データ構造とアルゴリズム

第4回　文字列の類似度計算、再帰、動的計画法

メディア情報コース
平居　悠（ひらい　ゆたか）

到達目標

- C言語を通じてデータ構造とアルゴリズムを理解する。
- 目的に応じて効率的なプログラムを自由に設計できる。

前回

第１回　構造体、共用体と線形リスト
第２回　探索とソート
第３回　メモリの動的確保
第４回　文字列の類似度計算、再帰、動的計画法
第５回　自由課題作成
第６回　自由課題作成
第７回　自由課題発表

前回の目標

必要なときに必要な分だけ配列や変数のメモリを確保できるようになる。

前回学んだこと

1. メモリの動的確保
2. メモリ確保領域の大きさの変更

メモリの動的確保

必要なときに変数や配列などのメモリを確保する機能

malloc関数

メモリを取得する。

ヘッダ：#include <stdlib.h>

形式：

```
void *malloc(size_t size);
```

malloc関数

```
void *malloc(size_t size);
```

- 「void *」型：どんな型にもキャストできる。
- sizeには割り当てるバイト数を指定する。

malloc関数

int型1個分の領域を確保する場合：

```
int *pi;
pi = (int *)malloc(sizeof(int));
```

malloc関数

- この関数が失敗したらNULLが返される。
- プログラマはmalloc関数の戻り値がNULLでないことを確認する必要がある。
- 動的に確保したメモリは不要になったら解放しなくてはいけない。

free関数

メモリを解放する。

ヘッダ：#include <stdlib.h>

形式：

```
void free(void *memblock);
```

memblockには、解放するメモリブロックを指定する。

calloc関数

メモリ配列を確保し、0で初期化する。

ヘッダ：#include <stdlib.h>

形式：

```
void *calloc(size_t num, size_t size);
```

calloc関数

```
void *calloc(size_t num, size_t size);
```

- num個の要素を持つ配列を確保する。
- 各要素のサイズはsizeバイト。
- 各要素は0で初期化される。
- 不要になったらfree関数で解放する。

realloc関数

メモリの再確保を行う。

ヘッダ：#include <stdlib.h>

形式：

```
void *realloc(void *memblock, size_t size);
```

realloc関数

```
void *realloc(void *memblock, size_t size);
```

- memblock: 既に割り当てられているメモリブロックへのポインタ。
- size: 新しいサイズをバイト単位で指定。

前回の問い

- 必要な分だけメモリを確保するにはどのようにすれば良いのか？
  - malloc関数を使って動的にメモリを確保する。
- 確保したメモリ領域の大きさを途中で変更するにはどうすれば良いのか？
  - realloc関数を用いる。

今回

第１回　構造体、共用体と線形リスト
第２回　探索とソート
第３回　メモリの動的確保
第４回　文字列の類似度計算、再帰、動的計画法
第５回　自由課題作成
第６回　自由課題作成
第７回　自由課題発表

今回の目標

2つの文字列の類似度を評価する
効率的なプログラムを作成できる
ようになる。

今回学ぶこと

1. 編集距離
2. レーベンシュタイン距離を求めるアルゴリズム
3. レーベンシュタイン距離を計算するプログラム

今回の問い

- 2つの文字列の類似度を調べるにはどのようにすれば良いか？
- 2つの文字列の類似度を調べるプログラムを設計するにはどのようなアプローチがあるか？

今回学ぶこと

1. 編集距離
2. レーベンシュタイン距離を求めるアルゴリズム
3. レーベンシュタイン距離を計算するプログラム

文字列の類似度評価

2つの文字列がどのくらい似ているか評価するには？



一致している文字数を数える

例：きっそう
　　うそつき



100%一致

文字は同じだが順番が異なる場合を評価できない。

文字列の類似度評価

2つの文字列がどのくらい似ているか評価するには？



順番に文字を比べて、一致している数を数える

例：**きっそう**
　　うそつき



0%一致

編集距離 (edit distance)

2つの文字列がどれくらい異なっているかを評価するための計量。自然言語処理（人間の言葉をコンピュータで分析する技術）で頻繁に使われる。

最も広く使われる編集距離の定義は、情報理論学者のウラジミール・レーベンシュタインが提案した「**レーベンシュタイン距離**」である。

レーベンシュタイン距離

2つの文字列の間のレーベンシュタイン距離を計算するには、1つの文字列をもう1つの文字列に変換するのに必要な1文字の最小回数を求める。

レーベンシュタイン距離

実行できる編集の種類：

- 1文字の**挿入** (例：cat→cast)
- 1文字の**削除** (例：bait→bat)
- 1文字の**置換** (例：cut→cat)

レーベンシュタイン距離の例

- 「おつかれさん」と「つかれました」のレーベンシュタイン距離＝4
- 「computer」と「Come Peter」のレーベンシュタイン距離＝5（スペースの挿入も1として数えられる）

今回の問い

- **2つの文字列の類似度を調べるにはどのようにすれば良いか？**
  - **編集距離を用いる。**
- 2つの文字列の類似度を調べるプログラムを設計するにはどのようなアプローチがあるか？

今回学ぶこと

1. 編集距離
2. レーベンシュタイン距離を求めるアルゴリズム
3. レーベンシュタイン距離を計算するプログラム

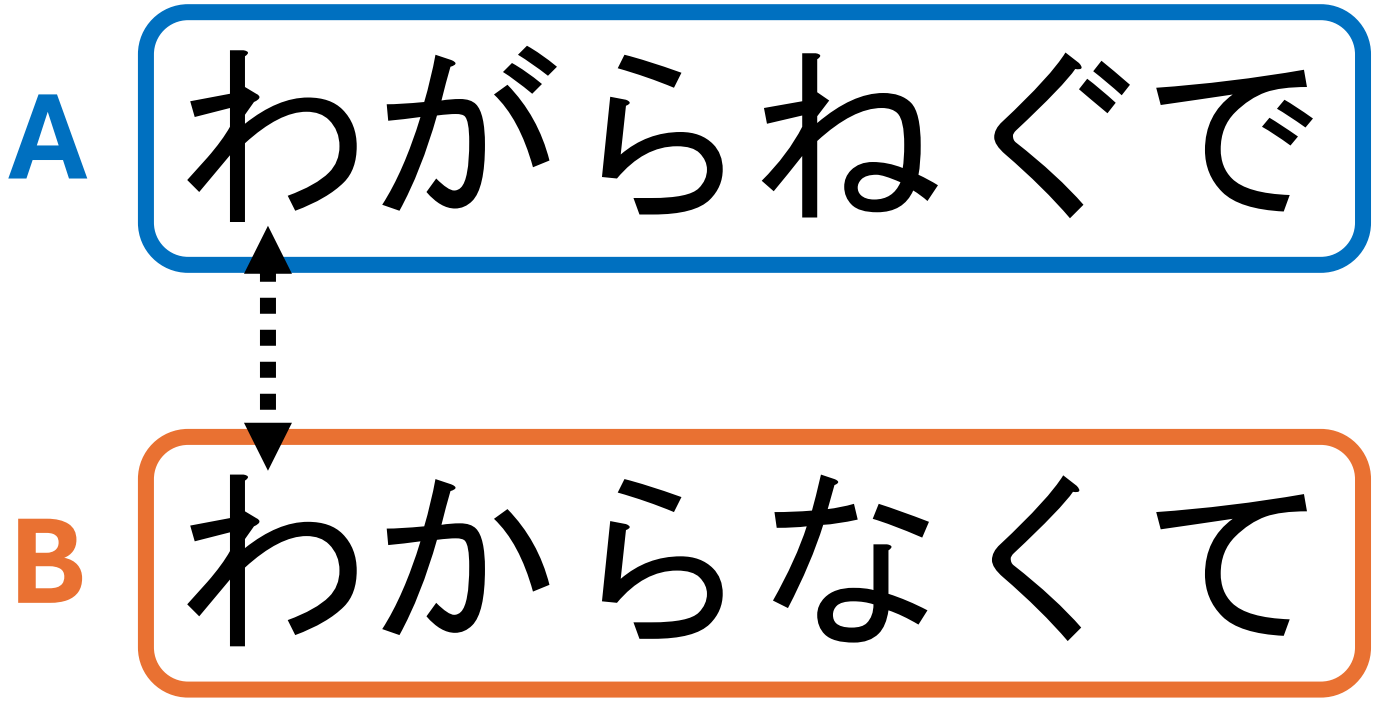
編集距離のアルゴリズム

①文字列AとBのどちらかが空なら、もう一方の長さを編集距離に足して終了



編集距離のアルゴリズム

②文字列AとBの1文字目を比べる



編集距離のアルゴリズム

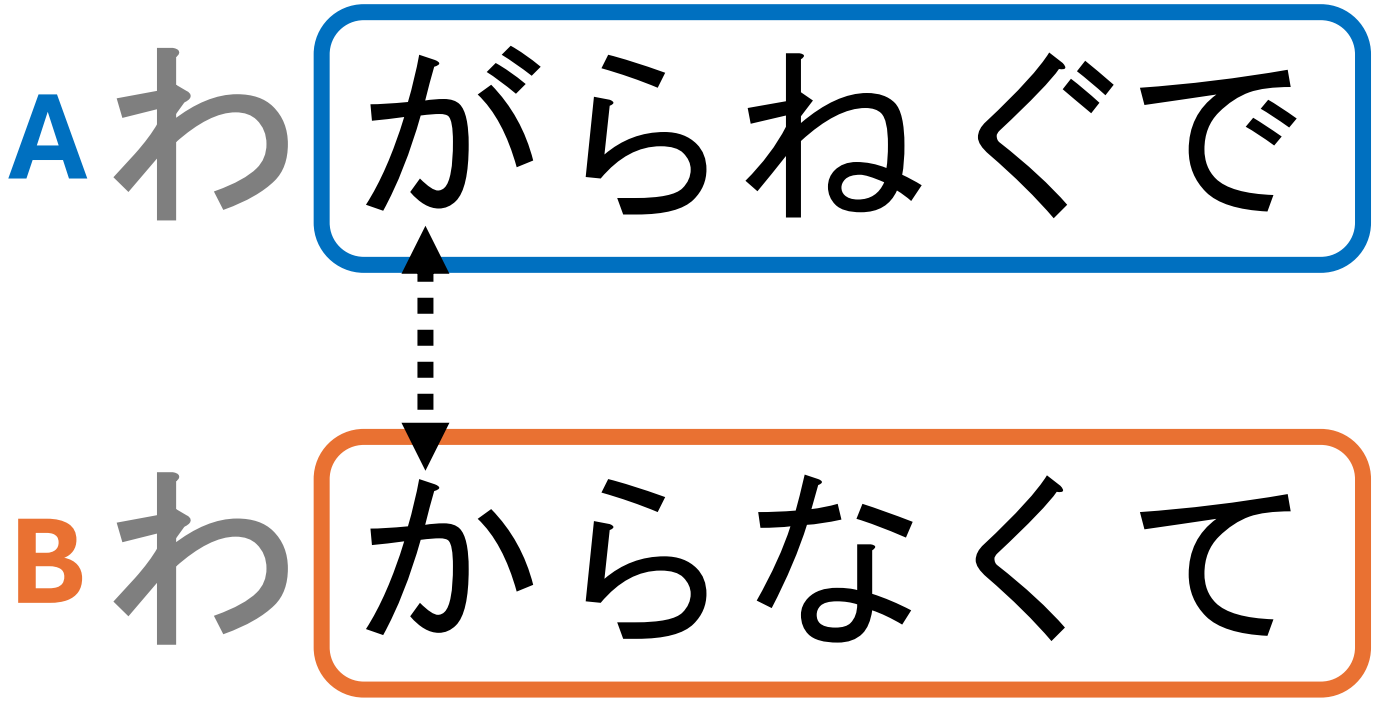
③同じなら次の文字に進んで①へ戻る

A わ がらねぐで

B わ からなくて

編集距離のアルゴリズム

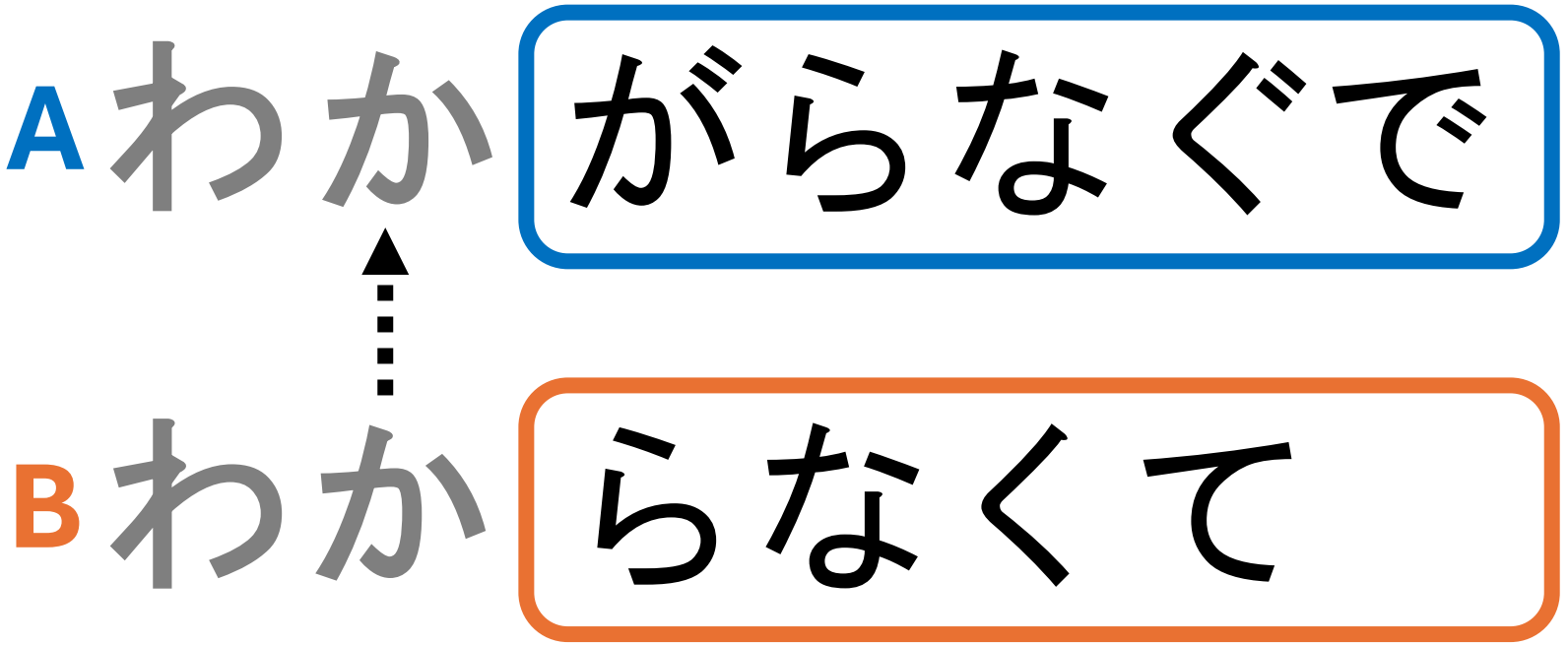
④異なるなら編集距離に1を足す。



編集距離のアルゴリズム

⑤3つの編集距離を計算する。

5-1 Bから1文字を削除 (=挿入)し①へ戻る



編集距離のアルゴリズム

⑤3つの編集距離を計算する。

5-2 Aから1文字を削除し①へ戻る

A わ

らねぐで

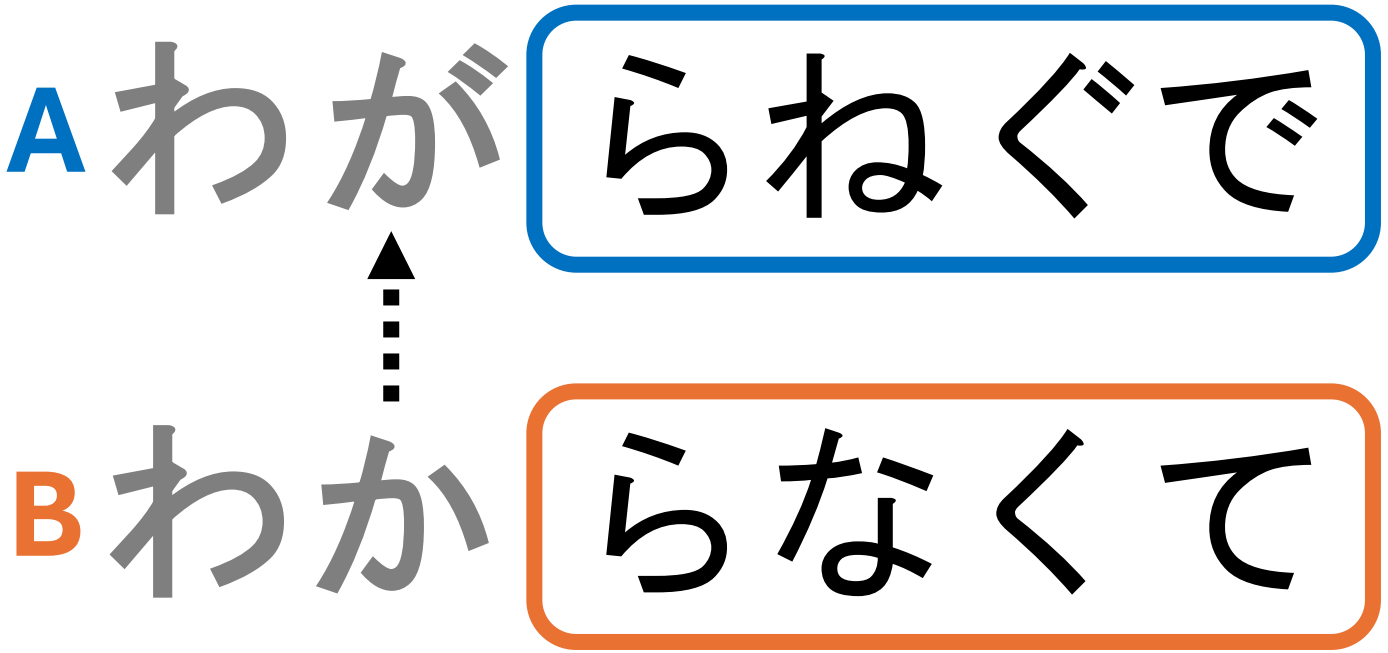
B わ

からなくて

編集距離のアルゴリズム

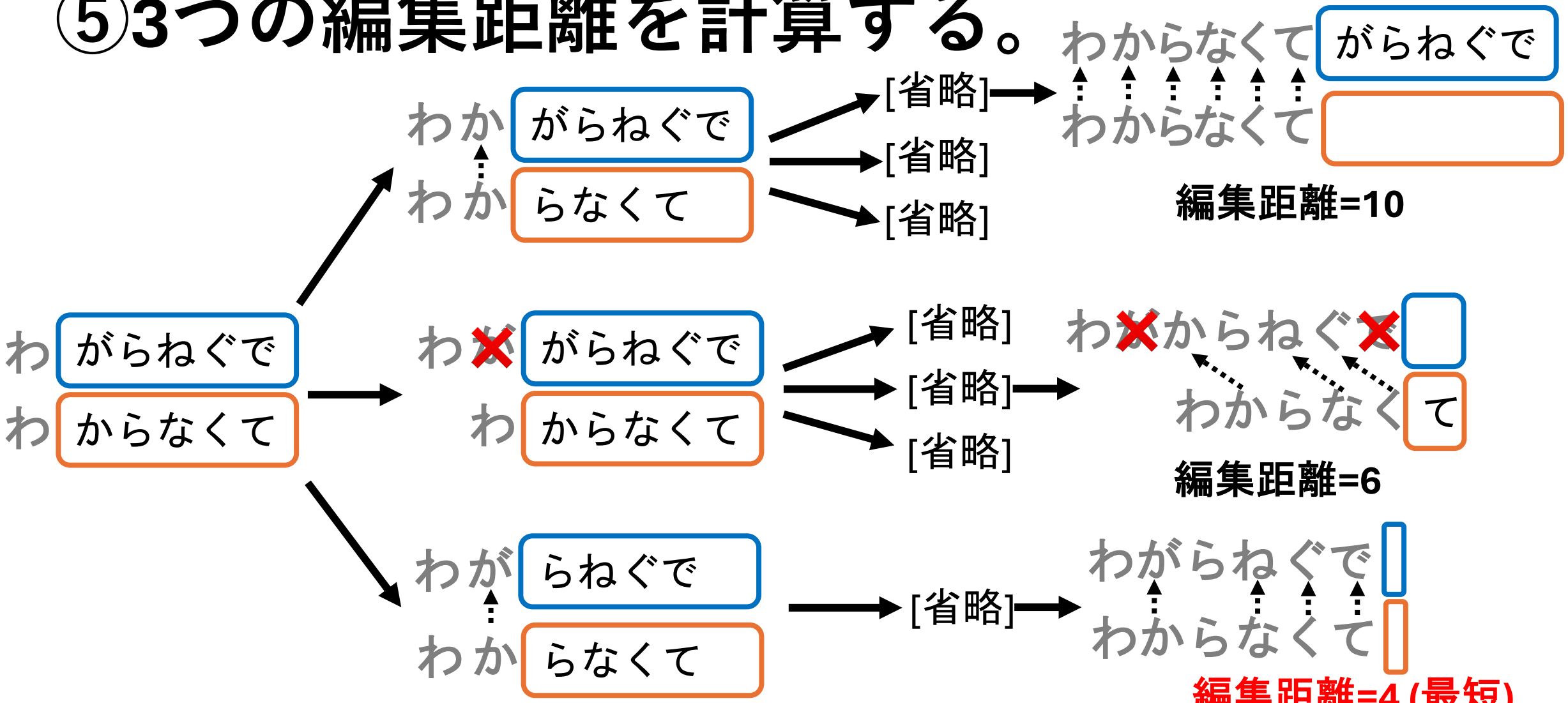
⑤3つの編集距離を計算する。

5-3 AとBから1文字を削除(=置換)し①へ戻る



編集距離のアルゴリズム

⑤3つの編集距離を計算する。



全ての計算が終わったら、最短の距離を結果とする。

今回学ぶこと

1. 編集距離
2. レーベンシュタイン距離を求めるアルゴリズム
3. **レーベンシュタイン距離を計算するプログラム**

再帰的関数を使ったアプローチ

編集距離の計算のように、簡単な作業を何回も繰り返しその結果を組み合わせることによって元の複雑な問題を解決できる場合、再帰処理を利用することが多い。

次ページのプログラム、levenshtein.cを作成し、実行せよ。

```c
#include <stdio.h>
#include <string.h>

int lowest(int values[], int n) { /* 整数の配列の中から最小値を返す */
   int lowest = values[0];
   int i = 0;
   for (i = 1; i < n; i++) {
      if (values[i] < lowest) {
         lowest = values[i];
      }
   }
   return lowest;
}

int eddist(char *str1, char *str2, int pos1, int pos2) {
   /* 編集距離（レーベンシュタイン距離）を計算する関数
   str1, str2 は対象の文字列
   pos1, pos2 は現在見ている文字の位置（添え字+1）*/
   int opt[3];
   if (pos1 == 0) { /* str1の終わりに到達した場合、str2の残りの文字数を返す */
      return pos2;
   } else if (pos2 == 0) { /* str2の終わりに到達した場合、str1の残りの文字数を返す */
      return pos1;
   } else if (str1[pos1-1] == str2[pos2-1]) { /* 両方の文字列が対象の位置に同じ文字を持つ場合、次の位置に移動する */
      return eddist(str1, str2, pos1-1, pos2-1);
   } else {
      opt[0] = eddist(str1, str2, pos1-1, pos2); /* str1から1文字を削除することと同じ */
      opt[1] = eddist(str1, str2, pos1, pos2-1); /* str1に1文字を挿入することと同じ */
      opt[2] = eddist(str1, str2, pos1-1, pos2-1); /* str1の1文字を置き換えることと同じ */
      return 1 + lowest(opt, 3);
   }
}
```

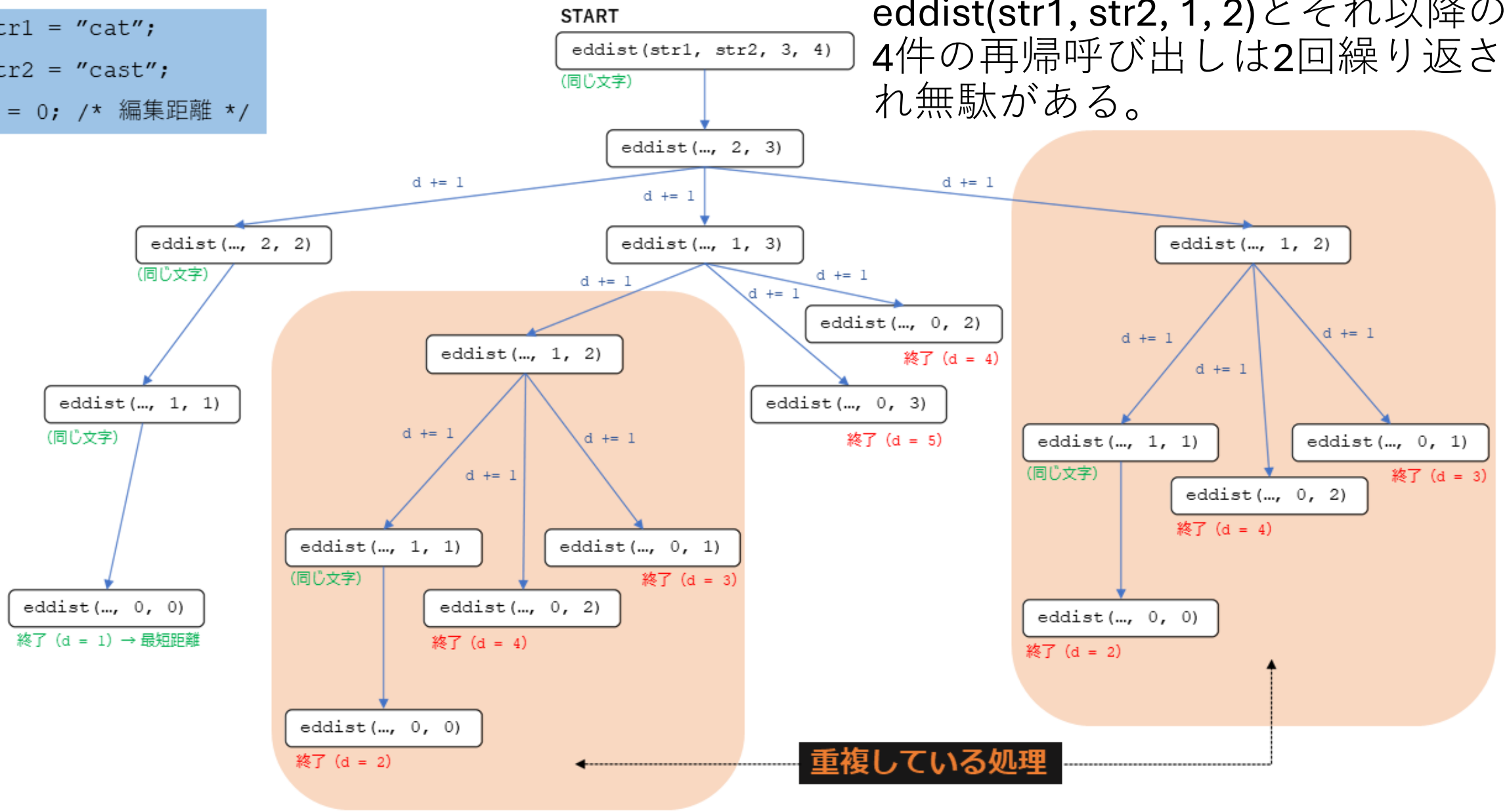
```c
int main()
{
   int dist;
   char str1[] = "programming";
   char str2[] = "promising";
   dist = eddist(str1, str2, strlen(str1), strlen(str2));
   printf("Edit distance: %d\n", dist);

   return 0;
}
```

“cat”と“cast”の距離を求める場合

```
str1 = "cat";
str2 = "cast";
d = 0; /* 編集距離 */
```

eddist(str1, str2, 1, 2)とそれ以降の4件の再帰呼び出しは2回繰り返され無駄がある。



再帰的関数を使ったアプローチの問題点

再帰呼び出しを行う度に処理が分岐する木構造になっているため、対象の文字列が長くなるに伴い必要な計算量が爆発的に増大してしまう。

動的計画法を使ったアプローチ

このような無駄を省くためには、一度計算した結果を保存し後で再び同じ処理を行う必要があれば記録された結果を再利用すれば良い。

このような方法は「**動的計画法**」または、「**ダイナミック・プログラミング**」と呼ばれる。

動的計画法を使ったアプローチ

次ページのプログラム (levenshteinDP.c) は、計算済みの編集距離をmemoという2次元配列に保存することによって処理速度の大幅な高速化を実現している。

手元の端末でlevenshteinDP.cを書いて実行してみよう。

```c
#include <stdio.h>
#include <string.h>

#define MAX_LEN 100 /* 対応できる文字列の長さの上限 */

int lowest(int values[], int n) { /* 整数の配列の中から最小値を返す */
  int lowest = values[0];
  int i = 0;
  for (i = 1; i < n; i++) {
    if (values[i] < lowest) {
      lowest = values[i];
    }
  }
  return lowest;
}

int eddist(char *str1, char *str2, int pos1, int pos2, int memo[MAX_LEN+1][MAX_LEN+1]) {
  /* 編集距離（レーベンシュタイン距離）を計算する関数
  str1, str2 は対象の文字列
  pos1, pos2 は現在見ている文字の位置（添え字+1）*/
  int opt[3];
  int dist;
  if (memo[pos1][pos2] > 0) {
    return memo[pos1][pos2];
  } else if (pos1 == 0) { /* str1の終わりに到達した場合、str2の残りの文字数を返す */
    return pos2;
  } else if (pos2 == 0) { /* str2の終わりに到達した場合、str1の残りの文字数を返す */
    return pos1;
  } else if (str1[pos1-1] == str2[pos2-1]) { /* 両方の文字列が対象の位置に同じ文字を持つ場合、次の位置に移動する */
    return eddist(str1, str2, pos1-1, pos2-1, memo);
  } else {
    opt[0] = eddist(str1, str2, pos1-1, pos2, memo); /* str1から1文字を削除することと同じ */
    opt[1] = eddist(str1, str2, pos1, pos2-1, memo); /* str1に1文字を挿入することと同じ */
    opt[2] = eddist(str1, str2, pos1-1, pos2-1, memo); /* str1の1文字を置き換えることと同じ */
    dist = 1 + lowest(opt, 3);
    memo[pos1][pos2] = dist;
    return dist;
  }
}

int main()
{
  int dist;
  char str1[] = "programming";
  char str2[] = "promising";
  int memo[MAX_LEN+1][MAX_LEN+1] = {0}; /* 全ての要素を 0 に初期化 */
  dist = eddist(str1, str2, strlen(str1), strlen(str2), memo);
  printf("Edit distance: %d\n", dist);

  return 0;
}
```

スタックフレーム

プログラムの中で関数が呼び出されると、メモリのスタック領域で「**スタックフレーム**」と呼ばれるブロックが割り当てられ、そこで関数の処理に必要な情報（関数の中で使われる変数など）が格納される。

例えば、右のプログラムを実行すればスタックメモリは次ページ図のように使用される。

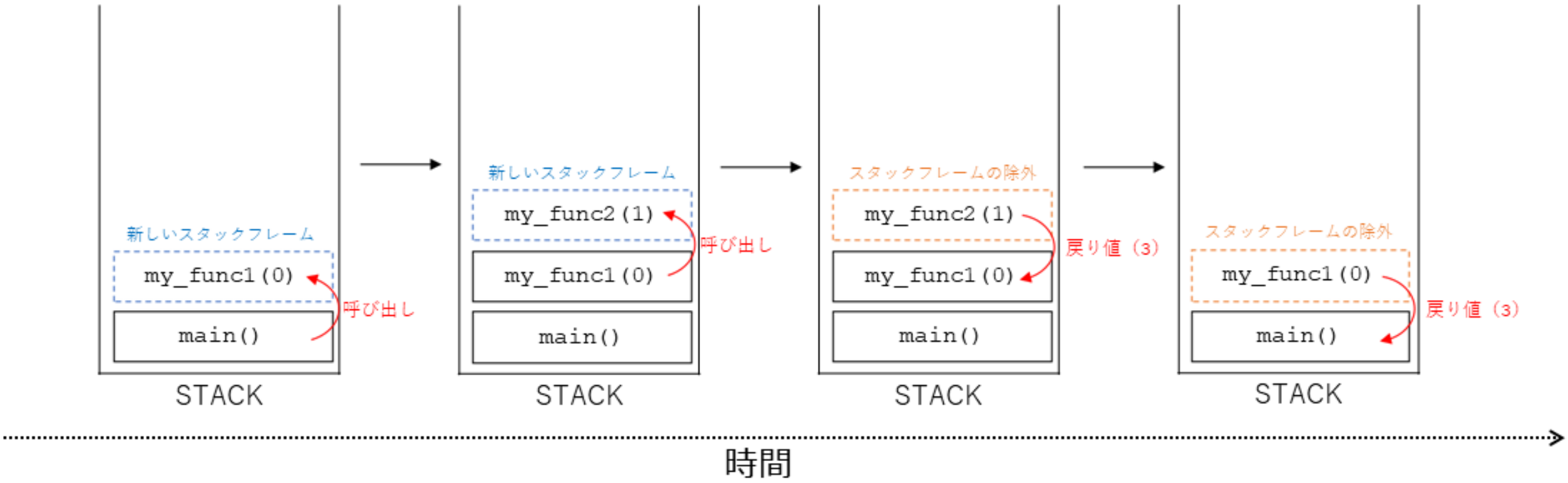
```
#include <stdio.h>

int my_func2(int z) {
   z += 2;
   return z; /* my_func2 のスタックフレームが除外される */
}

int my_func1(int y) {
   y++;
   y = my_func2(y); /* 新しいスタックフレームが作成される */
   return y; /* my_func1 のスタックフレームが除外される */
}

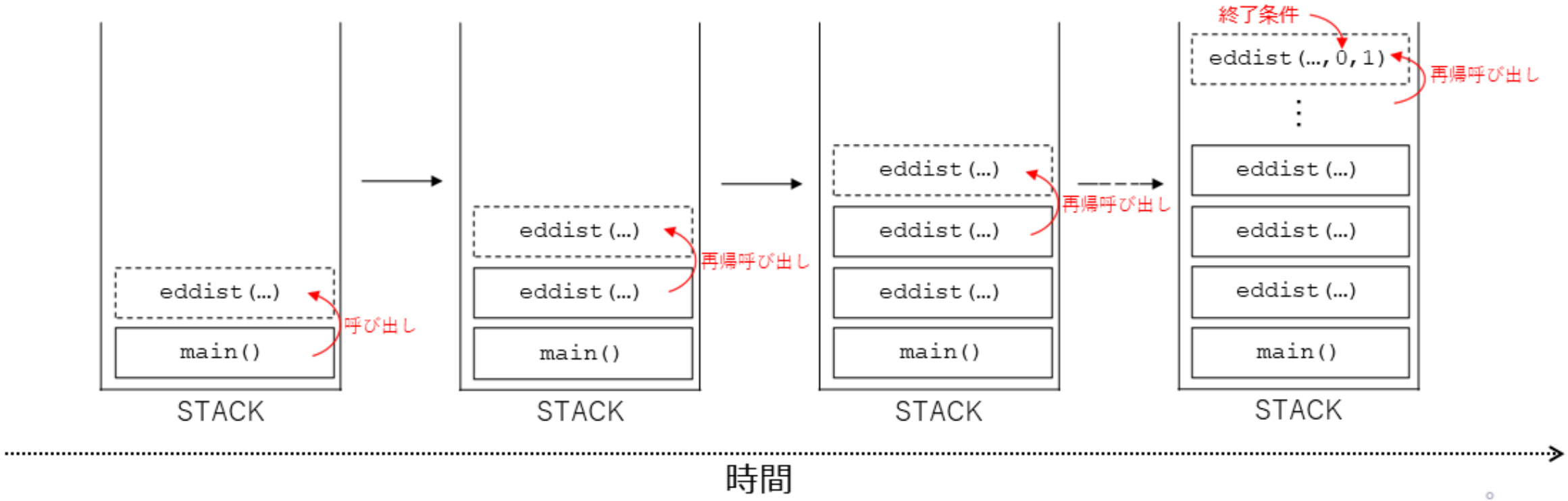
int main() {
   int x = 0;
   my_func1(x); /* 新しいスタックフレームが作成される */
   return 0; /* main のスタックフレームが除外される */
}
```

スタックフレーム



関数の実行が終了した後はスタックフレームが除外され解放されたメモリを再利用できる。

スタックフレーム



再帰的に定義された関数の場合、関数から抜ける前に次の呼び出しが行われるので、再帰の終了条件が成立するまでメモリ消費量が増加してしまう。

スタックオーバーフロー

再帰が深くなるとスタックが溢れてしまい（**スタックオーバーフロー**）プログラムの実行が中断される恐れがある。

ループを使ったアプローチ

for文やwhile文などによる反復処理（ループ）の場合は、処理を繰り返す度に同じスタックフレームが再利用されるため、オーバーフローする心配がない。

ループを使ったアプローチ

このような再帰処理の弱点を踏まえて、長い文字列にも対応できる編集距離の計算プログラムを作りたいならループを使って実装した方が良い。

ループを使ったアプローチ

具体的には、ダイナミック・プログラミングを使ったプログラムのmemo (計算結果を記録する表)を一つのループの中で作成する仕組みにすれば再帰呼び出しを行う必要がなくなる。

ループを使ったアプローチ

次ページの再帰呼び出しを行わずにレーベンシュタイン距離を計算するプログラム(levenshtein_loop.c)を書いて実行してみよう。

```
#include <stdio.h>
#include <string.h>

#define MAX_LEN 100 /* 対応できる文字列の長の上限 */

int lowest(int values[], int n) { /* 整数の配列の中から最小値を返す */
  int lowest = values[0];
  int i = 0;
  for (i = 1; i < n; i++) {
    if (values[i] < lowest) {
      lowest = values[i];
    }
  }
  return lowest;
}
```

```
int eddist_iter(char *str1, char *str2, int len1, int len2) {
  /* （再帰呼び出しを使わずに）編集距離（レーベンシュタイン距離）を計算する関数
  str1, str2 は対象の文字列
  len1, len2 は文字列の長さ */
  int matrix[MAX_LEN+1][MAX_LEN+1];
  int i, j;
  int opt[3];

  for (i = 0; i <= len1; i++) {
    for (j = 0; j <= len2; j++) {
      /* str1の終端に到達した場合、str2の残りの文字数を返す */
      if (i == 0) {
        matrix[i][j] = j;
      /* str2の終端に到達した場合、str1の残りの文字数を返す */
      } else if (j == 0) {
        matrix[i][j] = i;
      /* 両方の文字列が対象の位置に同じ文字を持つ場合、編集距離は増えない */
      } else if (str1[i-1] == str2[j-1]) {
        matrix[i][j] = matrix[i-1][j-1];
      /* 対象の位置にある文字が異なる場合、可能な編集を全て検討し
      最小値を求める */
      } else {
        opt[0] = matrix[i][j-1]; /* str1に1文字を挿入することと同じ */
        opt[1] = matrix[i-1][j]; /* str1から1文字を削除することと同じ */
        opt[2] = matrix[i-1][j-1]; /* str1の1文字を置き換えることと同じ */
        matrix[i][j] = 1 + lowest(opt, 3);
      }
    }
  }
  /* 処理が終わったら表の最後のセルに最小距離が入っているのでそれを返す */
  return matrix[len1][len2];
}
```

```
int main()
{
  int dist;
  char str1[] = "programming";
  char str2[] = "promising";
  dist = eddist_iter(str1, str2, strlen(str1), strlen(str2));
  printf("Edit distance: %d\n", dist);

  return 0;
}
```

メモリ消費の改善

メモリ消費の面でもっと深刻な問題がある。計算結果を記録する2次元配列 (matrix)に必要なメモリ容量はMAX_LENの二乗に比例している。

例えば、最大で1,000文字までの文字列の編集距離を計算したいなら、1,000×1,000 = 1,000,000セルの表を格納できるメモリを確保しなければならない。

メモリ消費の改善

levenshtein_loop.cでは、実際に参照されるのは現在の添字 (i, j)要素か、その直前 (i−1, j−1)の要素だけである。

この性質を利用すれば、2行だけの表で同じ処理を行うことができる。

メモリ消費の改善

具体的には、内側のループで今回新たに求まる値を1行に、もう1行には前回の結果を保管し外側のループで常に更新する。

このように改良したプログラムのメモリ消費はMAX_LENの二乗ではなくMAX_LENに比例する。

今回の問い

- 2つの文字列の類似度を調べるにはどのようにすれば良いか？
  - 編集距離を用いる。
- **2つの文字列の類似度を調べるプログラムを設計するにはどのような方法があるか？**
  - **再帰的関数や動的計画法を使った方法がある。**

今回学んだこと

1. 編集距離
2. レーベンシュタイン距離を求めるアルゴリズム
3. レーベンシュタイン距離を計算するプログラム

今回の目標

2つの文字列の類似度を評価する
効率的なプログラムを作成できる
ようになる。

課題

次ページのプログラムはメモリ消費を改善したlevenshtein_loop2.cである。このプログラムを編集し、ユーザーが入力した任意の文字列について編集距離を計算するプログラムalgorithm4.cを作成せよ。

提出先：yutaka.hirai@koeki-u.ac.jp

件名：データ構造とアルゴリズム第4回課題 [学籍番号]

提出物：algorithm4.c

締め切り：12月24日 (木)

```c
#include <stdio.h>
#include <string.h>

#define MAX_LEN 100 /* 対応できる文字列の長さ
の上限 */

void swap_pointers(int **p2p1, int **p2p2) {
  /* ２つのポインタが指すアドレスを交換する
  int *temp = *p2p1;
  *p2p1 = *p2p2;
  *p2p2 = temp;
}


int lowest(int values[], int n) { /* 整数の配列の中か
ら最小値を返す */
  int lowest = values[0];
  int i = 0;
  for (i = 1; i < n; i++) {
    if (values[i] < lowest) {
      lowest = values[i];
    }
  }
  return lowest;
}
```

```c
int eddist_iter(char *str1, char *str2, int len1, int len2) {
  /* （再帰呼び出しを使わずに）編集距離（レーベンシュタイン距離）を計算する関数
  str1, str2 は対象の文字列
  len1, len2 は文字列の長さ */
  int row1[MAX_LEN+1] = {0};
  int row2[MAX_LEN+1] = {0};
  int *prev = row1; /* 直前の行 */
  int *current = row2; /* 今見ている行 */
  int i, j;
  int opt[3];

  for (i = 0; i <= len1; i++) {
    for (j = 0; j <= len2; j++) {
      /* str1の終端に到達した場合、str2の残りの文字数を返す */
      if (i == 0) {
        current[j] = j;
      /* str2の終端に到達した場合、str1の残りの文字数を返す */
      } else if (j == 0) {
        current[j] = i;
      /* 両方の文字列が対象の位置に同じ文字を持つ場合、編集距離は増えない */
      } else if (str1[i-1] == str2[j-1]) {
        current[j] = prev[j-1];
      /* 対象の位置にある文字が異なる場合、可能な編集を全て検討し
      最小値を求める */
      } else {
        opt[0] = current[j-1]; /* str1に1文字を挿入することと同じ */
        opt[1] = prev[j]; /* str1から1文字を削除することと同じ */
        opt[2] = prev[j-1]; /* str1の1文字を置き換えることと同じ */
        current[j] = 1 + lowest(opt, 3);
      }
    }
    swap_pointers(&prev, &current); /* 今回調べた行（current）が今度 prev になる */
  }
  /* 処理が終わったら prev の最後の要素に最小距離が入っているのでそれを返す */
  return prev[len2];
}
```

```c
int main()
{
  int dist;
  char str1[] = "programming";
  char str2[] = "promising";
  dist = eddist_iter(str1, str2, strlen(str1), strlen(str2));
  printf("Edit distance: %d\n", dist);

  return 0;
}
```

次回

第１回　構造体、共用体と線形リスト
第２回　探索とソート
第３回　メモリの動的確保
第４回　文字列の類似度計算、再帰、動的計画法
第５回　自由課題作成
第６回　自由課題作成
第７回　自由課題発表