 $= \log_2 n + 1$

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



Trying again

~~Find~~ ^{Prove} a closed form for $T(n)$ via unwinding, assuming n is a power of 2

$$P(n): (n \text{ is a power of } 2) \Rightarrow T(n) = n \log_2 n + n$$

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

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

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

$$= 2\left(\frac{n}{2} \log_2 \frac{n}{2} + \frac{n}{2}\right) + n$$

$$= n \log_2 n + n$$

by $P(n/2)$, via IH

$n/2$ is a power of 2, and $n/2 < n$

$$n=10$$

$P(5)$ is True

Case 2: n is not a power of 2

$P(n)$ vacuously true

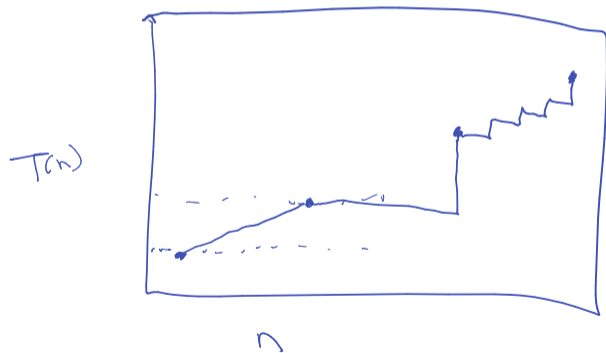
Case 3: $n = 2^0 = 1$

$$T(1) = 1, \text{ by defn } 1 = 1 \log_2 1 + 1 \text{ so } P(n)$$

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 ceil)



Prove $T(n)$ is
nondecreasing

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:

$$T(n) = \underline{a}T\left(\frac{n}{\underline{b}}\right) + f(n)$$

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)



Bin search: $a = 1$ $b = 2$

Divide-and-conquer

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

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 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))
```

$a = 2$ $b = 2$ $f(n) \approx n^2$ $T(n) = 2T(n/2) + n^2$

Closed form when cost per recursive call is quadratic?

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

