Analysis and Design of Algorithms

By
Syed Bakhtawar Shah Abid
Lecturer in Computer Science

What is Sorting?

An operation that segregates items into groups according to specified criterion or key

Examples:- Sorting Books in Library, Sorting Individuals by Height, Sorting Movies in Blockbuster Sorting Numbers, sorting student records.

A = { 3 1 6 2 1 3 4 5 9 0 }

A = { 0 1 1 2 3 3 4 5 6 9 }

Some Definitions

- Internal Sort
  - The data to be sorted is all stored in the computer's main memory.
- External Sort
  - Some of the data to be sorted might be stored in some external, slower, device.
- In Place Sort
  - The amount of extra space required to sort the data is constant with the input size.

Stability

- A STABLE sort preserves relative order of records with equal keys

Sorted on first key:

Aaron	4	A	664-480-0023	097 Little
Andrews	3	A	874-088-1212	121 Whitman
Battle	4	C	991-878-4944	308 Blair
Chen	2	A	884-232-5341	11 Dickinson
Fox	1	A	243-456-9091	101 Brown
Furia	3	A	766-093-9873	22 Brown
Gazsi	4	B	665-303-0266	113 Walker
Kanaga	3	B	898-122-9643	343 Forbes
Rohde	3	A	232-343-5555	115 Holder
Quilici	1	C	343-987-5642	32 McCosh

Sort file on second key:

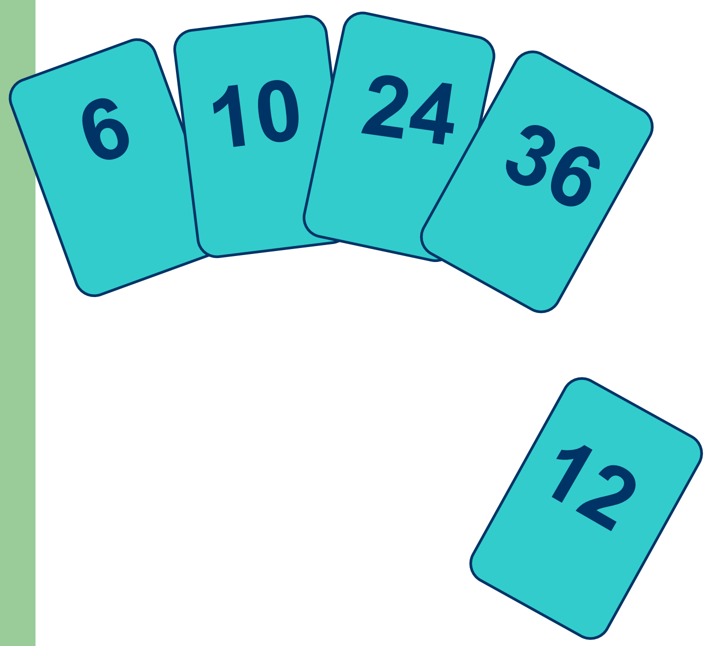
Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Furia	3	A	766-093-9873	22 Brown
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Gazsi	4	B	665-303-0266	113 Walker
Aaron	4	A	664-480-0023	097 Little

Records with key value 3 are not in order on first key!!

Insertion Sort

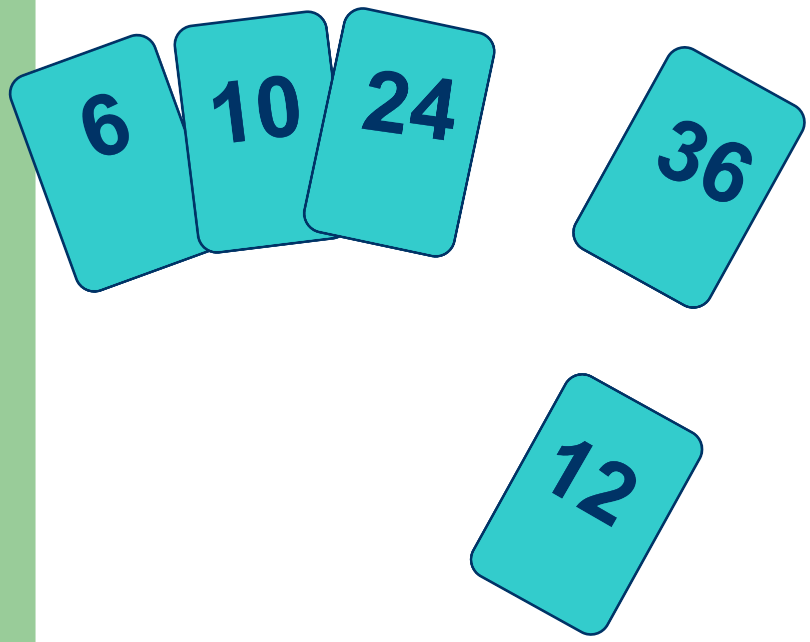
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

Insertion Sort

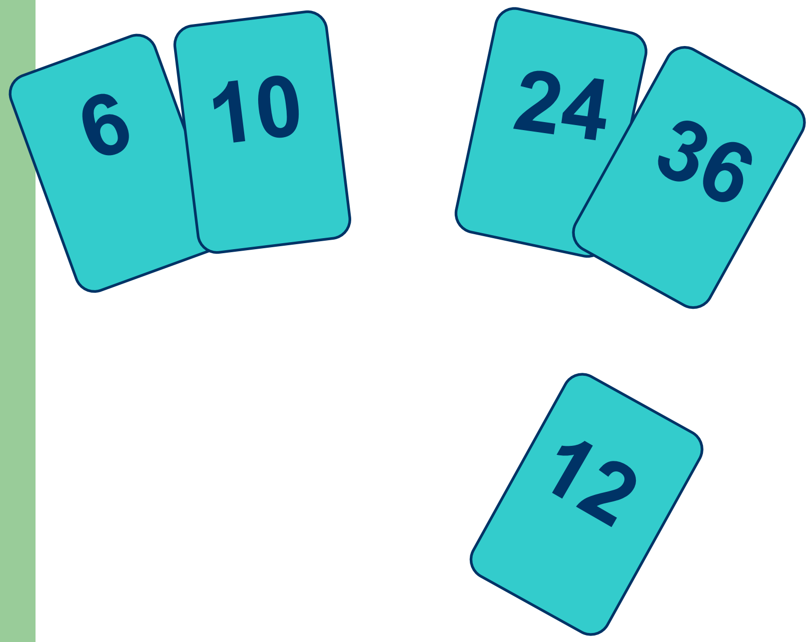


To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort



Insertion Sort



Insertion Sort Algorithm

- Pseudo code for Insertion sort algorithm — constant — times
- Insertion-sort (A , N)

Step	constant	times
– Repeat step 2 to 4 for K = 2 to N	C1	n
– Set Temp := A[K]	C2	n - 1
– Set PTR := K – 1	C3	n - 1
– Repeat while PTR > 0 AND Temp < A[PTR]	C4	$\sum_{k=2}^{n}(t_k)$
• A[PTR + 1] := A[PTR]	C5	$\sum_{k=2}^{n}(t_k - 1)$
• Set PTR := PTR – 1	C6	$\sum_{k=2}^{n}(t_k - 1)$
– [End of Loop]		
– Set A[PTR + 1] := Temp	C7	n - 1
– [End of Step 1 Loop]		

- Exit

Insertion Sort Algorithm

- The running time of the algorithm is the sum of running times for each statement executed.
- A statement that takes c_i steps to execute and executes n times will contribute $c_i n$ to the total running time.
  - So $T(n)$ = C1n + C2(n – 1) + C3(n – 1) + C4$\sum_{k=2}^{n}(t_k)$ + C5$\sum_{k=2}^{n}(t_k - 1)$ + C6$\sum_{k=2}^{n}(t_k - 1)$ + C7(n – 1)
- Running time for already sorted array(Best Case) is a linear function of n i.e. $T(n) = an + b$ whereas
- Running time for a reverse sored array(Worst Case) is a quadratic function of n i.e. $T(n) = an^2 + bn + c$

LOOP INVARIANTS AND THE CORRECTNESS OF INSERTION SORT

- **Initialization:** It is true prior to the first iteration of the loop i.e. the sub-array A[1..j-1], therefore, consists of just the single element A[1], which is in fact the original element in A[1]
- **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

THE DIVIDE-AND-CONQUER APPROACH

- Many useful algorithms are ***recursive*** in structure
- To solve a given problem, they call themselves recursively one or more times to deal with closely related sub-problems.
- These algorithms typically follow a ***divide-and-conquer*** approach:

THE DIVIDE-AND-CONQUER APPROACH

- These algorithms break the problem into several sub-problems that are similar to the original problem but smaller in size, solve the sub-problems recursively, and then combine these solutions to create a solution to the original problem.

THE DIVIDE-AND-CONQUER APPROACH

- The divide-and-conquer paradigm involves three steps at each level of the recursion:
  - **Divide** the problem into a number of sub-problems that are smaller instances of the same problem.
  - **Conquer** the sub-problems by solving them recursively.
  - **Combine** the solutions to the sub-problems into the solution for the original problem.

MERGE SORT

MERGE-SORT(A, p, r)

if $p < r$

    $q = \lfloor (p + r)/2 \rfloor$

    MERGE-SORT(A, p, q)

    MERGE-SORT$(A, q + 1, r)$

    MERGE(A, p, q, r)

MERGE PROCEDURE

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
        L[i] = A[p + i - 1]
for j = 1 to n2
        R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
        if L[i] ≤ R[j]
                A[k] = L[i]
                i = i + 1
        else A[k] = R[j]
                j = j + 1
```

LOOP INVARIANT

- At the start of each iteration of the **for** loop of lines 12–17,
  - The sub-array A[p . . . . K-1] contains the k - p smallest elements of L[1 …..n_1+1] and R[1 ….n_2+1], in sorted order.
  - Moreover, L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.

LOOP INVARIANT

- **Initialization:**
  - Prior to the first iteration of the loop, we have k = p, so k-p=0 and the sub-array A[p…..k-1] is empty and since i = j = 1, both L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.
  - **Maintenance: ?**
  - **Termination: ?**

ANALYZING DIVIDE-AND-CONQUER ALGORITHMS

- When an algorithm contains a recursive call to itself, its running time is often described by a ***recurrence equation*** or ***recurrence***
- ***Recurrence equation*** or ***recurrence*** describes the overall running time on a problem of size n in terms of the running time on smaller inputs.
- Mathematical tools are then used to solve the recurrence and to provide bounds on the performance of the algorithm.

ANALYZING DIVIDE-AND-CONQUER ALGORITHMS

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c , \\ aT(n/b) + D(n) + C(n) & \text{otherwise} . \end{cases}$$

- Here
  - Suppose that the division of the problem yields a sub-problems, each of which is $1/b$ the size of the original.
  - It takes time $T(n/b)$ to solve one sub-problem of size n/b, and so it takes time $aT(n/b)$ to solve a of them.
  - $D(n)$ is the time to divide the problem into sub-problems and
  - $C(n)$ is the time to combine the solutions to the sub-problems into the solution to the original problem,

ANALYSIS OF MERGE SORT ALGORITHM

- **Divide:** The divide step just computes the middle of the sub-array, which takes constant time. Thus, $D(n) = \theta(1)$.
- **Conquer:** We recursively solve two sub-problems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.
- **Combine:** We have already noted that the MERGE procedure on an n-element sub-array takes time $\theta(n)$, and so $C(n) = \theta(n)$.

ANALYSIS OF MERGE SORT ALGORITHM

- Recurrence for the worst-case running time $T(n)$ of merge sort

$$T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,, \end{cases}$$