Pattern Matching

- matching substrings
  - Not just ctrl-F in Word
  - base pairs in DNA
    - numerals in a real number expansion
- find pattern, $P$
- inside of text, $T$
- composed of alphabet, $\Sigma$
  - could be all of Unicode, which is large!
- Possibly will never write from scratch, but should know so understand performance of libraries and what your options are.
Brute Force

• Start at beginning of both P and T
  • compare current char of P with current char of T
    • match? compare until out of characters in P or mismatch
    • mismatch? move forward one char in T, start over in P
    • matched all of P? we have a match!

• $O(\text{len}(P) \times \text{len}(T)) = O(n*m)$
• example from webnotes
• find_brute.py
• alt_brute.py
Simplified Boyer-Moore (1977)

• “Hey, if there’s a mismatch, why not jump past that point?”

• *Character jump heuristic*
  • on mismatch, jump P as far forward in T as possible

• *Looking glass heuristic*
  • compare from end of P, more likely to have len(P) jumps instead of 1, 2, ... char jumps

• example from webnotes
Boyer-Moore, cont’d

• Preprocess P to get jump table
• `find_boyer_moore.py`
  • makes jump table a map based on actual letters in P
• Array jump table: $O(n \times m + |\Sigma|)$
• Map jump table: $O(n \times m)$
• ... did that get us anything? Maybe, depends upon if the data is well-suited for the heuristics (English prose is.)
• Non-simplified Boyer-Moore: $O(n + m + |\Sigma|)$
• **GNU grep uses Boyer-Moore**

- Make a *failure table* which tells exactly how far to skip ahead for any mismatch
- Can start later in P if jump can realign against prefix
- Example from webnotes
- $O(n + m)$
  - Theoretical best pattern matching time
  - Worst case, no match at all, still have to look at every char in P and every char in T at least once.
- `fail_kmp.py`
Dynamic Programming

• ... and now for something completely different
• bottom up construction of answer
• sub-answers are easy to compute
• sub-answers overlap so they can be reused
Fibonacci with Dynamic Programming

- from webnotes
- Basic recursion
  - grows ~ $1.6^{**}n$ in space
- *memoization*
  - cache values for reuse
  - $O(n)$ complexity, storage and stack space
- But wait, we can skip recursion and just write to table
- But wait again, we can skip the table and pull ourselves up by our own bootstraps! That is dynamic programming.