Advanced Algorithmic Problem Solving

Le 1 – Data Structures

Fredrik Heintz
Dept of Computer and Information Science
Linköping University

Outline

- Basic data structures (UVA 10107, UVA 902)
- Union-Find (lab 1.4)
- Fenwick Trees (lab 1.5)
- Segment Trees (UVA 11402)

Time Limits and Computational Complexity

- The normal time limit for a program is a few seconds.
- You may assume that your program can do about 100M operations within this time limit.

n	Worst AC Complexity	Comments
≤ [10..11]	$O(n!)$, $O(n^6)$	Enumerating permutations
≤ [15..18]	$O(2^n \times n^2)$	DP TSP
≤ [18..22]	$O(2^n \times n)$	DP with bitmask technique
≤ 100	$O(n^4)$	DP with 3 dimensions and O(n) loop
≤ 450	$O(n^3)$	Floyd Warshall's (APSP)
≤ 2K	$O(n^2 \log_2 n)$	2-nested loops + tree search
≤ 10K	$O(n^2)$	Bubble/Selection/Insertion sort
≤ 1M	$O(n \log_2 n)$	Merge Sort, Binary search
≤ 100M	$O(n)$, $O(\log_2)$, $O(1)$	Simulation, find average

Basic Data Structures

- Linear data structures
  - Pair, tuple (C++11)
  - static array
  - vector (ArrayList or Vector)
  - bitset (BitSet)
  - stack (Stack)
  - queue (Queue)
  - deque (Deque)
- Linked list data structures
  - list (LinkedList)
- Tree-like data structures
  - priority queue (PriorityQueue)
    - C++ max heap, Java min heap
  - set (TreeSet), multiset
  - map (TreeMap), multimap
  - unordered_map (HashMap/HashSet/HashTable), unordered_multimap (C++11)

Example Problem: UVA 10107 and 902

- UVA 10107: Compute the median of *n* integers
  - vector<int> that is extended and sorted allows to take out the median in O(1), O(n log n) => 1M elements
  - Linked list, insert in the right place to keep sorted (basically insertion sort)
  - Balanced tree, keep sorted (basically heap sort), find median element using binary search (?)
- UVA 902: Find the most frequent string of length *n* in a text *t*
  - Create a map<string, id> counting the frequency of each substring of length *n*, *O(t log tn)* => 1M elements

Union-Find Disjoint Sets

- Union-Find Disjoint Sets is a data structure for storing a set of disjoint sets where it is very efficient (~*O(1)*) to *find* which set an element belongs to and to *merge* two sets.
- The disjoint sets are represented by a *forest of trees*, where the root of a tree is the representative element for that set.
- To improve the performance use the *union-by-rank* and *path-compression* heuristics.
- Example usage: Finding connected components in an undirected graph or Kruskal's algorithm for finding a Minimum Spanning Tree.
- In Lab 1.5 you will implement this data structure
- In Exercise 1 (Almost Union-Find you will implement an extended version of the data structure which also supports *delete* and *move*)

Fenwick Tree

- A Fenwick Tree is an efficient data structure for computing range sum queries with updates, both in *O(log n)*.
  - An example range sum is cumulative frequencies, in which case *n* is the highest value in the data.
- If the data is static then the range sums can be precomputed in *O(n)* (rsq[i] = rsq[i-1] + A[i]).
- The cost of building a Fenwick Tree is *O(m log n)*, where *m* is the number of data points.
- A Fenwick Tree only stores range sums, not the original values, which makes it very space efficient (*O(n)*).
- A Fenwick Tree is a binary tree where element *i* stores the range sum query for [*i*-LSOne(*i*)+1, *i*-LSOne(*i*)+2, ..., *i*], where LSOne(*i*) is the least significant one in the binary representation of *i*.
- The range sum for any range [i,j] can be computed as rsq(j) – rsq(i-1).
- Fenwick Trees can be extended to d-dimensional data with query and update operations in $O(2^d \log^d n)$.

Segment Tree

- A Segment Tree is an efficient data structure for computing range queries with updates, both in *O(log n)*.
- Example range queries are range min/max queries and range sum queries.
- If the data is static then the range min/max queries can be precomputed in *O(n log n)*.
- A Segment Tree is a binary tree where the root has index 1 and the index of the left/right child of index *p* is 2*p*/2*p*+1.
- RMQ(i,i) = A[i].
  For RMQ(i,j), let p1=RMQ(i, (i+j)/2) and p2=RMQ((i+j)/2+1, j), RMQ(i,j)=p1 if A[p1]≤A[p2], otherwise p2.

Fenwick Tree vs Segment Tree

Feature	Segment Tree	Fenwick Tree
Build tree from array	$O(n)$	$O(m \log n)$
Dynamic RMin/RMaxQ	Ok	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

Example Problem: UVA 11402

Summary

- Learn to use basic data structures in standard libraries such as vector, map, stack, queue, priority queue and set.
- Use a Union-Find data structure to represent collections of disjoint sets when you need to efficiently check membership and merge sets. Can be extended to handle move and delete.
- Use a Fenwick Tree to compute range sum queries when the data needs to be updated between queries. Can be extended to *d*-dimensional data.
- Use a Segment Tree to compute range min/max queries when the data needs to be updated between queries.