

# OS Mid Sem Exam

## Practicals

Yoga Sri Varshan V  
CED18I058

Q. Develop a multithreaded version of Fibonacci Search and compare it with the performance of its equivalent binary search in terms of number of comparisons for large sized inputs and varied search keys. The demonstration must display the passes of the search strategies. Also carry out a detailed performance study on the effect of varying number of threads and its impact on the efficiency of the respective algorithms parallelized versions. Ensure that the testing explores large sized arrays, random distribution of elements. As an add on it would be preferred to have a data creator code which initializes an array of user required size randomly given some boundary conditions.

### Structure of the Report:

- What did I do?
- Fibonacci vs Binary Performance
- Codes and Outputs
- What happened when I tried to vary the number of threads!
- Conclusion

### What did I do?

- Tried understanding Fibonacci Searching from various online sources.

- Coded both Fibonacci and Binary searching algorithms in the same program in a two threaded setup - Two Keys to find - One in the first half of the array and the other in the second half.
- Executed them - found that for most of the test cases Fibonacci was doing better than Binary !
- Analysed the reason for above ( online sources used again! )
- Used a 4 threaded setup to see which one performs better - Varying the number of threads.
- Concluded that the 2 threaded setup was better than 4 threaded!!!!!!.

### Fibonacci Searching vs Binary Searching:

- Fibonacci search is used to search an element of a sorted array with the help of Fibonacci numbers. It studies the locations whose addresses have lower dispersion. Fibonacci number is subtracted from the index thereby reducing the size of the list.
- What happens in binary search is splitting up of the array into two halves every time we narrow in on our hunt for the key.
- Over in Fibonacci, we traverse in a much better way by searching inside segments bordered by fibonacci numbers.

### The complete algorithm of fibonacci -

Let arr[0..n-1] be the input array and element to be searched be x.

1. Find the smallest Fibonacci Number greater than or equal to n. Let this number be fibM [m'th Fibonacci Number]. Let the two Fibonacci numbers preceding it be fibMm1 [(m-1)'th Fibonacci Number] and fibMm2 [(m-2)'th Fibonacci Number].
2. While the array has elements to be inspected:
  1. Compare x with the last element of the range covered by fibMm2
  2. If x matches, return index
  3. Else If x is less than the element, move the three Fibonacci variables two Fibonacci down, indicating elimination of approximately rear two-third of the remaining array.
  4. Else x is greater than the element, move the three Fibonacci variables one Fibonacci down. Reset offset to index. Together these indicate elimination of approximately front one-third of the remaining array.
3. Since there might be a single element remaining for comparison, check if fibMm1 is 1. If Yes, compare x with that remaining element. If match, return index.

### Codes and Outputs Section:

### Fibonacci and Binary Searchings with 2 Threads:

// All the passes shown in the output !

### Code:

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

struct node{
    int *arr;
    int start;
    int end;
    int key;
    int status;
};

int min(int x, int y){ return (x<=y)? x : y; }
void *fib(void *arg){

    struct node* temp=(struct node*) arg;
    int x=temp->key;
    int start=temp->start;
    int n=temp->end;
    int *arr=temp->arr;
    temp->status=-1;
    /* Initialize fibonacci numbers */
    int fibMMm2 = 0;    // (m-2)'th Fibonacci No.
    int fibMMm1 = 1;    // (m-1)'th Fibonacci No.
    int fibM = fibMMm2 + fibMMm1;  // m'th Fibonacci
    /* fibM is going to store the smallest Fibonacci
       Number greater than or equal to n */
    while (fibM < n){
        fibMMm2 = fibMMm1;
        fibMMm1 = fibM;
        fibM = fibMMm2 + fibMMm1;
    }
    // Marks the eliminated range from front
    int offset = start-1;
```

```

/* while there are elements to be inspected. Note that
   we compare arr[fibMm2] with x. When fibM becomes 1,
   fibMm2 becomes 0 */
while (fibM > 1){
    // Check if fibMm2 is a valid location
    int i = min(offset+fibMMm2, n-1);
    /* If x is greater than the value at index fibMm2,
       cut the subarray array from offset to i */
    if (arr[i] < x){
        fibM = fibMMm1;
        fibMMm1 = fibMMm2;
        fibMMm2 = fibM - fibMMm1;
        offset = i;
        printf("Key %d greater than the min(offset+fibMMm2, n-1)
value%d\n",x,arr[i]);
    }
    /* If x is greater than the value at index fibMm2,
       cut the subarray after i+1 */
    else if (arr[i] > x){
        fibM = fibMMm2;
        fibMMm1 = fibMMm1 - fibMMm2;
        fibMMm2 = fibM - fibMMm1;
        printf("Key %d lesser than the min(offset+fibMMm2, n-1)
value%d\n",x,arr[i]);
    }
    /* element found. return index */
    if(arr[i]==x){
        printf("Key %d Found\n",x);
        temp->status=i;
        break;
    }
}
/* comparing the last element with x */
if(fibMMm1 && arr[offset+1]==x)
    temp->status= offset+1;
/*element not found. return -1 */
pthread_exit(NULL);
}

void* bs(void *arg) {

```

```

    struct node* temp=(struct node*) arg;
    temp->status=-1;
    while(temp->start<=temp->end) {
        int mid=temp->start+(temp->end-temp->start)/2;
        if(temp->arr[mid]==temp->key) {
            printf("Key %d found!\n",temp->key);
            temp->status=mid;break;
        }
        else if(temp->arr[mid]<temp->key) {
            printf("Key %d lesser than the mid value
%d\n",temp->key,temp->arr[mid]);
            temp->start=mid+1;
        }
        else{
            printf("Key %d greater than the mid value
%d\n",temp->key,temp->arr[mid]);
            temp->end=mid-1;
        }
    }
    pthread_exit(NULL);
}

void swap(int *p,int *q) {
    int temp=*p;
    *p=*q;
    *q=temp;
}

void bubblesort(int a[],int ch,int start ,int end) {
    int i,j;
    for(i=start;i<end-1;i++) {
        for(j=start;j<end-(i-start)-1;j++) {
            int choice=(ch==0)?(a[j]<a[j+1]):(a[j]>a[j+1]);
            if(choice==1) {
                swap(&a[j],&a[j+1]);
            }
        }
    }
}

```

```

void process3(int arr[],int n,int key1,int key2){
    clock_t t1,t2;
    pthread_t pth[2];
    struct node *temp=(struct node* )malloc(2*sizeof(struct node));
    temp[0].arr=arr;
    temp[0].start=0;
    temp[0].end=n/2-1;
    temp[0].key=key1;

    temp[1].arr=arr;
    temp[1].start=n/2;
    temp[1].end=n;
    temp[1].key=key2;

    t1=clock();

    pthread_create(&pth[0],NULL,bs,&temp[0]);
    pthread_create(&pth[1],NULL,bs,&temp[1]);

    pthread_join(pth[0],NULL);
    pthread_join(pth[1],NULL);

    t2=clock();

    printf("Binary Search:%lf\n",(t2-t1)/(double)CLOCKS_PER_SEC);

    if(temp[0].status!=-1)
        printf("Key1 found at first half\n");
    else
        printf("Key1 not found at first half\n");
    if(temp[1].status!=-1)
        printf("Key2 found at second half\n");
    else
        printf("Key2 not found at second half\n");
    //fib
}

void process4(int arr[],int n,int key1,int key2){
    clock_t t1,t2;
    pthread_t pth[2];

```

```

    struct node *temp=(struct node* )malloc(2*sizeof(struct node));
    temp[0].arr=arr;
    temp[0].start=0;
    temp[0].end=n/2;
    temp[0].key=key1;

    temp[1].arr=arr;
    temp[1].start=n/2;
    temp[1].end=n;
    temp[1].key=key2;

    t1=clock();

    pthread_create(&pth[0],NULL,fib,&temp[0]);
    pthread_create(&pth[1],NULL,fib,&temp[1]);

    pthread_join(pth[0],NULL);
    pthread_join(pth[1],NULL);

    t2=clock();

    printf("Fibonacci search:%lf\n", (t2-t1)/(double)CLOCKS_PER_SEC);

    if(temp[0].status!=-1)
        printf("Key1 found at first half\n");
    else
        printf("Key1 not found at first half\n");

    if(temp[1].status!=-1)
        printf("Key2 found at second half\n");
    else
        printf("Key2 not found at second half\n");
}

int main() {
    srand(time(0));
    int n = (rand() % 20 + 1) * 2;
    printf("n: %d\n", n);
    int arr[n];

```

```

    for (int i = 0; i < n; i++)
        arr[i] = rand() % 100 + 1;
    printf("Array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    // int n,i;
    // printf("Enter size of array: ");
    // int arr[n];
    // scanf("%d",&n);
    // printf("Enter %d numbers:\n",n);
    // for(i=0;i<n;i++)
    //     scanf("%d",&arr[i]);
    int key1,key2;

    printf("Enter key1 and key2 to be searched:\n");

    scanf("%d %d",&key1,&key2);
    bubblesort(arr,1,0,n/2);
    bubblesort(arr,1,n/2,n);
    printf("Elements after Sorting:\n");
    for(int i=0;i<n;i++)
        printf("%d ",arr[i]);
    printf("\n");

    process3(arr,n,key1,key2);
    process4(arr,n,key1,key2);

    return 0;
}

```

Output:



```

Sun Nov 1, 19:53:23
yoga@macbook-pro: ~/OS

File Edit View Search Terminal Help

yoga@macbook-pro:~/OS$ g++ -pthread binFib.cpp
yoga@macbook-pro:~/OS$ ./a.out
n: 20
Array: 87 58 46 4 97 61 81 91 99 71 41 77 92 45 85 91 77 19 6 48
Enter key1 and key2 to be searched:
19 77
Elements after Sorting:
4 46 58 61 71 81 87 91 97 99 6 19 41 45 48 77 77 85 91 92
Key 19 greater than the mid value 71
Key 19 greater than the mid value 46
Key 19 lesser than the mid value 4
Key 77 found!
Binary Search:0.001951
Key1 not found at first half
Key2 found at second half
Key 19 lesser than the min(offset+fibMMM2, n-1) value71
Key 19 lesser than the min(offset+fibMMM2, n-1) value46
Key 19 greater than the min(offset+fibMMM2, n-1) value4
Key 77 lesser than the min(offset+fibMMM2, n-1) value85
Key 77 greater than the min(offset+fibMMM2, n-1) value41
Key 77 greater than the min(offset+fibMMM2, n-1) value48
Key 77 Found
Fibonacci search:0.000846
Key1 not found at first half
Key2 found at second half
yoga@macbook-pro:~/OS$
```

### Note:

1. The input is randomised as per the question using a data creator( srand setup ).
2. Fibonacci performs much better than binary searching algorithms.

### Fibonacci and Binary Searchings with 4 Threads:

/\* All the passes not shown in the output !

Assumed it was enough to show the passes in the 2 Threads Setup \*/

## Code:

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

struct node
{
    int *arr;
    int start;
    int end;
    int key;
    int status;
};

int min(int x, int y) { return (x<=y)? x : y; }
/* Returns index of x if present, else returns -1 */
void *fib(void *arg) //void *arg
{
    struct node* temp=(struct node*)arg;
    int x=temp->key;
    int start=temp->start;
    int n=temp->end;
    int *arr=temp->arr;
    temp->status=-1;

    /* Initialize fibonacci numbers */
    int fibMMm2 = 0;    // (m-2)'th Fibonacci No.
    int fibMMm1 = 1;    // (m-1)'th Fibonacci No.
    int fibM = fibMMm2 + fibMMm1;    // m'th Fibonacci
    /* fibM is going to store the smallest Fibonacci
```

```

    Number greater than or equal to n */
while (fibM < n)
{
    fibMMm2 = fibMMm1;
    fibMMm1 = fibM;
    fibM = fibMMm2 + fibMMm1;
}
// Marks the eliminated range from front
int offset = start-1;
/* while there are elements to be inspected. Note that
we compare arr[fibMm2] with x. When fibM becomes 1,
fibMm2 becomes 0 */
while (fibM > 1)
{
    // Check if fibMm2 is a valid location
    int i = min(offset+fibMMm2, n-1);
    /* If x is greater than the value at index fibMm2,
    cut the subarray array from offset to i */
    if (arr[i] < x)
    {
        fibM = fibMMm1;
        fibMMm1 = fibMMm2;
        fibMMm2 = fibM - fibMMm1;
        offset = i;
    }
    /* If x is greater than the value at index fibMm2,
    cut the subarray after i+1 */
    else if (arr[i] > x)
    {
        fibM = fibMMm2;
        fibMMm1 = fibMMm1 - fibMMm2;
        fibMMm2 = fibM - fibMMm1;
    }
    /* element found. return index */
    if(arr[i]==x)
    {
        temp->status=i;
        break;
    }
}
}

```

```

    /* comparing the last element with x */
    if(fibMMm1 && arr[offset+1]==x)
        temp->status= offset+1;
    /*element not found. return -1 */
    pthread_exit(NULL);
}

void* bs(void *arg){
    struct node* temp=(struct node*)arg;
    temp->status=-1;
    while(temp->start<=temp->end){
        int mid=temp->start+(temp->end-temp->start)/2;
        if(temp->arr[mid]==temp->key)
            {temp->status=mid;break;}
        else if(temp->arr[mid]<temp->key)
            temp->start=mid+1;
        else
            temp->end=mid-1;
    }
    pthread_exit(NULL);
}

void swap(int *p,int *q){
    int temp=*p;
    *p=*q;
    *q=temp;
}

void bubblesort(int a[],int ch,int start ,int end){
    int i,j;
    for(i=start;i<end-1;i++)
    {
        for(j=start;j<end-(i-start)-1;j++)
        {
            int choice=(ch==0)?(a[j]<a[j+1]):(a[j]>a[j+1]);
            if(choice==1)
            {
                swap(&a[j],&a[j+1]);
            }
        }
    }
}

```

```

    }
    }
    }
}

void process3(int arr[],int n,int key1,int key2){
    clock_t t1,t2;
    pthread_t pth[4];
    struct node *temp=(struct node* )malloc(4*sizeof(struct node));
    temp[0].arr=arr;
    temp[0].start=0;
    temp[0].end=n/4-1;
    temp[0].key=key1;

    temp[1].arr=arr;
    temp[1].start=n/4;
    temp[1].end=n/2;
    temp[1].key=key1;

    temp[2].arr=arr;
    temp[2].start=n/2;
    temp[2].end=n/4+n/2-1;
    temp[2].key=key2;

    temp[3].arr=arr;
    temp[3].start=n/2+n/4;
    temp[3].end=n;
    temp[3].key=key2;

    t1=clock();

    for(int i=0;i<4;i++)
        pthread_create(&pth[i],NULL,bs,&temp[i]);

    for(int i=0;i<4;i++)
        pthread_join(pth[i],NULL);

    t2=clock();
}

```

```

printf("Binary Search:%lf\n", (t2-t1) / (double)CLOCKS_PER_SEC);

for(int i=0;i<4;i++){
    if(temp[i].status!=-1)
        printf("Key found at %d half\n",i+1);
    else
        printf("Key not found at %d half\n",i+1);
}
}

void process4(int arr[],int n,int key1,int key2){
    clock_t t1,t2;
    pthread_t pth[4];

    struct node *temp=(struct node* )malloc(4*sizeof(struct node));
    temp[0].arr=arr;
    temp[0].start=0;
    temp[0].end=n/4;
    temp[0].key=key1;

    temp[1].arr=arr;
    temp[1].start=n/4;
    temp[1].end=n/2;
    temp[1].key=key1;

    temp[2].arr=arr;
    temp[2].start=n/2;
    temp[2].end=n/4+n/2;
    temp[2].key=key2;

    temp[3].arr=arr;
    temp[3].start=n/2+n/4;
    temp[3].end=n;
    temp[3].key=key2;

    t1=clock();

    for(int i=0;i<4;i++)
        pthread_create(&pth[i],NULL,fib,&temp[i]);

```

```

    for(int i=0;i<4;i++)
        pthread_join(pth[i],NULL);

    t2=clock();

    printf("Fibonacci:%lf\n", (t2-t1)/(double)CLOCKS_PER_SEC);

    for(int i=0;i<4;i++){
        if(temp[i].status!=-1)
            printf("Key found at %d half\n",i+1);
        else
            printf("Key not found at %d half\n",i+1);
    }
}

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

    srand(time(0));
    int n = (rand() % 20 + 1) * 2;
    printf("n: %d\n", n);
    int arr[n];
    for (int i = 0; i < n; i++)
        arr[i] = rand() % 100 + 1;
    printf("Array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    int key1, key2;
    printf("Enter key1 and key2 to be searched:\n");
    scanf("%d %d", &key1, &key2);

    bubblesort(arr, 1, 0, n/4);
    bubblesort(arr, 1, n/4, n/2);
    bubblesort(arr, 1, n/2, (n/2+n/4));
    bubblesort(arr, 1, (n/2+n/4), n);
}

```

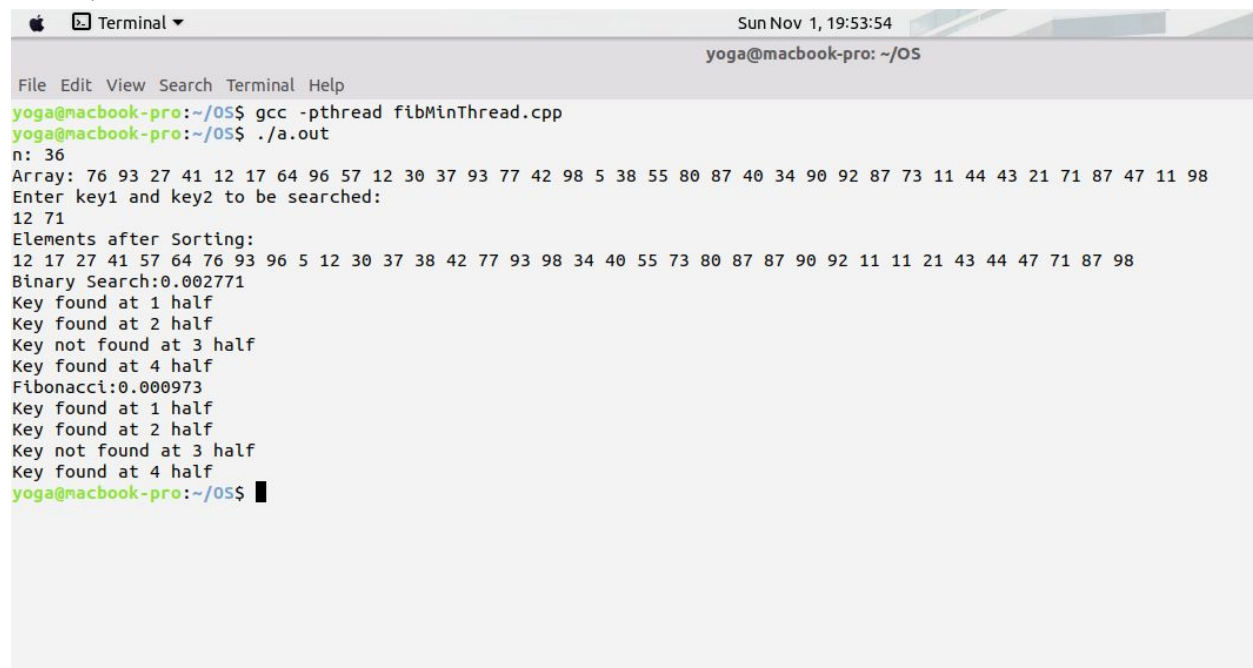
```

printf("Elements after Sorting:\n");
for(int i=0;i<n;i++)
    printf("%d ",arr[i]);
printf("\n");
process3(arr,n,key1,key2);
process4(arr,n,key1,key2);

return 0;
}

```

## Output:



A screenshot of a macOS Terminal window. The title bar shows 'Terminal' and the date/time 'Sun Nov 1, 19:53:54'. The user is 'yoga@macbook-pro' and the current directory is '~/OS'. The terminal shows the following output:

```

yoga@macbook-pro:~/OS$ gcc -pthread fibMinThread.cpp
yoga@macbook-pro:~/OS$ ./a.out
n: 36
Array: 76 93 27 41 12 17 64 96 57 12 30 37 93 77 42 98 5 38 55 80 87 40 34 90 92 87 73 11 44 43 21 71 87 47 11 98
Enter key1 and key2 to be searched:
12 71
Elements after Sorting:
12 17 27 41 57 64 76 93 96 5 12 30 37 38 42 77 93 98 34 40 55 73 80 87 87 90 92 11 11 21 43 44 47 71 87 98
Binary Search:0.002771
Key found at 1 half
Key found at 2 half
Key not found at 3 half
Key found at 4 half
Fibonacci:0.000973
Key found at 1 half
Key found at 2 half
Key not found at 3 half
Key found at 4 half
yoga@macbook-pro:~/OS$

```

## Note:

1. 4 Threaded setup performing poorly than the 2 threaded one.
2. Lot of test cases were tested and hence the conclusion.



### Varying the number of threads:

- It really depends on if/whether your threads are ever going to be waiting for i/o (disk access, network, etc.) and how long those waits will be versus the cost of switching threads.
- If your processor core is only ever going to be waiting on data from RAM, then it's unlikely the processor can switch threads faster than that.
- If on the other hand, the thread is going to be waiting on data from some network attached database, then the system might be able to switch threads and get something useful done while waiting for it. In that case it might be useful to have twice as many threads as cores.
- The best thing to do is try various numbers of threads and see what gives you the best performance. I'd try a couple of different test cases. If you have N cores, I'd try 1-thread, N-1 threads, N threads, N x 2 threads -- and see what gave you the best performance consistently. If N threads and N x 2 threads give similarly good results, try N x 1.5 and N x 3.
- I guess that's why my 2 threaded setup was better than the 4 threaded one !!!!

### Conclusion:

- Implemented both fibonacci and binary search algorithms in multiple threads - 2 && 4.

- Referred to online sources like GeeksforGeeks, stack overflow and other websites to understand key things.
- My Codes can be found in my Github repository - Link - <https://github.com/YogaVicky/OS-Assignments-Theory/tree/master/Midsem>

REALLY ENJOYED WRITING THE EXAM !