ENEE 641

# PROGRAMMING ASSIGNMENT 1

2 Layer Channel Routing Problem

Abhay Raina
Date of Submission:
15-10-14

**ABSTRACT**

*The problem is to find the minimum number of tracks for any n terminal channel routing problem, there is no vertical constraint on assignment of tracks but there is a horizontal constraint. Moreover, overlapping intervals on the same track are not allowed and our algorithm has to be linear in time. This solution uses arrays to hold various properties of the given intervals and then use radix sort three times to sort these arrays using different arrays as references each time. Finally, we use a stack to assign tracks to the intervals such that there is no overlap of tracks.*

## 1. INTRODUCTION

The Channel Routing Problem (CRP) is the problem of connecting terminals belonging to signal nets and located on two opposite sides of a rectangular channel so that the wiring uses minimum area. Ideally, other cost measures (e.g., the number of contacts) should also be minimized in the routing process. More formally, in a CRP we are given a channel of n intervals where each interval $N_i$ has two end points i.e. $(l_i, r_i)$. We have different types of constraints on the CRP in different situation here we don't have a vertical constraint but there is a horizontal constraint on the overlap of intervals. We call the problem a 2-terminal CRP, otherwise a multi-terminal CRP.

Because of the importance of channel routing in the design of layout systems [IIS, R], numerous heuristics and approximation algorithms have been proposed for a number of wiring models [BBL, D, H, 0, PL, RBM, RF, SP, YK]. In this 2-layer channel routing (in which two wires on the same track are not allowed to share intervals) the algorithm is expected to assign tracks to various intervals such that we won't have any interval overlapping i.e. on the same track wires joining different contacts aren't supposed to overlap each other. Keeping that in mind we are supposed to assign tracks such that all the intervals are accommodated on minimum number of tracks. Also, we are expected to use this algorithm for a huge number of input values hence, it needs to be linear in time so that we can get results in realistic time frame as the number of intervals in the input increases.

In this algorithm, initially intervals are read from an input file and then memory is allocated according to the number of intervals in that file (mentioned in the first line of the file). We basically allocate memory for three arrays initially and input the intervals from the file to the array 'a'. We fill the array b with 0 and 1, marking a left end point with 0 and marking a right end point with 1. In the next array 'c' we put in the interval number corresponding to the end point in the 'a' array. Then we perform a radix sort on array 'b' and change the array 'a' & 'c' values accordingly (to get all the left and right end points together). The we perform the radix sort on array 'a' and change 'b' & 'c' accordingly. Further a stack is used to assign tracks, with a pop operation being performed at left end point and push being performed at the right end point thereby giving us correct track allocations for correct interval numbers in the 'track' array. Another array 't' holds tracks according to each end point. Now 't' array is sorted using the radix sort and then we just output the values to file in the correct order as required with column having the track number and subsequent columns the end points.

We perform theoretical analysis on the code w.r.t the space and time by doing a line by line analysis of the code. Then we use our program to find the time required for the execution of the program and the space required for the program. Later we compare these two, and find that our algorithm is linear in both space and time.

- Section 2 describes the problem definition, Solution and Bound Analysis.
- Section 3 presents the code analysis.
- In Section 4 we discuss the experiment and its analysis with respect to the theoretical results.
- Section 5 contains the discussions about the process of development of the algorithm.
- Section 6 has the bonus part 1.
- Section 7 has the bonus part 2.
- Section 8 has the references for the problem.

## 2. PROBLEM DEFINITION, SOLUTION & BOUND ANALYSES

### 2.1 PROBLEM DEFINITION

1. Develop a linear time algorithm to find an optimal solution for the following problem.
Problem: Given a set of n closed intervals, assign them to a minimum number of tracks in such a way that no two intervals overlap on each track.

Let $I_j = [l_j, r_j]$ for j = 1, 2, …, n denote an interval having left end point $l_j$ and right end point $r_j$ where $l_j$ and $r_j$ are integers with $0 \leq l_j < r_j \leq m$ for a fixed integer m. (Note that m is the number of terminals on each side of a channel for the case of a two layer channel routing problem discussed in class). Two intervals $Ij = [l_j, r_j]$ and $I_k = [l_k, r_k]$ are said to overlap if $l_j < l_k \leq r_j < r_k$ or $l_k < l_j \leq r_k < r_j$.

### 2.2 SOLUTION

In the question n set of closed intervals are given to us. So we take following as the input to our program.

Important variables definitions, used in the program:
1. Total number of tracks (denoted by) = ‘n’
2. Left and right ends of each interval are put in array ‘A[2n]’
3. Alternate 0 and 1 are inserted in array ‘B[2n]’
4. Interval number is mentioned inside array ‘C[2n]’
5. Array Ares[2n] stores the sorted array A[2n]
6. Array Bres[2n] stores the changed array B[2n]
7. Array Cres[2n] stores the sorted array C[2n]
8. An array called ‘stack’ containing numbers 1 to n in order. i.e. stack[n]
9. An array ‘track’ containing the track numbers associated to the intervals to which its indexes point.
10. m is the highest value amongst the endpoints as given in the question.

**PSUEDOCODE:**

Initially I define a pseudo code for the Radixsort(array,b,c,size,large)

| | | Cost | Times |
|---|---|---|---|
| 1. | While(Largestnumber/SignificantDIgit>0) | c16 | d |
| 2. | Int bucket[10]={0} | 1 | d |
| 3. | for i← 0 to Size do | c17 | d*(2n+1) |
| 4. | bucket[(array[i] / significantDigit) % 10]++ | c18 | d*(2n) |
| 5. | for i← 1 to 9 do | c19 | d*11 |
| 6. | bucket[i] += bucket[i - 1] | 1 | d*10 |
| 7. | for i← size-1 to 0 do | c20 | d*(2n+1) |
| 8. | int l = --bucket[(array[i] / significantDigit) % 10]; | c21 | d*(2n) |
| 9. | semiSorted[l] = array[i]; | 1 | d*(2n) |
| 10. | bsort[l]=b[i]; | 1 | d*(2n) |
| 11. | csort[l]=c[i]; | 1 | d*(2n) |
| 12. | for (i = 0; i < size; i++) | c22 | d*(2n+1) |
| 13. | {array[i] = semiSorted[i]; | 1 | d*(2n) |
| 14. | b[i] = bsort[i]; | 1 | d*(2n) |
| 15. | c[i] = csort[i]; | 1 | d*(2n) |
| 16. | significantDigit *= 10; | 1 | d |

Now I Inputs the end points into the file in the order of left end point first followed by the right end point.

| | | | |
|---|---|---|---|
| 17. | for i ← 0 to 2n-1 do | c1 | 2n+1 |
| 18. | A[i] ← Input end points one by one. | c2 | 2n |
| 19. | for i ← 0 to n-1 do | c3 | n+1 |
| 20. | B[i]←0 | 1 | n |
| 21. | for i ← n to 2n-1 do | c4 | n+1 |
| 22. | B[i]←1 | 1 | n |
| 23. | j ← 1 | 1 | 1 |
| 24. | for i ← 0 to n-1 | c5 | n+1 |
| 25. | C[i]← i+1 | 1 | n |
| 26. | for i ← n to 2n-1 | 1 | n |
| 27. | C[i+n]←C[i+1] | 1 | n |
| 28. | stack[n-1]←0 | 1 | 1 |
| 29. | for i← n-1 to 0 | c14 | n+1 |
| 30. | stack[i]←stack[i]+1 | 1 | n |
| 31. | Radixsort(A,B,C,2n,m) | c30 | 1 |
| 32. | top← n-1 | 1 | 1 |
| 33. | for i← 0 to 2n-1 do | c13 | 2n+1 |
| 34. | if Bres[i]==0 | 1 | 2n |
| 35. | then track[Cres[i]]←stack [top] | 1 | 2n |
| 36. | top←top-1 | 1 | 2n |
| 37. | if mintop < stack[top] | 1 | 2n |
| 38. | then mintop← stack[top] | 1 | 2n |
| 39. | else if Bres[i]==1 | 1 | 2n |

| | | |
|---|---|---|
| 40. then top←top-1 | 1 | 2n |
| 41. stack [top] ← track[Cres[i]] | 1 | 2n |
| 42. for i=0 to i=2n-1 do | c27 | 2n+1 |
| 43.t[i]←track[c[i]] | 1 | 2n |
| 44. Radixsort(t,a,c,2n,n+1) | c31 | 1 |
| 45. Output the values back to the file | c32 | c33 |

Now the sum of all cost*times for Radixsort() is a constant hence we directly substitute constant values for it inside the program. Also the putting output back to the file requires only linear time as it uses only single for loops and comparisons hence we use the constants for that too.

## Time analysis:

Now we see that at each steps through 1-45 **Cost*Times** is of the form

$a_i * n + b_i$ where ai and bi are constants.

Now adding all these together will be like

$$\sum_{i=1}^{45} a_i * n + b_i$$

which is again of the form

$c * n + d$ , where c and d are constants.

Therefore its order of n.

$O(n)$ , where c and d are constants.

There above algorithm has **Linear time complexity.**

## Space Analysis:

We allocate space only in lines 17-30 therefore its analysis will be as follows

1. For Array ‘A’ we require 2*(2*n) bytes.
2. For Array ‘B’ we require 2*(2*n) bytes.
3. For Array ‘C’ we require 2*(2*n) bytes.
4. For Array ‘t’ we require 2*(2*n) bytes.
5. For stack we require 2*(n) bytes.
6. For ‘track’ array we require 2*(n)bytes
7. Also in each radix sort we will at max require 3*(2*(2*n) bytes.

Therefore total memory used will be approximately 32*n which is linear with respect to the input size and hence we can expect the experimental values to be similar. Also we are neglecting the space required for other variables as their values will be insignificant when compared to the calloc() memory requirements. Therefore its of O(n) space complexity.

## 3. CODE ANALYSIS:

Here in the code analysis we see the time requirements of the code, In case of for loops if they contain just comparisons and other statements not dependent upon 'n' we take the cost of the whole for loop to be constant. As the total time and space required by radix sort is constant we replace the right side with a constant whenever radix sort is called. The C followed by a number is just a random constant we assign to them.

For the memory assignment we consider only the major memory assignment steps like calloc() as others are insignificant.

| Program | Time | Memory |
|---|---|---|
| #include<stdio.h><br>#include <stdlib.h><br>#include<stdlib.h><br>#include <time.h><br>void radixSort(unsigned int*array, unsigned int * b, unsigned int * c ,int size, int large) | | |
| { int i; | 1 | |
| unsigned int *semiSorted,*bsort,*csort; | 1 | |
| semiSorted =(unsigned int*) calloc (2*size,sizeof(unsigned int)); | C1 | 2*2*n |
| bsort=(unsigned int*) calloc ( 2*size,sizeof(unsigned int)); | C2 | 2*2*n |
| csort=(unsigned int*) calloc ( 2*size,sizeof(unsigned int)); | C3 | 2*2*n |
| int significantDigit = 1; | 1 | |
| int largestNum = large; | 1 | |
| while (largestNum / significantDigit > 0) | C4 | |
| {int bucket[10] = { 0 } | 1 | |
| for (i = 0; i < size; i++) | C5(Full loop) | |
| bucket[(array[i] / significantDigit) % 10]++; | | |
| for (i = 1; i < 10; i++) | C6(Full loop) | |
| {bucket[i] += bucket[i - 1];} | | |
| for (i = size - 1; i >= 0; i--) | C7for the whole for loop as it has constant time operations | |
| { int l = --bucket[(array[i] / significantDigit) % 10]; | | |
| semiSorted[l] = array[i]; | | |
| bsort[l]=b[i]; | | |
| csort[l]=c[i];} | | |
| for (i = 0; i < size; i++) | C80(Full loop) | |
| {array[i] = semiSorted[i]; | C7 whole for loop it has constant time | |
| b[i] = bsort[i]; | | |
| c[i] = csort[i]; } | | |
| significantDigit *= 10; } | C50 | |
| free(semiSorted); | 1 | |
| free(bsort); | 1 | |
| free(csort);} | 1 | |
| void main(int argc, char *argv[]) | | |
| { char *inFileString, *outFileString; | C60 | |

| | | |
|---|---|---|
| clock_t t1, t2,t3,t4,t5,t6; | C70 | |
| t1 = clock(); | 1 | |
| FILE*infile; | 1 | |
| int i=0,n,j,k=0,m,count=0,memsize=0; | C8 | |
| inFileString = argv[1]; | 1 | |
| outFileString = argv[2]; | 1 | |
| infile=fopen(inFileString,"r"); | 1 | |
| if (infile == NULL) { | 1 | |
| fprintf(stderr, "Error! Could not open file.\n"); } | 1 | |
| fscanf(infile,"%d",&n); | 1 | |
| printf("Total number of intervals is %d \n",n); | 1 | |
| fscanf(infile,"%d",&m); | 1 | |
| printf("Maximum value of the input to the radix is %d \n",m); | 1 | |
| unsigned int *a,*b,*c,*stack,*track,*t; | 1 | |
| a = (unsigned int*) calloc ( 2*n,sizeof(unsigned int)); | C9 | 2*2*n |
| memsize=memsize+(2*(2*n)); | 1 | |
| b = (unsigned int*) calloc ( 2*n,sizeof(unsigned int)); | C10 | 2*2*n |
| memsize=memsize+(2*(2*n)); | 1 | |
| c = (unsigned int*) calloc ( 2*n,sizeof(unsigned int)); | C11 | 2*2*n |
| memsize=memsize+(2*(2*n)); | 1 | |
| stack = (unsigned int*) calloc ( n,sizeof(unsigned int)); | C12 | 2*n |
| memsize=memsize+(2*(n)); | 1 | |
| track = (unsigned int*) calloc ( n,sizeof(unsigned int)); | C13 | 2*n |
| memsize=memsize+(2*(n)); | 1 | |
| i=0;j=0; | 1 | |
| while (k<2*n) { | C14 for the | |
| if(count==0) | whole while | |
| {fscanf(infile,"%d",&a[i]); | loop as it has | |
| count=1; | constant time | |
| i++;} | operations | |
| else{ fscanf(infile,"%d",&a[n+j]); | C15 for the | |
| count=0; | whole while | |
| j++;} | loop as it has | |
| k++;} | constant time | |
| fclose(infile); | operations. | |
| for (i=0;i<n;i++) | C16 for whole | |
| b[i]=0; | for loop. | |
| for (i=n;i<2*n;i++) | C17 for whole | |
| b[i]=1; | for loop | |
| for (i=0;i<n;i=i++) | C18 for whole | |
| c[i]=i+1; | loop | |
| for (i=0;i<n;i=i) | C19 for the | |
| c[n+i]=i+1; | whole loop. | |
| | | |
| stack[0]=n; | 1 | |
| for (i=1;i<n;i++) | C20 | |
| stack[i]=stack[i-1]-1; | | |

| | | |
|---|---|---|
| int mintop=0; | 1 | |
| t3 = clock(); | 1 | |
| radixSort(a, b , c , 2*n , m); | 1 | |
| t4 = clock(); | 1 | |
| int top = n-1; | 1 | |
| for(i=0;i<2*n;i++) | C21 for the whole loop as it contains only comparisons and assignments. | |
| { if (b[i]==0) | | |
| { track[c[i]]=stack[top]; | | |
| top=top-1;} | | |
| if (b[i]==1) | | |
| { top=top+1; | | |
| stack[top]=track[c[i]];} | | |
| } | | |
| free(stack); | 1 | |
| free(b); | 1 | |
| t = (unsigned int*) calloc ( 2*n,sizeof(unsigned int)); | C22 | 2*2*n |
| memsize=memsize+(2*(2*n)); | 1 | |
| for(i=0; i<2*n; i++) | C23 | |
| { t[i]=track[c[i]]; | | |
| } | | |
| t5 = clock(); | 1 | |
| radixSort(t, a , c , 2*n , 211835); | C24 | |
| t6 = clock(); | 1 | |
| mintop=t[2*n-1]; | 1 | |
| printf("%d \n",mintop); | 1 | |
| FILE*fp1; | 1 | |
| fp1 =fopen(outFileString, "w+"); | 1 | |
| i=1; | 1 | |
| fprintf(fp1,"%s%d","T",i); | 1 | |
| free(c); | 1 | |
| free(track); | 1 | |
| for(i=0;i<((2*n)-1);i++) | C25 for whole for loop as it includes assignments comparions and prints which are linear in time. | |
| { fprintf(fp1, " %ld",a[i]); | | |
| if (t[i]!=t[i+1]) | | |
| fprintf(fp1, "\n%s%ld","T",t[i+1]);} | | |
| fprintf(fp1," %ld",a[2*n-1]); | | |
| fclose(fp1); | | |
| t2 = clock(); | | |
| printf("Total memory used in bytes is %d \n",memsize); } | | |

**Time bounds:**

So after adding all the values in the time column we get that is of the form $a_i * n + b_i$ where ai and bi are constants.

Now adding all these together will be like

$$\sum a_i * n + b_i$$

which is again of the form $c * n + d$ , where c and d are constants. Therefore its order of n. $O(n)$ , where c and d are constants.

Memory bounds:

After adding all the values in the memory column we get that is of the form

$$32 * n$$

Which is linear with respect to time hence we can observe that the memory should vary linearly with number of inputs. . Therefore its order of n. $O(n)$ , where c and d are constants.

There above algorithm has Linear time complexity w.r.t Memory & Time.

## 4. EXPERIMENTS AND ANALYSIS:

The experiment here was to calculate the time and space requirements theoretically and then to compare it with the practical values that we get from our program.

*Table 1: File Size, Memory Required, Execution time & Minimum Number of tracks required.*

| File Number | File Size In Number of intervals | Execution time in seconds | Memory Required in Bytes | No. of Tracks Required |
|---|---|---|---|---|
| 1 | 20000 | 0.38 | 640000 | 9994 |
| 2 | 50000 | 0.85 | 1600000 | 24922 |
| 3 | 80000 | 1.18 | 2560000 | 39750 |
| 4 | 110000 | 1.75 | 3520000 | 39676 |
| 5 | 140000 | 2.14 | 4480000 | 49975 |
| 6 | 170000 | 2.53 | 5440000 | 60718 |
| 7 | 200000 | 2.92 | 6400000 | 71570 |
| 8 | 230000 | 3.47 | 7360000 | 82438 |
| 9 | 260000 | 4.15 | 8320000 | 93591 |
| 10 | 2900000 | 4.28 | 9280000 | 104180 |
| 11 | 3200000 | 4.67 | 10240000 | 114843 |
| 12 | 3500000 | 6.90 | 11200000 | 125053 |
| 13 | 3800000 | 7.16 | 12160000 | 136497 |
| 14 | 4100000 | 7.55 | 13120000 | 147015 |
| 15 | 4400000 | 7.51 | 14080000 | 157928 |
| 16 | 4700000 | 7.67 | 15040000 | 168187 |
| 17 | 5000000 | 7.79 | 16000000 | 178863 |
| 18 | 5300000 | 8.21 | 16960000 | 189632 |
| 19 | 5600000 | 8.30 | 17920000 | 201184 |
| 20 | 5900000 | 8.66 | 18880000 | 211833 |

The above graph contains the experimental values obtained when the c program was run first column contains the file number, second has the Number of intervals in that file, third has the execution time required for the program in seconds, forth is the memory requirements of the code in bytes and fifth contains the minimum number of tracks for each of the files.

**4.1 TIME CALCULATION EXPERIMENT:**

For the time calculation a time counter was placed at the beginning and the end of the program to find the time required to go through the program. The same was repeated for the 20 files given and corresponding execution times were noted down. Table 1 contains the values of the same.

Also we plot the following graph using the values from Table 1.

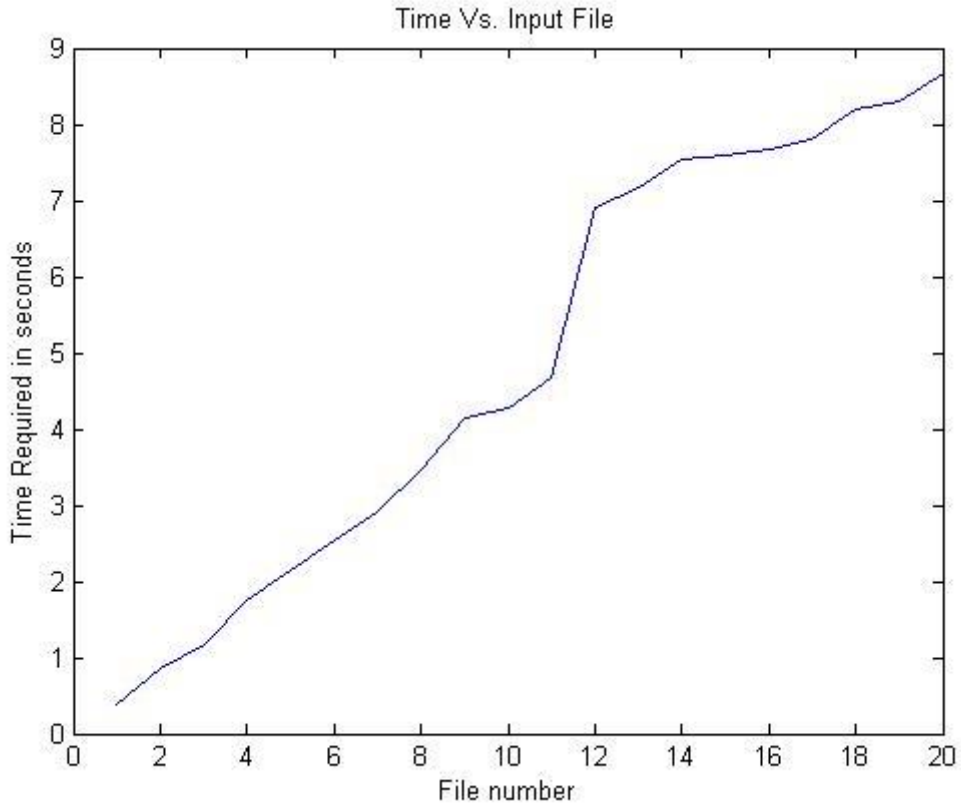


Figure 1: Graph of Execution Time vs. Input File

My theoretical analysis of the code showed that the runtime of the code is linear as there was no construct which contributed a nonlinear timing. When the experimental values were plotted in the graph above result was obtained. As we can see above the graph is largely linear with a few minor deviations at points 10-14. These deviations can happen due to the following reasons:

1. The Input file which was generated randomly is nearing the worst case scenario and we see a little deviation from the average case.
2. The glue system was busier comparatively when running files 10-14 than at other times thereby increasing the time factor by some minor values.

Hence from the above graph we can safely conclude that our code is working as desired from it i.e. in linear time.

**4.2 SPACE CALCULATION EXPERIMENT:**

The space calculation was done using counting the number of calloc() assignments in the program and finding the amount of memory that the reserve in each case and then adding them all up to get an idea of how much memory is required in general. The same was repeated for the 20 files given and corresponding space requirements were noted. Following table contains the values of the same. As at a time we are at max going to run one radix sort I have added the space requirements for one radix sort to indicate the maximum amount of memory that the program takes at any given time as that is the only limiting factor when it comes to the space requirements.

Also we plot the following graph using the above values from Table 1.

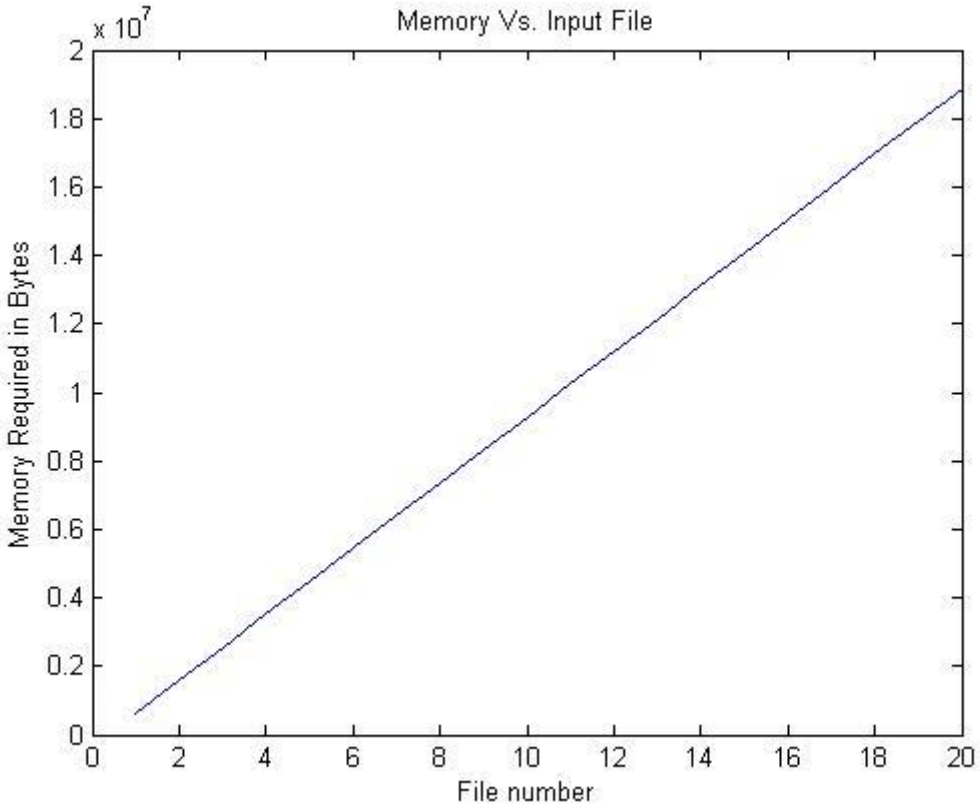


Figure 2: Graph of Memory Required Vs. Input File

The theoretical analysis of the code showed that the Memory space required by the code is linear as there was no construct which contributed a nonlinear space requirement. When the experimental values were plotted in the graph above result was obtained. As we can see above the graph is linear as it was expected to be as the amount of memory required is increasing has the input size is increasing.

Therefore from the above graph we can safely conclude that our code is working as desired from it i.e. the space requirements are linearly dependent upon the input size. As the input size becomes large the space requirement might serve as a constraint for the program to run.

## 5. DISSCUSSIONS

While designing the algorithm following problems were faced:

1. The radix sort function was supposed to sort not only the input array but other arrays too in such a fashion that as the elements of the primary array moved other array elements should have moved in the same manner. This issue was solved by using the variable supposed to index the primary array to perform indexing for the other arrays.

2. It was difficult to assign tracks in the case when the right and the left endpoints of two given intervals coincided, in that case same track was being assigned to two overlapping intervals. To solve this issue I initially inputted the values from the given file in such a way that all the left end points came before all the right endpoints which solved the problem of improper assignment of tracks in case of a overlap.

3. After assignment of the tracks to the 'track' array it was very difficult to print the tracks and intervals to the file in the required order in linear time. To solve this issue an array 't' was created which had all the tracks in the right places according to the correct interval numbers. This then when fed to radix sort gave me a structure using which the tracks and intervals could be written to the file in the correct order.

4. For some reason the output to the file was not coming correct when I used any input file with the size greater than 20k (first input file provided). The reason being that when feeding the array 't' to the radix sort, the algorithm no longer was bound by the upper limit of 9999 number as the track number could have well exceeded that value. This wasn't the case when arrays 'a' was fed to the radix sort as it was limited by the upper limit of 9999 (the upper limit was contained in a variable 'm' and passed but still the track numbers required could have well exceeded 'm' ).

The strengths of my approach would be as follows:

1. The algorithm gives the minimum number of tracks in linear time, hence in case of a increase in the number of input intervals the time taken would change exponentially.
2. The fact that I am reading from the files in all left end points followed by all right end points order I don't need any special logic to solve the collisions.
3. Same radix sort function sorts sort the tracks therefore no additionally code required.
4. The algorithm uses only arrays and hence is easy to understand and implement.

Weaknesses of my approach would be as follows:

1. As the number of inputs increase the amount of memory required at that moment increases and sometimes the system stack won't be having the hold such huge amount of numbers.
2. The radix sort is using some division and modulus operation which consume a lot of cycles therefore increasing the execution time.

Future improvements to my approach could be to improve the radix sort function such that the modulus and division operation can be divided thereby increasing my execution time significantly.

## 6. BONUS SECTION 1

The upper limit of the number of intervals that my program could execute in five minutes is as follows :

For input of 5,000,000 intervals my program took 70.32 seconds.

For input of 10,000,000 intervals my program took 184.02 seconds.

For an input of 15,000,000 intervals my program took 303.12 seconds.

And for an input of 20,000,000 intervals my program took 430.58 seconds.

As shown in the graph.

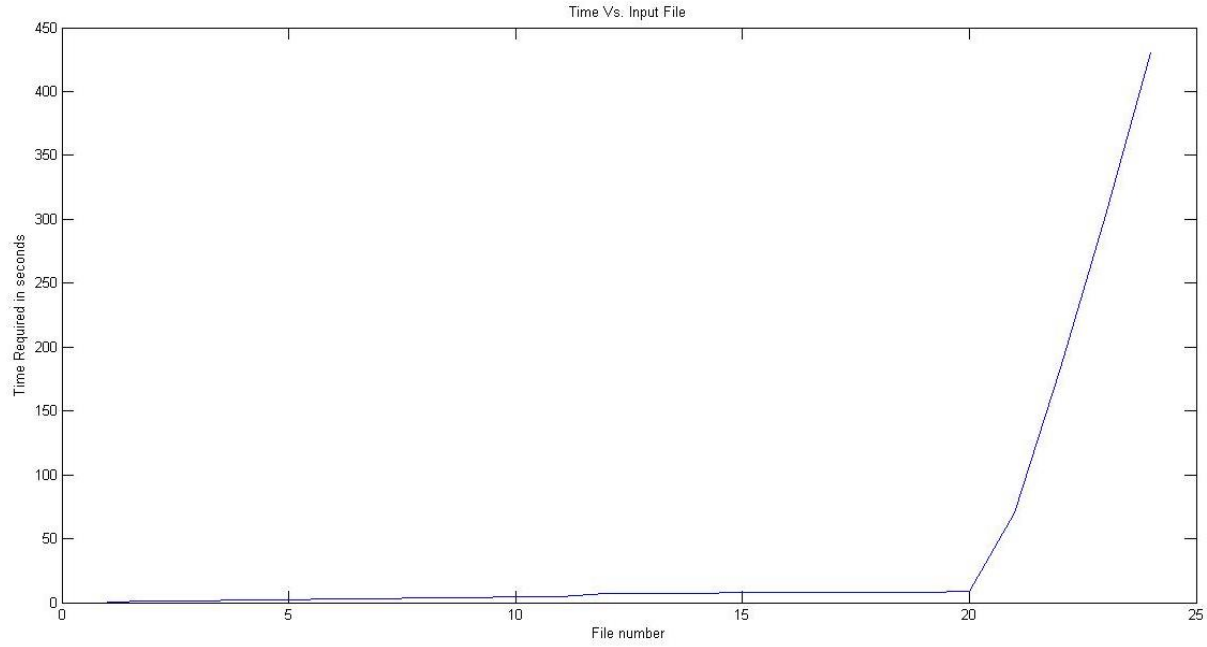


Figure3: Graph plotting n=10million, n=15 million & n=20 million

As we can see although my for files number 21, 22 & 23 the values are quite large but the graph is still following the linear curve.

From the above analysis I can see that my program crosses the five minute limit for an input size of 15,000,000.

## 7. BONUS SECTION 2

Before using the above shown approach to the given problem I had used another approach to perform the desired operation in real time. It had lesser complication with file input but included one extra radixsort() operation. It was observed that the one extra radixsort() induced significant time delay that was around 5 seconds for the 20.in file.

Following is the code for the same:

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

void radixSort(int * array, int * b, int * c ,int size, int large)
{  int i;
  int *semiSorted,*bsort,*csort;
  semiSorted =(int*) calloc ( 2*size,sizeof(int));
  bsort=(int*) calloc ( 2*size,sizeof(int));
  csort=(int*) calloc ( 2*size,sizeof(int));
  int significantDigit = 1;
  int largestNum = large;
  // Loop until we reach the largest significant digit
  while (largestNum / significantDigit > 0)
  { int bucket[10] = { 0 };
    // Counts the number of "keys" or digits that will go into each bucket
   for (i = 0; i < size; i++)
     bucket[(array[i] / significantDigit) % 10]++;
   for (i = 1; i < 10; i++)
          {
     bucket[i] += bucket[i - 1];}
   // Use the bucket to fill a "semiSorted" array
   for (i = size - 1; i >= 0; i--)
          { int l = --bucket[(array[i] / significantDigit) % 10];
            semiSorted[l] = array[i];
     bsort[l]=b[i];
           csort[l]=c[i];}
    for (i = 0; i < size; i++)
   {array[i] = semiSorted[i];
     b[i] = bsort[i];
     c[i] = csort[i];}
   // Move to next significant digit
   significantDigit *= 10;}
  free(semiSorted);
  free(bsort);
  free(csort);
}
```

```
void main(int argc, char *argv[])
{ char *inFileString, *outFileString;
  clock_t t1, t2;
  t1 = clock();
  FILE*infile;
  int i=0,n,m;
  inFileString = argv[1];
  outFileString = argv[2];
  infile=fopen(inFileString,"r");
   if (infile == NULL) {
      fprintf(stderr, "Error! Could not open file.\n");
   }
   fscanf(infile,"%d",&n);
          printf("%d \n",n);
          fscanf(infile,"%d",&m);
          printf("%d \n",m);
          int *a,*b,*c,*stack,*track,*t;
          a = (int*) calloc ( 2*n,sizeof(int));
          b = (int*) calloc ( 2*n,sizeof(int));
          c = (int*) calloc ( 2*n,sizeof(int));
          stack = (int*) calloc ( n,sizeof(int));
          track = (int*) calloc ( n,sizeof(int));
   while (i<2*n)
          {      fscanf(infile,"%d",&a[i]);
                              i++;}
          fclose(infile);

          for (i=0;i<2*n;i++) // Assign alternating 0's and 1's to array B[2n]
   {if (i%2==0) b[i]=0;
   else b[i]=1;}
          int j=1;

          for (i=0;i<2*n;i=i+2)// Put the interval number corresponding to the endpoints in
array C[2n]
     {c[i]=j;
      c[i+1]=c[i];
      j=j+1;}

   stack[0]=n; // Initialize a stack using array with 1 to n track numbers from right to left.
   for (i=1;i<n;i++)
     stack[i]=stack[i-1]-1;

  int mintop=0;
  //Sorting
  radixSort(b, a , c , 2*n , 1);
  radixSort(a, b , c , 2*n , m);
  // Stack
```

```
        int top = n-1;

        for(i=0;i<2*n;i++)
          {   if (b[i]==0)
            { track[c[i]]=stack[top];
             top=top-1;}

                    if (b[i]==1)
            { top=top+1;
              stack[top]=track[c[i]];}
            }

            free(stack);
            free(b);

        t = (int*) calloc ( 2*n,sizeof(int));
        for(i=0; i<2*n; i++)
        { t[i]=track[c[i]];
        }
        radixSort(t, a , c , 2*n , n+1);

        mintop=t[2*n-1];
        printf("%d \n",mintop);
  //writing back to the file
        FILE*fp1;
  fp1 =fopen(outFileString, "w+");
  i=1;
  fprintf(fp1,"%s%d","T",i);
  free(c);
        free(track);
        for(i=0;i<((2*n)-1);i++)
        { fprintf(fp1, " %ld",a[i]);

          if (t[i]!=t[i+1])
             fprintf(fp1, "\n%s%ld","T",t[i+1]);}

        fprintf(fp1," %ld",a[2*n-1]);
        fclose(fp1);
        free(a);
        free(t);
        t2 = clock();
        float diff = (((float)t2 - (float)t1)*0.000001);
        printf("Total Execution time  %f \n",diff);}
```

As has been highlighted above these two portions differ from the original code, now by introducing an extra radixsort() the timing analysis changed as follows.

*Table 2: Timing Analysis of Special Algorithm*

| File Number | Time taken in seconds |
|---|---|
| 1 | 0.73 |
| 2 | 1.62 |
| 3 | 3.21 |
| 4 | 3.89 |
| 5 | 4.52 |
| 6 | 5.13 |
| 7 | 5.89 |
| 8 | 7.09 |
| 9 | 7.48 |
| 10 | 7.63 |
| 11 | 7.94 |
| 12 | 8.43 |
| 13 | 10.04 |
| 14 | 10.76 |
| 15 | 11.62 |
| 16 | 12.21 |
| 17 | 12.78 |
| 18 | 13.01 |
| 19 | 13.45 |
| 20 | 13.96 |

*Figure 1 Timing Analysis of special Algorithm*

The graph for the original algorithm and the special algorithm is as follows

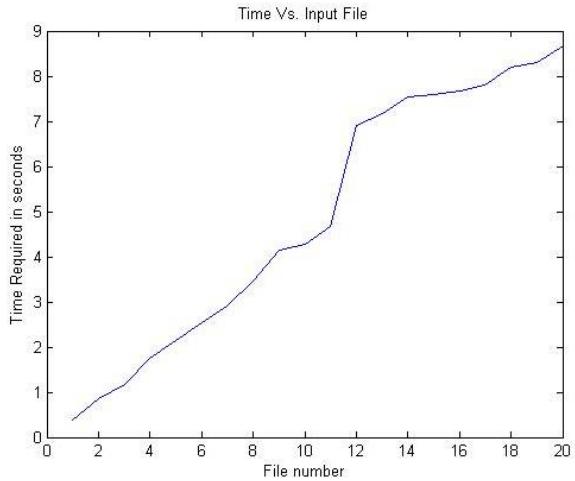


*Figure 4: Timing analysis of Original algorithm*

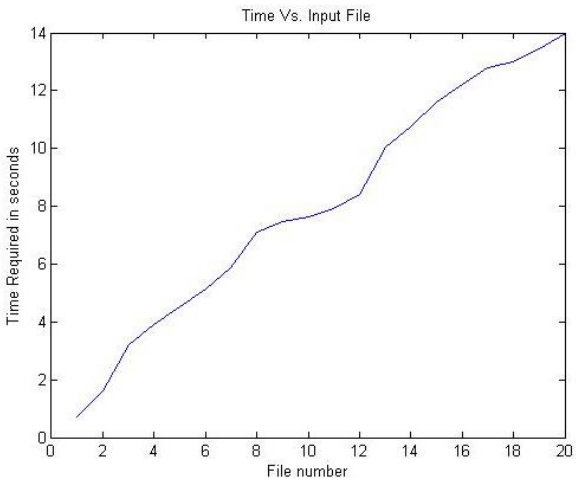


*Figure 5: Timing Analysis for Special Algorithm*

As we can see the graph though still being linear has greater execution time due use of one extra radixsort to sort the unordered input array ‘a’.

Overall we see a difference of 4-5 seconds in the execution of the program. Which is substantial when used for operations which require very fast execution.

## 6. REFERENCES:


1. http://www.tutorialspoint.com/c_standard_library/
2. http://cs.baylor.edu/~maurer/channelrouting.pdf
3. http://en.wikipedia.org/wiki/Channel_router
4. www.wikipedia.org/wiki/**Radix_sort**