Divide-And-Conquer With Searching

Make Every Algorithm Recursive For The Sake Of Algorithm Analysis Only

Binary Search

- Searching a “sorted” array for a value
  - Like looking up a dictionary for a word, or a phone book for a name
- Everyone is expected to write the binary search code (pseudocode or actual language) fluently both recursively and iteratively
  - Remember the “divide-and-conquer” nature
  - And also the “elimination” nature: You can eliminate half of the array once you compare with the middle value
  - What differences & benefits are there in each approach (recursive & iterative)?
    - Keep this question in mind when you experiment the code in the next slides

Recursive Binary Search

```
def binary_search_recursive(array, left, right, value):
    if left > right:
        return -1

    mid = (left + right) / 2
    if value == array[mid]:
        return mid
    if value > array[mid]:
        return binary_search_recursive(array, mid+1, right, value)
    # value < array[mid]
    return binary_search_recursive(array, left, mid-1, value)
```

Iterative Binary Search

```
def binary_search_iterative(array, value):
    left = 0
    right = len(array)-1

    while left <= right:
        mid = (left + right) / 2
        if value == array[mid]:
            return mid
        if value > array[mid]:
            left = mid + 1
        else:  # value < array[mid]
            right = mid – 1

    return -1  # No match
```

Analysis of Binary Search

- Best case: $T(n) = \Theta(1)$
  - Fixed # operations (1 mid calc op, 1 if, 1 return) when the mid entry is a hit
- Worst case
  - $T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(1)$
  - $T(0) = \Theta(1)$
- Easier to derive the recurrence from recursive code
- Fewer # steps with iterative code

```
def binary_search_iterative(array, value):
    left = 0
    right = len(array)-1

    while left <= right:
        mid = (left + right) / 2
        if value == array[mid]:
            return mid
        if value > array[mid]:
            left = mid + 1
        else:  # value < array[mid]
            right = mid - 1
    return -1  # No match
```

```
def binary_search_recursive(array, left, right, value):
    if left > right:
        return -1  # No match
    mid = (left + right) / 2
    if value == array[mid]:
        return mid
    if value > array[mid]:
        return binary_search_recursive(array, mid+1, right, value)
    # value < array[mid]
    return binary_search_recursive(array, left, mid-1, value)
```

Recursive vs. Iterative Binary Search

- Easier to write recursive code and derive recurrence from it
- Call stack overhead in recursive code : $O(\log n)$ space complexity
- Harder to write iterative code and analyze it
- Faster execution (constant factor speed up) with iterative code, no call stack overhead ($O(1)$ space complexity)
- Quite common to start out writing recursive implementation, then translate it to iterative code for optimization
- Possible variation: What if we need to return the index of the first match when there are multiple matches?

Sidebar: Binary Divide-And-Conquer For Searching Unsorted Array

- With an unsorted array, we can't eliminate the problem size by half, like with a sorted array and binary search
- Had to do sequential search, eliminating problem size by one at a time
- Can we do the binary divide-and-conquer with unsorted array as well?
  - Yes, we can. Can you code that?
  - But we won't get any benefit, as our recurrence will be:
    - $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(1) \rightarrow T(n) = \Theta(n)$, not $\Theta(\log n)$
- Can we derive general solution on $T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^c)$?

More Divide-And-Conquer Examples

Interesting Divide-And-Conquer Algorithms

Maximum Subarray Problem

- Given an example array

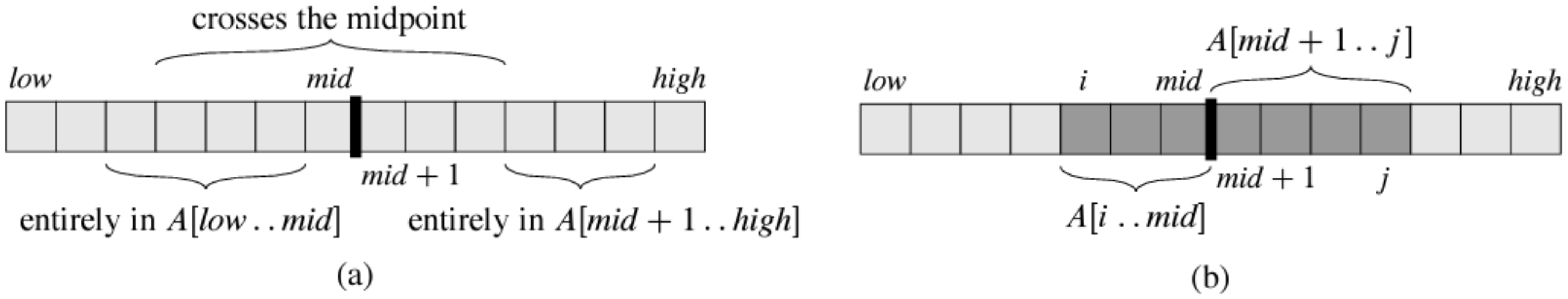
  A=[13,-3,-25,20,-3,-16,-23,18,20,-7,12,-5,-22,15,-4,7]
  - What is the subarray whose sum is the maximum of all subarray sums?
  - In this example, it's [18,20,-7,12] with sum 43.
    - You can confirm this yourself by whatever means
  - It doesn't make much difference between finding the max subarray sum (43) and finding the subarray itself ([18,20,-7,12]). Think about why.
  - Interesting only when there are negative numbers in the array.
  - Read CLRS 4.1 for a motivating application of this problem
    - Stock trading to maximize gain when daily change amounts are known.
      - Not a real stock trading technique!

Naïve, Brute-Force Solutions

- Evaluate sums of all A[i..j] for any possible i & j, and find the max.
    - max_subarray_sum = -infinity (or A[1])
    - for i=1 to n (assuming 1-starting array indexing)
        - for j=i to n
            - subarray_sum=0
            - for k=i to j
                - subarray_sum += A[k]
            - If subarray_sum > max_subarray_sum,
                - max_subarray_sum = subarray_sum
    - return max_subarray_sum
- $\Theta(n^3)$: Really naïve, repeating same summation many times
- $\Theta(n^2)$: By separating out summations, storing them in a 2D array, then doing comparisons, we can achieve this improvement.

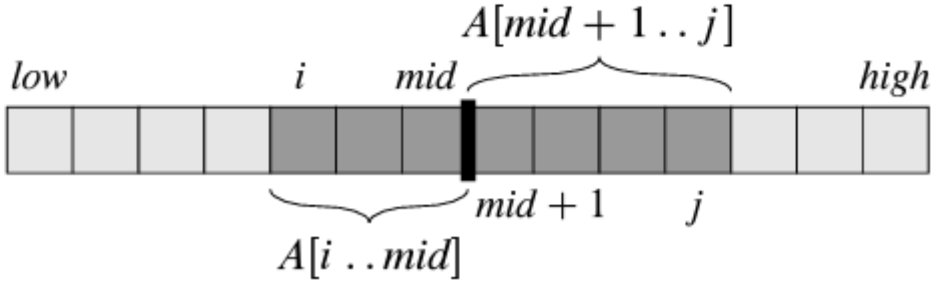
Can We Do Any Better?

- How about binary divide-and-conquer, like earlier?
  - Max subarray sum of A[low..high] is the maximum of:
    - Max subarray sum of A[low..mid] ← Recursively computed
    - Max subarray sum of A[mid+1..high] ← Recursively computed
    - Max of sums of subarrays straddling mid
      - Easier than original problem because this problem is constrained.



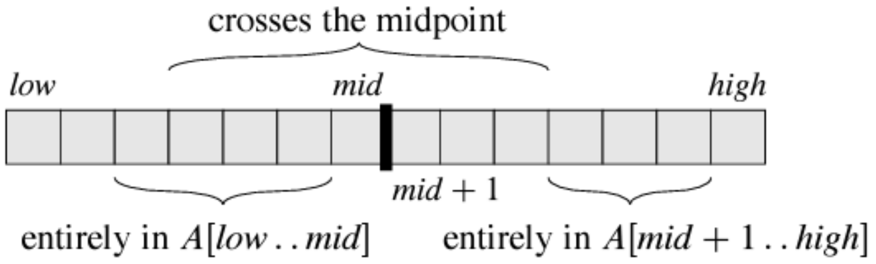
CLRS Fig. 4.4

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum = −∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = −∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```



(b)

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```



(a)

CLRS pp. 72

Analysis of Divide-And-Conquer Max Subarray

- $T(1) = \Theta(1)$: Base case. Recursive case is:
- $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + T_{crossing}(n) + \Theta(1)$ where:
  - $T_{crossing}(n) = \Theta(n)$
- $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$: Exactly the same as merge sort
- $T(n) = \Theta(n \log n)$
- Achieved $\Theta(n^2)$ to $\Theta(n \log n)$ improvement
  - Actually we can do better and achieve linear time ($\Theta(n)$)
    - Exercise 4.1-5 in pp. 75..
    - It's not a divide-and-conquer solution, though. It's rather a clever intuition.

Strassen's Matrix Multiplication Algorithm (CLRS 4.2)

- Multiplying 2 $n \times n$ matrices
  - Study Appendix D if you are not familiar with matrices and operations
- Simple straightforward algorithm, giving $\Theta(n^3)$

```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

CLRS pp.75

Simple D&C Matrix Multiplication

- Partition each of A, B, and C into 4 quarters:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{4.9}$$

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}. \tag{4.10}$$

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \tag{4.11}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \tag{4.12}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \tag{4.13}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \tag{4.14}$$

- Can we reduce # multiplications some way?

Strassen's Improvement

- Not sure how Strassen found this, but he observed that, by letting:

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} , \\
S_2 &= A_{11} + A_{12} , \\
S_3 &= A_{21} + A_{22} , \\
S_4 &= B_{21} - B_{11} , \\
S_5 &= A_{11} + A_{22} , \\
S_6 &= B_{11} + B_{22} , \\
S_7 &= A_{12} - A_{22} , \\
S_8 &= B_{21} + B_{22} , \\
S_9 &= A_{11} - A_{21} , \\
S_{10} &= B_{11} + B_{12} .
\end{aligned}
$$

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} , \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} , \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} , \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} , \\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} , \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} , \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} .
\end{aligned}
$$

Analysis of Strassen's MM Version

- Submatrix C_{ij} can be represented as follows:

  $C_{11} = P_5 + P_4 - P_2 + P_6$. $C_{12} = P_1 + P_2$, $C_{21} = P_3 + P_4$ $C_{22} = P_5 + P_1 - P_3 - P_7$,

- Algebraic proofs shown in CLRS pp. 81
- We've now got 7 $\frac{n}{2} \times \frac{n}{2}$ multiplications and 18 additions
  - 18 additions are still $\Theta(n^2)$.
  - Thus new recurrence is

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$$

  - Solution to this recurrence is $T(n) = \Theta\left(n^{\log_2 7}\right) \cong \Theta(n^{2.81})$
    (using master theorem, which will be presented later)

Methods For Solving Recurrences

Tools To Analyze Divide-And-Conquer Algorithms

Substitution Method For Solving Recurrences

- Given a recurrence,
  - Take a guess of the solution
    - Not easy, requiring intuitions, experiences
  - Then prove it by induction
    - Not easy either, but could be routine

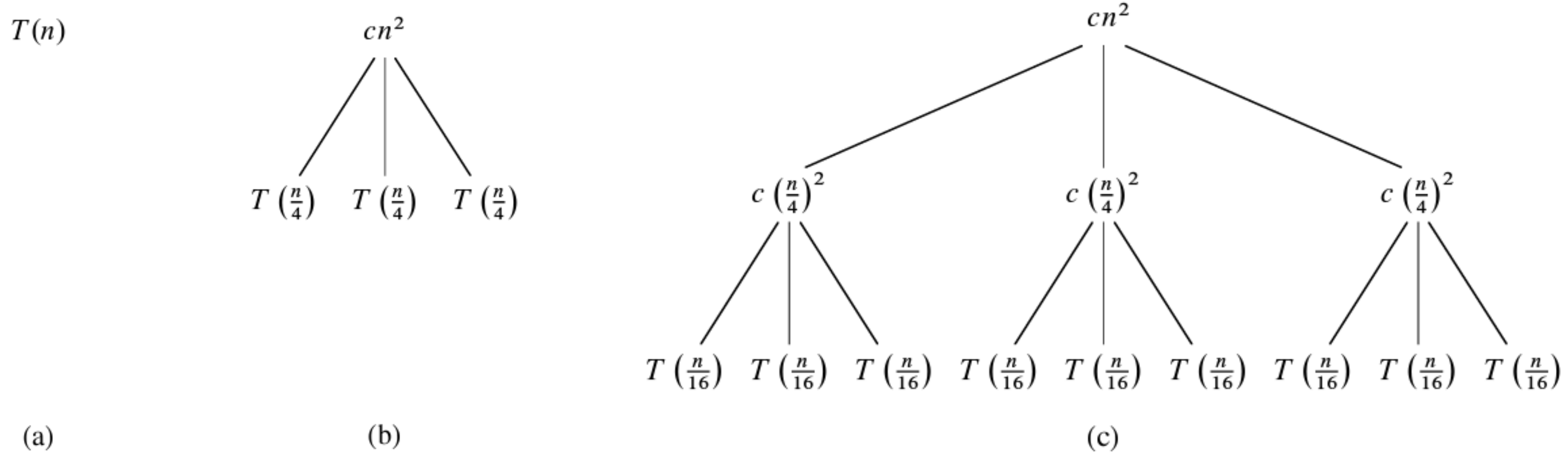
- E.g, $T(n) = 2T(\lfloor n/2 \rfloor) + n, T(1) = 1$
  - Guess that $T(n) = O(n \log_2 n)$
    - How? … From previous experiences?
  - Need to prove $T(n) \leq cn \log_2 n$ for some c (we get to choose) and for all $n \geq n_0$ (we get to choose n_0 as well).
    - Above statement is just the definition of $T(n) = O(n \log_2 n)$
    - We prove this by mathematical induction, especially strong induction

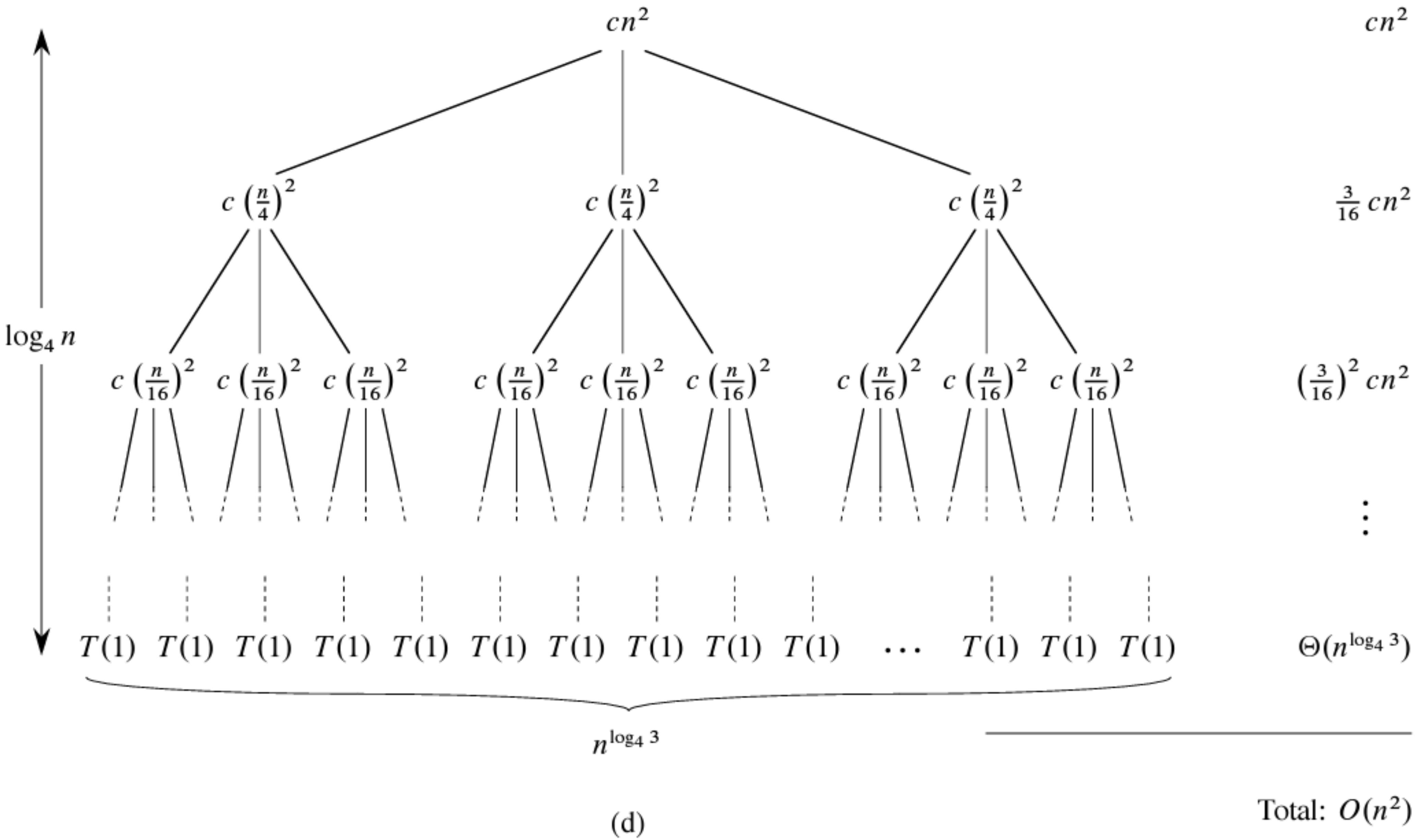
Proof By Induction Example

- Induction step
  - Hypothesis: Assume $T(m) \leq cm \log_2 m$ for all $m < n$
  - Prove: $T(n) \leq cn \log_2 n$
    - Again, don't forget that we get to choose c!
- Induction base
  - May need to choose a different starting value of n
    - Because $T(1) = 1$ may not meet the inequality
    - Remember we get to choose n_0, so don't be limited by the given base case.

Recursion Tree Method For Solving Recurrences

- Making a good guess for substitution method feels like a black magic!
- Visually expand original $T(n)$, using tree structure all the way down to base case, and reason about the outcome
- E.g., $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$,





cn^2
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$\log_4 n$
$T(1)$
$n^{\log_4 3}$
cn^2
$\frac{3}{16}\, cn^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$\Theta(n^{\log_4 3})$
Total: $O(n^2)$

(d)

Master Method For Solving Recurrences

- "Cookbook" method for solving recurrences of the form
$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$
- Proof is presented in CLRS 4.6, left as optional (not covered in class)
- Intuitive understanding: We compare $f(n)$ with $n^{\log_b a}$
  - If $f(n)$ is smaller (polynomially & asymptotically), then $T(n) = \Theta(n^{\log_b a})$.
  - If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$
  - If $f(n)$ is bigger with some more conditions (see Theorem 4.1), then $T(n) = \Theta(f(n))$

Master Method Examples

- $T(n) = 2T\left(\frac{n}{2}\right) + cn$ (merge sort)
  - $f(n) = cn$ and $n^{\log_b a} = n^{\log_2 2} = n^1 = n$
    - Which makes $f(n) = \Theta(n^{\log_b a})$, thus second case, and we get $T(n) = \Theta(n \log n)$.
- $T(n) = 8T\left(\frac{n}{2}\right) + cn^2$ (ordinary div.-and-conquer matrix mult.)
  - $f(n) = cn^2$ and $n^{\log_b a} = n^{\log_2 8} = n^3$
    - Which makes $f(n)$ smaller than $n^{\log_b a}$, thus first case, and we get $T(n) = \Theta(n^3)$.
- $T(n) = 7T\left(\frac{n}{2}\right) + cn^2$ (Strassen's div.-and-conquer matrix mult.)
  - $f(n) = cn^2$ and $n^{\log_b a} = n^{\log_2 7}$
    - Which makes $f(n)$ smaller than $n^{\log_b a}$, thus first case, and we get $T(n) = \Theta(n^{\log_2 7})$.