Big O Common Code Patterns

O(1) O(log n)	O(n)	O(n log n)	O(n²)	O(2 ⁿ)
Array Access Direct access of an element by index				
let arr = [1, 2, 3, 4, 5] let firstElement = arr[0] // 0(1)				
let lastElement = arr[arr.length - 1] // Common Use Cases: Quick lookups in fixed-size arra				
Simple Computation	ys			
Sum two integers				
<pre>func addOne(n: Int) { return n + 1 }</pre>				
Common Use Cases: Managing function calls, undo o	perations			
O(1) O(log n)	O(n)	O(n log n)	O(n²)	O(2 ⁿ)
Binary Search Halving the search space in each step				
<pre>func binarySearch(_ arr: [Int], _ target: var left = 0, right = arr.count - 1</pre>	Int) -> Int {			
while left <= right { let mid = (left + right) / 2				
<pre>if arr[mid] == target { return mi if arr[mid] < target { left = mid + 1</pre>	ld }			
<pre>} else { right = mid - 1 }</pre>				
} return -1 }				
Common Use Cases: Finding elements in sorted array	/S			
Height-Balanced BST Operations Operations on a balanced binary search tree				
class AVLTree {				
<pre>func insert(_ value: Int) { // O(log n) - always balanced root = insert(root, value)</pre>				
<pre>balance(root) }</pre>				
<pre>func search(_ value: Int) -> Bool { // O(log n) - tree height return search(root, value)</pre>				
} Occurred the Occupation of the second data with factors and the second data with factors and the second data with the second data wit				
Common Use Cases: Maintaining sorted data with fas	st operations			
O(1) O(log n)	O(n)	O(n log n)	O(n²)	O(2 ⁿ)
Simple Iteration				
Processing each elements of a collection func sum(_ arr: [Int]) {				
<pre>runc sum(_ arr: [Int]) { var sum = 0 for i in 0<arr.count +="arr[i]</pre" sum="" {=""></arr.count></pre>				
<pre>sum += arr[i] } return sum }</pre>				
Common Use Cases: Key-value storage with potentia	l collisions			
Array/String Search				
Finding an element in an unsorted array func linearSearch(_ arr: [Int], _ target:				
<pre>for (index, value) in arr.enumerated(if value == target { return index</pre>	() {			
} } return -1				
} Common Use Cases: Finding elements in unsorted da	ata			
O(1) O(log n)	O(n)	O(n log n)	O(n²)	O(2 ⁿ)
Merge Sort	O(n)	O(n log n)	O(n²)	O(2 ⁿ)
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] {		O(n log n)	O(n²)	O(2 ⁿ)
<pre>Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2</pre>	- }	O(n log n)	O(n²)	O(2 ⁿ)
<pre>Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mid let="" merge(left,="" pre="" right="mergeSort(Array(arr[mid.return" right)<=""></mid></pre>	- }	O(n log n)	O(n²)	O(2 ⁿ)
<pre>Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mid let="" merge(left,="" pre="" return="" right="mergeSort(Array(arr[mid" right)="" }<=""></mid></pre>	- } (1))	O(n log n)	O(n²)	O(2 ⁿ)
<pre>Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mid a="" cases:="" common="" efficient="" guaranteed="" let="" merge(left,="" of="" pre="" right="mergeSort(Array(arr[midreturn" right)))}="" sort<="" sorting="" stable="" use="" }=""></mid></pre>	- } (1))	O(n log n)	O(n²)	O(2 ⁿ)
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mic "def",="" "ghi"]<="" a="" built-in="" cases:="" common="" efficient="" guarantee="" guaranteed="" items='["abc",' let="" merge(left,="" of="" return="" right="mergeSort(Array(arr[mid" right)="" sort="" sorting="" stability="" stable="" td="" use="" with="" }=""><td>- } (1))</td><td>O(n log n)</td><td>O(n²)</td><td>O(2ⁿ)</td></mic>	- } (1))	O(n log n)	O(n²)	O(2 ⁿ)
<pre>Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mic "def",="" "ghi"]="" \$0="" \$1="" <="" a="" built-in="" cases:="" common="" efficient="" guarantee="" guaranteed="" items='["abc",' let="" merge(left,="" of="" pre="" right="mergeSort(Array(arr[midreturn" right)))="" sort="" sorted="items.sorted" sorting="" stability="" stable="" use="" with="" {="" }="" }<=""></mic></pre>	- } [])) .]))	O(n log n)	O(n²)	O(2 ⁿ)
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mic "def",="" "ghi"]="" \$0="" \$1="" <="" a="" built-in="" cases:="" common="" efficient="" guarantee="" guaranteed="" items='["abc",' let="" merge(left,="" of="" right="mergeSort(Array(arr[midreturn" right)))}="" sort="" sorted="items.sorted" sorting="" stability="" stable="" td="" use="" with="" {="" }<=""><td>- } [])) .]))</td><td>O(n log n)</td><td>O(n²)</td><td>O(2ⁿ)</td></mic>	- } [])) .]))	O(n log n)	O(n²)	O(2 ⁿ)
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mic "def",="" "ghi"]="" \$0="" \$1="" <="" a="" built-in="" cases:="" common="" efficient="" guarantee="" guaranteed="" items='["abc",' let="" merge(left,="" of="" right="mergeSort(Array(arr[midreturn" right)))}="" sort="" sorted="items.sorted" sorting="" stability="" stable="" td="" use="" with="" {="" }<=""><td>- } [])) .]))</td><td>O(n log n) O(n log n)</td><td>O(n²)</td><td>O(2ⁿ)</td></mic>	- } [])) .]))	O(n log n) O(n log n)	O(n²)	O(2 ⁿ)
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mic "def",="" "ghi"]="" \$0="" \$1="" <="" a="" and="" are="" built-in="" cases:="" common="" efficient="" general-purpose="" guarantee="" guaranteed="" items='["abc",' let="" merge(left,="" of="" return="" right="mergeSort(Array(arr[mid" right)="" some="" sort="" sorted="" sorted<="" sorting="" stability="" stable="" td="" the="" use="" when="" with="" {="" }=""><td>arrays</td><td></td><td></td><td></td></mic>	arrays			
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mic "def",="" "ghi"]="" \$0="" \$1="" <="" a="" built-in="" cases:="" common="" efficient="" general-purpose="" guarantee="" guaranteed="" items='["abc",' let="" merge(left,="" of="" right="mergeSort(Array(arr[midreturn" right))="" some="" sort="" sorted="items.sorted" sorti<="" sorting="" stability="" stable="" td="" the="" use="" when="" with="" {="" }=""><td>arrays</td><td></td><td></td><td></td></mic>	arrays			
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mic "def",="" "ghi"]="" \$0="" \$1="" +="" 0<arr.count="" 10="" <="" [int])="" a="" against="" arr:="" arr[i]="" arr[j]="=" built-in="" cases:="" common="" each="" efficient="" element="" every="" findallpairs(_="" for="" func="" general-purpose="" guarantee="" guaranteed="" i="" i+1<arr.count="" if="" in="" items='["abc",' j="" let="" merge(left,="" of="" other="" return="" right="mergeSort(Array(arr[mid" right))="" sort="" sorted="items.sorted" sorting="" stability="" stable="" td="" use="" when="" with="" {="" {<="" }=""><td>arrays O(n)</td><td></td><td></td><td></td></mic>	arrays O(n)			
<pre>Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mic "def",="" "ghi"]="" \$0="" \$1="" +="" 0<arr.count="" 10="" <="" [int])="" \(\chi_{i})="" \(arr[i]),="" \)="" a="" against="" arr:="" arr[i]="" arr[j]="=" built-in="" cases:="" common="" each="" efficient="" element="" every="" findallpairs(_="" for="" func="" general-purpose="" guarantee="" guaranteed="" i="" if="" in="" items='["abc",' iteration="" let="" merge(left,="" n)="" nested="" o(1)="" o(log="" of="" other="" pre="" print("pair:="" processing="" return="" right="mergeSort(Array(arr[mid" right))="" sort="" sorted="items.sorted" sorting="" stability="" stable="" use="" when="" with="" {="" }="" }<=""></mic></pre>	arrays O(n)			
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mid "def",="" "ghi"]="" \$0="" \$1="" <="" a="" built-in="" cases:="" common="" efficient="" general-purpose="" guarantee="" guaranteed="" i<="" in="" items='["abc",' let="" merge(left,="" of="" right="mergeSort(Array(arr[midreturn" right)))}="" some="" sort="" sorted="items.sorted" sorting="" stability="" stable="" td="" use="" when="" with="" {="" }=""><td>arrays O(n)</td><td></td><td></td><td></td></mid>	arrays O(n)			
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mic "def",="" "ghi"]="" \$0="" \$1="" +="" 0<arr.count="" 10="" <="" [int])="" \(ar<="" \(arr[i]),="" a="" against="" arr:="" arr[i]="" arr[j]="=" cases:="" common="" each="" efficient="" element="" every="" findallpairs(_="" for="" func="" general-purpose="" guarantee="" guaranteed="" i="" if="" in="" items='["abc",' let="" merge(left,="" of="" other="" print("pair:="" right="mergeSort(Array(arr[midreturn" right))="" sorted="items.sorted" sorting="" stability="" td="" use="" when="" with="" {="" }=""><td>arrays O(n)</td><td></td><td></td><td></td></mic>	arrays O(n)			
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mic "def",="" "ghi"]="" \$0="" \$1="" +="" 0<arr.count="" 10="" <="" \(arr[i]),="" \(arr[i]<="" a="" arr[i]="" arr[j]="=" built-in="" cases:="" common="" efficient="" for="" general-purpose="" guarantee="" guaranteed="" i="" if="" in="" items='["abc",' let="" merge(left,="" of="" print("pair:="" right="mergeSort(Array(arr[midreturn" right)))}="" sort="" sorted="items.sorted" sorting="" stability="" stable="" td="" use="" when="" with="" {="" }=""><td>arrays O(n)</td><td></td><td></td><td></td></mic>	arrays O(n)			
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mic "def",="" "ghi"]="" \$0="" \$1="" +="" 0<arr.count="" 10="" <="" \(arr[i]),="" \(arr[i]<="" a="" arr[i]="" arr[j]="=" built-in="" cases:="" common="" efficient="" for="" general-purpose="" guarantee="" guaranteed="" i="" if="" in="" items='["abc",' let="" merge(left,="" of="" print("pair:="" right="mergeSort(Array(arr[midreturn" right)))}="" sort="" sorted="items.sorted" sorting="" stability="" stable="" td="" use="" when="" with="" {="" }=""><td>stability matters O(n) On Int, _ high: In</td><td>O(n log n)</td><td></td><td></td></mic>	stability matters O(n) On Int, _ high: In	O(n log n)		
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<midlet "def",="" "ghi"]="" \$0="" \$1="" <="" a="" built-in="" cases:="" common="" efficient="" general-purpose="" guarantee="" guaranteed="" in="" items='["abc",' let="" merge(left,="" of="" right="mergeSort(Array(arr[mid.return" right))="" some="" sor<="" sort="" sorted="" sorting="" stability="" stable="" td="" use="" when="" with="" {="" }=""><td>arrays O(n) O(n) Int, _ high: Int</td><td>O(n log n)</td><td></td><td></td></midlet>	arrays O(n) O(n) Int, _ high: Int	O(n log n)		
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[mid.return merge(left, right)) } Common Use Cases: Guaranteed efficient sorting of some sorting with stability guarantee let items = ["abc", "def", "ghi"] let sorted = items.sorted { \$0 < \$1 } Common Use Cases: General-purpose sorting when some sorting each element against every other func findAllPairs(_ arr: [Int]) { for i in 0 <arr.count +="" 10="" \(ar<="" \(arr[i]),="" arr[i]="" arr[j]="=" if="" print("pair:="" td="" {=""><td>arrays O(n) O(n) Int, _ high: Int</td><td>O(n log n)</td><td></td><td></td></arr.count>	arrays O(n) O(n) Int, _ high: Int	O(n log n)		
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mic "def",="" "ghi"]="" \$0="" \$1="" (worst="" +="" -="" 0<arr.count="" 1)="" 1,="" 10="" <="" [int])="" [int],="" \(arr[i]),="" \\="" _="" against="" all="" already="" arr:="" arr[i]="" arr[j]="=" array="" built-in="" case)="" cases:="" combination="" common="" each="" efficient="" element="" else="" every="" findallpairs(_="" finding="" for="" func="" general-purpose="" guarantee="" guaranteed="" guard="" high="" high)="" i="" if="" in="" inout="" is="" items='["abc",' let="" low="" low,="" low:="" merge(left,="" o(n²)="" of:="" or="" other="" pivot="" possible="" print("pair:="" quick="" quicksort(&arr,="" quicksort(_="" return="" reverse="" right="mergeSort(Array(arr[midreturn" right))="" selection="" some="" sort="" sorted="" sorting="" stability="" stable="" td="" unfortunate="" use="" when="" with="" {="" }="" }<=""><td>arrays Stability matters O(n) O(n)</td><td>O(n log n)</td><td></td><td></td></mic>	arrays Stability matters O(n) O(n)	O(n log n)		
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mic "def",="" "ghi"]="" \$0="" \$1="" (worst="" +="" -="" 0(n²)="" 0<arr.count="" 1)="" 1,="" 10="" <="" [int])="" [int],="" \(arr[i]),="" \\="" _="" against="" all="" already="" arr:="" arr[i]="" arr[j]="=" array="" built-in="" case)="" cases:="" combination="" common="" each="" efficient="" element="" else="" every="" findallpairs(_="" finding="" for="" func="" general-purpose="" guarantee="" guaranteed="" guard="" high="" high)="" i="" if="" in="" inout="" is="" items='["abc",' let="" low="" low,="" low:="" merge(left,="" of:="" or="" other="" pivot="" possible="" print("pair:="" quick="" quicksort(&arr,="" quicksort(_="" return="" reverse="" right="mergeSort(Array(arr[midreturn" right))="" selection="" some="" sort="" sorted="" sorting="" stability="" stable="" td="" unfortunate="" use="" when="" with="" {="" }="" }<=""><td>arrays Stability matters O(n) O(n)</td><td>O(n log n)</td><td></td><td></td></mic>	arrays Stability matters O(n) O(n)	O(n log n)		
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mic "def",="" "ghi"]="" \$0="" \$1="" (worst="" +="" -="" 0<arr.count="" 1)="" 1,="" 10="" <="" [int])="" [int],="" \(arr[i]),="" \\="" _="" against="" all="" already="" and="" are="" arr:="" arr[i]="" arr[j]="=" array="" built-in="" case)="" cases:="" combination="" common="" each="" efficient="" element="" else="" every="" findallpairs(_="" finding="" for="" func="" general-purpose="" guarantee="" guaranteed="" guard="" high="" high)="" i="" if="" in="" inout="" is="" items="" items.="" iteration="" let="" low="" low,="" low:="" merge(left,="" n)="" nested="" o(1)="" o(log="" o(n²)="" of="" of:="" or="" other="" pivot="" possible="" print("pair:="" processing="" quick="" quicksort(&arr,="" quicksort(_="" return="" reverse="" right="mergeSort(Array(arr[midreturn" right))="" selection="" some="" sort="" sorted="" sorting="" stability="" stable="" td="" unfortunate="" use="" when="" with="" {="" }="" }<=""><td>arrays Stability matters O(n) O(n)</td><td>O(n log n)</td><td></td><td></td></mic>	arrays Stability matters O(n) O(n)	O(n log n)		
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[midreturn merge(left, right)) } Common Use Cases: Guaranteed efficient sorting of: Stable Sort Built-in sorting with stability guarantee let items = ["abc", "def", "ghi"] let sorted = items.sorted { \$0 < \$1 } Common Use Cases: General-purpose sorting when some sorting with stability guarantee let items = ["abc", "def", "ghi"] let sorted = items.sorted { \$0 < \$1 } Common Use Cases: General-purpose sorting when some sorting with stability guarantee let items = ["abc", "def", "ghi"] let sorted = items.sorted { \$0 < \$1 } Common Use Cases: General-purpose sorting when some sorting when some sorting when some sorting with stability guarantee func findAllPairs(_ arr: [Int]) { for i in 0 <arr.count (worst="" +="" -="" 0(n²)="" 1)="" 1,="" 10="" <="" [int],="" \(\)="" \(arr[i]),="" _="" all="" already="" arr:="" arr[i]="" arr[j]="=" array="" cas="" case)="" cases:="" combination="" common="" else="" fibonacci<="" finding="" func="" general="" guard="" high="" high)="" if="" inout="" is="" let="" low="" low,="" low:="" n)="" o(1)="" o(log="" or="" pivot="" possible="" print("pair:="" quick="" quicksort(&arr,="" quicksort(_="" recursive="" return="" reverse="" selection="" sort="" sorted="" sorting="" td="" unfortunate="" use="" when="" worst="" {="" }=""><td>arrays O(n) Int, _ high: Int is of pairs</td><td>O(n log n)</td><td>O(n²)</td><td>O(2ⁿ)</td></arr.count>	arrays O(n) Int, _ high: Int is of pairs	O(n log n)	O(n²)	O(2 ⁿ)
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mic) "def",="" "ghi"]="" \$0="" \$1="" (worst="" +="" -="" 0(n²)="" 0arr.count="" 1)="" 1,="" 10="" <="" [int])="" [int],="" \(\)="" \(arr[i]),="" _="" a="" against="" all="" already="" arr:="" arr[i]="" arr[j]="=" array="" built-in="" cas="" case)="" cases:="" combination="" common="" each="" efficient="" element="" else="" every="" fibonacci="" fibonacci(_="" findallpairs(_="" finding="" for="" func="" general="" general-purpose="" guarantee="" guaranteed="" guard="" high="" high)="" i="" if="" implementation="" in="" inout="" int)="" is="" items='["abc",' let="" low="" low,="" low:="" merge(left,="" n)="" n:="" naive="" o(1)="" o(log="" of="" other="" pivot="" possible="" print("pair:="" quick="" quicksort(&arr,="" quicksort(_="" recursive="" return="" right="mergeSort(Array(arr[midreturn" right))="" selection="" some="" sort="" sorted="" sorting="" stability="" stable="" unfortunate="" use="" when="" with="" worst="" {="" }=""> Int {</mic)>	arrays O(n) Int, _ high: Int is of pairs	O(n log n)	O(n²)	O(2 ⁿ)
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[<mic "def",="" "ghi"]="" \$0="" \$1="" (worst="" +="" -="" 0(n²)="" 1)="" 1,="" 10="" <="" [int],="" \(\)="" \(arr[i]),="" _="" all="" already="" arr:="" arr[i]="" arr[j]="=" array="" built-in="" cas="" case)="" cases:="" combination="" common="" efficient="" else="" fibonacci="" fibonacci(_="" finding="" for="" func="" general="" general-purpose="" guarantee="" guaranteed="" guard="" high="" high)="" i+1<arr.count="" if="" implementation="" in="" inout="" int)="" is="" items='["abc",' j="" let="" low="" low,="" low:="" merge(left,="" n)="" n:="" naive="" o(1)="" o(log="" of="" pivot="" possible="" print("pair:="" quick="" quicksort(&arr,="" quicksort(_="" recursive="" return="" right="mergeSort(Array(arr[midreturn" right))="" selection="" sort="" sorted="" sorting="" stability="" stable="" unfortunate="" use="" when="" with="" worst="" {="" }=""> Int { // Each call spawns two more calls if n <= 1 { return n } return fibonacci(n - 1) + fibona</mic>	arrays Stability matters O(n) O(n) Int, _ high: Int O(n)	O(n log n)	O(n²)	O(2 ⁿ)
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[\mid let right = mergeSort(Array(arr[midreturn merge(left, right)) } Common Use Cases: Guaranteed efficient sorting of: Stable Sort Built-in sorting with stability guarantee let items = ["abc", "def", "ghi"] let sorted = items.sorted { \$0 < \$1 } Common Use Cases: General-purpose sorting when some sorting each element against every other func findAllPairs(_ arr: [Int]) { for i in 0 rount findAllPairs(_ arr: [Int]) { for j in i+1 rount findAllPairs(_ arr: [Int]) { for j in i+1 Processing each element against every other func findAllPairs(_ arr: [Int]) { for j in i+1 rount findAllPairs(_ arr: [Int]) { for j in i+1 print("Pair: \((arr[i]), \(\) } } } Common Use Cases: Finding all possible combination Quick Sort (Worst Case) When pivot selection is unfortunate func quickSort(_ arr: inout [Int], _ low: // 0(n²) when array is already sorted // or reverse sorted guard low < high else { return } let pivot = partition(&arr, low, high quickSort(&arr, pivot + 1, high) } Common Use Cases: General sorting when worst case O(1) O(log n) Recursive Fibonacci Naive recursive implementation func fibonacci(_ n: Int) -> Int { // Each call spawns two more calls if n <= 1 { return n } return fibonacci(n - 1) + fibonacci(n	arrays Stability matters O(n) (arr[j])") In a high:	O(n log n)	O(n²)	O(2 ⁿ)
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[\mice let right = mergeSort(Array(arr[midreturn merge(left, right)) } Common Use Cases: Guaranteed efficient sorting of a stable Sort Built-in sorting with stability guarantee let items = ["abc", "def", "ghi"] let sorted = items.sorted { \$0 < \$1 } Common Use Cases: General-purpose sorting when a state of the sorted in a farricity of the sorted in a farricit	arrays Stability matters O(n) (arr[j])") In a high:	O(n log n)	O(n²)	O(2 ⁿ)
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[\mic let right = mergeSort(Array(arr[mid\mic return merge(left, right)) } Common Use Cases: Guaranteed efficient sorting of: Stable Sort Built-in sorting with stability guarantee let items = ["abc", "def", "ghi"] let sorted = items.sorted { \$0 < \$1 } Common Use Cases: General-purpose sorting when some sorting when some sorting with stability guarantee let items = ["abc", "def", "ghi"] let sorted = items.sorted { \$0 < \$1 } Common Use Cases: General-purpose sorting when some sorting when sorting all possible combination Processing each element against every other func findAllPairs(_ arr: [Int]) { for j in i+1varr.count { if arr[i] + arr[j] == 10 { print("Pair: \(arr[i]), \(\) } } } } Common Use Cases: Finding all possible combination Quick Sort (Worst Case) When pivot selection is unfortunate func quickSort(\(\) arr: inout [Int], _ low:	arrays stability matters O(n) O(n) Int, _ high: Int O(n) or - 2) roduction)	O(n log n)	O(n²)	O(2 ⁿ)
Merge Sort Divide and conquer sorting algorithm func mergeSort(_ arr: [Int]) -> [Int] { guard arr.count > 1 else { return arr let mid = arr.count / 2 let left = mergeSort(Array(arr[\mic let right = mergeSort(Array(arr[mid return merge(left, right)) } Common Use Cases: Guaranteed efficient sorting of: Stable Sort Built-in sorting with stability guarantee let items = ["abc", "def", "ghi"] let sorted = items.sorted { \$0 < \$1 } Common Use Cases: General-purpose sorting when : O(1) O(log n) Nested Iteration Processing each element against every other func findAllPairs(_ arr: [Int]) { for i in 0 <arr.count (worst="" +="" 10="" [int],="" \(arr[i]),="" \\="" _="" all="" arr:="" arr[i]="" arr[j]="=" case)="" cases:="" combination="" common="" finding="" func="" if="" inout="" is="" low:<="" pivot="" possible="" print("pair:="" quick="" quicksort(_="" selection="" sort="" td="" unfortunate="" use="" when="" {="" }=""><td>arrays stability matters O(n) O(n) Int, _ high: Int O(n) or - 2) roduction)</td><td>O(n log n)</td><td>O(n²)</td><td>O(2ⁿ)</td></arr.count>	arrays stability matters O(n) O(n) Int, _ high: Int O(n) or - 2) roduction)	O(n log n)	O(n²)	O(2 ⁿ)

[first] + \$0

}

}

let subsetsWithFirst = subsetsWithoutFirst.map {

return subsetsWithoutFirst + subsetsWithFirst

Common Use Cases: When all combinations must be considered