SoC 2022 : Competitive Coding

Week-2

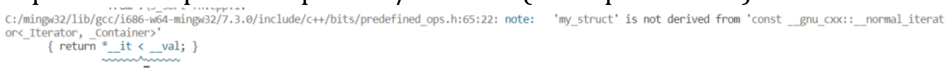
Sorting, Binary Search

Contents

Sorting

In problems, just use sort() from algorithm header.

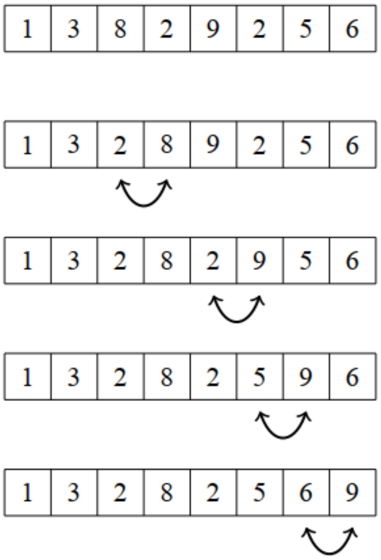
- in C++, sort(v.begin(),v.end()), sort(str.begin(),str.end()), list1.sort() in Python, etc. [done in wk-1].
- requires a comparison operator/function (if not predefined)



```
C:/mingw32/lib/gcc/i686-w64-mingw32/7.3.0/include/c++/bits/predefined_ops.h:65:22: note:   'my_struct' is not derived from 'const __gnu_cxx::__normal_iterat
or<_Iterator, _Container>'
      { return *__it < __val; }
               ~~~~~^~~~~~~
```

  Gives an error like this, if not predefined (e.g., user defined structs).
- Can overload operator<, or define a comparison function

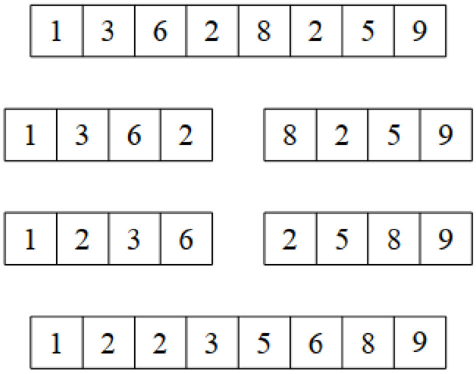
1. Bubble-sort
   - (see code)
   - At the end of first iteration, largest element will be at the correct location. This generalizes for k rounds.
   - Inversion: pair (i,j) such that i<j and a[i]>a[j].
   - No. of swaps in this algo = No. of inversions
   - Each swap removes exactly 1 inversion.
     Say a[i]>a[i+1]. When algo swaps these, then no other inversion pairs are affected, and only this one is removed.
   - O(n^2)



1	3	8	2	9	2	5	6
1	3	2	8	9	2	5	6
1	3	2	8	2	9	5	6
1	3	2	8	2	5	9	6
1	3	2	8	2	5	6	9

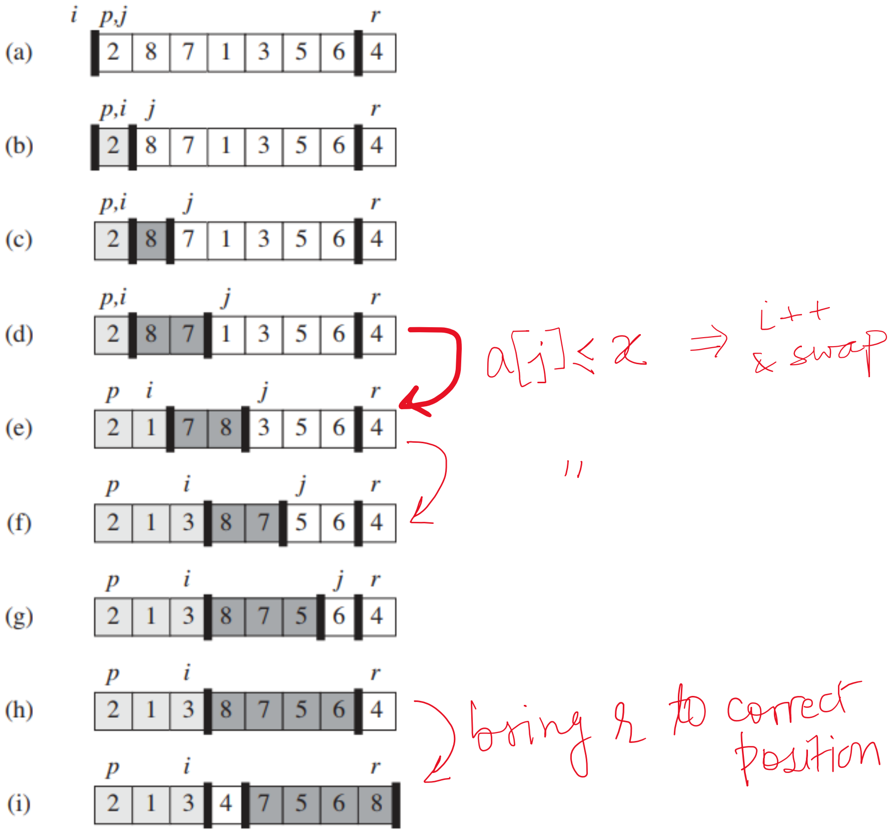
   - (first iteration)

2. Merge-sort
   - O(n.log(n))
     Based on the recurrence T(n) = 2*T(n/2) + O(n)



-
- Additional space required for storing the smaller arrays.

3. Quick-sort
   - Example: 17 12 6 19 23 8 5 10.
   - Consider 10 (last element) as a 'pivot'. Divide array into 2 parts – one with all elements <10, other with all ≥10. (This is done in O(n) time using 2 markers i,j).
   - Recurse on these smaller arrays.
   - Can do in-place (no additional space for arrays needed).
   - Average O(n.log(n))
     Worst case O(n^2) [Example 1 2 3 4 5 6 7 8 – smaller arrays after 1st call will be 1 2 3 4 5 6 7, and 8.]
   - Working of 'partition' [CLRS Section 7.1]
     Pivot is '4'. Note i<j at all points.


(a) i p,j r 2 8 7 1 3 5 6 4
(b) p,i j r 2 8 7 1 3 5 6 4
(c) p,i j r 2 8 7 1 3 5 6 4
(d) p,i j r 2 8 7 1 3 5 6 4
(e) p i j r 2 1 7 8 3 5 6 4
(f) p i j r 2 1 3 8 7 5 6 4
(g) p i j r 2 1 3 8 7 5 6 4
(h) p i r 2 1 3 8 7 5 6 4
(i) p i r 2 1 3 4 7 5 6 8
a[j] ≤ x ⇒ i++ & swap
''
bring r to correct position

Any comparison based sorting algo takes $\Omega(n \log(n))$ time [Ω ~ lower bound]
[CLRS Section 8.1]

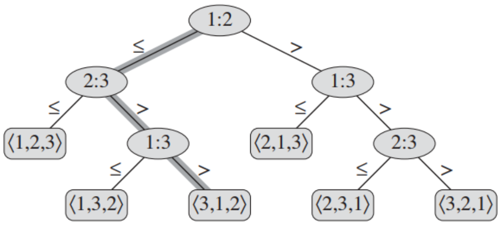


Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i:j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \le a_1 = 6 \le a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, and so the decision tree must have at least 6 leaves.

4. Counting Sort
    - Sorts an array in O(n) time assuming that every element in the array is an integer between 0 and c and c = O(n). [additional constraint]
    - Bookkeeping array (counts of each element)

1	3	6	9	9	3	5	9

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

1,3,3,5,6,9,9,9

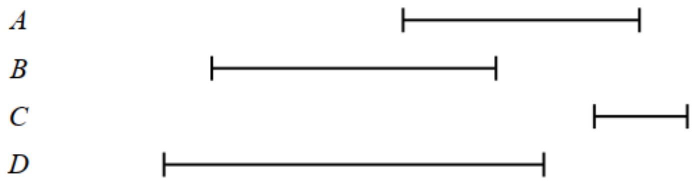
    - From this array, sorted array in O(n).

Use Sorting

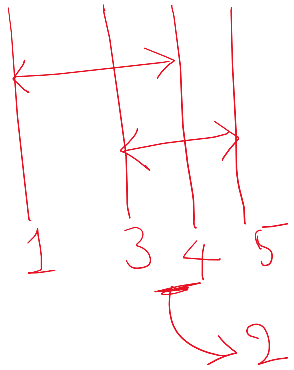
- Can help to get O(n^2) → O(n) or O(n log n)

- Q. Finding #unique elements in array.

- Q. Maximum number of overlapping intervals

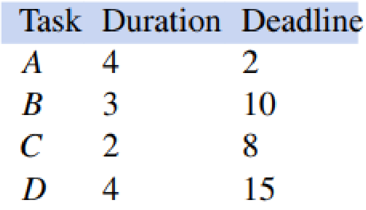


Make a vector (or array) of start and end times (all into one). Sort this vector.
Sweep from left to right. Maintain a counter. +1 if a 'start' is encountered, -1 if 'end' is encountered. Also maintain max value of counter ever achieved.
Say sorted vector is {(s,1),(s,3),(e,4),(e,5)}. Then counter = 0,1,2,1,0. max=2.

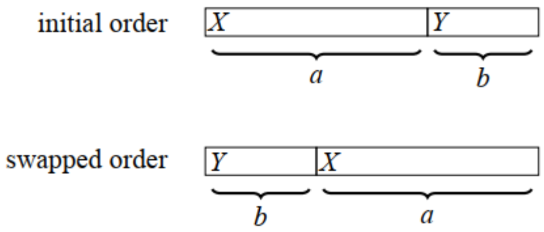


- Q. Given n events with their starting and ending times, find a schedule that includes as many events as possible.
  Greedy strategy, involves sorting (say, by end times).
  Take events with smallest end times, and discard any overlapping events.
  Proof of optimality: consider any solution, then replacing event with smallest end time in this solution (say ej), with the overall smallest end time event (e1), won't affect optimality.

- Q. Given n tasks with durations and deadlines, our task is to choose an order to perform the tasks. For each task, we earn d – x points where d is the task's deadline and x is the moment when we finish the task.



Task	Duration	Deadline
A	4	2
B	3	10
C	2	8
D	4	15



p2+a, p1-b ⇒ a-b

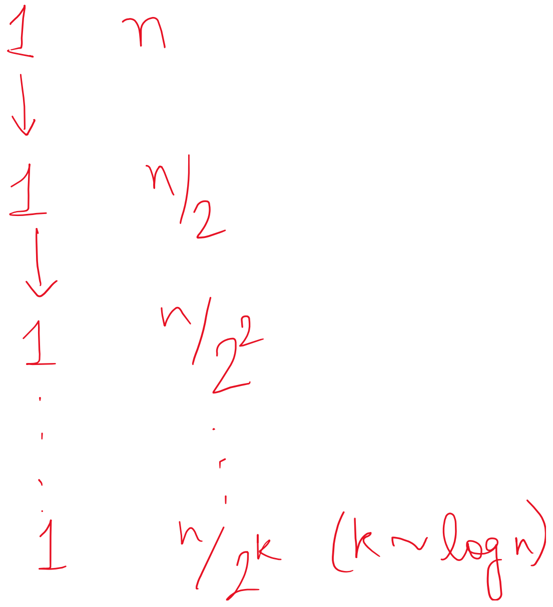
Observe that on swapping X and Y, Y gives 'a' more score, and X gives 'b' more score. So, for any X,Y such that duration(X)>duration(Y), the latter order is better.

This directly gives a greedy strategy: do tasks with increasing (here, sorting is needed) durations.

- Need to think optimality of solution.

Binary Search

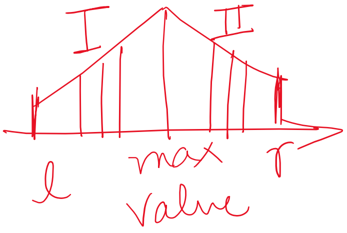
- Searching in a **sorted** array, in O(log n).
- Each step ~ halves the search range, so recurrence is T(n)=T(n/2)+O(1).
  Can solve by recursion tree, or just think that there are log(n) steps, and constant amount of work in each step, so O(log n).



- See code for iterative solution. Preferred to recursion, since it avoids building the entire function stack.

- **Ideas similar to binary search: (halving the search space)** →

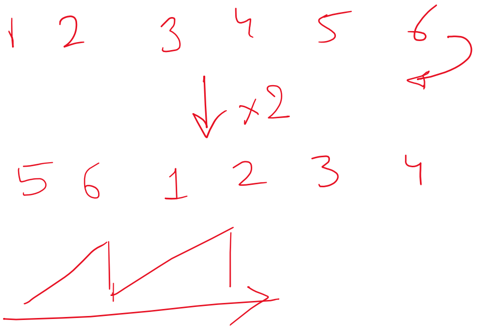
- Q. Find max value in unimodal array.



  Can use binary search for O(log n) solution.
  - Base Cases
  - If (a[mid] is greater than its next and previous value) done
  - Else if (a[mid] lies in case I) search for max in right half
  - Else (case II) search for max in left half

  Again, in each step, search space roughly halved.

- Q: Find the number of times a **sorted** array has been rotated.



Answer is (index of minimum element in this new array).
O(n) : linear search.
O(log n). Similar to binary search. Observe that the minimum element is the only element that has its previous element greater than itself.
https://www.geeksforgeeks.org/find-rotation-count-rotated-sorted-array/
(See code num_rotations.cpp)

- Q. First and last occurrence of an element in a **sorted** array.
  Can do in O(n) [linear traversal of array].
  But given sorted, so should think binary search.
  Find the first and last occurrences (say, indices f and l). Then answer is l-f+1.
  (See code occurrences.cpp)

- **Binary Search on the (optimal) answer, i.e., in the answer space –**

- Q. Painter Partition Problem.
  We have to paint n boards of length {A1, A2, ..., An}. There are k painters available and each takes 1 unit time to paint 1 unit of board. The problem is to find the minimum time to get this job done under the constraints that any painter will only paint continuous sections of boards.

  We need to minimize the maximum time that any painter takes.
  We can use binary search for this minimum time. The search space is the set of these 'maximum times'.

  Example: lengths {40,20,10,30} and 2 painters.
  Search space is 0 (theoretical minimum, won't be achieved) to (40+20+10+30) (or some arbitrary larger number).

P1 40 + ~~20~~ (∵ 60 > 50)

P2 20 + 10 + ~~30~~

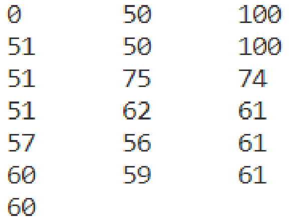
⇒ 50 ✗

⇒ search in 50 — 100

75

P1 40 + 20 + 10, P2 30 ✓. Search in 50 — 75

See code in painters-part.cpp.



```
0       50      100
51      50      100
51      75      74
51      62      61
57      56      61
60      59      61
60
```

Another method is to use DP.

- Q. Consider a problem where our task is to process k jobs using n machines. Each machine i is assigned an integer pi: the time to process a single job. What is the minimum time to process all the jobs?

  Use similar approach: binary search on the space of answers (minimize the maximum time taken by any machine).

First 5 problems of CSES 'Sorting and Searching'.
Will post rest on group.