ADS Assignment

String & text processing

Pattern Matching Algorithms :-

→ Brute force pattern matching algorithms.

In the classic pattern matching problems we are given a text string T of length n and a pattern string P of length m and want to find whether P is a substring of T.

→ we should find the lowest index i within T at which P begins such that T[i...i+m] equals P. or perhaps to find all indices of T at which pattern P begins.

The brute force algorithmic design pattern is a powerful technique for algorithm design when we have something

we wish to search for or when we wish to optimize some function.

PAGE NO.
DATE / /

we typically enumerate all possible configurations of the inputs involved & pick the best of all these enumerated configurations (Test all possible placements of P relative to T).

```
def find_brute(T,P):
"return the lowest index of T at which substring P begins(or else -1)"
n,m = len(T), len(P) # introduce convenient notations.
for i in range(n-m+1) # try every potential starting index within T
    k=0                # an index into pattern P.
    while k<m and T[i+k]==P[k] # kth character of P matches
        k+=1
    if k==m:           # if we reached the end of pattern.
        return i       # substring T[i:i+m] matches P.
return -1 # failed to find a match starting with any i.
```

* Boyer-moore algorithm: It can sometimes avoid comparisons between P and a sizable fraction of the characters in T.
→ The main goal is to improve running time of the ~~brute~~ brute force algorithm by adding two potential time saving heuristics.

i) Looking glass Heuristic: When testing a possible placement of P against T, begin the comparisons from the end of P and move backward to front of P.

i) Character Jump Heuristic : During the testing of a possible placement of P within actor T[k] is handled as follows. If c is not contained anywhere in P. then shift P completely past T[i] (for it cannot anywhere in P then shift P completely past T[i]). otherwise, Shift P until an occurence of character c in P gets aligned with T[i].

→ The efficiency of the Boyer-moore algorithm relies on creating in a lookup table that quickly determines where a mismatched character occurs elsewhere in the pattern.

→ If c is in P, last(c) is the index of the last occurence of c in P otherwise we conventionally define last(c) = -1.

```
def find_boyer_moore(T,P):
"Return the lowest index of T at which substring P
begins (or else -1)".
n, m = len(T), len(P) # introduce convenient notations
if m == 0 return 0 # trivial search for empty string
last = {} # build last dictionary.
for k in range(m):
    last[P[k]] = k # later occurance overwrites.
# align end of pattern of index m-1 of text.
i = m-1 # an index into T
k = m-1 # an index into P
while i < n:
    if T[i] == P[k]: # a matching character.
        if k == 0:
            return i # pattern begin at index i of text.
```

else:
    i -= 1
    k -= 1
else:
    j =
    i +=
    k = m-1
return -1

examine previous character of
both T and P

last[i][T[j]] is -1 if not found.
Case analysis for jump step
Restart at end of pattern.

The Knuth-Morris-Pratt algorithm: Achieves running time of O(n+m) which is asymptonically optimal. The main idea of the KMP algorithm is to precompute self overlaps between portions of the pattern so that when a mismatch occurs at one location. we immediately know the maximum amount to shift the pattern before continuing the search.

To implement the KMP algorithm we will preform precompute a failure function, f that indicate the proper shift of P upon a failed comparision the failure function f(k) is defined as the length of the longest prefix of P

that is a suffix of P[i-k+1].

def find-kmp (T, P):
"Return the lowest index of T which starting P beings (or else -1)"
n, m = len(T), len(P) # introduce convenient notation
if m == 0 return 0 # trivial search for empty string

The Huffman coding Algorithm begins with each of the a distinct characters of the string X to encode beign the root node of a single node binary tree. The algorithm proceeds in a series of rounds the algorithm takes the two binary tree.

Algorithm Huffman(X):
Input: String X of length n with d distinct characters
Output: coding tree of x.
compute the frequency f(c) of each character of x Initialize a priority queue Q
for each character c in x do.
create a single node binary tree T storing c
Insert T into Q with key f(c)

while (in(Q) > 1 do.
(f1, T1) = Q.remove_min()
(f2, T2) = Q.remove_min()
create a new binary tree t with left subtree T1 and right subtree T2
Insert T into Q with key f1+f2
(f, T) = Q.remove_min()
return tree T.

Greedy Method: Huffman's algorithm for building an optimal encoding is example of greedy method. This design pattern is applied to optimization problems,

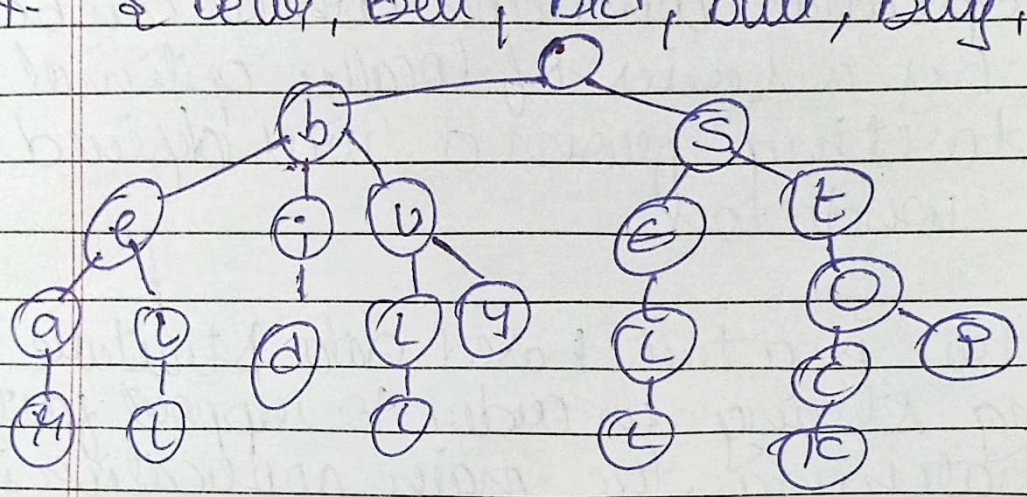
where we are trying to construct some structure while minimizing or maximizing some property of that structure. The general formula for the greedy method pattern is almost as simple as the brute force method. this approach does not always lead to an optimal solution.
For some problems it works for which passes the greedy-choice property which means that a global optimal condition which can be reduced by a series of locally optimal choices starting from a well defined starting condition.

Tries: A trie is a tree based data structure for storing strings in order to support fast pattern matching. the main application is information retrieval The primary query operations that tries support are pattern matching & prefix matching.

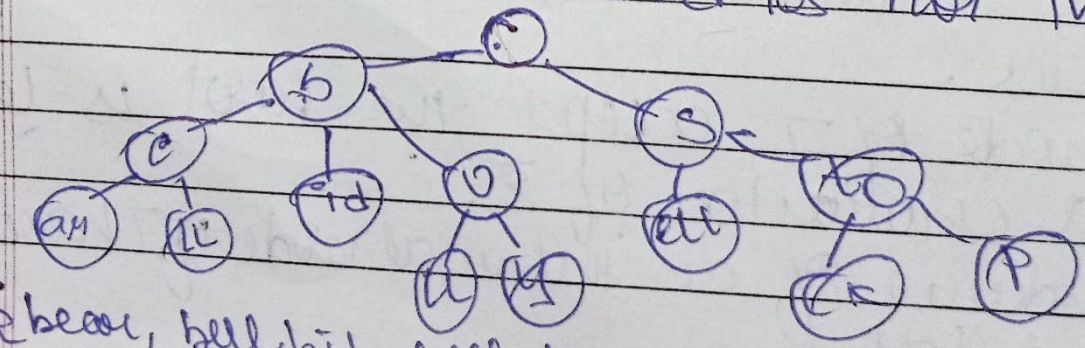
Standard tries: let S be a set of s strings from alphabet Σ such that no string in S is a prefix of another string A standard trie for S is an ordered tree T with the following properties.
→ Each node of T, except the root is labelled with a character of Σ.
→ The children of an internal node of T have distinct labels.

5) T has s leaves, each associated with a string S_i such that the concatenation of the labels of the nodes on the path from root to a leaf v of T yields the string of S associated with. Thus a trie T represents the strings of S with paths from the root to the leaves of T.

Ex: {bear, bell, bid, bull, buy, sell, stock, stop}

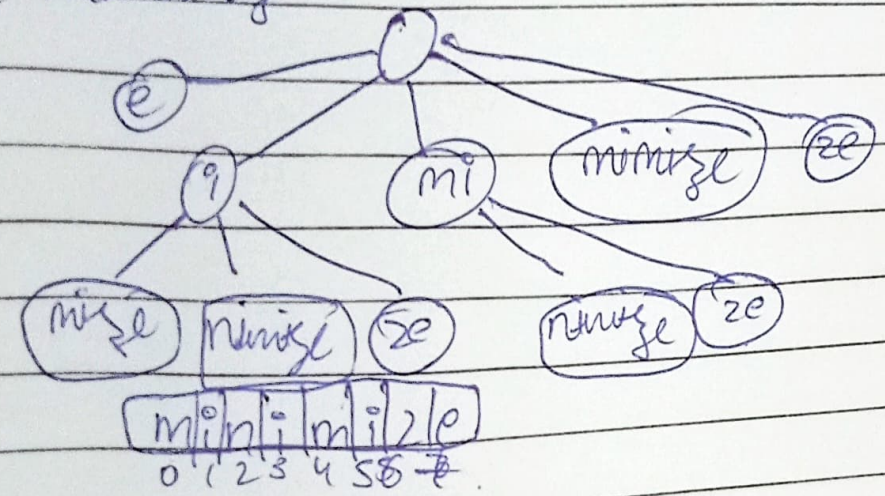


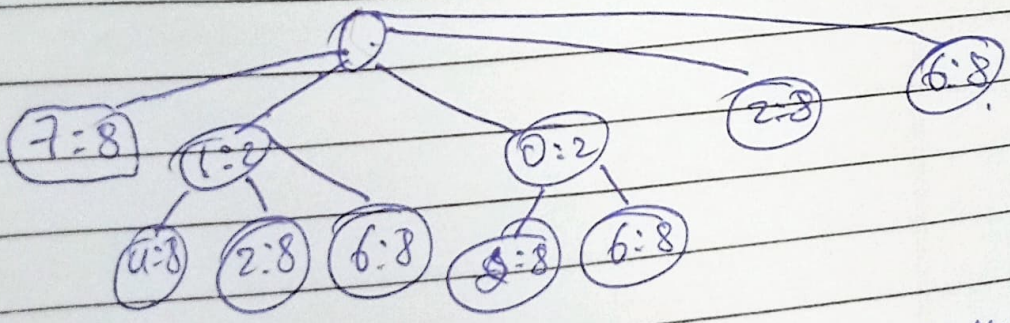
compressed trie is similar to standard trie but it ensures that each internal node in the trie has at least two children, It enforces this rule by compressing chains of single child node into individual edges let T be a standard trie we say that an internal node v of T is redundant if it has one child and is not the root.



{bear, bell, bid, bull, buy, sell, stock, stop}.

Suffix tries : one of the primary applications for tries is for the case when the strings in the collection is are all the suffies of a string X, such a trie is called Suffix trie of string X
ex = minimize.





Using a suffix trie allows us to save space over a standard trie by using several space compression technique

The suffix trie T for a string X can be used to efficiently perform pattern matching queries on text X.