Sorting Algorithms

1. Insertion Sorting

One of the simplest sorting algorithms. It consists of $n-1$ passes. For pass $j=2$ through $j=n$, it ensures that the elements in positions 1 through j in sorted order. Insertion sort makes use of the fact that elements in positions from 1 to $j-1$ are already in sorted order.

1. Insertion Sorting (cont.)

Algorithm Insertion(A)

1. $n \leftarrow length[A]$
2. for $j \leftarrow 2$ to n do
3. $key \leftarrow A[j]$
4. /* Insert $A[j]$ into sorted sequence $A[1\ldots j-1]$ */
5. $i \leftarrow j-1$
6. while $i > 0$ and $A[i] > key$ do
7. $A[i+1] \leftarrow A[i]$
8. $i \leftarrow i-1$
9. $A[i+1] \leftarrow key$

Time complexity analysis:

$\sum_{j=2}^{n}(2c_1 + \sum_{k=1}^{j-1} 2c_1 + c_1) + c_2 \leq \sum_{j=2}^{n}(2c_1 + 2(j-1)c_1 + c_1) + c_2 = O(n^2)$, where c_1 and c_2 are constants.

Example: $34, 8, 64, 51, 32, 21$

2. Bubble Sorting

It consists of $n-1$ passes. For pass $j=1$ through $j=n-1$, it ensures that the elements in positions 1 through j in sorted order.

Algorithm Bubble(A)

1. $n \leftarrow length[A]$
2. for $j \leftarrow 1$ to $n-1$ do
3. $key \leftarrow A[j]$
4. /* Insert the smallest element from $A[j+1..n]$ into sorted sequence $A[1\ldots j]$ */
5. for $i \leftarrow j+1$ to n do
6. if $A[i] < key$ then
7. $key \Leftrightarrow A[i]$
8. $A[j] \leftarrow key$

2. Bubble Sorting (cont.)

Time complexity analysis:

$\sum_{j=1}^{n-1}(c_1 + \sum_{j+1}^{n}(c_3 + 3c_1) + c_1) + c_2 = O(n^2)$,

where c_1 c_2, and c_3 are constants.

Example: $34, 8, 64, 51, 32, 21$

3. Shell sorting

Shellsort is referred to as diminishing increment sort, it works by comparing elements that are distance; the distance between comparisons decreases as the algorithm runs until the last phase, in which adjacent elements are compared. It uses a sequence, $h_1, h_2, \ldots, h_t$, called the increment sequence.

The general strategy to h_k-sort is for each position, i in h_k, $h_k+1, \ldots, n$, place the element in the correct spot among i, $i-h_k, i-2h_k$, etc.

A typical increment sequence is $h_t = \lfloor n/2 \rfloor$, and $h_k = \lfloor h_{k+1}/2 \rfloor$, and $h_1 = 1$, $k = 1, 2, \ldots, t$.

3. Shell Sorting (cont.)

```
Algorithm Shell(A)
 1   n ← length[A]
 2   gap = ⌊n/2⌋;
 3   while gap > 0 do
 4   /* each subsequence with skip gap elements is sorted by bubble sorting*/
 5         for   i ← 1 to n do
 7               temp ← A[i]
 8               j ← i
 9               while j ≥ i and j ≤ n − gap do
10                     if    temp > A[j] then
11                           temp ⇔ A[j]
12                     j ← j + gap
13               A[i] ← temp
14         gap ← gap/2
```


3. Shell Sorting (cont.)

Time complexity analysis:

$O(\sum_{k=1}^{t}(h_k(n/h_k)^2)) = O(n^2 \sum_{k=1}^{t} 1/h_k) = O(n^2)$,

while $\sum_{k=1}^{t} 1/h_k < 2$, if the increment sequence is a geometric series with common ratio 2.

Example: $81, 94, 11, 96, 12, 35, 17, 95, 28, 58, 41, 75, 5$

4. Mergesort

Given two **sorted arrays** $A[1..n]$ and $B[1..m]$, merge them into a sorted array $C[1..m+n]$. The merge algorithm is as follows.

```
Algorithm Merge(A, B, C)
 1   i ← 1; j ← 1;k ← 1;
 2   while i ≤ length[A] and j ≤ length[B] do
 3         if    A[i] ≤ B[j] then
 4               C[k] ← A[i]
 5               i ← i + 1;
 6               else   C[k] ← B[j]
 7                     j ← j + 1;
 8         k ← k + 1;
 9         Copy the rest of A or B to C;
```

Clearly the running time of merge algorithm is $O(m+n)$.

4. Mergesort (cont.)

Merge sort is an optimal sorting algorithm. Let $a_1, a_2, \ldots, a_n$ be a real sequence. The basic steps of mergesort are as follows:

Step 1. The original sequence of elements is partitioned into two subsequences, and the length of each of them is roughly equal to $n/2$.

Let $S_1 = \{a_1, a_2, \ldots, a_{\lceil n/2 \rceil}\}$ and $S_2 = \{a_{\lceil n/2 \rceil} + 1, \ldots, a_n\}$;

Step 2. Sort S_1 and S_2 using mergesort recursively.

Step 3. merge the sorted S_1' and S_2'.

Example: $81, 13, 92, 43, 31, 57, 65, 26, 75, 3$.

4. Mergesort (cont.)

Analysis of the Time Complexity of Mergesort: Assume $n = 2^k$, then $k = \log n$.

$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + cn = O(n \log n)$. If $n \neq 2^k$, assume that $2^k \leq n < 2^{k+1}$, let

$n' = 2^{k+1} > n$, then $T(n) \leq T(n') = O(n \log n)$.

5. Quicksort

Quick sort is a fast sorting algorithm in practice and very useful.

Let $a_1, a_2, \ldots, a_n$ be a real sequence. The basic steps of quicksort are as follows:

Step 1. pick an element $x = a_i$ randomly (usually the first element a_1) as the **pivot element**;

Step 2. partition the sequence into two disjoint subsequences, $S_1 = \{a_i : a_i \leq x\}$ and $S_2 = \{a_j : a_j \geq x\}$;

Step 3. sort S_1 and S_2 recursively, using quicksort;

Step 4. merge the sorted S_1, $\{x\}$, and S_2.

Example: $81, 13, 92, 43, 31, 57, 65, 26, 75, 3$.

5. Quicksort (cont.)

The running time of the quicksort algorithm is heavily effected by the choice of the pivot element.

- Consider a case where the sequence is sorted initially, to see what happens?
- If the median of the sequence is chosen as the pivot element, what is its running time?

5. Quicksort (cont.)

The time complexity of the quicksort algorithm is analyzed as follows. Let $T(n)$ be the time complexity of the quicksort algorithm for sorting n elements. Then,

$$T(n) = T(i) + T(n-i-1) + cn, \tag{1}$$

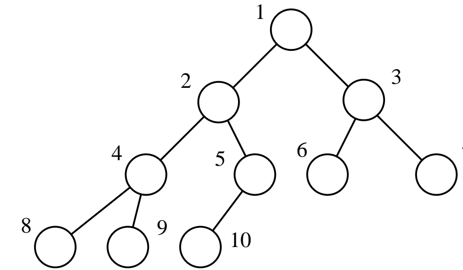
where $|S_1| = i$ and $|S_2| = n-i-1$.

- Worst Case: $T(n) = T(n-1) + cn = O(n^2)$
- Best Case: $T(n) = T(n/2) + cn = O(n \log n)$
- Average Case: $T(i) = T(n-i-1) = \frac{1}{n}\sum_{j=1}^{n-1} T(j)$ and $T(n) = O(n \log n)$

6. Heap Sort

A binary heap is a nearly complete binary tree, usually stored as an array. Each node (array element) contains a value (key) and perhaps data associated with the key.

The tree is completely filled on all levels except possibly the lowest, which is filled from the left.

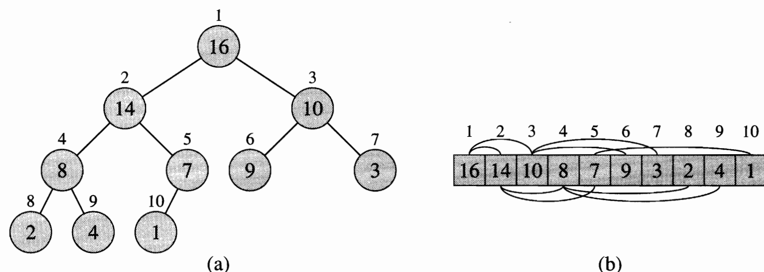


Array indexes are assigned top to bottom, left to right, as shown.

6. Heap Sort (continued)

There are two types of heap, the max-heap and the min-heap.

- For a **max-heap**, every node other than the root has a key less than or equal to the key in its parent.
- For **min-heap**, every node other than the root has a key greater than or equal to the key in its parent.



A max-heap (Cormen, p128)

6. Heap Sort (continued)

The tree structure does not need pointers, since the relatives of a node are stored in array positions that can be calculated:

- The root of the tree is $A[1]$.
- Given the index i of a node, we have:
  - $\text{PARENT}(i) = \lfloor i/2 \rfloor$, except for $i = 1$.
  - $\text{LEFT}(i) = 2i$, if the left child exists.
  - $\text{RIGHT}(i) = 2i + 1$, if the right child exists.

Basic properties of a max-heap:

- The key in the root is the largest key.
- Given any node of a max-heap, that node and all its descendants forms a max-heap.

6. Heap Sort (continued)

Let A be an array and let i be the index of an element.

- MAX-HEAPIFY(A,i) – Make the subtree rooted at i into a max-heap, assuming that the subtrees rooted at LEFT(i) and RIGHT(i) are max-heaps.
- BUILD-MAX-HEAP(A) – Make the unordered array A into a max-heap.
- HEAP-MAXIMUM(A) – Return the largest key in a max-heap A.
- MAX-HEAP-INSERT(A,key) – Insert a new key into a max-heap A.
- HEAP-EXTRACT-MAX(A) – Delete the largest key from a max-heap A.
- HEAP-INCREASE-KEY(A,i,key) – Increase the key of element i in a max-heap A.

6.1 Fixing errors in max-heaps

When an element of a max-heap violates the max-heap property, it might be possible to fix the error using one or more of the following operations.

- *Swap up.* If a key is greater than the key in the parent, swap them.
- *Swap down.* If a key is less than the key in a child, swap the key with the greatest key of a child.

6.2 MAX-HEAPIFY operation

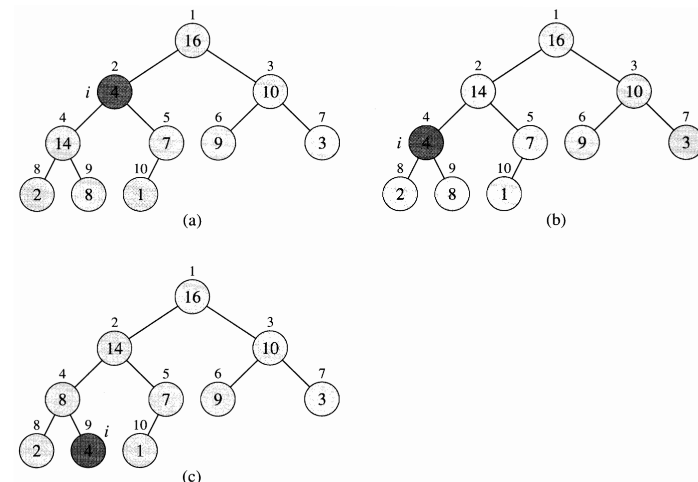
MAX-HEAPIFY(A,i) – Make the subtree rooted at i into a max-heap, assuming that the subtrees rooted at LEFT(i) and RIGHT(i) are max-heaps.
Method: swap down as necessary.

Algorithm MAX-HEAPIFY(A,i)

1. $l \leftarrow LEFT(i)$
2. $r \leftarrow RIGHT(i)$
3. if $(l \leq heap_size[A])$ & $(A[l] > A[i])$
4. then $largest \leftarrow l$
5. else $largest \leftarrow i$
6. if $(r \leq heap_size[A])$ & $(A[r] > A[largest])$
7. then $largest \leftarrow r$
8. if $largest \neq i$
9. then exchange $A[i] \leftrightarrow A[largest]$
10. MAX-HEAPIFY$(A, largest)$

6.2 MAX-HEAPIFY operation (example)

The invisible value is 4, which violates the max-heap property.



6.3 BUILD-MAX-HEAP operation

Given an unordered array A, turn it into a max-heap.

The idea is to fix one element at a time, working from the leaves up to the root. As each element's turn comes, its children are already fixed.

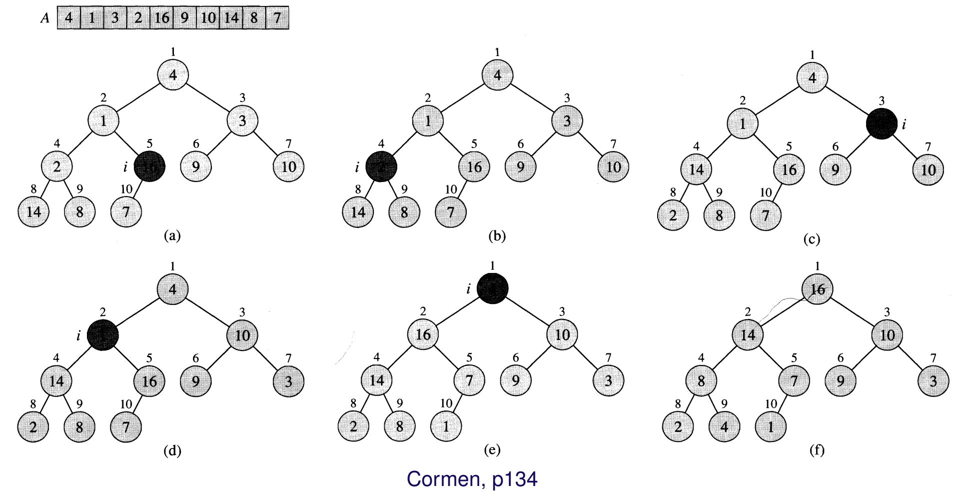
Entries $\lfloor length[A]/2 \rfloor$ and later are leaves, so they are already fixed.

BUILD-MAX-HEAP(A)

1 $heap_size[A] \leftarrow length[A]$
2 for $i \leftarrow \lfloor length[A]/2 \rfloor$ downto 1 do
3 MAX-HEAPIFY(A, i)

Time required: $O(n)$. **Why?**

6.3 BUILD-MAX-HEAP operation (example)



Cormen, p134

6.3.1. Heapsort

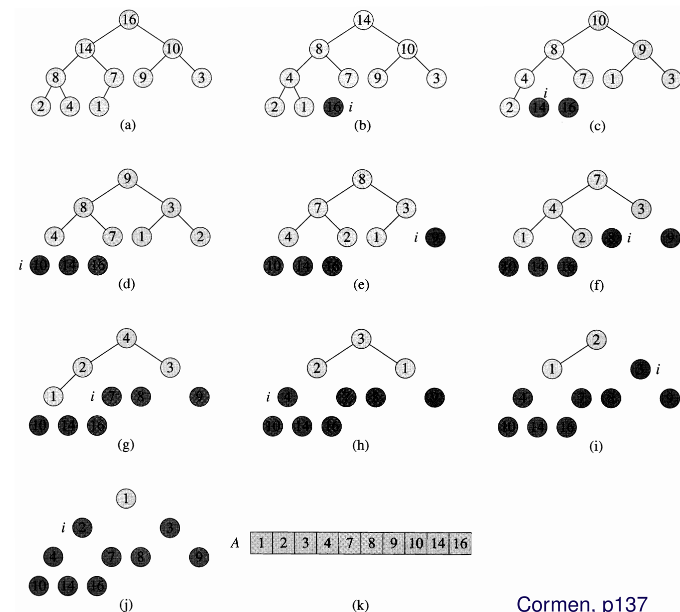
Heapsort is an algorithm for sorting an array.

Method: Turn the array into a max-heap. Then repeatedly remove the maximum element of the heap into the proper place in the array.

Algorithm HEAPSORT(A)

1 BUILD-MAX-HEAP(A)
2 for $i \leftarrow length[A]$ downto 2 do
3 $A[1] \Leftrightarrow A[i]$
4 $heap_size[A] \leftarrow heap_size[a] - 1$
5 MAX-HEAPIFY($A, 1$)

Time required: $O(n \lg n)$. **Why?**



Cormen, p137