ECE368

Weeks 7 and 8 (Notes)

(Incomplete slides will be worked out in class)

Comparing Running Times (adapted from Garey and Johnson)

Given that the program uses 1 microsecond per step (10^{-6}), s stands for second, m for minute, d for day, y for year, and c for century:

	Size of Input					
Time Complexity	10	20	30	40	50	60
n	.00001 s	.00002 s	.00003 s	.00004 s	.00005 s	.00006 s
n^2	.0001 s	.0004 s	.0009 s	.0016 s	.0025 s	.0036 s
n^3	.001 s	.008 s	.027 s	.064 s	.125 s	.216 s
n^5	.1 s	3.2 s	24.3 s	1.7 m	5.2 m	13 m
2^n	.001 s	1.0 s	17.9 m	12.7 d	35.7 y	366 c
3^n	.059 s	58 m	6.5 y	3855 c	2×10^8 c	1.3×10^{13} c

Efficiency Terminology (Summary)

O(1) means a constant computing time.

O(log n) is called logarithmic.

O(n) is called linear.

O(n^2) is quadratic.

O(n^3) is cubic.

O(2^n) is exponential.

Space Complexity

- Space required may depend upon the input (worst-case and average-case complexity)

- If input data has some natural form, the analyze the additional space requirement.

- O(1) additional space: *in place* algorithm

Sorting

Arranging elements in a certain order.

Time-space complexity

Stability: Original order among the same values is preserved.

Sorting

Internal vs External Sorting

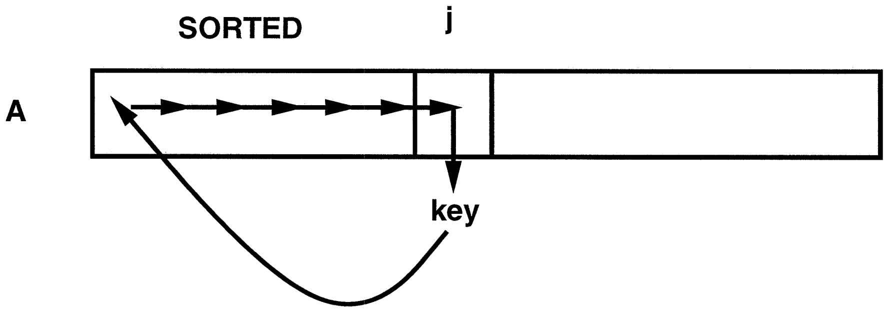
Exchange Sort

- Bubble Sort
- Quicksort

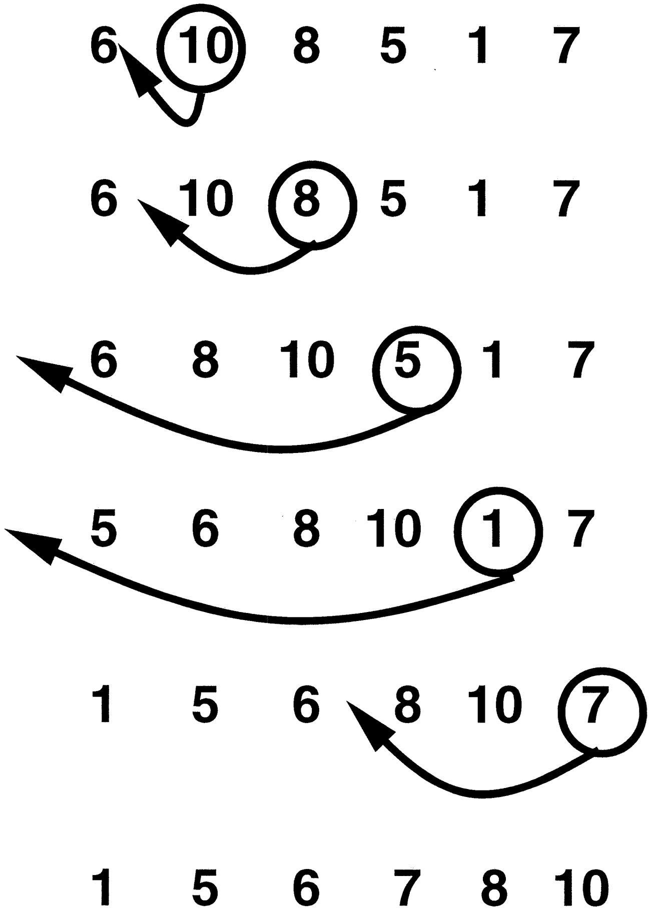
- Selection Sort
- Heap Sort
- Insertion Sort
- Shell Sort
- Merge Sort
- Radix Sort

Introductory Example: INSERTION-SORT

```
INSERTION-SORT($A$)
for $j \leftarrow 2$ to $length[A]$
      do $key \leftarrow A[j]$
         $\triangleright$ Insert $A[j]$ into the sorted sequence $A[1..j-1]$
         $i \leftarrow j-1$
         while $i > 0$ and $A[i] > key$
               do $A[i+1] \leftarrow A[i]$
                  $i \leftarrow i-1$
         $A[i+1] \leftarrow key$
```



INSERTION-SORT **Example:** $\langle 6, 10, 8, 5, 1, 7\rangle$



6 10 8 5 1 7

6 10 8 5 1 7

6 8 10 5 1 7

5 6 8 10 1 7

1 5 6 8 10 7

1 5 6 7 8 10

Pseudocode Notation

- Indentation reflects block structure.
- Looping and conditional constructs have Pascal semantics.
- $\triangleright$ is used for comments.
- $\leftarrow$ is used for assignment.
- Variables are local unless otherwise indicated.
- Array elements are accessed as in Pascal, and we can specify subranges using: $A[i..j]$. Arrays are compound data types with a length attribute accessed using $length[array\_name]$.
- To access the value of a field in a compound data type, use $field\_name[compound]$.
- Parameters are passed by value.
- Omit error handling used in real programs.

How to Characterize an Algorithm

What are the resources used in terms of memory and time?

Tools required to answer:

- execution model: RAM (random access machines)
- math tools:
  - discrete combinatorics
  - elementary probability
  - algebraic dexterity
  - methods of identifying most significant terms

Goal: Find the running time as a function of the input size.

Analyzing INSERTION-SORT

input size: number of items in the input (or number of bits, number of edges, etc.)

running time: number of computational steps in RAM model.

```
INSERTION-SORT(A)                                 cost     times
for $j \leftarrow 2$ to $length[A]$                $c_1$    $n$
    do $key \leftarrow A[j]$                       $c_2$    $n-1$
       ▷ Insert $A[j]$ into the sorted
       ▷ sequence A[1..$j$-1]
       $i \leftarrow j-1$                          $c_4$    $n-1$
       while $i > 0$ and $A[i] > key$              $c_5$    $\sum_{j=2}^{n} t_j$
           do $A[i+1] \leftarrow A[i]$             $c_6$    $\sum_{j=2}^{n} (t_j - 1)$
              $i \leftarrow i-1$                   $c_7$    $\sum_{j=2}^{n} (t_j - 1)$
       $A[i+1] \leftarrow key$                     $c_8$    $n-1$
```

Some Simple Summations

$$\sum_{j=1}^{n} j = 1 + 2 + 3 + \ldots + (n-1) + n$$

Gauss's trick for summing n numbers:

Add :

$$1 + 2 + 3 + 4 + \ldots + (n-1) + n$$

To:

$$n + (n-1) + (n-2) + (n-3) + \ldots + 2 + 1$$

Giving:

$$(n+1) + (n+1) + (n+1) + (n+1) + \ldots + (n+1) + (n+1)$$

Hence:

$$\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$$

To obtain a solution for $\sum_{j=1}^{n-1} j$, substitute in (n-1) for n to obtain:

$$\sum_{j=1}^{n-1} j = 1{+}2{+}3{+}\ldots{+}((n{-}1){-}1){+}(n{-}1) = \frac{(n-1)((n-1)+1)}{2} = \frac{n(n-1)}{2}$$

Some Simple Summations *continued*

How about the following?

$$\sum_{j=1}^{n} c$$

$$\sum_{j=2}^{n} j$$

$$\sum_{j=2}^{n} (j-1)$$

Analysis Concepts

The running time of INSERTION-SORT is calculated as follows:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$
$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Kinds of Time Analysis:

- **Worst-case:** $T(n)$ is the maximum time on any input of size n (usually we use this).
- **Average-case:** $T(n)$ is the average time (given some distribution) over all inputs of size n (we sometimes use this).
- **Best-case:** $T(n)$ is the minimum time on any input of size n (never use this).

Discussion of the INSERTION-SORT Analysis

What is the best-case running time for insertion sort? When does it occur?

Discussion of the INSERTION-SORT Analysis *continued*

What is the worst-case running time for insertion sort? When does it occur?

$$\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(\tfrac{n(n+1)}{2} - 1) \\
&\quad + c_6(\tfrac{n(n-1)}{2}) + c_7(\tfrac{n(n-1)}{2}) + c_8(n-1) \\
&= (\tfrac{c_5}{2} + \tfrac{c_6}{2} + \tfrac{c_7}{2})n^2 + (c_1 + c_2 + c_4 + \tfrac{c_5}{2} - \tfrac{c_6}{2} - \tfrac{c_7}{2} + c_8)n \\
&\quad -(c_2 + c_4 + c_5 + c_8)
\end{aligned}$$

Discussion of the INSERTION-SORT Analysis *continued*

What is the average-case running time for insertion sort?

How much space is needed in (best-case, average-case, worst-case)?

Exchange Sort:
Bubble Sort

Algorithm:

1. Keep scanning through the list of numbers comparing adjacent elements. If they are out of place exchange.

2. Keep scanning until you hit a scan with no exchanges.

Scan 0 25 57 48 37 12 92 86 33
(original file)

Designing Algorithms

The INSERTION-SORT algorithm uses an **incremental** approach (at any point we have a sorted partial subarray).

Another design approach involves **Divide-and-Conquer Algorithms**:

Divide a problem into subproblems.

Conquer, i.e., solve the subproblem (either by dividing again or by solving directly for small inputs).

Combine the solutions of the subproblems.

MERGE-SORT is an example of a divide-and-conquer algorithm.

Shell Sort (diminishing increment sort)

Insertion sort is slow because it exchanges only adjacent elements -- shell sort is a simple extension which gets around this by allowing long distance exchanges.

Insertion sort is efficient for almost sorted files and by the time we deal with all n elements the file is almost sorted.

Shell Sort: Sorts separate subfiles of a file, where the subfile contains every kth element of the original file.

- Once a file is k-sorted, we partition the file for some i, where i < k.

- We continue the process of partitioning, until we 1-sort the file.

File Partitioning Rule

Given increment k, we can get the i–th element of a subfile j using the following rule:

a[(i–1) * k + j - 1]

For example, for k = 3, we get...

subfile 1 : a[0], a[3], a[6], ...
subfile 2 : a[1], a[4], a[7], ...
subfile 3 : a[2], a[5], a[8], ...

- Sort each subfile by simple insertion

- Repeat the process for all increments with the final increment = 1.

Example

k=13, **13-sort subfiles** (Total 15 elements):

(1) a[0], a[13]
(2) a[1], a[14]
(3) a[2]

,

.

(13) a[13]

k-4, **4-sort subfiles**

(1) a[0], a[4], a[8], a[12]
(2) a[1], a[5], a[9], a[13]
(3) a[2], a[6], a[10], a[14]
(4) a[3], a[9], a[11]

k=1, **1-sort subfiles**

(1) a[0], a[1], ..., a[13], a[14]

Example

List:	25	57	48	37	12	92	86	33
pass1:	25	57	48	37	12	92	86	33

span=5

pass 2
span = 3

pass 3
span =1

Complexity

- Running times $O(n (\log n)^2)$ with an appropriate series of increments

- Good for moderate-sized files

- Stable? No!

Recommended increments:

Should be relatively prime

One possible approach:
Define a function h recursively, with :

$$h(1) = 1$$
$$h(i+1) = 3 * h(i) + 1$$

Let x be the smallest integer such that $\mathbf{h(x)} \geq \mathbf{n}$

Set numinc = x-2,
Set incmnts i = h(numinc - i + 1)

$1 \leq i \leq$ numinc

Ex: n = 25 $h(1) = 1$ <- incmnts 2

$h(2) = 4 = 3 * 1 + 1$ <- incmnts 1

$h(3) = 13 = 3 * 4 + 1$

x =4 $h(4) = 40 = 3 * 13 * 1$

Quicksort

(Developed by C.A.R. Hoare in 1960.)

- A divide and conquer algorithm, partitioning a file into 2 parts, sorting each part independently.

```
 i = partition (*a, lb, ub);
quicksort(*a, lb, i-1);
quicksort(*a, i+1, ub);
```

Comments

- A call quicksort (a, 0, N-1) will sort the elements in a from 0 to N-1 so long as we are able to define the partition function.

- Partition divides the file into 2 parts by finding some a[i] in its correct spot in the array such that:

  - a[lb] ... a[i-1] are $\leq$ a[i]
  - a[i+1] ... a[ub] are $>$ a[i]

List Partitioning and the Algorithm

- Arbitrarily choose some element to put into its final position, say a[lb]

- Scan the array from L --> R (use index L)
  Scan the array from R --> L (use index R)

- Stop L --> R scan whenever we hit an element > a[lb]

- Stop R -> L scan whenever we hit an element

  ≤ a[lb]

- These two elements are out of place so exchange them.

- Continue until R < L, then R is index of location for partition element.

- At this point, we know we got the right place for a[lb] to go. So, exchange a[lb] with the leftmost element of a[R].

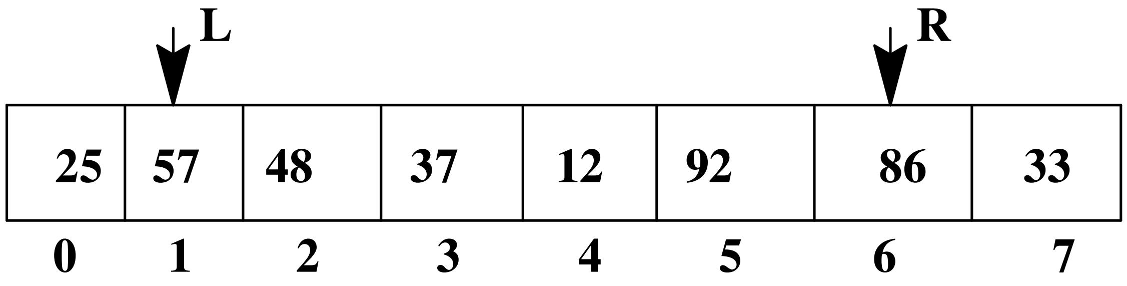
Quicksort Example

1.

L = 0, R =7,   a[0] = 25 = pivot

since, 25 > 25 and 33 ≤ 25, so continue both scans.

2.



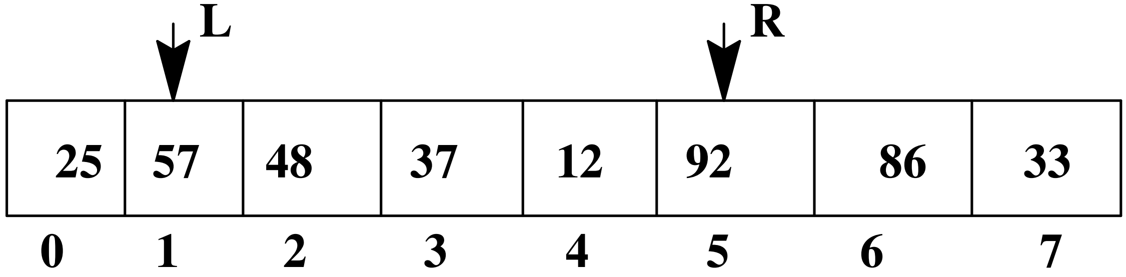
25	57	48	37	12	92	86	33
0	1	2	3	4	5	6	7

L =1, R =6,

since, 57 > 25 (stop L) and 86 ≤ 25 (continue R).

Example (contd')

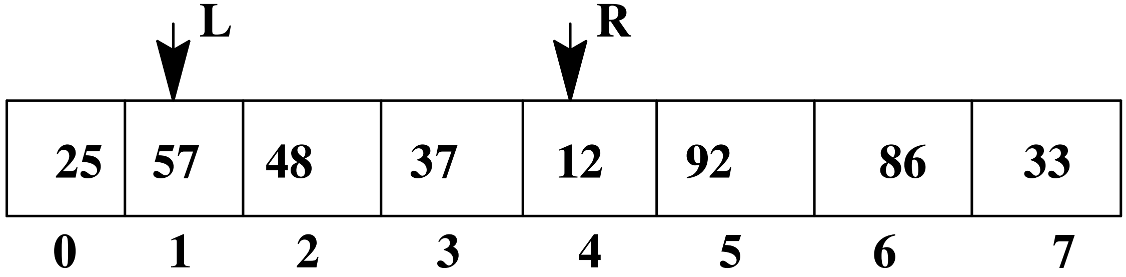
3.



0	1	2	3	4	5	6	7
25	57	48	37	12	92	86	33

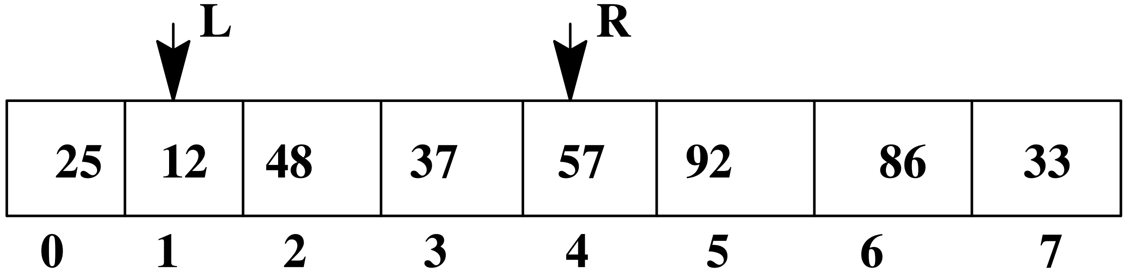
L =1, R =5, since, 92 ≤ 25 (continue R scan).

4.



0	1	2	3	4	5	6	7
25	57	48	37	12	92	86	33

L =1, R =4, Now 25 ≤ 12, So swap a[L] and a[R].

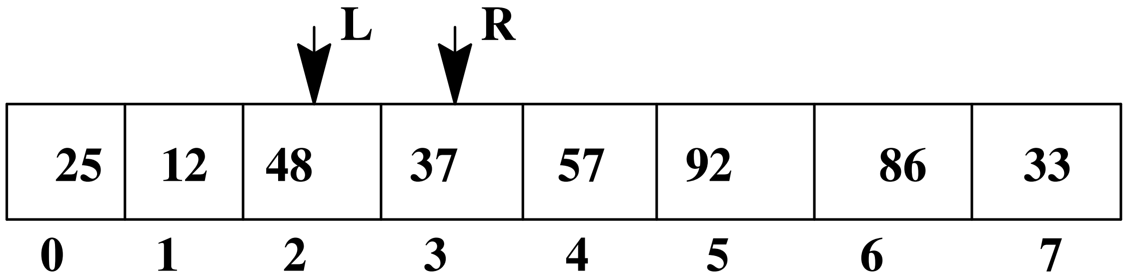


0	1	2	3	4	5	6	7
25	12	48	37	57	92	86	33

and continue scan both R and L.

Example (contd')

5.

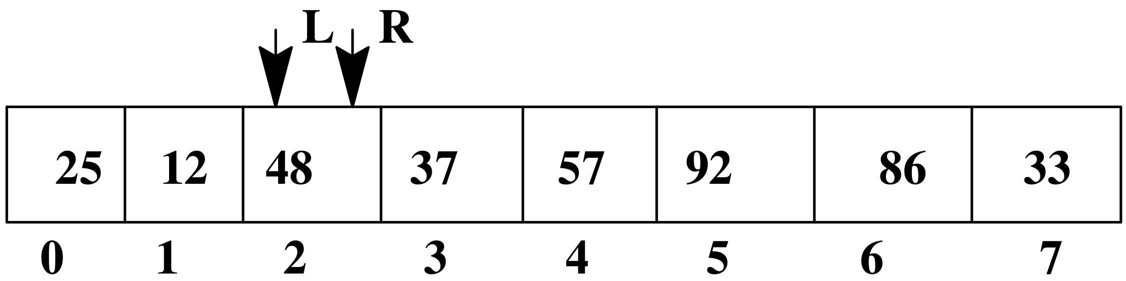


25	12	48	37	57	92	86	33
0	1	2	3	4	5	6	7

L =2, R =3,

Now 48 > 25 stop L, But 37 ≤ 25, so continue R.

6.



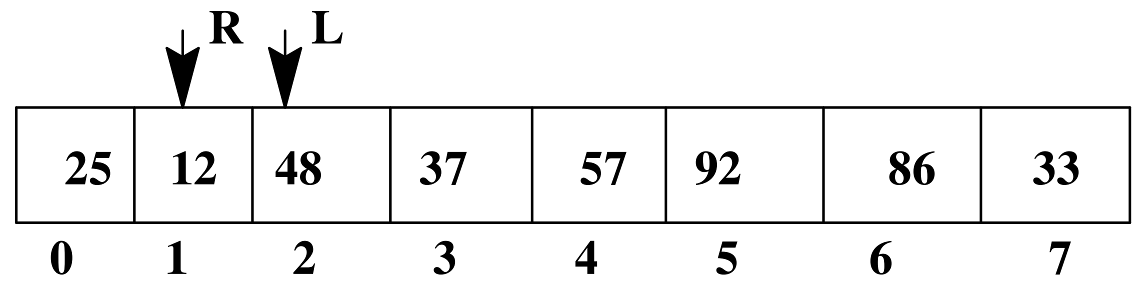
25	12	48	37	57	92	86	33
0	1	2	3	4	5	6	7

L =2, R =2,

Now 48 ≤ 25, so continue R (also R < L)

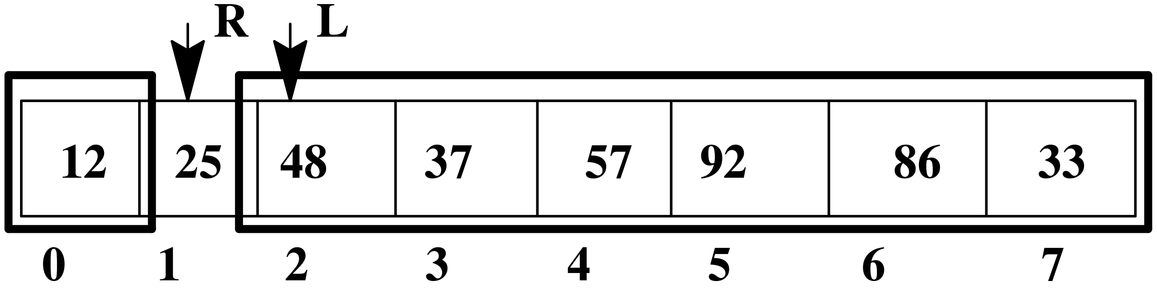
Example (contd')

7.



25	12	48	37	57	92	86	33
0	1	2	3	4	5	6	7

L =2, R =1, Now R < L and we swap a[0] with a[R]



12	25	48	37	57	92	86	33
0	1	2	3	4	5	6	7

At this point, every element greater than 25 is on its Right, and less than equal is on the Left.

Now we can quicksort two sub-arrays a[0] and a[2..7] by making a recursive call to quicksort. Pivot Index is i = 1.

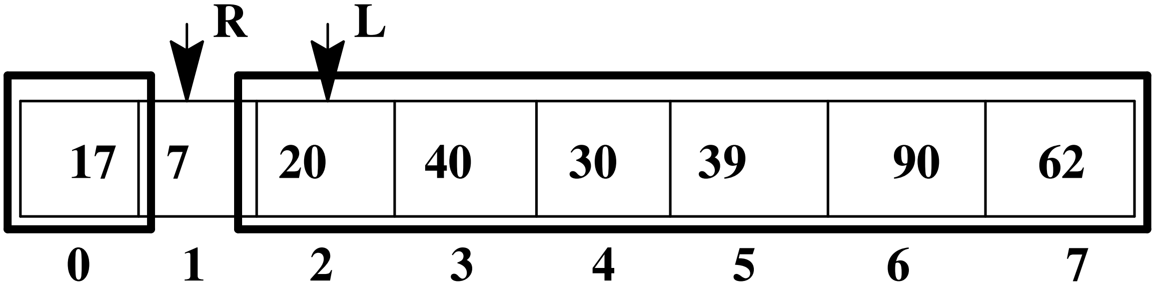
Note: a[0] is sorted so no more work for that half of the array!

Note: Every element is eventually put into place by being used as a partitioning element!

Example

a :	17	62	20	40	30	39	90	7
	0	1	2	3	4	5	6	7

(a) Quicksort(a,0,7) find partition :
(partition = 17, L = 0, R =7)



a :	17	7	20	40	30	39	90	62
	0	1	2	3	4	5	6	7

Partition

(b) Quicksort (a, 0, 0)

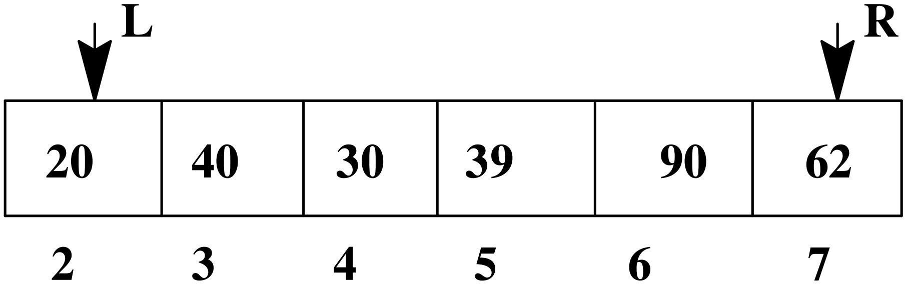
make 1 call
1 > 1
Done!

(c) Quicksort (a, 2, 7)

make call

Example (contd')

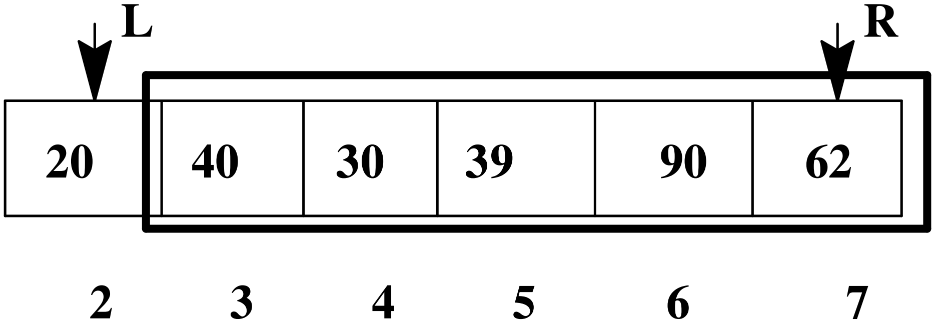
c) Quicksort (a, 2, 7)



20	40	30	39	90	62
2	3	4	5	6	7

Lb = 2 ub = 7 pivot = 2 0

L = 2 R = 7



20	40	30	39	90	62
2	3	4	5	6	7

(d) Quicksort (a, 2, 1) (e) Quicksort (a, 3, 7)

1 > 2 make call

Done

Example (contd')

Result of Quicksort:

a :	7	17	20	30	39	40	62	90
	0	1	2	3	4	5	6	7

Efficiency of Quicksort:

Assume: n = 2^k

Assuming pivot always divides the file exactly in half, we need:

- n comparisons to split it in 2 subarrays (= n)

- n/2*2 comparisons to split into 4 subarrays (=n)

- n/4 * 4 . . . . = n

.

.

.

- n/n * n . . . . = n

Total k such terms

Complexity: O(kn) = O(n log n) comparisons

Efficiency of Quicksort:

Runs inefficiently if file is already sorted (assumption is not valid).

If file is pre-sorted, x[lb] will always be in correct spot, so we will only knock 1 element off for each partition.

- n comparisons
- n-1 comparisons
- n-2 comparisons

.

.

- 3 comparisons
- 2 comparisons

This is O(n^2).

Improvements:

- Remove recursion.
- When we get down to small subfiles, apply a different method rather than doing Quicksort.
- Use a better partitioning element.

  (1) use random element to avoid problem of sorted file

  (2) take 3 elements from file and sort them use middle guy as partition element. (median of three method) reduces running time by 5% overall

What about extra space? Depends on # of nested recursive calls!

Heap Sort (Tree Sort):

Definition: A Descending Heap (Max Heap) of size n is an almost complete binary tree of n nodes

such that they key of each node is ≤ the key of its father. (Heap property)

Examples:

is a descending heap.

is a descending heap.

is a descending heap.

Examples:

is not a descending heap!

is not a descending heap.

is not a descending Heap!

Heap Generation/Insertion Procedure

Insertion is done by getting the position of next insertion as a leaf node such that an almost complete binary tree structure is maintained.

Then go up the tree to the 1st element $\geq$ the element to insert. As we go up the heap, each element less than the element to be inserted is shifted down, making room for the new element which is being inserted.

The root has the largest element.

Example

25   57   48   37   12   92   86   33

Insertion Algorithm

```
s = k
f = (s-1) /2         /* f is the father of s */
while  ( s > 0 && dpq[f] < elt )
   {
     dpq[s] = dpq[f];
       s = f;   /* advance up the tree */
        f = (s-1)/2;
  }   /* end while */
  dpq[s] = elt;
```

Analysis of Insertion

We can create a heap by inserting elements into the heap in such a manner that we miantain heap properties. Each insertion can be done in O(log n). Therefore, insertion of n elements into a heap can be done in O(n log n).

Heap Deletion Procedure

1. Remove the Root

2. Move the last element (remove its node) to the Root node. The new structure is again an almost complete binary tree.

3. Select among the largest value of the new root and its immediate children to become the new root.

4. Apply Step 3 again on the subtree with the root being the child with the new value.

Heap Sorting

It is a general selection sort using the input array as a heap.

Step 1:

Step 2:

Analysis

-Worst case behavior

-Avg. case behavior

- Space needed?

- Stability?

- Performance with sorted data?

Merge Sort

- Take 2 sorted files and merge them. Merge small files (size 1), then bigger (size 2), then bigger, ..., till all files are merged into one.
- Recursive Implementation

Merge Sort

[25] [57] [48] [37] [12] [92] [86] [33]

[25 57] [37 48] [12 92] [33 86]

[25 37 48 57] [12 33 86 92]

[12 25 33 37 48 57 86 92]

Analysis

- Maximum number of passes : log n
- Each pass requires n or fewer comparisons
- Overall complexity -> O(n log n)
- Requires extra space - O(n) for auxiliary array.
- Is Merge Sort stable? yes
- Sorted Data? Same running time.
- Reverse Sorted Data? Same running time.

Radix Sort

(Radix Exchange Sort)

– Based on the values of the digits of the key

– Base 10 – partition file into 10 groups on leading digit of key. 9 group > 8 group > ... > 0 group.

– Sort on next most–significant digit

– Repeat until you reach the least significant digit

– Must make room in array for items (array implementation)

Radix Sort (Alternate Approach)

– Process digits from least significant to most significant digit.

– Take each record with key, in the order it appears in the file, and place it in one of 10 queues depending on value of the digit.

– Return items to file in the order they were placed onto the queue, starting 0 queue 1st, 1 queue second, etc.

– Repeat the process for each digit until it is done on most significant digit –> file is sorted.

Algorithm

```
for (K = least significant to most significant digit )
{
       for  (i= 0; i < n; i++)
   {
               y =  x [i]
              j = Kth digit of y
              place y at rear of queue[j];
   }             /* end for */


       for (qu = 0; qu < 10; qu++)
          place elements of queue[qu]
          in next sequential position of x

 }    /* end  for  */
```

Example

Original Data:

25 57 48 37 12 92 86 33

i) First sort on least significant digit, in each bin (queues in form of linked list)

bin [0]

bin [1]

bin [2]

bin [3]

bin [4]

bin [5]

bin [6]

bin [7]

bin [8]

bin [9]

Example (cont'd)

ii) Now sort on the next most significant digit

12 92 33 25 86 57 37 48

bin [0]

bin [1]

bin [2]

bin [3]

bin [4]

bin [5]

bin [6]

bin [7]

bin [8]

bin [9]

Complexity

(1) Running time:

If *d* digits and *n* records and radix (no. of bins or queues) is *r*, each pass is O(*n*+*r*)
(as *n* elements are placed O(*n*) and *r* bins are initialized O(*r*) in each pass).

d passes --> ??

- very efficient if ***d*** and ***r*** are small relative to ***n***.

(2) Space required? --> queue [0] ... queue [9]

(3) Stability? Yes

(4) Behavior of algorithm with sorted data?

unaffected

How Fast Can We Sort?

Use a Decision/Comparison Tree to sort n elements.

Decision Tree for Sorting

– Binary tree. A node represents a comparison operation between two elements of the input data set.

– A path from the root to a leaf node represents the order of the sequence of comparison operations for a given input data set, that results in the output sequence represented by the leaf node.

– The order of the sequence of comparison operations (that is the selection of one of the paths) depends upon the input data set and the sorting algorithm.

– Depending upon the sorting algorithm, the tree structure may vary.

Example of Decision Tree for Sorting

Decision Tree for Sorting

- Number of leaf nodes =
- Height of the Tree:

---> Complexity of a sorting algorithm.
(Best of the worst case)

Avg. Case Complexity:
= External Path Length/Number of possible output sequences

External Path Length: Sum of the number of branches (arcs) traversed in going from the root once to every leaf node in the tree =

---> Avg. Case Complexity:

Sorting Algorithm	Time	Space
- Exchange Sort		
- Bubble Sort	O(n^2)	O(1)
- Quicksort	O(n log n) & O(n^2)	
Space Depends on # of nested recursive calls		
- Selection Sort	O(n^2)	O(1)
- Heap Sort	O(n log n)	O(1)
- Insertion Sort	O(n^2)	O(1)
- Shell Sort	O($n^{1.25}$)	O(1)
	O(n (log n)^2)	
- Merge Sort	O(n log n)	O(n)
- Radix Sort	O(n^2)	O(1)
- How fast can we sort?	O(n log n)	