

<pre> SelectionSort(arr, n): // arr: 배열, n: 배열의 크기 for i from 0 to n-2:     // 가장 작은 원소의 인덱스를 찾음     minIndex = i     for j from i+1 to n-1:         if arr[j] &lt; arr[minIndex]:             minIndex = j      // 찾은 가장 작은 원소와 현재 위치(i)의 원소를 교환     swap(arr[i], arr[minIndex]) </pre>	<pre> InsertionSort(arr, n): // arr: 배열, n: 배열의 크기 for i from 1 to n-1:     key = arr[i] // 현재 삽입하려는 원소     j = i - 1      // key보다 큰 원소들을 오른쪽으로 한 칸씩 이동     while j &gt;= 0 and arr[j] &gt; key:         arr[j+1] = arr[j]         j = j - 1      // key를 올바른 위치에 삽입     arr[j+1] = key </pre>
<pre> Partition(arr, low, high): // arr: 배열, low: 시작 인덱스, high: 끝 인덱스 pivot = arr[high] // 피벗을 마지막 원소로 선택 i = low - 1 // 피벗보다 작은 원소들의 마지막 위치  for j from low to high-1:     // 현재 원소가 피벗보다 작거나 같으면     if arr[j] &lt;= pivot:         i = i + 1         swap(arr[i], arr[j])  // 피벗을 올바른 위치로 이동 swap(arr[i+1], arr[high]) return (i + 1) // 피벗의 최종 위치 반환 </pre>	<pre> QuickSort(arr, low, high): if low &lt; high:     // pi는 파티션 후 피벗의 인덱스     pi = Partition(arr, low, high)      // 피벗을 기준으로 좌우 부분 배열을 각각 재귀적으로 정렬     QuickSort(arr, low, pi - 1)     QuickSort(arr, pi + 1, high) </pre>
<pre> Merge(arr, left, mid, right): // 부분 배열의 크기 계산 n1 = mid - left + 1 n2 = right - mid  // 임시 배열 L과 R 생성 L = new array[n1] R = new array[n2]  // 데이터 복사 for i from 0 to n1-1:     L[i] = arr[left + i] for j from 0 to n2-1:     R[j] = arr[mid + 1 + j]  // 병합 과정 i = 0 // L 배열의 인덱스 j = 0 // R 배열의 인덱스 k = left // 원본 arr 배열의 인덱스  while i &lt; n1 and j &lt; n2:     if L[i] &lt;= R[j]:         arr[k] = L[i]         i = i + 1     else:         arr[k] = R[j]         j = j + 1     k = k + 1  // 남은 원소들 복사 while i &lt; n1:     arr[k] = L[i]     i = i + 1     k = k + 1 while j &lt; n2:     arr[k] = R[j]     j = j + 1     k = k + 1 </pre>	<pre> MergeSort(arr, left, right): if left &lt; right:     // 오버플로우를 방지하는 중간값 계산     mid = left + (right - left) / 2      // 좌우 부분 배열 재귀 호출     MergeSort(arr, left, mid)     MergeSort(arr, mid + 1, right)  // 정렬된 부분 배열 병합 Merge(arr, left, mid, right) </pre>

<p><b>힙정렬 (Heapify)</b></p> <pre> Heapify(arr, n, i): // arr: 배열, n: 힙의 크기, i: heapify를 수행할 노드 인덱스 largest = i      // 부모 노드 (일단 자신이 가장 크다고 가정) left = 2*i + 1   // 왼쪽 자식 right = 2*i + 2  // 오른쪽 자식  // 왼쪽 자식이 힙 크기 내에 있고, 부모(현재 largest)보다 크면 if left &lt; n and arr[left] &gt; arr[largest]:     largest = left  // 오른쪽 자식이 힙 크기 내에 있고, (현재까지) 가장 큰 값보다 크면 if right &lt; n and arr[right] &gt; arr[largest]:     largest = right  // largest가 i가 아니라는 것은 자식 노드가 더 크다는 의미 (힙 속성 위반) if largest != i:     swap(arr[i], arr[largest]) // swap으로 인해 자식 노드(largest)의 서브트리가 깨졌을 수 있으므로 // 그 위치(largest)에서 재귀적으로 heapify 수행 Heapify(arr, n, largest) </pre>	<p><b>힙정렬</b></p> <pre> HeapSort(arr, n): // 1. 최대 힙 구성 (Build Max Heap) // (n/2 - 1)은 마지막 부모 노드의 인덱스 for i from (n / 2 - 1) down to 0:     Heapify(arr, n, i)  // 2. 힙 정렬 (Extract elements) for i from n-1 down to 1:     // 현재 힙의 루트(최대값)를 배열의 맨 뒤로 이동     swap(arr[0], arr[i])      // 힙 크기를 1 줄이고 (arr[0...i-1]), 루트에 대해 heapify 수행     Heapify(arr, i, 0) </pre>
<p><b>계수정렬</b></p> <pre> CountingSort(arr, n, k): // arr: 입력 배열, n: 배열 크기, k: 최대값 (0부터 k까지 범위)  // 1. 빈도수 계산 count = new array[k+1] initialized to 0 for i from 0 to n-1:     count[arr[i]] = count[arr[i]] + 1  // 2. 누적합 계산 (각 원소의 '끝' 위치) for i from 1 to k:     count[i] = count[i] + count[i-1]  // 3. 출력 배열 구성 (안정 정렬을 위해 역순 진행) output = new array[n] for i from n-1 down to 0:     value = arr[i]     position = count[value] - 1 // 0-indexed 위치     output[position] = value     count[value] = count[value] - 1 // 다음 동점자를 위해 위치 1 감소  // 4. 원본 배열에 복사 for i from 0 to n-1:     arr[i] = output[i] </pre>	<p><b>기수정렬</b></p> <pre> // 기수 정렬을 위한 보조 함수 (안정 정렬인 Counting Sort 사용) CountSortByDigit(arr, n, exp): // (exp는 현재 정렬 중인 자릿수, 예: 1, 10, 100...) output = new array[n] count = new array[10] initialized to 0 // 0~9까지 10개  // 1. 현재 자릿수의 빈도수 계산 for i from 0 to n-1:     digit = (arr[i] / exp) % 10     count[digit] = count[digit] + 1  // 2. 누적합 계산 for i from 1 to 9:     count[i] = count[i] + count[i-1]  // 3. 출력 배열 구성 (역순 -&gt; 안정성 보장) for i from n-1 down to 0:     digit = (arr[i] / exp) % 10     output[count[digit] - 1] = arr[i]     count[digit] = count[digit] - 1  // 4. 원본 배열에 복사 for i from 0 to n-1:     arr[i] = output[i]  // 기수 정렬 메인 함수 RadixSort(arr, n): // 1. 최대값 찾기 (자릿수 파악을 위해) max_val = findMax(arr, n)  // 2. 1의 자리부터 LSD(최하위 유효 숫자) 방식으로 정렬 exp = 1 while (max_val / exp) &gt; 0:     CountSortByDigit(arr, n, exp)     exp = exp * 10 </pre>

<p><b>레드 블랙 트리</b></p> <pre> // 노드 구조 Node:     key     color (RED or BLACK)     left, right, parent (NIL 센티넬을 가리킬 수 있음)  // 좌회전 LeftRotate(Tree T, Node x):     y = x.right // y는 x의 오른쪽 자식     x.right = y.left // y의 왼쪽 서브트리를 x의 오른쪽 서브트리로     if y.left != T.nil:         y.left.parent = x // y의 왼쪽 자식의 부모를 x로      y.parent = x.parent // y의 부모를 x의 부모로     if x.parent == T.nil: // x가 루트였다면         T.root = y // y가 새 루트     else if x == x.parent.left: // x가 왼쪽 자식이었다면         x.parent.left = y // 부모의 왼쪽 자식을 y로     else: // x가 오른쪽 자식이었다면         x.parent.right = y // 부모의 오른쪽 자식을 y로      y.left = x // x를 y의 왼쪽 자식으로     x.parent = y // x의 부모를 y로  // 삽입 후 재조정 InsertFixup(Tree T, Node z):     // (z는 새로 삽입된 RED 노드)     while z.parent.color == RED: // 속성 4 위반 (RED의 자식은 RED 불가)         // A. 부모가 할아버지의 왼쪽 자식인 경우         if z.parent == z.parent.parent.left:             uncle = z.parent.parent.right // 삼촌              // Case 1: 삼촌이 RED             if uncle.color == RED:                 z.parent.color = BLACK                 uncle.color = BLACK                 z.parent.parent.color = RED                 z = z.parent.parent // 할아버지 노드에서 다시 검사 시작              else: // (uncle.color == BLACK 또는 NIL)                 // Case 2: 삼촌이 BLACK이고, z가 오른쪽 자식 (Triangle)                 if z == z.parent.right:                     z = z.parent                     LeftRotate(T, z)                  // Case 3: 삼촌이 BLACK이고, z가 왼쪽 자식 (Line)                 // (Case 2가 Case 3로 전환됨)                 z.parent.color = BLACK                 z.parent.parent.color = RED                 RightRotate(T, z.parent)          // B. 부모가 할아버지의 오른쪽 자식인 경우 (A와 대칭)         else:             uncle = z.parent.parent.left             // (Case 1, 2, 3의 대칭 로직)             ...      T.root.color = BLACK // 속성 2 (루트는 BLACK) 보장 </pre>	<p><b>BST</b></p> <pre> // 노드 구조 Node:     key     left (child)     right (child)  // 삽입 (재귀) Insert(node, key):     // 1. 트리가 비었거나 리프 노드에 도달     if node == null:         return createNode(key)      // 2. 재귀적으로 삽입 위치 탐색     if key &lt; node.key:         node.left = Insert(node.left, key)     else if key &gt; node.key: // (중복을 허용하지 않거나, 같을 때 오른쪽으로 보낼 수 있음)         node.right = Insert(node.right, key)      // 3. (수정된) 서브트리의 루트 반환     // (AVL/RBT가 아니므로 node는 항상 자신)     return node  // 삭제 DeleteNode(root, key):     if root == null:         return root // 기저 사례: 빈 트리      // 1. 삭제할 노드 탐색     if key &lt; root.key:         root.left = DeleteNode(root.left, key)     else if key &gt; root.key:         root.right = DeleteNode(root.right, key)      // 2. 노드를 찾았을 때 (key == root.key)     else:         // Case 1: 자식이 없거나 하나         if root.left == null:             temp = root.right             free(root) // 메모리 해제             return temp // 오른쪽 자식(또는 null)을 부모에게 연결         else if root.right == null:             temp = root.left             free(root) // 메모리 해제             return temp // 왼쪽 자식(또는 null)을 부모에게 연결          // Case 2: 자식이 둘         // 오른쪽 서브트리에서 가장 작은 값(후계자, successor)을 찾음         temp = findMin(root.right)          // 후계자의 키를 현재 노드로 복사         root.key = temp.key          // (복사된) 후계자 노드를 오른쪽 서브트리에서 삭제         root.right = DeleteNode(root.right, temp.key)      return root // (수정된) 서브트리의 루트 반환 </pre>

<pre> <b>B 트리</b> // t = 최소 차수 (degree)  // 노드 구조 Node:     n (현재 키의 개수)     keys[2t-1] (키 배열)     children[2t] (자식 포인터 배열)     isLeaf (true/false)  // 검색 Search(node, key):     i = 0     // 1. 현재 노드에서 위치 찾기 (key보다 크거나 같은 첫 키를 찾을 때까지)     while i &lt; node.n and key &gt; node.keys[i]:         i = i + 1      // 2. 키를 찾을 때까지     if i &lt; node.n and key == node.keys[i]:         return (node, i)      // 3. 리프 노드인데 못 찾을 경우 (기저 사례)     if node.isLeaf:         return null      // 4. 자식 노드로 재귀 검색 (key는 children[i] 서브트리에 있어야 함)     return Search(node.children[i], key)  // 팍 찬 자식 노드 분할 // (parent의 i번째 자식인 fullChild를 분할) SplitChild(parent, i, fullChild):     // 1. 새 노드(newChild) 생성 (fullChild의 오른쪽 절반)     newChild = createNode()     newChild.isLeaf = fullChild.isLeaf     newChild.n = t - 1 // (키 개수)      // 2. fullChild의 키/자식을 newChild로 복사     // (키: t부터 2t-2까지 t-1개를 복사)     for j from 0 to t-2:         newChild.keys[j] = fullChild.keys[j + t]     // (자식: t부터 2t-1까지 t개를 복사)     if not fullChild.isLeaf:         for j from 0 to t-1:             newChild.children[j] = fullChild.children[j + t]      // 3. fullChild의 키 개수 업데이트 (중간 키 제외)     fullChild.n = t - 1      // 4. parent에 newChild를 위한 공간 확보 (자식 포인터)     // (i+1 뒤의 자식들을 오른쪽으로 한 칸씩 뺌)     for j from parent.n down to i+1:         parent.children[j+1] = parent.children[j]     parent.children[i+1] = newChild // newChild 연결      // 4-1. parent에 newChild를 위한 공간 확보 (키)     // (i 뒤의 키들을 오른쪽으로 한 칸씩 뺌)     for j from parent.n-1 down to i:         parent.keys[j+1] = parent.keys[j]      // 5. 중간 키(fullChild.keys[t-1])를 parent로 올림     parent.keys[i] = fullChild.keys[t-1]     parent.n = parent.n + 1 </pre>	<pre> <b>KD 트리</b> // k: 차원 수  // 구축 BuildKDTree(points, depth):     n = points.length     if n == 0:         return null      // 1. 현재 깊이에서 분할 축(axis) 결정     axis = depth % k      // 2. points를 axis 기준으로 정렬     // (O(n log n) -&gt; 비효율적. O(n)인 'nth_element' 또는 'median-of-medians' 사용 권장)     sort(points, by_axis=axis)      // 3. 중앙값(median)을 현재 노드로 선택     median_index = n / 2     median_point = points[median_index]      node = createNode(median_point, axis) // (축 정보도 저장)      // 4. 재귀적으로 좌우 서브트리 구축     // (중앙값 제외하고 좌/우 분할)     node.left = BuildKDTree(points[0 ... median_index-1], depth + 1)     node.right = BuildKDTree(points[median_index+1 ... n-1], depth + 1)      return node  // 범위 검색 RangeSearch(node, range, depth):     if node == null:         return      // 1. 현재 노드가 범위 내에 있는지 확인     if node.point is inside range:         add node.point to results      // 2. 현재 축(axis) 가져오기 (node.axis 또는 depth % k)     axis = node.axis      // 3. 왼쪽 서브트리 탐색 결정 (분할선이 범위의 오른쪽보다 왼쪽에 있음)     // (즉, "검색 범위"가 분할선의 "왼쪽" 영역과 겹침)     if range.min[axis] &lt;= node.point[axis]:         RangeSearch(node.left, range, depth + 1)      // 4. 오른쪽 서브트리 탐색 결정 (분할선이 범위의 왼쪽보다 오른쪽에 있음)     // (즉, "검색 범위"가 분할선의 "오른쪽" 영역과 겹침)     if range.max[axis] &gt;= node.point[axis]:         RangeSearch(node.right, range, depth + 1) </pre>

<p><b>체이닝</b></p> <pre>// HashTable 구조 HashTable:     size (테이블 크기, 버킷의 수)     table (연결 리스트의 배열)     count (총 항목 수)  // 삽입 Insert(ht, key, value):     // (로드 팩터 체크 및 재해싱)     // if (ht.count / ht.size) &gt; THRESHOLD:     Rehashing()      index = hash(key) % ht.size     // (음수 인덱스 방지: index = (index + ht.size) % ht.size)      // 1. 해당 인덱스의 체인(연결 리스트) 순회     // (키 중복 체크)     node = ht.table[index]     while node != null:         if node.key == key:             node.value = value // 키 중복 시 값 업데이트             return         node = node.next      // 2. 새 노드를 체인의 맨 앞에 삽입 (중복 없음)     newNode = createNode(key, value)     newNode.next = ht.table[index] // 기존 리스트를 새     // 노드의 next로     ht.table[index] = newNode // 새 노드를 리스트의     // head로     ht.count = ht.count + 1</pre>	<p><b>개방 주소법</b></p> <pre>// 테이블 항목 상태: EMPTY, OCCUPIED, DELETED  // 삽입 Insert(ht, key, value):     // (재해싱 체크: ht.count / ht.size &gt; THRESHOLD     // (e.g., 0.5))     // if (ht.count / ht.size) &gt; 0.5: Rehashing()      for i from 0 to ht.size-1:         index = (hash(key) + i) % ht.size // 선형 탐사         // (i=0, 1, 2...)          // 1. 빈 슬롯(EMPTY 또는 DELETED)을 찾으면 삽입         if ht.table[index].state == EMPTY or         ht.table[index].state == DELETED:             ht.table[index] = {key, value, OCCUPIED}             ht.count = ht.count + 1 // 항목 개수 증가             return          // 2. 키가 중복되면 값 업데이트         if ht.table[index].state == OCCUPIED and         ht.table[index].key == key:             ht.table[index].value = value             return      // (테이블이 꽉 참 - 재해싱 필요)  // 검색 Search(ht, key):     for i from 0 to ht.size-1:         index = (hash(key) + i) % ht.size          // 1. EMPTY를 만나면 탐사 종료 (키 없음)         if ht.table[index].state == EMPTY:             return null          // 2. DELETED를 만나면 탐사 계속         if ht.table[index].state == DELETED:             continue          // 3. 키를 찾으면 값 반환 (OCCUPIED 상태)         if ht.table[index].key == key: // (자동으로         // OCCUPIED 상태임)             return ht.table[index].value      // 테이블을 다 돌아도(EMPTY 못 만나고) 못 찾음     return null</pre>
<pre>// k번째로 작은 원소를 찾는 함수 (k는 0-indexed) QuickSelect(arr, left, right, k):     // 원소가 하나만 남으면 반환 (기저 사례)     if left == right:         return arr[left]      // Lomuto 파티션을 사용     pivotIndex = Partition(arr, left, right)      if k == pivotIndex:         // k번째 원소를 찾음 (피벗이 k번째였음)         return arr[k]     else if k &lt; pivotIndex:         // k가 피벗보다 왼쪽에 있음         // (왼쪽 부분 배열만 재귀 탐색)         return QuickSelect(arr, left, pivotIndex - 1,         k)     else: // k &gt; pivotIndex         // k가 피벗보다 오른쪽에 있음         // (오른쪽 부분 배열만 재귀 탐색)         return QuickSelect(arr, pivotIndex + 1,         right, k)</pre>	

--	--

<pre> // parent[]: 각 원소의 부모를 저장 // rank[]: 트리의 높이(또는 랭크)를 저장  // 초기화 MakeSet(n):     parent = new array[n]     rank = new array[n]     for i from 0 to n-1:         parent[i] = i // 자기 자신을 부모(루트)로         rank[i] = 0 // 높이 0  // 찾기 (Find) - Path Compression 적용 Find(x):     if parent[x] == x: // 자신이 루트이면         return x     else:         // 재귀적으로 루트를 찾고,         // 그 결과를 나의 부모(parent[x])로 바로 연결 (경로 압축)         parent[x] = Find(parent[x])         return parent[x] // (압축된) 부모(즉, 루트) 반환  // 합치기 (Union) - Union by Rank 적용 Union(x, y):     rootX = Find(x)     rootY = Find(y)      if rootX != rootY: // 두 원소가 다른 집합에 속할 때만 합침         // 1. 랭크 비교         if rank[rootX] &lt; rank[rootY]:             parent[rootX] = rootY // rootX를 rootY 아래에 붙임             (rootY가 루트)         else if rank[rootX] &gt; rank[rootY]:             parent[rootY] = rootX // rootY를 rootX 아래에 붙임             (rootX가 루트)          // 2. 랭크가 같을 때         else:             parent[rootY] = rootX // 한쪽을 붙이고             rank[rootX] = rank[rootX] + 1 // 합쳐진 트리의 랭크             (높이) 1 증가 </pre>	