Operating Systems Tutorial 7
Jason Runzer 100520993
Albert Fung 100520898
Nicholas Gregorio 100514374
11/8/2015

Conceptual Questions

1. Signals are software interrupts that are delivered to a process. They can report errors, asynchronous events, or report other situations to an executing program. Signals can be used as a communication tool to synchronize processes, or for parent processes to terminate child processes.
2. The signal SIGINT is used to terminate a process when the user types the INTR character or can suspend a program when the user enters the SUSP character. The SIGTSTP signal is a signal that is used for job control to stop a process. This signal can be handled and ignored by the program. The SIGCONT is used to resume a program when it is called. A program can call the SIGINT with SUSP character to suspend the program. Then the program can continue if the SIGCONT to resume it, then finally it can use the SIGTSTP signal to terminate the process.
3. The kill() function allows the program to select which process to kill. The kill function can be specified to notify a process that it has finished execution as well. This is done by using kill(pid,SIGCHILD), which will notify the parent when the child has finished execution. The waitpid() function suspends the execution of the process that called it until the selected pid that refers to a process finishes its execution.
4. A linked list is a data structure where each element in the list has a pointer to the successor in the list and the data portion. Linked lists are used to implement queues, stacks, or any other data structure that grows dynamically during the time of execution of a program. This has an advantage over an array since an initial size does not need to be defined. A queue uses a linked list to satisfy the first in, first out (FIFO) condition. An item is added to the end of the list, and traversal through the items is from the first one that was added, to the last one that was added in the queue. The common operations that a linked list implementation for a queue must have is: pop() which removes the top element in the list, and push() which adds the element to the end of the list. The rest of the functions are needed if you wish to change how a queue works.
5. A linked list in C is constructed by having a node struct, and a data struct. The node contains an instance of the data struct, and also contains a pointer to the next node struct in the list. To add a value to the queue, you would iterate through the list until the node is null, then allocate memory for the node, add the data to the node, then set the next node to null. To remove a specific value from the list, you would iterate through the linked list until the data is found, then assign the previous node’s pointer to the node after the one that contains the specific data. Then you would free the node that you are removing to ensure that there are no memory leaks.

Application Questions

1.
CODE

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>

typedef struct {
	char name[256];
	int priority;
	int pid;
	int runtime;
} proc;

int MAX_BUFF_SIZE = 256;

typedef struct node {
	proc process;
	struct node *next;
} node_t;

node_t *queue;

void push(proc process) {
	node_t *current = queue; //set the current node to the first one in the queue

	//traverse through the linked list until it finds the next node = to NULL
	while (current->next != NULL) {
		current = current->next;
	}
	//allocate and and the process to the list
	current->next = (node_t *) malloc(sizeof(node_t));

	current->next->process = process;
	current->next->next = NULL; //set the next node to NULL to specify the end of the list
}
```

```
int main(void) {
	//initalize the first node with null
	queue = NULL;
	//the temp process when pushing to the list
	//proc tempProcess;

	//allocate memory for the
	queue = malloc(sizeof(node_t));
	queue->next = NULL;

	//open the file to get the processes
	FILE *fp = fopen("processes.txt", "r");

	if (!fp) {
		printf("Error opening file");
		exit(EXIT_FAILURE);
	}

	//the line for each process
	size_t len = 0;
	ssize_t read;
	char *line = NULL;

	//while ((read = getline(&line, &len, fp)) != -1) {

	//go though the text file and add each process to the list
	while ((read = getline(&line, &len, fp)) != -1) {
		//tokenize the line with , to get each entity of the process
		proc tempProcess;
		char * tokens = NULL;
		tokens = strtok(line, ",");
		strcpy(tempProcess.name, tokens);
		tokens = strtok(NULL, ",");
		tempProcess.priority = atoi(tokens); //atoi casts a char * to an integer
		tokens = strtok(NULL, ",");
		tempProcess.pid = atoi(tokens);
		tokens = strtok(NULL, ",");
		tempProcess.runtime = atoi(tokens);
		push(tempProcess); //push the process to the list

	}
	fclose(fp);
```

```
	//traverse through and display each item in the linked list.
	node_t * current = queue;
	//get rid of the sentinel node
	current = current->next;

	//traverse through the whole linked list
	while (current != NULL) {
		proc tempProcess = current->process;
		printf("Name: %s\nPriority: %d\npid: %d\nRuntime: %d\n\n",
				tempProcess.name, tempProcess.priority, tempProcess.pid,
				tempProcess.runtime);
		current = current->next; //advance the node
	}
	free(line);
	return 0;

}
```

OUTPUT

```
JasonR@ubuntu:~/git-projects/Tutorial7/Q1$ ./Q1
Name: systemd
Priority: 0
pid: 1
Runtime: 5

Name: bash
Priority: 0
pid: 1000
Runtime: 8

Name: vim
Priority: 1
pid: 11992
Runtime: 3

Name: emacs
Priority: 3
pid: 11993
Runtime: 1

Name: chrome
Priority: 1
pid: 11996
Runtime: 2

Name: chrome
Priority: 1
pid: 11997
Runtime: 3

Name: chrome
Priority: 1
pid: 11998
Runtime: 1

Name: gedit
Priority: 2
pid: 12235
Runtime: 4

Name: eclipse
Priority: 2
pid: 14442
Runtime: 2

Name: clang
Priority: 1
pid: 9223
Runtime: 3

JasonR@ubuntu:~/git-projects/Tutorial7/Q1$
```

2.
CODE:

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>


typedef struct{
	char name[256];
	int priority;
	int pid;
	int runtime;
} proc;

int MAX_BUFF_SIZE = 256;

typedef struct node{
	proc process;
	struct node *next;
} node_t;


proc tempProc;

node_t *queue;

//adds a process to the end of the list
void push (proc process){
	node_t *current = queue; //set the current node to the first one in the queue

	//traverse through the linked list until it finds the next node = to NULL
	while(current -> next != NULL){
		current = current -> next;
	}
	//allocate and and the process to the list
	current -> next =  (node_t *) malloc(sizeof(node_t));
	current -> next-> process = process;
```

```
	current-> next-> next = NULL;  //set the next node to NULL to specify the end of the list
}

//pops the first node in the list and returns a pointer to the process that was deleted
proc * pop(){
	node_t * next_node = NULL;
	if(queue -> next == NULL){
		printf("in NULL");

		return NULL;
	}

	next_node = queue -> next -> next; // get the next node after the sentinal
	node_t * currentNode = queue -> next;
	tempProc = currentNode-> process;

	//strcpy(tempProc.name,currentNode -> process.name);

	free(currentNode);
	queue -> next = next_node;

	return &tempProc;



}


//prints the process that is given
void printProc(proc *pro){

	if(pro != NULL){
		printf("Name: %s\nPriority: %d\npid: %d\nRuntime: %d\n\n",
				(pro -> name),pro ->priority,pro ->pid,pro ->runtime);
	}
}

/* Prints the list processes that is given.*/
void printList(node_t *list){
	node_t * current = list;
	//get rid of the sentinal
	current = current -> next;
```

```
        while(current != NULL){

                tempProc = current ->process;
                printProc(&tempProc); //print the current process
                current = current -> next; //advance the node
        }

}


//deletes the node with the name specified, If not there, returns null
proc * delete_name(char *name){
        node_t *current = queue -> next;
        node_t *previous = queue;
        while(current != NULL){

                //check the current process' name
                tempProc = current -> process;
                if(strcmp(tempProc.name,name) == 0){
                        node_t *deletedNode = current;
                        previous -> next = current -> next;
                        free(deletedNode); //free up the memory of the node
                        return &tempProc;
                }
                //get the next node in the list
                previous = current;
                current = current -> next;
        }
        printf("process not found\n");
        return NULL;
}

//deletes the node with the pid specified, If not there, returns null
proc * delete_pid(int pid){
        node_t *current = queue -> next;
        node_t *previous = queue;
        while(current != NULL){
                //check the current process' pid
                tempProc = current -> process;
                if(tempProc.pid == pid){
                        node_t *deletedNode = current;
                        previous -> next = current -> next;
                        free(deletedNode); //free up the deleted nodes memory
```

```
                        return &tempProc; //return the process that was deleted
                }
                //get the next item in the list
                previous = current;
                current = current -> next;
        }
        printf("process not found\n");
        return NULL;
}

int main(void)
{
        //initalize the first node with null
        queue = NULL;


        //allocate memory for the
        queue = malloc(sizeof(node_t));
        queue->next = NULL;


        //open the file to get the processes
        FILE *fp = fopen("processes.txt","r");

        if(!fp){
                printf("Error opening file");
        }

        //the line for each process

        size_t len = 0;
        ssize_t read;
        char *line = NULL;

        //go though the text file and add each process to the list
        while ((read = getline(&line, &len, fp)) != -1) {
                //tokenize the line with , to get each entity of the process
                char * tokens = NULL;
                tokens = strtok(line,",\n");
                strcpy(tempProc.name,tokens);
                tokens = strtok(NULL,",\n");
                tempProc.priority = atoi(tokens); //atoi casts a char * to an integer
                tokens = strtok(NULL,",\n");
```

```
            tempProc.pid = atoi(tokens);
            tokens = strtok(NULL,",\n");
            tempProc.runtime = atoi(tokens);
            push(tempProc); //push the process to the list
        }
        fclose(fp);



        //traverse through and display each item in the linked list.
        printList(queue);

        printf("\n\n\n");

        //delete the processes
        delete_name("emacs");
        delete_pid(12235);

        printf("POP\n");
        while(queue -> next != NULL){
            printProc(pop());

        }


        free(line);
        return 0;

}
```

OUTPUT

```
JasonR@ubuntu:~/git-projects/Tutorial7/Q2$ ./Q2
Name: systemd
Priority: 0
pid: 1
Runtime: 5

Name: bash
Priority: 0
pid: 1000
Runtime: 8

Name: vim
Priority: 1
pid: 11992
Runtime: 3

Name: emacs
Priority: 3
pid: 11993
Runtime: 1

Name: chrome
Priority: 1
pid: 11996
Runtime: 2

Name: chrome
Priority: 1
pid: 11997
Runtime: 3

Name: chrome
Priority: 1
pid: 11998
Runtime: 1

Name: gedit
Priority: 2
pid: 12235
Runtime: 4

Name: eclipse
Priority: 2
pid: 14442
Runtime: 2

Name: clang
Priority: 1
pid: 9223
Runtime: 3
```

```
POP
Name: systemd
Priority: 0
pid: 1
Runtime: 5

Name: bash
Priority: 0
pid: 1000
Runtime: 8

Name: vim
Priority: 1
pid: 11992
Runtime: 3

Name: chrome
Priority: 1
pid: 11996
Runtime: 2

Name: chrome
Priority: 1
pid: 11997
Runtime: 3

Name: chrome
Priority: 1
pid: 11998
Runtime: 1

Name: eclipse
Priority: 2
pid: 14442
Runtime: 2

Name: clang
Priority: 1
pid: 9223
Runtime: 3

JasonR@ubuntu:~/git-projects/Tutorial7/Q2$
```

3.

Code:

```
/*
 * Q3.c
 *
 *  Created on: Nov 4, 2015
 *      Author: uoitstudent
 */

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

```
#include <stdbool.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#define SHELL "/bin/sh"

int main(int argc, char *argv[]) {
	int status;
	pid_t pid;
	const char *command ="./process";
	pid = fork();


	if (pid < 0) {
		//Failed
	} else if (pid == 0) {
		//Child Process
		execv(command, argv);
		exit(0);
		_exit(EXIT_FAILURE);


	} else {
		//Parent Process
		sleep(5);
		kill(pid, SIGINT);
		if(wait(&status)){
			printf("Child terminated");
		}
	}
}
```

Output:

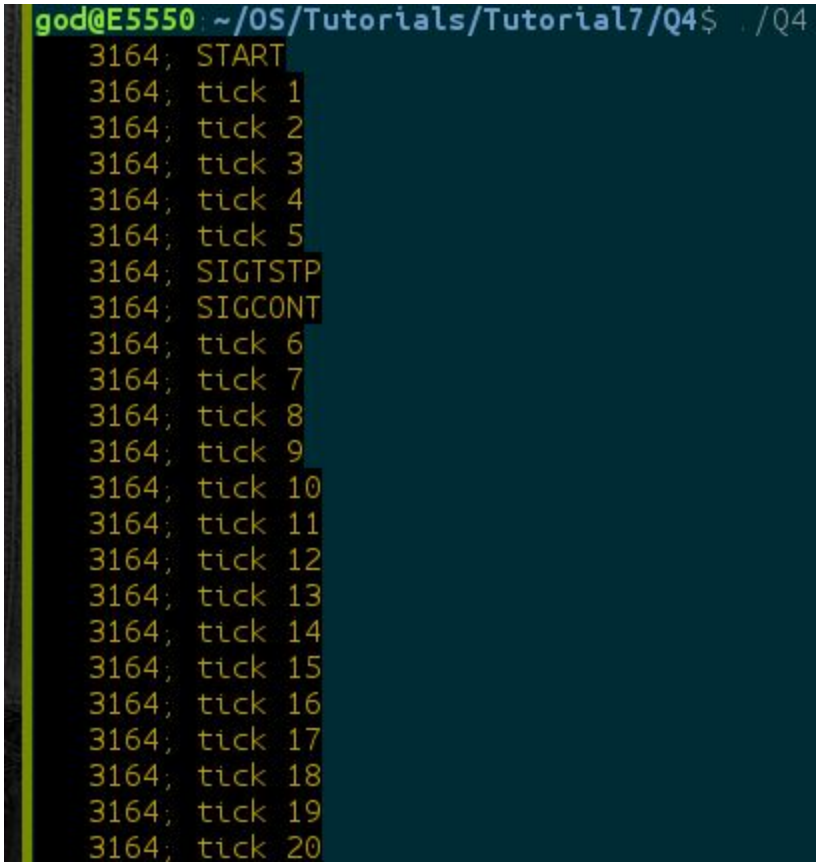
4.

Code:

```
//includes
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>

#define SHELL "/bin/sh"

int main(int argc, char *argv[]) {
	//command to execute
	const char *command = "./process";
	int status;
	pid_t pid;

	//fork the child
	pid = fork();
	if (pid == 0) { //CHILD
		//execute command
		execv(command, argv);
		exit(0);
		_exit(EXIT_FAILURE);
	} else if (pid < 0) {
		//FORK FAILED
		status = -1;
	} else { //PARENT
		//sleep for 5 seconds
		sleep(5);
```

```
                //send SIGTSTP to suspend the proc
                kill(pid, SIGTSTP);
                //sleep for 10 seconds
                sleep(10);
                //send SIGCONT to resume the proc
                kill(pid, SIGCONT);
                //wait for child to terminate
                waitpid(pid, &status, 0);
                //if child terminates normally
                if (status == 0) {
                        exit(0);
                }
                return status;
        }
}
```

Output:



5.

Code:

```
#include <stddef.h>
```

```
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>

#define SHELL "/bin/sh"

typedef struct {
	char name[256];
	int priority;
	int pid;
	int runtime;
} proc;

int MAX_BUFF_SIZE = 256;

typedef struct node {
	proc process;
	struct node *next;
} node_t;

node_t *queue;
proc tempProc;

//the methods below are taken from Q1 and Q2

void push(proc process) {
	node_t *current = queue; //set the current node to the first one in the queue
	//traverse through the linked list until it finds the next node = to NULL
	while (current->next != NULL) {
		current = current->next;
	}
	//allocate and and the process to the list
	current->next = (node_t *) malloc(sizeof(node_t));
	current->next->process = process;
	current->next->next = NULL; //set the next node to NULL to specify the end of the list
}

proc * delete_name(char *name) {
```

```
	node_t *current = queue->next;
	node_t *previous = queue;
	while (current != NULL) {

		tempProc = current->process;
		if (strcmp(tempProc.name, name) == 0) {
			node_t *deletedNode = current;
			previous->next = current->next;
			free(deletedNode);
			return &tempProc;
		}
		previous = current;
		current = current->next;
	}
	printf("process not found\n");
	return NULL;
}

//pops the first node in the list and returns a pointer to the process that was deleted
proc * pop() {
	node_t * next_node = NULL;
	if (queue->next == NULL) {
		printf("in NULL");

		return NULL;
	}

	next_node = queue->next->next; // get the next node after the sentinal
	node_t * currentNode = queue->next;
	tempProc = currentNode->process;

	//strcpy(tempProc.name,currentNode -> process.name);

	free(currentNode);
	queue->next = next_node;

	return &tempProc;
}

//prints the process that is given
void printProc(proc *pro) {
	if (pro != NULL) {
		printf("Name: %s\nPriority: %d\npid: %d\nRuntime: %d\n\n", (pro->name),
```

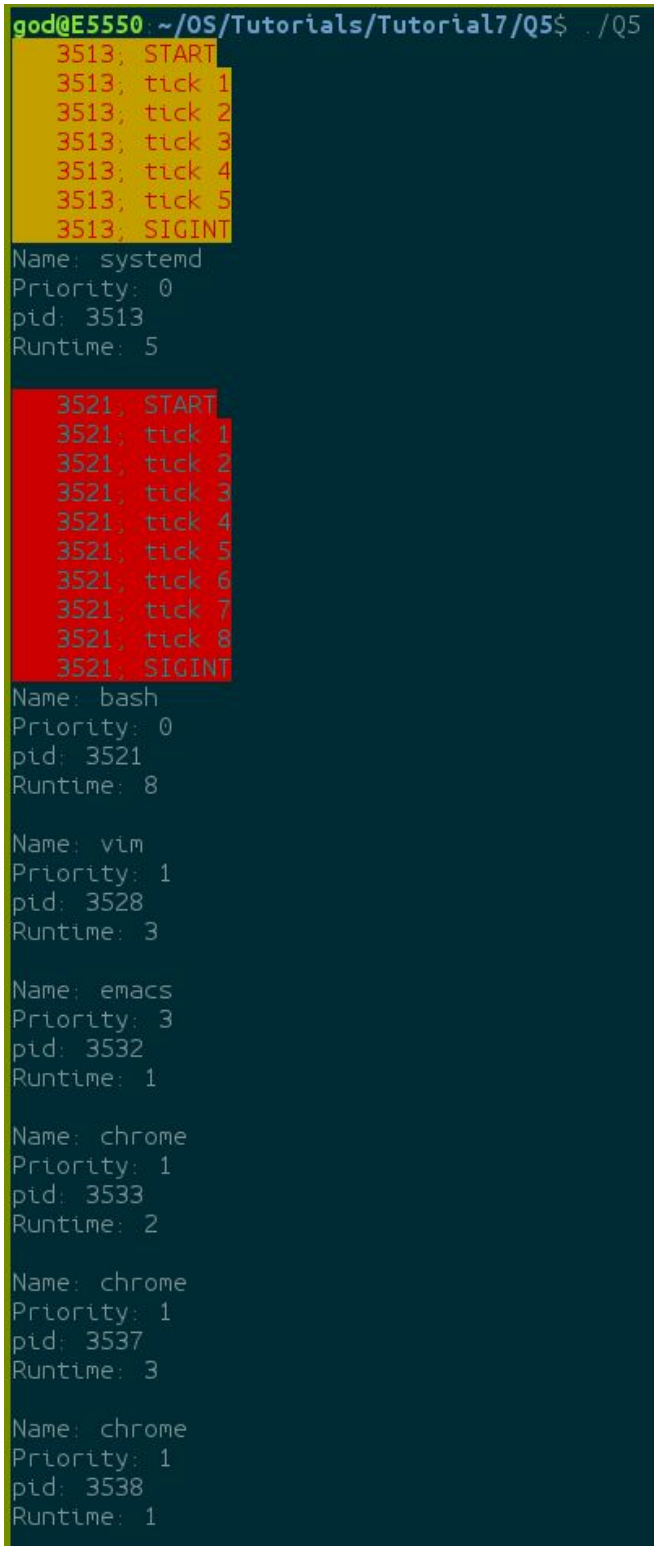
```
                                pro->priority, pro->pid, pro->runtime);
        }
}

//prints the contents of the list given
void printList(node_t *list) {
        node_t * current = list;
        //get rid of the sentinel
        current = current->next;
        while (current != NULL) {
                tempProc = current->process;
                printProc(&tempProc); //print the current process
                current = current->next; //advance the node
        }
}

int main(int argc, char *argv[]) {
        //initalize the first node with null
        queue = NULL;
        //allocate memory for the linkedlist
        queue = malloc(sizeof(node_t));
        queue->next = NULL;
        //open the file to get the processes

        proc tempProcess;

        ////////////////FILE STRUCTURE/////////////////
        //NAME, PRIORITY, PID(0), RUNTIME IN SECONDS
        ////////////////////////////////////////////

        FILE *fp = fopen("processes_q5.txt", "r");

        if (!fp) {
                printf("Error opening file");
                exit(EXIT_FAILURE);
        }

        //the line for each process
        size_t len = 0;
        ssize_t read;
        char *line = NULL;

        //go though the text file and add each process to the list
```

```
while ((read = getline(&line, &len, fp)) != -1) {
        //tokenize the line with , to get each entity of the process
        char * tokens = NULL;
        tokens = strtok(line, ",\n");
        strcpy(tempProcess.name, tokens);
        tokens = strtok(NULL, ", \n");
        tempProcess.priority = atoi(tokens); //atoi casts a char * to an integer
        tokens = strtok(NULL, ", \n");
        tempProcess.pid = 0;
        tempProcess.runtime = atoi(tokens);
        push(tempProcess); //push the process to the list
}
//close the file
fclose(fp);
node_t * current = queue;
//get rid of the sentinel node
current = current->next;
while (current != NULL) {
        tempProcess = current->process; //advance the node
        if (tempProcess.priority == 0) { //priority zero queue
                //command to execute
                const char *command = tempProcess.name;
                int status;
                pid_t pid;
                //fork the child
                pid = fork();
                if (pid == 0) { //CHILD
                        //execute command
                        execv(command, argv);
                        //remove from process struc
                        delete_name(tempProcess.name);
                        //execute the process binary
                        execv("./process", argv);
                        exit(0);
                } else if (pid < 0) { //FORK FAILED
                        status = -1;
                } else { //PARENT
                        //sleep for the runtime after exec, then kill the proc with SIGINT
                        sleep(tempProcess.runtime);
                        //send SIGINT to suspend the proc
                        kill(pid, SIGINT);
                        //wait for child to terminate
                        waitpid(pid, &status, 0);
```

```
                        if (status == 0) { //if child terminates normally
                                //set pid returned from exec()
                                tempProcess.pid = pid;
                                printProc(&tempProcess);
                                //iterate to next item in the linkedlist
                                current = current->next;
                        }
                }
        } else { //rest of the queue
                //command to execute
                const char *command = tempProcess.name;
                int status;
                pid_t pid;
                //fork the child
                pid = fork();
                if (pid == 0) { //CHILD
                        //execute command
                        execv(command, argv);
                        exit(0);
                } else if (pid < 0) { //FORK FAILED
                        status = -1;
                } else { //PARENT
                        //sleep for the runtime after exec, then kill the proc with SIGINT
                        sleep(tempProcess.runtime);
                        //send SIGINT to suspend the proc
                        kill(pid, SIGINT);
                        //wait for child to terminate
                        waitpid(pid, &status, 0);
                        if (status == 0) { //if child terminates normally
                                //set pid returned from exec()
                                tempProcess.pid = pid;
                                //prints name, priority, pid and runtime of process
                                printProc(&tempProcess);
                                //iterate to next item in the linkedlist
                                current = current->next;
                                pop(); //pops the remaining process off the queue
                        }
                }
        }
}
/*
printf("Printing out rest of queue: \n");
printList(queue);
```

```
        */
        return 0;
}
```

Output:



```
Name: gedit
Priority: 2
pid: 3542
Runtime: 4

Name: eclipse
Priority: 2
pid: 3547
Runtime: 2

Name: clang
Priority: 1
pid: 3550
Runtime: 3
```