Computing Laboratory

Sorting and Searching Techniques

Malay Bhattacharyya

Associate Professor

MIU, CAIML, TIH
Indian Statistical Institute, Kolkata

November, 2023

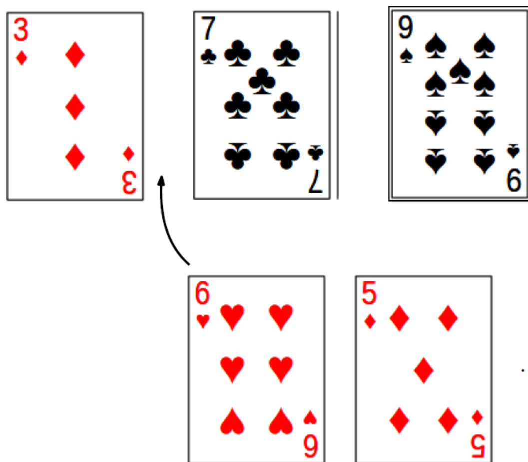
1 Sorting Techniques
- Bubble sort
- Insertion sort
- Merge sort
- Heap sort
- Bucket sort
- Practical implementation

2 Searching Techniques
- Basics
- Searching on Linear Data Structures
- Searching on Nonlinear Data Structures
- Other Searching Problems

Bubble sort

```
for(i = 0; i < n-1; i++)
    for(j = 0; j < n-i-1; j++)
        if(A[j+1] < A[j])
            swap(A, j, j+1);
```

Insertion sort

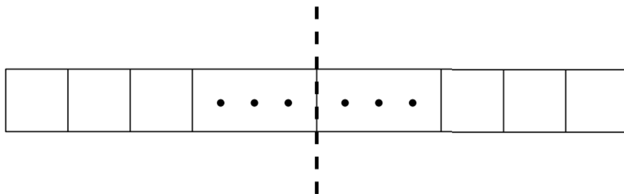


Insertion sort

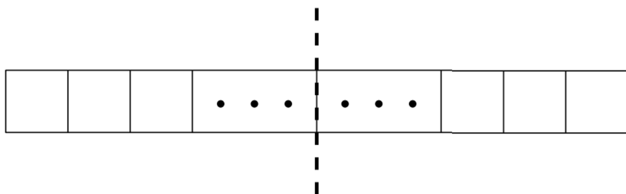
Insertion sort


```
for(i = 1; i < n; i++){
    key = A[i];
    /* Find the right place for A[i] in A[0 ... i-1] */
    for(j = i-1; j >= 0 && A[j] > key; j--)
        A[j+1] = A[j];
    A[j+1] = key;
}
```

Merge sort



Merge sort



```
void msort(int *A, int beginning, int end){
     if(beginning < end){
        middle = (beginning + end) / 2;
        msort(A, beginning, middle);
        msort(A, middle+1, end);
        merge(A, beginning, middle, end);
     }
}
```

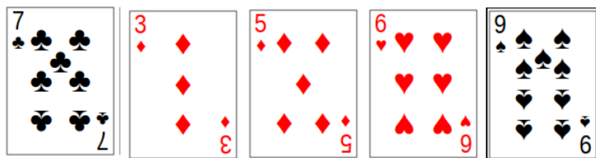
Merge sort

```
void merge(A, b, m, e){
     int i, j, k;
     /* Allocate space for auxiliary array B */
     for(i = b, j = m+1, k = 0; i <= m && j <= e; )
         if(A[i] < A[j])
            B[k++] = A[i++];
         else if(A[i] > A[j])
            B[k++] = A[j++];
         else
            B[k++] = A[i++], B[k++] = A[j++];
     while(i <= m) B[k++] = A[i++];
     while(j <= e) B[k++] = A[j++];
     assert(...);
}
```

Heap sort

```
void heapsort(void *a, int N, size_t element_size,
            int (*comparator)(void *, int, int)){
    int k;
    HEAP h;
    h.element_size = element_size;
    h.num_allocated = h.num_used = N;
    h.array = a;
    h.comparator = comparator;
    /* Make heap out of array */
    for(k = N/2; k >= 1; k--)
        swapDown(&h, k);
    /* Sort by successive deleteMax */
    while(h.num_used > 1){
        swap(&h, 1, h.num_used); // move max to end
        h.num_used--;
        swapDown(&h, 1);
    }
}
```

Bucket sort



Spades				Hearts				Diamonds				Clubs			
2	3	...	Ace	2	3	...	Ace	2	3	...	Ace	2	3	...	Ace

Bucket sort

```
/* Allocate space for array A */
for(i = 0; i < n; i++){
    /* Scan the value in i
    A[i] = i;
}
```

Let us recall: function pointers

- Declaring function pointers

```
<return type> (* <function name>) ( <parameter list> )
```

  The brackets around the function name are important!!!
  Example:

```
int *aFunction(int), *(*aFunctionPointer)(int);
```

- Using function pointers

```
(*f)(...)
```

- Setting function pointer variables / passing function pointers as arguments: simply use the name of the function
  Example:

```
aFunctionPointer = aFunction;
```

Let us recall: void pointers

- Pointers to a generic chunk of memory
- Programmer needs to know the actual type in order to do anything useful
- Useful for writing generic (type independent) code

```
char c, *cp;
char *str, **sp;
int i, *ip;
float f;

void *voidPointer;
```

```
voidPointer = (void *) &c;
* ((char *) voidPointer) = 'A';
printf("%c\n", c);

voidPointer = (void *) &i;
* ((int *) voidPointer) = 10;
printf("%d\n", i);
```

Generic sort routine

```
#include<stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Comparator routine: examples

```
int compare_int (void *elem1, void *elem2){
    int *ip1 = elem1;
    int *ip2 = elem2;
    return *ip1 - *ip2;
    /* Or more explicitly:
       int i1 = *((int *) elem1);
       int i2 = *((int *) elem2);
       return i1 - i2;
     */
}

int compare_strings (void *elem1, void *elem2){
    char **s1 = elem1; // Alt.: char *s1 = *((char **) elem1);
    char **s2 = elem2; // Alt.: char *s2 = *((char **) elem2);
    return strcmp (*s1, *s2);   // Alt.: return strcmp(s1, s2);
}
```

Using qsort

```
char **strings;
int *a;
int num_strings, N;

qsort(a, N, sizeof(int), compare_int);
qsort(strings, num_strings, sizeof(char *),
                                    compare_strings);
```

Timsort

Timsort is a highly optimized and stable version of merge sort that effectively combines insertion sort.

Timsort was implemented by Tim Peters in 2002 for use in the Python programming language.

- https://en.wikipedia.org/wiki/Timsort
- https://realpython.com/sorting-algorithms-python/#pythons-built-in-sorting-algorithm

Timsort

Input: Unsorted collection
Output: Sorted collection

```
// Calculate run length
runLength := calculateRunLength(array);
// Perform insertion sort on each run
for start ← 0 to array.length by runLength do
    end := min(array.length − 1, start + runLength − 1);
    insertionSort(array, start, end);
end
// Recursively merge adjacent runs
mergeSize := runLength;
while mergeSize < array.length do
    for left ← 0 to array.length by size ∗ 2 do
        mid := left + size − 1;
        right := min(array.length − 1, left + (2 ∗ size) − 1);
        if mid < right then
            merge(array, left, mid, right);
        end
    end
    mergeSize := mergeSize ∗ 2;
end
```

The searching problem

Given a data structure S over a domain of data items D, verify (search) whether a given data item s belongs to D or not. If it belongs, then return the position (index) of s in D with respect to its organization in S.

Searching on Linear Data Structures

- Linear search
- Jump search
- Binary search
- Interpolation search
- Fibonacci search

Searching on Linear Data Structures

Searching Method	On Arrays	On Linked Lists
Linear	Possible	Possible
Binary	Possible	Only possible with alternative implementation (no deletion applied)
Jump	Possible	Only possible with alternative implementation (no deletion applied)
Interpolation	Possible	Only possible with alternative implementation (no deletion applied)
Fibonacci	Possible	Possible with alternative implementation (no deletion applied)

Linear search

Let $LIST[] = \{30, 10, 20, 40, 50\}$ and $s = 40$.

Linear search

Let $LIST[] = \{30, 10, 20, 40, 50\}$ and $s = 40$.

$i = 0$	$\downarrow$					
Whether $LIST[i] = s$	30	10	20	40	50	FALSE

Linear search

Let $LIST[] = \{30, 10, 20, 40, 50\}$ and $s = 40$.

$i = 0$	↓					
Whether $LIST[i] = s$	30	10	20	40	50	FALSE
$i = i + 1$		↓				
Whether $LIST[i] = s$	30	10	20	40	50	FALSE

Linear search

Let $LIST[] = \{30, 10, 20, 40, 50\}$ and $s = 40$.

$i = 0$	$\downarrow$					
Whether $LIST[i] = s$	30	10	20	40	50	FALSE
$i = i + 1$		$\downarrow$				
Whether $LIST[i] = s$	30	10	20	40	50	FALSE
$i = i + 1$			$\downarrow$			
Whether $LIST[i] = s$	30	10	20	40	50	FALSE

Linear search

Let $LIST[] = \{30, 10, 20, 40, 50\}$ and $s = 40$.

$i = 0$	$\downarrow$					
Whether $LIST[i] = s$	30	10	20	40	50	FALSE
$i = i + 1$		$\downarrow$				
Whether $LIST[i] = s$	30	10	20	40	50	FALSE
$i = i + 1$			$\downarrow$			
Whether $LIST[i] = s$	30	10	20	40	50	FALSE
$i = i + 1$				$\downarrow$		
Whether $LIST[i] = s$	30	10	20	40	50	TRUE

Return $i = 3$.

Linear search

```
int LinearSearch(int *LIST, int SIZE, int s){
    int i;
    for(i=0; i<SIZE; i++)
        if(*(LIST + i) == s)
            return i; // Data is at the index i
    return; // Data is nowhere
}
```

Jump search

Let $LIST[] = \{10, 20, 30, 40, 50\}$ and $s = 40$. Further assume that the block size (of jump) is $b = 2$.

Jump search

Let $LIST[] = \{10, 20, 30, 40, 50\}$ and $s = 40$. Further assume that the block size (of jump) is $b = 2$.

$pre_i = 0, i = b$			↓			
Whether $LIST[i] < s$	10	20	30	40	50	TRUE

Jump search

Let $LIST[] = \{10, 20, 30, 40, 50\}$ and $s = 40$. Further assume that the block size (of jump) is $b = 2$.

$pre_i = 0, i = b$			↓			
Whether $LIST[i] < s$	10	20	30	40	50	TRUE

$pre_i = i, i = i + b$					↓	
Whether $LIST[i] < s$	10	20	30	40	50	FALSE

Jump search

Let $LIST[] = \{10, 20, 30, 40, 50\}$ and $s = 40$. Further assume that the block size (of jump) is $b = 2$.

Step						
$pre_i = 0, i = b$			↓			
Whether $LIST[i] < s$	10	20	30	40	50	TRUE
$pre_i = i, i = i + b$					↓	
Whether $LIST[i] < s$	10	20	30	40	50	FALSE
$pre_i = pre_i + 1$				↓		
Whether $LIST[pre_i] = s$	10	20	30	40	50	TRUE

Jump search

Let $LIST[] = \{10, 20, 30, 40, 50\}$ and $s = 40$. Further assume that the block size (of jump) is $b = 2$.

Step						
$pre_i = 0, i = b$			↓			
Whether $LIST[i] < s$	10	20	30	40	50	TRUE
$pre_i = i, i = i + b$					↓	
Whether $LIST[i] < s$	10	20	30	40	50	FALSE
$pre_i = pre_i + 1$				↓		
Whether $LIST[pre_i] = s$	10	20	30	40	50	TRUE

Note: The block size in jump search is taken as $b = \sqrt{n}$, where n denotes the size of the list.

Jump search

```
int JumpSearch(int *LIST, int SIZE, int b, int s){
    int pre_i = 0, i = b;
    while(*(LIST + Min_Func(i, SIZE) - 1) < s)
        pre_i = i, i += b;
        if(pre_i >= SIZE)
            return;
    // Linear search between pre_i and i
    for(j=pre_i+1; j<i; j++)
        if(*(LIST + j) == s)
            return i; // Data is at the index j
    return; // Data is nowhere
}
int Min_Func(int x, int y){
    return x < y ? x : y;
}
```

Binary search

Let $LIST[] = \{10, 20, 30, 40, 50\}$ and $s = 40$.

Binary search

Let $LIST[] = \{10, 20, 30, 40, 50\}$ and $s = 40$.

$l = 0, r = 4, i = l + (r - l)/2 = 2$			$\downarrow$			
Whether $LIST[i] = s$	10	20	30	40	50	FALSE

Binary search

Let $LIST[] = \{10, 20, 30, 40, 50\}$ and $s = 40$.

$l = 0, r = 4, i = l + (r - l)/2 = 2$			↓			
Whether $LIST[i] = s$	10	20	30	40	50	FALSE

			↓			
Whether $LIST[i] < s$	10	20	30	40	50	TRUE

Binary search

Let $LIST[] = \{10, 20, 30, 40, 50\}$ and $s = 40$.

			↓			
$l = 0, r = 4, i = l + (r - l)/2 = 2$			↓			
Whether $LIST[i] = s$	10	20	30	40	50	FALSE
			↓			
Whether $LIST[i] < s$	10	20	30	40	50	TRUE
$l = i+1, i = l+(r-l)/2 = 3$				↓		
Whether $LIST[i] = s$	10	20	30	40	50	TRUE

Binary search

```
int BinarySearch(int *LIST, int l, int r, int s){
    int i;
    if(r >= l){
        i = l + (r - l)/2;
        if(*(LIST + i) == s)
            return i; // Data is in the middle
        if(*(LIST + i) < s) // Data is in the right
            return BinarySearch(LIST, i+1, r, s);
        // Data is in the left
        return BinarySearch(LIST, l, i-1, s);
   }
    return; // Data is nowhere
}
```

Comparing linear/jump/binary search

	Linear search	**Jump search**	**Binary search**
Requirements	Nothing	Ordered list	Ordered list
Backward scan required	No	Once	Multiple times
Time complexity (best case)	$O(1)$	$O(b)$	$O(1)$
Time complexity (average case)	$O(n)$	$O(\sqrt{n})$	$O(\log n)$
Time complexity (worst case)	$O(n)$	$O(\sqrt{n})$	$O(\log n)$

Interpolation search

Let $LIST[] = \{10, 20, 30, 40, 50\}$ and $s = 40$.

Interpolation search

Let $LIST[] = \{10, 20, 30, 40, 50\}$ and $s = 40$.

$i = 0 + (40 - 10) * (4 - 0)/(50 - 10) = 3$				$\downarrow$		
Whether $LIST[i] = s$	10	20	30	40	50	TRUE

Interpolation search

Let $LIST[] = \{10, 20, 30, 40, 50\}$ and $s = 40$.

$i = 0 + (40 - 10) * (4 - 0)/(50 - 10) = 3$				↓		
Whether $LIST[i] = s$	10	20	30	40	50	TRUE

Note: The interpolation search is efficient over binary search for the ordered lists having uniformly distributed data items.

Interpolation search

```
int InterpolationSearch(int *LIST, int SIZE, int s){
    int l = 0, r = SIZE - 1, p;
    while (l<=r && s>=*(LIST + l) && s<=*(LIST + r)){
        // Interpolate the position
        p = l + (s-*(LIST + l)) * (r-l) /
                        (*(LIST + r)-*(LIST + l));
        if(*(LIST + p) == s)
            return p;
        if (*(LIST + p) < s) // Data is in the right
            left = p + 1;
        else // Data is in the left
            right = p - 1;
    }
    return; // Data is nowhere
}
```

Fibonacci search

Let $LIST[] = \{10, 20, 30, 40, 50\}$ and $s = 40$. Note that, the index of largest Fibonacci number that is greater than or equal to the length of given array is $f = 5$.

Fibonacci search

Let $LIST[] = \{10, 20, 30, 40, 50\}$ and $s = 40$. Note that, the index of largest Fibonacci number that is greater than or equal to the length of given array is $f = 5$.

$i = Fibonacci(f-2) = 2$			↓			
Whether $LIST[i] = s$	10	20	30	40	50	FALSE

Fibonacci search

Let $LIST[] = \{10, 20, 30, 40, 50\}$ and $s = 40$. Note that, the index of largest Fibonacci number that is greater than or equal to the length of given array is $f = 5$.

$i = Fibonacci(f-2) = 2$			↓			
Whether $LIST[i] = s$	10	20	30	40	50	FALSE

			↓			
Whether $LIST[i] < s$	10	20	30	40	50	TRUE

Fibonacci search

Let $LIST[] = \{10, 20, 30, 40, 50\}$ and $s = 40$. Note that, the index of largest Fibonacci number that is greater than or equal to the length of given array is $f = 5$.

			↓			
$i = Fibonacci(f-2) = 2$			↓			
Whether $LIST[i] = s$	10	20	30	40	50	FALSE
			↓			
Whether $LIST[i] < s$	10	20	30	40	50	TRUE
$f = 4 - 2 + 1 = 3, i = i + Fibonacci(f-2) = 3$				↓		
Whether $LIST[i] = s$	10	20	30	40	50	TRUE

Fibonacci search

Let $LIST[] = \{10, 20, 30, 40, 50\}$ and $s = 40$. Note that, the index of largest Fibonacci number that is greater than or equal to the length of given array is $f = 5$.

$i = Fibonacci(f-2) = 2$			$\downarrow$			
Whether $LIST[i] = s$	10	20	30	40	50	FALSE
			$\downarrow$			
Whether $LIST[i] < s$	10	20	30	40	50	TRUE
$f = 4-2+1 = 3, i = i + Fibonacci(f-2) = 3$				$\downarrow$		
Whether $LIST[i] = s$	10	20	30	40	50	TRUE

Note: Fibonacci search is a suitable replacement of binary search where the list is unbounded. Unlike binary search, it does not use the costly division operation (though avoidable with bitwise shift).

Searching on Nonlinear Data Structures

1. Searching on graphs
   - Breadth-first search
   - Depth-first search
2. Searching on trees
   - Searching in binary trees
   - Searching in heaps
   - Searching in binary search trees
   - Searching in balanced search trees

Other Searching Problems

- Substring search
- Sublist search
- Range search
- Searching in streams

Range search

A database query may ask to find out all the students with GPA between g_1 and g_2 (say 7.5 and 8.5), and salary between s_1 and s_2 (say 90K and 100K).

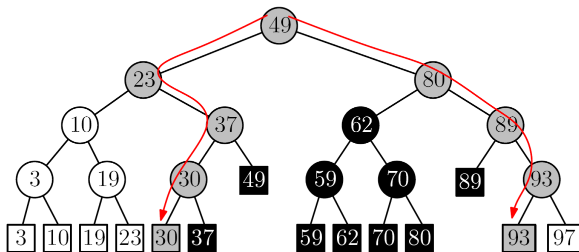
Roll No.	Salary	GPA
1	90K per mensem	7.2
2	80K per mensem	7.7
3	100K per mensem	7.1
4	90K per mensem	7.3
5	70K per mensem	8.2

Different variants of range search

1-D range query problem: Given a set of n points (data items) on $\mathbb{R}$ (real line) and an interval (1-D query range), efficiently find the points that stay within the interval.

Different variants of range search

1-D range query problem: Given a set of n points (data items) on $\mathbb{R}$ (real line) and an interval (1-D query range), efficiently find the points that stay within the interval.



A 1-D range search with query [25, 90] (using search tree)

Note: The gray nodes are only visited (traversed).

Different variants of range search

2-D range query problem: Given a set of n points (coordinates) on $\mathbb{R}^2$ (2-D space) and a boundary (2-D query range), efficiently find the points that stay within the boundary.

Different variants of range search

2-D range query problem: Given a set of *n* points (coordinates) on $\mathbb{R}^2$ (2-D space) and a boundary (2-D query range), efficiently find the points that stay within the boundary.

A 2-D range search with query boundary (using *k*-dimensional tree)

Problems – Day 18

1. The password file on a GNU/Linux machine looks like the example available from `https://www.dropbox.com/s/7x17j7z3alfkfdc/etc-passwd.txt?dl=0`. This file lists the accounts of MTCS students of ISI from the 2017-19 batch. It consists of multiple columns, separated by '`:`'. Columns 1,3, and 5 correspond respectively to a student's roll number, user ID and name.
   Write a C program to read and store this information in the form of an array of structures, with each structure representing a single student.
   Now write appropriate comparator functions so that the `qsort` function can be used to sort the array in
   - ascending order of roll numbers;
   - descending order of roll numbers;
   - alphabetic order by name.

Problems – Day 18

2. Let us define a sequence $a_0, a_1, a_2, \ldots, a_{n-1}$ as Λ-*bitonic* if there exists a j, $0 \leq j < n$, such that $a_0 < a_1 < \cdots < a_j > a_{j+1} > \cdots > a_{n-1}$. Consider an $m \times n$ matrix A consisting of integer entries, such that each row and each column of the matrix forms a Λ-bitonic sequence. Write a program to efficiently find the largest element of the matrix.
3. Write a program to implement Fibonacci search on a generic array having ordered list of data items. Consider that the comparator() function works on various data types in a usual sense.