МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Вятский государственный университет»
Факультет автоматики и вычислительной техники
Кафедра электронных вычислительных машин

Лабораторная работа № 3 по курсу «Технологии программирования»

Выполнил студент группы ИВТ-21 _____/Рзаев А. Э./
Проверил доцент кафедры ЭВМ _____/Долженкова М. Л./

Киров 2017

1 Задание

Написать программу с использованием WinAPI для поиска различий по словам в двух текстовых файлах. На главном окне приложения должны быть расположены компоненты для выбора текстовых файлов для сравнения и запуска процесса сравнения. В двух дочерних модальных окнах должно отображаться содержимое выбранных текстовых файлов, в которых отличающиеся отрывки переведены в верхний регистр.

2 Результат работы

Экранные формы приведены в приложении А.

3 Листинг программы

Листинг разработанной программы приведен в приложении Б.

4 Вывод

В ходе выполнения лабораторной работы была написана программа с использованием функций WinAPI. В программа был реализован поиск различий по словам в двух текстовых файлах, используя алгоритм нахождения наибольшей общей подпоследовательности. Для сравнения между собой слов использовалась хеш-функция. В случае, если оба текста совпадают, выводится сообщение о совпадении.

Приложение А
(обязательное)
Экранные формы

Рисунок 1 – Главное окно программы

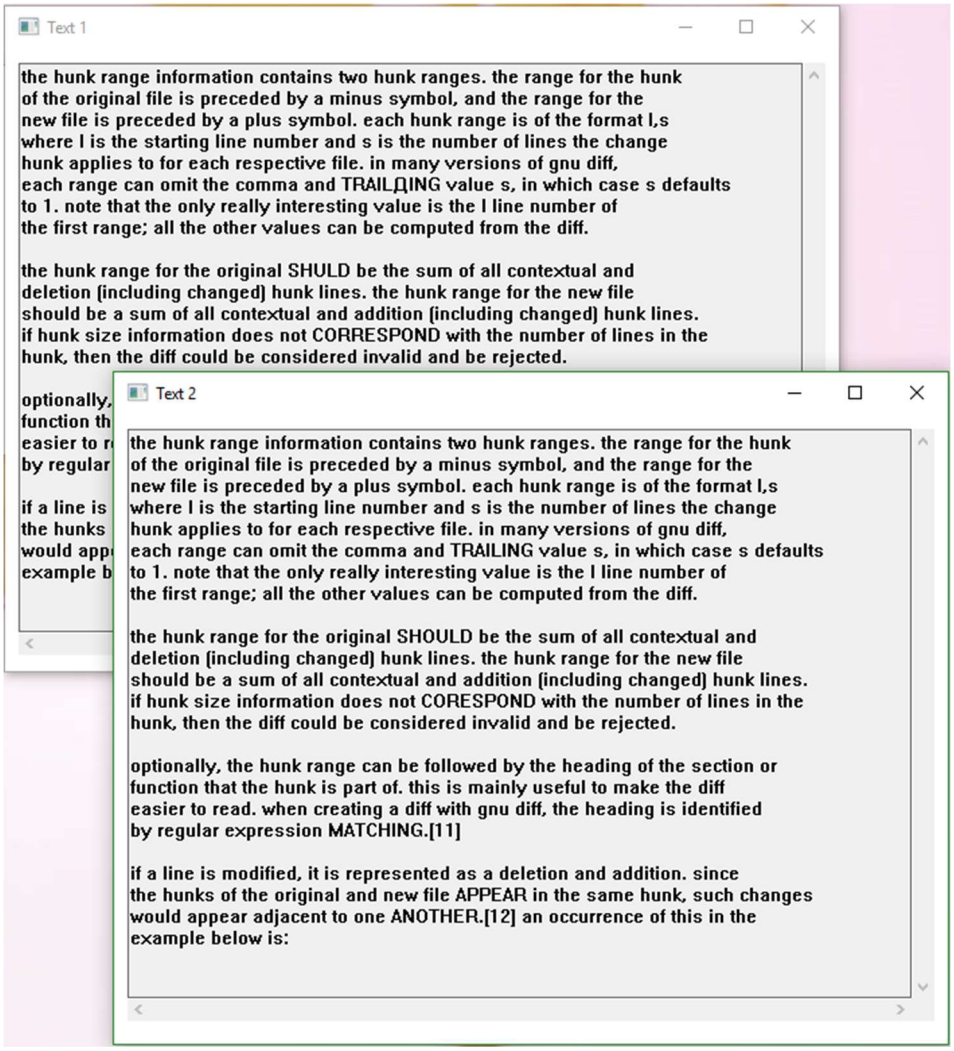

Рисунок 2 – Дочерние окна с содержимым файлов

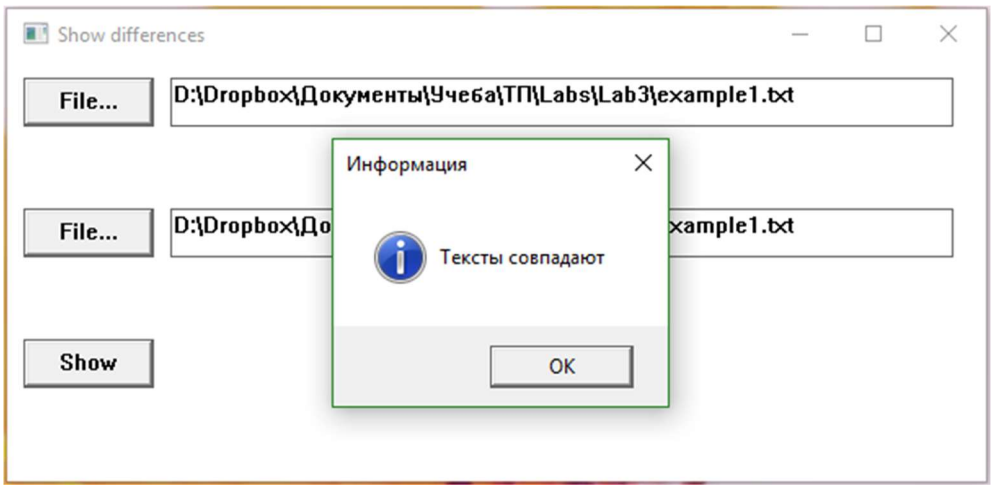

Рисунок 3 – Сообщение об одинаковых файлах

Приложение Б
(обязательное)
Листинг программы

main.cpp

```
#include <Windows.h>
#include <bits\stdc++.h>
#include "TextRoutines.h"
#include "FileDialog.h"
#include "Widgets.h"

using std::make_shared;
using std::make_tuple;
using std::shared_ptr;
using std::tuple;
using std::tie;
using std::endl;

std::ofstream logs("logs.txt");

void WmCommandProc(WPARAM wParam, LPARAM
lParam) {
  HWND hwnd = (HWND)lParam;
  switch (LOWORD(wParam)) {
  case BN_CLICKED:
    getWidget(hwnd)->onClick(wParam,
lParam);
    break;
  default:
    break;
  }
}

LRESULT CALLBACK WndProc(
  HWND hwnd,
  UINT msg,
  WPARAM wParam,
  LPARAM lParam) {

  switch (msg) {
  case WM_CLOSE:
    getWidget(hwnd)->hide();
    break;
  case WM_COMMAND:
    WmCommandProc(wParam, lParam);
    break;
  case WM_DESTROY:
    PostQuitMessage(0);
    break;
  default:
    return DefWindowProc(hwnd, msg,
wParam, lParam);
  }
  return 0;
}

tuple<
  shared_ptr<W32Window>,
  shared_ptr<W32TextEdit>>
CreateTextViewWindow(
  const char* title,
  shared_ptr<W32Application> app,
  shared_ptr<W32Window> parent) {
  auto fileTextWindow =
make_shared<W32Window>(title, 640, 520,
*app, *parent);
  auto fileTextView =
make_shared<W32TextEdit>(600, 440, 10,
10, *fileTextWindow);

  addWidget(fileTextWindow);
  addWidget(fileTextView);

  return make_tuple(fileTextWindow,
fileTextView);
}

tuple<
  shared_ptr<W32Window>,
  shared_ptr<W32Button>,
  shared_ptr<W32TextField>,
  shared_ptr<W32Button>,
  shared_ptr<W32TextField>,
  shared_ptr<W32Button>>
CreateMainWindow(shared_ptr<W32Applicatio
n> app) {
  auto mainWindow =
make_shared<W32Window>("Show
differences", 620, 300, *app);
  auto openTextFile1 =
make_shared<W32Button>("File...", 80, 30,
10, 10, *mainWindow);
  auto filePathEdit1 =
make_shared<W32TextField>(480, 30, 100,
10, *mainWindow);
  auto openTextFile2 =
make_shared<W32Button>("File...", 80, 30,
10, 90, *mainWindow);
  auto filePathEdit2 =
make_shared<W32TextField>(480, 30, 100,
90, *mainWindow);
  auto openFilesButton =
make_shared<W32Button>("Show", 80, 30,
10, 170, *mainWindow);
```

```
  addWidget(mainWindow);
  addWidget(openTextFile1);
  addWidget(filePathEdit1);
  addWidget(openTextFile2);
  addWidget(filePathEdit2);
  addWidget(openFilesButton);

  openTextFile1-
>setOnClickHandler([mainWindow,
filePathEdit1](WPARAM w, LPARAM l) {
    auto path =
getOpenFileName(*mainWindow, "All\0
* .*\0Text\0 * .txt\0");
    filePathEdit1->setText(path);
  });

  openTextFile2-
>setOnClickHandler([mainWindow,
filePathEdit2](WPARAM w, LPARAM l) {
    auto path =
getOpenFileName(*mainWindow, "All\0
* .*\0Text\0 * .txt\0");
    filePathEdit2->setText(path);
  });

  return make_tuple(mainWindow,
openTextFile1, filePathEdit1,
openTextFile2, filePathEdit2,
openFilesButton);
}

shared_ptr<W32Application>
CreateApplication(HINSTANCE hInstance,
int nCmdShow) {
  auto app =
make_shared<W32Application>("Lab3",
WndProc, hInstance);

  // main window
  shared_ptr<W32Window> mainWindow;
  shared_ptr<W32Button> openTextFile1;
  shared_ptr<W32TextField> filePathEdit1;
  shared_ptr<W32Button> openTextFile2;
  shared_ptr<W32TextField> filePathEdit2;
  shared_ptr<W32Button> openFilesButton;
  tie(
    mainWindow,
    openTextFile1,
    filePathEdit1,
    openTextFile2,
    filePathEdit2,
    openFilesButton) =
CreateMainWindow(app);

  // first text view window: tuple
<window, textedit>
  auto fileTextWindow1 =
CreateTextViewWindow("Text 1", app,
mainWindow);

  // second text view window: tuple
<window, textedit>
  auto fileTextWindow2 =
CreateTextViewWindow("Text 2", app,
mainWindow);

  openFilesButton-
>setOnClickHandler([=](WPARAM w, LPARAM
l) {
    std::hash<std::string> hash;
    auto text1 =
LoadTextFromFile(filePathEdit1->text());
    auto text2 =
LoadTextFromFile(filePathEdit2->text());

    for (auto& c : text1) {
      c = ansi_toupper(c);
    }
    for (auto& c : text2) {
      c = ansi_toupper(c);
    }

    if (hash(text1) == hash(text2)) {
      MessageBox(
        mainWindow->hwnd(),
        "Òåêñòû ñîâïàäàþò",
        "Èíôîðìàöèÿ",
        MB_ICONINFORMATION);
      return;
    }

    auto diffs = ShowDifferences(text1,
text2);

    std::get<1>(fileTextWindow1)-
>setText(diffs.first);
    std::get<1>(fileTextWindow2)-
>setText(diffs.second);
    std::get<1>(fileTextWindow1)-
>setReadOnly(TRUE);
    std::get<1>(fileTextWindow2)-
>setReadOnly(TRUE);

    std::get<0>(fileTextWindow1)-
>setModal(TRUE);
    std::get<0>(fileTextWindow2)-
>setModal(TRUE);
    std::get<0>(fileTextWindow1)->show();
    std::get<0>(fileTextWindow2)->show();
  });

  mainWindow->show(nCmdShow);
```

```
  return app;
}

int WINAPI WinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPSTR lpCmdLine,
  int nCmdShow) {

  auto app = CreateApplication(hInstance,
nCmdShow);

  return app->exec();
}
```

Widgets.h

```
#pragma once
#include <bits\stdc++.h>
#include <Windows.h>

using std::make_shared;
using std::shared_ptr;
using std::endl;

class W32Widget;
shared_ptr<W32Widget> getWidget(HWND
hwnd);

class WException : public std::exception
{
public:
  WException() = default;
  WException(const char* msg) :
std::exception(msg) {}
};

class W32Application {
private:
  WNDCLASSEX mWndCl;
public:
  W32Application() = delete;
  W32Application(const char* name,
WNDPROC wndProc, HINSTANCE hInstance) {
    mWndCl.cbSize = sizeof(WNDCLASSEX);
    mWndCl.style = 0;
    mWndCl.lpfnWndProc = wndProc;
    mWndCl.cbClsExtra = 0;
    mWndCl.cbWndExtra = 0;
    mWndCl.hInstance = hInstance;
    mWndCl.hIcon = LoadIcon(NULL,
IDI_APPLICATION);
    mWndCl.hCursor = LoadCursor(NULL,
IDC_ARROW);
    mWndCl.hbrBackground =
(HBRUSH)(COLOR_WINDOW + 1);
    mWndCl.lpszMenuName = NULL;
    mWndCl.lpszClassName = name;
    mWndCl.hIconSm = LoadIcon(NULL,
IDI_APPLICATION);

    if (!RegisterClassEx(&mWndCl)) {
      throw WException();
    }
  }

  HINSTANCE instance() const {
    return mWndCl.hInstance;
  }

  const char* name() const {
    return mWndCl.lpszClassName;
  }

  WPARAM exec() {
    MSG Msg;
    while (GetMessage(&Msg, NULL, 0, 0) >
0)
    {
      TranslateMessage(&Msg);
      DispatchMessage(&Msg);
    }
    return Msg.wParam;
  }
};

class W32Widget {
public:
  using Handler = std::function <
void(WPARAM, LPARAM) >;

protected:
  HWND mHwnd;
  Handler mOnClickHandler;

public:
  HWND hwnd() const {
    return mHwnd;
  }

  virtual bool show(int nCmdShow =
SW_SHOW) {
    return ShowWindow(mHwnd, nCmdShow);
  }

  virtual bool update() {
    return UpdateWindow(mHwnd);
```

```
  }

  virtual void destroy() {
    DestroyWindow(mHwnd);
  }

  virtual void hide() {
    ShowWindow(mHwnd, SW_HIDE);
  }

  void setOnClickHandler(Handler handler)
{
    mOnClickHandler = handler;
  }

  void onClick(WPARAM wParam, LPARAM
lParam) {
    if (mOnClickHandler) {
      mOnClickHandler(wParam, lParam);
    }
  }
};

class W32Window : public W32Widget {
private:
  bool mModal = false;
  int modalChildren = 0;
  W32Window(const char* title, int width,
int height, const W32Application& app,
HWND parent) {
    mHwnd = CreateWindowEx(
      WS_EX_CLIENTEDGE,
      app.name(), title,
      WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT, CW_USEDEFAULT,
      width, height,
      parent, NULL, app.instance(),
NULL);

    if (mHwnd == NULL) {
      throw WException();
    }
  }

  void incModalChildren() {
    ++modalChildren;
    EnableWindow(mHwnd, FALSE);
  }

  void decModalChildren() {
    if (modalChildren > 0) {
      --modalChildren;
    }
    if (modalChildren == 0) {
      EnableWindow(mHwnd, TRUE);
    }
  }

public:
  W32Window() = default;
  W32Window(const char* title, int width,
int height, const W32Application& app)
    : W32Window(title, width, height,
app, NULL) {}
  W32Window(const char* title, int width,
int height, const W32Application& app,
const W32Widget& parent)
    : W32Window(title, width, height,
app, parent.hwnd()) {}

  void setModal(bool flag) {
    W32Window& parent =
dynamic_cast<W32Window&>(*getWidget(GetWi
ndow(mHwnd, GW_OWNER)));
    mModal = flag;
    if (flag) { // disable parent window
      parent.incModalChildren();
    }
    else { // enable parent window
      parent.decModalChildren();
    }
  }

  bool modal() const {
    return mModal;
  }

  void hide() override {
    if (GetWindow(mHwnd, GW_OWNER) ==
NULL) {
      destroy();
    }
    else {
      W32Widget::hide();
      if (mModal) {
        setModal(FALSE);
      }
    }
  }

  //bool show(int show = SW_SHOW)
override {
  //  bool res = W32Widget::show(show);
  //  BringWindowToTop(mHwnd);

  //  return res;
  //}
};

class W32Button : public W32Widget {
public:
  W32Button() = default;
```

```
  W32Button(
    const char* label,
    int width, int height,
    int x, int y,
    const W32Widget parent) {
    mHwnd = CreateWindow(
      "BUTTON", label,
      WS_TABSTOP | WS_VISIBLE | WS_CHILD
| BS_DEFPUSHBUTTON,
      x, y, width, height,
      parent.hwnd(), NULL,

(HINSTANCE)GetWindowLong(parent.hwnd(),
GWL_HINSTANCE), NULL);

    if (mHwnd == NULL) {
      throw WException();
    }
  }
};

class W32AbstractTextEdit : public
W32Widget {
public:
  bool readOnly() const {
    return ES_READONLY &
GetWindowLong(mHwnd, GWL_STYLE);
  }

  void setReadOnly(bool flag) {
    SendMessage(mHwnd, EM_SETREADONLY,
(flag ? TRUE : FALSE), 0);
  }

  void setText(const std::string& text) {
    SetWindowTextA(mHwnd, text.c_str());
  }

  void setText(const char* text) {
    SetWindowTextA(mHwnd, text);
  }

  std::string text() const {
    int length =
GetWindowTextLengthA(mHwnd);
    std::vector<char> buffer(length + 1);
    GetWindowTextA(mHwnd, buffer.data(),
length + 1);
    return std::string(buffer.begin(),
buffer.end() - 1);
  }
};

class W32TextEdit : public
W32AbstractTextEdit {
public:
  W32TextEdit() = default;
  W32TextEdit(
    int width, int height,
    int x, int y,
    const W32Widget parent) {
    mHwnd = CreateWindow(
      "EDIT", "",
      WS_TABSTOP | WS_VISIBLE | WS_CHILD
| WS_BORDER | WS_VSCROLL | WS_HSCROLL |
      ES_AUTOHSCROLL | ES_AUTOVSCROLL |
ES_MULTILINE,
      x, y, width, height,
      parent.hwnd(), NULL,

(HINSTANCE)GetWindowLong(parent.hwnd(),
GWL_HINSTANCE), NULL);

    if (mHwnd == NULL) {
      throw WException();
    }
  }
};

class W32TextField : public
W32AbstractTextEdit {
public:
  W32TextField() = default;
  W32TextField(
    int width, int height,
    int x, int y,
    const W32Widget parent) {
    mHwnd = CreateWindow(
      "EDIT", "",
      WS_TABSTOP | WS_VISIBLE | WS_CHILD
| WS_BORDER |
      ES_AUTOHSCROLL | ES_AUTOVSCROLL,
      x, y, width, height,
      parent.hwnd(), NULL,

(HINSTANCE)GetWindowLong(parent.hwnd(),
GWL_HINSTANCE), NULL);

    if (mHwnd == NULL) {
      throw WException();
    }
  }
};

std::map<HWND, shared_ptr<W32Widget>>
widgets;

void addWidget(shared_ptr<W32Widget>
widget) {
  widgets.emplace(widget->hwnd(),
widget);
}
```

```
shared_ptr<W32Widget> getWidget(HWND
hwnd) {
  return widgets.at(hwnd);
}
```

TextRoutines.h

```
#pragma once

#include <Windows.h>
#include <bits\stdc++.h>

#define mt std::make_tuple
#define mp std::make_pair

extern std::ofstream logs;

std::string LoadTextFromFile(const
std::string& name) {
  std::ifstream file(name);
  std::string text;
  std::string line;
  while (std::getline(file, line)) {
    text += line;
    text += "\r\n";
  }

  return text;
}

int ansi_tolower(int ch) {
  int ruA = 'À';
  int ruZ = 'ß';
  int rua = 'à';

  int enA = 'A';
  int enZ = 'Z';
  int ena = 'a';
  if (ruA <= ch && ch <= ruZ) {
    return ch + (rua - ruA);
  }
  else if (enA <= ch && ch <= enZ) {
    return ch + (ena - enA);
  }
  else {
    return ch;
  }
}

int ansi_toupper(int ch) {
  int ruA = 'À';
  int rua = 'à';
  int ruz = 'ÿ';

  int enA = 'A';
  int enz = 'z';
  int ena = 'a';
  if (rua <= ch && ch <= ruz) {
    return ch - (rua - ruA);
  }
  else if (ena <= ch && ch <= enz) {
    return ch - (ena - enA);
  }
  else {
    return ch;
  }
}

namespace internal {
  using pii = std::pair < int, int > ;
  using std::tuple;
  using std::get;
  using std::tie;

  struct word_t {
    using hash_t =
std::hash<std::string>;
    std::string str;
    size_t hash;

    word_t(const std::string& str, size_t
hash) :
      str(str), hash(hash) {}

    bool operator==(const word_t& rhs)
const {
      return hash == rhs.hash;
    }

    bool operator!=(const word_t& rhs)
const {
      return hash != rhs.hash;
    }

    word_t& operator=(const word_t& rhs)
= default;

    word_t to_upper() const {
      std::string upp_str = str;

      for (char& c : upp_str) {
        c = ansi_toupper(c);
      }

      return word_t{ upp_str,
hash_t()(upp_str) };
    }
```

```
    word_t to_lower() const {
      std::string low_str = str;

      for (char& c : low_str) {
        c = ansi_tolower(c);
      }

      return word_t{ low_str,
hash_t()(low_str) };
    }
  };

  std::vector<word_t> SplitLines(const
std::vector<std::string>& lines) {
    std::vector<word_t> words;
    std::hash<std::string> hash;

    for (const auto& line : lines) {
      std::stringstream stream{ line };
      std::string word;
      while (stream >> word) {
        words.emplace_back(word,
hash(word));
      }

      words.emplace_back("\r\n",
hash("\r\n"));
    }

    return words;
  }

  std::vector<std::string> GetLines(const
std::string& str) {
    std::vector<std::string> lines;
    std::stringstream stream{ str };
    std::string line;
    std::hash<std::string> hash;

    while (std::getline(stream, line)) {
      lines.emplace_back(line);
    }

    return lines;
  }

  std::string JoinLines(const
std::vector<word_t>& lines) {
    std::string text;
    size_t sz = 0;
    for (const auto& word : lines) {
      sz += word.str.size();
    }

    text.reserve(sz);
    for (const auto& word : lines) {
      text += word.str;
      //text += "\r\n";
      if (word.str != "\r\n") {
        text += " ";
      }
    }

    return text;
  }

  class Differences {
  private:
    std::map<pii, int>& d;
    std::map<pii, pii>& p;
  public:
    Differences(std::map<pii, int>& d,
std::map<pii, pii>& p) :
      d(d), p(p) {}
    // <length, index1, index2>
    tuple<int, int, int>
    operator()(
      const std::vector<word_t>& result1,
      const std::vector<word_t>& result2,
      int n1,
      int n2
    ) {
      auto key = mp(n1, n2);
      if (n1 < 0 || n2 < 0) {
        return mt(0, -1, -1);
      }
      else if (d.find(key) != d.end()) {
        int i1, i2;
        tie(i1, i2) = p[key];
        if (result1[n1] != result2[n2]) {
          return mt(d[key], i1, i2);
        }
        else {
          return mt(d[key], n1, n2);
        }
      }

      if (result1[n1] == result2[n2]) {
        int i1, i2, res;
        tie(res, i1, i2) =
operator()(result1, result2, n1 - 1, n2 -
1);
        res += 1;

        d[key] = res;
        p[key] = mp(i1, i2);
        return mt(res, n1, n2);
      }
      else {
        int i1, i2, res;
```

```
        auto t1 = operator()(result1,
result2, n1 - 1, n2);
        auto t2 = operator()(result1,
result2, n1, n2 - 1);

        if (get<0>(t1) > get<0>(t2)) {
          tie(res, i1, i2) = t1;
        }
        else {
          tie(res, i1, i2) = t2;
        }
        d[key] = res;
        p[key] = mp(i1, i2);
        return mt(res, i1, i2);
      }
    }
  };
}

std::pair<
  std::string,
  std::string
> ShowDifferences(const std::string&
text1, const std::string& text2) {
  using namespace internal;
  auto result1 =
SplitLines(GetLines(text1));
  auto result2 =
SplitLines(GetLines(text2));

  std::map<pii, int> d;
  std::map<pii, pii> p;
  Differences diffs(d, p);

  int i1, i2;
  std::tie(std::ignore, i1, i2) =
diffs(result1, result2, result1.size() -
1, result2.size() - 1);

  result1[i1] = result1[i1].to_lower();
  result2[i2] = result2[i2].to_lower();

  for (int f = 0, s = 0;;) {
    std::tie(f, s) = p[mp(i1, i2)];

    if (f < 0 || s < 0) {
      break;
    }

    result1[f] = result1[f].to_lower();
    result2[s] = result2[s].to_lower();

    i1 = f, i2 = s;
  }

  return mp(JoinLines(result1),
JoinLines(result2));
}

#undef mt
#undef mp
```