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);
elseA[n/2] <= val
    //val is in second half of the array
    return binarySearch(A[n/2...n], val);
end if
```

Example:
Find 9 in the following array

{3 5 7 8 9 12 15}

What is difficult about finding the runtime of this algorithm?

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.
(3) **Combine** subproblem solutions.

Binary search:

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

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⌉]);
mergeSort(A[⌈n/2⌉ + 1 ... n]);
Merge the two sorted lists
return;
```

Merge sort:

(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?

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

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

Let $T(n)$ represent the runtime of mergeSort:

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

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

Recurrence for mergeSort:

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

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

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.

.

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 [illegible]

- $n^{\log_b a}$ = 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