タイトル	プログラミングＢレポート
課題番号	課題 3
課題提出年月日	令和 3 年 1 月 7 日
教員名	首藤 裕一
提出者	09B20065 , ブアマニー　タンピモン

1 課題内容

I did the Fundamental project (基本課題) and advanced assignment (発展課題). Fundamental project can be separate into 2 assignments:

1. Defining the address of location in Japan using Japanese ZIP code
2. Searching for places in japan with indicated words

Advanced project's assignment is the continuation of Fundamental project's(2) which assign the program to scale down the possible places using new indicated words.

2 アルゴリズムの説明

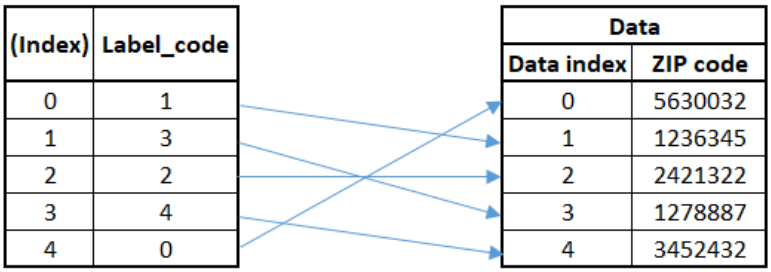
The processing in this program can be defined into 2 orders: the pre-processing and the searching. As the address data is given in form of text file, the program runs through the file and saves all data in form of list "data" declared as global variables that collects prefecture, city and town along with their code. As the data are not sorted, pre-processing is required for the search to be done quickly and efficiently. After the data is sorted, the program will need to search in accordance to 2 assignments given.

2.1 Pre-processing

The data from file is being saved in the list "data" in forms of unsorted array. As moving data location directly requires change in the location of all variables (code, prefecture, city, and town) which will waste a lot of time, the program instead give out number index to each data and declare array that will define the correct order of the data. For the ZIP code pre-processing, the program use quick sort to arrange integer and describe the order of index in the array called "label_code". Other variables including prefecture and city the program will group the data with same name together in array variables called "label_pref" and "label_city" respectively. After processing, data will no longer not be accessed by data's numerical order but instead by label_code, label_pref or label_city order depending on what sorted-data the program is needed so that it will be sorted accordingly.

2.1.1 Pre-processing ZIP code - Quick sort

The image 1 express how label code works. Instead of moving ZIP code and other data directly that would require a lot of memory usage along with time for moving several variables in one move, the program move only the index part. It could be seen that the number on the left: 1,3,2,4,0 is not in a numerical order but if we use those number as a data index and convert it to ZIP code, the number will be 1236345 , 1278887 , 2421322 , 3452432 , 5630032 which is in numerical order. The program use quick sort to sort the following data which will be explained in 3.6.1 Pre-processing (init function).



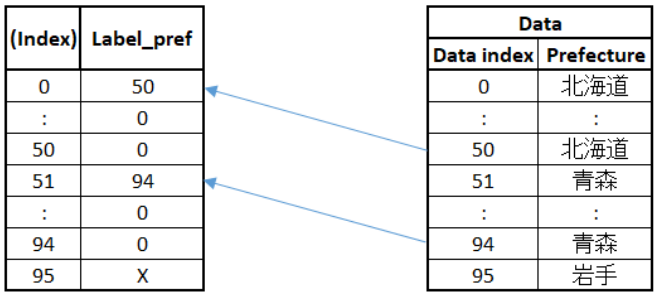
(Index)	Label_code
0	1
1	3
2	2
3	4
4	0

Data	
Data index	ZIP code
0	5630032
1	1236345
2	2421322
3	1278887
4	3452432

図 1: Label_code's function

2.1.2 Pre-processing prefecture and city - Grouping

In the second part of the program, the user will input part of the name and the program has to output all possible choices of the address with the name mentioned. As the name might not be input from starting letter, it's impossible to sort data and binary search it like how we do it ZIP code. Instead, we can group the data with the same name so that it will reduce the time used to search for it. For example in prefecture data, there's a total of 124341 data units with only 47 varieties: from Hokkaido to Okinawa. With that, the program will declare array label_pref with the same size as the amount of data, and initialize the element's data to 0. While preprocessing the data, the index of the last data with the same name as the first data will be put in the first data with that name so that when searching, only the first data will be compared and then skip to other different words. Using images 2 as an example, the program will first set the latest data index as 0 and prefecture name as 北海道 and run the program until they find the first data with different prefecture name (Data index 51:青森). The program will then save the last data index with same prefecture name (Data index 50) to label_pref of the first data index (index 0). The program will do the same for other 47 prefectures. For example 青森 is the prefecture of data 51 - 94, so the program will record number 94 on Label_pref number 51.



(Index)	Label_pref
0	50
:	0
50	0
51	94
:	0
94	0
95	X

Data	
Data index	Prefecture
0	北海道
:	:
50	北海道
51	青森
:	:
94	青森
95	岩手

図 2: Label_pref and Label_city's function

2.2 Search

2.2.1 Search for address using ZIP code

As shown on image 3, binary search is used quickly find location for the mentioned ZIP code. The program started with variable pointing at the very left and right of the array. Middle variable is indicated from averaging the right and the left. The ZIP code of the index of the middle index mentioned will be compared to the ZIP code that we want to find. If the desired ZIP code is more than that, right variable's number will be replaced by middle number. Else, if the desired ZIP code is less than the ZIP code in the middle, the left variable's number will be replaced by middle number. In the case that the ZIP code is the same, it means that we are able to find the ZIP code we wanted so the program will know the index number of that ZIP code which is the same as the ZIP code of the address.
In some cases like ZIP code: 0191834, the code represent more than one address mentioned below:

0191834: 秋田県大仙市南外揚土山
0191834: 秋田県大仙市南外北田黒瀬

To be able to present all data with the same ZIP code, after the program found one data with the search ZIP code, the program will continue search with the adjacent index in label code until the leftmost and the rightmost of the data consists of data with different ZIP code, then the program will print out all data with the same ZIP code from the left to the right.

(Index)	Label_code
0	1
1	5
2	2
3	4
4	6
5	3
6	0

Data	
Data index	ZIP code
0	5630032
1	1236345
2	2421322
3	4212243
4	2743127
5	1278887
6	3452432

Input ZIP code : 5630032

Search round	variable				ZIP code of middle	compare result
	left	right	middle	Label_code		
1	0	6	3	4	2743127	<
2	4 = (3+1)	6	5	3	4212243	>
3	4	4 = (5-1)	4	6	3452432	=

図 3: Searching for ZIP code

2.2.2 Search for places in japan using indicated word

As shown on the image 4, the program first start searching on prefecture, then city and town respectively. At first, the program would check if the prefecture name on that certain index is the same for the one their search for or not. If the result is not the same, the program would start comparing cities in those prefecture. If there's no cases with similarity in city, the program would start comparing each town name. The program will check in the order of prefecture name, one city name and finish checking all town in the city before moving on to the next city name. With that, when the program would lessen time going through similar prefecture name or city name in searching once.

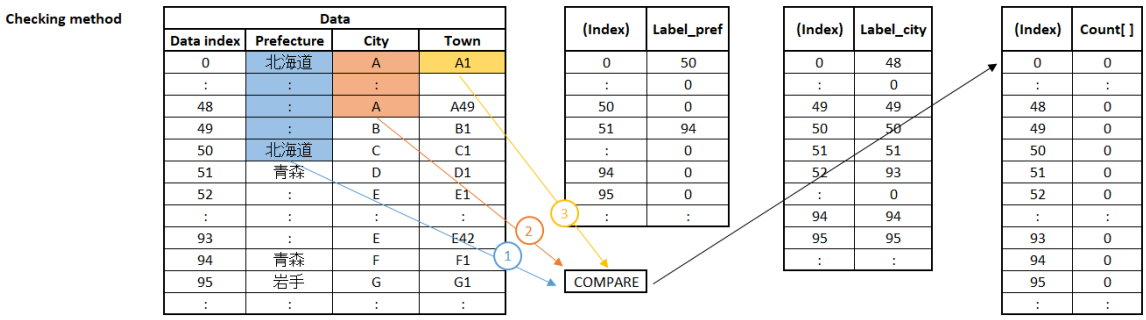


図 4: Searching for places using indicated word

If the same result is found on prefecture name, the program would go to label_pref to check the range of prefecture with that certain prefecture name(2) and convert all count[] variable in that range to 1(3).To prevent checking at the those data that has already considered as the same to search data, the program would then set the data index to the number after the ending range. The image 5 show searching process when query are prefecture name. 青森 is located from index 51 to 94 so after searching, the index is then set to 94+1 = 95) and then continue searching in 95 position onward. The program then repeat the process until it reach the last index of the prefecture.

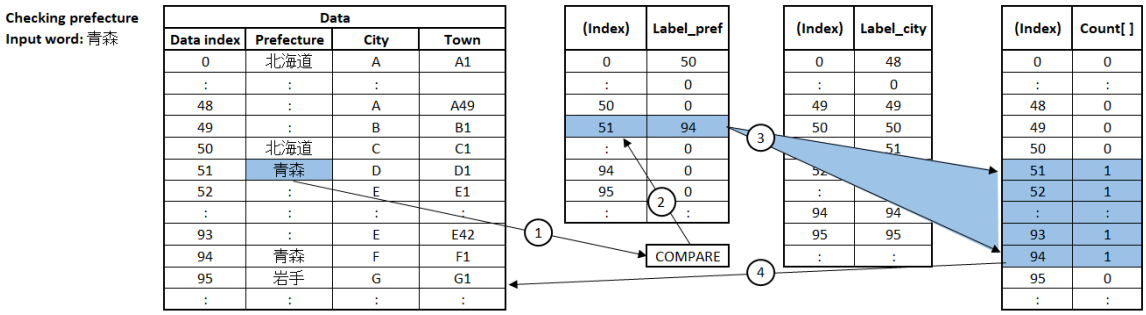


図 5: Ordering for prefecture search

When the keyword is not in the prefecture, after finish searching one prefecture, it would start working on searching each city in that prefecture. As they're many places with the same city name, Label_city is created to decrease the time use for checking all places with same city name at once. After checking if the name has the same word to the search word or not, the program would check in Label_city the range of the city with that same name and automatically set the count data from the current index number to

that last number to be the same. Then the index would move on to the number next to the ending range. The image 6 show searching process when query are city name. As the range is 0-48, after checking, the program would then point to index 49).In case that all city names are different, program would start searching for town name among those same city name directly from the first town to the last town.

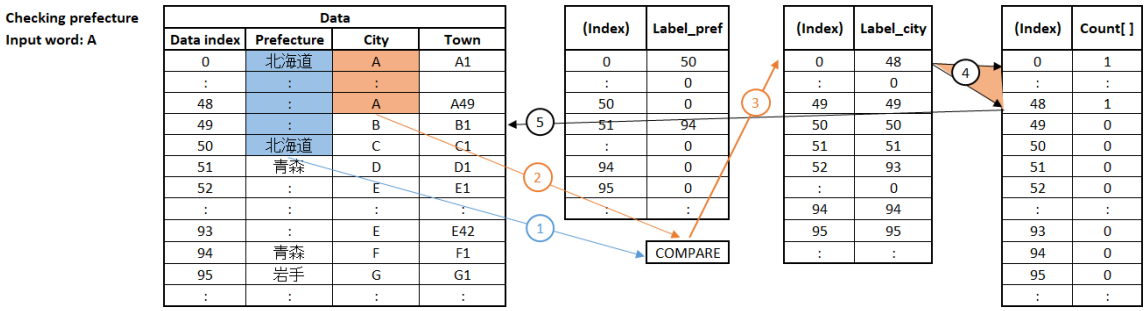


図 6: Ordering in case that city name is searched

2.2.3 (Advance assignments) Search for places in japan using several indicated words

For the advance assignments, the only difference from the normal search in 2.2.2 is that in the added words difference from setting count=1 like how the normal search does, the program would instead add 1 to count (1->2 / 2->3) which will indicate how many words that are similar to the word that was searched. In the advanced assignment, the program want to get the address of places with ALL indicated words so we save the variable max_count to count how many word we have and at the end of the program we would compare the number in count variable in each address to max_count and would only print the data if the 2 data have the same number.

3 プログラムの説明

3.1 入力形式

The program receives input in forms of numbers and Japanese letters from keyboard. At the start, the user input number between 0-2 answering what kind of search they prefer the program to do.The program could be seen in image 7.

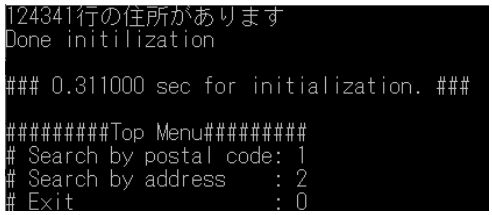


図 7: Starting program

If the user input '1', stating that the program would search for postal code, the program would ask the user to input the postal code they want to search. The program could be seen in image 8.

図 8: Search by postal code

Else, if the user input '2', the program would ask the uset to input string of word they want to search. After seeing the results, the program will let the user input 0/1 whether the user want to refine the data by continue searching or not. If the user wish to continue searching, the program will continue taking input from the user in form of words they want to use to refine data.The program could be seen in image 9.

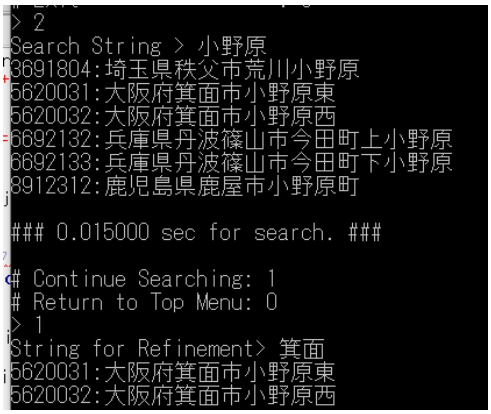


図 9: Search by address

3.2 出力形式

Program sends formatted output to the screen. English's used in asking questions and Japanese's used when accessing data. Questions's outputted on the screen and wait for input to continue program. In the first assignment, program sends output of address of the given postal code. In the second and extra assignments, program prints out all possible address that are conformed to the given conditions.

3.3 データ構造

The address data this program used is Shift_JIS CSV format files retrieved from the university webpage CLE. Japan post, a government-owned postal and package delivery service in Japan, provided the original data in forms of zip file inside the company's website which the link is attached below.

http://www.post.japanpost.jp/zipcode/dl/kogaki/zip/ken_all.zip

When program is complied, data from CSV format files is read and saved to the program in struct data

variable called "dataset" which contain integer variable code that's used to save postal code, and string variable pref, city, and town that's used to save prefecture, city and town name respectively.

3.4 大域変数

The global variables in this program are summarized in table 1.

表 1: Global Variables in the program

Variable name	Variable type	Usage
Dataset	Array of struct data variable Member: Integer - code String - pref, city, town	Collect essential postal data Code - collect postal code Pref - collect prefecture name City - collect city name Town - collect town name
Dataset_length	Long	The amount of postal data
Mode	Int (Number between 0/1/2)	Keep the mode of search the user want to do (0:Nothing,1:Postal Search,2:Keyword Search)
Refine_flag	Int (Number between 0/1)	Check if there's refinement done or not (0: No, 1: Yes)
Query	String (Array of char length of 200)	Collect postal code/Keyword inputted by user
Label_code	Array of int	Collect the index of postal code in dataset in their postal-code order
Label_pref	Array of int	Collect the range of index with same prefecture name (putting the last index number with that name on the first label_pref index with same name)
Label_city	Array of int	Collect the range of index with same city name (putting the last index number with that name on the first label_pref index with same name)
Correct	Array of int	Collect the amount of time the dataset in that certain index have part of the keyword that has been searched
Maxcorrect	Int	Collect amount of time keywords've been searched

3.5 関数の設計

There's a total of 19 functions in this program (main function included). The connection of each functions are shown in image 10.

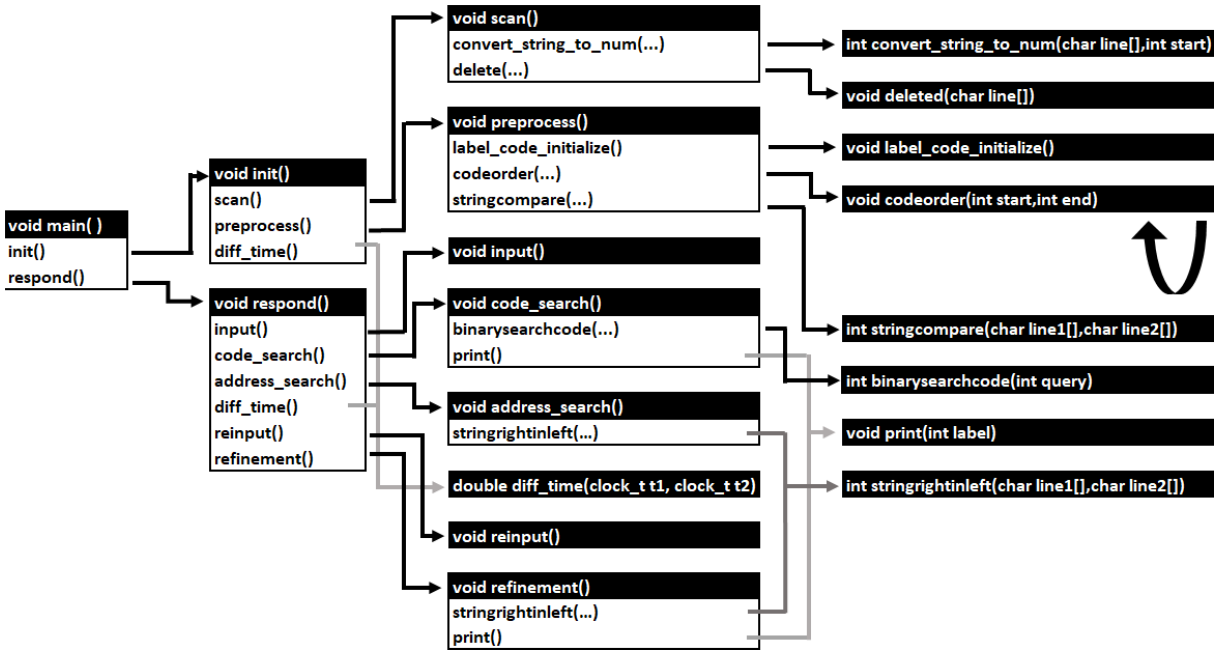


図 10: How each function are connected to each other

It could be seen on the left is the main function, the first function to be executed. In each function name, downward white boxes show the function which is called when executing the upper black function. The arrow will point from those white box to black box that explains the input and output if data together with the function called in that certain function. Table 2 below explains the input and output of these functions.

表 2: Function in the program

Function name	Input	Output
Main()	(void)	(void)
Init()	(void)	(void)
Respond()	(void)	(void)
Scan()	(void)	(void)
Preprocess()	(void)	(void)
Input()	(void)	(void)
Code_search()	(void)	(void)
Address_search()	(void)	(void)
Diff_time()	(clock_t)t1: time before processing (clock_t)t2: time after processing	(double) difference between two processor time
Reinput()	(void)	(void)
Refinement()	(void)	(void)
Convert_string_to_num()	(char) line[]: the string that will be converted (int) start: the starting digit of conversion	(int) the string line[] in form of integer
Deleted()	(char) line[]: the string with ""	(void)
Label_code_initialize()	(void)	(void)
Codeorder()	(int)start: the starting index that will be sorted (int)end: the ending index that'll be sorted	(void)
Stringcompare()	(char)line1[] : 1st string that'll be compared (char)line2[]: 2nd string that'll be compared	(int) if 2 string are the same: 1 else :0
Binarysearchcode()	(int) query: ZIP code that'll be searched	(int) index of the postal code if not found:-1
Print()	(int) label: index that ZIP code'll be printed	(void)
Stringrightinleft()	(char) line1[]: long string of postal data (char)line2[]: word that'll be searched in line1[]	(int) if line2 is in line1: 1 else :0

3.6 Explanation of each functions

As stated in 「2. アルゴリズムの説明」 The processing in this program can be defined into 2 orders: the pre-processing and the searching. In the main function, it can be seen that init() is in charge of pre-processing and respond() is in charge of searching.

3.6.1 Pre-processing (init function)

As init() function are in charge of preprocessing, 3 functions inside init() including scan() ,preprocess() and diff_time() works as follow:

a scan():

The program check whether it can be read or not. If it's readable, the program get the data row by row and separated them to be postal code, prefecture, city and town respectively. As much of data are raw, convert_string_to_num() and deleted() are used to make data easier to be processed.

a1 convert_string_to_num():

this function process string of postal code and convert it to integer. At first the the string are in form 7 character number with 2 quotation marks on the very front and back (for example: "5630032"). This function received the string along with number stating which index it should start with (1 as the 0 index have the quotation marks). The program would read the letter one by one, and if the letter ascii number is between '0'(48) to '9'(57) it would convert the letter to number by decreasing it with '0'(48) and adding it to variable ans. As it's 10-based number, as the new digit number is to be inserted, the program will times the ans variable by 10 to increase the digit up by 1. After the whole string is bring run through, the sum in ans variable is returned.

a2 deleted():

As raw data have quotation marks before and after the word(for example: "北海道"), delete program will delete those quotation marks by moving each letter's position up by 1. At the last position, the program will converted the letter to the end of string. As when you pass an array to the function you're passing a pointer, the modification are effective without using return function.

b preprocess():

As the program come in form of unsorted data, the program need to preprocess it to make sure that it will search fast and efficiently. In preprocess, the function call label_code_initialize, Codeorder() to preprocess ZIP code data, and Stringcompare() to preprocess other information.

b1 Label_code_initialize():

This function initialize label_code data by setter number in every array elements the same as their index. (For example: label_code[10] = 10)

b2 Codeorder():

This function use to sort the postal code in order as it was a high-efficient sorting algorithm. As the postal data is conjoined to a lot of data in the dataset struct, the program will not moved the postal data directly but instead move the index data in label_code instead. Using recursive function, the function codeorder(start, end) will pick up the variable in the middle(start+end /2) and compare every data between start and end to see if the data is more or less than the middle data. All of the data less than the middle data will be pushed to the left and others will be pushed to the right. To save memory, instead of ordering data in the new array the program will push data in the blank element. If there's no blank element at the current moment the program will push one unprocessed data out, push the processed data in and start working on that unprocessed data. Image 11 is the example of how this program's quick sort works in small amount of data. This comparison work on array between indexes 0-7. First the program get data from index 4, the middle number between 0 and 7, and set it as a comparison number. The arrow represent the non-yet-arranged index. The program start working from index 0 by comparing each index's number to the middle number. If the number is less than

the middle number such as case 1, the number is pushed to the not-yet-arranged index on the left. In contrast, if the number is more than the middle number like in case 2, the number is pushed to the not-yet arranged index on the right. It could be seen that the program want to push 18 in index 7 but there's number 12 in there, so the program push 18 in and compare 12 next. If there's no number in the block we want to push in (number inside is -1) the program will just put on that variable and move on to the next right block from the left. From case 8 it could be seen that all number have been compared and the arrow ended up at the same location. At that case we will put the middle number in that block and consider as the end of separating once. The program then repeat this process but working on array between 0-3 and 5-7 so that all array is sorted.

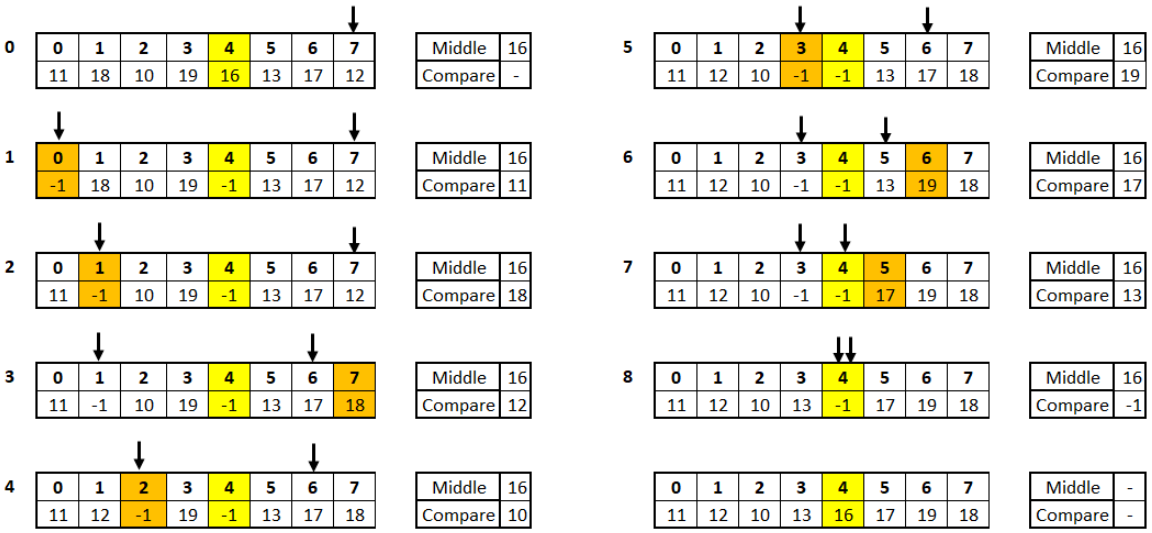


図 11: How quick sort works

b3 Stringcompare():

The program directly compare two strings, character by character to see if all the character in the same index are the same. If any character are different, the program will return 0. After checking, the program would check once again if the ending of array are at the same place. If it's in the same place, it will print out 1, or else the program will return 0.

c diff_time() :

From the start and at the very end of init() function, the program has record the starting processing time and ending processing time. Using this function, the program will find the difference between this two time interval and return the number which represent the time used for initialization back to init() function. The number will then be retrieved and printed in init() function.

3.6.2 Searching (respond() function)

3.6.2 Searching (respond() function) The respond() function will call for 6 different function: input(), code_search(), address_search(), diff_time(), reinput() and refinement(). As diff_time has already been explained in the 3.6.1 Pre-processing, the other five will be explained below.

a input():

Input function asks user about type of searching they prefer between

- search by postal code
- search by address
- exit

The answer is saved in global variable mode. If mode=1,2 the program will ask for the search data and save it as a form of string in string variable called query.

b code_search():

The given code is searched by using binarysearchcode() function to find the one of the index with that postal code. As there might be more than one place with the same postal code, data before and after that index will be checked to find the range of data with the given postal code.

b1 Binarysearchcode():

The variable left and right are first initialized as index of the start(0) and the end of data. Middle are the average between left and right (left+right/2). As dataset[label_code[middle]].code are checked, if the data is more than the query, variable right will be convert to middle-1 as the data is sort and all data between middle and right are more than query. Vise versa, if the data is less than query, variable left will be converted to middle+1. The program will keep on lessen the range until the middle index contain the given query, which will be returned to code_search() function.

c address_search():

As explained in "2.2.2. Search for places in japan using indicated word", the program is searched in order of prefecture, city and town respectively. Function stringrightinleft() is used to check if the query exist in the address word or not. If the program found the query in the search string, correct array of that position will be converted to 1. After checking all data, the program will run the correct array and print out all data in the index that correct[index] = 1.

c1 stringrightinleft():

2 string will be compared letter by letter. If one of the letter different, the program will restart comparing from the first position. If all the letter in the right string are in left string and in the right order, the program will output 1, else, it will output 0.

d re_input():

The program ask if user want to refine the recent search by adding more query. If so, the program will retrieved the data and put it in query, preparing for refinement.

e refinement(): Similar to address_search() the program will compare in the order of prefecture, city and town. However, instead of changing the correct array in that index to 1, the program plus 1 to the array. Maxcorrect variable collect the amount of time this refinement has been done. When printing out the data, correct[index] will be compared to maxcorrect to see if all the query is in that address then print out all data with the following conditions.

4 工夫した点

In my opinion, the hardest part to work on is pre-processing where I did the postal code sort and grouping. As I believe that these kind of program have a fixed data which will not be changed very often, good pre-processing, even though it might take a long time, is the most important to make the data process fast afterward.

Doing the indirect quick sort where the index moved instead of the number is very confusing. To get the data from dataset, the sorting position have to go through label_code to get the index for the dataset which is more complex than the index-to-elements array we often do. Sorting this way require only moving the index number which is faster from moving all data. (Example: number ranking in the middle of the dataset=dataset[label_code[middle]].code)

For grouping, I realized when seeing the raw data that despite the fact that the postal code is not in order, data like prefecture and city are sorted nicely. With that I decided to group it up by the original index number. With that, the same string will be checked only once, and if the string is what we searched for the other data in the same range will immediately be changed as well. Still, as the postal code are not yet sort, the result of print or not will be saved using the correct array and printed out later by order. Top-down search from a prefecture name to the town name will decrease the time as the program can skip the city and town search immediately if the similarity is already found on the prefecture string, or skip the town search if the similarity is found on the city string. The correct array is kept in form of numbers stating the times it ’s similar to the given data as in real life search engine, a mistype or overtyping might occur. Using the number in correct[] we will be able to rank the relevancy of data.

5 感想

Theoretically, quick sort and merge sort have similar average time complexity of n log n [1]. Still, despite the fact that the postal is not completely sort beforehand, many of the data are not very far from their destination and often the order are only swapped within the prefecture. With that, it might be practically true for merge sort to work faster than quick sort. A further experiment using different type of sorting is needed to improve the preprocessing time.

6 考察

I feel interested from the start that I heard about this assignment. However I cannot start this project right away as I spent the first 3 practice weeks (演習) fixing my computer as it was not configured for using Japanese language. I used to believe that using English is enough in programming but that was wrong, learning in Japan required me to process a lot of data in Japanese. I’m looking forward to buy new japanese computer soon! Despite the short time I have for this project, I tried searching for ways that google did their search engine [2] and learn a lot of new stuff. I have problem doing quick sort as I often confuse the index of label_code and the label code elements itself so I wrote normal quick sort first and then change it bit by bit. I hope I get to know the average processing time of the class and the max and min for me to know my place and improve myself better.

7 作業工程

- アルゴリズム設計 - I searched for ways that google did [2] but it didn't seems to suit with our data style. So I tried to imagine what I would do if I were to sit in the computer and hand processing the data. The idea of making label_code, label_prefecture and label_city come up as I believe it would make the program quicker than simply searching alone.
- プログラム設計 - Before writing, I tried to draw up some programming ideas and test those ideas using small amount of data. Hand coding make me understand my program more and allow me to keep track to even small details. Creating variable and function name that shows the purpose of the use helped me or even readers understanding the code easily.
- コーディング - Program template was received from CLE. Complex function was written in a different file and was merged in after completed. I tried to write easy, common program first and develop it. For example, my index-position change quick sort causes me a lot of problem because layers of data I have to compare is confusing (dataset[label_code[num]].code instead of num). So I tried simplify it by writing normal quick sort at first and gradually change the variable to suit the real program.
- デバッグ - To debug the mistake, all variable was printed out along with data which are supposed to be change out to check it after almost every line of code. After sending to professor, I got a feedback on test case I didn't know exist (Problem 1 Test case 0191834: two places with same postal code), so the code was edit according to it.
- レポート作成 - The work started on Microsoft word at first and moved to LateX after finish writing the details. Before writing, I tried to come up with ideas of what to write on each part, to make sure that I wouldn't talk about same thing twice. Images were made via Microsoft excel before each writing section starts but after brainstorming to make sure it'd be explained clearly.

8 参考文献

[1] Analysis of different sorting techniques - GeeksforGeeks. (2020). Retrieved 10 December 2020, from https://www.geeksforgeeks.org/analysis-of-different-sorting-techniques/

[2] How Search Works (for beginners) | Google Search Central. (2020). Retrieved 7 December 2020, from https://developers.google.com/search/docs/basics/how-search-works