2018/10/15 22:00 修正

計算言語学

文(書)→単語列

東京大学生産技術研究所
吉永 直樹

site: http://www.tkl.iis.u-tokyo.ac.jp/~ynaga/class/cl/

テキストを計算機でどう表現するか？

- 計算機上でのテキストのデータ表現: 文字列
  - 文字を表現(符号化)したバイト(列)の系列

文字を要素とした系列	文(書)＝アトミックな要素
彼|は|越|谷|に|引|っ|越|し|た	彼は東京に引っ越した
H|e| |m|o|v|e|d| |t|o| |K|o|s|h|i|g|y|a|.	He moved to Tokyo.
各要素の意味が多様過ぎる 通言語的な一貫性に乏しい	要素の種類数が多過ぎる （メモリに載らない）

単語の系列としてテキスト

- 単語＝テキストを構成する部分文字列
  - 音韻的な語: アクセントやイントネーションのまとまり
  - 統語的な語: 品詞などの統語的役割を割り当てる単位
  - 正書法に基づく語: 空白など言語の正書法で決まる単位

- 何を単語とする(べき)かはデータ・応用依存

  Dr.␣York’s␣father␣moved␣to␣New␣York.

  Dr. | York’s | father | moved | to | New | York.

  Dr. | York’s | father | moved | to | New | York |.

  Dr. | York | ’s | father | moved | to | New | York |.

  Dr. | York | ’s | father | moved | to | New York |.

コーパス: テキストの集積

- (生)コーパス: 自然言語で書かれた文書や音声を(大規模に)記録・集積したデータ
  - 新聞記事，ウェブ(Wikipedia など)，電子カルテ etc.

  ウェブ出現以降，超大規模・多様化

- 注釈付きコーパス: 生コーパス＋(言語的)注釈
  - Brown コーパス: 様々なテキスト(新聞，小説等)＋品詞
  - Penn TreeBank: Wall Street Journal + 品詞・構文
  - 京都大学テキストコーパス: 毎日新聞＋品詞・構文・意味情報

  クラウドソーシング等により低コストで開発可能だが注釈の専門性が高い場合は大規模化は困難

テキストと単語に関する経験則 (1/3)

- Herdan's Law (1960) (or Heap's law (1978))
  テキストの単語数 N とテキストに含まれる単語の種類数(語彙サイズ) $|V|$ に関する経験則

$$|V| = kN^{\beta}$$

k, β はコーパス依存の定数

Corpus	N	$\|V\|$
Brown コーパス	~1,000,000	38,000
Penn TreeBank	1,173,766	57,389
京都大学テキストコーパス	972,894	39,431
Goolge N-grams	~1,000,000,000	13,000,000

テキストと単語に関する経験則 (2/3)

- Zipf's Law (1935):
  テキストに含まれる各単語の出現頻度 f とその単語のテキスト中での頻度順位 r に関する経験則

$$f \propto \frac{1}{r}$$

Penn TreeBank
(N=1173766,
$|V|$=57389)

Word	Rank (r)	Freq (f)	$f\,r$
,	1	60,484	60,484
the	2	50,975	101,950
to	5	27,249	136,245
for	10	9890	98900
A	100	1108	110,800
Bay	1000	136	136,000
Namely	10,000	7	70,000

テキストと単語に関する経験則 (3/3)

- Zipf's Law (1935):
  テキストに含まれる各単語の出現頻度 f とその単語のテキスト中での頻度順位 r に関する経験則

$$f \propto \frac{1}{r}$$

京大コーパス
(N= 972894,
$|V|$= 39431)

Word	Rank (r)	Freq (f)	$f\,r$
の	1	56,993	56,993
、	2	51,544	103,088
は	5	32,235	161,175
」	10	10,417	104,170
後	100	798	79,800
治療	1000	104	104,000
取り戻し	10,000	7	70,000

テキストから単語列への変換

文分割	珈琲，美味しかったな… | 根津にあったカフェ。	I’m eating cakes. | They included nuts.
tokenization (単語分割)	根津 | に | あった | カフェ	I | ’m | eating | cakes | .
lemmatization (見出し語化)	根津 に ある カフェ	I be eat cake .

大半の言語処理タスクの前処理であり高速な処理が求められる

テキストから単語列への変換

文分割	珈琲，美味しかったな… | 根津にあったカフェ。	I’m eating cakes. | They included nuts.
tokenization (単語分割)	根津|に|あった|カフェ	I | ’m | eating | cakes | .
lemmatization (見出し語化)	根津 に ある カフェ	I be eat cake .

大半の言語処理タスクの前処理であり高速な処理が求められる

文分割

- 句点や疑問符など文末記号が手がかりとなる
  - ただし，記号が曖昧だったり，ない場合も

モーニング娘。がデビューして20年以上たつのか…

Apple Inc. was founded by S. Jobs, S. Wozniak, and R. Wayne.

余談: LaTeX は大文字で始まる単語末尾の.を文末記号として扱わないので適切な空白を入れるには \@ を入れて文末を示唆する必要がある
例) I bought a ticket to Vancouver\@. The ticket costed …

- 文脈を考慮して適切に文分割を行うには機械学習に基づく二値分類器を用いる

テキストから単語列への変換 (再掲)

文分割	珈琲，美味しかったな… | 根津にあったカフェ。	I’m eating cakes. | They included nuts.
tokenization (単語分割)	根津 | に | あった | カフェ	I | ’m | eating | cakes | .
lemmatization (見出し語化)	根津 に ある カフェ	I be eat cake .

大半の言語処理タスクの前処理であり高速な処理が求められる

単語分割と正規化 (1/2):

- 正書法で空白により入力が分割される言語では，空白での分割を基本として微調整を行う
  - 英語では Penn Treebank で採用された正規表現に基づく Penn Treebank tokenization が用いられる

“The San Francisco-based restaurant,” they said, “doesn’t charge $10”.

“ | The | San | Francisco-based | restaurant | , | ” | they | said |, | “ | does | n’t | charge | $ | 10 | ” | .

- 応用により，単語の正規化 (US→USA, cooool->cool) を行ったり，固有名詞を一語にするなどする

単語分割と正規化 (2/2):

- 単語境界が正書法で明示されない言語（日本語，中国語，タイ語など）ではより高度な処理が必要

他特別喜欢北京烤鸭

他 | 特別 | 喜欢 | 北京烤鸭

外国人参政権

外国人 | 参政権

- 辞書を用いた最長一致法が高速
- (機械学習に基づく)最小コスト法や点推定が高精度

例) すもも も ももも も ももの うち

最長一致法: 辞書に基づく決定的単語分割

- **最長一致法**: テキストの先頭から辞書に定義された最長の単語を順に切り出すアルゴリズム

```
fun MaxMatch(sent, dict)
  i = 0; words = []
  while i < len (sent):
    w = the longest word in dict
        starting from sent[i]
    if len (w) > 0:
      words.append (w)
      i = i + len (w)
    else:
      words.push (sent[i])
      i += 1
  return words
```

dic={外,外国,外国人,人参,参政権}
sent="外国人参政権"

i = 0, words = []

i = 3, words = ["外国人"]

i = 6, words = ["外国人", "参政権"]

最長一致法: 辞書に基づく決定的単語分割

- **最長一致法**: テキストの先頭から辞書に定義された最長の単語を順に切り出すアルゴリズム
  - 注) 英語ではうまく動作しない

```
fun MaxMatch(sent, dict)
  i = 0; words = []
  while i < len (sent):
    w = the longest word in dict
        starting from sent[i]
    if len (w) > 0:
      words.append (w)
      i = i + len (w)
    else:
      words.push (sent[i])
      i += 1
  return words
```

```
sent="wecanonlyseeashortdistanceahead"

words=["we","canon","l","y","see",
       "ash", "ort", "distance",
       "ahead"]
```

最長一致法: 辞書に基づく決定的単語分割

- **最長一致法**: テキストの先頭から辞書に定義された最長の単語を順に切り出すアルゴリズム
  - 注) 英語ではうまく動作しない
  - 効率的な辞書引きアルゴリズムが別途必要

```
fun MaxMatch(sent, dict)
  i = 0; words = []
  while i < len (sent):
    w = the longest word in dict
        starting from sent[i]
    if len (w) > 0:
      words.append (w)
      i = i + len (w)
    else:
      words.push (sent[i])
      i += 1
  return words
```

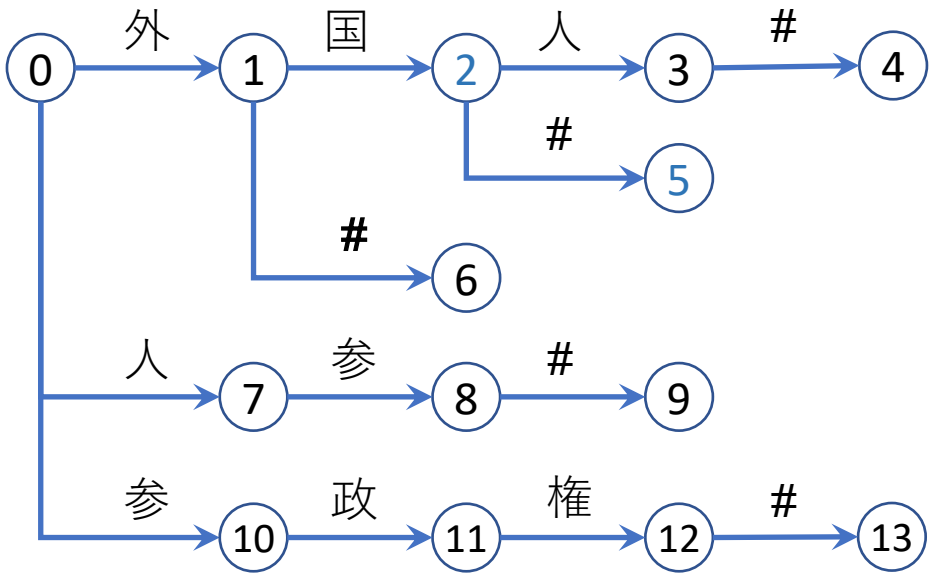
sent="wecanonlyseeashortdistanceahead"

words=["we","canon","l","y","see",
"ash", "ort", "distance",
"ahead"]

トライに基づく最長一致法

- トライ(trie): エッジに文字が割り当てられた木構造
  - 接頭辞の重複する文字列の集合を効率よく格納
  - 終端文字 # をエッジは登録文字列の終了を表現

辞書： {外, 外国, 外国人, 人参, 参政権}

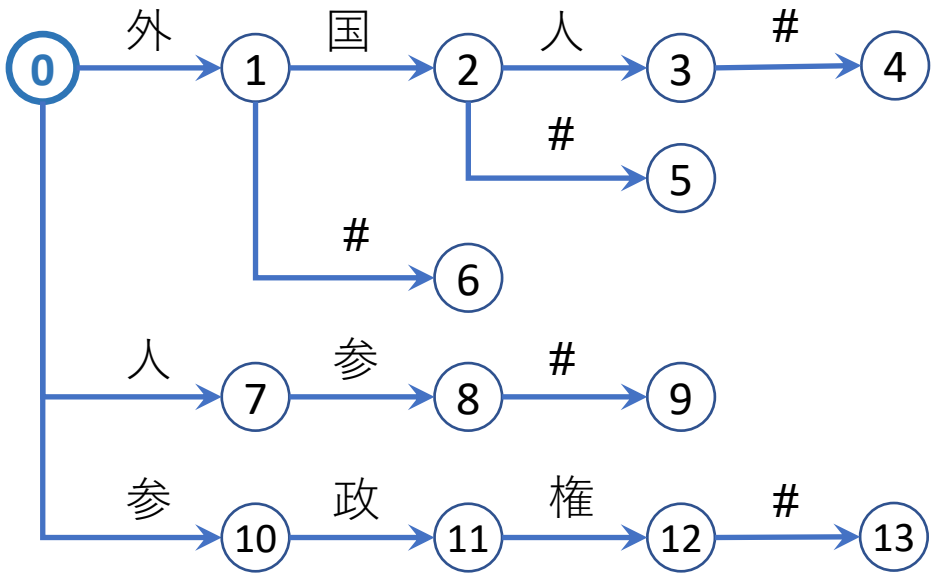


外国人参政権

トライに基づく最長一致法

- トライ(trie): エッジに文字が割り当てられた木構造
  - 接頭辞の重複する文字列の集合を効率よく格納
  - 終端文字 # をエッジは登録文字列の終了を表現

辞書： {外, 外国, 外国人, 人参, 参政権}



単語探索の開始位置

↓

外国人参政権に…

トライに基づく最長一致法

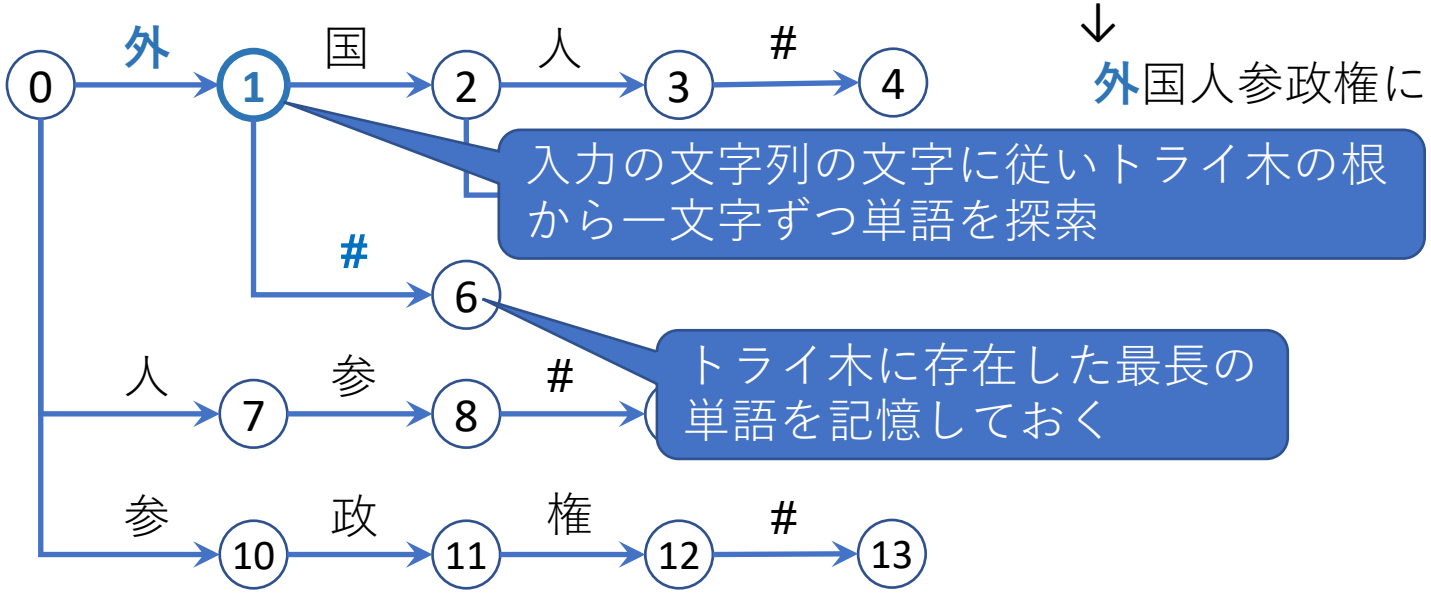
- トライ(trie): エッジに文字が割り当てられた木構造
  - 接頭辞の重複する文字列の集合を効率よく格納
  - 終端文字 # をエッジは登録文字列の終了を表現

辞書： {外, 外国, 外国人, 人参, 参政権}

単語探索の開始位置

↓

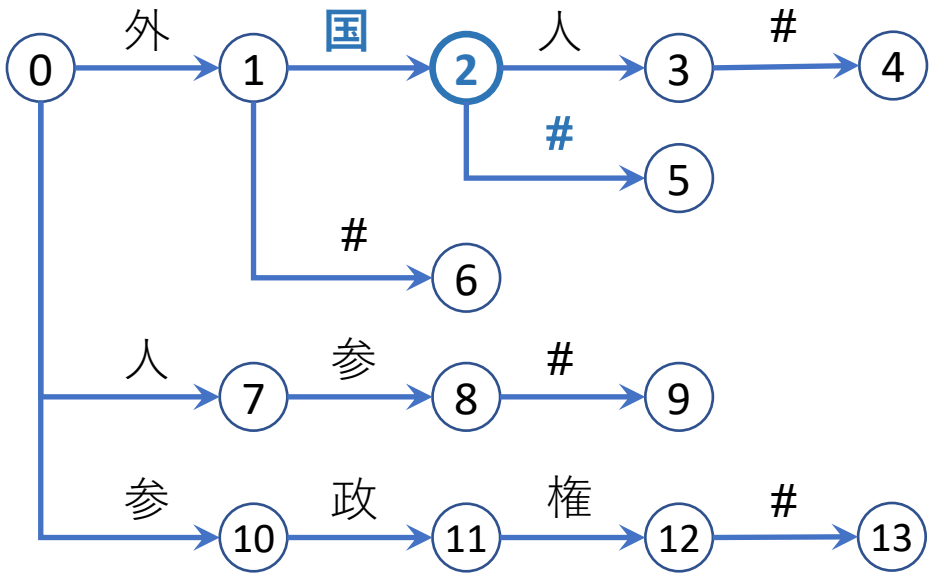
外国人参政権に…



トライに基づく最長一致法

- トライ(trie): エッジに文字が割り当てられた木構造
  - 接頭辞の重複する文字列の集合を効率よく格納
  - 終端文字 # をエッジは登録文字列の終了を表現

辞書： {外, 外国, 外国人, 人参, 参政権}



単語探索の開始位置

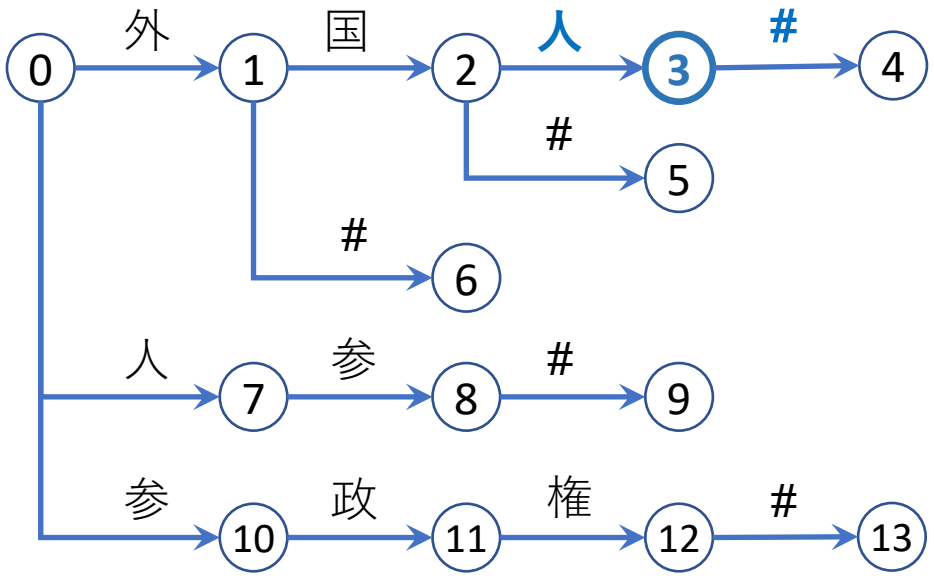
↓

外国人参政権に…

トライに基づく最長一致法

- トライ(trie): エッジに文字が割り当てられた木構造
  - 接頭辞の重複する文字列の集合を効率よく格納
  - 終端文字 # をエッジは登録文字列の終了を表現

辞書： {外, 外国, 外国人, 人参, 参政権}



単語探索の開始位置

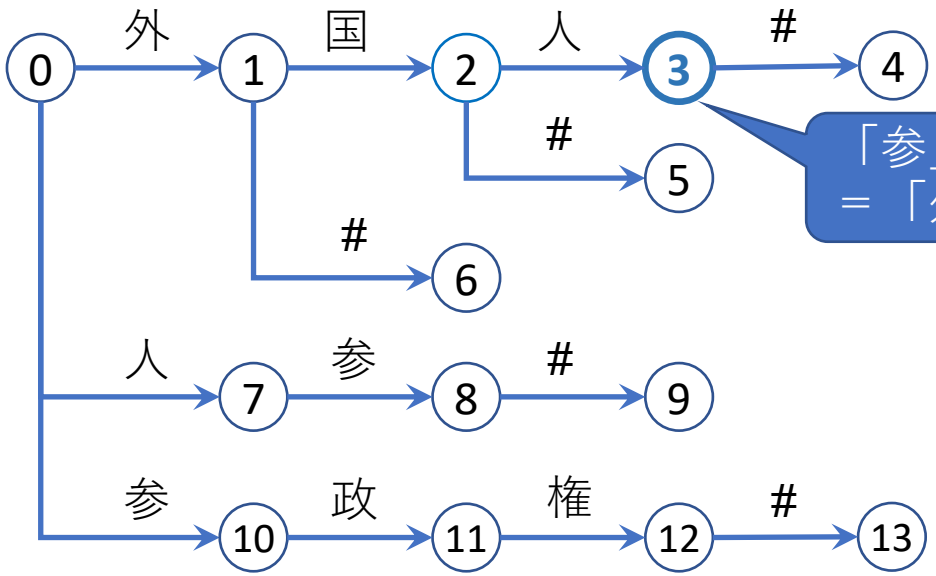
↓

外国人参政権に…

トライに基づく最長一致法

- トライ(trie): エッジに文字が割り当てられた木構造
  - 接頭辞の重複する文字列の集合を効率よく格納
  - 終端文字 # をエッジは登録文字列の終了を表現

辞書： {外, 外国, 外国人, 人参, 参政権}



単語探索の開始位置

↓

外国人参政権に…

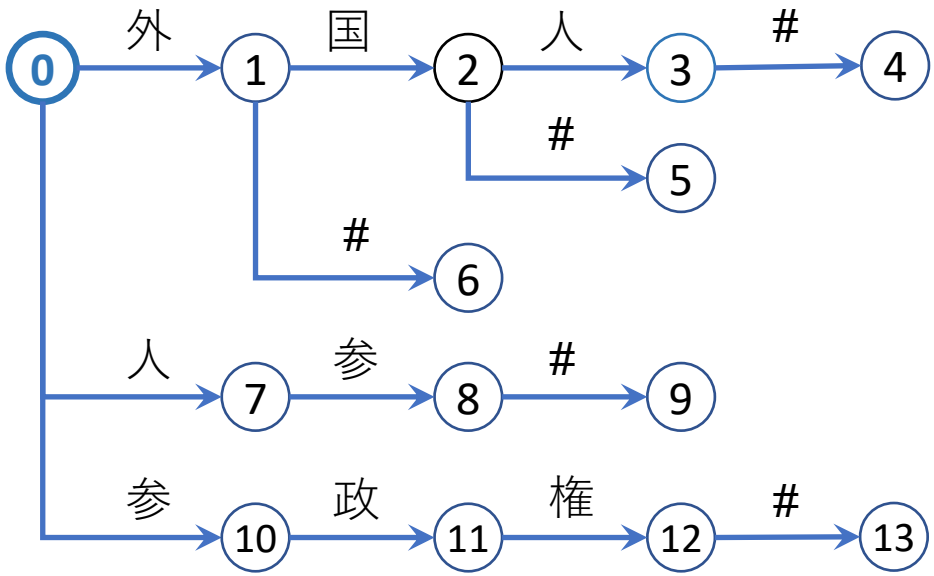
「参」で辿れるエッジが存在しない
＝「外国人参」で始まる単語がない

最長の文字列が見つかると
直ちに探索が終了する

トライに基づく最長一致法

- トライ(trie): エッジに文字が割り当てられた木構造
  - 接頭辞の重複する文字列の集合を効率よく格納
  - 終端文字 # をエッジは登録文字列の終了を表現

辞書： {外, 外国, 外国人, 人参, 参政権}



単語探索の開始位置

↓

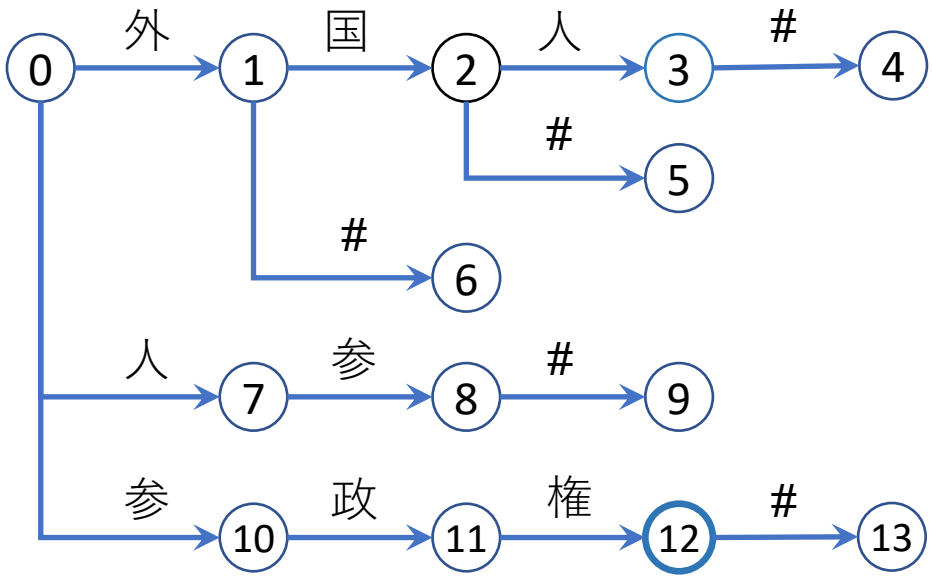
外国人 | 参政権に…

トライ木の根に戻り
入力中の単語を探索

トライに基づく最長一致法

- トライ(trie): エッジに文字が割り当てられた木構造
  - 接頭辞の重複する文字列の集合を効率よく格納
  - 終端文字 # をエッジは登録文字列の終了を表現

辞書： {外, 外国, 外国人, 人参, 参政権}



単語探索の開始位置

↓

外国人｜参政権｜に…

効率的なトライ実装

- ダブル配列(Aoe, 1989)
  - 2つの整数値配列でトライを表現
    - Base: 同じ親ノードを持つ子ノードのアドレスのオフセット
    - Check: 親から子ノードへの遷移を確認するための補助配列
  - UnicodeはUTF-8でバイトを文字として扱うのが効率的

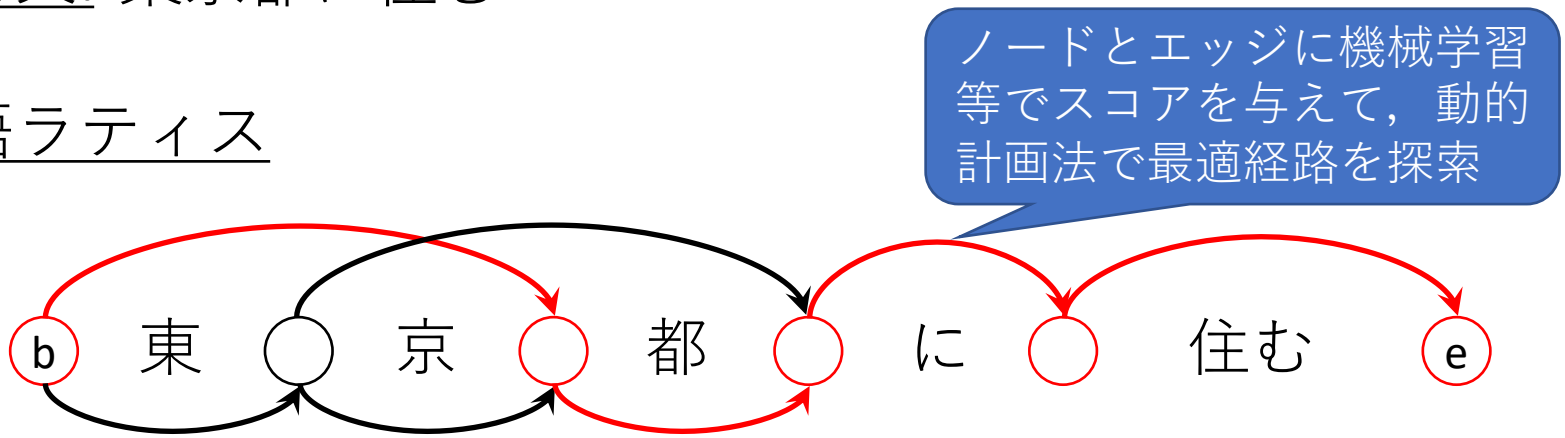
- LOUDS (Jacobson, 1989)
  - 木構造を表現可能な簡潔データ構造の一種
  - 日本語入力などメモリ制約の強い応用でのみ用いられる

より高度な単語分割: 最小コスト法と点推定

- 最小コスト法: 辞書ベースの単語分割候補の列挙＋(機械学習と) 動的計画法による解選択

入力文: 東京都に住む

単語ラティス



- 点推定: 二値分類器を用いた分割間の相互依存を考慮しない単純分割

テキストから単語列への変換 (再掲)

文分割	珈琲，美味しかったな… | 根津にあったカフェ。	I’m eating cakes. | They included nuts.
tokenization (単語分割)	根津|に|あった|カフェ	I | ’m | eating | cakes | .
lemmatization (見出し語化)	根津 に ある カフェ	I be eat cake .

大半の言語処理タスクの前処理であり高速な処理が求められる

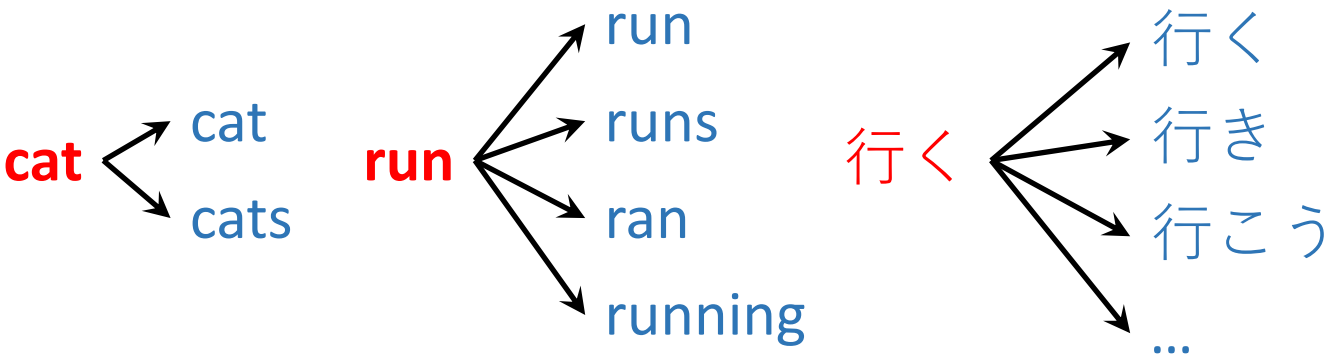
Lemmatization (見出し語化)

- **Lemmatization**: テキスト中の各語を lemma に変換

  He is reading detective stories .

  He be read detective story .

  - Lemma: 語形変化(inflection; 屈折)により同一語幹, 語義, 品詞の語派生した語(語形)に対する標準形
    ≒辞書の見出し語



品詞タグ付け＋辞書を用いることで見出し語化を行うが
品詞の曖昧性の少ない日本語では辞書引きのみでも大体解ける

Stemming

- **Stemming:** 単語末尾の接辞を切って語幹に変換

  例) Porter stemmer:
  https://tartarus.org/martin/PorterStemmer/
  書き換え規則の連続適用に基づく英語の stemmer

  John obtained better results.

  - 文脈を考慮しないため高速に動作するが
    不適切な正規化を行うことがある
    - Over-stemming: organization → organ, paste → past, useful → us
    - Under-stemming: european → european, triangular → triangular

未知語と自然言語処理

- 辞書やコーパスなど既存の言語資源に基づく手法では未知語が大きな問題となる
  - 辞書に含まれない単語
  - (統計的手法で) 学習コーパスに含まれない単語
- 言語は動的な側面を持つため，網羅的な単語集合を作っても新しい語が毎日のように発生する
  - バイナンス，艦これ，ですよ。

- 対策
  - 未知語処理，また言語資源(辞書等) の(継続的)拡充
    例) https://github.com/neologd/mecab-ipadic-neologd
  - 未知語が少なくなるような単語分割手法の採用

サブワード: 未知語への対応

- 低頻度語をより短い部分文字列（サブワード）の系列で表現し，語彙サイズ・未知語を削減
  - サブワード: (言語学的な)単語と文字の中間の分割単位
    例) 足利義満=足利義+満，lower=low+er

    語彙サイズの制限の大きいニューラル言語処理で活用

- 入力テキストと初期単語集合から単語(サブワード)集合を計算
  - Byte-pair encoding (BPE) [Gage 1994, Sennrich+ 2016]
    1文字＝単語から始めて貪欲的に連結した単語を追加
  - ユニグラム言語モデル [Kudo 2018]
    大規模語彙を元に文の尤度が高くなるよう単語を分割

Byte-pair encoding (BPE) [Sennrich+ 2016]

- 1文字＝1単語から始めて，連結した際に最も頻度が高くなる2つの単語を連結して単語集合に追加
  - 事前に指定した語彙サイズを下回る限り繰り返す

単語	頻度
l o w ·	5
l o w e s t ·	2
n e w e r ·	6
w i d e r ·	3
n e w ·	2

単語集合

{ l, o, w, e, s, t, n, r, i, d }

Byte-pair encoding (BPE) [Sennrich+ 2016]

- 1文字＝1単語から始めて，連結した際に最も頻度が高くなる2つの単語を連結して単語集合に追加
  - 事前に指定した語彙サイズを下回る限り繰り返す

単語	頻度		単語集合
l o w ·	5		{ l, o, w, e, s, t, n, r, i, d }
l o w e s t ·	2	{ e, r }	{ l, …, d, er }
n e w er ·	6		
w i d er ·	3		
n e w ·	2		

Byte-pair encoding (BPE) [Sennrich+ 2016]

- 1文字＝1単語から始めて，連結した際に最も頻度が高くなる2つの単語を連結して単語集合に追加
  - 事前に指定した語彙サイズを下回る限り繰り返す

単語	頻度		単語集合
l o w ·	5		{ l, o, w, e, s, t, n, r, i, d }
l o w e s t ·	2	{ e, r }	{ l, …, d, er }
n e w er·	6	{ er, · }	{ l, …, d, er, er· }
w i d er·	3		
n e w ·	2		

Byte-pair encoding (BPE) [Sennrich+ 2016]

- 1文字＝1単語から始めて，連結した際に最も頻度が高くなる2つの単語を連結して単語集合に追加
  - 事前に指定した語彙サイズを下回る限り繰り返す

単語	頻度		単語集合
l o w ·	5		{ l, o, w, e, s, t, n, r, i, d }
l o w e s t ·	2	{ e, r }	{ l, …, d, er }
ne w er·	6	{ er, · }	{ l, …, d, er, er· }
w i d er·	3	{ n, e }	{ l, …, d, er, er·, ne }
ne w ·	2		

Byte-pair encoding (BPE) [Sennrich+ 2016]

- 1文字＝1単語から始めて，連結した際に最も頻度が高くなる2つの単語を連結して単語集合に追加
  - 事前に指定した語彙サイズを下回る限り繰り返す

単語	頻度		単語集合
l o w ·	5		{ l, o, w, e, s, t, n, r, i, d }
l o w e s t ·	2	{ e, r }	{ l, …, d, er }
new er·	6	{ er, · }	{ l, …, d, er, er· }
w i d er·	3	{ n, e }	{ l, …, d, er, er·, ne }
new ·	2	{ ne, w }	{ l, …, d, er, er·, ew, new }

Byte-pair encoding (BPE) [Sennrich+ 2016]

- 1文字＝1単語から始めて，連結した際に最も頻度が高くなる2つの単語を連結して単語集合に追加
  - 事前に指定した語彙サイズを下回る限り繰り返す

単語	頻度		単語集合
lo w ·	5		{ l, o, w, e, s, t, n, r, i, d }
lo w e s t ·	2	{ e, r }	{ l, …, d, er }
new er·	6	{ er, · }	{ l, …, d, er, er· }
w i d er·	3	{ n, e }	{ l, …, d, er, er·, ne }
new ·	2	{ ne, w }	{ l, …, d, er, er·, ew, new }
		{ l, o }	{ l, …, d, er, er·, ew, new, lo }

Byte-pair encoding (BPE) [Sennrich+ 2016]

- 1文字＝1単語から始めて，連結した際に最も頻度が高くなる2つの単語を連結して単語集合に追加
  - 事前に指定した語彙サイズを下回る限り繰り返す

単語	頻度		語彙集合
low ·	5		{ l, o, w, e, s, t, n, r, i, d }
low e s t ·	2	{ e, r }	{ l, …, d, er }
new er ·	6	{ er, · }	{ l, …, d, er, er· }
w i d er ·	3	{ n, e }	{ l, …, d, er, er·, ne }
new ·	2	{ ne, w }	{ l, …, d, er, er·, ew, new }
		{ l, o }	{ l, …, d, er, er·, ew, new, lo }
		{ lo, w }	{ l, …, d, er, er·, ew, new, lo, low }

Byte-pair encoding (BPE) [Sennrich+ 2016]

- 1文字＝1単語から始めて，連結した際に最も頻度が高くなる2つの単語を連結して単語集合に追加
  - 事前に指定した語彙サイズを下回る限り繰り返す
  - 得られた連結規則を獲得順に適用し単語分割を行う

		語彙集合
l o w e r ·		{ l, o, w, e, s, t, n, r, i, d }
l o w er ·	{ e, r }	{ l, …, d, er }
l o w er·	{ er, · }	{ l, …, d, er, er· }
	{ n, e }	{ l, …, d, er, er·, ne }
	{ ne, w }	{ l, …, d, er, er·, ew, new }
lo w er·	{ l, o }	{ l, …, d, er, er·, ew, new, lo }
low er·	{ lo, w }	{ l, …, d, er, er·, ew, new, lo, low }
	…	…

異なる単語間の近さをどうモデル化するか？

- 単語自体の表層的な近さ
  - 最小編集距離: 要素に対する既定の編集操作により，一方の語を他方に変換する最小操作回数

例) 最小編集距離に基づく単語アラインメント

```
僕 は 友 達 が 少 な い
|  |  |  |  |  |  |  |
＊ は ＊ ＊ が ＊ な い
d    d  d    d
```

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

- 単語の出現文脈の近さ
  - 分布仮説 [Harris 1954, Firth 1957]

a *word* is characterized by *the company it keeps*

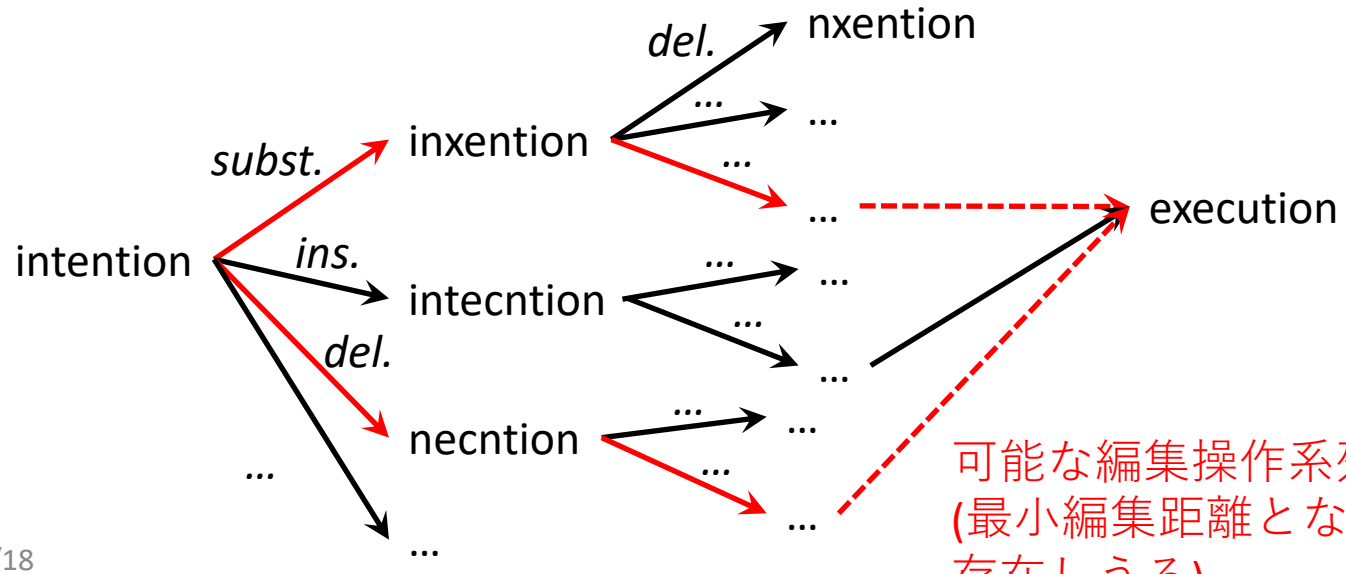
The small dog barks louder.
His dog runs fast.
Eyes of the dog was very small.

Foxes are barking in the distance.
A large fox ran to catch rabbit.
The fox lost one of his eyes.

最小編集距離の計算

- 最小編集距離
  与えられた2つの文字列 *x, y* に対して，*x* を *y* に変換するのに必要な要素の編集操作の最少回数
  - Substitution
  - Deletion
  - Insertion

  各操作に重みを割り当てて最小化することも可能 (全て1のとき Levenstein 距離と呼ぶ)



可能な編集操作系列は指数的に存在
(最小編集距離となる系列も複数存在しうる)

動的計画法による最小編集距離の効率的計算

- 動的計画法 (dynamic programming)
  問題を部分問題に分解して解き結果を再利用することで
  部分問題に対して重複する計算を一度で済ませる

挿入・削除コスト1，置換コスト2の
最小編集距離の計算（動的計画法）

```
fun MinEditDist(src,target):
  # 初期化
  d[[0,0]] = 0
  for i in len (s):
    d[[i,0]] = d[[i-1,0]] + 1 # del.
  for j in len (t):
    d[[0,j]] = d[[0,j-1]] + 1 # ins.
  # 部分問題の結果を利用して構成的に求解
  for i in len (s):
    for j in len (s):
      d[[i,j]] = min (d[[i-1, j]] + 1,
                      d[[i-1, j-1]] + 2,
                      d[[i,j-1]] + 1)
return d
```

src=INTENTION; trg=EXECUTION

	#	E	X	E	C	U	T	I	O	N
#	0	1	2	3	4	5	6	7	8	9
I	1	2	3	4	5	6	7	6	7	8
N	2	3	4	5	6					
T	3	4	5	6	7					
E	4	3	4	5	6	7	8	9	10	9
N	5	4	5	6	7	8	9	10	11	10
T	6	5	6	7	8	9	8	9	10	11
I	7	6	7	8	9	10	9	8	9	10
O	8	7	8	9	10	11	10	9	8	9
N	9	8	9	10	11	12	11	10	9	8

INTE と EXEC の
最小編集距離

動的計画法による最小編集距離の効率的計算

- 動的計画法 (dynamic programming)
  問題を部分問題に分解して解き結果を再利用することで
  部分問題に対して重複する計算を一度で済ませる

挿入・削除コスト1，置換コスト2の
最小編集距離の計算（動的計画法）

```
fun MinEditDist(src,target):
  # 初期化
  d[[0,0]] = 0
  for i in len (s):
    d[[i,0]] = d[[i-1,0]] + 1 # del.
  for j in len (t):
    d[[0,j]] = d[[0,j-1]] + 1 # ins.
  # 部分問題の結果を利用して構成的に求解
  for i in len (s):
    for j in len (s):
      d[[i,j]] = min (d[[i-1, j]] + 1,
                      d[[i-1, j-1]] + 2,
                      d[[i,j-1]] + 1)
return d
```

src=INTENTION; trg=EXECUTION

	#	E	X	E	C	U	T	I	O	N
#	0	← 1	← 2	← 3	← 4	← 5	← 6	← 7	← 8	← 9
I	1									
N	2									
T	3									
E	4									
N	5									
T	6									
I	7									
O	8									
N	9									

動的計画法による最小編集距離の効率的計算

- 動的計画法 (dynamic programming)
  問題を部分問題に分解して解き結果を再利用することで
  部分問題に対して重複する計算を一度で済ませる

挿入・削除コスト1，置換コスト2の
最小編集距離の計算（動的計画法）

```
fun MinEditDist(src,target):
  # 初期化
  d[[0,0]] = 0
  for i in len (s):
    d[[i,0]] = d[[i-1,0]] + 1 # del.
  for j in len (t):
    d[[0,j]] = d[[0,j-1]] + 1 # ins.
  # 部分問題の結果を利用して構成的に求解
  for i in len (s):
    for j in len (s):
      d[[i,j]] = min (d[[i-1, j]] + 1,
                      d[[i-1, j-1]] + 2,
                      d[[i,j-1]] + 1)
return d
```

src=INTENTION; trg=EXECUTION

	#	E	X	E	C	U	T	I	O	N
#	0	1	2	3	4	5	6	7	8	9
I	1	2								
N	2									
T	3									
E	4									
N	5									
T	6									
I	7									
O	8									
N	9									

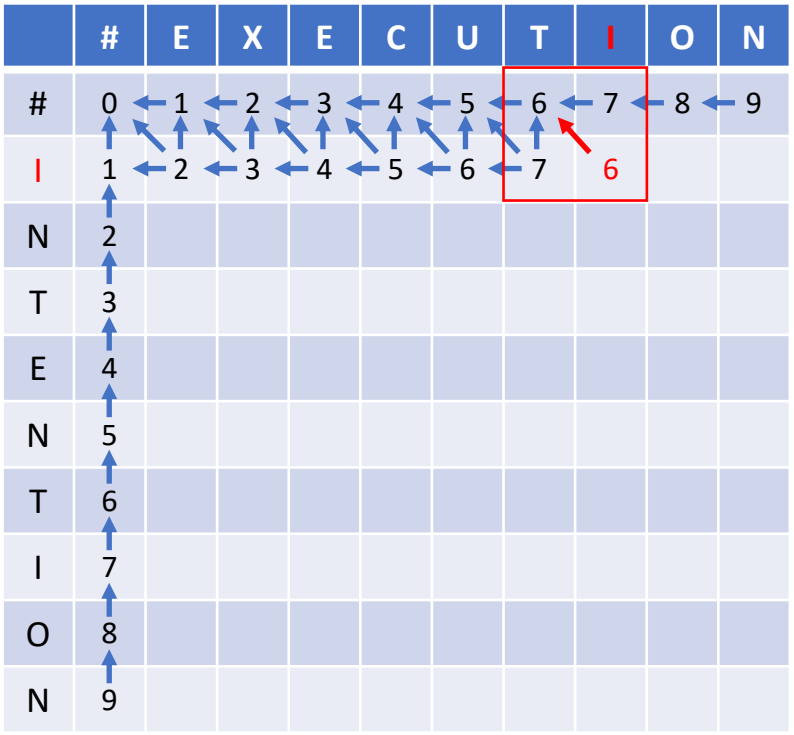
動的計画法による最小編集距離の効率的計算

- 動的計画法 (dynamic programming)
  問題を部分問題に分解して解き結果を再利用することで
  部分問題に対して重複する計算を一度で済ませる

挿入・削除コスト1，置換コスト2の
最小編集距離の計算（動的計画法）

```
fun MinEditDist(src,target):
  # 初期化
  d[[0,0]] = 0
  for i in len (s):
    d[[i,0]] = d[[i-1,0]] + 1 # del.
  for j in len (t):
    d[[0,j]] = d[[0,j-1]] + 1 # ins.
  # 部分問題の結果を利用して構成的に求解
  for i in len (s):
    for j in len (s):
      d[[i,j]] = min (d[[i-1, j]] + 1,
                      d[[i-1, j-1]] + 2,
                      d[[i,j-1]] + 1)
return d
```

src=INTENTION; trg=EXECUTION



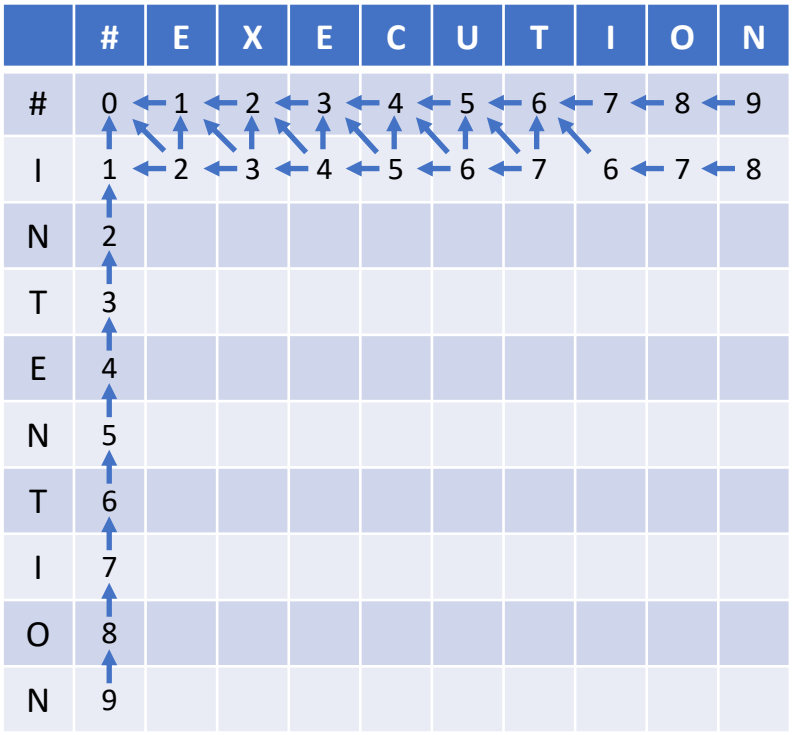
動的計画法による最小編集距離の効率的計算

- 動的計画法 (dynamic programming)
  問題を部分問題に分解して解き結果を再利用することで
  部分問題に対して重複する計算を一度で済ませる

挿入・削除コスト1，置換コスト2の
最小編集距離の計算（動的計画法）

```
fun MinEditDist(src,target):
  # 初期化
  d[[0,0]] = 0
  for i in len (s):
    d[[i,0]] = d[[i-1,0]] + 1 # del.
  for j in len (t):
    d[[0,j]] = d[[0,j-1]] + 1 # ins.
  # 部分問題の結果を利用して構成的に求解
  for i in len (s):
    for j in len (s):
      d[[i,j]] = min (d[[i-1, j]] + 1,
                      d[[i-1, j-1]] + 2,
                      d[[i,j-1]] + 1)
return d
```

src=INTENTION; trg=EXECUTION



動的計画法による最小編集距離の効率的計算

- 動的計画法 (dynamic programming)
  問題を部分問題に分解して解き結果を再利用することで
  部分問題に対して重複する計算を一度で済ませる

挿入・削除コスト1，置換コスト2の
最小編集距離の計算（動的計画法）

```
fun MinEditDist(src,target):
  # 初期化
  d[[0,0]] = 0
  for i in len (s):
    d[[i,0]] = d[[i-1,0]] + 1 # del.
  for j in len (t):
    d[[0,j]] = d[[0,j-1]] + 1 # ins.
  # 部分問題の結果を利用して構成的に求解
  for i in len (s):
    for j in len (s):
      d[[i,j]] = min (d[[i-1, j]] + 1,
                      d[[i-1, j-1]] + 2,
                      d[[i,j-1]] + 1)
return d
```

src=INTENTION; trg=EXECUTION

	#	E	X	E	C	U	T	I	O	N
#	0	1	2	3	4	5	6	7	8	9
I	1	2	3	4	5	6	7	6	7	8
N	2	3	4	5	6	7	8	7	8	7
T	3	4	5	6	7	8	7	8	9	8
E	4	3	4	5	6	7	8	9	10	9
N	5	4	5	6	7	8	9	10	11	10
T	6	5	6	7	8	9	8	9	10	11
I	7	6	7	8	9	10	9	8	9	10
O	8	7	8	9	10	11	10	9	8	9
N	9	8	9	10	11	12	11	10	9	8

動的計画法による最小編集距離の効率的計算

- 動的計画法 (dynamic programming)
  問題を部分問題に分解して解き結果を再利用することで
  部分問題に対して重複する計算を一度で済ませる

挿入・削除コスト1，置換コスト2の
最小編集距離の計算（動的計画法）

```
fun MinEditDist(src,target):
  # 初期化
  d[[0,0]] = 0
  for i in len (s):
    d[[i,0]] = d[[i-1,0]] + 1 # del.
  for j in len (t):
    d[[0,j]] = d[[0,j-1]] + 1 # ins.
  # 部分問題の結果を利用して構成的に求解
  for i in len (s):
    for j in len (s):
      d[[i,j]] = min (d[[i-1, j]] + 1,
                      d[[i-1, j-1]] + 2,
                      d[[i,j-1]] + 1)
return d
```

src=INTENTION; trg=EXECUTION

	#	E	X	E	C	U	T	I	O	N
#	0	1	2	3	4	5	6	7	8	9
I	1	2	3	4	5	6	7	6	7	8
N	2	3	4	5	6	7	8	7	8	7
T	3	4	5	6	7	8	7	8	9	8
E	4	3	4	5	6	7	8	9	10	9
N	5	4	5	6	7	8	9	10	11	10
T	6	5	6	7	8	9	8	9	10	11
I	7	6	7	8	9	10	9	8	9	10
O	8	7	8	9	10	11	10	9	8	9
N	9	8	9	10	11	12	11	10	9	8

動的計画法による最小編集距離の効率的計算

- 動的計画法 (dynamic programming)
  問題を部分問題に分解して解き結果を再利用することで
  部分問題に対して重複する計算を一度で済ませる

挿入・削除コスト1，置換コスト2の
最小編集距離の計算（動的計画法）

```
fun MinEditDist(src,target):
  # 初期化
  d[[0,0]] = 0
  for i in len (s):
    d[[i,0]] = d[[i-1,0]] + 1 # del.
  for j in len (t):
    d[[0,j]] = d[[0,j-1]] + 1 # ins.
  # 部分問題の結果を利用して構成的に求解
  for i in len (s):
    for j in len (s):
      d[[i,j]] = min (d[[i-1, j]] + 1,
                      d[[i-1, j-1]] + 2,
                      d[[i,j-1]] + 1)
return d
```

src=INTENTION; trg=EXECUTION

	#	E	X	E	C	U	T	I	O	N
#	0	1	2	3	4	5	6	7	8	9
I	1	2	3	4	5	6	7	6	7	8
N	2	3	4	5	6	7	8	7	8	7
T	3	4	5	6	7	8	7	8	9	8
E	4	3	4	5	6	7	8	9	10	9
N	5	4	5	6	7	8	9	10	11	10
T	6	5	6	7	8	9	8	9	10	11
I	7	6	7	8	9	10	9	8	9	10
O	8	7	8	9	10	11	10	9	8	9
N	9	8	9	10	11	12	11	10	9	8

コスト計算に使われたセルを辿ることでアラインメントを復元可

本日のまとめ

- 単語＝テキストを計算機で扱う上での基本単位
  - 音韻的，統語的，正書法による定義
  - 未知語問題とサブワード
- テキスト→単語列
  - 文分割
  - 単語分割 (tokenization)
  - 見出し語化 (lemmatization)
- 単語（文字列）の近さの計算
  - 最小編集距離
  - 分布仮説