Task 6.3 The issue that really makes the above algorithm run in at least $O(2^n)$ is the redundant calls to `is_word()` for substrings that have already been checked before. We can instead create a table that will grow to hold each word and its result as is checked. Then each time we call `is_word()` again, we check that table to see if we already checked it, an operation that can reduce the runtime of `is_word()` to constant.

Then the task is simply to rearrange the for loops to check each word (in $O(n^2)$ time) and by then we already know which combinations would possibly result in a real word. Further optimizations will change how we look through the search space, along with a table of word frequencies according to Zipf's Law to help resolve conflicts or ambiguities.

Task 9.1 Assuming a perfect implementation, with 4 bytes per id, we have 2^{32} possible IDs. Storing each ID will take 4 bytes (to hold the value of each ID), we need $2^{32} * 4$ bytes, which amounts to precisely 16GB. My laptop has 16GB physical memory, so I would crash as soon as I tried to run an operating system underneath this program. However, my Ubuntu dual boot has a 8 gig swap partition, so running it there would work, so long as I closed Chrome first.

Task 9.2 We still have 2^{32} IDs, but now need 1 bit per, rather than 32 as before. This decreases our required memory to *only* 512MB. This would easily run on my laptop, my cell phone, and even the tiny VM I run my website off of.

Task 9.3 Initially, when our load factor is quite low, we'll have constant time allocate and release since we won't have any cache collisions. Indeed, the first M calls to allocate will be ideal, assuming a perfect hash. But after that point, we'll start to run proportional to the load factor λ. Assuming linear probing, we'll need more and more compares to find an unused ID, to eventually take an average of $n/2$ compares for the final alloc.

A hash function with better spread will give better results with linear probing assuming that similar IDs are allocated and then released in close proximity. However if release calls are taken as random, then any hash function will suffice as long as it hits every entry in the table in equal proportion.

Task 9.4 My short script suggests that will a namespace of 2^{32} IDs and only 5 left unused, it will take an average of $7.095 \cdot 10^8$ compares to fill just 1 of those slots using linear probing.

9.5 It's not too hard to come up with a variety of implementations that will have constant runtime for both methods that will simply grow in memory as IDs are freed.

If there's always a few more allocs than frees, we can keep a counter of the lowest ID not yet freed and a linked list IDs that have been freed. Free adds to either end of the list, and alloc will return the head of the list if nonempty, or return counter and then increment counter. This will fall apart apart when many IDs are freed and not many new ones are allocated. Methods still take constant time, but we'll consume more and more memory in the degenerate cases. This pessimal case doesn't sound too impossible however, so let's keep working.

My real solution keeps a linked list of ranges of free values (or equivalently, ranges of allocated IDs). We start with the range $[0, M]$ and when alloc is called, return the lower bound of the range. When IDs are freed, it takes proportional to the number of ranges to either adjust the appropriate range or insert a new one into the list. Once the ranges are suitably complex, the alloc function will prioritize removing ranges of length 1, which will decrease the search time for future calls. In the worst cases, this implementation is $O(n)$ in alloc, free, and memory usage, but those cases are very hard to find. In many cases, inclduing when very many or very few IDs are allocated, each method will take near-constant time and memory. If my description here was unclear, I'll include a barebones implementation in my submission.