CS3233
Competitive Programming

Dr. Steven Halim

Week 02 – Data Structures & Libraries

Focus on Bit Manipulation & Binary Indexed Tree
Outline

• Mini Contest #1 + Break + Discussion + Admins

• **Data Structures With Built-in Libraries**
  – Just a *very quick* walkthrough
    • The pace of this lecture may be frightening for some students...
      – Additional help session on Saturday, 26 Jan 2013, 10am-12pm @ PL6
    • Read the book (Chapter 2) + experiment with the details on your own
      – Linear Data Structures (CS1010/1st half of CS2020)
      – Non Linear Data Structures (CS2010/2nd half of CS2020)

• **Data Structures With Our-Own Libraries**
  – Focus on Binary Indexed (Fenwick) Tree
Basic knowledge that all ICPC/IOI-ers must have!

LINEAR DATA STRUCTURES WITH BUILT-IN LIBRARIES
I am...

1. A pure C coder
2. A pure C++ coder
3. A mix between C/C++ coder
4. A pure Java coder
5. A multilingual coder: C/C++/Java
Linear DS + Built-In Libraries (1)

1. Static Array, built-in support in C/C++/Java

2. Resize-able: C++ STL \texttt{vector}, Java \texttt{Vector}
   - Both are very useful in ICPCs/IOIs
   - PS: Java \texttt{ArrayList} \textit{may be} slightly faster

• There are 2 very common operations on Array:
  - Sorting
  - Searching
  - Let’s take a look at efficient ways to do them
Two “fundamental” CS problems

SORTING + SEARCHING
INVOLVING ARRAY
Sorting (1)

• Definition:
  – Given unsorted stuffs, sort them...

• Popular Sorting Algorithms
  – O(n^2) algorithms: Bubble/Selection/Insertion Sort
  – O(n \log n) algorithms: Merge/Quick/Heap Sort
  – Special purpose: Counting/Radix/Bucket Sort

• Reference:
Sorting (2)

• In ICPC, you can “forget” all these...
  – In general, if you need to sort something..., just use the $O(n \log n)$ sorting library:
    • C++ STL `algorithm:: sort`
    • Java `Collections.sort`

• In ICPC, sorting is either used as *preliminary step* for more complex algorithm or to *beautify output*
  – Familiarity with sorting libraries is a must!
Sorting (3)

• Sorting routines in C++ STL algorithm
  – sort – a bug-free implementation of *introsort*
    • Fast, it runs in $O(n \log n)$
    • Can sort basic data types (ints, doubles, chars), Abstract Data Types (C++ class), multi-field sorting ($\geq 2$ criteria)
  – partial_sort – implementation of *heapsort*
    • Can do $O(k \log n)$ sorting, if we just need top-k sorted!
  – stable_sort
    • If you need to have the sorting ‘stable’, keys with same values appear in the same order as in input
Searching in Array

• Two variants:
  – When the array is sorted versus not sorted
• Must do $O(n)$ linear scan if not sorted - trivial
• Can use $O(\log n)$ binary search when sorted
  – PS: must run an $O(n \log n)$ sorting algorithm once
• Binary search is ‘tricky’ to code!
  – Instead, use C++ STL `algorithm::lower_bound`
    or Java `Collections.binarySearch`
Linear DS + Built-In Libraries (2)

3. Array of Boolean: C++ STL **bitset**
   - Should be faster than **array of Booleans** or **vector<bool>**!
   - No specific API in Java that is similar to this

4. Bitmask (One important point of this lecture)
   - a.k.a. lightweight set of Boolean or bit string
   - Explanation via:
Linear DS + Built-In Libraries (3)

5. Linked List, C++ STL **list**, Java **LinkedList**
   – Usually not used... just use **vector**!

6. Stack, C++ STL **stack**, Java **Stack**
   – Used by default in Recursion, Postfix Conversion/Calculation, Bracket Matching, etc

7. Queue, C++ STL **queue**, Java **Queue**
   – Used in Breadth First Search, Topological Sort, etc

8. Deque, C++ STL **deque**, Java **Deque**
   – Used in algorithms for ‘Sliding Window’ problem, etc
More efficient data structures

NON-LINEAR DATA STRUCTURES
WITH BUILT-IN LIBRARIES
Binary Search Tree (1)

- ADT Table (key $\rightarrow$ data)
- Binary Search Tree (BST)
  - Advertised $O(\log n)$ for insert, search, and delete
  - Requirement: the BST must be balanced!
    - AVL tree, Red-Black Tree, etc... *argh*
- Fret not, just use: C++ STL map (Java TreeMap)
  - UVa 10226 (Hardwood Species)*
Binary Search Tree (2)

- ADT Table (key exists or not)
- Set (Single Set)
  - C++ STL `set`, similar to C++ STL `map`
    - map stores a `(key, data)` pair
    - set stores just the `key`
  - In Java: `TreeSet`
- Example:
  - UVa 11849 – CD
Heap

- Heap
  - C++ STL **algorithm** has some heap algorithms
    - `partial_sort` uses heapsort
  - C++ STL **priority_queue** (Java **PriorityQueue**) is heap
    - Prim’s and Dijkstra’s algorithms use priority queue
- But, we rarely see pure heap problems in ICPC
Hash Table

- Hash Table
  - Advertised $O(1)$ for insert, search, and delete, but:
    - The hash function must be good!
    - There is no Hash Table in C++ STL (∃ in Java API)
  - Nevertheless, $O(\log n)$ using `map` is usually ok

- Direct Addressing Table (DAT)
  - Rather than hashing, we more frequently use DAT
  - UVa 11340 (Newspaper)
Quick Check

1. I can cope with this pace...
2. I am lost with so many new information in the past few slides
5 Minutes Break

- One data structures *without* built-in libraries will be discussed in the last part...
  - Binary Indexed (Fenwick) Tree
  - Graph, Union-Find Disjoint Sets, and Segment Tree are not discussed in this year’s CS3233 Week02
    - Graph DS is covered in details in CS2010/CS2020
    - UFDS is covered briefly in CS2010/CS2020
    - Please study Segment Tree on your own
      - *We try* not to set any contest problem involving Segment Tree
Graph (not discussed today, revisited in Week06/07/08)
Union-Find Disjoint Sets (not discussed today, read Ch2 on your own)
Segment Tree (not discussed today, read Ch2 on your own)
Fenwick Tree (discussed today)

DATA STRUCTURES
WITHOUT BUILT-IN LIBRARIES
Fenwick Tree – Basics (1)

• Cumulative Frequency Table
  – Example, \( s = \{2,4,5,5,6,6,7,7,8,9\} \) (already sorted)

<table>
<thead>
<tr>
<th>Index/Score/Symbol</th>
<th>Frequency</th>
<th>Cumulative Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>- (index 0 is ignored)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Fenwick Tree – Basics (2)

• Fenwick Tree (inventor = Peter M. Fenwick)
  – Also known as “Binary Indexed Tree”, very aptly named
  – Implemented as an array, let call the array name as \textbf{ft}
  – We will frequently use this bit manipulation, remember!
    • \textbf{LSOne(i)} = Least Significant One of \textbf{i} computed via \textbf{i} & \textbf{(-i)}
Fenwick Tree – Basics (3)

- Each index \(i\) of \(ft\) is responsible for certain range: [i-LSOne(i)+1 .. i]
- \(ft[i]\) stores the cumulative frequency of elements: \{i-LSOne(i)+1, i-LSOne(i)+2, i-LSOne(i)+3, ..., i\}

<table>
<thead>
<tr>
<th>Key/I</th>
<th>Binary</th>
<th>Range</th>
<th>F</th>
<th>CF</th>
<th>FT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>[1..1]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>[1..2]</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>[3..3]</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>[1..4]</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>[5..5]</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>[5..6]</td>
<td>3</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>[7..7]</td>
<td>2</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>[1..8]</td>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>[9..9]</td>
<td>1</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>[9..10]</td>
<td>0</td>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>
Fenwick Tree – RSQ (1)

• To get the cumulative frequency from index 1 to b, use \texttt{rsq}(b)
  – The answer is the sum of sub-frequencies stored in array \texttt{ft} with indices related to b via this formula \( b' = b - \text{LSOne}(b) \)
  – Apply this formula iteratively until b is 0
  – Example: \( \text{rsq}(6) \)
    • \( b = 6 = 0110 \), \( b' = b - \text{LSOne}(b) = 0110 - 0010, b' = 4 = 0100 \)
    • \( b' = 4 = 0100 \), \( b'' = b' - \text{LSOne}(b') = 0100 - 0100, b'' = 0 \rightarrow \text{stop} \)
    • Sum \( \text{ft}[6]+\text{ft}[4] = 5+2 = 7 \) (the pink area covers range \([1..4]+[5..6] = [1..6]\) )

Analysis:
This is O(log n)

Why?
Fenwick Tree – RSQ (2)

• To get the cumulative frequency from index $a$ to $b$, use $rsq(a, b)$
  – If $a$ is greater than one, we use: $rsq(b) - rsq(a-1)$
  – Example: $rsq(4, 6)$
    • $rsq(4, 6) = rsq(6) - rsq(4-1) = rsq(6) - rsq(3) = (5+2) - (0+1) = 7 - 1 = 6$

Analysis:
This is $O(2 \log n) = O(\log n)$

Why?
Fenwick Tree – Update

- To update the frequency of an key/index $k$ by $v$ ($v$ is either positive or negative), use $\text{adjust}(k, v)$
  - Indices that are related to $k$ via $k' = k + \text{LSOne}(k)$ will be updated by $v$ when $k < \text{ft.size}()$
  - Example: $\text{adjust}(5, 1)$
    - $k = 5 = 0101$, $k' = k + \text{LSOne}(k) = 0101 + 0001$, $k' = 6 = 0110$
    - $k' = 6 = 0110$, $k'' = k' + \text{LSOne}(k') = 0110 + 0010$, $k'' = 8 = 1000$
    - $k'' = 8 = 1000$, $k''' = k'' + \text{LSOne}(k'') = 1000 + 1000$, $k''' = 16 = 10000 \rightarrow \text{stop}$
  - Observe that the pink line in the figure below stabs through the ranges that are under the responsibility of indices 5, 6, and 8
    - $\text{ft}[5]$, 2 updated to 3
    - $\text{ft}[6]$, 5 updated to 6
    - $\text{ft}[8]$, 10 updated to 11

Analysis:
This is also $O(\log n)$

Why?
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; }
};

FT/BIT is in IOI syllabus!
Fenwick Tree – Sample Application

- Fenwick Tree is very suitable for dynamic RSQs (cumulative frequency table) where update occurs on a certain index only
- Now, think of potential real-life applications!
  - [http://uhunt.felix-halim.net/id/32900](http://uhunt.felix-halim.net/id/32900)
  - Consider code runtime of [0.000 - 9.999]s for a particular UVa problem
    - There are up to 10+ million submissions/codes
      - About thousands submissions per problem
    - If your code runs in 0.342 secs, what is your rank?
- How to use Fenwick Tree to deal with this problem?
Quick Check

1. I am lost with Fenwick Tree
2. I understand the basics of Fenwick Tree, but since this is new for me, I may/may not be able to recognize problems solvable with FT
3. I have solved several FT-related problems before
Summary

• There are a lot of *great* Data Structures out there
  – We need the most efficient one for our problem
    • Different DS suits different problem!

• Many of them have **built-in libraries**
  – For some others, we have to build **our own (focus on FT)**
    • Study these libraries! Do not rebuild them during contests!

• From Week03 onwards and future ICPCs/IOIs, use C++ STL and/or Java API and our built-in libraries!
  – Now, your team should be in rank 30-45 (from 60)
    (still solving ~1-2 problems out of 10, but faster)
References

• **Competitive Programming 2.9**, Chapter 2
  – Steven, Felix 😊

• **A new data structure for cumulative frequency table**
  – Peter M Fenwick

• **Fenwick Tree @ TopCoder**
  – By bobal5551
Study These Visualizations


- You can use your smart phones/tablet PCs to access them 😊
- Google searches (as of last year), there is no other visualizations on bitmask/BIT like these
- PS: Report bugs to Steven, if any