Mergesort and Quicksort

Data Structures & Algorithms

Based on Robert Sedgewick & Kevin Wayne

Algorithms, 4th Edition (Addison-Wesley, 2011)

Chapters 2.2 & 2.3

Duration: 4 hours (with breaks)

Learning Objectives

By the end of this lecture, students will be able to:

1. Explain the divide-and-conquer paradigm and identify its three fundamental steps in the context of sorting algorithms.

2. Implement top-down (recursive) and bottom-up (iterative) mergesort, trace their execution on sample inputs, and prove their O(n log n) time complexity.

3. Implement quicksort with both Lomuto and Hoare partitioning schemes, analyze best-case, worst-case, and average-case performance, and explain the role of randomization.

4. Apply practical improvements to quicksort including median-of-three pivot selection, small-subarray cutoff to insertion sort, and Dijkstra's three-way partitioning for duplicate keys.

5. Compare mergesort and quicksort along the dimensions of time complexity, space complexity, stability, cache performance, and suitability for different application domains.

6. Solve problems requiring algorithm traces, correctness proofs, and complexity analysis of mergesort and quicksort variants.

---

1. The Divide-and-Conquer Paradigm

1.1 What Divide-and-Conquer Means

In Lectures 2 and 3 we studied elementary sorting algorithms -- selection sort, insertion sort, and shellsort -- that operate by iteratively building up a sorted portion of the array. Those algorithms are fundamentally *incremental*: they process one element at a time, inserting or swapping it into position. Their worst-case performance is quadratic, $O(n^2)$, because each element may need to be compared against every other element.

Divide-and-conquer is a fundamentally different algorithmic strategy. Instead of processing elements one at a time, we break the entire problem into smaller subproblems, solve those subproblems recursively, and then combine the solutions. This strategy is one of the most powerful ideas in all of algorithm design, and it underlies both of the algorithms we study today: mergesort and quicksort.

The intuition is simple but profound. Sorting an array of one million elements is hard. But sorting two arrays of 500,000 elements is only slightly easier in absolute terms -- yet if we can combine two sorted halves efficiently, we cut the problem in half at each level of recursion, and this geometric reduction leads to dramatically better performance.

1.2 The Three Steps

Every divide-and-conquer algorithm follows three steps:

Step 1 -- Divide. Split the problem into two or more smaller subproblems of the same type. In mergesort, we divide the array into two equal halves. In quicksort, we divide the array around a pivot element.

Step 2 -- Conquer. Solve each subproblem recursively. When the subproblem is small enough (typically a single element or an empty array), solve it directly -- this is the *base case*.

Step 3 -- Combine. Merge the solutions to the subproblems into a solution for the original problem. In mergesort, the combine step is the merge operation. In quicksort, the combine step is trivial because the partitioning has already placed elements in the correct relative positions.

An essential observation is that mergesort and quicksort differ in *where the work happens*. In mergesort, the divide step is trivial (just compute the midpoint) and the combine step does the heavy lifting (merging two

sorted halves). In quicksort, the divide step does the heavy lifting (partitioning around the pivot) and the combine step is trivial (no work needed after recursion returns). This duality is aesthetically pleasing and important to understand.

1.3 Why Divide-and-Conquer Leads to Efficient Algorithms

The efficiency of divide-and-conquer algorithms stems from the fact that splitting a problem of size n into two subproblems of size n/2, with O(n) work at each level, produces only O(log n) levels of recursion. The total work is therefore O(n log n).

Consider the contrast with an incremental algorithm. If processing the k-th element requires O(k) work, then the total work is:

```
1 + 2 + 3 + ... + n = n(n+1)/2 = O(n^2)
```

With divide-and-conquer, we have O(n) work at each of O(log n) levels:

```
n + n + n + ... + n  (log n terms) = O(n log n)
```

For n = 1,000,000, the difference between n^2 = 10^12 and n log n ~ 20,000,000 is a factor of 50,000. This is the difference between a computation that takes seconds and one that takes days.

1.4 Recurrence Relations

The running time of a divide-and-conquer algorithm is naturally expressed as a *recurrence relation*. For both mergesort and quicksort (in the best/average case), the recurrence has the form:

```
T(n) = 2T(n/2) + O(n)
T(1) = O(1)
```

This says: the time to solve a problem of size n equals the time to solve two subproblems of size n/2 plus O(n) work for the divide/combine steps. We will solve this recurrence explicitly when we analyze mergesort in Section 2.

DEFINITION 1.1 -- RECURRENCE RELATION

A recurrence relation is an equation that defines a function in terms of its values on smaller inputs. For divide-and-conquer algorithms, the recurrence captures the recursive structure of the algorithm: the cost at each level is the sum of the costs of the subproblems plus the cost of the divide and combine steps.

Lecture 4 of 12 -- Data Structures & Algorithms -- Based on Sedgewick's Algorithms, 4th Edition

---

2. Mergesort

2.1 The Idea

Mergesort is the prototypical divide-and-conquer sorting algorithm. It was invented by John von Neumann in 1945, making it one of the oldest sorting algorithms still in widespread use. The idea is beautifully simple:

1. Divide the array into two halves.
2. Recursively sort each half.
3. Merge the two sorted halves into a single sorted array.

The key insight is that merging two sorted arrays is a linear-time operation. Given two sorted arrays of size n/2 each, we can produce a sorted array of size n by repeatedly comparing the smallest unprocessed elements from each half and copying the smaller one.

2.2 Abstract In-Place Merge

Before presenting the full mergesort algorithm, let us first understand the merge operation in detail. The merge takes an array `a[lo..hi]` where `a[lo..mid]` and `a[mid+1..hi]` are each individually sorted, and rearranges the array so that `a[lo..hi]` is fully sorted.

We use an auxiliary array `aux[]` of the same size as the input. The merge proceeds as follows:

```
merge(a, aux, lo, mid, hi):
    // Copy a[lo..hi] to aux[lo..hi]
    for k = lo to hi:
        aux[k] = a[k]

    // Merge back to a[lo..hi]
    i = lo          // index into left half  (aux[lo..mid])
    j = mid + 1     // index into right half (aux[mid+1..hi])

    for k = lo to hi:
        if       i > mid:          a[k] = aux[j++]   // left half exhausted
        else if j > hi:            a[k] = aux[i++]   // right half exhausted
        else if aux[j] < aux[i]:   a[k] = aux[j++]   // right element smaller
        else:                      a[k] = aux[i++]   // left element smaller or equal
```

PROPOSITION 2.1

The merge operation uses at most n comparisons to merge two sorted halves of total size n, and it preserves the relative order of equal elements (it is *stable*).

Proof. Each iteration of the main loop places exactly one element into its final position and advances either `i` or `j`. Each comparison involves exactly one element from each half. Since we place n elements total, we make at most n comparisons. The stability follows from the `else` clause: when `aux[j]` equals `aux[i]`, we take the element from the left half (`aux[i]`), preserving the original relative order of equal elements. QED.

2.3 Top-Down Mergesort (Recursive)

Top-down mergesort is the direct recursive implementation of the divide-and-conquer strategy:

```
sort(a, aux, lo, hi):
    if hi <= lo: return          // base case: subarray of size 0 or 1
    mid = lo + (hi - lo) / 2     // divide: compute midpoint
    sort(a, aux, lo, mid)        // conquer: sort left half
    sort(a, aux, mid + 1, hi)    // conquer: sort right half
    merge(a, aux, lo, mid, hi)   // combine: merge sorted halves
```

The top-level call is `sort(a, aux, 0, n-1)` where `aux` is pre-allocated with size n.

2.4 Trace Through a Complete Example

Let us trace mergesort on the array `[M, E, R, G, E, S, O, R, T]` (indices 0 through 8). We represent each recursive call showing the subarray being processed, with indentation indicating recursion depth.

```
Initial array: [M, E, R, G, E, S, O, R, T]
                0  1  2  3  4  5  6  7  8


sort(0, 8)
  sort(0, 4)                         -- left half
    sort(0, 2)                       -- left quarter
      sort(0, 1)
        sort(0, 0)                   -- base case: [M]
        sort(1, 1)                   -- base case: [E]
        merge(0, 0, 1)               -- merge [M] and [E]
        Result: [E, M, ...]
      sort(2, 2)                     -- base case: [R]
      merge(0, 1, 2)                 -- merge [E,M] and [R]
      Result: [E, M, R, ...]
    sort(3, 4)                       -- right quarter (of left half)
      sort(3, 3)                     -- base case: [G]
      sort(4, 4)                     -- base case: [E]
      merge(3, 3, 4)                 -- merge [G] and [E]
      Result: [..., E, G, ...]
    merge(0, 2, 4)                   -- merge [E,M,R] and [E,G]
    Result: [E, E, G, M, R, ...]

  sort(5, 8)                         -- right half
    sort(5, 6)
      sort(5, 5)                     -- base case: [S]
      sort(6, 6)                     -- base case: [O]
      merge(5, 5, 6)                 -- merge [S] and [O]
      Result: [..., O, S, ...]
    sort(7, 8)
      sort(7, 7)                     -- base case: [R]
      sort(8, 8)                     -- base case: [T]
      merge(7, 7, 8)                 -- merge [R] and [T]
      Result: [..., R, T]
    merge(5, 6, 8)                   -- merge [O,S] and [R,T]
    Result: [..., O, R, S, T]

  merge(0, 4, 8)                     -- final merge
  Result: [E, E, G, M, O, R, R, S, T]
```

Let us trace the final merge `merge(0, 4, 8)` in detail, where left half = `[E, E, G, M, R]` and right half = `[O, R, S, T]`:

```
aux:  [E, E, G, M, R, O, R, S, T]
       i                 j

Step 1: aux[i]=E < aux[j]=O  -> a[0]=E, i++
Step 2: aux[i]=E < aux[j]=O  -> a[1]=E, i++
Step 3: aux[i]=G < aux[j]=O  -> a[2]=G, i++
Step 4: aux[i]=M < aux[j]=O  -> a[3]=M, i++
Step 5: aux[j]=O < aux[i]=R  -> a[4]=O, j++
Step 6: aux[i]=R = aux[j]=R  -> a[5]=R, i++  (take from left for stability)
Step 7: j=7: aux[j]=R < aux[i]=*  (i>mid, left exhausted)
        Actually: i > mid, so a[6]=R, j++
Step 8: i > mid  -> a[7]=S, j++
Step 9: i > mid  -> a[8]=T, j++


Final: [E, E, G, M, O, R, R, S, T]
```

The array is now sorted. Notice that the two E's and the two R's maintained their original relative order -- this is a consequence of mergesort's stability.

2.5 Bottom-Up Mergesort (Iterative)

An alternative to the recursive top-down approach is bottom-up mergesort, which processes the array in passes. In the first pass, we merge subarrays of size 1 into sorted subarrays of size 2. In the second pass, we merge subarrays of size 2 into sorted subarrays of size 4. We continue doubling until the entire array is sorted.

```
bottom_up_mergesort(a, n):
    aux = new array of size n
    sz = 1
    while sz < n:
        lo = 0
        while lo < n - sz:
            merge(a, aux, lo, lo + sz - 1, min(lo + 2*sz - 1, n - 1))
            lo = lo + 2 * sz
        sz = 2 * sz
```

Trace on `[M, E, R, G, E, S, O, R, T]`:

```
Pass 1 (sz=1): merge pairs of size 1
  merge(0,0,1): [M,E] -> [E,M]
  merge(2,2,3): [R,G] -> [G,R]
  merge(4,4,5): [E,S] -> [E,S]
  merge(6,6,7): [O,R] -> [O,R]
  (T at index 8 has no pair -- left alone)
  Array: [E, M, G, R, E, S, O, R, T]

Pass 2 (sz=2): merge pairs of size 2
  merge(0,1,3): [E,M] + [G,R] -> [E,G,M,R]
  merge(4,5,7): [E,S] + [O,R] -> [E,O,R,S]
  (T at index 8 has no pair)
  Array: [E, G, M, R, E, O, R, S, T]

Pass 3 (sz=4): merge pairs of size 4
  merge(0,3,7): [E,G,M,R] + [E,O,R,S] -> [E,E,G,M,O,R,R,S]
  (T at index 8 has no pair)
  Array: [E, E, G, M, O, R, R, S, T]

Pass 4 (sz=8): merge the two remaining pieces
  merge(0,7,8): [E,E,G,M,O,R,R,S] + [T] -> [E,E,G,M,O,R,R,S,T]

Final: [E, E, G, M, O, R, R, S, T]
```

Bottom-up mergesort has the same asymptotic complexity as top-down mergesort but avoids the overhead of recursion. It is also the natural approach for sorting linked lists, where random access is expensive but sequential traversal is cheap.

2.6 Analysis of Mergesort

THEOREM 2.1

Top-down mergesort uses between 1/2 n lg n and n lg n comparisons to sort an array of n elements.

Proof. Let C(n) denote the number of comparisons to sort an array of n elements. We have the recurrence:

```
C(n) = C(n/2) + C(n/2) + cost_of_merge(n)
C(1) = 0
```

The merge of two halves of total size n uses between n/2 and n comparisons. The minimum occurs when every element of one half is smaller than every element of the other half (so we exhaust one half in n/2 comparisons and copy the rest without comparing). The maximum occurs when elements alternate between halves.

Upper bound (n lg n): Using the worst case for merge (n comparisons at each level):

```
C(n) <= 2 C(n/2) + n
     <= 2(2 C(n/4) + n/2) + n = 4 C(n/4) + 2n
     <= 4(2 C(n/8) + n/4) + 2n = 8 C(n/8) + 3n
     ...
     <= 2^k C(n/2^k) + kn
```

Setting k = lg n (so that n/2^k = 1 and C(1) = 0):

```
C(n) <= n * 0 + n lg n = n lg n
```

Lower bound (1/2 n lg n): Using the best case for merge (n/2 comparisons at each level):

```
C(n) >= 2 C(n/2) + n/2
```

By the same expansion, C(n) >= (1/2) n lg n. QED.

We can visualize this with the *recurrence tree*:

```
Level 0:                    [n]                    cost: n
                          /     \
Level 1:            [n/2]     [n/2]                cost: n
                   /    \     /    \
Level 2:        [n/4] [n/4] [n/4] [n/4]            cost: n

                 ...    ...    ...    ...

                 ...    ...    ...    ...
Level lg n:     [1][1][1].............[1]          cost: 0


Total levels: lg n + 1
Cost per level: at most n (for the merge operations)
Total cost: at most n lg n
```
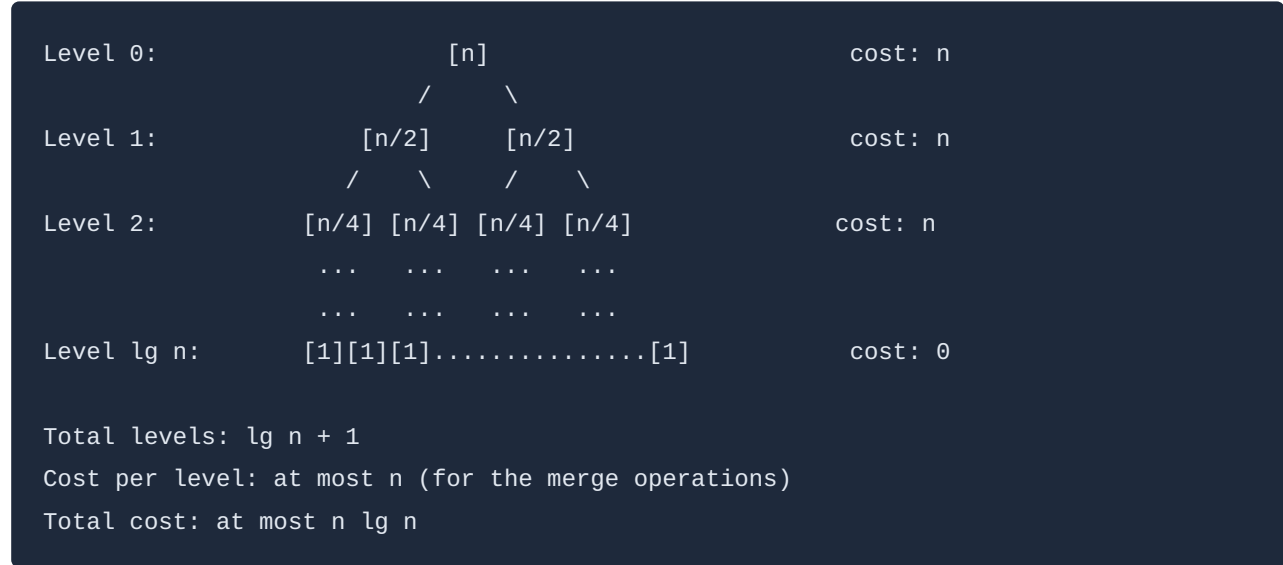
Each level of the tree represents one level of recursion. At level k, there are 2^k subproblems each of size n/2^k. The merge cost at level k is at most n (since the total number of elements merged at each level is n). There are lg n levels where merging occurs (level 0 through level lg n - 1), giving at most n lg n total comparisons.

PROPOSITION 2.2

Mergesort uses O(n) extra space for the auxiliary array.

Proof. The auxiliary array `aux[]` has size n. Although there are lg n levels of recursion (using O(lg n) stack space), the dominant space cost is the auxiliary array. Therefore the total extra space is O(n). QED.

PROPOSITION 2.3

Mergesort is a *stable* sorting algorithm.

Proof. Equal elements are never reordered. In the merge step, when two elements are equal, we always take the element from the left subarray first. Since the left subarray contains elements that originally appeared before the right subarray's elements, the original relative order of equal elements is preserved. By induction on the recursion, the entire sort is stable. QED.

2.7 Practical Improvements

Sedgewick and Wayne describe three practical improvements to mergesort that are important in real implementations:

Improvement 1: Cutoff to insertion sort for small subarrays. Mergesort has higher overhead per comparison than insertion sort due to the merge operation and recursive calls. For small subarrays (typically n <= 7 to 15), insertion sort is faster. We modify the base case:

```
sort(a, aux, lo, hi):
    if hi - lo < CUTOFF:
        insertion_sort(a, lo, hi)
        return
    mid = lo + (hi - lo) / 2
    sort(a, aux, lo, mid)
    sort(a, aux, mid + 1, hi)
    merge(a, aux, lo, mid, hi)
```

This can improve running time by 10-15% in practice.

Improvement 2: Test whether the array is already sorted. If `a[mid] <= a[mid+1]`, then the two halves are already in sorted order and we can skip the merge entirely:

```
sort(a, aux, lo, hi):
    if hi <= lo: return
    mid = lo + (hi - lo) / 2
    sort(a, aux, lo, mid)
    sort(a, aux, mid + 1, hi)
    if a[mid] <= a[mid + 1]: return    // already sorted
    merge(a, aux, lo, mid, hi)
```

This makes mergesort linear for nearly-sorted inputs while preserving O(n log n) worst case.

Improvement 3: Eliminate the copy to auxiliary array. Instead of copying from `a` to `aux` at the start of each merge, we can alternate the roles of `a` and `aux` at each level of recursion. This eliminates half the array-copy operations, at the cost of a more complex implementation.

Lecture 4 of 12 -- Data Structures & Algorithms -- Based on Sedgewick's Algorithms, 4th Edition

3. Quicksort

3.1 The Idea

Quicksort, invented by Tony Hoare in 1960, is arguably the most important sorting algorithm in practice. It is the default general-purpose sorting algorithm in many standard libraries and is typically 2-3 times faster than mergesort in practice, despite having the same O(n log n) average-case complexity.

Like mergesort, quicksort is a divide-and-conquer algorithm. But whereas mergesort divides the array into two equal halves and does the hard work during the combine step, quicksort does the hard work during the divide step and has a trivial combine step.

The basic quicksort algorithm works as follows:

1. **Partition:** Choose a *pivot* element. Rearrange the array so that all elements less than the pivot come before it, all elements greater than the pivot come after it, and the pivot is in its final sorted position.
2. **Recurse:** Recursively sort the elements before the pivot and the elements after the pivot.
3. **Combine:** Nothing to do -- the array is already sorted.

```
quicksort(a, lo, hi):
    if hi <= lo: return
    j = partition(a, lo, hi)     // partition and get pivot index
    quicksort(a, lo, j - 1)    // sort left part
    quicksort(a, j + 1, hi)    // sort right part
```

The beauty of quicksort is that after partitioning, the pivot element `a[j]` is in its final position: every element to its left is smaller (or equal) and every element to its right is larger (or equal). No further work is needed to combine the sorted subarrays.

3.2 Lomuto Partition Scheme

The Lomuto partition scheme is the simpler of the two common partition implementations. It is commonly used in textbooks for its clarity, although Hoare's original scheme is more efficient.

The Lomuto scheme uses the last element as the pivot and maintains two regions: elements less than the pivot (from `lo` to `i`) and elements greater than or equal to the pivot (from `i+1` to `j-1`). Elements from `j` to `hi-1` are unprocessed.

```
lomuto_partition(a, lo, hi):
    pivot = a[hi]               // choose last element as pivot
    i = lo - 1                  // boundary of "less than" region

    for j = lo to hi - 1:
        if a[j] < pivot:
            i = i + 1
            swap(a[i], a[j])

    swap(a[i + 1], a[hi])       // place pivot in correct position
    return i + 1
```

Detailed trace on `[Q, U, I, C, K, S, O, R, T]` (using last element T as pivot):

```
Initial: [Q, U, I, C, K, S, O, R, T]    pivot = T
          0  1  2  3  4  5  6  7  8      i = -1

j=0: a[0]=Q < T? Yes. i=0. swap(a[0],a[0]). [Q, U, I, C, K, S, O, R, T]
j=1: a[1]=U < T? No.                        [Q, U, I, C, K, S, O, R, T]
j=2: a[2]=I < T? Yes. i=1. swap(a[1],a[2]). [Q, I, U, C, K, S, O, R, T]
j=3: a[3]=C < T? Yes. i=2. swap(a[2],a[3]). [Q, I, C, U, K, S, O, R, T]
j=4: a[4]=K < T? Yes. i=3. swap(a[3],a[4]). [Q, I, C, K, U, S, O, R, T]
j=5: a[5]=S < T? Yes. i=4. swap(a[4],a[5]). [Q, I, C, K, S, U, O, R, T]
j=6: a[6]=O < T? Yes. i=5. swap(a[5],a[6]). [Q, I, C, K, S, O, U, R, T]
j=7: a[7]=R < T? Yes. i=6. swap(a[6],a[7]). [Q, I, C, K, S, O, R, U, T]

Final swap: swap(a[7], a[8]) -> swap(U, T)
Result: [Q, I, C, K, S, O, R, T, U]
                                 ^
                       pivot at index 7

Everything left of T (index 7) is < T.  Everything right of T is >= T.
```

3.3 Hoare Partition Scheme

Hoare's original partition scheme is more efficient than Lomuto's because it uses fewer swaps on average (roughly three times fewer). It works by scanning inward from both ends of the array.

```
hoare_partition(a, lo, hi):
    pivot = a[lo]                // choose first element as pivot
    i = lo - 1
    j = hi + 1

    while true:
        do: i = i + 1
        while a[i] < pivot

        do: j = j - 1
        while a[j] > pivot

        if i >= j: return j

        swap(a[i], a[j])
```

DEFINITION 3.1 -- PARTITIONING INVARIANT (HOARE)

After `hoare_partition(a, lo, hi)` returns index `j`, we have: (1) all elements in `a[lo..j]` are less than or equal to the pivot, (2) all elements in `a[j+1..hi]` are greater than or equal to the pivot, and (3) `lo <= j < hi`.

Detailed trace on `[K, R, A, T, E, L, E, P, U, C]` (using first element K as pivot):

```
Initial: [K, R, A, T, E, L, E, P, U, C]     pivot = K
          0  1  2  3  4  5  6  7  8  9      i = -1, j = 10

Iteration 1:
  Scan i right: i=0, a[0]=K, K < K? No. Stop. i=0.
  Scan j left:  j=9, a[9]=C, C > K? No. Stop. j=9.
  i < j, so swap(a[0], a[9]):
  [C, R, A, T, E, L, E, P, U, K]

Iteration 2:
  Scan i right: i=1, a[1]=R, R < K? No. Stop. i=1.
  Scan j left:  j=8, a[8]=U, U > K? Yes. j=7, a[7]=P, P > K? Yes.
                j=6, a[6]=E, E > K? No. Stop. j=6.
  i < j, so swap(a[1], a[6]):
  [C, E, A, T, E, L, R, P, U, K]

Iteration 3:
  Scan i right: i=2, a[2]=A, A < K? Yes. i=3, a[3]=T, T < K? No. Stop. i=3.
  Scan j left:  j=5, a[5]=L, L > K? Yes. j=4, a[4]=E, E > K? No. Stop. j=4.
  i < j, so swap(a[3], a[4]):
  [C, E, A, E, T, L, R, P, U, K]

Iteration 4:
  Scan i right: i=4, a[4]=T, T < K? No. Stop. i=4.
  Scan j left:  j=3, a[3]=E, E > K? No. Stop. j=3.
  i >= j (4 >= 3), so return j = 3.

Result: [C, E, A, E | T, L, R, P, U, K]
          <=K region     >=K region
          a[0..3]        a[4..9]
```

Note that in Hoare's scheme, the pivot is not necessarily at the partition boundary. The return value `j` indicates that `a[lo..j]` contains only elements <= pivot and `a[j+1..hi]` contains only elements >= pivot. The recursive calls are then `quicksort(a, lo, j)` and `quicksort(a, j+1, hi)`.

Sedgewick's textbook uses a variant that is closer to Hoare's scheme but places the pivot in its final position. The version in the textbook uses `a[lo]` as pivot, scans `i` from `lo+1` and `j` from `hi`, and swaps the pivot into position at the end:

```
sedgewick_partition(a, lo, hi):
    pivot = a[lo]
    i = lo
    j = hi + 1

    while true:
        // scan right, find element >= pivot
        while a[++i] < pivot:
            if i == hi: break

        // scan left, find element <= pivot
        while a[--j] > pivot:
            if j == lo: break     // redundant: a[lo] = pivot

        if i >= j: break
        swap(a[i], a[j])

    swap(a[lo], a[j])        // put pivot in position
    return j                  // a[j] is now in its final place
```

3.4 Complete Quicksort Trace

Let us trace the full quicksort execution on `[Q, U, I, C, K, S, O, R, T]` using Sedgewick's partition (first element as pivot):

```
Initial: [Q, U, I, C, K, S, O, R, T]
         0  1  2  3  4  5  6  7  8

quicksort(0, 8):
  partition(0, 8) with pivot Q:
    i scans right: a[1]=U >= Q, stop. i=1.
    j scans left:  a[8]=T > Q? Yes. a[7]=R > Q? Yes. a[6]=O > Q? No. j=6.
    i < j: swap(a[1], a[6]): [Q, O, I, C, K, S, U, R, T]
    i scans right: a[2]=I < Q? Yes. a[3]=C < Q? Yes. a[4]=K < Q? Yes. a[5]=S >= Q. i=5.
    j scans left: a[5]=S > Q? Yes. a[4]=K > Q? No. j=4.
    i >= j (5 >= 4): break.
    swap(a[0], a[4]): [K, O, I, C, Q, S, U, R, T]
    return 4.


  Array: [K, O, I, C, Q, S, U, R, T]
                      ^
                Q in final position (index 4)


  quicksort(0, 3):   sort [K, O, I, C]
    partition(0, 3) with pivot K:
      i scans right: a[1]=O >= K, stop. i=1.
      j scans left: a[3]=C > K? No. j=3.
      i < j: swap(a[1], a[3]): [K, C, I, O]
      i scans right: a[2]=I < K? Yes. a[3]=O >= K. i=3.
      j scans left: a[2]=I > K? No. j=2.
      i >= j (3 >= 2): break.
      swap(a[0], a[2]): [I, C, K, O]
      return 2.


    quicksort(0, 1):  sort [I, C]
      partition(0, 1) with pivot I:
        i scans right: a[1]=C < I? Yes. i=2 (past hi). break.
        j scans left: a[1]=C > I? No. j=1. Then a[0]=I > I? No. j=0.
        Actually: j starts at hi+1=2, --j=1. a[1]=C > I? No. j=1.
        i >= j (2 >= 1): break.
        swap(a[0], a[1]): [C, I]
        return 1.
      quicksort(0, 0): base case.
      quicksort(2, 1): base case (lo > hi).
```

```
    quicksort(3, 3): base case.

  quicksort(5, 8):   sort [S, U, R, T]
    partition(5, 8) with pivot S:
       i scans right: a[6]=U >= S, stop. i=6.
       j scans left: a[8]=T > S? Yes. a[7]=R > S? No. j=7.
       i < j: swap(a[6], a[7]): [S, R, U, T]  (indices 5-8)
       i scans right: a[7]=U >= S. i=7.
       j scans left: a[6]=R > S? No. j=6.
       i >= j (7 >= 6): break.
       swap(a[5], a[6]): [R, S, U, T] (indices 5-8)
       return 6.

    quicksort(5, 5): base case.
    quicksort(7, 8): sort [U, T]
       partition(7, 8) with pivot U:
          i scans right: a[8]=T < U? Yes. i=9 (past hi). break.
          j scans left: a[8]=T > U? No. j=8.
          i >= j: break.
          swap(a[7], a[8]): [T, U]
          return 8.
       quicksort(7, 7): base case.
       quicksort(9, 8): base case.

 Final array: [C, I, K, O, Q, R, S, T, U]
```

3.5 Analysis of Quicksort

Best Case: O(n log n)

The best case occurs when every partition splits the array exactly in half. The recurrence is:

```
T(n) = 2T(n/2) + O(n)
```

This is the same as mergesort, giving T(n) = O(n log n).

Worst Case: O(n^2)

THEOREM 3.1

The worst-case running time of quicksort is O(n^2).

Proof. The worst case occurs when the partition always produces one subproblem of size n-1 and one of size 0. This happens when the pivot is always the smallest or largest element. The recurrence becomes:

```
T(n) = T(n-1) + T(0) + O(n) = T(n-1) + O(n)
```

Expanding:

```
T(n) = O(n) + O(n-1) + O(n-2) + ... + O(1) = O(n^2)
```

This worst case arises when the input is already sorted (or reverse sorted) and the first element is chosen as pivot. QED.

Average Case: O(1.39 n lg n)

THEOREM 3.2

The average number of comparisons C_n used by quicksort to sort n distinct elements (with all n! permutations equally likely) satisfies:

```
C_n = 2(n + 1) H_n - 2n  ~  1.39 n lg n
```

where H_n = 1 + 1/2 + 1/3 + ... + 1/n is the n-th harmonic number.

Proof sketch. Each pair of elements is compared at most once (when one of them is the pivot). Consider elements with final sorted positions i and j (i < j). They are compared if and only if one of them is the first element in the range [i, j] to be chosen as a pivot. The probability of this is 2/(j - i + 1).

The expected total number of comparisons is:

```
C_n = sum_{i=1}^{n} sum_{j=i+1}^{n} 2/(j - i + 1)

    = 2 sum_{i=1}^{n} sum_{k=2}^{n-i+1} 1/k     (substituting k = j - i + 1)

    = 2 sum_{i=1}^{n} (H_{n-i+1} - 1)

    = 2(n+1) H_n - 4n + 2

    ~ 2(n+1) ln n - 4n

    ~ 2n ln n

    = 2n (ln 2)(log_2 n)

    ~ 1.39 n lg n
```

So on average, quicksort uses about 39% more comparisons than the best case of n lg n, but this is still O(n log n). QED.

PROPOSITION 3.1

Quicksort uses O(log n) extra space on average for the recursion stack.

Proof. On average, the partition is reasonably balanced, giving O(log n) levels of recursion. In the worst case, the recursion depth is O(n), but this can be mitigated by always recursing on the smaller subarray first (tail-call optimization on the larger subarray). QED.

3.6 Why Randomization Helps

The worst case for quicksort occurs with specific input patterns (sorted, reverse sorted, all equal elements). In practice, we can avoid these patterns by *randomly shuffling the array* before sorting:

```
randomized_quicksort(a, n):
    shuffle(a)                  // random permutation
    quicksort(a, 0, n - 1)
```

After shuffling, every permutation is equally likely, so the expected running time is O(1.39 n lg n) regardless of the original input order. This is a probabilistic guarantee: no specific input can reliably cause worst-case behavior.

An alternative is to choose a random pivot at each partition step. Both approaches achieve the same effect.

Sedgewick and Wayne emphasize this point strongly: **always shuffle before quicksorting.** The cost of the shuffle is O(n), which is negligible compared to the O(n log n) sorting time, and it provides a probabilistic guarantee against worst-case inputs.

3.7 Comparison with Mergesort

Property	Mergesort	Quicksort
Worst-case time	O(n log n)	O(n^2)
Average-case time	O(n log n)	O(1.39 n lg n)
Best-case time	O(1/2 n lg n)	O(n log n)
Extra space	O(n)	O(log n)
Stable?	Yes	No (in standard implementations)
In-place?	No	Yes (except recursion stack)
Cache performance	Moderate	Excellent
Practical speed	Good	Fastest in practice

Quicksort is generally preferred for arrays because: (1) it sorts in place, using only O(log n) stack space vs O(n) auxiliary space for mergesort; (2) its inner loop is extremely tight (increment pointer, compare, branch), leading to excellent cache performance; (3) despite the O(n^2) worst case, randomization makes it O(n log n) with high probability.

Mergesort is preferred when: (1) worst-case O(n log n) is required as a guarantee (not just a probabilistic guarantee); (2) stability is required; (3) sorting linked lists (where mergesort's space overhead disappears and quicksort's random access pattern is a liability).

Lecture 4 of 12 -- Data Structures & Algorithms -- Based on Sedgewick's Algorithms, 4th Edition

4. Quicksort Improvements

4.1 Median-of-Three Pivot Selection

Choosing the first element as pivot is simple but can lead to poor partitions. A better strategy is the *median-of-three*: examine the first, middle, and last elements, and use the median (middle value) as the pivot.

```
median_of_three(a, lo, hi):
    mid = lo + (hi - lo) / 2
    // Sort a[lo], a[mid], a[hi] among themselves
    if a[mid] < a[lo]: swap(a[lo], a[mid])
    if a[hi]  < a[lo]: swap(a[lo], a[hi])
    if a[hi]  < a[mid]: swap(a[mid], a[hi])
    // Now a[lo] <= a[mid] <= a[hi]
    swap(a[mid], a[lo])    // place median at a[lo] for standard partition
    return partition(a, lo, hi)
```

Why this helps: The median of three random values is a better estimate of the true median than a single random value. The expected partition ratio improves from 1:1 (ideal) to roughly 1/3 : 2/3 in the worst common case, which reduces the constant factor in the average-case comparison count.

PROPOSITION 4.1

With median-of-three pivot selection, the average number of comparisons is approximately 1.19 n lg n, compared to 1.39 n lg n for random pivot selection. This is a roughly 14% improvement.

Median-of-three also serves as a sentinel: after the three-way sort, `a[lo]` is the smallest of the three and `a[hi]` is the largest. This means `a[hi]` serves as a natural sentinel for the left-to-right scan, and `a[lo]` serves as a natural sentinel for the right-to-left scan, eliminating bounds-checking in the inner loops.

4.2 Cutoff to Insertion Sort for Small Subarrays

Just as with mergesort, quicksort has relatively high overhead for very small subarrays. The solution is the same: switch to insertion sort when the subarray size falls below a cutoff (typically 5-15 elements).

```
quicksort(a, lo, hi):
    if hi - lo < CUTOFF:
        insertion_sort(a, lo, hi)
        return
    j = partition(a, lo, hi)
    quicksort(a, lo, j - 1)
    quicksort(a, j + 1, hi)
```

Sedgewick recommends a cutoff of about 5-15 elements. The optimal value depends on the machine and implementation. An alternative approach is to skip the recursion for small subarrays entirely and run a single pass of insertion sort over the whole array at the end (this works because elements in small subarrays are already close to their final positions).

4.3 Three-Way Partitioning (Dijkstra's Dutch National Flag)

Standard quicksort performs poorly on arrays with many duplicate keys. If all elements are equal, standard partitioning still partitions into two halves (one of which has size n-1), leading to O(n^2) behavior.

The solution is *three-way partitioning*, which partitions the array into three regions: elements less than the pivot, elements equal to the pivot, and elements greater than the pivot. Equal elements are never examined again after partitioning.

This problem is closely related to *Dijkstra's Dutch National Flag problem*, which asks: given an array of elements, each colored red, white, or blue, rearrange them so that all reds come first, then all whites, then all blues, using only swaps.

DEFINITION 4.1 -- THREE-WAY PARTITION

A three-way partition of array `a[lo..hi]` with respect to pivot value `v` rearranges the array such that for some indices `lt` and `gt` with `lo <= lt <= gt <= hi`: (1) `a[i] < v` for all `i` in `[lo, lt-1]`, (2) `a[i] = v` for all `i` in `[lt, gt]`, and (3) `a[i] > v` for all `i` in `[gt+1, hi]`.

The implementation maintains three pointers:

```
three_way_partition(a, lo, hi):
    lt = lo          // a[lo..lt-1] < v
    i  = lo          // a[lt..i-1]  = v
    gt = hi          // a[gt+1..hi] > v
    v  = a[lo]       // pivot value


    while i <= gt:
        if a[i] < v:
            swap(a[lt], a[i])
            lt++
            i++
        else if a[i] > v:
            swap(a[i], a[gt])
            gt--
            // do NOT increment i: swapped element needs examination
        else:    // a[i] == v
            i++

    // Now: a[lo..lt-1] < v, a[lt..gt] = v, a[gt+1..hi] > v
    quicksort(a, lo, lt - 1)
    quicksort(a, gt + 1, hi)
```

Trace on `[R, B, W, W, R, W, B, R, R, W, B, R]` (using R as pivot, treating B < R < W):

```
Initial: [R, B, W, W, R, W, B, R, R, W, B, R]
          lt                           gt
          i


Step 1: a[i]=R = v.  i++.
        [R, B, W, W, R, W, B, R, R, W, B, R]
         lt  i                           gt


Step 2: a[i]=B < v.  swap(a[lt],a[i]). lt++, i++.
        [B, R, W, W, R, W, B, R, R, W, B, R]
            lt  i                         gt


Step 3: a[i]=W > v.  swap(a[i],a[gt]). gt--.
        [B, R, R, W, R, W, B, R, R, W, B, W]
            lt  i                       gt


Step 4: a[i]=R = v.  i++.
        [B, R, R, W, R, W, B, R, R, W, B, W]
            lt     i                    gt


Step 5: a[i]=W > v.  swap(a[i],a[gt]). gt--.
        [B, R, R, B, R, W, B, R, R, W, W, W]
            lt     i                 gt


Step 6: a[i]=B < v.  swap(a[lt],a[i]). lt++, i++.
        [B, B, R, R, R, W, B, R, R, W, W, W]
               lt     i               gt


Step 7: a[i]=R = v.  i++.
        [B, B, R, R, R, W, B, R, R, W, W, W]
               lt        i            gt


Step 8: a[i]=W > v.  swap(a[i],a[gt]). gt--.
        [B, B, R, R, R, W, B, R, R, W, W, W]
        Wait -- a[gt]=W, swap gives:
        [B, B, R, R, R, W, B, R, R, W, W, W]
               lt        i        gt
        Actually gt was at 9 initially...
```

```
Let me redo this more carefully with indices:


Initial: [R, B, W, W, R, W, B, R, R, W, B, R]
          0  1  2  3  4  5  6  7  8  9 10 11
          lt=0, i=0, gt=11, v=R


i=0: a[0]=R=v. i=1.
i=1: a[1]=B<v. swap(a[0],a[1]). lt=1, i=2.  [B, R, W, W, R, W, B, R, R, W, B, R]
i=2: a[2]=W>v. swap(a[2],a[11]). gt=10.     [B, R, R, W, R, W, B, R, R, W, B, W]
i=2: a[2]=R=v. i=3.
i=3: a[3]=W>v. swap(a[3],a[10]). gt=9.       [B, R, R, B, R, W, B, R, R, W, W, W]
i=3: a[3]=B<v. swap(a[1],a[3]). lt=2, i=4.  [B, B, R, R, R, W, B, R, R, W, W, W]
i=4: a[4]=R=v. i=5.
i=5: a[5]=W>v. swap(a[5],a[9]). gt=8.         [B, B, R, R, R, W, B, R, R, W, W, W]
     Wait: a[9]=W, so swap(W,W)=no change. gt=8.
i=5: a[5]=W>v. swap(a[5],a[8]). gt=7.         [B, B, R, R, R, R, B, R, W, W, W, W]
i=5: a[5]=R=v. i=6.
i=6: a[6]=B<v. swap(a[2],a[6]). lt=3, i=7.   [B, B, B, R, R, R, R, R, W, W, W, W]
i=7: a[7]=R=v. i=8.
i=8 > gt=7: STOP.


Result: [B, B, B, R, R, R, R, R, W, W, W, W]
          < v         = v               > v
          [0..2]    [3..7]           [8..11]
```

The array is now partitioned into three regions. Recursion only processes the B-region and the W-region; the five R's are already in their final positions.

4.4 Handling Duplicate Keys Efficiently

The three-way partition is particularly important when there are many duplicate keys. Consider sorting an array where every element is the same value. Standard quicksort would make $O(n^2)$ comparisons. Three-way quicksort handles this in $O(n)$ because after the first partition, all elements are in the "equal" region and no recursion is needed.

More generally, when the keys come from a small alphabet or have many duplicates, three-way quicksort adapts to the structure of the input.

4.5 Entropy-Optimal Sorting

THEOREM 4.1

Three-way quicksort is *entropy-optimal*: it uses at most a constant factor more comparisons than the information-theoretic lower bound for any comparison-based sorting algorithm when there are duplicate keys.

Let $n_1, n_2, ..., n_k$ be the frequencies of the k distinct key values, with $n_1 + n_2 + ... + n_k = n$. The *Shannon entropy* of the key distribution is:

```
H = - sum_{i=1}^{k} (n_i / n) lg(n_i / n)
```

The information-theoretic lower bound on the number of comparisons is $n * H$. Three-way quicksort uses $O(n * H)$ comparisons on average.

When all keys are distinct, $H = \lg n$, and we recover the $O(n \log n)$ bound. When there are many duplicates, $H < \lg n$, and three-way quicksort is faster. In the extreme case of all identical keys, $H = 0$, and three-way quicksort runs in $O(n)$ time.

Lecture 4 of 12 -- Data Structures & Algorithms -- Based on Sedgewick's Algorithms, 4th Edition

5. Practical Considerations

5.1 Mergesort vs Quicksort in Practice

Despite their similar average-case complexity, quicksort is typically faster than mergesort in practice for sorting arrays of primitive types. There are several reasons:

Cache performance. Quicksort's partitioning accesses elements sequentially from both ends of the subarray, exhibiting excellent spatial locality. Mergesort's merge step alternates between two halves and an auxiliary array, leading to more cache misses, especially for large arrays that do not fit in L1 or L2 cache.

Overhead per comparison. Quicksort's inner loop is extremely tight: increment a pointer, compare, branch. Mergesort has additional overhead from copying to and from the auxiliary array.

Space. Quicksort uses O(log n) stack space. Mergesort requires O(n) auxiliary space. For very large arrays, this extra allocation can be expensive and may cause garbage collection pressure in managed languages.

Constant factors. Empirically, optimized quicksort (with median-of-three, cutoff to insertion sort, and three-way partitioning) uses approximately 1.1 n lg n comparisons on average, compared to mergesort's approximately n lg n comparisons. The slightly higher comparison count is more than offset by quicksort's lower overhead per comparison.

5.2 When to Use Which

Use quicksort when:
- Sorting arrays of primitive types
- Average-case performance matters most
- Memory is limited
- Input is not adversarially chosen (or you can randomize)

Use mergesort when:
- Worst-case O(n log n) guarantee is needed
- Stability is required (preserving order of equal elements)
- Sorting linked lists
- External sorting (data on disk)
- Sorting objects where comparison is expensive relative to data movement

5.3 System Sort Implementations

Real-world sorting libraries use sophisticated hybrid algorithms:

Language/Library	Algorithm for Objects	Algorithm for Primitives
Java (`Arrays.sort`)	TimSort (mergesort variant)	Dual-pivot quicksort
C++ (`std::sort`)	IntroSort (quicksort + heapsort)	IntroSort
Python (`sorted`)	TimSort	TimSort
C (`qsort`)	Implementation-defined (often quicksort)	Same
Go (`sort.Sort`)	Pattern-defeating quicksort (pdqsort)	Same

TimSort (by Tim Peters, 2002) is a hybrid of mergesort and insertion sort. It exploits existing order in the input by identifying already-sorted "runs" and merging them. It is guaranteed O(n log n) worst case, is stable, and performs well on partially sorted data. Java uses it for sorting objects because stability is required (the Java specification mandates that `Arrays.sort` for objects is stable).

IntroSort (by David Musser, 1997) is a hybrid of quicksort and heapsort. It starts with quicksort, but if the recursion depth exceeds 2 lg n, it switches to heapsort to guarantee O(n log n) worst case. C++ uses this because it provides the speed of quicksort with the worst-case guarantee of heapsort.

Dual-pivot quicksort (by Vladimir Yaroslavskiy, 2009) uses two pivots to partition the array into three parts. It is used in Java for primitive types because: (1) primitives do not need stability; (2) empirical testing showed it was faster than single-pivot quicksort due to fewer cache misses; (3) it uses O(log n) space.

5.4 Stability Considerations

DEFINITION 5.1 -- STABILITY

A sorting algorithm is *stable* if it preserves the relative order of elements with equal keys. That is, if element `a[i]` has the same key as element `a[j]` and `i < j` in the input, then `a[i]` appears before `a[j]` in the output.

Stability matters when sorting by multiple criteria. For example, sorting a list of students first by name, then by grade: a stable sort on grade preserves the alphabetical ordering among students with the same grade.

Algorithm	Stable?	Notes
Insertion sort	Yes	Never moves equal elements past each other
Selection sort	No	Swaps can move equal elements past each other
Shellsort	No	Long-distance exchanges disrupt order
Mergesort	Yes	Takes from left half on ties
Quicksort	No	Partitioning swaps disrupt order
Three-way quicksort	No	Same reason

Quicksort can be made stable, but the standard in-place implementations are not stable. Making quicksort stable requires O(n) extra space, at which point mergesort is generally a better choice.

5.5 Memory Allocation Patterns

The way algorithms interact with the memory hierarchy significantly affects their practical performance:

Quicksort accesses memory in a sequential scan pattern (two pointers moving from the ends toward the middle). This is very cache-friendly. The working set at each level of recursion is the subarray being partitioned, and elements are accessed in order.

Mergesort accesses three memory regions during the merge: the left half, the right half, and the auxiliary array. For large arrays, these three regions may map to different cache lines, causing more cache misses. The copy to the auxiliary array is an additional memory access pattern.

For external sorting (where data does not fit in main memory), mergesort is preferred because its access pattern is sequential -- it reads and writes data in large blocks, which is efficient for disk I/O. Quicksort's random access pattern is poorly suited to disk-based storage.

Lecture 4 of 12 -- Data Structures & Algorithms -- Based on Sedgewick's Algorithms, 4th Edition

6. Worked Examples and Practice Problems

Problem 1: Mergesort Trace

Problem: Trace top-down mergesort on the array `[5, 2, 8, 3, 1, 6, 4, 7]`.

Solution:

```
Initial: [5, 2, 8, 3, 1, 6, 4, 7]
          0  1  2  3  4  5  6  7

sort(0,7)
  sort(0,3)
    sort(0,1)
      sort(0,0) -> [5]
      sort(1,1) -> [2]
      merge(0,0,1): [5,2] -> [2,5]
    sort(2,3)
      sort(2,2) -> [8]
      sort(3,3) -> [3]
      merge(2,2,3): [8,3] -> [3,8]
    merge(0,1,3): [2,5] + [3,8] -> [2,3,5,8]
  sort(4,7)
    sort(4,5)
      sort(4,4) -> [1]
      sort(5,5) -> [6]
      merge(4,4,5): [1,6] -> [1,6]
    sort(6,7)
      sort(6,6) -> [4]
      sort(7,7) -> [7]
      merge(6,6,7): [4,7] -> [4,7]
    merge(4,5,7): [1,6] + [4,7] -> [1,4,6,7]
  merge(0,3,7): [2,3,5,8] + [1,4,6,7] -> [1,2,3,4,5,6,7,8]

Final: [1, 2, 3, 4, 5, 6, 7, 8]
```

The final merge `merge(0,3,7)` processes:

```
Left:  [2, 3, 5, 8]    Right: [1, 4, 6, 7]
         i                       j

1 < 2 -> take 1 from right. j++
2 < 4 -> take 2 from left.  i++
3 < 4 -> take 3 from left.  i++
4 < 5 -> take 4 from right. j++
5 < 6 -> take 5 from left.  i++
6 < 8 -> take 6 from right. j++
7 < 8 -> take 7 from right. j++
left exhausted: take 8.


Result: [1, 2, 3, 4, 5, 6, 7, 8]
Comparisons used in this merge: 7 (which is n-1)
```

Total comparisons: 1 + 1 + 3 + 1 + 1 + 3 + 7 = 17. The maximum for n=8 is n lg n = 8 * 3 = 24. Our input required fewer comparisons because some subarrays were partially ordered.

Problem 2: Bottom-Up Mergesort Trace

Problem: Trace bottom-up mergesort on `[6, 3, 8, 1, 5, 2, 7, 4]`.

Solution:

```
Initial: [6, 3, 8, 1, 5, 2, 7, 4]

Pass 1 (sz=1): merge adjacent pairs of size 1
  merge(0,0,1): [6,3] -> [3,6]
  merge(2,2,3): [8,1] -> [1,8]
  merge(4,4,5): [5,2] -> [2,5]
  merge(6,6,7): [7,4] -> [4,7]
  Array: [3, 6, 1, 8, 2, 5, 4, 7]

Pass 2 (sz=2): merge adjacent pairs of size 2
  merge(0,1,3): [3,6] + [1,8] -> [1,3,6,8]
  merge(4,5,7): [2,5] + [4,7] -> [2,4,5,7]
  Array: [1, 3, 6, 8, 2, 4, 5, 7]

Pass 3 (sz=4): merge the two halves
  merge(0,3,7): [1,3,6,8] + [2,4,5,7] -> [1,2,3,4,5,6,7,8]

Final: [1, 2, 3, 4, 5, 6, 7, 8]
```

The three passes correspond to the three levels of the mergesort recurrence tree, processed bottom-up instead of top-down.

Problem 3: Quicksort Partition

Problem: Perform Sedgewick's partition on `[7, 2, 9, 4, 3, 8, 6, 1]` using the first element (7) as the pivot. Show each swap.

Solution:

```
Initial: [7, 2, 9, 4, 3, 8, 6, 1]    pivot = 7
          lo                    hi      i = lo, j = hi+1

Scan i right from lo+1: a[1]=2<7, a[2]=9>=7. Stop. i=2.
Scan j left from hi: a[7]=1>7? No. Stop. j=7.
i<j: swap(a[2],a[7]): [7, 2, 1, 4, 3, 8, 6, 9]

Scan i right: a[3]=4<7, a[4]=3<7, a[5]=8>=7. Stop. i=5.
Scan j left: a[8]=9>7, yes. Wait, we need to be careful about indices.
  j starts from previous position. j=6, a[6]=6>7? No. Stop. j=6.
i<j: swap(a[5],a[6]): [7, 2, 1, 4, 3, 6, 8, 9]

Scan i right: a[6]=8>=7. Stop. i=6.
Scan j left: a[5]=6>7? No. Stop. j=5.
i>=j (6>=5): break.

swap(a[lo], a[j]) = swap(a[0], a[5]): [6, 2, 1, 4, 3, 7, 8, 9]

Return j=5.

Result: [6, 2, 1, 4, 3, | 7 | 8, 9]
          all < 7           pivot  all > 7
```

The pivot 7 is now in its final sorted position at index 5. Everything to its left is less than 7, everything to its right is greater than 7.

Problem 4: Proving Mergesort Correctness

Problem: Prove that mergesort correctly sorts any array of n comparable elements.

Solution:

We prove this by strong induction on n.

Base case (n <= 1): An array of 0 or 1 elements is trivially sorted. Mergesort returns immediately in this case.

Inductive step: Assume mergesort correctly sorts any array of fewer than n elements. We must show it correctly sorts an array of n elements (where n >= 2).

Mergesort computes `mid = lo + (hi - lo) / 2`. This gives two subarrays: `a[lo..mid]` of size `ceil(n/2)` and `a[mid+1..hi]` of size `floor(n/2)`. Both sizes are strictly less than n (since n >= 2, both are at least 1 and at most n-1).

By the inductive hypothesis, the recursive calls correctly sort `a[lo..mid]` and `a[mid+1..hi]`.

It remains to show that the merge operation correctly combines two sorted subarrays into one sorted array. The merge examines the smallest unprocessed element from each half and places the smaller one into the output. Since both halves are sorted, the smallest unprocessed element from each half is at the front. Therefore, at each step, the merge places the globally smallest remaining element, producing a sorted output.

By induction, mergesort correctly sorts any array of n elements. QED.

Problem 5: Three-Way Partition Trace

Problem: Perform Dijkstra's three-way partition on `[3, 1, 4, 1, 5, 3, 3, 2]` using the first element (3) as pivot.

Solution:

```
Initial: [3, 1, 4, 1, 5, 3, 3, 2]
         0  1  2  3  4  5  6  7
         lt=0, i=0, gt=7, v=3

i=0: a[0]=3=v.  i=1.
i=1: a[1]=1<v.  swap(a[0],a[1]). lt=1, i=2.  [1, 3, 4, 1, 5, 3, 3, 2]
i=2: a[2]=4>v.  swap(a[2],a[7]). gt=6.        [1, 3, 2, 1, 5, 3, 3, 4]
i=2: a[2]=2<v.  swap(a[1],a[2]). lt=2, i=3.  [1, 2, 3, 1, 5, 3, 3, 4]
i=3: a[3]=1<v.  swap(a[2],a[3]). lt=3, i=4.  [1, 2, 1, 3, 5, 3, 3, 4]
i=4: a[4]=5>v.  swap(a[4],a[6]). gt=5.        [1, 2, 1, 3, 3, 3, 5, 4]
i=4: a[4]=3=v.  i=5.
i=5: a[5]=3=v.  i=6.
i=6 > gt=5: STOP.

Result: [1, 2, 1, 3, 3, 3, 5, 4]
         < 3       = 3      > 3
        [0..2]    [3..5]   [6..7]
```

The three 3's are in their final positions. Recursion will sort `[1, 2, 1]` and `[5, 4]`.

Problem 6: Complexity Analysis

Problem: Suppose we modify mergesort to split the array into three equal parts instead of two (three-way mergesort). What is the asymptotic running time?

Solution:

The recurrence is:

```
T(n) = 3 T(n/3) + O(n)
```

The O(n) term accounts for merging three sorted subarrays (a three-way merge uses at most 2n comparisons since at each step we compare the minimums of three lists, requiring 2 comparisons).

By the Master Theorem (case 2, with a=3, b=3, so log_b(a) = 1, and f(n) = O(n) = O(n^{log_b(a)})):

```
T(n) = O(n log n)
```

More precisely, there are log_3(n) levels of recursion, each costing O(n), so:

```
T(n) = O(n log_3 n) = O(n * (ln n / ln 3)) = O(n log n)
```

The asymptotic complexity is the same as standard mergesort. However, the constant factor changes: we have n * log_3(n) = n * (log_2(n) / log_2(3)) ~ n * (log_2(n) / 1.585). Since each level requires roughly 2n comparisons (for the three-way merge), the total is approximately 2n * log_3(n) ~ 2n * ln(n)/ln(3) ~ 1.26 n lg n. This is slightly fewer comparisons than standard mergesort's worst case of n lg n per level? Actually, let us be more precise.

Standard two-way mergesort: at most n comparisons per level, lg_2(n) levels = n lg n comparisons.

Three-way mergesort: at most 2n comparisons per level (since three-way merge needs 2 comparisons per element in the worst case), log_3(n) levels = 2n log_3(n) = 2n * (lg n / lg 3) ~ 2n * (lg n / 1.585) ~ 1.26 n lg n.

So three-way mergesort uses approximately 26% more comparisons than two-way mergesort. It is not an improvement in terms of comparisons, but it may offer benefits in terms of cache behavior or parallelism in certain contexts.

Problem 7: Worst-Case Input for Quicksort

Problem: Construct a worst-case input of size 8 for quicksort with first-element pivot selection (no shuffle).

Solution:

The worst case occurs when every partition produces one subproblem of size 0 and one of size n-1. This happens when the pivot is always the minimum (or maximum) element.

For first-element pivot selection, a sorted array produces worst-case behavior:

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

Trace:

- partition(0,7): pivot=1. Everything else is >= 1. j=0 after scanning. Result: pivot stays at 0. Subarrays: [] and [2,3,4,5,6,7,8].
- partition(1,7): pivot=2. j=1. Subarrays: [] and [3,4,5,6,7,8].
- partition(2,7): pivot=3. j=2. Subarrays: [] and [4,5,6,7,8].
- ... and so on.

Number of comparisons: $(n-1) + (n-2) + ... + 1 = n(n-1)/2 = 28$ for n=8.

A reverse-sorted array `[8, 7, 6, 5, 4, 3, 2, 1]` also produces worst-case behavior. The first partition with pivot 8 places 8 at the end and leaves [7, 6, 5, 4, 3, 2, 1] to be sorted. Each subsequent partition similarly produces a maximally unbalanced split.

This is precisely why Sedgewick recommends shuffling the array before sorting: it makes any specific worst-case input occur with probability only $1/n!$, which is negligibly small.

Problem 8: Counting Comparisons in Merge

Problem: What are the minimum and maximum number of comparisons needed to merge two sorted arrays of sizes m and n?

Solution:

Minimum comparisons: min(m, n).

This occurs when every element of the smaller array is less than every element of the larger array. For example, merging [1, 2, 3] with [4, 5, 6, 7]. We compare 1 < 4, 2 < 4, 3 < 4 (3 comparisons, exhausting the left array), then copy 4, 5, 6, 7 without comparisons. In general, we exhaust the smaller array in min(m,n) comparisons.

Maximum comparisons: m + n - 1.

This occurs when elements alternate between the two arrays. For example, merging [1, 3, 5] with [2, 4, 6, 7]. The merge interleaves elements: 1 < 2, 2 < 3, 3 < 4, 4 < 5, 5 < 6, 6 (one array exhausted after 6 comparisons = 3 + 4 - 1 = 6).

Each comparison places exactly one element. The last element is placed without a comparison (when one array is exhausted, the remaining elements are copied). So the maximum is m + n - 1.

Summary

In this lecture we studied two of the most important sorting algorithms in computer science: mergesort and quicksort. Both are based on the divide-and-conquer paradigm, but they differ in where the work happens.

Mergesort divides the array into equal halves (trivial divide), recursively sorts each half, and merges the sorted halves (where the real work is). It guarantees O(n log n) worst-case performance, is stable, but requires O(n) extra space.

Quicksort partitions the array around a pivot element (where the real work is), then recursively sorts the two partitions (trivial combine). Its average-case performance is O(1.39 n lg n), and it sorts in place using only O(log n) stack space. However, its worst case is O(n^2), which we mitigate by shuffling the input.

We studied several improvements to quicksort: median-of-three pivot selection, cutoff to insertion sort for small subarrays, and Dijkstra's three-way partitioning for handling duplicate keys. Three-way quicksort is entropy-optimal, meaning it adapts to the distribution of keys in the input.

In practice, modern system sorts are hybrids: Java uses TimSort (a mergesort variant) for objects and dual-pivot quicksort for primitives; C++ uses IntroSort (quicksort with heapsort fallback); Python uses TimSort for everything.

The key takeaway is that the divide-and-conquer paradigm, combined with careful engineering, produces sorting algorithms that are both theoretically optimal and practically fast.

Algorithm	Time (worst)	Time (average)	Space	Stable	In-place
Mergesort	O(n log n)	O(n log n)	O(n)	Yes	No
Quicksort	O(n^2)	O(1.39 n lg n)	O(log n)	No	Yes
3-way quicksort	O(n^2)	O(n H)	O(log n)	No	Yes

In the next lecture, we will study priority queues and heapsort, completing our survey of comparison-based sorting algorithms.

References

1. Sedgewick, R. and Wayne, K. (2011). *Algorithms*, 4th Edition. Addison-Wesley Professional. Chapters 2.2 (Mergesort) and 2.3 (Quicksort).

2. Hoare, C. A. R. (1962). "Quicksort." *The Computer Journal*, 5(1), 10-16.

3. Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Edition. Addison-Wesley. Sections 5.2.4 (Sorting by Merging) and 5.2.2 (Sorting by Exchanging).

4. Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*, 3rd Edition. MIT Press. Chapters 2 (Merge Sort) and 7 (Quicksort).

5. Dijkstra, E. W. (1976). *A Discipline of Programming.* Prentice Hall. Chapter 14 (The Dutch National Flag Problem).

6. Musser, D. R. (1997). "Introspective Sorting and Selection Algorithms." *Software: Practice and Experience*, 27(8), 983-993.

7. Peters, T. (2002). "Timsort." Python source code, `Objects/listsort.txt`. Available at: https://github.com/python/cpython/blob/main/Objects/listsort.txt

8. Yaroslavskiy, V. (2009). "Dual-Pivot Quicksort." Technical report, available at: http://codeblab.com/wp-content/uploads/2009/09/DualPivotQuicksort.pdf

9. Bentley, J. L. and McIlroy, M. D. (1993). "Engineering a Sort Function." *Software: Practice and Experience*, 23(11), 1249-1265.

10. Sedgewick, R. (1978). "Implementing Quicksort Programs." *Communications of the ACM*, 21(10), 847-857.