University of Waterloo
CS 240 Winter 2018
Review

Theorem: Any comparison based implementation of size$-n$ ADT dictionary requires $\Omega(\log n)$ comparison for some search operations.
Proof: Any algorithm defines a binary decision tree with comparisons at the nodes and actions at the leafs. There are at least $n+1$ answers: returning an item or "not found". Therefore, the decision tree has at least $n+1$ leaves, and the height is $\Omega(\log n)$. This implies that some leaf is at level $\log(n)$ or lower, and the input that led to this answer requires $\Omega(\log n)$ comparison.

Hashing

Definition: The load factor of a hash table is

$$\alpha := \frac{n}{M} = \frac{\#\text{ of keys}}{\text{size of table}}$$

We can choose $M \implies$ We can choose α.

Hashing with chaining:
Assuming uniform hashing, average bucket size is exactly α.
Analysis of operations and space:

- search: $\Theta(1+\alpha)$ average-case, $\Theta(n)$ worst-case
- insert: $O(1)$ worst-case since we always insert at the front
- delete: same as search: $\Theta(1+\alpha)$ average-case, $\Theta(n)$ worst-case
- space: $O(M)$, we can keep it at $\Theta(n)$ through rehashing

If we maintain $M \in \Theta(n)$, then average costs are all $O(1)$. This can be accomplished by rehashing whenever $n < c_1 M$ or $n > c_2 M$, for some constatns c_1, c_2 with $0 < c_1 < c_2$.

Rehashing:
We start with some small M.
During insert/delete, keep track of n, M, α.
If α gets "too big", say $\alpha > \frac{1}{2}$,

- create a new table, size $= M' \geq 2M$
- find a new hash-function $h' : U \to \{0, \ldots, M'-1\}$
- create a new hash-table $T'[0 \ldots M'-1]$

– for each item in old hash-table T, insert it in the new hash-table T'

Complexity: $O(M' + n \cdot \text{insert})$
Since new α is small, every insert should be fast, i.e. $O(1)$. Rehashing happens rarely so we will not count this time towards the insertion time (amortized analysis).

Open addressing with probe sequence:

For each key k, we have a probe sequence $\langle h(k,0), h(k,1), h(k,2), \dots \rangle$. To insert, try each value in the probe sequence.
Possible probe sequences:

- Linear probing: $h(k,i) = (h(k) + i) \mod M$
- Quadratic probing: $h(k,i) = (h(k) + c_1 i + c_2 i^2) \mod M$
- Double hashing: $h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod M$ where $h_2(k) \neq 0$ for any k, assuming we have 2 *independent* hash functions h_1, h_2.
  Note: to get valid probe sequences, we need $\gcd(h_2(k), M) = 1 \implies$ choose M prime

Deletiton:
We cannot leave an empty spot behind as the next search might otherwise not go far enough. We can use *lazy deletion*: mark the item as "deleted", then adjust insert/search to deal with deleted keys.

Open addressing with cuckoo hashing:
An item with key k can only be in $T[h_1(k)]$ or $T[h_2(k)]$, assuming we have two hash functions $h_1 : U \to \{0, \dots, M-1\}$, $h_2 : U \to \{0, \dots, M-1\}$. Search and delete therefore take $O(1)$ time.
Insertion:

- Always try h_1 first.
- If $T[h_1(k)]$ is occupied: "kick out" the other item, which we then attempt to re-insert into its alternate position.
- Safety check: loop at most n times to avoid an infinite loop of "kicking out".
- In case of failure, rehash with a larger M and new hash functions.

```
1 cuckoo-insert(T, x):
2 // T: hash table, x: new item to insert
3 y → x, i → h1(x.key)
4 do at most n times:
5    swap(y, T[i])
6    if y is "empty" then return "success"
```

```
7     // swap i to be the other hash-location
8     if i = h1(y.key) then i → h2(y.key)
9     else i → h1(y.key)
10 return "failure"
```

Can show: if $\alpha = \frac{n}{M} < \frac{1}{2} - \epsilon$, then insert has $O(1)$ expected run-time.

Choosing hash functions:

- easy to compute
- avoid patterns in data
- depend on all parts of the key

Modular method: $h(k) = k \mod M$, where M is prime
Multiplication method: $h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$, for some constant floating-point number A with $0 < A < 1$. Knuth suggests $A = \frac{\sqrt{5}-1}{2} \approx 0.618$.

Universal hashing: randomization to avoid bad input
On start up, and when rehashing, do:

- find a prime number P that is $\approx$ the desired table size
- randomly choose $a, b \in \{0, \ldots, P-1\}, a \neq 0$
- use as hash function $h(k) = (ak + b) \mod P$ in a table of size P

Hashing vs. Balanced Search Trees:
Advantages of Balanced Search Trees:

- $O(\log n)$ worst-case operation cost
- Does not require any assumptions, special functions, or known properties of input distribution
- Predictable space usage (exactly n nodes)
- Never need to rebuild the entire structure
- supports ordered dictionary operations (rank, select etc.)

Advantages of Hash Tables:

- $O(1)$ operations if hashes well-spread and load factor small
- We can choose space-time tradeoff via load factor
- Cuckoo hashing achieves $O(1)$ worst-case for search and delete

Range-Searching in Dictionaries for Points

Range Search Query:
Multi-dimensional data: each item has d aspects, and aspect values are numbers. Each item corresponds to a point in d-dimensional space.

Quadtrees
Assume: All points are within a sqaure R, ideally the width/height of R is a power of 2.
How to build the quadtree on $S = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$:

- Root r of the quadtree corresponds to R
- If R contains 0 or 1 point(s), then root r is a leaf that stores point
- Else split: Partition R into four equal subsquares (quadrants) $R_{NE}, R_{NW}, R_{SW}, R_{SE}$
- Root has four children $v_{NE}, v_{NW}, v_{SW}, v_{SE}$; v_i is associated with R_i
- Recursively repeat this process at each child

Rules: Points on split lines belong to **top/right** side. We could delete leaves without point (but then need edge labels).

Quadtree Dictionary Operations:
Insert: search for the point, split the leaf if there are two points.
Delete: search for the point, remove the point, if its parent has only one child left, delete that child and continue the process toward the root.

Quadtrees are similar to tries:
point coordinates = bit strings (binary number, padded to same length)
On level i, we split by ith bit of both x-coordinate and y-coordinate. (still 4 children)

Quadtree Range Search: 4 cases

- $R \subseteq A$: All points are in subtree
- $R \cap A$ is empty: no points in subtree
- Leaf: subtree only has one point
- can't decide: delegate to children (recursion)

```
1 QTree-RangeSearch(T, A):
2 // T: the root of a quadtree, A: query rectangle
3 let R be the rectangle associate with T
4 if (R ⊆ A) then
```

```
5     report all points in T; return
6  if (R ∩ A is empty) then
7     return
8  if(T stores a single point p) then
9     if p is in A return p
10    else return
11 for each child v of T do:
12    QTree-RangeSearch(v, A)
```

Quadtree Analysis:
spread factor of points S: $\beta(S) = \frac{\text{sidelength of R}}{d_{\min}}$, where $d_{\min}$ is the minimum distance between two points in S
height of quadtree: $h \in \Theta(\log \beta(S))$
Complexity to build initial tree: $\Theta(nh)$ worst-case
Complexity of range search: $\Theta(nh)$ worst-case even if the answer is $\emptyset$. In practice it is much faster.
Space is potentially wasteful, but not if points are well-distributed

kd-trees
(Point-based) kd-tree idea: Split the region such that (roughly) half the points are in each subtree
Each node of the kd-tree keeps track of a splitting line in one dimension (2D: either vertical or horizontal)
Continue splitting, switching between vertical and horizontal lines, until every point is in a separte region
Tie-breaking: points on splitting lines go to **bottom/left**

Assume first that no two points have the same x-coordinate or y-coordinate. Then the split always put $\lfloor \frac{n}{2} \rfloor$ points on one side and $\lceil \frac{n}{2} \rceil$ points on the other, so height $h(n)$ satisfies the recursion $h(n) \leq h(\lceil \frac{n}{2} \rceil) + 1$. This resolves to $h(n) \leq \lceil \log(n) \rceil$.

Constructing kd-trees:
Build kd-tree with initial split for x (vertical line) on points S:

- If $|S| \leq 1$ create a leaf and return
- Else find median X of x-coordinates in S
- Partition S into $S_{x \leq X}$ and $S_{x > X}$ by comparing points' x-coordinate with X
- Create left child with recursive call (splitting on y) for points $S_{x \leq X}$
- Create right child with recursive call (splitting on y) for points $S_{x > X}$

Analysis:
Find median and partition in linear time.

$\Theta(n)$ work on each level in the tree (summed over all nodes)
Total is $\Theta(\text{height} \cdot n) = \Theta(n \log n)$.

```
1  kdTree-RangeSearch(T, R, A):
2  // T: the root of a kd-tree, R: region associated with T, A: query rectangle
3  if (R ⊆ A) then report all points in T; return
4  if (R ∩ A is empty) then return
5  if (T stores a single point p) then
6     if p is in A return p
7     else return
8  if T stores split "is x ≤ X"?
9     R_ℓ ← R ∩ {(x, y) : x ≤ X}
10    R_r ← R ∩ {(x, y) : x > X}
11    kdTree-RangeSearch(T.left, R_ℓ, A)
12    kdTree-RangeSearch(T.right, R_r, A)
13 else // root node splits by y-coordinate
14     ...
```

kd-tree Analysis:
The complexity is $O(s + Q(n))$ where
- s is the number of keys reported (output-size)
- s can be anything from 0 to n
- No range-search can work in $o(s)$ time since it must report the points
- $Q(n)$ is the number of nodes for which `kdTreeRangeSearch` was called
- can show: $Q(n)$ satisfies the following recurrence relation: $Q(n) \leq 2Q(\frac{n}{4}) + O(1)$, which resolves to $Q(n) \in O(\sqrt(n))$.
Therefore, the complexity of range search in kd-trees is $O(s + \sqrt{n})$.

kd-tree: d-dimensional space:

- At the root the point set is partitioned based on the first coordinate
- At the children of the root the partition is based on the second coordinate
- At depth $d - 1$ the partition is based on the last coordinate
- At depth d we start all over again, partitioning on first coordinate

Storage: $O(n)$
Construction time: $O(n \log n)$
Range query time: $O(s + n^{1-\frac{1}{d}})$
This assumes that $o(n)$ points share coordinates and d is a constant.

Range Trees:

```
1  BST-RangeSearch(T, k1, k2):
2     // T: root of a bst, k1, k2: search keys
3      // returns keys in T that are in range [k1, k2]
4      if T = null then return
5      if k1 ≤ key(T) ≤ k2 then
6        L ← BST-RangeSearch(T.left, k1, k2)
7        R ← BST-RangeSearch(T.right, k1, k2)
8        return L ∪ {key(T)} ∪ R
9      if key(T) < k1 then
10       return BST-RangeSearch(T.right, k1, k2)
11     if key(T) > k2 then
12       return BST-RangeSearch(T.left, k1, k2)
```

This results in unnecessary searches.
Rephrase:
Search for left boundary k_1 giving path P_1, for right boundary k_2 giving path P_2.
Partition nodes of T into 3 groups:

1. boundary nodes: nodes in P_1 or P_2
2. inside nodes: nodes that are right of P_1 and left of P_2
3. outside nodes: nodes that are left of P_1 and right of P_2

Report all inside nodes, then test each boundary node and report it if it is in range.

Analysis:
Assuming that the binary search tree is balanced, then search for both paths take $O(\log n)$ time. There are $O(\log n)$ boundary nodes, but could have many inside nodes.
We only need the topmost of them: allocation node v that satisfies

- not in P_1 or P_2, but parent is in P_1 or P_2 but not both
- if parent is in P_1, then v is right child
- if parent is in P_2, then v is left child

There are $O(\log n)$ allocation nodes; for each of them report all descendants.
Run-time: $O(\#$ boundary nodes $+ \#$ reported points$) = O(\log n + s)$

2-dimensional Range Trees:
We have n points $P = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$
A range tree is a tree of trees (multi-level data structure)
Primary structure: binary search tree T that stores P and sorted by x-coordinate
Each node v of T has an auxiliary structure $T(v)$:
- Let $P(v)$ be all points at descendants of v in T (including v)
- $T(v)$ stores $P(v)$ in a binary search tree, sorted by y-coordinate

- v is not necessarily the root of $T(v)$

Space analysis:
Primary tree uses $O(n)$ space, associate tree $T(v)$ uses $O(|P(v)|)$ space.
$w \in P(v)$ means that v is an ancestor of w in T.
- Every node has $O(\log n)$ ancestors in T
- Every node belongs to $O(\log n)$ sets $P(v)$
- So $\Sigma_v |P(v)| \leq n \cdot O(\log n)$
Range tree space usage: $O(n \log n)$

Insert: first, insert point by x-coordinate into T. Then, walk back up to the root and insert the point by y-coordinate in al $T(v)$ of nodes v on path to the root.

To perform a range search query $A = [x_1, x_2] \times [y_1, y_2]$:

- Perform a range search (on the x-coordinates) for the interval $[x_1, x_2]$ in primary tree T (BST-RangeSearch(T, x_1, x_2))
- Obtain boundary, topmost outside and allocation nodes as before
- For every allocation node v, perform a range search on the y-coordinates for the interval $[y_1, y_2]$ in $T(v)$
- For every boundary node, test to see if the corresponding point is within the region A

Query Run-time:
$O(\log n)$ time to find boundary and allocation nodes in primary tree.
There are $O(\log n)$ allocation nodes.
$O(\log n + s_v)$ time for each allcation node v, where s_v is the number of points in $T(v)$ that are reported
Every point is reported in exactly one auxiliary structure, so $\Sigma s_v = s$
Hence the time for range-query in range tree is $O(s + \log^2 n)$.

For d-dimensional space:
- storage: $O(n(\log n)^{d-1})$
- construction time: $O(n(\log n)^{d-1})$
- range query time: $O(s + (\log n)^d)$

Summary:

Quadtrees:

- simple
- work well only if points are well distributed

– wastes space for higher dimensions

kd-trees:

– linear space

– query time $O(\sqrt{n})$

– inserts/deletes destroy balance

– care needed for duplicate coordinates

range trees:

– fastest range search $O(\log^2 n)$

– wastes more space

– insert and delete more complicated

String Matching

DFA

Matching time on a text stirng of length n is $\Theta(n)$.
Preprocessing can clearly be done in $O(m^3|\Sigma|)$ time, but there exists an algorithm to do it in $O(m|\Sigma|)$ time.
Altogether it has run-time $O(m|\Sigma| + n)$.

Knuth-Morris-Pratt(KMP) Algorithm

Use a new type of transition $\times$ ("failure") only if no other fitsl. It does *not* consume a character. Even though computations of this automaton are deterministic, it is formally not a valid DFA (pseudo-DFA).
Stores failure-function in an array $F[0 \ldots m-1]$. The failure arc from state j leads to $F[j-1]$.

```
1  KMP(T,P), to return the first match
2  // T: String of length n (text), P: string of length m (pattern)
3  F ← failureArray(P)
4  i ← 0 // current character of T to parse
5  j ← 0 // current state that we are in
6  while i < n do:
7     if P[j] = T[i] then
8        if j = m - 1 then
9           return i - m + 1 //match
10       else
11          i ← i + 1
12          j ← j + 1
```

```
13    else // P[j] ≠ T[i]
14       if j > 0 then
15          j ← F[j-1]
16       else
17          i ← i + 1
18 return FAIL // no match
```

$F[j]$ is the length of the longest prefix of $P[0 \ldots j]$ that is a suffix of $P[1 \ldots j]$. Can be computed in $\Theta(m)$ time.

KMP run-time analysis:
At each iteration of the while loop, at least one of the following happens:
1. i increases by one, or
2. the index $i - j$ increases by at least one ($F[j-1] < j$)
There are no more than $2n$ iterations of the while loop
Running time: $\Theta(n + m)$

Boyer-Moore Algorithm

Ideas:

- Reverse-order searching: compare P with a subsequence of T moving backwards
- Bad character jumps: When a mismatch occurs at $T[i] = c$, use location of c in P (if any) to eliminate guesses
- Good suffix jumps: When a mismatch occurs, then use recently seen suffix to P to eliminate guesses (Similar to failure arrays in KMP)

```
1  BoyerMoore(T,P):
2     L ← last occurence array computed from P
3     S ← good suffix array computed from P
4     k ← 0 // current guess
5     while k ≤ n - m:
6        i ← k + m - 1, j ← m - 1
7        while j ≥ 0 and T[i] = P[j] then
8           i ← i - 1
9           j ← j - 1
10       if j = -1 return k
11       else
12          k ← k + max(1, j-L[T[i]], m-1-S[j])
13    return FAIL
```

Last occurrence Function: $L(c)$ is defined as the largest index i such that $P[i] = c$ or -1 if no such index exists.
Good suffix array: $S[j]$ is the maximum index ℓ such that

- $P[j+1 \dots m-1]$ is a suffix of $P[0 \dots \ell]$ and $P[j] \neq P[\ell - m + j + 1]$, or
- $P[0 \dots \ell]$ is a suffix of $P[j+1, \dots, m-1]$, or
- -1 if neither of the above holds

Boyer-Moore Run-time analysis:
Worst-case running time $\in O(n + m + |\Sigma|)$ if P is not in T.

Rabin-Karp Fingerprint Algorithm
Idea: compute hash function for each text position, then compare with pattern hash. If a match of the hash value of the pattern and a text position found, then compares the pattern with the substring by naive approach.
To improve run-time: use a rolling hash function.
We can compute $h(T[k+1 \dots k+m])$ from $h(T[k \dots k+m-1])$ in constant time.

$$h(T[k \dots k+m-1]) = (\sum_{i=0}^{m} T[k+i] \cdot 10^{m-1-i}) \mod M$$
$$h(T[k+1 \dots k+m]) = ((h(T[k \dots k+m-1]) - T[k] \cdot 10^{m-1}) \cdot 10 + T[k+m])$$

Overall expected run-time: $O(m+n)$
Worst-case run-time: $O(mn)$ but highly unlikely

Tries of suffixes and suffix trees

If we want to search for many pattens P within the same fixed text T, we should preprocess T instead.
Observation: P is a substring of T if and only if P is a prefix of some suffix of T.
Suffix tree: a compressed trie storing all suffixes of T as indices. Can be built in $\Theta(n)$ time.

Pattern matching:
Assume that P does not have \$.
In the suffix tree, search for P until one of the following occurs:

- If search fails due to "no such child" then P is not in T
- If we reach end of P, say at node v, then jump to leaf ℓ in subtree of v, presuming that suffix trees stores such shortcuts (need to check since it is compressed)
- Else we reach a leaf $\ell = v$ while characters of P left

Either way, left index at ℓ gives the shift we should check. This takes $O(|P|)$ time.

Analysis:
Run-time: $\Theta(m)$

Space: $\Theta(n)$ since we store only indices

	Brute force	Rabin Karp	KMP	Boyer-Moore	Suffix trees
preprocessing		$\Theta(m)$	$\Theta(m)$	$\Theta(m + \|\Sigma\|)$	$\Theta(n)$
search time	$\Theta(mn)$	$\Theta(n + m)$(exp)	$\Theta(n)$	$\Theta(n)$	$\Theta(m)$
extra space		$O(1)$	$\Theta(m)$	$\Theta(m + \|\Sigma\|)$	$\Theta(n)$
When to use	m small or close to n	searching for same P in many different texts			searching for many P in the same text

Compression

Encoding

Have: source text S over alphabet Σ_S,
Want: convert S into new text C over Σ_C (usually {0,1}) such that C is smaller
compression ratio =

$$\frac{|C| \cdot \log |\Sigma_C|}{|S| \cdot \log |\Sigma_S|}$$

Character-by-character encoding:
Map each character from the source alphabet to a string in coded alphabet.
$E : \Sigma_S \to \Sigma_C^*$
For $c \in \Sigma_S$, we call $E(c)$ the codeword of c.
Fixed-length code: All codewords have the same length (e.g. ASCII).

The decoding algorithm must map Σ_C^* to Σ_S^*.
The code must be *uniquely decodable*. From now on, we only consider prefix-free codes E:
$E(c)$ is not a prefix of $E(c')$ for any $c, c' \in \Sigma_S$.
A prefix-free code naturally corresponds to a trie with characters of Σ_S only at the leaves.

Huffman's Algorithm

1. Determine frequency of each character $c \in \Sigma$ in S.
2. For each $c \in \Sigma$, create a height-0 trie holding c.
3. Assign a "weight" to each trie: sum of frequencies of all letters in trie. Initially, these are just the character frequencies.
4. Find the two tries with the minimum weight.
5. Merge these tries with new interior node; new weight is the sum. (corresponds to adding one bit to the encoding of each character)
6. Repeat steps 4-5 until there is only 1 trie left; this is D.

Use a min-oriented heap to make this efficient.

Run-time analysis:
Encoding: $O(n + |\Sigma_S| \log |\Sigma_S| + |D| + n) \in O(n + |\Sigma_S| \log |\Sigma_S|)$
Decoding is faster; asymmetric scheme.
The constructed trie is optimal in the sense that C is shortest (among all prefix-free character-encodings with $\Sigma_C = \{0, 1\}$).

Run-Length Encoding (RLE)

- Give the first bit of S (either 0 or 1)
- Then give a sequence of integers indicating run lengths

Prefix-free Encoding for Integers:
Encode the length of k in unary, followed by the actual value of k in binary.
The binary length of k is $\text{len}(k) = \lfloor \log k \rfloor + 1$. Since $k \geq 1$, we will encode len(k)-1, which is at least 0.
The prefix-free encoding of the positive integer k is in two parts:
1. $\lfloor \log k \rfloor$ copies of 0, followed by
2. The binary representation of k which always start with 1

Lempel-Ziv-Welch (LZW) Compression

Fixed-width encoding using k bits. First $|\Sigma_S|$ entries are for single characters, remaining entries involve *multiple* characters.
Encoding: after encoding a substring x of S, add xc to D where c is the character that follows x in S.
Decoding: after decoding a substring y of S, add xc to D, where x is previously encoded/decoded substring of S, and c is the first character of y.

```
1  LZW-encode(S):
2  // S: stream of characters
3  w ← NIL
4  C ← empty string
5  while there is input in S do
6     K ← next symbol from S
7     if wK exists in the dictionary
8        w ← wK
9     else
10       C.append(index(w))
11       add wK to the dictionary
12       w ← K
13 C.append(index(w))
14 return C
```

Generally, decoder is "one step behind" in creating dictionary, so problem occurs if we want to use a code that we're about to build. However, there is enough information about what is going on.
Suppose we are seeing code-number k at the time that we are assigning k. Decoder knows s_{prev} (decoded in previous step), and wants s (what k decodes to). We know that the encoder assigns k to $s_{prev} + s[0]$, and also that $s[0]$ = first character of what k decodes to. Therefore $s = s_{prev} + s_{prev}[0]$.

```
1  LZW-decode(C):
2  // C: stream of integers
3  D ← dictionary that maps \{0,...,127\} to ASCII
4  idx ← 128
5  S ← empty string
6  code ← first code from C
7  s ← D(code); S.append(s)
8  while there are more codes in C do:
9     s_prev ← s
10    code ← next code of C
11    if code = idx
12       s ← s_prev+s_prev[0]
13    else
14       s ← D(code)
15    S.append(s)
16    D.insert(idx,  s_prev+s[0])
17    idx++
18 return S
```

Summary

Huffman	**Run-length encoding**	**Lempel-Ziv-Welch**
variable-length	variable-length	fixed-length
single-character	multi-character	multi-character
2-pass	1-pass	1-pass
60% compression on English text	bad on text	45% compression on English text
optimal 01-prefix-code	good on long runs	good on English text
must send dictionary	can be worse than ASCII	can be worse than ASCII
rarely used directly	rarely used directly	frequently used
part of pkzip, JPEG, MP3	fax machines, old picture-formats	GIF, some variants of PDF, Unix compress

Text transformations

Move-to-front transformation (MTFT):
Start with an initial dictionary stored in an array. If you have a long run of characters in S, then you would have a long run of 0's in C.

Burrows-Wheeler Transform:
Transforms source text to a coded text with the same letters, just in a different order.
The coded text will be more easily compressible with MTFT (long runs of characters).
Encoding needs all of S (can't use streaming) as BWT is a block compression method.

BWT Encoding:
Assume the source text S ends with a special end-of-word character \$ that occurs nowhere else in S.

1. Place all cyclic shifts of S in a list L.
2. Sort the strings in L lexicographically.
3. C is the list of trailing characters of each stirng in L.

S = alf_eats_alfalfa\$
C = asff\$f_e_lllaaata

BWT Decoding:
Idea: Given C, we can generate the first column of the array by sorting. This tells us which character comes after each character in S.

Analysis:
Encoding cost: $O(n^2)$ using radix sort
Sorting cyclic shifts is equivalent to sorting suffixes, possible in $O(n)$ time

Decoding cost: $O(n)$, asymmetric

bzip2 compression:
BWT $\rightarrow$ MTFT $\rightarrow$ RLE $\rightarrow$ Huffman