Algorithmic Problem Solving
AAPS21
Le 2 – Data Structures

Fredrik Heintz
Dept of Computer and Information Science
Linköping University

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

- **UVA 10107:** Compute the median of *n* integers
  - vector<int> that is sorted allows to take out the median in O(1) time.
  - 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.

Example Problem: UVA 902

- **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)*.

Interval Cover (Greedy)

Input: Goal interval $[s, f]$ and the intervals $[s_1, f_1], .., [s_n, f_n]$ such that $[s, f] \subset \bigcup [s_i, f_i]$.

Output: A smallest subset $S \subset \{1, \ldots, n\}$ such that $[s, f] \subset \bigcup_{i \in S} [s_i, f_i]$.

INTERVALCOVER($[s_1, f_1], \ldots, [s_n, f_n]$)

(1) $S \leftarrow \emptyset, t \leftarrow s$
(2) **while** $t < f$
(3) $f_i \leftarrow \max\{f_j | s_j \leq t \leq f_j\}$
(4) $S \leftarrow S \cup \{i\}$
(5) $t \leftarrow f_i$
(6) **return** S

(pseudocode by Wendin et al.)

- ***Exercise 1:*** *Why is this greedy algorithm optimal?*
- ***Exercise 2:*** *How can this be improved to O(n log n)?*

Knapsack (DP)

- Given a capacity ***C***, and some objects that each have a (possibly unique) weight and value, find the highest valued set of objects that you can carry.
- Can be solved with dynamic programming.

Knapsack (DP)

- Subproblem = "should I put the item with index *i* in the knapsack if I carry *w* weight"?
- Decision state = *"(i, w)"*.
- Store the solutions for each such subproblem in an array ***dp[|objects|][max weight]***.
- Time complexity: ***O(|objects| x max weight)***.

The knapsack problem generalizes MANY well-known optimization problems.

Disjoint Set (Union-Find)

- The **disjoint set** 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* ("**unify**") 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.

Disjoint Set (Union-Find)

- To improve the performance use *path-compression*.
- Example usage: Finding connected components in an undirected graph or Kruskal's algorithm for finding a Minimum Spanning Tree.
- In Almost Union-Find you implemented 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)*.
- Naively, *O(n)*, so this is a great improvement.
- A Fenwick tree only stores range sums, not the original values.
- Basic idea: Each integer can be represented as sum of powers of two. In the same way, cumulative frequency can be represented as sum of sets of subfrequencies. In our case, each set contains some successive number of non-overlapping frequencies.

Fenwick Tree

- 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 $O(2^d \log^d n)$.

Fenwick Tree

- **Tip 1:** Read the original paper by Fenwick! It is very concise and highly explanatory.
- **Tip 2:** Implementing the Fenwick tree only takes a few lines of code: Look at implementations online for inspiration (LSB)!
- **Tip 3:** Learn how to use the Fenwick tree.
- Also known as BIT (binary indexed tree).

Segment Tree

- A Segment Tree is an efficient data structure for computing range queries with range 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(n 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

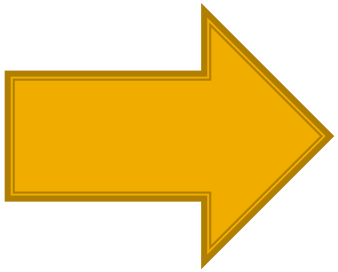
Tips

- ***Debugging:*** If you get stuck for too long, ask me for test data that breaks your algorithm. You will learn a lot by trying to understand why certain "more complex" input data kills your program.
- If you're still stuck, take a break or try another problem.

Tips

- If you are using C++:
  #include <bits/stdc++.h> can be very helpful during problem-solving sessions.

```
3  #include <cstdio>
4  #include <cmath>
5  #include <cstring>
6  #include <cctype>
7  #include <string>
8  #include <vector>
9  #include <list>
10 #include <set>
11 #include <unordered_set>
12 #include <map>
13 #include <unordered_map>
14 #include <queue>
15 #include <stack>
16 #include <algorithm>
17 #include <bitset>
18 #include <functional>
19 #include <climits>
20 #include <limits>
21 #include <cstdlib>
22 #include <ctime>
23 #include <iomanip>
```



```
1 #include <bits/stdc++.h>
2 using namespace std;
```

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.