PSL UNIVERSITÉ PARIS

Lab 1 — Detailed Solutions

Algorithms, Complexity & Divide-and-Conquer

TAOCP2 — Week 1 — **Solutions**

1 Warm-up: Complexity Quick-fire

1.1 Rank by growth rate

Solution

$$1 \;<\; \log n \;<\; \sqrt{n} \;<\; n \;<\; n \log n \;<\; n^2 \;<\; n^3 \;<\; 2^n \;<\; n!$$

Explanation

Key insight: polynomial functions (n^k) always grow slower than exponential functions (c^n for $c > 1$), which grow slower than factorial. Logarithmic functions grow slower than any polynomial. Also, $\sqrt{n} = n^{1/2}$, so it fits between $\log n$ and n.

1.2 True or false?

Solution

#	Statement	T/F	Justification
a	$n^2 = O(n^3)$	**T**	$n^2 \leqslant 1 \cdot n^3$ for all $n \geqslant 1$. Take $c = 1,\ n_0 = 1$.
b	$n^3 = O(n^2)$	**F**	$n^3/n^2 = n \to \infty$, so no constant c can bound the ratio.
c	$2^{n+1} = O(2^n)$	**T**	$2^{n+1} = 2 \cdot 2^n$. Take $c = 2,\ n_0 = 1$.
d	$2^{2n} = O(2^n)$	**F**	$2^{2n} = (2^n)^2 = 4^n$. The ratio $4^n/2^n = 2^n \to \infty$.
e	$\log_2 n = \Theta(\log_{10} n)$	**T**	$\log_2 n = \frac{\log_{10} n}{\log_{10} 2} \approx 3.32 \cdot \log_{10} n$. Constant factor.
f	$n^2 + 10^6 n = \Theta(n^2)$	**T**	For $n \geqslant 10^6$: $n^2 \leqslant n^2 + 10^6 n \leqslant 2n^2$. Take $c_1 = 1, c_2 = 2, n_0 = 10^6$.

Common Mistakes

- **(d):** Students often confuse 2^{n+1} and 2^{2n}. Stress that $2^{n+1} = 2 \cdot 2^n$ (constant factor) while $2^{2n} = (2^n)^2$ (squaring).
- **(e):** Some students think different log bases matter. Remind them: $\log_a n = \frac{\log_b n}{\log_b a}$, so all logarithmic bases differ by a constant factor.
- **(f):** The 10^6 coefficient may mislead students into thinking it dominates. Emphasize: constants don't matter asymptotically, only the growth rate.

1.3 What is the complexity?

Solution

(a) $\boldsymbol{\Theta(n^2)}$ — Two nested loops, each running n times. Total operations: $n \times n = n^2$.

(b) $\boldsymbol{\Theta(n^2)}$ — Inner loop runs $0 + 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = \Theta(n^2)$.

(c) $\boldsymbol{\Theta(\log n)}$ — Variable `i` starts at n and is halved each iteration. Number of iterations: $\lfloor \log_2 n \rfloor$.

(d) $\boldsymbol{\Theta(n \log n)}$ — Outer loop: n iterations. Inner loop: `j` doubles from 1 to n, so $\lfloor \log_2 n \rfloor$ iterations. Total: $n \cdot \log_2 n$.

(e) $\boldsymbol{\Theta(n)}$ — The outer loop runs for $i = 1, 2, 4, 8, \ldots$ up to n (so $\lfloor \log_2 n \rfloor + 1$ iterations). The inner loop does i work at each step. Total work:

$$\sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^k = 2^{\lfloor \log_2 n \rfloor + 1} - 1 \leqslant 2n - 1 = \Theta(n)$$

This is a geometric series where the last term dominates.

Common Mistakes

(e) is the trickiest. Many students will answer $\Theta(n \log n)$ by reasoning "outer loop is $\log n$, inner loop is up to n". The key is that the inner loop does *variable* work: 1, 2, 4, 8, ... The total is a geometric sum, not $n \times \log n$.

2 Proving Asymptotic Bounds

2.1 Prove that $3n^2 + 5n + 2 = O(n^2)$

Solution

We need to find $c > 0$ and $n_0 > 0$ such that $3n^2 + 5n + 2 \leqslant c \cdot n^2$ for all $n \geqslant n_0$.
For $n \geqslant 1$, we have $n \leqslant n^2$ and $1 \leqslant n^2$, so:

$$3n^2 + 5n + 2 \leqslant 3n^2 + 5n^2 + 2n^2 = 10n^2$$

Therefore, with $c = 10$ and $n_0 = 1$, we have $3n^2 + 5n + 2 \leqslant 10n^2$ for all $n \geqslant 1$.

Alternative (tighter): For $n \geqslant 5$: $5n \leqslant n^2$ and $2 \leqslant n^2$, so $3n^2 + 5n + 2 \leqslant 3n^2 + n^2 + n^2 = 5n^2$. Thus $c = 5, n_0 = 5$.

Grading Notes

Accept any valid pair (c, n_0). The most common choice is $c = 10, n_0 = 1$. Students should show the chain of inequalities clearly. Deduct if they just state "drop the lower-order terms" without formal justification.

2.2 Prove that $n^2 \neq O(n)$

Solution

Proof by contradiction.
Assume $n^2 = O(n)$. Then $\exists\, c > 0,\ n_0 > 0$ such that:

$$n^2 \leqslant c \cdot n \quad \text{for all } n \geqslant n_0$$

Dividing both sides by $n > 0$:

$$n \leqslant c \quad \text{for all } n \geqslant n_0$$

But this is a contradiction, since n can be made arbitrarily large (take $n = \lceil c \rceil + 1$, then $n > c$).
Therefore, $n^2 \neq O(n)$. □

2.3 Prove that $n \log n = O(n^2)$ and $n \log n = \Omega(n)$

Solution

Part 1: $n \log n = O(n^2)$.
For $n \geqslant 1$: $\log n \leqslant n$ (since $e^x \geqslant x$ for all x, or simply by checking that $\log n$ grows slower than n). Therefore:

$$n \log n \leqslant n \cdot n = n^2$$

Take $c = 1,\ n_0 = 1$. ✓

Part 2: $n \log n = \Omega(n)$.
For $n \geqslant 2$: $\log n \geqslant 1$ (using log base 2, or $\log_e n \geqslant 1$ for $n \geqslant 3$). Therefore:

$$n \log n \geqslant n \cdot 1 = n$$

Take $c = 1,\ n_0 = 2$ (base 2) or $n_0 = 3$ (natural log). ✓

Conclusion: We have shown $n \log n = O(n^2)$ and $n \log n = \Omega(n)$. However, this does **not** mean $n \log n = \Theta(n)$ or $\Theta(n^2)$. The tightest we can say is:

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

We proved it sits *between* n and n^2, but n is not a tight lower bound and n^2 is not a tight upper bound.

2.4 (Bonus) Prove that $\log(n!) = \Theta(n \log n)$

Solution

Upper bound: $n! = 1 \cdot 2 \cdot 3 \cdots n \leqslant n^n$. Therefore:

$$\log(n!) \leqslant \log(n^n) = n \log n$$

So $\log(n!) = O(n \log n)$ with $c = 1$.

Lower bound: $n! = 1 \cdot 2 \cdots \lfloor n/2 \rfloor \cdots n \geqslant \left(\frac{n}{2}\right)^{n/2}$ (the last $n/2$ factors are each $\geqslant n/2$). Therefore:

$$\log(n!) \geqslant \frac{n}{2} \log\left(\frac{n}{2}\right) = \frac{n}{2}(\log n - \log 2) = \frac{n \log n}{2} - \frac{n \log 2}{2}$$

For large enough n, the first term dominates: $\log(n!) \geqslant \frac{n \log n}{4}$ for $n \geqslant 4$ (e.g.).
So $\log(n!) = \Omega(n \log n)$ with $c = 1/4$.

Conclusion: $\log(n!) = \Theta(n \log n)$. □

Note: Stirling's approximation gives $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, *so* $\log(n!) \approx n \log n - n \log e + \frac{1}{2}\log(2\pi n) \sim n \log n$.

3 Master Theorem

3.1 Apply the master theorem

Solution

#	Recurrence	a	b	$n^{\log_b a}$	Case	Solution
1	$T(n) = 4T(n/2) + n$	4	2	n^2	1	$\Theta(n^2)$
2	$T(n) = 2T(n/2) + n^2$	2	2	n^1	3	$\Theta(n^2)$
3	$T(n) = 3T(n/4) + n \log n$	3	4	$n^{0.79\ldots}$	3	$\Theta(n \log n)$
4	$T(n) = 2T(n/2) + n \log n$	2	2	n^1	N/A	$\Theta(n \log^2 n)$
5	$T(n) = 9T(n/3) + n^2$	9	3	n^2	2	$\Theta(n^2 \log n)$
6	$T(n) = T(n/2) + 1$	1	2	$n^0 = 1$	2	$\Theta(\log n)$

Explanation

Detailed justifications:
#1: $a = 4, b = 2 \Rightarrow \log_2 4 = 2$. We have $f(n) = n = O(n^{2-1})$, so Case 1 applies with $\varepsilon = 1$. Solution: $\Theta(n^2)$.
#2: $a = 2, b = 2 \Rightarrow \log_2 2 = 1$. We have $f(n) = n^2 = \Omega(n^{1+1})$, so Case 3 with $\varepsilon = 1$. Check regularity: $2f(n/2) = 2(n/2)^2 = n^2/2 \leqslant (1/2) \cdot n^2$. ✓ Solution: $\Theta(n^2)$.
#3: $a = 3, b = 4 \Rightarrow \log_4 3 = \ln 3 / \ln 4 \approx 0.792$. We have $f(n) = n \log n = \Omega(n^{0.792+\varepsilon})$ for, say, $\varepsilon = 0.1$ (since $n^{0.892}$ grows slower than n). Case 3. Regularity: $3f(n/4) = 3(n/4)\log(n/4) \leqslant (3/4)n \log n$ for large n. ✓
#4: $a = 2, b = 2 \Rightarrow \log_2 2 = 1$. $f(n) = n \log n$. For Case 3, we would need $f(n) = \Omega(n^{1+\varepsilon})$ for some $\varepsilon > 0$. But $n \log n / n = \log n$, which grows slower than n^ε for any $\varepsilon > 0$. **The master theorem does not apply.** Use the recursion tree (see below).
#5: $a = 9, b = 3 \Rightarrow \log_3 9 = 2$. $f(n) = n^2 = \Theta(n^2)$. Case 2. Solution: $\Theta(n^2 \log n)$.
#6: $a = 1, b = 2 \Rightarrow \log_2 1 = 0$. $f(n) = 1 = \Theta(n^0) = \Theta(1)$. Case 2. Solution: $\Theta(n^0 \cdot \log n) = \Theta(\log n)$.

Common Mistakes

- **#4:** Most common error — students apply Case 3 or Case 2. Stress that Case 2 requires $f(n) = \Theta(n^{\log_b a})$ exactly (here n vs $n \log n$), and Case 3 requires a *polynomial* gap.
- **#2/#3:** Forgetting to check the regularity condition for Case 3.
- **#6:** Some students are confused by $a = 1$. Remind them that $\log_2 1 = 0$ and $n^0 = 1$.

3.2 Recursion tree for #4

Solution

$T(n) = 2T(n/2) + n \log n$.
(a) Number of levels: The problem size halves at each level. Starting from n, after k levels: $n/2^k$. The recursion bottoms out when $n/2^k = 1$, i.e., $k = \log_2 n$.
So there are $\log_2 n + 1$ levels (levels $0, 1, \ldots, \log_2 n$).

(b) Work at level k:
At level k, there are 2^k subproblems, each of size $n/2^k$. The work per subproblem is:

$$\frac{n}{2^k} \cdot \log\left(\frac{n}{2^k}\right) = \frac{n}{2^k}(\log n - k)$$

Total work at level k:

$$2^k \cdot \frac{n}{2^k}(\log n - k) = n(\log n - k)$$

(c) Total work:

$$\begin{aligned} T(n) &= \sum_{k=0}^{\log_2 n} n(\log n - k) = n \sum_{k=0}^{\log_2 n} (\log n - k) \\ &= n \sum_{j=0}^{\log_2 n} j \quad \text{(substituting } j = \log n - k\text{)} \\ &= n \cdot \frac{(\log n)(\log n + 1)}{2} \\ &= \Theta(n \log^2 n) \end{aligned}$$

Conclusion: $T(n) = \Theta(n \log^2 n)$. □

4 Implement Merge Sort

4.1 Reference implementation

Solution

```python
def merge(left: list, right: list) -> list:
    """Merge two sorted lists into a single sorted list."""
    result = []
    i, j = 0, 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    # Append remaining elements
    result.extend(left[i:])
    result.extend(right[j:])
    return result

def merge_sort(arr: list) -> list:
    """Sort a list using the merge sort algorithm."""
    if len(arr) <= 1:
        return arr.copy()   # or arr[:] -- return a new list
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)
```

Common Mistakes

- Forgetting to copy the input in the base case (returning `arr` directly means mutations propagate).
- Using `<` instead of `<=` in the merge comparison — this makes the sort **unstable** (not incorrect, but worth discussing).
- Using `pop(0)` instead of index tracking — this is $O(n)$ per pop, making merge $O(n^2)$.
- Forgetting to append the remaining elements after the main loop.

4.2 Tests

Solution

All assertions pass with the reference implementation:

```python
assert merge_sort([]) == []
assert merge_sort([42]) == [42]
assert merge_sort([1, 2, 3, 4, 5]) == [1, 2, 3, 4, 5]
assert merge_sort([5, 4, 3, 2, 1]) == [1, 2, 3, 4, 5]
assert merge_sort([38, 27, 43, 3, 9, 82, 10]) == [3, 9, 10, 27, 38, 43, 82]
assert merge_sort([3, 3, 1, 1, 2, 2]) == [1, 1, 2, 2, 3, 3]
print("All tests passed!")
```

4.3 Count comparisons

Solution

```python
def merge_count(left: list, right: list) -> tuple[list, int]:
    """Merge two sorted lists, counting comparisons."""
    result = []
    i, j, comparisons = 0, 0, 0
    while i < len(left) and j < len(right):
        comparisons += 1
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result, comparisons

def merge_sort_count(arr: list) -> tuple[list, int]:
    """Return (sorted_list, num_comparisons)."""
    if len(arr) <= 1:
        return arr[:], 0
    mid = len(arr) // 2
    left, left_cmp = merge_sort_count(arr[:mid])
    right, right_cmp = merge_sort_count(arr[mid:])
    merged, merge_cmp = merge_count(left, right)
    return merged, left_cmp + right_cmp + merge_cmp
```

Expected output (approximately):

```
n=   100: comparisons=     541, n*floor(log2(n))=      600, ratio=0.90
n=  1000: comparisons=    8701, n*floor(log2(n))=     9000, ratio=0.97
n= 10000: comparisons=  120474, n*floor(log2(n))=   130000, ratio=0.93
```

The ratio is consistently close to 1, confirming the $\Theta(n \log n)$ complexity.

Explanation

The exact number of comparisons depends on the input, but for random data:

- **Best case:** $\frac{n}{2} \log_2 n$ (when one half is always exhausted first in each merge)
- **Worst case:** $n \log_2 n - n + 1$ (when elements alternate perfectly between halves)
- **Average case:** $\approx n \log_2 n - 1.26n$ (information-theoretic analysis)

5 Divide-and-Conquer Challenges

5.1 Fast Exponentiation

Solution

```python
def fast_power(base: float, exp: int) -> float:
    """Compute base^exp using O(log exp) multiplications."""
    if exp == 0:
        return 1
    if exp < 0:
        return 1 / fast_power(base, -exp)
    if exp % 2 == 0:
        half = fast_power(base, exp // 2)
        return half * half
    else:
        return base * fast_power(base, exp - 1)
```

Answers to the questions:

1. **Recurrence:** $T(n) = T(n/2) + O(1)$ (each step either halves n or reduces by 1 then halves).
2. **Master theorem:** $a = 1, b = 2, \log_2 1 = 0$. $f(n) = O(1) = O(n^0)$. Case 2: $T(n) = \Theta(\log n)$.
3. **Comparison:** A naïve loop (`result *= base` repeated n times) performs n multiplications ($O(n)$). Fast power performs $O(\log n)$ multiplications. For $n = 1000$: 1000 vs ≈10 multiplications.
   Note: Python's built-in `**` operator already uses fast exponentiation internally, so the comparison is with an explicit naïve loop, not with `**`.
4. **Verification:**

```python
assert fast_power(2, 100) == 2**100
assert fast_power(3, 50) == 3**50
assert fast_power(2, 0) == 1
assert fast_power(5, 1) == 5
print("All tests passed!")
```

Explanation

The iterative version (binary exponentiation) is also acceptable:

```python
def fast_power_iter(base, exp):
    result = 1
    while exp > 0:
        if exp % 2 == 1:
            result *= base
        base *= base
        exp //= 2
    return result
```

This avoids recursion overhead and is the preferred form in competitive programming.

5.2 Counting Inversions

Solution

Naïve approach — $O(n^2)$:

```python
def count_inversions_naive(arr: list) -> int:
    """Count inversions using brute force."""
    count = 0
    n = len(arr)
    for i in range(n):
        for j in range(i + 1, n):
            if arr[i] > arr[j]:
                count += 1
    return count
```

D&C approach — $O(n \log n)$:

```python
def count_inversions(arr: list) -> tuple[list, int]:
    """Return (sorted_array, inversion_count) using modified merge sort."""
    if len(arr) <= 1:
        return arr[:], 0

    mid = len(arr) // 2
    left, left_inv = count_inversions(arr[:mid])
    right, right_inv = count_inversions(arr[mid:])

    # Merge and count split inversions
    merged = []
    inversions = left_inv + right_inv
    i, j = 0, 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
            # All remaining elements in left[i:] form inversions
            # with right[j-1]
            inversions += len(left) - i

    merged.extend(left[i:])
    merged.extend(right[j:])
    return merged, inversions
```

Verification:

```python
# Sorted array: 0 inversions
assert count_inversions([1, 2, 3, 4, 5]) == ([1, 2, 3, 4, 5], 0)

# Reverse sorted: n*(n-1)/2 = 5*4/2 = 10 inversions
assert count_inversions([5, 4, 3, 2, 1]) == ([1, 2, 3, 4, 5], 10)

# Example from the problem statement
assert count_inversions([2, 4, 1, 3, 5]) == ([1, 2, 3, 4, 5], 3)

# Cross-check with naive on random data
import random
arr = [random.randint(0, 1000) for _ in range(1000)]
_, dc_count = count_inversions(arr)
naive_count = count_inversions_naive(arr)
assert dc_count == naive_count
print(f"Random array: {dc_count} inversions (verified)")
```

Explanation

Key insight: During the merge step, when we pick an element from the right half (say `right[j]`), it means `right[j]` is smaller than all remaining elements in the left half (`left[i]`, `left[i+1]`, ...). Each of these pairs is a "split inversion" — an inversion where one element is in the left half and the other in the right half.
The number of such inversions is exactly `len(left) - i`.

Complexity: Same as merge sort: $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$.

6 Benchmarking — Theory vs Practice

Solution

```python
import time, random
import matplotlib.pyplot as plt

def insertion_sort(arr):
    a = arr.copy()
    for i in range(1, len(a)):
        key = a[i]
        j = i - 1
        while j >= 0 and a[j] > key:
            a[j + 1] = a[j]
            j -= 1
        a[j + 1] = key
    return a

sizes = [100, 500, 1000, 2000, 5000, 10000]
times_insertion = []
times_merge = []
times_builtin = []

for n in sizes:
    arr = [random.randint(0, 10**6) for _ in range(n)]

    start = time.perf_counter()
    insertion_sort(arr)
    times_insertion.append(time.perf_counter() - start)

    start = time.perf_counter()
    merge_sort(arr)
    times_merge.append(time.perf_counter() - start)

    start = time.perf_counter()
    sorted(arr)
    times_builtin.append(time.perf_counter() - start)

    print(f"n={n:>6}: insertion={times_insertion[-1]:.4f}s, "
          f"merge={times_merge[-1]:.4f}s, "
          f"sorted={times_builtin[-1]:.6f}s")
```

Typical output (values vary by machine):

```
n=   100: insertion=0.0002s, merge=0.0001s, sorted=0.000005s
n=   500: insertion=0.0042s, merge=0.0007s, sorted=0.000029s
n=  1000: insertion=0.0170s, merge=0.0015s, sorted=0.000063s
n=  2000: insertion=0.0670s, merge=0.0033s, sorted=0.000139s
n=  5000: insertion=0.4200s, merge=0.0092s, sorted=0.000390s
n= 10000: insertion=1.6800s, merge=0.0200s, sorted=0.000850s
```

Plot code:

```python
plt.figure(figsize=(10, 6))
plt.plot(sizes, times_insertion, 'ro-', label='Insertion Sort $O(n^2)$')
plt.plot(sizes, times_merge, 'bs-', label='Merge Sort $O(n \\log n)$')
plt.plot(sizes, times_builtin, 'g^-', label='Built-in sorted() (Timsort)')
plt.xlabel('Input size n')
plt.ylabel('Time (seconds)')
plt.title('Sorting Algorithm Benchmarks')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig('benchmark.png', dpi=150)
plt.show()
```

Explanation

Answers to the questions:

1. **Crossover point:** Merge sort typically starts winning around $n \approx 50$–200 in Python. For very small arrays, insertion sort is faster due to lower overhead (no recursion, no list creation).
2. **Ratio at $n = 10\,000$:** Typically around 80–100× faster for merge sort. The theoretical ratio is $n/\log_2 n \approx 10000/13.3 \approx 750$, but constants matter in practice: merge sort has higher constant factors (memory allocation, function calls) than insertion sort.
3. **Built-in `sorted()`:** Python's Timsort is 20–50× faster than our merge sort because:
   - It's implemented in C, not Python.
   - It exploits existing order in the data (natural runs).
   - It uses insertion sort for small subarrays.
   - It avoids creating new lists (in-place merge).

Bonus Exercises

Bonus 1 — Binary Search Variants

Solution

```python
def binary_search(arr: list, target) -> int:
    """Return the index of target in sorted arr, or -1 if not found."""
    lo, hi = 0, len(arr) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            lo = mid + 1
        else:
            hi = mid - 1
    return -1

def lower_bound(arr: list, target) -> int:
    """Return the index of the first element >= target."""
    lo, hi = 0, len(arr)
    while lo < hi:
        mid = (lo + hi) // 2
        if arr[mid] < target:
            lo = mid + 1
        else:
            hi = mid
    return lo

def upper_bound(arr: list, target) -> int:
    """Return the index of the first element > target."""
    lo, hi = 0, len(arr)
    while lo < hi:
        mid = (lo + hi) // 2
        if arr[mid] <= target:
            lo = mid + 1
        else:
            hi = mid
    return lo

def count_occurrences(arr: list, target) -> int:
    """Count occurrences of target in sorted arr in O(log n)."""
    return upper_bound(arr, target) - lower_bound(arr, target)

# Tests
arr = [1, 2, 2, 2, 3, 4, 5, 5, 6]
assert lower_bound(arr, 2) == 1
assert upper_bound(arr, 2) == 4
assert count_occurrences(arr, 2) == 3
assert count_occurrences(arr, 5) == 2
assert count_occurrences(arr, 7) == 0
print("All binary search tests passed!")
```

Explanation

Key difference between the three variants:

- `binary_search`: uses `lo <= hi` and checks equality at `mid`.
- `lower_bound`: uses `lo < hi`, searches for the *leftmost* position where we could insert `target`.
- `upper_bound`: same structure, but `<=` instead of `<` in the comparison — finds the *rightmost* insertion point.

These correspond exactly to C++'s `std::lower_bound` and `std::upper_bound`, and to Python's `bisect.bisect_left` and `bisect.bisect_right`.

Bonus 2 — Merge Sort in C++

Solution

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <random>

std::vector<int> merge(const std::vector<int>& left,
                       const std::vector<int>& right) {
    std::vector<int> result;
    result.reserve(left.size() + right.size());
    size_t i = 0, j = 0;
    while (i < left.size() && j < right.size()) {
        if (left[i] <= right[j]) {
            result.push_back(left[i++]);
        } else {
            result.push_back(right[j++]);
        }
    }
    while (i < left.size()) result.push_back(left[i++]);
    while (j < right.size()) result.push_back(right[j++]);
    return result;
}

std::vector<int> merge_sort(const std::vector<int>& arr) {
    if (arr.size() <= 1) return arr;
    size_t mid = arr.size() / 2;
    std::vector<int> left(arr.begin(), arr.begin() + mid);
    std::vector<int> right(arr.begin() + mid, arr.end());
    return merge(merge_sort(left), merge_sort(right));
}

int main() {
    std::mt19937 rng(42);
    for (int n : {1000, 10000, 100000, 1000000}) {
        std::vector<int> arr(n);
        std::uniform_int_distribution<int> dist(0, 1000000);
        for (auto& x : arr) x = dist(rng);

        // Our merge sort
        auto a1 = arr;
        auto t0 = std::chrono::high_resolution_clock::now();
        auto sorted1 = merge_sort(a1);
        auto t1 = std::chrono::high_resolution_clock::now();
        double ms_merge = std::chrono::duration<double,
                          std::milli>(t1 - t0).count();

        // std::sort (introsort)
        auto a2 = arr;
        t0 = std::chrono::high_resolution_clock::now();
        std::sort(a2.begin(), a2.end());
        t1 = std::chrono::high_resolution_clock::now();
        double ms_std = std::chrono::duration<double,
                        std::milli>(t1 - t0).count();

        // std::stable_sort (merge sort variant)
        auto a3 = arr;
        t0 = std::chrono::high_resolution_clock::now();
        std::stable_sort(a3.begin(), a3.end());
        t1 = std::chrono::high_resolution_clock::now();
        double ms_stable = std::chrono::duration<double,
                           std::milli>(t1 - t0).count();
```

Explanation

Key observations:

- Our C++ merge sort is already 50–100× faster than the Python version for the same n, thanks to compiled code and no interpreter overhead.
- `std::sort` (introsort = quicksort + heapsort fallback) is $\approx$ 3–4× faster than our merge sort because it sorts in-place (better cache locality, no memory allocation during merges).
- `std::stable_sort` (a highly optimized merge sort) is slightly slower than `std::sort` but preserves the relative order of equal elements (stability). It is still $\approx$ 3× faster than our naïve merge sort due to optimizations (in-place merging with temporary buffers, insertion sort for small subarrays).
- All three are $O(n \log n)$; the differences are in constant factors.

Bonus 3 — Karatsuba Multiplication

Solution

```python
def karatsuba(x: int, y: int) -> int:
    """Multiply two integers using Karatsuba's algorithm."""
    # Base case: single-digit multiplication
    if x < 10 or y < 10:
        return x * y

    # Determine the size (number of digits)
    n = max(len(str(abs(x))), len(str(abs(y))))
    half = n // 2

    # Split x = a * 10^half + b, y = c * 10^half + d
    power = 10 ** half
    a, b = divmod(x, power)
    c, d = divmod(y, power)

    # Three recursive multiplications (instead of four)
    ac = karatsuba(a, c)
    bd = karatsuba(b, d)
    ad_bc = karatsuba(a + b, c + d) - ac - bd

    return ac * (10 ** (2 * half)) + ad_bc * power + bd

# Tests
assert karatsuba(1234, 5678) == 1234 * 5678
assert karatsuba(12345678, 87654321) == 12345678 * 87654321
assert karatsuba(0, 12345) == 0
assert karatsuba(999, 999) == 999 * 999
print("All Karatsuba tests passed!")
```

Complexity: The recurrence is $T(n) = 3T(n/2) + O(n)$, where n is the number of digits. Applying the master theorem: $a = 3, b = 2, \log_2 3 \approx 1.585$. Since $f(n) = O(n) = O(n^{1.585-0.585})$, this is Case 1: $T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$.
This is better than the naïve $O(n^2)$ digit-by-digit multiplication. For very large numbers (hundreds of digits), the speedup is significant.

End of Solutions — The Art of Computer Programming 2 — Lab 1