Problem 1

The two major concerns of any software project are how much it will cost and how long it will take. The budget ensures the project stays within financial limits while time focuses on delivering the software within the scheduled deadlines. Budget is more important between the two, as keeping costs under control is critical. The idea of complete functionality combines these two ideas. It must be balanced with time constraints, requiring teams to prioritize core features and defer less critical ones to future iterations.

Problem 2

The five phases in Agile software development in each iteration are requirements analysis, designing, coding, testing, and maintenance. Agile emphasizes continuous adaptation. You could potentially skip some smaller substeps in the phases, but I don't think it's a good idea to do that. Ultimately, it wouldn't save time overall because poor planning can spiral other things out of control. Thus, repeating all five phases ensures flexibility and responsiveness to evolving requirements, a core principle of Agile. Saving time by skipping phases could undermine the adaptability and quality Agile promotes.

Problem 3

The Waterfall method's main phases are requirements gathering, design, implementation (coding), testing, deployment, and maintenance. These phases are sequential, meaning each must be completed before the next begins. Waterfall includes more upfront phases like detailed requirements gathering and comprehensive design, which Agile streamlines or repeats in each iteration. While these phases are essential in Waterfall for thorough planning, they can become rigid if the project needs change. These extra Waterfall phases could be necessary for the medical or aerospace industries that need highly regulated software. For example, when compliance with strict standards requires detailed documentation or design approvals upfront, an Agile team might temporarily incorporate more formal requirements gathering to meet regulatory needs while maintaining flexibility in development.

Problem 4

What is a "user story"?: A user story is a simple description of a software feature from the end-user's perspective. It captures what the user needs and why, without getting into technical details.
What is "blueskying"?: Blueskying refers to brainstorming sessions where you and the customer discuss what they would like to achieve, without limitations.
What are four things that user stories SHOULD do?:

- Define the **WHAT** of your project clearly.
- Be customer-oriented, focusing on the user's needs.

- Be manageable and allow for estimates to be provided.
- Break down into smaller tasks that can be tracked .

What are three things that user stories SHOULD NOT do?:

- They should not be too large, as that can make them difficult to estimate.
- They should not dictate implementation details to the development team.
- They should not be vague or incomplete, as this leads to misunderstandings .

Does the Waterfall method have user stories?: No, the Waterfall method typically does not include user stories. Waterfall is a sequential design process where all requirements are gathered upfront and formalized in documentation.

Problem 5

All assumptions are bad, and no assumption is a good assumption. - I disagree with this statement, since some assumptions can be made in good faith, as long as there is ample reasoning to support making that assumption as opposed to asking for clarification. For example, you may assume that your client wants you to optimize your code, even if it runs, because a faster runtime isn't a detriment to the vast majority of projects.

A big user story estimate is a bad user story estimate. - Generally, I agree with this statement, since increasing the scope of a project makes it more difficult to prioritize, track, estimate time scales, and generally slows down the production process.

Problem 6

You can dress me up as a use case for a formal occasion: User Story
The more of me there are, the clearer things become: User Story
I help you capture EVERYTHING: Blueskying, Observation
I help you get more from the customer: Role-playing, Observation
In court, I'd be admissible as firsthand evidence: Observation
Some people say I'm arrogant, but really I'm just about confidence: Estimate
Everyone's involved when it comes to me: Blueskying

Problem 7

A "better than best-case estimate" is a scenario in which the actual time taken or resources consumed to complete a task or project is even less than what was considered the most optimistic forecast. Normally, a best-case estimate represents the most ideal and efficient outcome, assuming no interruptions or issues, but a "better than best-case" estimate exceeds these expectations in a positive, often unexpected way.

Problem 8

In our opinion, the best time to tell a customer that we cannot meet a delivery schedule is as soon as possible, since putting it off simply compounds expectations that the schedule is on track. Additionally, telling the customer early gives them time to help create a plan to readjust according to the needs of the team, or, if not acceptable, then drop the project and end a waste of their resources. It would be a difficult conversation to have, but it is made less difficult the earlier that you voice concerns, as honesty (as well as presenting alternative plans) are received a lot better than simply failing to deliver.

Problem 9

Branching in our software configuration would be great to implement, because there are so many moving parts to our project; How the environment system (rain, sun, etc.) is entirely different than the inventory system or UI that other developers would concurrently be implementing, and so minimizing the amount of conflicts when creating so many different systems at once would be incredibly important.

Problem 10

The most prominent “build tool” that we have is Unity, since all of our game will be ultimately compiled, tested, and ran through it. It is great for computer scientists to use and manipulate, since most of the specific ways that the system runs can be custom configured easily by using code, but as a result is a lot more needy when it comes to actually creating a new system, as opposed to making an instance of an already made one, since it requires a lot of specific details in order to be compiled in an acceptable form.