The Art of Giving Up and B.S.

Winston Churchill once said, “Never, never, never, never give up.” Though different sources of the quote differ on the number of times ‘never’ appears in the quote, we can confidently ascertain Churchill’s intentions nonetheless. But was he right? In the context of World War II, certainly a statement of this type from a leader makes enormous sense. In some sense, everything was at stake and the consequences of giving up were incredibly dire.

In terms of everyday situations and solving problems however, giving up is a necessary part of life. Imagine what would have happened if I never gave up on my dream to be an NFL quarterback? At 5’ 7”, with an arm that can barely heave a football 40 yards, no amount of practice or training could have earned me a contract playing quarterback for even the Miami Dolphins, who have a rather hapless quarterback situation. This certainly begs the question, at what point do we decide to give up on a certain goal or path and decide to pursue another one?

I was reminded of the fact that knowing when to give up is indeed a tricky and difficult art this past weekend while solving two programming contest problems. I had solved both problems incorrectly in a timed contest setting and had decided that I wanted to spend my spare time during the weekend fixing my solutions, without the timed constraint of a contest.

Ironically, in one problem I attempted a direct problem solving approach that wasn’t working which I finally fixed with a binary search (B.S.), while in the other problem I changed to a direct approach when every attempt to tweak my binary search had failed. Before I solved either of the two problems, I had to decide to give up on my previous technique. As was illustrated in this case, binary search works well for some problems, and not others.

Most of us are first introduced to binary search as children through the guessing game. Someone picks a secret number in between 1 and 100 and asks the other person to guess it. After the guess, the player is provided with feedback telling her whether her guess was too big or too small. In this manner, the game continues until the player guesses the correct number. After several iterations of playing the game, most children figure out that the ideal technique is to guess in the middle of the possible range of numbers. Thus, we first guess 50, followed by either a guess of 25 or 75, depending on the response to our first guess. In its basic form, a binary search allows one to quickly find whether or not an item appears in a sorted list.

While this doesn’t sound terribly useful, the process of making a guess and removing half of the possible candidate answers after each guess is a very powerful problem solving technique. It makes problems that would otherwise be very complicated to solve, relatively easy to solve. The trick, of course, is figuring out how to configure your search, and how to appropriately define the search space and cut it in half each time.

The first time I used a binary search to solve a problem at work was when I was 18 years old, working for a company that made crystals that oscillate. I had access to a function that, given the amount of time a crystal was etched, calculated its output frequency. But, my job was the opposite. I was given the end frequency, and needed to calculate how much time to etch the crystal so that it arrived at that frequency. In mathematics, my job was to find the inverse of the function I was given. (An inverse is essentially a function that “undoes” the transformation that the original function performs. For example, if a function doubles a number, its inverse would be dividing the number by 2. Namely, when I take 5 and double it, I obtain 10. Given 10, to get back the original number, I need to divide it by 2.)

I had just finished my freshman year at MIT and felt pretty confident that I could invert the function given to me. I started writing out some math on paper and tried all sorts of tricks I learned through my years of math contests. None of them worked. I probably spent about three hours before having to go to lunch. After lunch, I came back and tried some more, with no success. I was feeling pretty down. This was just my second week on the job and my first week was largely met with failure to read and understand legacy Fortran code, and not being able to invert a function, something that I was supposed to be good at, was very troubling. I don’t know why, but I decided to give up. I just decided that inverting the function was too hard, and asked myself if I could use a different technique to approximate the answer. After all, I realized, I didn’t need to know the exact time it took to achieve the desired frequency, just the amount of time within 1 second. The amount of time we could dip a crystal to be etched was always a whole number of seconds, so there was no point in making any calculations to greater accuracy. Besides a host of other real-life factors made the real-life performances of these etches vary by several seconds from what the equations predicted. In about five minutes I realized that the function I was plugging into was an increasing function. Namely, if you etch a crystal for more time, the frequency of its oscillation would increase as well. Thus, if I made a guess as to how much time to etch the crystal, I could look at the corresponding frequency and know immediately if my guess was too small or too big – BINARY SEARCH!!! It took me about 10 minutes to code the whole thing up, and amazingly it worked! Problem solved. Naturally, I was a bit dismayed that I had “wasted” the whole morning and part of the afternoon, but felt lucky that I did eventually choose to give up on that approach, which led me to discover one that worked.

The first problem I tackled this weekend was from the Google Code Jam, Round 1B. It involved judging on a reality show and determining what percentage of votes each contestant needed to avoid being kicked off. Almost immediately, I came to the conclusion that there was a fixed cut-off of points that everyone needed. From there, I figured out, for each contestant, how many points they needed to get to that cut-off, and converted that to a percentage. When I submitted my solution in the contest for the “small data sample”, I passed! I was reasonably “sure”, that my approach was correct. However, after the contest was over, I found out that my program failed the large sample data. As I started working on this problem again this past Friday night, I kept to my assumption that I just had a slightly wrong formula for calculating my fixed cut-off for not getting eliminated. I tweaked this formula, checked a few cases, and then decided to try solving the large data set posted online after the contest. Yet, my solution failed. I tried a second modification, which still didn’t work. It was nearing 1am and I was getting tired, but I didn’t want to go to bed without solving the problem. After a more careful analysis of my program’s output, I realized I was outputting negative percentages in a few cases, which is non-sensical. I put in a quick fix which simply changed any negative percentages to 0. Yet, even with this fix, I still failed the large test case data. Finally, I stopped. I asked myself if my initial assumption, that I knew the number of cut-off points, was correct. After saying, “what if I am wrong, what if I need to first figure out this cut-off without a direct formula?”, I drew a picture of a bunch of vertical bars and a horizontal line representing the cut-off point value needed to avoid elimination. From this picture it was clear to me that I DIDN'T know where to draw the line. BUT, if I tried to draw the line, it would be very easy for me to verify if my line was too low or too high – BINARY SEARCH! It took no more than 10 minutes to add in this binary search to find that correct cut-off score. Once I knew that, the final calculation was very similar to what I had and my solution finally worked. At 1:30am, after finally solving the problem, I headed to bed.

The second problem I started working on Saturday was from the 2011 Southeast Regional Programming Tournament. Some of our students correctly solved this problem during the contest itself. Even my own pickup team, solved the problem correctly during the contest. Though all three of us helped, Stephen Fulwider actually wrote nearly all the code. As part of his solution, he used a binary search. When we submitted it during the contest, we got the problem incorrect at first. Stephen tweaked his binary search and we got the problem correct. The following week, I tried to write my own solution to the problem. Much to my chagrin, even after 5 submissions, no matter how I tweaked my binary search, I got a wrong answer. I was determined to figure out how/why I couldn't get my tweak to work back in November. When I had free time on Saturday, I tried several different tweaks to my binary search. No matter what I tried, I'd either get wrong answers, or my search would take too long. I tried to modify how long my search ran for specific cases, but that approach ran into disaster. Finally, I decided to give up. I decided to try to find a direct formula to calculate what Stephen had used a binary search to guess. Essentially, I was doing the opposite of what I did back when I was a freshman calculating etch times. I carefully wrote out my equations, and got a solution I needed for a time variable. Unfortunately, I got wildly incorrect answers. I had to leave to do other things and left my code in this state. When I returned to it, I realized that my binary search was useful. It at least got the correct answer, if it ran long enough. I decided to use it to find mistakes in my other approach. Slowly, as I found cases for which my new approach wasn't working, I'd find a sign I had incorrectly switched in my math, or another minor issue. After fixing one case, I'd move onto another, only to discover more problems. The continued for about three iterations, before I couldn't find an individual case for which my new approach didn't work. Finally, I decided to plug in my new approach to my old solution, getting rid of the binary search. It solved the problem correctly in about 15 seconds, where my other solution had been taking over 2 minutes. In this case, avoiding guessing and doing the extra work to find a direct answer was the key for me solving the problem.

Unfortunately, even with my weekend experience, I don't think I am any closer to defining when one ought to give up on a certain approach to a problem. I can say with certainty, however, that giving up is sometime necessary to make progress and that occasionally taking a step back and forcing yourself to remove tacit assumptions can help solve some difficult problems. Last, but not least, B.S. only helps you solve problems SOME of the time!