Outline and brief explanation of your task runner &
Explanation of the main paradigm used in your concurrent implementation &
Explanation of how the tasks are scheduled, how new tasks are spawned and how the implementation runs in parallel.

The task runner implementation is as follows: Multi-threaded Depth-First Search (DFS) with work stealing (crossbeam).

Multi-threaded:
In our implementation, threads are utilized as opposed to tokio async tasks due to the fact that the task is doing CPU-bound work. That is, the thread would spend the majority of its time occupying the CPU to get its work done. The number of threads spawned in the task runner is equal to the number of physical CPU cores available in the machine.

Depth-First Search
To ensure that the task runner is running with a reasonable memory consumption for large input sizes, the task runner is implemented in a DFS manner. DFS is used as its space complexity is O(height*max number of children), which is significantly better than Breadth-First Search (BFS) that has a space complexity of O(height^max number of children). In terms of time complexity, DFS is the same as BFS, which is O(total number of tasks).

Work stealing (crossbeam)
The crossbeam crate is used as it provides a lock-free concurrent work stealing deque. The lock-free queue is non-blocking and generally provides higher performance as compared to a queue that uses locks. Additionally, the deque ensures that multiple threads are able to access the data structure in a thread-safe manner and a work-stealing mechanism can be easily implemented into the task runner. When a thread runs out of work, the thread will steal work from the other threads, ensuring maximum concurrency and parallelism where every thread will be executing some work until all tasks are exhausted.

We used crossbeam to implement these data structures:
Injector: a First In First Out (FIFO) queue, where tasks are pushed and stolen from opposite ends. It is shared among threads and contains the initial tasks.

Worker: a Last In First Out (LIFO) queue, in which tasks are pushed and popped from the same end. Each Worker is owned by a single thread and supports only push and pop operations.

Stealer: A stealer can be created from each Worker, that may be shared among threads and can steal tasks from its respective Worker. Tasks are stolen from the end opposite to where they get pushed.

A thread will first try to pop from its local Worker, or else use the Stealer to steal a task from a Worker in another thread, or else steal from the Injector. When the execution of a task is completed, the children's tasks will be pushed onto the local worker's LIFO queue. This ensures that all threads will be occupied with some tasks, when their local Worker runs out of task, ensuring maximum concurrency.

std::sync::Arc and std::sync::Mutex: are used to wrap around count_map, output and injector. These data structures are shared among multiple threads, arc + mutex are used to prevent data races for these data structures.

Summary:
Overall, the injector, worker and stealer data structure work together to ensure efficient, concurrent, and parallel execution of a large number of tasks. The injector contains the initial set of tasks generated by the "generate_initial" method. We then generate workers and its corresponding stealers based on the number of physical CPU cores available in the machine. Each of these workers will then be moved to a newly spawned thread, which is also based on the number of physical CPU cores available in the machine. When a worker thread needs to get a new task, it will first try to pop a task from its own stack (which operates as a LIFO queue). If its local stack is empty, the worker will attempt to steal a task from another thread's worker stack. Specifically, the worker will attempt to steal from the bottom of the stack. However, if there are no tasks available in any of the other worker threads' stacks, the worker will attempt to steal a task from the injector. If the worker manages to get or steal a task, it will then execute the CPU Bound tasks in a DFS manner. When the execution of a task is completed, the children's tasks will be pushed onto the local worker's LIFO queue. Otherwise, if the thread could not get or steal a task, it will complete its execution and await joining.

By implementing this work-stealing algorithm, we can ensure that no threads remain idle and the threads are efficiently used to carry out the CPU-Bound work concurrently and in parallel, making the task runner a concurrent and efficient program to run tasks on.

Bonus task runner (bonus folder)
For the above implementation of task runner, it's space complexity is O(height*max number of children), which will still use a lot of memory for extremely large input sizes (i.e. starting_height = 10,000 and max_children = 10,000). The key observation here is that since we are using DFS, there is no need to generate all the children, we only need to generate one. Hence, we have modified task.rs to lazily generate children, reducing the memory usage significantly. Instead of generating all children task, we generate this struct instead:

```
struct LazyTask {
    pub height: usize,
    pub count: usize,
    pub rng: Arc<Mutex<rand_chacha::ChaCha20Rng>>,
    pub max_children: usize,
}
```

where count represents the number of children tasks left to generate and the other fields are required to generate the child task.

For example, instead of generating 10000 children per parent, we encapsulate the parameters that generate the children and the number of children to be generated into 1 LazyTask object. This means that a child task is generated only during execution, as opposed to generating 10,000 children per parent in advance.

Because we only store 1 LazyTask per height for each thread, the space complexity for each height is O(1). Therefore, the overall space complexity is just O(height), significantly improving the overall memory efficiency.

Implementations attempted:

No.	Implementation	Description	Speed	Memory
1	Tokio asynchronous recursive with box	- Parents will recursively spawn async tokio tasks for their children, leading to an explosion of tokio tasks.	Faster	Huge memory usage: Number of children and tokio tasks increases extremely fast.
2	Tokio asynchronous bfs	- Tasks will be executed in an async bfs manner and children will be appended to shared taskq. - The number of tokio tasks is capped.	Fastest	Huge memory usage: Number of Task increases extremely fast due to the nature of bfs, space complexity of O(height^max number of children)
3	Tokio asynchronous dfs	- Tasks will be executed in an async dfs manner and children will be appended to local stack. - The number of tokio tasks is capped.	Fast	Low memory usage: Due to the nature of dfs, the memory usage is lower than bfs. Dfs space complexity is O(height*max number of children)
4	Multi-threaded dfs	same as above (3.), just that its using threads instead of tokio async	Fast	same as above (3.)
5	**Multi-threaded dfs with work stealing (Current implementation)**	- Tasks will be executed in a multi-threaded dfs manner and children will be appended to local stack. - The number of threads is equal to the number of physical cpu cores. - Work stealing is implemented using	Fastest	Low memory usage: Due to the nature of dfs, the space complexity is O(height*max number of children). The memory usage is

		crossbeam::deque		also slightly lower than 3. due to work stealing.

6.	**Multi-threaded dfs with work stealing and lazy generation of children (Bonus implementation)**	Similar to above (5.), but children are now generated in a lazy manner. Instead of generating all children task, we generate this struct instead:
		`struct LazyTask {`
		`    pub height: usize,`
		`    pub count: usize,`
		`    pub rng: Arc<Mutex<rand_chacha::ChaCha20Rng>>,`
		`    pub max_children: usize,`
		`}`
		where count represents the number of children left to generate and the other fields are required to generate the child task.
			Fastest	Extremely low memory usage:
				
				e.g. instead of generating 10k children per parent, we generate only 1 LazyTask. A child task is only generated before execution.
				
				Because we only store 1 LazyTask per height for each thread, the space complexity for each height is O(1). Therefore, the overall space complexity is just O(height).