

Project 4, Program Design

Problem 1- Pointers as array arguments and using pointers to process arrays (50 points)

Suppose you are given an array of positive integers of length n . An array is called good if the elements of the arrays have the same parity (even or odd). For example, elements of array [1, 5, 7] have the same parity odd, elements of array [2, 3, 4] do not have the same parity.

Example input/output #1:

```
Enter the length of the array: 7
Enter the elements of the array: 3 5 1 4 9 8 11
Output: not good
```

Example input/output #2:

```
Enter the length of the array: 4
Enter the elements of the array: 2 16 8 4
Output: good, all elements are even
```

Example input/output #3:

```
Enter the length of the array: 3
Enter the elements of the array: 5 13 17
Output: good, all elements are odd
```

1. In the main function, ask the user to enter the length of the input array, declare the input array. Then ask the user to enter the elements of the array.
2. The program should include the following function. **Do not modify the function prototype.**

```
void find_parity(int *a, int n, int * all_even, int
`*all_odd);
```

a represents the input array with length n. The `all_even` and `all_odd` parameters point to variables in which the function will store whether all of the array elements are all even or all odd.

This function should use pointer arithmetic– not subscripting – to visit array elements. In other words, eliminate the loop index variables and all use of the `[]` operator in the function.

3. In the main function, call the `find_parity` function to find whether all of the array elements are all even or all odd, or neither.
4. The `main` function displays the result.

Problem 2 – Check the inventory (50 points)

A certain computer factory has grown over the years, and it now must improve its production lines to meet the demand. The first problem that the managers identified is the lack of organization in their inventory. They cannot even compute how many computers they can build using the parts available in storage. Luckily, they know that there are at most **K** different part types and each part has a unique code from **1** to **K**. Also, they know they have **N** parts in storage through a list of part codes that was created during this reorganization. Knowing that a computer is composed of **K** distinct parts, one of each of the existing part types, can you write a program that computes how many computers can be assembled with the parts in storage?

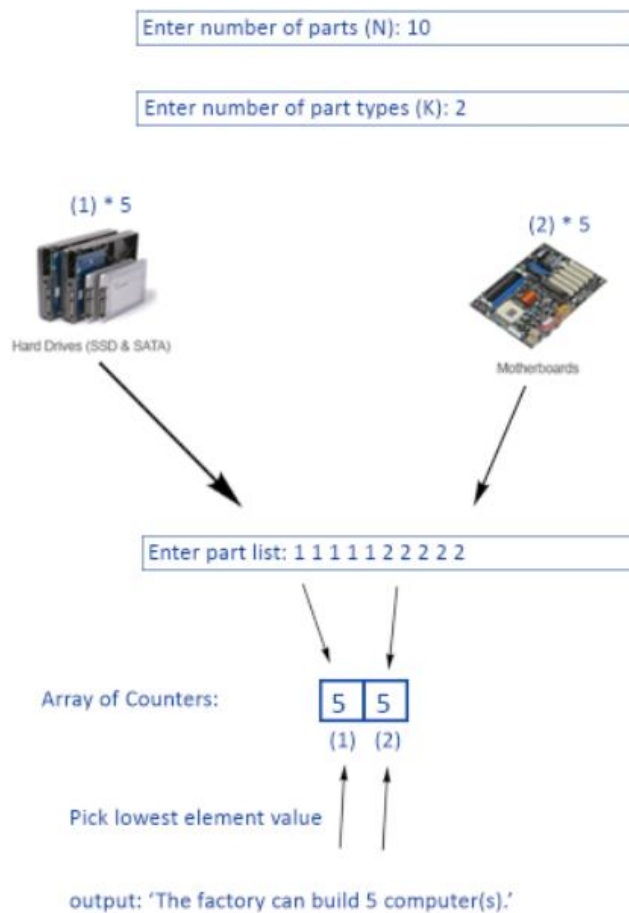
1. The program takes the number of parts in storage (**N**), the number of part types (**K**) and a list of **N** part codes from **1** to **K**. It computes the number of computers that can be assembled with the existing parts.
2. Input validation: the program validates the number of parts in storage ($1 \leq N \leq 1000000$), the number of part types ($1 \leq K \leq 10000$), and the code c_i of the i -th part in storage ($1 \leq c_i \leq 1000$). If any of the input values are invalid, the program prints a message and exits.

Hint: use the number of part types to create one array of counters. For example, in Example #1 below, the array of counters for the part list will be [5 5] since there are 5 of part type 1, and 5 of part type 2, therefore the factory can build 5 computers. In Example #2 below, the array of counters will be [4 4 4 6 2], therefore the factory can build 2 computers.

Example #1:

Enter number of parts (N): 10
Enter number of part types (K): 2
Enter part list:
1 1 1 1 1 2 2 2 2 2
The factory can build 5 computer(s).

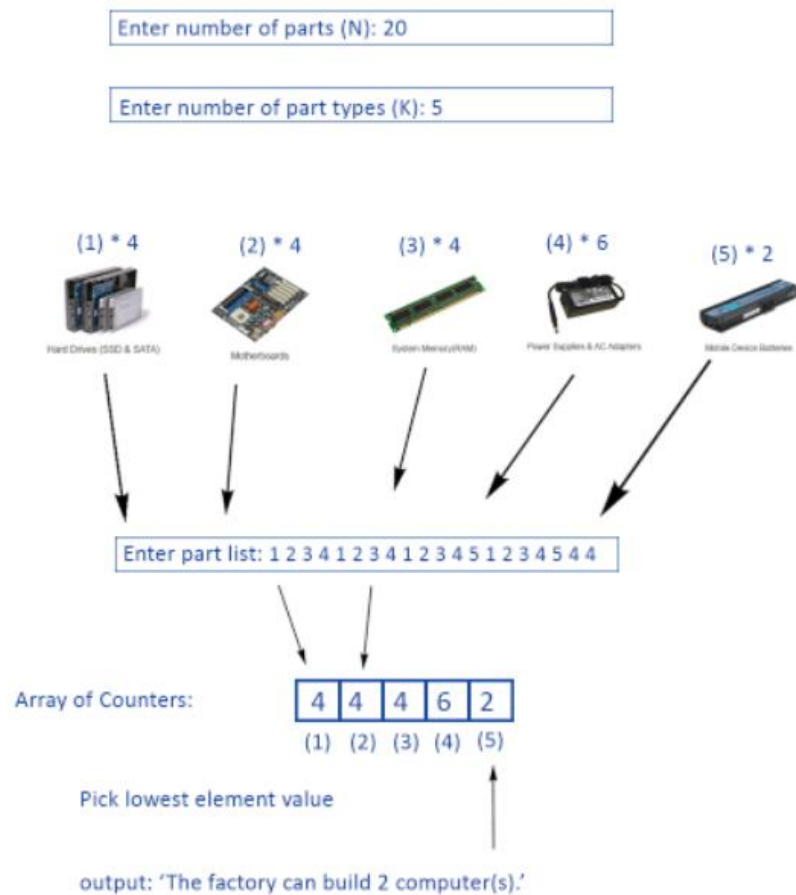
Illustration for Example #1:



Example #2:

Enter number of parts (N): 20
Enter number of part types (K): 5
Enter part list:
1 2 3 4 1 2 3 4 1 2 3 4 5 1 2 3 4 5 4 4
The factory can build 2 computer(s).

Illustration for Example #2:



1. In the main function, ask the user to enter number of parts, number of part types, and the part list.
2. The program should include the following function. **Do not modify the function prototype.**

```
int find_minimum(int *a, int n);
```

a represents the array of counters with length n. The function returns the minimum value of the array a .

This function should use pointer arithmetic– not subscripting – to visit array elements. In other words, eliminate the loop index variables and all use of the [] operator in the function.

3. In the main function, call the `find_minimum` function to find the minimum value of the array of counters.
4. The `main` function displays the result.

Other requirements and submission:

1. Program name:
project4_parity.c
project4_inventory.c
2. Compile on *student cluster (sc.rc.usf.edu)*.

```
gcc -Wall project4_parity.c  
gcc -Wall project4_inventory.c
```

3. Change Unix file permission on Unix:

```
chmod 600 project4_parity.c  
chmod 600 project4_inventory.c
```

4. Test your program with the shell script on the student cluster:

```
chmod +x try_project4_parity  
./try_project4_parity
```

```
chmod +x try_project4_inventory  
./try_project4_inventory
```

5. Download the program *project4_parity.c* and *project4_inventory.c* from student cluster and submit it on Canvas>Assignments.

Grading

Total points: 100

1. A program that does not compile will result in a zero.
2. Runtime error and compilation warning 5%
3. Commenting and style 15%
4. Functionality 80%

Programming Style Guidelines

The major purpose of programming style guidelines is to make programs easy to read and understand. Good programming style helps make it possible for a person knowledgeable in the application area to quickly read a program and understand how it works.

1. Your program should begin with a comment that briefly summarizes what it does. This comment should also include your name.
2. In most cases, a function should have a brief comment above its definition describing what it does. Other than that, comments should be written only *needed* in order for a reader to understand what is happening.
3. Variable names and function names should be sufficiently descriptive that a knowledgeable reader can easily understand what the variable means and what the function does. If this is not possible, comments should be added to make the meaning clear.
4. Use consistent indentation to emphasize block structure.
5. Full line comments inside function bodies should conform to the indentation of the code where they appear.
6. Macro definitions (`#define`) should be used for defining symbolic names for numeric constants. For example: **`#define PI 3.141592`**
7. Use names of moderate length for variables. Most names should be between 2 and 12 letters long.
8. Use underscores to make compound names easier to read: **`tot_vol`** or **`total_volumn`** is clearer than `totalvolumn`.