Data Structures & Programming

Sorting Algorithms

Golnar Sheikhshab

Interview question

When you do laundry, what sort algorithm do you use to sort your sucks?

Sorting algorithms we have mentioned so far

Algorithm	**Time complexity (worst case scenario)**
Insertion sort	O(n^2)
Selection sort	O(n^2)
Merge sort	O(n lg(n))
Heap sort	O(n lg(n))
Binary search sort	O(n^2) If binary search is kept balanced: O(n lg(n))

Sorting algorithms we have not mentioned yet

Algorithm	**Time complexity (worst case scenario)**
bubble sort	O(n^2)
Quick sort	O(n^2)
Bucket sort	O(n)
Radix sort	O(n)

Bucket sort and Radix sort are not applicable in general

General Sorting

Based on comparison

Best possible time complexity: O(n lg n)

Why?

$$\log(n!) \geq \log\left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2}\log\frac{n}{2}$$

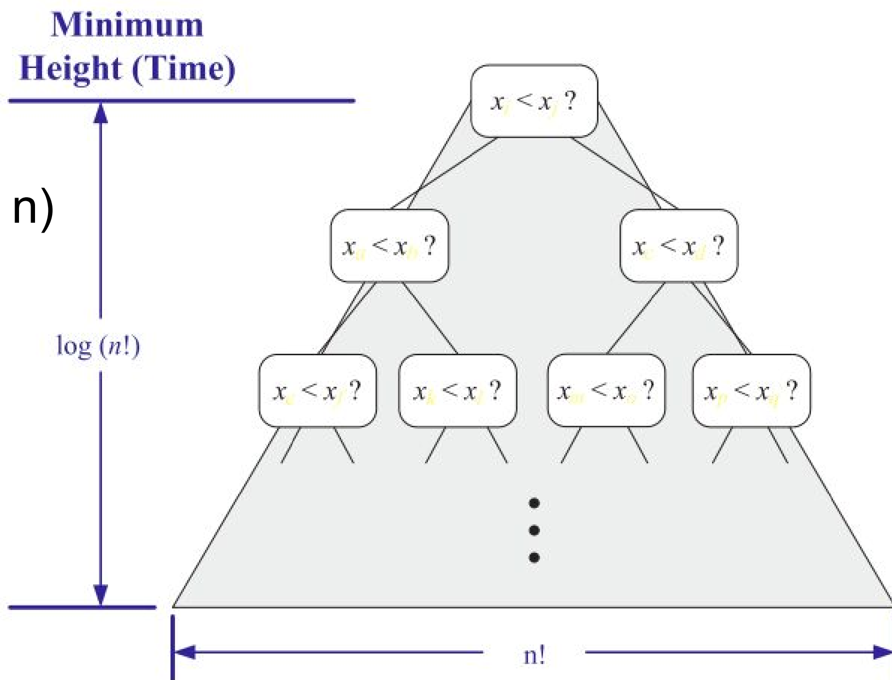


Figure 11.15: Visualizing the lower bound for comparison-based sorting.

Inefficient Sorting

Insertion sort:

Repeatedly insert items at their right place

Selection sort:

Repeatedly find the min/max of the remaining and put it in its right place

Bubble sort:

Repeatedly swap consequent items if their order is wrong

Check out inefficient_sorts.cpp

Efficient sorting

Heap sort

Worst case and average case time complexity: $O(n \log n)$

Need for extra space: $O(1)$

Merge sort

Worst case and average case time complexity: $O(n \log n)$

Need for extra space: $O(n)$

Quick sort

Worst case time complexity: $O(n^2)$

Average case time complexity: $O(n \log n)$

Need for extra space: $O(1)$

Heap Sort

Make a heap in-place

Repeatedly removeMin/removeMax and keep it in the same vector/array

Merge sort

Find the middle

Sort the first half recursively

Sort the second half recursively

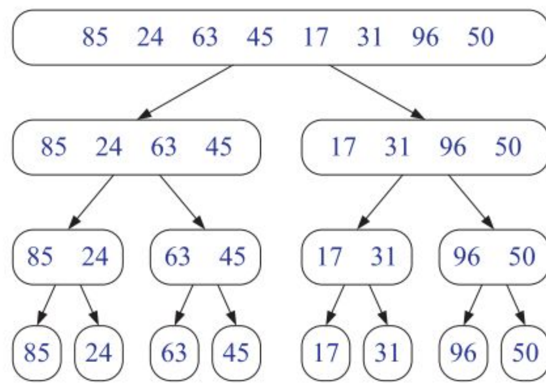
Merge the two already-sorted halves (Merge is explained in the next slide)

Check out merge_sort.cpp

Merge(start, middle, end)

1. Assumes that the container is sorted in [started, middle] and in [middle+1, end]
2. Keeps two cursors in the two intervals starting at start and middle+1
3. Repeatedly inserts the minimum of the two elements that the cursors are pointing to into an auxiliary container and advances the pointer pointing to that minimum
4. Stops when both cursors hit the end of their intervals (middle and end)
5. Copies everything from the auxiliary container back to the original container

```
void merge (vector<int>& v, int start, int middle, int end){
        int cursor1 = start, cursor2 = middle + 1, infinity = INT_MAX;
        vector<int> aux;
        while ( (cursor1<=middle) || (cursor2 <= end) ){
                int c1 = (cursor1<=middle)?v[cursor1]:infinity;
                int c2 = (cursor2<=end)?v[cursor2]:infinity;
                if (c1 < c2){
                        aux.push_back(c1); cursor1++;
                }
                else{
                        aux.push_back(c2); cursor2++;
                }
        }
        for (int i=0; i<aux.size(); i++)
                v[start+i] = aux[i];
}
```



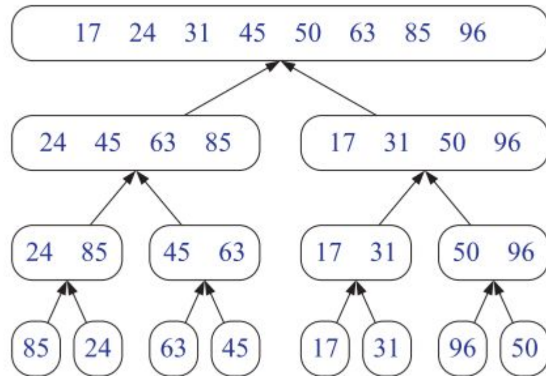
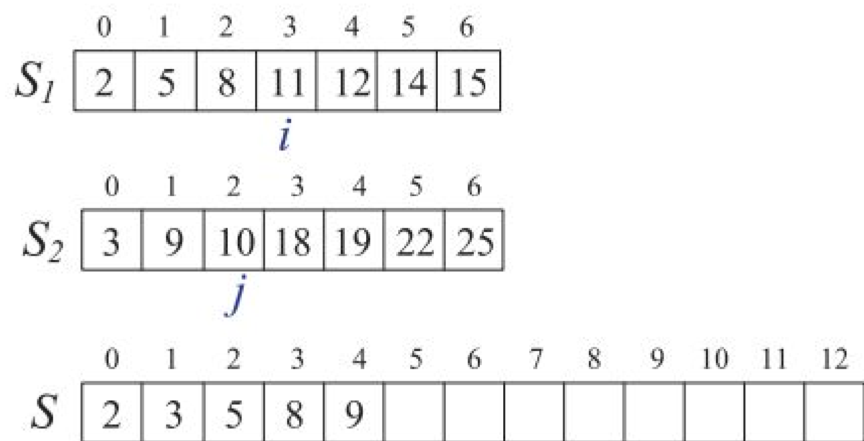


Figure 11.1: Merge-sort tree T for an execution of the merge-sort algorithm on a sequence with eight elements: (a) input sequences processed at each node of T; (b) output sequences generated at each node of T.



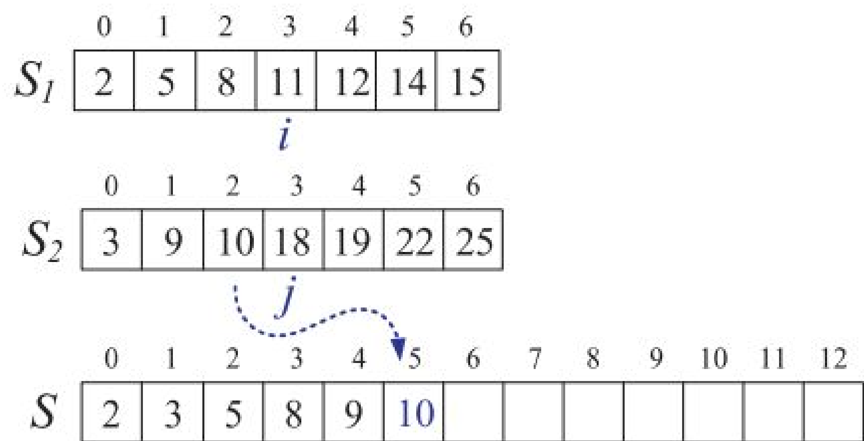


Figure 11.5: A step in the merge of two sorted arrays: (a) before the copy step; (b) after the copy step.

Quick Sort

1. Pick a pivot
2. Partition the container so that (Partitioning explained in the next slides)
    a. pivot is at its right place
    b. if x <= pivot, x is before pivot
    c. if x >= pivot, x is after pivot
3. Recursively sort everything to the left of the pivot
4. Recursively sort everything to the right of the pivot

Check out quick_sort.cpp

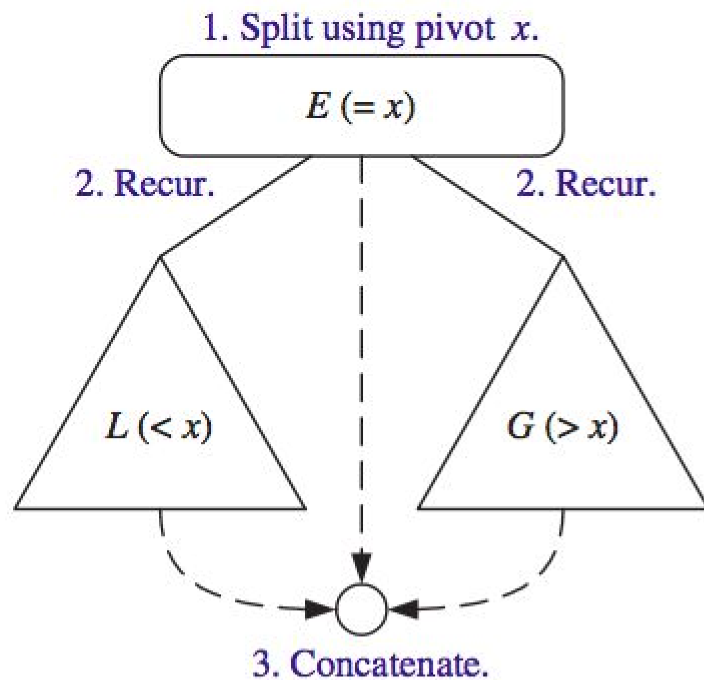


Figure 11.8: A visual schematic of the quick-sort algorithm.

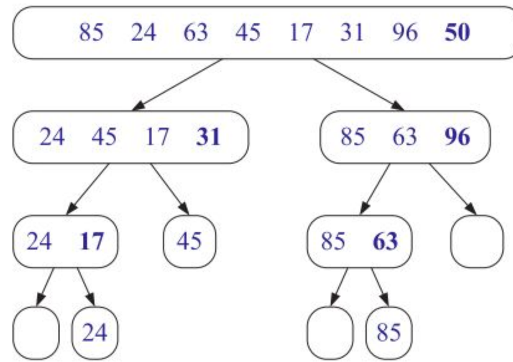
Quick Sort

```
void in_place_quick_sort(vector<int>& v, int start, int end){
    if (end <= start)
        return;
    int pivot_index = partition(v, start, end);
    in_place_quick_sort(v,start, pivot_index - 1);
    in_place_quick_sort(v,pivot_index + 1, end);
}
```

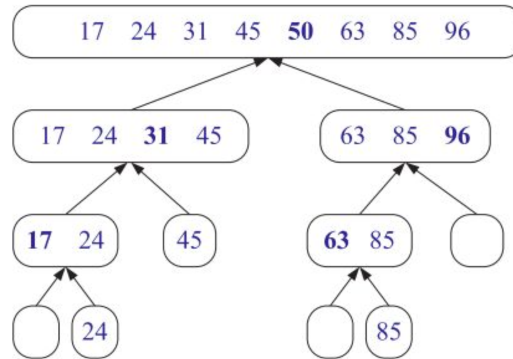
Partition(start, end)

1. Assume item at index end is the pivot
   a. If it's not swap it with item at index end
2. Keep two cursors cursor1 and cursor2 that are initialized to start and end-1
3. cursor1 moves forward until what it points to is greater than pivot
4. cursor2 moves backward until what it points to is smaller than pivot
5. We then swap the items at cursor1 and cursor12 and continue until they pass each other (when cursor2 < cursor1)
6. At this point we swap the item at cursor1 with pivot (item at index end) and report cursor1 as the new index of the pivot

```
int partition (vector<int>& v, int start, int end){
    int pivot_index = end, cursor_1 = start, cursor_2 = end - 1;
    while (cursor_1 < cursor_2){
        while (v[cursor_1] <= v[pivot_index]){
            cursor_1++;
            if (cursor_1 > cursor_2)  break;
        }
        while (v[cursor_2] >= v[pivot_index]){
            cursor_2--;
            if (cursor_1 > cursor_2)  break;
        }
        if (cursor_1 < cursor_2)
            swap(v, cursor_1, cursor_2);
    }
    swap(v, pivot_index, cursor_1);
    return cursor_1;
}
```



(a)



(b)

Figure 11.9: Quick-sort tree T for an execution of the quick-sort algorithm on a sequence with eight elements: (a) input sequences processed at each node of T; (b) output sequences generated at each node of T. The pivot used at each level of the recursion is shown in bold.

Linear Sorts

Bucket sort (a.k.a count sort)

For items in a specific range

Radix sort

For items that contain constant number of components, each in a specific range. Ex: constant length strings

Check out linear_sorts.cpp

Bucket Sort

Keeps an array/vector of counts

Iterates over array/vector and output x count[x] times

Can we use an unordered_map instead of an array/vector?

Radix Sort

Start from rightmost component and go left

Do a **stable** bucket sort based on each component

Reading Material

Sections 11.1 - 11.3 of the textbook