Chapter 3

Sorting Algorithms

- **The array sorting problem**: Given an array A rearrange array elements such that A[i] <= A[i+1] for i = 0, 1, 2, …, n-2.

- Note input can be a file or a linked list. Also, items may each consist of a record that contains several pieces of information. However, the sorting is performed based on an identifier or a key contained in each record.

- Categorize sorting algorithms :

  - $\Theta(n^2)$ algorithms: Insertion sort, Bubble sort, Selection sort, etc. These algorithms are usually simple and consist of two nested loops.

  - $\Theta(n^\alpha),\ 1 < \alpha < 2$ algorithms: various versions of Shell sort.

  - $\Theta(n \lg n)$ algorithms: Quick sort, Merge sort, Heap sort, etc.

3.1 Insertion Sort

One of the simplest sorting algorithms - consists of n-1 passes. In pass p, move p-th element to left until its correct position.

Example:

Original	34	8	64	51	32	21	positions moved
After p=2	8	34	64	51	32	21	**1**
After p=3	8	34	64	51	32	21	**0**
After p=4	8	34	51	64	32	21	**1**
After p=5	8	32	34	51	64	21	**3**
After p=6	8	21	32	34	51	64	**4**

```
void InsertionSort (int A[ ], int n)
{
        int temp;

        A[0] = MinVal      // put a very small value
        for (int p = 2; p <= n;  p++)
        {
                temp = A[p];
                for ( int j = p; temp < A[j-1]; j--)
                        A[j] = A[j-1];
                A[j] = temp;
        }
}
```

3.1.1 Analysis of Insertion Sort

Worst case: array is in reverse order, each element is moved p positions

$$C_w(n) = \sum_{p=2}^{n} \sum_{j=0}^{p} 1 = \sum_{p=2}^{n} p = \frac{n(n+1)}{2} - 1$$

$C_w(n) = \Theta(n^2)$. On average p/2 comparisons:

$$C_a(n) = \sum_{p=2}^{n} \frac{p}{2} = \frac{1}{2}\left(\frac{n(n+1)}{2} - 1\right); \quad C_a(n) = \Theta(n^2)$$

If array is already sorted, the inner for loop will not iterate and the complexity is $C(n) = \Theta(n)$.

- **Inversion**: There is an inversion if $i<j$ but $A[i] > A[j]$.

Example : 34 8 64 51 32 21

8 inversions: (34,8), (34,32), (64,51), (64,32), …(32,21).

# of inversions n_I = # of swaps in the inner for loop.

Complexity is $O(n + n_I)$.

Worst case: array is reverse ordered, $n_I \quad \frac{n(n-1)}{2}$

Exercise: Show that the average inversions is $\frac{n(n-1)}{4}$.

- Insertion sort is an "in-place" method.

3.2 Selection Sort

This algorithm is quadratic complexity.

Step 1: Set i= 0;
Step 2: Scan array from A[i] to A[n-1] and find smallest element
Step 3: Swap the smallest element with A[i]
Step 4: Increment i and repeat steps 2 and 3.

Exercise:
(a) Apply Selection sort to the example given in Section 3.1
(b) Write a function to implement Selection sort
(c) Show that the complexity of selection sort is $\Theta(n^2)$.

3.3 Shell Sort

Compare elements that are distant in array. The distance decreases as the algorithm progresses → *diminishing increment* sort.

- Shell sort uses increment sequence $\{h_t, h_{t-1}, \cdots, h_2, h_1\}$, where $h_1 = 1$.

# of increments t and the increments $h_t > h_{t-1} > \cdots > h_2 > 1$ to be chosen. Some choices result in better complexity. An example of the sequence is {5, 3, 1} .

- After a phase with increment h_k, $1 \le k \le t$, for all i we have $A[i] \le A[i + h_k]$ (where this makes sense), i.e. elements that are spaced h_k apart are relatively sorted, and the array is called h_k sorted.

Example:

Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	**35**	**17**	**11**	**28**	**12**	**41**	**75**	**15**	**96**	**58**	**81**	**94**	***95***
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

- **Shell sort property**: an h_k sorted file, that is then h_{k-1} sorted, remains h_k sorted.

- The action of h_k sort is to perform an insertion sort on h_k independent sub-arrays. In the above example these three sub-arrays after 5 sort that are to be 3-sorted. These sub-arrays are
(35, 28, 75, 58, 95),
(17, 12, 15, 81),
(11, 41, 96, 94).

```
void ShellSort ( int a[ ], int n)
{
        int  i, j, incr, t, temp;
        int  h[];
        h[0] = 1; h[1] = 3; h[2] = 5;  // 3 increments are (5, 3,1)
        for ( t = 2; t<0; t--)
        {
                incr = h[t];
                for ( j = incr;  j < n; j++)
                        i = j – incr;
                        temp = A[j];
                        while( i >= 0)
                        {
                                if ( temp > A[i])
                                        break;
                                else
                                {
                                        A[i+ incr] = A[i];
                                        i = i – incr;
                                }
                        A[i+incr] = temp;
                        }
        }
}
```

3.3.1 Complexity of Shell Sort

- The complexity of Shell sort depends significantly on the choice of increments. Shell proposed the increments
  $[n/2], [n/4], [n/8], \cdots, 1.$
  - not a particularly good choice.

- Worst case complexity of Shell sort with his increments is $\Theta(n^2)$.

Consider a "bad" input with n is a power of 2 with $n/2$ largest elements in even positions and $n/2$ elements are in the odd positions, e.g. for n = 16

1 **9** 2 **10** 3 **11** 4 **12** 5 **13** 6 **14** 7 **15** 8 **16**

Shell's suggested increments : 8, 4, 2, 1.

After 8-sort, 4-sort, and 2-sort phases, no useful work is done. Restoring only the i-th element to its correct position, requires moving (i-1) spaces in the array. Since there are n/2 smallest elements, the work needed is at least $\sum_{i=1}^{n/2}(i-1)$ or $\Omega(n^2)$.

To show $O(n^2)$ bound, note that a pass with increment h_k consists of h_k insertion sorts of about n/h_k elements. Since insertion sort is quadratic, the total cost of a pass is $O\left(h_k(n/h_k)^2\right) = O(n^2/h_k)$ Summing over all passes gives the bound

$$O\left(\sum_{k=1}^{t}\frac{n^2}{h_k}\right) = O\left(n^2\sum_{k=1}^{t}\frac{1}{h_k}\right) = O(n^2) \quad \text{since} \sum_{k=1}^{t}\frac{1}{h_k} < 2.$$

Thus the complexity of Shell sort with his suggested increment is $\Theta(n^2)$.

- Hibbard sequence is 1, 3, 7, 15, …, $2^k - 1$. It is similar to Shell sequence (1, 2, 4, 8, 16, …). However, Hibbard sequence has no common factor.

- With Hibbard sequence: worst case complexity of $\Theta(n^{3/2})$.

  Its average complexity analysis is still an open question, but simulations have shown that it is $O(n^{5/4})$.

- Sedgewick sequence: (1, 5, 19, 41, 109, …) Obtained from alternately from $9(4^i) - 9(2^i) + 1$ and $4^i - 3(2^i) + 1$. Worst case complexity with Sedgewick's sequence is $O(n^{4/3})$.
- Simulation studies have shown $O(n^{7/6.})$.

- Performance of the Shell sort algorithm is generally good for large input size. Although it is a relatively simple algorithm, its complexity analysis is very hard.

3.4 Divide and Conquer Sorting Algorithms

- Decomposing (breaking) a problem of size n into several (say m) instances of the same problems with smaller sizes n_i, $i = 1, 2, \cdots, m$.

- Solving each problem of size n_i.
- Combining solutions of smaller size problems to get solution of original problem.

Pseudo-code for divide and conquer algorithm.

```
Solve (P)  // solve problem P
 n = size (P);
 if (n <= SmallSize)
     Solve (P)   // solve the problem directly;
 else
     divide P into P1, P2, ..., Pm
     for each i=1 to m
         Si = Solve (Pi);
      Combine (S1, S2, ... Sm)
Return solution
```

Note that for divide and conquer algorithm to be useful, decomposing, solving the smaller size sub-problems and combining them should requires less work than solving the original large problem.

3.5 Quick Sort

Very efficient algorithms. Let A be the array

1. If number of elements in a is 0 or 1 return

2. Pick an element of A, say p. This element is called the pivot.

3. Put p aside, and partition the remaining elements into to sub-arrays:

$$A_1 = \{x \in A[] \mid x < p \}$$
$$A_2 = \{x \in A[] \mid x > p \}$$

4. Apply Quicksort to A_1 and A2

5. Go to Step 1, and then combine the sub-arrays.

Example:

Original Array (2 5 1 8 9 12 4 3 20)

Partition (2 5 1 3 4) **(8)** (9 12 20)
Subarray 1 Pivot Subarray 2

Partition (1 2) **(3)** (4 5) **(8)** (9) **(12)** (20)

pivots

Combine (1 2 3 4 5 8 9 12 20)

- Quicksort recursively solves 2 sub-problems, and requires linear additional work for partitioning. The two sub-arrays are not guaranteed to have equal size.

- Quicksort is an in-place sorting algorithm.

3.5.1 Selecting the Pivot

Choice of pivot is critical, complexity of depends on this choice.
Best pivot divides the array into two arrays of almost equal size

- First element: This is usually a bad choice, specially when the input file is almost sorted.

- Random selection: Generally good, but generating a random index in the range adds to the overhead.

- Median of three: The best choice would be median of all n array elements. However, this is expensive. A better choice would be the median of first, last and middle elements

- Example

8 1 4 9 6 3 5 2 7 0

left (8), center (6), right (0)

1. Median (left, center, right) = 6
2. Swap center and right

8 1 4 9 0 3 5 2 7 **6**

i = left (8) →

← j = right - 1 (7)

3. Move i to the right until element > pivot, then stop

4. Move j to the left until element < pivot, then stop

8 1 4 9 0 3 5 2 7 **6**

i j

5. Swap elements at positions i and j, and perform Steps 3 and 4

2 1 4 9 0 3 5 8 7 **6**

i j

2 1 4 5 0 3 9 8 7 **6**

5. Continue until i and j cross, do not swap.

2 1 4 5 0 3 9 8 7 **6**

j i

6 Finally, swap the element at position i with the pivot

2 1 4 5 0 3 6 8 7 9

sub-array 1 < pivot sub-array 2 > pivot

6. Apply above steps to each of two sub arrays.

- If i or j encounter an element that is equal to pivot, stop and swap. This may produce many swaps, but

sub-arrays will be more balanced than the case where no swaps are made.

Example

8 8 8 8 8 8 8 8 8

i j pivot

Since elements at i and j are equal to the pivot, swap.

8 8 8 8 8 8 8 8 8

j i

Now restore the pivot

8 8 8 8 8 8 8 8 8

sub-array 1 pivot sub-array 2

If we do not swap and continue until i and j cross

8 8 8 8 8 8 8 8 **8**

i j

8 8 8 8 8 8 8 8 8

sub-array 1 sub-array 2

The sub-arrays are unbalanced.

- Small Files: Quicksort is not efficient for small and almost sorted sub-files or sub-arrays. During the application of Quicksort, when the sub-files get small (Cutoff size of 10 to 20), apply insertion sort.

3.5.2 Implementation of Quicksort

```
int  median3 (int A[], int left, int right)
{
int center;
center = ( left + right)/2;
if (A[left] > A[center] )
        Swap(A[left], A[center]);  // Swap is a function
If (A[left] > A[right])
        Swap(A[left], A[right]);
If (A[center] > A[right])
        Swap(A[center], A[right]);
Swap(A[center], A[right-1]);        // put the pivot to a
side for now
Return A[right-1];
}
```

Note that in the above, the largest of the three is placed in A[right], pivot is in A[right-1], and the smallest of the three is in A[left]. Thus
i = left + 1 and j = right –2.

```
void Quicksort( int A[], int n)
{
   Qsort(A, 1, n);
    Isort(A, n);
}

void Qsort( int A[], int left, int right)
{
  int i, j, pivot;
// use Cutoff to stop Quicksort and start Insertion sort.
  if((left + Cutoff <= right))
     {
              pivot = median3 (A, left, right);
      i = left;
      J = right – 1;
      for(  ;  ; )
         {
              while(A[++i] < pivot);
              while( A[--j]  > pivot);
              if(i  <  j)
                 Swap(A[i], A[j]);
              else
                 break;
         }
      Swap ( A[i], A[right-1]);       //restore pivot
      Qsort(A, left, i – 1);
      Qsort(A, i + 1, right);
  }
}
```

3.5.4 Complexity Analysis of Quicksort

If the number of elements in one subfile is k, then the complexity of the two sufiles will be C(k) and C(n-k-1) where pivot is excluded. The complexity of partitioning (number of comparisons) is linear in the number of items. Thus

$$C(n) = C(k) + C(n-k-1) + a\,n$$
$$C(0) = C(1) = 1.$$

Worst Case Complexity: The pivot is the smallest element all the time. This happens when the array is already sorted (and we don't know this). Then k = 0, C(0) = 1 and

$$C_w(n) = C(n-1) + a\,n + 1$$
$$C_w(n-1) = C(n-2) + a\,(n-1) + 1$$
$$C_w(n-2) = C(n-3) + a\,(n-2) + 1$$
$$\vdots$$
$$C_w(2) = T(1) + 2a + 1$$

$$C_w(n) = C_w(1) + a\sum_{i=2}^{n} i + \sum_{i=2}^{n} 1 = 1 + a(\frac{n(n+1)}{2} - 1) + n - 1$$

Thus $C_w(n)$ is $\Theta(n^2)$, and the algorithm is quadratic in this worst case.

Best Case Complexity: Partitioning produces two sub-files of equal size all the time, i.e.

$C(k) \approx C(n-k-1) \approx C(n/2)$.

$C_b(n) = 2C_b(n/2) + a\,n$

solution to this recurrence relation

$$C_b(n) = an\lg n + n$$

$$C_b(n) \text{ is } \Theta(n\lg n)$$

Average Case Complexity: Each sub-file size, i.e. 1, 2, …, n-1, is equally likely with a probability of $\frac{1}{n}$.

$$C_a(n) = 2\left[\frac{1}{n}\sum_{k=0}^{n-1} C_a(k)\right] + a\,n$$

Multiply by n, then substitute n-1 for n:

$$nC_a(n) = 2\left[\sum_{k=0}^{n-1} C_a(k)\right] + a\,n^2$$

$$(n-1)C_a(n-1) = 2\left[\sum_{k=0}^{n-2} C_a(k)\right] + a\,(n-1)^2$$

Subtract, and divide by n(n+1)

$$\frac{C_a(n)}{n+1} = \frac{C_a(n-1)}{n} + \frac{2a}{n+1}$$

Since the above is valid for all n:

$$\frac{C_a(n-1)}{n} = \frac{C_a(n-2)}{n-1} + \frac{2a}{n}$$
$$\frac{C_a(n-2)}{n-1} = \frac{C_a(n-3)}{n-2} + \frac{2a}{n-1}$$
$$\vdots$$
$$\frac{C_a(2)}{3} = \frac{C_a(1)}{2} + \frac{2a}{3}$$

Adding all the above equations and simplifying

$$\frac{C_a(n)}{n+1} = \frac{C_a(1)}{2} + 2a\sum_{i=3}^{n+1}\frac{1}{i}$$

Now from Chapter 1, $\sum_{i=3}^{n+1}\frac{1}{i} \approx \ln(n+1) - 1$, and

$$C_a(n) = \frac{n+1}{2} + (n+1)\ln n - 2a$$

Average case complexity of Quicksort is

$C_a(n)$ is $\Theta(n \lg n)$.

3.6 Merge Sort

A a divide and conquer type algorithm.
Merging of two ordered (sorted) arrays or sequences A and B into an initially empty array C.

A: 2 5 9 10 B: 6 11 17 18 20 C:

Apos (→ 2) Bpos (→ 6) Cpos

if(A[Apos] < B[Bpos])
    Copy A[Apos] into C[Cpos]
    Apos++ and Cpos++
else
    Copy B[Bpos] into C[Cpos]
    Bpos++ and Cpos++

After one iteration:
A: 2 5 9 10 B: 6 11 17 18 20 C: 2

Apos (→ 5) Bpos (→ 6) Cpos

Note: The above simple algorithm must be modified to check if Apos has reached the end of its array, and if so the rest of B is copied into C. Similarly, if Bpos has reached the end its array, the rest of A must be copied into C. The complexity of merge is $\Theta(n_1 + n_2)$, where n_1 and n_2 are subarray sizes.

```
void MergeMain (int A, int n)
{
  int *TempArray;
  TempArray = (int ) malloc ((n+1)sizeof (int));
  if TempArray !=NULL
      {
       MergeSort(A, TempArray, 1, n);
       free (TempArray);
      }
   else error( )  // Call a function to deal with error
}
void MergeSort( int A[], int TempArray[], int left, int right)
{
  int center;
  if ( left < right)
       {
       center = (right + left)/2;
       MergeSort(A, TempArray, left, center);
       MergeSort(A, TempArray, center+1, right);
       // MergeSort divides  array recursively until each subarray
       // size is 1
       Merge(A, TempArray, left, center+1, right);
       }}

void Merge(int A[], int TempArray[], int Lpos, int Rpos, int Rend)
// Lpos is start of left half,  Rpos is start of right half, Rendis end of
// right half
{
  int i, Lend, Num, TempPos; //
  Lend = Rpos –1;            //
 TempPos = Lpos;           //
 Num = Rend –Lpos +1;   //
                          //        Lpos          Lend Rpos          Rend

while((Lpos <=Lend) && (Rpos <= Rend))
```

```
  if(A[Lpos] <= A[Rpos])
     TempArray[TempPos++] = A[Lpos++];
  else
     TempArray[TempPos++] = A[Rpos++];
while (Lpos <= Lend)    // Copy rest of first half
  TempArray[TempPos++] = A[Lpos++];
while (Rpos <= Rend)    // Copy rest of second half
  TempArray[TempPos++] = A[Rpos++];
// Now copy TempArray back into the original array
for(i =1; i <= Num; i++, Rend--)
  A[Rend] = TempArray [Rend];
}
```

- Note: There is only one TempArray active at any point since TempArray is not declared locally to the Merge function. If this were not the case, we would need many (i.e. lg n) temporary arrays, or would have to do dynamic allocations and freeing.
- Note: Merge sort is not an in place operation since a copy of the array is made.

3.6.1Complexity Analysis of Merge Sort

To simplify assume $n = 2^k, \quad k = 1, 2, \cdots$

$k = \lg n$

Complexity of sorting n items C(n) consists of sorting recursively two sub arrays each of size n/2, and a constant

$$C(n) = 2C(n/2) + a\,n$$

with $C(1) = 1$, where a is a proportionality constant.
Solution:

$$C(n) = n \lg n + a\,n$$

Therefore C(n) is $\Theta(n \lg n)$.

- If n is not a power of 2, it can still be shown that C(n) is $\Theta(n \lg n)$.

- Merge sort requires linear extra space for copying of TempArray into A which slows the algorithm.

- Copying can be avoided by changing the roles of A and TempArray at alternate levels of recursion.

- Because of the extra overhead associated with copying, merge sort is not a particularly good sorting method when all data fit into computer memory.
- However, it is the only choice for external sorting where all data do not fit into memory, and chunks of data must be processed sequentially.

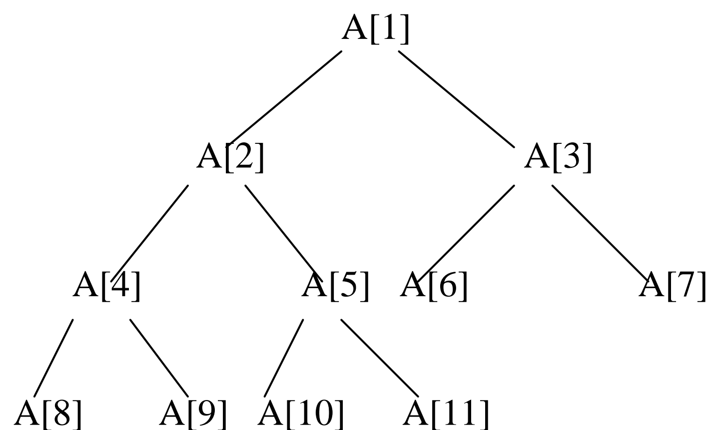
3.7 Heap Sort

A divide and conquer algorithm.

3.7.1 Heaps

Heap is a data structure that is represented by a binary tree but is implemented using arrays. A heap has two properties:

- **Structure Property**: A heap is a complete binary tree such that:
  - All tree levels, except possibly the last level, have the maximum number of nodes and are completely filled.
  - The last level is filled from left to right.



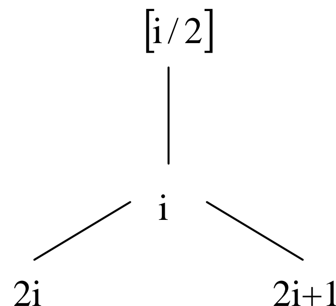
If one node in the last level, $n = 2^h$ When the last level is completely filled

$$n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1.$$

$$2^h \leq n \leq 2^{h+1} - 1, \quad \text{and} \quad h = [\lg n]$$

Note: Because of the regularity of the complete binary tree, it can be implemented using an array, called a heap array H.

Note: Element positions:



- **Order Property**: Heap types – max heap and min heap. In max heaps $A[i/2] > A[i]$. Thus in a max heap the key at the root is the largest.

  In a min heap $A[i/2] < A[i]$, and the key at the root is the smallest.

- **Heap Operations:** Two main heap operations: insert and delete.

Insert adds the new element at location n+1, and move it up, if necessary until the heap order property is satisfied.

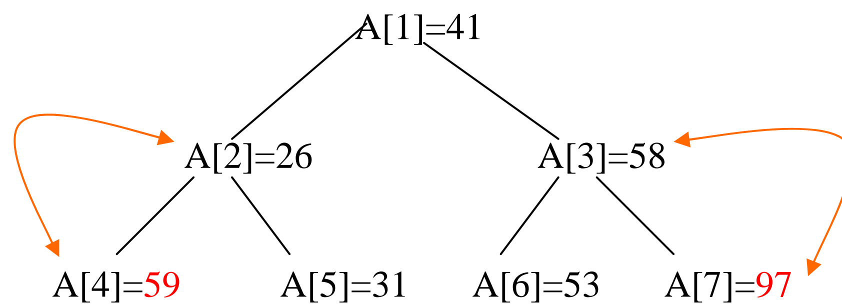
Delete removes the element at the root, reduces the current size by one, and place the last element H[n] at the now

empty root, and moves it down, if necessary to satisfy the order property.

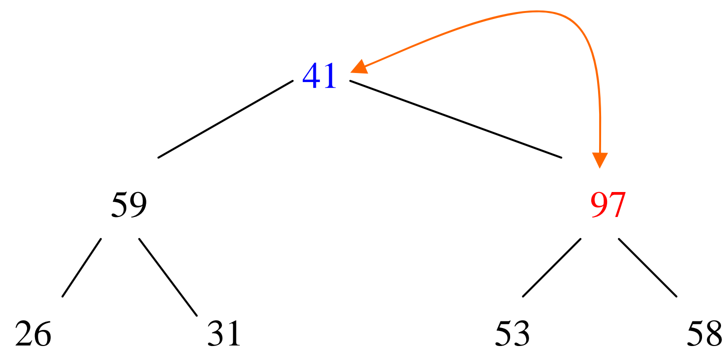
- A sketch of the heap sort algorithm is

```
for i = n/2 down to 1
Percolate down A[i] // Build a heap A from an
                    //unsorted array
for i = 1 to n
  Move A[1] to A[n-i]
     Reheap from A[1] to A[n-i-1]
               // A[1] is empty before reheap
```

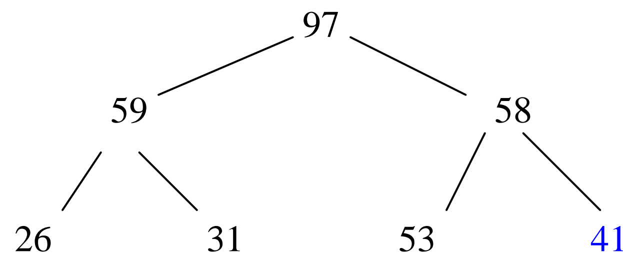
Example:
This example shows how to build a max heap from a given array.



We start at i= $\lceil n/2 \rceil$=3, compare 58 with the maximum of its children which is 97, and since 97 > 58, swap 97 with 58. Next we go to i-1 =2, and compare 26 with max(59, 31)=59 and swap 26 with 59. The result is



Now decrement i to 1, and compare 41 with max(59, 97) =97, then swap 97 with 41. Since 41 is smaller than max(53, 58)=59, it has to be swapped with 58. The result is a maximum heap as follows



The corresponding heap array is

Position	0	1	2	3	4	5	6	7
Element		97	59	58	26	31	53	41

Now swap the root 97 with element in last position 41, and reheap to put the new root (41) in its appropriate location among items in positions 2 to 6. The element 97 will remain there as one sorted element. After reheaping the array will become

Position	0	1	2	3	4	5	6	7
Element		59	41	58	26	31	53	97

Now the root 59 is swapped with element in position 6, to get

Position	0	1	2	3	4	5	6	7
Element		53	41	58	26	31	59	97

At this time, two elements are sorted (59 and 97), and a reheaping must be done to put 53 at the correct location. The procedure is then repeated until all elements are sorted.

```
void HeapSort(int A[], int n)
{
  for (int i=n/2; i > 0; i--)  //build a max heap
  Percolate(A, i, n);

 for(i = n; i >=2; i--)
   {
       Swap(A[1], A[i]);
       Percolate( A, 1, i-1);   // reheap
   }
}

void Percolate ( int A, int i, int n)
{
int Child, Temp;

Temp = A[i];
for( ; i*2 <= n; i = Child)
{
       Child = i*2;
       if(Child != n && A[Child+1] > A[Child] )  //find max
                                                 // child
           Child ++;
       if ( Temp < A[Child])
```

```
            A[i] = A[Child];
        else
                break;
}
        A[i] = Temp;
```

3.7.2 **Complexity Analysis of Heap Sort**

Heap sort consists of
1. building the heap $C_{build}(n)$,
2. swapping the root $C_{swap}(n)$, and
3. reheaping $C_{reheap}(n)$.

In the build heap process, each node may have to be percolated down all the way to a leaf. The number of comparisons for each node is $2h_i$, where h_i is the height of the node i, and 2 is for the comparisons (one to find the max child, and another to compare max child with the node). For a perfect tree total number of comparisons is

$$C_{build}(n) = 2\sum_{i=0}^{h} 2^i (h-i)$$

$$= 2\left[h + 2(h-1) + 4(h-2) + 8(h-3) \cdots + 2^{h-1}(1)\right]$$

$$2C_{build}(n) = 2\left[\quad 2h + 4(h-1) + 8(h-2) + 16(h-3) \cdots + 2^h(1)\right]$$

$$2C_{build}(n) - C_{build}(n) =$$

$$C_{build}(n) = 2\left[-h + 2 + 4 + 8 + \cdots + 2^{h-1} + 2^h\right] = 2[(2^{h+1} - 1) - (h+1)]$$

For a perfect binary tree $(2^{h+1} - 1) = n$

$$C_{build}(n) = 2n - 2\lg(n+1) \approx 2n - 2\lg n$$

2. $C_{swap}(n) = n$ since each element at the root is swapped once.

3. Reheap consists of moving the new root to its correct position. This requires 2h comparisons for each element in the worst case since the root is at height h, and moving down each level requires 2 comparisons.

$$C_{reheap}(n) = 2nh = 2n[\lg(n+1) - 1] \approx 2n\lg n$$

Therefore

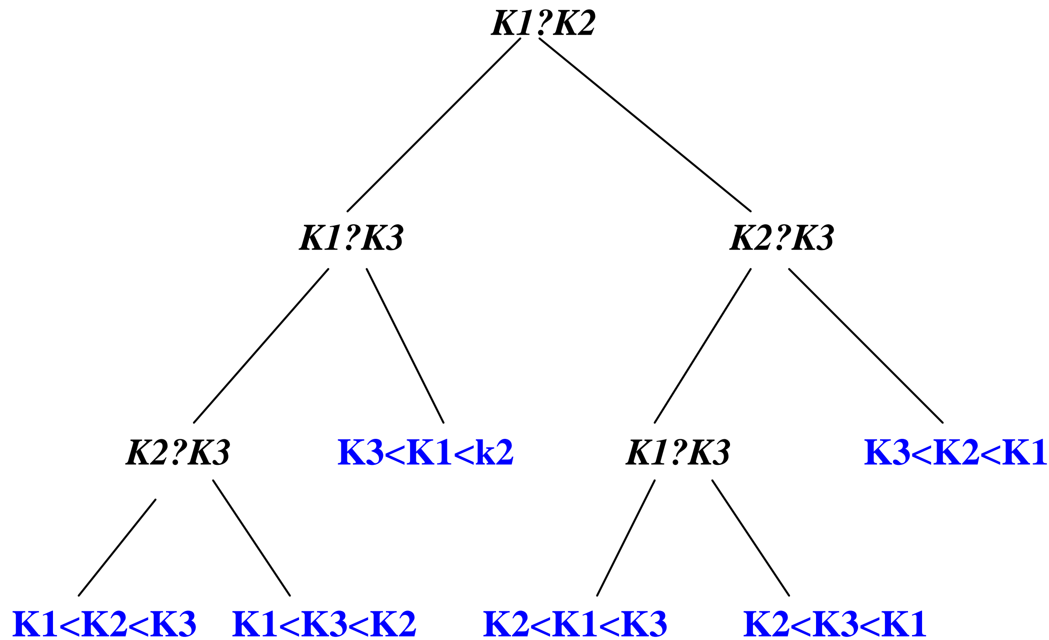
$$C_w(n) = 3n + 2n\lg n - 2\lg n \in \Theta(n\lg n)$$

3.8 Lower Bounds for Sorting

- **Question:** Is it possible to devise an algorithm that uses key comparisons only, and that can do the sorting with a worst case complexity of better than $\Theta(n\lg n)$ for an arbitrary input of size n?

Use of decision trees. Associate with a sorting algorithm an n, a decision tree that describes the sequence of comparisons carried out by the algorithm.

Example: Suppose we have 3 keys K1, K2, K3 to be sorted. The associated decision tree is shown below, where Ki ? Kj means compare the two key, and if Ki < Kj go to the left child, otherwise go to the right child.



- In general , the action of any sorting method is to find a path from the root to a leaf. The tree must have at least n! leaves because there are n! ways in which the keys can be permutated.

- The number of comparisons in the worst case is equal to the depth of deepest leaf.

- **Result 1**: A binary tree of height h has $L \le 2^h$ leaves.

  To prove the above statement note that
  $h = 0 \rightarrow L = 1$ (root only)

  $h > 0 \rightarrow L = L_l + L_r$ (i.e. # of leaves in left and right subtrees).

  Now $$h_l \le h - 1, \quad h_r \le h - 1$$

  $L_l \le 2^{d-1}, \quad L_r \le 2^{d-1}$

  $$L = L_l + L_r \le 2^h$$

This gives

$$h \ge \lceil \lg L \rceil$$

- Result 2: Any sorting algorithm that uses only comparisons between keys requires at least $\lceil \lg n! \rceil$ comparisons in the worst case. Alternatively it requires $\Omega(n \lg n)$ comparisons.

  To prove, observe that a decision tree has n! leaves. The number of comparisons in the worst case

$$C_w(n) \geq h \geq \lceil \lg L \rceil = \lceil \lg n! \rceil$$

Also

$$n! \geq n(n-1)(n-2)\cdots(\lceil n/2 \rceil)$$

$$\geq (n/2)^{n/2}$$

Thus

$$C_w(n) \geq \frac{n}{2}\lg(n/2) = \frac{n}{2}\lg n - \frac{n}{2}$$

Finally,

$$C_w(n) \text{ is } \Omega(n \lg n)$$

This is called "information theoretic lower bound".

- Since the average height of leaves of a binary tree can be shown to satisfy $h_a \leq \lg L$, then we have $C_a(n)$ is $\Omega(n \lg n)$.

- The information theoretic lower bound, $\Omega(n \lg n)$, assumes that only key comparisons are made, and no other information is available on the array elements. If additional information is known about the input, more efficient algorithms can be devised. Bucket sort is an example of such an algorithm.

3.8.1 Bucket Sort

Given the integer array A[i] , i= 0, 1, …, n-1, suppose that it is known that $0 \leq A[i] < m$ for all i. The following algorithm is a simple version of bucket sort:

1. Initialize an array B (called bucket) of size m to a value outside the range of 0 to m , e.g. B[j] = -1 for j=0, 1, …, m-1.
2. Read A[i], starting with A[0], and increment B[A[i]] by one until A[n-1] is read
3. Scan B and print (or copy back into A) the indexes in B that are in the 0 to m range.

Example:

A: 8 5 7 10 2 5

$0 \le A[i] < 10, \quad m = 10$

B[j] = -1, j=0, 1,2, …, m-1

B[8] =1, B[5]=1, B[7] =1, B[10]=1, B[2]=1, B[5]=5

Now B[0] and B[1] are -1, so don't print any thing, then B[2] = 1 so print 2. Next B[3] =B[4] = -1 so do not print. However, B[5] =2, so print 5, 5. Continuing, this gives: 2, 5, 5, 7, 8, 10.