**№8**

**Лабораторная работа**

**«Разработка полнофункционального программного продукта для решения задач в профессиональной области»**

**Цель работы:**

Разработка полноценного программного продукта для решения задачи шифрования с применением криптографических возможностей Python

**Краткая теория:**

В Python существуют инструменты и стандартные библиотеки реализующие шифрование и библиотеки хешинга. Рассмотрим применение сторонних пакетах: PyCrypto и cryptography. Рассмотрим шифрование и расшифровывание текстовой строки с применением этих библиотек. Хеширование. Если нужно защитить хэши или алгоритм дайджеста сообщений, то для этого прекрасно подойдет модуль стандартной библиотеки Python hashlib. Он включает в себя безопасные алгоритмы хеширования FIPS, такие как SHA1, SHA224, SHA256, SHA384, а также SHA512 и MD5. Python поддерживает функции хеширования adler32 и crc32 - они содержатся в модуле zlib. Самым распространенным применением хеширования является хранение хеша пароля, вместо самого пароля. Алгоритм хеширования должен быть грамотно выбран для поставленной задачи, иначе он может быть расшифрован. Хеширование часто применяется для файлов, отправляемых по сети или через интернет. ВНИМАНИЕ Файл и его хеш отправляются по отдельности. Получатель файла может запустить хеш в полученном файле, чтобы убедиться в том, что файл соответствует отправленному хешу. Если это так, значит никто не менял файл, с момента его отправки.

**Задание**

Создать две программы выполняющие функции шифрования и дешифрования текстового файла. (выбор шифрования студент выбирает самостоятельно)

**ХОД РАБОТЫ**

Для выполнения задания было принято решение воспользоваться алгоритмом кодирования Хаффмана. Данный алгоритм позволяет не только закодировать текст создав ключ для декодирования, но еще и сжать текст. Также для реализации интерфейса потребуется встроенный в Python модуль Tkinter.

Создадим два файла, один из которых будет модульным, в него пропишем код алгоритма Хаффмана.

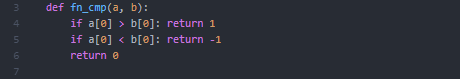


Рисунок 1 - сравнение 2-х элементов по количеству

На рисунке 1 показано сравнение 2-х элементов по количеству.

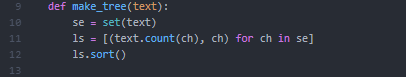


Рисунок 2 – код функция дерева Хаффмана

Начинаем построение дерева Хаффмана по тексту. Для этого как показано на рисунке 2 создаем функцию дерева. Для начала необходимо выделить множество символов. Далее создаем частотный список, с помощью генератора списков, пробегая все элементы в множестве se. В качестве результирующего элемента берем пару количество символов в строке и сам символ. Следом производим сортировку для того, чтобы символы с меньшей чистотой были слева, а с большей частотой справа.

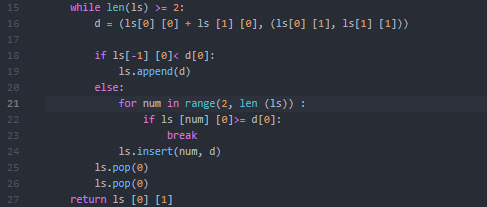


Рисунок 3 – код построения двоичного дерева по полученному отсортированному списку

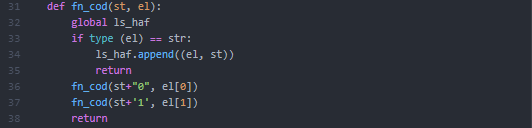


Рисунок 4 – код рекурсивного бинарного кода листов элемента

На рисунке 4 создается рекурсивно бинарный код листов элемента. Если элемент «символ» т.е. его тип str (строка), то к списку Хаффмана, прибавляем код и элемент, если этот элемент дерево, то применяем к его элементам саму эту функцию. Т.о. рекурсивная функция позволяет обойти дерево целиком и создать словарь.

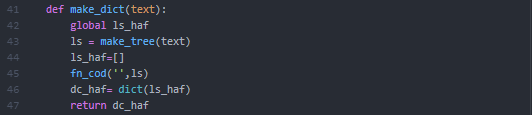


Рисунок 5 – функция словаря Хаффмана

Далее пишем новую функцию, в которой будет создавать словарь Хаффмана по тексту. Список Хаффмана обозначаем, как глобальную переменную. Для того, чтобы к ней можно было обращаться так же вне этой функции. Далее создаем дерево Хаффмана и применяем к нему рекурсивную функцию, описанную ранее, которая из дерева создаст код. Превращаем полученный список в словарь и возвращаем его.

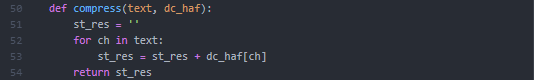


Рисунок 6 – функция сжатия текста

После создания словаря приступаем к сжатию текста. Где будет текст кодироваться в двоичный код, и создаваться ключ для расшифровки. Для начала сделаем заготовку результирующей строки, как пустую, а затем к строке добавляем значение словаря, проходим в цикле по всему тексту и возвращаем сжатый текст.

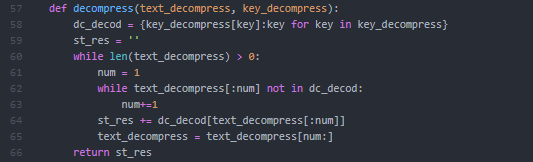


Рисунок 7 – код декодирования

На рисунке 7 представлен код декодирования. Аналогично, как и в случае с сжатием, данная функция работает с обратным словарем. Это очень просто сделать с помощью генератора словарей. Используем словарь Хаффмана и с помощью генератора словаря обращаем его, чтобы левым элементом было последовательность бит, а правым элементом был символ. Также создаем пустую результирующую строку. И потом выполняем побитный проход по строке, пока длинна строки больше 0 считываем все символы, до тех пор, пока они не будут находиться в словаре декодирования. Как только найдется в словаре декодирования последовательность бит, прибавляем к результирующей строке элемент словаря декодирования и продолжаем цикл с меньшей длинной. И в конце возвращаем результирующую строку символов. Имея сжатый текст в двоичном виде, а также ключ сопоставляем их.

После написания основного тела алгоритма, отвечающего за логику шифрование и дешифровку текста, напишем функцию отвечающую за взаимодействие с самим текстом. Т.к. нам требуется создать помимо зашифрованного текста еще и ключ, учтем это и сделаем две функции, одна будет отвечать за текст, другая за ключ.

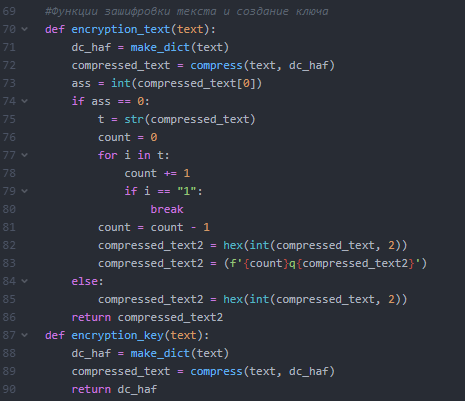


Рисунок 8 – код функции взаимодействия с текстом

Теперь осталось лишь прописать функцию дешифровки.

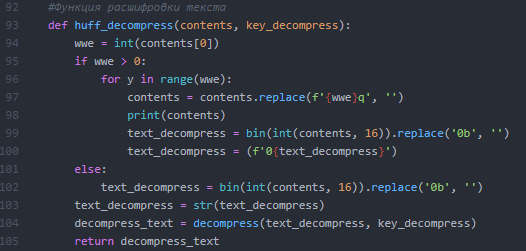


Рисунок 9 – функция дешифровки

На этом с алгоритмом шифрования закончили. Теперь как и планировалось приступим к написанию кода второго файла .py. В него будет помещен интерфейс программы.

Первым делом импортируем требуемые модули. В том числе и созданный модуль алгоритма Хаффмана.

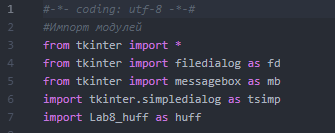


Рисунок 10 – импорт модулей в программу интерфейса

Перед написанием непосредственно самого интерфейса пропишем логику работы с программы. Первым делом пропишем открытие текста, в данном случае не важно зашифрованного или же обычного текста, одна функция будет и на шифрование и на расшифровку. Далее пропишем функцию открытия ключа. Текст и ключ передаем в глобальные переменные.

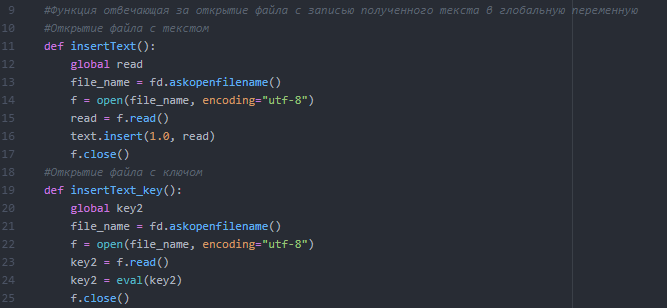


Рисунок 11 – код открытия файла и ключа

Далее пропишем код функции, отвечающей за сохранение зашифрованного текста в том формате, в котором желает пользователь. Также, сразу же сохраним ключ по тому же пути, что и зашифрованный текст. Ключу дадим расширение .dat.

Следом создадим функцию сохранения уже расшифрованного файла.



Рисунок 12 – код функций сохранения файлов

Далее создаем функции отвечающие за зашифровку и расшифровку текста. Они будут получать текст и ключ, передавать их в модуль, в котором и будет происходить процесс обработки. После успешного выполнения задачи, выведем окно об успешном выполнении.

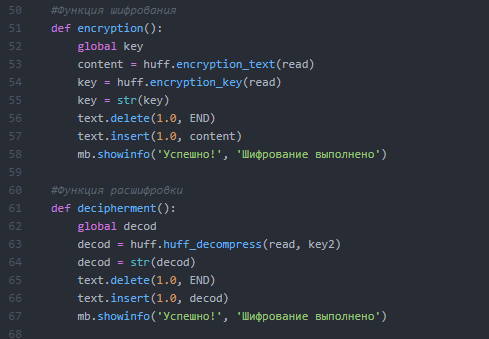


Рисунок 13 – код функций шифрования и расшифровки

Теперь перейдем непосредственно к самому интерфейсу и его частям. Так как у нас два направления работы программы (шифрование и расшифровка), для упрощения работы с программой создадим условие. При нажатии нужной радиокнопки (Radiobutton), будет появляться тот или иной метод обработки текста. При переходе от одного метода к другому, кнопки, отвечающие за действия, будут меняться.



Рисунок 14 – код условий выбора метода работы программы

Далее напишем код радиокнопок, текстового поля и названия окна программы.

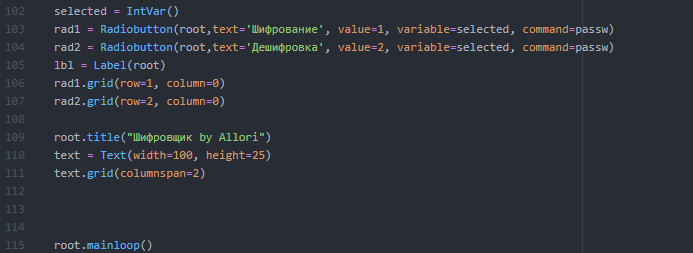


Рисунок 15 – код создания Radiobutton, текстового поля и названия окна программы

**ЛИСТИНГ ПРОГРАММЫ**

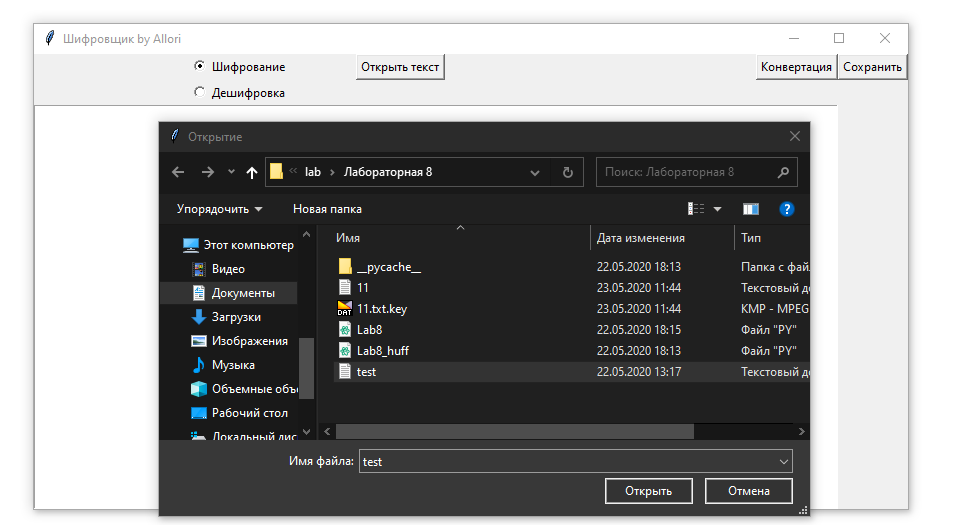
**Модуль с алгоритмом Хаффмана**

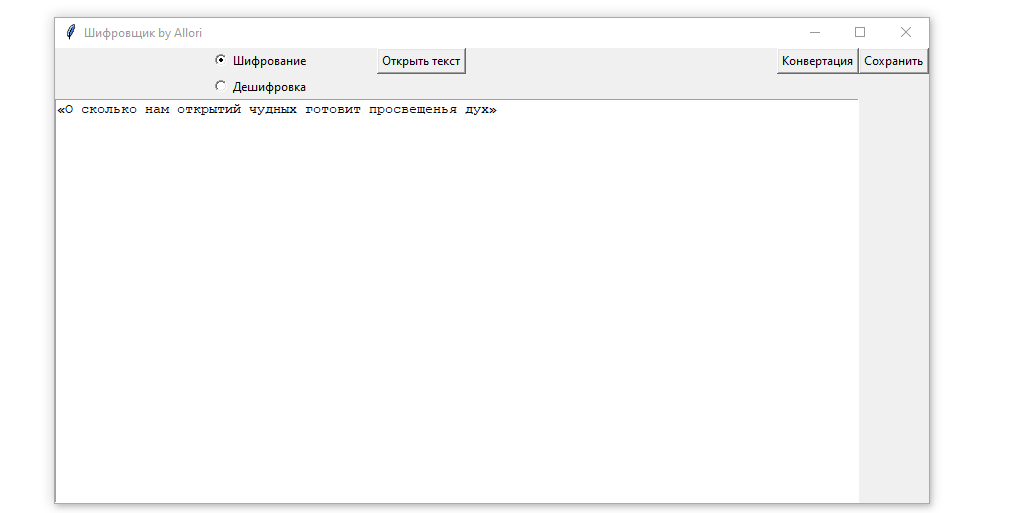
|  |
| --- |
| # -\*- coding: utf8 -\*-  def fn\_cmp(a, b):  if a[0] > b[0]: return 1  if a[0] < b[0]: return -1  return 0  #-- создать дерево хаффмана по тексту -  def make\_tree(text):  se = set(text)  ls = [(text.count(ch), ch) for ch in se]  ls.sort()  #-- построить двоичное дерево по этому списку  while len(ls) >= 2:  d = (ls[0] [0] + ls [1] [0], (ls[0] [1], ls[1] [1]))  if ls[-1] [0]< d[0]:  ls.append(d)  else:  for num in range(2, len (ls)) :  if ls [num] [0]>= d[0]:  break  ls.insert(num, d)  ls.pop(0)  ls.pop(0)  return ls [0] [1]  #-- рекурсивно создать бинарный код листов элемента  def fn\_cod(st, el):  global ls\_haf  if type (el) == str:  ls\_haf.append((el, st))  return  fn\_cod(st+"0", el[0])  fn\_cod(st+'1', el[1])  return  #-- создать словарь хаффмана по тексту -  def make\_dict(text):  global ls\_haf  ls = make\_tree(text)  ls\_haf=[]  fn\_cod('',ls)  dc\_haf= dict(ls\_haf)  return dc\_haf  #-- сжать по хаффману  def compress(text, dc\_haf):  st\_res = ''  for ch in text:  st\_res = st\_res + dc\_haf[ch]  return st\_res  #-- decompress  def decompress(text\_decompress, key\_decompress):  dc\_decod = {key\_decompress[key]:key for key in key\_decompress}  st\_res = ''  while len(text\_decompress) > 0:  num = 1  while text\_decompress[:num] not in dc\_decod:  num+=1  st\_res += dc\_decod[text\_decompress[:num]]  text\_decompress = text\_decompress[num:]  return st\_res  #Функции зашифровки текста и создание ключа  def encryption\_text(text):  dc\_haf = make\_dict(text)  compressed\_text = compress(text, dc\_haf)  ass = int(compressed\_text[0])  if ass == 0:  t = str(compressed\_text)  count = 0  for i in t:  count += 1  if i == "1":  break  count = count - 1  compressed\_text2 = hex(int(compressed\_text, 2))  compressed\_text2 = (f'{count}q{compressed\_text2}')  else:  compressed\_text2 = hex(int(compressed\_text, 2))  return compressed\_text2  def encryption\_key(text):  dc\_haf = make\_dict(text)  compressed\_text = compress(text, dc\_haf)  return dc\_haf  #Функция расшифровки текста  def huff\_decompress(contents, key\_decompress):  wwe = int(contents[0])  if wwe > 0:  for y in range(wwe):  contents = contents.replace(f'{wwe}q', '')  print(contents)  text\_decompress = bin(int(contents, 16)).replace('0b', '')  text\_decompress = (f'0{text\_decompress}')  else:  text\_decompress = bin(int(contents, 16)).replace('0b', '')  text\_decompress = str(text\_decompress)  decompress\_text = decompress(text\_decompress, key\_decompress)  return decompress\_text |

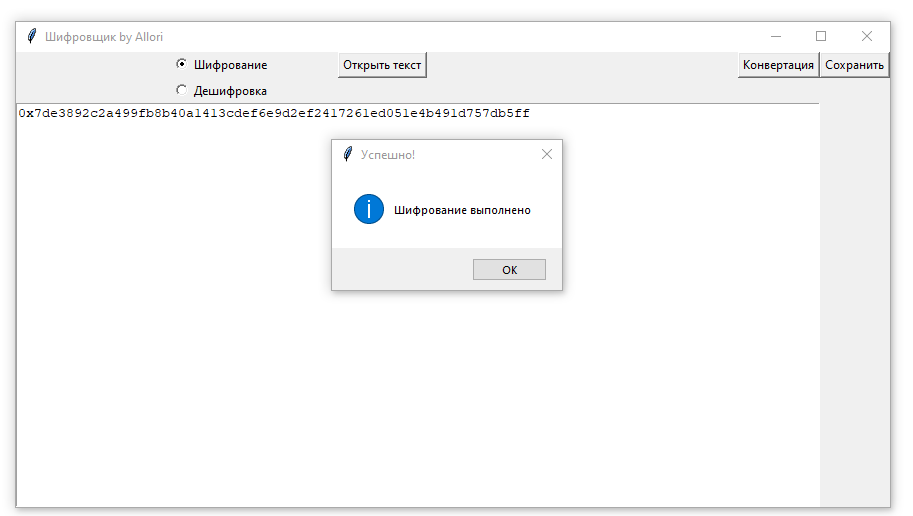
**Код интерфейса**

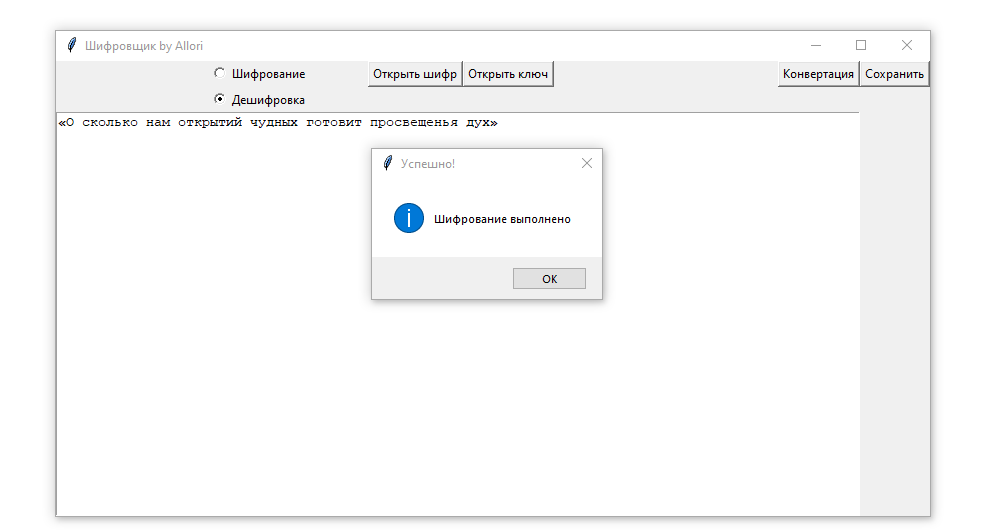
|  |
| --- |
| #-\*- coding: utf-8 -\*-#  #Импорт модулей  from tkinter import \*  from tkinter import filedialog as fd  from tkinter import messagebox as mb  import tkinter.simpledialog as tsimp  import Lab8\_huff as huff  #Функция отвечающая за открытие файла с записью полученного текста в глобальную переменную  #Открытие файла с текстом  def insertText():  global read  file\_name = fd.askopenfilename()  f = open(file\_name, encoding="utf-8")  read = f.read()  text.insert(1.0, read)  f.close()  #Открытие файла с ключом  def insertText\_key():  global key2  file\_name = fd.askopenfilename()  f = open(file\_name, encoding="utf-8")  key2 = f.read()  key2 = eval(key2)  f.close()  #Функция отвечающая за сохранение обработанного текста в новый файл  def extractText\_encry():  file\_name = fd.asksaveasfilename(filetypes=(("TXT files", "\*.txt"),  ("HTML files", "\*.html;\*.htm"),  ("All files", "\*.\*") ))  #Сохранение зашифрованного файла  f = open(file\_name, 'w', encoding="utf-8")  s = text.get(1.0, END)  f.writelines(s)  f.close()  #Сохранение ключа с расширением .dat  f = open(f'{file\_name}.key.dat', 'w', encoding="utf-8")  f.write(key)  f.close()  #Функция отвечающая за сохранение расшифрованного файла  def extractText\_decip():  file\_name = fd.asksaveasfilename(filetypes=(("TXT files", "\*.txt"),  ("HTML files", "\*.html;\*.htm"),  ("All files", "\*.\*") ))  f = open(file\_name, 'w', encoding="utf-8")  f.writelines(decod)  f.close()  #Функция шифрования  def encryption():  global key  content = huff.encryption\_text(read)  key = huff.encryption\_key(read)  key = str(key)  text.delete(1.0, END)  text.insert(1.0, content)  mb.showinfo('Успешно!', 'Шифрование выполнено')  #Функция расшифровки  def decipherment():  global decod  decod = huff.huff\_decompress(read, key2)  decod = str(decod)  text.delete(1.0, END)  text.insert(1.0, decod)  mb.showinfo('Успешно!', 'Шифрование выполнено')  #Основной код программы отвечающий за создание элементов интерфейса  root = Tk()  #Функция отвечающая за переключение между зашифровкой и расшифровкой  def passw():  global b1, b2, b3, b4, b5, b6, b7  if selected.get() == 1:  b1 = Button(text="Открыть текст", command=insertText)  b1.grid(row=1, sticky=E)  b2 = Button(text="Сохранить", command=extractText\_encry)  b2.grid(row=1, column=2)  b3 = Button(text="Конвертация", command=encryption)  b3.grid(row=1, column=1, sticky=E)  b4.grid\_remove()  b5.grid\_remove()  b6.grid\_remove()  b7.grid\_remove()  if selected.get() == 2:  b4 = Button(text="Конвертация", command=decipherment)  b4.grid(row=1, column=1, sticky=E)  b5 = Button(text="Открыть шифр", command=insertText)  b5.grid(row=1, sticky=E)  b6 = Button(text="Открыть ключ", command=insertText\_key)  b6.grid(row=1, column=1, sticky=W)  b7 = Button(text="Сохранить", command=extractText\_decip)  b7.grid(row=1, column=2)  b1.grid\_remove()  b2.grid\_remove()  b3.grid\_remove()  selected = IntVar()  rad1 = Radiobutton(root,text='Шифрование', value=1, variable=selected, command=passw)  rad2 = Radiobutton(root,text='Дешифровка', value=2, variable=selected, command=passw)  lbl = Label(root)  rad1.grid(row=1, column=0)  rad2.grid(row=2, column=0)  root.title("Шифровщик by Allori")  text = Text(width=100, height=25)  text.grid(columnspan=2)  root.mainloop() |

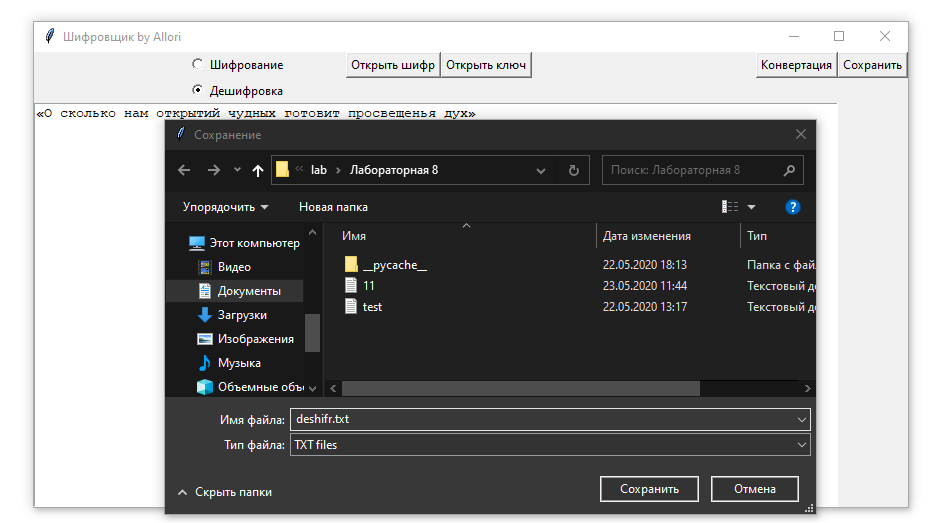
**КОНТРОЛЬНЫЙ ВАРИАНТ**











**КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. **Что такое хеширование и для чего оно применяется**

Алгоритм хеширования (также хеш-функций) уместен для криптовалютных блокчейн платформ, например Эфириум или Биткоин. Процедура хеширования занимает короткий промежуток времени, за который входящий массив данных произвольной длины преобразовывается в битовую строку после обработки системным алгоритмом.

1. **Сравните алгоритмы хеширования SHA1, SHA224, SHA256, SHA384, SHA512, MD5.**

Это разные алгоритмы хеш-функции. Создатели программного обеспечения часто указывают хеш для загружаемых файлов. Алгоритм хеширования должен создавать уникальные хеши для каждого возможного ввода. Бит имеет два возможных значения: 0 и 1. Возможное количество уникальных хешей может быть выражено как количество возможных значений увеличенных на количество бит. Для SHA-256 имеется 2 256 возможных комбинаций. Итак, 2 256 комбинаций.

 SHA – это защитный алгоритм хеширования. SHA-1 и SHA-2 – это две разные версии этого алгоритма. Они различаются в конструкциях (как создается хеш из исходных данных) и в битовой длине подписи. Следует воспринимать SHA-2, как преемника SHA-1. В первую очередь, люди сосредоточены на бит-длине, как на важном различии. SHA-1 — это 160-битный хеш. SHA-2, на самом деле, является «семейством» хешей и имеет множество длин, наиболее популярной из которых является 256-бит. Разнообразие хешей SHA-2 может привести к путанице, поскольку сайты и авторы обозначают их по-разному. Если «SHA-2», «SHA-256» или «SHA-256 бит», эти имена относятся к одной и той же вещи. Если «SHA-224», «SHA-384» или «SHA-512», это относится к чередованию бит-длины SHA-2. Вы также можете увидеть, что некоторые сайты выписывают как алгоритм, так и длину бита, например «SHA-2 384.».

С 2011 по 2015 год SHA-1 был основным алгоритмом. Растущее число исследований, показывающих недостатки SHA-1, вызвало переоценку. Фактически, Google даже зашел так далеко, что создал конфликт SHA-1 (когда две части разрозненных данных создают одно и то же значение хеш-функции). Таким образом, с 2016 года SHA-2 является новым стандартом. Если вы получаете сегодня сертификат SSL/TLS, он должен использовать эту подпись. Иногда будут встречаться сертификаты, использующие SHA-2 384-бита. Реже будут встречаться 224-битные типы, поскольку они не одобрены для использования с публично доверенными сертификатами, или 512-битные типы, которые очень редко поддерживаются программным обеспечением.

В 2015 году индустрия SSL прошла «переход на SHA-2». Это включало повторную выдачу тысяч существующих сертификатов, чтобы новые файлы могли быть созданы и подписаны с SHA-2. Это также связано с крупными обновлениями программного обеспечения для выпуска, которым работают централизованные доверенные центры сертификации (их десятки).

MD5 был разработан в 1991 году профессором Рональдом Л. Ривестом. Алгоритм его часто называют алгоритмом шифрования, но на самом деле это утверждение ошибочно. Главным преимуществом MD5 является то, что зашифрованные данные восстановить нельзя.

Предположим, у нас есть некоторый набор данных. Для простоты будем рассматривать натуральные числа от 1 до 106. И пусть есть некоторая функция, в которой один параметр — натуральное число от 1 до 106, а возвращаемое значение — натуральное число от 1 до 1000. Нам не важно, что именно делает эта функция, нам важно то, что она каждому натуральному числу от 1 до 106 ставит в соответствие другое натуральное число от 1 до 1000. Это и есть простая хэш-функция. Если мы знаем параметр функции, то однозначно можем сказать, какой будет результат.