9. **UVa 10038 - Jolly Jumpers *** (use 1D boolean flags to check $[1..n-1]$)
10. UVa 10050 - Hartals (1D boolean flag)
11. UVa 10260 - Soundex (Direct Addressing Table for soundex code mapping)
12. UVa 10978 - Let's Play Magic (1D string array manipulation)
13. *UVa 11093 - Just Finish it up* (linear scan, circular array, a bit challenging)
14. UVa 11192 - Group Reverse (character array)
15. UVa 11222 - Only I did it (use several 1D arrays to simplify this problem)
16. **UVa 11340 - Newspaper *** (DAT; see Hashing in Section 2.3)
17. UVa 11496 - Musical Loop (store data in 1D array, count the peaks)
18. UVa 11608 - No Problem (use three arrays: created; required; available)
19. UVa 11850 - Alaska (for each integer location from 0 to 1322; can Brenda reach (anywhere within 200 miles of) any charging stations?)
20. *UVa 12150 - Pole Position* (simple manipulation)
21. ***UVa 12356 - Army Buddies **** (similar to deletion in doubly linked lists, but we can still use a 1D array for the underlying data structure)

- 2D Array Manipulation
  1. UVa 00101 - The Blocks Problem ('stack' like simulation; but we need to access the content of each stack too, so it is better to use 2D array)
  2. UVa 00434 - Matty's Blocks (a kind of visibility problem in geometry, solvable with using 2D array manipulation)
  3. UVa 00466 - Mirror Mirror (core functions: rotate and reflect)
  4. UVa 00541 - Error Correction (count the number of '1's for each row/col; all of them must be even; if $\exists$ an error, check if it is on the same row and col)
  5. UVa 10016 - Flip-flop the Squarelotron (tedious)
  6. UVa 10703 - Free spots (use 2D boolean array of size 500×500)
  7. **UVa 10855 - Rotated squares *** (string array, 90^o clockwise rotation)
  8. **UVa 10920 - Spiral Tap *** (simulate the process)
  9. UVa 11040 - Add bricks in the wall (non trivial 2D array manipulation)
  10. UVa 11349 - Symmetric Matrix (use long long to avoid issues)
  11. UVa 11360 - Have Fun with Matrices (do as asked)
  12. **UVa 11581 - Grid Successors *** (simulate the process)
  13. UVa 11835 - Formula 1 (do as asked)
  14. *UVa 12187 - Brothers* (simulate the process)
  15. *UVa 12291 - Polyomino Composer* (do as asked, a bit tedious)
  16. *UVa 12398 - NumPuzz I* (simulate backwards, do not forget to mod 10)
- C++ STL `algorithm` (Java `Collections`)
  1. UVa 00123 - Searching Quickly (modified comparison function, use `sort`)
  2. **UVa 00146 - ID Codes *** (use `next_permutation`)
  3. UVa 00400 - Unix ls (this command very frequently used in UNIX)
  4. UVa 00450 - Little Black Book (tedious sorting problem)
  5. *UVa 00790 - Head Judge Headache* (similar to UVa 10258)
  6. UVa 00855 - Lunch in Grid City (sort, median)
  7. UVa 01209 - Wordfish (LA 3173, Manila06) (STL `next` and `prev_permutation`)
  8. *UVa 10057 - A mid-summer night ...* (involves the median, use STL `sort`, `upper_bound`, `lower_bound` and some checks)

9. **UVa 10107 - What is the Median? *** (find median of a *growing*/dynamic list of integers; still solvable with multiple calls of `nth_element` in `algorithm`)
10. UVa 10194 - Football a.k.a. Soccer (multi-fields sorting, use `sort`)
11. **UVa 10258 - Contest Scoreboard *** (multi-fields sorting, use `sort`)
12. *UVa 10698 - Football Sort* (multi-fields sorting, use `sort`)
13. UVa 10880 - Colin and Ryan (use `sort`)
14. UVa 10905 - Children's Game (modified comparison function, use `sort`)
15. UVa 11039 - Building Designing (use `sort` then count different signs)
16. UVa 11321 - Sort Sort and Sort (be careful with negative mod!)
17. UVa 11588 - Image Coding (`sort` simplifies the problem)
18. UVa 11777 - Automate the Grades (`sort` simplifies the problem)
19. UVa 11824 - A Minimum Land Price (`sort` simplifies the problem)
20. *UVa 12541 - Birthdates* (LA6148, HatYai12, `sort`, pick youngest and oldest)

- Bit Manipulation (both C++ STL `bitset` (Java `BitSet`) and bitmask)
  1. UVa 00594 - One Little, Two Little ... (manipulate bit string with `bitset`)
  2. UVa 00700 - Date Bugs (can be solved with `bitset`)
  3. UVa 01241 - Jollybee Tournament (LA 4147, Jakarta08, easy)
  4. ***UVa 10264 - The Most Potent Corner ****** (heavy bitmask manipulation)
  5. *UVa 11173 - Grey Codes* (D & C pattern or one liner bit manipulation)
  6. UVa 11760 - Brother Arif, ... (separate row+col checks; use two bitsets)
  7. **UVa 11926 - Multitasking *** (use 1M `bitset` to check if a slot is free)
  8. **UVa 11933 - Splitting Numbers *** (an exercise for bit manipulation)
  9. IOI 2011 - Pigeons (this problem becomes simpler with bit manipulation but the final solution requires much more than that.)
- C++ STL `list` (Java `LinkedList`)
  1. **UVa 11988 - Broken Keyboard ... *** (rare linked list problem)
- C++ STL `stack` (Java `Stack`)
  1. UVa 00127 - "Accordian" Patience (shuffling `stack`)
  2. **UVa 00514 - Rails *** (use `stack` to simulate the process)
  3. **UVa 00732 - Anagram by Stack *** (use `stack` to simulate the process)
  4. ***UVa 01062 - Containers ****** (LA 3752, WorldFinals Tokyo07, simulation with `stack`; maximum answer is 26 stacks; $O(n)$ solution exists)
  5. UVa 10858 - Unique Factorization (use `stack` to help solving this problem)

  Also see: implicit `stacks` in recursive function calls and Postfix conversion/evaluation in Section 9.27.
- C++ STL `queue` and `deque` (Java `Queue` and `Deque`)
  1. UVa 00540 - Team Queue (modified 'queue')
  2. **UVa 10172 - The Lonesome Cargo ... *** (use both `queue` and `stack`)
  3. **UVa 10901 - Ferry Loading III *** (simulation with `queue`)
  4. UVa 10935 - Throwing cards away I (simulation with `queue`)
  5. **UVa 11034 - Ferry Loading IV *** (simulation with `queue`)
  6. *UVa 12100 - Printer Queue* (simulation with `queue`)
  7. *UVa 12207 - This is Your Queue* (use both `queue` and `deque`)

  Also see: `queues` in BFS (see Section 4.2.2)

2.3 Non-Linear DS with Built-in Libraries

For some problems, linear storage is not the best way to organize data. With the efficient implementations of non-linear data structures shown below, you can operate on the data in a quicker fashion, thereby speeding up the algorithms that rely on them.

For example, if you need a *dynamic*[11] collection of pairs (e.g. key → value pairs), using C++ STL `map` below can provide you $O(\log n)$ performance for insertion/search/deletion operations with just a few lines of code (that you still have to write yourself), whereas storing the same information inside a static array of `struct`s may require $O(n)$ insertion/search/deletion, and you will need to write the longer traversal code yourself.

- Balanced Binary Search Tree (BST): C++ STL `map/set` (Java `TreeMap/TreeSet`)
  The BST is one way to organize data in a tree structure. In each subtree rooted at x, the following BST property holds: Items on the left subtree of x are smaller than x and items on the right subtree of x are greater than (or equal to) x. This is essentially an application of the Divide and Conquer strategy (also see Section 3.3). Organizing the data like this (see Figure 2.2) allows for $O(\log n)$ `search(key)`, `insert(key)`, `findMin()/findMax()`, `successor(key)/predecessor(key)`, and `delete(key)` since in the worst case, only $O(\log n)$ operations are required in a root-to-leaf scan (see [7, 5, 54, 12] for details). However, this only holds if the BST is balanced.

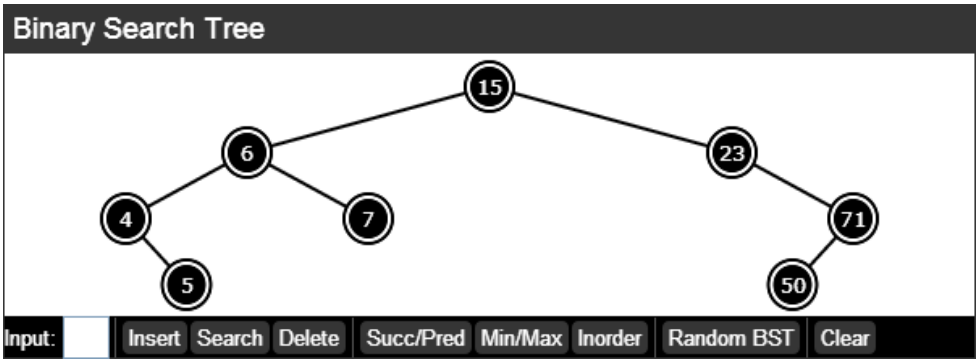

Figure 2.2: Examples of BST

Implementing *bug-free* balanced BSTs such as the Adelson-Velskii Landis (AVL)[12] or Red-Black (RB)[13] Trees is a tedious task and is difficult to achieve in a time-constrained contest environment (unless you have prepared a code library beforehand, see Section 9.29). Fortunately, C++ STL has `map` and `set` (and Java has `TreeMap` and `TreeSet`) which are *usually* implementations of the RB Tree which guarantees that major BST operations like insertions/searches/deletions are done in $O(\log n)$ time. By mastering these two C++ STL template classes (or Java APIs), you can save a lot of precious coding time during contests! The difference between these two data structures is simple: the C++ STL `map` (and Java `TreeMap`) stores (key → data) pairs whereas the C++

[11] The contents of a dynamic data structure is frequently modified via insert/delete/update operations.

[12] The AVL tree was the first self-balancing BST to be invented. AVL trees are essentially traditional BSTs with an additional property: The heights of the two subtrees of any vertex in an AVL tree can differ *by at most one*. Rebalancing operations (rotations) are performed (when necessary) during insertions and deletions to maintain this invariant property, hence keeping the tree roughly balanced.

[13] The Red-Black tree is another self-balancing BST, in which every vertex has a color: red or black. In RB trees, the root vertex, all leaf vertices, and both children of every red vertex are black. Every simple path from a vertex to any of its descendant leaves contains *the same number of black vertices*. Throughout insertions and deletions, an RB tree will maintain all these invariants to keep the tree balanced.

STL `set` (and Java `TreeSet`) only stores the key. For most (contest) problems, we use a `map` (to really map information) instead of a `set` (a `set` is only useful for efficiently determining the existence of a certain key). However, there is a small drawback. If we use the library implementations, it becomes difficult or impossible to augment (add extra information to) the BST. Please attempt **Exercise 2.3.5*** and read Section 9.29 for more details.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/bst.html`

Source code: `ch2_05_map_set.cpp/java`

- Heap: C++ STL `priority_queue` (Java `PriorityQueue`)
  The heap is another way to organize data in a tree. The (Binary) Heap is also a binary tree like the BST, except that it must be a *complete*[14] tree. Complete binary trees can be stored efficiently in a compact 1-indexed array of size $n + 1$, which is often preferred to an explicit tree representation. For example, the array A = {N/A, 90, 19, 36, 17, 3, 25, 1, 2, 7} is the compact array representation of Figure 2.3 with index 0 ignored. One can navigate from a certain index (vertex) i to its parent, left child, and right child by using simple index manipulation: $\lfloor \frac{i}{2} \rfloor$, $2 \times i$, and $2 \times i + 1$, respectively. These index manipulations can be made faster using bit manipulation techniques (see Section 2.2): `i >> 1`, `i << 1`, and `(i << 1) + 1`, respectively.

  Instead of enforcing the BST property, the (Max) Heap enforces the Heap property: in each subtree rooted at x, items on the left **and** right subtrees of x are smaller than (or equal to) x (see Figure 2.3). This is also an application of the Divide and Conquer concept (see Section 3.3). The property guarantees that the top (or root) of the heap is always the maximum element. There is no notion of a 'search' in the Heap (unlike BSTs). The Heap instead allows for the fast extraction (deletion) of the maximum element: `ExtractMax()` and insertion of new items: `Insert(v)`—both of which can be easily achieved by in a $O(\log n)$ root-to-leaf or leaf-to-root traversal, performing swapping operations to maintain the (Max) Heap property whenever necessary (see [7, 5, 54, 12] for details).

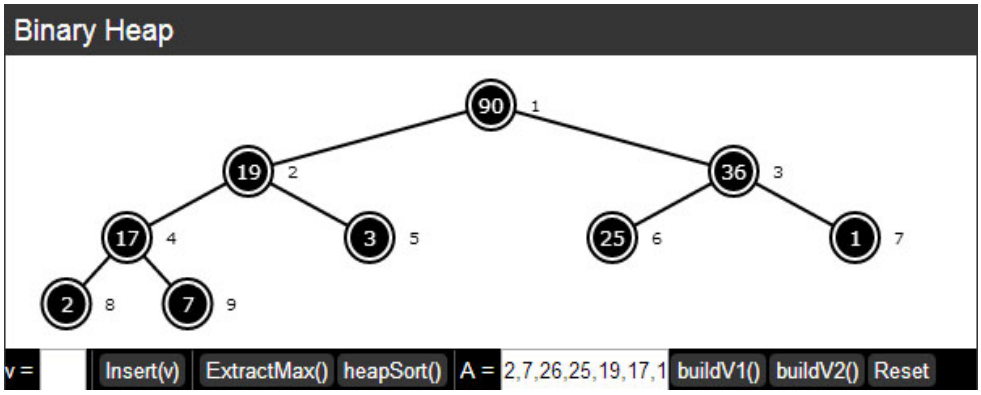

Figure 2.3: (Max) Heap Visualization

The (Max) Heap is a useful data structure for modeling a Priority Queue, where the item with the highest priority (the maximum element) can be dequeued (`ExtractMax()`)

[14] A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled. All vertices in the last level must also be filled from left-to-right.

and a new item v can be enqueued (`Insert(v)`), both in $O(\log n)$ time. The implementation[15] of `priority_queue` is available in the C++ STL `queue` library (or Java `PriorityQueue`). Priority Queues are an important component in algorithms like Prim's (and Kruskal's) algorithms for the Minimum Spanning Tree (MST) problem (see Section 4.3), Dijkstra's algorithm for the Single-Source Shortest Paths (SSSP) problem (see Section 4.4.3), and the A* Search algorithm (see Section 8.2.5).

This data structure is also used to perform `partial_sort` in the C++ STL `algorithm` library. One possible implementation is by processing the elements one by one and creating a Max[16] Heap of k elements, removing the largest element whenever its size exceeds k (k is the number of elements requested by user). The smallest k elements can then be obtained in descending order by dequeuing the remaining elements in the Max Heap. As each dequeue operation is $O(\log k)$, `partial_sort` has $O(n \log k)$ time complexity[17]. When $k = n$, this algorithm is equivalent to a heap sort. Note that although the time complexity of a heap sort is also $O(n \log n)$, heap sorts are often slower than quick sorts because heap operations access data stored in distant indices and are thus not cache-friendly.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/heap.html`

Source code: `ch2_06_priority_queue.cpp/java`

- Hash Table: C++11 STL `unordered_map`[18] (and Java `HashMap`/`HashSet`/`HashTable`)
  The Hash Table is another non-linear data structure, but we do not recommend using it in programming contests unless absolutely necessary. Designing a well-performing hash function is often tricky and only the new C++11 has STL support for it (Java has Hash-related classes).

  Moreover, C++ STL `map`s or `set`s (and Java `TreeMap`s or `TreeSet`s) are usually fast enough as the typical input size of (programming contest) problems is usually not more than $1M$. Within these bounds, the $O(1)$ performance of Hash Tables and $O(\log 1M)$ performance for balanced BSTs do not differ by much. Thus, we do not discuss Hash Tables in detail in this section.

  However, a simple form of Hash Tables can be used in programming contests. 'Direct Addressing Tables' (DATs) can be considered to be Hash Tables where the keys themselves are the indices, or where the 'hash function' is the identity function. For example, we may need to assign all possible ASCII characters [0-255] to integer values, e.g. 'a' → '3', 'W' → '10', ..., 'I' → '13'. For this purpose, we do not need the C++ STL `map` or any form of hashing as the key itself (the value of the ASCII character) is unique and sufficient to determine the appropriate index in an array of size 256. Some programming exercises involving DATs are listed in the previous Section 2.2.

[15]The default C++ STL `priority_queue` is a Max Heap (dequeuing yields items in descending key order) whereas the default Java `PriorityQueue` is a Min Heap (yields items in ascending key order). Tips: A Max Heap containing numbers can easily be converted into a Min Heap (and vice versa) by inserting the negated keys. This is because negating a set of numbers will reverse their order of appearance when sorted. This trick is used several times in this book. However, if the priority queue is used to store *32-bit signed integers*, an overflow will occur if -2^{31} is negated as $2^{31} - 1$ is the maximum value of a 32-bit signed integer.

[16]The default `partial_sort` produces the smallest k elements in ascending order.

[17]You may have noticed that the time complexity $O(n \log k)$ where k is the output size and n is the input size. This means that the algorithm is 'output-sensitive' since its running time depends not only on the input size but also on the amount of items that it has to output.

[18]Note that C++11 is a new C++ standard, older compilers may not support it yet.

Exercise 2.3.1: Someone suggested that it is possible to store the key → value pairs in a *sorted array* of `structs` so that we can use the $O(\log n)$ binary search for the example problem above. Is this approach feasible? If no, what is the issue?

Exercise 2.3.2: We will not discuss the basics of BST operations in this book. Instead, we will use a series of sub-tasks to verify your understanding of BST-related concepts. We will use Figure 2.2 as an *initial reference* in all sub-tasks except sub-task 2.

1. Display the steps taken by `search(71)`, `search(7)`, and then `search(22)`.
2. Starting with an *empty* BST, display the steps taken by `insert(15)`, `insert(23)`, `insert(6)`, `insert(71)`, `insert(50)`, `insert(4)`, `insert(7)`, and `insert(5)`.
3. Display the steps taken by `findMin()` (and `findMax()`).
4. Indicate the *inorder traversal* of this BST. Is the output sorted?
5. Display the steps taken by `successor(23)`, `successor(7)`, and `successor(71)`.
6. Display the steps taken by `delete(5)` (a leaf), `delete(71)` (an internal node with one child), and then `delete(15)` (an internal node with two children).

Exercise 2.3.3*: Suppose you are given a reference to the root R of a binary tree T containing n vertices. You can access a node's left, right and parent vertices as well as its key through its reference. Solve each of the following tasks below with the best possible algorithms that you can think of and analyze their time complexities. Let's assume the following constraints: $1 \leq n \leq 100K$ so that $O(n^2)$ solutions are theoretically infeasible in a contest environment.

1. Check if T is a BST.

2*. Output the elements in T that are within a given range $[a..b]$ in ascending order.

3*. Output the contents of the *leaves* of T in *descending order*.

Exercise 2.3.4*: The inorder traversal (also see Section 4.7.2) of a standard (not necessarily balanced) BST is known to produce the BST's element in sorted order and runs in $O(n)$. Does the code below also produce the BST elements in sorted order? Can it be made to run in a total time of $O(n)$ instead of $O(\log n + (n-1) \times \log n) = O(n \log n)$? If possible, how?

```
x = findMin(); output x
for (i = 1; i < n; i++)                    // is this loop O(n log n)?
  x = successor(x); output x
```

Exercise 2.3.5*: Some (hard) problems require us to write *our own* balanced Binary Search Tree (BST) implementations due to the need to augment the BST with additional data (see Chapter 14 of [7]). Challenge: Solve UVa 11849 - CD which is a pure balanced BST problem with *your own* balanced BST implementation to test its performance and correctness.

Exercise 2.3.6: We will not discuss the basics of Heap operations in this book. Instead, we will use a series of questions to verify your understanding of Heap concepts.

1. With Figure 2.3 as the initial heap, display the steps taken by `Insert(26)`.
2. After answering question 1 above, display the steps taken by `ExtractMax()`.

Exercise 2.3.7: Is the structure represented by a 1-based compact array (ignoring index 0) sorted in descending order a Max Heap?

Exercise 2.3.8*: Prove or disprove this statement: "The second largest element in a Max Heap with $n \geq 3$ distinct elements is always one of the direct children of the root". Follow up question: What about the third largest element? Where is/are the potential location(s) of the third largest element in a Max Heap?

Exercise 2.3.9*: Given a 1-based compact array A containing n integers ($1 \leq n \leq 100K$) that are guaranteed to satisfy the Max Heap property, output the elements in A that are greater than an integer v. What is the best algorithm?

Exercise 2.3.10*: Given an unsorted array S of n distinct integers ($2k \leq n \leq 100000$), find the largest and smallest k ($1 \leq k \leq 32$) integers in S in $O(n \log k)$. Note: For this written exercise, assume that an $O(n \log n)$ algorithm is *not* acceptable.

Exercise 2.3.11*: One heap operation *not* directly supported by the C++ STL `priority_queue` (and Java `PriorityQueue`) is the `UpdateKey(index, newKey)` operation, which allows the (Max) Heap element at a certain index to be updated (increased or decreased). Write *your own* binary (Max) Heap implementation with this operation.

Exercise 2.3.12*: Another heap operation that may be useful is the `DeleteKey(index)` operation to delete (Max) Heap elements at a certain index. Implement this!

Exercise 2.3.13*: Suppose that we only need the `DecreaseKey(index, newKey)` operation, i.e. an `UpdateKey` operation where the update *always* makes `newKey` smaller than its previous value. Can we use a simpler approach than in **Exercise 2.3.11**? Hint: Use lazy deletion, we will use this technique in our Dijkstra code in Section 4.4.3.

Exercise 2.3.14*: Is it possible to use a balanced BST (e.g. C++ STL `set` or Java `TreeSet`) to implement a Priority Queue with the same $O(\log n)$ enqueue and dequeue performance? If yes, how? Are there any potential drawbacks? If no, why?

Exercise 2.3.15*: Is there a better way to implement a Priority Queue if the keys are all integers within a small range, e.g. $[0 \ldots 100]$? We are expecting an $O(1)$ enqueue and dequeue performance. If yes, how? If no, why?

Exercise 2.3.16: Which non-linear data structure should you use if you have to support the following three dynamic operations: 1) many insertions, 2) many deletions, and 3) many requests for the data in sorted order?

Exercise 2.3.17: There are M **strings**. N of them are unique ($N \leq M$). Which non-linear data structure discussed in this section should you use if you have to index (label) these **M** strings with integers from `[0..N-1]`? The indexing criteria is as follows: The first string must be given an index of 0; The next different string must be given index 1, and so on. However, if a string is re-encountered, it must be given the same index as its earlier copy! One application of this task is in constructing the connection graph from a list of city names (which are not integer indices!) and a list of highways between these cities (see Section 2.4.1). To do this, we first have to map these city names into integer indices (which are far more efficient to work with).

Programming exercises solvable with library of non-linear data structures:

- C++ STL `map` (and Java `TreeMap`)
  1. UVa 00417 - Word Index (generate all words, add to `map` for auto sorting)
  2. UVa 00484 - The Department of ... (maintain frequency with `map`)
  3. UVa 00860 - Entropy Text Analyzer (frequency counting)
  4. *UVa 00939 - Genes* (`map` child name to his/her gene and parents' names)
  5. *UVa 10132 - File Fragmentation* (N = number of fragments, B = total bits of all fragments divided by $N/2$; try all $2 \times N^2$ concatenations of two fragments that have length B; report the one with highest frequency; use `map`)
  6. *UVa 10138 - CDVII* (map plates to bills, entrace time and position)
  7. **UVa 10226 - Hardwood Species *** (use hashing for a better performance)
  8. UVa 10282 - Babelfish (a pure dictionary problem; use `map`)
  9. UVa 10295 - Hay Points (use `map` to deal with Hay Points dictionary)
  10. UVa 10686 - SQF Problem (use `map` to manage the data)
  11. UVa 11239 - Open Source (use `map` and `set` to check previous strings)
  12. **UVa 11286 - Conformity *** (use `map` to keep track of the frequencies)
  13. UVa 11308 - Bankrupt Baker (use `map` and `set` to help manage the data)
  14. *UVa 11348 - Exhibition* (use `map` and `set` to check uniqueness)
  15. ***UVa 11572 - Unique Snowflakes ****** (use `map` to record the occurrence index of a certain snowflake size; use this to determine the answer in $O(n \log n)$)
  16. UVa 11629 - Ballot evaluation (use `map`)
  17. UVa 11860 - Document Analyzer (use `set` and `map`, linear scan)
  18. UVa 11917 - Do Your Own Homework (use `map`)
  19. *UVa 12504 - Updating a Dictionary* (use `map`; string to string; a bit tedious)
  20. *UVa 12592 - Slogan Learning of Princess* (use `map`; string to string)

  Also check frequency counting section in Section 6.3.
- C++ STL `set` (Java `TreeSet`)
  1. UVa 00501 - Black Box (use `multiset` with efficient iterator manipulation)
  2. **UVa 00978 - Lemmings Battle *** (simulation, use `multiset`)
  3. UVa 10815 - Andy's First Dictionary (use `set` and `string`)
  4. UVa 11062 - Andy's Second Dictionary (similar to UVa 10815, with twists)
  5. **UVa 11136 - Hoax or what *** (use `multiset`)
  6. **UVa 11849 - CD *** (use `set` to pass the time limit, better: use hashing!)
  7. *UVa 12049 - Just Prune The List* (`multiset` manipulation)
- C++ STL `priority_queue` (Java `PriorityQueue`)
  1. **UVa 01203 - Argus *** (LA 3135, Beijing04; use `priority_queue`)
  2. **UVa 10954 - Add All *** (use `priority_queue`, greedy)
  3. **UVa 11995 - I Can Guess ... *** (`stack`, `queue`, and `priority_queue`)

  Also see the usage of `priority_queue` for topological sorts (see Section 4.2.1), Kruskal's[19] (see Section 4.3.2), Prim's (see Section 4.3.3), Dijkstra's (see Section 4.4.3), and the A* Search algorithms (see Section 8.2.5)

[19]This is another way to implement the edge sorting in Kruskal's algorithm. Our (C++) implementation shown in Section 4.3.2 simply uses `vector` + `sort` instead of `priority_queue` (a heap sort).

2.4 Data Structures with Our Own Libraries

As of 24 May 2013, important data structures shown in this section do not have built-in support yet in C++ STL or Java API. Thus, to be competitive, contestants should prepare bug-free implementations of these data structures. In this section, we discuss the key ideas and example implementations (see the given source code too) of these data structures.

2.4.1 Graph

The graph is a pervasive structure which appears in many Computer Science problems. A graph ($G = (V, E)$) in its basic form is simply a set of vertices (V) and edges (E; storing connectivity information between vertices in V). Later in Chapter 3, 4, 8, and 9, we will explore many important graph problems and algorithms. To prepare ourselves, we will discuss three basic ways (there are a few other rare structures) to represent a graph G with V vertices and E edges in this subsection[20].

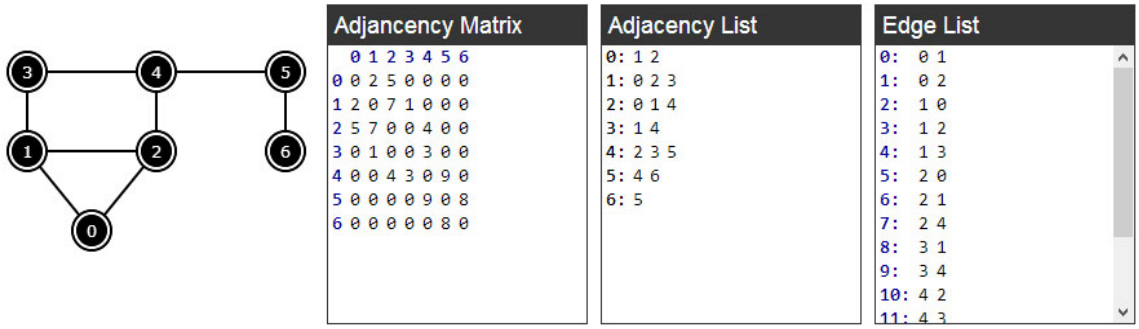

Figure 2.4: Graph Data Structure Visualization

A). The Adjacency Matrix, usually in the form of a 2D array (see Figure 2.4).

In (programming contest) problems involving graphs, the number of vertices V is usually known. Thus we can build a 'connectivity table' by creating a static 2D array: `int AdjMat[V][V]`. This has an $O(V^2)$ *space*[21] complexity. For an unweighted graph, set `AdjMat[i][j]` to a non-zero value (usually 1) if there is an edge between vertex i-j or zero otherwise. For a weighted graph, set `AdjMat[i][j] = weight(i,j)` if there is an edge between vertex i-j with `weight(i,j)` or zero otherwise. Adjacency Matrix cannot be used to store multigraph. For a simple graph without self-loops, the main diagonal of the matrix contains only zeroes, i.e. `AdjMat[i][i] = 0, ∀i ∈ [0..V-1]`.

An Adjacency Matrix is a good choice if the connectivity between two vertices in a *small dense graph* is frequently required. However, it is not recommended for *large sparse graphs* as it would require too much space ($O(V^2)$) and there would be many blank (zero) cells in the 2D array. In a competitive setting, it is usually infeasible to use Adjacency Matrices when the given V is larger than ≈ 1000. Another drawback of Adjacency Matrix is that it also takes $O(V)$ time to enumerate the list of neighbors of a vertex v—an operation common to many graph algorithms—even if a vertex only has a handful of neighbors. A more compact and efficient graph representation is the Adjacency List discussed below.

[20]The most appropriate notation for the cardinality of a set S is $|S|$. However, in this book, we will often overload the meaning of V or E to also mean $|V|$ or $|E|$, depending on the context.

[21]We differentiate between the *space* and *time* complexities of data structures. The *space* complexity is an asymptotic measure of the memory requirements of a data structure whereas the *time* complexity is an asymptotic measure of the time taken to run a certain algorithm or an operation on the data structure.

B). The Adjacency List, usually in the form of a vector of vector of pairs (see Figure 2.4).
Using the C++ STL: `vector<vii> AdjList`, with `vii` defined as in:
`typedef pair<int, int> ii; typedef vector<ii> vii; // data type shortcuts`
Using the Java API: `Vector< Vector < IntegerPair > > AdjList`.
`IntegerPair` is a simple Java class that contains a pair of integers like `ii` above.

In Adjacency Lists, we have a `vector` of `vector` of pairs, storing the list of neighbors of each vertex u as 'edge information' pairs. Each pair contains two pieces of information, the index of the neighbouring vertex and the weight of the edge. If the graph is unweighted, simply store the weight as 0, 1, or drop the weight attribute[22] entirely. The space complexity of Adjacency List is $O(V + E)$ because if there are E bidirectional edges in a (simple) graph, this Adjacency List will only store $2E$ 'edge information' pairs. As E is usually much smaller than $V \times (V-1)/2 = O(V^2)$—the maximum number of edges in a complete (simple) graph, Adjacency Lists are often more space-efficient than Adjacency Matrices. Note that Adjacency List can be used to store multigraph.

With Adjacency Lists, we can also enumerate the list of neighbors of a vertex v efficiently. If v has k neighbors, the enumeration will require $O(k)$ time. Since this is one of the most common operations in most graph algorithms, it is advisable to use Adjacency Lists as your first choice of graph representation. Unless otherwise stated, most graph algorithms discussed in this book use the Adjacency List.

C). The Edge List, usually in the form of a vector of triples (see Figure 2.4).
Using the C++ STL: `vector< pair<int, ii> > EdgeList`.
Using the Java API: `Vector< IntegerTriple > EdgeList`.
`IntegerTriple` is a class that contains a triple of integers like `pair<int, ii>` above.

In the Edge List, we store a list of all E edges, usually in some sorted order. For directed graphs, we can store a bidirectional edge twice, one for each direction. The space complexity is clearly $O(E)$. This graph representation is very useful for Kruskal's algorithm for MST (Section 4.3.2), where the collection of undirected edges need to be sorted[23] by ascending weight. However, storing graph information in Edge List complicates many graph algorithms that require the enumeration of edges incident to a vertex.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/graphds.html`

Source code: `ch2_07_graph_ds.cpp/java`

Implicit Graph

Some graphs do *not* have to be stored in a graph data structure or explicitly generated for the graph to be traversed or operated upon. Such graphs are called *implicit* graphs. You will encounter them in the subsequent chapters. Implicit graphs can come in two flavours:

1. The edges can be determined easily.

   Example 1: Navigating a 2D grid map (see Figure 2.5.A). The vertices are the cells in the 2D character grid where '.' represents land and '#' represents an obstacle. The edges can be determined easily: There is an edge between two neighboring cells in the

[22] For simplicity, we will always assume that the second attribute exists in all graph implementations in this book although it is not always used.

[23] `pair` objects in C++ can be easily sorted. The default sorting criteria is to sort on the first item and then the second item for tie-breaking. In Java, we can write our own `IntegerPair`/`IntegerTriple` class that implements `Comparable`.

grid if they share an N/S/E/W border and if both are '.' (see Figure 2.5.B).

Example 2: The graph of chess knight movements on an 8×8 chessboard. The vertices are the cells in the chessboard. Two squares in the chessboard have an edge between them if they differ by two squares horizontally and one square vertically (or two squares vertically and one square horizontally). The first three rows and four columns of a chessboard are shown in Figure 2.5.C (many other vertices and edges are not shown).

2. The edges can be determined with some rules.

   Example: A graph contains N vertices (`[1..N]`). There is an edge between two vertices i and j if $(i+j)$ is a prime. See Figure 2.5.D that shows such a graph with $N = 5$ and several more examples in Section 8.2.3.

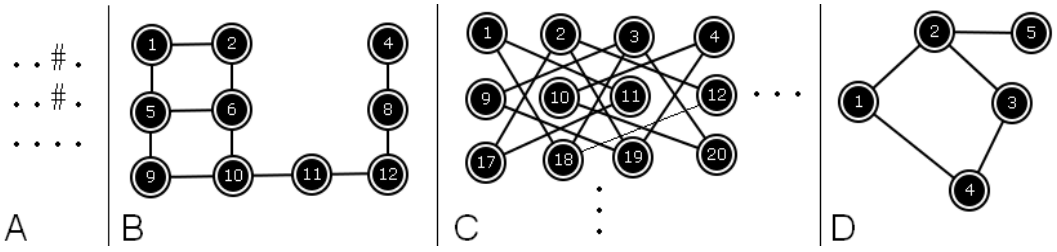

Figure 2.5: Implicit Graph Examples

Exercise 2.4.1.1*: Create the Adjacency Matrix, Adjacency List, and Edge List representations of the graphs shown in Figure 4.1 (Section 4.2.1) and in Figure 4.9 (Section 4.2.9). Hint: Use the graph data structure visualization tool shown above.

Exercise 2.4.1.2*: Given a (simple) graph in one representation (Adjacency Matrix/AM, Adjacency List/AL, or Edge List/EL), *convert* it into another graph representation in the most efficient way possible! There are 6 possible conversions here: AM to AL, AM to EL, AL to AM, AL to EL, EL to AM, and EL to AL.

Exercise 2.4.1.3: If the Adjacency Matrix of a (simple) graph has the property that it is equal to its transpose, what does this imply?

Exercise 2.4.1.4*: Given a (simple) graph represented by an Adjacency Matrix, perform the following tasks in the most efficient manner. Once you have figured out how to do this for Adjacency Matrices, perform the same task with Adjacency Lists and then Edge Lists.

1. Count the number of vertices V and directed edges E (assume that a bidirectional edge is equivalent to two directed edges) of the graph.
2*. Count the in-degree and the out-degree of a certain vertex v.
3*. Transpose the graph (reverse the direction of each edges).
4*. Check if the graph is a complete graph K_n. Note: A complete graph is a simple undirected graph in which *every pair* of distinct vertices is connected by a single edge.
5*. Check if the graph is a tree (a connected undirected graph with $E = V - 1$ edges).
6*. Check if the graph is a star graph S_k. Note: A star graph S_k is a complete bipartite $K_{1,k}$ graph. It is a tree with only one internal vertex and k leaves.

Exercise 2.4.1.5*: Research other possible methods of representing graphs other than the ones discussed above, especially for storing special graphs!

2.4.2 Union-Find Disjoint Sets

The Union-Find Disjoint Set (UFDS) is a data structure to model a collection of *disjoint sets* with the ability to efficiently[24]—in $\approx O(1)$—determine which set an item belongs to (or to test whether two items belong to the same set) and to unite two disjoint sets into one larger set. Such data structure can be used to solve the problem of finding connected components in an undirected graph (Section 4.2.3). Initialize each vertex to a separate disjoint set, then enumerate the graph's edges and join every two vertices/disjoint sets connected by an edge. We can then test if two vertices belong to the same component/set easily.

These seemingly simple operations are not *efficiently* supported by the C++ STL `set` (and Java `TreeSet`), which is not designed for this purpose. Having a `vector` of `set`s and looping through each one to find which set an item belongs to is expensive! C++ STL `set_union` (in `algorithm`) will not be efficient enough although it combines two sets in *linear time* as we still have to deal with shuffling the contents of the `vector` of `set`s! To support these set operations efficiently, we need a better data structure—the UFDS.

The main innovation of this data structure is in choosing a representative 'parent' item to represent a set. If we can ensure that each set is represented by only one unique item, then determining if items belong to the same set becomes far simpler: The representative 'parent' item can be used as a sort of identifier for the set. To achieve this, the Union-Find Disjoint Set creates a tree structure where the disjoint sets form a forest of trees. Each tree corresponds to a disjoint set. The root of the tree is determined to be the representative item for a set. Thus, the representative set identifier for an item can be obtained simply by following the chain of parents to the root of the tree, and since a tree can only have one root, this representative item can be used as a unique identifier for the set.

To do this efficiently, we store the index of the parent item and (the upper bound of) the height of the tree of each set (`vi p` and `vi rank` in our implementation). Remember that `vi` is our shortcut for a vector of integers. `p[i]` stores the immediate parent of item `i`. If item `i` is the representative item of a certain disjoint set, then `p[i] = i`, i.e. a self-loop. `rank[i]` yields (the upper bound of) the height of the tree rooted at item `i`.

In this section, we will use 5 disjoint sets {0, 1, 2, 3, 4} to illustrate the usage of this data structure. We initialize the data structure such that each item is a disjoint set by itself with rank 0 and the parent of each item is initially set to itself.

To unite two disjoint sets, we set the representative item (root) of one disjoint set to be the new parent of the representative item of the other disjoint set. This effectively merges the two trees in the Union-Find Disjoint Set representation. As such, `unionSet(i, j)` will cause both items '`i`' and '`j`' to have the same representative item—directly or indirectly. For efficiency, we can use the information contained in `vi rank` to set the representative item of the disjoint set with *higher rank* to be the new parent of the disjoint set with *lower rank*, thereby `minimizing` the rank of the resulting tree. If both ranks are the same, we arbitrarily choose one of them as the new parent and increase the resultant root's rank. This is the 'union by rank' heuristic. In Figure 2.6, top, `unionSet(0, 1)` sets `p[0]` to 1 and `rank[1]` to 1. In Figure 2.6, middle, `unionSet(2, 3)` sets `p[2]` to 3 and `rank[3]` to 1.

For now, let's assume that function `findSet(i)` simply calls `findSet(p[i])` recursively to find the representative item of a set, returning `findSet(p[i])` whenever `p[i] != i` and `i` otherwise. In Figure 2.6, bottom, when we call `unionSet(4, 3)`, we have `rank[findSet(4)] = rank[4] = 0` which is smaller than `rank[findSet(3)] = rank[3] = 1`, so we set `p[4] = 3` *without* changing the height of the resulting tree—this is the 'union by rank' heuristic

[24] M operations of this UFDS data structure with 'path compression' and 'union by rank' heuristics run in $O(M \times \alpha(n))$. However, since the inverse Ackermann function $\alpha(n)$ grows very slowly, i.e. its value is just less than 5 for practical input size $n \leq 1M$ in programming contest setting, we can treat $\alpha(n)$ as constant.

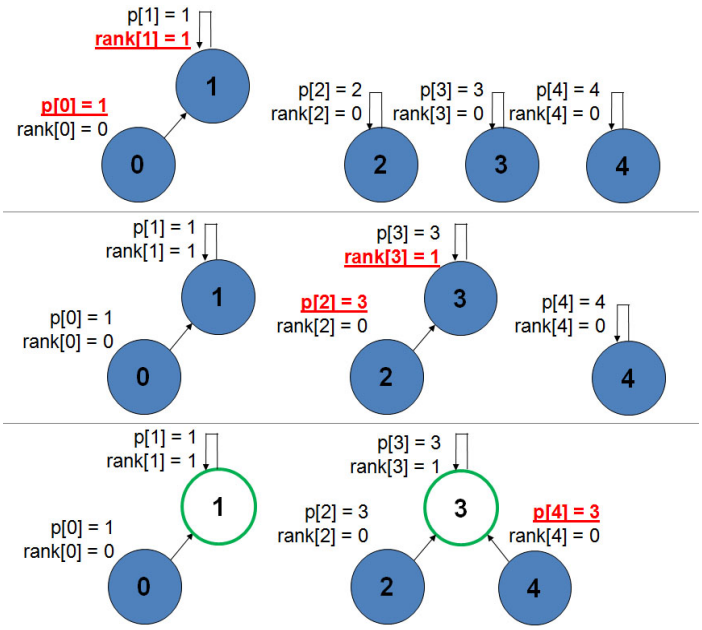

Figure 2.6: `unionSet(0, 1)` → `(2, 3)` → `(4, 3)` and `isSameSet(0, 4)`

at work. With the heuristic, the path taken from any node to the representative item by following the chain of 'parent' links is effectively minimized.

In Figure 2.6, bottom, `isSameSet(0, 4)` demonstrates another operation for this data structure. This function `isSameSet(i, j)` simply calls `findSet(i)` and `findSet(j)` and checks if both refer to the same representative item. If they do, then '`i`' and '`j`' both belong to the same set. Here, we see that `findSet(0) = findSet(p[0]) = findSet(1) = 1` is not the same as `findSet(4)= findSet(p[4]) = findSet(3) = 3`. Therefore item 0 and item 4 belongs to *different* disjoint sets.

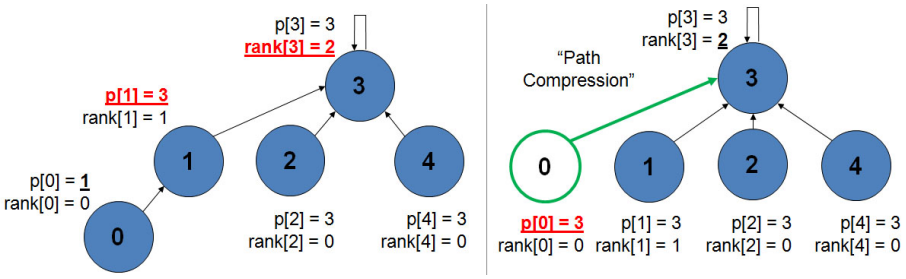

Figure 2.7: `unionSet(0, 3)` → `findSet(0)`

There is a technique that can vastly speed up the `findSet(i)` function: Path compression. Whenever we find the representative (root) item of a disjoint set by following the chain of 'parent' links from a given item, we can set the parent of *all items* traversed to point directly to the root. Any subsequent calls to `findSet(i)` on the affected items will then result in only one link being traversed. This changes the structure of the tree (to make `findSet(i)` more efficient) but yet preserves the actual constitution of the disjoint set. Since this will occur any time `findSet(i)` is called, the combined effect is to render the runtime of the `findSet(i)` operation to run in an extremely efficient amortized $O(M \times \alpha(n))$ time.

In Figure 2.7, we demonstrate this 'path compression'. First, we call `unionSet(0, 3)`. This time, we set `p[1] = 3` and update `rank[3] = 2`. Now notice that `p[0]` is unchanged, i.e. `p[0] = 1`. This is an *indirect* reference to the (true) representative item of the set, i.e. `p[0] = 1` → `p[1] = 3`. Function `findSet(i)` will actually require more than one step to

traverse the chain of 'parent' links to the root. However, once it finds the representative item, (e.g. 'x') for that set, it will *compress the path* by setting `p[i] = x`, i.e. `findSet(0)` sets `p[0] = 3`. Therefore, subsequent calls of `findSet(i)` will be just $O(1)$. This simple strategy is aptly named the 'path compression' heuristic. Note that `rank[3] = 2` now no longer reflects the *true height* of the tree. This is why `rank` only reflects the *upper bound* of the actual height of the tree. Our C++ implementation is shown below:

```
class UnionFind {                                              // OOP style
private: vi p, rank;                            // remember: vi is vector<int>
public:
  UnionFind(int N) { rank.assign(N, 0);
    p.assign(N, 0); for (int i = 0; i < N; i++) p[i] = i; }
  int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
  bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
  void unionSet(int i, int j) {
    if (!isSameSet(i, j)) {                          // if from different set
      int x = findSet(i), y = findSet(j);
      if (rank[x] > rank[y]) p[y] = x;          // rank keeps the tree short
      else {                 p[x] = y;
                             if (rank[x] == rank[y]) rank[y]++; }
} } };
```

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/ufds.html`

Source code: `ch2_08_unionfind_ds.cpp/java`

Exercise 2.4.2.1: There are two more queries that are commonly performed in this data structure. Update the code provided in this section to support these two queries efficiently: `int numDisjointSets()` that returns the number of disjoint sets currently in the structure and `int sizeOfSet(int i)` that returns the size of set that currently contains item `i`.

Exercise 2.4.2.2*: Given 8 disjoint sets: {0, 1, 2, ..., 7}, please create a sequence of `unionSet(i, j)` operations to create a tree with rank = 3! Is this possible for rank = 4?

Profiles of Data Structure Inventors

George Boole (1815-1864) was an English mathematician, philosopher, and logician. He is best known to Computer Scientists as the founder of Boolean logic, the foundation of modern digital computers. Boole is regarded as the founder of the field of Computer Science.

Rudolf Bayer (born 1939) has been Professor (emeritus) of Informatics at the Technical University of Munich. He invented the Red-Black (RB) tree used in the C++ STL `map`/`set`.

Georgii Adelson-Velskii (born 1922) is a Soviet mathematician and computer scientist. Along with Evgenii Mikhailovich Landis, he invented the AVL tree in 1962.

Evgenii Mikhailovich Landis (1921-1997) was a Soviet mathematician. The name of the AVL tree is an abbreviation of the two inventors: Adelson-Velskii and Landis himself.

2.4.3 Segment Tree

In this subsection, we will discuss a data structure which can efficiently answer *dynamic*[25] range queries. One such range query is the problem of finding the index of the minimum element in an array within range `[i..j]`. This is more commonly known as the Range Minimum Query (RMQ) problem. For example, given an array `A` of size $n = 7$ below, `RMQ(1, 3) = 2`, as the index 2 contains the minimum element among `A[1]`, `A[2]`, and `A[3]`. To check your understanding of RMQ, verify that in the array A below, `RMQ(3, 4) = 4`, `RMQ(0, 0) = 0`, `RMQ(0, 1) = 1`, and `RMQ(0, 6) = 5`. For the next few paragraphs, assume that array A is the same.

Array	Values	18	17	13	19	15	11	20
A	Indices	0	1	2	3	4	5	6

There are several ways to implement the RMQ. One trivial algorithm is to simply iterate the array from index `i` to `j` and report the index with the minimum value, but this will run in $O(n)$ time per query. When n is large and there are many queries, such an algorithm may be infeasible.

In this section, we answer the dynamic RMQ problem with a Segment Tree, which is another way to arrange data in a binary tree. There are several ways to implement the Segment Tree. Our implementation uses the same concept as the 1-based compact array in the binary heap where we use `vi` (our shortcut for `vector<int>`) `st` to represent the binary tree. Index 1 (skipping index 0) is the root and the left and right children of index p are index $2 \times p$ and $(2 \times p) + 1$ respectively (also see Binary Heap discussion in Section 2.3). The value of `st[p]` is the RMQ value of the segment associated with index p.

The root of segment tree represents segment `[0, n-1]`. For each segment `[L, R]` stored in index p where `L != R`, the segment will be split into `[L, (L+R)/2]` and `[(L+R)/2+1, R]` in a left and right vertices. The left sub-segment and right sub-segment will be stored in index $2 \times p$ and $(2 \times p)+1$ respectively. When `L = R`, it is clear that `st[p] = L` (or `R`). Otherwise, we will recursively build the segment tree, comparing the minimum value of the left and the right sub-segments and updating the `st[p]` of the segment. This process is implemented in the `build` routine below. This `build` routine creates up to $O(1+2+4+8+\ldots+2^{\log_2 n}) = O(2n)$ (smaller) segments and therefore runs in $O(n)$. However, as we use simple 1-based compact array indexing, we need `st` to be at least of size $2 * 2^{\lfloor(\log_2(n)\rfloor+1}$. In our implementation, we simply use a loose upper bound of space complexity $O(4n) = O(n)$. For array `A` above, the corresponding segment tree is shown in Figure 2.8 and 2.9.

With the segment tree ready, answering an RMQ can be done in $O(\log n)$. The answer for `RMQ(i, i)` is trivial—simply return `i` itself. However, for the general case `RMQ(i, j)`, further checks are needed. Let `p1 = RMQ(i, (i+j)/2)` and `p2 = RMQ((i+j)/2 + 1, j)`. Then `RMQ(i, j)` is `p1` if `A[p1]` $\leq$ `A[p2]` or `p2` otherwise. This process is implemented in the `rmq` routine below.

Take for example the query `RMQ(1, 3)`. The process in Figure 2.8 is as follows: Start from the root (index 1) which represents segment `[0, 6]`. We cannot use the stored minimum value of segment `[0, 6] = st[1] = 5` as the answer for `RMQ(1, 3)` since it is the minimum value over a larger[26] segment than the desired `[1, 3]`. From the root, we only have to go to the left subtree as the root of the right subtree represents segment `[4, 6]` which is outside[27] the desired range in `RMQ(1, 3)`.

[25] For dynamic problems, we need to frequently *update* and query the data. This makes pre-processing techniques useless.

[26] Segment `[L, R]` is said to be larger than query range `[i, j]` if `[L, R]` is not outside the query range and not inside query range (see the other footnotes).

[27] Segment `[L, R]` is said to be outside query range `[i, j]` if `i > R || j < L`.

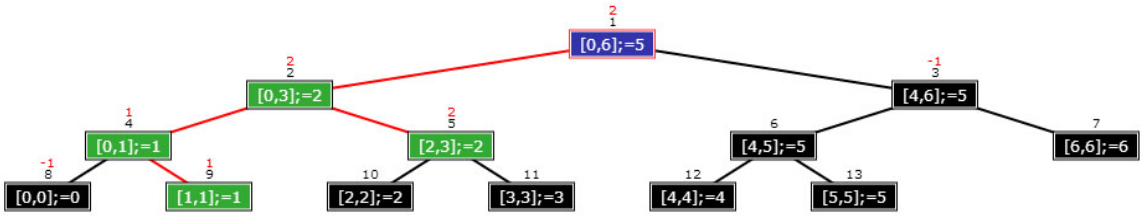

Figure 2.8: Segment Tree of Array A = {18, 17, 13, 19, 15, 11, 20} and `RMQ(1, 3)`

We are now at the root of the left subtree (index 2) that represents segment `[0, 3]`. This segment `[0, 3]` is still larger than the desired `RMQ(1, 3)`. In fact, `RMQ(1, 3)` intersects *both* the left sub-segment `[0, 1]` (index 4) and the right sub-segment `[2, 3]` (index 5) of segment `[0, 3]`, so we have to explore *both* subtrees (sub-segments).

The left segment `[0, 1]` (index 4) of `[0, 3]` (index 2) is not yet inside the `RMQ(1, 3)`, so another split is necessary. From segment `[0, 1]` (index 4), we move right to segment `[1, 1]` (index 9), which is now inside[28] `[1, 3]`. At this point, we know that `RMQ(1, 1) = st[9] = 1` and we can return this value to the caller. The right segment `[2, 3]` (index 5) of `[0, 3]` (index 2) is inside the required `[1, 3]`. From the stored value inside this vertex, we know that `RMQ(2, 3) = st[5] = 2`. We do *not* need to traverse further down.

Now, back in the call to segment `[0, 3]` (index 2), we now have `p1 = RMQ(1, 1) = 1` and `p2 = RMQ(2, 3) = 2`. Because `A[p1] > A[p2]` since `A[1] = 17` and `A[2] = 13`, we now have `RMQ(1, 3) = p2 = 2`. This is the final answer.

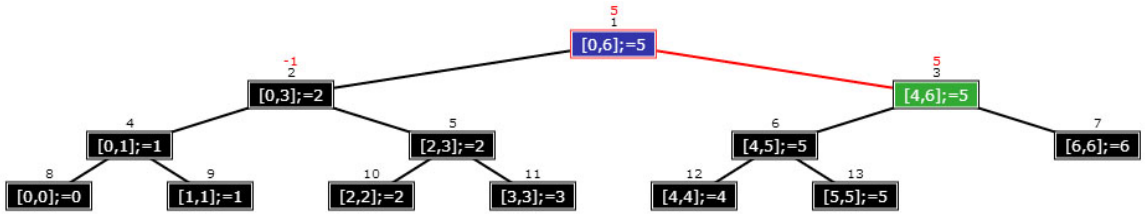

Figure 2.9: Segment Tree of Array A = {18, 17, 13, 19, 15, 11, 20} and `RMQ(4, 6)`

Now let's take a look at another example: `RMQ(4, 6)`. The execution in Figure 2.9 is as follows: We again start from the root segment `[0, 6]` (index 1). Since it is larger than the `RMQ(4, 6)`, we move right to segment `[4, 6]` (index 3) as segment `[0, 3]` (index 2) is outside. Since this segment exactly represents `RMQ(4, 6)`, we simply return the index of minimum element that is stored in this vertex, which is 5. Thus `RMQ(4, 6) = st[3] = 5`.

This data structure allows us to avoid traversing the unnecessary parts of the tree! In the worst case, we have *two* root-to-leaf paths which is just $O(2 \times \log(2n)) = O(\log n)$. Example: In `RMQ(3, 4) = 4`, we have one root-to-leaf path from `[0, 6]` to `[3, 3]` (index $1 \rightarrow 2 \rightarrow 5 \rightarrow 11$) and another root-to-leaf path from `[0, 6]` to `[4, 4]` (index $1 \rightarrow 3 \rightarrow 6 \rightarrow 12$).

If the array `A` is static (i.e. unchanged after it is instantiated), then using a Segment Tree to solve the RMQ problem is *overkill* as there exists a Dynamic Programming (DP) solution that requires $O(n \log n)$ one-time pre-processing and allows for $O(1)$ per RMQ. This DP solution will be discussed later in Section 9.33.

Segment Tree is useful if the underlying array is frequently updated (dynamic). For example, if `A[5]` is now changed from 11 to 99, then we just need to update the vertices along the leaf to root path in $O(\log n)$. See path: `[5, 5]` (index 13, `st[13]` is unchanged) $\rightarrow$ `[4, 5]` (index 6, `st[6] = 4` now) $\rightarrow$ `[4, 6]` (index 3, `st[3] = 4` now) $\rightarrow$ `[0, 6]` (index

[28] Segment `[L, R]` is said to be inside query range `[i, j]` if `L` $\geq$ `i && R` $\leq$ `j`.

1, st[1] = 2 now) in Figure 2.10. For comparison, the DP solution presented in Section 9.33 requires another $O(n \log n)$ pre-processing to update the structure and is ineffective for such dynamic updates.

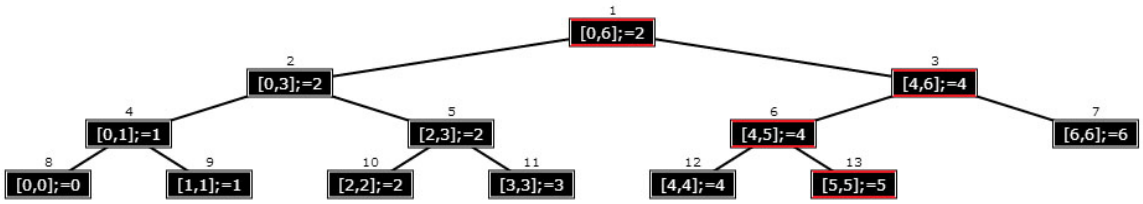

Figure 2.10: Updating Array A to {18, 17, 13, 19, 15, **99**, 20}

Our Segment Tree implementation is shown below. The code shown here supports only *static* RMQs (*dynamic* updates are left as an exercise to the reader).

```
class SegmentTree {         // the segment tree is stored like a heap array
private: vi st, A;            // recall that vi is: typedef vector<int> vi;
  int n;
  int left (int p) { return p << 1; }     // same as binary heap operations
  int right(int p) { return (p << 1) + 1; }

  void build(int p, int L, int R) {                                  // O(n)
    if (L == R)                              // as L == R, either one is fine
      st[p] = L;                                            // store the index
    else {                                  // recursively compute the values
      build(left(p) , L              , (L + R) / 2);
      build(right(p), (L + R) / 2 + 1, R          );
      int p1 = st[left(p)], p2 = st[right(p)];
      st[p] = (A[p1] <= A[p2]) ? p1 : p2;
  } }

  int rmq(int p, int L, int R, int i, int j) {                  // O(log n)
    if (i >  R || j <  L) return -1; // current segment outside query range
    if (L >= i && R <= j) return st[p];               // inside query range

     // compute the min position in the left and right part of the interval
    int p1 = rmq(left(p) , L              , (L+R) / 2, i, j);
    int p2 = rmq(right(p), (L+R) / 2 + 1, R          , i, j);

    if (p1 == -1) return p2;   // if we try to access segment outside query
    if (p2 == -1) return p1;                                 // same as above
    return (A[p1] <= A[p2]) ? p1 : p2;               // as in build routine
  }

public:
  SegmentTree(const vi &_A) {
    A = _A; n = (int)A.size();              // copy content for local usage
    st.assign(4 * n, 0);            // create large enough vector of zeroes
    build(1, 0, n - 1);                                  // recursive build
  }
```

```
  int rmq(int i, int j) { return rmq(1, 0, n - 1, i, j); }   // overloading
};

int main() {
  int arr[] = { 18, 17, 13, 19, 15, 11, 20 };         // the original array
  vi A(arr, arr + 7);
  SegmentTree st(A);
  printf("RMQ(1, 3) = %d\n", st.rmq(1, 3));             // answer = index 2
  printf("RMQ(4, 6) = %d\n", st.rmq(4, 6));             // answer = index 5
} // return 0;
```

Visualization: www.comp.nus.edu.sg/∼stevenha/visualization/segmenttree.html

Source code: ch2_09_segmenttree_ds.cpp/java

Exercise 2.4.3.1*: Draw the Segment Tree corresponding to array `A` = {`10, 2, 47, 3, 7, 9, 1, 98, 21`}. Answer RMQ(1, 7) and RMQ(3, 8)! Hint: Use the Segment Tree visualization tool shown above.

Exercise 2.4.3.2*: In this section, we have seen how Segment Trees can be used to answer Range Minimum Queries (RMQs). Segment Trees can also be used to answer dynamic Range Sum Queries (`RSQ(i, j)`), i.e. a sum from `A[i] + A[i + 1] + ...+ A[j]`. Modify the given Segment Tree code above to deal with RSQ.

Exercise 2.4.3.3: Using a similar Segment Tree to **Exercise 2.4.3.1** above, answer the queries `RSQ(1, 7)` and `RSQ(3, 8)`. Is this a good approach to solve the problem if array `A` is never changed? (also see Section 3.5.2).

Exercise 2.4.3.4*: The Segment Tree code shown above lacks the (point) `update` operation as discussed in the body text. Add the $O(\log n)$ `update` function to update the value of a certain index (point) in array `A` and simultaneously update the corresponding Segment Tree!

Exercise 2.4.3.5*: The (point) update operation shown in the body text only changes the value of a certain index in array `A`. What if we delete existing elements of array `A` or insert a new elements into array `A`? Can you explain what will happen with the given Segment Tree code and what you should do to address it?

Exercise 2.4.3.6*: There is also one more important Segment Tree operation that has not yet been discussed, the *range* update operation. Suppose a certain subarray of `A` is updated to a certain common value. Can we update the Segment Tree efficiently? Study and solve UVa 11402 - Ahoy Pirates—a problem that requires range updates.

2.4.4 Binary Indexed (Fenwick) Tree

Fenwick Tree—also known as **Binary Indexed Tree** (BIT)—were invented by *Peter M. Fenwick* in 1994 [18]. In this book, we will use the term Fenwick Tree as opposed to BIT in order to differentiate with the standard *bit manipulations*. The Fenwick Tree is a useful data structure for implementing *dynamic cumulative frequency tables*. Suppose we have[29] test scores of `m = 11` students `f = {2,4,5,5,6,6,6,7,7,8,9}` where the test scores are *integer values* ranging from `[1..10]`. Table 2.1 shows the frequency of each individual test score ∈ `[1..10]` and the cumulative frequency of test scores ranging from `[1..i]` denoted by `cf[i]`—that is, the sum of the frequencies of test scores `1, 2, ..., i`.

Index/ Score	Frequency `f`	Cumulative Frequency `cf`	Short Comment
0	-	-	Index 0 is ignored (as the sentinel value).
1	0	0	`cf[1] = f[1] = 0.`
2	1	1	`cf[2] = f[1] + f[2] = 0 + 1 = 1.`
3	0	1	`cf[3] = f[1] + f[2] + f[3] = 0 + 1 + 0 = 1.`
4	1	2	`cf[4] = cf[3] + f[4] = 1 + 1 = 2.`
5	2	4	`cf[5] = cf[4] + f[5] = 2 + 2 = 4.`
6	3	7	`cf[6] = cf[5] + f[6] = 4 + 3 = 7.`
7	2	9	`cf[7] = cf[6] + f[7] = 7 + 2 = 9.`
8	1	10	`cf[8] = cf[7] + f[8] = 9 + 1 = 10.`
9	1	11	`cf[9] = cf[8] + f[9] = 10 + 1 = 11.`
10	0	11	`cf[10] = cf[9] + f[10] = 11 + 0 = 11.`

Table 2.1: Example of a Cumulative Frequency Table

The cumulative frequency table can also be used as a solution to the Range Sum Query (RSQ) problem mentioned in **Exercise 2.4.3.2***. It stores `RSQ(1, i)` $\forall i \in$ `[1..n]` where `n` is the largest integer index/score[30]. In the example above, we have `n = 10`, `RSQ(1, 1) = 0`, `RSQ(1, 2) = 1`, ..., `RSQ(1, 6) = 7`, ..., `RSQ(1, 8) = 10`, ..., and `RSQ(1, 10) = 11`. We can then obtain the answer to the RSQ for an arbitrary range `RSQ(i, j)` when $i \neq 1$ by subtracting `RSQ(1, j) - RSQ(1, i - 1)`. For example, `RSQ(4, 6) = RSQ(1, 6) - RSQ(1, 3) = 7 - 1 = 6`.

If the frequencies are *static*, then the cumulative frequency table as in Table 2.1 can be computed efficiently with a simple $O(n)$ loop. First, set `cf[1] = f[1]`. Then, for `i` ∈ `[2..n]`, compute `cf[i] = cf[i - 1] + f[i]`. This will be discussed further in Section 3.5.2. However, when the frequencies are frequently updated (increased or decreased) and the RSQs are frequently asked afterwards, it is better to use a *dynamic* data structure.

Instead of using a Segment Tree to implement a *dynamic* cumulative frequency table, we can implement the *far simpler* Fenwick Tree instead (compare the source code for both implementations, provided in this section and in the previous Section 2.4.3). This is perhaps one of the reasons why the Fenwick Tree is currently included in the IOI syllabus [20]. Fenwick Tree operations are also extremely efficient as they use fast bit manipulation techniques (see Section 2.2).

In this section, we will use the function `LSOne(i)` (which is actually `(i & (-i))`) extensively, naming it to match its usage in the original paper [18]. In Section 2.2, we have seen that the operation `(i & (-i))` produces the first Least Significant One-bit in `i`.

[29]The test scores are shown in sorted order for simplicity, they do not have to be sorted.

[30]Please differentiate `m` = the number of data points and `n` = the largest integer value among the `m` data points. The meaning of `n` in Fenwick Tree is a bit different compared to other data structures in this book.

The Fenwick Tree is typically implemented as an array (we use a `vector` for size flexibility). The Fenwick Tree is a tree that is indexed by the *bits* of its *integer* keys. These integer keys fall within the fixed range `[1..n]`—skipping[31] index 0. In a programming contest environment, `n` can approach $\approx 1M$ so that the Fenwick Tree covers the range `[1..1M]`—large enough for many practical (contest) problems. In Table 2.1 above, the scores `[1..10]` are the integer keys in the corresponding array with size `n = 10` and `m = 11` data points.

Let the name of the Fenwick Tree array be `ft`. Then, the element at index `i` is responsible for elements in the range `[i-LSOne(i)+1..i]` and `ft[i]` stores the cumulative frequency of elements {`i-LSOne(i)+1, i-LSOne(i)+2, i-LSOne(i)+3, .., i`}. In Figure 2.11, the value of `ft[i]` is shown in the circle above index `i` and the range `[i-LSOne(i)+1..i]` is shown as a circle and a bar (if the range spans more than one index) above index `i`. We can see that `ft[4] = 2` is responsible for range `[4-4+1..4] = [1..4]`, `ft[6] = 5` is responsible for range `[6-2+1..6] = [5..6]`, `ft[7] = 2` is responsible for range `[7-1+1..7] = [7..7]`, `ft[8] = 10` is responsible for range `[8-8+1..8] = [1..8]` etc[32].

With such an arrangement, if we want to obtain the cumulative frequency between `[1..b]`, i.e. `rsq(b)`, we simply add `ft[b]`, `ft[b']`, `ft[b'']`, ... until index `b`^{i} is 0. This sequence of indices is obtained via subtracting the Least Significant One-bit via the bit manipulation expression: `b' = b - LSOne(b)`. Iteration of this bit manipulation effectively *strips off* the least significant one-bit of `b` at each step. As an integer `b` only has $O(\log b)$ bits, `rsq(b)` runs in $O(\log n)$ time when `b = n`. In Figure 2.11, `rsq(6) = ft[6] + ft[4] = 5 + 2 = 7`. Notice that indices 4 and 6 are responsible for range `[1..4]` and `[5..6]`, respectively. By combining them, we account for the entire range of `[1..6]`. The indices 6, 4, and 0 are related in their binary form: `b` = 6_{10} = $(1\underline{1}0)_2$ can be transformed to `b'` = 4_{10} = $(\underline{1}00)_2$ and subsequently to `b''` = 0_{10} = $(000)_2$.

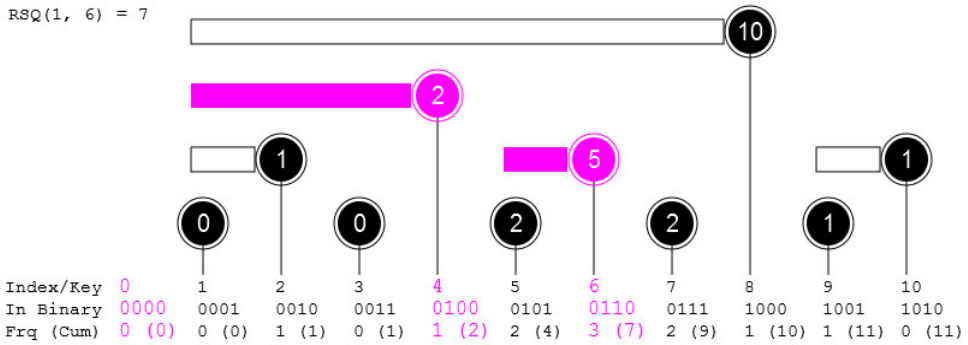

Figure 2.11: Example of `rsq(6)`

With `rsq(b)` available, obtaining the cumulative frequency between two indices `[a..b]` where `a != 1` is simple, just evaluate `rsq(a, b) = rsq(b) - rsq(a - 1)`. For example, if we want to compute `rsq(4, 6)`, we can simply return `rsq(6) - rsq(3) = (5+2) - (0+1) = 7 - 1 = 6`. Again, this operation runs in $O(2 \times \log b) \approx O(\log n)$ time when `b = n`. Figure 2.12 displays the value of `rsq(3)`.

When updating the value of the element at index `k` by adjusting its value by `v` (note that `v` can be either positive or negative), i.e. calling `adjust(k, v)`, we have to update `ft[k]`, `ft[k']`, `ft[k'']`, ... until index `k`^{i} exceeds n. This sequence of indices are obtained

[31] We have chosen to follow the original implementation by [18] that ignores index 0 to facilitate an easier understanding of the bit manipulation operations of Fenwick Tree. Note that index 0 has no bit turned on. Thus, the operation `i +/- LSOne(i)` simply returns `i` when `i = 0`. Index 0 is also used as the terminating condition in the `rsq` function.

[32] In this book, we will not detail why this arrangement works and will instead show that it allows for efficient $O(\log n)$ update and RSQ operations. Interested readers are advised to read [18].

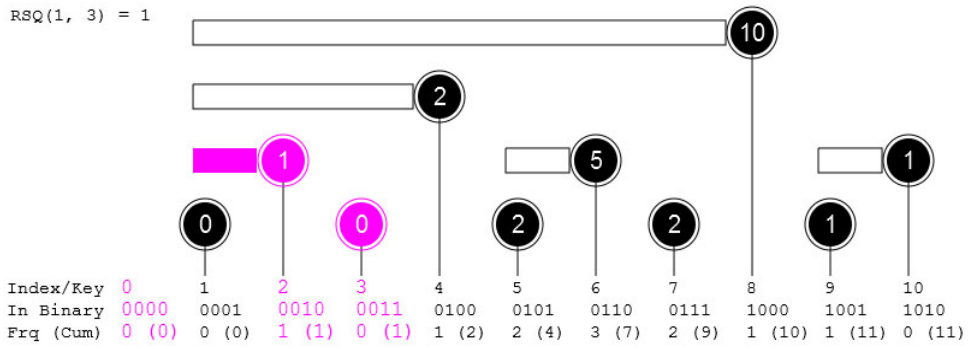

Figure 2.12: Example of `rsq(3)`

via this similar iterative bit manipulation expression: `k' = k + LSOne(k)`. Starting from any integer `k`, the operation `adjust(k, v)` will take at most $O(\log n)$ steps until `k` $> n$. In Figure 2.13, `adjust(5, 1)` will affect (add +1 to) `ft[k]` at indices `k` $= 5_{10} = (101)_2$, `k'` $= (101)_2$ `+` $(001)_2 = (110)_2 = 6_{10}$, and `k''` $= (110)_2 + (010)_2 = (1000)_2 = 8_{10}$ via the expression given above. Notice that if you project a line upwards from index 5 in Figure 2.13, you will see that the line indeed *intersects* the ranges under the responsibility of index 5, index 6, and index 8.

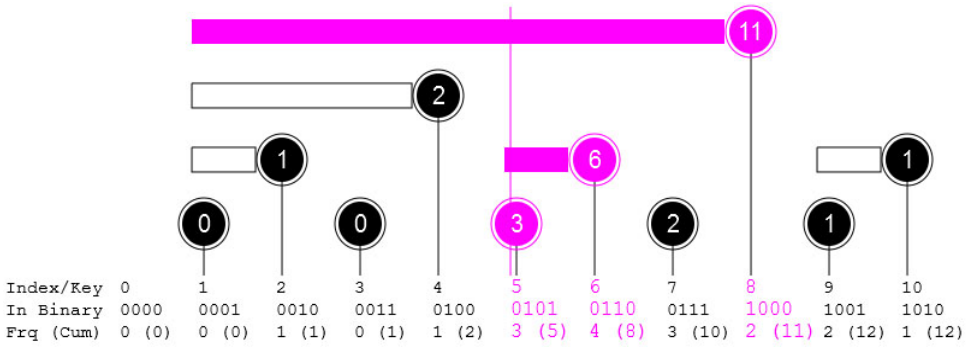

Figure 2.13: Example of `adjust(5, 1)`

In summary, Fenwick Tree supports both RSQ and update operations in just $O(n)$ space and $O(\log n)$ time given a set of m integer keys that ranges from `[1..`n`]`. This makes Fenwick Tree an ideal data structure for solving *dynamic* RSQ problems on with discrete arrays (the *static* RSQ problem can be solved with simple $O(n)$ pre-processing and $O(1)$ per query as shown earlier). Our *short* C++ implementation of a basic Fenwick Tree is shown below.

```
class FenwickTree {
private: vi ft;                 // recall that vi is: typedef vector<int> vi;
public: FenwickTree(int n) { ft.assign(n + 1, 0); }    // init n + 1 zeroes
  int rsq(int b) {                                     // returns RSQ(1, b)
    int sum = 0; for (; b; b -= LSOne(b)) sum += ft[b];
    return sum; }                                  // note: LSOne(S) (S & (-S))
  int rsq(int a, int b) {                              // returns RSQ(a, b)
    return rsq(b) - (a == 1 ? 0 : rsq(a - 1)); }
  // adjusts value of the k-th element by v (v can be +ve/inc or -ve/dec)
  void adjust(int k, int v) {                     // note: n = ft.size() - 1
    for (; k < (int)ft.size(); k += LSOne(k)) ft[k] += v; }
};
```

```
int main() {
  int f[] = { 2,4,5,5,6,6,6,7,7,8,9 };                    // m = 11 scores
  FenwickTree ft(10);          // declare a Fenwick Tree for range [1..10]
      // insert these scores manually one by one into an empty Fenwick Tree
  for (int i = 0; i < 11; i++) ft.adjust(f[i], 1);    // this is O(k log n)
  printf("%d\n", ft.rsq(1, 1));  // 0 => ft[1] = 0
  printf("%d\n", ft.rsq(1, 2));  // 1 => ft[2] = 1
  printf("%d\n", ft.rsq(1, 6));  // 7 => ft[6] + ft[4] = 5 + 2 = 7
  printf("%d\n", ft.rsq(1, 10)); // 11 => ft[10] + ft[8] = 1 + 10 = 11
  printf("%d\n", ft.rsq(3, 6));  // 6 => rsq(1, 6) - rsq(1, 2) = 7 - 1
  ft.adjust(5, 2);               // update demo
  printf("%d\n", ft.rsq(1, 10)); // now 13
} // return 0;
```

Visualization: www.comp.nus.edu.sg/∼stevenha/visualization/bit.html

Source code: ch2_10_fenwicktree_ds.cpp/java

Exercise 2.4.4.1: Just a simple exercise of the two basic bit-manipulation operations used in the Fenwick Tree: What are the values of `90 - LSOne(90)` and `90 + LSOne(90)`?

Exercise 2.4.4.2: What if the problem that you want to solve includes an element at integer key 0? Recall that the standard integer key range in our library code is is `[1..n]` and that this implementation cannot use index 0 since it is used as the terminating condition of `rsq`.

Exercise 2.4.4.3: What if the problem that you want to solve uses non-integer keys? For example, what if the test scores shown in Table 2.1 above are `f = {5.5, 7.5, 8.0, 10.0}` (i.e. allowing either a 0 or a 5 after the decimal place)? What if the test scores are `f = {5.53, 7.57, 8.10, 9.91}` (i.e. allowing for two digits after the decimal point)?

Exercise 2.4.4.4: The Fenwick Tree supports an additional operation that we have decided to leave as an exercise to the reader: Find the smallest index with a given cumulative frequency. For example, we may need to determine the minimum index/score `i` in Table 2.1 such that there are at least 7 students covered in the range `[1..i]` (index/score 6 in this case). Implement this feature.

Exercise 2.4.4.5*: Solve this dynamic RSQ problem: UVa 12086 - Potentiometers using *both* a Segment Tree and Fenwick Tree. Which solution is easier to produce in this case? Also see Table 2.2 for a comparison between these two data structures.

Exercise 2.4.4.6*: Extend the 1D Fenwick Tree to 2D!

Exercise 2.4.4.7*: Fenwick Trees are normally used for point update and range (sum) query. Show how to use a Fenwick Tree for *range update* and point queries. For example, given lots of intervals with small ranges (from 1 to at most 1 million) determine the number of intervals encompassing index i.

Profile of Data Structure Inventors

Peter M. Fenwick is a Honorary Associate Professor in the University of Auckland. He invented the Binary Indexed Tree in 1994 [18] as "cumulative frequency tables of arithmetic compression". The BIT has since been included in the IOI syllabus [20] and used in many contest problems for its efficient yet easy to implement data structure.

Feature	Segment Tree	Fenwick Tree
Build Tree from Array	$O(n)$	$O(m \log n)$
Dynamic RMin/MaxQ	OK	Very limited
Dynamic RSQ	OK	OK
Query Complexity	$O(\log n)$	$O(\log n)$
Point Update Complexity	$O(\log n)$	$O(\log n)$
Length of Code	Longer	Shorter

Table 2.2: Comparison Between Segment Tree and Fenwick Tree

Programming exercises that use the data structures discussed and implemented:

- Graph Data Structures Problems
  1. **UVa 00599 - The Forrest for the Trees *** ($v-e$ = number of connected components, keep a `bitset` of size 26 to count the number of vertices that have some edge. Note: Also solvable with Union-Find)
  2. **UVa 10895 - Matrix Transpose *** (transpose adjacency list)
  3. UVa 10928 - My Dear Neighbours (counting out degrees)
  4. UVa 11550 - Demanding Dilemma (graph representation, incidence matrix)
  5. **UVa 11991 - Easy Problem from ... *** (use the idea of an Adj List)

  Also see: More graph problems in Chapter 4
- Union-Find Disjoint Sets
  1. **UVa 00793 - Network Connections *** (trivial; application of disjoint sets)
  2. UVa 01197 - The Suspects (LA 2817, Kaohsiung03, Connected Components)
  3. UVa 10158 - War (advanced usage of disjoint sets with a nice twist; memorize list of enemies)
  4. UVa 10227 - Forests (merge two disjoint sets if they are consistent)
  5. **UVa 10507 - Waking up brain *** (disjoint sets simplifies this problem)
  6. UVa 10583 - Ubiquitous Religions (count disjoint sets after all unions)
  7. UVa 10608 - Friends (find the set with the largest element)
  8. UVa 10685 - Nature (find the set with the largest element)
  9. **UVa 11503 - Virtual Friends *** (maintain set attribute (size) in rep item)
  10. UVa 11690 - Money Matters (check if total money from each member is 0)
- Tree-related Data Structures
  1. UVa 00297 - Quadtrees (simple quadtree problem)
  2. UVa 01232 - SKYLINE (LA 4108, Singapore07, a simple problem if input size is small; but since $n \leq 100000$, we have to use a Segment Tree; note that this problem is not about RSQ/RMQ)
  3. **UVa 11235 - Frequent Values *** (range maximum query)
  4. UVa 11297 - Census (Quad Tree with updates or use 2D segment tree)
  5. UVa 11350 - Stern-Brocot Tree (simple tree data structure question)
  6. **UVa 11402 - Ahoy, Pirates *** (segment tree with *lazy* updates)
  7. UVa 12086 - Potentiometers (LA 2191, Dhaka06; pure dynamic range sum query problem; solvable with Fenwick Tree or Segment Tree)
  8. ***UVa 12532 - Interval Product ** (clever usage of Fenwick/Segment Tree)

  Also see: DS as part of the solution of harder problems in Chapter 8

2.5 Solution to Non-Starred Exercises

Exercise 2.2.1*: Sub-question 1: First, sort S in $O(n \log n)$ and then do an $O(n)$ linear scan starting from the second element to check if an integer and the previous integer are the same (also read the solution for **Exercise 1.2.10**, task 4). Sub-question 6: Read the opening paragraph of Chapter 3 and the detailed discussion in Section 9.29. Solutions for the other sub-questions are not shown.

Exercise 2.2.2: The answers (except sub-question 7):

1. S & $(N-1)$
2. $(S$ & $(S-1)) == 0$
3. S & $(S-1)$
4. $S \parallel (S+1)$
5. S & $(S+1)$
6. $S \parallel (S-1)$

Exercise 2.3.1: Since the collection is dynamic, we will encounter frequent insertion and deletion queries. An insertion can potentially change the sort order. If we store the information in a static array, we will have to use one $O(n)$ iteration of an insertion sort after each insertion and deletion (to close the gap in the array). This is inefficient!

Exercise 2.3.2:

1. `search(71)`: root (15) → 23 → 71 (found)
   `search(7)`: root (15) → 6 → 7 (found)
   `search(22)`: root (15) → 23 → empty left subtree (not found).
2. We will eventually have the same BST as in Figure 2.2.
3. To find the min/max element, we can start from root and keep going left/right until we encounter a vertex with no left/right subtrees respectively. That vertex is the answer.
4. We will obtain the sorted output: 4, 5, 6, 7, 15, 23, 50, 71. See Section 4.7.2 if you are not familiar with the inorder tree traversal algorithm.
5. `successor(23)`: Find the minimum element of the subtree rooted at the right of 23, which is the subtree rooted at 71. The answer is 50.
   `successor(7)`: 7 has no right subtree, so 7 must be the maximum of a certain subtree. That subtree is the subtree rooted at 6. The parent of 6 is 15 and 6 is the left subtree of 15. By the BST property, 15 must be the successor of 7.
   `successor(71)`: 71 is the largest element and has no successor.
   Note: The algorithm to find the predecessor of a node is similar.
6. `delete(5)`: We simply remove 5, which is a leaf, from the BST
   `delete(71)`: As 71 is an internal vertex with one child, we cannot simply delete 71 as doing so will disconnect the BST into *two* components. We can instead reshuffle the subtree rooted at the parent of 71 (which is 23), causing 23 to has 50 as its right child.

7. `delete(15)`: As 15 is a vertex with two children, we cannot simply delete 15 as doing so will disconnect the BST into *three* components. To deal with this issue, we need to find the successor of 15 (which is 23) and use the successor to replace 15. We then delete the old 23 from the BST (not a problem now). As a note, we can also use predecessor(key) instead of successor(key) during delete(key) for the case when the key has two children.

Exercise 2.3.3*: For Sub-task 1, we run inorder traversal in $O(n)$ and see if the values are sorted. Solutions to other sub-tasks are not shown.

Exercise 2.3.6: The answers:

1. `Insert(26)`: Insert 26 as the left subtree of 3, swap 26 with 3, then swap 26 with 19 and stop. The Max Heap array A now contains {-, 90, 26, 36, 17, 19, 25, 1, 2, 7, 3}.
2. `ExtractMax()`: Swap 90 (maximum element which will be reported after we fix the Max Heap property) with 3 (the current bottom-most right-most leaf/the last item in the Max Heap), swap 3 with 36, swap 3 with 25 and stop. The Max Heap array A now contains {-, 36, 26, 25, 17, 19, 3, 1, 2, 7}.

Exercise 2.3.7: Yes, check that all indices (vertices) satisfy the Max Heap property.

Exercise 2.3.16: Use the C++ STL `set` (or Java `TreeSet`) as it is a balanced BST that supports $O(\log n)$ dynamic insertions and deletions. We can use the inorder traversal to print the data in the BST in sorted order (simply use C++ `iterator`s or Java `Iterator`s).

Exercise 2.3.17: Use the C++ STL `map` (Java `TreeMap`) and a counter variable. A hash table is also a possible solution but not necessary for programming contests. This trick is quite frequently used in various (contest) problems. Example usage:

```
char str[1000];
map<string, int> mapper;
int i, idx;
for (i = idx = 0; i < M; i++) {                          // idx starts from 0
  scanf("%s", &str);
  if (mapper.find(str) == mapper.end())   // if this is the first encounter
  // alternatively, we can also test if mapper.count(str) is greater than 0
    mapper[str] = idx++;       // give str the current idx and increase idx
}
```

Exercise 2.4.1.3: The graph is undirected.

Exercise 2.4.1.4*: Subtask 1: To count the number of vertices of a graph: Adjacency Matrix/Adjacency List → report the number of rows; Edge List → count the number of distinct vertices in all edges. To count the number of edges of a graph: Adjacency Matrix → sum the number of non-zero entries in every row; Adjacency List → sum the length of all the lists; Edge List → simply report the number of rows. Solutions to other sub-tasks are not shown.

Exercise 2.4.2.1: For `int numDisjointSets()`, use an additional integer counter `numSets`. Initially, during `UnionFind(N)`, set `numSets = N`. Then, during `unionSet(i, j)`, decrease `numSets` by one if `isSameSet(i, j)` returns false. Now, `int numDisjointSets()` can simply return the value of `numSets`.

For `int sizeOfSet(int i)`, we use another `vi setSize(N)` initialized to all ones (each set has only one element). During `unionSet(i, j)`, update the `setSize` array by performing `setSize[find(j)] += setSize[find(i)]` (or the other way around depending on rank) if `isSameSet(i, j)` returns false. Now `int sizeOfSet(int i)` can simply return the value of `setSize[find(i)];`

These two variants have been implemented in `ch2_08_unionfind_ds.cpp/java`.

Exercise 2.4.3.3: `RSQ(1, 7) = 167` and `RSQ(3, 8) = 139`; No, using a Segment Tree is overkill. There is a simple DP solution that uses an $O(n)$ pre-processing step and takes $O(1)$ time per RSQ (see Section 9.33).

Exercise 2.4.4.1: `90 - LSOne(90) =` $(1011010)_2$ `-` $(10)_2$ `=` $(1011000)_2$ `= 88` and `90 + LSOne(90) =` $(1011010)_2$ `+` $(10)_2$ `=` $(1011100)_2$ `= 92`.

Exercise 2.4.4.2: Simple: shift all indices by one. Index $i$in the 1-based Fenwick Tree now refers to index $i - 1$ in the actual problem.

Exercise 2.4.4.3: Simple: convert the floating point numbers into integers. For the first task, we can multiply every number by two. For the second case, we can multiply all numbers by one hundred.

Exercise 2.4.4.4: The cumulative frequency is sorted, thus we can use a *binary search*. Study the 'binary search for the answer' technique discussed in Section 3.3. The resulting time complexity is $O(\log^2 n)$.

2.6 Chapter Notes

The basic data structures mentioned in Section 2.2-2.3 can be found in almost every data structure and algorithm textbook. References to the C++/Java built-in libraries are available online at: `www.cppreference.com` and `java.sun.com/javase/7/docs/api`. Note that although access to these reference websites are usually provided in programming contests, we suggest that you try to master the syntax of the most common library operations to minimize coding time during actual contests!

One exception is perhaps the *lightweight set of Boolean* (a.k.a bitmask). This *unusual* technique is not commonly taught in data structure and algorithm classes, but it is quite important for competitive programmers as it allows for significant speedups if applied to certain problems. This data structure appears in various places throughout this book, e.g. in some iterative brute force and optimized backtracking routines (Section 3.2.2 and Section 8.2.1), DP TSP (Section 3.5.2), DP with bitmask (Section 8.3.1). All of them use bitmasks instead of `vector<boolean>` or `bitset<size>` due to its efficiency. Interested readers are encouraged to read the book "Hacker's Delight" [69] that discusses bit manipulation in further detail.

Extra references for the data structures mentioned in Section 2.4 are as follows. For Graphs, see [58] and Chapters 22-26 of [7]. For Union-Find Disjoint Sets, see Chapter 21 of [7]. For Segment Trees and other geometric data structures, see [9]. For the Fenwick Tree, see [30]. We remark that all our implementation of data structures discussed in Section 2.4 avoid the usage of pointers. We use either arrays or vectors.

With more experience and by reading the source code we have provided, you can master more tricks in the application of these data structures. Please spend time exploring the source code provided with this book at `sites.google.com/site/stevenhalim/home/material`.

There are few more data structures discussed in this book—string-specific data structures (**Suffix Trie/Tree/Array**) are discussed in Section 6.6. Yet, there are still many other data structures that we cannot cover in this book. If you want to do better in programming contests, please research data structure techniques beyond what we have presented in this book. For example, **AVL Trees**, **Red Black Trees**, or even **Splay Trees** are useful for certain problems that require you to implement and augment (add more data to) balanced BSTs (see Section 9.29). **Interval Tree**s (which are similar to Segment Trees) and **Quad Tree**s (for partitioning 2D space) are useful to know as their underlying concepts may help you to solve certain contest problems.

Notice that many of the efficient data structures discussed in this book exhibit the 'Divide and Conquer' strategy (discussed in Section 3.3).

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	12	18 (+50%)	35 (+94%)
Written Exercises	5	12 (+140%)	14+27*=41 (+242%)
Programming Exercises	43	124 (+188%)	132 (+6%)

The breakdown of the number of programming exercises from each section is shown below:

Section	Title	Appearance	% in Chapter	% in Book
2.2	**Linear DS**	79	60%	5%
2.3	**Non-Linear DS**	30	23%	2%
2.4	Our-own Libraries	23	17%	1%


ACM-ICPC Asia Phuket Regional
Contest 2009
3-4 November 2009 Phuket
University Phuket Campus
L-R: Adhiraj, Steven
ACM-ICPC Asia Phuket Regional
Programming Contest 2009
Prince of Songkla University Phuket Campus
30,000.-
acm
L-R: Victor, Hung, Dr CJ Hwang, Phong, Duc, Tien


THAILAND
2011
L-R: Hubert, Chuanqi, Zi Chun, Steven, Zhan Xiong, Raymond, Mr Cheong

Chapter 3

Problem Solving Paradigms

If all you have is a hammer, everything looks like a nail
— **Abraham Maslow, 1962**

3.1 Overview and Motivation

In this chapter, we discuss *four* problem solving paradigms commonly used to attack problems in programming contests, namely Complete Search (a.k.a Brute Force), Divide and Conquer, the Greedy approach, and Dynamic Programming. All competitive programmers, including IOI and ICPC contestants, need to master these problem solving paradigms (and more) in order to be able to attack a given problem with the appropriate 'tool'. Hammering *every* problem with Brute Force solutions will not enable anyone to perform well in contests. To illustrate, we discuss four simple tasks below involving an array `A` containing $n \leq 10K$ small integers $\leq 100K$ *(e.g.* $A = \{10, 7, 3, 5, 8, 2, 9\}$, $n = 7$*)* to give an overview of what happens if we attempt every problem with Brute Force as our sole paradigm.

1. Find the largest and the smallest element of `A`. *(10 and 2 for the given example).*
2. Find the k^{th} smallest element in `A`. *(if $k = 2$, the answer is 3 for the given example).*
3. Find the largest gap g such that $x, y \in$ `A` and $g = |x - y|$. *(8 for the given example).*
4. Find the longest increasing subsequence of `A`. *($\{3, 5, 8, 9\}$ for the given example).*

The answer for the first task is simple: Try each element of `A` and check if it is the current largest (or smallest) element seen so far. This is an $O(n)$ **Complete Search** solution.

The second task is a little harder. We can use the solution above to find the smallest value and replace it with a large value (e.g. $1M$) to 'delete' it. We can then proceed to find the smallest value again (the second smallest value in the original array) and replace it with $1M$. Repeating this process k times, we will find the k^{th} smallest value. This works, but if $k = \frac{n}{2}$ (the median), this Complete Search solution runs in $O(\frac{n}{2} \times n) = O(n^2)$. Instead, we can sort the array `A` in $O(n \log n)$, returning the answer simply as `A[k-1]`. However, a better solution for a small number of queries is the expected $O(n)$ solution shown in Section 9.29. The $O(n \log n)$ and $O(n)$ solutions above are **Divide and Conquer** solutions.

For the third task, we can similarly consider all possible two integers x and y in `A`, checking if the gap between them is the largest for each pair. This Complete Search approach runs in $O(n^2)$. It works, but is slow and inefficient. We can prove that g can be obtained by finding the difference between the smallest and largest elements of `A`. These two integers can be found with the solution of the first task in $O(n)$. No other combination of two integers in `A` can produce a larger gap. This is a **Greedy** solution.

For the fourth task, trying all $O(2^n)$ possible subsequences to find the longest increasing one is not feasible for all $n \leq 10K$. In Section 3.5.2, we discuss a simple $O(n^2)$ **Dynamic Programming** solution and also the faster $O(n \log k)$ Greedy solution for this task.

Here is some advice for this chapter: Please do not just memorize the solutions for each problem discussed, but instead remember and internalize the thought process and problem solving strategies used. Good problem solving skills are more important than memorized solutions for well-known Computer Science problems when dealing with (often creative and novel) contest problems.

3.2 Complete Search

The Complete Search technique, also known as brute force or recursive backtracking, is a method for solving a problem by traversing the entire (or part of the) search space to obtain the required solution. During the search, we are allowed to prune (that is, choose not to explore) parts of the search space if we have determined that these parts have no possibility of containing the required solution.

In programming contests, a contestant *should* develop a Complete Search solution when there is clearly no other algorithm available (e.g. the task of enumerating *all* permutations of $\{0, 1, 2, \ldots, N-1\}$ clearly requires $O(N!)$ operations) or when better algorithms exist, but are *overkill* as the input size happens to be small (e.g. the problem of answering Range Minimum Queries as in Section 2.4.3 but on static arrays with $N \leq 100$ is solvable with an $O(N)$ loop for each query).

In ICPC, Complete Search should be the first solution considered as it is usually easy to come up with such a solution and to code/debug it. Remember the 'KISS' principle: Keep It Short and Simple. A *bug-free* Complete Search solution should *never* receive the Wrong Answer (WA) response in programming contests as it explores the *entire* search space. However, many programming problems do have better-than-Complete-Search solutions as illustrated in the Section 3.1. Thus a Complete Search solution may receive a Time Limit Exceeded (TLE) verdict. With proper analysis, you can determine the likely outcome (TLE versus AC) before attempting to code anything (Table 1.4 in Section 1.2.3 is a good starting point). If a Complete Search is likely to pass the time limit, then go ahead and implement one. This will then give you more time to work on harder problems in which Complete Search will be too slow.

In IOI, you will usually need better problem solving techniques as Complete Search solutions are usually only rewarded with very small fractions of the total score in the subtask scoring schemes. Nevertheless, Complete Search should be used when you cannot come up with a better solution—it will at least enable you to score some marks.

Sometimes, running Complete Search on *small instances* of a challenging problem can help us to understand its structure through patterns in the output (it is possible to *visualize* the pattern for some problems) that can be exploited to design a faster algorithm. Some combinatorics problems in Section 5.4 can be solved this way. Then, the Complete Search solution can also act as a verifier for *small instances*, providing an additional check for the faster but non-trivial algorithm that you develop.

After reading this section, you may have the impression that Complete Search only works for 'easy problems' and it is usually not the intended solution for 'harder problems'. This is not entirely true. There exist hard problems that are only solvable with creative Complete Search algorithms. We have reserved those problems for Section 8.2.

In the next two sections, we give several *(easier)* examples of this simple yet possibly challenging paradigm. In Section 3.2.1, we give examples that are implemented *iteratively*. In Section 3.2.2, we give examples on solutions that are implemented *recursively* (with backtracking). Finally, in Section 3.2.3, we provide a few tips to give your solution, especially your Complete Search solution, a better chance to pass the required Time Limit.

3.2.1 Iterative Complete Search

Iterative Complete Search (Two Nested Loops: UVa 725 - Division)

Abridged problem statement: Find and display all pairs of 5-digit numbers that collectively use the digits 0 through 9 once each, such that the first number divided by the second is equal to an integer N, where $2 \leq N \leq 79$. That is, `abcde / fghij = N`, where each letter represents a different digit. The first digit of one of the numbers is allowed to be zero, e.g. for `N = 62`, we have `79546 / 01283 = 62; 94736 / 01528 = 62`.

Quick analysis shows that `fghij` can only range from `01234` to `98765` which is at most $\approx 100K$ possibilities. An even better bound for `fghij` is the range `01234` to `98765 / N`, which has at most $\approx 50K$ possibilities for $N = 2$ and becomes smaller with increasing N. For each attempted `fghij`, we can get `abcde` from `fghij * N` and then check if all 10 digits are different. This is a doubly-nested loop with a time complexity of at most $\approx 50K \times 10 = 500K$ operations per test case. This is small. Thus, an iterative Complete Search is feasible. The main part of the code is shown below (we use a fancy bit manipulation trick shown in Section 2.2 to determine digit uniqueness):

```
for (int fghij = 1234; fghij <= 98765 / N; fghij++) {
  int abcde = fghij * N;  // this way, abcde and fghij are at most 5 digits
  int tmp, used = (fghij < 10000); // if digit f=0, then we have to flag it
  tmp = abcde; while (tmp) { used |= 1 << (tmp % 10); tmp /= 10; }
  tmp = fghij; while (tmp) { used |= 1 << (tmp % 10); tmp /= 10; }
  if (used == (1<<10) - 1)              // if all digits are used, print it
    printf("%0.5d / %0.5d = %d\n", abcde, fghij, N);
}
```

Iterative Complete Search (Many Nested Loops: UVa 441 - Lotto)

In programming contests, problems that are solvable with a *single* loop are usually considered *easy*. Problems which require doubly-nested iterations like UVa 725 - Division above are more challenging but they are not necessarily considered difficult. Competitive programmers must be comfortable writing code with *more than two* nested loops.

Let's take a look at UVa 441 which can be summarized as follows: Given $6 < k < 13$ integers, enumerate all possible subsets of size 6 of these integers in sorted order.

Since the size of the required subset is always 6 and the output has to be sorted lexicographically (the input is already sorted), the easiest solution is to use *six* nested loops as shown below. Note that even in the largest test case when $k = 12$, these six nested loops will only produce ${}_{12}C_6 = 924$ lines of output. This is small.

```
for (int i = 0; i < k; i++)                       // input: k sorted integers
  scanf("%d", &S[i]);
for (int a = 0    ; a < k - 5; a++)                     // six nested loops!
  for (int b = a + 1; b < k - 4; b++)
    for (int c = b + 1; c < k - 3; c++)
      for (int d = c + 1; d < k - 2; d++)
        for (int e = d + 1; e < k - 1; e++)
          for (int f = e + 1; f < k    ; f++)
            printf("%d %d %d %d %d %d\n",S[a],S[b],S[c],S[d],S[e],S[f]);
```

Iterative Complete Search (Loops + Pruning: UVa 11565 - Simple Equations)

Abridged problem statement: Given three integers A, B, and C ($1 \leq A, B, C \leq 10000$), find three other distinct integers x, y, and z such that $x + y + z = A$, $x \times y \times z = B$, and $x^2 + y^2 + z^2 = C$.

The third equation $x^2 + y^2 + z^2 = C$ is a good starting point. Assuming that C has the largest value of 10000 and y and z are one and two (x, y, z have to be distinct), then the possible range of values for x is $[-100 \ldots 100]$. We can use the same reasoning to get a similar range for y and z. We can then write the following triply-nested iterative solution below that requires $201 \times 201 \times 201 \approx 8M$ operations per test case.

```
bool sol = false; int x, y, z;
for (x = -100; x <= 100; x++)
  for (y = -100; y <= 100; y++)
    for (z = -100; z <= 100; z++)
      if (y != x && z != x && z != y &&      // all three must be different
          x + y + z == A && x * y * z == B && x * x + y * y + z * z == C) {
        if (!sol) printf("%d %d %d\n", x, y, z);
        sol = true; }
```

Notice the way a short circuit AND was used to speed up the solution by enforcing a *lightweight* check on whether `x`, `y`, and `z` are all different *before* we check the three formulas. The code shown above already passes the required time limit for this problem, but we can do better. We can also use the second equation $x \times y \times z = B$ and assume that $x = y = z$ to obtain $x \times x \times x < B$ or $x < \sqrt[3]{B}$. The new range of x is $[-22 \ldots 22]$. We can also prune the search space by using `if` statements to execute only some of the (inner) loops, or use `break` and/or `continue` statements to stop/skip loops. The code shown below is now much faster than the code shown above (there are a few other optimizations required to solve the extreme version of this problem: UVa 11571 - Simple Equations - Extreme!!):

```
bool sol = false; int x, y, z;
for (x = -22; x <= 22 && !sol; x++) if (x * x <= C)
  for (y = -100; y <= 100 && !sol; y++) if (y != x && x * x + y * y <= C)
    for (z = -100; z <= 100 && !sol; z++)
      if (z != x && z != y &&
          x + y + z == A && x * y * z == B && x * x + y * y + z * z == C) {
        printf("%d %d %d\n", x, y, z);
        sol = true; }
```

Iterative Complete Search (Permutations: UVa 11742 - Social Constraints)

Abridged problem statement: There are $0 < n \leq 8$ movie goers. They will sit in the front row in n consecutive open seats. There are $0 \leq m \leq 20$ seating constraints among them, i.e. movie goer **a** and movie goer **b** must be at most (or at least) **c** seats apart. The question is simple: How many possible seating arrangements are there?

The key part to solve this problem is in realizing that we have to explore **all** permutations (seating arrangements). Once we realize this fact, we can derive this simple $O(m \times n!)$ 'filtering' solution. We set `counter = 0` and then try all possible $n!$ permutations. We increase the `counter` by 1 if the current permutation satisfies all m constraints. When all $n!$ permutations have been examined, we output the final value of `counter`. As the maximum

n is 8 and maximum m is 20, the largest test case will still only require $20 \times 8! = 806400$ operations—a perfectly viable solution.

If you have never written an algorithm to generate all permutations of a set of numbers (see **Exercise 1.2.3**, task 7), you may still be unsure about how to proceed. The simple C++ solution is shown below.

```
#include <algorithm>              // next_permutation is inside this C++ STL
// the main routine
int i, n = 8, p[8] = {0, 1, 2, 3, 4, 5, 6, 7};     // the first permutation
do {   // try all possible O(n!) permutations, the largest input 8! = 40320
  ...              // check the given social constraint based on 'p' in O(m)
}                           // the overall time complexity is thus O(m * n!)
while (next_permutation(p, p + n));   // this is inside C++ STL <algorithm>
```

Iterative Complete Search (Subsets: UVa 12455 - Bars)

Abridged problem statement[1]: Given a list `l` containing $1 \leq n \leq 20$ integers, is there a subset of list `l` that sums to another given integer X?

We can try all 2^n possible subsets of integers, sum the selected integers for each subset in $O(n)$, and see if the sum of the selected integers equals to X. The overall time complexity is thus $O(n \times 2^n)$. For the largest test case when $n = 20$, this is just $20 \times 2^{20} \approx 21M$. This is 'large' but still viable (for reason described below).

If you have never written an algorithm to generate all subsets of a set of numbers (see **Exercise 1.2.3**, task 8), you may still be unsure how to proceed. An easy solution is to use the *binary representation* of integers from 0 to $2^n - 1$ to describe all possible subsets. If you are not familiar with bit manipulation techniques, see Section 2.2. The solution can be written in simple C/C++ shown below (also works in Java). Since bit manipulation operations are (very) fast, the required $21M$ operations for the largest test case are still doable in under a second. Note: A faster implementation is possible (see Section 8.2.1).

```
// the main routine, variable 'i' (the bitmask) has been declared earlier
for (i = 0; i < (1 << n); i++) {                  // for each subset, O(2^n)
  sum = 0;
  for (int j = 0; j < n; j++)                       // check membership, O(n)
    if (i & (1 << j))        // test if bit 'j' is turned on in subset 'i'?
      sum += l[j];                                    // if yes, process 'j'
  if (sum == X) break;                  // the answer is found: bitmask 'i'
}
```

Exercise 3.2.1.1: For the solution of UVa 725, why is it better to iterate through `fghij` and not through `abcde`?

Exercise 3.2.1.2: Does a 10! algorithm that permutes `abcdefghij` work for UVa 725?

Exercise 3.2.1.3*: Java does *not* have a built-in `next_permutation` function yet. If you are a Java user, write your own recursive backtracking routine to generate all permutations! This is similar to the recursive backtracking for the 8-Queens problem.

Exercise 3.2.1.4*: How would you solve UVa 12455 if $1 \leq n \leq \underline{30}$ and each integer can be as big as 1000000000? Hint: See Section 8.2.4.

[1]This is also known as the 'Subset Sum' problem, see Section 3.5.3.

3.2.2 Recursive Complete Search

Simple Backtracking: UVa 750 - 8 Queens Chess Problem

Abridged problem statement: In chess (with an 8×8 board), it is possible to place eight queens on the board such that no two queens attack each other. Determine *all* such possible arrangements given the position of one of the queens (i.e. coordinate (a, b) must contain a queen). Output the possibilities in lexicographical (sorted) order.

The most naïve solution is to enumerate all combinations of 8 different cells out of the $8 \times 8 = 64$ possible cells in a chess board and see if the 8 queens can be placed at these positions without conflicts. However, there are ${}_{64}C_8 \approx 4B$ such possibilities—this idea is not even worth trying.

A better but still naïve solution is to realize that each queen can only occupy one column, so we can put exactly one queen in each column. There are only $8^8 \approx 17M$ possibilities now, down from $4B$. This is still a 'borderline'-passing solution for this problem. If we write a Complete Search like this, we are likely to receive the Time Limit Exceeded (TLE) verdict especially if there are multiple test cases. We can still apply the few more easy optimizations described below to further reduce the search space.

We know that no two queens can share the same column *or the same row*. Using this, we can further simplify the original problem to the problem of finding valid *permutations* of 8! row positions. The value of `row[i]` describes the row position of the queen in column `i`. Example: `row = {1, 3, 5, 7, 2, 0, 6, 4}` as in Figure 3.1 is one of the solutions for this problem; `row[0] = 1` implies that the queen in column 0 is placed in row 1, and so on (the index starts from 0 in this example). Modeled this way, the search space goes *down* from $8^8 \approx 17M$ to $8! \approx 40K$. This solution is already fast enough, but we can still do more.

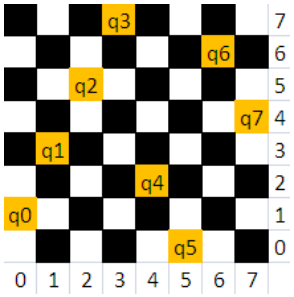

Figure 3.1: 8-Queens

We also know that no two queens can share any of the two diagonal lines. Let queen A be at `(i, j)` and queen B be at `(k, l)`. They attack each other iff `abs(i-k) == abs(j-l)`. This formula means that the vertical and horizontal distances between these two queens are equal, i.e. queen A and B lie on one of each other's two diagonal lines.

A *recursive backtracking* solution places the queens one by one in columns 0 to 7, observing all the constraints above. Finally, if a candidate solution is found, check if at least one of the queens satisfies the input constraints, i.e. `row[b] == a`. This *sub* (i.e. lower than) $O(n!)$ solution will obtain an AC verdict.

We provide our implementation below. If you have never written a recursive backtracking solution before, please scrutinize it and perhaps re-code it in your own coding style.

```
#include <cstdlib>                          // we use the int version of 'abs'
#include <cstdio>
#include <cstring>
using namespace std;

int row[8], TC, a, b, lineCounter;            // ok to use global variables

bool place(int r, int c) {
  for (int prev = 0; prev < c; prev++)     // check previously placed queens
    if (row[prev] == r || (abs(row[prev] - r) == abs(prev - c)))
      return false;         // share same row or same diagonal -> infeasible
  return true; }
```

```
void backtrack(int c) {
  if (c == 8 && row[b] == a) {         // candidate sol, (a, b) has 1 queen
    printf("%2d      %d", ++lineCounter, row[0] + 1);
    for (int j = 1; j < 8; j++) printf(" %d", row[j] + 1);
    printf("\n"); }
  for (int r = 0; r < 8; r++)                       // try all possible row
    if (place(r, c)) {          // if can place a queen at this col and row
      row[c] = r; backtrack(c + 1);      // put this queen here and recurse
}   }

int main() {
  scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &a, &b); a--; b--;         // switch to 0-based indexing
    memset(row, 0, sizeof row); lineCounter = 0;
    printf("SOLN       COLUMN\n");
    printf(" #      1 2 3 4 5 6 7 8\n\n");
    backtrack(0);           // generate all possible 8! candidate solutions
    if (TC) printf("\n");
} } // return 0;
```

Source code: ch3_01_UVa750.cpp/java

More Challenging Backtracking: UVa 11195 - Another n-Queen Problem

Abridged problem statement: Given an $n \times n$ chessboard ($3 < n < \mathbf{15}$) where some of the cells are bad (queens cannot be placed on those bad cells), how many ways can you place n queens in the chessboard so that no two queens attack each other? Note: Bad cells *cannot* be used to block queens' attack.

The recursive backtracking code that we have presented above is *not* fast enough for $n = 14$ and no bad cells, the worst possible test case for this problem. The *sub-*$O(n!)$ solution presented earlier is still OK for $n = 8$ but not for $n = 14$. We have to do better.

The major issue with the previous n-queens code is that it is quite slow when checking whether the position of a new queen is valid since we compare the new queen's position with the previous `c-1` queens' positions (see function `bool place(int r, int c)`). It is better to store the same information with three boolean arrays (we use `bitset`s for now):

```
bitset<30> rw, ld, rd;      // for the largest n = 14, we have 27 diagonals
```

Initially all n rows (`rw`), $2 \times n - 1$ left diagonals (`ld`), and $2 \times n - 1$ right diagonals (`rd`) are unused (these three `bitset`s are all set to `false`). When a queen is placed at cell `(r, c)`, we flag `rw[r] = true` to disallow this row from being used again. Furthermore, all `(a, b)` where `abs(r - a) = abs(c - b)` also cannot be used anymore. There are two possibilities after removing the `abs` function: `r - c = a - b` and `r + c = a + b`. Note that `r + c` and `r - c` represent indices for the two diagonal axes. As `r - c` can be negative, we add an *offset* of `n - 1` to both sides of the equation so that `r - c + n - 1 = a - b + n - 1`. If a queen is placed on cell `(r, c)`, we flag `ld[r - c + n - 1] = true` and `rd[r + c] = true` to disallow these two diagonals from being used again. With these additional data structures and the additional problem-specific constraint in UVa 11195 (`board[r][c]` cannot be a bad cell), we can extend our code to become:

```
void backtrack(int c) {
  if (c == n) { ans++; return; }                                  // a solution
  for (int r = 0; r < n; r++)                              // try all possible row
    if (board[r][c] != '*' && !rw[r] && !ld[r - c + n - 1] && !rd[r + c]) {
      rw[r] = ld[r - c + n - 1] = rd[r + c] = true;               // flag off
      backtrack(c + 1);
      rw[r] = ld[r - c + n - 1] = rd[r + c] = false;               // restore
}   }
```

Visualization: www.comp.nus.edu.sg/~stevenha/visualization/recursion.html

Exercise 3.2.2.1: The code shown for UVa 750 can be further optimized by pruning the search when '`row[b] != a`' *earlier* during the recursion (not only when c == 8). Modify it!

Exercise 3.2.2.2*: Unfortunately, the updated solution presented using `bitset`s: `rw`, `ld`, and `rd` will still obtain a TLE for UVa 11195 - Another n-Queen Problem. We need to further speed up the solution using bitmask techniques and another way of using the left and right diagonal constraints. This solution will be discussed in Section 8.2.1. For now, use the (non Accepted) idea presented here for UVa 11195 to speed up the code for UVa 750 and two more similar problems: UVa 167 and 11085!

3.2.3 Tips

The biggest gamble in writing a Complete Search solution is whether it will or will not be able to pass the time limit. If the time limit is 10 seconds (online judges do not usually use large time limits for efficient judging) and your program currently runs in ≈ 10 seconds on several (can be more than one) test cases with the largest input size as specified in the problem description, yet your code is still judged to be TLE, you may want to tweak the 'critical code'[2] in your program instead of re-solving the problem with a faster algorithm which may not be easy to design.

Here are some tips that you may want to consider when designing your Complete Search solution for a certain problem to give it a higher chance of passing the Time Limit. Writing a good Complete Search solution is an art in itself.

Tip 1: Filtering versus Generating

Programs that examine lots of (if not all) candidate solutions and choose the ones that are correct (or remove the incorrect ones) are called 'filters', e.g. the naïve 8-queens solver with ${}_{64}C_8$ and 8^8 time complexity, the iterative solution for UVa 725 and UVa 11742, etc. Usually 'filter' programs are written iteratively.

Programs that gradually build the solutions and immediately prune invalid partial solutions are called 'generators', e.g. the improved recursive 8-queens solver with its *sub-*$O(n!)$ complexity plus diagonal checks. Usually, 'generator' programs are easier to implement when written recursively as it gives us greater flexibility for pruning the search space.

Generally, filters are easier to code but run slower, given that it is usually far more difficult to prune more of the search space iteratively. Do the math (complexity analysis) to see if a filter is good enough or if you need to create a generator.

[2]It is said that every program spends most of its time in only about 10% of its code—the critical code.

Tip 2: Prune Infeasible/Inferior Search Space Early

When generating solutions using recursive backtracking (see the tip no 1 above), we may encounter a partial solution that will never lead to a full solution. We can prune the search there and explore other parts of the search space. Example: The diagonal check in the 8-queens solution above. Suppose we have placed a queen at `row[0] = 2`. Placing the next queen at `row[1] = 1` or `row[1] = 3` will cause a diagonal conflict and placing the next queen at `row[1] = 2` will cause a row conflict. Continuing from any of these infeasible partial solutions will never lead to a valid solution. Thus we can prune these partial solutions at this juncture and concentrate only on the other valid positions: `row[1] = {0, 4, 5, 6, 7}`, thus reducing the overall runtime. As a rule of thumb, the earlier you can prune the search space, the better.

In other problems, we may be able to compute the 'potential worth' of a partial (and still valid) solution. If the potential worth is inferior to the worth of the current best found valid solution so far, we can prune the search there.

Tip 3: Utilize Symmetries

Some problems have symmetries and we should try to exploit symmetries to reduce execution time! In the 8-queens problem, there are 92 solutions but there are only 12 unique (or fundamental/canonical) solutions as there are rotational and line symmetries in the problem. You can utilize this fact by only generating the 12 unique solutions and, if needed, generate the whole 92 by rotating and reflecting these 12 unique solutions. Example: `row = {7-1, 7-3, 7-5, 7-7, 7-2, 7-0, 7-6, 7-4} = {6, 4, 2, 0, 5, 7, 1, 3}` is the horizontal reflection of the configuration in Figure 3.1.

However, we have to remark that it is true that sometimes considering symmetries can actually complicate the code. In competitive programming, this is usually not the best way (we want shorter code to minimize bugs). If the gain obtained by dealing with symmetry is not significant in solving the problem, just ignore this tip.

Tip 4: Pre-Computation a.k.a. Pre-Calculation

Sometimes it is helpful to generate tables or other data structures that accelerate the lookup of a result prior to the execution of the program itself. This is called Pre-Computation, in which one trades memory/space for time. However, this technique can rarely be used for recent programming contest problems.

For example, since we know that there are only 92 solutions in the standard 8-queens chess problem, we can create a 2D array `int solution[92][8]` and then fill it with all 92 valid permutations of the 8-queens row positions! That is, we can create a generator program (which takes some time to run) to fill this 2D array `solution`. Afterwards, we can write *another* program to simply and quickly print the correct permutations within the 92 pre-calculated configurations that satisfy the problem constraints.

Tip 5: Try Solving the Problem Backwards

Some contest problems look far easier when they are solved 'backwards' [53] (from a *less obvious* angle) than when they are solved using a frontal attack (from the more obvious angle). Be prepared to attempt unconventional approaches to problems.

This tip is best illustrated using an example: UVa 10360 - Rat Attack: Imagine a 2D array (up to 1024×1024) containing rats. There are $n \leq 20000$ rats spread across the cells. Determine which cell `(x, y)` should be gas-bombed so that the number of rats killed in

a square box `(x-d, y-d)` to `(x+d, y+d)` is maximized. The value d is the power of the gas-bomb ($d \leq 50$), see Figure 3.2.

An immediate solution is to attack this problem in the most obvious fashion possible: bomb each of the 1024^2 cells and select the most effective location. For each bombed cell `(x, y)`, we can perform an $O(d^2)$ scan to count the number of rats killed within the square-bombing radius. For the worst case, when the array has size 1024^2 and $d = 50$, this takes $1024^2 \times 50^2 = 2621M$ operations. TLE[3]!

Another option is to attack this problem backwards: Create an array `int killed[1024][1024]`. For each rat population at coordinate `(x, y)`, add it to `killed[i][j]`, where $|i-x| \leq d$ and $|j-y| \leq d$. This is because if a bomb was placed at `(i, j)`, the rats at coordinate `(x, y)` will be killed. This pre-processing takes $O(n \times d^2)$ operations. Then, to determine the most optimal bombing position, we can simply find the coordinate of the highest entry in array `killed`, which can be done in 1024^2 operations. This approach only requires $20000 \times 50^2 + 1024^2 = 51M$ operations for the worst test case ($n = 20000, d = 50$), ≈ 51 times faster than the frontal attack! This is an AC solution.

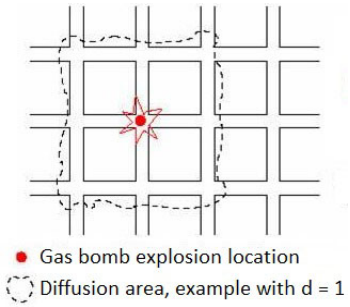

Figure 3.2: UVa 10360 [47]

Tip 6: Optimizing Your Source Code

There are many tricks that you can use to optimize your code. Understanding computer hardware and how it is organized, especially the I/O, memory, and cache behavior, can help you design better code. Some examples (not exhaustive) are shown below:

1. A biased opinion: Use C++ instead of Java. An algorithm implemented using C++ usually runs faster than the one implemented in Java in many online judges, including UVa [47]. Some programming contests give Java users extra time to account for the difference in performance.

2. For C/C++ users, use the faster C-style `scanf/printf` rather than `cin/cout`. For Java users, use the faster `BufferedReader/BufferedWriter` classes as follows:

```
BufferedReader br = new BufferedReader(                    // speedup
  new InputStreamReader(System.in));
// Note: String splitting and/or input parsing is needed afterwards

PrintWriter pr = new PrintWriter(new BufferedWriter(       // speedup
  new OutputStreamWriter(System.out)));
// PrintWriter allows us to use the pr.printf() function
// do not forget to call pr.close() before exiting your Java program
```

3. Use the *expected* $O(n \log n)$ but cache-friendly quicksort in C++ STL `algorithm::sort` (part of 'introsort') rather than the true $O(n \log n)$ but non cache-friendly heapsort (its root-to-leaf/leaf-to-root operations span a wide range of indices—lots of cache misses).

4. Access a 2D array in a row major fashion (row by row) rather than in a column major fashion—multidimensional arrays are stored in a row-major order in memory.

[3]Although 2013 CPU can compute $\approx 100M$ operations in a few seconds, $2621M$ operations will still take too long in a contest environment.

5. Bit manipulation on the built-in integer data types (up to the 64-bit integer) is more efficient than index manipulation in an array of booleans (see bitmask in Section 2.2). If we need more than 64 bits, use the C++ STL `bitset` rather than `vector<bool>` (e.g. for Sieve of Eratosthenes in Section 5.5.1).

6. Use lower level data structures/types at all times if you do not need the extra functionality in the higher level (or larger) ones. For example, use an `array` with a slightly larger size than the maximum size of input instead of using resizable `vector`s. Also, use 32-bit `int`s instead of 64-bit `long long`s as the 32-bit `int` is faster in most 32-bit online judge systems.

7. For Java, use the faster `ArrayList` (and `StringBuilder`) rather than `Vector` (and `StringBuffer`). Java `Vector`s and `StringBuffer`s are *thread safe* but this feature is not needed in competitive programming. Note: In this book, we will stick with `Vector`s to avoid confusing bilingual C++ and Java readers who use both the C++ STL `vector` and Java `Vector`.

8. Declare most data structures (especially the bulky ones, e.g. large arrays) once by placing them in global scope. Allocate enough memory to deal with the largest input of the problem. This way, we do not have to pass the data structures around as function arguments. For problems with multiple test cases, simply clear/reset the contents of the data structure before dealing with each test case.

9. When you have the option to write your code either iteratively or recursively, choose the iterative version. Example: The iterative C++ STL `next_permutation` and iterative subset generation techniques using bitmask shown in Section 3.2.1 are (far) faster than if you write similar routines recursively (mainly due to overheads in function calls).

10. Array access in (nested) loops can be slow. If you have an array `A` and you frequently access the value of `A[i]` (without changing it) in (nested) loops, it may be beneficial to use a local variable `temp = A[i]` and works with `temp` instead.

11. In C/C++, *appropriate* usage of macros or inline functions can reduce runtime.

12. For C++ users: Using C-style character arrays will yield faster execution than when using the C++ STL string. For Java users: Be careful with `String` manipulation as Java `String` objects are immutable. Operations on Java `String`s can thus be very slow. Use Java `StringBuilder` instead.

Browse the Internet or relevant books (e.g. [69]) to find (much) more information on how to speed up your code. Practice this 'code hacking skill' by choosing a harder problem in UVa online judge where the runtime of the best solution is not 0.000s. Submit several variants of your Accepted solution and check the runtime differences. Adopt hacking modification that consistently gives you faster runtime.

Tip 7: Use Better Data Structures & Algorithms :)

No kidding. Using better data structures and algorithms will always outperform any optimizations mentioned in Tips 1-6 above. If you are sure that you have written your fastest Complete Search code, but it is still judged as TLE, abandon the Complete Search approach.

Remarks About Complete Search in Programming Contests

The main source of the 'Complete Search' material in this chapter is the USACO training gateway [48]. We have adopted the name 'Complete Search' rather than 'Brute-Force' (with its negative connotations) as we believe that some Complete Search solutions can be clever and fast. We feel that the term 'clever Brute-Force' is also a little self-contradictory.

If a problem is solvable by Complete Search, it will also be clear when to use the iterative or recursive backtracking approaches. Iterative approaches are used when one can derive the different states *easily* with some formula relative to a certain *counter* and (almost) all states have to be checked, e.g. scanning all the indices of an array, enumerating (almost) all possible subsets of a small set, generating (almost) all permutations, etc. Recursive Backtracking is used when it is hard to derive the different states with a simple index and/or one also wants to (heavily) prune the search space, e.g. the 8-queens chess problem. If the search space of a problem that is solvable with Complete Search is large, then recursive backtracking approaches that allow early pruning of infeasible sections of the search space are usually used. Pruning in iterative Complete Searches is not impossible but usually difficult.

The best way to improve your Complete Search skills is to solve more Complete Search problems. We have provided a list of such problems, separated into several categories below. Please attempt as many as possible, especially those that are highlighted with the **must try *** indicator. Later in Section 3.5, readers will encounter further examples of recursive backtracking, but with the addition of the 'memoization' technique.

Note that we will discuss some *more* advanced search techniques later in Section 8.2, e.g. using bit manipulation in recursive backtracking, harder state-space search, Meet in the Middle, A* Search, Depth Limited Search (DLS), Iterative Deepening Search (IDS), and Iterative Deepening A* (IDA*).

Programming Exercises solvable using Complete Search:

- Iterative (One Loop, Linear Scan)
  1. UVa 00102 - Ecological Bin Packing (just try all 6 possible combinations)
  2. UVa 00256 - Quirksome Squares (brute force, math, pre-calculate-able)
  3. **UVa 00927 - Integer Sequence from ... *** (use sum of arithmetic series)
  4. **UVa 01237 - Expert Enough *** (LA 4142, Jakarta08, input is small)
  5. **UVa 10976 - Fractions Again ? *** (total solutions is asked upfront; therefore do brute force twice)
  6. UVa 11001 - Necklace (brute force math, maximize function)
  7. UVa 11078 - Open Credit System (one linear scan)
- Iterative (Two Nested Loops)
  1. UVa 00105 - The Skyline Problem (height map, sweep left-right)
  2. UVa 00347 - Run, Run, Runaround ... (simulate the process)
  3. UVa 00471 - Magic Numbers (somewhat similar to UVa 725)
  4. UVa 00617 - Nonstop Travel (try all integer speeds from 30 to 60 mph)
  5. UVa 00725 - Division (elaborated in this section)
  6. **UVa 01260 - Sales *** (LA 4843, Daejeon10, check all)
  7. UVa 10041 - Vito's Family (try all possible location of Vito's House)
  8. **UVa 10487 - Closest Sums *** (sort and then do $O(n^2)$ pairings)

9. *UVa 10730 - Antiarithmetic?* (2 nested loops with pruning can pass possibly pass the weaker test cases; note that this brute force solution is too slow for the larger test data generated in the solution of UVa 11129)
10. **UVa 11242 - Tour de France *** (plus sorting)
11. *UVa 12488 - Start Grid* (2 nested loops; simulate overtaking process)
12. *UVa 12583 - Memory Overflow* (2 nested loops; be careful of overcounting)

- Iterative (Three Or More Nested Loops, Easier)

1. UVa 00154 - Recycling (3 nested loops)
2. UVa 00188 - Perfect Hash (3 nested loops, until the answer is found)
3. **UVa 00441 - Lotto *** (6 nested loops)
4. UVa 00626 - Ecosystem (3 nested loops)
5. UVa 00703 - Triple Ties: The ... (3 nested loops)
6. **UVa 00735 - Dart-a-Mania *** (3 nested loops, then count)
7. **UVa 10102 - The Path in the ... *** (4 nested loops will do, we do not need BFS; get max of minimum Manhattan distance from a '1' to a '3'.)
8. UVa 10502 - Counting Rectangles (6 nested loops, rectangle, not too hard)
9. UVa 10662 - The Wedding (3 nested loops)
10. UVa 10908 - Largest Square (4 nested loops, square, not too hard)
11. UVa 11059 - Maximum Product (3 nested loops, input is small)
12. *UVa 11975 - Tele-loto* (3 nested loops, simulate the game as asked)
13. *UVa 12498 - Ant's Shopping Mall* (3 nested loops)
14. *UVa 12515 - Movie Police* (3 nested loops)

- Iterative (Three-or-More Nested Loops, Harder)

1. UVa 00253 - Cube painting (try all, similar problem in UVa 11959)
2. UVa 00296 - Safebreaker (try all 10000 possible codes, 4 nested loops, use similar solution as 'Master-Mind' game)
3. UVa 00386 - Perfect Cubes (4 nested loops with pruning)
4. UVa 10125 - Sumsets (sort; 4 nested loops; plus binary search)
5. UVa 10177 - (2/3/4)-D Sqr/Rects/... (2/3/4 nested loops, precalculate)
6. UVa 10360 - Rat Attack (also solvable using 1024^2 DP max sum)
7. UVa 10365 - Blocks (use 3 nested loops with pruning)
8. *UVa 10483 - The Sum Equals ...* (2 nested loops for a, b, derive c from a, b; there are 354 answers for range [0.01 .. 255.99]; similar with UVa 11236)
9. **UVa 10660 - Citizen attention ... *** (7 nested loops, Manhattan distance)
10. UVa 10973 - Triangle Counting (3 nested loops with pruning)
11. UVa 11108 - Tautology (5 nested loops, try all $2^5 = 32$ values with pruning)
12. ***UVa 11236 - Grocery Store **** (3 nested loops for a, b, c; derive d from a, b, c; check if you have 949 lines of output)
13. UVa 11342 - Three-square (pre-calculate squared values from 0^2 to 224^2, use 3 nested loops to generate the answers; use `map` to avoid duplicates)
14. *UVa 11548 - Blackboard Bonanza* (4 nested loops, string, pruning)
15. **UVa 11565 - Simple Equations *** (3 nested loops with pruning)
16. UVa 11804 - Argentina (5 nested loops)
17. UVa 11959 - Dice (try all possible dice positions, compare with the 2nd one)

Also see Mathematical Simulation in Section 5.2

- Iterative (Fancy Techniques)
  1. UVa 00140 - Bandwidth (max n is just 8, use `next_permutation`; the algorithm inside `next_permutation` is iterative)
  2. *UVa 00234 - Switching Channels* (use `next_permutation`, simulation)
  3. UVa 00435 - Block Voting (only 2^{20} possible coalition combinations)
  4. UVa 00639 - Don't Get Rooked (generate 2^{16} combinations and prune)
  5. ***UVa 01047 - Zones ** ** (LA 3278, WorldFinals Shanghai05, notice that $n \leq 20$ so that we can try all possible subsets of towers to be taken; then apply inclusion-exclusion principle to avoid overcounting)
  6. *UVa 01064 - Network* (LA 3808, WorldFinals Tokyo07, permutation of up to 5 messages, simulation, mind the word 'consecutive')
  7. UVa 11205 - The Broken Pedometer (try all 2^{15} bitmask)
  8. UVa 11412 - Dig the Holes (`next_permutation`, find one possibility from 6!)
  9. **UVa 11553 - Grid Game *** (solve by trying all $n!$ permutations; you can also use DP + bitmask, see Section 8.3.1, but it is overkill)
  10. UVa 11742 - Social Constraints (discussed in this section)
  11. *UVa 12249 - Overlapping Scenes* (LA 4994, KualaLumpur10, try all permutations, a bit of string matching)
  12. *UVa 12346 - Water Gate Management* (LA 5723, Phuket11, try all 2^n combinations, pick the best one)
  13. *UVa 12348 - Fun Coloring* (LA 5725, Phuket11, try all 2^n combinations)
  14. *UVa 12406 - Help Dexter* (try all 2^p possible bitmasks, change '0's to '2's)
  15. ***UVa 12455 - Bars ** (discussed in this section)
- Recursive Backtracking (Easy)
  1. UVa 00167 - The Sultan Successor (8-queens chess problem)
  2. UVa 00380 - Call Forwarding (simple backtracking, but we have to work with strings, see Section 6.2)
  3. UVa 00539 - The Settlers ... (longest simple path in a *small* general graph)
  4. **UVa 00624 - CD *** (input size is small, backtracking is enough)
  5. UVa 00628 - Passwords (backtracking, follow the rules in description)
  6. UVa 00677 - All Walks of length "n" ... (print all solutions with backtracking)
  7. UVa 00729 - The Hamming Distance ... (generate all possible bit strings)
  8. UVa 00750 - 8 Queens Chess Problem (discussed in this section with sample source code)
  9. UVa 10276 - Hanoi Tower Troubles Again (insert a number one by one)
  10. UVa 10344 - 23 Out of 5 (rearrange the 5 operands and the 3 operators)
  11. UVa 10452 - Marcus, help (at each pos, Indy can go forth/left/right; try all)
  12. **UVa 10576 - Y2K Accounting Bug *** (generate all, prune, take max)
  13. **UVa 11085 - Back to the 8-Queens *** (see UVa 750, pre-calculation)
- Recursive Backtracking (Medium)
  1. UVa 00222 - Budget Travel (looks like a DP problem, but the state cannot be memoized as 'tank' is floating-point; fortunately, the input is not large)
  2. *UVa 00301 - Transportation* (2^{22} with pruning is possible)
  3. UVa 00331 - Mapping the Swaps ($n \leq 5$...)
  4. UVa 00487 - Boggle Blitz (use `map` to store the generated words)
  5. **UVa 00524 - Prime Ring Problem *** (also see Section 5.5.1)

6. UVa 00571 - Jugs (solution can be suboptimal, add flag to avoid cycling)
7. **UVa 00574 - Sum It Up *** (print all solutions with backtracking)
8. UVa 00598 - Bundling Newspaper (print all solutions with backtracking)
9. *UVa 00775 - Hamiltonian Cycle* (backtracking suffices because the search space cannot be that big; in a dense graph, it is more likely to have a Hamiltonian cycle, so we can prune early; we do NOT have to find the best one like in TSP problem)
10. *UVa 10001 - Garden of Eden* (the upperbound of 2^{32} is scary but with efficient pruning, we can pass the time limit as the test case is not extreme)
11. *UVa 10063 - Knuth's Permutation* (do as asked)
12. *UVa 10460 - Find the Permuted String* (similar nature with UVa 10063)
13. UVa 10475 - Help the Leaders (generate and prune; try all)
14. **UVa 10503 - The dominoes solitaire *** (max 13 spaces only)
15. *UVa 10506 - Ouroboros* (any valid solution is AC; generate all possible next digit (up to base 10/digit [0..9]); check if it is still a valid Ouroboros sequence)
16. *UVa 10950 - Bad Code* (sort the input; run backtracking; the output should be sorted; only display the first 100 sorted output)
17. UVa 11201 - The Problem with the ... (backtracking involving strings)
18. *UVa 11961 - DNA* (there are at most 4^{10} possible DNA strings; moreover, the mutation power is at most $K \leq 5$ so the search space is much smaller; sort the output and then remove duplicates)

- Recursive Backtracking (Harder)

1. *UVa 00129 - Krypton Factor* (backtracking, string processing check, a bit of output formatting)
2. UVa 00165 - Stamps (requires some DP too; can be pre-calculated)
3. **UVa 00193 - Graph Coloring *** (Max Independent Set, input is small)
4. UVa 00208 - Firetruck (backtracking with some pruning)
5. **UVa 00416 - LED Test *** (backtrack, try all)
6. UVa 00433 - Bank (Not Quite O.C.R.) (similar to UVa 416)
7. UVa 00565 - Pizza Anyone? (backtracking with lots of pruning)
8. *UVa 00861 - Little Bishops* (backtracking with pruning as in 8-queens recursive backtracking solution; then pre-calculate the results)
9. UVa 00868 - Numerical maze (try row 1 to N; 4 ways; some constraints)
10. ***UVa 01262 - Password **** (LA 4845, Daejeon10, sort the columns in the two 6×5 grids first so that we can process common passwords in lexicographic order; backtracking; important: skip two similar passwords)
11. UVa 10094 - Place the Guards (this problem is like the n-queens chess problem, but must find/use the pattern!)
12. *UVa 10128 - Queue* (backtracking with pruning; try up to all N! (13!) permutations that satisfy the requirement; then pre-calculate the results)
13. UVa 10582 - ASCII Labyrinth (simplify complex input first; then backtrack)
14. *UVa 11090 - Going in Cycle* (minimum mean weight cycle problem; solvable with backtracking with important pruning when current running mean is greater than the best found mean weight cycle cost)

3.3 Divide and Conquer

Divide and Conquer (abbreviated as D&C) is a problem-solving paradigm in which a problem is made *simpler* by 'dividing' it into smaller parts and then conquering each part. The steps:

1. Divide the original problem into *sub*-problems—usually by half or nearly half,
2. Find (sub)-solutions for each of these sub-problems—which are now easier,
3. If needed, combine the sub-solutions to get a complete solution for the main problem.

We have seen examples of the D&C paradigm in the previous sections of this book: Various sorting algorithms (e.g. Quick Sort, Merge Sort, Heap Sort) and Binary Search in Section 2.2 utilize this paradigm. The way data is organized in Binary Search Tree, Heap, Segment Tree, and Fenwick Tree in Section 2.3, 2.4.3, and 2.4.4 also relies upon the D&C paradigm.

3.3.1 Interesting Usages of Binary Search

In this section, we discuss the D&C paradigm in the well-known Binary Search algorithm. We classify Binary Search as a 'Divide' and Conquer algorithm although one reference [40] suggests that it should be actually classified as 'Decrease (by-half)' and Conquer as it does not actually 'combine' the result. We highlight this algorithm because many contestants know it, but not many are aware that it can be used in many other non-obvious ways.

Binary Search: The Ordinary Usage

Recall that the *canonical* usage of Binary Search is searching for an item in a *static sorted array*. We check the middle of the sorted array to determine if it contains what we are looking for. If it is or there are no more items to consider, stop. Otherwise, we can decide whether the answer is to the left or right of the middle element and continue searching. As the size of search space is halved (in a binary fashion) after each check, the complexity of this algorithm is $O(\log n)$. In Section 2.2, we have seen that there are built-in library routines for this algorithm, e.g. the C++ STL `algorithm::lower_bound` (and the Java `Collections.binarySearch`).

This is *not* the only way to use binary search. The pre-requisite for performing a binary search—a *static sorted sequence (array or vector)*—can also be found in other uncommon data structures such as in the root-to-leaf path of a tree (not necessarily binary nor complete) that satisfies the *min heap* property. This variant is discussed below.

Binary Search on Uncommon Data Structures

This original problem is titled 'My Ancestor' and was used in the Thailand ICPC National Contest 2009. Abridged problem description: Given a weighted (family) tree of up to $N \leq 80K$ vertices with a special trait: *Vertex values are increasing from root to leaves.* Find the *ancestor* vertex closest to the root from a starting vertex v that has weight at least P. There are up to $Q \leq 20K$ such *offline* queries. Examine Figure 3.3 (left). If $P = 4$, then the answer is the vertex labeled with 'B' with value 5 as it is the ancestor of vertex v that is closest to root 'A' and has a value of ≥ 4. If $P = 7$, then the answer is 'C', with value 7. If $P \geq 9$, there is no answer.

The naïve solution is to perform a linear $O(N)$ scan per query: Starting from the given vertex v, we move up the (family) tree until we reach the first vertex whose direct parent has value $< P$ or until we reach the root. If this vertex has value $\geq P$ and it is not vertex v

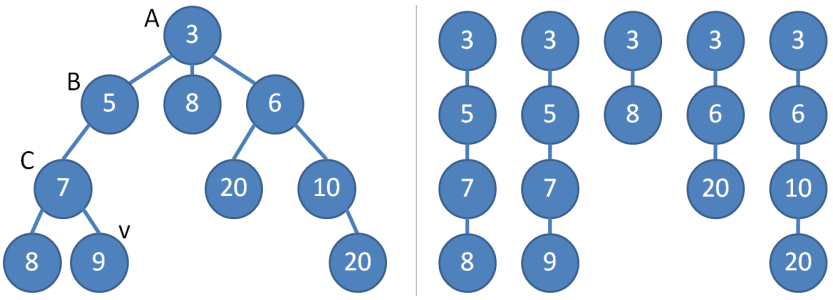

Figure 3.3: My Ancestor (all 5 root-to-leaf paths are sorted)

itself, we have found the solution. As there are Q queries, this approach runs in $O(QN)$ (the input tree can be a sorted linked list, or rope, of length N) and will get a TLE as $N \leq 80K$ and $Q \leq 20K$.

A better solution is to store all the $20K$ queries (we do not have to answer them immediately). Traverse the tree *just once* starting from the root using the $O(N)$ preorder tree traversal algorithm (Section 4.7.2). This preorder tree traversal is slightly modified to remember the partial root-to-current-vertex sequence as it executes. The array is always sorted because the vertices along the root-to-current-vertex path have increasing weights, see Figure 3.3 (right). The preorder tree traversal on the tree shown in Figure 3.3 (left) produces the following partial root-to-current-vertex sorted array: {{3}, {3, 5}, {3, 5, 7}, {3, 5, 7, 8}, backtrack, {3, 5, 7, 9}, backtrack, backtrack, backtrack, {3, 8}, backtrack, {3, 6}, {3, 6, 20}, backtrack, {3, 6, 10}, and finally {3, 6, 10, 20}, backtrack, backtrack, backtrack (done)}.

During the preorder traversal, when we land on a queried vertex, we can perform a $O(\log N)$ **binary search** (to be precise: `lower_bound`) on the partial root-to-current-vertex weight array to obtain the ancestor closest to the root with a value of at least P, recording these solutions. Finally, we can perform a simple $O(Q)$ iteration to output the results. The overall time complexity of this approach is $O(Q \log N)$, which is now manageable given the input bounds.

Bisection Method

We have discussed the applications of Binary Searches in finding items in static sorted sequences. However, the binary search **principle**[4] can also be used to find the root of a function that may be difficult to compute directly.

Example: You buy a car with loan and now want to pay the loan in monthly installments of d dollars for m months. Suppose the value of the car is originally v dollars and the bank charges an interest rate of i% for any unpaid loan at the end of each month. What is the amount of money d that you must pay per month (to 2 digits after the decimal point)?

Suppose $d = 576.19$, $m = 2$, $v = 1000$, and $i = 10\%$. After one month, your debt becomes $1000 \times (1.1) - 576.19 = 523.81$. After two months, your debt becomes $523.81 \times (1.1) - 576.19 \approx 0$. If we are only given $m = 2$, $v = 1000$, and $i = 10\%$, how would we determine that $d = 576.19$? In other words, find the root d such that the debt payment function $f(d, m, v, i) \approx 0$.

An *easy* way to solve this root finding problem is to use the bisection method. We pick a reasonable range as a starting point. We want to fix d within the range `[a..b]` where

[4]We use the term 'binary search principle' to refer to the D&C approach of halving the range of possible answers. The 'binary search algorithm' (finding index of an item in a sorted array), the 'bisection method' (finding the root of a function), and 'binary search the answer' (discussed in the next subsection) are all instances of this principle.

$a = 0.01$ as we have to pay at least one cent and $b = (1 + i\%) \times v$ as the earliest we can complete the payment is $m = 1$ if we pay exactly $(1 + i\%) \times v$ dollars after one month. In this example, $b = (1 + 0.1) \times 1000 = 1100.00$ dollars. For the bisection method to work[5], we must ensure that the function values of the two extreme points in the initial Real range `[a..b]`, i.e. $f(a)$ and $f(b)$ have opposite signs (this is true for the computed `a` and `b` above).

a	**b**	**d** $= \frac{a+b}{2}$	**status:** $f(d, m, v, i)$	**action**
0.01	1100.00	550.005	undershoot by 54.9895	increase d
550.005	1100.00	825.0025	overshoot by 522.50525	decrease d
550.005	825.0025	687.50375	overshoot by 233.757875	decrease d
550.005	687.50375	618.754375	overshoot by 89.384187	decrease d
550.005	618.754375	584.379688	overshoot by 17.197344	decrease d
550.005	584.379688	567.192344	undershoot by 18.896078	increase d
567.192344	584.379688	575.786016	undershoot by 0.849366	increase d
...	...	...	a **few** iterations later ...	...
...	...	576.190476	stop; error is now less than ϵ	answer = 576.19

Table 3.1: Running Bisection Method on the Example Function

Notice that bisection method only requires $O(\log_2((b - a)/\epsilon))$ iterations to get an answer that is good enough (the error is smaller than the threshold error ϵ that we can tolerate). In this example, bisection method only takes $\log_2 1099.99/\epsilon$ tries. Using a small $\epsilon =$ `1e-9`, this yields only ≈ 40 iterations. Even if we use a smaller $\epsilon =$ `1e-15`, we will still only need ≈ 60 tries. Notice that the number of tries is *small*. The bisection method is much more efficient compared to exhaustively evaluating each possible value of $d =$`[0.01..1100.00]`$/\epsilon$ for this example function. Note: The bisection method can be written with a loop that tries the values of $d \approx 40$ to 60 times (see our implementation in the 'binary search the answer' discussion below).

Binary Search the Answer

The abridged version of UVa 11935 - Through the Desert is as follows: Imagine that you are an explorer trying to cross a desert. You use a jeep with a 'large enough' fuel tank – initially full. You encounter a series of events throughout your journey such as 'drive (that consumes fuel)', 'experience gas leak (further reduces the amount of fuel left)', 'encounter gas station (allowing you to refuel to the original capacity of your jeep's fuel tank)', 'encounter mechanic (fixes all leaks)', or 'reach goal (done)'. You need to determine the *smallest possible* fuel tank capacity for your jeep to be able to reach the goal. The answer must be precise to three digits after decimal point.

If we know the jeep's fuel tank capacity, then this problem is just a simulation problem. From the start, we can simulate each event in order and determine if the goal can be reached without running out of fuel. The problem is that we do not know the jeep's fuel tank capacity—this is the value that we are looking for.

From the problem description, we can compute that the range of possible answers is between `[0.000..10000.000]`, with 3 digits of precision. However, there are $10M$ such possibilities. Trying each value sequentially will get us a TLE verdict.

Fortunately, this problem has a property that we can exploit. Suppose that the correct answer is X. Setting your jeep's fuel tank capacity to any value between `[0.000..X-0.001]`

[5]Note that the requirements for the bisection method (which uses the binary search principle) are slightly different from the binary search algorithm which needs a sorted array.

will *not* bring your jeep safely to the goal event. On the other hand, setting your jeep fuel tank volume to any value between [X..10000.000] will bring your jeep safely to the goal event, usually with some fuel left. This property allows us to binary search the answer X! We can use the following code to obtain the solution for this problem.

```
#define EPS 1e-9  // this value is adjustable; 1e-9 is usually small enough
bool can(double f) {               // details of this simulation is omitted
  // return true if the jeep can reach goal state with fuel tank capacity f
  // return false otherwise
}

// inside int main()
  // binary search the answer, then simulate
  double lo = 0.0, hi = 10000.0, mid = 0.0, ans = 0.0;
  while (fabs(hi - lo) > EPS) {         // when the answer is not found yet
    mid = (lo + hi) / 2.0;                          // try the middle value
    if (can(mid)) { ans = mid; hi = mid; } // save the value, then continue
    else          lo = mid;
  }

  printf("%.3lf\n", ans);     // after the loop is over, we have the answer
```

Note that some programmers choose to use a constant number of refinement iterations instead of allowing the number of iterations to vary dynamically to avoid precision errors when testing fabs(hi - lo) > EPS and thus being trapped in an infinite loop. The only changes required to implement this approach are shown below. The other parts of the code are the same as above.

```
  double lo = 0.0, hi = 10000.0, mid = 0.0, ans = 0.0;
  for (int i = 0; i < 50; i++) {    // log_2 ((10000.0 - 0.0) / 1e-9) ~= 43
    mid = (lo + hi) / 2.0;     // looping 50 times should be precise enough
    if (can(mid)) { ans = mid; hi = mid; }
    else          lo = mid;
  }
```

Exercise 3.3.1.1: There is an alternative solution for UVa 11935 that does not use 'binary search the answer' technique. Can you spot it?

Exercise 3.3.1.2*: The example shown here involves binary-searching the answer where the answer is a floating point number. Modify the code to solve 'binary search the answer' problems where the answer lies in an *integer range*!

Remarks About Divide and Conquer in Programming Contests

The Divide and Conquer paradigm is usually utilized through popular algorithms that rely on it: Binary Search and its variants, Merge/Quick/Heap Sort, and data structures: Binary Search Tree, Heap, Segment Tree, Fenwick Tree, etc. However—based on our experience, we reckon that the most commonly used form of the Divide and Conquer paradigm in