# Part I

# Problems

Chapter

# 1 Primitive Types

*Representation is the essence of programming.*

— "*The Mythical Man Month*,"
F. P. Brooks, 1975

A program updates variables in memory according to its instructions. Variables come in types—a type is a classification of data that spells out possible values for that type and the operations that can be performed on it.

A type can be provided by the language or defined by the programmer. Many languages provide types for Boolean, integer, character and floating point data. Often, there are multiple integer and floating point types, depending on signedness and precision. The width of these types is the number of bits of storage a corresponding variable takes in memory. For example, most implementations of C++ use 32 or 64 bits for an `int`. In Java an `int` is always 32 bits.

## 1.1 Computing the parity of a word

The parity of a binary word is 1 if the number of 1s in the word is odd; otherwise, it is 0. For example, the parity of 1011 is 1, and the parity of 10001000 is 0. Parity checks are used to detect single bit errors in data storage and communication. It is fairly straightforward to write code that computes the parity of a single 64-bit word.

How would you compute the parity of a very large number of 64-bit words?

*Hint:* Use a lookup table, but don't use $2^{64}$ entries!

**Solution:** The brute-force algorithm iteratively tests the value of each bit while tracking the number of 1s seen so far. Since we only care if the number of 1s is even or odd, we can store the number modulo 2.

```
public static short parity(long x) {
  short result = 0;
  while (x != 0) {
    result ^= (x & 1);
    x >>= 1;
  }
  return result;
}
```

The time complexity is $O(n)$, where $n$ is the word size.

On Page **??** we showed how to erase the lowest set bit in a word in a single operation. This can be used to improve performance in the best- and average-cases.

```
public static short parity(long x) {
  short result = 0;
  while (x != 0) {
    result ^= 1;
    x &= (x - 1); // Drops the lowest set bit of x.
  }
  return result;
}
```

Let $k$ be the number of bits set to 1 in a particular word. (For example, for 10001010, $k = 3$.) Then time complexity of the algorithm above is $O(k)$.

The problem statement refers to computing the parity for a very large number of words. When you have to perform a large number of parity computations, and, more generally, any kind of bit fiddling computations, two keys to performance are processing multiple bits at a time and caching results in an array-based lookup table.

First we demonstrate caching. Clearly, we cannot cache the parity of every 64-bit integer—we would need $2^{64}$ bits of storage, which is of the order of ten trillion exabytes. However, when computing the parity of a collection of bits, it does not matter how we group those bits, i.e., the computation is associative. Therefore, we can compute the parity of a 64-bit integer by grouping its bits into four nonoverlapping 16 bit subwords, computing the parity of each subwords, and then computing the parity of these four subresults. We choose 16 since $2^{16} = 65536$ is relatively small, which makes it feasible to cache the parity of all 16-bit words using an array. Furthermore, since 16 evenly divides 64, the code is simpler than if we were, for example, to use 10 bit subwords.

We illustrate the approach with a lookup table for 2-bit words. The cache is $\langle 0, 1, 1, 0 \rangle$—these are the parities of (00), (01), (10), (11), respectively. To compute the parity of (11001010) we would compute the parities of (11), (00), (10), (10). By table lookup we see these are $0, 0, 1, 1$, respectively, so the final result is the parity of $0, 0, 1, 1$ which is 0.

To lookup the parity of the first two bits in (11101010), we right shift by 6, to get (00000011), and use this as an index into the cache. To lookup the parity of the next two bits, i.e., (10), we right shift by 4, to get (10) in the two least-significant bit places. The right shift does not remove the leading (11)—it results in (00001110). We cannot index the cache with this, it leads to an out-of-bounds access. To get the last two bits after the right shift by 4, we bitwise-AND (00001110) with (00000011) (this is the "mask" used to extract the last 2 bits). The result is (00000010). Similar masking is needed for the two other 2-bit lookups.

```
public static short parity(long x) {
  final int WORD_SIZE = 16;
  final int BIT_MASK = 0xFFFF;
  return (short) (
      precomputedParity[(int)((x >>> (3 * WORD_SIZE)) & BIT_MASK)]
      ^ precomputedParity[(int)((x >>> (2 * WORD_SIZE)) & BIT_MASK)]
```

```
      ^ precomputedParity[(int)((x >>> WORD_SIZE) & BIT_MASK)]
      ^ precomputedParity[(int)(x & BIT_MASK)]);
}
```

The time complexity is a function of the size of the keys used to index the lookup table. Let $L$ be the width of the words for which we cache the results, and $n$ the word size. Since there are $n/L$ terms, the time complexity is $O(n/L)$, assuming word-level operations, such as shifting, take $O(1)$ time. (This does not include the time for initialization of the lookup table.)

The XOR of two bits is 0 if both bits are 0 or both bits are 1; otherwise it is 1. XOR has the property of being associative (as previously described), as well as commutative, i.e., the order in which we perform the XORs does not change the result. The XOR of a group of bits is its parity. We can exploit this fact to use the CPU's word-level XOR instruction to process multiple bits at a time.

For example, the parity of $\langle b_{63}, b_{62}, \ldots, b_3, b_2, b_1, b_0 \rangle$ equals the parity of the XOR of $\langle b_{63}, b_{62}, \ldots, b_{32} \rangle$ and $\langle b_{31}, b_{30}, \ldots, b_0 \rangle$. The XOR of these two 32-bit values can be computed with a single shift and a single 32-bit XOR instruction. We repeat the same operation on 32-, 16-, 8-, 4-, 2-, and 1-bit operands to get the final result. Note that the leading bits are not meaningful, and we have to explicitly extract the result from the least-significant bit.

We illustrate the approach with an 8-bit word. The parity of (11010111) is the same as the parity of (1101) XORed with (0111), i.e., of (1010). This in turn is the same as the parity of (10) XORed with (10), i.e., of (00). The final result is the XOR of (0) with (0), i.e., 0. Note that the first XOR yields (11011010), and only the last 4 bits are relevant going forward. The second XOR yields (11101100), and only the last 2 bits are relevant. The third XOR yields (10011010). The last bit is the result, and to extract it we have to bitwise-AND with (00000001).

```
public static short parity(long x) {
  x ^= x >>> 32;
  x ^= x >>> 16;
  x ^= x >>> 8;
  x ^= x >>> 4;
  x ^= x >>> 2;
  x ^= x >>> 1;
  return (short)(x & 0x1);
}
```

The time complexity is $O(\log n)$, where $n$ is the word size.

Note that we could have combined caching with word-level operations, e.g., by doing a lookup once we get to 16 bits.

Chapter

# 2 Arrays

*The machine can alter the scanned symbol and its behavior is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behavior of the machine.*

— "Intelligent Machinery,"
A. M. Turing, 1948

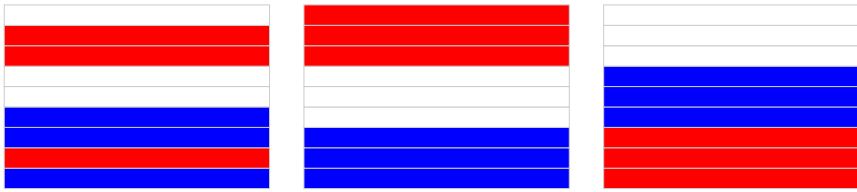
The simplest data structure is the *array*, which is a contiguous block of memory. It is usually used to represent sequences. Given an array $A$, $A[i]$ denotes the $(i+1)$th object stored in the array. Retrieving and updating $A[i]$ takes $O(1)$ time. Insertion into a full array can be handled by resizing, i.e., allocating a new array with additional memory and copying over the entries from the original array. This increases the worst-case time of insertion, but if the new array has, for example, a constant factor larger than the original array, the average time for insertion is constant since resizing is infrequent. Deleting an element from an array entails moving all successive elements one over to the left to fill the vacated space. For example, if the array is $\langle 2, 3, 5, 7, 9, 11, 13, 17\rangle$, then deleting the element at index 4 results in the array $\langle 2, 3, 5, 7, 11, 13, 17, 0\rangle$. (We do not care about the last value.) The time complexity to delete the element at index $i$ from an array of length $n$ is $O(n - i)$.

## 2.1 The Dutch national flag problem

The quicksort algorithm for sorting arrays proceeds recursively—it selects an element (the "pivot"), reorders the array to make all the elements less than or equal to the pivot appear first, followed by all the elements greater than the pivot. The two subarrays are then sorted recursively.

Implemented naively, quicksort has large run times and deep function call stacks on arrays with many duplicates because the subarrays may differ greatly in size. One solution is to reorder the array so that all elements less than the pivot appear first, followed by elements equal to the pivot, followed by elements greater than the pivot. This is known as Dutch national flag partitioning, because the Dutch national flag consists of three horizontal bands, each in a different color.

As an example, assuming that black precedes white and white precedes gray, Figure 2.1(b) on the following page is a valid partitioning for Figure 2.1(a) on the next page. If gray precedes black and black precedes white, Figure 2.1(c) on the following page is a valid partitioning for Figure 2.1(a) on the next page.



(a) Before partitioning.

(b) A three-way partitioning resembling the Dutch national flag.

(c) Another three-way partitioning: the Russian national flag.

**Figure 2.1:** Illustrating the Dutch national flag problem.

Write a program that takes an array $A$ and an index $i$ into $A$, and rearranges the elements such that all elements less than $A[i]$ (the "pivot") appear first, followed by elements equal to the pivot, followed by elements greater than the pivot.

*Hint:* Think about the partition step in quicksort.

**Solution:** The problem is trivial to solve with $O(n)$ additional space, where $n$ is the length of $A$. We form three lists, namely, elements less than the pivot, elements equal to the pivot, and elements greater than the pivot. Consequently, we write these values into $A$. The time complexity is $O(n)$.

We can avoid using $O(n)$ additional space at the cost of increased time complexity as follows. In the first stage, we iterate through $A$ starting from index 0, then index 1, etc. In each iteration, we seek an element smaller than the pivot—as soon as we find it, we move it to the subarray of smaller elements via an exchange. This moves all the elements less than the pivot to the start of the array. The second stage is similar to the first one, the difference being that we move elements greater than the pivot to the end of the array. Code illustrating this approach is shown below.

```
public static enum Color { RED, WHITE, BLUE }

public static void dutchFlagPartition(int pivotIndex, List<Color> A) {
  Color pivot = A.get(pivotIndex);
  // First pass: group elements smaller than pivot.
  for (int i = 0; i < A.size(); ++i) {
    // Look for a smaller element.
    for (int j = i + 1; j < A.size(); ++j) {
      if (A.get(j).ordinal() < pivot.ordinal()) {
        Collections.swap(A, i, j);
        break;
      }
    }
  }
  // Second pass: group elements larger than pivot.
  for (int i = A.size() - 1; i >= 0 && A.get(i).ordinal() >= pivot.ordinal();
       --i) {
    // Look for a larger element. Stop when we reach an element less
    // than pivot, since first pass has moved them to the start of A.
    for (int j = i - 1; j >= 0 && A.get(j).ordinal() >= pivot.ordinal();
         --j) {
```

```
      if (A.get(j).ordinal() > pivot.ordinal()) {
        Collections.swap(A, i, j);
        break;
      }
    }
  }
}
```

The additional space complexity is now $O(1)$, but the time complexity is $O(n^2)$, e.g., if $i = n/2$ and all elements before $i$ are greater than $A[i]$, and all elements after $i$ are less than $A[i]$. Intuitively, this approach has bad time complexity because in the first pass when searching for each additional element smaller than the pivot we start from the beginning. However, there is no reason to start from so far back—we can begin from the last location we advanced to. (Similar comments hold for the second pass.)

To improve time complexity, we make a single pass and move all the elements less than the pivot to the beginning, In the second pass we move the larger elements to the end. It is easy to perform each pass in a single iteration, moving out-of-place elements as soon as they are discovered.

```
public static enum Color { RED, WHITE, BLUE }

public static void dutchFlagPartition(int pivotIndex, List<Color> A) {
  Color pivot = A.get(pivotIndex);
  // First pass: group elements smaller than pivot.
  int smaller = 0;
  for (int i = 0; i < A.size(); ++i) {
    if (A.get(i).ordinal() < pivot.ordinal()) {
      Collections.swap(A, smaller++, i);
    }
  }
  // Second pass: group elements larger than pivot.
  int larger = A.size() - 1;
  for (int i = A.size() - 1; i >= 0 && A.get(i).ordinal() >= pivot.ordinal();
       --i) {
    if (A.get(i).ordinal() > pivot.ordinal()) {
      Collections.swap(A, larger--, i);
    }
  }
}
```

The time complexity is $O(n)$ and the space complexity is $O(1)$.

The algorithm we now present is similar to the one sketched above. The main difference is that it performs classification into elements less than, equal to, and greater than the pivot in a single pass. This reduces runtime, at the cost of a trickier implementation. We do this by maintaining four subarrays: *bottom* (elements less than pivot), *middle* (elements equal to pivot), *unclassified*, and *top* (elements greater than pivot). Initially, all elements are in *unclassified*. We iterate through elements in *unclassified*, and move elements into one of *bottom*, *middle*, and *top* groups according to the relative order between the incoming unclassified element and the pivot.

As a concrete example, suppose the array is currently $A = \langle -3, 0, -1, 1, 1, ?, ?, ?, 4, 2 \rangle$, where the pivot is 1 and ? denotes unclassified elements. There are three possibilities for the first unclassified element, $A[5]$.

- $A[5]$ is less than the pivot, e.g., $A[5] = -5$. We exchange it with the first 1, i.e., the new array is $\langle -3, 0, -1, -5, 1, 1, ?, ?, 4, 2 \rangle$.
- $A[5]$ is equal to the pivot, i.e., $A[5] = 1$. We do not need to move it, we just advance to the next unclassified element, i.e., the array is $\langle -3, 0, -1, 1, 1, 1, ?, ?, 4, 2 \rangle$.
- $A[5]$ is greater than the pivot, e.g., $A[5] = 3$. We exchange it with the last unclassified element, i.e., the new array is $\langle -3, 0, -1, 1, 1, ?, ?, 3, 4, 2 \rangle$.

Note how the number of unclassified elements reduces by one in each case.

```
public static enum Color { RED, WHITE, BLUE }

public static void dutchFlagPartition(int pivotIndex, List<Color> A) {
  Color pivot = A.get(pivotIndex);

  /**
   * Keep the following invariants during partitioning:
   * bottom group: A.subList(0, smaller).
   * middle group: A.subList(smaller, equal).
   * unclassified group: A.subList(equal, larger).
   * top group: A.subList(larger, A.size()).
   */
  int smaller = 0, equal = 0, larger = A.size();
  // Keep iterating as long as there is an unclassified element.
  while (equal < larger) {
    // A.get(equal) is the incoming unclassified element.
    if (A.get(equal).ordinal() < pivot.ordinal()) {
      Collections.swap(A, smaller++, equal++);
    } else if (A.get(equal).ordinal() == pivot.ordinal()) {
      ++equal;
    } else { // A.get(equal) > pivot.
      Collections.swap(A, equal, --larger);
    }
  }
}
```

Each iteration decreases the size of *unclassified* by 1, and the time spent within each iteration is $O(1)$, implying the time complexity is $O(n)$. The space complexity is clearly $O(1)$.

**Variant:** Assuming that keys take one of three values, reorder the array so that all objects with the same key appear together. The order of the subarrays is not important. For example, both Figures 2.1(b) and 2.1(c) on Page 6 are valid answers for Figure 2.1(a) on Page 6. Use $O(1)$ additional space and $O(n)$ time.

**Variant:** Given an array $A$ of $n$ objects with keys that takes one of four values, reorder the array so that all objects that have the same key appear together. Use $O(1)$ additional space and $O(n)$ time.