Data Structures

Merge, quick and heap sort

The notes for the lesson “Merge, quick and heap sort”: chapters 6.3 to 6.4, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

2 chapters | Chapters 6.3 to 6.4

About these notes

These are the notes for the University free n easy video lesson **Merge, quick and heap sort** on Data Structures: chapters 6.3 to 6.4, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

6.3 Merge sort and quick sort

Both sorts use divide and conquer: split the array, sort the parts recursively, combine. Merge sort does the trivial split (cut in the middle) and the real work in the combine step (merging two sorted runs). Quick sort does the real work in the split (partitioning around a pivot so that the combine step is nothing). Both run in O(n log n) on average, which is as good as any comparison sort can be (chapter 6.6), and both are asked as "trace with a divide-and-conquer diagram".

You need: recursion and solving recurrences (Unit 1), and the simple sorts of chapter 6.2 for comparison.

Merge sort

1. If the range has one element, it is sorted: return.
2. Find mid = (low + high)/2; sort a[low..mid] and a[mid+1..high] recursively.
3. **Merge**: walk two pointers i and j along the two sorted halves, copy the smaller of a[i] and a[j] into a temporary array, advance that pointer; when one half is exhausted, copy the rest of the other; copy the temporary array back.

Merging two runs whose lengths add to n takes at most n − 1 comparisons and exactly n copies. The recursion has $\lceil \log_2 n \rceil$ levels, and each level merges n elements in total, so T(n) = 2T(n/2) +

cn, which gives Θ(n log n) in the best, average and worst case. The price is O(n) extra space for the temporary array, and the merge is stable because when a[i] equals a[j] the left one is copied first. Merge sort is the natural sort for linked lists (no extra array needed, just pointer rewiring) and for data on disk (chapter 6.6).

```
void merge(int a[], int lo, int mid, int hi, int tmp[]) {       /* C */
    int i = lo, j = mid + 1, k = lo;
    while (i <= mid && j <= hi) tmp[k++] = (a[i] <= a[j]) ? a[i++] : a[j++];
    while (i <= mid) tmp[k++] = a[i++];
    while (j <= hi)  tmp[k++] = a[j++];
    for (k = lo; k <= hi; k++) a[k] = tmp[k];
}
void merge_sort(int a[], int lo, int hi, int tmp[]) {
    if (lo >= hi) return;
    int mid = lo + (hi - lo) / 2;
    merge_sort(a, lo, mid, tmp); merge_sort(a, mid + 1, hi, tmp);
    merge(a, lo, mid, hi, tmp);
}
```

Quick sort

Choose a **pivot** element. **Partition** the array so that everything ≤ pivot is to its left and everything > pivot to its right; the pivot is now in its final sorted position. Sort the two sides recursively; nothing needs combining. The partition below is Lomuto's, the one in most Indian textbooks and the easiest to trace; the pivot is the last element.

1. pivot = a[high]; i = low − 1 (i marks the end of the "≤ pivot" zone).
2. For j from low to high − 1: if a[j] ≤ pivot, increment i and swap a[i] with a[j].
3. Swap a[i+1] with a[high]. The pivot now sits at p = i + 1; return p.
4. Recursively quick-sort a[low..p−1] and a[p+1..high].

Hoare's original partition (many exam papers use it) keeps the pivot at the first position and moves two indices inward, swapping out-of-place pairs, until they cross. Either way a partition of n elements costs about n comparisons (exactly n − 1 for Lomuto's) and no extra array.

If the pivot splits the range evenly every time, $T(n) = 2T(n/2) + n$, $\Theta(n \log n)$. If the pivot is always the smallest or largest, one side is empty: $T(n) = T(n-1) + n$, which sums to $n(n+1)/2$, $\Theta(n^2)$. That happens with a last-element pivot on data that is already sorted. The average over random inputs is about $1.39\, n \log_2 n$ comparisons, still $\Theta(n \log n)$, and the constant is small because partition touches memory sequentially; that is why quick sort is usually the fastest in-memory sort. Choosing the pivot as the median of a[low], a[mid], a[high], or at random, makes the $\Theta(n^2)$

case practically impossible. Quick sort is not stable, and uses O(log n) stack space on average, O(n) at worst.

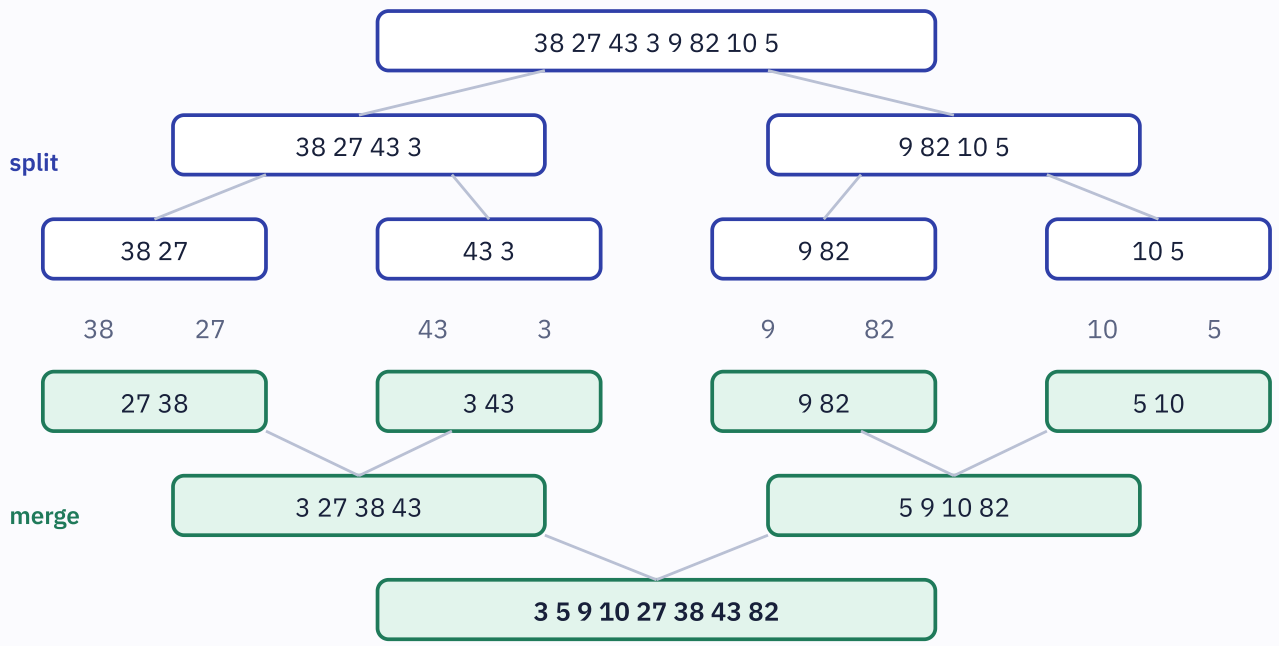


Figure 6.3.1 Merge sort on eight numbers. The top half only splits (no comparisons); the bottom half merges pairs of sorted runs: at most 4, 6 and 7 comparisons on the three levels. Three levels of merging for n = 8 is $\log_2 8$; each level copies all eight elements once, hence n log n.

WORKED EXAMPLE

Question (a). Sort {38, 27, 43, 3, 9, 82, 10, 5} by merge sort. Show each merge and count the comparisons.

Solution. Splitting gives singletons as in Figure 6.3.1. Merges, level by level:

MERGE	INPUTS	COMPARISONS MADE	OUTPUT
1–4	[38][27], [43][3], [9][82], [10][5]	1 each = 4	[27 38], [3 43], [9 82], [5 10]
5	[27 38] and [3 43]	3 vs 27, 27 vs 43, 38 vs 43 = 3	[3 27 38 43]
6	[9 82] and [5 10]	9 vs 5, 9 vs 10, 82 vs 10 = 3	[5 9 10 82]
7	[3 27 38 43] and [5 9 10 82]	3v5, 27v5, 27v9, 27v10, 27v82, 38v82, 43v82 = 7	[3 5 9 10 27 38 43 82]

Total **17 comparisons** and 8 × 3 = 24 element copies. Bubble sort on the same input would make 28 comparisons.

Question (b). Partition {29, 10, 14, 37, 13} with Lomuto's method (pivot = last) and continue quick sort to the end.

Solution. pivot = 13, i = −1.

J	A[J]	A[J] ≤ 13?	I	ACTION	ARRAY
0	29	no	−1	—	29 10 14 37 13
1	10	yes	0	swap a[0], a[1]	10 29 14 37 13
2	14	no	0	—	10 29 14 37 13
3	37	no	0	—	10 29 14 37 13
end				swap a[i+1]=a[1] with a[4]	10 **13** 14 37 29

Pivot 13 is final at index 1 (4 comparisons). Left part [10] is one element. Right part [14, 37, 29], pivot = 29: j=2, 14 ≤ 29 → i = 2, swap with itself; j=3, 37 > 29; swap a[3] with a[4] → 14 **29** 37 (2 comparisons). Both sides are single elements. Result **10 13 14 29 37** with 6 comparisons in total. Had the input been sorted, 10 13 14 29 37, the pivot 37 would leave an empty right side each time: 4 + 3 + 2 + 1 = 10 comparisons, the $\Theta(n^2)$ pattern.

EXAM TRAPS

- Quick sort worst case is $\Theta(n^2)$, not $\Theta(n \log n)$. The examiner's favourite question: "when does it happen?" Answer: pivot always extreme, e.g. first or last element on sorted or reverse-sorted input.
- Merge sort is $\Theta(n \log n)$ in *every* case; it is the one to name when the question says "guaranteed".
- In Lomuto's partition the pivot ends at i + 1, not at i; forgetting the final swap is the usual trace error.
- Merge sort needs O(n) extra space for arrays, but O(1) extra for linked lists. Quick sort is in place apart from the recursion stack.
- Do not say "quick sort is stable". It is not; merge sort is.

CHECK YOURSELF

– REMEMBER **Write the recurrence for merge sort and its solution.** 2 marks

$T(n) = 2T(n/2) + cn$, $T(1) = c$; solution $\Theta(n \log n)$ (Master theorem case 2, or a recursion tree with $\log_2 n$ levels of cost cn).

– REMEMBER **Which step does the real work in merge sort, and which in quick sort?** 2 marks

Merge sort: the combine (merge) step. Quick sort: the divide (partition) step; combine is empty.

– UNDERSTAND **Why does a random pivot make quick sort's worst case unlikely rather than impossible?** 3 marks

A bad sequence of pivots still exists for every input, but no fixed input can force it; the probability that all n pivots are extreme is about $2^n/n!$, negligible for $n > 20$. The expected cost becomes $\Theta(n \log n)$ for every input.

– APPLY **Merge [2, 7, 9, 15] with [3, 4, 11]. List the comparisons.** 3 marks

2v3→2, 7v3→3, 7v4→4, 7v11→7, 9v11→9, 15v11→11, then copy 15. Output 2 3 4 7 9 11 15; 6 comparisons = 7 − 1.

– APPLY **Partition {7, 2, 9, 4, 6} with pivot = last (Lomuto). Show the result and the pivot's index.** 4 marks

pivot 6, i=−1. j=0: 7 no. j=1: 2 yes, i=0, swap → 2 7 9 4 6. j=2: 9 no. j=3: 4 yes, i=1, swap → 2 4 9 7 6. Final swap a[2]↔a[4] → 2 4 6 7 9; pivot at index 2.

– ANALYZE **For sorting a 2 GB file of fixed-size records on a 1 GB machine, which of the two is suitable and why?** 4 marks

Merge sort: it reads and writes runs sequentially, so it works when the data does not fit in memory (external merge sort, chapter 6.6). Quick sort needs random access to the whole array for partitioning and would thrash the disk.

– ANALYZE **Quick sort does about 1.39 n $\log_2 n$ comparisons on average, more than merge sort's n $\log_2 n$, yet is usually faster. Why?** 4 marks

Partition works in place with sequential memory access and no copying, so each comparison costs less; merge sort copies every element at every level into a temporary array, roughly doubling memory traffic. Constant factors and cache behaviour, not the growth rate, decide the winner.

BEYOND THE SYLLABUS

Not for the exam. Real libraries blend these ideas: C++ `std::sort` is introsort (quick sort that switches to heap sort if recursion gets too deep, and to insertion sort for tiny ranges); Java and Python sort objects with Timsort, a merge sort that detects already-sorted runs; Java sorts primitives with a dual-pivot quick sort; Rust's standard library recently moved to new hybrids (driftsort for stable, ipnsort for unstable sorting). GPUs and databases sort in parallel with merge-based networks. Search for "introsort", "Timsort runs" and "pdqsort pattern-defeating" to see how the $\Theta(n^2)$ case is engineered away.

6.4 Heap sort

Heap sort turns the array itself into a max-heap (a complete binary tree stored in the array where every parent is at least as large as its children), then repeatedly swaps the root, which is the maximum, to the end and shrinks the heap by one. It is the only simple sort that is O(n log n) in the worst case *and* in place, which is why it is the fallback inside introsort. The whole algorithm is two ideas you already have: the array-as-tree index rule and the sift-down (heapify) operation.

You need: binary heaps and the array representation of a complete binary tree (Unit 4, Binary heaps and priority queues). This chapter only recalls the index rule.

The array is the tree

With 0-based indices, the children of node i are 2i + 1 and 2i + 2, and its parent is ⌊(i − 1)/2⌋. Leaves are the indices from ⌊n/2⌋ to n − 1; they are already one-node heaps. The **max-heap property**: a[i] ≥ a[2i+1] and a[i] ≥ a[2i+2] wherever those children exist. Nothing is said about left versus right child, so a heap is not a search tree.

Heapify (sift down) and build-heap

1. **heapify(i, n)**: let largest = i; if the left child 2i+1 < n and a[2i+1] > a[largest], largest = 2i+1; same test for the right child 2i+2.
2. If largest ≠ i, swap a[i] with a[largest] and heapify(largest, n): the displaced element keeps sinking until both children are smaller or it reaches a leaf.
3. **build-heap**: for i from ⌊n/2⌋ − 1 down to 0, call heapify(i, n). Working bottom-up guarantees that when node i is fixed, both its subtrees are already heaps.

heapify costs O(height of i). Summing over all nodes, most of which sit near the bottom with tiny height, gives build-heap a total of O(n), not O(n log n): there are at most $n/2^{h+1}$ nodes of height h, and $\Sigma\, h/2^h$ converges to 2.

The sort

1. build-heap on a[0..n−1].
2. For end = n − 1 down to 1: swap a[0] (the maximum) with a[end]; then heapify(0, end) so that a[0..end−1] is a heap again.
3. Each swap fixes one more element at the right end; after n − 1 rounds the array is sorted ascending.

```
void heapify(int a[], int n, int i) {                    /* C, sift down */
    int l = 2*i + 1, r = 2*i + 2, big = i;
    if (l < n && a[l] > a[big]) big = l;
    if (r < n && a[r] > a[big]) big = r;
    if (big != i) { int t = a[i]; a[i] = a[big]; a[big] = t; heapify(a, n, big); }
}
void heap_sort(int a[], int n) {
    for (int i = n/2 - 1; i >= 0; i--) heapify(a, n, i);          /* build-heap: O(n) */
    for (int end = n - 1; end > 0; end--) {
        int t = a[0]; a[0] = a[end]; a[end] = t;                    /* max goes to the end
        heapify(a, end, 0);                                         /* O(log n) */
    }
}
```

Cost: O(n) to build, then n – 1 heapify calls on a heap of shrinking size, each O(log n): O(n log n) in the best, average and worst case. Extra space O(1) (the recursion in heapify can be written as a loop). Heap sort is not stable, and in practice it runs slower than quick sort because the parent-child jumps across the array are unfriendly to caches. To sort *descending*, build a min-heap instead.

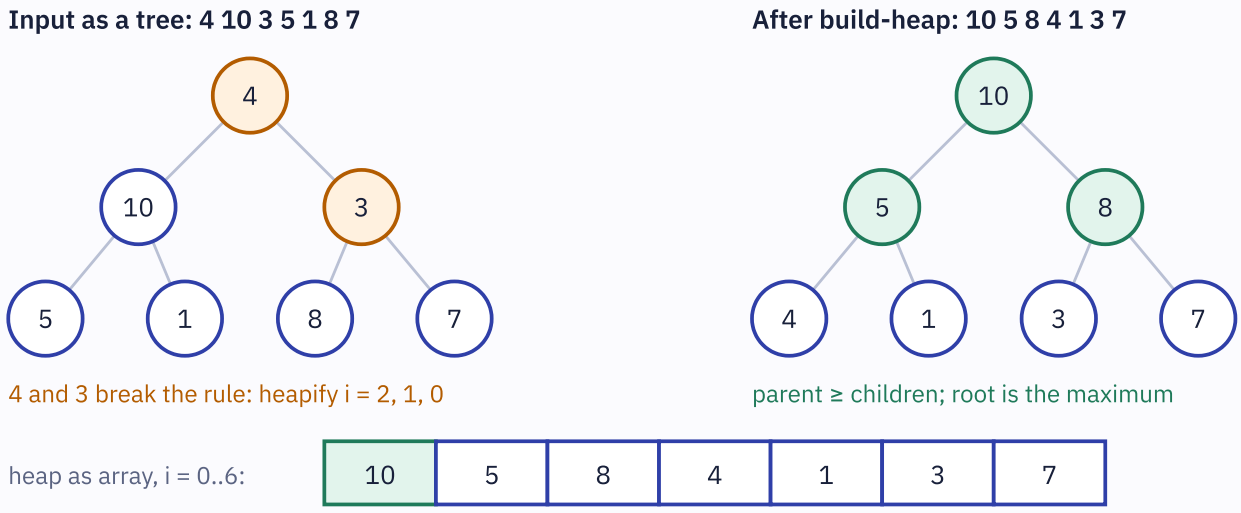


Figure 6.4.1 Build-heap on 4 10 3 5 1 8 7. The tree is only a way of reading the array: node i has children 2i+1 and 2i+2. After build-heap the root holds the maximum 10; the sort then swaps it with the last element and sifts the new root down.

WORKED EXAMPLE

Question. Sort {4, 10, 3, 5, 1, 8, 7} using heap sort. Show the array after build-heap and after each extraction.

Solution. n = 7, so build-heap runs heapify for i = 2, 1, 0.

- i = 2: a[2] = 3, children a[5] = 8, a[6] = 7; largest is 8 → swap: 4 10 **8** 5 1 **3** 7.
- i = 1: a[1] = 10, children 5 and 1 → no change.
- i = 0: a[0] = 4, children 10 and 8; largest 10 → swap: **10 4** 8 5 1 3 7; sift 4 at index 1: children 5 and 1, largest 5 → swap: 10 **5** 8 **4** 1 3 7.

Max-heap: **10 5 8 4 1 3 7**. Now extract n − 1 = 6 times; the part after | is sorted and no longer in the heap.

ROUND	SWAP A[0] WITH A[END]	HEAPIFY(0, END)	ARRAY
1	10 ↔ 7 (end 6)	7 sinks: 7 vs 5, 8 → swap with 8; at index 2, child 3 only → stop	8 5 7 4 1 3 | 10
2	8 ↔ 3 (end 5)	3 vs 5, 7 → swap with 7; at index 2, no children in heap	7 5 3 4 1 | 8 10
3	7 ↔ 1 (end 4)	1 vs 5, 3 → swap with 5; at index 1, child 4 → swap	5 4 3 1 | 7 8 10
4	5 ↔ 1 (end 3)	1 vs 4, 3 → swap with 4; at index 1, no children	4 1 3 | 5 7 8 10
5	4 ↔ 3 (end 2)	3 vs 1 → stays	3 1 | 4 5 7 8 10
6	3 ↔ 1 (end 1)	heap of one element	1 | 3 4 5 7 8 10

Sorted output: **1 3 4 5 7 8 10**. Build-heap used 3 swaps; the six rounds used 6 root swaps and 5 sift-down swaps.

EXAM TRAPS

- Build-heap is O(n), not O(n log n). The examiner asks precisely this; the n log n belongs to the extraction phase.
- Start build-heap at index ⌊n/2⌋ − 1 (0-based) or ⌊n/2⌋ (1-based), going *down* to the root. Starting at the root and going down the array does not produce a heap.
- A max-heap sorts ascending; students who draw a min-heap and then expect ascending output get the order reversed.
- After swapping the root to the end, heapify with the *reduced* size (end), or the sorted tail gets pulled back into the heap.
- Heap sort is not stable and, despite its guaranteed O(n log n), is usually slower than quick sort in practice; do not claim it is "the fastest".

CHECK YOURSELF

– REMEMBER **Give the index of the parent and children of node i in a 0-based heap array.** 2 marks

Children 2i + 1 and 2i + 2; parent ⌊(i − 1)/2⌋.

– REMEMBER **State the time and space complexity of heap sort.** 2 marks

O(n log n) in best, average and worst case; O(1) extra space (in place).

– UNDERSTAND **Why does heapify start from the last non-leaf and move backwards during build-heap?** 3 marks

heapify(i) assumes the subtrees of i are already heaps. Leaves are heaps trivially, and processing indices from ⌊n/2⌋ − 1 down to 0 fixes every node after its children, so the assumption always holds.

– APPLY **Build a max-heap from {1, 14, 3, 7, 9, 2} and write the array.** 4 marks

n = 6, heapify i = 2: 3 vs child 2 → stays. i = 1: 14 vs 7, 9 → stays. i = 0: 1 vs 14, 3 → swap: 14 1 3 7 9 2; sift 1 at index 1: 1 vs 7, 9 → swap with 9: 14 9 3 7 1 2. Heap: 14 9 3 7 1 2.

– APPLY **Starting from the heap 14 9 3 7 1 2, do one extraction round and show the array.** 3 marks

Swap 14 ↔ 2: 2 9 3 7 1 | 14. heapify(0, 5): 2 vs 9, 3 → swap with 9: 9 2 3 7 1; at index 1: 2 vs 7, 1 → swap with 7: 9 7 3 2 1 | 14.

– ANALYZE **You must find the 10 largest of 10 crore transaction amounts in a stream. Compare full heap sort with keeping a min-heap of size 10.** 4 marks

Full heap sort needs all data in memory and O(n log n) ≈ $10^8 \times 27$ operations. A min-heap of the 10 largest seen so far costs O(log 10) per arriving element when it beats the heap's minimum, O(n log k) ≈ $10^8 \times 3.3$, uses 10 slots of memory and works on a stream. Choose the size-k heap.

BEYOND THE SYLLABUS

Not for the exam. Heap sort is rarely the main sort in a library, but its guarantee is what makes introsort safe: when quick sort's recursion exceeds $2 \log_2 n$ levels, the C++ standard library hands the range to heap sort. The heap itself is everywhere: priority queues in Dijkstra's algorithm (Unit 5), timers in operating-system kernels, Top-k queries in search engines. Newer variants like bottom-up heapsort and the d-ary heap cut cache misses; search for "bottom-up heapsort" and "d-ary heap Dijkstra".

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
Merge sort	T(n) = 2T(n/2) + n → Θ(n log n) always; O(n) space; stable	6.3
Quick sort	Partition around pivot; avg Θ(n log n), worst Θ(n^2) with extreme pivot; in place; not stable	6.3
Heap sort	build-heap O(n), then n−1 sift-downs; Θ(n log n) always; in place; not stable	6.4

Exam question bank

Long-answer exam questions on these chapters, each with an outline of the points an examiner looks for. Answer one in full on paper before you read its outline.

Remember and understand

- **3. Explain the divide-and-conquer structure of merge sort, write the merge procedure, and solve its recurrence. (8 marks)**

  Outline: split at mid, recursive sorts, merge with two pointers and at most n−1 comparisons; $T(n) = 2T(n/2) + cn$ solved by recursion tree ($\log_2 n$ levels × cn) or Master theorem; O(n) extra space; stability.

Apply

- **7. Trace quick sort on {40, 55, 70, 85, 30, 20, 10, 60, 50} with the first element as pivot (Hoare's partition), showing the array after each partition and the recursion tree. State the number of comparisons and whether this input is a good or a bad case. (12 marks)**

  Outline: pivot 40; i moves right to 55, j moves left to 10, swap → 40 10 70 85 30 20 55 60 50; i to 70, j to 20, swap → 40 10 20 85 30 70 55 60 50; i to 85, j to 30, swap → 40 10 20 30 85 70 55 60 50; i and j cross, swap pivot with a[j] = 30 → 30 10 20 **40** 85 70 55 60 50; recurse on {30, 10, 20} and {85, 70, 55, 60, 50}; draw the tree; about n + 1 comparisons per partition; the split 3 : 5 is reasonably balanced, a good case, whereas sorted input with a first-element pivot gives $\Theta(n^2)$.

- **8. Sort {18, 7, 23, 4, 15, 9} by heap sort. Show the max-heap after build-heap and the array after every extraction. (10 marks)**

  Outline: heapify i = 2 (23 ≥ 9), i = 1 (7 vs 4, 15 → swap with 15), i = 0 (18 vs 15, 23 → swap with 23): heap 23 15 18 4 7 9; six rounds: 18 15 9 4 7 | 23; 15 7 9 4 | 18 23; 9 7 4 | 15 18 23; 7 4 | 9 …; 4 | 7 …; output 4 7 9 15 18 23.

Glossary (terms used in this lesson)

Build-heap	Turning an array into a heap by sifting down from the last non-leaf to the root; O(n).
Comparison sort	A sort that learns the order of keys only by comparing pairs; bounded below by Ω(n log n).
Heapify (sift down)	Restore the heap property at a node by swapping it downward with its larger child.
Pivot	The element quick sort partitions around; ends in its final position.

Run	A sorted block of records written to disk in the first phase of external sorting.
Stable sort	A sort that keeps equal keys in their original relative order.

Sources and further reading

- Mark Allen Weiss, *Data Structures and Algorithm Analysis in C*, 2nd edition: the chapters on hashing and sorting are prescribed reading in many syllabi.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*: heapsort, quicksort, lower bounds and linear-time sorts, hash tables, and string matching (KMP).
- Ellis Horowitz, Sartaj Sahni and Susan Anderson-Freed, *Fundamentals of Data Structures in C*: sorting (including external sorting) and hashing.
- Robert Sedgewick and Kevin Wayne, *Algorithms*, 4th edition: shellsort, hash tables, tries and substring search, with the free companion site.
- Donald E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*: the source of the probe-count formulas for linear probing and of the multiplication-method constant.
- Wikipedia, "Linear probing": en.wikipedia.org/wiki/Linear_probing, for the expected-cost bounds in terms of the load factor (opened Nov 2026).
- Wikipedia, "Knuth–Morris–Pratt algorithm": en.wikipedia.org/wiki/Knuth–Morris–Pratt_algorithm, for the 1977 date and the Θ(m) + Θ(n) bounds (opened Nov 2026).
- Wikipedia, "Shellsort": en.wikipedia.org/wiki/Shellsort, for Shell's 1959 paper, the gap sequences and their worst-case bounds (opened Nov 2026).
- Syllabi: the universal syllabus and a range of university course documents.

Built for the University free n easy YouTube channel from the universal syllabus of Data Structures (Oct 2026). Written with top universities' syllabi and standard textbooks; check a formula against your prescribed book before an exam, and report mistakes on the channel.