Knuth-Morris-Pratt
O(m+n). Compare from left to right. When mismatch occurs, shift to the longest proper prefix: longest prefix of P[0:j-1] is also a suffix of P[1:j-1]. Failure function gives the length of this prefix.

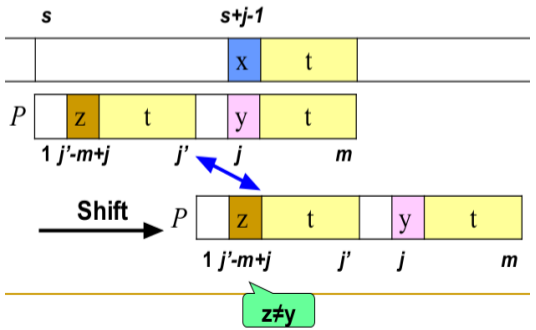
```
Algorithm failureFunction(P)
    F[0] ← 0
    i ← 1
    j ← 0
    m ← length(P)
    while i < m
    if P[i] = P[j]
    {we have matched j + 1 chars}
        F[i] ← j + 1
        i ← i + 1
        j ← j + 1
    else if j > 0 then
    {use failure function to shift P}
        j ← F[j − 1]
    else
        F[i] ← 0 { no match }
        i ← i + 1
```

Boyar-Moore
Bad Character Rule
Upon mismatch, shift the pattern to the right such that the largest position in the left of the pattern matches the char in the text. If no such char exists, move the pattern start 1 position to the right.

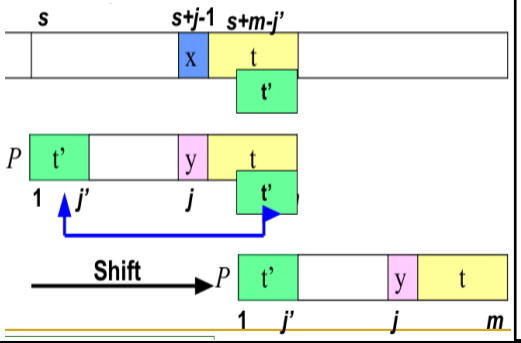
Good Suffix Rule 1
Upon mismatch, move to the largest shift such that pattern heads are different, but the right side matches with text's right side.

Good Suffix Rule 2
Upon mismatch check if there's a suffix right of current text head that is a prefix of the pattern. If so, shift to align these.

Good Suffix Rule 1



Good Suffix Rule 2



Shift amount
max(bad_char, min(good_suffix1, good_suffix2))

Preprocessing
O(m+Sigma)
All Matches
O(mn), if at all positions
O(m+n), no match

Rabin-Karp Fingerprint
f_0 = (P[m-1] + base*(P[m-2]) + base^2* P[m-3] …) mod q
f = ((f – T[s] * base^(m-1) mod q) * base + T[s+m]) mod q
Preprocessing: O(m), total time: O(n-m), Worst case: O(nm)

Rabin-Karp Fingerprint
f_0 = (P[m-1] + base*(P[m-2]) + base^2* P[m-3] …) mod q
f = ((f – T[s] * base^(m-1) mod q) * base + T[s+m]) mod q

Preprocessing: O(m), Iter. over text: O(n-m), loop to compare with pattern: O(m), overall: O(nm). Choosing prime, randomized: O(m).

Finite Automata Search
Construct transition table such that phi(s) = state after transition. Sigma (x) = for pattern p, length of longest prefix of p that is also a suffix of x.

For p = ab, sigma(ccaca) = 1, sigma(epsilon) = 0, etc.

Transition function
delta(state, letter) = sigma(P[:state] + letter)

For P = ababaca, delta(5,b) = sigma(ababab) = |abab| = 4

Search: O(n) -- length of text,
Preprocessing: O(m * |Sigma|)
Memory: O(m * |Sigma|)

BITAP Shift-And
Bitshift(j): Rotate one to right/down, set first index to 1
U(letter) = [1 if pos[i] = letter else 0]
col-{j+1} = Bitshift(col-j) bitwise-and U(T[j])
Start: col-j is all zeros.

A bit is 1 if there's a previous match and a matching char occurs. If last index is 1, pattern match occurs.

Runtime: O(m * n), *Space:* O(m) since only two columns are necessary.

Exact Matching – O(mn)
Multiple Matching – O(kmn) for k patterns.

Keyword Trees

Build a keyword in O(N) time, N=total length of all patterns
Naive threading O(N + nm)
Aho-Corasick O(N + m)

Construct a n-node tree, each path from a leaf gives a pattern in your dictionary. During construction, do a search to find a path to diverge from. At search time, push the characters one by one down the tree. If you reach a leaf, you have a match. *Too much space!*

Suffix Trees - Trie

Instead of pushing chars; list the suffixes first, then construct the tree.
Three cases:
2. Root is empty – create a child directly.
3. No common prefix with current root's children – append a new child with current prefix as a leaf
4. There's a no-empty prefix match with a child – create a new node with common prefix, assign it as a child of the parent. Create edges labeled unmatching parts of the child and the current string, connect them to this new parent.

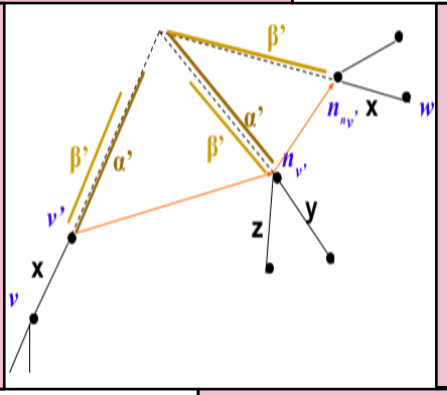
If you have multiple patterns, assign different end symbol to each.
<u>Longest common substring:</u> Find the node with the largest depth that has a descendant from all strings.
<u>At search time</u>, check if a substring starting from current index matches a child of the current root. If you reach a leaf, you have a match. Leaves also give the length of the pattern for easy retrieval.

Pattern Matching:
O(m) construction for length-m pattern, O(n) query. O(n+m) in total.

Multiple Pattern Matching:
O(n) construction for the text, O(n+km) for querying k patterns.

Aho-Corasick

Construct an NFA with skip connections named *failure links* over a keyword tree. Remembers partial matches.
Runtime: O(n+m), O(n) construction, O(m) threading. n=|text|, m=|pattern|.
Construction:
1. If the root or root's children, failure node(fn) is the root.
2. If there's an edge from the root and fn labelled x, failure link(fl) is x-child of the previous failure link.
3. If no common edge is found, traverse to the next fn in hierarchy until you reach the root, searching for a common edge labelled x and a common proper suffix y.



Suffix Arrays

Lexicographically sorted suffix-start index pairs. Suffixes are not stored explicitly. Insert a char $ at the end which has the smallest lexicographical value in alphabet.

Naive construction: In-place sorting with insertion sort, O(m^2). O(mlogm) possible, not discussed.

O(n) space, n= |string|. Increased query time.

Querying is based on binary search, O(m * logm) complexity: From left to right, find range of suffixes that start with the letter i of pattern p at index j. Left and right pointers are adjusted each iteration, continue until the prefix of a suffix matches with the pattern. Indices of the matching cells give you the occurences.

Shortcomings of suffix arrays:
1. No approximate matches
2. Change is expensive and require rebuilding the suffix array.

Burrows-Wheeler Transformation (BWT)

Reorders text for improved locality and better compression. Append a char $ at the end with smallest lexicographic value, generate all rotations. First column becomes the suffix array, second one becomes the BWT array.

Alternatively, BWT[i] = T[SA[i] – 1] if i > 0, else $.
In other words, characters left of the suffix arrays is BWT characters on the same row.

LF Map

Nth character c in BWT is the nth character in suffix array. Suffix array brings together same-type characters and orders them lexicographically.

Have an array of character counts and a matrix of occurences of all characters until each index. Also, keep an array indicating first occurence of a character in the original string, named ranks array – not directly relevant to LF map but will be useful later.

[Cnt('$') +
Cnt('i') +
Cnt('m') +
Cnt('p') = 8]
+ [Occ(9, 's')= 3]
= 11

before 's' →

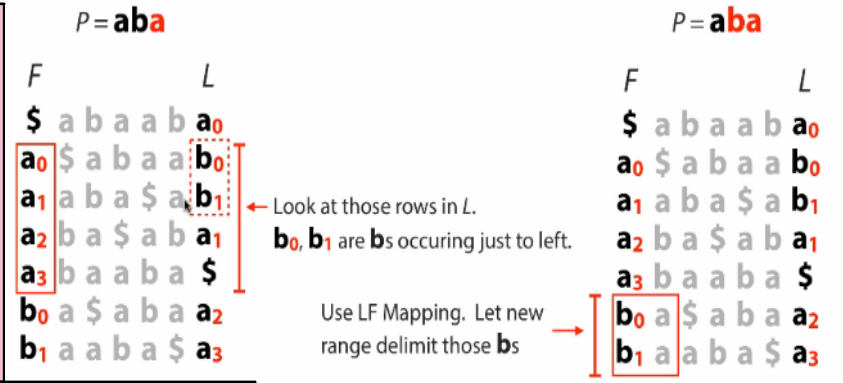
's' section →

Cnt('c')

$	i	m	p	s
1	4	1	2	4

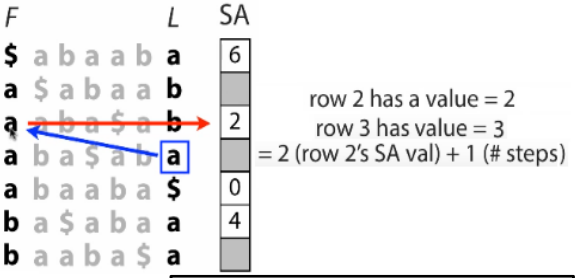
1	$	m	i	s	s	i	s	s	i	p	p	i
2	i	$	m	i	s	s	i	s	s	i	p	p
3	i	p	p	i	$	m	i	s	s	i	s	s
4	i	s	s	i	p	p	i	$	m	i	s	s
5	i	s	s	i	s	s	i	p	p	i	$	m
6	m	i	s	s	i	s	s	i	p	p	i	$
7	p	i	$	m	i	s	s	i	s	s	i	p
8	p	p	i	$	m	i	s	s	i	s	s	i
9	s	i	p	p	i	$	m	i	s	s	i	s
10	s	i	s	s	i	p	p	i	$	m	i	s
11	s	s	i	p	p	i	$	m	i	s	s	i
12	s	s	i	s	s	i	p	p	i	$	m	i

FM Index
Uses BWT + Suffix Array + Rank array + occurence table mentioned before. Space-efficient way of pattern matching at the cost of query time.
Query is executed from right to left, last char first. The idea is to do binary search on suffixes, then to scan the BWT characters for a match based on the next character in the pattern, do LF mapping to find the range containing suffixes that start with this character and continue until match.



Issues: Last-character search is O(m). Storing occurences is space-expensive. Finding the exact location of the matching string is nontrivial.
Solution 1: Fast Occurence(rank) Calculation
Keep a sparse occurence table with checkpoints, compute the ranks based on these values. If checkpoint is before you, add one to checkpoint rank when you see an identical character. If checkpoint is after you, substract one from the checkpoint when the same happens.

This speeds up the scan and saves space, fixes the first two problems.
Solution 2: Finding Matching Index
Keep a sparse suffix array without the strings. Select suffixes having evenly-spaced lengths: 0,3,6,9, etc. If you match to a checkpoint in the suffix array, return the location directly. Otherwise, do LF mapping and count the number of steps until you hit a checkpoint. Return checkpoint + # steps.



$m = \|T\|, n = \|P\|,$ $k = \#\ P \text{ in } T$	Suffix tree	Suffix array	FM Index
Time: Does P occur?	$O(n)$	$O(n \log m)$	$O(n)$
Time: Count k occurrences of P	$O(n+k)$	$O(n \log m)$	$O(n)$
Time: Report k locations of P	$O(n+k)$	$O(n \log m + k)$	$O(n+k)$
Space	$O(m)$	$O(m)$	$O(m)$
Needs T?	*yes*	*yes*	*no*
Bytes per input character	>15	~4	~0.5

a: fraction of Suffix Array rows we keep
b: fraction of Occ rows we keep
Components of FM Index:
First column (F): ~ | ∑ | integers
Last column (L): m characters
SA sample: m · a integers, a is fraction of SA elements kept
Checkpoints: m · | ∑ | · b integers, b is fraction of occ kept
For DNA alphabet (2 bits / nt), T = human genome, a = 1/32, b = 1/128 :
First column (F): 16 bytes
Last column (L): 2 bits * 3 billion chars = 750 MB
SA sample: 3 billion chars * 4 bytes / 32 = ~ 400 MB
Checkpoints: 3 billion * 4 alphabet chars * 4 bytes / 128 = ~ 400 MB
Total ≈ 1.5 GB
~0.5 bytes per input char

For the same data, a suffix tree takes > 45 GB, a suffix array with the suffixes > 12 GB.

Dynamic Programming
Change Problem: Given c1,c2,c3,.. min coins to make a given value.
minCoins(m) = min(minCoins(m-c1)+c1,minCoins(m-c2)+c2,minCoins(m-c3)+c3,..)
Manhattan Tourist Problem: From source to sink, collect most attractions
Si,j = max(Si-1,j + edge i-1,j; Si,j-1 + edge i, j-1; Si-1,j-1 + edge i-1,j-1)
Compute the solution: source -> sink
Obtain pathing by backtracking: sink -> source
Traverse horizontal, vertical, or diagonally.

Longest Common Subsequence (LCS)
Doesn't need to be continous, there can be gaps. Basically MTP, O(mn).
Hamming Distance: penalty 1 if nonmatch, 0 else
Levenshtein (edit) distance: # elementary operations to morph one string to another
Needleman-Wunsch
Global alignment as follows:
S ij = max(S i-1, j-1 + 1 if match S i-1, else j-1 – nu, S i-1, j – sigma, S i, j-1 – sigma)
1: match score, nu: mismatch penalty, sigma: indel penalty
Initialize [0,:] and [:,0] with -nu*index, start 0,0 = 0.

Affine Gaps
F i,j down = max (F i-1,j – sigma, G i-1,j – (ro+sigma))
E ij right = max(E i,j-1 – sigma, G i, j-1 – (ro+sigma)
G i,j = max(G i-1,j-1 + delta, F i,j, E i,j), delta = +nu if match, -nu if mismatch
Final score = max(G i,j, F i,j, E i,j)
F, E tracks possible gaps in down and right directions. Affine gaps come from biological observations suggesting consecutive gaps should be penalized less than individual ones.

Smith-Waterman Local Alignment
Find longest path among arbitrary vertices.
S i,j = max(0, S i-1,j-1 + delta, S i-1,j + sigma, S i,j-1 + sigma), delta = +nu if match, -nu if mismatch.
With zero, we restart alignment. To retrieve the alignment, search for the highest score in DP table, backtrack until you get to a zero.

Substring Problem: Find occurance of every query in a dictionary upon inputing a text t.
Construct a generalized suffix tree based on D with unique identifiers, which can be done in linear time. Thread the string and retrieve the identifiers that are children all the leaves that are a descendant of the destination. This traversal is linear in size of the dictionary. In total, thread and traversal is quadratic.

DNA Contamination Problem: Given strings S1, S2 and positive integer i, find all substrings of S1 longer than l that is a substring of S2.
Create all j-suffixes of S1 where j > l and create a suffix tree T over this set. Thread S2 over T and report all matches.
Listing suffixes of S1 and creating T are linear in |S1|.Threading S2 is linear in |S2|, overall O(m+n).

Longest Common Substring of Multiple Strings: Let S1,..,Sn be strings of total length N. Find the longest common substring between all such Si.
Construct a generalized suffix tree T over Si using n unique identifies.
Traverse the tree post-order, collecting the unique identifiers in its children and the edgepath; as well as the node with largest unique identifiers and its edgepath. Report this edgepath at the end.
Constructing T is O(N), traversal is O(N). Overall, O(N).

Matching Statistics: Length of the longest subtring of T starting at i which matches a substring of P.
If P occurs at i of T, ms(i) = |P| = p
For each position of i, thread T[i..n] over suffix tree S created on P. If target node is a leaf, ms(i) = length of this suffix. Else, collect backwards edgepath and set ms(i) to be the its length.
Constructing suffix tree is O(n), treading each i is O(m), for n times. Overall, O(nm).

Maximal Pair Problem: For S, a and b are maximal strings if they are identical and cannot be extended to either directons without being unequal. Compute the MP for given i,j.
Construct suffix tree T on S. Traverse the tree post-order. For each node, let p be its edgepath. If all children are leaves, MP(node) = (c1,c2,p) where :
c1 = |S| - the shortest child - |p|
c1 = |S| - the longest child - |p|.
If no children, MP(node) = (none, none, 0)
Otherwise, invoke the function on each children with updated edgepaths and let Cx be child with longest px. Then, MP(node) = MP(Cx). Report the node with largest MP.
Linear in construction of suffix tree and traversal.

Circular String Problem: For S, choose an index i where S[i:n] + s[1:i] has the smallest lexical order.
Suffix tree for SS$, $ has largest lexical value.
Traverse the tree until you reach an edgepath of length n where at each step, choose the path with smallest lexical order.
Suffix tree is O(n), traversal is O(n), O(n) overall.

Longest Common Extension(LCE):
Let S1,S2 be strings of total length N. For (i,j) indices, find longest prefix of S1[i:n] which is also a prefix of S2[j:m].
Generalized suffix tree for S1, S2.
Thread S1, post-order traverse starting from this node. If $,# both in children report edgepath. Else, backtrack towards root until $,# both in children. Do another traversal to find largest such edgepath.
Suffix tree linear, traversal linear.

Maximal Palindromes: Find the longest palindrome in given string S.
Can be solved with DP or suffix trees. If DP, base cases are length 1 and 2. General case check if previous string is a palindrome and chars at the end are equal.
With suffix trees, construct a generalized suffix tree for S and S-reverse. Traverse the tree to collect depth of the nodes what have $,# as children. Report the node having the largest depth.
Quadratic since DP.

Is Subsequence: Check if P is a subsequence of T.
In linear time, keep two pointers, one to T and one to P. Increament both if T[i] = P[j], otherwise increment i. Stop if i or j are out of bounds. Return True if j = |p|.

If there are wildcards in P, assume they are correct matches. If there are wildcards in T, they are incorrect matches.

K-Mismatch Problem: Given pattern P and text T, find the |p|-length substring of T that has at most k-mismatches with P.
For all indices of T, check subsequence match with T[i:|p|] and P, counting mismatches. If mismatches > k, increment i.

Also, keep track of start and end characters and mismatch counts. At each step, you don't need to recompute the mismatches in the middle.

Cooptimal Alignments: Find cooptimal local/global alignments for S1 of length n and S2 of length m.
Let Smax be largest value in all directions. Sum each direction having equal weight to Smax, set the current weight to this sum. Repeat for all i,j report n,m.

Inexact Repeated Substring with Local Alignment: Construct local alignment table with diagonals having -infinity weight.
Tandem Repeats: Let Pm be the string where each character of P is repeated m times. Find the local alignment between Pm and T.
Let P be listed row-wise and T column-wise. Compute the local alignment for each column. Afterwards, set the first element of the column to the last element and repeat column computation. Repeat for all columns. When reporting optimal path, if you reach column head (position 0) go to the end of the column and keep backtracking.

Interwoven Strings: Let S1,S2, S3 be strings where |S3| = |S1| + |S2|. Check if S3 can be created by interwoving S1 and S2.
The idea is to check at all times whether character S3[i+j] is equal to either S1[i] or S2[j], and increment one of the indices i or j. Quadratic with DP.

Character-Repeated Subsequence: For P, T find the largest k such that Pk is a subsequence of T where Pk is constructed by repeating each character of P k times.
The idea is to keep a lookup table for each character of P and count the number of consecutive occurrences of each character. Afterwards, the smallest such count gives the largest repeat k.

Motif Finding Problem
Given a set of DNA sequences T find the pattern, namely motif, that is implanted in each of the invidual sequences.
Assume start of each motif is Si. Lineup the patterns and construct a profile counting frequencies of character occurences. Concensus has the highest score in each column.

```
Alignment   a G g t a c T t
            C c A t a c g t
            a c g t T A g t
            a c g t C c A t
            C c g t a c g G

Profile     A 3 0 1 0 3 1 1 0
            C 2 4 0 0 1 4 0 0
            G 0 1 4 0 0 0 3 1
            T 0 0 0 5 1 0 1 4

Consensus   A C G T A C G T
```

- Line up the patterns by their start indexes

$$\mathbf{s} = (s_1, s_2, \ldots, s_t)$$

- Construct profile matrix with frequencies of each nucleotide in columns
- Consensus nucleotide in each position has the highest score in column

Scoring Concensus

```
        l
DNA:  a G g t a c T t
      C c A t a c g t
      a c g t T A g t
      a c g t C c A t
      C c g t a c g G   t

      A 3 0 1 0 3 1 1 0
      C 2 4 0 0 1 4 0 0
      G 0 1 4 0 0 0 3 1
      T 0 0 0 5 1 0 1 4

Consensus  a c g t a c g t

Score  3+4+4+5+3+4+3+4=30
```

Formal definition: Given a set of DNA sequences, find *l*-mer that maximizes concensus score.
Brute-force solution is exponential: compare every *l*-mer for every position in the input set.

Median String Problem: Given a set of t sequences find the pattern that appears in all sequences with minimal errors.
Hamming Distance (HD): Number of characters that mismatch.
HD(AAAAAA, ACAAAC) = 2

Total Distance: Given pattern v, for each sequence i, compute the distance d_si of v to l-mer starting at position si and find the *l*-mer with minimal distance. Total distance of pattern v to input DNA sequences is the sum of all minimal distances computed previously:
TotalDistance(v,DNA) = Sum min_s HD(v,s), for all s in DNA

Solution of Median String Problem minimizes TotalDistance(v,DNA).

Motif Finding is a maximization problem while Median String is a minimization. They are equivalent.

Motif Finding examines all index combinations while Median String examines all string combinations which is significantly smaller.

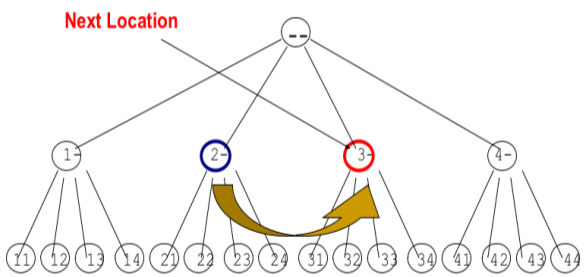
Search for l-mers among all string combinations can be optimized by **search trees.**

Sequences are contained in the *leaves* and parent of a node is the *prefix* of its children.

<u>NextLeaf:</u> Addition in base-(seq_length – extend + 1) + 1, excluding 0.
N=5, l=2, base-5 with no zeros indices.

<u>AllLeaves:</u> Visit all permutations in ascending order: 11, 12, 13, 14, 21, 22, 23, 24, 31, 32, 33, 34, 41, 42, 43, 44.
<u>NextVertex:</u> Location after k-vertex moves. Can be done with by Depth-First Search(DFS).
<u>Bypass:</u> given a prefix, skip all its children and find the next vertex in sequence. Increment most-significant-digit, set least-significant-digit to empty.

- Bypassing the descendants of "2-":

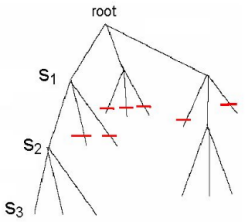


- Sets of $\mathbf{s}=(s_1, s_2, \ldots, s_t)$ may have a weak profile for the first *i* positions $(s_1, s_2, \ldots, s_i)$
- Every row of alignment may add at most ℓ to Score
- <u>Optimism</u>: if all subsequent *(t-i)* positions $(s_{i+1}, \ldots s_t)$ add

$$(t - i) * \ell \text{ to Score}(\mathbf{s}, i, DNA)$$

- If Score(**s**,*i*,*DNA*) + $(t - i) * \ell$ < **BestScore**, it makes no sense to search in vertices of the current subtree
  - Use **ByPass()**

- Since each level of the tree goes deeper into search, discarding a prefix discards all following branches



- This saves us from looking at $(n - \ell + 1)^{t-i}$ leaves
  - Use **NextVertex()** and **ByPass()** to navigate the tree

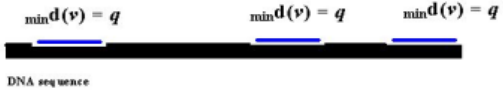
```
BranchAndBoundMotifSearch(DNA,t,n,ℓ)
s ← (1,...,1)
bestScore ← 0
i ← 1
while i > 0
    if i < t
        optimisticScore ← Score(s, i, DNA) +(t – i ) * ℓ
        if optimisticScore < bestScore
            (s, i) ← Bypass(s,i, n-ℓ+1)
        else
            (s, i) ← NextVertex(s, i, n-ℓ+1)
    else
        if Score(s,DNA) > bestScore
            bestScore ← Score(s)
            bestMotif ← (s1, s2, s3, ..., si)
            (s,i) ← NextVertex(s,i,t,n-ℓ+ 1)
return bestMotif
```

Nucleotides can be numbered 1,2,3,4 arbitrarily to organize the search space of 4^l *l*-mers, as if searching in base-5 except 0.

Remember that if TotalDistance(prefix) > Best Distance, bypass the branch.

Given an l-mer, divide it into a prefix u and a suffix v. Compute the minimum distance of u to the sequence. No such prefix u can have a distance more than the minimum. But, this doesn't suggest u is part of any motif.
If we repeat this for v as well, we can ascertain that since u and w are substrings, minimal distance of w must be more than the sum for u and v.

Searching for prefix V
We may find many instances of prefix V with a minimum distance q



Likewise for U

But for U and V combined, U is not at its minimum distance location, neither is V

$_{min}d(\ q + 1 \quad z + 2\ )$

But at least we know w (prefix u suffix v) cannot have distance *less* than $_{min}d(v)$ + $_{min}d(u)$

This means that if *d(prefix)* + *d(suffix)* >= *best_distance*, we can bypass the node.

Implementation can be done with linked lists, but traversal becomes expensive.

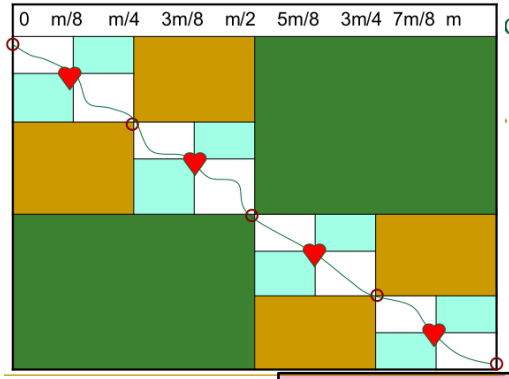
Better Bounded Median String Search

```
ImprovedBranchAndBoundMedianString(DNA,t,n,ℓ)
s = (1, 1, ..., 1)
bestdistance = ∞
i = 1
while i > 0
    if i < ℓ
        prefix = nucleotide string corresponding to (s1, s2, s3, ..., si )
        optimisticPrefixDistance = TotalDistance (prefix, DNA)
        if (optimisticPrefixDistance < bestsubstring[ i ])
            bestsubstring[ i ] = optimisticPrefixDistance
            if (ℓ- i < i )
                optimisticSufxDistance = bestsubstring[ℓ-i ]
        else
            optimisticSufxDistance = 0;
        if optimisticPrefixDistance + optimisticSufxDistance ≥ bestDistance
            (s, i ) = Bypass(s, i, ℓ, 4)
        else
            (s, i ) = NextVertex(s, i, ℓ,4)
    else
        word = nucleotide string corresponding to (s1,s2, s3, ..., st)
        if TotalDistance( word, DNA) < bestDistance
            bestDistance = TotalDistance(word, DNA)
            bestWord = word
            (s,i) = NextVertex(s, i,ℓ, 4)
return bestWord
```

Banded Alignment: - Ukkonen's Algorithm: If we limit the gaps in the word to q where q << n and q << m, we need a DP table having 2q+1-length window around the diagonal, no need to full the rest of the cells.Then, runtime becomes linear in length of the text since constant number of table elements for each index of T is required.

Divide-and-Conquer Approach to Alignment: Alignment score can be obtained in Linear Space-Complexity. To compute it, we only need the previous and the current columns, not the entire table. By determining a middle 'vertex' in the alignment path, we can simplify the problem into smaller ones.

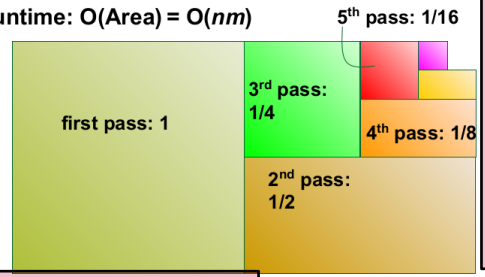
Let length(i) be the length of the longest path that passes through (i, m/2). Define mid as the point the longest alignment path crosses the middle column.
Assume prefix(i) is the length of the longest path from (0,0) to (i,m/2). Compute this using DP.
Similarly, let suffix(i) be the length of the longest path from (i,m/2) to (m,n), and compute it using DP in reverse order.
Let length(i) = prefix(i) + suffix(i). *mid* becomes the value i which maximizes this quantity.



Geometric Reduction At Each Iteration

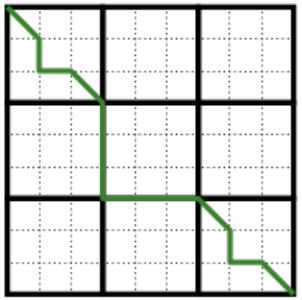
$1 + ½ + ¼ + \ldots + (½)^k \leq 2$

Runtime: O(Area) = O(*nm*)

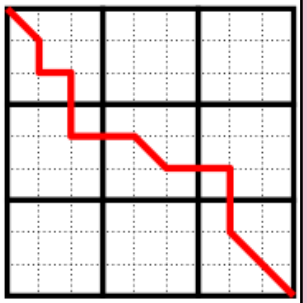


Block Alignment: Partition the DP table into txt blocks such that alignment problem transformed into (n/t)^2 sequence to sequence alignment problems. Global alignment has to follow the edges of these blocks, cannot go through them.

Block alignment of two sequences u, v aligns the entire block of u to the entirety of v: gaps insert entire blocks and backtrack pass through their corners.



valid



invalid

To solve it, compute alignment score for each pair of blocks. There are (n/t)^2 blocks, each is a mini-alignment problem of size txt.
Runtime: O((n/t) x (n/2) * t * t) = O(n^2)