The Hazards of Hack and Slash in Programming

By Arjun Rasodha (LS216) – January 12, 2022

It's a beautiful summer day and you and some friends decide to go on a road trip to the nearest beach. You and your friend decide to take separate cars. Your friend says, "Since neither of us have been to this beach before, we should probably put this in the GPS so we know which way to go". You reply, "No, I know how to get there. It's not too complicated. I'll be alright without the GPS". You and your friend both leave your houses at the same time. Your friend has looked at the GPS, made some notes on which route is the best and any potential obstacles that may arise such as, construction, heavy traffic or railroad crossings. Eventually on the map there is a point where there is construction and as a result there are two alternative routes to choose from. You say, “Well, I’ll just take the first route, there shouldn’t be much difference and I’ll get to the destination around the same time. I know what I’m doing”.

On the other hand, your friend is already expecting some construction in that area ahead of time and determines the best alternative route based on their current location, the traffic on the two routes and the difference in distance. Needless to say, your friend arrives a lot sooner to the beach, consumes less fuel and was less stressed out during the drive as they created a plan and were prepared.

Benjamin Franklin has a famous quote, “If you fail to plan, you are planning to fail”. You may not have failed to arrive at the destination but the likeliness of that being the result is higher and the obstacles are tougher without a well thought out plan.

This relates to writing code as a carefully thought out plan also know as an algorithm is essential to the chances of you writing the code successfully. An algorithm is a well-defined step-by-step procedure of how you are going to solve a problem. Algorithms are precise instructions that tell the programmer how to write the code. They are usually written in pseudocode with some implementation level details. They describe the logic for how the input will be converted into the correct output. In terms of the first example, an algorithm is specific directions for how you will be getting to your desired destination with any challenges accounted for.

Without an algorithm we would be diving straight into the coding part of solving the problem. This is a very dangerous approach. Without a well thought out plan it is very likely that you will run into many different issues and not have a clear idea of how you want to solve the problem. In addition, within more challenging problems there are usually many different test cases to consider when writing the logic. Without an algorithm it is extremely difficult to satisfy them all and the coding process can become far more difficult, frustrating and error prone.

At this point we know that we must have a system for how to break down a problem description, identify the explicit and implicit requirements and develop an algorithm that we are confident in. This gives us the best chance of understanding the problem well enough to create a plan that will produce the correct result. The algorithm should be written in a way that gives thorough detail to the programmer on how to write the code. The goal for the algorithm is to be accurate enough so that the programmer is just translating it into the correct syntax when writing the code. However, even though the logic has been thoroughly thought out, trade offs have been considered and obstacles have been accounted for during the algorithm phase, there are still possibilities of considerations being overlooked.

When converting your algorithm into code, everything may be going well. Your code is nice and clean, you are feeling confident and you think that you have become really good at this process. You run your first test case and it passes easily. You run your second test case and there is a slight issue with one of the requirements. You run your third case and it does not pass at all. Now you start to experience feelings of stress. Anxiety starts to creep in, your confidence slowly starts to decrease and your mind becomes foggy, making it difficult for you to think clearly about the next step. You look at the timer and you are past the halfway mark for the interview. You say to yourself, “I don’t have time to adjust the algorithm, I know how to fix this, I just need to change a line here or add an extra condition here or change this operator”. You make the change quickly, one test case is fixed, but the one that was working before is not working anymore. This is starting to get worse rather than better. What should you do now?

This process here is called “Hacking and Slashing” and occurs when you are trying to solve an issue in your program through the use of trial and error. You are not looking at the code at a high enough level of abstraction to see the larger picture of what your code is doing. You begin to experience tunnel vision. You think that going back to the algorithm to write pseudocode at this point is going to waste time so instead you decide to go straight to making tweaks to the code. At this point you are no longer being intentional with your code and are straying away from your intentional problem-solving approach. This is a risky mode to get into where it is not very likely for a successful outcome to be the result, especially in more difficult coding problems. The only thing worse than this is not having an algorithm at all and jumping straight into coding from the very beginning.

Some of the main dangers of being in hack and slash mode are wasting precious interview time falling down the wrong path. This can occur because you have not thought about the pros and cons of using that approach in the solution. Just because it works for one test case does not mean that it will work for all of them. Another downside to being in hack and slash mode is that it is highly unorganized and easy to forget components that you want to include in your solution. Whereas, when you have it written down in a clear algorithm, you will not forget any parts of it. Another drawback of hack and slash is using more energy as you will likely run into more problems due to thinking about the solution in a very low-level way. Thinking about the problem in a higher level of abstraction allows you to think about the alternative approaches without committing to any implementation level details. To add to that, when you are in hack and slash mode, you come off as less knowledgeable to the interviewer since you are not able to articulate your thoughts and write code simultaneously. Whereas, writing an algorithm allows for the interviewee to communicate the technical aspects of the solution and different approaches easier.

Referring back to the earlier example of you and your friend going to the beach. Without the GPS, you decided to guess which alternative route you should take when encountering the construction. This guess would be using a hack and slash approach as it is not calculated, similar to a guess and definitely along the lines of a trial and error approach. On the other hand, your friend decided to do some research on the map, compare the distance of each route and look for the route with the least traffic and then made their decision. Your friend created a clear plan, with steps and considered edge cases and other challenges that needed to be thought through prior to the trip which is similar to creating an algorithm.

Back to a coding interview scenario, it is far more efficient and effective to go back to the algorithm and make the adjustments there. This is more superior than using the hack and slash approach, to work your way out of a problem that your algorithm overlooked. You can read your algorithm over from the first line and follow it down to the area where you are finding an issue as to why your test case is not passing. At that point you can try to debug your code and use some `console log` statements to see what is going on right before the point where you are finding your problem. After this, you can determine different ways to overcome this obstacle while considering all of the different test cases and their requirements. Once you have adjusted your algorithm and feel confident in your new way of going about the problem, you can go ahead and adjust the code according to it.

This approach is more effective and efficient than the “hack and slash” approach as it will save you time and be more likely to work since you have thought about the obstacle from a higher level of abstraction and considered the other trade offs of the change in relation to the various test cases.

Let’s take a look at these two approaches with a JavaScript coding problem and see how they both work:

Question (Title Case): A string is considered to be in title case if each word in the string is either (a) capitalized (that is, only the first letter of each word is in uppercase) or (b) considered to be an exception and put entirely into lowercase unless it is the first word, which is always capitalized.

Write a function that will convert a string into title case, given a list of exceptions (minor words). The list of minor words will be given as a string with each word separated by a space. Your function should ignore the case of the minor words string — it should behave in the same way even if the case of the minor word string is changed.

First Argument: the original string to be converted. Can be an empty string. Return input string if so.

Second Argument: space-delimited list of minor words that must always be lowercase except for the first word in the string.

Test Cases

```
titleCase('a clash of KINGS', 'a an the of'); // should return: 'A Clash of Kings'
titleCase('THE WIND IN THE WILLOWS', 'The In'); // should return: 'The Wind in the Willows'
titleCase('', 'The In'); // should return: ''
```

This question seems quite straight-forward. We are given two strings. Take the first string and return a new string with every word in the first string, in the same order and in title case. Title case means the first letter of each word is in uppercase and the rest of the word is in lowercase unless the word is included in the second string. If the word is included in the second string, it should remain lowercase in the output. However, if the word is in the second string and is the first word in the first input string, the first letter should be uppercase while the rest of the letters of that word is lowercased.

If we create an algorithm for this question it may look something like this:

Algorithm

```
- if the first input string is an empty string, return the first input
string
-  create `wordsArr` and initialize it to the first input string, set to
lowercase and split by a single space
- create `minorWordsArr` and initialize it to the second input string, set
to lowercase and split by a single space
- map over `wordsArr` and on each round of iteration:
  - if the current `index` is `0`
   - Return the word with the first letter capitalized and the rest of the
word set to lowercase
  - otherwise if the word is included in `minorWordsArr`
     - return the word
   - otherwise
     - return the word with the first letter set to uppercase and the rest
of the word set to lowercase

- return the mapped array joined with a single space
```

The algorithm is looking quite good. Let’s go ahead and write it in code now:

```
function titleCase(title, minorWords) {
  if (title === '') return title;
  let wordsArr = title.toLowerCase().split(' ');
  let minorWordsArr = minorWords.toLowerCase().split(' ');

  return wordsArr.map((word, idx) => {
    if (idx === 0) {
      return word[0].toUpperCase() + word.slice(1);
    } else if (minorWordsArr.includes(word)) {
      return word;
    } else {
      return word[0].toUpperCase() + word.slice(1);
    }
 }).join(' ');
}
```

Okay, everything looks good. Let's run our code and see how it works.

First test case has passed. Great! Second test case is working. Awesome! Third test case is working. This is wonderful!

However, the interviewer says that you have missed an edge case to check for. Will your code work with this test case:

```
titleCase(' ', 'The In'); // should return: ‘ '
```

Your confidence is high and you say, "yes, lets run the code. My algorithm and code is great". You run the code and this error is raised:

```
  return word[0].toUpperCase() + word.slice(1);
           ^
TypeError: Cannot read property 'toUpperCase' of undefined
```

It looks like we missed something in our algorithm. We are handling empty strings but not strings with white-spaces. According to this question an empty string and a white-space string should be handled the same way. But not to worry, we can just go ahead and add one line of code to fix this problem.

```
 if (title === '' || title === ' ') return title;
```

This results in the new test case passing. But the interviewer says, "we should be able to pass an empty string with any amount of space as the first argument into this function and it should still work".

You say, "okay, no problem. I just need to modify this line of code once more and it will work".

```
if (title.split('').length === 0) return title;
```

Now you run your code and all test cases work but the last one is raising an error. You are confused, scratching your head and were sure that an empty string, split into an array of characters, should have resulted in an array with the length of `0`. You try to modify the line a few more times and are running the code to check each time in panic. Hopefully you have noticed by now that you are in hack and slash mode now.

A way to have avoided all of this wasted time and effort is to steer clear from making any changes to the code without first referring to your algorithm. As soon as the test case failed you should have went back to the algorithm and followed it from step 1 to the point where the error has been raised. Then we could think about the different solutions at a higher level of abstraction and see how each solution would impact the rest of the test cases. Then we should add the solution into our written algorithm and then go through the test cases to make sure that we have considered all edge cases. Once a level of satisfaction with the algorithm adjustments has been achieved we could then go ahead and make the changes to the code.

When going back to our algorithm we would have noticed that splitting a white-space string into an array of characters with a single space would result in an array of white-spaces. We must filter those white-spaces out of the array and then check its length since white-space characters are not valid word characters. This fix works for empty strings, white-space strings and all of our other test cases.

The updated algorithm may look like so:

Algorithm

```
- if the first input string, split into an array of characters, filtered to
remove any single space strings, array length === 0, return the first input
string (CHANGE MADE ^)
-  create `wordsArr` and initialize it to the first input string, set to
lowercase and split by a single space
- create `minorWordsArr` and initialize it to the second input string, set
to lowercase and split by a single space
- map over `wordsArr` and on each round of iteration:
  - if the current `index` is `0`
   - Return the word with the first letter capitalized and the rest of the
word set to lowercase
  - otherwise if the word is included in `minorWordsArr`
     - return the word
   - otherwise
     - return the word with the first letter set to uppercase and the rest
of the word set to lowercase

- return the mapped array joined with a single space
```

Then we can go ahead and verify our change by manually walking through the test cases. Once we are happy with this we can go ahead and adjust our code according to the algorithm. It would look like so:

```
function titleCase(title, minorWords) {
  if (title.split('').filter.(char => char !== ' ').length === 0) return
title;
// ^ change made
  let wordsArr = title.toLowerCase().split(' ');
  let minorWordsArr = minorWords.toLowerCase().split(' ');

  return wordsArr.map((word, idx) => {
    if (idx === 0) {
      return word[0].toUpperCase() + word.slice(1);
    } else if (minorWordsArr.includes(word)) {
      return word;
    } else {
      return word[0].toUpperCase() + word.slice(1);
    }
 }).join(' ');
}
```

Now regardless of how many spaces there are in the empty string provided as the first argument, we can handle those test cases without effecting the rest of the code that was working for the other test cases.

Now we can attempt running a test case like so:

```
titleCase('      ', 'The In'); // should return: '      '
```

And our code will work just as we expect it too.

In a nutshell, you should always plan before coding. This means break down the problem, make sure you understand all the requirements, develop test cases to verify your understanding and create a thorough step-by-step algorithm that outlines how you plan to convert your input to the desired output. If you find that you have ran into an issue working on a coding exercise, take a moment to collect yourself and remain calm and revisit your algorithm right away. Walk through the test case that is causing a problem and try to identify the part of the algorithm causing the issue. Adjust your algorithm and check it with the test cases manually. Once you are confident at this stage, go ahead and make the changes to your code. It may not seem more efficient and effective to go back to the algorithm in the middle of coding but overtime you will surely realize that it is.

Hopefully, this was a helpful and encouraging article that demonstrated why to avoid the hack and slash method when working on code problems and how using a well planned algorithm and referring back to it in times of uncertainty is a far better way to go. Hack and slash is very tempting but a dangerous gamble that does more harm than good.