BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

CS-F211: Data Structures and Algorithms

Lab 6: Quicksort and its variations

Introduction

Welcome to week 6 of Data Structures and Algorithms! We have been looking at a variety of sorting algorithms over the past few weeks. We started off by looking at $O(n^2)$ sorting algorithms like Insertion Sort. Then we discussed Merge Sort which is better at $O(n \lg n)$ in the worst case. Today we shall be looking at Quicksort. The quicksort algorithm has a worst-case running time of $\Theta(n^2)$ on an input array of n numbers. Despite this slow worst-case running time, quicksort is often the best practical choice for sorting because it is remarkably efficient on average. Its expected running time is $O(n \lg n)$, and the constant factors hidden in the $O(n \lg n)$ notation are quite small.

Topics to be covered in this lab sheet

- Introduction to Quicksort
    - Partitioning
    - Pivot Selection
- Converting Recursive Quick Sort to Iterative
- Hybrid Quicksort: Quicksort with Insertion sort

Introduction to Quicksort

Quicksort is a divide-and-conquer algorithm for sorting a typical subarray **A[lo .. hi]:**

1. Divide: Partition (rearrange) the array **A[lo .. hi]** into two (possibly empty) subarrays **A[lo .. piv - 1]** and **A[piv + 1 .. hi]** such that each element of **A[lo .. piv - 1]** is less than or equal to **A[piv]**, which is, in turn, less than or equal to each element of **A[piv + 1 .. hi]**. Compute the index **piv** as part of this partitioning procedure.
2. Conquer: Sort the two subarrays **A[lo .. piv - 1]** and **A[piv + 1 .. hi]** by recursive calls to quicksort.
3. Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array **A[lo .. hi]** is now sorted.

Thus, it depends on two major sub-problems — pivot selection and partitioning around the pivot. Let us look at each of them in detail.

Partitioning

Given a pivot element, partitioning is the problem of rearranging the elements into two contiguous parts such that elements in the first part are less than or equal to the pivot while the elements in the second part are strictly greater than it. The partition algorithm does not care about the ordering of the individual elements within the two parts.

Consider the example illustrated in Fig 1.

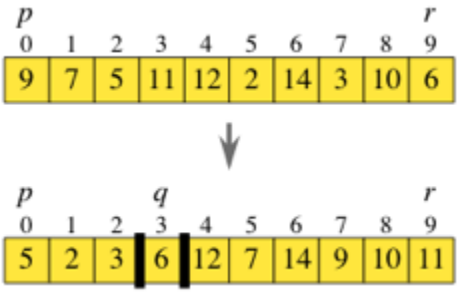

Fig 1. An array being partitioned around the pivot 6

Here, the array of integers is shown which is then partitioned around the pivot element 6. After the partition operation, all elements that are preceding pivot 6 in the array are now smaller than (or equal to) 6, and all elements that appear after the pivot are greater than 6.

Let us look at how to perform this operation now. Consider the same array as shown in Fig 1 before partitioning. Now, try to think of how we could partition it. One might consider having two auxiliary arrays **LT** and **GT**. We could traverse the array and for each element, if it belongs to the left partition, we could copy it to the array **LT** and if it belongs to the right partition, we

could copy it to **GT**. After putting each element in one of these arrays, we can simply copy the elements in **LT** followed by those in **GT** back to the original array.

Now the above approach suffers from the problem of using extra auxiliary space. While the time complexity of this approach is `O(n)`, its space complexity is also `O(n)`. Also, here we end up copying every single element twice even if it is possible that the original location it was at is already less than the pivot. Let us instead take a look at an algorithm that gets rid of these redundant copies and auxiliary space usage.

Hoare Partitioning

We shall now learn the Hoare Partitioning Scheme. The Hoare partitioning scheme works on the intuition that if while scanning an array from left to right, we encounter an element that belongs to the right partition, there must be a corresponding element that is currently ahead of the correct pivot location but is less than the pivot element. So, these two elements can be swapped.

The algorithm thus uses two pointers, one starting from the leftmost element and the other from the rightmost element, to scan the array. The left pointer moves right until it finds an element that is greater than the pivot, and the right pointer moves left until it finds an element that is less than or equal to the pivot. The two elements are then swapped, and this process is repeated until the pointers meet in the middle.

The Hoare partitioning algorithm can be implemented as follows:

```
// Ls[lo..hi] is the input array; Ls[pInd] is the pivot
int part(int Ls[], int lo, int hi, int pInd)
{
    swap(Ls, pInd, lo);
    int pivPos, lt, rt, pv;
    lt = lo + 1;
    rt = hi;
    pv = Ls[lo];
    while (lt < rt)
    {
        for (; lt <= hi && Ls[lt] <= pv; lt++);
        // Ls[j]<=pv  for j in lo..lt-1
        for (; Ls[rt] > pv; rt--);
        // Ls[j]>pv  for j in rt+1..hi
        if (lt < rt)
        {
```

```
            swap(Ls, lt, rt);
            lt++;
            rt--;
        }
    }
    if (Ls[lt] < pv && lt <= hi)
        pivPos = lt;
    else
        pivPos = lt - 1;
    swap(Ls, lo, pivPos);
    // Postcond.: (Ls[j]<=pv for j in lo..pivPos-1) and (Ls[j]>pv for j in
pivPos+1..hi)
    return pivPos;
}
```

The above function partitions the array into two halves and returns the index of the pivot in the partitioned array.

Lomuto Partitioning

There is another `O(n)` technique called the Lomuto partitioning scheme. In Lomuto partitioning, a single pointer is used to iterate over the array from left to right. Initially, the pivot is swapped with the element at the last location. As the pointer moves from left to right, it keeps track of the correct position of the pivot based on the elements swept past till now and swaps any element that is less than or equal to the pivot to the left of the pivot position. Once the pointer reaches the end of the array, the pivot is swapped with the element at the pivot position, effectively partitioning the array into two parts.

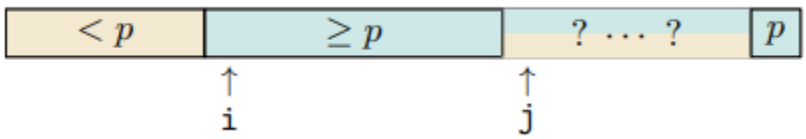

Fig 2. Lomuto Partitioning

Lomuto's partitioning scheme uses two additional variables **i** and **j** and maintains the invariant (conditions) displayed in Figure 2. The variable **j** is incremented from **1** to **n−1** using a for-loop. When **A[j]** is inspected, it is compared to the pivot **p**. If it is smaller than the pivot, **A[i]** and **A[j]** are swapped, and **i** is incremented. **A[0 ... i − 1]** consists of elements smaller than **p**, **A[i ... j −**

1] consists of elements at least as large as **p**; **A[j ... n − 2]** has not been looked at and **A[n-1]** has the pivot.

Three-Way Partitioning

Three-way partitioning is one where we create three partitions — less than the pivot, equal to the pivot and greater than the pivot. This ensures that when we have multiple occurrences of the pivot element, all of them are bunched together and we get two partitions — one strictly less than the pivot and the other strictly lesser than the pivot.

The three-way partitioning version of Hoare's partitioning algorithm is given as follows:

```
int threePart(int Ls[], int lo, int hi, int pInd)
{
    swap(Ls, pInd, hi - 1);
    int pivPos, lt, rt, mid, pv;
    lt = lo;
    rt = hi - 2;
    mid = lo;
    pv = Ls[hi - 1];
    while (mid <= rt)
    {
        if (Ls[mid] < pv)
        {
            swap(Ls, lt, mid);
            lt++;
            mid++;
        }
        else if (Ls[mid] > pv)
        {
            swap(Ls, mid, rt);
            rt--;
        }
        else
        {
            mid++;
        }
    }
    swap(Ls, mid, hi - 1);
    return mid;
}
```

Consider the following illustration:

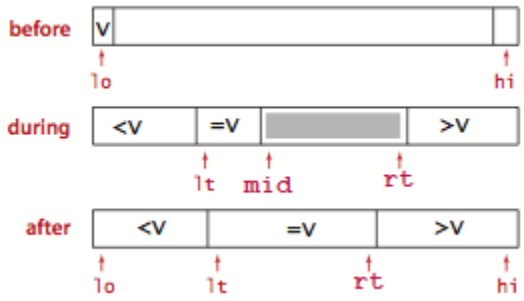

Fig 3. 3-way partitioning in action

In Fig 3, the grey portion shows the unsorted section of the array. As **lt** and **mid** move rightward and **rt** moves leftwards while maintaining the condition shown in the figure. When **mid** meets **rt**, the array is now partitioned into three partitions.

Task 1: Implement both two-way and three-way versions of Hoare partitioning as well as Lumotu partitioning for partitioning elements of struct Person based on the height. The struct is defined as:

```
struct person
{
    int id;
    char *name;
    int age;
    int height;
    int weight;
};
```

Read the file dat1000.csv (as in Lab 4) into three different arrays and compare the three algorithms in terms of the time taken to partition it for a few random indices as pivots. [We have seen that all three of these algorithms have an asymptotic running time of **O(n)**. But still one of these would be running faster than the other two because of the constant terms present in the time complexity.]

Task 2: You are given an array of 0s and 1s in random order. Segregate 0s on the left side and 1s on the right side of the array [Basically you have to sort the array]. Traverse the array only once.

Home Exercise 1: Consider the following structure definition:

```
typedef struct bitsian
{
    char name[20];
    char rollno[15];
}bitsian;
```

Example B.E. roll numbers are "2020A7PS0081P" and “2020A7PS0013P” where 2020 refers to the year of joining, A7 is the branch code, PS or TS refers to whether the student is opting for Practice School or Thesis, 0081 (or 0013) is the student’s ID and P refers to the campus (P - Pilani, G - Goa, H - Hyderabad, D - Dubai). An example M.Sc. roll number is “2019B5A19876P”, where 2019 refers to the year of joining, B5 is the primary branch code, A1 is the dual branch code, 9876 is the student's ID and P again refers to the Pilani campus.

A branch code beginning with A is for B.E. students and one beginning with B is for M.Sc. students. However, in the first year, the dual degree hasn't been allotted to the M.Sc. students yet and they still have "PS" in the corresponding field.

In the first year, courses are common for all branches, but the students are divided into two groups based on which courses they would be doing in the first and second semesters respectively.

This division is based on the branch code.

Suppose that in some particular year, students from {A1, A2, …, A8, A9} are in the first group and students from {AA, AB, B1, …, B5} are in the second group. You have been given the details of students in a file named "bitsians.csv" in the following format:

```
N
rollno1, name1
rollno2, name2
...
rollnoN, nameN
```

Write a C program to read the details of the students from the file and store them in an array of structures. Now, divide the students into two groups based on which group of courses they would be doing in the first semester as described above. Your algorithm should run in $O(n)$ time and have an auxiliary space complexity of $O(1)$ apart from the array of structures holding the original student data (which should **not** be modified by your algorithm).

Pivot Selection

Pivot selection refers to the problem of selecting a pivot around which you would partition the array into sub-arrays. In principle, any pivot would be correct and still lead to a sorted array. However, the choice affects the time complexity of the quicksort algorithm. We ideally want balanced partitions to recurse into.

In the extreme case of unbalanced partitions (1 : n-1), we end up with a complexity of $O(n^2)$ for quicksort. The intuition behind this is that in each iteration exactly one element is placed in it correct spot (the partition of size 1). So the number of iterations required to sort the array would be n and the complexity of each iteration is that of the partition algorithm, ie., O(n). So, the worst-case complexity of quicksort is $O(n^2)$ if we are not able to select a "good" pivot.

As we saw earlier, a desirable quality of the pivot is that it should create approximately equal-sized partitions each time. Thus the problem of selecting a good pivot reduces to finding the median of the array. Let us look at various pivot selection techniques.

Selecting the first (or, equivalently, the last) element

This technique would work well if the input array is randomised. However, suppose that the input is nearly sorted in either ascending or descending order — one of the partitions would end up being of size 1. This gives us our worst-case time complexity of **$O(n^2)$**, which is not desirable. The more unbalanced each pivot choice is, the worse our performance gets. Can we do better?

Median-of-Three

'Median-of-three' is a technique where we take the median of the first, middle, and last elements and use it as our pivot. This tackles the problem of nearly sorted arrays. So, we do get the desired complexity of quicksort as $O(n \lg n)$. While the partitions are in general not perfectly balanced, the overhead of computing the pivot in this technique is negligible leading it to be used at times in practice.

Fig 4. Illustrating Median-of-Three

As illustrated in fig 4. we first find the median of the first, middle and last elements, which are 2, 13, and 6 respectively (of which, the median is 6), and then choose that median as the pivot.

Median-of-Medians

'Median-of-medians' is one technique which can be used to find the k^{th} smallest element in an array. In the median of medians algorithm, we divide the input arrays first into groups of 5. Since 5 is a small constant number, for a given list of 5 integers, we can find the median of the list in $O(1)$ time. Now, we need to find the median of these n/5 medians which you can do by recursing into the list of n/5 elements. It can be mathematically proven (and students taking the course Design and Analysis of Algorithms in future would be doing so themselves) that this algorithm has a time complexity of $O(n)$[1].

```
// L[] is the input array of length n
// kth smallest element is returned
int select(int L[], int n, int k)
{
    if (k == 0)
        return L[0];
    if (n <= 5){
        for (int i = 1; i < n; i++)
            for (int j = i-1; j >= 0; j--)
                if (L[j] > L[j+1])
                    swap(L, j, j+1);
                else
                    break;
        return L[k-1];
```

[1] For more details, students may refer to section 9.3 of Cormen T.H., Leiserson, C.E., Rivest, R.L., and C. Stein. *Introduction to Algorithms*, MIT Press, 3rd Edition, 2009.

```
    }

    // partition L into subsets of five elements each
    //     (there will be n/5 subsets total).

    int numGroups;
    if (n % 5 == 0)
        numGroups = n/5;
    else
        numGroups = n/5 + 1;

    int medians[numGroups];

    for (int i = 0; i < numGroups; i++)
    {
        medians[i] = select(L + i*5, min(5, n - i*5), min(5, n - i*5)/2);
    }

    int M = select(medians, numGroups, (numGroups+1)/2)

    // Partition array into two halves, L1, {M} and L2, such that
    // L1 contains elements <= M, {M} contains one instance of M and L2
contains all elements > M
    int mInd;
    for (int i = 0; i < n; i++)
    {
        if (L[i] == M)
        {
            mInd = i;
            break;
        }
    }

    int pInd = part(L, 0, n-1, mInd);

    if (k <= pInd)
        return select(L, pInd, k);

    else if (k > pInd + 1)
        return select(L + pInd + 1, n - pInd - 1, k - pInd - 1);
```

```
    else
        return L[pInd];
}
```

Quickselect

Quickselect is an algorithm used to find the k^{th} smallest element in an array. It is very similar to quicksort in that it chooses a pivot and partitions the array based on it. However, in quickselect, we recurse into only one partition which would contain the required element. The basic idea behind QuickSelect is to use the partitioning step from QuickSort to divide the array into two parts: elements smaller than (or equal to) the pivot and elements larger than the pivot. By selecting the pivot carefully, it's possible to eliminate a large portion of the array without having to sort it. Thus, we end up with an average time complexity of `O(n)` for quickselect and not the `O(n lg n)` the we had obtained in the case of quicksort.

The algorithm for quickselect is given as follows:

```
// L[] is the input array of length n
// kth smallest element is returned
int qselect(int L[], int n, int k)
{
    int pivot = 0;
    int lo = 0;
    int hi = n - 1;
    int pInd = part(L, lo, hi, pivot);
    if (k <= pInd)
        return qselect(L, pInd, k);
    else if (k > pInd + 1)
        return qselect(L + pInd + 1, n - pInd - 1, k - pInd - 1);
    else
        return L[pInd];
}
```

Random Pivot

Here, we select the pivot index uniformly randomly between the first and last indices. This is another way to avoid the worst case of sorted arrays that we had with picking the first element. This technique may not always lead to extremely balanced arrays, but it saves the overhead of computing the pivot.

Generating truly random numbers is a very difficult computational task. In practice, we generally use a pseudo-random generator, which approximates the properties of random numbers to a good extent. In C, random integers are generated using the **`rand()`** function of the `<stdlib.h>` header file. The C library function `int rand(void)` returns a pseudo-random number in the range of `0` to `RAND_MAX`. `RAND_MAX` is a constant whose default value may vary between implementations but it is granted to be at least `32767`. The number can be restricted to a range by using the modulo operation on the integer returned by the function.

A random seed (or seed state, or just seed) is a number used to initialise a pseudorandom number generator. A pseudorandom number generator's number sequence is completely determined by the seed: thus, if a pseudorandom number generator is reinitialized with the same seed, it will produce the same sequence of numbers. In C, the **`rand()`** function is seeded through the **`srand()`** function.

If we do not seed the random number generator with a new value every run, it would produce the same output every time it is run. Thus it is a common practice to use the current time as the seed as it would vary the numbers generated every time the program is run.

Consider the following example illustrating the use of the `rand()` and `srand()` functions to generate a random number between two integers provided by the user.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main()
{
    int a, b;
    printf("Enter a and b: ");
    scanf("%d %d", &a, &b);
    srand(time(NULL));
    int r = rand() % (b - a + 1) + a;
    printf("Random number between %d and %d is %d\n", a, b, r);
    return 0;
}
```

Now, this same technique can be used to select a pivot randomly with just the following lines of code:

```c
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <sys/time.h>
int randomPivot(int lo, int hi);
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    srand(tv.tv_usec * 1000000 + tv.tv_sec);
);
    return (rand() % (b - a)) + a;
}
```

Note, both the libraries `<time.h>` and `<sys/time.h>` can be used to access the current time in different ways as illustrated in the above two examples.

Home Exercise 2: Let S be an (unsorted) array of n distinct real numbers and let k ≤ n be a positive integer (k may not be a constant, ie., it can be in terms of n as well). Write an algorithm, running in `O(n)` time, that determines the k numbers in S that are closest to the median of S. [Hint: You might have to apply multiple `O(n)` operations on the array. While there are no explicit space restrictions for this instance of the problem, a solution with `O(1)` additional space complexity does exist].

Example 1:
S = [1,3,5,7,9], k = 3
Here, the median is 5. So the 3 closest elements to 5 are 3, 5 and 7.

Example 2:
S = [2,3,9,10,11,12], k = 4
Here, the median is 9.5. The 4 elements closest to the median are 9, 10, 11 and 12.

Putting it all together

Now that we have seen various pivot selection and partitioning techniques, we are now equipped to implement quicksort. The template for the same is given as follows:

```c
void qs(int Ls[], int lo, int hi)
{
    if (lo < hi)
    {
        int p = pivot(Ls, lo, hi);    // Ls[p] is the pivot
        p = part(Ls, lo, hi, p); // Ls[p] is the pivot
```

```
        /*
        (Ls[j]<=Ls[p] for j in lo..pPos-1) and
        (Ls[j]>Ls[p] for j in pPos+1..hi)
        */
        qs(Ls, lo, p - 1);
        qs(Ls, p + 1, hi);
    }
}
```

Obviously, here we can use any of the pivot selection and partition techniques as described above for the implementation of pivot() and partition() method.

Task 3: Implement quicksort with the best partitioning scheme obtained in Task 1 and with each of the pivot selection mechanisms described above for sorting integers. Compare the running time of the quicksort algorithm when implemented with the different pivot selection techniques for the files of the form *intX.csv* provided to you.

Is there a technique that works universally well for each of the files? You are encouraged to go through the files yourselves to understand why a particular technique may or may not perform well on a particular file.

Recursion to Iteration

We saw last week how tail recursion can be eliminated and the advantages of iterative algorithms over recursive versions in terms of saving call stack space. Each recursive call results in a new function call frame being pushed to the call stack. This call stack frame contains a lot of information and there is a huge overhead in storing all of this information. In order to avoid this, we convert our recursive programs into iterative ones.

Let us look at our initial recursive quicksort implementation first:

```
void qs(int Ls[], int lo, int hi)
{
    if (lo < hi)
    {
        int p = pivot(Ls, lo, hi);   // Ls[p] is the pivot
        p = part(Ls, lo, hi, p); // Ls[p] is the pivot
        /*
        (Ls[j]<=Ls[p] for j in lo..pPos-1) and
        (Ls[j]>Ls[p] for j in pPos+1..hi)
        */
        qs(Ls, lo, p - 1);
        qs(Ls, p + 1, hi);
    }
}
```

Here, there are two recursive calls to the function qs(). If we look at the second occurrence of call, we can appreciate that it is a tail call. That is the last line executed in the function. Thus we can eliminate this call using the methods described in the previous lab. We simply update the actual parameters of the function and loop its execution iteratively instead of recursively passing new parameters.

The result of this technique would yield the following implementation now:

```
void qsort_iter_attempt1(int Ls[], int lo, int hi)
{
    while (lo < hi)
    {
        int pInd = pivot(Ls, lo, hi);
        int p = part(Ls, lo, hi, pInd);
        qsort(Ls, lo, p - 1);
        lo = p + 1;
    }
```

```
}
```

So we have achieved some progress. Instead of the two recursive calls to qs(), now we have reduced it to only one. Let us try to remove this call as well..

This cannot be done as easily as it is not a tail-recursive call nor can it be converted into one. Thus, the only way to eliminate this is by using a method that was briefly hinted at in the previous labsheet — using an **explicit stack**. In this method, we essentially write explicit code to simulate the recursive call stack. A new stack is created, the function parameters are pushed onto this new stack (instead of a recursive call), and the function body is executed in a loop with the parameters on the top of the stack until the stack becomes empty.

When we apply this method on qsort_iter_attempt1(), we get the following implementation:

```
void qs_iter_attemp2(int ls[], int lo, int hi)
{
    Stack *s = newStack();
    Element ele = {lo, hi};
    push(s, ele);
    while (!isEmpty(s))
    {
        Element e = *top(s);
        pop(s);
        lo = e.lo;
        hi = e.hi;
        while (lo < hi)
        {
            int p = pivot(ls, lo, hi);
            p = part(ls, lo, hi, p);
            push(s, (Element){lo, p - 1});
            lo = p + 1;
        }
    }
}
```

Here, we have used the linked_list implementation of the stack from lab sheet 3, where the only change we have made is to the definition of the **struct Element** which instead of consisting of an integer **int_value** and a float **float_value**, now consists of two integers **lo** and **hi**.

Thus, we have converted quicksort to a completely iterative algorithm now using both tail-call elimination and an explicit stack.

Many times in practice (and this applies to programs in general), the processor may not be able to yield enough memory to store all the call frames in a recursive code. In such cases, the explicit stack method comes to the rescue. And even if the explicit stack cannot fit in memory, we could simply maintain the stack in a file when dealing with humongous amounts of data. While this would slow down the algorithm quite a bit (as file IO is highly expensive), sometimes it is inevitable.

Home Exercise 3: In this section, the pivot selection could have been any of the techniques described formerly. If you look at the algorithms for median-of-medians and quickselect, you would realise that these algorithms are also recursive in nature. Convert these algorithms to their iterative versions using the techniques described in this section.

Hybrid Quicksort

Let us revisit the median-of-medians algorithm. You would have noticed that for arrays of size less than 5, we had sorted the array using insertion-sort and then selected the median. Would this have been more efficient if we had used an asymptotically better algorithm like quicksort or mergesort?

The answer surprisingly is no. For small arrays, insertion sort is seen to perform *better* than most other sorting algorithms. (Is this a violation of asymptotic complexity?)

Still, for larger arrays, the $O(n \lg n)$ factor in quicksort does beat the $O(n^2)$ factor of insertion sort. So we try to explore the question: is there some way to get the best of both worlds here? The answer to this is **hybrid quicksort**.

Here, we start with the usual implementation of recursive quicksort. However, when the subarrays become smaller than a fixed size, insertion sort is called on them instead of quicksort. An implementation of this algorithm is given below:

```
void qsort_hybrid(int Ls[], int lo, int hi)
{
    if (hi - lo < 10)
    {
        insertionsort(Ls, lo, hi);
        return;
    }
    else if (lo < hi)
    {
        int p = pivot(Ls, lo, hi);
        p = part(Ls, lo, hi, p);
        qsort_hybrid(Ls, lo, p - 1);
        qsort_hybrid(Ls, p + 1, hi);
    }
}
```

While we have chosen 10 as the value for the threshold here, you are encouraged to experiment with different threshold values to see which beats the vanilla quicksort best.

Task 4: There is another way of designing a hybrid of quicksort and insertion sort. In this method, instead of running insertion sort on arrays smaller than a threshold, those arrays are simply left unsorted by quicksort. After the entire run of quicksort is completed, now insertion

sort is run on the entire array. This algorithm also works well because insertion sort has a good performance on nearly sorted arrays.

Implement this version of hybrid-quicksort and compare it with the quicksort_hybrid() function implemented in the above section. Compare the performance of both algorithms on the *intX.txt* set of files provided to you.

Task 5: You compared the space and time efficiencies of merge sort and insertion sort on the set of *datX.csv* files in lab4. Now, on the same set of files, also compare (by plotting graphs) for the running time of the following algorithms against the input size:

1. Recursive Quicksort using two-way Hoare partitioning and random pivot selection
2. Recursive Quicksort using Lomuto partitioning and median-of-three pivot selection
3. Iterative Quicksort using two-way Hoare partitioning and iterative median-of-medians pivot selection
4. Hybrid Quicksort using three-way Hoare partitioning and recursive quickselect

Observe the difference in the time taken between merge sort and the different implementations of quicksort in spite of them all `O(n lg n)` on average.

Home Exercise 4: The BITS library has a mammoth collection of books on a wide variety of subjects. You have been provided with a sizable subset of the BITS library database[2] (as on February 2023!) "*BITS Library Database.csv*". Write a program to read the different fields from the CSV file and sort the database based on the **callnumber** field. Note that the call number must be read as a string and not a number.

Note that not all fields are present in all the records. **If a record does not contain a call number, skip that record while reading the input file.** For the other fields, you may consider an appropriate default value for records where the particular field is missing. You can explore the different fields that are present by going through the file yourselves.

Create a structure that can store all the fields included in the file and read the file into a dynamically allocated array of this structure. Implement hybrid quicksort (any pivot selection and partitioning technique you want) to sort the array lexicographically based on the call number. Now write the records in this sorted order to a new file called *sorted_books.csv*.

[2] Special thanks to the librarian Dr. Rajan Sinha Thakur for providing access to the database.