PROJECT 1

Distributed median finding algorithm
with multiple processes

Name: Agowun Muhammad Altaf

Contents

Introduction

This report covers the steps and pseudo code used for the implementation of the distributed median finding algorithm using multi processes (5 processes K = 5).

Median is the middle/ average of the two middle numbers in an ordered list but since median is not affected by the values above or below the median value/s, an algorithm that aims to find the median does not need to order all the values. It is concerned with only the number/s that is halfway from the start and end.

The algorithm can therefore only count the quantity of numbers above a specific number to determine whether it is the median/part of it. In this implementation we will consider the quantity of numbers above and equal to the median, we denote this number as k.

Thus, if we can get the quantity of numbers above and equal to a randomly selected number, denoted by m, we can tell whether it is the median or not by comparing it with k.

- If k = m, then it is the median
- If m > k then that value is below the median, we can then remove values than are below and equal to that randomly selected number to reduce the quantity of numbers to consider later (equal also since it is not the median).
- If m < k then that value is above the median, we can then remove values than are above and equal to that randomly selected number to reduce the quantity of numbers to consider later (equal also since it is not the median).
  Since we are using the quantity of numbers above and equal to the median, we also need to update our value of k, k = k – m.

Example: first consider <u>odd</u> quantity of numbers:

1	2	3	4	5	6	7

k = (7 / 2) + 1 = 4, 4th number from the end should be the median value (4).

1	2	3	4	5	6	7

4 values

Hence if 4 is randomly selected then,

There would be 4 numbers (4,5,6 and 7) greater and equal to 4, hence m = 4, since k is also 4 then k(4) = m(4), thus median is found.

In case 2 is randomly selected then,

There would be 6 numbers (2,3,4,5,6 and 7) greater and equal to 2, hence m(6) not equal to k(4). Meaning it is not the median.

Following this step, we can drop values below and equal to 2 to reduce work later.

1	2	3	4	5	**6**	**7**

6 values

4 values

In case number 6 is randomly selected,

Then there would be 2 numbers (6 and 7) greater and equal to 6, hence m(2) not equal to k(4). Meaning it is not the median.

1	2	3	4	5	6	7

$$1 \quad 2 \quad 3 \quad 4 \quad 5 \quad \underbrace{6 \quad 7}_{\text{2 values}}$$

Following this step, we can drop values greater and equal to 6, to reduce work later.

1	2	3	4	5	6	7

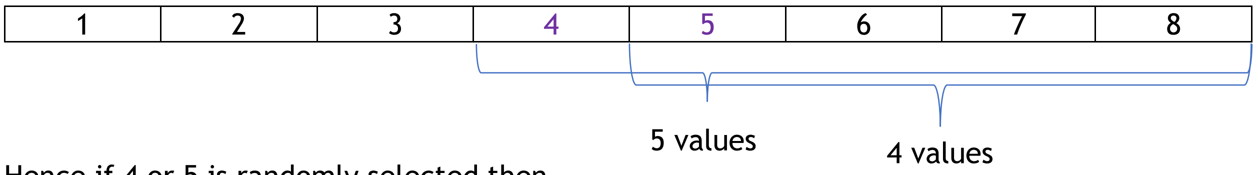
$$1 \quad 2 \quad 3 \quad \underbrace{4 \quad 5}_{\text{2 values}} \quad 6 \quad 7$$

But we also need to update k since the quantity of numbers greater than the median will be lower. k = k – m, k = 4 – 2 = 2.

now consider even quantity of numbers:

1	2	3	4	5	6	7	8

k = (8 / 2) + 1 = 5, 5th and 4th number from the end should be the median value ((4 + 5)/ 2).



Hence if 4 or 5 is randomly selected then,

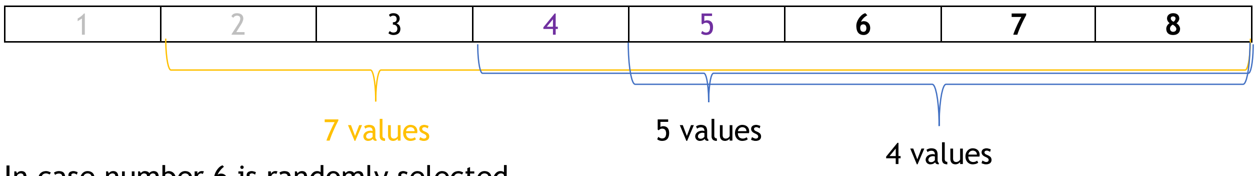
If 5 is selected: There would be 4 numbers (5,6,7 and 8) greater and equal to 5, hence m = 4, since one k is also 4 then k(4) = m(4), thus one component of the median is found. Moreover, we can clear values greater than and equal to it to reduce work. Set k = 1 also since all values above are cleared and quantity of numbers greater and equal to the median is only one.

If 4 is selected: There would be 5 numbers (4,5,6,7 and 8) greater and equal to 4, hence m = 5, since one k is also 5 then k(4) = m(4), thus one component of the median is found. Moreover, we can clear values less than and equal to it to reduce work.

In case 2 is randomly selected then,

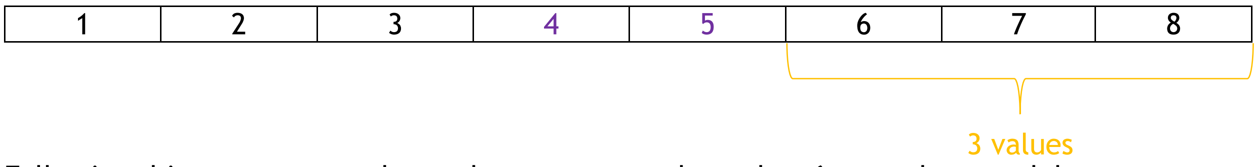
There would be 7 numbers (2,3,4,5,6,7 and 8) greater and equal to 2, hence m(7) not equal to k(4 or 5). Meaning it is not the median.

Following this step, we can drop values below and equal to 2 to reduce work later.

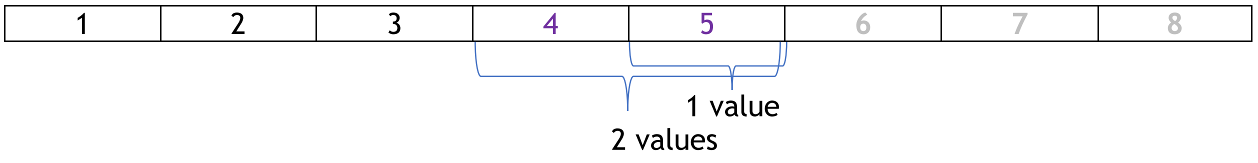


In case number 6 is randomly selected,

Then there would be 3 numbers (6,7 and 8) greater and equal to 6, hence m(2) not equal to k(4 or 5). Meaning it is not the median.



Following this step, we can drop values greater and equal to 6, to reduce work later.



But we also need to update k since the quantity of numbers greater than the median will be lower. k = k - m, k = 4 - 3 = 1 and k = 5 - 2 = 2.

Set up

Main (program entry point)

From the main (entry point) we have to create the pipes and processes

- Create 10 pipes (using a for loop we can create K*2 pipes (10 pipes))
- Using a for loop we can create the 5 processes, use if statement to
    - (pid > 0) ensure only the parent process is creating child processes.
    - (pid == 0) run the code for the child processes in child process only
        - Note the child receives an ID to tell from which pipe it has to write/read from
    - (pid < 0) handle error creating process

Fork after creating the pipes so that both the parent and children have a copy of the file descriptor to the pipes.

main (entry point) pseudo code:

```
function main(int argc, char const *argc[]) do
   initialize int pidArr[NUM_CHILD] // use to close processes one by one when exiting
   initialize pid = 1 // 1 in order to run the parent code for the first time

   // create the pipes
   for (i in NUM_CHILD) do
      pipe(parent_fd[i])
      pipe(child_fd[i])
   end
   for (i in NUM_CHILD) do
      if (pid > 0) do
         pid - fork()
         pidArr[i] = pid
      end
      // catch error during creating child process
      if (pid < 0) do
         error "could not create child process"
         closeChildren(pidArr) // close child that were created
         exit(1) // exit with error
      end
      // launch child process
      if (pid == 0) do
         childProcess(i) // i to let the child know which pipe to use / child ID
         break // exit the for loop since all children code is in the function call
      end
   end
   // run parent process
   if (pid > 0) do // ensure we are in the parent process
      parentProcess(pidArr);
   end
end
```

Parent process

Wait for child processes to be ready

The parent waits for the child to be ready to get the quantity of numbers in total (n) and to ensure median finding algorithm can start. The “READY” code is an integer corresponding to the quantity of numbers the child contains.

Calculate the initial value of k

Check if there are even or odd quantity of numbers. If even, then set the even flag

Then calculate the value of k. k = (n / 2) + 1, + 1 since the median will be included in the comparison between k and m

Parent process pseudo code related to set up

```
function parentProcess(int pidArr) do
    initialize k = 0 // values will be added to it so set to 0
    initialize evenFlag = 0 // set flag for even quantity of number to 0 (false)

    for (i in NUM_CHILD) do
        initialize childArrSize
        if (read READY response from child i into childArrSize == -1) do
            error "could not read READY response"
            closeChildren(pidArr)
            exit(2)
        end

        if (childArrSize < 0) do
            error "child responded with wrong quantity of numbers"
            closeChildren(pidArr)
            exit(3)
        end
        increment k by childArrSize
    end

    // check if even quantity of numbers
    if (k % 2 == 0) do
        set evenFlag to 1
    end

    k = (k / 2) + 1 // set k to point to median position
```

Child process

Close unused file descriptor to pipes (childPipeSetUp function below Set up Appendix)

Register kill signal (handler function below Set up Appendix)

Read file (using the function readFile below pseudo code)

Communicate the quantity of numbers it read from its file

Child process pseudo code related to set up

```
function childProcess(int id) do

    childPipeSetUp(id) // close all unused pipe by process

    register kill signal handle through handler function

    initialize numIndex = readFile(id) // return the quantity of numbers in it

    // display numbers read from file to user
    print "content of child = "
    for (i in numIndex-1) do
        print numPtr[i] + " "
    end
    print numPtr[numIndex]

    print "child sends ready to parent"
    if (send ready signal with quantity of numbers < 0) do
        error "error sending ready code/ quantity of numbers"
        free(numPtr)
        exit(1);
    end
```

ReadFile function

```
function readFile(int id) do
    intialize numIndex = 0
    intialize buffer
    intialize file pointer, fp
    intialize filename

    // create filename
    snprintf(filename, sizeof(filename), "input_%d.txt", processID)

    // read file
    if ((open file for read) == NULL) do
        error "file could not be opened"
        if (send FILE_ERR code to parent) do
            error "could not send file error code to parent"
            exit(1)
        end
        exit(4);
    end

    // create dynamic array
    if (numPtr = malloc(sizeof(int)) == NULL) do
        error "could not create pointer"
        exit(5);
    end

    while (fscanf(fp, "%d", %buffer) == 1) do
        if ((numPtr = realloc(numPtr, INT_SIZE * (numIndex + 1))) == NULL) do
            error "could not create increase dynamic array"
            free(numPtr) // free whatever was already allocated
            exit(5);
        end
        store buffer value in newly created numPtr space numPtr[numIndex]
        increment numIndex
    end

    if (fclose(fp) == EOF) do
        error "could not close file pointer"
    end

    return numIndex // to let calling function know the size of the array

end
```

Set up Appendix

Handle kill signal

```
function handler(int num) do
    close(child_fd[processID][1]);
    close(parent_fd[processID][0]);
    free(numPtr);
    exit(0);
end
```

childPipeSetUp function

```
function childPipeSetUp(int id) do
    for (i in NUM_CHILD) do
        if (i != id) do
            close(parent_fd[i][0])
            close(parent_fd[i][1])
            close(child_fd[i][0])
            close(child_fd[i][1])
        end else do
            close(child_fd[i][0]);
            close(parent_fd[i][1]);
        end
    end
end
```

Distributed median finding algorithm

Overall

Parent: sends request to a random child to send a random number it holds to become the pivot.

Randomly selected child: randomly select a number that it holds and sends it to the parent (runs the request function)

Parent: read the pivot being sent by the randomly selected child

Parent checks if child sent a correct pivot, if not then flag it and ensure that there are still numbers left overall to continue

Parent: broadcast PIVOT signal followed by the pivot value received from randomly selected child to all children.

Children: then count and send quantity of number greater than the pivot, to parent. (runs the pivot function)

Parent: sum each child quantity of numbers greater than the pivot in m

Parent: compare m and k and sends signal accordingly

- If m = k, then
  - If odd quantity of numbers then
    - median is found display it and ends
  - if even quantity of numbers then
    - if already have a value for median stored then
      - display average of the median found with that stored and ends
    - else store the value
      - remove element greater than that number if higher median value, remove lower if lower median value
- If m > k, the pivot is a value below the median, then
  - Sends SMALL code to remove element that are below the pivot to each children
- If m < k, the pivot is a value above the median, then
  - Sends LARGE code to remove element that are above the pivot to each children
  - Update k to account for removal of numbers above the median (k = k - m)

Children: receive SMALL/ LARGE code, and does the corresponding action

- SMALL: remove numbers it contains that are less than the pivot(runs the small function)
- LARGE: remove numbers it contains that are greater than the pivot(runs the large function)

Parent process code

Parent process pseudo code related to median finding algorithm

```
function parentProcess(int pidArr) do
    // initialize array to flag that the processs' array is empty
    initialize emptyProcesses[NUM_CHILD] = {0}
    srand(time(NULL)) // use to set the seed for the random number generator
    initialize medianBuf = 0 // use to store the value median average calculation

    // infinite loop
    while (1) do
        initialize m = 0 // store the quantity of numbers greater median
        initialize randChild // store the ID of the randomly selected child

        while (emptyProcesses[(randChild = rand() % K)] == 1) // see explanation below

        int code = REQUEST // store the REQUEST code to be sent in pipe
        print "parent send request to child i"
        if (write code to randChild) do
            error "error sending request to child i"
            closeChildren(pidArr)
            exit(1)
        end

        if (read response (pivot)) do
            error "error reading the pivot"
            closeChildren(pidArr)
            exit(2)
        end
        if (pivot >= 0) do
            print "pivot = ${pivot}"
        end else do
            // child has no pivot
            print "child i has no pivot"
            set emptyProcesses[randChild] flag to 1
            initialize numEmptyChild to 0 // count number of empty child process
            for (i in NUM_CHILD) do
                increment numEmptyChild if emptyProcesses[i] is set
            end
            if (numEmptyChild == K) do
                print "all children empty ending program"
                closeChildren(pidArr)
                exit(4)
            end

            continue // skip this loop
        end
```

```
int code = PIVOT // store the PIVOT code to be sent in pipe
for (i in NUM_CHILD) do
    if (send code to child i < 0) do
        error "could not send pivot"
        closeChildren(pidArr)
        exit(1)
    end

    if(send pivot value < 0) do
        error "error sending pivot"
        closeChildren(pidArr)
        exit(1);
    end
end

print "m ="
// display the value and calualtion for m
for (i in NUM_CHILD) do
    int buf // store the value read from child response
    if (read response from child into buf == -1) do
        error "could not read amount of nubmers"
        closeChildren(pidArr)
        exit(2);
    end
    increment m with value of buf
    print buf followed by either 6 of = if last one
end
print value of m

// check if we have a/the median
if ( k == m || (evenFlag && !medianBuf && (k - 1) == m) ) do
    // check if even
    if (evenFlag) do
        // check if we already have a median value
        if (medianBuf == 0) do
            medianBuf = pivot // save median in medianBuf

            // check if value was found through k or k-1
            if (k == m) do
                decrement k by 1 // update default k value since k for even will not be used
                code = SMALL
            end else do
                code = LARGE
                k = 1
            end
            print "first median component of median is" + medianBuf
```

```
                for (i in NUM_CHILD) do
                    if (send code in code variable to child i < 0) do
                        error "could not send ${SMALL/LARGE} code to child "
                        closeChildren(pidArr)
                        exit(1)
                    end
                 end

                continue // since we still need the other component
            end

            print "second component received " + pivot
            print "median = " + (pivot + medianBuf) / 2
        end else do
            // not even quantity of numbers thus median found
            print "median = " + pivot
        end

        print "parent sending kill signal to children"
        closeChildren(pidArr)
        exit(0) // end successfully

    end else do
        // not a/the median
        // store the code to be sent
        int drop = m > k ? SMALL : LARGE
        if (drop == SMALL) do
            print m +" > " + k + "sending SMALL code"
        end else do
            print m +" < " + k + "sending LARGE code"

            print "k - m, since value larger than k being dropped"
            k -= m
            print k
        end

        for (i in NUM_CHILD) do
            if (send code in drop variable to child i) do
                error "could not send ${SMALL/LARGE} code to child "
                closeChildren(pidArr)
                exit(1)
            end
        end

    end
  end
end
```

(evenFlag && !medianBuf && (k - 1) == m)

Ensures that the following code only runs if evenFlag is set and that medianBuf is empty

- There is an even quantity of numbers and
- Since even quantity of numbers, we need 2 values for median. MedianBuf stores the 1st value hence we need to test if it has not been stored yet, otherwise have the second k value (k-1)

Avoid random child that are empty

```
while (emptyProcesses[(randChild = rand() % K)] == 1)
```

loops while the randomly selected child's array is empty
set the value for the randChild variable

rand() % K rand return value between 0 and RAND_MAX
using modulus the value is force to be between 0 and K inclusive

CloseChildren function

```
// call by parent when exiting program
function closeChildren() do
    for (i in NUM_CHILD) do
        close child_fd[i] (read file descriptor for pipe)
        close parent_fd[i] (write file descriptor for pipe)

        send kill signal to child_fd[i]
        wait child termination
        print child i terminated
    end
end
```

Child process code

Child pseudo code related to median finding algorithm

```
function childProcess(int id) do
   initialize pivotNum to store the pivot received

   // infinite loop
   while (1) do
      initialize code // stoer the code received from parent

      if (read signal from parent == -1) do
         error "error reading code"

         free(numPtr)
         exit(2)
      end

      switch(code) do
         case REQUEST:
            request(id, numPtr, numIndex);
            break;
         case PIVOT:
            pivotNum = pivot(id, numPtr, numIndex);
            break;
         case SMALL:
            numIndex = small(pivotNum, numPtr, numIndex);
            break;
         case LARGE:
            numIndex = large(pivotNum, numPtr, numIndex);
            break;
         default:
            // no code was received
            break;
      end
   end

end
```

Request function

```
function request(int id, int *childArr, int arraySize) do
    initialize ret = EMPTY_ARR

    if (arraySize != 0) do
        int randomIndex = rand() % arraySize // get a random index
        ret = childArr[randomIndex]
    end

    if (send ret to parent < 0) do
        error "error sending pivot"
        free(childArr)
        exit(1)
    end

end
```

Pivot function

```
function pivot(int id, int *childArr, int arraySize) do
    initialize pivotNum
    int ret = 0

    if (read pivot from parent) do
        error "could not read pivot"
        free(childArr)
        exit(2)
    end

    if (arraySize != 0) do
        for (i in arraySize) do
            if (childArr[i] >= pivotNum) do
                increment ret
            end
        end
    end

    print ret

    if (send quantity of number greater than pivot to parent) do
        error "error sending number quantity than pivot"
        free(childArr)
        exit(1)
    end

    return pivot

end
```

Small function

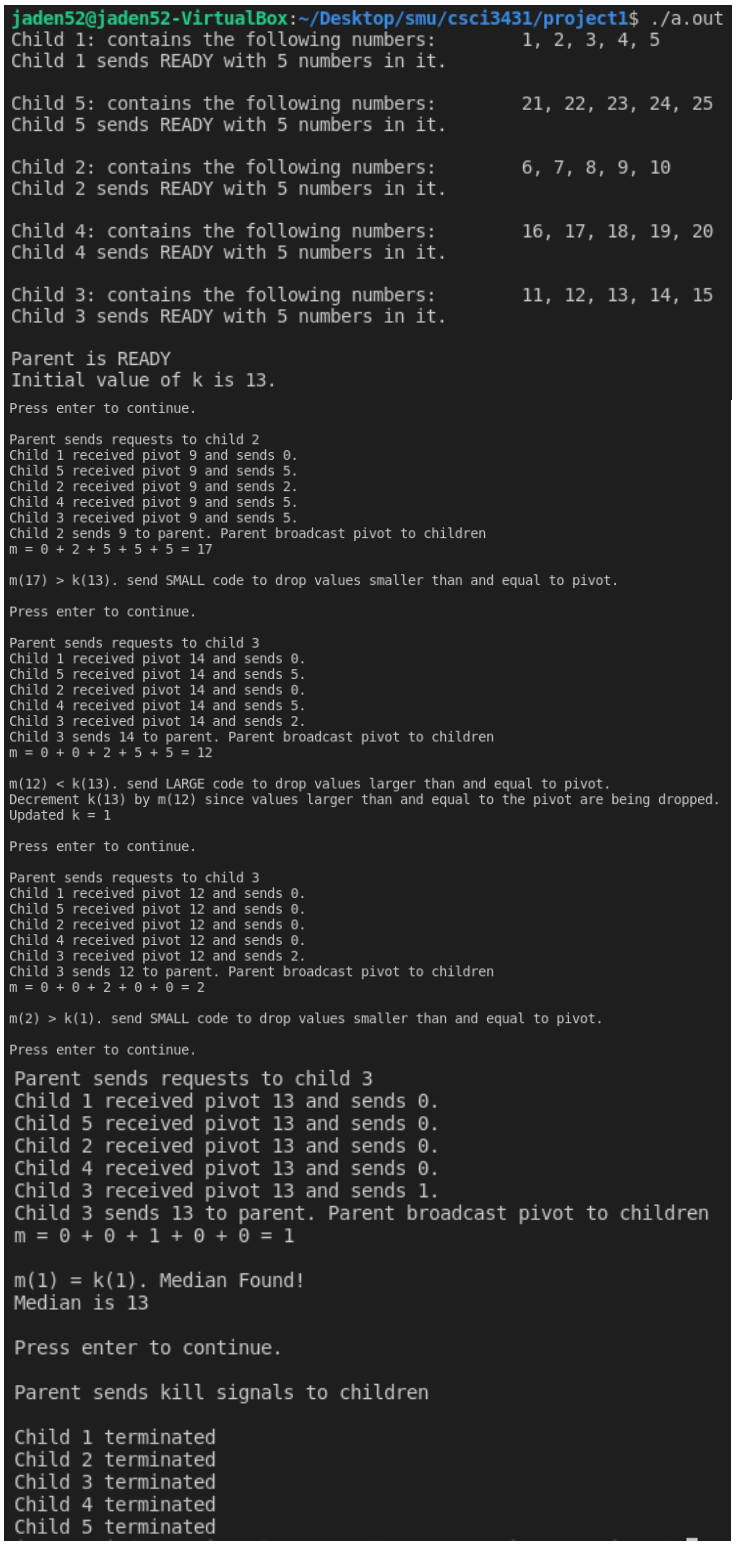
```
function small(int pivot, int *childArr, int arraySize)) do
    initialize index = 0 // new index
    for (i in arraySize) do
        if (childArr[i] > pivot) do
            childArr[index] = childArr[i]
            increment index
        end
    end

    return index
end
```

Large function

```
function large(int pivot, int *childArr, int arraySize) do
    initialize index = 0 // new index
    for (i in arraySize) do
        if (childArr[i] < pivot) do
            childArr[index] = childArr[i]
            increment index
        end
    end

    return index
end
```

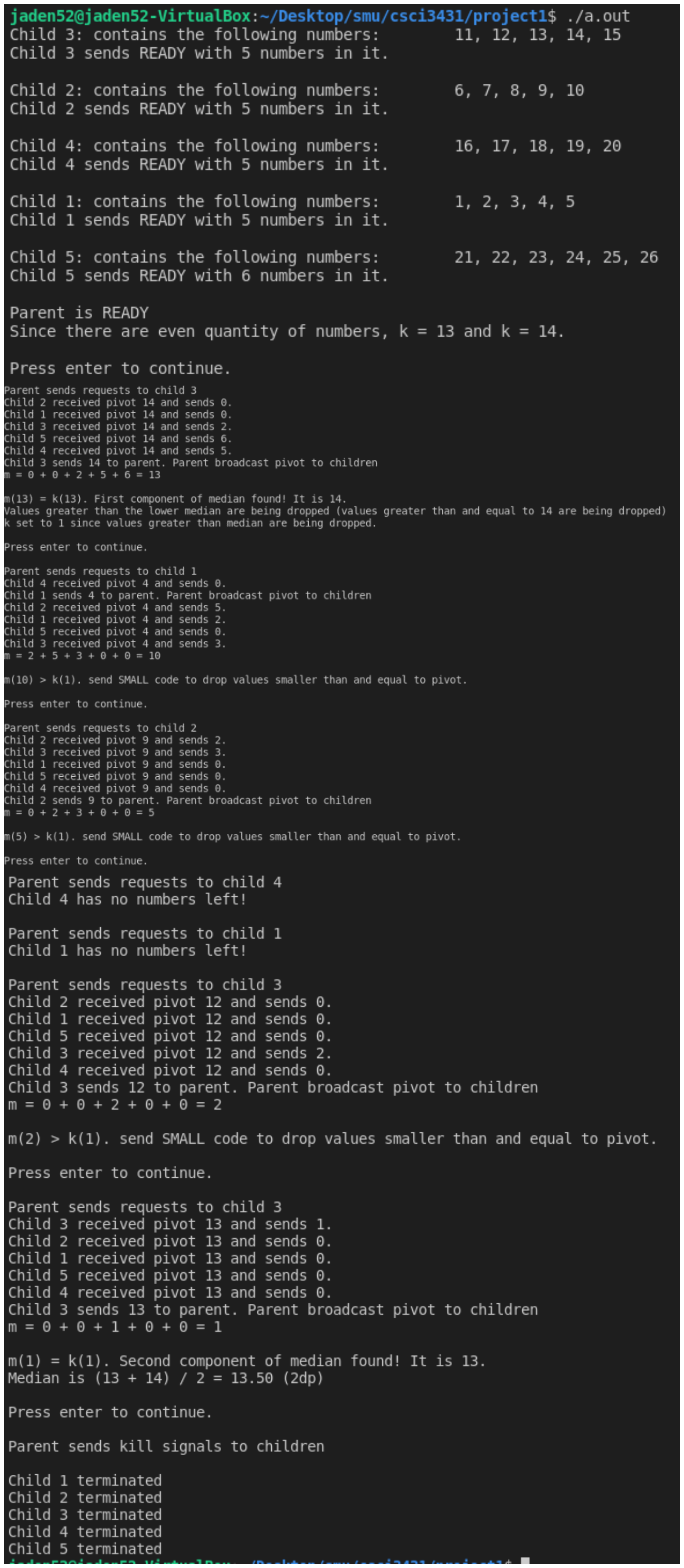
DEMO (screenshots)

Odd quantity of numbers



```
jaden52@jaden52-VirtualBox:~/Desktop/smu/csci3431/project1$ ./a.out
Child 1: contains the following numbers:        1, 2, 3, 4, 5
Child 1 sends READY with 5 numbers in it.

Child 5: contains the following numbers:        21, 22, 23, 24, 25
Child 5 sends READY with 5 numbers in it.

Child 2: contains the following numbers:        6, 7, 8, 9, 10
Child 2 sends READY with 5 numbers in it.

Child 4: contains the following numbers:        16, 17, 18, 19, 20
Child 4 sends READY with 5 numbers in it.

Child 3: contains the following numbers:        11, 12, 13, 14, 15
Child 3 sends READY with 5 numbers in it.

Parent is READY
Initial value of k is 13.
Press enter to continue.

Parent sends requests to child 2
Child 1 received pivot 9 and sends 0.
Child 5 received pivot 9 and sends 5.
Child 2 received pivot 9 and sends 2.
Child 4 received pivot 9 and sends 5.
Child 3 received pivot 9 and sends 5.
Child 2 sends 9 to parent. Parent broadcast pivot to children
m = 0 + 2 + 5 + 5 + 5 = 17

m(17) > k(13). send SMALL code to drop values smaller than and equal to pivot.

Press enter to continue.

Parent sends requests to child 3
Child 1 received pivot 14 and sends 0.
Child 5 received pivot 14 and sends 5.
Child 2 received pivot 14 and sends 0.
Child 4 received pivot 14 and sends 5.
Child 3 received pivot 14 and sends 2.
Child 3 sends 14 to parent. Parent broadcast pivot to children
m = 0 + 0 + 2 + 5 + 5 = 12

m(12) < k(13). send LARGE code to drop values larger than and equal to pivot.
Decrement k(13) by m(12) since values larger than and equal to the pivot are being dropped.
Updated k = 1

Press enter to continue.

Parent sends requests to child 3
Child 1 received pivot 12 and sends 0.
Child 5 received pivot 12 and sends 0.
Child 2 received pivot 12 and sends 0.
Child 4 received pivot 12 and sends 0.
Child 3 received pivot 12 and sends 2.
Child 3 sends 12 to parent. Parent broadcast pivot to children
m = 0 + 0 + 2 + 0 + 0 = 2

m(2) > k(1). send SMALL code to drop values smaller than and equal to pivot.

Press enter to continue.

Parent sends requests to child 3
Child 1 received pivot 13 and sends 0.
Child 5 received pivot 13 and sends 0.
Child 2 received pivot 13 and sends 0.
Child 4 received pivot 13 and sends 0.
Child 3 received pivot 13 and sends 1.
Child 3 sends 13 to parent. Parent broadcast pivot to children
m = 0 + 0 + 1 + 0 + 0 = 1

m(1) = k(1). Median Found!
Median is 13

Press enter to continue.

Parent sends kill signals to children

Child 1 terminated
Child 2 terminated
Child 3 terminated
Child 4 terminated
Child 5 terminated
```

Even quantity of numbers



```
jaden52@jaden52-VirtualBox:~/Desktop/smu/csci3431/project1$ ./a.out
Child 3: contains the following numbers:        11, 12, 13, 14, 15
Child 3 sends READY with 5 numbers in it.

Child 2: contains the following numbers:        6, 7, 8, 9, 10
Child 2 sends READY with 5 numbers in it.

Child 4: contains the following numbers:        16, 17, 18, 19, 20
Child 4 sends READY with 5 numbers in it.

Child 1: contains the following numbers:        1, 2, 3, 4, 5
Child 1 sends READY with 5 numbers in it.

Child 5: contains the following numbers:        21, 22, 23, 24, 25, 26
Child 5 sends READY with 6 numbers in it.

Parent is READY
Since there are even quantity of numbers, k = 13 and k = 14.

Press enter to continue.
Parent sends requests to child 3
Child 2 received pivot 14 and sends 0.
Child 1 received pivot 14 and sends 0.
Child 3 received pivot 14 and sends 2.
Child 5 received pivot 14 and sends 6.
Child 4 received pivot 14 and sends 5.
Child 3 sends 14 to parent. Parent broadcast pivot to children
m = 0 + 0 + 2 + 5 + 6 = 13

m(13) = k(13). First component of median found! It is 14.
Values greater than the lower median are being dropped (values greater than and equal to 14 are being dropped)
k set to 1 since values greater than median are being dropped.

Press enter to continue.

Parent sends requests to child 1
Child 4 received pivot 4 and sends 0.
Child 1 sends 4 to parent. Parent broadcast pivot to children
Child 2 received pivot 4 and sends 5.
Child 1 received pivot 4 and sends 2.
Child 5 received pivot 4 and sends 0.
Child 3 received pivot 4 and sends 3.
m = 2 + 5 + 3 + 0 + 0 = 10

m(10) > k(1). send SMALL code to drop values smaller than and equal to pivot.

Press enter to continue.

Parent sends requests to child 2
Child 2 received pivot 9 and sends 2.
Child 3 received pivot 9 and sends 3.
Child 1 received pivot 9 and sends 0.
Child 5 received pivot 9 and sends 0.
Child 4 received pivot 9 and sends 0.
Child 2 sends 9 to parent. Parent broadcast pivot to children
m = 0 + 2 + 3 + 0 + 0 = 5

m(5) > k(1). send SMALL code to drop values smaller than and equal to pivot.

Press enter to continue.
Parent sends requests to child 4
Child 4 has no numbers left!

Parent sends requests to child 1
Child 1 has no numbers left!

Parent sends requests to child 3
Child 2 received pivot 12 and sends 0.
Child 1 received pivot 12 and sends 0.
Child 5 received pivot 12 and sends 0.
Child 3 received pivot 12 and sends 2.
Child 4 received pivot 12 and sends 0.
Child 3 sends 12 to parent. Parent broadcast pivot to children
m = 0 + 0 + 2 + 0 + 0 = 2

m(2) > k(1). send SMALL code to drop values smaller than and equal to pivot.

Press enter to continue.

Parent sends requests to child 3
Child 3 received pivot 13 and sends 1.
Child 2 received pivot 13 and sends 0.
Child 1 received pivot 13 and sends 0.
Child 5 received pivot 13 and sends 0.
Child 4 received pivot 13 and sends 0.
Child 3 sends 13 to parent. Parent broadcast pivot to children
m = 0 + 0 + 1 + 0 + 0 = 1

m(1) = k(1). Second component of median found! It is 13.
Median is (13 + 14) / 2 = 13.50 (2dp)

Press enter to continue.

Parent sends kill signals to children

Child 1 terminated
Child 2 terminated
Child 3 terminated
Child 4 terminated
Child 5 terminated
```

Known weakness

Due to how the quantity of numbers above the median is counted the code will not be able to get the median if the median value appears more than once, such that the position of the median is never reached. This happens as even if the randomly selected value is the median, when the algorithm counts the quantity of values above the pivot, the other occurrences will make the count wrong.

An example:

Consider odd quantity of numbers, and the median being the value denoted by X.

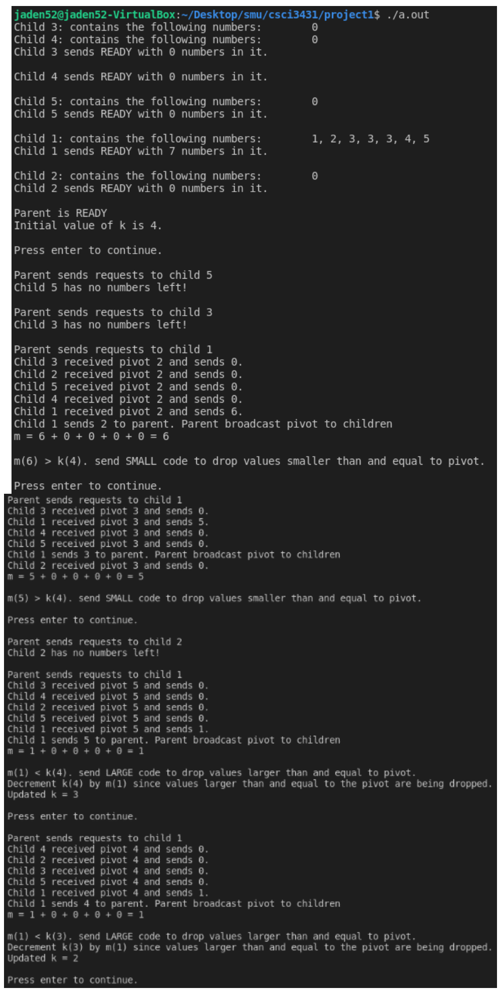
…	X	X (median)	X	…

k will point to the index of the X in the middle, but m will always be above it of below.

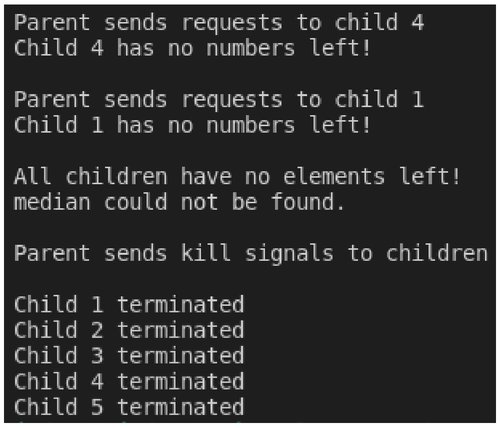
Consider the case that the pivot is the value of X, when the count is done, the value always points to the wrong number.

DEMO:

1	2	3	3 (median)	3	4	5



```
jaden52@jaden52-VirtualBox:~/Desktop/smu/csci3431/project1$ ./a.out
Child 3: contains the following numbers:        0
Child 4: contains the following numbers:        0
Child 3 sends READY with 0 numbers in it.

Child 4 sends READY with 0 numbers in it.

Child 5: contains the following numbers:        0
Child 5 sends READY with 0 numbers in it.

Child 1: contains the following numbers:        1, 2, 3, 3, 3, 4, 5
Child 1 sends READY with 7 numbers in it.

Child 2: contains the following numbers:        0
Child 2 sends READY with 0 numbers in it.

Parent is READY
Initial value of k is 4.

Press enter to continue.

Parent sends requests to child 5
Child 5 has no numbers left!

Parent sends requests to child 3
Child 3 has no numbers left!

Parent sends requests to child 1
Child 3 received pivot 2 and sends 0.
Child 2 received pivot 2 and sends 0.
Child 5 received pivot 2 and sends 0.
Child 4 received pivot 2 and sends 0.
Child 1 received pivot 2 and sends 6.
Child 1 sends 2 to parent. Parent broadcast pivot to children
m = 6 + 0 + 0 + 0 + 0 = 6

m(6) > k(4). send SMALL code to drop values smaller than and equal to pivot.

Press enter to continue.
Parent sends requests to child 1
Child 3 received pivot 3 and sends 0.
Child 1 received pivot 3 and sends 5.
Child 4 received pivot 3 and sends 0.
Child 5 received pivot 3 and sends 0.
Child 1 sends 3 to parent. Parent broadcast pivot to children
Child 2 received pivot 3 and sends 0.
m = 5 + 0 + 0 + 0 + 0 = 5

m(5) > k(4). send SMALL code to drop values smaller than and equal to pivot.

Press enter to continue.

Parent sends requests to child 2
Child 2 has no numbers left!

Parent sends requests to child 1
Child 3 received pivot 5 and sends 0.
Child 4 received pivot 5 and sends 0.
Child 2 received pivot 5 and sends 0.
Child 5 received pivot 5 and sends 0.
Child 1 received pivot 5 and sends 1.
Child 1 sends 5 to parent. Parent broadcast pivot to children
m = 1 + 0 + 0 + 0 + 0 = 1

m(1) < k(4). send LARGE code to drop values larger than and equal to pivot.
Decrement k(4) by m(1) since values larger than and equal to the pivot are being dropped.
Updated k = 3

Press enter to continue.

Parent sends requests to child 1
Child 4 received pivot 4 and sends 0.
Child 2 received pivot 4 and sends 0.
Child 3 received pivot 4 and sends 0.
Child 5 received pivot 4 and sends 0.
Child 1 received pivot 4 and sends 1.
Child 1 sends 4 to parent. Parent broadcast pivot to children
m = 1 + 0 + 0 + 0 + 0 = 1

m(1) < k(3). send LARGE code to drop values larger than and equal to pivot.
Decrement k(3) by m(1) since values larger than and equal to the pivot are being dropped.
Updated k = 2

Press enter to continue.
```



```
Parent sends requests to child 4
Child 4 has no numbers left!

Parent sends requests to child 1
Child 1 has no numbers left!

All children have no elements left!
median could not be found.

Parent sends kill signals to children

Child 1 terminated
Child 2 terminated
Child 3 terminated
Child 4 terminated
Child 5 terminated
```

Appendix

Snprintf combine string

https://www.geeksforgeeks.org/snprintf-c-library

```
char buffer[50];

snprintf(buffer, sizeof(buffer), "%s\n", s);
```

1st argument: variable to store the resulting string

2nd argument: max size that can be stored in the variable

3rd argument: string format for the resulting string

4th onward arguments: variable to be use in the string format (3rd argument)

fscanf read from file

https://www.geeksforgeeks.org/scanf-and-fscanf-in-c/

```
int buffer
while ((fscanf(fp, "%d", &buffer)) == 1)
{
    // content read stored in buffer
}
```

read the next integer in the file
fscanf returns number of matches, hence 1 since %d is the only match to be made if successful

srand and rand for random number generator

https://stackoverflow.com/questions/822323/how-to-generate-a-random-int-in-c

```
#include <time.h> // required to get current time function time

srand(time(NULL));   // Initialization, should only be called once.
int randomVal = rand() % val;
```

srand set the seed for the pseudo random number generator, time is used to get a random generator each time the program is run

% val, since rand return a large value, we use modulus to get a value that is between 0 and (val – 1)

waitpid to wait for any child process

https://stackoverflow.com/questions/21248840/example-of-waitpid-in-use

waitpid(pidArr[i], NULL, 0)

1st argument: pidArr[i] stores the id of the child process upon creation, use to wait for that specific process to end

getchar to wait for user to press enter before continuing

https://stackoverflow.com/questions/18801483/press-any-key-to-continue-function-in-c

use to pause the program until the user press enter

```
void halt()
{
    printf("\npress enter to continue.");
    getchar();
}
```