CS213/293 Data Structure and Algorithms 2023

Lecture 14: Sorting

Instructor: Ashutosh Gupta

IITB India

Compile date: 2023-10-02

Sorting

We almost always handle data in a sorted manner.

Computers spend a large percentage of their time in sorting.

We need efficient sorting algorithms.

Many algorithms

There are many sorting algorithms based on various design techniques.

The lower bound of sorting is $\Omega(n \log n)$.

Sorting algorithms

We will discuss the following algorithms for sorting.

- Merge sort
- Quick sort
- Radix sort
- Bucket sort

Topic 14.1

Merge sort

Merge sort : Divide, conquer, and combine

- Divide: Divide the array into two roughly equal parts
- Conquer: Sort each part using mergesort
- Combine: merge the sorted parts

Merge sort

The following code sort array $A[\ell : u-1]$.

Algorithm 14.1: MERGESORT(int* A, int ℓ, int u)

if $u \leq \ell + 1$ **then** **return**;
$mid := (u + \ell)/2$;
MERGESORT(A, ℓ, mid);
MERGESORT(A, mid, u);
MERGE(A, ℓ, p, u);

Merge

Algorithm 14.2: MERGE(int* A, int ℓ, int p, int u)

```
int B[u − ℓ];
i := ℓ;
j := mid;
for  k = 0; k < u − ℓ; k++ do
    if i < mid and (j ≥ u or A[i] ≤ A[j]) then
        B[k] := A[i];
        i := i + 1;
    else
        B[k] := A[j];
        j := j + 1;
MEMCOPY( B, A + ℓ, u − ℓ)
```

We cannot do merge in place.

The running time of MERGE is $\Theta(u - \ell)$.

Exercise 14.1

Give an optimization that will make MERGE *for an ordered array efficient.*

Example: MergeSort

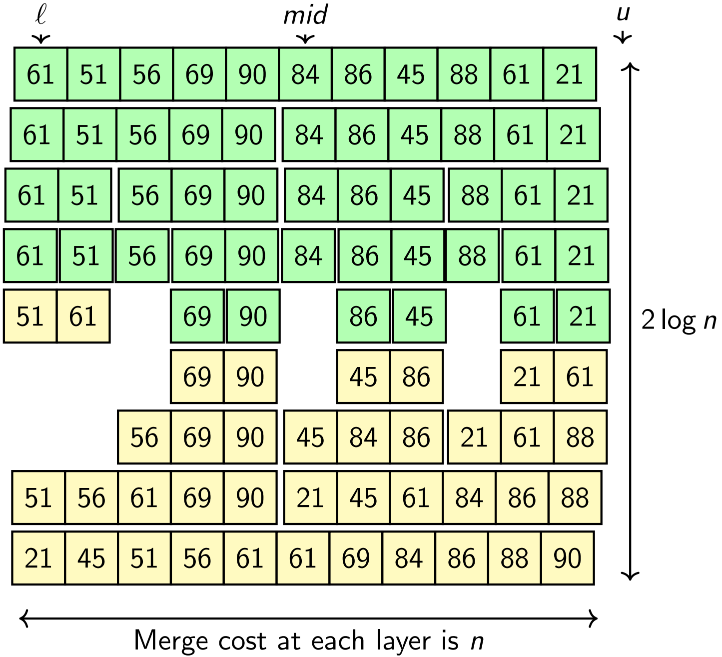
Example 14.1

Consider the following array.

We split the array in the middle and sort the first part.

When partitions are of size one, we start merging. (yellow arrays)

Worst-case complexity: $O(n \log n)$ (visual proof)



Formal proof of the complexity

Let n be the length of the array. Let $T(n)$ be the worst-case complexity of MergeSort.

The following recursive equation defines $T(n)$.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + T(n/2) + \Theta(n) & \text{Otherwise.} \end{cases}$$

After a recursive substitution,

$$T(n) = 4T(n/4) + 2\Theta(n/2) + \Theta(n).$$

Formal proof of the complexity(2)

After k recursive substitution,

$$T(n) = 2^k T(n/(2^k)) + k\Theta(n).$$

If $n = 2^k$,

$$T(n) = 2^k T(1) + k\Theta(n).$$

Therefore,

$$T(n) = n + k\Theta(n).$$

Therefore,

$$T(n) \in O(n \log n).$$

Topic 14.2

Quick sort

Quick sort : Divide and conquer

- Divide: Partition array in two parts such that elements in the lower part $\leq$ elements in the higher part
- Conquer: recursively sort the arrays

Partition

Algorithm 14.3: PARTITION(int* A, int ℓ, int u)

```
pivot := A[ℓ];
i := ℓ − 1;
j := u + 1;
while true do
    do i := i + 1 while pivot > A[i];
    do j := j − 1 while pivot < A[j];
    if i ≥ j then return j ;
    SWAP(A[i], A[j])
```

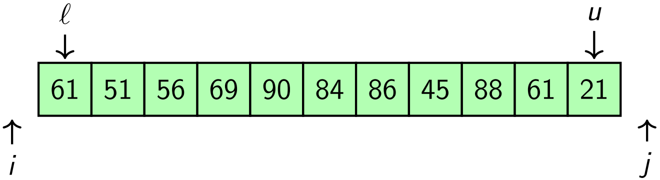
i is incremented at least once.

The running time of PARTITION is $\Theta(n)$. (Why not $O(n)$?)

Example : partition

Example 14.2

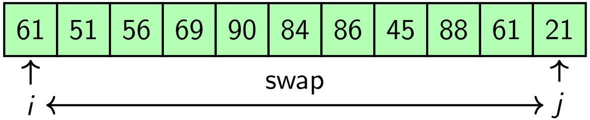
Consider the following input to partition



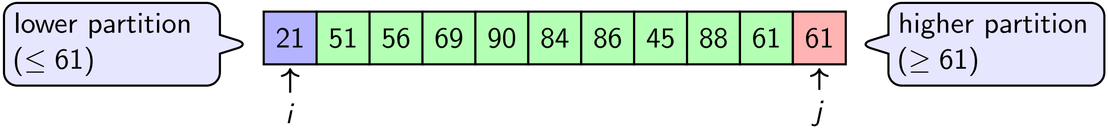
pivot is 61.

Example : the first iteration of partition

After scan

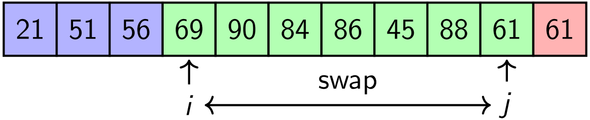


After swap

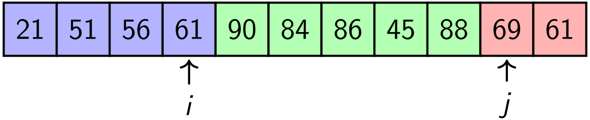


Example : second iteration of partition

After scan

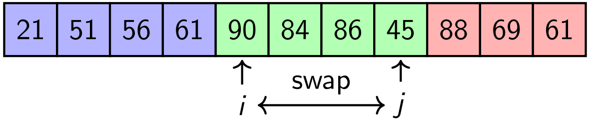


After swap

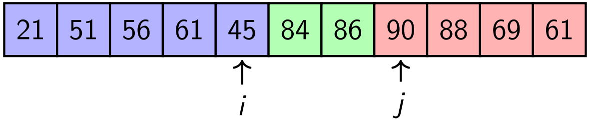


Example : third iteration of partition

After scan

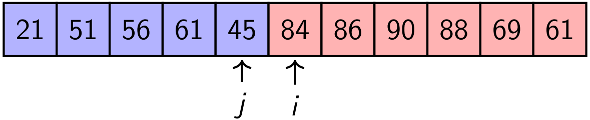


After swap



Example : final iteration of partition

After scan



Since $j \leq i$, the algorithm stops and returns j.

Exercise 14.2

Modify the partition where elements equal to the pivot are moved to the lower partition.

Quick sort

Algorithm 14.4: QUICKSORT(int* A, int ℓ, int u)

```
if u ≥ ℓ then return;
p := PARTITION( A, ℓ, u );
QUICKSORT( A, ℓ, p );
QUICKSORT( A, p + 1, u );
```

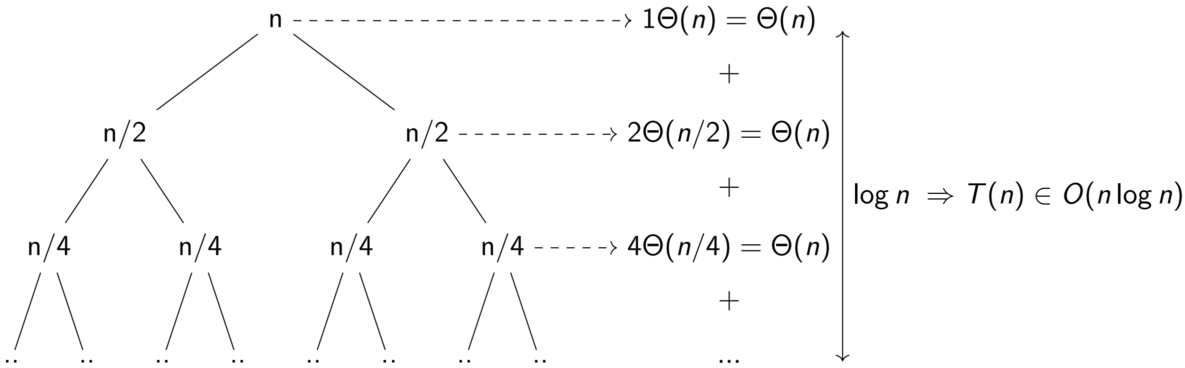
Topic 14.3

Analysis of quick sort

Best case execution

Every time the array splits into two halves.

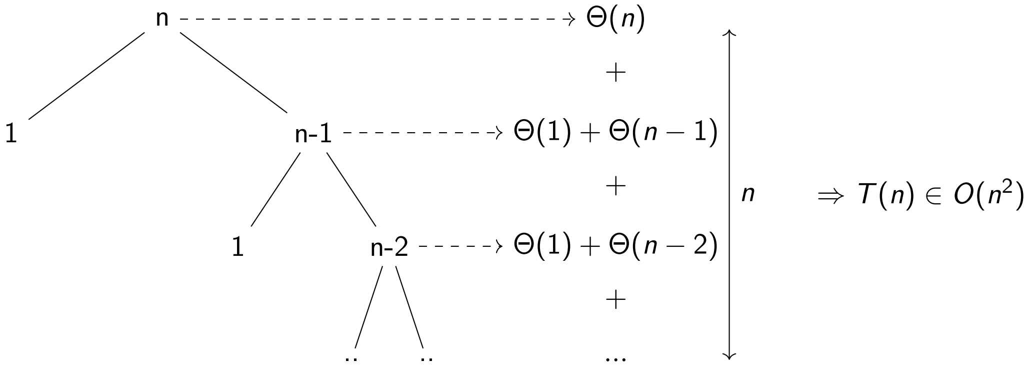
$$T(n) = 2T(n/2) + \Theta(n)$$



Worst case execution

Every time the array splits into an array of size 1 and the rest.

$$T(n) = T(1) + T(n-1) + \Theta(n)$$



When does the worst case occur?

- Worst case occurs on sorted or reverse sorted array. ¯_(ツ)_/¯

Exercise 14.3

Can we modify quick sort such that on a sorted array its running time is $O(n)$*?*

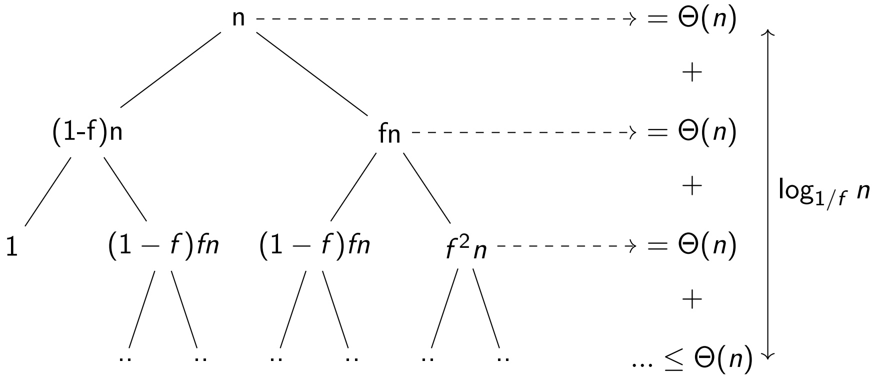
Properties of quick sort

- In-place sorting
- practical, average $O(n \log n)$(with small constants), but worst $O(n^2)$

Other factional splits

Every time the array splits into fn and $(1-f)n$ arrays, where $f > 1/2$.

$$T(n) = T(fn) + T((1-f)n) + \Theta(n)$$



$\Rightarrow T(n) \in O(n \log n)$

Topic 14.4

Randomized quicksort

How to get to the average case?

- Partition around the "middle" element?
- Partition around the random element.

Randomized quicksort

Algorithm 14.5: RANDOMQUICKSORT(int* A, int ℓ, int u)

```
if u ≥ ℓ then return;
i := RANDOM(ℓ, u);
SWAP(A[i], A[ℓ]);
p := PARTITION( A, ℓ, u );
QUICKSORT( A, ℓ, p );
QUICKSORT( A, p + 1, u );
```

Analyzing randomized quicksort

Assume all elements are distinct.

Due to partition around a random element, all splits are equally likely.

Analyzing Randomized quicksort

Definition 14.1

Let $T(n)$ be the expected number of comparisons needed to quicksort n numbers.

Since each split occurs with probability $1/n$, $T(n)$ has value $T(i-1) + T(n-i) + n - 1$ with probability $1/n$.

$$T(n) = \frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i) + n - 1) = \frac{2}{n}\sum_{i=0}^{n-1} T(i) + n - 1$$

cc by nc sa

Solving the recurrence relation

We have seen the recurrence relation before in inserting permutations of numbers in BST.

We had proven $T(n) \in O(n \log n)$

What does expected running time mean?

- Algorithm may have different running time at different times
- Running times are input-independent. It depends on random number generators.

Topic 14.5

Radix sort

Radix partition

Sort the array according to the bth bit.

Algorithm 14.6: RADIXPARTITION(int* A, int b, int ℓ, int u)

$i := \ell - 1$; $j := u + 1$;
while *true* **do**
    do $i := i + 1$ **while** $2^b \& A[i] = 0$;
    do $j := j - 1$ **while** $2^b \& A[i] \neq 0$;
    if $i \geq j$ **then** **return** j ;
    SWAP$(A[i], A[j])$

Example: radix partition

Example 14.3

Consider the following numbers.

- 61 = 0111101
- 51 = 0110011
- 56 = 0111000
- 69 = 1000101
- 90 = 1011010
- 84 = 1010100
- 86 = 1010110
- 45 = 0101101
- 88 = 1011000
- 61 = 0111101
- 21 = 0010101

radix partition divides the numbers as follows.

- 61 = 0111101
- 51 = 0110011
- 56 = 0111000
- 45 = 0101101
- 61 = 0111101
- 21 = 0010101
- 69 = 1000101
- 90 = 1011010
- 84 = 1010100
- 86 = 1010110
- 88 = 1011000

RadixSort

We start from the most significant bit (leftmost bit).

Algorithm 14.7: RadixSort(int* A, int ℓ, int u)

Let b be the number of bits needed for the largest number in A;
RadixSortRec(A, b, ℓ, u);

Algorithm 14.8: RadixSortRec(int* A, int b, int ℓ, int u)

if $u \geq \ell$ *and* $b \geq 0$ **then return**;
$p :=$ RadixPartition(A, b, ℓ, u);
RadixSortRec(A, $b - 1$,ℓ, p);
RadixSortRec(A, $b - 1$,$p + 1$, u);

Worst-case run time complexity: $O(bn)$

Exercise 14.4

Is this complexity better than $O(n \log n)$?

Topic 14.6

Bucket sort

Bucket sort

Radix sort does well when $b < \log n$, which implies low variation and high repetition among keys.

Let us suppose that we know that the numbers in the array are from set $\{0, ..., m-1\}$.

We can put the keys in a hash table of size m, where the hash function is identity. Keys will be automatically sorted.

Complexity: $O(m+n)$.

Exercise 14.5

Give a situation when bucket sort will be the best sorting algorithm?

Topic 14.7

Why $O(n \log n)$?

Comparison sort

A sorting algorithm takes an input sequence and produces a permutation of the input.

Definition 14.2

A comparison sort determines the sorted order only based on comparisons between the input elements.

Let us assume our input has only distinct elements. All we need to do is strict comparison.

Decision tree model of executions.

Imagine a comparison sort algorithm, that needs to identify the sorting permutation.

It executes a sequence of comparisons over the elements of inputs.

Example: Comparison sort

Let us consider BUBBLESORT, which is an example comparison sort.

Algorithm 14.9: BUBBLESORT(int* A, int n)

```
for  i = 0; i < n − 1; i++ do
    for  j = 0; j < n − i − 1; j++ do
        if A[j] > A[j + 1] then
            SWAP(A, j, j+1);
```

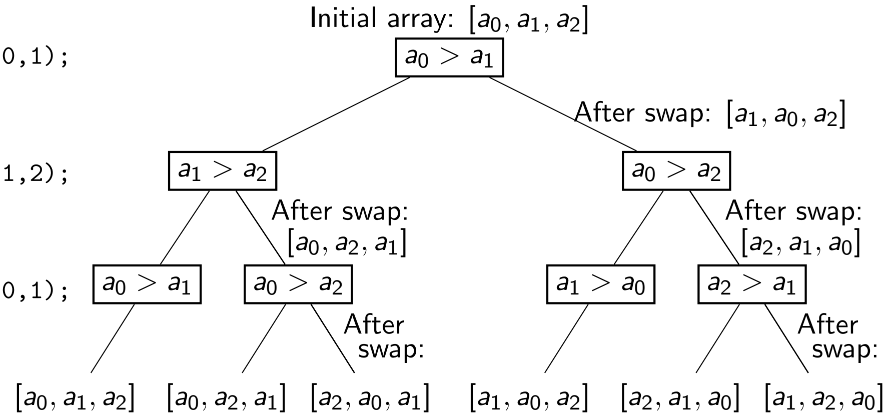
Let us suppose $A = [a_0, a_1, a_2]$.

We will run of the algorithm in all possible executions and create an execution tree.

Example: Execution tree of BubbleSort

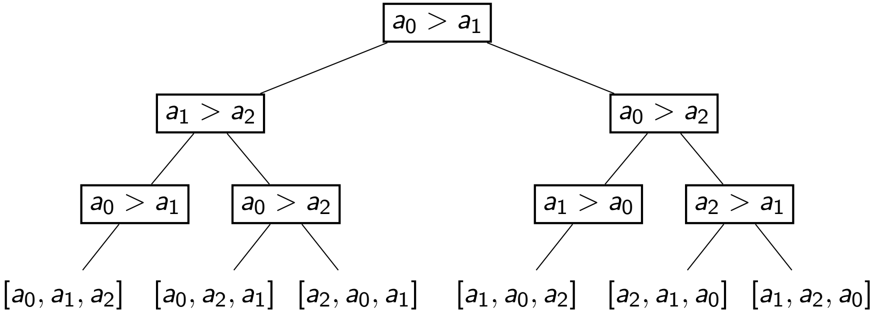
Now let us consider all possible executions of BubbleSort. We compare elements of the array if the condition holds(right child), we run swap.

```
if(A[0]>A[1]) swap(A,0,1);

if(A[1]>A[2]) swap(A,1,2);

if(A[0]>A[1]) swap(A,0,1);
```



Structure of the execution tree of any possible comparison sort

Let us ignore the array updates and the intermediate states. In the tree, we only track the elements (not positions) compared and the final permutation.



We can create the abstraction of the execution tree of any comparison sort algorithm.

The above is a binary tree that has $n!$ leaves.

Best worst-case performance

Theorem 14.1

Any comparison sort algorithm requires $\Omega(n \log n)$ *comparisons in the worst case.*

Proof.

An execution tree has $n!$ leaves.

The minimum height of the tree is $O(\log(n!))$, which is $O(n \log n)$. Hence proved! $\square$

Topic 14.8

Tutorial Problems

Exercise: sorting a sorted array

Exercise 14.6

a. Modify the PARTITION *such that it detects that the array is already sorted. b. Can we use this modification to improve quick sort?*

Exercise: straight radix sort

Exercise 14.7

Modify RADIXSORT *such that it considers bits from right to left.*

Commentary: We need to ensure that later iterations of sorting do not disrupt earlier work.

Exercise: Execution tree for Heap sort

Exercise 14.8

Draw the execution tree for the Heap sort. Let us suppose the input array is of size 3.

Exercise: Complexity

Exercise 14.9

Prove that $O(\log(n!)) = O(n \log n)$. *Hint: Stirling's approximation*

Topic 14.9

Problem

Exercise: In-place sorting

Exercise 14.10

Give the algorithms that are not in-place sorting algorithms. An algorithm is in-place sorting algorithm if does not use more than $O(1)$ extra space and update is only via replace or swap.

End of Lecture 14