Systems Programming Project

fileReader.c

```
#include <stdio.h>

#define STRLEN 20
#define PDEBUG 1
#define DPRINT(fmt, ...) do { if (PDEBUG) fprintf(stderr, "%s:%d:%s(): "fmt,\
                __FILE__, __LINE__, __func__, ##__VA_ARGS__); } while (0)

int main(int argc, char* argv[]) {
        if(argc != 2) {
                DPRINT("Usage: fileReader <filename>\n");
                return -1;
        }

        FILE* in;
        if((in = fopen(argv[1], "r")) == NULL) {
                DPRINT("Error: fopen returned NULL\n");
                return -1;
        }

        char str[STRLEN];
        while(fscanf(in, "%s", str) != EOF) printf("%s\n", str);

        fclose(in);
    return 0;
}
```

fileReader.c is pretty simple, it makes sure it has the correct number of arguments, and exits if it doesn’t. It then opens the file given as an argument, and if this is successful it uses fscanf to read the file string by string until EOF. Every time it reads a string, it prints it to stdout. Finally, it closes the file and returns 0.

aSorter.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define STRLEN 20
#define PDEBUG 1
#define DPRINT(fmt, ...) do { if (PDEBUG) fprintf(stderr, "%s:%d:%s():
"fmt,\
        __FILE__, __LINE__, __func__, ##__VA_ARGS__); } while (0)

void bsort(char** arr, const int len) {
    for(int i = 0; i < len; i++) {
        for(int j = 0; j < (len-i-1); j++) {
            if(strcmp(arr[j], arr[j+1]) > 0) {
                char* tmp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = tmp;
            }
        }
    }
}

int main(int argc, char* argv[]) {
    if(argc != 2) {
        DPRINT("Usage: aSorter <number>\n");
        return -1;
    }

    int strsize = sizeof(char)*STRLEN;
    int currsize = strsize*10;
    int count = 0;
    char** strarr = (char**)malloc(currsize);
    char str[STRLEN];

    while(fscanf(stdin, "%s", str) != EOF) {
        if((count*strsize) > currsize) {
            currsize += currsize;
            strarr = (char**)realloc(strarr, currsize);
        }

        strarr[count] = strdup(str);
        count++;
    }

    bsort(strarr, count);

    for(int i = 0; i < count; i++) printf("%s%s\n", argv[1], strarr[i]);

    for(int i = 0; i < count; i++) free(strarr[i]);
    free(strarr);

    return 0;
}
```

aSorter.c first checks to make sure it has the correct number of arguments, then it creates a string array with an initial size of 10 and begins reading string by string from stdin until EOF using fscanf. These strings are put into our string array and, if required, the string array is resized to fit more strings. When the while loop ends, the string array is sorted using the bsort function, and then the sorted array is printed to stdout along with a preceding number that was specified through arv[1]. Finally, the array is freed and the program finishes.

theMerger.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define STRLEN 20
#define PDEBUG 1
#define DPRINT(fmt, ...) do { if (PDEBUG) fprintf(stderr, "%s:%d:%s():
"fmt,\
        __FILE__, __LINE__, __func__, ##__VA_ARGS__); } while (0)

int get_num(char* str) {
    char num[10];
    int j = 0;
    for(int i = 0; i < strlen(str); i++) {
        if(str[i] >= '0' && str[i] <= '9') {
            num[j] = str[i];
            j++;
        } else {
            return atoi(num);
        }
    }
}

char** merge(char** arr1, char** arr2, int size1, int size2) {
    char** ret = (char**)malloc(sizeof(char*)*(size1+size2));
    int i = 0, count1 = 0, count2 = 0;
    while(count1 < size1 && count2 < size2) {
        char str1[10];
        char str2[10];
        int num1 = get_num(arr1[count1]);
        int num2 = get_num(arr2[count2]);
        sprintf(str1, "%d", num1);
        sprintf(str2, "%d", num2);
        char str4[10];
        char str5[10];
        strncpy(str4, &((arr1[count1])[strlen(str1)]), strlen(arr1[count1]) -
strlen(str1));
        strncpy(str5, &((arr2[count2])[strlen(str2)]), strlen(arr2[count2]) -
strlen(str2));
        if(strcmp(str4, str5) < 0) {
            ret[i++] = strdup(arr1[count1++]);
        } else {
            ret[i++] = strdup(arr2[count2++]);
        }
    }
    while(count1 < size1) {
        ret[i++] = strdup(arr1[count1++]);
    }
    while(count2 < size2) {
        ret[i++] = strdup(arr2[count2++]);
    }
    return ret;
}

void freearr(char** arr, int size) {
    for(int i = 0; i < size; i++) free(arr[i]);
    free(arr);
}

int main(int argc, char* argv[]) {
    FILE* out;

    if(argc == 1) out = stdout;
    else if(argc == 2) {
        if((out = fopen(argv[1], "w")) == NULL) {
            DPRINT("Error: fopen returned NULL\n");
            return -1;
        }
```

```
}
else {
	DPRINT("Usage: theMerger <filename>\n");
	return -1;
}

int strsize = sizeof(char)*STRLEN;
int currsize = strsize*10;
int count1 = 0;
int count2 = 0;
int num1 = -1;
int num2 = -1;
int curr;

char** arr1 = (char**)malloc(currsize);
char** arr2 = (char**)malloc(currsize);
char str[STRLEN];

while(scanf("%s", str) != EOF) {
	if((count1*strsize) > currsize || (count2*strsize) > currsize) {
		currsize += currsize;
		arr1 = (char**)realloc(arr1, currsize);
		arr2 = (char**)realloc(arr2, currsize);
	}

	curr = get_num(str);

	if(num1 == -1) {
		num1 = curr;
		arr1[count1] = strdup(str);
		count1++;
	}
	else if(num2 == -1 && curr != num1) {
		num2 = curr;
		arr2[count2] = strdup(str);
```

```
			count2++;
		}
		else if(curr == num1) {
			arr1[count1] = strdup(str);
			count1++;
		}
		else if(curr == num2) {
			arr2[count2] = strdup(str);
			count2++;
		}
		else {
			DPRINT("Error: invalid case in while loop!\n");
			return -1;
		}
	}

	char** merged = merge(arr1, arr2, count1, count2);

	for(int i = 0; i < (count1 + count2); i++) fprintf(out, "%s\n", merged[i]);

	freearr(arr1, count1);
	freearr(arr2, count2);
	freearr(merged, count1 + count2);

	fclose(out);

	return 0;
}
```

theMerger.c first determines whether or not the arguments specified are valid, as well as whether or not a file was specified in the arguments or if we will be writing to stdout at the end. Then, it creates two string arrays of initial size 10 and then goes into a loop reading string by string from stdin. It uses the get_num() function to put each string into the proper array based on the preceding number it has, resizing the arrays if more space is needed. After the while loop, the two arrays are merged into a third array of double the size by calling the merge() function, and then each string in this merged (and sorted) array is printed to either stdout or the specified file if an argument was given for that. Finally, the three arrays are freed, the file is closed, and the program finishes.

myStarter.c (simple)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

#define PDEBUG 1
#define DPRINT(fmt, ...) do { if (PDEBUG) fprintf(stderr, "%s:%d:%s():
"fmt,\
                __FILE__, __LINE__, __func__, ##__VA_ARGS__); } while (0)

int* createpipe() {
        int* arr = (int*)malloc(sizeof(int)*2);
        if(pipe(arr) == -1) {
                DPRINT("Error: pipe returned -1\n");
                exit(-1);
        }
        return arr;
}

int createchild() {
        int pid;
        if((pid = fork()) == -1) {
                DPRINT("Error: fork returned -1\n");
                exit(-1);
        }
        return pid;
}
```

```
int main(int argc, char* argv[]) {
    char* child5arg;

    if(argc == 6) {
        child5arg = NULL;
    }
    else if(argc == 7) {
        child5arg = argv[6];
    }
    else {
        DPRINT("Usage: myStarter <child1/2 program> <child1
argument> <child2 argument> <child3/4 program> <child5 program>
<child5 argument>\n");
        return -1;
    }

    int* pipe1 = createpipe();
    int child1 = createchild();

    if(!child1) {
        close(pipe1[0]);

        if(dup2(pipe1[1], STDOUT_FILENO) == -1) {
            DPRINT("Error: dup2 returned -1 in child1\n");
            exit(-1);
        }

        if(execlp(argv[1], "fileReader1", argv[2], NULL) == -1) {
            DPRINT("Error: execlp returned -1 in child1\n");
            exit(-1);
        }

        exit(0);
    }
    int* pipe2 = createpipe();
    int child2 = createchild();

    if(!child2) {
        close(pipe1[0]);
        close(pipe1[1]);
        close(pipe2[0]);

        if(dup2(pipe2[1], STDOUT_FILENO) == -1) {
            DPRINT("Error: dup2 returned -1 in child2\n");
            exit(-1);
        }

        if(execlp(argv[1], "fileReader2", argv[3], NULL) == -1) {
            DPRINT("Error: execlp returned -1 in child2\n");
            exit(-1);
        }

        exit(0);
    }

    int* pipe3 = createpipe();
    int child3 = createchild();

    if(!child3) {
        close(pipe1[1]);
        //close(pipe2[0]);
        close(pipe2[1]);
        close(pipe3[0]);

        if(dup2(pipe1[0], STDIN_FILENO) == -1 || dup2(pipe3[1],
STDOUT_FILENO) == -1) {
            DPRINT("Error: dup2 returned -1 in child3\n");
            exit(-1);
        }

        if(execlp(argv[4], "aSorter1", "42", NULL) == -1) {
            DPRINT("Error: execlp returned -1 in child3\n");
            exit(-1);
        }
        exit(0);
    }

    int child4 = createchild();

    if(!child4) {
        //close(pipe1[0]);
        close(pipe1[1]);
        close(pipe2[1]);
        close(pipe3[0]);

        if(dup2(pipe2[0], STDIN_FILENO) == -1 || dup2(pipe3[1],
STDOUT_FILENO) == -1) {
            DPRINT("Error: dup2 returned -1 in child4\n");
            exit(-1);
        }

        if(execlp(argv[4], "aSorter2", "16", NULL) == -1) {
            DPRINT("Error: execlp returned -1 in child4\n");
            exit(-1);
        }

        exit(0);
    }

    int child5 = createchild();

    if(!child5) {
        close(pipe1[0]);
        close(pipe1[1]);
        close(pipe2[0]);
        close(pipe2[1]);
        close(pipe3[1]);

        if(dup2(pipe3[0], STDIN_FILENO) == -1) {
            DPRINT("Error: dup2 returned -1 in child5\n");
            exit(-1);
        }

        if(execlp(argv[5], "theMerger", child5arg, NULL) == -1) {
            DPRINT("Error: execlp returned -1 in child5\n");
            exit(-1);
        }

        exit(0);
    }

    waitpid(child1, NULL, 0);
    close(pipe1[1]);

    waitpid(child2, NULL, 0);
    close(pipe2[1]);

    waitpid(child3, NULL, 0);
    close(pipe1[0]);
    close(pipe3[1]);

    waitpid(child4, NULL, 0);
    waitpid(child5, NULL, 0);

    free(pipe1);
    free(pipe2);
    free(pipe3);
```

```
	return 0;
}
```

This is the unsophisticated myStarter.c, as each child is created by hand and there are no helper functions for execlp and dup2 (although there are helper functions for createpipe() and createchild()). It creates 5 children and 3 pipes as per the diagram for the simple pipe starter, and then it waits for all the children to finish before freeing the pipes and returning.

<u>myStarter2.c</u> (complex)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

#define PDEBUG 1
#define DPRINT(fmt, ...) do { if (PDEBUG) fprintf(stderr, "%s:%d:%s(): 
"fmt,\
		__FILE__, __LINE__, __func__, ##__VA_ARGS__); } while (0)

int* createpipe() {
	int* arr = (int*)malloc(sizeof(int)*2);
	if(pipe(arr) == -1) {
		DPRINT("Error: pipe returned -1\n");
		exit(-1);
	}
	return arr;
}

int createchild() {
	int pid;
	if((pid = fork()) == -1) {
		DPRINT("Error: fork returned -1\n");
		exit(-1);
```

```
    }
    return pid;
}

void createdupe(int oldfd, int newfd) {
    if(dup2(oldfd, newfd) == -1) {
        DPRINT("Error: dup2 returned -1\n");
        exit(-1);
    }
}

void runprog(char* path, char* name, char* arg) {
    if(execlp(path, name, arg, NULL) == -1) {
        DPRINT("Error: execlp returned -1\n");
        exit(-1);
    }
}

int main(int argc, char* argv[]) {
    //determine if user specified an output file
    char* output;
    if(argc == 9) output = argv[8];
    else if(argc == 8) output = NULL;
    else {
        DPRINT("Invalid arguments!\n");
        exit(-1);
    }

    //initialize pipe/children array
    int* pipes[7];
    int children[11];
    for(int i = 0; i < 7; i++) {
        pipes[i] = createpipe();
    }
```

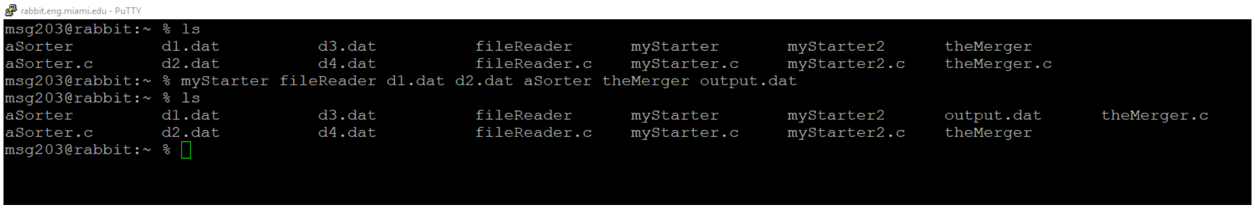
```
//4 fileReader children
for(int i = 0; i < 4; i++) {
        children[i] = createchild();
        if(!children[i]) {
                createdupe(pipes[i][1], STDOUT_FILENO);
                runprog(argv[1], "fileReader", argv[i+2]);
                exit(0);
        }
}
for(int i = 0; i < 4; i++) close(pipes[i][1]);

//4 aSorter children
for(int i = 4; i < 8; i++) {
        children[i] = createchild();
        if(!children[i]) {
                if(i < 6) {
                        createdupe(pipes[i-4][0], STDIN_FILENO);
                        createdupe(pipes[4][1], STDOUT_FILENO);
                        runprog(argv[6], "aSorter", "16");
                } else {
                        createdupe(pipes[i-4][0], STDIN_FILENO);
                        createdupe(pipes[5][1], STDOUT_FILENO);
                        runprog(argv[6], "aSorter", "42");
                }
                exit(0);
        }
}
for(int i = 4; i < 6; i++) close(pipes[i][1]);

//2 intermediate merger children
for(int i = 8; i < 10; i++) {
        children[i] = createchild();
        if(!children[i]) {
                createdupe(pipes[i-4][0], STDIN_FILENO);
                createdupe(pipes[6][1], STDOUT_FILENO);
```

```
            runprog(argv[7], "aMerger", NULL);
            exit(0);
        }
    }
    for(int i = 0; i < 4; i++) close(pipes[i][0]);
    close(pipes[6][1]);

    //final merge child
    children[10] = createchild();
    if(!children[10]) {
        createdupe(pipes[6][0], STDIN_FILENO);
        runprog(argv[7], "theMerger", output);
        exit(0);
    }

    //wait for all children to finish
    for(int i = 0; i < 11; i++) waitpid(children[i], NULL, 0);

    //free the memory allocated for pipes
    for(int i = 0; i < 7; i++) free(pipes[i]);

    return 0;
}
```

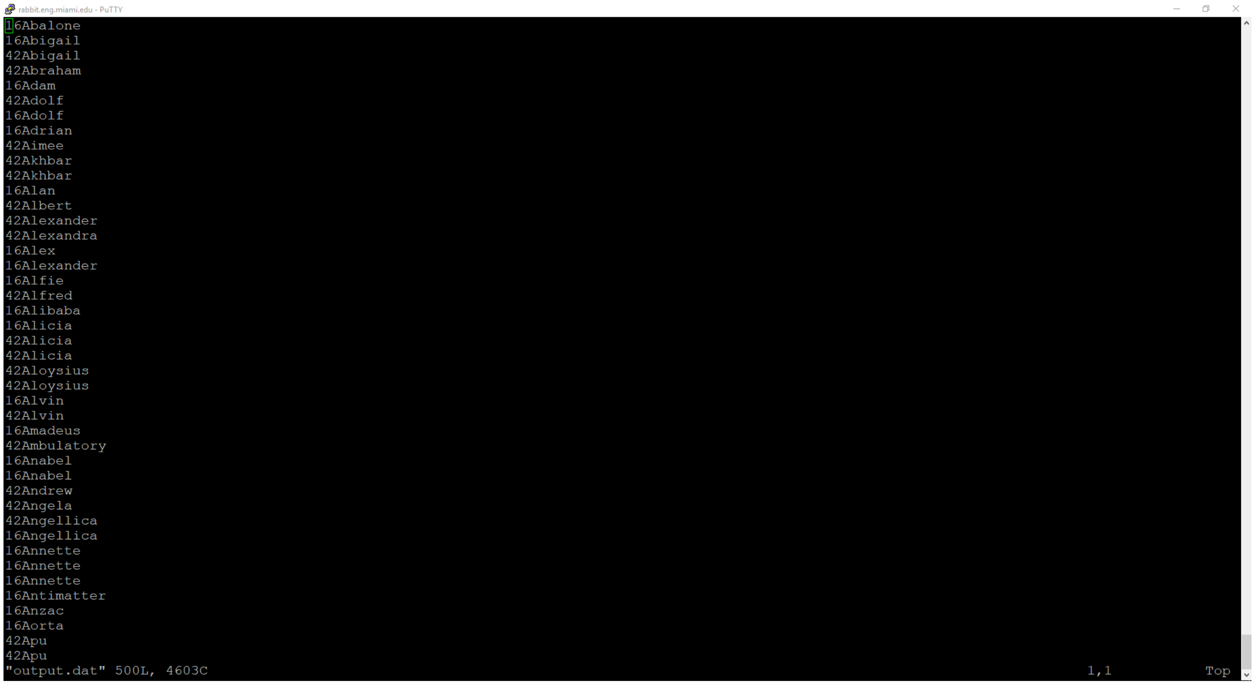
myStarter2.c is pretty much better than myStarter.c in every way. Despite having a more complex child/pipe arrangement with 11 children and 7 pipes, the source code is much shorter and much more readable. This is thanks to using for loops and child/pipe arrays to create the 4 readers, 4 sorters, and 3 mergers. Though the for loops are hard coded, it should be quite easy to change a few numbers and expand this child/pipe arrangement to an even more complicated arrangement. The createdupe() and runprog() functions also help to make the source code more compact and readable, as they abstract the error checking of dup2 and execlp so it doesn’t have to be explicitly written out each time. After all the reader, sorter, and merger children are made in their respective for loops, there are two final for loops, one for calling waitpid for each child and another for freeing our pipes afterwards. Once each child has been waited for, the program returns successfully.

Sample Runs

myStarter.c (simple)

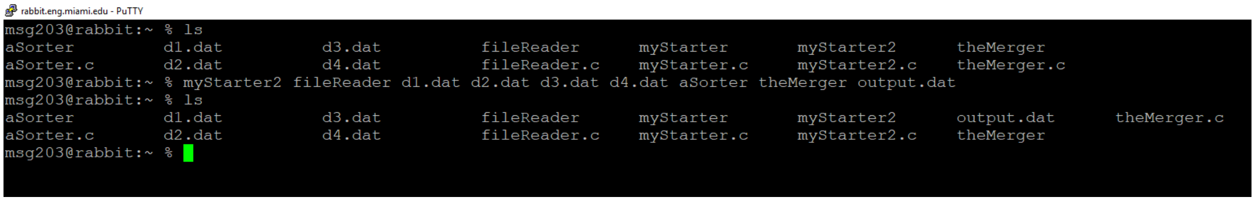


```
msg203@rabbit:~ % ls
aSorter         d1.dat          d3.dat          fileReader      myStarter       myStarter2      theMerger
aSorter.c       d2.dat          d4.dat          fileReader.c    myStarter.c     myStarter2.c    theMerger.c
msg203@rabbit:~ % myStarter fileReader d1.dat d2.dat aSorter theMerger output.dat
msg203@rabbit:~ % ls
aSorter         d1.dat          d3.dat          fileReader      myStarter       myStarter2      output.dat      theMerger.c
aSorter.c       d2.dat          d4.dat          fileReader.c    myStarter.c     myStarter2.c    theMerger
msg203@rabbit:~ %
```

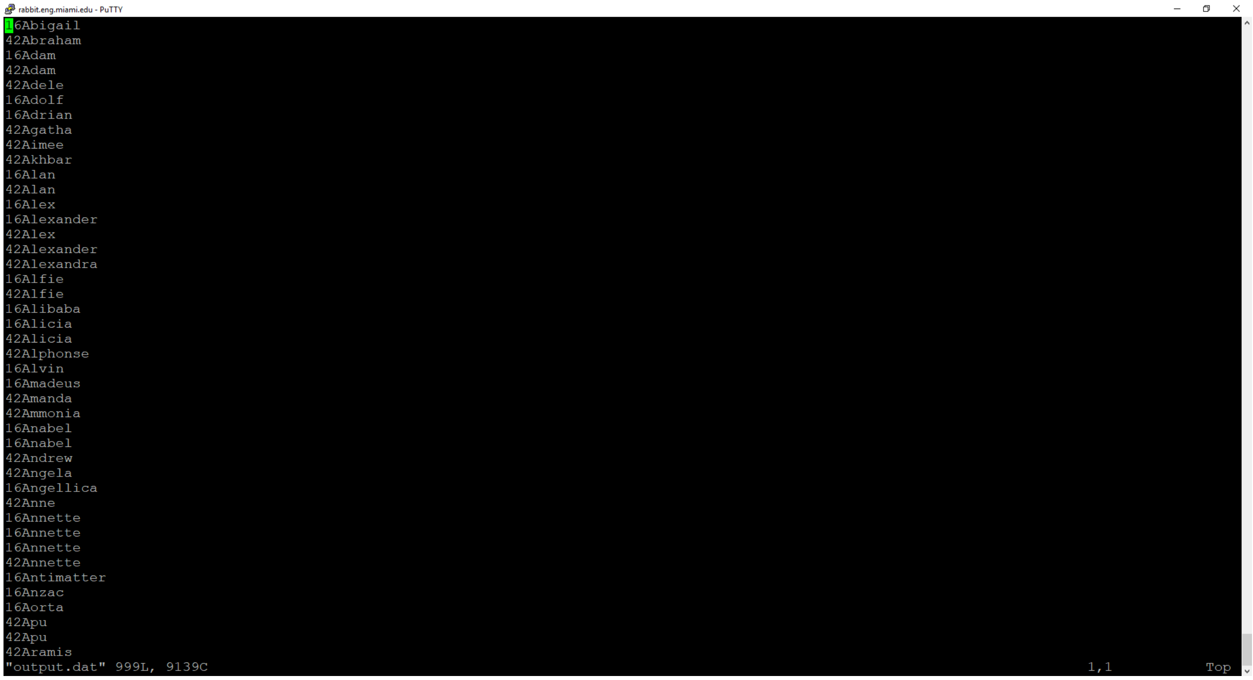


```
16Abalone
16Abigail
42Abigail
42Abraham
16Adam
42Adolf
16Adolf
16Adrian
42Aimee
42Akhbar
42Akhbar
16Alan
42Albert
42Alexander
42Alexandra
16Alex
16Alexander
16Alfie
42Alfred
16Alibaba
16Alicia
42Alicia
42Alicia
42Aloysius
42Aloysius
16Alvin
42Alvin
16Amadeus
42Ambulatory
16Anabel
16Anabel
42Andrew
42Angela
42Angellica
16Angellica
16Annette
16Annette
16Annette
16Antimatter
16Anzac
16Aorta
42Apu
42Apu
"output.dat" 500L, 4603C                                                                            1,1           Top
```

myStarter2.c (complex)



```
msg203@rabbit:~ % ls
aSorter         d1.dat          d3.dat          fileReader      myStarter       myStarter2      theMerger
aSorter.c       d2.dat          d4.dat          fileReader.c    myStarter.c     myStarter2.c    theMerger.c
msg203@rabbit:~ % myStarter2 fileReader d1.dat d2.dat d3.dat d4.dat aSorter theMerger output.dat
msg203@rabbit:~ % ls
aSorter         d1.dat          d3.dat          fileReader      myStarter       myStarter2      output.dat      theMerger.c
aSorter.c       d2.dat          d4.dat          fileReader.c    myStarter.c     myStarter2.c    theMerger
msg203@rabbit:~ %
```



```
16Abigail
42Abraham
16Adam
42Adam
42Adele
16Adolf
16Adrian
42Agatha
42Aimee
42Akhbar
16Alan
42Alan
16Alex
16Alexander
42Alex
42Alexander
42Alexandra
16Alfie
42Alfie
16Alibaba
16Alicia
42Alicia
42Alphonse
16Alvin
16Amadeus
42Amanda
42Ammonia
16Anabel
16Anabel
42Andrew
42Angela
16Angellica
42Anne
16Annette
16Annette
16Annette
42Annette
16Antimatter
16Anzac
16Aorta
42Apu
42Apu
42Aramis
"output.dat" 999L, 9139C                                                                                                                  1,1           Top
```