



SORTING + STRING

COMP 321 – McGill University

These slides are mainly compiled from the following resources.

- Professor Jaehyun Park' slides CS 97SI
- Top-coder tutorials.
- Programming Challenges book.

SORTING

- Practical applications in computing require things to be in order.
- To consider:
 - Runtime.
 - Memory Space.
 - In-place algorithms ???
 - Stability.
 - What happens to elements that are comparatively the same?

SORTING

- Practical applications in computing require things to be in order.
- To consider:
 - Runtime.
 - Memory Space.
 - In-place algorithms => without creating copies of the data
 - Stability.
 - What happens to elements that are comparatively the same?
 - Those elements whose comparison key is the same will remain in the same relative order after sorting as they were before sorting.

Bubble Sort

- To pass through the data and swap two adjacent elements whenever the first is greater than the last. Thus, the smallest elements will “bubble” to the surface.
- $O(n^2)$.
- Simple to understand and code from memory + Stable + In-place.

```
for (int i = 0; i < data.Length; i++)
    for (int j = 0; j < data.Length - 1; j++)
        if (data[j] > data[j + 1])
        {
            tmp = data[j];
            data[j] = data[j + 1];
            data[j + 1] = tmp;
        }
```

Insertion Sort

- It seeks to sort a list one element at a time. With each iteration, it takes the next element waiting to be sorted, and adds it, in proper location, to those elements that have already been sorted.

- $O(n^2)$.

- it works very efficiently for lists that are nearly sorted initially

```
for (int i = 0; i <= data.Length; i++) {
    int j = i;
    while (j > 0 && data[i] < data[j - 1])
        j--;
    int tmp = data[i];
    for (int k = i; k > j; k--)
        data[k] = data[k - 1];
    data[j] = tmp;
}
```

Merge Sort

- A merge sort works recursively (divide and conquer). Divide the unsorted list into n sublists, each containing 1 element. Then, merge sublists to produce new sorted sublists.
- $O(n \log n)$.
- Fairly efficient + can be used to solve other problems.

```
int[] mergeSort (int[] data) {
    if (data.Length == 1)
        return data;
    int middle = data.Length / 2;
    int[] left = mergeSort(subArray(data, 0, middle - 1));
    int[] right = mergeSort(subArray(data, middle, data.Length - 1));
    int[] result = new int[data.Length];
```

Heap Sort

- All data from a list is inserted into a heap, and then the root element is repeatedly removed and stored back into the list.
- $O(n \log n)$
- Not stable

```
Heap h = new Heap();
for (int i = 0; i < data.Length; i++)
    h.Add(data[i]);
int[] result = new int[data.Length];
for (int i = 0; i < data.Length; i++)
    data[i] = h.RemoveLowest();
```

Quick Sort

- Divide the data into two groups of “high” values and “low” values. Then, recursively process the two halves. Finally, reassemble the now sorted list.

- $O(n^2)$

• dependent upon how successfully an accurate midpoint value is selected

```
Array quickSort(Array data) {
    if (Array.Length <= 1)
        return;
    middle = Array[Array.Length / 2];
    Array left = new Array();
    Array right = new Array();
    for (int i = 0; i < Array.Length; i++)
        if (i != Array.Length / 2) {
            if (Array[i] <= middle)
                left.Add(Array[i]);
            else
                right.Add(Array[i]);
        }
    return concatenate(quickSort(left), middle, quickSort(right));
}
```


Radix Sort

- Sort data without having to directly compare elements to each other. It groups keys by the individual digits which share the same significant position and value.
- $O(n * k)$, where k is the size of the key.
- Some types of data may use very long keys, or may not easily lend itself to a representation that can be processed

Sorting Libraries

- Java API, and C++ STL all provide some built-in sorting capabilities.
- Check the interface called Comparable => you add a method `int compareTo (object other)`, which returns a negative value if less than, 0 if equal to, or a positive value if greater than the parameter.
- Also check the interface called Comparator. which defines a single method `int Compare (object obj1, object obj2)`, which returns a value indicating the results of comparing the two parameters.



STRINGS

String Matching Problem

- Given a text T and a pattern P , find all occurrences of P within T
- Notations:
 - n and m : lengths of P and T
 - Σ : set of alphabets (of constant size)
 - P_i : i th letter of P (1-indexed)
 - a, b, c : single letters in Σ
 - x, y, z : strings

String Matching Problem

- $T = \text{AGCATGCTGCAGTCATGCTTAGGCTA}$
- $P = \text{GCT}$
- P appears three times in T
- A naive method takes $O(mn)$ time
 - Initiate string comparison at every starting point
 - Each comparison takes $O(m)$ time
- We can do much better!

String Matching Problem - Hash

- Main idea: preprocess T to speedup queries
 - Hash every substring of length k
 - k is a small constant
- For each query P , hash the first k letters of P to retrieve all the occurrences of it within T
- Don't forget to check collisions!

String Matching Problem - Hash

- Pros:
 - Easy to implement
 - Significant speedup in practice
- Cons:
 - Doesn't help the asymptotic efficiency
 - Can still take $O(nm)$ time if hashing is terrible or data is difficult
 - Can you give me an example of the worst case?
- A lot of memory consumption

String Matching Problem - Hash

- Pros:
 - Easy to implement
 - Significant speedup in practice
- Cons:
 - Doesn't help the asymptotic efficiency
 - Can still take $O(nm)$ time if hashing is terrible or data is difficult
 - Can you give me an example of the worst case? => When all the characters of pattern and text are same. $T=AAAAAAA\dots$ $P=AAA$.
- A lot of memory consumption

SMP - Knuth-Morris-Pratt (KMP)

- A linear time (!) algorithm that solves the string matching problem by preprocessing P in $O(m)$ time
 - Main idea is to skip some comparisons by using the previous comparison result.
- Uses an auxiliary array π that is defined as the following:
 - $\pi[i]$ is the largest integer smaller than i such that $P_1 \dots P_{\pi[i]}$ is a suffix of $P_1 \dots P_i$
 - e.g., $\pi[6] = 4$ since abab is a suffix of ababab
 - e.g., $\pi[9] = 0$ since no prefix of length ≤ 8 ends with c

i	1	2	3	4	5	6	7	8	9	10
P_i	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1



Question for you

- Why is π useful?

SMP - Knuth-Morris-Pratt (KMP)

- T = ABC ABCDAB ABCDABCDABDE
- P = ABCDABD
- $\pi = (0,0,0,0,1,2,0)$
- Start matching at the first position of T:

12345678901234567890123
ABC ABCDAB ABCDABCDABDE
ABCD**ABD**
1234567

- Mismatch at the 4th letter of P!

SMP - Knuth-Morris-Pratt (KMP)

- We matched $k = 3$ letters so far, and $\pi[k] = 0$
 - Thus, there is no point in starting the comparison at T_2 , T_3
- Shift P by $k - \pi[k] = 3$ letters

12345678901234567890123
ABC ABCDAB ABCDABCDABDE
ABCDABD
1234567

- Mismatch at the 4th letter of P !

SMP - Knuth-Morris-Pratt (KMP)

- We matched $k = 0$ letters so far
- Shift P by $k - \pi[k] = 1$ letter (we define $\pi[0] = -1$)

12345678901234567890123
ABC ABCDAB ABCDABCDABDE
ABCDABD
1234567

- Mismatch at i_{11} !

SMP - Knuth-Morris-Pratt (KMP)

- $\pi[6] = 2$ means P_1P_2 is a suffix of $P_1 \dots P_6$
- Shift P by $6 - \pi[6] = 4$ letters

12345678901234567890123
ABC ABCDAB ABCDABCDABDE
ABCDABD

||
ABCDABD
1234567

- Again, no point in shifting P by 1, 2, or 3 letters

SMP - Knuth-Morris-Pratt (KMP)

- Mismatch at T_{11} again!

12345678901234567890123
ABC ABCDAB ABCDABCDABDE
ABCDABD
1234567

- Currently 2 letters are matched
- Shift P by $2 - \pi[2] = 2$ letters

SMP - Knuth-Morris-Pratt (KMP)

- Mismatch at T_{11} again!

1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3

ABC ABCDAB ABCDABCDABDE

ABCDABD

1 2 3 4 5 6 7

- Currently no letters are matched
- Shift P by 0 – $\pi[0] = 1$ letter

SMP - Knuth-Morris-Pratt (KMP)

- Mismatch at T18

1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3

ABC ABCDAB ABCDABCDABDE

ABCDABD

1 2 3 4 5 6 7

- Currently 6 letters are matched
- Shift P by 6 - $\pi[6]$ = 4 letters

SMP - Knuth-Morris-Pratt (KMP)

- Finally, there it is!

12345678901234567890123
ABC ABCDAB ABCDABCDABDE
ABCDABD
1234567

- After recording this match (at T16 . . . T22, we shift P again in order to find other matches
 - Shift by $7 - \pi[7] = 7$ letters

SMP - Knuth-Morris-Pratt (KMP)

- Computing π .
- Obs1 \Rightarrow if $P_1 \dots P_{\pi[i]}$ is a suffix of $P_1 \dots P_i$, then $P_1 \dots P_{\pi[i]-1}$ is a suffix of $P_1 \dots P_{i-1}$
- Obs2 \Rightarrow all the prefixes of P that are a suffix of $P_1 \dots P_i$ can be obtained by recursively applying to i
 - e.g., $P_1 \dots P_{\pi[i]}$, $P_1 \dots$, $P_{\pi[\pi[i]]}$, $P_1 \dots$, $P_{\pi[\pi[\pi[i]]]}$ are all suffixes of $P_1 \dots P_i$

SMP - Knuth-Morris-Pratt (KMP)

- Computing π .
- Obs3 (not obvious) =>
 - First, let's write $\pi^{(k)}[i]$ as $\pi[\cdot]$ applied k times to i
 - e.g., $\pi^{(2)}[i] = \pi[\pi[i]]$
 - $\pi[i]$ is equal to $\pi^{(k)}[i - 1] + 1$, where k is the smallest integer that satisfies $P_{\pi^{(k)}[i-1]+1} = P_i$
 - If there is no such k , $\pi[i] = 0$
- Intuition: we look at all the prefixes of P that are suffixes of $P_1 \dots P_{i-1}$, and find the longest one whose next letter matches P_i

SMP - Knuth-Morris-Pratt (KMP)

- Implementation π .

```
pi[0] = -1;
int k = -1;
for(int i = 1; i <= m; i++) {
    while(k >= 0 && P[k+1] != P[i])
        k = pi[k];
    pi[i] = ++k;
}
```

SMP - Knuth-Morris-Pratt (KMP)

- Implementation KMP.

```
int k = 0;
for(int i = 1; i <= n; i++) {
    while(k >= 0 && P[k+1] != T[i])
        k = pi[k];
    k++;
    if(k == m) {
        // P matches T[i-m+1..i]
        k = pi[k];
    }
}
```

SMP – Suffix trie

- Suffix trie of a string T is a rooted tree that stores all the suffixes (thus all the substrings)
- Each node corresponds to some substring of T
- Each edge is associated with an alphabet
- For each node that corresponds to ax , there is a special pointer called suffix link that leads to the node corresponding to x
- Surprisingly easy to implement!

Applications of Suffix Tries (1)

Check whether q is a **substring** of T :

Follow the path for q starting from the root.
If you exhaust the query string, then q is in T .

Check whether q is a **suffix** of T :

Follow the path for q starting from the root.
If you end at a leaf at the end of q , then q is a suffix of T .

Count # of occurrences of q in T :

Follow the path for q starting from the root.
The number of leaves under the node you end up in is the number of occurrences of q .

Find the longest repeat in T :

Find the deepest node that has at least 2 leaves under it.

Find the lexicographically (alphabetically) first suffix:

Start at the root, and follow the edge labeled with the lexicographically (alphabetically) smallest letter.

Suppose we want to build suffix trie for string:

$s = \text{abbacabaa}$

We will walk down the string from left to right:

abbacabaa
→

building suffix tries for $s[0], s[0..1], s[0..2], \dots, s[0..n]$

⏟
To build suffix trie for $s[0..i]$, we will use the suffix trie for $s[0..i-1]$ built in previous step

To convert $\text{SufTrie}(S[0..i-1]) \rightarrow \text{SufTrie}(s[0..i])$, add character $s[i]$ to all the suffixes:

abbacabaa
 $i=4$

Need to add nodes for the suffixes:

abbac
bbac
bac
ac
c

Purple are suffixes that will exist in $\text{SufTrie}(s[0..i-1])$ Why?

How can we find these suffixes quickly?

To build SufTrie(s[0..i]) from SufTrie(s[0..i-1]):

CurrentSuffix = longest (aka deepest suffix)

Repeat:

Add child labeled s[i] to CurrentSuffix.

Follow suffix link to set CurrentSuffix to next shortest suffix.

until you reach the root or the current node already has an edge labeled s[i] leaving it.

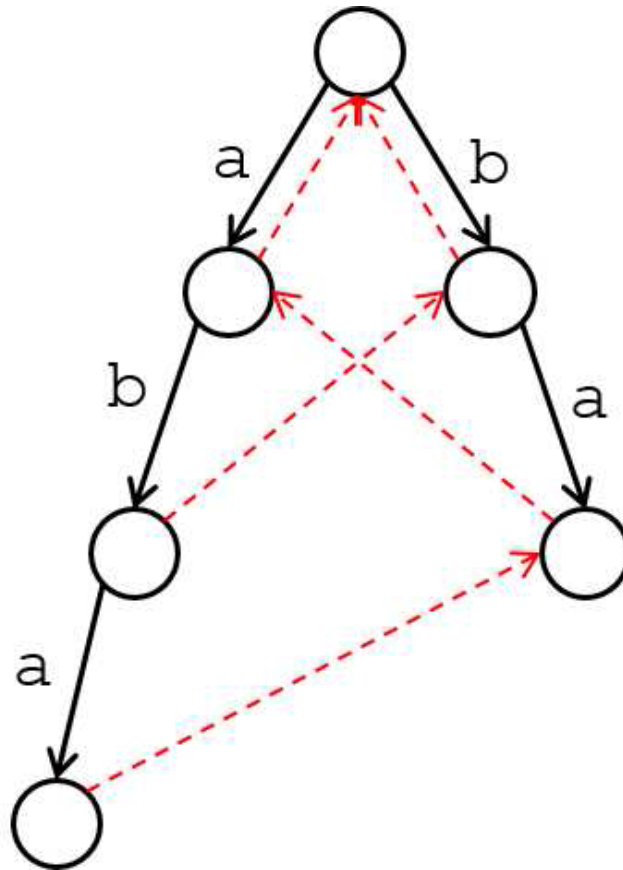
Add suffix links connecting nodes you just added in the order in which you added them.

Because if you already have a node for suffix $\alpha s[i]$ then you have a node for every smaller suffix.

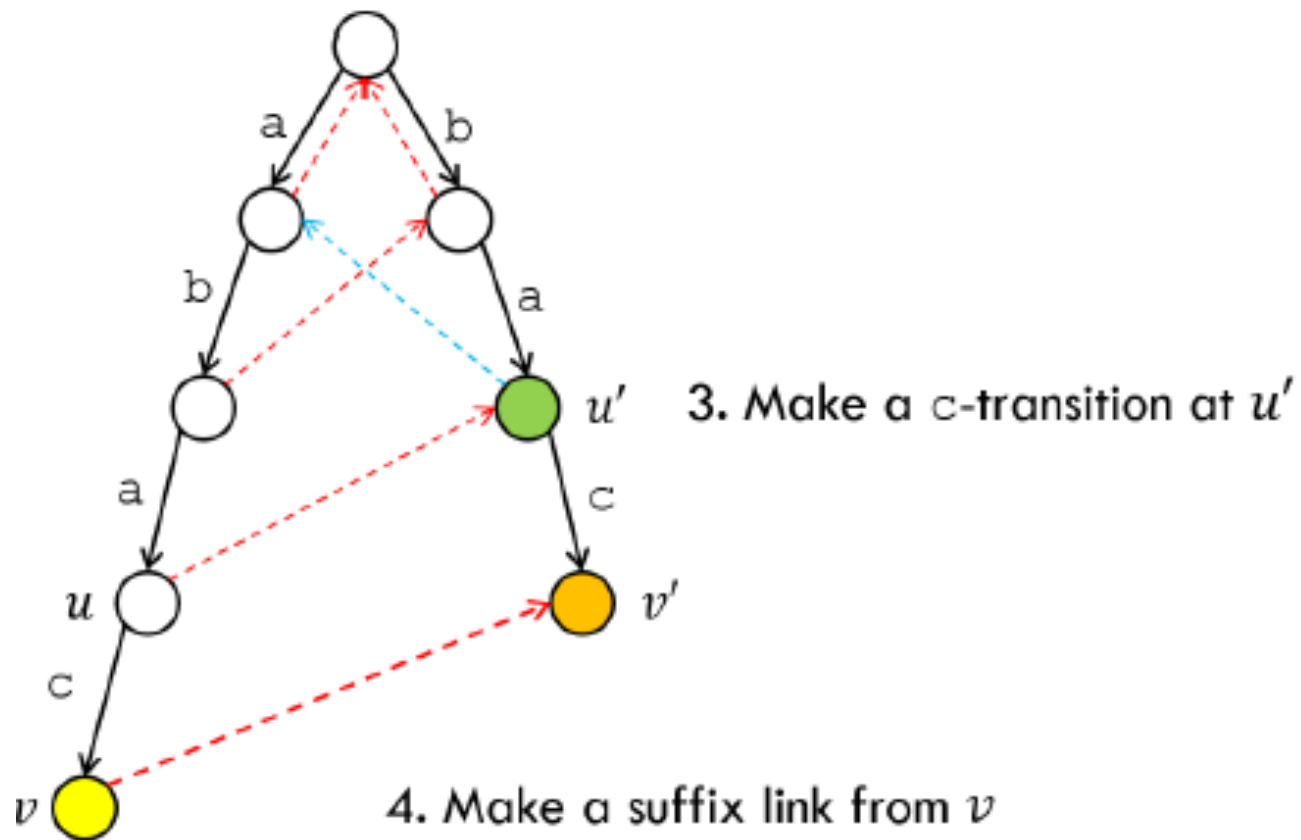
In practice, you add these links as you go along, rather than at the end.

SMP – Suffix trie

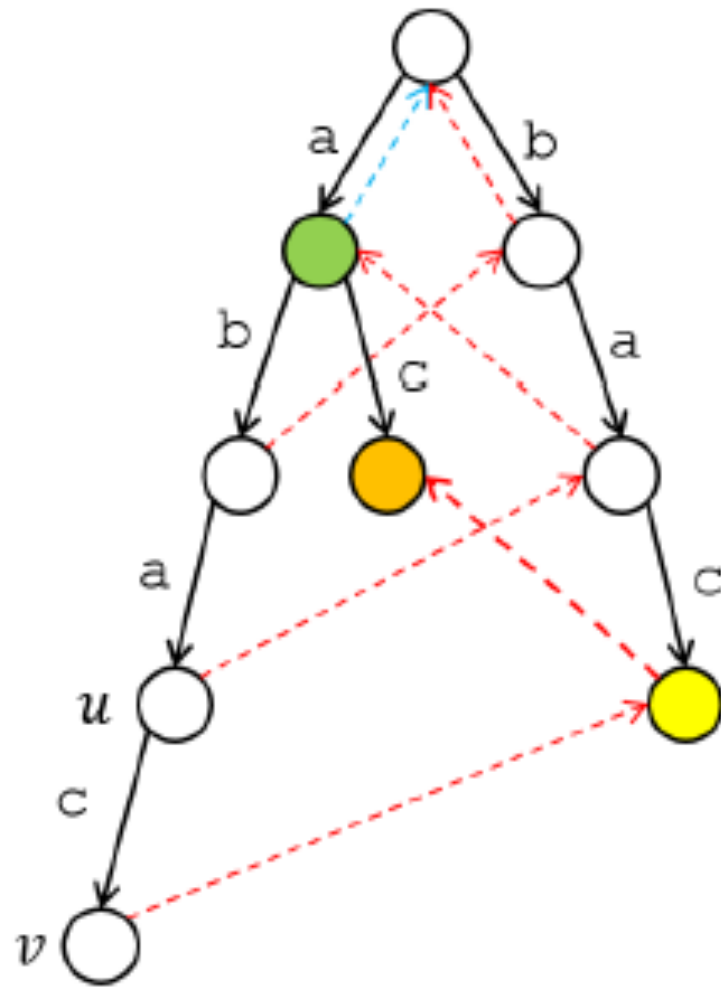
- Given the suffix trie for aba, we want to add a new letter c



SMP – Suffix trie



SMP – Suffix trie



SMP-Suffix Array

Input string	Get all suffixes	Sort the suffixes	Take the indices
BANANA	1 BANANA	6 A	6, 4, 2, 1, 5, 3
	2 ANANA	4 ANA	
	3 NANA	2 ANANA	
	4 ANA	1 BANANA	
	5 NA	5 NA	
	6 A	3 NANA	

SMP – Suffix Array

- Memory usage is $O(n)$
- Has the same computational power as suffix trie
- Can be constructed in $O(n)$ time (!)
 - But it's hard to implement

Notes

- Always be aware of the null-terminators
- Simple hash works so well in many problems
- If a problem involves rotations of some string, consider concatenating it with itself and see if it helps
- It is a smart idea to have the implementation of suffix arrays and KMP in your notebook.