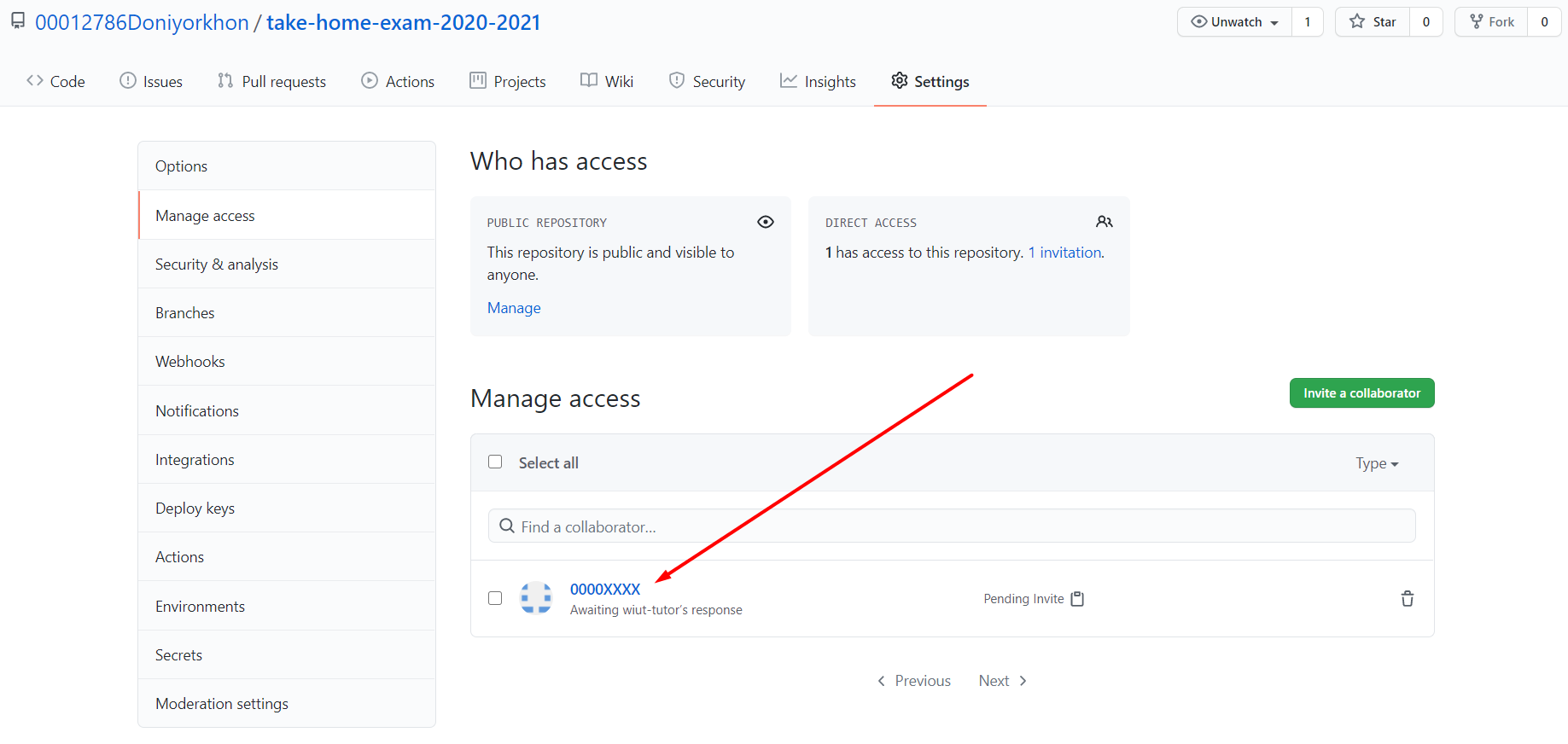
# Task 1

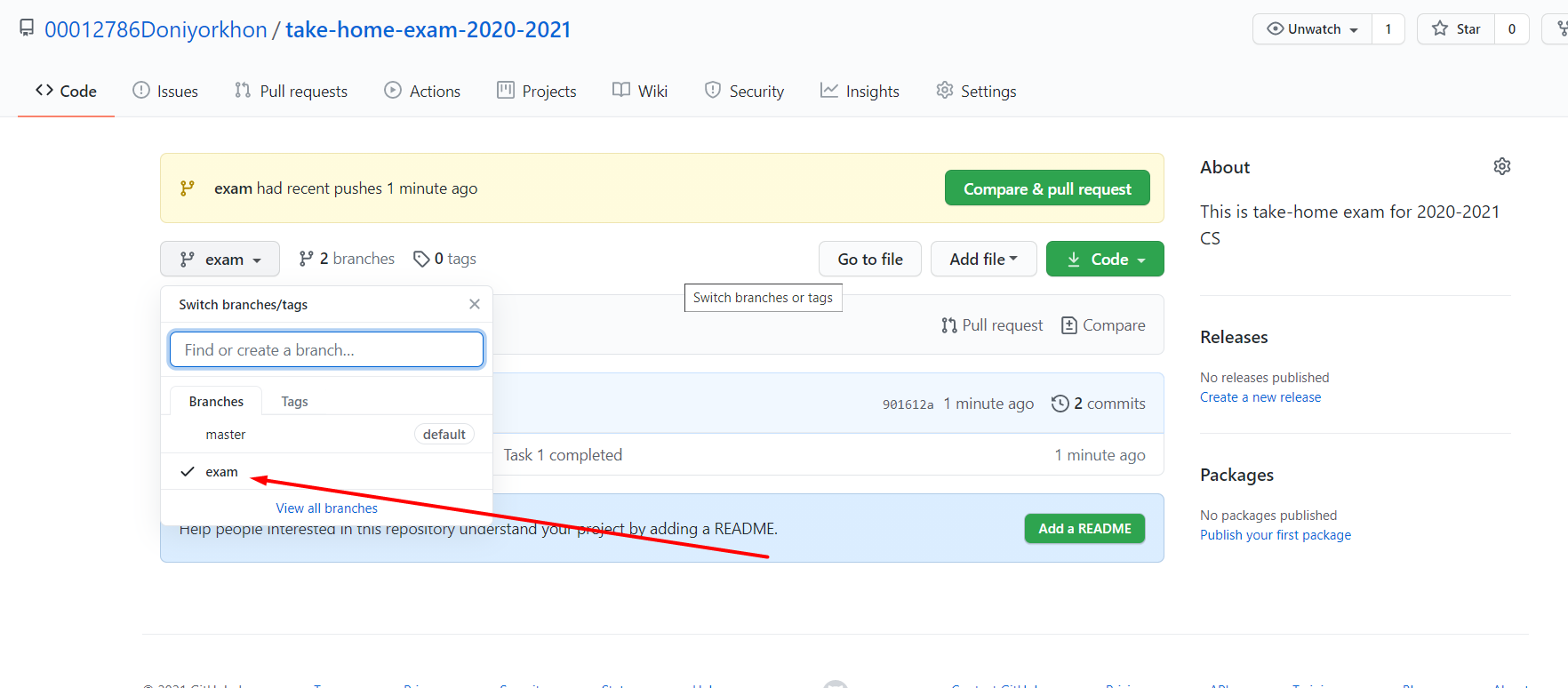
For this part of the assignment, new repository on GitHub platform was created that is accessible by url: <https://github.com/00012786Doniyorkhon/take-home-exam-2020-2021/>

In order to keep versioning the assignment purposes, new branch *exam* was opened.

As this assignment is being done using new repository, one collaboration request has been sent to a GitHub user: *wiut-tutor*:



As soon as Task 1 finishes, new commit is required, so when it was committed to new branch *exam*, such branch started appearing:



# Task 2

ID number: *12786*

|  |  |  |
| --- | --- | --- |
| 2 | 12786 |  |
| 2 | 6393 | 0 |
| 2 | 3196 | 1 |
| 2 | 1598 | 0 |
| 2 | 799 | 0 |
| 2 | 399 | 1 |
| 2 | 199 | 1 |
| 2 | 99 | 1 |
| 2 | 49 | 1 |
| 2 | 24 | 1 |
| 2 | 12 | 0 |
| 2 | 6 | 0 |
| 2 | 3 | 0 |
| 2 | 1 | 1 |
| 2 | 0 | 1 |
| 1278610=110001111100102 | | |

**a)** 1. In order to convert decimal number to binary numeric system the steps below are taken (<https://www.instructables.com/How-to-Convert-From-Decimal-to-Binary/>):

Step 1.

|  |  |  |
| --- | --- | --- |
| 2 | 12786 |  |
|  | 6393 | 0 |

In order to convert a decimal number to binary, the decimal number in our case 12786 is divided by 2 which is on left side and remainder is kept (0) at right side like above.

Step 2. Division product (6393) is kept divided by 2 (which is on left side) again and the remainder is saved on right side (which is 1).

Step 3. After the last iteration, binary representation of original decimal number is completed on the right column as shown above. Only requirement is to read number sequence from bottom to top.

2. Conversion of decimal number to hexadecimal is logical continuation of decimal-to-binary conversion and it involves the steps below (<https://www.instructables.com/Convert-Decimal-to-Hexadecimal/>):

Step 1. It is required to rewrite product of decimal-to-binary conversion product from right to left every four bits separated by a space character and required to finish by leading extra zeros on the left to complete the last group of four bits:

1278610 = 00 11 0001 1111 00102

Step 2. These four bits are compared to their hexadecimal analogue from this binary-to-hex table (<https://www.wikihow.com/Convert-Binary-to-Hexadecimal>):

|  |  |
| --- | --- |
| **Binary** | **Hexadecimal** |
| **0** | **0** |
| **1** | **1** |
| **10** | **2** |
| **11** | **3** |
| **100** | **4** |
| **101** | **5** |
| **110** | **6** |
| **111** | **7** |
| **1000** | **8** |
| **1001** | **9** |
| **1010** | **A** |
| **1011** | **B** |
| **1100** | **C** |
| **1101** | **D** |
| **1110** | **E** |
| **1111** | **F** |

1278610 = 0011 0001 1111 00102

3 1 F 2

1278610 = 00 11 0001 1111 00102 = 31F216

**b)** According to steps taken for decimal-to-binary conversion of numbers:

9999910 = 0001 1000 0110 1001 11112

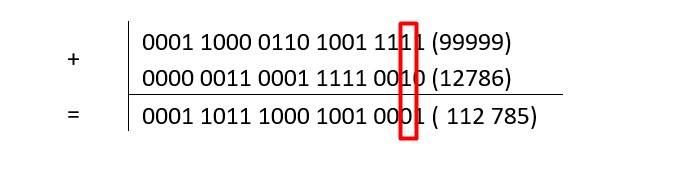
1278610 = 0000 0011 0001 1111 00102

*Addition*. Such operation in binary system is pretty much the same as in decimal system with only one nuance is that binary addition carries 1 addition result equals to 2 (<https://www.calculator.net/binary-calculator.html>):

*0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, 1 + 1 = 0, carry over the 1, i.e. 10*

For example, let’s add those two numbers (99999 and 12786) in binary format:

|  |  |
| --- | --- |
| + | 0001 1000 0110 1001 1111 (99999) |
| 0000 0011 0001 1111 0010 (12786) |
| = | 0001 1011 1000 1001 0001 ( 112 785) |

In the picture above it is shown that how carrying over the 1 happens *when 1 + 1* is done.

*Subtraction.* Borrowing happens when the number that is subtracted is larger than the number it is being subtracted from (when 1 is subtracted from 0). When this happens, the 0 in the borrowing column becomes "2" while reducing the 1 in the column being borrowed from by 1. Borrowing will have to occur from each subsequent column until a column with a value of 1 can be reduced to 0 (ibid.):

*0 - 0 = 0*

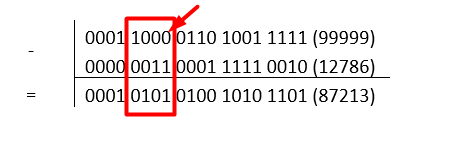
*0 - 1 = 1, borrow 1, resulting in -1 carried over*

*1 - 0 = 1*

*1 - 1 = 0*

For example, let’s subtract those two numbers (99999 and 12786) in binary format:

|  |  |
| --- | --- |
| - | 0001 1000 0110 1001 1111 (99999) |
| 0000 0011 0001 1111 0010 (12786) |
| = | 0001 0101 0100 1010 1101 (87213) |



The picture above clearly shows how borrowing happens from subsequent digits.

**c)** Primary ***use case*** of a hexadecimal number is to simplify binary representation of data by computer scientists. As hex is 2 power of 4, this means there is linear relationship between these two, i.e. hex digit is equivalent to 4 binary digits. Computers use binary system while humans prefer hexadecimal system to shorten binary representation whereby making it legible and easy to read. Another usages of hex are as well below:

\* Hexadecimals can define locations in memory. Two hex digits characterize one byte whereas eight binary digits do the same.

\* Hexadecimals represent colors. Each of RGBs is characterized by two hex digits: #RRGGBB.

\* Hexadecimals represent MAC addresses, data contract of standards like Modbus. MAC address consists of 12-digit hex number with formats of MMMM-MMSS-SSSS or MM:MM:MM:SS:SS:SS (first six hex digits represent Adapter manufacturer ID and another six hex digits show Adapter serial number). RS-232 interface applying Modbus protocol transfers ASCII or Hex-represented values, for example, Slave can send function codes represented in hex format to a Master in order to perform some operation (<https://www.picotech.com/library/oscilloscopes/modbus-serial-protocol-decoding>).

Hexadecimal systems are well known for their some advantages:

* Signifying a number using a base of 16 means comes up with less number of digits compared to binary or decimal. Hex system allows to store more info using less space. Hence, large binary numbers are written using hex system.
* As hex system is human-legible, it is used to group binary numbers for easier understanding like in Task 1.
* Moreover, such concise representation of binary numbers using hex format lowers error occurring possibility of binary numbers sequence.

# Task 3

Father’s and Mother’s names combination: *Botir&Aziza*

Tree:

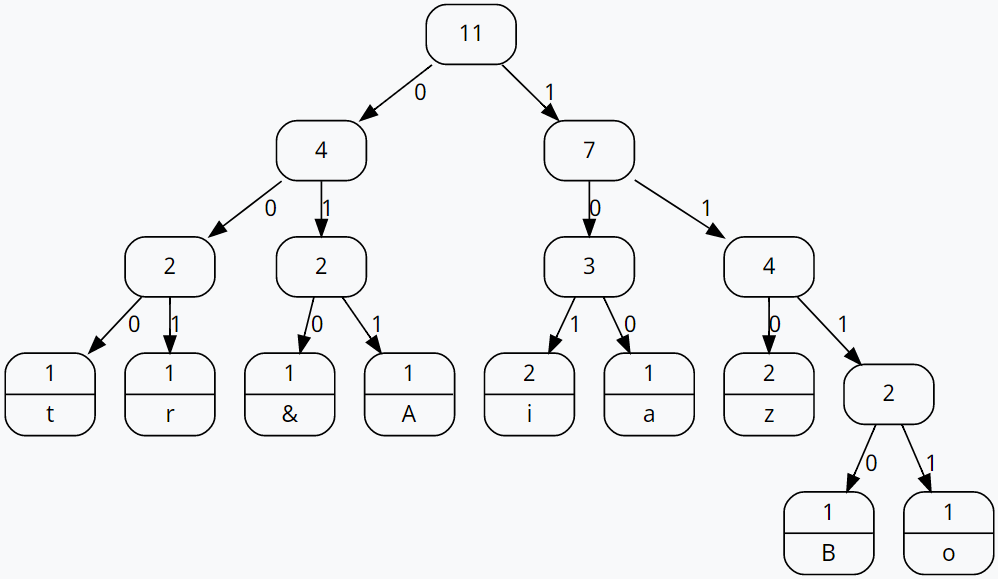


Table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Character** | **Frequency** | **Code** | **Code length** |
| B | 1 | 1110 | 4 |
| o | 1 | 1111 | 4 |
| t | 1 | 000 | 3 |
| i | 2 | 101 | 3 |
| r | 1 | 001 | 3 |
| A | 1 | 011 | 3 |
| z | 2 | 110 | 3 |
| A | 1 | 100 | 3 |
| & | 1 | 010 | 3 |

Total number of bits of resulting encoded message using frequency of characters:

Freq(B)\*codeLength(B) + Freq(o)\*codeLength(o) + … Freq(&)\*codeLength(&) = 1\*4+1\*4+1\*3+…+1\*3+1.3 = 35

In order to find number of bits representing those characters, these should be calculated:

Sum of frequencies = 9

Total number of bits = 8 \* 9 = 72 bits

According to Huffman encoding, total number of bits is calculated as:

1\*4 (B) +1\*4 (o) + 2\*3 (z) + 1\*3 (a) + 2\*3 (i) + 1\*3 (A) + 1\*3 (&) + 1\*3 (r) + 1\*3 (t) = 35

Bits saved = 72 - 35 = 37

# Task 4

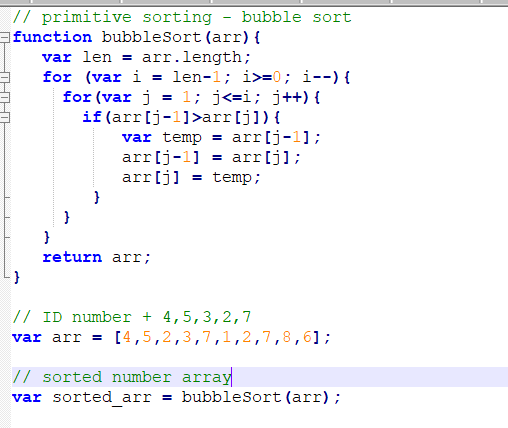
Given number: 45237

ID number: 12786

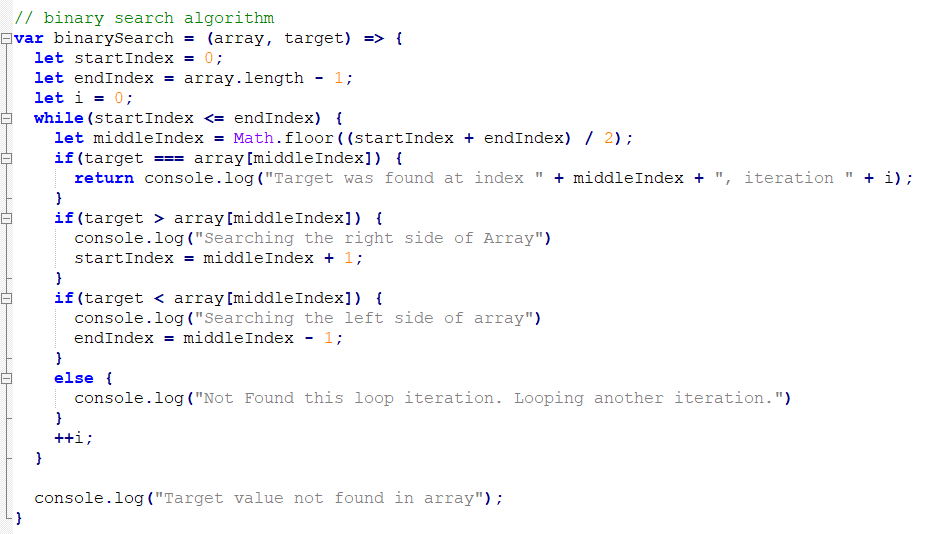
Numbers: *4523712786*

The code that describes binary search algorithm over sorted array of numbers is shown in auxiliary file **task4.js** attached to this Word file document.

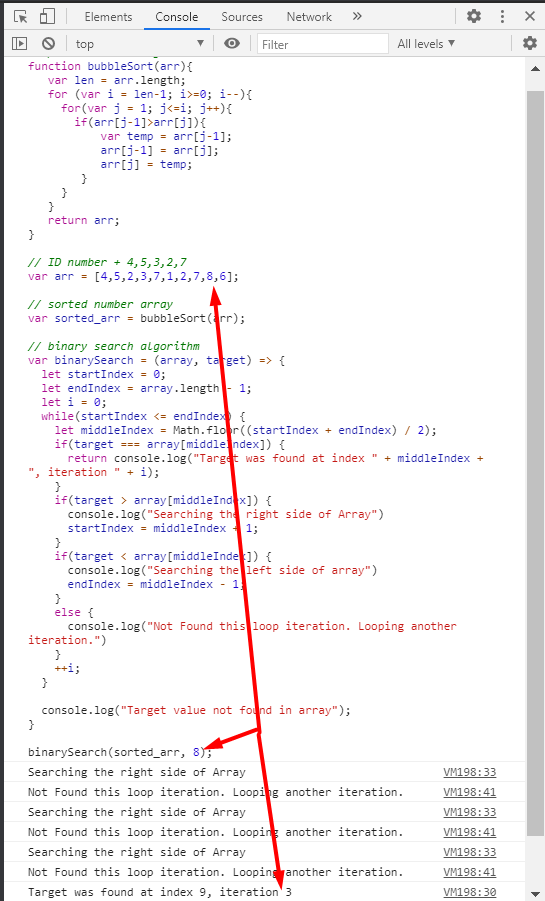
Firstly, array of numbers is sorted using bubble sort algorithm:



Secondly, binary search algorithm has been written:



Thirdly, the code is checked using V8 engine of Chrome web browser:



The picture above shows that number *8* is found in third iteration within sorted array.

# Task 5

1. In paging memory management process address space is broken into equal size blocks (pages) (512 - 8192 bytes). Process size is measured in the number of pages. Main memory is divided into small fixed-sized blocks of (physical) memory (frames) and the frame size is kept the same as that of a page to have main memory optimum utilization whereby avoiding external fragmentation. It is often contrasted with segmentation, so it is decided to enumerate main difference between the two:

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Key** | **Paging** | **Segmentation** |
| 1 | Memory Size | Process address space consists of fixed-sized blocks - pages. | Process address space consists of varying-sized blocks - sections. |
| 2 | Accountability | OS divides the memory into pages. | Compiler calculates the segment size, the virtual address and actual address. |
| 3 | Size | Available memory determines page size. | User determines section size. |
| 4 | Speed | This technique is faster in terms of memory access. | Segmentation is slower than paging. |
| 5 | Fragmentation | Paging can cause internal fragmentation where some pages may not be utilized. | Segmentation can cause external fragmentation where some memory block is not even used. |
| 6 | Logical Address | A logical address consists of page number and offset. | A logical address consists of section number and offset. |
| 7 | Data Storage | Page table stores the page data. | Segmentation table stores the segmentation data. |

(<https://www.tutorialspoint.com/difference-between-paging-and-segmentation>)

.One of the advantages of using this technique is that it eliminates the issue of External Fragmentation so that it is easy to swap since everything is the same size.

One of the advantages of using this technique is that paging increases the computer hardware cost as page addresses are linked to hardware.

Memory stores store information about page tables. Some memory space is underutilized when available blocks are insufficient for address space for jobs to run. Since physical memory is of equal-size split, it allows internal fragmentation (<https://medium.com/@esmerycornielle/memory-management-paging-43b85abe6d2f>).

2. a) A logical address of <2, 85> means the 2nd page with offset of 85. In order to find out corresponding physical address it is required to look up frame number from PMT table. According to the table *frame number = 5*. And it is required to *multiply frame number by page size and add the offset* which is = 5 \* 1024 + 85 = 5205

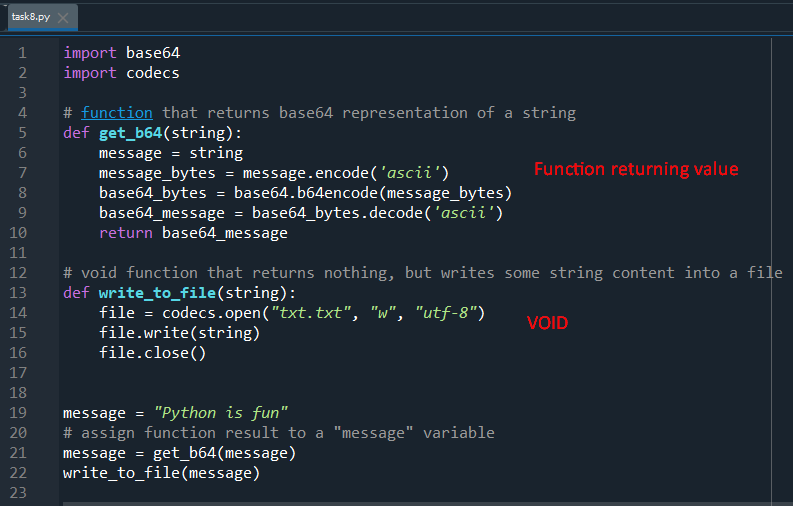
b) In given sample problem, the offset (1026) is larger than the frame size, so it is illegal address.

# Task 6

# Task 7

# Task 8

Void functions are used in order to perform some action status of which is not interesting to a software engineer whereas functions (that return some data after operation happens) return some indicator in order to tell about operation success or some calculated value that will be useful in later programming activities. For example, let’s observe this code (**task8.py** file is auxiliary to this paper and can be found next to this Word File):



This code shows how a text’s base64 is generated and written into a file called txt.txt.

There is a usage of get\_b64() function that converts string byte array to base64 string format and *returns* the result. There is also write\_to\_file() custom void function used that does only write operation which *does not require returning* the status. This operation can also return something that shows some success message of write operation, but it all depends on technical specification of the software.

# Task 9

# Reference list