

[ Chapter 7]Introduction toComputational Complexity:The Sorting Problem

# 4

### **Two Types of Complexity Analysis**

- Algorithm Analysis
  - Different approaches to solve the same problem with the hope of finding efficient algorithms.
  - Time and space complexity.
- Problem Analysis
  - In matrix multiplication problem, there is an algorithm in  $\Theta(n^3)$ 
    - But the time complexity of Strassen's is  $\Theta(n^{2.81})$  and it is even possible to be in  $\Theta(n^{2.38})$ .
  - Therefore, we do not want to know the complexity of an algorithm, but the complexity of a problem.
  - It is a computational complexity analysis.
    - The result is stated by a lower bound for a problem.
    - (e.g.)  $\Omega(n^2)$  for the matrix multiplication. Meaning?



### **Computational Complexity**

#### Objective

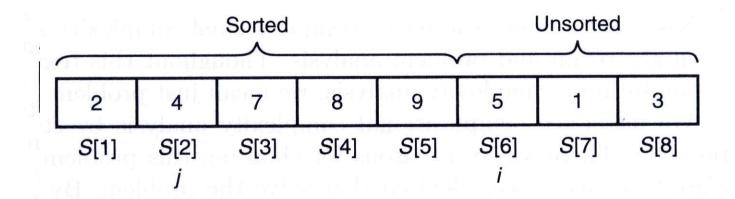
- To develop a  $\Theta(f(n))$  algorithm for the problem of  $\Omega(f(n))$ .
- It is impossible to develop an algorithm that is lower than a lower bound for the problem.
- Example: Sorting problem
  - Exchange sort:  $\Theta(n^2)$
  - Merge sort:  $\Theta(n \lg n)$
  - The lower bound for the sorting problem is  $\Omega(n \lg n)$
  - Fortunately, we found an algorithm that touches the lower bound of the sorting problem.
- Our new objective
  - How to find such a lower bound

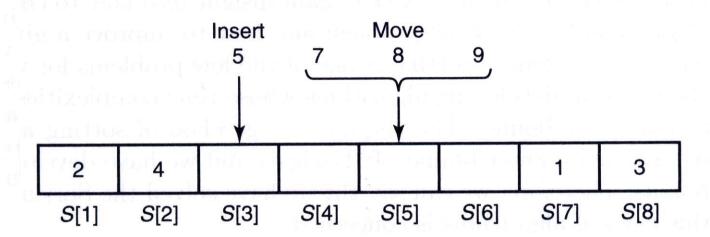
# Insertion Sort (1/2)

- By inserting records in an existing sorted array.
  - ProblemSort n keys in non-decreasing order.
  - Inputs
    Positive integer n; array of keys S indexed from 1 to n.
  - Outputs
     the array S containing the keys in non-decreasing order.



### **Insertion Sort (2/2)**





### **Inserting Sort Algorithm 7.1**

```
void insertionsort(int n, keytype S[]){
    index i,j;
    keytype x;
    for (i=2; i \le n; i++) {
      x = S[i];
      j = i - 1;
      while (j>0 \&\& S[j]>x) {
         S[j+1] = S[j];
        j−−;
      S[j+1] = x;
```

### **Inserting Sort Algorithm Analysis**

- Worst-case Time Complexity Analysis
  - At most (i-1) comparisons for each i.

$$W(n) = \sum_{i=2}^{n} (i-1) = \frac{n(n-1)}{2}$$

Average-case Time Complexity Analysis

Index 
$$i$$
  $i-1$   $\cdots$  2 1 #Comparison 1 2  $\cdots$   $i-1$   $i-1$ 

The number of comparisons to insert 
$$x$$

$$1\frac{1}{i} + 2\frac{1}{i} + \dots + (i-1)\frac{1}{i} + (i-1)\frac{1}{i} = \frac{1}{i}\sum_{k=1}^{i-1} k + \frac{i-1}{i} = \frac{(i-1)i}{2i} + \frac{i-1}{i} = \frac{i+1}{2} - \frac{1}{i}$$

Therefore, the average number of comparisons to insert

$$\sum_{i=2}^{n} \left(\frac{i+1}{2} - \frac{1}{i}\right) = \sum_{i=2}^{n} \frac{i+1}{2} - \sum_{i=2}^{n} \frac{1}{i} \approx \frac{(n+4)(n-1)}{4} - \ln n \approx \frac{n^2}{4}$$

## Selection Sort

- By selecting records in order and putting them in their proper positions in an array.
  - ProblemSort n keys in non-decreasing order.
  - Inputs
     Positive integer n; array of keys S indexed from 1 to n.
  - Outputs
     the array S containing the keys in non-decreasing order.

### **Selection Sort Algorithm 7.2**

```
void selectionsort(int n, keytype S[]) {
    index i,j,smallest;
    for (i=1; i<=n-1; i++) {
       smallest = i;
       for (j=i+1; j \le n; j++) {
         if(S[j] < S[smallest])</pre>
           smallest = j;
       exchange S[i] and S[smallest];
```

### **Selection Sort Algorithm Analysis**

- Every-case Time Complexity Analysis
  - The number of comparisons

Index 1 2 
$$\cdots$$
  $n-1$  #Comparison  $n-1$   $n-2$   $\cdots$  1

• Therefore, 
$$T(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

- The number of assignments
  - 3 assignments for an exchange
  - Therefore, T(n) = 3(n 1)

## 4

### **Analysis summary for Exchange Sort, Insertion Sort, and Selection Sort**

Algorithm	Comparisons of Keys	Assignments of Records	Extra Space Usage
Exchange Sort	$T\left(n\right) = \frac{n^2}{2}$	$W\left(n\right) = \frac{3n^2}{2}$	In-place
. rebno gnase,		$A\left(n\right) = \frac{3n^2}{4}$	
Insert Sort	$W\left(n\right) = \frac{n^2}{2}$	$W\left(n\right) = \frac{n^2}{2}$	In-place
	$A\left(n\right) = \frac{n^2}{4}$	$A\left(n\right) = \frac{n^2}{4}$	And the proof of the second
Selection Sort	$T\left(n\right) = \frac{n^2}{2}$	$T\left( n\right) =3n$	In-place



## Lower Bounds for Algorithms that Remove at Most one inversion per comparison

- [1, 2, 3, ... n]
  - n! permutations
  - If i < j and  $k_i > k_j$  then it's an inversion.
  - (e.g) How many inversions in [3, 2, 4, 1, 6, 5]?
    - 5 inversions
- Thm 7.1 Any algorithm that sorts n distinct keys only by comparisons of keys and removes one inversion (at most) after each comparison must in the worst case do at least  $\frac{n(n-1)}{2}$  comparisons of keys and  $\frac{n(n-1)}{2}$  on the average.

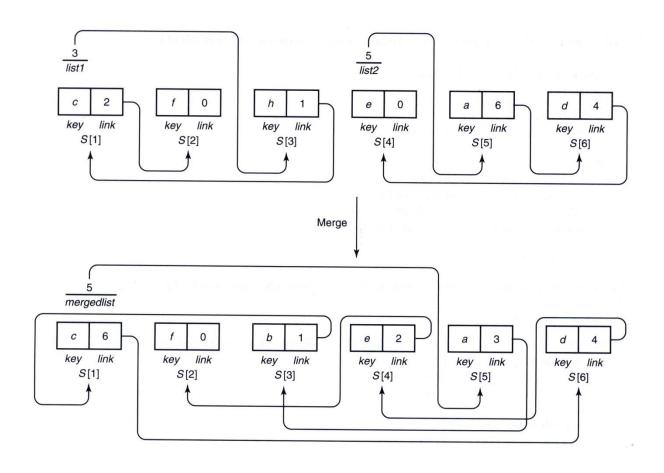
#### **Proof for Thm. 7.1**

- Worst case
  - The permutation with  $\frac{1}{2}$  n(n-1) inversions, [n, n-1, ... 2, 1] is the case.
- Average case
  - $P = [k_1, k_2, ..., k_n]$ , and its transpose  $P^T = [k_n, ..., k_2, k_1]$
  - An arbitrary pair (s, r) belongs to P or  $P^T$ , always.
  - In other words,  $I(P) + I(P^T) = \frac{1}{2} n(n-1)$ .
  - Therefore, on the average, there are  $\frac{1}{4} n(n-1)$ .

#### **Merge Sort Revisited**

- Think [3, 4] and [1, 2].
  - After 3 and 1 are compared, 1 is put in the 1<sup>st</sup> slot, then two inversions, (3, 1) and (4, 1) are removed.
  - After 3 and 2 are compared, (3, 2) and (4, 2) are removed
- The worst case time complexity
  - W(h, m) = W(h) + W(m) + h + m 1=  $n \lg n + n - 1$
  - Removing >= 1 inversions after a comparison.
  - We gained significantly!
- 3 improvements
  - Dynamic programming version
  - Linked list version
  - More complex merge algorithm

# Figure 7.3 Merging using links. Arrows are used to show how the links work. The Keys are letters to avoid confusion with indices.



### **Quick Sort Revisited**

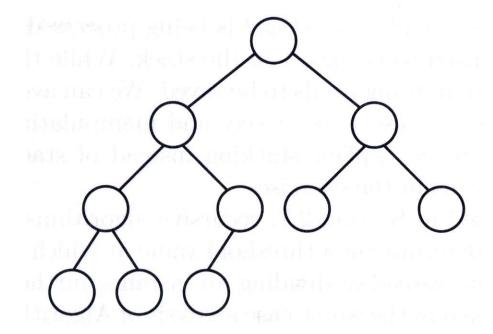
Time complexity

$$W(n) = \Theta(n^2)$$
 and  $A(n) \approx 1.38(n+1) \lg n$ 

- A(n) is not much worse than Mergesort().
- Technically, QS is not an in-place sorting.
- 5 different ways to reduce the amount of extra space.
  - Check yourself that how much the extra space complexity can be reduced!!
  - The worst-case height of the stack is  $\Theta(\lg n)$

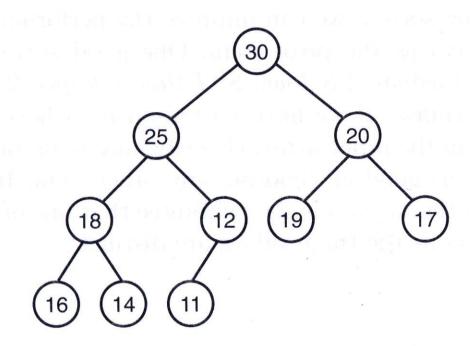
# Heap Sort

- Unlike MS and QS, HS is truly an in-place sorting algorithm.
  - Worst-case time complexity is  $\Theta(n \lg n)$
- Complete binary tree
  - All internal nodes have 2 children.
  - All leaf nodes have a depth of d.
- Essentially CBT
  - It's a complete binary tree up to a depth of d-1.
  - The nodes with depth d are as far to the left as possible.
- Heap
  - An ECBT with heap property.

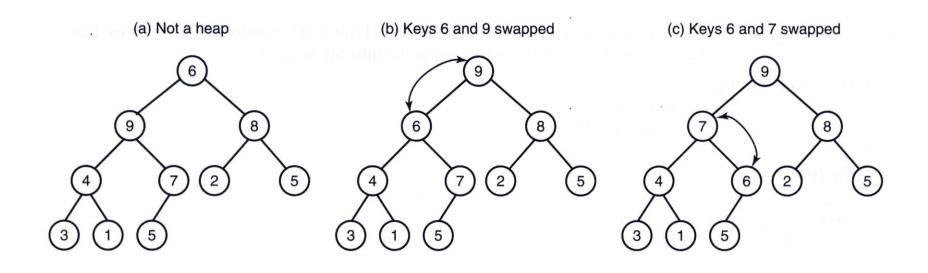


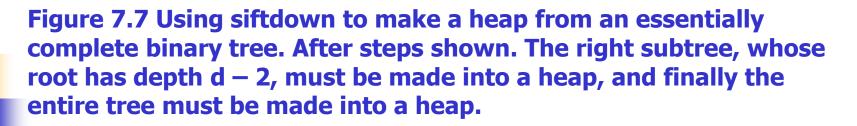


#### Figure 7.5 A heap.

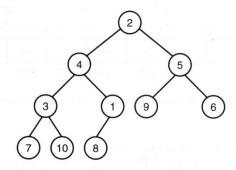


## Figure 7.6 Procedure *siftdown* sifts 6 down until the heap property is restored.

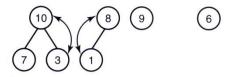




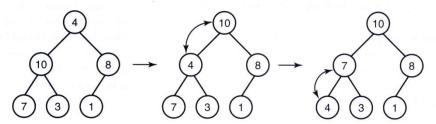
(a) The initial structure



(b) The subtrees, whose roots have depth d-1, are made into heaps.



(c) The left subtree, whose root has depth d-2, are made into a heap.



### Figure 7.8 The array representation of the heap in Figure 7.5.

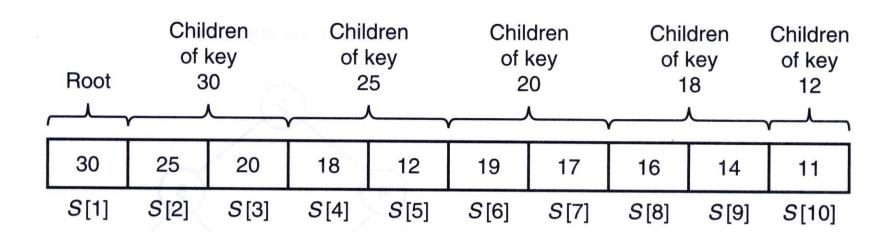


Figure 7.9 An illustration using n = 8, Showing that if an essentially complete binary tree has n node and n is a power of 2, then the depth d of the tree is lg n, there is one node with depth d, and that node has d ancestors. The three ancestors of that node are marked "A".

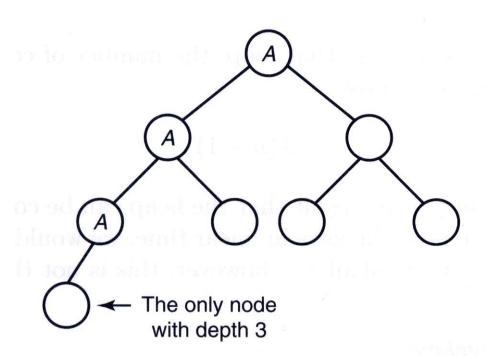
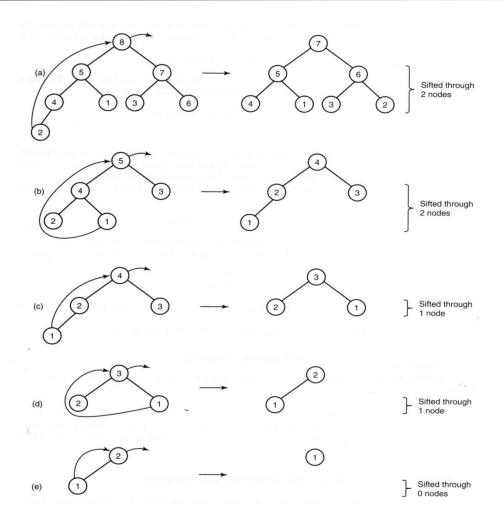


Figure 7.10 Removing the Keys from a heap with eight nodes. The removal of the first Key is depicted in (a): the fourth in (b): the fifth in (c): the sixth in (d): and the seventh in (e). The key moved to the root is sifted through the number of nodes shown on the right.



#### leapsort void siftdown(heap& H) { node parent, largerchild; parent = root of H; largerchild = parent's child containing larger key; while (key at parent is smaller than key at largerchild) { exchange key at parent and key at largerchild; parent = largerchild; largerchild = parent's child containing larger key; keytype root(heap& H) { keytype keyout; keyout = key at the root; move the key at the bottom node to the root; delete the bottom node; siftdown(H); return keyout;

```
void removekeys(int n, heap H, keytype S[]){
  index i;
 for(i=n; i>=1; i--)
S[i] = root(H);
void makeheap(int n, heap& H) {
  index i;
  heap Hsub;
  for(i=d-1; i>=0; i--)
    for (all subtree Hsub whose roots have depth i)
      siftdown(Hsub);
}
void heapsort(int n, heap H, keytype S[]) {
  makeheap(n,H);
  removekeys (n,H,S);
```

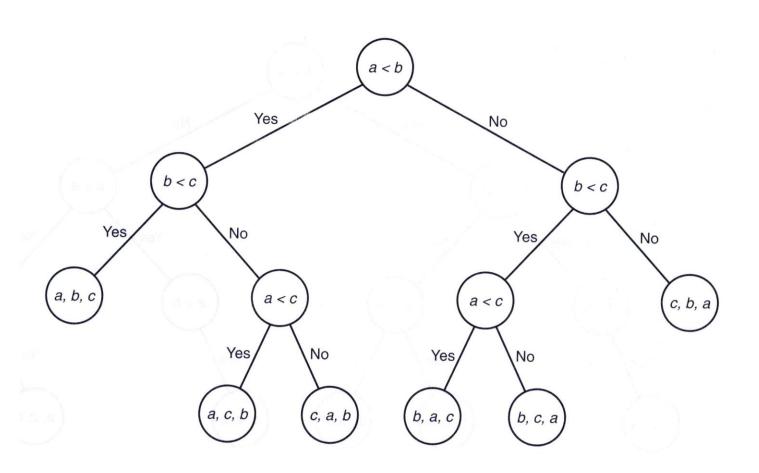
### **Heapsort Analysis: How to?**

- S: the greatest number of sift operations.
  - S1 = The number of sift operations for a node before d-1
  - S2 = The number of sift operations for the last node at d
  - S = S1 + S2
- At least 2 comparisons per siftdown
  - Therefore, the total number of worst-case comparison operations is 2 times S.
- Analysis of removekeys()
  - R: the number of comparisons for removekeys()
- Heapsort all together
  - S+R is  $\Theta(n \lg n)$

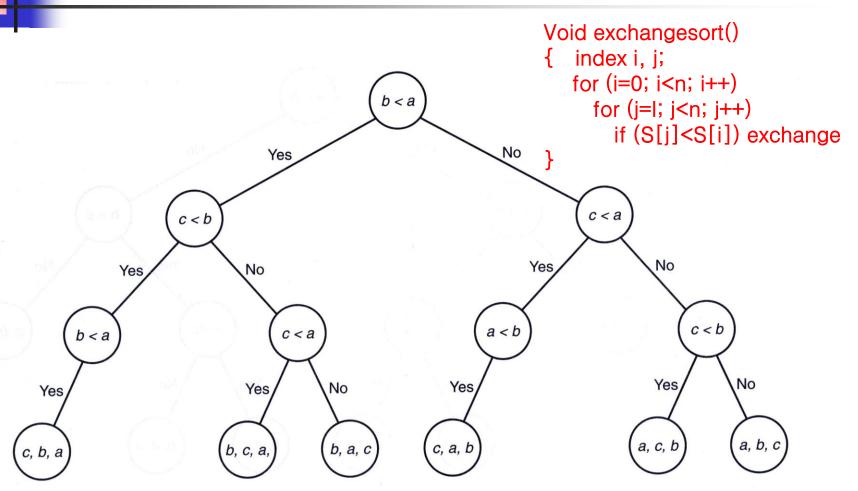
## Table 7.2 Analysis summary for Θ (*n lg n*) sorting algorithms\*

Algorithm	Comparison of Keys	Assignments of Records	Extra Space Usage
Mergesort	$W\left(n\right) = n\lg n$	$T\left(n\right) = 2n\lg n$	$\Theta(n)$ records
(Algorithm 2.4)	$A\left(n\right) = n\lg n$		
Mergesort	$W\left(n\right) = n\lg n$	$T\left( n\right) =0^{\dagger}$	$\Theta(n)$ links
(Algorithm 7.4)	$A\left(n\right) = n\lg n$		. ,
Quicksort	$W\left(n\right) = n^2/2$		$\Theta(\lg n)$ indices
(with improvements)	$A(n) = 1.38n \lg n$	$A\left(n\right) = 0.69n \lg n$	
Heapsort	$W\left(n\right) = 2n\lg n$	$W\left(n\right) = n\lg n$	In-place
	$A\left(n\right) = 2n\lg n$	$A(n) = n \lg n$	

### Figure 7.11 The decision tree corresponding to procedure *sortthree*.



### Figure 7.12 The decision tree corresponding to Exchange Sort when sorting three Keys.



# 1

## **Lower Bounds for Sorting Only by Comparison of Keys**

#### Terms

- Decision tree
- A decision tree is valid if there is a path from the root to a leaf node for each permutation of n keys.

#### Lemma 7.1

 To every deterministic algorithm for sorting n distinct keys, there corresponds a pruned, valid, binary decision tree containing exactly n! leaves.

#### Lemma 7.2

 The worst-case number of comparisons done by a decision tree is equal to its depth.

#### Lemma 7.3

- Suppose m is the number of leaves in a binary tree and d is its depth.
- Then  $d \ge \lceil \lg m \rceil$

# 1

## Lower Bounds for Sorting Only by Comparison of Keys (Cont'd)

- Theorem 7.2
  - Any deterministic algorithm that sorts n distinct keys only by comparison of keys must in the worst case do at least  $\lg(n!)$  comparison of keys.
- Lemma 7.4
  - For any positive integer n,

$$\lg(n!) = \lg[n(n-1)(n-2)\cdots 2]$$

$$= \sum_{i=2}^{n} \lg i$$

$$\geq \int_{1}^{n} \lg x \, dx$$

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

$$\geq n \lg n - 1.45n$$

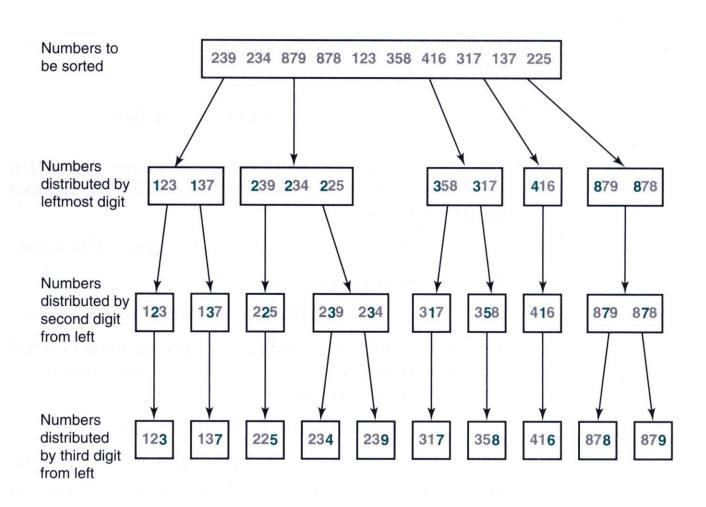


## Lower Bounds for Sorting Only by Comparison of Keys (Final)

#### Theorem 7.3

Any deterministic algorithm that sorts n distinct keys only by comparison of keys must in the worst case do at least,  $\lceil n \lg n - 1.45n \rceil$  comparisons of keys.

### Figure 7.14 Sorting by distribution while inspecting the digits from left to right.



### Figure 7.15 Sorting by distribution while inspecting the digits from right to left.

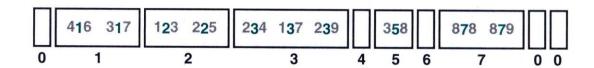


239 234 879 879 123 358 416 317 137 225

Numbers distributed by rightmost digit



Numbers distributed by second digit from right



Numbers distributed by third digit from right

