1 Tutorial problems

Use multiple tracks, tapes, or other variations on the Turing theme to find machines that accept the following languages (or compute certain functions). Again, worry more about the high level description, and understanding how, while these variations may improve efficiency, their computations could all be accomplished by a standard TM

1. *Compute the n^{th} Fibonacci number f_n (i.e. on input a^n, arrange that a^{f_n} is written on a working tape). Take $f_0 = f_1 = 1$ and for $n > 1$, $f_n = f_{n-1} + f_{n-2}$.*

   Here's a multi-tape approach:

   - Use three working tapes in addition to the main tape (call them T0, T1, and T2).
   - Initialise T0 and tape T1 to have a single a, and T2to be empty.
   - Until you reach a blank move one step right on the input tape and then:
     - Copy the complete contents of T0 to T2
     - Append the complete contents of T1 to T2
     - Overwrite T0 with the contents of T1 (there will always be more a's)
     - Overwrite T1 with the complete contents of T2.

     When you reach the end of the input, T0 will contain a^{f_n}. A certain amount of overwriting can be avoided by using state to cycle through the tapes, i.e., concatenate T0 and T1 to T2, then T1 and T2 to T0 then T2 and T0 to T1 etc. but then you also have to remember which tape has the final answer.

2. *Convert binary numbers to unary. That is, on input $w \in \{0,1\}^*$ arrange output on a working tape of a^n where n is the value of w interpreted as a binary number.*

   Start with two blank working tapes. Move the read-write head on the main tape rightwards. Each time you encounter a 1, append the material from the second working tape to the first, append an a on the first tape, and then copy the first tape onto the second tape. Each time you read a 0 do the same except don't append an a.

3. *Convert unary numbers to binary ones.*

   This is of course easiest to do if you work from least significant digit to most significant digit from left to right. So assume we have an output tape. On the input tape, containing a^n proceed rightwards changing every second a to A (do this to the first a of each pair). When you reach the end, if there was no unpaired a (you'll know this by the state), add a 0 to the output tape (the total number of a's was even). Otherwise, add a 1 (notice you've changed the final a to A). The number of a's remaining is $n/2$ (rounded down). So now repeat (possibly doing it from right to left rather than going all the way back to the beginning). Stop when you find no more a's. If you must have things in "standard" order, now reverse the contents of the output tape (i.e., use the output tape as a working tape and write the reverse onto the real output tape).

4. *Accept the language $L = \{a^p \mid p \text{ is prime}\}$.*

   - *Think about the standard loop version "for each $2 \leq i < p$ check whether the remainder when p divided by i is 0. If so, reject. If all these tests succeed, accept.*

     Use two working tapes. Initialise one (T0) to be the same as the input, and the other (T1) to contain aa. Now move forwards across T0 and T1 simultaneously. At some point you'll reach a blank on one or both tapes:

     – (Blank on T0, not on T1) The number on T1 was not a factor. Add an a to T1, move both heads back to the left and repeat.
     – (Blank on T1, not on T0) Move head on T1 back to beginning (or just past first a – be careful of "off by one" errors).
     – (Blank on both) Check to see if T0 and T1 contain the same number of a's. If so, announce prime, if not, announce composite.

   - *There is an improvement to this method where you only look at $i \leq \sqrt{p}$ since if a number is composite it has a proper factor less than or equal to its square root – how might that be implemented?*

     Use a third tape where each time you reach "blank on T1 not on T0" you add a marker. When you hit a "blank on T0 not on T1" situation (recognising that your current trial is not a factor) check to see whether there are more marks on the third tape than on T1. If so, erase it and proceed. If not, halt and reject.

Note, the only time we literally have to check $\sqrt{n}$ as a non-trivial factor is when $n = p^2$, where p is a prime. In the $p-1$ round, we'll put $p+1$ marks on the third tape since we'd be testing whether $p-1$ is a factor and $p^2 = (p-1)(p+1)+1$.

- *Does using non-determinism seem to help for this problem? What about for the complement of L, i.e. the set of strings a^n where n is composite?*

  There's no obvious method for using non-determinism in a primality test. Here's a highly non-obvious one (the Lucas primality test and see also the Pratt primality certificate) to test if n is prime:

  - Check that n is odd if it's greater than 2!
  - Ask the genie for the prime factorisation of $n-1$, and an integer a.
  - Check that what's provided is the prime factorisation of $n-1$ (by checking – using this test, that the 'primes' really are prime and that the product is correct) – this seems weird but the sum of the bit-lengths of the primes occurring will be at most the bit-length of n minus 1 (because 2 will definitely be a factor of $n-1$) so this recursion will in practice be quite fast.
  - Check that a^{n-1} has a remainder of 1 when divided by n (computing remainders of exponentials can be done quickly by working with remainders throughout and using repeated squaring).
  - Check that for each prime factor q of $n-1$ (from the genie's list) that $a^{(n-1)/q}$ does not leave a remainder of 1 on division by n.

  If you can do all that, then n is prime!

  On the other hand using non-determinism for the complementary language is trivial – just ask the genie for a proper factor of n.