Developing Algorithms

Strategies –

- realize your problem is another well-known problem in disguise
  - it is searching or sorting
  - there's a data structure for that
  - it is a graph problem
- develop a new algorithm
  - divide-and-conquer
  - iterative
  - series of choices – greedy, recursive backtracking, dynamic programming

Searching

- searching for a key in a collection
- worth considering distinct from dictionaries in situations where the structures are static (no insertion or deletion)
- basic approaches
  - sequential search
  - binary search

Searching

- sequential search typically fastest for small collections (≤ 20 elements)
- sequential search is simpler to implement
  - 17 years between invention of binary search and the first correct implementation published
- are some items accessed more often than others? – put near the beginning of the search
  - easier to do for sequential search
  - for BST build *optimal BST*
- might access frequencies change over time? – *self-organizing lists* or trees (splay trees)
  - can extend the useful range of sequential search up to 100 elements
- is the key close by? – *one-sided binary search*
- can you guess where the key should be? – *interpolation search*
  - but: it takes work to optimize enough to beat binary search, and a highly-tuned structure is very sensitive to a change in distribution
- does all the data fit in memory? – *B-trees* or *van Emde Boas trees*
  - cluster keys into pages to minimize disk accesses

Sorting

- how many keys will be sorted?
  - for $n \leq 100$, any quadratic-time algorithm will do
    - insertion sort faster, simpler, less likely to be buggy than bubblesort
    - shellsort much faster than insertion sort but requires getting the right insertion sequences
  - for $n > 100$, need $O(n \log n)$ algorithm
    - heapsort, quicksort, mergesort
  - for $n > 100{,}000{,}000$, need external-memory sorting algorithm to minimize disk accesses
    - B-trees
    - multiway mergesort

Sorting

- duplicate keys?
  - is there a secondary key to break ties?
  - a *stable* sort preserves the initial ordering
    - if important, probably better to explicitly code position as a secondary key
    - most of the quadratic-time algorithms are stable but most of the O(n log n) ones are not
- what do you know about your data?
  - is it already partially sorted? – insertion sort is O(n) best case
  - are the keys randomly or uniformly distributed? – bucket or distribution sort
  - are the keys very long or hard to compare?
    - use a short prefix, then resolve ties using the full key
    - radix sort
  - is the range of possible keys very small? – utilize a bit vector

Sorting

- best general-purpose internal sorting algorithm is quicksort
  - requires tuning for max performance
- use a library function instead of coding quicksort yourself
  - e.g. a poorly-written quicksort is likely slower than a poorly-written heapsort

Algorithmic Applications

Many problems boil down to direct applications of the right data structure.

e.g. basic (and not so basic) ADTs –

- Stack – reversing, matching closest, depth-first search
- Queue – breadth-first search
- PriorityQueue – sorting, best-first search, greedy algorithms
- suffix trees – many string processing applications

e.g. hashing –

- can gain speed improvements when comparing the elements themselves is expensive
  - e.g. string matching, duplicate detection
- verification, proof of possession

Match the task with a data structure / technique that can be used to solve it.

all occurrences of a pattern in a text	suffix tree
finding duplicates	hashing
finding the most common element	hashing
fingerprinting	hashing
inexact pattern matching	hashing
longest palindrome	suffix tree
longest substring common to a set of strings	suffix tree
lookup	dictionary
plagiarism detection	hashing
proof of unmodified content	hashing
sorting when deletions follow insertions	sorting algorithm
sorting when deletions are interleaved with insertions	priority queue

String Matching

- does text string *t* contain the pattern string *p* as a substring, and if so, where?
- simple algorithm
  - for each index i of t, check if p matches t[i..i+|p|-1]
  - running time: O(|t||p|)
- Rabin-Karp algorithm
  - observations
    - if $h(s_1)$ and $h(s_2)$ are different, s_1 and s_2 are different
    - if $h(s_1)$ and $h(s_2)$ are the same, s_1 and s_2 are very likely the same
  - algorithm: for each index i of t, check if p matches t[i..i+|p|-1] only if h(p) == h(t[i..i+|p|-1])
  - running time
    - computing h(s) is O(|s|) but the incremental computation of h(s[j+1...j+k+1]) given h(s[j..j+k]) is O(1)
    - total time: O(|t|+|p|) to compute all of the hashes + O(|p|) per collision x number of false collisions = O(|t|+|p|) expected
      - expected probability of a false collision is 1/N
      - with N = |t|, expect one false collision over the course of checking substrings of t
      - with N = $|t|^2$, expect very small chance of any false collisions

Rabin-Karp

- Michael Rabin, 1931 –
  - Israeli mathematician and computer scientist
  - known for work in computational complexity theory
    - Miller-Rabin primality test
    - Rabin cryptosystem
  - received Turing Award in 1976
    - for introducing the idea of non-deterministic finite automata

https://en.wikipedia.org/wiki/Michael_O._Rabin
https://en.wikipedia.org/wiki/Richard_M._Karp

Rabin-Karp

- Richard M Karp, 1935 –
  - American computer scientist
  - known for work in algorithms and complexity
    - 21 NP-complete problems
    - Edmonds-Karp algorithm – maximum flow
    - Hopcroft-Karp algorithm – maximum cardinality matchings in bipartite graphs
    - Held-Karp algorithm – exact exponential-time algorithm for TSP
    - Karp-Lipton theorem – if SAT can be solved by Boolean circuits with a polynomial number of logic gates, the polynomial hierarchy collapses
  - received Turing Award in 1985
    - development of efficient algorithms for network flow and other combinatorial optimization problems
    - connection of polynomial-time computability and algorithmic efficiency
    - a standard methodology for proving problems NP-complete

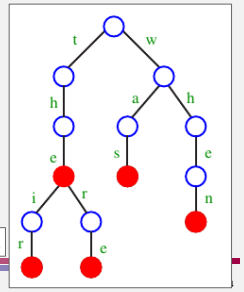
https://en.wikipedia.org/wiki/Michael_O._Rabin
https://en.wikipedia.org/wiki/Richard_M._Karp

Duplication Detection and Verification

- in a large collection of documents –
  - is a given document different from all the rest?
  - is part of a document plagiarized?
- verifying integrity
  - e.g. to ensure that the file downloaded is the authentic original file and not something substituted by a hacker
- password verification
- proof of possession
  - e.g. to prevent cheating in a closed-bid auction

Suffix Trees

In its simplest instantiation, a suffix tree is simply a *trie* of the n suffixes of an n-character string S. A trie is a tree structure, where each edge represents one character, and the root represents the null string. Each path from the root represents a string, described by the characters labeling the edges traversed. Every finite set of words defines a distinct trie, and two words with common prefixes branch off from each other at the first distinguishing character. Each leaf denotes the end of a string. Figure 15.1 illustrates a simple trie.

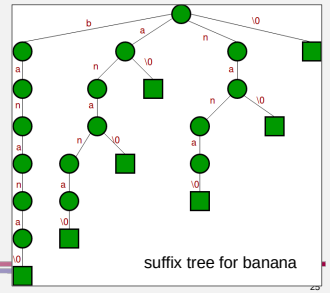


A trie on strings *the, their, there, was,* and *when*

Suffix Trees

- all occurrences of a pattern in a text

Find all occurrences of q as a substring of S – Just as with a trie, we can walk from the root to the node n_q associated with q. The positions of all occurrences of q in S are represented by the descendants of n_q, which can be identified using a depth-first search from n_q. With a collapsed suffix tree, it takes $O(|q| + k)$ time to find the k occurrences of q in S.



suffix tree for banana

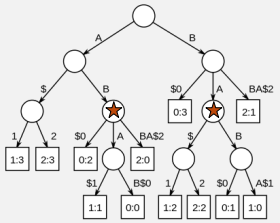
https://www.geeksforgeeks.org/pattern-searching-using-suffix-tree/

Suffix Trees

- longest substring common to a set of strings

Longest substring common to a set of strings – Build a single collapsed suffix tree containing all suffixes of all strings, with each leaf labeled with its original string. In the course of doing a depth-first search on this tree, we mark each node with both the length of its common prefix and the number of distinct strings that are children of it. From this information, the best node can be selected in linear time.

The longest common substrings of a set of strings can be found by building a generalized suffix tree for the strings, and then finding the deepest internal nodes which have leaf nodes from all the strings in the subtree below it. The figure on the right is the suffix tree for the strings "ABAB", "BABA" and "ABBA", padded with unique string terminators, to become "ABAB$0", "BABA$1" and "ABBA$2". The nodes representing "A", "B", "AB" and "BA" all have descendant leaves from all of the strings, numbered 0, 1 and 2.



https://en.wikipedia.org/wiki/Longest_common_substring

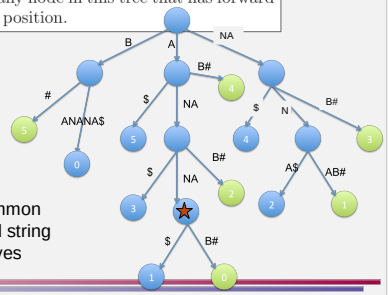
Suffix Trees

- longest palindrome

Find the longest palindrome in S – A *palindrome* is a string that reads the same if the order of characters is reversed, such as *madam*. To find the longest palindrome in a string S, build a single suffix tree containing all suffixes of S and the reversal of S, with each leaf identified by its starting position. A palindrome is defined by any node in this tree that has forward and reversed children from the same position.

S_f = banana$

S_r = ananab#



longest palindrome is the longest common substring of the forward and reversed string
→ lowest node with both $ and # leaves

https://stackoverflow.com/questions/7043778/longest-palindrome-in-a-string-using-suffix-tree