

CCES – College of Computer Engineering and Sciences

Prince Sattam Bin Abdulaziz University

CS3401 – Algorithm Design and Analysis

Chapter 4

Decrease & Conquer and Divide & Conquer

Academic Year 2026 – 2027



Dr. Khalil Chebil

A white capital letter 'E' centered within a white circle.

Dr. Esraa Eldesouky

Chapter Overview

Recursive Reduction — A Unified View

Decrease by a Constant

Decrease by a Constant Factor

Variable-Size Decrease

Divide & Conquer — Introduction

Mergesort

Quicksort

Binary Trees & Closest Pair

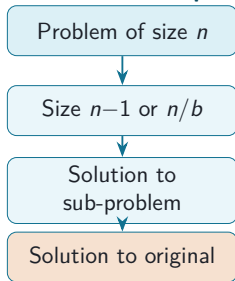
Section 1

Recursive Reduction — A Unified View

Two Sides of the Same Coin

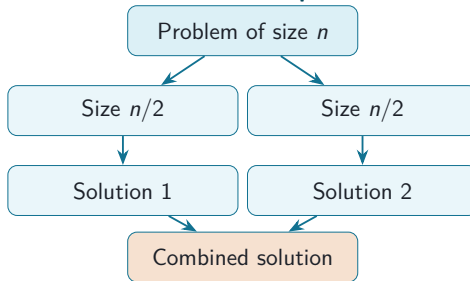
Both strategies rely on **recursive reduction**: *solve a large instance by solving one or more smaller instances.*

Decrease & Conquer



ONE smaller sub-problem

Divide & Conquer



TWO (or more) equal sub-problems + combine step

Three Variants of Decrease & Conquer

1. Decrease by a constant (usually by 1)

$$T(n) \rightarrow T(n-1) + \text{work}$$

- ▶ Insertion sort, topological sort
- ▶ Generating permutations and subsets

2. Decrease by a constant factor (usually by 2)

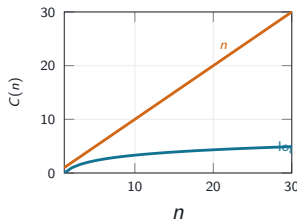
$$T(n) \rightarrow T(n/2) + \text{work}$$

- ▶ Binary search, fake-coin problem
- ▶ Exponentiation by squaring, Russian peasant mult.

3. Variable-size decrease

$$T(n) \rightarrow T(n') + \text{work}, \text{ where } n' \text{ varies}$$

- ▶ Euclid's GCD, Quickselect, BST operations



Decrease-by-factor $\Rightarrow \mathcal{O}(\log n)$

Decrease-by-1 $\Rightarrow \mathcal{O}(n)$ (or $\mathcal{O}(n \log n)$)

Section 2

Decrease by a Constant

Insertion Sort

Idea: To sort $A[0..n-1]$, first sort $A[0..n-2]$, then *insert* $A[n-1]$ into the sorted part.

Algorithm

```
1: for  $i \leftarrow 1$  to  $n-1$ 
2:    $v \leftarrow A[i]; j \leftarrow i - 1$ 
3:   while  $j \geq 0$  and  $A[j] > v$ 
4:      $A[j+1] \leftarrow A[j]; j \leftarrow j - 1$ 
5:   end while
6:    $A[j+1] \leftarrow v$ 
7: end for
```

Trace on $[5, 3, 8, 1]$:

5 3 8 1
3 5 8 1
3 5 8 1
1 3 5 8

Comparisons

Best (sorted)	$n - 1$	$\Theta(n)$
Worst (reverse)	$n(n - 1)/2$	$\Theta(n^2)$
Average	$n^2/4$	$\Theta(n^2)$
Stable: Yes In-place: Yes		

Topological Sorting

Definition

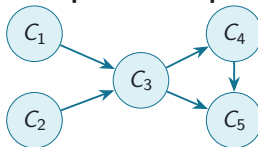
A **topological sort** of a DAG is a linear ordering of its vertices such that for every directed edge (u, v) , u appears before v .

Source-removal algorithm:

1. Compute in-degree for all vertices
2. Enqueue all vertices with in-degree = 0
3. Repeat until queue is empty:
 - Dequeue vertex v , add to sorted list
 - Decrease in-degree of all neighbors of v
 - Enqueue neighbors whose in-degree becomes 0
4. If sorted list has $< n$ vertices: **cycle detected**

Complexity: $\mathcal{O}(|V| + |E|)$

Example: course prerequisites



Result: C_1, C_2, C_3, C_4, C_5

In-degrees:

$C_1=0, C_2=0, C_3=2, C_4=1, C_5=2$

Topological Sorting — DFS Method

Second method: depth-first search.

1. Run DFS on the digraph
2. When a vertex becomes a *dead end* (all neighbours visited), **pop** it and push it onto a stack
3. Reversing the popping-off order gives a valid topological sort

Cycle detection

If a **back edge** is encountered during DFS, the digraph is *not* a DAG and topological sorting is impossible.

Complexity: $\mathcal{O}(|V| + |E|)$

DFS traversal stack

$C_{2_{5,5}}$
$C_{5_{4,1}}$
$C_{4_{3,2}}$
$C_{3_{2,3}}$
$C_{1_{1,4}}$

Popping-off order:

C_5, C_4, C_3, C_1, C_2

Topologically sorted list (reverse):

C_2, C_1, C_3, C_4, C_5

Generating Subsets — Binary Representation

Bit-string representation: Each subset of $\{a_1, \dots, a_n\} \leftrightarrow$ a binary string $b_1 b_2 \dots b_n$ of length n ($b_i = 1$: a_i included, $b_i = 0$: excluded). Generate the binary numbers from 0 to $2^n - 1$.

Algorithm

- 1: **for** $k \leftarrow 0$ **to** $2^n - 1$
- 2: $b_1 b_2 \dots b_n \leftarrow k$ written with n bits
- 3: output $\{a_i \mid b_i = 1\}$
- 4: **end for**

Note: not squashed order ($\{a_3\}$ before $\{a_1\}$).
Reading the bits right to left gives the squashed order (next slide).

All subsets of $\{a_1, a_2, a_3\}$:

Bits	Subset
000	$\emptyset$
001	$\{a_3\}$
010	$\{a_2\}$
011	$\{a_2, a_3\}$
100	$\{a_1\}$
101	$\{a_1, a_3\}$
110	$\{a_1, a_2\}$
111	$\{a_1, a_2, a_3\}$

Total: $2^3 = 8$ subsets

$\mathcal{O}(2^n)$

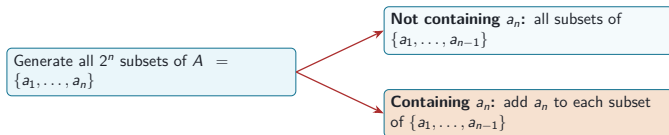
Generating Subsets — Bottom-Up

Idea: Build the 2^n subsets of $A = \{a_1, \dots, a_n\}$ *incrementally*, adding one element at a time.

Key observation: Every subset of $\{a_1, \dots, a_n\}$ either contains a_n or it does not.

Recurrence:

$$\text{Subsets}(n) = \text{Subsets}(n-1) \cup \{S \cup \{a_n\} \mid S \in \text{Subsets}(n-1)\}$$



n	subsets
0	$\emptyset$
1	$\emptyset$ $\{a_1\}$
2	$\emptyset$ $\{a_1\}$ $\{a_2\}$ $\{a_1, a_2\}$
3	$\emptyset$ $\{a_1\}$ $\{a_2\}$ $\{a_1, a_2\}$ $\{a_3\}$ $\{a_1, a_3\}$ $\{a_2, a_3\}$ $\{a_1, a_2, a_3\}$

not containing a_3

containing a_3

This order is the **squashed order**: subsets containing a_j appear only after all the subsets of $\{a_1, \dots, a_{j-1}\}$.

Generating Subsets — Binary Reflected Gray Code

Idea: A **minimal-change** algorithm generating the 2^n bit strings for all subsets of an n -element set, so that *successive* strings differ in exactly one bit.

Algorithm BRGC(n)

```
1: if  $n = 1$  then
2:   make list  $L$  of bit strings 0 and 1
3: else
4:    $L_1 \leftarrow \text{BRGC}(n - 1)$ 
5:    $L_2 \leftarrow$  copy of  $L_1$  in reverse order
6:   add 0 in front of each string in  $L_1$ 
7:   add 1 in front of each string in  $L_2$ 
8:    $L \leftarrow L_1$  appended with  $L_2$ 
9: end if
10: return  $L$ 
```

$n = 2$:

00 01 11 10

$n = 3$:

000 001 011 010 110 111 101 100

$n = 4$:

0000 0001 0011 0010
0110 0111 0101 0100
1100 1101 1111 1110
1010 1011 1001 1000

Each string differs from the next in exactly
one bit .

Section 3

Decrease by a Constant Factor

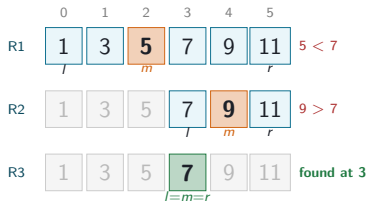
Binary Search

Problem: Find key K in a *sorted* array $A[0..n-1]$.

Algorithm

```
1:  $l \leftarrow 0$ ;  $r \leftarrow n - 1$ 
2: while  $l \leq r$ 
3:    $m \leftarrow \lfloor (l + r) / 2 \rfloor$ 
4:   if  $K = A[m]$  then return  $m$ 
5:   else if  $K < A[m]$  then
6:      $r \leftarrow m - 1$ 
7:   else
8:      $l \leftarrow m + 1$ 
9:   end if
10: end while
11: return  $-1$ 
```

Trace: search for $K = 7$ in $[1, 3, 5, 7, 9, 11]$



$$T(n) = T(n/2) + \mathcal{O}(1) \implies \mathcal{O}(\log n)$$

Fake Coin Problem

Problem: Among n identical-looking coins, one is counterfeit (lighter). Find it using a balance scale.

Decrease-by-2 (halving):

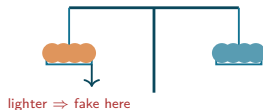
- ▶ Split coins into two equal groups
- ▶ Weigh them — the lighter group contains the fake
- ▶ Recurse on the lighter half

$$W(n) = W(\lfloor n/2 \rfloor) + 1 \implies W(n) = \lfloor \log_2 n \rfloor$$

Decrease-by-3 (thirds) — even better:

- ▶ Split into three equal groups; weigh two of them
- ▶ One weighing eliminates 2/3 of candidates

$$W(n) = W(\lceil n/3 \rceil) + 1 \implies W(n) = \lceil \log_3 n \rceil$$



Key insight

Decrease-by-3 requires $\lceil \log_3 n \rceil$ weighings, fewer than $\lfloor \log_2 n \rfloor$ for $n > 3$.

Russian Peasant Multiplication

Problem: Compute $m \times n$ using only halving, doubling, and addition.

Recurrence:

$$m \cdot n = \begin{cases} \frac{m}{2} \cdot 2n & m \text{ even} \\ \frac{m-1}{2} \cdot 2n + n & m \text{ odd} \end{cases}$$

Add n whenever m is *odd*; always halve m and double n .

Example: 20×26

m	n	add?
20	26	—
10	52	—
5	104	+104
2	208	—
1	416	+416

Result = 104 + 416 = 520

$\mathcal{O}(\log n)$



Section 4

Variable-Size Decrease

Quickselect — Finding the k -th Smallest

Problem: Find the k -th smallest element in an unsorted array of n elements (e.g., the *median*).

Lomuto partition around pivot p (first element):

$$\underbrace{A[0] \cdots A[s-1]}_{\leq p} \quad p \quad \underbrace{A[s+1] \cdots A[n-1]}_{\geq p}$$

After partition, pivot is at index s :

- ▶ If $s = k - 1$: **done**, return $A[s]$
- ▶ If $s > k - 1$: recurse on left part
- ▶ If $s < k - 1$: recurse on right part

Complexities:

Case	Complexity
Best	$\Theta(n)$
Worst	$\Theta(n^2)$
Average	$\Theta(n)$

Real-world use

Finding the median salary among 1 million records *without* sorting them all — used in statistics, databases, and data science pipelines.

Two Partitioning Algorithms

Partition a list around a pivot p (the first element) so the left part holds all elements $\leq p$, then p , then all elements $\geq p$.

Lomuto — one-directional scan

Input: Subarray $A[l..r]$

```
1:  $p \leftarrow A[l]; \quad s \leftarrow l$ 
2: for  $i \leftarrow l + 1$  to  $r$ 
3:   if  $A[i] < p$  then
4:      $s \leftarrow s + 1; \text{SWAP}(A[s], A[i])$ 
5:   end if
6: end for
7:  $\text{SWAP}(A[l], A[s])$ 
8: return  $s$ 
```

Single index s marks the end of the “ $< p$ ” region;
simple, one swap per smaller element.

Hoare — two-directional scan

Input: Subarray $A[l..r]$

```
1:  $p \leftarrow A[l]; \quad i \leftarrow l; \quad j \leftarrow r + 1$ 
2: repeat
3:   repeat  $i \leftarrow i + 1$ 
4:   until  $A[i] \geq p$ 
5:   repeat  $j \leftarrow j - 1$ 
6:   until  $A[j] \leq p$ 
7:    $\text{SWAP}(A[i], A[j])$ 
8: until  $i \geq j$ 
9:  $\text{SWAP}(A[i], A[j])$ 
10:  $\text{SWAP}(A[l], A[j])$ 
11: return  $j$ 
```

▷ undo last

Partitioning Example — Lomuto vs Hoare

Partition $A = [5, 3, 8, 4, 2, 7, 1]$ around pivot $p = A[0] = 5$.

Lomuto (scan i left→right, grow s):

Array	s action
[<u>5</u> 3 8 4 2 7 1]	0 start, $p=5$
[5 <u>3</u> 8 4 2 7 1]	1 $3 < 5$: swap self
[5 3 <u>8</u> 4 2 7 1]	1 $8 \geq 5$: skip
[5 3 4 <u>8</u> 2 7 1]	2 $4 < 5$: swap 8,4
[5 3 4 2 <u>8</u> 7 1]	3 $2 < 5$: swap 8,2
[5 3 4 2 8 <u>7</u> 1]	3 $7 \geq 5$: skip
[5 3 4 2 1 <u>7</u> 8]	4 $1 < 5$: swap 8,1
[1 3 4 2 5 7 8]	4 swap $A[0], A[4]$

Pivot lands at $s = 4$; **5 swaps** total.

Hoare (scan $i \rightarrow$ right, $j \leftarrow$ left):

Array	i, j	action
[5 3 8 4 2 7 1]	$i=2, j=6$	$A[i]=8 \geq 5, A[j]=1 \leq 5$
[5 3 1 4 2 7 8]	swap	$8 \leftrightarrow 1$
[5 3 1 4 2 7 8]	$i=5, j=4$	$A[i]=7 \geq 5, A[j]=2 \leq 5$
	$i \geq j$	stop, undo
[2 3 1 4 5 7 8]	$j=4$	swap $A[0], A[4]$

Pivot lands at $j = 4$; only **2 swaps**.

Key difference

Both put the pivot in its final sorted position and return a split index, but **Hoare** does *fewer swaps* (two-way scan) while **Lomuto** is *simpler* (one-way scan). The two partitions may order the sub-arrays differently, yet both are valid.

Interpolation Search

Idea: Like binary search, but probe position is estimated from the *value* of the key, assuming uniform distribution.

Algorithm sketch

```
1: while  $l \leq r$  and  $v \in [A[l], A[r]]$ 
2:   compute probe position  $x$  via formula
3:   if  $A[x] = v$  then return  $x$ 
4:   else if  $A[x] < v$  then
5:      $l \leftarrow x + 1$ 
6:   else
7:      $r \leftarrow x - 1$ 
8:   end if
9: end while
10: return  $-1$ 
```

$$x = l + \left\lfloor \frac{(v - A[l])(r - l)}{A[r] - A[l]} \right\rfloor$$

Case	Complexity
Best (1st probe hits)	$\mathcal{O}(1)$
Average (uniform)	$\mathcal{O}(\log \log n)$
Worst	$\mathcal{O}(n)$

When to prefer over binary search

Only for *very large* sorted arrays with *uniformly distributed* values. Otherwise binary search is safer.

Interpolation vs Binary Search — Example

Search for $v = 18$ in the sorted array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	12	13	16	18	19	20	21	22	23	24	33	35	42	47

Binary Search — 4 steps

1. $p = \lfloor \frac{0+14}{2} \rfloor = 7$, $A[7]=21 \geq 18 \Rightarrow$ go left $[0..6]$
2. $p = \lfloor \frac{0+6}{2} \rfloor = 3$, $A[3]=16 \leq 18 \Rightarrow$ go right $[4..6]$
3. $p = \lfloor \frac{4+6}{2} \rfloor = 5$, $A[5]=19 \geq 18 \Rightarrow$ go left $[4..4]$
4. $p = \lfloor \frac{4+4}{2} \rfloor = 4$, $A[4]=18$ **found!**

Position ignores the *value* of the key — always halves.

Interpolation Search — 2 steps

1. $p = 0 + \lfloor \frac{(18-10)(14-0)}{47-10} \rfloor = \lfloor \frac{112}{37} \rfloor = 3$,
 $A[3]=16 \leq 18 \Rightarrow$ go right $[4..14]$
2. $p = 4 + \lfloor \frac{(18-16)(14-4)}{47-18} \rfloor = 4 + 0 = 4$,
 $A[4]=18$ **found!**

The formula jumps near the key using its value — $\mathcal{O}(\log \log n)$ when data is uniform.

Section 5

Divide & Conquer — Introduction

Divide & Conquer: The General Recurrence

Recurrence

$$T(n) = a T(n/b) + f(n), \quad f(n) \in \Theta(n^d), \quad d \geq 0$$

Master Theorem

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \text{ (combine dominates)} \\ \Theta(n^d \log n) & \text{if } a = b^d \text{ (balanced)} \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \text{ (recursion dominates)} \end{cases}$$

Recurrence	a	b	d	Case	Result
$3T(n/2) + n$	3	2	1	$a > b^d$	$\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$
$3T(n/2) + n^2$	3	2	2	$a < b^d$	$\Theta(n^2)$
$4T(n/2) + n^2$	4	2	2	$a = b^d$	$\Theta(n^2 \log n)$
$2T(n/2) + n$	2	2	1	$a = b^d$	$\Theta(n \log n)$

Does not apply when:

- ▶ $a < 1$ or $d < 0$
- ▶ $f(n)$ is not polynomial
- ▶ a is not a constant

Memory aid

Compare the *branching factor* a with the *shrink-work ratio* b^d :

- ▶ $a < b^d$: work at root dominates
- ▶ $a = b^d$: cost spreads evenly
- ▶ $a > b^d$: leaves dominate

Section 6

Mergesort

Mergesort — Algorithm

Idea: Split array in half, sort each half recursively, then *merge* the two sorted halves.

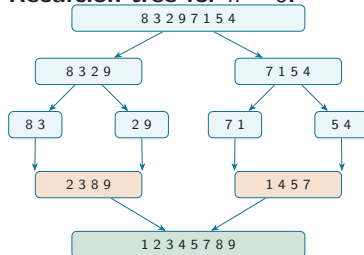
Mergesort

Input: Array $A[0..n-1]$

- 1: **if** $n > 1$ **then**
- 2: split A into $B[0..\lfloor n/2 \rfloor - 1]$ and $C[...]$
- 3: MERGESORT(B)
- 4: MERGESORT(C)
- 5: MERGE(B, C, A)
- 6: **end if**

Merge: compare heads of B and C , copy smaller.
Runs in $\Theta(p + q)$ where $|B| = p$, $|C| = q$.

Recursion tree for $n = 8$:



Mergesort — The Merge Step

Merge($B[0..p-1]$, $C[0..q-1]$, A)

```
1:  $i \leftarrow 0$ ;  $j \leftarrow 0$ ;  $k \leftarrow 0$ 
2: while  $i < p$  and  $j < q$ 
3:   if  $B[i] \leq C[j]$  then
4:      $A[k] \leftarrow B[i]$ ;  $i \leftarrow i + 1$ 
5:   else
6:      $A[k] \leftarrow C[j]$ ;  $j \leftarrow j + 1$ 
7:   end if
8:    $k \leftarrow k + 1$ 
9: end while
10: if  $i = p$  then copy  $C[j..q-1]$  to  $A[k..]$ 
11: else copy  $B[i..p-1]$  to  $A[k..]$ 
12: end if
```

Takes $\Theta(p + q)$: each element of B and C is copied exactly once.

Merge $B = [1, 13, 24, 26]$, $C = [2, 15, 27, 38]$:

B				C				A							
1	13	24	26	2	15	27	38	1							
1	13	24	26	2	15	27	38	1	2						
1	13	24	26	2	15	27	38	1	2	13					
1	13	24	26	2	15	27	38	1	2	13	15				
1	13	24	26	2	15	27	38	1	2	13	15	24			
1	13	24	26	2	15	27	38	1	2	13	15	24	26		
1	13	24	26	2	15	27	38	1	2	13	15	24	26	27	
1	13	24	26	2	15	27	38	1	2	13	15	24	26	27	38

The smaller head (highlighted) is copied to A each step; when one list empties, the rest of the other is appended.

Mergesort — Analysis

Recurrence:

$$T(n) = 2T(n/2) + \Theta(n), \quad T(1) = \mathcal{O}(1)$$

Master Theorem: $a = 2, b = 2, d = 1 \implies a = b^d$

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

All cases are $\Theta(n \log n)$ (the merge always takes $\Theta(n)$).

Number of comparisons (worst case) $\approx n \log_2 n$, close to the information-theoretic lower bound

$$\lceil \log_2 n! \rceil \approx n \log_2 n - 1.44n.$$

Mergesort Bubble

Best	$\Theta(n \log n)$	$\Theta(n)$
Worst	$\Theta(n \log n)$	$\Theta(n^2)$
Average	$\Theta(n \log n)$	$\Theta(n^2)$
Space	$\Theta(n)$	$\Theta(1)$
Stable	Yes	Yes

Trade-off

Mergesort is not *in-place* — it requires $\Theta(n)$ extra space for the merge step.
Prefer it for *linked lists*.



Section 7

Quicksort

Quicksort — Algorithm

Idea: Choose a *pivot*, partition the array so that all elements $\leq$ pivot come first, then recursively sort both parts.

Quicksort

Input: Subarray $A[l..r]$

```
1: if  $l < r$  then
2:    $s \leftarrow \text{PARTITION}(A[l..r])$ 
3:    $\text{QUICKSORT}(A[l..s-1])$ 
4:    $\text{QUICKSORT}(A[s+1..r])$ 
5: end if
```

Hoare partition scans inward from both ends, swapping out-of-place elements until the pointers cross.

Pivot strategies:

- **First element** — simple, **bad on sorted input**
- **Median-of-three** — good average behaviour
- **Random element** — avoids adversarial inputs

Complexities:

Best / Average	$\Theta(n \log n)$
Worst	$\Theta(n^2)$
Space (avg)	$\mathcal{O}(\log n)$ stack
Stable	No
In-place	Yes

Quicksort — Full Example (Hoare partition)

Sort $A = [5, 3, 1, 9, 8, 2, 4, 7]$. Pivot is always the first element.

Partition($A[0..7]$), pivot 5:

5	3	1	9	8	2	4	7	swap 9, 4
5	3	1	4	8	2	9	7	swap 8, 2
5	3	1	4	2	8	9	7	$i \geq j$
2	3	1	4	5	8	9	7	swap pivot

Partition($A[0..7]$) $\Rightarrow$ 4
 $\Rightarrow$ [2, 3, 1, 4] **5** [8, 9, 7]

Quicksort($A[0..3]$): pivot 2 $\Rightarrow$ split 1

2 3 1 4 $\rightarrow$ 1 2 3 4

$Q(A[0..0])=1$, $Q(A[2..3])=[3, 4]$

Quicksort($A[5..7]$): pivot 8 $\Rightarrow$ split 6

8 9 7 $\rightarrow$ 7 8 9

$Q(A[5..5])=7$, $Q(A[7..7])=9$

Final result: 1 2 3 4 5 7 8 9

Quicksort vs Mergesort

Property	Quicksort	Mergesort
Average time	$\Theta(n \log n)$	$\Theta(n \log n)$
Worst time	$\Theta(n^2)$	$\Theta(n \log n)$
Extra space	$\mathcal{O}(\log n)$	$\Theta(n)$
In-place	Yes	No
Stable	No	Yes
Cache perf.	Better	Good

Guideline

- ▶ Prefer **Quicksort** for arrays in practice (better cache behaviour, in-place)
- ▶ Prefer **Mergesort** for linked lists (in-place merge in $\mathcal{O}(1)$ extra space)
- ▶ Use **Timsort** (hybrid) for general-purpose libraries (Python, Java)

Section 8

Binary Trees & Closest Pair

Binary Tree Traversals

A **binary tree** is naturally processed by Divide & Conquer: divide into root + left subtree + right subtree.

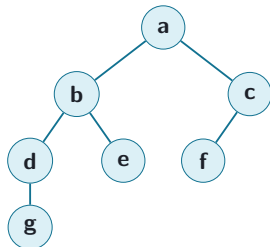
Height of a binary tree

```
1: if  $T = \emptyset$  then return  $-1$ 
2: end if
3: return  $\max(\text{Height}(T_L), \text{Height}(T_R)) + 1$ 
```

Traversal orders:

- ▶ **Preorder**: root, left, right
- ▶ **Inorder**: left, root, right $\leftarrow$ produces sorted order for BST
- ▶ **Postorder**: left, right, root

All traversals: $\Theta(n)$



Preorder: a, b, d, g, e, c, f

Inorder: g, d, b, e, a, f, c

Postorder: g, d, e, b, f, c, a

Closest Pair — Divide & Conquer

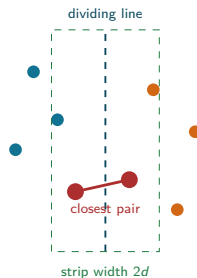
Problem: Find the two closest points in n points in the plane. Brute force: $\mathcal{O}(n^2)$.

D&C algorithm sketch:

1. Sort points by x -coordinate
2. Split at median x into P_L and P_R
3. Recurse: $d_L = \text{closestPair}(P_L)$,
 $d_R = \text{closestPair}(P_R)$
4. $d = \min(d_L, d_R)$
5. Check the "strip" of width $2d$ around the dividing line (at most 8 point-pairs per strip element)

Recurrence: $T(n) = 2T(n/2) + \mathcal{O}(n)$

$\implies T(n) = \mathcal{O}(n \log n)$



Chapter 4 Summary

Decrease & Conquer:

Insertion sort $\mathcal{O}(n^2)$ worst
Topological sort $\mathcal{O}(|V| + |E|)$
Binary search $\mathcal{O}(\log n)$
Quickselect $\mathcal{O}(n)$ avg

Divide & Conquer:

Mergesort $\Theta(n \log n)$ all cases
Quicksort $\Theta(n \log n)$ avg
Closest pair $\Theta(n \log n)$
Tree traversal $\Theta(n)$

Master Theorem

$$T(n) = aT(n/b) + \Theta(n^d):$$

- ▶ $a < b^d$: $\Theta(n^d)$
- ▶ $a = b^d$: $\Theta(n^d \log n)$
- ▶ $a > b^d$: $\Theta(n^{\log_b a})$

Next chapter

Chapter 5: Transform & Conquer — presorting, balanced trees, heaps, and problem reduction (including linear programming).