1. Review of Algorithmic Techniques

Fill in the following table by marking, for each algorithm, those generic algorithmic techniques that were employed in its design. Think carefully: what is the exact problem each algorithm solves and what approach its authors have taken to tackle it. You might see more than one suitable option for some algorithms. Be ready to explain.

<table>
<thead>
<tr>
<th>Algorithm/Algorithm (Type)</th>
<th>Exhaustive Search</th>
<th>Iterative improvement</th>
<th>Greedy</th>
<th>Divide &amp; Conquer</th>
<th>Single recurrence</th>
<th>Multiple recurrence / DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bellman-Ford’s algorithm (SSSP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary search</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dijkstra’s algorithm (SSSP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floyd-Warshall’s algorithm (APSP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ford-Fulkerson’s algorithm (Max flow)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prim’s algorithm (MST)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Topological sorting</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kruskal’s algorithm (MST)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linear search</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radix sort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quicksort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Write out the time complexity of each algorithm near each line.
2. Algorithmic complexity attacks

In the lecture you saw an example of an application that had to take a lot of untrusted input and store it in a key-value store, which was implemented as a simple hash map. Consequently, the application was vulnerable to an algorithmic complexity attack on the hash map. Come up with (or find in the internet) another potential or actual example of an algorithmic complexity attack (not necessarily involving a hash map).

3. Universal hashing

A student in the lecture proposed the following parameterized hash function\(^1\):

```c
uint8_t hash(uint8_t key, uint32_t val) {
    return key ^ (val >> (32-8));
}
```

Here `key` is the parameter of the hash family and `val` is the value being hashed. Is this function universal or \(\varepsilon\)-almost-universal for some \(\varepsilon < 1\)? Prove your claim.

**Hint:** recall that a hash function family \(\{h_k\}\) is \(\varepsilon\)-almost universal if for any fixed inputs \(x\) and \(y\) and a key \(k\) picked uniformly at random, the probability of a collision is

\[
\Pr[h_k(x) = h_k(y)] \leq \varepsilon.
\]

The hash family is universal if it is \(1/m\)-almost-universal, where \(m\) is the cardinality of the domain of the function (\(2^8\) in our case).

4. Monte-Carlo sorting

Propose an example of a meaningful Monte-Carlo sorting algorithm. What are you trading off for what? When would you use such an approach?

**Hint:** Think about drastically reducing number of necessary comparisons. Think about, say, combining counting sort with Bloom-filters or hashing. Think about the need to sort with respect to a complicated ordering (e.g. graph vertices, etc). Think web-scale.

5. Ternary Huffman code

In the lecture we discussed the Huffman coding algorithm for deriving an optimal binary code (i.e. the code which maps each source concept into a sequence of 0s and 1s). Recall that the algorithm went by recursively joining pairs of least-probable nodes. It turns out that analogous approach (recursively joining \(n\)-ples of least probable nodes) produces optimal code for any destination alphabet.

---

\(^1\)This is plain C. \(\gg\) denotes bitwise shift-right. \(^\wedge\) denotes bitwise XOR. `uint8_t` and `uint32_t` are 8- and 32-bit unsigned integer types (commonly used by including the `stdint.h` header).
Implement ternary Huffman coding algorithm. That is, write a function

```python
function ternary_huffman_code(string input) {
    ...
}
```

which takes a string of letters as input, computes letter occurrence counts, builds a ternary Huffman tree (i.e. a tree in which each non-leaf node has three children rather than two) and finally prints the mapping of each source letter to the corresponding codeword.

For example:

```bash
$ ternary_huffman_code("WORKMORE")
O -> 0
R -> 1
W -> 20
K -> 21
M -> 220
E -> 221
```

**Hint:** Note that if the number of different letters in the source text is even, you need to add one “dummy” letter with occurrence count 0 in order to build the tree.

**Bonus (1pt):** It is possible to implement this function in less than 15 lines of Python (as few as 7 lines are possible, in fact, if you sacrifice some readability). Make your implementation no longer than 15 lines (no matter what language you use) and get 1 bonus point.

6*. Lempel-Ziv-Welch algorithm

Implement the LZW coding and decoding. Have the encoding implemented as:

```python
function encode(string s) {
    ...
    return a list of pairs (int, char)
}
```

and the decoding function as:

```python
function decode(list of pairs (int, char)) {
    ...
    return decoded string
}
```

Don’t worry about the efficiency of your encoding implementation, but do think about the data structures used to implement efficient LZW encoding in real-life applications.