Searching & Sorting

Data Structures & Algorithms

CONTENT

- **Searching**
  - Sequential Search
  - Binary Search
- **Sorting**
  - Insertion Sort
  - Selection Sort
  - Bubble Sort
  - Quick Sort
  - Merge Sort

CONTENT

- **Searching**
  - Sequential Search
  - Binary Search
- **Sorting**
  - Insertion Sort
  - Selection Sort
  - Bubble Sort
  - Quick Sort
  - Merge Sort

- How do you find your name on a class list?
- How do you search a book in a library?
- How do you find a word in a dictionary?

⇨ **Search is very often operation**

- Sequential Search
- Binary Search

- Suppose we have a unsorted list of *n* numbers

  ⇨ Find the position of the number 531

```
# define MAX_SIZE 100
typedef struct {
   int  key;
   /* other fileds */
} element;
element  list[MAX_SIZE];
```

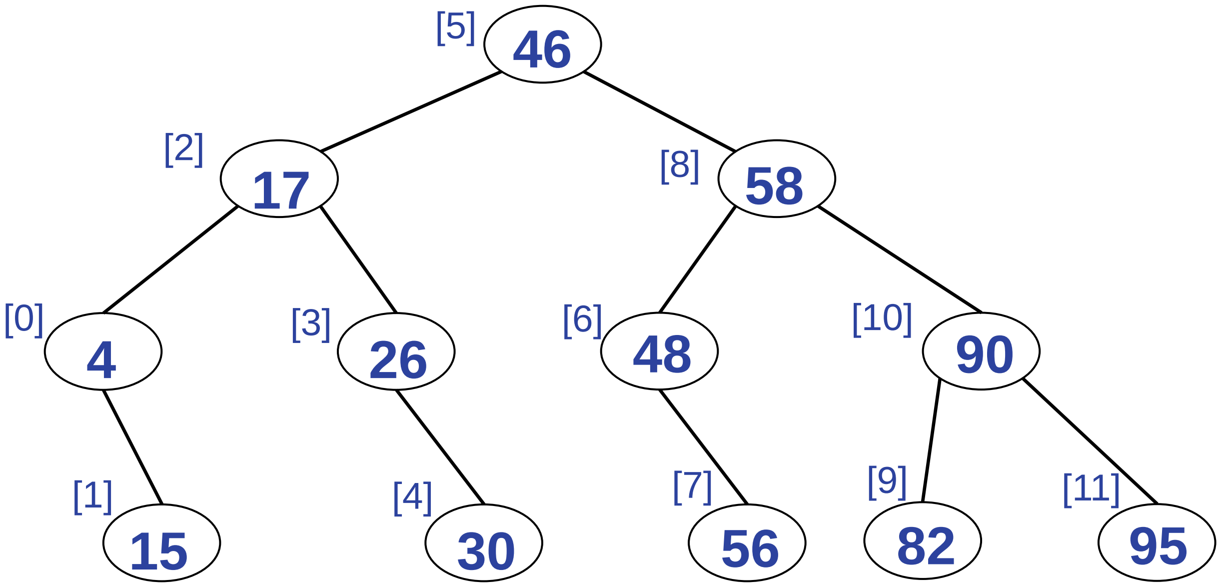
```
int   seqSearch(int list[], int searchNum, int n){
        /* Search an array, list, that has n numbers.
        Return i, if list[i] = searchNum.  Return -1, if searchNum is not in the list */
        list[n].key = searchNum;          // Sentinel that signals the end of the list
        int  i=0;
        while (list[i].key != searchNum)  i++;
        return ((i < n) ? i : -1);
}
```

- **Analysis**
  - Example: 44, 55, 12, 42, 94, 18, 06, 67
  - unsuccessful search: n+1
  - The average number of comparisons for a successful search is

$$\sum_{i=0}^{n-1}(i+1)\,/\,n=\frac{n+1}{2}$$

- **Binary Search**
  - Input: **sorted** list
    (i.e. list[0].key ≤ list[1].key ≤ ... ≤ list[n-1].key)
  - Compare **searchNum** and **list[middle].key**, where middle = (n-1)/2, there are three possible outcomes:
    - searchNum < list[middle].key
      ⇨ Search: list[0] … list[middle-1]
    - searchNum = list[middle].key
      ⇨ Search terminal successfully: return **middle**
    - searchNum > list[middle].key
      ⇨ Search: list[middle+1] … list[n-1]

- **Example**
  - Input : 4, 15, 17, 26, 30, 46, 48, 56, 58, 82, 90, 95



Decision tree for binary search

⇨Binary search makes no more than **O(log n)** comparisons (i.e. height of tree)

```
int binSearch(element list[], int searchNum, int n){
        int left = 0, right = n-1, middle;
        while (left <= right){
                middle = (left + right) / 2;
                switch (COMPARE(list[middle].key, searchNum)){
                        case -1 : left = middle + 1;
                                  break;
                        case 0 : return middle;
                        case 1 : right = middle - 1;
                }
        }
        return -1;
}
```

CONTENT

- **Searching**
  - Sequential Search
  - Binary Search
- **Sorting**
  - Insertion Sort
  - Selection Sort
  - Bubble Sort
  - Quick Sort
  - Merge Sort

- **List Verification**
  - Determine if all elements on one list appear on another list
    - Assume the 2 lists have m and n elements, respectively, and there are no repetitions
  - 1st approach (O(mn) time)
    1. Assume both lists are not sorted
    2. for each element in the 1st list do
    3. search the element in the 2nd list
    4. end for
  - 2nd approach (O(mlogm + nlogn + m + n) time)
    1. Sort the 1st list in O($m$log$m$) time
    2. Sort the 2nd list in O($n$log$n$) time
    3. Compare the two sorted lists in O(m + n) time

- **Sorting Problem**
  - We are given a list of records $(R_0, R_1, \ldots, R_{n-1})$. each record R_i, has a key value K_i
  - Find a permutation *p* such that
    - **Sorted:** $K_{p(i-1)} \leq K_{p(i)}$, for $0 < i \leq n-1$
    - **Stable:** If $i<j$ and $K_i=K_j$ in the list, then R_i precedes R_j in the sorted list.
  - Sorting algorithm is **internal** if it makes use of only memory
  - Sorting algorithm is **external** if it stores intermediate results on hard disks

- **Insertion sort**
  - *n* records are stored in an array *A*
  - *n*-1 steps in the insertion sorting algorithm
  - During step *i*, *i*-1 elements *A*[0],*A*[1], …, *A*[*i*-1] are in sorted order
  - We need to insert *A*[*i*] into *A*[0],*A*[1], …, *A*[*i*-1] such that these *i* elements are sorted
    - the element A[i] is compared with *A*[0],*A*[1], …,*A*[*i*-1], one by one, until a place, say *k*, is decided for *A*[*i*]
    - the elements A[k],A[k+1], …, A[i-1] are moved to the right one position and *A*[*i*] is copied to *A*[*k*]

- **Example**

26	5	77	1	61	11	59	15	48	19
5	26	77	1	61	11	59	15	48	19
5	26	77	1	61	11	59	15	48	19
1	5	26	77	61	11	59	15	48	19
1	5	26	61	77	11	59	15	48	19
1	5	11	26	61	77	59	15	48	19
1	5	11	26	59	61	77	15	48	19
1	5	11	15	26	59	61	77	48	19
1	5	11	15	26	48	59	61	77	19
1	5	11	15	19	26	48	59	61	77

```
void  insertionSort(element list[], int n){
        int i, j;
        element  next;
        for (i=1; i<n; i++) {
                next = list[i];
                for (j=i-1; j>=0 && next.key<list[j].key ; j--)
                        list[j+1] = list[j];
                list[j+1] = next;
        }
}
```

- **Analysis**
  - During step i, we need at most i+1 comparisons and movements
  - In the **worst case**, we need (1+2+…+n-1) = $n(n-1)/2$= O(n^2) comparisons and movements
  - a record $A[i]$ is said ***left-out-of-order*** (LOO) if $A[i] < max\{A[0];A[1];A[2];...;A[i-1]\}$
    - Each LOO record induces at most n comparisons
    - Each non-LOO record induces 1 comparison
  - Suppose there are k LOO records and n - k non-LOO records
  - We need at most $kn + (n - k)$ comparisons
  - Since $k <= n$, time is O(n^2)

⇨This algorithm is **stable**

- **Variation**
  - Binary insertion sort
    - sequential search --> binary search
    - reduce # of comparisons
    - # of moves unchanged
  - List insertion sort
    - array --> linked list
    - sequential search
    - no movement, adjust pointers only

- **Selection Sort**
  - *n* records are stored in an array *A*
  - loop *n*-1 steps in the selection sorting algorithm
  - During step *i*, *A*[0],*A*[1], …, *A*[*i*-1] are the smallest *i* elements, arranged in sorted order
  - Then the smallest among *A*[*i*],*A*[*i*+1], ... ,*A*[*n*-1] is selected and is exchanged with *A*[*i*]

- **Example**

26	5	77	**1**	61	11	59	15	48	19
1	26	77	**5**	61	11	59	15	48	19
1	26	77	**5**	61	11	59	15	48	19
1	**5**	77	**26**	61	11	59	15	48	19
1	5	77	26	61	**11**	59	15	48	19
1	5	**11**	26	61	**77**	59	15	48	19
1	5	11	26	61	77	59	**15**	48	19
1	5	11	**15**	61	77	59	**26**	48	19
1	5	11	15	61	77	59	26	48	**19**
1	5	11	15	**19**	77	59	26	48	**61**
1	5	11	15	19	77	59	**26**	48	61
1	5	11	15	19	**26**	59	**77**	48	61
1	5	11	15	19	26	59	77	**48**	61
1	5	11	15	19	26	**48**	77	**59**	61
1	5	11	15	19	26	48	77	**59**	61
1	5	11	15	19	26	48	**59**	**77**	61
1	5	11	15	19	26	48	59	77	**61**
1	5	11	15	19	26	48	59	61	77

```
void  selectionSort(element list[], int n){
        int i, j, min;
        element tmp;
        for (i=0; i<=n-2; i++)
                for (j=i+1; j<=n-1; j++)
                        if (list[i].key > list[j].key) {
                        /* exchange two elements -> should be improved */
                                tmp = list[i];
                                list[i] = list[j];
                                list[j] = tmp;
                        }
}
```

```
void selectionSort'(element list[], int n){
        int i, j, min;
        element tmp;
        for (i=0; i<=n-2; i++) {
                min = i;                        /* search the smallest element */
                for (j=i+1; j<=n-1; j++)
                        if (list[min].key > list[j].key) min = j;
                If (i != min) {                 /* exchange two elements */
                        tmp = list[i];
                        list[i] = list[min];
                        list[min] = tmp;
                }
        }
}
```

- **Analysis**
  - During step i, there are at most $n - i$ comparisons
  - In the worst case, there are

$$(n - 1 + n - 2 + \ldots + 1) = n(n - 1)/2 = O(n^2) \qquad \text{comparisons}$$

- **Bubble Sort**
  - *n* records are stored in an array *A*
  - *n* steps in the bubble sorting algorithm
  - During step *i*, *A*[0],*A*[1], …, *A*[*i*-1] are the smallest *i* elements, arranged in sorted order
  - Then among *A*[*i*],*A*[*i*+1], ... ,*A*[*n*-1], each pair ((*A*[*n*-2], *A*[*n*-1]) , …) will be exchanged if they are out of order, so *A*[*i*] will be the smallest
    - At step *i*, the smallest element bubbles

Example

5	5	5	5	5	**0**	0	0	0	0	0	0	0	0	0	0
1	1	1	1	**0**	**5**	5	5	5	**1**	1	1	1	1	1	1
7	7	7	**0**	**1**	1	1	1	**1**	**5**	5	5	**4**	4	4	4
4	4	**0**	**7**	7	7	7	**4**	**4**	4	4	**4**	**5**	5	**5**	5
0	**0**	**4**	4	4	4	**4**	**7**	7	7	**6**	**6**	6	**6**	**6**	**6**
6	**6**	6	6	6	6	**6**	6	6	6	**7**	7	7	**7**	7	**7**

```
void bubbleSort(element list[], int n){
        int i, j, min;
        element tmp;
        for (i = 0; i < n-1; i++)
                for (j = n-1; j > i; j--)
                        if (list[j-1].key > list[j].key) {
                                /* exchange two elements */
                                tmp = list[j-1];
                                list[j-1] = list[j];
                                list[j] = tmp;
                        }
        }
}
```

- **Quick Sort**
  - n records are stored in an array A
  - Choose a pivot $A[i]$
  - Re-arrange the list so that $A[0], A[1], \ldots, A[i-1]$ are smaller than $A[i]$ while $A[i+1], A[i+2], \ldots, A[n-1]$ are all greater than $A[i]$
  - Then, recursively apply quick sort to the first half $A[0], A[1], \ldots, A[i-1]$ and the second half $A[i+1], A[i+2], \ldots, A[n-1]$, respectively

- **Divide and Conquer algorithm**

  Two phases:

- Partition phase: **divide** the list into half

≤ pivot	**pivot**	**> pivot**

- Sort phase:
  - **Conquers** the halves: apply the same algorithm to each half

≤ pivot **> pivot**

≤ p’	**p’**	**> p’**	**pivot**	**≤ p”**	**p”**	**> p”**

- **Example**

R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	left	right
{ 26	5	37	1	61	11	59	15	48	19 }	0	9
{ 11	5	19	1	15 }	26	{ 59	61	48	37 }	0	4
{ 1	5 }	11	{19	15 }	26	{ 59	61	48	37 }	0	1
1	5	11	15	19	26	{ 59	61	48	37 }	3	4
1	5	11	15	19	26	{ 48	37 }	59	{ 61 }	6	9
1	5	11	15	19	26	37	48	59	{ 61 }	6	7
1	5	11	15	19	26	37	48	59	61	9	9
1	5	11	15	19	26	37	48	59	61		

```
#define SWAP(a,b,t)   {int  t;   t=a;   a=b;   b=t; }
void quickSort(element list[], int left, int right){
        int pivot, i, j;              element temp;
        if (left < right) {                          /* divide */
                i = left;   j = right+1;
                pivot = list[left].key;
                do {
                do i++;  while (list[i].key < pivot);
                do j--;    while (list[j].key > pivot);
                if (i < j)   SWAP(list[i], list[j], temp);
        } while (i < j);
     SWAP(list[left], list[j], temp);          /* put pivot a good position*/
     quickSort(list, left, j-1);               /* conquer */
     quickSort(list, j+1, right);
   }
}
```

- **Analysis**
  - Time complexity
    - Worst case: $O(n^2)$
    - Best case: O(nlogn)
    - Average case: O(nlogn)
  - Space complexity
    - Worst case: O(n)
    - Best case: O(logn)
    - Average case: O(logn)
- **Unstable**

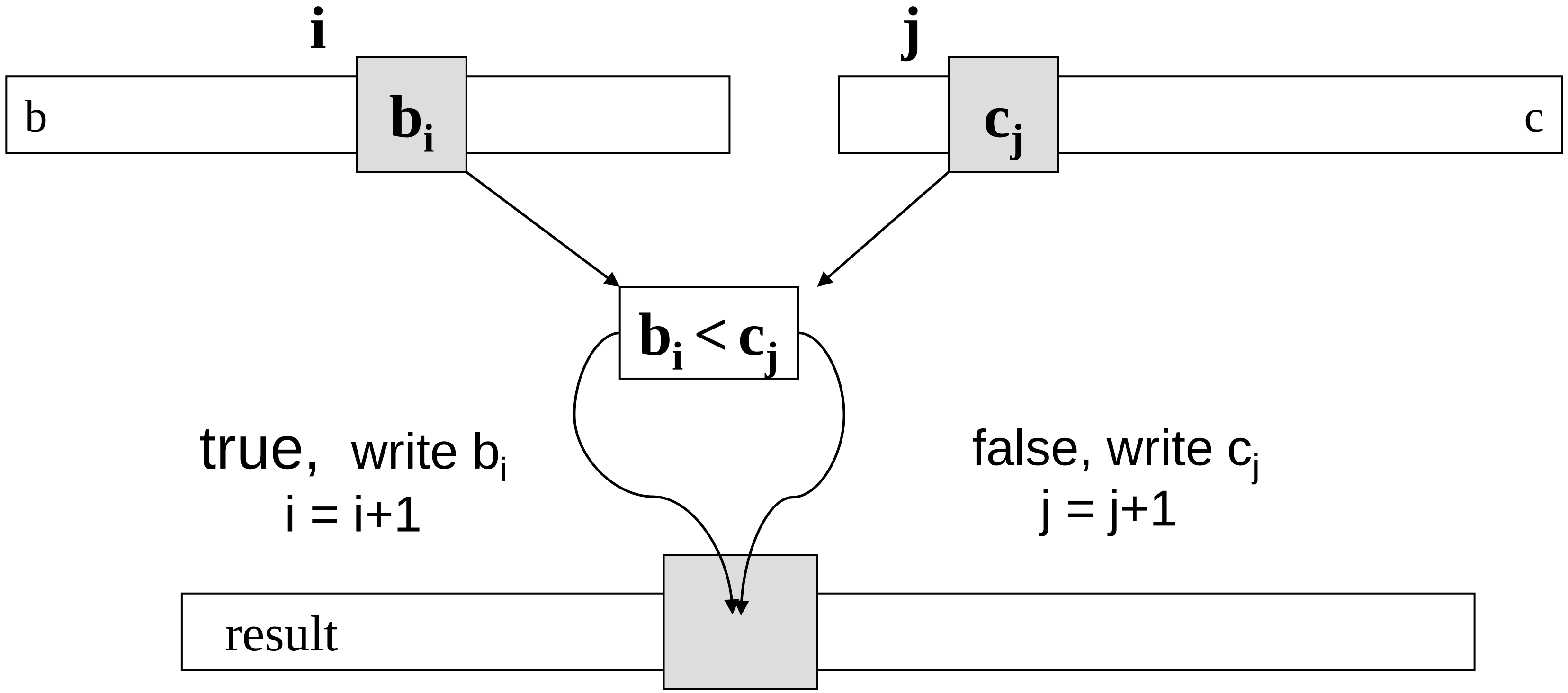
- **Merge Sort**

Divide and Conquer algorithm

1. Cut the list in 2 halves
2. Sort each half, respectively, probably by merge sort recursively
3. Merge the 2 sorted halves

⇨ **How to merge two sorted lists** (list[i], ... , list[m] and list[m+1], ... , list[n]) **into single sorted list,** (sorted[i], ... , sorted[n])**?**

- **Merge 2 sorted lists**

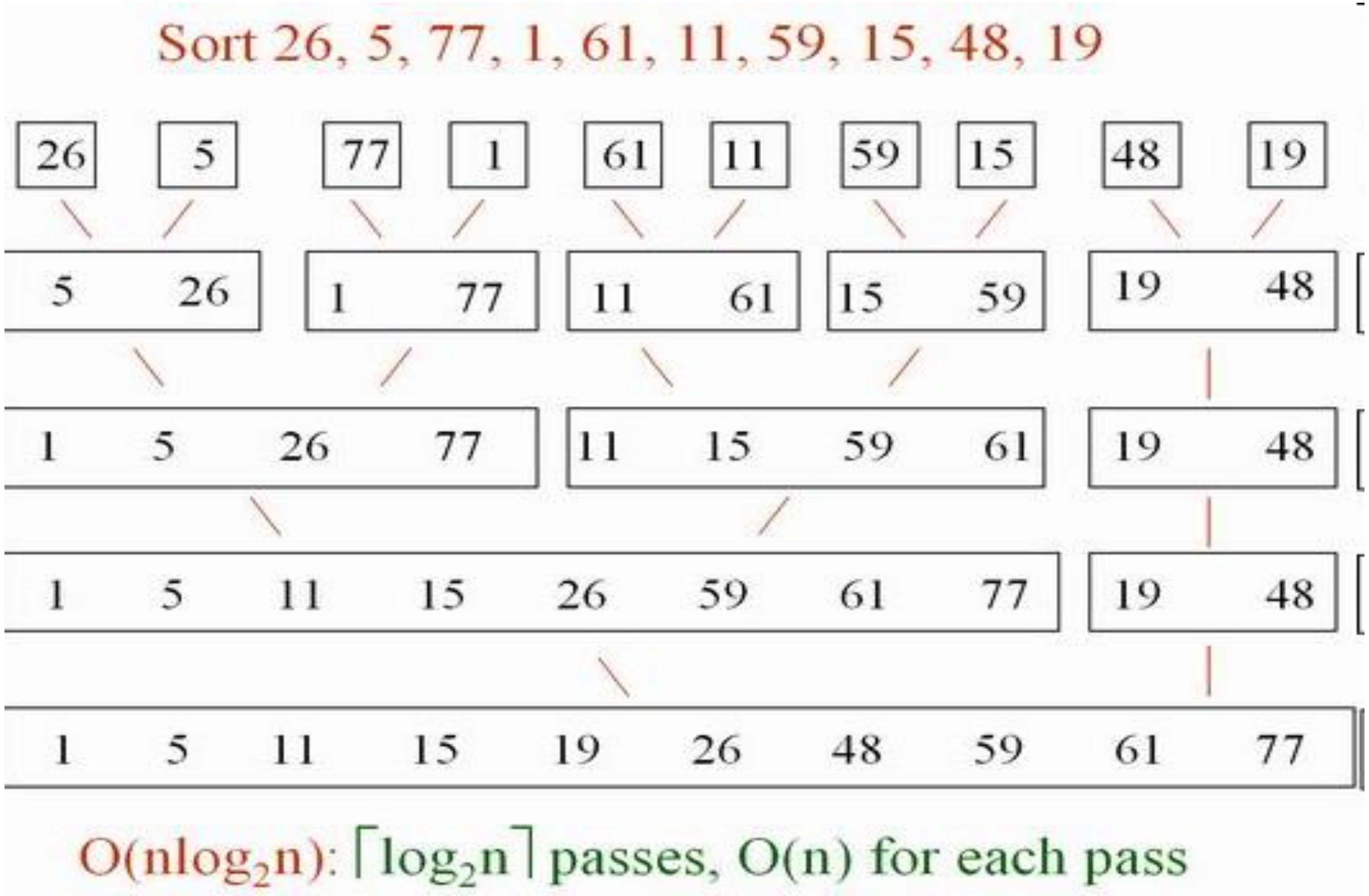


```
void merge(element list[], element sorted[], int i, int m, int n){
        int j, k, t;
        j = m+1;
        k = i;
        while (i<=m && j<=n) {
                if (list[i].key<=list[j].key) sorted[k++]= list[i++];
                else sorted[k++]= list[j++];
        }
        if (i>m)
                for (t=j; t<=n; t++)  sorted[k+t-j]= list[t];
        else
                for (t=i; t<=m; t++) sorted[k+t-i] = list[t];
}
```

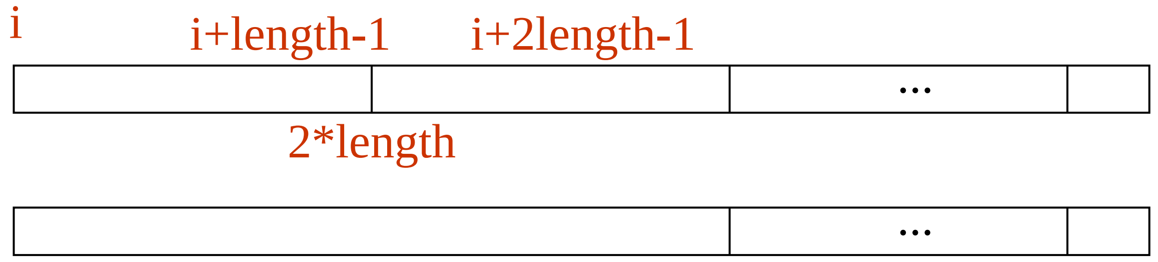
Time complexity: O(n)
Space complexity: O(n)

- **Iterative Merge Sort**
  - We assume that the input sequence has n sorted lists each of length 1
  - We merge these lists pair-wise to obtain n/2 list of size 2
  - We then merge the n/2 lists pair-wise, and so on, until a single list remains

- **Iterative Merge Sort Example**

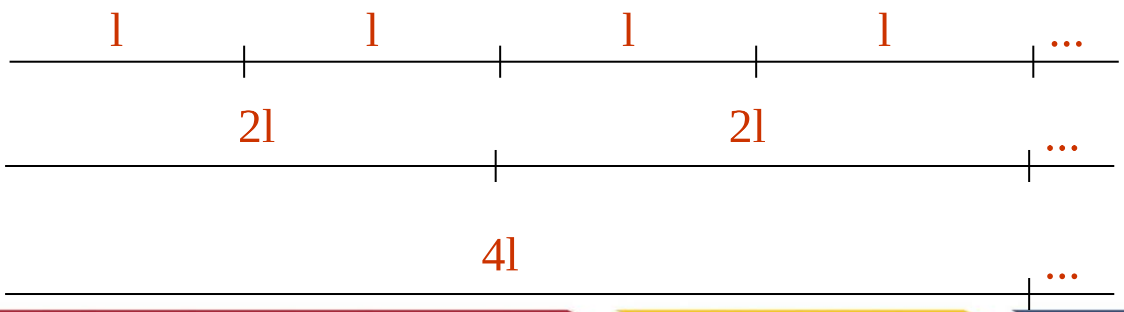


```
void mergePass(element list[], element sorted[],int n, int length){
        int i, j;
        for (i=0; i<n-2*length; i+=2*length)
                merge(list,sorted,i,i+length-1,i+2*length-1);
        if (i+length<n)        //One complement segment and one partial segment
                merge(list, sorted, i, i+length-1, n-1);
        else                   //Only one segment
                for (j=i; j<n; j++)
                        sorted[j]= list[j];
}
```



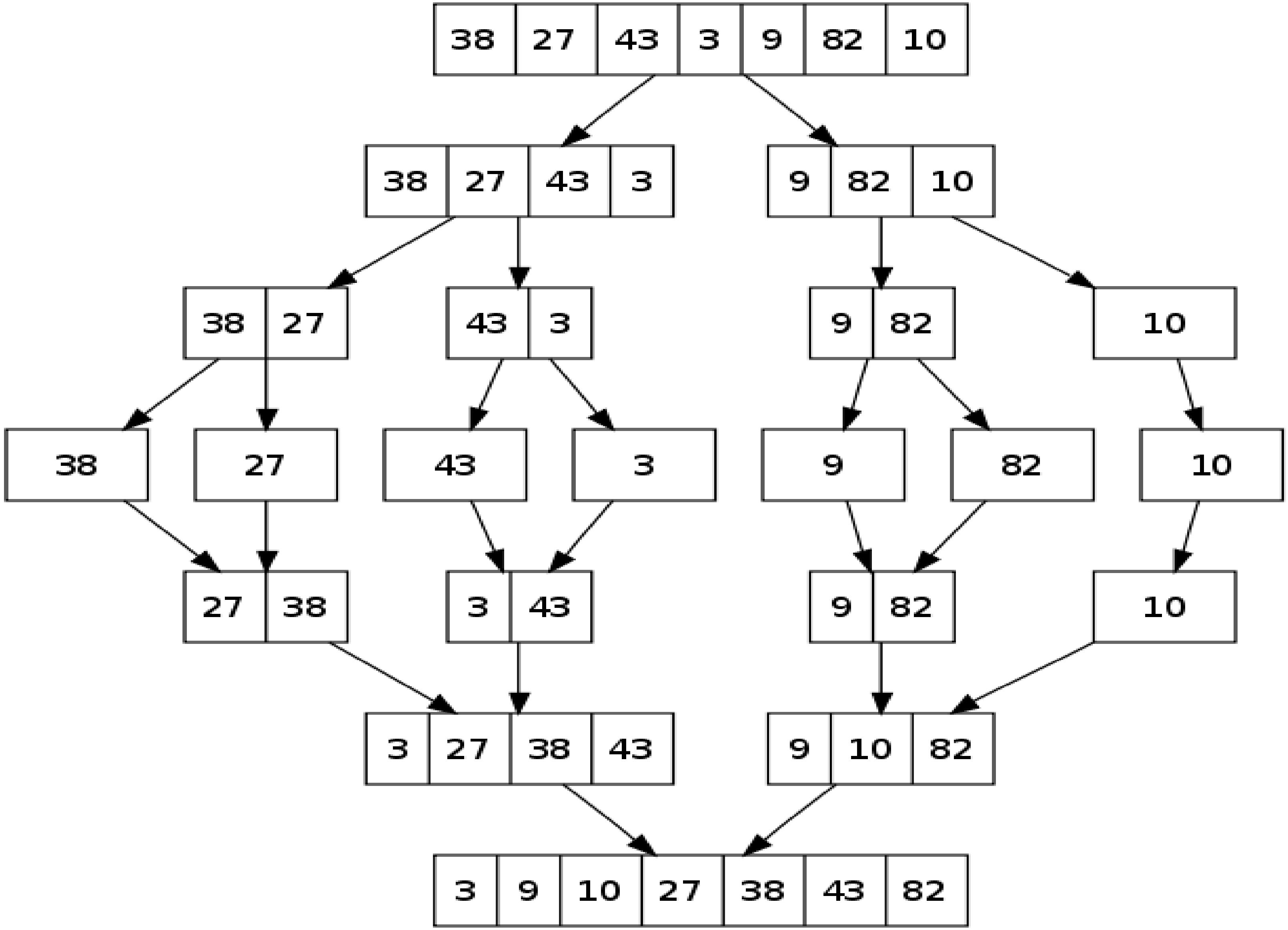
- **Iterative Merge Sort**

```
void mergeSort(element list[], int n){
        int length=1;
        element extra[MAX_SIZE];
        while (length<n) {
                mergePass(list, extra, n, length);
                length *= 2;
                mergePass(extra, list, n, length);
                length *= 2;
        }
}
```



- **Recursive Merge Sort**
  - Given *n* elements (in an array) to be sorted
  - Cut them into two halves, each with *n*/2 elements
  - Sort each half by the same merge sort algorithm recursive
  - Finally merge the two sorted halves into a single array

- **Example**


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

- **Recursive Merge Sort**

```
void mSort(int list[], int left, int right) {
        int mid;
        if (right > left){
                mid = (right + left) / 2;
                mSort(list, left, mid);
                mSort(list, mid+1, right);
                merge(list, left, mid+1, right);
        }
}

void mergeSort(element list[], int array_size) {
        mSort(list, 0, array_size - 1);
}
```

- **Recursive Merge Sort**

```
void merge(element list[], int left, int mid, int right) {
  int i, left_end, num_elements, tmp_pos;
  element temp[right - left + 1];
  left_end = mid - 1;
  tmp_pos = left;
  num_elements = right - left + 1;
  while ((left <= left_end) && (mid <= right)) {
    if (list[left].key <= list[mid]).key {
      temp[tmp_pos] = list[left];
      tmp_pos = tmp_pos + 1;
      left = left +1;
    } else {
      temp[tmp_pos] = list[mid];
      tmp_pos = tmp_pos + 1;
      mid = mid + 1;
    }
  }
while (left <= left_end) {
    temp[tmp_pos] = list[left];
    left = left + 1;
    tmp_pos = tmp_pos + 1;
  }
  while (mid <= right) {
    temp[tmp_pos] = list[mid];
    mid = mid + 1;
    tmp_pos = tmp_pos + 1;
  }
  for (i=0; i < num_elements; i++) {
    list[right] = temp[right];
    right = right - 1;
  }
}
```

- Analysis
  - **Time complexity**
    - $T(n) = 2T(n/2) + n-1$
    - $O(n \log n)$ in time
    - $O(n)$ in space

    ⇨Can be improved by an approach with O(1) in space
  - **Stable**

SUMMARY

- **Searching**
  - Sequential Search
  - Binary Search
- **Sorting**
  - Insertion Sort
  - Selection Sort
  - Bubble Sort
  - Quick Sort
  - Merge Sort

Enjoy the Course…!