

Tablice sufiksów i drzewa sufiksów

dr inż. Marcin Ciura

mgc@agh.edu.pl

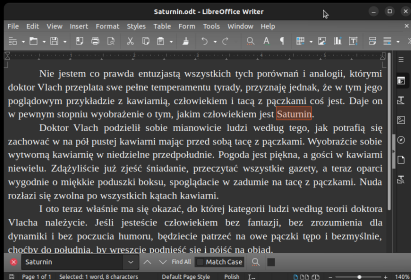
Wydział Informatyki, Akademia Górniczo-Hutnicza

Plan wykładu

- Tablice sufiksów
- Zastosowania tablic sufiksów
- Drzewa sufiksów
- Zastosowania drzew sufiksów
- Algorytm Ukkonena budowania drzew sufiksów

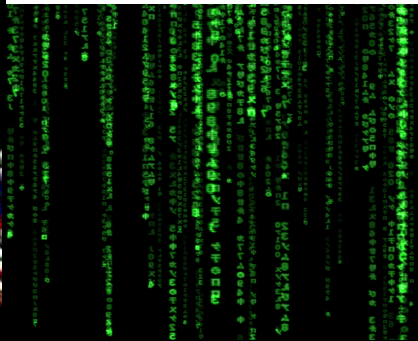
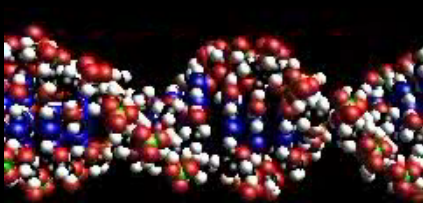
**Po co nam tablice sufiksów i drzewa
sufiksów?**

Żeby wyszukiwać wzorce w tekstach, i nie tylko po to :-)



Mozart
Sonata in E Minor, K. 304

The image shows the first page of the musical score for Mozart's Sonata in E Minor, K. 304. The score is for Violin and Piano. The tempo is marked 'Allegro'. The key signature is one sharp (F#). The score includes staves for both instruments, with various musical notations such as notes, rests, and dynamic markings like 'p' (piano) and 'f' (forte).



Prehistoria

AARON'S

2

Aaron will seek.
 16. 27. 28. M. and A. a. w. up. after
 29. 30. A. shall put up upon after
 31. 32. A. shall put up upon after
 33. 34. A. shall put up upon after
 35. 36. A. shall put up upon after
 37. 38. A. shall put up upon after
 39. 40. A. shall put up upon after
 41. 42. A. shall put up upon after
 43. 44. A. shall put up upon after
 45. 46. A. shall put up upon after
 47. 48. A. shall put up upon after
 49. 50. A. shall put up upon after
 51. 52. A. shall put up upon after
 53. 54. A. shall put up upon after
 55. 56. A. shall put up upon after
 57. 58. A. shall put up upon after
 59. 60. A. shall put up upon after
 61. 62. A. shall put up upon after
 63. 64. A. shall put up upon after
 65. 66. A. shall put up upon after
 67. 68. A. shall put up upon after
 69. 70. A. shall put up upon after
 71. 72. A. shall put up upon after
 73. 74. A. shall put up upon after
 75. 76. A. shall put up upon after
 77. 78. A. shall put up upon after
 79. 80. A. shall put up upon after
 81. 82. A. shall put up upon after
 83. 84. A. shall put up upon after
 85. 86. A. shall put up upon after
 87. 88. A. shall put up upon after
 89. 90. A. shall put up upon after
 91. 92. A. shall put up upon after
 93. 94. A. shall put up upon after
 95. 96. A. shall put up upon after
 97. 98. A. shall put up upon after
 99. 100. A. shall put up upon after

Pr. 33. 34. Blood and A. as before III.
 35. 36. Blood and A. as before III.
 37. 38. Blood and A. as before III.
 39. 40. Blood and A. as before III.
 41. 42. Blood and A. as before III.
 43. 44. Blood and A. as before III.
 45. 46. Blood and A. as before III.
 47. 48. Blood and A. as before III.
 49. 50. Blood and A. as before III.
 51. 52. Blood and A. as before III.
 53. 54. Blood and A. as before III.
 55. 56. Blood and A. as before III.
 57. 58. Blood and A. as before III.
 59. 60. Blood and A. as before III.
 61. 62. Blood and A. as before III.
 63. 64. Blood and A. as before III.
 65. 66. Blood and A. as before III.
 67. 68. Blood and A. as before III.
 69. 70. Blood and A. as before III.
 71. 72. Blood and A. as before III.
 73. 74. Blood and A. as before III.
 75. 76. Blood and A. as before III.
 77. 78. Blood and A. as before III.
 79. 80. Blood and A. as before III.
 81. 82. Blood and A. as before III.
 83. 84. Blood and A. as before III.
 85. 86. Blood and A. as before III.
 87. 88. Blood and A. as before III.
 89. 90. Blood and A. as before III.
 91. 92. Blood and A. as before III.
 93. 94. Blood and A. as before III.
 95. 96. Blood and A. as before III.
 97. 98. Blood and A. as before III.
 99. 100. Blood and A. as before III.

ABSORBENCE

Da. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. 13. 14. 15. 16. 17. 18. 19. 20. 21. 22. 23. 24. 25. 26. 27. 28. 29. 30. 31. 32. 33. 34. 35. 36. 37. 38. 39. 40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50. 51. 52. 53. 54. 55. 56. 57. 58. 59. 60. 61. 62. 63. 64. 65. 66. 67. 68. 69. 70. 71. 72. 73. 74. 75. 76. 77. 78. 79. 80. 81. 82. 83. 84. 85. 86. 87. 88. 89. 90. 91. 92. 93. 94. 95. 96. 97. 98. 99. 100.

ABHORREST

ABHORREST
 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. 13. 14. 15. 16. 17. 18. 19. 20. 21. 22. 23. 24. 25. 26. 27. 28. 29. 30. 31. 32. 33. 34. 35. 36. 37. 38. 39. 40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50. 51. 52. 53. 54. 55. 56. 57. 58. 59. 60. 61. 62. 63. 64. 65. 66. 67. 68. 69. 70. 71. 72. 73. 74. 75. 76. 77. 78. 79. 80. 81. 82. 83. 84. 85. 86. 87. 88. 89. 90. 91. 92. 93. 94. 95. 96. 97. 98. 99. 100.

3

ABHORREST
 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. 13. 14. 15. 16. 17. 18. 19. 20. 21. 22. 23. 24. 25. 26. 27. 28. 29. 30. 31. 32. 33. 34. 35. 36. 37. 38. 39. 40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50. 51. 52. 53. 54. 55. 56. 57. 58. 59. 60. 61. 62. 63. 64. 65. 66. 67. 68. 69. 70. 71. 72. 73. 74. 75. 76. 77. 78. 79. 80. 81. 82. 83. 84. 85. 86. 87. 88. 89. 90. 91. 92. 93. 94. 95. 96. 97. 98. 99. 100.

ABIHU

ABIHU
 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. 13. 14. 15. 16. 17. 18. 19. 20. 21. 22. 23. 24. 25. 26. 27. 28. 29. 30. 31. 32. 33. 34. 35. 36. 37. 38. 39. 40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50. 51. 52. 53. 54. 55. 56. 57. 58. 59. 60. 61. 62. 63. 64. 65. 66. 67. 68. 69. 70. 71. 72. 73. 74. 75. 76. 77. 78. 79. 80. 81. 82. 83. 84. 85. 86. 87. 88. 89. 90. 91. 92. 93. 94. 95. 96. 97. 98. 99. 100.

- Liang, Franklin Mark, 722.
- Library card sorting, 7-8.
- Liddy, George Gordon, 395.
- LIFO, 148, 299, 305, *see* Stacks.
- Lin, Andrew Damon, 547, 578.
- Lin, Shen (林牲), 188, 195, 202-206.
- Lineal chart, 424.
- Linear algorithm for median, 214-215, 695.
- Linear algorithms for sorting, 5, 102, 176-179, 196, 616.
- Linear arrangements, optimum, 408.
- Linear congruential sequence, 383.
- Linear hashing, 548-549.
- Linear lists, 248, 385, 459, *see also* List sorting.
 representation of, 96-97, 163-164, 471-473, 479-480, 491, 547.
- Linear order, 4, 181.
- Linear probing, 526-528, 533-534, 536-539, 543-544, 547, 548, 551-553, 555, 556.
 optimum, 532.
- Ling, Huei (林輝), 578.
- Linial, Nathen (נתן ליניאל), 660.
- Linked allocation, 74-75, 96, 99-102, 104, 164-165, 170-173, 399, 405, 459, 547.
- Linn, John Charles, 425.
- Lint, Jacobus Hendricus van, 729, 747.
- Lipski, Witold, Jr., 548.
- Lissajous, Jules Antoine, 395.
- List head, 267, 462.
- List insertion sort, 95-98, 104, 380, 382.
- List merge sort, 164-168, 183, 381, 382, 390.
- List sorting, 74-75, 80, 164-168, 171-178, 382, 390, 698.
- Littlewood, Dudley Ernest, 610, 612.
- LSD: Least significant digit, 218.
- Lucas, François Édouard Anatole, 611.
 numbers L_n , 467, 708.
- Luczak, Tomasz Jan, 734.
- Lueker, George Schick, 742.
- Luhn, Hans Peter, 440, 547.
- Lukasiewicz, Jan, 395, 672.
- Lum, Vincent Yu-sun (林耀榮), 520, 578.
- Lynch, William Charles, 682, 709.
- m-d tree, see k-d tree.*
- m-d trie, see k-d trie.*
- Machiavelli, Niccolò di Bernardo, 1.
- MacLaren, Malcolm Donald, 176, 178, 179, 380, 618.
- MacLeod, Iain Donald Graham, 617.
- MacMahon, Percy Alexander, 8, 16-17, 20, 27, 33, 43, 45, 59, 61, 70, 600, 613, 653.
 Master Theorem, 33-34.
- Macro language, 457.
- Magic trick, 370.
- Magnetic tapes, 6-10, 248-251, 267-357, 403-407.
 reliability of, 337.
- Magnus, Wilhelm, 131.
- Mahmoud, Hosam Mahmoud
 (حسام محمود محمود), 713, 721.
- Mahon, Maurice Harlang (= Magenta), xi.
- Maier, David, 477.
- Majewski, Bohdan Stanisław, 513.
- Major index, *see* Index.
- Mallach, Efreim Gershon, 533.
- Mallows, Colin Lingwood, 44, 604, 733.
- Maly, Kurt, 721.
- Manacher, Glenn Keith, 192, 204.

Tablice sufiksów



Izraelski informatyk. W 1990 roku wraz z Genem Myersem wprowadził pojęcie tablicy sufiksów. Był wiceprezesem w Amazonie, Google i YouTube.

Gene Myers (31.12.1953–)



Amerykański informatyk i bioinformatyk. Jest współautorem programu BLAST (basic local alignment search tool), który porównuje łańcuchy w biologicznych bazach danych. Artykuł z 1990 roku, w którym opisano BLAST, był do 2024 roku cytowany ponad 180 tysięcy razy.

Tablica sufixsów zawiera posortowane indeksy tych pozycji w danym łańcuchu T , od których zaczynają się sufixy tego łańcucha. Tak sortuję elementy tablicy sufixsów:

Jeśli pewien sufix S_1 poprzedza inny sufix S_2 w porządku leksykograficznym, to indeks tej pozycji łańcucha T , od której zaczyna się sufix S_1 , znajduje się w tablicy sufixsów łańcucha T przed indeksem tej pozycji łańcucha T , od której zaczyna się sufix S_2

Tablice sufixów – przykład

Tablica sufixów łańcucha **ananas** to **[0 2 4 1 3 5]**

od pozycji **0** łańcucha **ananas** zaczyna się jego sufix **ananas**

od pozycji **2** łańcucha **ananas** zaczyna się jego sufix **anas**

od pozycji **4** łańcucha **ananas** zaczyna się jego sufix **as**

od pozycji **1** łańcucha **ananas** zaczyna się jego sufix **nanas**

od pozycji **3** łańcucha **ananas** zaczyna się jego sufix **nas**

od pozycji **5** łańcucha **ananas** zaczyna się jego sufix **s**

Tablice sufiksów – najprostszy algorytm tworzenia

```
type suffix struct {  
    s []byte  
    n int  
}  
  
func sortedSuffixes(text []byte) []suffix {  
    suffixes := make([]suffix, len(text))  
    for i := range suffixes {  
        suffixes[i].s = text[i:]  
        suffixes[i].n = i  
    }  
    sort.Slice(suffixes, func(i, j int) bool {  
        return bytes.Compare(suffixes[i].s, suffixes[j].s) < 0  
    })  
    return suffixes  
}
```

Tablice sufiksów – najprostszy algorytm tworzenia

```
// Index implementuje indeks łańcucha `text`. Ten indeks  
// składa się z tablicy sufiksów łańcucha `text`  
// i z łańcucha `text`  
type Index struct {  
    suffixes []int  
    text     []byte  
}
```

Tablice sufiksów – najprostszy algorytm tworzenia

// SuffixArray zwraca indeks łańcucha `text`

```
func SuffixArray(text []byte) Index {  
    index := Index{  
        make([]int, len(text)),  
        make([]byte, len(text)),  
    }  
    index.text = text  
    for i, suf := sortedSuffixes(text) {  
        index.suffixes[i] = suf.n  
    }  
    return index  
}
```

Tablice sufiksów – najprostszy algorytm tworzenia

Za pomocą funkcji `SuffixArray` mogę stworzyć tablicę sufiksów łańcucha `text` w czasie $O(|\text{text}|^2 \log |\text{text}|)$

Istnieją szybsze algorytmy tworzenia tablic sufiksów. Najszybsze z tych algorytmów działają w czasie $O(|\text{text}|)$

Tablice sufiksów – zajmowana pamięć

```
// Index implementuje indeks łańcucha `text`. Ten indeks  
// składa się z tablicy sufiksów łańcucha `text`  
// i z łańcucha `text`  
type Index struct {  
    suffixes []int  
    text     []byte  
}
```

Wewnątrz struktury **Index**:

- każdy element tablicy **suffixes** zajmuje 1 słowo maszynowe
- każdy znak łańcucha **text** zajmuje 1 bajt

Zatem tablica sufiksów łańcucha **text** zajmuje $(\text{sizeof}(\text{int})+1)*\text{len}(\text{text})$ bajtów

Zastosowania tablic sufiksów

Zastosowania tablic sufiksów – 1

Tak wyszukuję wszystkie wystąpienia danego wzorca P w danym tekście T :

- Buduję tablicę sufiksów **suffixes** łańcucha T
- Znajduję indeks **i** pierwszego takiego elementu tablicy **suffixes**, który nie poprzedza wzorca T w porządku leksykograficznym
- Znajduję indeks **j** pierwszego takiego elementu tablicy **suffixes**, którego prefiks o długości $|P|$ występuje w porządku leksykograficznym za wzorcem P
- Wycinek **suffixes[i:j]** zawiera indeksy tych pozycji, od których zaczynają się wystąpienia wzorca P w tekście T

```
// Suffix zwraca ten sufiks łańcucha `x.text`, który  
// zaczyna się od `i`-tej pozycji tego łańcucha  
func (x Index) Suffix(i int) []byte {  
    return x.text[i:]  
}
```

Zastosowania tablic sufiksów – 1

```
// LookupAll zwraca wycinek, którego elementami są wszystkie  
// te pozycje, na których łańcuch `s` występuje w łańcuchu `x.text`  
func (x Index) LookupAll(s []byte) []int {  
    i := sort.Search(len(x.suffixes), func(i int) bool {  
        return bytes.Compare(x.Suffix(x.suffixes[i]), s) >= 0  
    })  
    j := i + sort.Search(len(x.suffixes) - i, func(j int) bool {  
        return !bytes.HasPrefix(x.Suffix(x.suffixes[i+j]), s)  
    })  
    return x.suffixes[i:j]  
}
```

Zastosowania tablic sufiksów – 1

Tak wyszukuję wszystkie wystąpienia wzorca **issi** w tekście **mississippi**:

```
10: i
 7: ippi
>4: issippi
>1: ississippi
 0: mississippi
 9: pi
 8: ppi
 6: sippi
 3: sissippi
 5: ssippi
 2: ssissippi
```

Wzorzec **issi** występuje w tekście **mississippi** na pozycjach 4 i 1

Korzystając z tablicy sufiksów, mogę znaleźć wszystkie wystąpienia danego wzorca P w danym tekście T w czasie $O(\log|T|)$ dzięki temu, że w funkcji **sort.Search** zastosowano wyszukiwanie binarne, które działa w czasie $O(\log|T|)$

Zastosowania tablic sufiksów – 2

```
// lenOfCommonPrefix zwraca długość najdłuższego  
// wspólnego prefiksu łańcuchów `s` i `t`  
func lenOfCommonPrefix(s, t []byte) int {  
    k := 0  
    for ; k < min(len(s), len(t)); k++ {  
        if s[k] != t[k] {  
            return k  
        }  
    }  
    return k  
}
```


Zastosowania tablic sufiksów – 2

*// LongestRepeatingSubstring znajduje najdłuższy taki łańcuch,
// który występuje w łańcuchu `text` co~najmniej `k` razy*

```
func LongestRepeatingSubstring(text []byte, k int) []byte {  
    index := SuffixArray(text)  
    substr := []byte{}  
    for i := 0; i + k <= len(a); i++ {  
        m := lenOfCommonPrefix(  
            index.Suffix(i), index.Suffix(i+k-1))  
        if m > len(substr) {  
            substr = index.Suffix(i)[:m]  
        }  
    }  
    return substr  
}
```

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście **mississippi**:

10: i

7: ippi

4: issippi

1: ississippi

0: mississippi

9: pi

8: ppi

6: sippi

3: sissippi

5: ssippi

2: ssissippi

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście **mississippi**:

10: i

7: ippi

4: issippi

1: ississippi

0: mississippi

9: pi

8: ppi

6: sippi

3: sissippi

5: ssippi

2: ssissippi

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście **mississippi**:

10: i

7: **i**ppi

4: **i**ssippi

1: ississippi

0: mississippi

9: pi

8: ppi

6: sippi

3: sissippi

5: ssippi

2: ssissippi

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście **mississippi**:

10: i

7: ippi

4: **issi**ppi

1: **issi**ssippi

0: mississippi

9: pi

8: ppi

6: sippi

3: sissippi

5: ssippi

2: ssissippi

issi

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście
mississippi:

10: i

7: ippi

4: issippi

1: ississippi

0: mississippi

9: pi

8: ppi

6: sippi

3: sissippi

5: ssippi

2: ssissippi

issi

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście

mississippi:

10: i

7: ippi

4: issippi

1: ississippi

0: mississippi

9: pi

8: ppi

6: sippi

3: sissippi

5: ssippi

2: ssissippi

issi

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście

mississippi:

10: i

7: ippi

4: issippi

1: ississippi

0: mississippi

9: pi

8: ppi

6: sippi

3: sissippi

5: ssippi

2: ssissippi

issi

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście
mississippi:

10: i

7: ippi

4: issippi

1: ississippi

0: mississippi

9: pi

8: ppi

6: sippi

3: sissippi

5: ssippi

2: ssissippi

issi

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście

mississippi:

10: i

7: ippi

4: issippi

1: ississippi

0: mississippi

9: pi

8: ppi

6: sippi

3: sissippi

5: ssippi

2: ssissippi

issi

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście

mississippi:

10: i

7: ippi

4: issippi

1: ississippi

0: mississippi

9: pi

8: ppi

6: sippi

3: sissippi

5: ssippi

2: ssissippi

issi

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście

mississippi:

10: i

7: ippi

4: issippi

1: ississippi

0: mississippi

9: pi

8: ppi

6: sippi

3: sissippi

5: ssippi

2: ssissippi

issi

Korzystając z tablic sufiksów mogę znaleźć najdłuższy taki łańcuch, który występuje co najmniej k razy w łańcuchu T , w czasie $O(|T|^2)$

Drzewa sufiksów



Amerykański informatyk i przedsiębiorca. Był promotorem doktoratu Briana Kernighana. Założył Interactive Systems Corporation, pierwszą komercyjną firmę zajmującą się UNIX-em. Wprowadził pojęcie drzewa sufiksów. Jest autorem algorytmu tworzenia drzew sufiksów, który działa w czasie liniowym względem długości łańcucha. Donald Knuth nazwał ten algorytm „algorytmem roku 1973”.

Drzewo trie – takie drzewo wyszukiwania, którego krawędzie są opisane przez fragmenty kluczy wyszukiwania

Nazwa **trie** pochodzi od wyrazu **retrieval**

Nieskompresowane drzewo trie – takie drzewo trie, którego krawędzie są opisane przez pojedyncze znaki

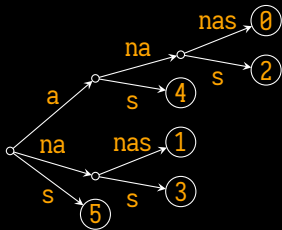
Skompresowane drzewo trie – takie drzewo trie, w którym każdy taki węzeł, z którego wychodzi dokładnie 1 krawędź, został połączony ze swoim rodzicem w 1 węzeł. Krawędzie skompresowanego drzewa trie są opisane przez niepuste podłańcuchy kluczy wyszukiwania

Drzewo sufiksów łańcucha T to takie skompresowane drzewo trie, w którym:

- Każda krawędź między węzłami jest opisana przez pewien niepusty podłańcuch łańcucha T
- Z każdego węzła wewnętrznego wychodzą 2 krawędzie lub więcej krawędzi
- Z żadnego węzła nie wychodzą takie 2 krawędzie, których opis zaczynałby się tym samym znakiem
- Jest $|T|$ liści. Te liście mają numery od 0 do $T - 1$
- Gdy połączę etykiety każdej krawędzi na ścieżce od korzenia drzewa do i -tego liścia tego drzewa, otrzymam sufix $T[i :]$

Drzewa sufiksów – przykład

ananas



Drzewa sufiksów



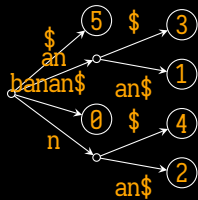
Jeśli pewien sufiks łańcucha T jest prefiksem innego sufiksu łańcucha T , czyli jeśli pewien sufiks łańcucha T jest równy innemu podłańcuchowi łańcucha T , to drzewo sufiksów łańcucha T ma mniej liści niż powinno mieć

Na przykład w drzewie sufiksów łańcucha **banan** brakuje liści 3 i 4 na końcach krawędzi **an** i **n**

W takim przypadku budujemy drzewa sufiksów takich łańcuchów, na których końcu jest dodany **wartownik**. Ten wartownik jest zwykle oznaczany znakiem dolara **\$**

Drzewa sufiksów – przykład

banan\$



Drzewo sufiksów – implementacja

Jak przechowywać węzły drzewa sufiksów? Jest kilka możliwości:

- Tablica wskaźników do węzłów potomnych, która ma tyle elementów, ile alfabet ma znaków
- Lista wskaźników do węzłów potomnych, posortowana według znaków
- Drzewo, na przykład drzewo binarne. Etykietami krawędzi tego drzewa są znaki, a węzły tego drzewa zawierają wskaźniki do węzłów potomnych
- Tablica haszująca znaki na wskaźniki do węzłów potomnych

Drzewo sufiksów – zajmowana pamięć

Przykładowa implementacja węzła drzewa sufiksów:

```
type node struct {  
    start, end, link int  
    next           [256]int  
}
```

- 3 słowa maszynowe – pozycje początku i końca podłańcucha, który opisuje krawędź, oraz **łącze do sufiksu**, o którym będzie mowa później
- 256 słów maszynowych na indeksy węzłów potomnych
- bajty na łańcuch **text**
- W drzewie sufiksowym jest co najwyżej **$2 * \text{len}(\text{text})$** węzłów

Zatem drzewo sufiksów zajmuje co najwyżej

$(259 * \text{sizeof}(\text{int}) + 1) * \text{len}(\text{text})$ bajtów

Drzewa sufiksów – wyzwanie 1

Udowodnię, że drzewo sufiksów łańcucha T :

- ma tyle samo liści, ile łańcuch T ma znaków
- ma mniej wierzchołków wewnętrznych niż łańcuch T ma znaków,

czyli że drzewo sufiksów łańcucha T ma rozmiar $O(|T|)$

Drzewa sufiksów – wyzwanie 2

Spośród łańcuchów, które mają 5 znaków, podam łańcuch, którego drzewo sufiksów ma najwięcej krawędzi

Zastosowania drzew sufiksów

Tak sprawdzam, czy wzorzec P występuje w tekście T :

- Buduję drzewo sufiksów łańcucha T
- Szukam w drzewie sufiksów łańcucha T takiej ścieżki od korzenia, że połączone etykiety krawędzi, które składają się na tę ścieżkę, tworzą łańcuch P
- Jeśli w drzewie sufiksów istnieje taka ścieżka, to znaczy, że wzorzec P występuje w tekście T
- Jeśli w drzewie sufiksów nie ma takiej ścieżki, to znaczy, że wzorzec P nie występuje w tekście T

Zastosowania drzew sufiksów – 2

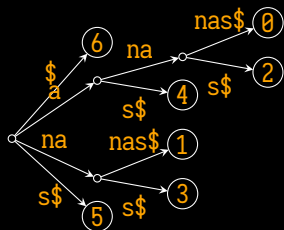
Znam tekst T . Ten tekst się nie zmienia. Chcę znaleźć wszystkie wystąpienia każdego z wielu wzorców P_1, P_2, \dots, P_n w tym tekście

Tworzę drzewo sufiksów łańcucha T

Przechodzę ścieżką od korzenia drzewa wzdłuż tych krawędzi, których etykiety łączą się we wzorec P_i . W ten sposób dochodzę do pewnego wężła N . Potem przeszukuję wężły potomne wężła N , na przykład w głąb. Każdy liść drzewa, który znajduję, zawiera indeks pozycji, od której zaczyna się pewne wystąpienie wzorca P_i w tekście

W ten sposób mogę znaleźć w tekście T wszystkie wystąpienia dowolnego wzorca P_i w czasie $O(|P_i| + \text{liczba wystąpień wzorca } P_i \text{ w tekście } T)$. Ten czas nie zależy od długości łańcucha T

Zastosowania drzew sufiksów – 2



Wystąpienia wzorca **na** zaczynają się od pozycji 1 i 3

Wystąpienia wzorca **an** zaczynają się od pozycji 0 i 2

Wystąpienia wzorca **a** zaczynają się od pozycji 0, 2 i 4

Wystąpienie wzorca **x** nie ma

Jak znaleźć wszystkie te łańcuchy ze zbioru $\{T_1, T_2, \dots, T_n\}$, które zawierają dany wzorzec P ?

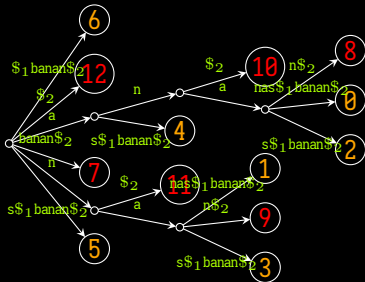
Tworzę drzewo sufiksów łańcucha $T_1\$_1T_2\$_2 \dots T_n\$_n$

Przechodzę ścieżką od korzenia drzewa sufiksów łańcucha $T_1\$_1T_2\$_2 \dots T_n\$_n$ wzdłuż tych krawędzi, których etykiety łączą się we wzorzec P . W ten sposób dochodzę do pewnego wężła N

Potem przeszukuję wężły potomne wężła N , na przykład w głąb. Każdy liść drzewa, który znajduję, zawiera indeks pozycji, od której zaczyna się pewne wystąpienie wzorca P w tekście

Zastosowania drzew sufiksów – 3

Jak znaleźć wszystkie te łańcuchy ze zbioru $\{T_1, T_2, \dots, T_n\}$, które zawierają dany wzorec P ?



Wyznaczam te łańcuchy, które zawierają dany wzorec, na podstawie pozycji, od których zaczynają się wystąpienia tego wzorca w tekście **ananas\$1banan\$2**

Wzorec **an** występuje w łańcuchach **1** i **2**

Wzorec **na** występuje w łańcuchach **1** i **2**

Wzorec **as** występuje w łańcuchu **1**

Wzorec **ban** występuje w łańcuchu **2**

Wzorec **x** nie występuje w żadnym z tych łańcuchów

Zastosowania drzew sufiksów – 3

Jak znaleźć wszystkie łańcuchy ze zbioru $\{T_1, T_2, \dots, T_n\}$, które zawierają dany wzorec P ?

Przykład zastosowania: Baza danych DNA mitochondrialnego żołnierzy USA

Pobieram DNA mitochondrialne od żyjących żołnierzy. Sekwencjonuję mały fragment DNA każdego żołnierza. Wybieram ten fragment tak, żeby był różny u różnych żołnierzy. Zapisuję ten fragment w bazie danych jako identyfikator żołnierza

Tak identyfikuję poległych żołnierzy:

Pobieram DNA mitochondrialne z ciał poległych żołnierzy. Sekwencjonuję ten sam fragment DNA. Porównuję ten fragment z identyfikatorami żołnierzy. Może się zdarzyć, że z ciała nie da się pobrać pełnego fragmentu DNA. Dlatego szukam takich identyfikatorów, które **zawierają** pobrany fragment

Zastosowania drzew sufiksów – 4

Jak znaleźć najdłuższy wspólny podłańcuch 2 łańcuchów T_1 i T_2 ?

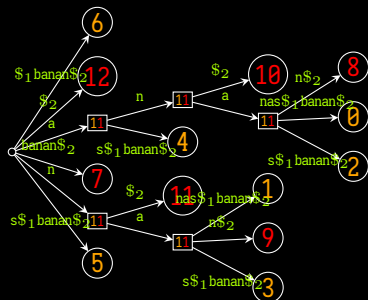
Buduję drzewo sufiksów łańcucha $T_1\$_1T_2\$_2$. Dodaję do każdego z węzłów wewnętrznych tego drzewa 2 bity

W każdym węźle wewnętrznym N tego drzewa ustawiam odpowiednie bity:

- bit 0, jeśli dowolny liść tego poddrzewa, którego korzeniem jest węzeł N , odpowiada pewnemu sufiksowi łańcucha $T_1\$_1T_2\$_2$
- bit 1, jeśli dowolny liść tego poddrzewa, którego korzeniem jest węzeł N , odpowiada pewnemu sufiksowi łańcucha $T_2\$_2$
- bit 0 i bit 1, jeśli dowolny liść tego poddrzewa, którego korzeniem jest węzeł N , odpowiada pewnemu sufiksowi łańcucha $T_1\$_1T_2\$_2$ i dowolny liść tego poddrzewa, którego korzeniem jest węzeł N , odpowiada pewnemu sufiksowi łańcucha $T_2\$_2$

Obliczam 2 bity dodane do każdego węzła wewnętrznego jako sumę logiczną (OR) 2 bitów dodanych do dzieci tego węzła, przeszukując drzewo sufiksów w głąb w kolejności wstecznej (post-order), czyli odwiedzając najpierw dzieci każdego węzła wewnętrznego, a potem ten węzeł

Jak znaleźć najdłuższy wspólny podłańcuch łańcuchów **ananas** i **banan**?



Przeszukuję od korzenia te węzły drzewa sufiksów, w których jest ustawiony i bit 0, i bit 1. Etykiety tych krawędzi, wzdłuż których przechodzę, łączą się w takie podłańcuchy łańcucha **ananas** s_1 **banan** s_2 , które występują i w łańcuchu **ananas**, i w łańcuchu **banan**. Szukam najdłuższego takiego wspólnego podłańcucha. Najdłuższy wspólny podłańcuch łańcuchów **ananas** i **banan** to łańcuch **ana**

Jak wykrywać zanieczyszczenia w DNA?

Przykład: DNA rzekomo wyekstrahowane z kości dinozaura okazało się bardziej podobne do ludzkiego DNA niż do DNA ptaków i krokodyli

Dany jest łańcuch DNA T_1 , połączone łańcuchy DNA możliwych zanieczyszczeń T_2 i liczba całkowita k . Chcę znaleźć wszystkie takie podłańcuchy łańcucha T_2 , które występują w łańcuchu T_1 i mają długość większą niż k

Jak wykrywać zanieczyszczenia w DNA?

Buduję drzewo sufiksów łańcucha $T_1\$_1T_2\$_2$. Dodaję do każdego z węzłów wewnętrznych tego drzewa 2 bity

W każdym węźle wewnętrznym N tego drzewa ustawiam odpowiednie bity na 1:

- bit 0, jeśli dowolny liść tego poddrzewa, którego korzeniem jest węzeł N , odpowiada pewnemu sufiksowi łańcucha $T_1\$_1T_2\$_2$
- bit 1, jeśli dowolny liść tego poddrzewa, którego korzeniem jest węzeł N , odpowiada pewnemu sufiksowi łańcucha $T_2\$_2$
- bit 0 i bit 1, jeśli dowolny liść tego poddrzewa, którego korzeniem jest węzeł N , odpowiada pewnemu sufiksowi łańcucha $T_1\$_1T_2\$_2$ i dowolny liść tego poddrzewa, którego korzeniem jest węzeł N , odpowiada pewnemu sufiksowi łańcucha $T_2\$_2$

Wyznaczam 2 bity dodane do każdego węzła wewnętrznego jako sumę logiczną (OR) 2 bitów dodanych do dzieci tego węzła, przeszukując drzewo sufiksów w głąb w kolejności wstecznej (post-order), czyli odwiedzając najpierw dzieci każdego węzła wewnętrznego, a potem ten węzeł

Przeszukuję od korzenia te węzły drzewa sufiksów, w których jest ustawiony i bit 0, i bit 1. Etykiety tych krawędzi, wzdłuż których przechodzę, łączą się w takie podłańcuchy łańcucha $T_1\$_1T_2\$_2$, które występują i w łańcuchu T_1 , i w łańcuchu T_2 . Szukam wszystkich takich podłańcuchów, których długość jest większa niż dana stała k

Ile razy dany wzorzec P występuje w danym tekście T ?

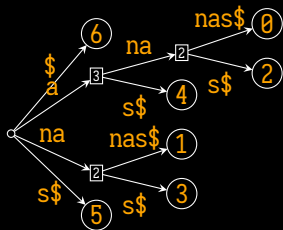
Buduję drzewo sufiksów łańcucha T

Dodaję licznik do każdego węzła tego drzewa. W tym liczniku zapamiętuję, ile liści ma poddrzewo, którego korzeniem jest ten węzeł. Wyznaczam wartość licznika dodanego do każdego węzła jako sumę wartości liczników dodanych do dzieci tego węzła, przeszukując drzewo sufiksów w głąb w kolejności wstecznej (post-order), czyli odwiedzając najpierw dzieci każdego węzła wewnętrznego, a potem ten węzeł

Ile razy dany wzorzec P występuje w danym tekście T ?

Przechodzę ścieżką od korzenia tego drzewa sufiksów wzdłuż tych krawędzi, których etykiety składają się we wzorzec P . W ten sposób dochodzę do pewnego wężła N . Odczytuję z licznika wężła N liczbę liści w poddrzewie wężła N . Gdy nie ma takiego wężła, to znaczy, że wzorzec P występuje 0 razy w tekście T

Ile razy dany wzorec występuje w danym tekście?



Jak znaleźć najdłuższy podłańcuch wspólny dla wielu różnych łańcuchów T_1, T_2, \dots, T_n , ale niekoniecznie wspólny dla wszystkich tych łańcuchów?

Przykład: DNA wielu różnych gatunków. Mutacje tych fragmentów DNA, które kodują krytyczne funkcje, i mutacje innych fragmentów DNA zdarzają się z jednakowym prawdopodobieństwem. Jednak mutacje tych fragmentów DNA, które kodują krytyczne funkcje, częściej powodują przedwczesną śmierć organizmu, przez co ten organizm nie ma potomków. Zatem te fragmenty DNA, które kodują krytyczne funkcje, mniej się zmieniają z pokolenia na pokolenie.

Gdy znajdę najdłuższe wspólne podłańcuchy DNA wielu gatunków, dowiem się, które fragmenty DNA kodują krytyczne funkcje

Jak znaleźć najdłuższy podłańcuch wspólny dla wielu różnych łańcuchów T_1, T_2, \dots, T_n , ale niekoniecznie wspólny dla wszystkich tych łańcuchów?

Buduję drzewo sufiksów łańcucha $T_1\$_1T_2\$_2 \dots T_n\$_n$. Przypisuję do każdego z węzłów wewnętrznych tego drzewa tablicę n bitów. k -ty bit tej tablicy w węźle wewnętrznym N jest ustawiony na 1, gdy dowolny liść tego poddrzewa, którego korzeniem jest węzeł N , odpowiada pewnemu sufiksowi łańcucha $T_1\$_1 \dots T_k\$_k \dots T_n\$_n$, który zaczyna się w łańcuchu T_k

Zastosowania drzew sufiksów – 7

Obliczam tablicę bitów każdego węzła wewnętrznego jako sumę logiczną (OR) tablic bitów dodanych do dzieci tego węzła, przeszukując drzewo sufiksów w głąb w kolejności wstecznej (post-order), czyli odwiedzając najpierw dzieci każdego węzła wewnętrznego, a potem ten węzeł

W tablicy bitów węzła N tyle bitów jest ustawionych na 1, ile różnych numerów łańcuchów pojawia się w liściach tego poddrzewa, którego korzeniem jest węzeł N

Przy ustalonej liczbie ℓ przeszukuję od korzenia takie węzły drzewa sufiksów, że w tablicach bitów tych węzłów co najmniej ℓ bitów jest ustawionych. Etykiety tych krawędzi, wzdłuż których przechodzę, łączą się w podłańcuchy łańcucha $T_1\$1 T_2\$2 \dots T_n\$n$. Szukam najdłuższego takiego podłańcucha

Algorytm Ukkonena

Tę część wykładu opracowałem na podstawie odpowiedzi Johannes Gollera ze Stack Overflow: <https://stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english>

Esko Ukkonen (26.1.1950–)



Fiński informatyk. Emerytowany profesor Uniwersytetu Helsińskiego. W 1995 roku opublikował jeden z prostszych algorytmów budowania drzew sufiksów. Ten algorytm działa w czasie liniowym względem długości łańcucha. W 2000 roku Esko Ukkonen otrzymał najwyższe odznaczenie Finlandii, Order Białej Róży Finlandii.

Drzewo sufiksów to skompresowane drzewo trie. Etykiety jego krawędzi to pary liczb całkowitych $[od, do]$. Liczby od i do to indeksy pozycji początku podłańcucha i pozycji końca podłańcucha w tekście. Chociaż etykieta każdej krawędzi może mieć dowolną długość, zajmuje tylko $O(1)$ komórek pamięci

Algorytm Ukkonena wykonuje $kroki$, przeglądając znaki łańcucha T od początku do końca tego łańcucha. Na każdy znak ciągu przypada jeden krok. Chociaż każdy krok może obejmować więcej niż jedną pojedynczą operację, wykażę, że całkowita liczba operacji wynosi $O(|T|)$, i że każda z tych operacji zajmuje czas $O(1)$

abca**b**xabcd

Krok 1: Zaczynam od lewej. Najpierw dodaję do drzewa sufiksów tylko pojedynczy znak **a**, tworząc krawędź od korzenia drzewa, czyli od węzła **root**, do liścia i oznaczając ją $[0, \#]$. Oznacza to, że krawędź reprezentuje podłańcuch, który zaczyna się na pozycji 0 i kończący na bieżącym końcu. Symbolem $\#$ oznaczam bieżący koniec, który znajduje się na pozycji 1 (zaraz za **a**)

Zatem początkowe drzewo sufiksów wygląda tak:



abcbxabcd

Krok 2: Teraz przechodzę do pozycji 2 (zaraz za **b**). Celem każdego kroku jest dodać wszystkie sufiksy aż do bieżącej pozycji. Aby osiągnąć ten cel:

- rozszerzam istniejącą krawędź **a** do **ab**
- wstawiam jedną nową krawędź dla **b**

Teraz drzewo sufiksów wygląda tak:



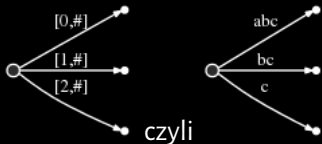
:

- Reprezentacja krawędzi **ab** jest taka sama jak w początkowym drzewie: $[0, \#]$. Jej znaczenie samo się zmieniło, bo zmieniłem bieżącą pozycję $\#$ z 1 na 2
- Każda krawędź zajmuje $O(1)$ miejsca, bo składa się tylko z dwóch wskaźników do tekstu, niezależnie od tego, jak długiemu podłańcuchowi odpowiada

ab^cab^cd

Krok 3: Znowu zwiększam pozycję o 1 i aktualizuję drzewo sufiksów, dodając znak ^c na końcu każdej istniejącej krawędzi i wstawiając jedną nową krawędź dla nowego sufiksu ^c

Teraz drzewo sufiksów wygląda tak:

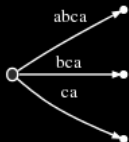


Zauważam, że:

- Po każdym kroku drzewo to trie, które buduję, jest poprawnym drzewem sufiksów prefiksu łańcucha T aż do bieżącej pozycji
- Wykonałem tyle kroków, ile przeczytałem znaków
- W każdym kroku wykonałem $O(1)$ operacji, bo zwiększanie zmiennej $\#$ o 1 zajmuje czas $O(1)$ i jedną nową krawędź dla ostatniego znaku można wstawić w czasie $O(1)$
- Zatem drzewo sufiksów takiego łańcucha **text**, w którym znaki się nie powtarzają, można zbudować w czasie $O(|\text{text}|)$

abca**bx**abcd

Krok 4: Przesuwam # na pozycję 4. W ten sposób niejawnie przesuwam końce wszystkich krawędzi aż do tej pozycji. Teraz drzewo sufiksów wygląda tak:



Teraz należy wstawić do korzenia **a**, ostatni sufix bieżącego kroku

Zanim to zrobię, wprowadzam oprócz zmiennej # jeszcze 2 zmienne:

- Aktywne położenie: trójka (**active_node**, **active_edge**, **active_length**)
- **backlog**: liczba całkowita, która wskazuje, ile nowych sufixów należy dodać do drzewa

- Kiedy wstawiałem do drzewa sufiksów znaki **abc**, aktywnym położeniem było zawsze (**root**, '\$', 0), czyli aktywnym węzłem był korzeń drzewa, aktywną krawędzią był znak wartownika, a aktywna długość była równa 0. Zatem każda nowa krawędź, którą dodawałem do drzewa sufiksów, była dodawana do korzenia drzewa
- Na początku każdego kroku zmienna **backlog** była ustawiana na 1. Oznaczało to, że na końcu każdego kroku należało wstawić do drzewa sufiksów tylko 1 sufiks

Algorytm Ukkonena

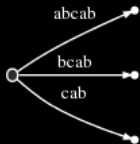
Teraz znak **a** znowu występuje w łańcuchu. Dlatego aktywne położenie i zmienna **backlog** będą się zmieniać. Kiedy wstawiam do korzenia drzewa bieżący ostatni znak **a**, zauważam, że z korzenia już wychodzi krawędź, która zaczyna się od znaku **a**: krawędź **abca**. W takim przypadku:

- Nie dodaję nowej krawędzi [4,#] do korzenia drzewa. Zamiast tego zauważam, że sufix **a** już jest dodany do drzewa. Nie przeszkadza mi to, że ten sufix kończy się wewnątrz dłuższej krawędzi.
- Ustawiam aktywne położenie na (**root**, '**a**', 1). Oznacza to, że aktywne położenie znajduje się za pozycją 1 tej krawędzi, która wychodzi z węzła **root** i ma etykietę **a**. Zauważam, że pierwszy znak krawędzi jednoznacznie wyznacza tę krawędź, bo z danego węzła może wychodzić tylko jedna krawędź, która zaczyna się od danego znaku
- Zwiększam wartość zmiennej **backlog** o 1. Teraz ta zmienna ma wartość 2

Zauważam, że gdy ostatni sufix, który należy wstawić do drzewa sufixów, już występuje w drzewie sufixów, nie zmieniam drzewa sufixów. W takim przypadku zmieniam tylko bieżące położenie i wartość zmiennej **backlog**. Mogę je zmienić w czasie $O(1)$

abca**b**xabcd

Krok 5: Zmieniam bieżącą pozycję # na 5. Teraz drzewo sufiksów wygląda tak:



Zmienna **backlog** ma wartość 2, więc należy wstawić do drzewa sufiksów 2 ostatnie sufiksy tego podłańcucha, który kończy się na bieżącej pozycji

- Sufiks **a** z poprzedniego kroku nie został wstawiony do drzewa sufiksów. W bieżącym kroku zwiększyłem bieżącą pozycję o 1, więc ten sufiks urósł z **a** do **ab**
- Należy też wstawić do drzewa sufiksów nowy ostatni sufiks, czyli **b**

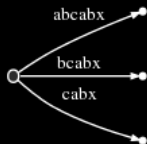
Chcę wstawiać te 2 sufiksy do drzewa sufiksów w kolejności od najdłuższego do najkrótszego sufiksu

Aktywne położenie wskazuje, gdzie kończy się podłańcuch **a** i należy wstawić sufiks **b**: za pierwszym znakiem **a** na krawędzi **abcab**. Okazuje się, że znak **b** już jest na krawędzi **abcab** za pierwszym znakiem **a**. Zatem nie zmieniam drzewa sufiksów, tylko:

- Zmieniam aktywne położenie na (**root**, '**a**', 2). Oznacza to, że aktywne położenie znajduje się na tej samej krawędzi **abcab** wychodzącej z tego samego węzła **root**, o 1 znak dalej, czyli za pierwszym znakiem **b**
- Zwiększam wartość zmiennej **backlog** do 3

abca**bx**abcd

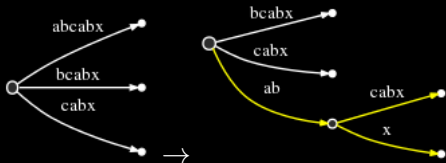
Krok 6: Zwiększam wartość zmiennej # o 1. Teraz drzewo sufiksów wygląda tak:



Algorytm Ukkonena

abcbxabcd

Zmienna **backlog** ma wartość 3. Zatem należy wstawić do drzewa sufiksów sufiksy **abx**, **bx** i **x**. Aktywne położenie wskazuje, gdzie kończy się podłańcuch **ab** i należy wstawić sufiks **x**: za pierwszym znakiem **b** krawędzi **abcbx** wychodzącej z węzła **root**. Znaku **x** nie ma w tym miejscu krawędzi **abcbx**. Zatem dzielę krawędź **abcbx** i wstawiam do drzewa sufiksów nowy węzeł wewnętrzny:



Algorytm Ukkonena

Każda krawędź zawiera indeksy pozycji początku i końca podłańcucha, zatem mogę podzielić krawędź i wstawić do drzewa sufiksów nowy węzeł wewnętrzny w czasie $O(1)$

Poradziłem sobie z sufiksem **abx**. Zmniejszam wartość zmiennej **backlog** do 2. Chcę wstawić do drzewa sufiksów następny zaległy sufiks, czyli **bx**. Najpierw jednak należy zaktualizować aktywne położenie. Korzystam z tej reguły:

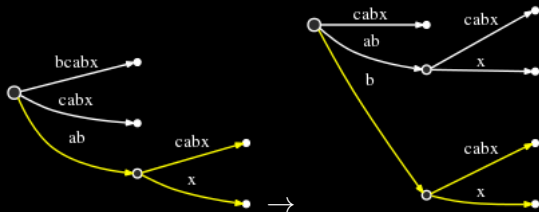
Reguła 1: Jeśli **active_node == root** i **active_length > 0**, to:

- **active_node** nadal będzie mieć wartość **root**
- Przesuwam **active_edge** w prawo do pierwszego znaku następnego zaległego sufiksu
- Zmniejszam **active_length** o 1

Algorytm Ukkonena

abcabxabcd

Zatem nowe aktywne położenie (root, 'b', 1) wskazuje, że następny sufix należy wstawić na krawędzi **bcabx**, za pierwszym znakiem, czyli za znakiem **b**. Mogę znaleźć to miejsce w czasie $O(1)$ i sprawdzić, czy znak **x** już tam jest, czy nie. Gdyby znak **x** tam był, zakończyłbym krok 6. Ale znaku **x** tam nie ma, więc wstawiam go, dzieląc krawędź **bcabx**:



W czasie $O(1)$ podzieliłem krawędź i zmieniłem:

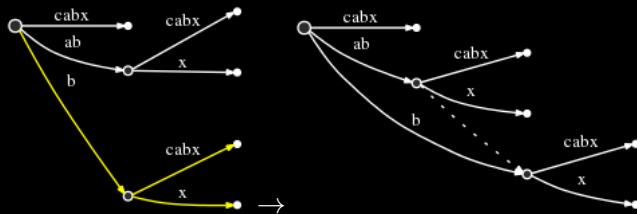
- wartość zmiennej **backlog** na 1
- aktywne położenie na (**root**, '**x**', **0**) zgodnie z Regułą 1

Należy zrobić jeszcze jedną rzecz. Mówi o tym ta reguła:

Reguła 2: Kiedy podzieliłem jakąś krawędź i dodałem do drzewa sufiksów nowy węzeł, i nie jest to pierwszy węzeł, który dodałem do drzewa w bieżącym kroku, łączę ten węzeł, który dodałem jako przedostatni, z tym węzłem, który dodałem jako ostatni, specjalnym **łączem do sufiksu**. Później będzie mowa o tym, do czego się przydają te łącza

abcbxabc

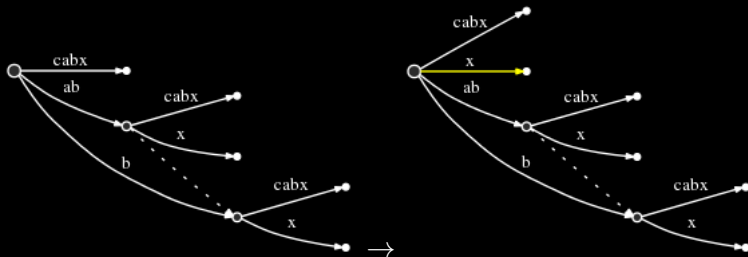
Kropkowana linia oznacza łącze do sufiksu:



Algorytm Ukkonena

ab cab x abcd

Nadal chcę wstawić do drzewa sufiksów ostatni sufiks bieżącego kroku, czyli sufiks **x**. Wartość zmiennej **active_length** spadła do 0, więc wstawiam ten sufiks do korzenia drzewa. Z korzenia drzewa nie wychodzi żadna krawędź, która zaczynałaby się od znaku **x**, więc wstawiam do korzenia drzewa nową krawędź:



Krok 7: Zmieniam wartość zmiennej # na 7. W ten sposób dodaję znak **a** na końcu wszystkich tych krawędzi, które prowadzą do liści drzewa sufiksów. Potem próbuję wstawić nowy ostatni znak, czyli **a**, w **active_node**, czyli w węźle **root**. Z węzła **root** już wychodzi krawędź, która zaczyna się od znaku **a**, więc:

- Zmieniam aktywne położenie na (**root**, '**a**', 1)
- Kończę ten krok

Krok 8: Zmieniam wartość zmiennej `#` na 8. W ten sposób dodaję znak `b` na końcu wszystkich tych krawędzi, które prowadzą do liści drzewa sufiksów. Na krawędzi `ab`, która wychodzi z węzła `root`, za jej pierwszym znakiem już jest znak `b`, więc podobnie jak w poprzednich krokach zmieniam aktywne położenie na `(root, 'a', 2)` i wartość zmiennej `backlog` na 2. W czasie $O(1)$ zauważam, że aktywne położenie znajduje się teraz na końcu krawędzi `ab`. Zatem zmieniam je na `(node_ab, '\0', 0)`. Węzłem `node_ab` nazywam ten węzeł, do którego prowadzi z węzła `root` krawędź z etykietą `ab`

ab^cabxabc^d

Krok 9: Należy wstawić do drzewa sufiksów znak **c**. Zmieniam wartość zmiennej # na 9. W ten sposób w czasie $O(1)$ dodaję znak **c** na końcu wszystkich tych krawędzi, które prowadzą do liści drzewa sufiksów. Z węzła **node_ab** już wychodzi krawędź, której etykieta zaczyna się od znaku **c**.

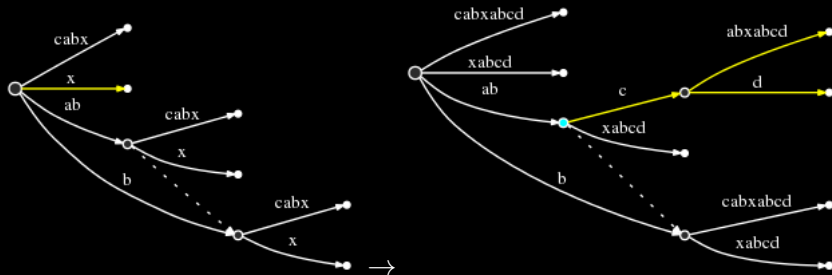
Zatem:

- Zmieniam aktywne położenie na (**node_ab**, '**c**', 1)
- Zwiększam wartość zmiennej **backlog** o 1, do 4
- Kończę ten krok

Algorytm Ukkonena

abcabxabcd

Krok 10: Zwiększam wartość zmiennej # do 10. Zmienna **backlog** ma wartość 4, więc wstawiam do drzewa sufiksów sufiks **abcd** w aktywnym położeniu. Powoduje to podział krawędzi **cabxabcd** w czasie $O(1)$:



Węzeł **active_node** jest oznaczony kolorem. Oto ostatnia reguła:

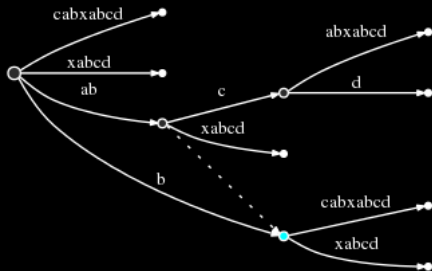
Reguła 3: Gdy podzieliłem krawędź od węzła **active_node**, który nie jest węzłem **root**:

- Jeśli z węzła **active_node** wychodzi łącze do sufiksu, to ustawiam węzeł **active_node** na ten węzeł, na który wskazuje to łącze do sufiksu
- Jeśli z węzła **active_node** nie wychodzi łącze do sufiksu, to ustawiam węzeł **active_node** na węzeł **root**
- Nie zmieniam wartości zmiennych **active_edge** i **active_length**

Algorytm Ukkonena

abcabxabcd

Aktywnym położeniem jest teraz (node_b, 'c', 1). Węzeł node_b jest oznaczony kolorem

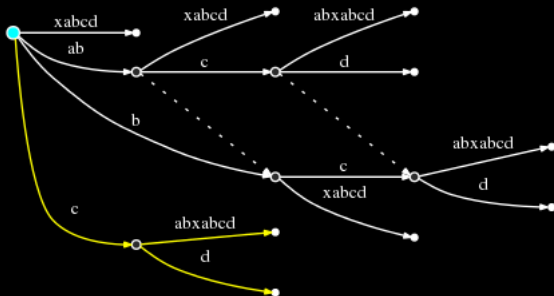


Dzięki łączom do sufiksów mogę zmieniać aktywne położenie w czasie $O(1)$

Algorytm Ukkonena

abcabxabcd

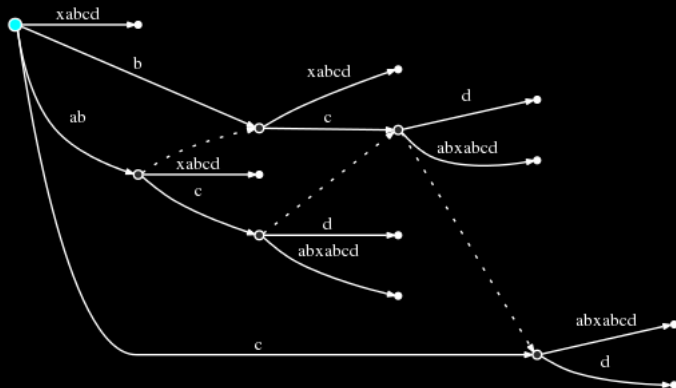
Krok 10 jeszcze się nie skończył. Zmienna **backlog** ma wartość 2. Trzeba skorzystać z reguły 3 i znowu zmienić aktywne położenie. Z węzła **active_node** nie wychodzi łączy do sufiksu, więc ustawiamy ten węzeł na węzeł **root**. Aktywne położenie to teraz (**root**, **c**, 1). Wstawiam znak **d** za pierwszym znakiem tej krawędzi wychodzącej z węzła **root**, która ma etykietę **cabxabcd**. Powoduje to kolejny podział krawędzi:



Algorytm Ukkonena

abxabcd

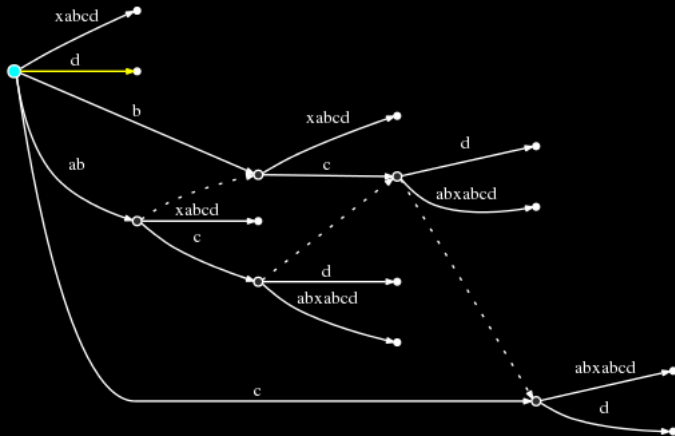
Stosuję regułę 2. Dodaję łącze do sufiksu do przedostatniego węzła utworzonego w tym kroku:



Algorytm Ukkonena

abcabxabcd

Teraz zmienna **backlog** ma wartość 1. Węzeł **active_node** to węzeł **root**.
Korzystając z reguły 1 zmieniam aktywne położenie na (**root**, 'd', 0).
Ostatnie wstawienie w tym kroku to wstawienie sufiksu **d** w węźle **root**:



Algorytm Ukkonena

Gdy tekst ma n znaków, algorytm Ukkonena wykonuje n kroków. W każdym kroku albo tylko zmienia wartości zmiennych w czasie $O(1)$, albo ponadto dodaje do drzewa zaległe sufiksy, też w czasie $O(1)$. Zmienna **backlog** wskazuje, ile razy algorytm nie dodawał sufiksów do drzewa w poprzednich krokach. Ta zmienna zmniejsza się o 1 zawsze wtedy, kiedy algorytm dodaje sufiks do drzewa. Zatem algorytm Ukkonena dodaje do drzewa sufiksy $O(n)$ razy. Zatem całkowita złożoność czasowa algorytmu Ukkonena to $O(n)$.

Podsumowanie

- Tablice sufiksów
- Zastosowania tablic sufiksów
- Drzewa sufiksów
- Zastosowania drzew sufiksów
- Algorytm Ukkonena budowania drzew sufiksów

Pomysły, uwagi, pytania, sugestie

Proszę wysyłać podpisane pomysły, uwagi, pytania, sugestie na temat wykładów lub laboratoriów na adres mgc@agh.edu.pl

lub wpisywać anonimowe pomysły, uwagi, pytania, sugestie pod adresem <https://tiny.cc/algorytmy-tekstowe>

Do zobaczenia na następnym wykładzie

Jego tematem będą pomysłowe algorytmy tekstowe

Žródła ilustracji

Zdeněk Jirotko, Saturnin – zrzut ekranu

https://commons.wikimedia.org/wiki/File:DNA123_rotated.png

<https://classic.csunplugged.org/activities/text-compression/>

Marshall Custiss Hazard, A Complete Concordance to the American Standard Version of the Holy Bible – zrzut ekranu z Archive.org

Donald E. Knuth, The Art of Computer Programming, tom 3

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

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

<https://www.linkedin.com/in/pgweiner/>

<https://www.cs.helsinki.fi/u/ukkonen/>

<https://stackoverflow.com/questions/9452701/>

ukkonens-suffix-tree-algorithm-in-plain-english

Wygaszacz ekranu GLMatrix: Copyright © 1999-2003 by Jamie Zawinski.
Permission to use, copy, modify, distribute, and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. No representations are made about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.