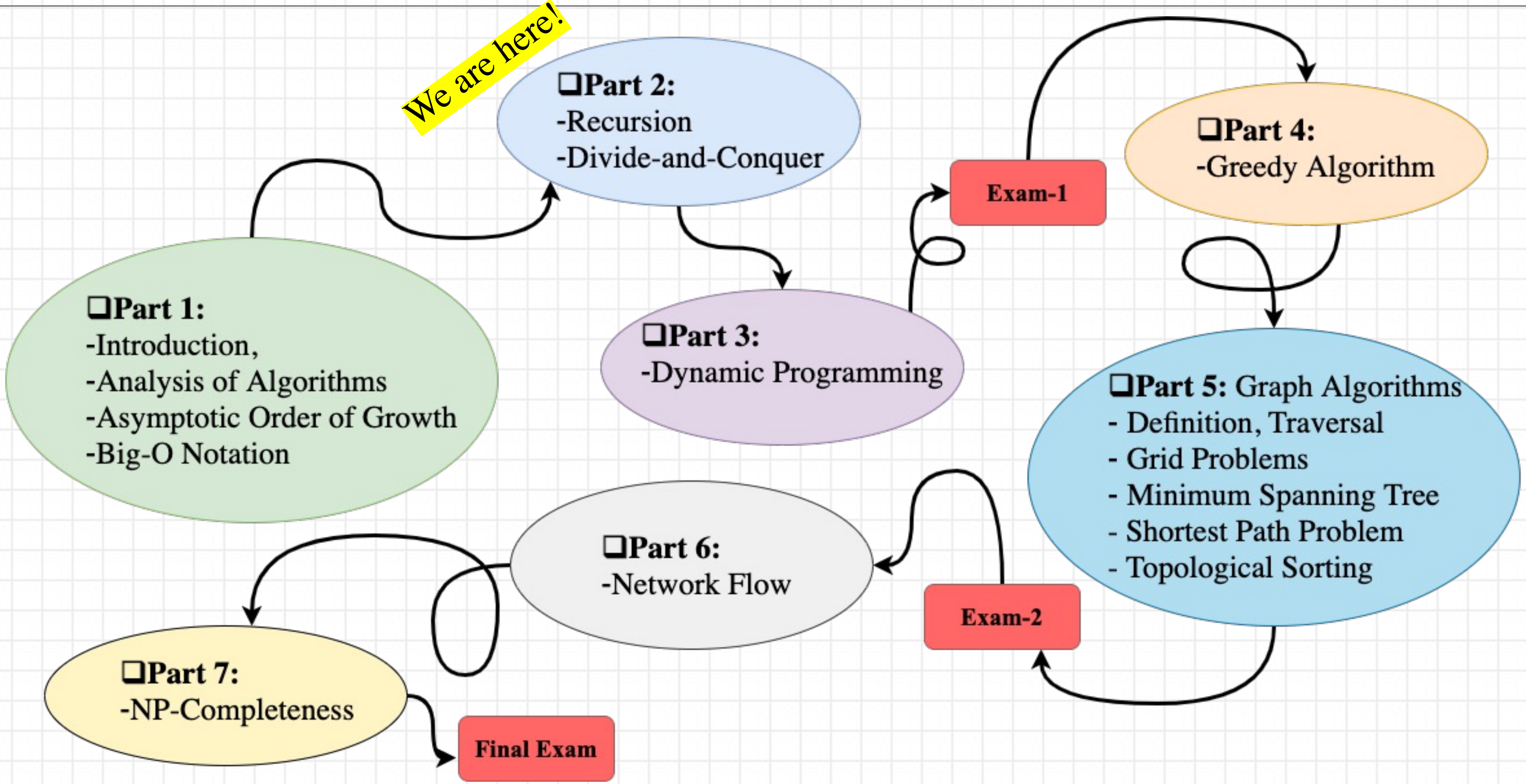
CS-3510:
Design and Analysis of Algorithms

Divide-and-Conquer I

Instructor: Shahrokh Shahi

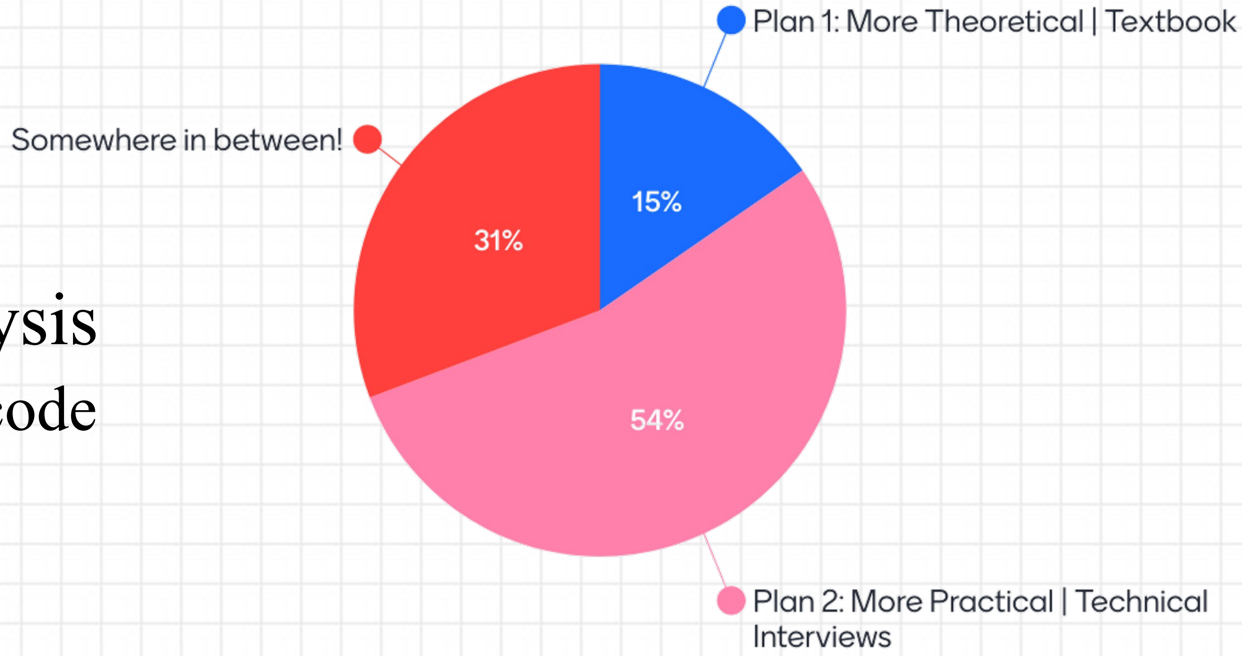
College of Computing
Georgia Institute of Technology
Summer 2022

Roadmap


We are here!
❑Part 1:
-Introduction,
-Analysis of Algorithms
-Asymptotic Order of Growth
-Big-O Notation
❑Part 2:
-Recursion
-Divide-and-Conquer
❑Part 3:
-Dynamic Programming
Exam-1
❑Part 4:
-Greedy Algorithm
❑Part 5: Graph Algorithms
- Definition, Traversal
- Grid Problems
- Minimum Spanning Tree
- Shortest Path Problem
- Topological Sorting
Exam-2
❑Part 6:
-Network Flow
❑Part 7:
-NP-Completeness
Final Exam

Previous Lecture (1/3)

- Introduction, course logistic
  - Course website
  - Lecture will be streamed / recorded using Zoom and will be accessible on Canvas
  - Course content
    - Poll result



- Algorithms; Design and Analysis
  - Designing, describing, pseudocode
  - Correctness
  - Time and space complexity

Previous Lecture (2/3)

- Review of running time and space complexity
  - Model of computation: random-access machine (RAM)
    - Single processor, no concurrency, supports common instructions
  - Runtime: number of steps taken by an algorithm, given the input size
    - Best-case, Worst-case, Average-case
  - Time complexity:
    - Providing a number (function) T(n), where is the input size
    - T(n): max amount time taken on any input of size n ≊ worst-case runtime
    - We mostly care about the rate of growth, i.e., how the runtime is scaling up w.r.t. the input size
    - Asymptotic analysis
      - Ω, O, and Θ notations: lower bound, upper bound, tight bound
    - Time complexity → rate of growth of the worst-case runtime → upper bound → Big-O
    - $T(n) \in O(g(n))$

Previous Lecture (3/3)

- Search problem
  - Linear search, $O(n)$
  - Binary search $O(\log\ n)$
    - Recursive algorithm
    - Divide-and-Conquer paradigm

A Note about Recursive Algorithms

- In general, recursive algorithms can be used in various setups:
  - Backtracking
    - Ex. Enumerating all subsets of a given set or array
    - Usually (not always!), in these cases we can expect an exponential runtime $O(a^n)$, where a is the number of possible options to choose at each step which is equal to the number branches after each node in the recursion tree.
  - Divide-and-Conquer (D&C)
  - Dynamic programming (DP)
  - Traversing a graph or tree using the depth-first search (DFS) approach

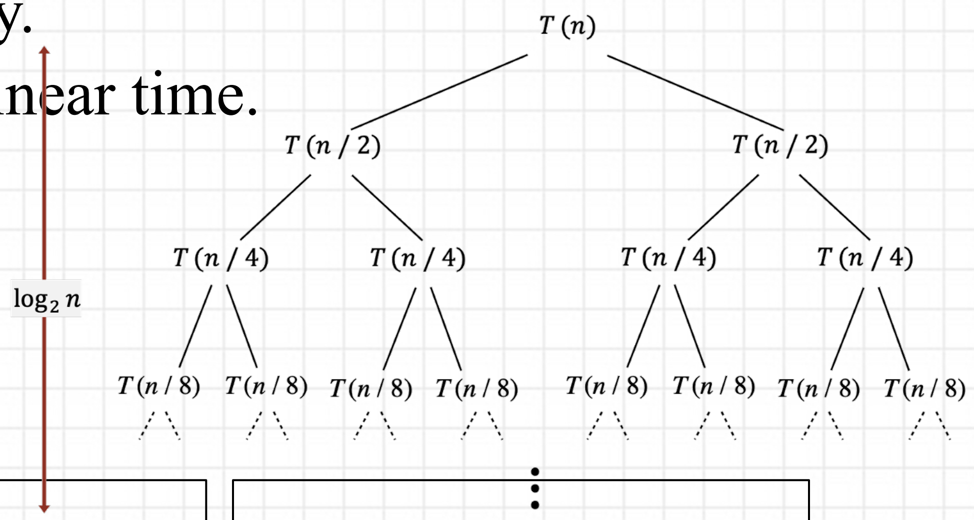
Divide-and-Conquer (D&C)

- Main idea:
  - Break the problem into smaller pieces to solve them easier. Then, combine the solution of these sub-problems to form the overall solution.

- Main steps
  - Divide up problems into several subproblems (of the same type)
  - Solve (conquer) each subproblem (usually recursively)
  - Combine the solutions

Divide-and-Conquer (D&C)

- Main steps
  - Divide up problems into several subproblems (of the same type).
  - Solve (conquer) each subproblem (usually recursively).
  - Combine the solutions.
- Most common framework
  - Divide the problem of size n into two subproblems of size $n/2$ in linear time
  - Solve (conquer) the two subproblems recursively.
  - Combine two solutions into overall solution in linear time.



Divide-and-Conquer (D&C)

- Let's start with some examples!
  - **Binary-search** (previous lecture) — Search Algorithm
  - **Merge-sort** (this lecture) — Sorting Algorithm
  - **Quick-sort** — Sorting Algorithm
  - **Matrix multiplication**
  - **Closest pair of points**

D&C Example: Binary-search (revisit)

- Search Problem: Given a sorted array, including integer numbers, and a target number, design an algorithm which returns True if the target number is in the given array, and False otherwise.
- Binary-search:
  - At each step compare the mid element with the target:
    - if mid == target: return True
    - if mid < target: discard the left sub-array and continue to search the right sub-array
    - if mid > target: discard the right sub-array and continue to search the left sub-array

D&C Example: Binary-search (revisit)

- Binary-search:
- At each step compare the mid element of with the target:
  - if mid == target: return True
  - if mid < target:
    discard the left sub-array and continue to search the right sub-array
  - if mid > target:
    discard the right sub-array and continue to search the left sub-array

- Time complexity?
  - Recursive Algorithms
    - Recursion tree
    - Substitution | Guess and prove by induction
    - Master theorem (this lecture!)

Algorithm 3: Binary Search (Recursive)

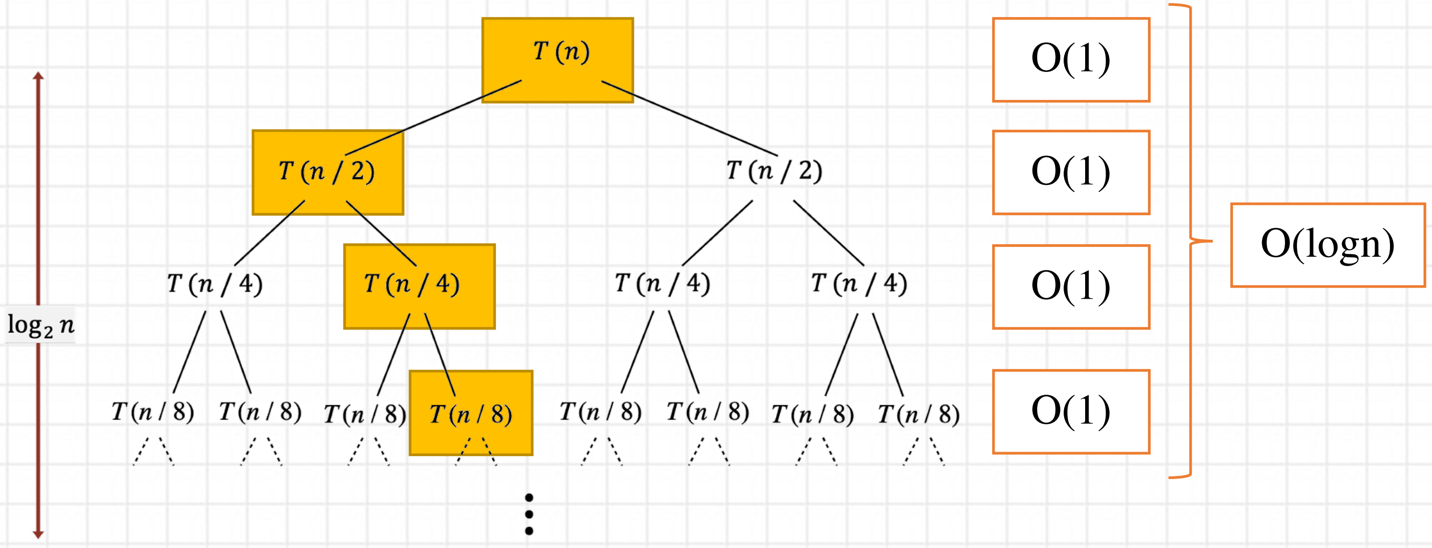
Input: $A = \{a_1, a_2, ..., a_n\}, k, lo = 1, hi = n$
Result: $True$ or $False$

```
// base case
if (lo > hi) then
    return False
end

// recurrence relation
mid ← [(lo + hi)/2]
if (a_mid == k) then
    // the target is in the array
    return True
else if (a_mid < k) then
    return BinarySearch(A, k, mid+1, hi)
else
    return BinarySearch(A, k, lo, mid-1)
end
```

D&C Example: Binary-search (revisit)

- Binary-search:
- At each step, compare the mid element of with the target:
  - if mid == target: return True
  - if mid < target: discard the left sub-array and continue to search the right sub-array
  - if mid > target: discard the right sub-array and continue to search the left sub-array



The recurrence: $T(n) = \begin{cases} 1 & , n = 1 \\ 1 + T(\frac{n}{2}), & n > 1 \end{cases}$

$$T(n) = 1 + T\left(\frac{n}{2}\right) = 1 + 1 + T\left(\frac{n}{4}\right) = 1 + 1 + 1 + T\left(\frac{n}{8}\right)$$

$$= \overbrace{1 + 1 + \ \ldots + 1}^{1+\log(n)} \in O(\log(n))$$

D&C Example: Merge-sort

- Sorting Problem: Given an input of n elements, re-arrange the elements in ascending (or descending) order.

- Applications:
  - Direct: Sort a list of numbers, names, etc.
  - Indirect: Sorting can make other tasks easier.
    - Ex. Given an array of n distinct integers, find three that sum to 0.
      - Brute force: $O(n^3)$
      - Sort the array first, then run two-sum algorithm on the sorted array: $O(n^2)$

D&C Example: Merge-sort

- Sorting Problem: Given an input of n elements, re-arrange the elements in ascending (or descending) order.

- Algorithms:

Array Sorting Algorithms

Algorithm	Running time			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

https://www.bigocheatsheet.com/
http://www.cs3510.com/resources/

D&C Example: Merge-sort

- Sorting Problem: Given an input of n elements, re-arrange the elements in ascending (or descending) order.

- Algorithms:

Array Sorting Algorithms

Algorithm	Running time			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Ex. of D&C: $\Theta(n \log n)$

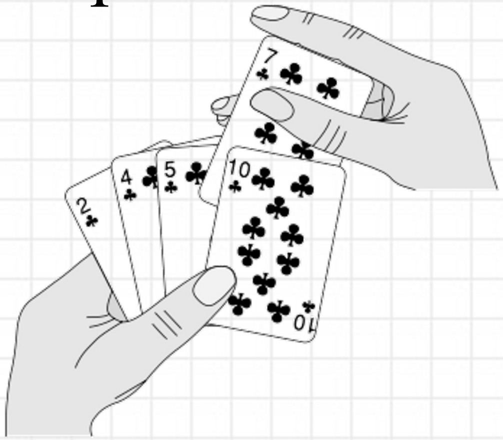
Ex. of Brute force: $\Theta(n^2)$

FWIW, Python built-in sort function

https://www.bigocheatsheet.com/
http://www.cs3510.com/resources/

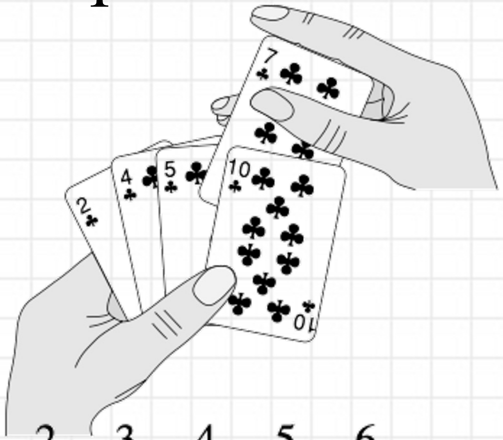
Extra Slide: Insertion-sort

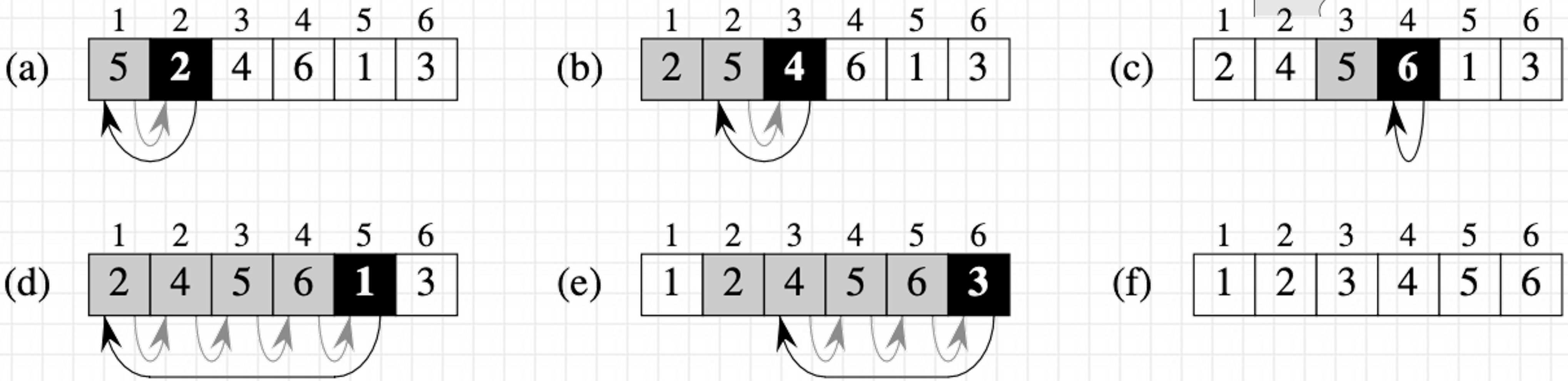
- Insertion-sort and Bubble-sort are two well-known examples of brute-force sorting algorithm
- Insertion sort:
  - Works the way people sort a hand of playing cards.
  - We start with an empty hand
  - The cards face down on the table.
  - We then remove one card at a time from the table and insert it into the correct position.
  - To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left



Extra Slide: Insertion-sort

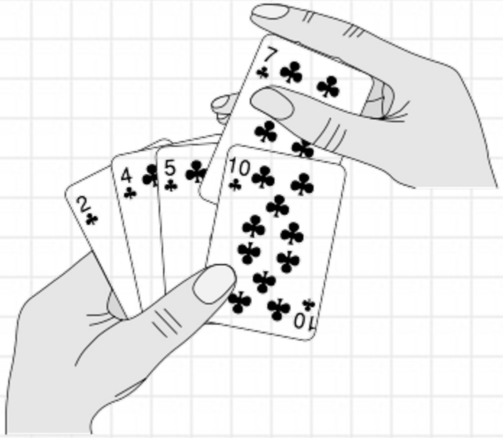
- Insertion-sort and Bubble-sort are two well-known examples of brute-force sorting algorithm
- Insertion sort:
- Ex. Input: A = [5, 2, 4, 6, 1, 3]





Extra Slide: Insertion-sort

- Insertion-sort and Bubble-sort are two well-known examples of brute-force sorting algorithm
- Insertion sort:



```
INSERTION-SORT(A)
for j ← 2 to length[A]
      do key ← A[j]
         ▷ Insert A[j] into the sorted sequence A[1 . . j − 1].
         i ← j − 1
         while i > 0 and A[i] > key
             do A[i + 1] ← A[i]
                i ← i − 1
         A[i + 1] ← key
```

- Demo code

Time complexity: $O(n^2)$
Space complexity: $O(1)$

D&C Example: Merge-sort

- Now, let's back to our D&C business and see how merge-sort works. Then, we can compare its performance with a brute force sorting algorithm like insertion sort.
- From a few slides ago:

- Main steps
  - Divide up problems into several subproblems (of the same type).
  - Solve (conquer) each subproblem (usually recursively).
  - Combine the solutions.

- Most common framework
  - Divide the problem of size n into two subproblems of size $n/2$ in linear time
  - Solve (conquer) the two subproblems recursively.
  - Combine two solutions into overall solution in linear time.

D&C Example: Merge-sort

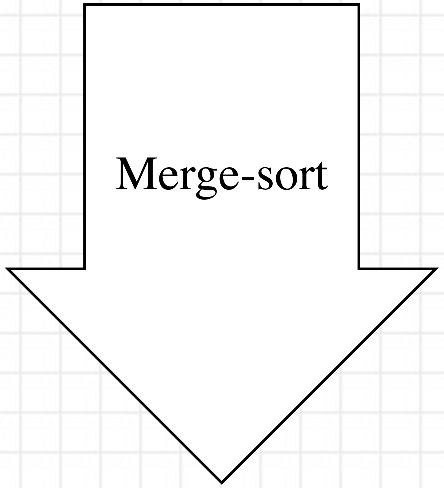
- Main steps
  - Divide up problems into several subproblems (of the same type).
  - Solve (conquer) each subproblem (usually recursively).
  - Combine the solutions.
- Most common framework
  - Divide the problem of size n into two subproblems of size $n/2$ in linear time
  - Solve (conquer) the two subproblems recursively.
  - Combine two solutions into overall solution in linear time.

- Merge-sort:
  - Divide: Divide the array into two halves
  - Conquer: Sort each half (by recursively executing merge-sort on each half)
  - Combine: Merge two halves to make a sorted array.

D&C Example: Merge-sort

- Ex. A = [39, 28, 40, 2, 8, 79, 11]

39	28	40	2	8	79	11

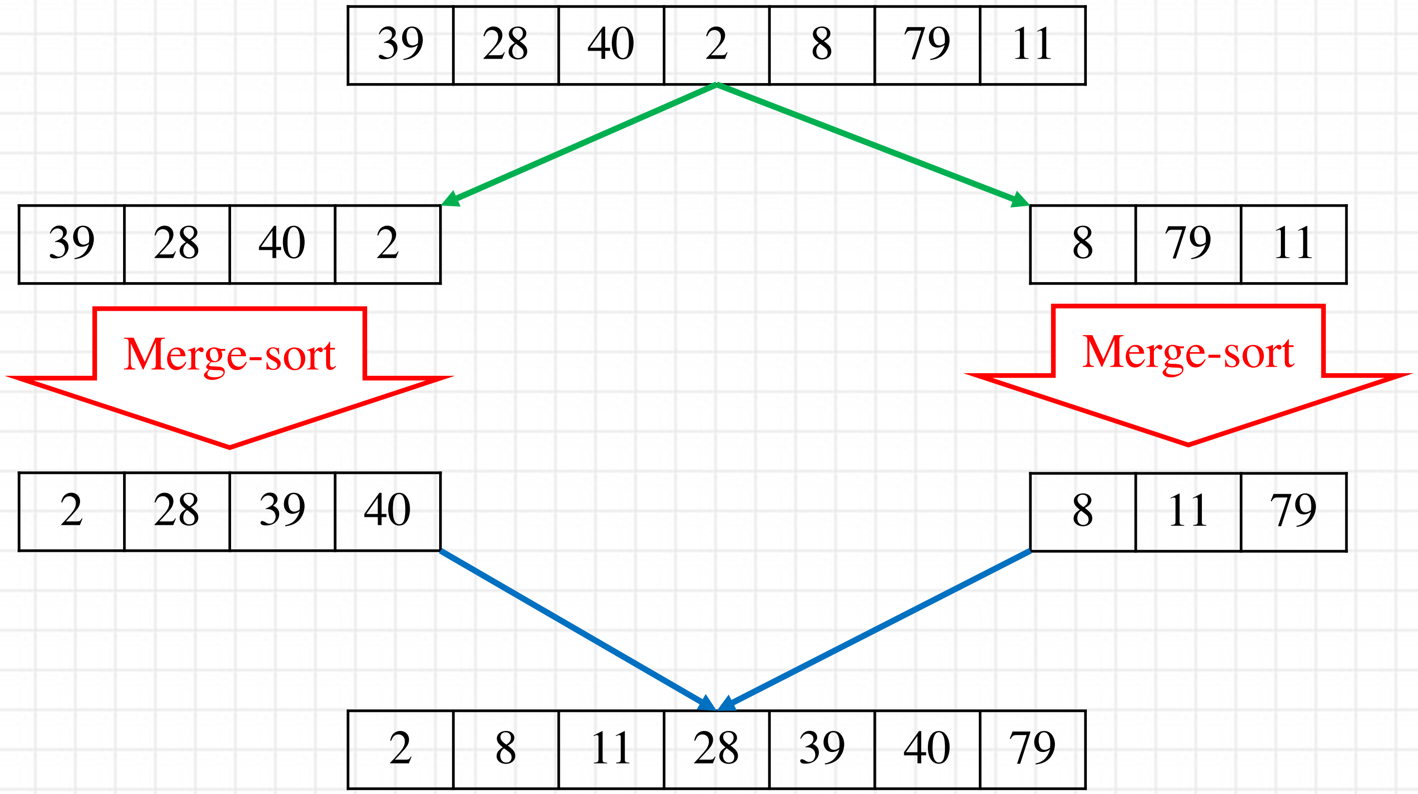


- <u>Divide</u>:
  Divide the array into two halves
- <u>Conquer</u>:
  Sort each half (by recursively executing merge-sort on each half)
- <u>Combine</u>:
  Merge two halves to make a sorted array

2	8	11	28	39	40	79

D&C Example: Merge-sort

- Ex. A = [39, 28, 40, 2, 8, 79, 11]

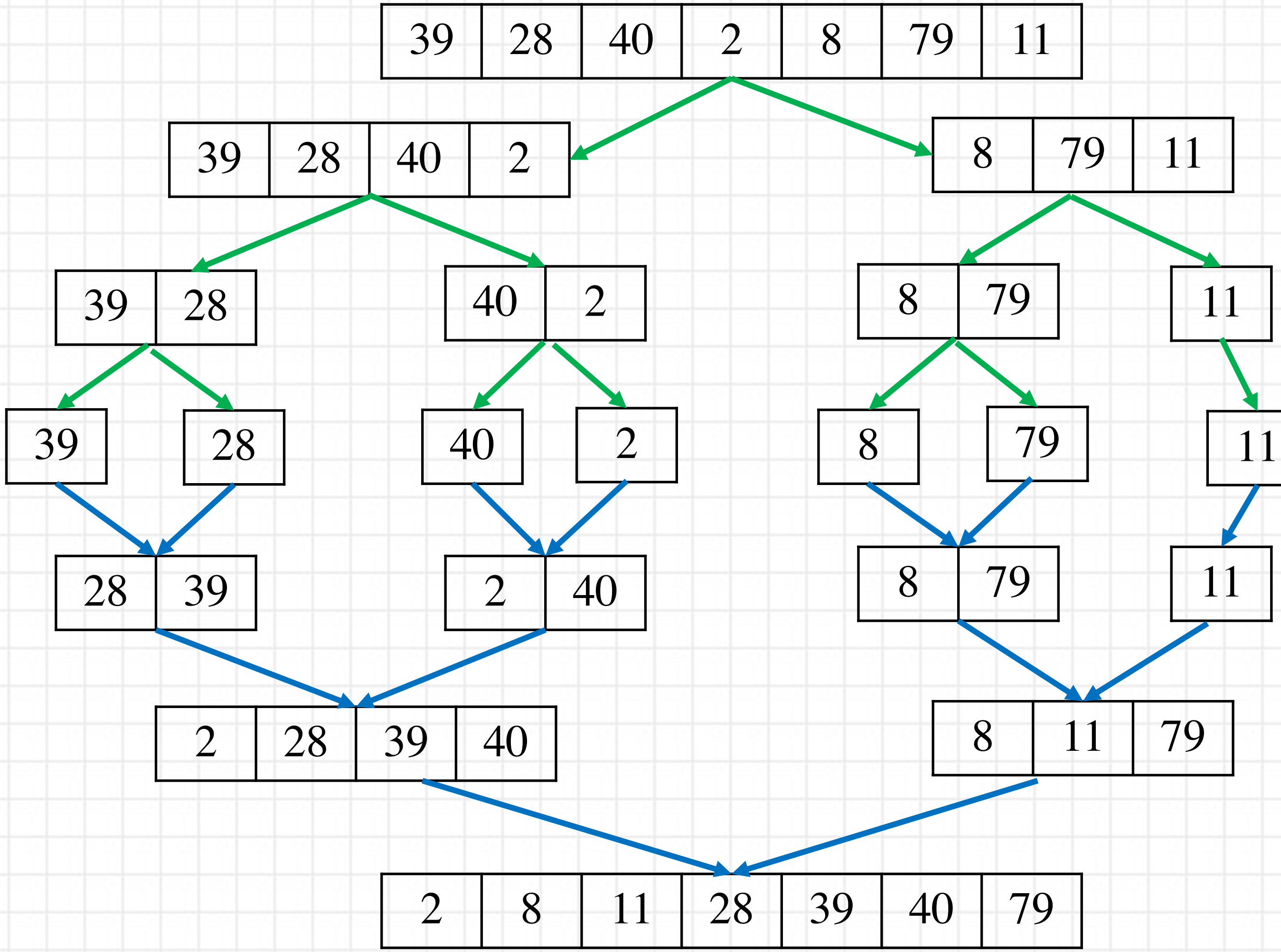


Divide: Divide the array into two halves

Conquer: Sort each half (by recursively executing merge-sort on each half)

Combine: Merge two halves to make a sorted array.

D&C Example: Merge-sort



Divide: Divide the array into two halves

Conquer: Sort each half (by recursively executing merge-sort on each half)

Combine: Merge two halves to make a sorted array.

D&C Example: Merge-sort

MERGE-SORT(L)

IF (list L has one element)

RETURN L.

Divide the list into two halves A and B.

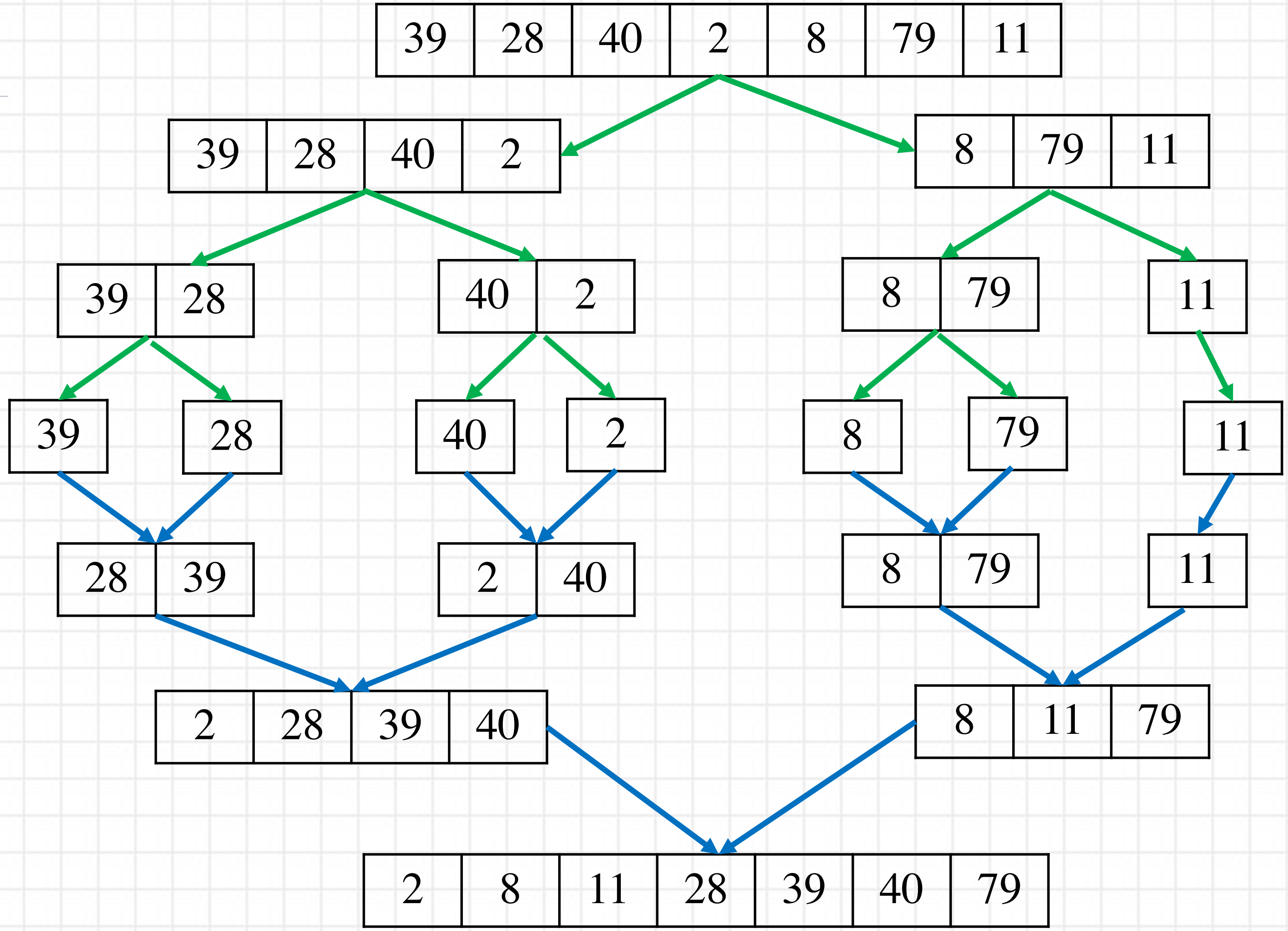
$A \leftarrow$ *MERGE-SORT*(A).

$B \leftarrow$ *MERGE-SORT*(B).

$L \leftarrow$ *MERGE*(A, B).

RETURN L.

Demo code



D&C Example: Merge-sort

- Merge-sort:
  - Divide: Divide the array into two halves
  - Conquer: Sort each half (by recursively executing merge-sort on each half)
  - Combine: Merge two halves to make a sorted array.

- Time complexity?

MERGE-SORT(*L*)

IF (*list L has one element*)

  RETURN *L*.

Divide the list into two halves *A* and *B*.

$A \leftarrow$ MERGE-SORT(A).

$B \leftarrow$ MERGE-SORT(B).

$L \leftarrow$ MERGE(A, B).

RETURN *L*.

D&C Example: Merge-sort

- Merge-sort:
  - Divide: Divide the array into two halves
  - Conquer: Sort each half (by recursively executing merge-sort on each half)
  - Combine: Merge two halves to make a sorted array.

- Time complexity?
  - Recursive Algorithms
    - Recursion tree
    - Substitution | Guess and prove by induction
    - Master theorem

MERGE-SORT(L)

IF (list L has one element)

  RETURN L.

Divide the list into two halves A and B.

$A \leftarrow$ MERGE-SORT(A). $\longleftarrow$ $T(n/2)$

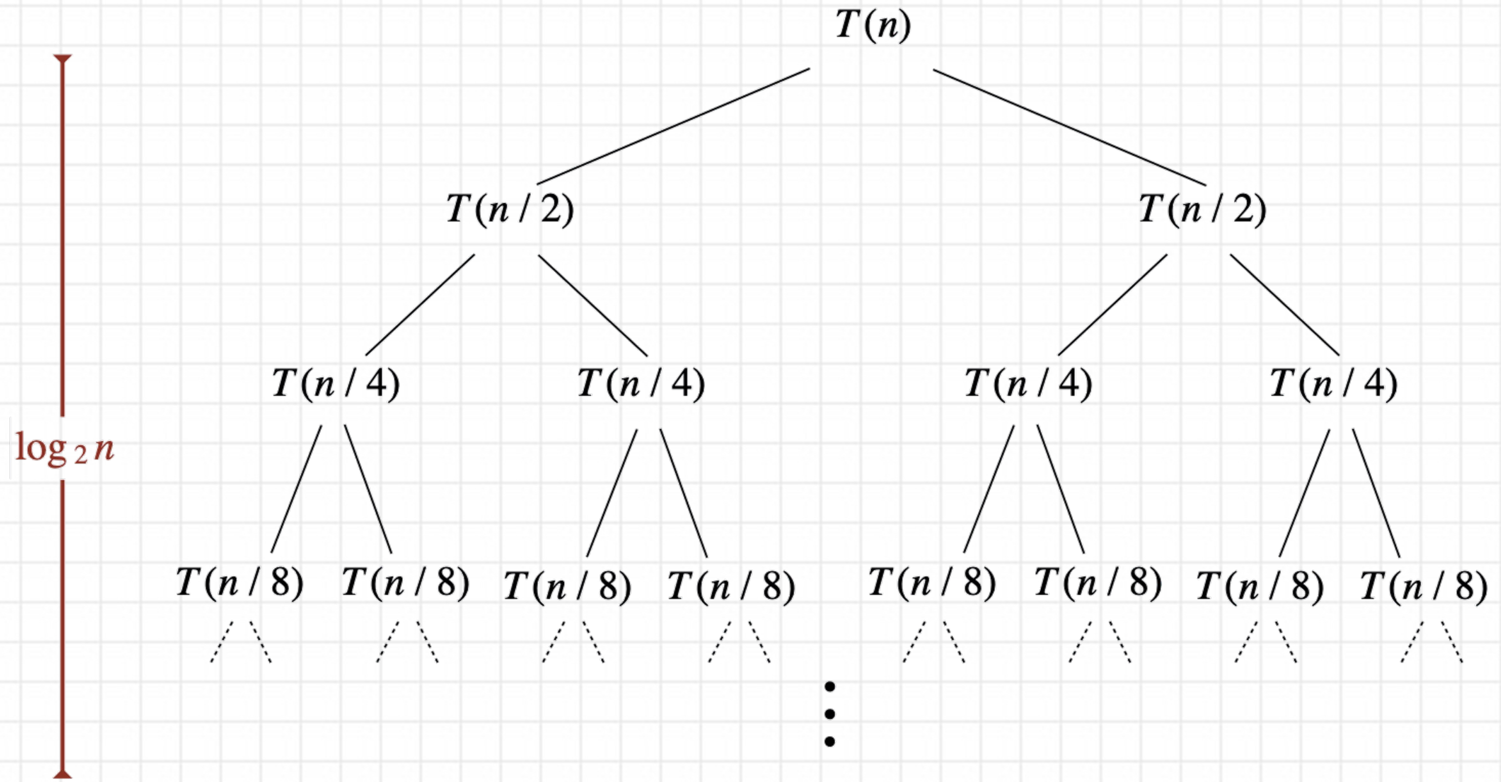
$B \leftarrow$ MERGE-SORT(B). $\longleftarrow$ $T(n/2)$

$L \leftarrow$ MERGE(A, B). $\longleftarrow$ $\Theta(n)$

RETURN L.

D&C Example: Merge-sort

- Recursion tree



$n = n$

$2\,(n/2) = n$

$4\,(n/4) = n$

$8\,(n/8) = n$

$\vdots$

$T(n) = n \log_2 n$

D&C Example: Merge-sort

- Recurrence
  - $T(n)$ = max number of compares to merge-sort a list of length n.

$$T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n & \text{if } n > 1 \end{cases}$$

  - Solution: $T(n)$ is $O(n\log_2 n)$
  - Proof by induction

D&C Example: Merge-sort

- Recurrence
  - Simplifying assumption: n is power of 2. Then,

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

  - Proof by induction:
    - Base case: when $n = 1$, $T(1) = 0 = n \log_2 n$.
    - Inductive hypothesis: assume $T(n) = n \log_2 n$.
    - We need to show that $T(2n) = 2n \log_2 (2n)$.

D&C Example: Merge-sort

- Recurrence
- Simplifying assumption: n is power of 2. Then,

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

- Proof by induction:
  - Base case: when $n = 1$, $T(1) = 0 = n \log_2 n$.
  - Inductive hypothesis: assume $\mathrm{T(n) = n \log_2 n}$.
  - We need to show that $T(2n) = 2n \log_2 (2n)$.

recurrence

$$\begin{aligned} T(2n) &= 2\,T(n) + 2n \\ &= 2\,n \log_2 n + 2n \\ &= 2\,n\,(\log_2 (2n) - 1) + 2n \\ &= 2\,n \log_2 (2n). \quad \blacksquare \end{aligned}$$

inductive hypothesis

Master Theorem

- Goal. Recipe for solving common divide-and-conquer recurrences,

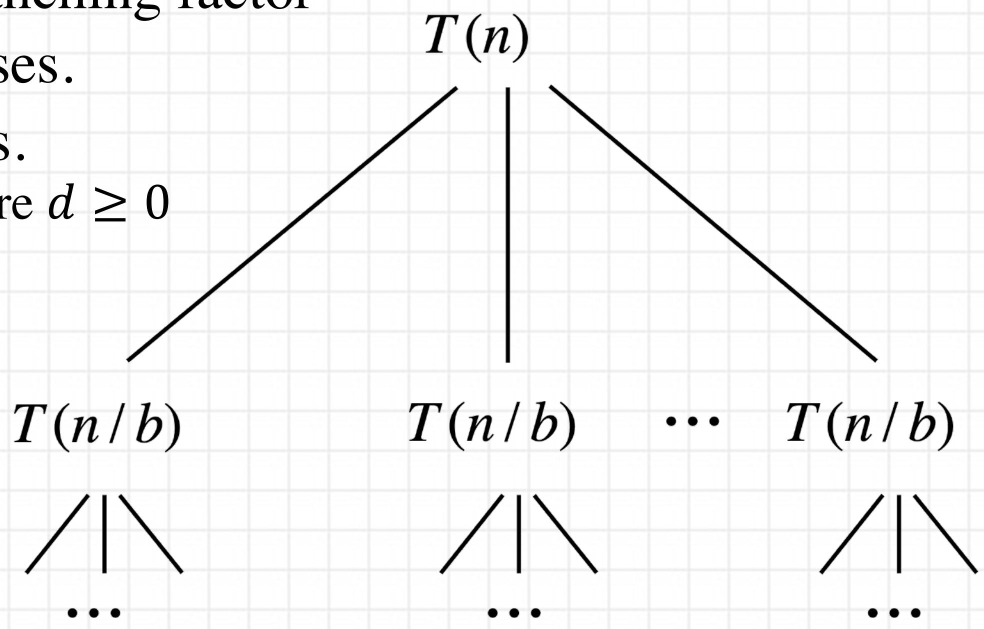
$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

where $T(0) = 0$ and $T(1) = \Theta(1)$.

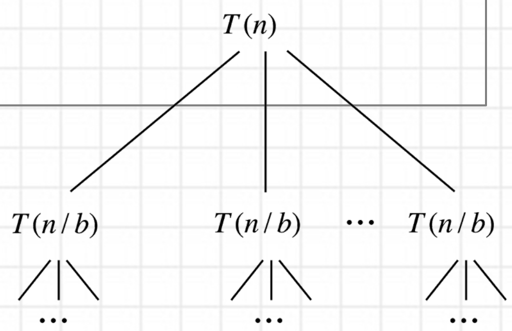
- $a \geq 1$ is the number of subproblems, also known as “branching factor”
- $b \geq 2$ is the factor by which the subproblem size decreases.
- $f(n) \geq 0$ is the work to divide and combine subproblems.
  - $f(n)$ usually takes polynomial time, i.e., $f(n)$ is $\Theta(n^d)$, where $d \geq 0$

Note:

- a^i = number of subproblems at level i
- $k = \log_b n$ levels, i.e., the depth of the recursion tree
- $\frac{n}{b^i}$ = size of subproblem at level i



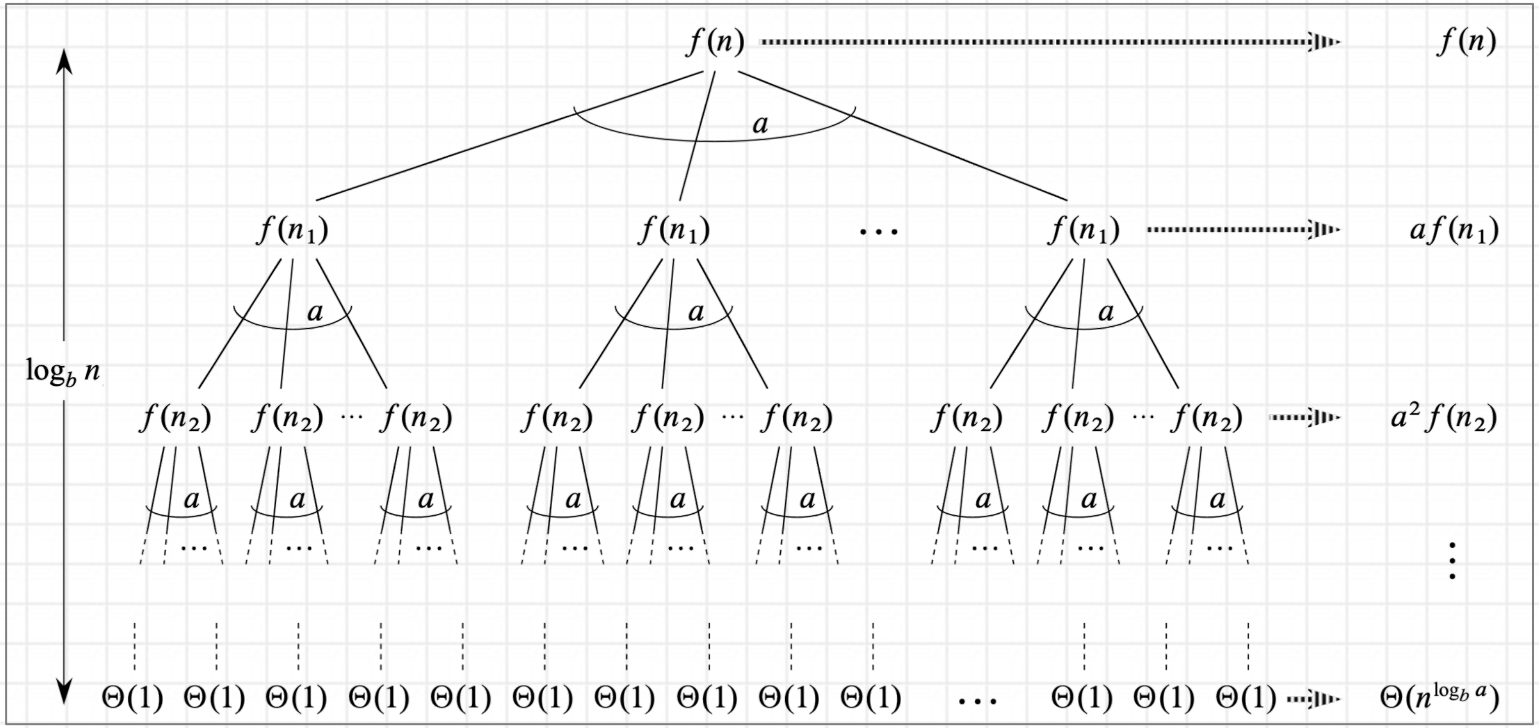
Master Theorem



- Goal. Recipe for solving common divide-and-conquer recurrences,

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

where $T(0) = 0$ and $T(1) = \Theta(1)$.



- Three cases can happen…

Master Theorem

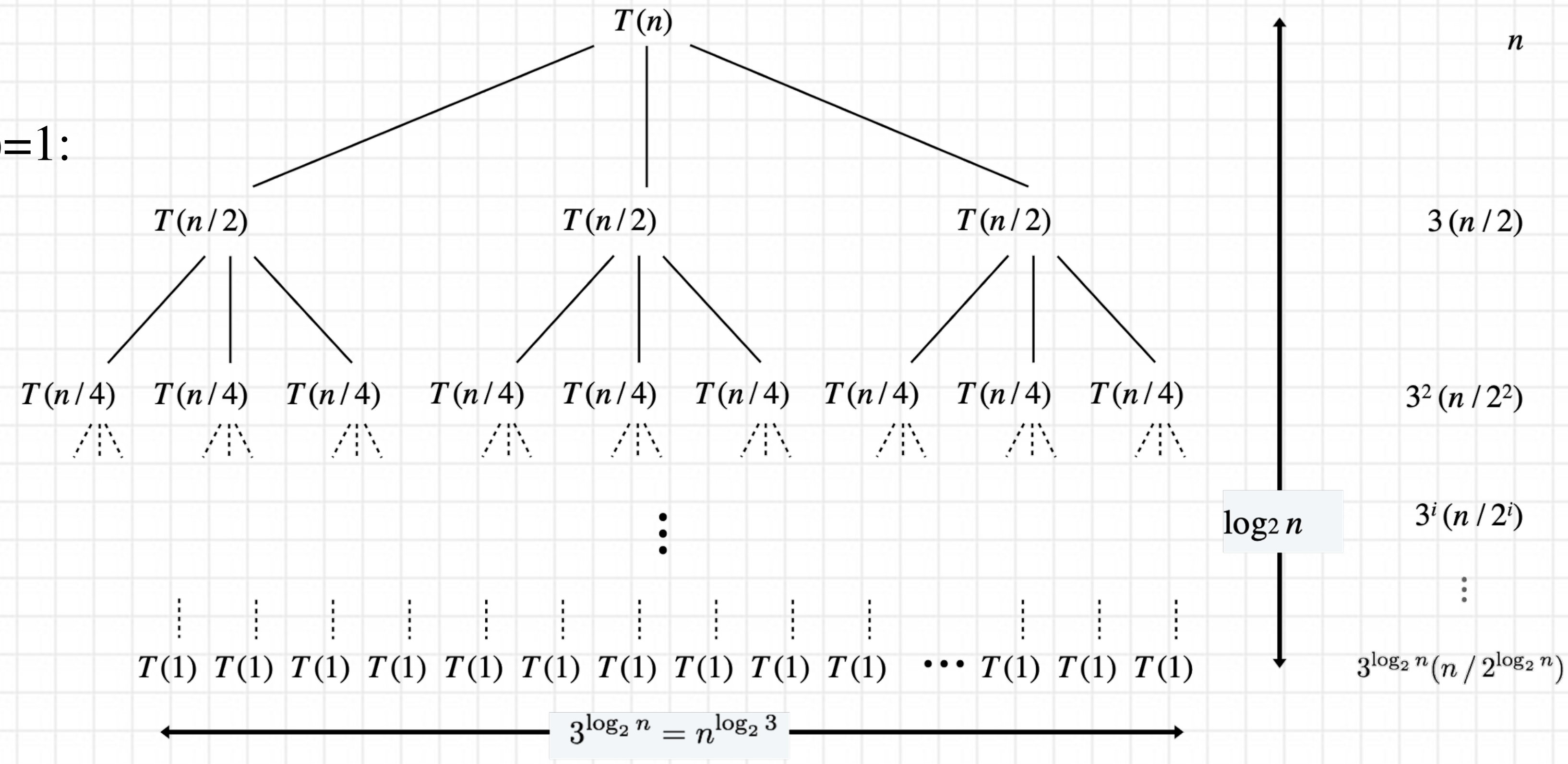
- Three cases can happen…
- But before talking about that, let's have a quick review about "Geometric Series"
  - Geometric series: sum of finite or infinite number of terms that have a constant ratio between each two consecutive terms.
  - Can be written as $a + ar + ar^2 + ar^3 + \cdots$, where a is the coefficient of each term and r is the common ratio between adjacent terms.
  - It can be shown that:
    - If $r \neq 1$, $1 + r + r^2 + r^3 + \cdots + r^{k-1} = \frac{1-r^k}{1-r}$
    - If $r = 1$, $1 + r + r^2 + r^3 + \cdots + r^{k-1} = k$
    - If $r < 1$, $1 + r + r^2 + r^3 + \cdots = \frac{1}{1-r}$

Master Theorem

- Case 1: Total computational cost is dominated by cost of leaves.
- Example:

Let T(n) = 3T(n/2)+n with T(1)=1:

Then, T(n)= $\Theta(n^{\log_2 3})$



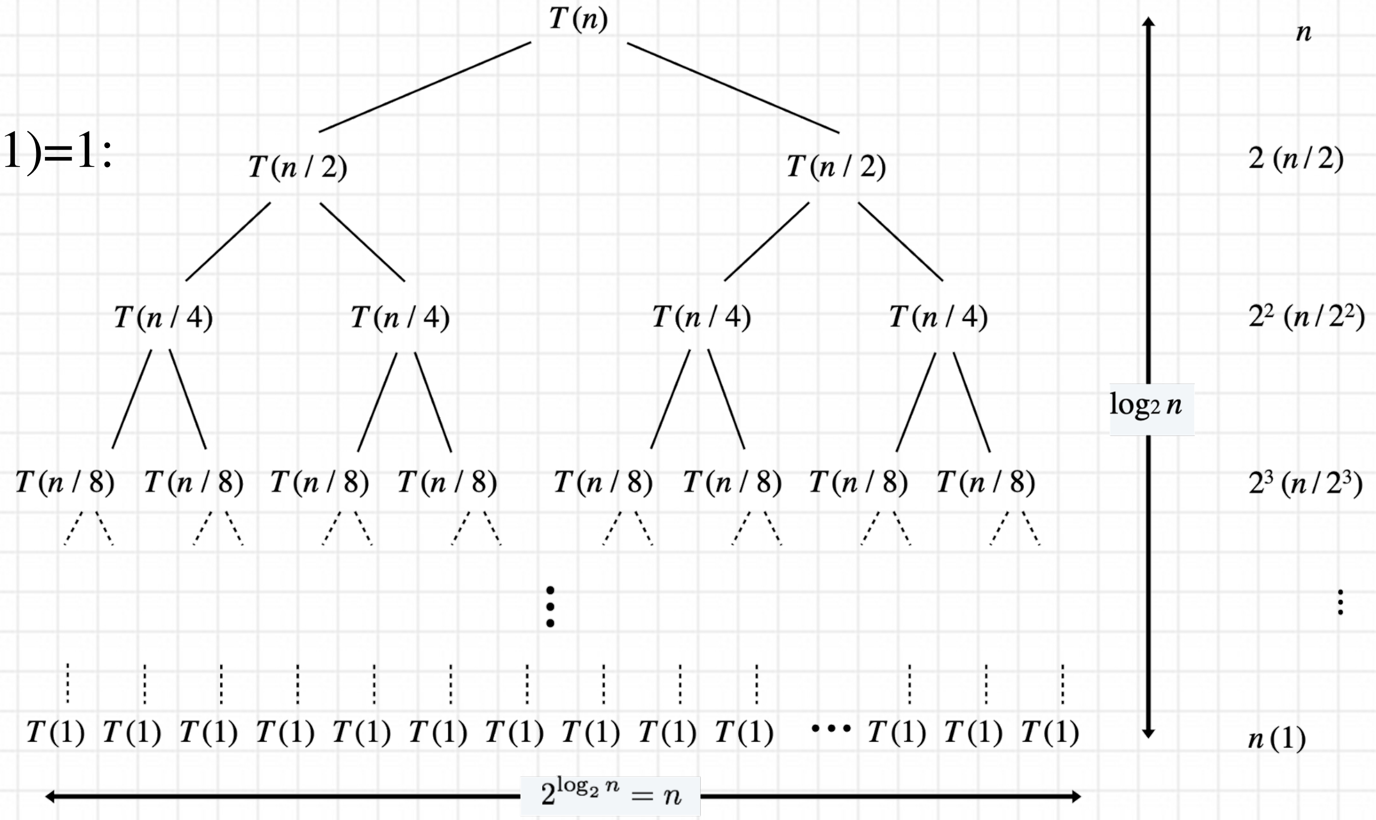
$r = 3/2 > 1 \qquad T(n) = (1 + r + r^2 + r^3 + \ldots + r^{\log_2 n})\, n = \dfrac{r^{1+\log_2 n} - 1}{r - 1} n = 3n^{\log_2 3} - 2n$

Master Theorem

- Case 2: Total computational cost is evenly distributed among levels
- Example:

Let T(n) = 2T(n/2)+n with T(1)=1:

Then, T(n)= $\Theta(n \log n)$



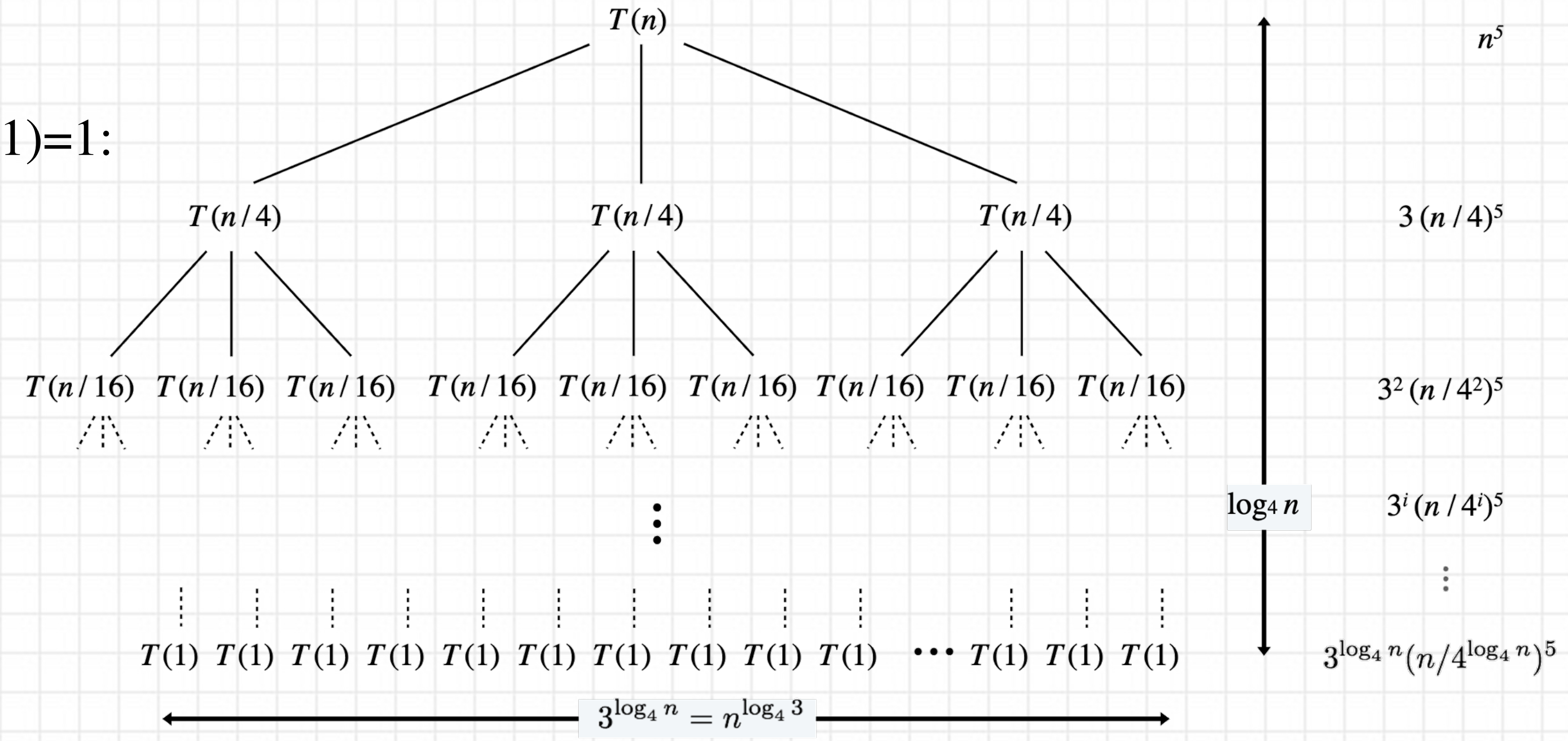
$r = 1$ $\quad T(n) = (1 + r + r^2 + r^3 + \ldots + r^{\log_2 n})\, n \; = \; n\,(\log_2 n + 1)$

Master Theorem

- Case 3: Total computational cost is dominated by cost of root
- Example:

Let $T(n) = 3T(n/4)+n^5$ with $T(1)=1$:

Then, $T(n)= \Theta(n^5)$



$r = 3/4^5 < 1 \qquad n^5 \leq T(n) \leq (1 + r + r^2 + r^3 + \ldots)\, n^5 \leq \frac{1}{1-r} n^5$

Master Theorem

- Goal. Recipe for solving common divide-and-conquer recurrences,

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

where $T(0) = 0$ and $T(1) = \Theta(1)$.

- $a \geq 1$ is the number of subproblems, also known as "branching factor"
- $b \geq 2$ is the factor by which the subproblem size decreases.
- $f(n) \geq 0$ is the work to divide and combine subproblems.
- If $f(n)$ is $\Theta(n^d)$, where $d \geq 0$:

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right), & \text{if } a > b^d \quad \text{(case 1)} \\ \Theta\left(n^d \log n\right), & \text{if } a = b^d \quad \text{(case 2)} \\ \Theta\left(n^d\right), & \text{if } a < b^d \quad \text{(case 3)} \end{cases}$$

Master Theorem

- Goal. Recipe for solving common divide-and-conquer recurrences,

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

$$\boxed{T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right), & \text{if } a > b^d \quad \text{(case 1)} \\ \Theta\left(n^d \log n\right), & \text{if } a = b^d \quad \text{(case 2)} \\ \Theta\left(n^d\right), & \text{if } a < b^d \quad \text{(case 3)} \end{cases}}$$

- Limitation. Master theorem cannot be used if
  - $T(n)$ is not monotone, e.g., $T(n) = \sin(n)$
  - $f(n)$ is not polynomial, e.g., $T(n) = 2\,T\left(\frac{n}{2}\right) + 2^n$
  - b cannot be expressed as a constant, e.g., $T(n) = a\,T(\sqrt{n}) + f(n)$

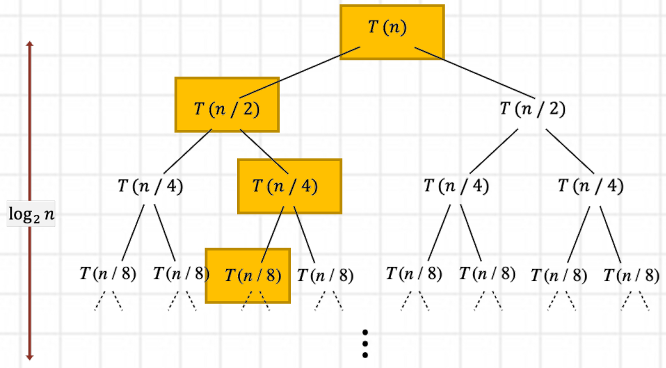
$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

Master Theorem

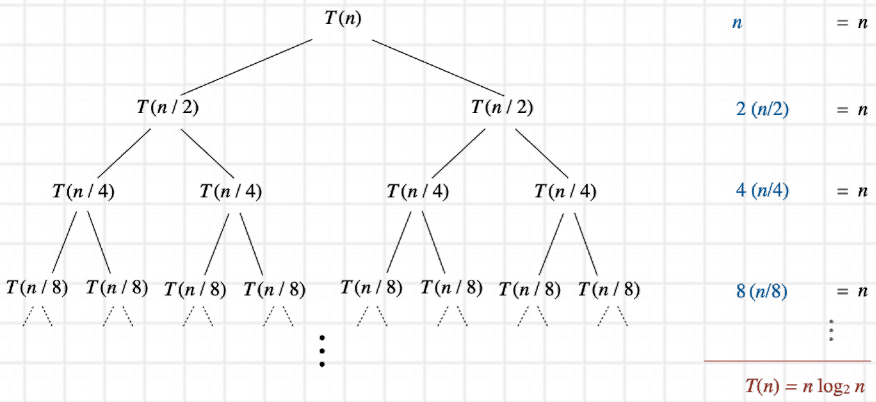
$$T(n) \in \begin{cases} \Theta(n^{\log_b a}), & \text{if } a > b^d \quad \text{(case 1)} \\ \Theta(n^d \log n), & \text{if } a = b^d \quad \text{(case 2)} \\ \Theta(n^d), & \text{if } a < b^d \quad \text{(case 3)} \end{cases}$$

- Now, we can apply master theorem to binary-search and merge-sort:

- Binary search:
  - Recurrence: $\mathrm{T(n)} = \mathrm{T}\left(\frac{n}{2}\right) + 1$
  - Therefore, $a = 1$, $\mathrm{b} = 2$, and $f(n) = 1 = \Theta(n^0)$, i.e., $d = 0$
  - $a = b^d \Longrightarrow \mathrm{T(n)} \in \Theta(n^0 \log n) = \Theta(\log n)$



- Merge sort:
  - Recurrence: $\mathrm{T(n)} = 2\mathrm{T}\left(\frac{n}{2}\right) + n$
  - Therefore, $a = 2$, $\mathrm{b} = 2$, and $f(n) = n = \Theta(n^1)$, i.e., $d = 1$
  - $a = b^d \Longrightarrow \mathrm{T(n)} \in \Theta(n^1 \log n) = \Theta(n \log n)$



References

- The lecture slides are heavily based on the suggested textbooks and the corresponding published lecture notes:

  - CLRS: Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009.
  - KT: Kleinberg, J., & Tardos, E. Algorithm design. Pearson/Addison-Wesley, 2006.
  - DPV: Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. Algorithms, McGraw-Hill Higher Education., 2008.
  - Slides by Kevin Wayne. Copyright © 2005 Pearson-Addison Wesley.
  - Slides by Erik D. Demaine and Charles E. Leiserson. Copyright © 2001-5