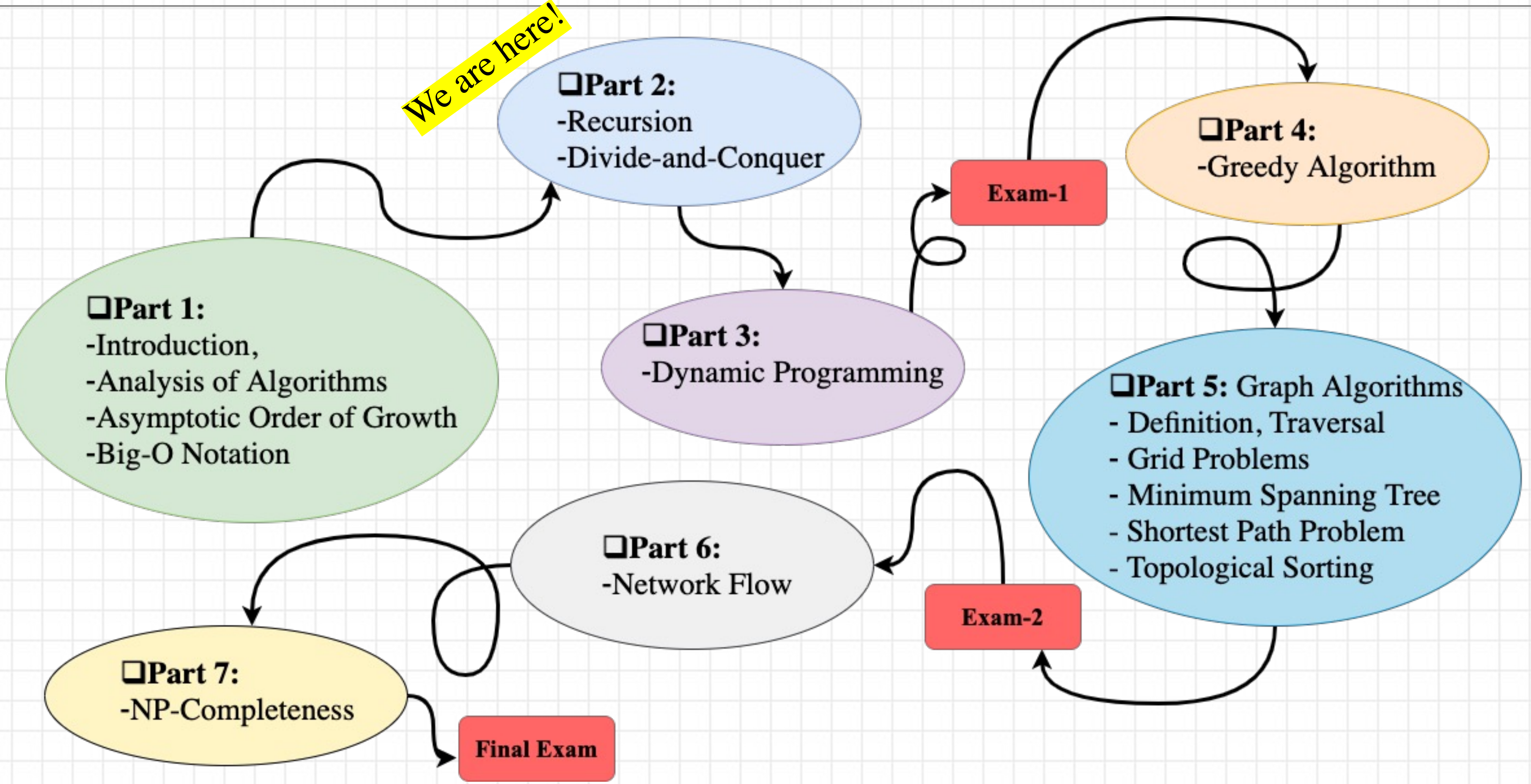
CS-3510: Design and Analysis of Algorithms

Divide-and-Conquer II

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

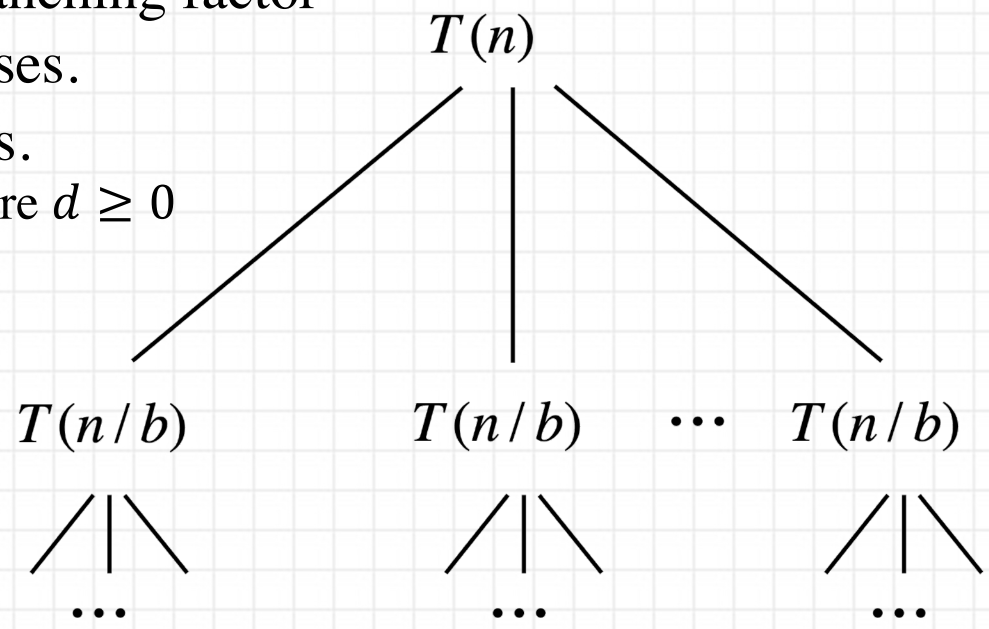
Master Theorem

- 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 $T(1) = \Theta(1)$.

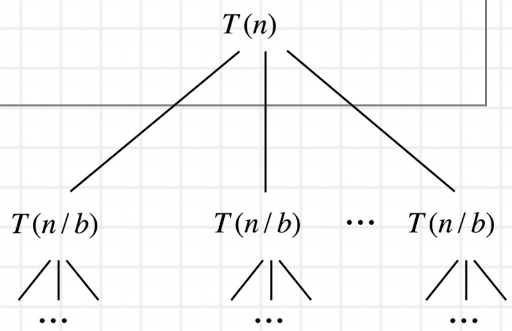
- $a \geq 1$ is the number of subproblems, also known as "branching factor"
- $b \geq 2$ is the factor by which the subproblem size decreases.
- $f(n) \geq 0$ is the work to divide and combine subproblems.
  - $f(n)$ usually takes polynomial time, i.e., $f(n)$ is $\Theta(n^d)$, where $d \geq 0$



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

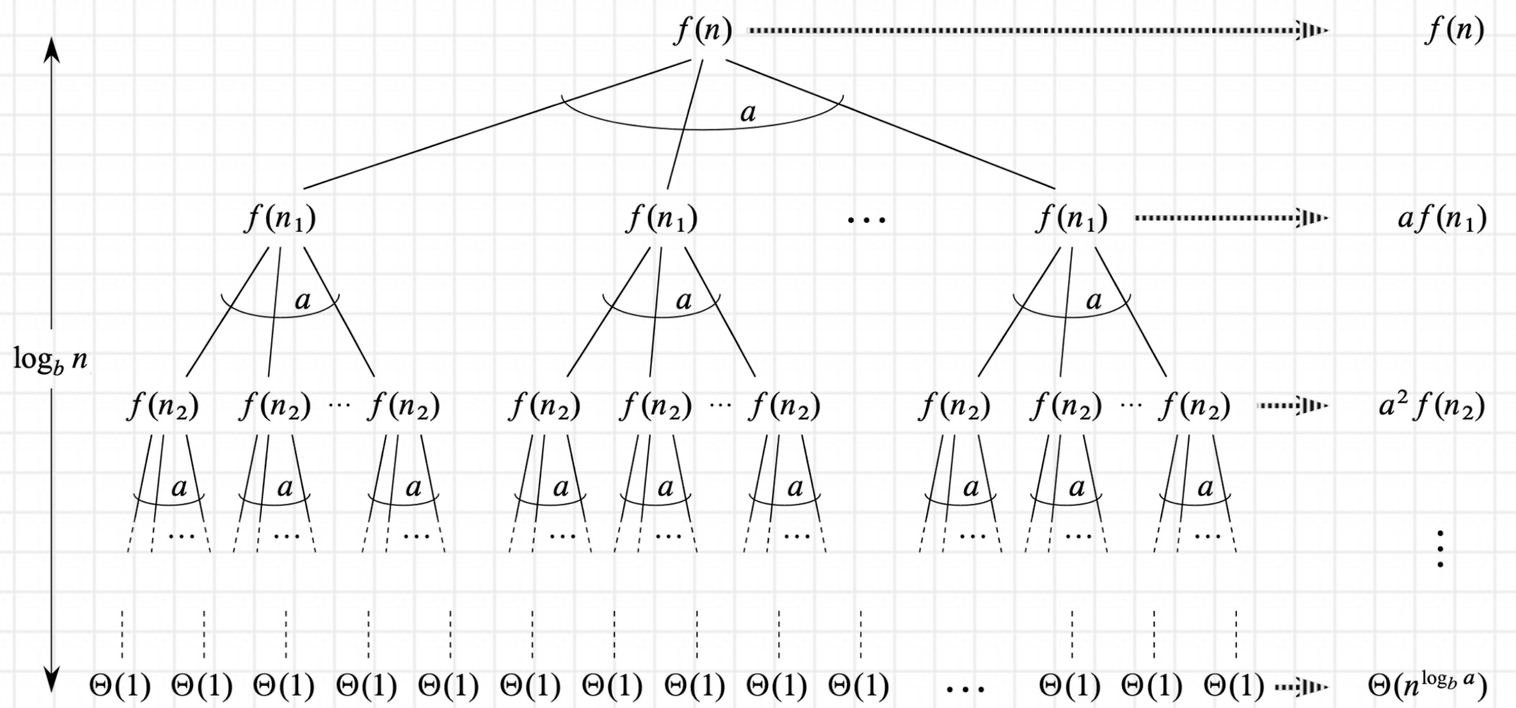
Master Theorem



- 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

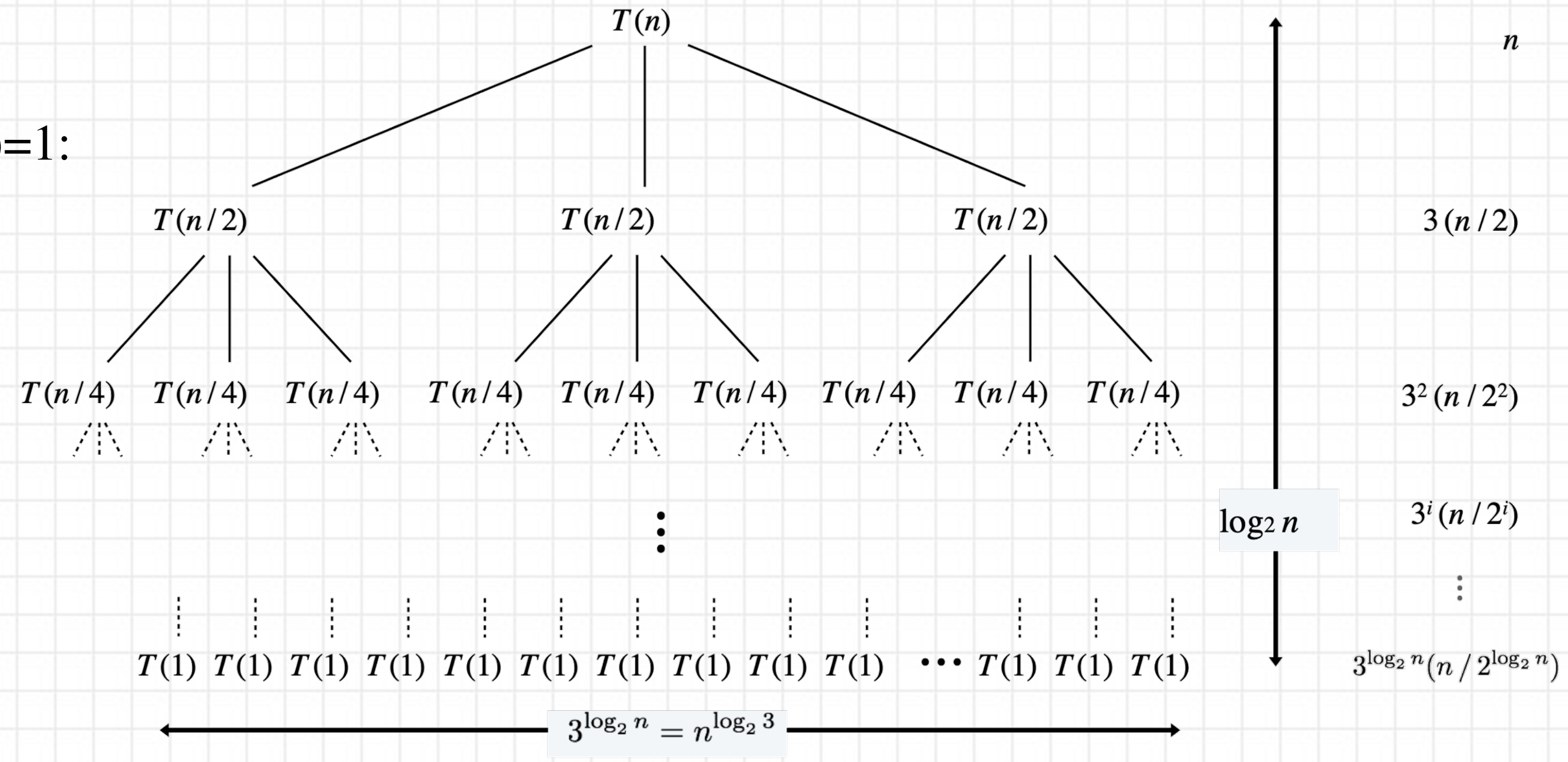
- Three cases can happen…
- But before talking about that, let's have a quick review about "Geometric Series"
  - Geometric series: sum of finite or infinite number of terms that have a constant ratio between each two consecutive terms.
  - Can be written as $a + ar + ar^2 + ar^3 + \cdots$, where a is 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)= $\Theta(n^{\log_2 3})$



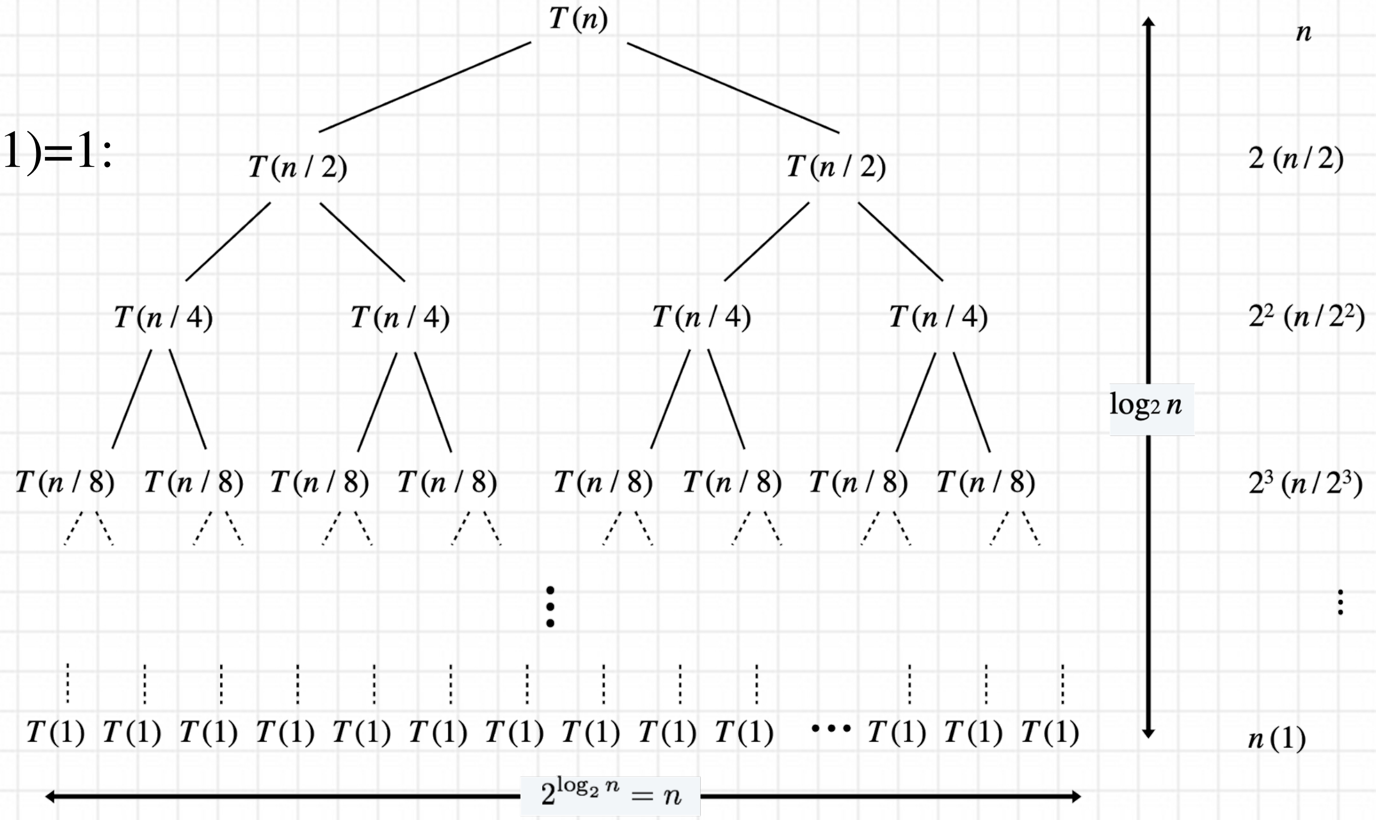
$r = 3/2 > 1$ $\quad T(n) = (1 + r + r^2 + r^3 + \ldots + r^{\log_2 n})\, n = \dfrac{r^{1+\log_2 n} - 1}{r - 1}\, n = 3n^{\log_2 3} - 2n$

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)= $\Theta(n \log n)$



$r = 1$

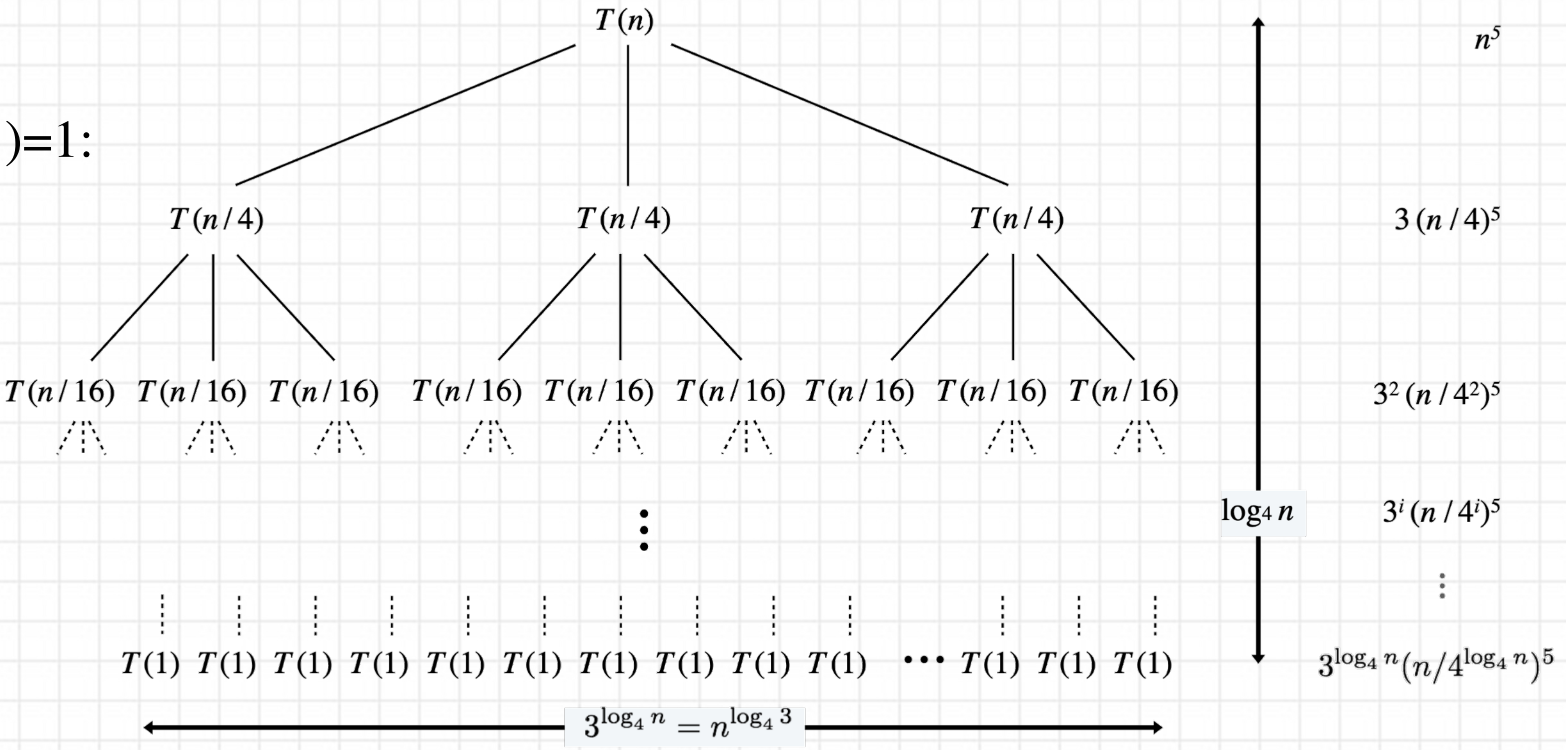
$$T(n) = (1 + r + r^2 + r^3 + \ldots + r^{\log_2 n})\, n \;=\; n\,(\log_2 n + 1)$$

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



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

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., $T(n) = a\,T(\sqrt{n}) + f(n)$

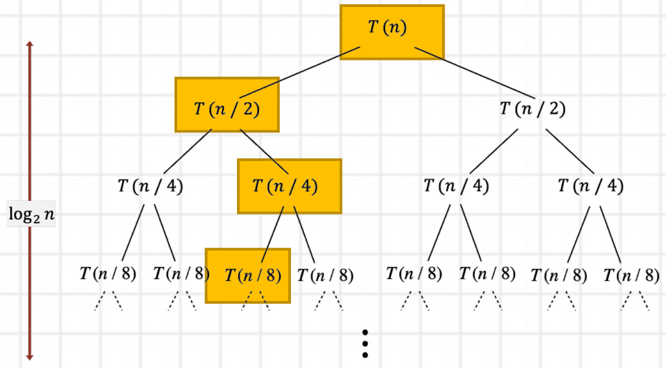
$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

Master Theorem

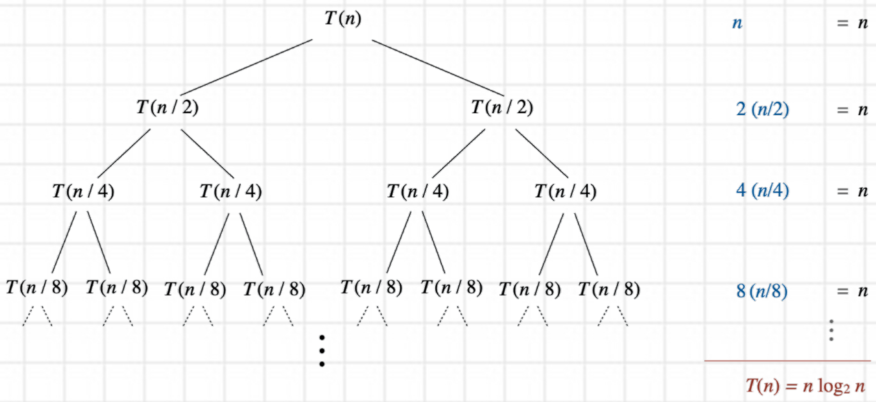
$$T(n) \in \begin{cases} \Theta(n^{\log_b a}), & \text{if } a > b^d \quad \text{(case 1)} \\ \Theta(n^d \log n), & \text{if } a = b^d \quad \text{(case 2)} \\ \Theta(n^d), & \text{if } a < b^d \quad \text{(case 3)} \end{cases}$$

- 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., $d = 1$
  - $a = b^d \Longrightarrow \mathrm{T(n)} \in \Theta(n^1 \log n) = \Theta(n \log n)$



Master Theorem

- More examples:
- Let $\mathrm{T(n)} = \mathrm{T}\left(\frac{n}{2}\right) + \frac{1}{2}n^2 + n$
- $a = 1, b = 2, d = 2$
- $a < b^d$ (case 3)

- $T(n) \in \Theta(n^2)$

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

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

Master Theorem

- More examples:
- Let $\mathrm{T(n)} = 2\mathrm{T}\left(\frac{n}{4}\right) + \sqrt{n} + 8$
- $a = 2, b = 4, d = \frac{1}{2}$
- $a = b^d$ (case 2)

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

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

- $T(n) \in \Theta\left(n^d \log n\right) = \Theta(\sqrt{n} \log n)$

Master Theorem

- More examples:
- Let $\mathrm{T(n)} = 3\mathrm{T}\left(\frac{n}{2}\right) + \frac{3}{4}n + 1$
- $a = 3, b = 2, d = 1$
- $a > b^d$ (case 1)
- $T(n) \in \Theta\left(n^{\log_2 3}\right)$

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

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

D&C Example: Quick-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: $\Theta(n^2)$

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

Quick-sort (CLRS 7.1)

- Similar to merge-sort applies divide-and-conquer paradigm.
- 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.

- Quick-sort:
  - Divide: Partition (rearrange) the array into three parts: $\overbrace{A[1{:}p-1]}^{A_{\text{left}}}, \overbrace{A[p]}^{a_p}, \overbrace{A[p+1{:}n]}^{A_{\text{right}}}$, such that all elements of $A_{\text{left}} < A[p]$ and all elements of $A_{\text{right}} \geq A[p]$. Also, $a_p = A[p]$ is known as the pivot element. Return index p.
  - Conquer: Sort the two sub-arrays A_{left} and A_{right} by recursive calls to quick-sort on each half.
  - Combine: Because the subarrays are already sorted, no additional work is required for combining the results. The entire array is now sorted

Quick-sort (CLRS 7.1)

- Similar to merge-sort applies divide-and-conquer paradigm.
- 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.

Key part of merge-sort

- Quick-sort:

Key part of quick-sort

  - Divide: Partition (rearrange) the array into three parts: $\overbrace{A[1{:}p-1]}^{A_{left}}, \overbrace{A[p]}^{a_p}, \overbrace{A[p+1{:}n]}^{A_{right}}$, such that all elements of $A_{left} < A[p]$ and all elements of $A_{right} \geq A[p]$. Also, $a_p = A[p]$ is known as the pivot element. Return index p.
  - Conquer: Sort the two sub-arrays A_{left} and A_{right} by recursive calls to quick-sort on each half.
  - Combine: Because the subarrays are already sorted, no additional work is required for combining the results. The entire array is now sorted

Quick-sort (CLRS 7.1)

- Quick-sort:
  - Divide: Partition (rearrange) the array into three parts: $\overbrace{A[1{:}p-1]}^{A_{left}}, \overbrace{A[p]}^{a_p}, \overbrace{A[p+1{:}n]}^{A_{right}}$, such that all elements of $A_{left} < A[p]$ and all elements of $A_{right} \geq A[p]$. Also, $a_p = A[p]$ is known as the pivot element. Return index p.
  - Conquer: Sort the two sub-arrays A_{left} and A_{right} by recursive calls to quick-sort on each half.
  - Combine: Because the subarrays are already sorted, no additional work is required for combining the results. The entire array is now sorted

```
Quicksort(A, lo, hi):
    if lo < hi:
        p = partition(A, lo, hi)
        Quicksort(A, lo, p-1)
        Quicksort(A, p+1, r)
```

Quick-sort (CLRS 7.1)

- Quick-sort:
  - Divide: Partition (rearrange) the array into three parts: $\overbrace{A[1:p-1]}^{A_{left}}, \overbrace{A[p]}^{a_p}, \overbrace{A[p+1:n]}^{A_{right}}$, such that all elements of $A_{left} < A[p]$ and all elements of $A_{right} \geq A[p]$. Also, $a_p = A[p]$ is known as the pivot element. Return index p.
  - Conquer: Sort the two sub-arrays A_{left} and A_{right} by recursive calls to quick-sort on each half.
  - Combine: Because the subarrays are already sorted, no additional work is required for combining the results. The entire array is now sorted
- Key component: Partition
  - Returns the final index of the pivot
  - Maintains two subarrays which grow

```
Quicksort(A, lo, hi):
    if lo < hi:
        p = partition(A, lo, hi)
        Quicksort(A, lo, p-1)
        Quicksort(A, p+1, r)
```

Quick-sort (CLRS 7.1)

```
Quicksort(A, lo, hi):
    if lo < hi:
        p = partition(A, lo, hi)
        Quicksort(A, lo, p-1)
        Quicksort(A, p+1, r)
```

```
Partition(A, lo, hi):
    choose a pivot element p ∈ [lo, hi]
    exchange A[p] with A[hi]
    pivot_index ← lo
    for each i = lo : hi-1
        if A[i] < A[hi]:
            exchange A[i] with A[pivot_index]
            pivot_index ++
    exchange A[hi] with A[pivot_index]
    return pivot_index
```

- Key component: Partition
  - Returns the final index of the pivot
  - Maintains two subarrays which grow
  - p returns is the position of the pivot element in the final sorted array

Quick-sort (CLRS 7.1)

- Key component: Partition
  - Returns the final index of the pivot
  - Maintains two subarrays which grow
  - p returns is the position of the pivot element in the final sorted array
- Ex. lo hi
- Let A = [..., 30, 50, 15, 5, 25, 8, 6, 20, ...]
- P = Partition (A, lo, hi)

```
Partition(A, lo, hi):
    choose a pivot element p ∈ [lo, hi]
    exchange A[p] with A[hi]
    // we can always choose p = hi.
    // In that case no exchange is required
    pivot_index ⟵ lo
    for each i = lo : hi-1
        if A[i] < A[hi]:
            exchange A[i] with A[pivot_index]
            pivot_index ++
    exchange A[hi] with A[pivot_index]
    return pivot_index
```

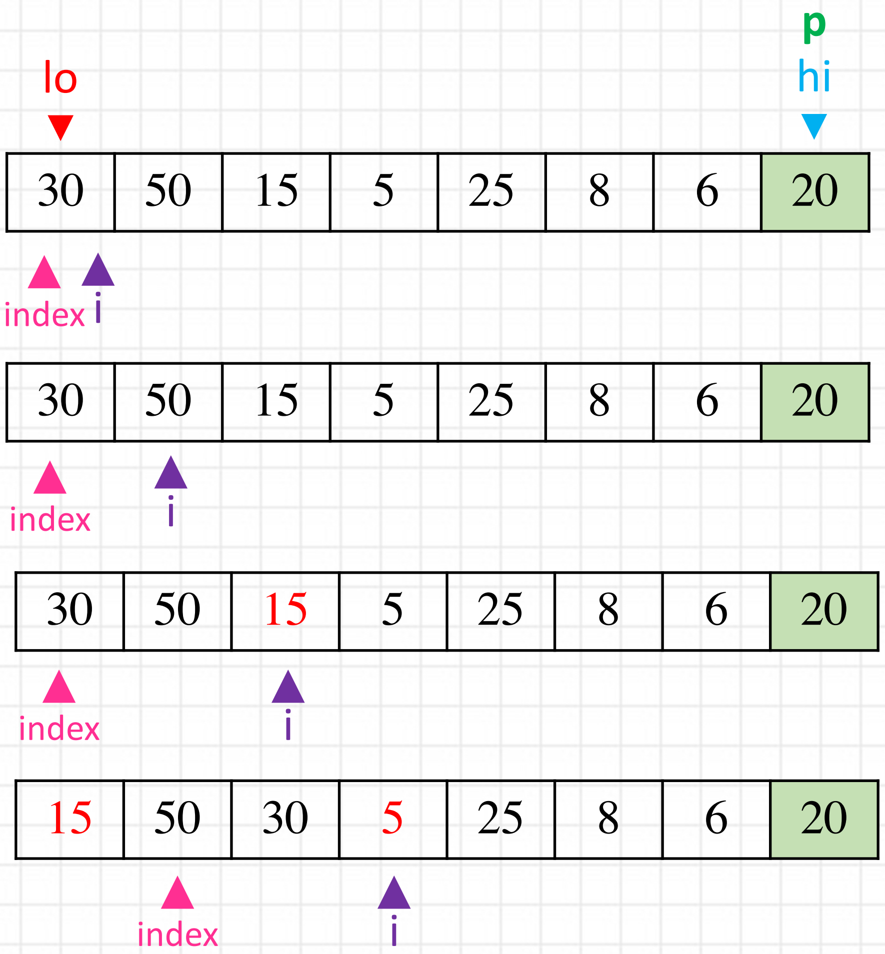
Quick-sort (CLRS 7.1)

- Key component: Partition
  - Returns the final index of the pivot
  - Maintains two subarrays which grow
  - p returns is the position of the pivot element in the final sorted array
- Ex. lo hi
- Let A = […, 30, 50, 15, 5, 25, 8, 6, 20, …]
- P = Partition (A, lo, hi)

```
Partition(A, lo, hi):
    choose a pivot element p ∈ [lo, hi]
    exchange A[p] with A[hi]
    // we can always choose p = hi.
    // In that case no exchange is required
    pivot_index ⟵ lo
    for each i = lo : hi-1
        if A[i] < A[hi]:
            exchange A[i] with A[pivot_index]
            pivot_index ++
    exchange A[hi] with A[pivot_index]
    return pivot_index
```

lo							p hi
30	50	15	5	25	8	6	20
index	i						

Quick-sort (CLRS 7.1)




```
Partition(A, lo, hi):
    choose a pivot element p ∈ [lo, hi]
    exchange A[p] with A[hi]
    // we can always choose p = hi.
    // In that case no exchange is required
    pivot_index ⟵ lo
    for each i = lo : hi-1
        if A[i] < A[hi]:
            exchange A[i] with A[pivot_index]
            pivot_index ++
    exchange A[hi] with A[pivot_index]
    return pivot_index
```

Quick-sort (CLRS 7.1)

15	5	30	50	25	8	6	20
		index		i			

15	5	30	50	25	8	6	20
		index			i		

15	5	8	50	25	30	6	20
			index			i	

15	5	8	6	25	30	50	20
				index		i	

```
Partition(A, lo, hi):
    choose a pivot element p ∈ [lo, hi]
    exchange A[p] with A[hi]
    // we can always choose p = hi.
    // In that case no exchange is required
    pivot_index ⟵ lo
    for each i = lo : hi-1
        if A[i] < A[hi]:
            exchange A[i] with A[pivot_index]
            pivot_index ++
    exchange A[hi] with A[pivot_index]
    return pivot_index
```

Quick-sort (CLRS 7.1)

15	5	30	50	25	8	6	20
		index		i			

15	5	30	50	25	8	6	20
		index			i		

15	5	8	50	25	30	6	20
			index			i	

15	5	8	6	25	30	50	20
				index		i	

15	5	8	6	20	30	50	25
				index		i	

Return index = 4

(Note A[4] = 20 is in its right place in the final sorted array)

```
Partition(A, lo, hi):
    choose a pivot element p ∈ [lo, hi]
    exchange A[p] with A[hi]
    // we can always choose p = hi.
    // In that case no exchange is required
    pivot_index ⟵ lo
    for each i = lo : hi-1
        if A[i] < A[hi]:
            exchange A[i] with A[pivot_index]
            pivot_index ++
    exchange A[hi] with A[pivot_index]
    return pivot_index
```

Quick-sort (CLRS 7.1)

- Let A = [..., 30, 50, 15, 5, 25, 8, 6, 20, ...]
- P = Partition (A, lo, hi)

lo							p hi
30	50	15	5	25	8	6	20
index i							

30	50	15	5	25	8	6	20
index	i						

30	50	15	5	25	8	6	20
index		i					

15	50	30	5	25	8	6	20
	index		i				

15	5	30	50	25	8	6	20
		index		i			

15	5	30	50	25	8	6	20
		index			i		

15	5	8	50	25	30	6	20
			index			i	

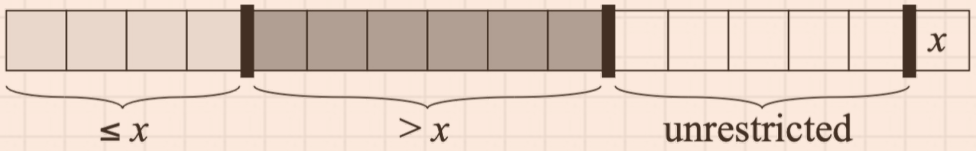
15	5	8	6	25	30	50	20
				index		i	

15	5	8	6	20	30	50	25
				index		i	

Return index = 4

Quick-sort (CLRS 7.1)

- Important Notes:
  - Let x= pivot =A[p]. Then, at each step of the for loop, we have four regions:
    1) A[1 : index-1] all elements < x
    2) A[index : i] all elements ≥ x
    3) A[i+1 : hi-1] not specified yet!
    4) A[hi] = x (the pivot element)



  - After calling p = Partition (A, lo, hi), all elements before pivot = A[p] are less than (<) pivot and all elements after pivot are not less than (≥) pivot

```
Partition(A, lo, hi):
    choose a pivot element p ∈ [lo, hi]
    exchange A[p] with A[hi]
    // we can always choose p = hi.
    // In that case no exchange is required
    pivot_index ⟵ lo
    for each i = lo : hi-1
        if A[i] < A[hi]:
            exchange A[i] with A[pivot_index]
            pivot_index ++
    exchange A[hi] with A[pivot_index]
    return pivot_index
```

Quick-sort (CLRS 7.1)

```
Quicksort(A, lo, hi):
    if lo < hi:
        p = partition(A, lo, hi)
        Quicksort(A, lo, p-1)
        Quicksort(A, p+1, r)
```

- Demo

```
Partition(A, lo, hi):
    choose a pivot element p ∈ [lo, hi]
    exchange A[p] with A[hi]
    pivot_index ← lo
    for each i = lo : hi-1
        if A[i] < A[hi]:
            exchange A[i] with A[pivot_index]
            pivot_index ++
    exchange A[hi] with A[pivot_index]
    return pivot_index
```

Quick-sort (CLRS 7.1)

- Running time?
  - It depends!
    - Whether the partitioning is balanced or unbalanced.
    - Therefore, it depends on which elements are used for partitioning.
  - If the partitioning is <u>balanced</u>
    - Asymptotically as fast as merge-sort $\Theta(n \log n)$
  - If the partitioning is <u>unbalanced</u>
    - Asymptotically as slow as insertion-sort $\Theta(n^2)$

Quick-sort (CLRS 7.1)

- Running time? (not a formal proof)
  - Worst-case: when the partitioning is <u>unbalanced</u>
    - The partition routine produces one subproblem with $n-1$ elements and one with 0 element. In the worst case, this will happen in each recursive call.
    - This can happen when the input array is sorted. (maximally unbalanced)
    - $T(n) = T(n-1) + T(0) + \underbrace{\Theta(n)}_{\text{Partitioning}} = T(n-1) + \Theta(n) \in \Theta(n^2)$
    - Asymptotically as slow as insertion-sort $\Theta(n^2)$

Quick-sort (CLRS 7.1)

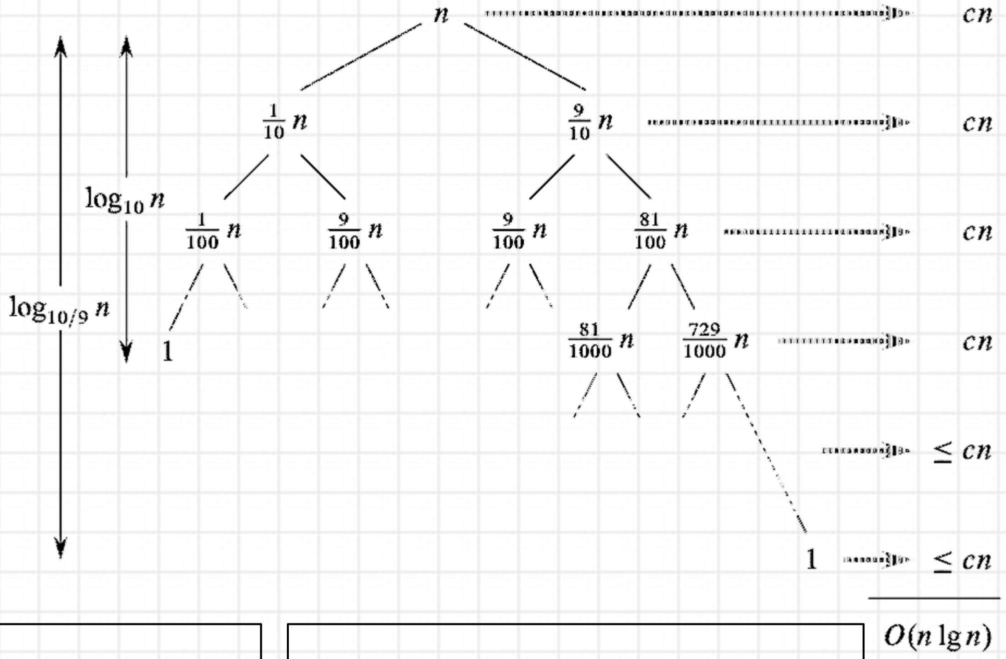
- Running time? (not a formal proof)
  - Best-case: most even possible split
    - The partition routine produces two subproblems, each of size no more than $n/2$

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \in \Theta(n \log n)$$

  - Average-case
    - Much closer to the best case than to the worst case
    - Ex. Assume the Partition subroutine always produces 9-to-1 proportional split

$$T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \Theta(n)$$

$$\mathrm{T(n)} \in \Theta(n \log n)$$



Quick-sort (CLRS 7.1)

- Running time? (not a formal proof)
  - In practice:
    - For not-worst-case inputs, quick-sort usually outperforms merge-sort.
    - Commonly used in sorting libraries.
  - Strategies to avoid $\Theta(n^2)$
    - Choosing the pivot element randomly
    - Choosing the pivot as the median of three random elements
    - Still, the worst case is possible, but highly unlikely

```
Partition(A, lo, hi):
    choose a pivot element p ∈ [lo, hi]
    exchange A[p] with A[hi]
    pivot_index ⟵ lo
    for each i = lo : hi-1
        if A[i] < A[hi]:
            exchange A[i] with A[pivot_index]
            pivot_index ++
    exchange A[hi] with A[pivot_index]
    return pivot_index
```

Merge-sort vs. Quick-sort

- Merge-sort: (bottom-up: main action during the combining the subproblem solutions)
  - 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

	Best	Average	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))

- Quick-sort: (top-down: main action during the breaking the problem into subproblems)
  - Divide: Partition (rearrange) the array into three parts: $A_{\text{left}}, A[p], A_{\text{right}}$, such that all elements of $A_{\text{left}} < A[p]$ and all elements of $A_{\text{right}} \geq A[p]$. Also, $a_p = A[p]$ is known as the pivot element. Return index p. As we divide into subproblems, we find the right position of the pivot element.
  - Conquer: Sort the two sub-arrays A_{left} and A_{right} by recursive calls to quick-sort on each half.
  - Combine: Because the subarrays are already sorted, no additional work is required for combining the results. The entire array is now sorted

Merge-sort vs. Quick-sort

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

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/

https://www.toptal.com/developers/sorting-algorithms
http://www.cs3510.com/resources/

D&C Example: Matrix Multiplication

Matrix multiplication. Given two n-by-n matrices A and B, compute $\underbrace{C = AB}$.

Grade-school. $\Theta(n^3)$ arithmetic operations.

$$c_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{kj}$$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$$\begin{bmatrix} .59 & .32 & .41 \\ .31 & .36 & .25 \\ .45 & .31 & .42 \end{bmatrix} = \begin{bmatrix} .70 & .20 & .10 \\ .30 & .60 & .10 \\ .50 & .10 & .40 \end{bmatrix} \times \begin{bmatrix} .80 & .30 & .50 \\ .10 & .40 & .10 \\ .10 & .30 & .40 \end{bmatrix}$$

D&C Example: Matrix Multiplication

$$\underbrace{\begin{bmatrix} 152 & 158 \\ 504 & 526 \end{bmatrix}}_{C_{11}}$$

$$\begin{bmatrix} 152 & 158 & 164 & 170 \\ 504 & 526 & 548 & 570 \\ 856 & 894 & 932 & 970 \\ 1208 & 1262 & 1316 & 1370 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \times \begin{bmatrix} 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 \\ 28 & 29 & 30 & 31 \end{bmatrix}$$

C_{11}, A_{11}, A_{12}, B_{11}, B_{21}

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} = \begin{bmatrix} 0 & 1 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 16 & 17 \\ 20 & 21 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \times \begin{bmatrix} 24 & 25 \\ 28 & 29 \end{bmatrix} = \begin{bmatrix} 152 & 158 \\ 504 & 526 \end{bmatrix}$$

D&C Example: Matrix Multiplication

To multiply two n-by-n matrices A and B:

- Divide: partition A and B into ½n-by-½n blocks.
- Conquer: multiply 8 pairs of ½n-by-½n matrices, recursively.
- Combine: add appropriate products using 4 matrix additions.

n-by-n matrices

$$C = A \times B$$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

½n-by-½n matrices

8 matrix multiplications (of ½n-by-½n matrices)

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

4 matrix additions (of ½n-by-½n matrices)

D&C Example: Matrix Multiplication

To multiply two n-by-n matrices A and B:

- Divide: partition A and B into ½n-by-½n blocks.
- Conquer: multiply 8 pairs of ½n-by-½n matrices, recursively.
- Combine: add appropriate products using 4 matrix additions.

n-by-n matrices

$$C = A \times B$$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

½n-by-½n matrices

8 matrix multiplications (of ½n-by-½n matrices)

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

4 matrix additions (of ½n-by-½n matrices)

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

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

Runtime?

Using master theorem:

$$T(n) = \underbrace{8T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}}$$

$$\Rightarrow T(n) = \Theta(n^3)$$

D&C Example: Matrix Multiplication

- Fast matrix multiplication
  - Strassen's trick

Key idea. Can multiply two 2-by-2 matrices via 7 scalar multiplications (plus 11 additions and 7 subtractions).

scalars

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_1 + P_5 - P_3 - P_7 \end{aligned}$$

$$\begin{aligned} P_1 &\leftarrow A_{11} \times (B_{12} - B_{22}) \\ P_2 &\leftarrow (A_{11} + A_{12}) \times B_{22} \\ P_3 &\leftarrow (A_{21} + A_{22}) \times B_{11} \\ P_4 &\leftarrow A_{22} \times (B_{21} - B_{11}) \\ P_5 &\leftarrow (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ P_6 &\leftarrow (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ P_7 &\leftarrow (A_{11} - A_{21}) \times (B_{11} + B_{12}) \end{aligned}$$

7 scalar multiplications

Pf.
$$\begin{aligned} C_{12} &= P_1 + P_2 \\ &= A_{11} \times (B_{12} - B_{22}) + (A_{11} + A_{12}) \times B_{22} \\ &= A_{11} \times B_{12} + A_{12} \times B_{22}. \checkmark \end{aligned}$$

D&C Example: Matrix Multiplication

- Fast matrix multiplication
  - Strassen's trick
  - To multiply n-by-n matrices:
  - Divide:
    partition A and B into $\frac{1}{2}n$-by-$\frac{1}{2}n$ blocks.
  - Compute:
    14 $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices via 10 matrix additions.
  - Conquer:
    multiply 7 pairs of $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices, recursively.
  - Combine:
    7 products into 4 terms using 8 matrix additions.

Key idea. Can multiply two 2-by-2 matrices via 7 scalar multiplications (plus 11 additions and 7 subtractions).

scalars

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= P_1 + P_5 - P_3 - P_7
\end{aligned}$$

$$\begin{aligned}
P_1 &\leftarrow A_{11} \times (B_{12} - B_{22}) \\
P_2 &\leftarrow (A_{11} + A_{12}) \times B_{22} \\
P_3 &\leftarrow (A_{21} + A_{22}) \times B_{11} \\
P_4 &\leftarrow A_{22} \times (B_{21} - B_{11}) \\
P_5 &\leftarrow (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\
P_6 &\leftarrow (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\
P_7 &\leftarrow (A_{11} - A_{21}) \times (B_{11} + B_{12})
\end{aligned}$$

7 scalar multiplications

Pf.
$$\begin{aligned}
C_{12} &= P_1 + P_2 \\
&= A_{11} \times (B_{12} - B_{22}) + (A_{11} + A_{12}) \times B_{22} \\
&= A_{11} \times B_{12} + A_{12} \times B_{22}. \checkmark
\end{aligned}$$

D&C Example: Matrix Multiplication

- Fast matrix multiplication
  - Strassen's trick
  - To multiply n-by-n matrices:
  - <u>Divide:</u>
    partition A and B into
    $\frac{1}{2}n$-by-$\frac{1}{2}n$ blocks.
  - <u>Compute:</u>
    14 $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices via
    10 matrix additions.
  - <u>Conquer:</u>
    multiply 7 pairs of $\frac{1}{2}n$-by-$\frac{1}{2}n$
    matrices, recursively.
  - <u>Combine:</u>
    7 products into 4 terms using
    8 matrix additions.

Runtime?

Using master theorem:

Assume n is power of 2.

$$T(n) = \underbrace{7T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, subtract}}$$

$$T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

D&C Example: Matrix Multiplication

- History of arithmetic complexity of matrix multiplication

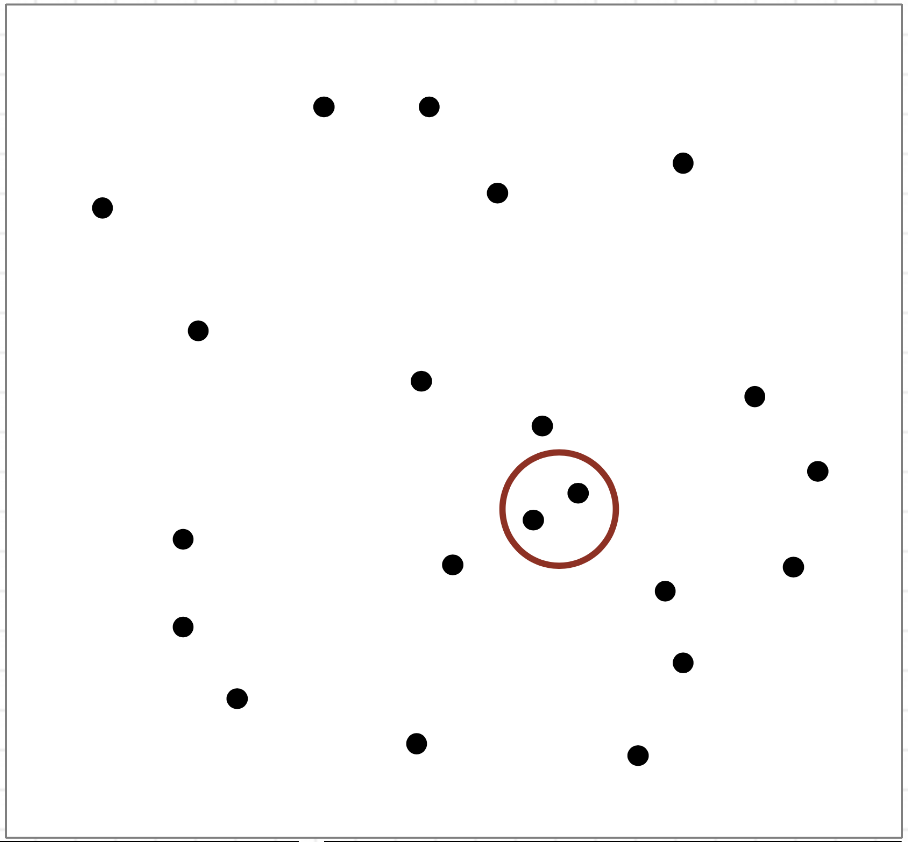
year	algorithm	arithmetic operations
1858	"grade school"	$O(n^3)$
1969	Strassen	$O(n^{2.808})$
1978	Pan	$O(n^{2.796})$
1979	Bini	$O(n^{2.780})$
1981	Schönhage	$O(n^{2.522})$
1982	Romani	$O(n^{2.517})$
1982	Coppersmith–Winograd	$O(n^{2.496})$
1986	Strassen	$O(n^{2.479})$
1989	Coppersmith–Winograd	$O(n^{2.3755})$
2010	Strother	$O(n^{2.3737})$
2011	Williams	$O(n^{2.372873})$
2014	Le Gall	$O(n^{2.372864})$
	???	$O(n^{2+\varepsilon})$

galactic algorithms

- Conjecture: $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$

D&C Example: Closest Pair of Points

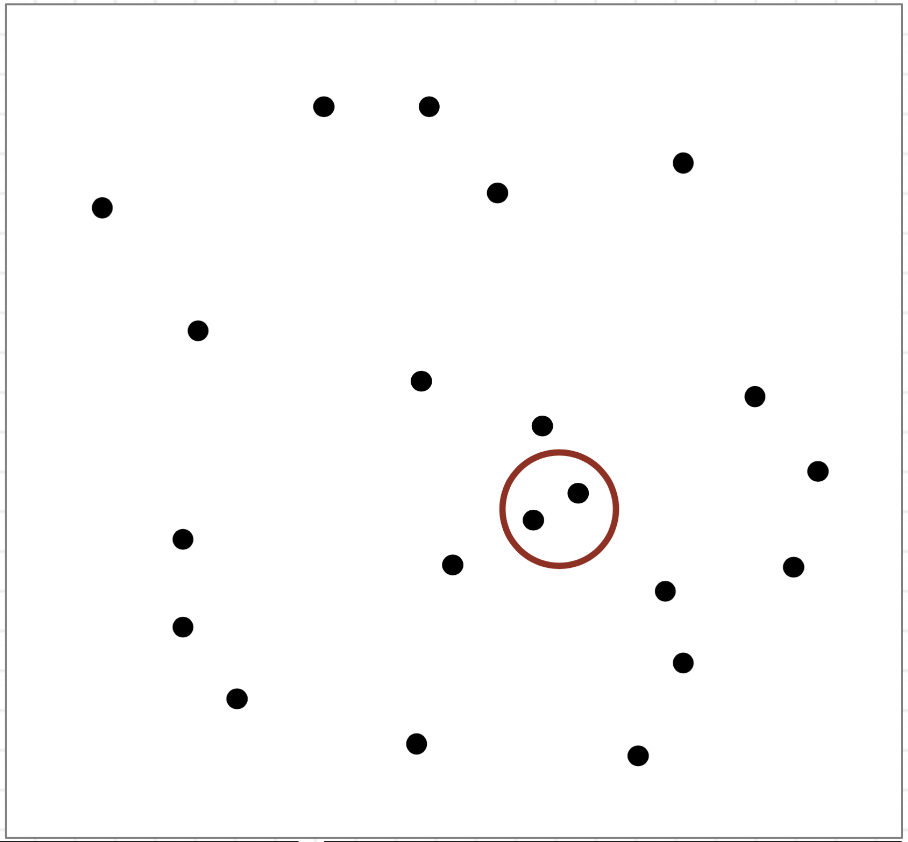
- Problem: Given n points in the plane, find a pair of points with the smallest Euclidean distance between them.



- Applications
  - Fundamental geometric primitive.
  - Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.
  - Special case of nearest neighbor

D&C Example: Closest Pair of Points

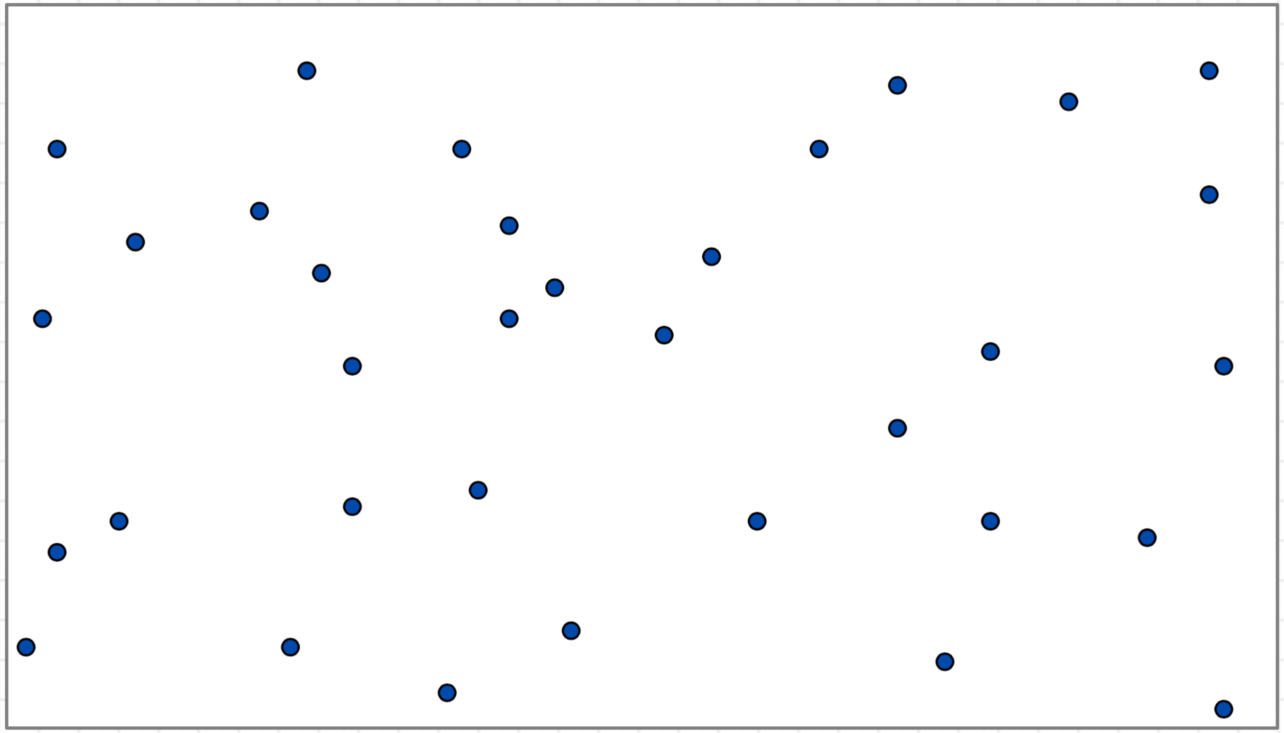
- Problem: Given n points in the plane, find a pair of points with the smallest Euclidean distance between them.



- Brute force.
  - Check all pairs with $\Theta(n^2)$ distance calculations.
- 1D version.
  - Easy $O(n \log n)$ algorithm if points are on a line.
- Non-degeneracy assumption.
  - No two points have the same x-coordinate.

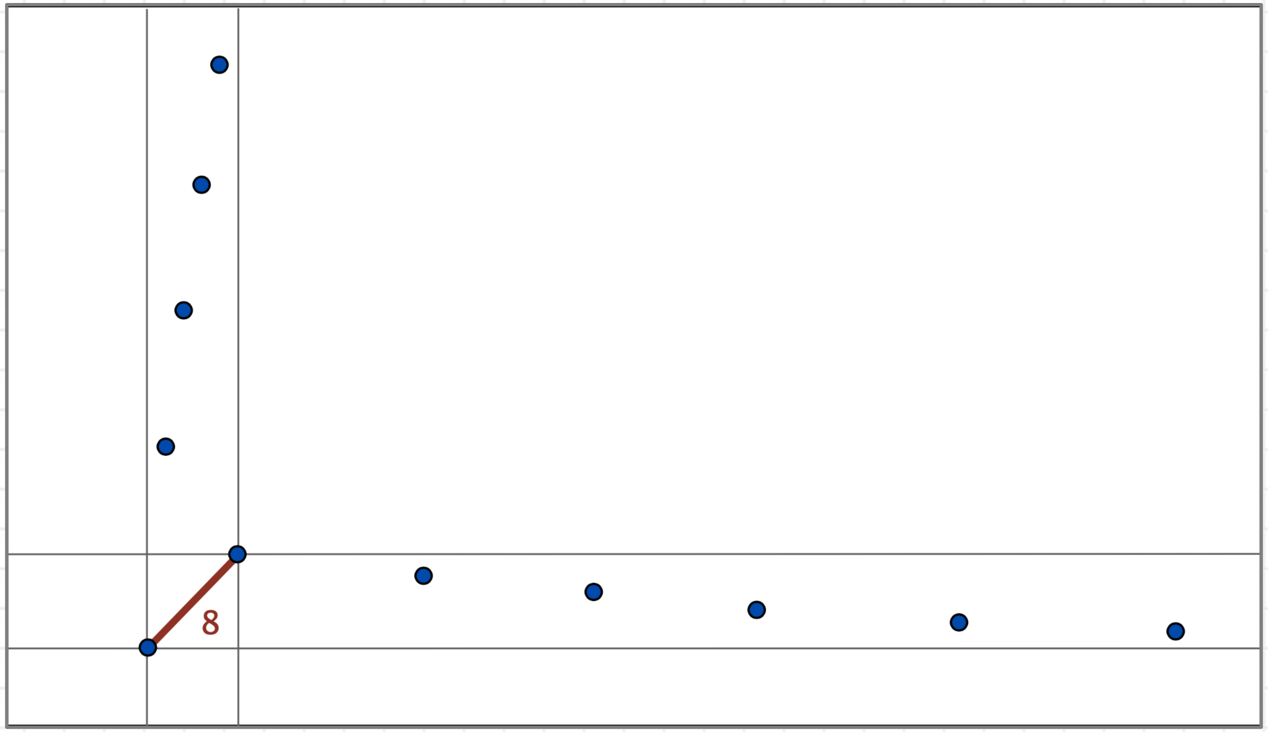
D&C Example: Closest Pair of Points

- Sorting solution?
  - Sort by x-coordinate and consider nearby points.
  - Sort by y-coordinate and consider nearby points.



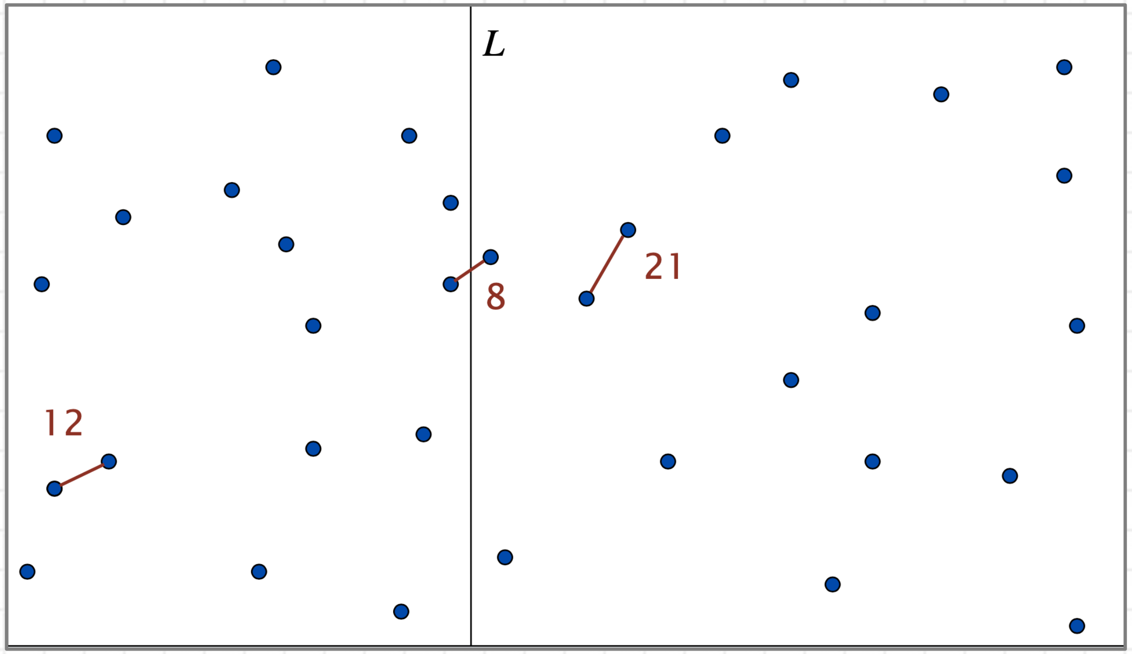
D&C Example: Closest Pair of Points

- Sorting solution? ✗
  - Sort by x-coordinate and consider nearby points.
  - Sort by y-coordinate and consider nearby points.



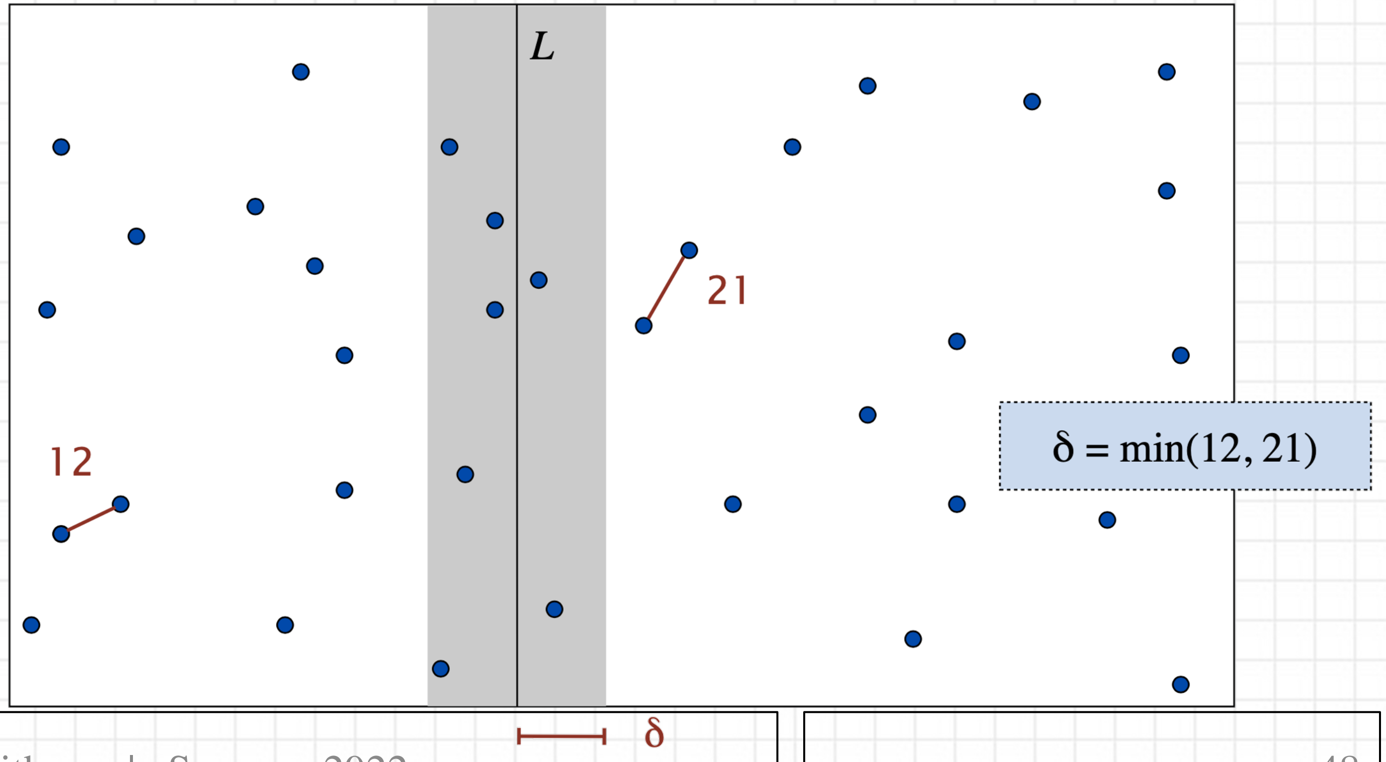
D&C Example: Closest Pair of Points

- Divide-and-Conquer
  - Divide: draw vertical line L so that $n / 2$ points on each side.
  - Conquer: find closest pair in each side recursively.
  - Combine: find closest pair with one point in each side.
    - (How? seems like $\Theta(n2)$?!)
  - Return best of 3 solutions.



D&C Example: Closest Pair of Points

- Divide-and-Conquer
  - Finding closest pair with one point in each side, assuming that distance < δ.
  - Observation: suffices to consider only those points within δ of line *L*.

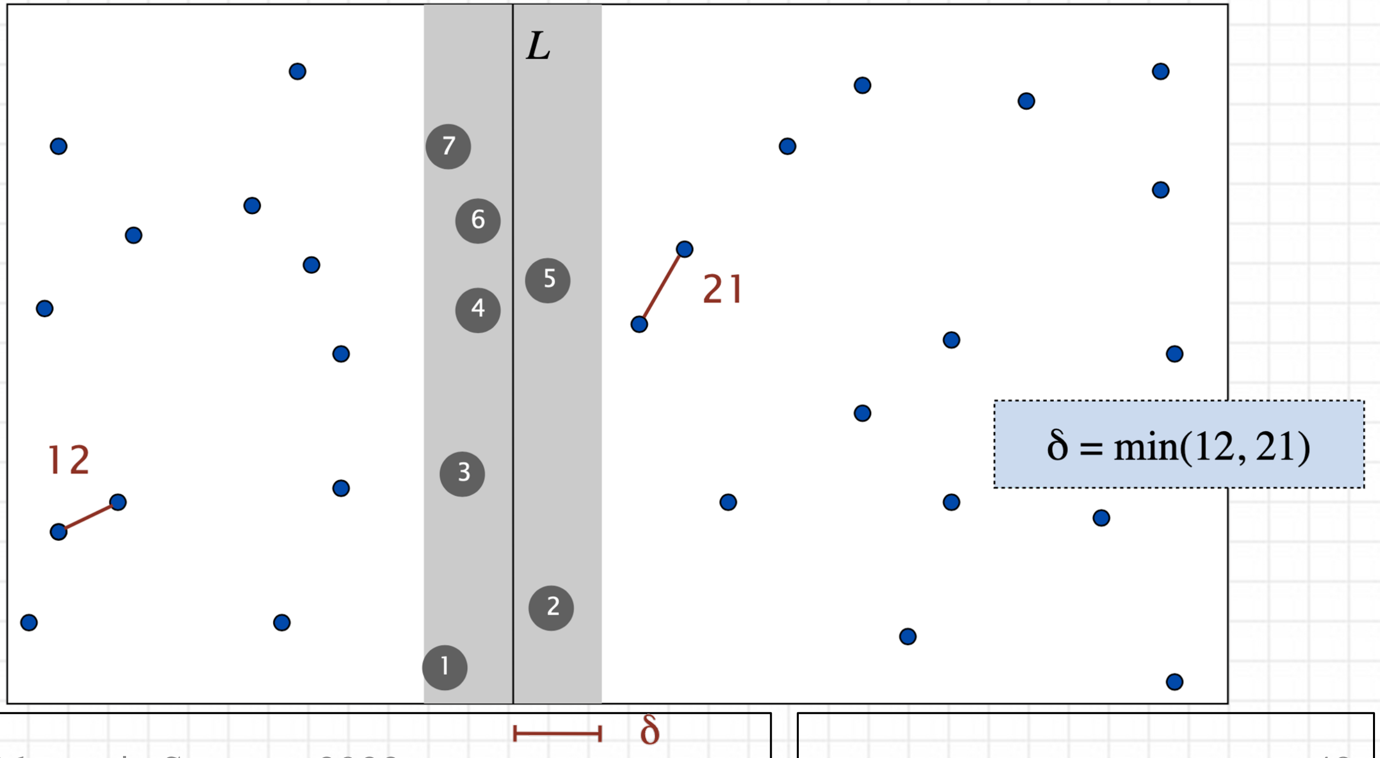


D&C Example: Closest Pair of Points

- Divide-and-Conquer
  - Finding closest pair with one point in each side, assuming that distance < δ.
  - Observation: suffices to consider only those points within δ of line *L*.

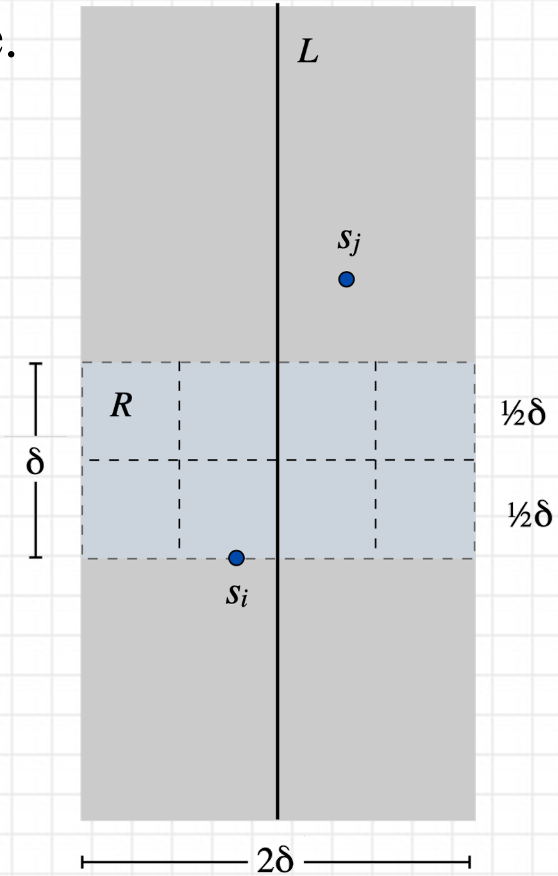
  - Sort points in 2 δ-strip by their *y*-coordinate.

  - Check distances of only those points within 7 positions in sorted list!



D&C Example: Closest Pair of Points

- Divide-and-Conquer
  - Finding closest pair with one point in each side, assuming that distance < δ.
  - Let s_i be the point in the 2 δ-strip, with the i^{th} smallest y-coordinate.
  - Claim: If $|j - i| > 7$, then the distance between s_i and s_j is at least δ.



  - Proof:
    - Consider the 2δ-by-δ rectangle R in strip whose min y-coordinate is y-coordinate of s_i
    - Distance between s_i and any point s_j above R is ≥ δ
    - Subdivide R into 8 squares.
    - At most 1 point per square.

      diameter is $\delta / \sqrt{2} < \delta$
    - At most 7 other points can be in R. ▪

D&C Example: Closest Pair of Points

- Divide-and-Conquer
  - <u>Divide</u>:
    draw vertical line L so
    that n / 2 points on each side.
  - <u>Conquer</u>:
    find closest pair in each side
    recursively.
  - <u>Combine</u>:
    find closest pair with one point
    in each side.
  - Return best of 3 solutions.

CLOSEST-PAIR($p_1, p_2, \ldots, p_n$)

Compute vertical line L such that half the points are on each side of the line. ← $O(n)$

$\delta_1 \leftarrow$ CLOSEST-PAIR(points in left half). ← $T(n / 2)$

$\delta_2 \leftarrow$ CLOSEST-PAIR(points in right half). ← $T(n / 2)$

$\delta \leftarrow \min \{ \delta_1 , \delta_2 \}$.

Delete all points further than δ from line L. ← $O(n)$

Sort remaining points by y-coordinate. ← $O(n \log n)$

Scan points in y-order and compare distance between each point and next 7 neighbors. If any of these distances is less than δ, update δ. ← $O(n)$

RETURN δ.

D&C Example: Closest Pair of Points

- Divide-and-Conquer
- Runtime?

$$T(n) \leq 2T(n/2) + O(n \log n)$$

$$\Rightarrow T(n) = O(n \log^2 n)$$

Q. Can we achieve O(n log n)?

A. Yes. Don't sort points in strip from scratch each time. Each recursive returns two lists: all points sorted by y coordinate, and all points sorted by x coordinate. Sort by merging two pre-sorted lists.

CLOSEST-PAIR($p_1, p_2, \ldots, p_n$)

Compute vertical line L such that half the points are on each side of the line.	$O(n)$
$\delta_1 \leftarrow$ CLOSEST-PAIR(points in left half).	$T(n / 2)$
$\delta_2 \leftarrow$ CLOSEST-PAIR(points in right half).	$T(n / 2)$
$\delta \leftarrow \min \{ \delta_1, \delta_2 \}$.	
Delete all points further than δ from line L.	$O(n)$
Sort remaining points by y-coordinate.	$O(n \log n)$
Scan points in y-order and compare distance between each point and next 7 neighbors. If any of these distances is less than δ, update δ.	$O(n)$
RETURN δ.	

$$T(n) \leq 2T(n/2) + O(n) \quad \Rightarrow \quad T(n) = O(n \log n)$$

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.