Divide-and-Conquer Algorithms

We can quickly search for the location of a given value in a sorted array using binary search.

The idea is that we look at the middle element of the remaining array. If the value we're searching for is smaller than the middle element then the desired value will be in the first half of the array. Otherwise it will be in the second half of the array.

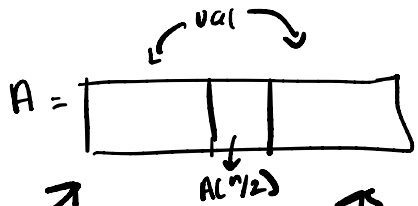
Consider the following pseudo code for binary search:

Algorithm 1 int binarySearch(int $A[1 \ldots n]$, int val)

```
if A[n/2] == val then
    //Found val at the current location
    return True;
else if n == 1 then
    //val is not in the array
    return False;
else if A[n/2] > val then
    //val is in first half of the array
    return binarySearch(A[1...n/2], val);
else A[n/2] < val
    //val is in second half of the array
    return binarySearch(A[n/2...n], val);
end if
```

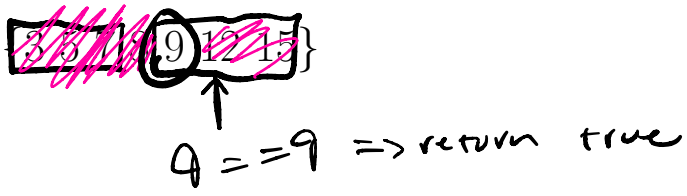
Base cases

recursive cases



Example: n = 7

Find 9 in the following array



What is difficult about finding the runtime of this algorithm? Its recursive

The divide-and-conquer design paradigm

(1) **Divide** the problem into subproblems of sizes that are fractions of the original problem size.
(2) **Conquer** the subproblems by solving them the subproblems by solving them recursively. solves sub problems
(3) **Combine** subproblem solutions.

Binary search:

(1) **Divide:** Check the middle element.
(2) **Conquer:** Recursively search 1 subarray.
(3) **Combine:** Trivial.
↳ return above result

Another example, Merge sort:

Algorithm 2 void mergeSort(int $A[1 \dots n]$)

```
if n == 1 then
    //Array of 1 element is trivially sorted
    return;
end if
mergeSort(A[1 ... ⌈n/2⌉]);//1st 1/2 of A
mergeSort(A[⌈n/2⌉ + 1 ... n]); //2nd 1/2 of A
Merge the two sorted lists
return;
```

Base case

//upon sorting both 1/2's we merge into a single sorted Array

Merge sort:

∴ comparisons are linear

(1) **Divide:** Trivial.
(2) **Conquer:** Recursively sort 2 subarrays of size $n/2$
(3) **Combine:** Linear time key subroutine Merge

How do you merge two sorted arrays?

n/2 {2 7 8 20} n/2 {1 10 13 21}

// we start at 0th elements and compare putting them in sorted order until no elements left

=> {1, 2, 7, 8, 10, 13, 20, 21}

n as e/a comparison takes 1, and n is # of comparisons of array elements

$\Theta(n)$ time to merge a total of n elements.

Let $T(n)$ represent the runtime of mergeSort: time to sort n elements on array of n elements

Algorithm 3 void mergeSort(int $A[1 \dots n]$)

```
if n == 1 then
    //Array of 1 element is trivially sorted
    return;
end if
mergeSort(A[1 ... ⌈n/2⌉]);          — T(n/2)   // as this is 1/2 of original array
mergeSort(A[⌈n/2⌉ + 1 ... n]);      — T(n/2)
Merge the two sorted lists
return;
```

T(n) = Θ(1) + T(n/2) + T(n/2) + Θ(n)

= 2T(n/2) + Θ(n) + Θ(1)

= constant

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

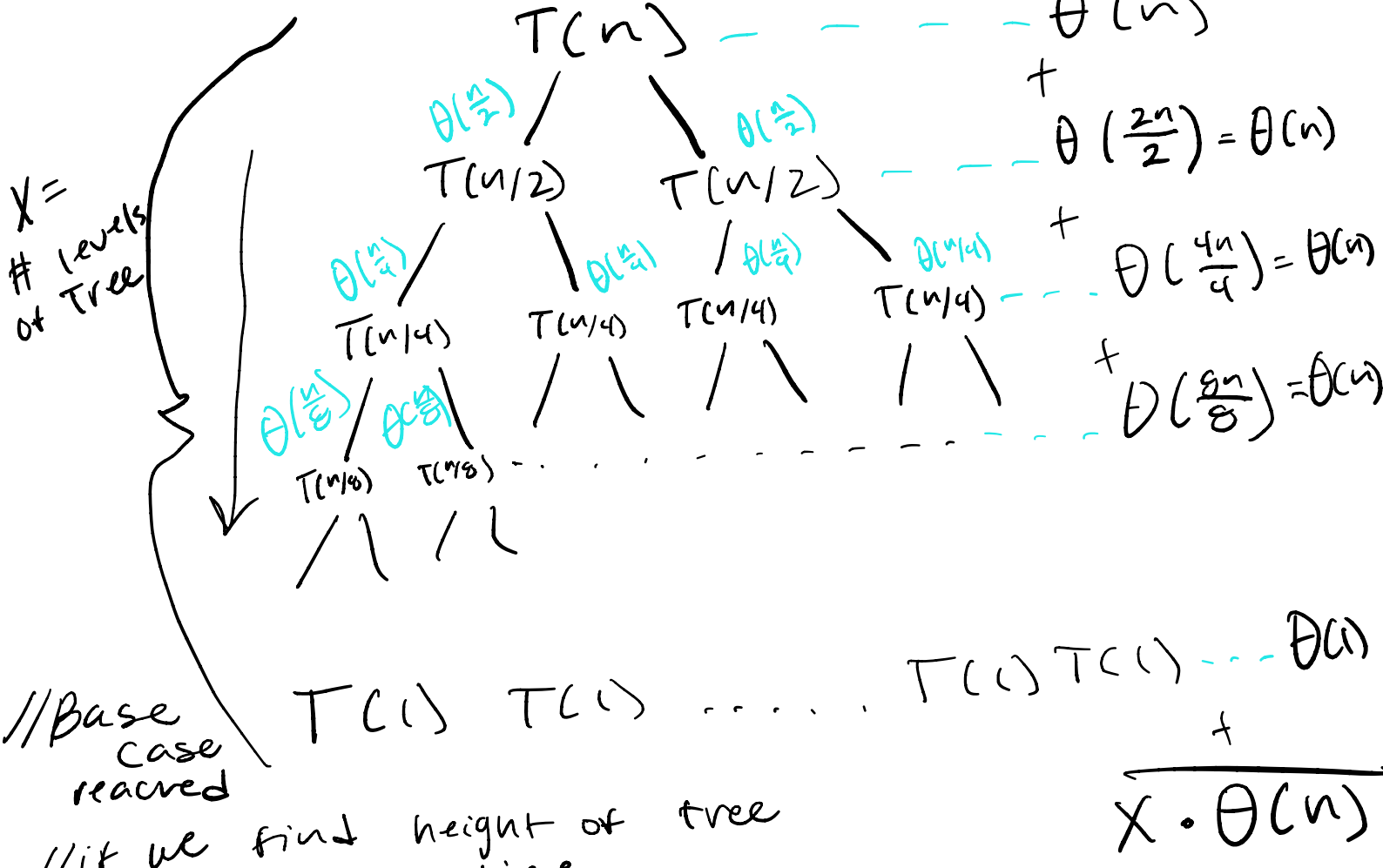
Recurrence for mergeSort:

$\Rightarrow T(n) = 2T(n/2) + \theta(n)$

↳ we need a way to find actual runtime

But what does $T(n)$ solve to? (e.g., n, n^2, n^3, or ...)

Try repeatedly plugging in definition of $T(n)$:
(Recursion Tree Method)



//if we find height of tree we can find runtime.

$\Rightarrow \frac{n}{2^x} = 1$ // how many divisions of 2 until we reach 1? (base case)

//n is power of 2

$\Rightarrow n = 2^x$ //log both sides

$\Rightarrow \log_2 n = x$

runtime $\theta(n \log n)$

$x \cdot \theta(n) = \log_2 n \cdot n = n \log n$

// We find c, n_0 values

The above method can be used to estimate a runtime but to be sure our guess is correct we would need to prove it with strong induction. // what we want to prove

$T(n) \in O(n \log n)$

$\Rightarrow T(n) \leq c \cdot f(n) = \boxed{T(n) \leq c \cdot n \log n}$

$c > 0, n_0 > 0$

$\forall n \geq n_0$

↳ Base case determined by n_0

// we pick our base case

- **Base Case** (Not base case from code)

$\Rightarrow$ When $n_0 = 1$ (Try)

LHS: $T(1) = \theta(1) = 1$ // let d = constant value

RHS: $c \cdot n \log n = c \cdot 1 \cdot \log_2(1) = c \cdot 1 \cdot 0 = 0$

↳ How long to sort 1 element array once?

// $n_0 = 1$ is not a true base case, thus we must try a different n_0 value

$\therefore$ To finish proof, we Prove: LHS $\leq$ RHS

$1 \leq 0$? (untrue)

$\Rightarrow$ When $n_0 = 2$

// we got this using our recurrence relation $\Rightarrow T(n) = 2 \cdot T(n/2) + \theta(n)$

LHS: $T(2) = 2 \cdot T(2/2) + \theta(2)$

$= 2 \cdot T(1) + 2$

↳ single element = 1

$= 2 + 2 = 4$ steps to sort 2 element Array

RHS: $c \cdot 2 \cdot \log_2(2) = c \cdot 2 \cdot 1 = 2c$

$\Rightarrow$ Prove: LHS $\leq$ RHS $= 4 \leq 2c$

$= 2 \leq c$ // True, just select a $c \geq 2$

- **Inductive Step** (strong induction) $\Rightarrow$

Assume not just 1st prior case is true, but all prior cases are true.

$\Rightarrow$ if this was weak induction, we could only assume $T(n-1) \leq c \cdot n-1 \log(n-1)$

Assume: $T(x) \leq c \cdot x \log x$

$\forall n_0 \leq x \leq (n-1)$

↳ Base case ↳ 1 prior to what we are proving

Prove: $T(n) \leq c \cdot n \log n$

Idea

$\Rightarrow$ Assume prior cases are true, and using this $\Rightarrow$ prove next case is true

$\Rightarrow$ Proving every next case must be true

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

one of cases in assumption $2 \leq n/2 \leq (n-1)$

// $n_0 = 2$

$2 \leq n/2 \leq (n-1)$

↳ run into issues if $n_0 = 3, 4$ thus we would need to do additional base cases for those values. Otherwise, this is true which lets us use assumption

// goal = $n \log n$

$\Rightarrow T(n) = 2T(n/2) + n \leq 2 \cdot c \cdot \frac{n}{2} \log\left(\frac{n}{2}\right) + n$

// $T(n/2) \in X$

// subbed in $c \cdot x \log(x)$ for $T(n/2)$ as $\frac{n}{2}$ satisfies assumption

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

$= cn(\log_2 n - \log_2 2) + n$ ($\log_2 2 = 1$)

$= cn \log_2 n - cn + n$

$cn \log_2 n$ ✓

$-cn + n$ ⊗ // need to get rid of this

// $c \geq 2$

$\Rightarrow -2n + n = -n$

$\leq cn \log n + 0$ // adding 0 as this is greater than (-1)

□

Master Theorem // does not always work

This above method is fairly difficult. Is there an easier way?

Merge sort: $f(n) = n^{\log_b a}$ thus...

$\Rightarrow T(n) \in \Theta(n \cdot \log n)$

(1) **Divide:** Trivial.
(2) **Conquer:** Recursively sort 2 subarrays of size $n/2$
(3) **Combine:** Linear time key subroutine Merge

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

$a \geq 1$

$b > 1$

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

a = # of recursive calls

b ↳ size of recursion

$f(n)$ ↳ asymptotically positive

$f(n)$ cannot be kn

- $n^{\log_b a}$ = recursive runtime
- we are comparing $f(n)$ w/ our recursive runtime

$n^{\log_b a}$ => What is X?

$b^X = a$

Binary search:

(1) **Divide:** Check the middle element.
(2) **Conquer:** Recursively search 1 subarray.
(3) **Combine:** Trivial.

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

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

$\Rightarrow n^{\log_2 1} = n^0 = 1 \quad \therefore 2^0 = 1$

$\Rightarrow f(n) = = n^{\log_b a}$ // case 2

$\Rightarrow T(n) \in \Theta(\log n)$ & $f(n) \in \Theta(n^{\log_b a})$

Example 1: $T(n) = 4T(n/2) + \sqrt{n}$

$a = 4$
$b = 2$
$f(n) = \sqrt{n}$

$\Rightarrow n^{\log_b a} = n^{\log_2 4} = n^2$

$2^2 = 4$
$\log_2 4 = 2$

// case 1

$n^2 > \sqrt{n} = n^{1/2}$

$\Rightarrow f(n) \in O(n^{\log_b a - \varepsilon})$
$= n^{1/2} \in O(n^{2-\varepsilon})$ // Find $\varepsilon > 0$ that makes this true
$= n^{1/2} \in O(n^1)$ => let $\varepsilon = 1$

// guess trick
=> let $\varepsilon = 1.5$
=> $n^{1/2} \in O(n^{1/2})$

exact amount both are correct

let ε = anything to make statement true

Example 2: $T(n) = 4T(n/2) + n^3$

case 1: $f(n) = n^{\log_b a}$

case 2: $f(n)$ grows slower

case 3: $f(n)$ grows faster

Example 3: $T(n) = 4T(n/2) + \frac{n^2}{\log n}$

$a = 4,\ b = 2,\ f(n) = \frac{n^2}{\log n}$

$\Rightarrow n^{\log_b a} = n^{\log_2 4} = n^2$

$\Rightarrow n^2$ vs. $\frac{n^2}{\log n}$ // which faster?

$\Rightarrow f(n) \in O(n^{2-\varepsilon})$

$= \frac{n^2}{\log n} \in O\left(\frac{n^2}{n^\varepsilon}\right)$

// we want to find n^ε that is similar to $\log n$

$\Rightarrow$ turns out nothing we can do with ε will make the statement true

$n^{0.000002} > \log n$

$\Rightarrow$ None of the cases apply here, we would need to do a inductive proof here

let $n = 100$

$100^2 = 10{,}000$

$\frac{100^2}{\log(100)} = \frac{10{,}000}{2} = 5000$

$n^2 > \frac{n^2}{\log n}$ as $\log n$ in denom reduces amount

thus case 1 as $f(n)$ grows slower