Chapter

6 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

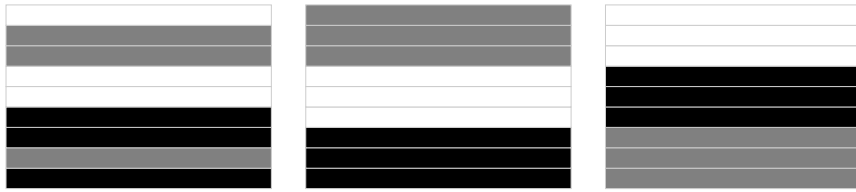
The simplest data structure is the *array*, which is a contiguous block of memory. It is usually used to represent sequences. Given an array A, $A[i]$ denotes the $(i+1)$th object stored in the array. Retrieving and updating $A[i]$ takes $O(1)$ time. Insertion into a full array can be handled by resizing, i.e., allocating a new array with additional memory and copying over the entries from the original array. This makes the worst-case time of insertion high 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 $\langle 2, 3, 5, 7, 11, 13, 17, 0\rangle$. (The last value is a don't care.) The time complexity to delete the element at index i from an array of length n is $O(n - i)$.

6.1 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 6.1(b) on the facing page is a valid partitioning for Figure 6.1(a) on the next page. If gray precedes black and black precedes white, Figure 6.1(c) on the facing page is a valid partitioning for Figure 6.1(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 6.1: 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 $A[i]$—as soon as we find it, we exchange it with the element at the current starting index. This moves all the elements less than $A[i]$ to the start of the array. The second stage is very similar to the first one, the difference beginning that we move elements equal to $A[i]$ to the middle of the array. (The remaining elements must be greater than $A[i]$, and so are already in their correct location.) Note the resemblance to selection sort—a key difference is that we don't need to find the smallest elements, just the first element less than $A[i]$ in the first pass. 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, the above approach has bad time complexity because when searching for an element smaller than $A[i]$ we start at $i + 1$. However, there is no reason to start from $i + 1$—we can begin from the last location we advanced to.

A better algorithm is to make one pass to move all the elements less than the pivot to the beginning, and then make a second pass over the subarray consisting of the remaining elements, moving elements equal to the pivot to the start of that subarray. It is easy to perform each pass with an iteration, exchanging elements out of place. The time complexity is $O(n)$ and the space complexity is $O(1)$. The program below implements this approach. We simplify the code slightly by proceeding backwards in the second pass, and moving the larger elements to the end—this allows us to avoid having to record where the smaller elements end.

```
void DutchFlagPartition(int pivot_index, vector<int>* A_ptr) {
```

```
  vector<int>& A = *A_ptr;
  int pivot = A[pivot_index];
  // First pass: group elements smaller than pivot.
  int smaller = 0;
  for (int i = 0; i < A.size(); ++i) {
    if (A[i] < pivot) {
      swap(A[i], A[smaller++]);
    }
  }
  // Second pass: group elements larger than pivot.
  int larger = A.size() - 1;
  for (int i = A.size() - 1; i >= 0 && A[i] >= pivot; --i) {
    if (A[i] > pivot) {
      swap(A[i], A[larger--]);
    }
  }
}
```

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.

```
void DutchFlagPartition(int pivot_index, vector<int>* A_ptr) {
  vector<int>& A = *A_ptr;
  int pivot = A[pivot_index];
  /**
   * Keep the following invariants during partitioning:
   * bottom group: A[0 : smaller - 1].
   * middle group: A[smaller : equal - 1].
   * unclassified group: A[equal : larger].
   * top group: A[larger + 1 : A.size() - 1].
   */
  int smaller = 0, equal = 0, larger = A.size() - 1;
  // Keep iterating as long as there is an unclassified element.
  while (equal <= larger) {
    // A[equal] is the incoming unclassified element.
```

```
    if (A[equal] < pivot) {
      swap(A[smaller++], A[equal++]);
    } else if (A[equal] == pivot) {
      ++equal;
    } else {  // A[equal] > pivot.
      swap(A[equal], A[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 6.1(b) and 6.1(c) on Page 57 are valid answers for Figure 6.1(a) on Page 57. 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.

Variant: Given an array A of n objects with Boolean-valued keys, reorder the array so that objects that have the key `false` appear first. Use $O(1)$ additional space and $O(n)$ time.

Variant: Given an array A of n objects with Boolean-valued keys, reorder the array so that objects that have the key `false` appear first. The relative ordering of objects with key `true` should not change. Use $O(1)$ additional space and $O(n)$ time.

6.2 Increment an arbitrary-precision integer

Write a program which takes as input an array of digits encoding a decimal number D and updates the array to represent the number $D + 1$. For example, if the input is $\langle 1, 2, 9 \rangle$ then you should update the array to $\langle 1, 3, 0 \rangle$. Your algorithm should work even if it is implemented in a language that has finite-precision arithmetic.

Hint: Experiment with concrete examples.

Solution: A brute-force approach might be to convert the array of digits to the equivalent integer, increment that, and then convert the resulting value back to an array of digits. For example, if the array is $\langle 1, 2, 9 \rangle$, we would derive the integer 129, add one to get 130, then extract its digits to form $\langle 1, 3, 0 \rangle$. When implemented in a language that imposes a limit on the range of values an integer type can take, this approach will fail on inputs that encode integers outside of that range.

We can avoid overflow issues by operating directly on the array of digits. Specifically, we mimic the grade-school algorithm for adding numbers, which entails adding digits starting from the least significant digit, and propagate carries. If the result has an additional digit, e.g., 99 + 1 = 100, all digits have to be moved to the right by one.

For the given example, we would update 9 to 0 with a carry out of 1. We update 2 to 3 (because of the carry in). There is no carry out, so we stop—the result is $\langle 1, 3, 0 \rangle$.

```
vector<int> PlusOne(vector<int> A) {
  ++A.back();
  for (int i = A.size() - 1; i > 0 && A[i] == 10; --i) {
    A[i] = 0, ++A[i - 1];
  }
  if (A[0] == 10) {
    // Need additional digit as the most significant digit (i.e., A[0]) has a
    // carry-out.
    A[0] = 0;
    A.insert(A.begin(), 1);
  }
  return A;
}
```

The time complexity is $O(n)$, where n is the length of A.

Variant: Write a program which takes as input two strings s and t of bits encoding binary numbers B_s and B_t, respectively, and returns a new string of bits representing the number $B_s + B_t$.

6.3 Multiply two arbitrary-precision integers

Certain applications require arbitrary precision arithmetic. One way to achieve this is to use arrays to represent integers, e.g., with one digit per array entry, with the most significant digit appearing first, and a negative leading digit denoting a negative integer. For example, $\langle 1, 9, 3, 7, 0, 7, 7, 2, 1 \rangle$ represents 193707721 and $\langle -7, 6, 1, 8, 3, 8, 2, 5, 7, 2, 8, 7 \rangle$ represents -761838257287.

Write a program that takes two arrays representing integers, and returns an integer representing their product. For example, since $193707721 \times -761838257287 = -147573952589676412927$, if the inputs are $\langle 1, 9, 3, 7, 0, 7, 7, 2, 1 \rangle$ and $\langle -7, 6, 1, 8, 3, 8, 2, 5, 7, 2, 8, 7 \rangle$, your function should return $\langle -1, 4, 7, 5, 7, 3, 9, 5, 2, 5, 8, 9, 6, 7, 6, 4, 1, 2, 9, 2, 7 \rangle$.

Hint: Use arrays to simulate the grade-school multiplication algorithm.

Solution: As in Solution 6.2 on the preceding page, the possibility of overflow precludes us from converting to the integer type.

Instead we can use the grade-school algorithm for multiplication which consists of multiplying the first number by each digit of the second, and then adding all the resulting terms.

From a space perspective, it is better to incrementally add the terms rather than compute all of them individually and then add them up. The number of digits required for the product is at most $n + m$ for n and m digit operands, so we use an array of size $n + m$ for the result. Indexing into the arrays is simplified if we reverse them—the first entry becomes the least significant digit.

For example, when multiplying 123 with 987, we would form $7 \times 123 = 861$, then we would form $8 \times 123 \times 10 = 9840$, which we would add to 861 to get 10701. Then we would form $9 \times 123 \times 100 = 110700$, which we would add to 10701 to get the final result 121401. (All numbers shown are represented using arrays of digits.)

```
vector<int> Multiply(vector<int> num1, vector<int> num2) {
  bool is_negative = (num1.front() < 0 && num2.front() >= 0) ||
                     (num1.front() >= 0 && num2.front() < 0);
  num1.front() = abs(num1.front()), num2.front() = abs(num2.front());

  // Reverses num1 and num2 to make multiplication easier.
  reverse(num1.begin(), num1.end());
  reverse(num2.begin(), num2.end());
  vector<int> result(num1.size() + num2.size(), 0);
  for (int i = 0; i < num1.size(); ++i) {
    for (int j = 0; j < num2.size(); ++j) {
      result[i + j] += num1[i] * num2[j];
      result[i + j + 1] += result[i + j] / 10;
      result[i + j] %= 10;
    }
  }

  // Skips the leading 0s and keeps one 0 if all are 0s.
  while (result.size() != 1 && result.back() == 0) {
    result.pop_back();
  }
  // Reverses result to get the most significant digit as the start of array.
  reverse(result.begin(), result.end());
  if (is_negative) {
    result.front() *= -1;
  }
  return result;
}
```

There are m partial products, each with at most $n + 1$ digits. We perform $O(1)$ operations on each digit in each partial product, so the time complexity is $O(nm)$.

Variant: Solve the same problem when numbers are represented as lists of digits.

6.4 Advancing through an array

In a particular board game, a player have to try to advance through a sequence of positions. Each position has a nonnegative integer associated with it, representing the maximum you can advance from that position in one move. For example, let $A = \langle 3, 3, 1, 0, 2, 0, 1 \rangle$ represent the board game, i.e., the ith entry in A is the maximum we can advance from i. Then the following sequence of advances leads to the last position: $\langle 1, 3, 2 \rangle$. If $A = \langle 3, 2, 0, 0, 2, 0, 1 \rangle$, it is not possible to advance past position 3.

Write a program which takes an array of n integers, where $A[i]$ denotes the maximum you can advance from index i, and returns whether it is possible to advance to the last index starting from the beginning of the array.

Hint: Analyze each location, starting from the beginning.

Solution: It is natural to try advancing as far as possible in each step. This approach does not always work, because it potentially skips indices containing large entries. For example, if $A = \langle 2, 4, 1, 1, 0, 2, 3 \rangle$, then it advances to index 2, which contains a 1, which leads to index 3, after which it cannot progress. However, advancing to index 1, which contains a 4 lets us proceed to index 5, from which we can advance to index 6.

The above example suggests iterating through all entries in A. As we iterate through the array, we track the furthest index we know we can advance to. The furthest we can advance from index i is $i + A[i]$. If, for some i before the end of the array, i is the furthest index that we have demonstrated that we can advance to, we cannot reach the last index. Otherwise, we reach the end.

For example, if $A = \langle 3, 3, 1, 0, 2, 0, 1 \rangle$, we iteratively compute the furthest we can advance to as $0, 3, 4, 4, 4, 6, 6, 7$, which reaches the last index, 6. If $A = \langle 3, 2, 0, 0, 2, 0, 1 \rangle$, we iteratively update the furthest we can advance to as $0, 3, 3, 3, 3$, after which we cannot advance, so it is not possible to reach the last index.

The code below implements this algorithm. Note that it is robust with respect to negative entries, since we track the maximum of how far we proved we can advance to and $i + A[i]$.

```
bool CanReachEnd(const vector<int>& max_advance_steps) {
  int furthest_reach_so_far = 0;
  for (int i = 0; i <= furthest_reach_so_far &&
                  furthest_reach_so_far < max_advance_steps.size() - 1;
       ++i) {
    furthest_reach_so_far =
        max(furthest_reach_so_far, max_advance_steps[i] + i);
  }
  return furthest_reach_so_far >= max_advance_steps.size() - 1;
}
```

The time complexity is $O(n)$, and the additional space complexity (beyond what is used for A) is three integer variables, i.e., $O(1)$.

Variant: Write a program to compute the minimum number of steps needed to advance to the last location.

6.5 Delete a key from an array

This problem is concerned with writing a remove function for arrays. For example, if the array is $\langle 5, 3, 7, 11, 2, 3, 13, 5, 7 \rangle$ and the key to remove is 3, then $\langle 5, 7, 11, 2, 13, 5, 7, 0, 0 \rangle$ is an acceptable update to the array. (The last two entries are don't cares—$\langle 5, 7, 11, 2, 13, 5, 7, 5, 7 \rangle$ would also be acceptable.) Many languages have library functions for performing this operation. You cannot use these functions.

Implement a function which takes as input an array and a key, and updates the array so that all occurrences of the input key have been removed and the remaining elements have been shifted left to fill the emptied indices. Return the number of remaining elements. There are no requirements as to the values stored beyond the last valid element.

Hint: Don't delete entries one-at-a-time.

Solution: A brute-force approach might be to traverse the input array, storing entries not equal to the key into a new array, and then copying the entries of the new array back over to the input array. The time and space complexity are both $O(n)$, where n is the length of the input array.

We can avoid the additional space complexity with an increased time complexity by traversing the array, and every time an entry equals the key, moving all subsequent entries one entry to the left. This approach performs poorly when the key appears frequently, since subarrays are repeatedly shifted left. The time complexity is $O(n^2)$, where n is the length of the array. For example, when all entries are equal to the key, the number of shifts is $(n-1)+(n-2)+\cdots+2+1$.

The key to improving time complexity while sticking to $O(1)$ space is avoiding wasted copies. Note that if we iterate forward through the array, any element not equal to the key that we move left never needs to be moved again. At a top level, our algorithm skips over elements equal to the key, and tracks the location the next value not equal to the key should be written to.

For the given example, we would iterate to the first 3, which is at index 1. We would then write subsequent elements one to the left till we get to the next 3. At this point the array would be $\langle 5,7,11,2,2,3,13,5,7\rangle$. To skip the next 3 we write subsequent elements two to the left. We continue till the end of the array, which is now $\langle 5,7,11,2,13,5,7,5,7\rangle$. Since we are not required to reset the invalid entries, we simply return 7, the index of the next entry to write to, which is also the number of valid elements.

```
// Returns the number of valid entries after deletion.
size_t DeleteKey(int key, vector<int>* A_ptr) {
  auto& A = *A_ptr;
  size_t write_idx = 0;
  for (size_t i = 0; i < A.size(); ++i) {
    if (A[i] != key) {
      A[write_idx++] = A[i];
    }
  }
  return write_idx;
}
```

The time complexity is $O(n)$, where n is the length of the array. The additional space complexity is $O(1)$.

6.6 Delete duplicates from a sorted array

This problem is concerned with deleting repeated elements from a sorted array. For example, for the array $\langle 2,3,5,5,7,11,11,11,13\rangle$, then after deletion, the array is $\langle 2,3,5,7,11,13,0,0,0\rangle$. After deleting repeated elements, there are 6 valid entries. There are no requirements as to the values stored beyond the last valid element.

Write a program which takes as input a sorted array and updates it so that all duplicates have been removed and the remaining elements have been shifted left to fill the

emptied indices. Return the number of valid elements. Many languages have library functions for performing this operation—you cannot use these functions.

Hint: There is an $O(n)$ time and $O(1)$ space solution.

Solution: Let A be the array and n its length. If we allow ourselves $O(n)$ additional space, we can solve the problem by iterating through A and recording values that have not appeared previously into a hash table. (The hash table is used to determine if a value is new.) New values are also written to a list. The list is then copied back into A.

Here is a brute-force algorithm that uses $O(1)$ additional space—iterate through A, testing if $A[i]$ equals $A[i+1]$, and, if so, shift all elements at and after $i+2$ to the left by one. As in Solution 6.5 on the previous page, the same worst-case input demonstrates the time complexity is $O(n^2)$, where n is the length of the array.

The intuition behind achieving a better time complexity is to reduce the amount of shifting. Since the array is sorted, repeated elements must appear one-after-another, so we do not need an auxiliary data structure to check if an element has appeared already. We move just one element, rather than an entire subarray, and ensure that we move it just once.

For the given example, $\langle 2,3,5,5,7,11,11,11,13\rangle$, when processing the $A[3]$, since we already have a 5 (which we know by comparing $A[3]$ with $A[2]$), we advance to $A[4]$. Since this is a new value, we move it to the first vacant entry, namely $A[3]$. Now the array is $\langle 2,3,5,7,7,11,11,11,13\rangle$, and the first vacant entry is $A[4]$. We continue from $A[5]$.

```
// Returns the number of valid entries after deletion.
int DeleteDuplicates(vector<int>* A_ptr) {
  vector<int>& A = *A_ptr;
  if (A.empty()) {
    return 0;
  }

  int write_index = 1;
  for (int i = 1; i < A.size(); ++i) {
    if (A[write_index - 1] != A[i]) {
      A[write_index++] = A[i];
    }
  }
  return write_index;
}
```

The time complexity is $O(n)$, and the space complexity is $O(1)$, since all that is needed is the two additional variables.

Variant: Write a program which takes as input a sorted array A of integers and a positive integer m, and updates A so that if x appears m times in A it appears exactly $\min(2,m)$ times in A. The update to A should be performed in one pass, and no additional storage may be allocated.

6.7 Buy and sell a stock once

This problem is concerned with the problem of optimally buying and selling a stock once, as described on Page 2. As an example, consider the following sequence of stock prices: $\langle 310, 315, 275, 295, 260, 270, 290, 230, 255, 250\rangle$. The maximum profit that can be made with one buy and one sell is 30—buy at 260 and sell at 290. Note that 260 is not the lowest price, nor 290 the highest price.

Write a program that takes an array denoting the daily stock price, and returns the maximum profit that could be made by buying and then selling that one share of that stock.

Hint: Identifying the minimum and maximum is not enough since the minimum may appear after the maximum height. Focus on valid differences.

Solution: We developed several algorithms for this problem in the introduction. Specifically, on Page 2 we showed how to compute the maximum profit by computing the difference of the current entry with the minimum value seen so far as we iterate through the array.

For example, the array of minimum values seen so far for the given example is $\langle 310, 310, 270, 270, 260, 260, 260, 230, 230, 230\rangle$. The maximum profit that can be made by selling on each specific day is the difference of the current price and the minimum seen so far, i.e., $\langle 0, 5, 0, 20, 0, 10, 30, 0, 25, 20\rangle$. The maximum profit overall is 30, corresponding to buying 260 and selling for 290.

```
double BuyAndSellStockOnce(const vector<double>& prices) {
  double min_price_so_far = numeric_limits<double>::max(), max_profit = 0;
  for (const double& price : prices) {
    double max_profit_sell_today = price - min_price_so_far;
    max_profit = max(max_profit, max_profit_sell_today);
    min_price_so_far = min(min_price_so_far, price);
  }
  return max_profit;
}
```

The time complexity is $O(n)$ time and the space complexity is $O(1)$ space, where n is the length of the array.

Variant: Write a program that takes an array of integers and finds the length of a longest subarray all of whose entries are equal.

6.8 Buy and sell a stock twice

The max difference problem, introduced on Page 1, formalizes the maximum profit that can be made by buying and then selling a single share over a given day range.

Write a program that computes the maximum profit that can be made by buying and selling a share at most twice. The second buy must be made after the first sale.

Hint: What do you need to know about the first i elements when processing the $(i+1)$th element?

Solution: The brute-force algorithm which examines all possible combinations of buy-sell-buy-sell days has complexity $O(n^4)$. The complexity can be improved to $O(n^2)$ by applying the $O(n)$ algorithm to each pair of subarrays formed by splitting A.

The inefficiency in the above approaches comes from not taking advantage of previous computations. Suppose we record the best solution for $A[0 : j]$, j between 1 and $n - 1$, inclusive. Now we can do a reverse iteration, computing the best solution for a single buy-and-sell for $A[j : n - 1]$, j between 1 and $n - 1$, inclusive. For each day, we combine this result with the result from the forward iteration for the previous day—this yields the maximum profit if we buy and sell once before the current day and once at or after the current day.

For example, suppose the input array is $\langle 12, 11, 13, 9, 12, 8, 14, 13, 15\rangle$. Then the most profit that can be made with a single buy and sell by Day i is $F = \langle 0, 0, 2, 2, 3, 3, 5, 5, 7\rangle$. Working backwards, the most profit that can be made with a single buy and sell on or after Day i is $B = \langle 7, 7, 7, 7, 7, 7, 2, 2, 0\rangle$. Combining these two we get $M = \langle 7, 7, 7, 9, 9, 10, 5, 7, 5\rangle$, i.e., the maximum profit is 10.

```
double BuyAndSellStockTwice(const vector<double>& prices) {
  double max_total_profit = 0;
  vector<double> first_buy_sell_profits(prices.size(), 0);
  double min_price_so_far = numeric_limits<double>::max();
  // Forward phase. For each day, we record maximum profit if we
  // sell on that day.
  for (int i = 0; i < prices.size(); ++i) {
    min_price_so_far = min(min_price_so_far, prices[i]);
    max_total_profit = max(max_total_profit, prices[i] - min_price_so_far);
    first_buy_sell_profits[i] = max_total_profit;
  }
  // Backward phase. For each day, find the maximum profit if we make
  // the second buy on that day.
  double max_price_so_far = numeric_limits<double>::min();
  for (int i = prices.size() - 1; i > 0; --i) {
    max_price_so_far = max(max_price_so_far, prices[i]);
    max_total_profit = max(max_total_profit, max_price_so_far - prices[i] +
                                             first_buy_sell_profits[i -
                                                 1]);
  }
  return max_total_profit;
}
```

The time complexity is $O(n)$, and the additional space complexity is $O(n)$, which is the space used to store the best solutions for the subarrays.

Variant: Solve the same problem in $O(n)$ and $O(1)$ space.

6.9 Enumerate all primes to n

A natural number is called a prime if it is bigger than 1 and has no divisors other than 1 and itself.

Write a program that takes a integer argument and returns all the primes between 1 and that integer. For example, if the input is 18, you should return $\langle 2, 3, 5, 7, 11, 13, 17\rangle$.

Hint: Exclude the multiples of primes.

Solution: The natural brute-force algorithm is to iterate over all i from 2 to n, where n is the input to the program. For each i, we test if i is prime; if so we add it to the result. We can use "trial-division" to test if i is prime, i.e., by dividing i by each integer from 2 to the square root of i, and checking if the remainder is 0. (There is no need to test beyond the square root of i, since if i has a divisor other than 1 and itself, it must also have a divisor that is no greater than its square root.) Since each test has time complexity $O(\sqrt{n})$, the time complexity of the entire computation is upper bounded by $O(n \times \sqrt{n})$, i.e., $O(n^{3/2})$.

Intuitively, the brute-force algorithm tests each number from 1 to n independently, and does not exploit the fact that we need to compute *all* primes from 1 to n. Heuristically, a better approach is to compute the primes and when a number is identified as a prime, to "sieve" it, i.e., remove all its multiples from future consideration.

We use a Boolean array to encode the candidates, i.e., if the ith entry in the array is true, then i is potentially a prime. Initially, every number greater than or equal to 2 is a candidate. Whenever we determine a number is a prime, we will add it to the result, which is an array. The first prime is 2. We add it to the result. None of its multiples can be primes, so remove all its multiples from the candidate set by writing false in the corresponding locations. The next location set to true is 3. It must be a prime since nothing smaller than it and greater than 1 is a divisor of it. As before, we add it to result and remove its multiples from the candidate array. We continue till we get to the end of the array of candidates.

As an example, if $n = 10$, the candidate array is initialized to $\langle F, F, T, T, T, T, T, T, T, T \rangle$, where T is true and F is false. (Entries 0 and 1 are false, since 0 and 1 are not primes.) We begin with index 2. Since the corresponding entry is one, we add 2 to the list of primes, and sieve out its multiples. The array is now $\langle F, F, T, T, F, T, F, T, F, T \rangle$. The next nonzero entry is 3, so we add it to the list of primes, and sieve out its multiples. The array is now $\langle F, F, T, T, F, T, F, T, F, F \rangle$. The next nonzero entries are 5 and 7, and neither of them can be used to sieve out more entries.

```
// Given n, return all primes up to and including n.
vector<int> GeneratePrimes(int n) {
  vector<int> primes;
  // is_prime[p] represents whether p is prime or not.
  // Initially, set each to true. Then use sieving to eliminate non primes.
  deque<bool> is_prime(n + 1, true);
  is_prime[0] = is_prime[1] = false;
  for (int p = 2; p < n; ++p) {
    if (is_prime[p]) {
      primes.emplace_back(p);
      // Sieve p's multiples.
      for (int j = p; j <= n; j += p) {
        is_prime[j] = false;
      }
    }
  }
  return primes;
```

```
}
```

We justified the sifting approach over the trial-division algorithm on heuristic grounds. The time to sift out the multiples of p is proportional to n/p, so the overall time complexity is $O(n/2 + n/3 + n/5 + n/7 + n/11 + \ldots)$. Although not obvious, this sum asymptotically tends to $n \log \log n$, yielding an $O(n \log \log n)$ time bound. The space complexity is dominated by the storage for P, i.e., $O(n)$.

The bound we gave for the trial-division approach, namely $O(n^{3/2})$, is based on an $O(\sqrt{n})$ bound for each individual test. Since most numbers are not prime, the actual time complexity of trial-division is actually lower on average, since the test early returns false. It is known that the time complexity of the trial-division approach is $O(n^{3/2}/(\log n)^2)$, so sieving is in fact superior to trial-division.

We can improve runtime by sieving p's multiples from p^2 instead of p, since all numbers of the form kp, where $k < p$ have already been sieved out. The storage can be reduced by ignoring even numbers. The code below reflects these optimizations.

```
// Given n, return all primes up to and including n.
vector<int> GeneratePrimes(int n) {
  const int kSize = floor(0.5 * (n - 3)) + 1;
  vector<int> primes;
  primes.emplace_back(2);
  // is_prime[i] represents (2i + 3) is prime or not.
  // Initially, set each to true. Then use sieving to eliminate non primes.
  deque<bool> is_prime(kSize, true);
  for (long i = 0; i < kSize; ++i) {
    if (is_prime[i]) {
      int p = (i * 2) + 3;
      primes.emplace_back(p);
      // Sieving from p^2, where p^2 = 4i^2 + 12i + 9 whose index in is_prime
      // is 2i^2 + 6i + 3 because is_prime[i] represents 2i + 3.
      for (long j = ((i * i) * 2) + 6 * i + 3; j < kSize; j += p) {
        is_prime[j] = false;
      }
    }
  }
  return primes;
}
```

The asymptotic time and space complexity are the same as that for the basic sieving approach.

6.10 Permute the elements of an array

A permutation is a rearrangement of members of a sequence into a new sequence. For example, there are 24 permutations of $\langle a, b, c, d \rangle$; some of these are $\langle b, a, d, c \rangle$, $\langle d, a, b, c \rangle$, and $\langle a, d, b, c \rangle$.

A permutation can be specified by an array P, where $P[i]$ represents the location of the element at i in the permutation. For example, the array $\langle 2, 0, 1, 3 \rangle$ represents the permutation that maps the element at location 0 to location 2, the element at location 1 to location 0, the element at location 2 to location 1, and keep the element

at location 3 unchanged. A permutation can be applied to an array to reorder the array. For example, the permutation $\langle 2, 0, 1, 3 \rangle$ applied to $A = \langle a, b, c, d \rangle$ yields the array $\langle b, c, a, d \rangle$.

Given an array A of n elements and a permutation P, apply P to A.

Hint: Any permutation can be viewed as a set of cyclic permutations. For an element in a cycle, how would you identify if it has been permuted?

Solution: It is simple to apply a permutation-array to a given array if additional storage is available to write the resulting array. We allocate a new array B of the same length, set $B[i] = A[P[i]]$ for each i, and then copy B to A. The time complexity is $O(n)$, and the additional space complexity is $O(n)$.

A key insight to improving space complexity is to decompose permutations into simpler structures which can be processed incrementally. For example, consider the permutation $\langle 3, 2, 1, 0 \rangle$. To apply it to an array $A = \langle \alpha, b, c, \delta \rangle$, we move the element at index 0 (α) to index 3 and the element already at index 3 (δ) to index 0. Continuing, we move the element at index 1 (b) to index 2 and the element already at index 2 (c) to index 1. Now all elements have been moved according to the permutation, and the result is $\langle \delta, c, b, \alpha \rangle$.

This example generalizes: every permutation can be represented by a collection of independent permutations, each of which is *cyclic*, that is, it moves all elements by a fixed offset, wrapping around.

This is significant, because a single cyclic permutation can be performed one element at a time, i.e., with constant additional storage. Consequently, if the permutation is described as a set of cyclic permutations, it can easily be applied using a constant amount of additional storage by applying each cyclic permutation one-at-a-time. Therefore, we want to identify the disjoint cycles that constitute the permutation.

To find and apply the cycle that includes entry i we just keep going forward (from i to $P[i]$) till we get back to i. After we are done with that cycle, we need to find another cycle that has not yet been applied. It is trivial to do this by storing a Boolean for each array element.

One way to perform this without explicitly using additional $O(n)$ storage is to use the sign bit in the entries in the permutation-array. Specifically, we subtract n from $P[i]$ after applying it. This means that if an entry is in $P[i]$ is negative, we have performed the corresponding move.

For example, to apply $\langle 3, 1, 2, 0 \rangle$, we begin with the first entry, 3. We move $A[0]$ to $A[3]$, first saving the original $A[3]$. We update the permutation to $\langle -1, 1, 2, 0 \rangle$. We move $A[3]$ to $A[0]$. Since $P[0]$ is negative we know we are done with the cycle starting at 0. We also update the permutation to $\langle -1, 1, 2, -4 \rangle$. Now we examine $P[1]$. Since it is not negative, it means the cycle it belongs to cannot have been applied. We continue as before.

```
void ApplyPermutation(vector<int>* perm_ptr, vector<int>* A_ptr) {
  vector<int> &perm = *perm_ptr, &A = *A_ptr;
  for (int i = 0; i < A.size(); ++i) {
    // Check if the element at index i has not been permutated
```

```
    // by seeing if perm[i] is nonnegative.
    if (perm[i] >= 0) {
      CyclicPermutation(i, &perm, &A);
    }
  }

  // Restore perm.
  for_each(perm.begin(), perm.end(), [&](int& x) { x += perm.size(); });
}

void CyclicPermutation(int start, vector<int>* perm_ptr, vector<int>* A_ptr) {
  vector<int> &perm = *perm_ptr, &A = *A_ptr;
  int i = start;
  int temp = A[start];
  do {
    int next_i = perm[i];
    int next_temp = A[next_i];
    A[next_i] = temp;
    // Subtracts perm.size() from an entry in perm to make it negative, which
    // indicates the corresponding assignment has been performed.
    perm[i] -= perm.size();
    i = next_i, temp = next_temp;
  } while (i != start);
}
```

The program above will apply the permutation in $O(n)$ time. The space complexity is $O(1)$, assuming we can temporarily modify the sign bit from entries in the permutation array.

If we cannot use the sign bit, we can allocate an array of n Booleans indicating whether the element at index i has been processed. Alternately, we can avoid using $O(n)$ additional storage by going from left-to-right and applying the cycle only if the current position is the leftmost position in the cycle.

```
void ApplyPermutation(const vector<int>& perm, vector<int>* A_ptr) {
  vector<int>& A = *A_ptr;
  for (int i = 0; i < A.size(); ++i) {
    // Traverses the cycle to see if i is the minimum element.
    bool is_min = true;
    int j = perm[i];
    while (j != i) {
      if (j < i) {
        is_min = false;
        break;
      }
      j = perm[j];
    }

    if (is_min) {
      CyclicPermutation(i, perm, &A);
    }
  }
}
```

```
void CyclicPermutation(int start, const vector<int>& perm, vector<int>* A_ptr)
    {
  vector<int>& A = *A_ptr;
  int i = start;
  int temp = A[start];
  do {
    int next_i = perm[i];
    int next_temp = A[next_i];
    A[next_i] = temp;
    i = next_i, temp = next_temp;
  } while (i != start);
}
```

Testing whether the current position is the leftmost position entails traversing the cycle once more, which increases the run time to $O(n^2)$.

Variant: Given an array A of integers representing a permutation, update A to represent the inverse permutation using only constant additional storage.

6.11 Compute the next permutation

There exist exactly $n!$ permutations of n elements. These can be totally ordered using the *dictionary ordering*—define permutation p to appear before permutation q if in the first place where p and q differ in their array representations, starting from index 0, the corresponding entry for p is less than that for q. For example, $\langle 2, 0, 1\rangle < \langle 2, 1, 0\rangle$. Note that the permutation $\langle 0, 1, 2\rangle$ is the smallest permutation under dictionary ordering, and $\langle 2, 1, 0\rangle$ is the largest permutation under dictionary ordering.

Write a program that takes as input a permutation, and returns the next permutation under dictionary ordering. If the permutation is the last permutation, return the empty array. For example, if the input is $\langle 1, 0, 3, 2\rangle$ your function should return $\langle 1, 2, 0, 3\rangle$. If the input is $\langle 3, 2, 1, 0\rangle$, return $\langle\rangle$.

Hint: Study concrete examples.

Solution: A brute-force approach might be to find all permutations whose length equals that of the input array, sort them according to the dictionary order, then find the successor of the input permutation in that ordering. Apart from the enormous space and time complexity this entails, simply computing all permutations of length n is a nontrivial problem; see Problem 16.3 on Page 273 for details.

The key insight is that we want to increase the permutation by as little as possible. The loose analogy is to how a car's odometer increments; the difference is that we cannot change values, only reorder them. We will use the permutation $\langle 6, 2, 1, 5, 4, 3, 0\rangle$ to develop this approach.

Specifically, we start from the right, and look at the longest decreasing suffix, which is $\langle 5, 4, 3, 0\rangle$ for our example. We cannot get the next permutation just by modifying this suffix, since it is already the maximum it can be.

Instead we look at the entry e that appears just before the longest decreasing suffix, which is 1 in this case. (If there's no such element, i.e., the longest decreasing suffix

is the entire permutation, the permutation must be $\langle n-1, n-2, \ldots, 2, 1, 0\rangle$, for which there is no next permutation.)

Observe that e must be less than some entries in the suffix (since the entry immediately after e is greater than e). Intuitively, we should swap e with the smallest entry s in the suffix which is larger than e so as to minimize the change to the prefix (which is defined to be the part of the sequence that appears before the suffix).

For our example, e is 1 and s is 3. Swapping s and e results in $\langle 6, 2, 3, 5, 4, 1, 0\rangle$.

We are not done yet—the new prefix is the smallest possible for all permutations greater than the initial permutation, but the new suffix may not be the smallest. We can get the smallest suffix by sorting the entries in the suffix from smallest to largest. For our working example, this yields the suffix $\langle 0, 1, 4, 5\rangle$.

As an optimization, it is not necessary to call a full blown sorting algorithm on suffix. Since the suffix was initially decreasing, and after replacing s by e it remains decreasing, reversing the suffix has the effect of sorting it from smallest to largest.

The general algorithm for computing the next permutation is as follows.

(1.) Find k such that $p[k] < p[k+1]$ and entries after index k appear in decreasing order.

(2.) Find the smallest $p[l]$ such that $p[l] > p[k]$ (such an l must exist since $p[k] < p[k+1]$).

(3.) Swap $p[l]$ and $p[k]$ (note that the sequence after position k remains in decreasing order).

(4.) Reverse the sequence after position k.

```
vector<int> NextPermutation(vector<int> perm) {
  int k = perm.size() - 2;
  while (k >= 0 && perm[k] >= perm[k + 1]) {
    --k;
  }
  if (k == -1) {
    return {};  // perm is the last permutation.
  }

  // Swap the smallest entry after index k that is greater than perm[k]. We
  // exploit the fact that perm[k + 1 : perm.size() - 1] is decreasing so if we
  // search in reverse order, the first entry that is greater than perm[k] is
  // the smallest such entry.
  for (int i = perm.size() - 1; i > k; --i) {
    if (perm[i] > perm[k]) {
      swap(perm[k], perm[i]);
      break;
    }
  }

  // Since perm[k + 1 : perm.size() - 1] is in decreasing order, we can build
  // the smallest dictionary ordering of this subarray by reversing it.
  reverse(perm.begin() + k + 1, perm.end());
  return perm;
}
```

Each step is an iteration through an array, so the time complexity is $O(n)$. All that we use are a few local variables, so the additional space complexity is $O(1)$.

Variant: Compute the kth permutation under dictionary ordering, starting from the identity permutation (which is the first permutation in dictionary ordering).

Variant: Given a permutation p, return the permutation corresponding to the *previous* permutation of p under dictionary ordering.

6.12 Sample offline data

This problem is motivated by the need for a company to select a random subset of its customers to roll out a new feature to. For example, a social networking company may want to see the effect of a new UI on page visit duration without taking the chance of alienating all its users if the rollout is unsuccessful.

Implement an algorithm that takes as input an array of distinct elements and a size, and returns a subset of the given size of the array elements. All subsets should be equally likely. Return the result in input array itself.

Hint: How would you construct a random subset of size $k+1$ given a random subset of size k?

Solution: Let the input array be A, its length n, and the specified size k. A naive approach is to iterate through the input array, selecting entries with probability k/n. Although the average number of selected entries is k, we may select more or less than k entries in this way.

Another approach is to enumerate all subsets of size k and then select one at random from these. Since there are $\binom{n}{k}$ subsets of size k, the time and space complexity are huge. Furthermore, enumerating all subsets of size k is nontrivial (Problem 16.5 on Page 276).

The key to efficiently building a random subset of size exactly k is to first build one of size $k-1$ and then adding one more element, selected randomly from the rest. The problem is trivial when $k = 1$. We make one call to the random number generator, take the returned value mod n (call it r), and swap $A[0]$ with $A[r]$. The entry $A[0]$ now holds the result.

For $k > 1$, we begin by choosing one element at random as above and we now repeat the same process with the $n-1$ element subarray $A[1 : n-1]$. Eventually, the random subset occupies the slots $A[0 : k-1]$ and the remaining elements are in the last $n-k$ slots.

Intuitively, if all subsets of size k are equally likely, then the construction process ensures that the subsets of size $k+1$ are also equally likely. A formal proof, which we do not present, uses mathematical induction—the induction hypothesis is that every permutation of every size k subset of A is equally likely to be in $A[0 : k-1]$.

As a concrete example, let the input be $A = \langle 3, 7, 5, 11 \rangle$ and the size be 3. In the first iteration, we use the random number generator to pick a random integer in the interval $[0, 3]$. Let the returned random number be 2. We swap $A[0]$ with $A[2]$—now the array is $\langle 5, 7, 3, 11 \rangle$. Now we pick a random integer in the interval $[1, 3]$. Let the returned random number be 3. We swap $A[1]$ with $A[3]$—now the resulting array is $\langle 5, 11, 3, 7 \rangle$. Now we pick a random integer in the interval $[2, 3]$. Let the

returned random number be 2. When we swap $A[2]$ with itself the resulting array is unchanged. The random subset consists of the first three entries, i.e., $\{5, 11, 3\}$.

```
void RandomSampling(int k, vector<int>* A_ptr) {
  vector<int>& A = *A_ptr;
  default_random_engine seed((random_device())());  // Random num generator.
  for (int i = 0; i < k; ++i) {
    // Generate a random index in [i, A.size() - 1].
    uniform_int_distribution<int> rand_idx_gen(i, A.size() - 1);
    swap(A[i], A[rand_idx_gen(seed)]);
  }
}
```

The algorithm clearly runs in additional $O(1)$ space. The time complexity is $O(k)$ to select the elements.

The algorithm makes k calls to the random number generator. When k is bigger than $\frac{n}{2}$, we can optimize by computing a subset of $n - k$ elements to remove from the set. For example, when $k = n - 1$, this replaces $n - 1$ calls to the random number generator with a single call.

Variant: The `rand()` function in the standard C library returns a uniformly random number in $[0, \texttt{RAND\_MAX} - 1]$. Does `rand()` mod n generate a number uniformly distributed $[0, n - 1]$?

6.13 Compute a random permutation

Generating random permutations is not as straightforward as it seems. For example, iterating through $\langle 0, 1, \ldots, n - 1 \rangle$ and swapping each element with another randomly selected element does *not* generate all permutations with equal probability. One way to see this is to consider the case $n = 3$. The number of permutations is $3! = 6$. The total number of ways in which we can choose the elements to swap is $3^3 = 27$ and all are equally likely. Since 27 is not divisible by 6, some permutations correspond to more ways than others, so not all permutations are equally likely.

Design an algorithm that creates uniformly random permutations of $\{0, 1, \ldots, n - 1\}$. You are given a random number generator that returns integers in the set $\{0, 1, \ldots, n-1\}$ with equal probability; use as few calls to it as possible.

Hint: If the result is stored in A, how would you proceed once $A[n - 1]$ is assigned correctly?

Solution: A brute-force approach might be to iteratively pick random numbers between 0 and $n - 1$, inclusive. If number repeats, we discard it, and try again. A hash table is a good way to store and test values that have already been picked.

For example, if $n = 4$, we might have the sequence 1, 2, 1 (repeats), 3, 1 (repeats), 2 (repeat), 0 (done, all numbers from 0 to 3 are present). The corresponding permutation is $\langle 1, 2, 3, 0 \rangle$.

It is fairly clear that all permutations are equally likely with this approach. The space complexity beyond that of the result array is $O(n)$ for the hash table. The time complexity is slightly challenging to analyze. Early on, it takes very few iterations to

get more new values, but it takes a long time to collect the last few values. Computing the average number of tries to complete the permutation in this way is known as the Coupon Collector's Problem. It is known that the number of tries on average (and hence the average time complexity) is $O(n \log n)$.

Clearly, the way to improve time complexity is to avoid repeats. We can do this by restricting the set we randomly choose the remaining values from. If we apply Solution 6.12 on Page 73 to $\langle 0, 1, 2, \ldots, n-1 \rangle$ with $k = n$, at each iteration the array is partitioned into the partial permutation and remaining values. Although the subset that is returned is unique (it will be $\{0, 1, \ldots, n-1\}$), all $n!$ possible orderings of the elements in the set occur with equal probability. For example, let $n = 4$. We begin with $\langle 0, 1, 2, 3 \rangle$. The first random number is chosen between 0 and 3, inclusive. Suppose it is 1. We update the array to $\langle 1, 0, 2, 3 \rangle$. The second random number is chosen between 1 and 3, inclusive. Suppose it is 3. We update the array to $\langle 1, 3, 0, 2 \rangle$. The third random number is chosen between 2 and 3, inclusive. Suppose it is 3. We update the array to $\langle 1, 3, 2, 0 \rangle$. This is the returned result.

```
vector<int> ComputeRandomPermutation(int n) {
  vector<int> permutation(n);
  // Initializes permutation to 0, 1, 2, ..., n - 1.
  iota(permutation.begin(), permutation.end(), 0);
  RandomSampling(permutation.size(), &permutation);
  return permutation;
}
```

The time complexity is $O(n)$, and, as an added bonus, no storage outside of that needed for the permutation array itself it needed.

6.14 Compute a random subset

The set $\{0, 1, 2, \ldots, n-1\}$ has $\binom{n}{k} = n!/((n-k)!k!)$ subsets of size k. We seek to design an algorithm that returns any one of these subsets with equal probability.

Write a program that takes as input a positive integer n and a size $k \leq n$, and returns a size-k subset of $\{0, 1, 2, \ldots, n-1\}$. The subset should be represented as an array. All subsets should be equally likely and, in addition, all permutations of elements of the array should be equally likely. You may assume the you have a function which takes as input a nonnegative integer t and returns an integer in the set $\{0, 1, \ldots, t-1\}$ with uniform probability.

Hint: Simulate Solution 6.12 on Page 73, using an appropriate data structure to reduce space.

Solution: Similar to the brute-force algorithm presented in Solution 6.13 on the preceding page, we could iteratively choose random numbers between 0 and $n-1$ until we get k distinct values. This approach suffers from the same performance degradation when k is close to n, and it also requires $O(k)$ additional space.

We could mimic the offline sampling algorithm described in Solution 6.12 on Page 73, with $A[i] = i$ initially, stopping after k iterations. This requires $O(n)$ space and $O(n)$ time to create the array. After creating $\langle 0, 1, 2, \ldots, n-1 \rangle$, we need $O(k)$ time to produce the subset.

Note that when $k \ll n$, most of the array is untouched, i.e., $A[i] = i$. The key to reducing the space complexity to $O(k)$ is simulating A with a hash table. We do this by only tracking entries whose values are modified by the algorithm—the remainder have the default value, i.e., the value of an entry is its index.

Specifically, we maintain a hash table H whose keys and values are from $\{0, 1, \ldots, n-1\}$. Conceptually, H tracks entries of the array which have been touched in the process of randomization—these are entries $A[i]$ which may not equal i. The hash table H is updated as the algorithm advances.

- If i is in H, then its value in H is the value stored at $A[i]$ in the brute-force algorithm.
- If i is not in H, then this implicitly implies $A[i] = i$.

Since we track no more than k entries, when k is small compared to n, we save time and space over the brute-force approach, which has to initialize and update an array of length n.

Initially, H is empty. We do k iterations of the following. Choose a random integer r in $[0, n-1-i]$, where i is the current iteration count, starting at 0. There are four possibilities, corresponding to whether the two entries in A that are being swapped are already present or not present in H. The desired result is in $A[0 : k-1]$, which can be determined from H.

For example, suppose $n = 100$ and $k = 4$. In the first iteration, suppose we get the random number 28. We update H to $(0, 28), (28, 0)$. This means that $A[0]$ is 28 and $A[28]$ is 0—for all other i, $A[i] = i$. In the second iteration, suppose we get the random number 42. We update H to $(0, 28), (28, 0), (1, 42), (42, 1)$. In the third iteration, suppose we get the random number 28 again. We update H to $(0, 28), (28, 2), (1, 42), (42, 1), (2, 0)$. In the third iteration, suppose we get the random number 64. We update H to $(0, 28), (28, 2), (1, 42), (42, 1), (2, 0), (3, 64), (64, 4)$. The random subset is the 4 elements corresponding to indices $0, 1, 2, 3$, i.e., $\langle 28, 42, 0, 64 \rangle$.

```
// Returns a random k-sized subset of {0, 1, ..., n - 1}.
vector<int> RandomSubset(int n, int k) {
  unordered_map<int, int> changed_elements;
  default_random_engine seed((random_device())());  // Random num generator.
  for (int i = 0; i < k; ++i) {
    // Generate a random index in [i, n - 1].
    uniform_int_distribution<int> rand_idx_gen(i, n - 1);
    int rand_idx = rand_idx_gen(seed);
    auto ptr1 = changed_elements.find(rand_idx),
         ptr2 = changed_elements.find(i);
    if (ptr1 == changed_elements.end() && ptr2 == changed_elements.end()) {
      changed_elements[rand_idx] = i;
      changed_elements[i] = rand_idx;
    } else if (ptr1 == changed_elements.end() &&
               ptr2 != changed_elements.end()) {
      changed_elements[rand_idx] = ptr2->second;
      ptr2->second = rand_idx;
    } else if (ptr1 != changed_elements.end() &&
               ptr2 == changed_elements.end()) {
      changed_elements[i] = ptr1->second;
      ptr1->second = i;
```

```
    } else {
      int temp = ptr2->second;
      changed_elements[i] = ptr1->second;
      changed_elements[rand_idx] = temp;
    }
  }

  vector<int> result;
  for (int i = 0; i < k; ++i) {
    result.emplace_back(changed_elements[i]);
  }
  return result;
}
```

The time complexity is $O(k)$, since we perform a bounded number of operations per iteration. The space complexity is also $O(k)$, since H and the result array never contain more than k entries.

6.15 Sample online data

This problem is motivated by the design of a packet sniffer that provides a uniform sample of packets for a network session.

Design a program that takes as input a size k, and reads packets, continuously maintains a uniform random subset of size k of the read packets.

Hint: Suppose you have a procedure which selects k packets from the first $n \geq k$ packets as specified. How would you deal with the $(n+1)$th packet?

Solution: A brute force approach would be to store all the packets read so far. After reading in each packet, we apply Solution 6.12 on Page 73 to compute a random subset of k packets. The space complexity is high—$O(n)$, after n packets have been read. The time complexity is also high—$O(nk)$, since each packet read is followed by a call to Solution 6.12 on Page 73.

At first glance it may seem that it is impossible to do better than the brute-force approach, since after reading the nth packet, we need to choose k packets uniformly from this set. However, suppose we have read the first n packets, and have a random subset of k of them. When we read the $(n+1)$th packet, it should belong to the new subset with probability $k/(n+1)$. If we choose one of the packets in the existing subset uniformly randomly to remove, the resulting collection will be a random subset of the $n+1$ packets.

The formal proof that the algorithm works correctly, uses induction on the number of packets that have been read. Specifically, the induction hypothesis is that all k-sized subsets are equally likely after $n \geq k$ packets have been read.

As an example, suppose $k = 2$, and the packets are read in the order p, q, r, t, u, v. We keep both the first two packets in the subset, which is $\{p, q\}$. We select the next packet, r, with probability 2/3. Suppose it is not selected. Then the subset after reading the first three packets is still $\{p, q\}$. We select the next packet, t, with probability 2/4. Suppose it is selected. Then we choose one of the packets in $\{p, q\}$ uniformly, and

replace it with t. Let q be the selected packet—now the subset is $\{p, t\}$. We select the next packet u with probability 2/5. Suppose it is selected. Then we choose one of the packets in $\{p, t\}$ uniformly, and replace it with u. Let t be the selected packet—now the subset is $\{p, u\}$. We select the next packet v with probability 2/6. Suppose it is not selected. The random subset remains $\{p, u\}$.

```
vector<int> OnlineRandomSample(istringstream* sin, int k) {
  int x;
  vector<int> running_sample;
  // Stores the first k elements.
  for (int i = 0; i < k && *sin >> x; ++i) {
    running_sample.emplace_back(x);
  }

  // After the first k elements.
  int num_seen_so_far = k;
  while (*sin >> x) {
    default_random_engine seed((random_device())());  // Random num generator.
    // Generate a random number in [0, num_seen_so_far], and if this number is
    // in [0, k - 1], we replace that element from the sample with x.
    uniform_int_distribution<int> rand_idx_gen(0, num_seen_so_far);
    int idx_to_replace = rand_idx_gen(seed);
    if (idx_to_replace < k) {
      running_sample[idx_to_replace] = x;
    }
    ++num_seen_so_far;
  }
  return running_sample;
}
```

The time complexity is proportional to the number of elements in the stream, since we spend $O(1)$ time per element. The space complexity is $O(k)$.

Note that at each iteration, every subset is equally likely. However, the subsets are not independent from iteration to iteration—successive subsets differ in at most one element. In contrast, the subsets computed by brute-force algorithm are independent from iteration to iteration.

6.16 Generate nonuniform random numbers

Suppose you need to write a load test for a server. You have studied the inter-arrival time of requests to the server over a period of one year. From this data you have computed a histogram of the distribution of the inter-arrival time of requests. In the load test you would like to generate requests for the server such that the inter-arrival times come from the same distribution that was observed in the historical data. The following problem formalizes the generation of inter-arrival times.

You are given n numbers as well as probabilities $p_0, p_1, \ldots, p_{n-1}$, which sum up to 1. Given a random number generator that produces values in $[0, 1]$ uniformly, how would you generate one of the n numbers according to the specified probabilities? For example, if the numbers are $3, 5, 7, 11$, and the probabilities are $9/18, 6/18, 2/18, 1/18$,

then in 1000000 calls to your program, 3 should appear roughly 500000 times, 5 should appear roughly 333333 times, 7 should appear roughly 111111 times, and 11 should appear roughly 55555 times.

Hint: Look at the graph of the probability that the selected number is less than or equal to α. What do the jumps correspond to?

Solution: First note that actual values of the numbers is immaterial—we want to choose from one of n outcomes with probabilities $p_0, p_1, \ldots, p_{n-1}$. If all probabilities were the same, i.e., $1/n$, we could make a single call to the random number generator, and choose outcome i if the number falls lies between i/n and $(i+1)/n$.

For the case where the probabilities are not the same, we can solve the problem by partitioning the unit interval $[0, 1]$ into n disjoint segments, in a way so that the length of the jth interval is proportional to p_j. Then we select a number uniformly at random in the unit interval, $[0, 1]$, and return the number corresponding to the interval the randomly generated number falls in.

An easy way to create these intervals is to take use $p_0, p_0+p_1, p_0+p_1+p_2, \ldots, p_0+p_1+p_2+\cdots+p_{n-1}$ as the endpoints. Using the example given in the problem statement, the four intervals are $[0.0, 0.5), [0.5, 0.833), [0.833, 0.944), [0.944, 1.0]$. Now, for example, if the random number generated uniformly in $[0.0, 1.0]$ is 0.873, since 0.873 lies in $[0.833, 0.944)$, which is the third interval, we return the third number, which is 7.

In general, searching an array of n disjoint intervals for the interval containing a number takes $O(n)$ time. However, we can do better. Since the array $\langle p_0, p_0 + p_1, p_0 + p_1 + p_2, \ldots, p_0 + p_1 + p_2 + \cdots + p_{n-1} \rangle$ is sorted, we can use binary search to find the interval in $O(\log n)$ time.

```
int NonuniformRandomNumberGeneration(const vector<int>& values,
                                     const vector<double>& probabilities) {
  vector<double> prefix_sums_of_probabilities;
  prefix_sums_of_probabilities.emplace_back(0.0);
  // Creating the endpoints for the intervals corresponding to the
  // probabilities.
  partial_sum(probabilities.cbegin(), probabilities.cend(),
              back_inserter(prefix_sums_of_probabilities));

  default_random_engine seed((random_device())());
  double uniform_0_1 =
      generate_canonical<double, numeric_limits<double>::digits>(seed);
  // Find the index of the interval that uniform_0_1 lies in, which is the
  // return value of upper_bound() minus 1.
  int interval_idx =
      distance(prefix_sums_of_probabilities.cbegin(),
               upper_bound(prefix_sums_of_probabilities.cbegin(),
                           prefix_sums_of_probabilities.cend(), uniform_0_1)) -
      1;
  return values[interval_idx];
}
```

The time complexity to compute a single value is $O(n)$, which is the time to create the array of intervals. This array also implies an $O(n)$ space complexity.

Once the array is constructed, computing each additional result entails one call to the uniform random number generator, followed by a binary search, i.e., $O(\log n)$.

Variant: Given a random number generator that produces values in $[0, 1]$ uniformly, how would you generate a value X from T according to a continuous probability distribution, such as the exponential distribution?

Multidimensional arrays

Thus far we have focused our attention in this chapter on one-dimensional arrays. We now turn our attention to multidimensional arrays. A 2D array in an array whose entries are themselves arrays; the concept generalizes naturally to k dimensional arrays.

Multidimensional arrays arise in image processing, board games, graphs, modeling spatial phenomenon, etc. Often, but not always, the arrays that constitute the entries of a 2D array A have the same length, in which case we refer to A as being an $m \times n$ rectangular array (or sometimes just an $m \times n$ array), where m is the number of entries in A, and n the number of entries in $A[0]$. The elements within a 2D array A are often referred to by their *row* and *column* indices i and j, and written as $A[i][j]$.

6.17 The Sudoku checker problem

Sudoku is a popular logic-based combinatorial number placement puzzle. The objective is to fill a 9×9 grid with digits subject to the constraint that each column, each row, and each of the nine 3×3 sub-grids that compose the grid contains unique integers in $[1, 9]$. The grid is initialized with a partial assignment as shown in Figure 6.2(a); a partial solution is shown in Figure 6.2(b).

5	**3**			**7**				
6			**1**	**9**	**5**			
	9	**8**					**6**	
8				**6**				**3**
4			**8**		**3**			**1**
7				**2**				**6**
	6					**2**	**8**	
			4	**1**	**9**			**5**
				8			**7**	**9**

(a) Partial assignment.

5	**3**	4	6	**7**	8	9	1	2
6	7	2	**1**	**9**	**5**	3	4	8
1	**9**	**8**	3	4	2	5	**6**	7
8	5	9	7	**6**	1	4	2	**3**
4	2	6	**8**	5	**3**	7	9	**1**
7	1	3	9	**2**	4	8	5	**6**
9	**6**	1	5	3	7	**2**	**8**	4
2	8	7	**4**	**1**	**9**	6	3	**5**
3	4	5	2	**8**	6	1	**7**	**9**

(b) A complete solution.

Figure 6.2: Sudoku configurations.

Check whether a 9×9 2D array representing a partially completed Sudoku is valid. Specifically, check that no row, column, and 3×3 2D subarray contains duplicates. A 0-value in the 2D array indicates that entry is blank; every other entry is in $[1, 9]$.

Hint: Directly test the constraints. Use an array to encode sets.

Solution: There is no real scope for algorithm optimization in this problem—it's all about writing clean code.

We need to check nine row constraints, nine column constraints, and nine sub-grid constraints. It is convenient to use bit arrays to test for constraint violations, that is to ensure no number in $[1, 9]$ appears more than once.

```
// Check if a partially filled matrix has any conflicts.
bool IsValidSudoku(const vector<vector<int>>& partial_assignment) {
  // Check row constraints.
  for (int i = 0; i < partial_assignment.size(); ++i) {
    if (HasDuplicate(partial_assignment, i, i + 1, 0,
         partial_assignment.size(),
                     partial_assignment.size())) {
      return false;
    }
  }

  // Check column constraints.
  for (int j = 0; j < partial_assignment.size(); ++j) {
    if (HasDuplicate(partial_assignment, 0, partial_assignment.size(), j, j +
         1,
                     partial_assignment.size())) {
      return false;
    }
  }

  // Check region constraints.
  int region_size = sqrt(partial_assignment.size());
  for (int I = 0; I < region_size; ++I) {
    for (int J = 0; J < region_size; ++J) {
      if (HasDuplicate(partial_assignment, region_size * I,
                       region_size * (I + 1), region_size * J,
                       region_size * (J + 1), partial_assignment.size())) {
        return false;
      }
    }
  }
  return true;
}

// Return true if subarray partial_assignment[start_row : end_row -
     1][start_col
// : end_col - 1]
// contains any duplicates in {1, 2, ..., num_elements}; otherwise return
     false.
bool HasDuplicate(const vector<vector<int>>& partial_assignment, int start_row,
                  int end_row, int start_col, int end_col, int num_elements) {
  deque<bool> is_present(num_elements + 1, false);
  for (int i = start_row; i < end_row; ++i) {
    for (int j = start_col; j < end_col; ++j) {
      if (partial_assignment[i][j] != 0 &&
          is_present[partial_assignment[i][j]]) {
        return true;
      }
      is_present[partial_assignment[i][j]] = true;
```

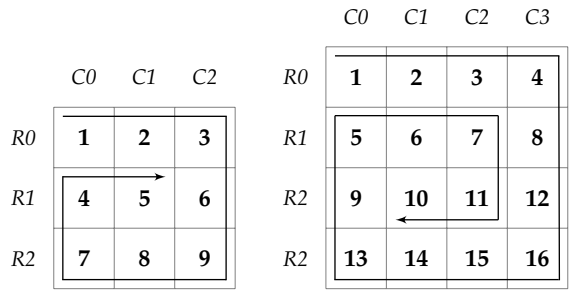
```
    }
  }
  return false;
}
```

The time complexity of this algorithm for an $n \times n$ Sudoku grid with $\sqrt{n} \times \sqrt{n}$ subgrids is $O(n^2)+O(n^2)+O(n^2/(\sqrt{n})^2 \times (\sqrt{n})^2) = O(n^2)$; the terms corresponds to the complexity to check n row constraints, the n column constraints, and the n subgrid constraints, respectively. The memory usage is dominated by the bit array used to check the constraints, so the space complexity is $O(n)$.

Solution 16.9 on Page 282 describes how to solve Sudoku instances.

6.18 Compute the spiral ordering of a 2D array

A 2D array can be written as a sequence in several orders—the most natural ones being row-by-row or column-by-column. In this problem we explore the problem of writing the 2D array in spiral order. For example, the spiral ordering for the 2D array in Figure 6.3(a) is $\langle 1, 2, 3, 6, 9, 8, 7, 4, 5 \rangle$. For Figure 6.3(b), the spiral ordering is $\langle 1, 2, 3, 4, 8, 12, 16, 15, 14, 13, 9, 5, 6, 7, 1, 10 \rangle$.



(a) Spiral ordering for a 3×3 array.

(b) Spiral ordering for a 4×4 array.

Figure 6.3: Spiral orderings. Column and row ids are specified above and to the left of the matrix, i.e., 1 is at entry $(0, 0)$.

Write a program which takes an $n \times n$ 2D array and returns the spiral ordering of the array.

Hint: Use case analysis and divide-and-conquer.

Solution: It is natural to solve this problem starting from the outside, and working to the center. The naive approach begins by adding the first row, which consists of n elements. Next we add the $n-1$ remaining elements of the last column, then the $n-1$ remaining elements of the last row, and then the $n-2$ remaining elements of the first column. The lack of uniformity makes it hard to get the code right.

Here is a uniform way of adding the boundary. Add the first $n-1$ elements of the first row. Then add the first $n-1$ elements of the last column. Then add the last $n-1$

elements of the last row in reverse order. Finally, add the last $n-1$ elements of the first column in reverse order.

After this, we are left with the problem of adding the elements of an $(n-2)\times(n-2)$ 2D array in spiral order. This leads to an iterative algorithm that adds the outermost elements of $n \times n, (n-2) \times (n-2), (n-4) \times (n-4), \ldots$ 2D arrays. Note that a matrix of odd dimension has a corner-case, namely when we reach its center.

As an example, for the 3×3 array in Figure 6.3(a) on the facing page, we would add $1, 2$ (first two elements of the first row), then $3, 6$ (first two elements of the last column), then $9, 8$ (last two elements of the last row), then $7, 4$ (last two elements of the first column). We are now left with the 1×1 array, whose sole element is 5. After processing it, all elements are processed.

For the 4×4 array in Figure 6.3(b) on the preceding page, we would add $1, 2, 3$ (first three elements of the first row), then $4, 8, 12$ (first three elements of the last column), then $16, 15, 14$ (last three elements of the last row), then $13, 9, 5$ (last three elements of the first column). We are now left with a 2×2 matrix, which we process similarly in the order $6, 7, 11, 10$, after which all elements are processed.

```
vector<int> MatrixInSpiralOrder(const vector<vector<int>> &square_matrix) {
  vector<int> spiral_ordering;
  for (int offset = 0; offset < ceil(0.5 * square_matrix.size()); ++offset) {
    MatrixLayerInClockwise(square_matrix, offset, &spiral_ordering);
  }
  return spiral_ordering;
}

void MatrixLayerInClockwise(const vector<vector<int>> &square_matrix,
                            int offset, vector<int> *spiral_ordering) {
  if (offset == square_matrix.size() - offset - 1) {
    // square_matrix has odd dimension, and we are at the center of
    // square_matrix.
    spiral_ordering->emplace_back(square_matrix[offset][offset]);
    return;
  }

  for (int j = offset; j < square_matrix.size() - offset - 1; ++j) {
    spiral_ordering->emplace_back(square_matrix[offset][j]);
  }
  for (int i = offset; i < square_matrix.size() - offset - 1; ++i) {
    spiral_ordering->emplace_back(
        square_matrix[i][square_matrix.size() - offset - 1]);
  }
  for (int j = square_matrix.size() - offset - 1; j > offset; --j) {
    spiral_ordering->emplace_back(
        square_matrix[square_matrix.size() - offset - 1][j]);
  }
  for (int i = square_matrix.size() - offset - 1; i > offset; --i) {
    spiral_ordering->emplace_back(square_matrix[i][offset]);
  }
}
```

The time complexity is $O(n^2)$ and the space complexity is $O(1)$.

An alternate solution writes 0 into array entries to indicate they have been processed, and a `shift` 2D array to compress the four iterations above into a single iteration parameterized by `shift`: The time complexity is $O(n^2)$ and the space complexity is $O(1)$.

```
vector<int> MatrixInSpiralOrder(vector<vector<int>> square_matrix) {
  const array<array<int, 2>, 4> shift = {
      {{{0, 1}}, {{1, 0}}, {{0, -1}}, {{-1, 0}}}};
  int dir = 0, x = 0, y = 0;
  vector<int> spiral_ordering;

  for (int i = 0; i < square_matrix.size() * square_matrix.size(); ++i) {
    spiral_ordering.emplace_back(square_matrix[x][y]);
    square_matrix[x][y] = 0;
    int next_x = x + shift[dir][0], next_y = y + shift[dir][1];
    if (next_x < 0 || next_x >= square_matrix.size() || next_y < 0 ||
        next_y >= square_matrix.size() || square_matrix[next_x][next_y] == 0) {
      dir = (dir + 1) % 4;
      next_x = x + shift[dir][0], next_y = y + shift[dir][1];
    }
    x = next_x, y = next_y;
  }
  return spiral_ordering;
}
```

Variant: Given a dimension d, write a program to generate a $d \times d$ 2D array which in spiral order is $\langle 1, 2, 3, \ldots, d^2 \rangle$. For example, if $d = 3$, the result should be

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 8 & 9 & 4 \\ 7 & 6 & 5 \end{bmatrix}.$$

Variant: Given a sequence of integers P, compute a 2D array A whose spiral order is P. (Assume $|P|$ for some integer n.)

Variant: Write a program to enumerate the first n pairs of integers (a, b) in spiral order, starting from $(0, 0)$ followed by $(1, 0)$. For example, if $n = 10$, your output should be $(0, 0), (1, 0), (1, -1), (0, -1), (-1, -1), (-1, 0), (-1, 1), (0, 1), (1, 1), (2, 1)$.

Variant: Compute the spiral order for an $m \times n$ 2D array A.

Variant: Compute the last element in spiral order for an $m \times n$ 2D array A in $O(1)$ time.

Variant: Compute the kth element in spiral order for an $m \times n$ 2D array A in $O(1)$ time.

6.19 Rotate a 2D array

Image rotation is a fundamental operation in computer graphics. Figure 6.4 on the facing page illustrates the rotation operation on a 2D array representing a bit-map of

an image. Specifically, the image is rotated by 90 degrees clockwise.

1	**2**	**3**	**4**
5	**6**	**7**	**8**
9	**10**	**11**	**12**
13	**14**	**15**	**16**

(a) Initial 4×4 2D array.

13	**9**	**5**	**1**
14	**10**	**6**	**2**
15	**11**	**7**	**3**
16	**12**	**8**	**4**

(b) Array rotated by 90 degrees clockwise.

Figure 6.4: Example of 2D array rotation.

Write a function that takes as input an $n \times n$ 2D array, and rotates the array by 90 degrees clockwise. Assume that $n = 2^k$ for some positive integer k.

Hint: Focus on the boundary elements.

Solution: With a little experimentation, it is easy to see that ith column of the rotate matrix is the ith row of the original matrix. For example, the first row, $\langle 13, 14, 15, 16 \rangle$ of the initial array in 6.4 becomes the first column in the rotated version. Therefore, a brute-force approach is to allocate a new $n \times n$ 2D array, write the rotation to it (writing rows of the original matrix into the columns of the new matrix), and then copying the new array back to the original one. The last step is needed since the problem specified we were to update the original array. The time and additional space complexity are both $O(n^2)$.

Since we are not explicitly required to allocate a new array, it is natural to ask if we can perform the rotation inplace, i.e., with $O(1)$ additional storage. The first insight is that we can perform the rotation in a layer-by-layer fashion—different layers can be processed independently. Furthermore, within a layer, we can exchange groups of four elements at a time to perform the rotation, e.g., send 1 to 4's location, 4 to 16's location, 16 to 13's location, and 13 to 1's location, then send 2 to 8's location, 8 to 15's location, 15 to 9's location, and 9 to 2's location, etc. The program below works its way into the center of the array from the outermost layers, performing exchanges within a layer iteratively using the four-way swap just described.

```
void RotateMatrix(vector<vector<int>>* square_matrix_ptr) {
  vector<vector<int>>& square_matrix = *square_matrix_ptr;
  for (int i = 0; i < (square_matrix.size() / 2); ++i) {
    for (int j = i; j < square_matrix.size() - 1 - i; ++j) {
      // Perform a 4-way exchange.
      int temp = square_matrix[i][j];
      square_matrix[i][j] = square_matrix[square_matrix.size() - 1 - j][i];
      square_matrix[square_matrix.size() - 1 - j][i] =
          square_matrix[square_matrix.size() - 1 - i][square_matrix.size() - 1
              -
                                                      j];
      square_matrix[square_matrix.size() - 1 - i][square_matrix.size() - 1 -
                                                  j] =
          square_matrix[j][square_matrix.size() - 1 - i];
```

```
      square_matrix[j][square_matrix.size() - 1 - i] = temp;
    }
  }
}
```

The time complexity is $O(n^2)$ and the additional space complexity is $O(1)$.

Interestingly, we can perform effectively rotate in $O(1)$ space and time complexity, albeit with some limitations. Specifically, we return an object r that composes the original matrix A. A read of the element at indices i and j in r is converted into a read from A at index $[n-1-j][i]$. Writes are handled similarly. The time to create r is constant, since it simply consists of a reference to A. The time to perform reads and writes is unchanged. This approach breaks when there are clients of the original A object, since writes to r change A. Even if A is not written to, if methods on the stored objects change their state, the system gets corrupted. Copy-on-write can be used to solve these issues.

```
class RotatedMatrix {
 public:
  explicit RotatedMatrix(const vector<vector<int>>& square_matrix)
      : square_matrix_(square_matrix) {}

  int ReadEntry(int i, int j) const {
    return square_matrix_[square_matrix_.size() - 1 - j][i];
  }

  void WriteEntry(int i, int j, int v) {
    square_matrix_[square_matrix_.size() - 1 - j][i] = v;
  }

 private:
  vector<vector<int>> square_matrix_;
};
```

Variant: Implement an algorithm to reflect A, assumed to be an $n \times n$ 2D array, about the horizontal axis of symmetry. Repeat the same for reflections about the vertical axis, the diagonal from top-left to bottom-right, and the diagonal from top-right to bottom-left. Assume that $n = 2^k$ for some positive integer k.

6.20 Compute rows in Pascal's Triangle

Figure 6.5 on the facing page shows the first five rows of a graphic that is known as Pascal's triangle. Each row contains one more entry than the previous one. Except for entries in the last row, each entry is adjacent to one or two numbers in the row below it. The first row holds 1. Each entry holds the sum of the numbers in the adjacent entries above it.

Write a program which takes as input a nonnegative integer n and returns the first n rows of Pascal's triangle.

Hint: Write the given fact as an equation.

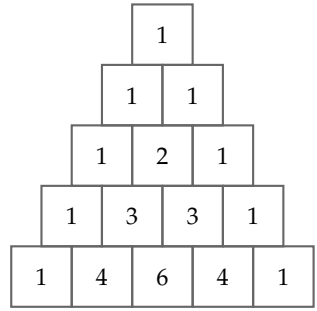


Figure 6.5: A Pascal triangle.

Solution: A brute-force approach might be to organize the arrays in memory similar to how they appear in the figure. The challenge is to determine the correct indices to range over and to read from.

A better approach is to keep the arrays left-aligned, that is the first entry is at location 0. Now it is simple: the jth entry in the ith row is 1 if $j = 0$ or $j = i$, otherwise it is the sum of the $(j-1)$th and jth entries in the $(i-1)$th row. The first row R_0 is $\langle 1 \rangle$. The second row R_1 is $\langle 1, 1 \rangle$. The third row R_2 is $\langle 1, R_1[0] + R_1[1] = 2, 1 \rangle$. The fourth row R_3 is $\langle 1, R_2[0] + R_2[1] = 3, R_2[1] + R_2[2] = 3, 1 \rangle$.

```
vector<vector<int>> GeneratePascalTriangle(int num_rows) {
  vector<vector<int>> pascal_triangle;
  for (int i = 0; i < num_rows; ++i) {
    vector<int> curr_row;
    for (int j = 0; j <= i; ++j) {
      // Sets this entry to the sum of the two above adjacent entries if they
      // exist.
      curr_row.emplace_back((0 < j && j < i)
                            ? pascal_triangle.back()[j - 1] +
                                  pascal_triangle.back()[j]
                            : 1);
    }
    pascal_triangle.emplace_back(curr_row);
  }
  return pascal_triangle;
}
```

Since each element takes $O(1)$ time to compute, the time complexity is $O(1+2+\cdots+n) = O(n(n+1)/2) = O(n^2)$. Similarly, the space complexity is $O(n^2)$.

It is a fact that the ith entry in the nth row of Pascal's triangle is $\binom{n}{i}$. This in itself does not trivialize the problem, since computing $\binom{n}{i}$ itself is tricky. (In fact, Pascal's triangle can be used to compute $\binom{n}{i}$.)

Variant: Compute the nth row of Pascal's triangle using $O(n)$ space.