Lecture 2: Divide&Conquer Paradigm, Merge sort and Quicksort

Instructor: Saravanan Thirumuruganathan

Outline

1. Divide and Conquer
2. Merge sort
3. Quick sort

In-Class Quizzes

- **URL:** `http://m.socrative.com/`
- **Room Name: 4f2bb99e**

Divide And Conquer Paradigm

- D&C is a popular algorithmic technique
- Lots of applications
- Consists of three steps:
    1. **Divide** the problem into a number of sub-problems
    2. **Conquer** the sub-problems by solving them *recursively*
    3. **Combine** the solutions to sub-problems into solution for original problem

Divide And Conquer Paradigm

When can you use it?

- The sub-problems are easier to solve than original problem
- The number of sub-problems is **small**
- Solution to original problem can be obtained easily, once the sub-problems are solved

Recursion and Recurrences

- Typically, D&C algorithms use **recursion** as it makes coding simpler
- Non-recursive variants can be designed, but are often slower
- If all sub-problems are of equal size, can be analyzed by the recurrence equation

$$T(n) = aT(\frac{n}{b}) + D(n) + C(n)$$

- a: number of sub-problems to solve
- b: how fast the problem size shrinks
- $D(n)$: time complexity for the divide step
- $C(n)$: time complexity for the combine step

D&C Approach to Sorting

How to use D&C in Sorting?

- Partition the array into sub-groups
- Sort each sub-group recursively
- Combine sorted sub-groups if needed

Why study Merge Sort?

- One of the simplest and efficient sorting algorithms
- Time complexity is $\Theta(n \log n)$ (vast improvement over Bubble, Selection and Insertion sorts)
- Transparent application of D&C paradigm
- Good showcase for time complexity analysis

Merge Sort

High Level Idea :

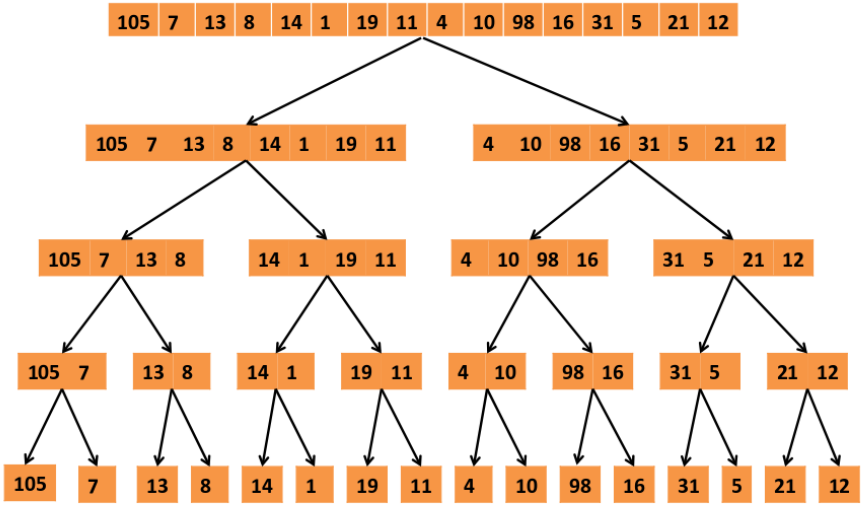
- Divide the array into two **equal** partitions - L and R
    - If not divisible by 1, L has $\lfloor \frac{n}{2} \rfloor$ elements and R has $\lceil \frac{n}{2} \rceil$
- Sort left partition L recursively
- Sort right partition R recursively
- Merge the two sorted partitions into the output array

Merge Sort Pseudocode

Pseudocode:

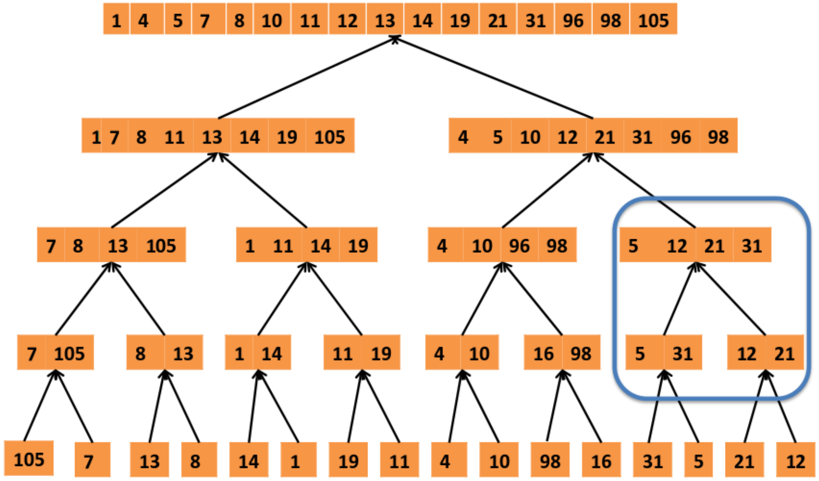
```
MergeSort(A, p, r):
    if p < r:
        q = (p+r)/2
        Mergesort(A, p , q)
        Mergesort(A, q+1, r)
        Merge(A, p, q, r)
```

Merge Sort - Divide[1]



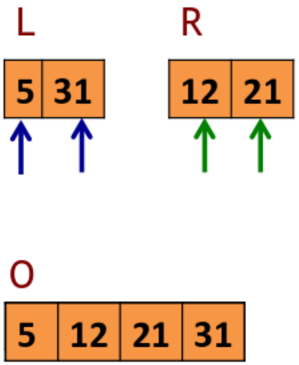
[1]http://web.stanford.edu/class/cs161/slides/0623_mergesort.pdf

Merge Sort - Combine[2]



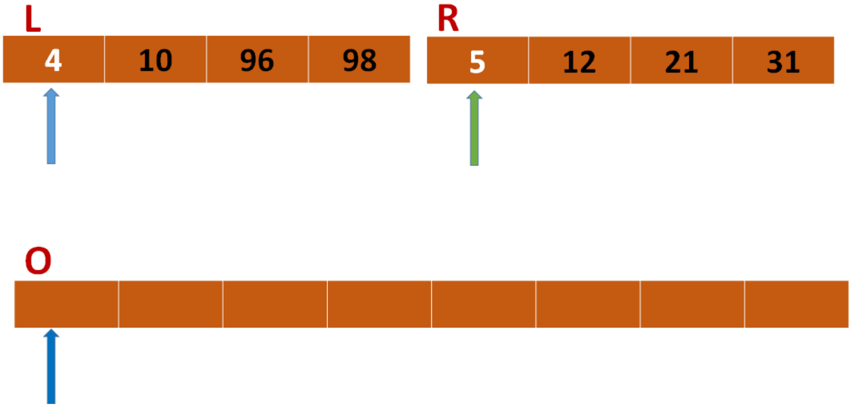
[2] http://web.stanford.edu/class/cs161/slides/0623_mergesort.pdf

Merging Two Sorted Lists[3]

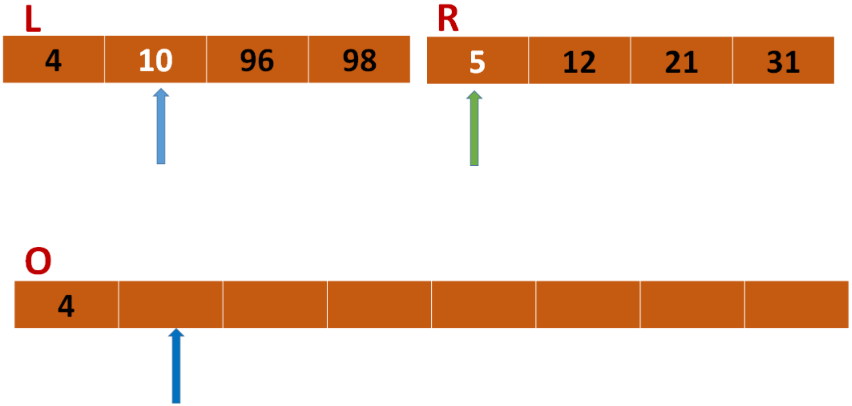


[3]http://web.stanford.edu/class/cs161/slides/0623_mergesort.pdf

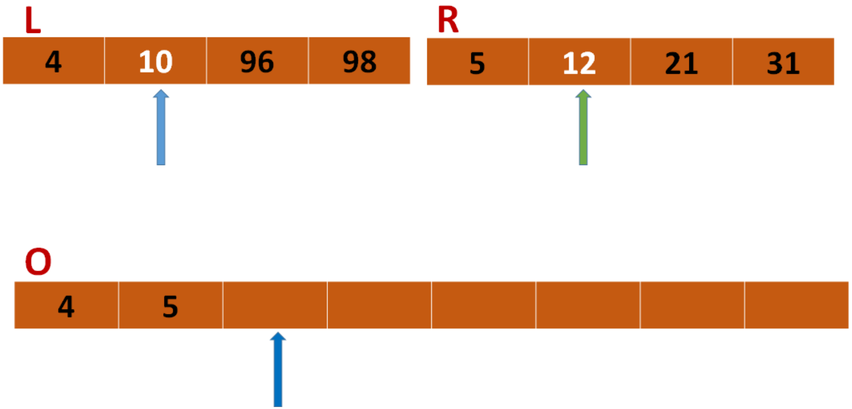
Merging Two Sorted Lists



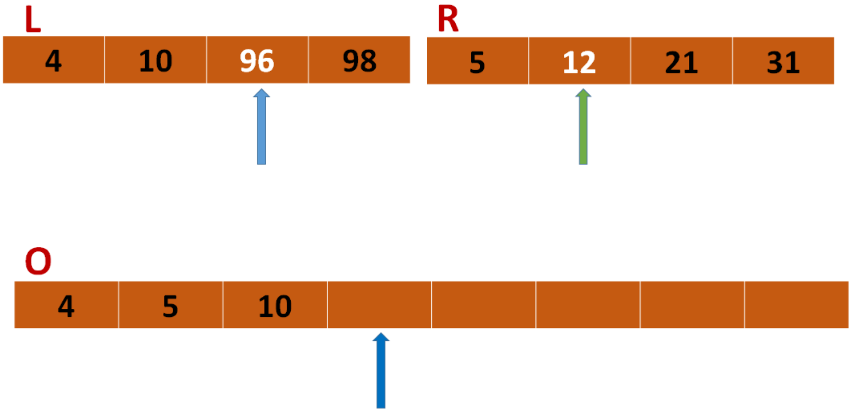
Merging Two Sorted Lists



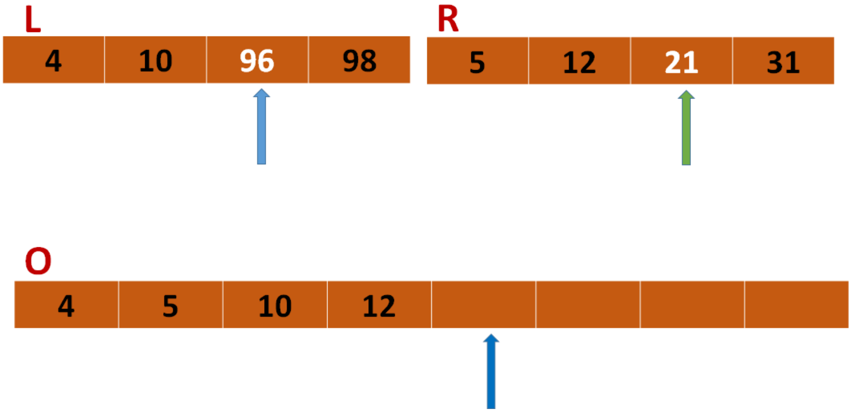
Merging Two Sorted Lists



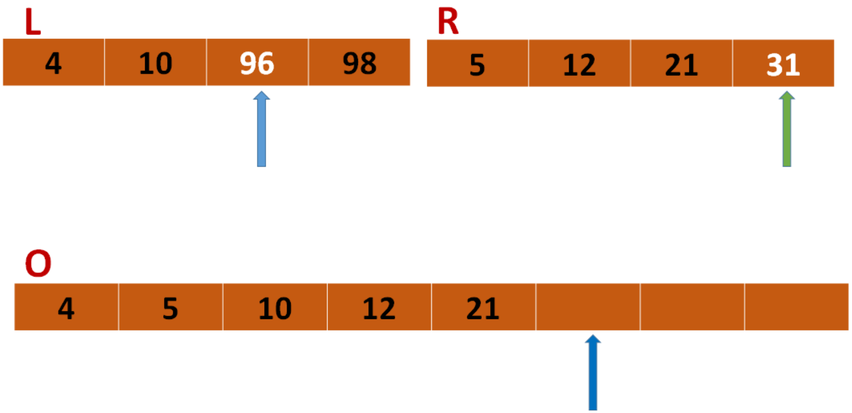
Merging Two Sorted Lists



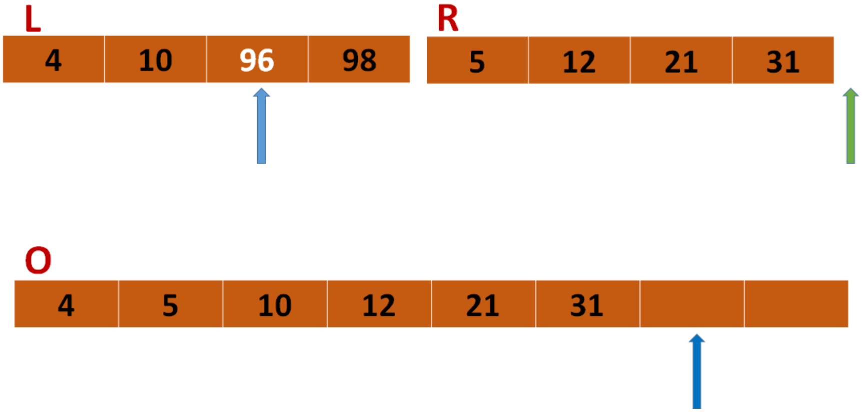
Merging Two Sorted Lists



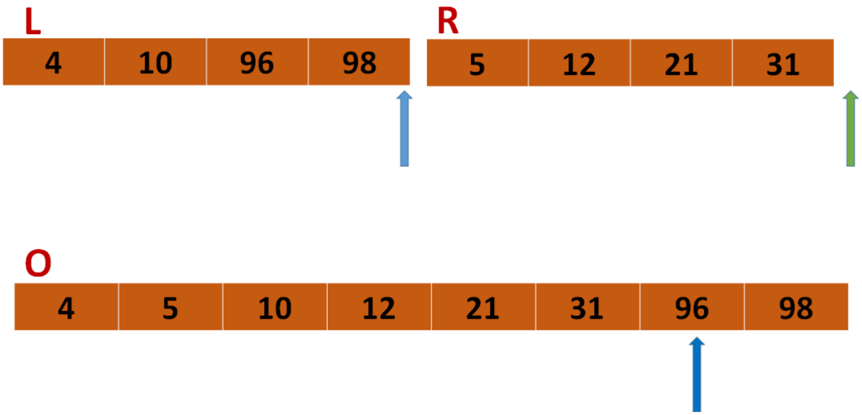
Merging Two Sorted Lists



Merging Two Sorted Lists



Merging Two Sorted Lists



Merging Two Sorted Lists

Merge Pseudocode:

```
Merge(A,B,C):
    i = j = 1
    for k = 1 to n:
        if A[i] < B[j]:
            C[k] = A[i]
            i = i + 1
        else: (A[i] > B[j])
            C[k] = B[j]
            j = j + 1
```

Analyzing Merge Sort: Master Method

Quiz!

- General recurrence formula for D&C is $T(n) = aT(\frac{n}{b}) + D(n) + C(n)$
- What is a?
- What is b?
- What is $D(n)$?
- What is $C(n)$?

Analyzing Merge Sort: Master Method

Quiz!

- General recurrence formula for D&C is

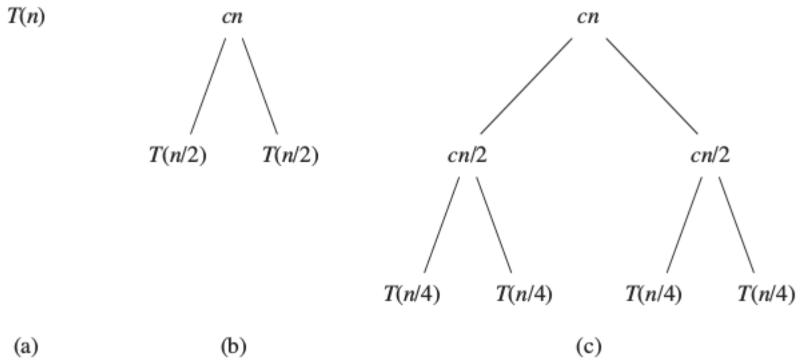
$$T(n) = aT(\frac{n}{b}) + D(n) + C(n)$$

- $a = 2,\ b = 2$
- $D(n) = O(1)$
- $C(n) = O(n)$
- Combining, we get:

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

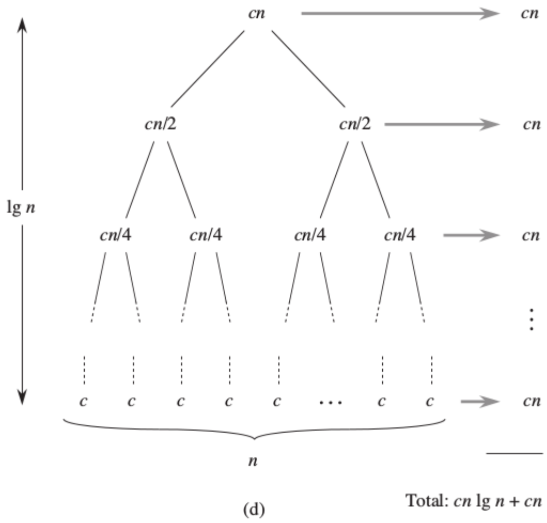
- Using Master method, we get $T(n) = O(n \log n)$
- If you are picky, $T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + O(n)$

Analyzing Merge Sort: Recursion Tree[4]



[4] CLRS Book

Analyzing Merge Sort: Recursion Tree[5]



[5] CLRS Book

Merge Sort Vs Insertion Sort

- Merge Sort is very fast in general ($O(n \log n)$) than Insertion sort ($O(n^2)$)
- For "nearly" sorted arrays, Insertion sort is faster
- Merge sort has $\Theta(n \log n)$ (i.e. both best and worst case complexity is $n \log n$
- Overhead: recursive calls and extra space for copying
- Insertion sort is in-place and adaptive
- Merge sort is easily parallizable

Quicksort

- Quicksort is a *very* popular and elegant sorting algorithm
- Invented by Tony Hoare
    - Also invented the concept of NULL (he called it a Billion dollar mistake! - why?)
    - "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

Quicksort

- Fastest of the fast sorting algorithms and lot (and lots) of ways to tune it.
- Default sorting algorithm in most languages
- Simple but innovative use of D&C
- On average it takes $\Theta(n \log n)$ but in worst case might require $O(n^2)$
    - Occurs rarely in practice **if** coded properly

Quicksort

- Quicksort is a D&C algorithm and uses a different style than Merge sort
- It does more work in Divide phase and almost no work in Combine phase
    - One of the very few algorithms with this property

Partitioning Choices[6]

- Sorting by D&C - Divide to two sub-arrays, sort each and merge
- Different partitioning ideas leads to different sorting algorithms
- Given an array A with n elements, how to split to two sub-arrays L and R
    - L has first $n-1$ elements and R has last element
    - R has largest element of A and L has rest of $n-1$ elements
    - L has the first $\lfloor \frac{n}{2} \rfloor$ elements and R has the rest
    - Chose a pivot p, L has elements less than p and R has elements greater than p

[6] From
`http://www.cs.bu.edu/fac/gkollios/cs113/Slides/quicksort.ppt`

Partitioning Choices

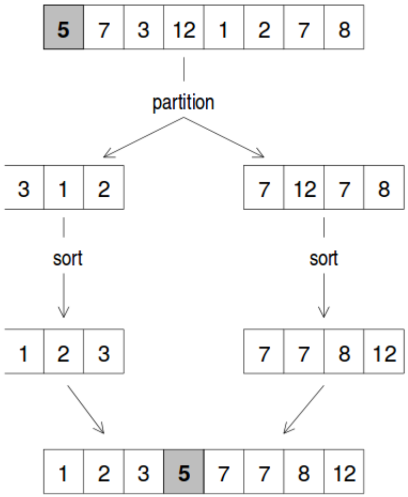
- D&C is not a silver bullet!
- Different partitioning ideas leads to different sorting algorithms
  - **Insertion Sort** $O(n^2)$: L has first $n-1$ elements and R has last element
  - **Bubble Sort** $O(n^2)$: R has largest element of A and L has rest of $n-1$ elements
  - **Merge Sort** $O(n \log n)$: L has the first $\lfloor \frac{n}{2} \rfloor$ elements and R has the rest
  - **Quick Sort** $O(n \log n)$ (average case): Chose a pivot p, L has elements less than p and R has elements greater than p

Quicksort

Pseudocode:

```
QuickSort(A, p, r):
    if p < r:
        q = Partition(A, p, r)
        QuickSort(A, p, q-1)
        QuickSort(A, q+1, r)
```

QuickSort


5 7 3 12 1 2 7 8
partition
3 1 2
7 12 7 8
sort
sort
1 2 3
7 7 8 12
1 2 3 5 7 7 8 12

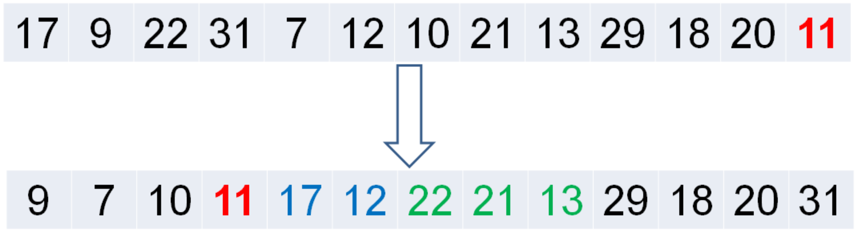
Quicksort Design Objectives

- Choose a good pivot
- Do the partitioning - efficiently and in-place

Partition Subroutine

- Given a pivot, partition A to two sub-arrays L and R.
- All elements less then pivot are in L
- All elements greater than pivot are in R
- Return the new index of the pivot after the rearrangement
- Note: Elements in L and R need not be sorted *during* partition (just $\leq$ pivot and $\geq$ pivot respectively)

Partition Subroutine

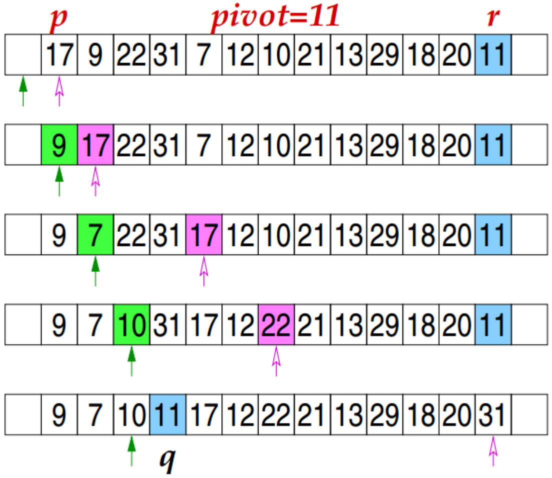


Partition Subroutine

Note: CLRS version of Partition subroutine. Assumes last element as pivot. To use this subroutine for other pivot picking strategies, swap pivot element with last element.

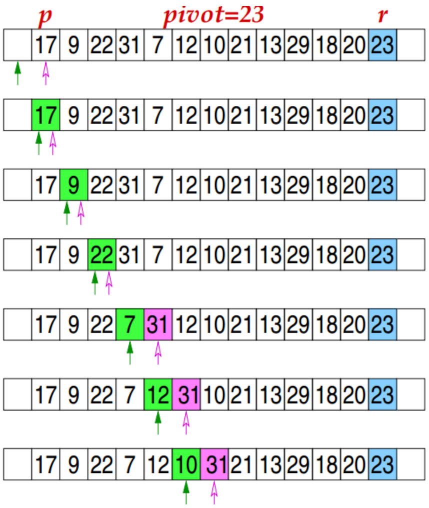
```
Partition(A, p, r):
    x = A[r] // x is the pivot
    i = p - 1
    for j = p to r-1
        if A[j] <= x
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i+1] with A[r]
    return i+1
```

Partition Example 1[7]



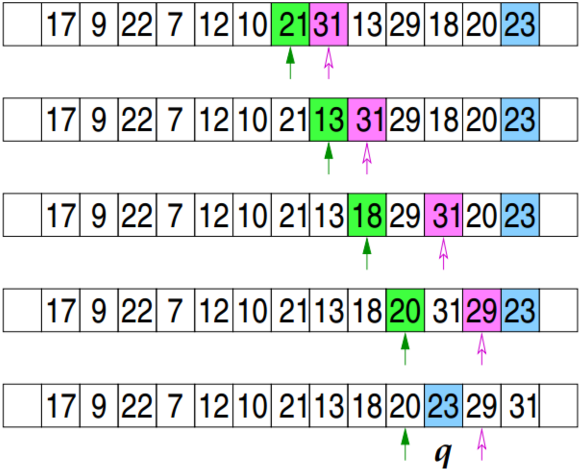
[7] `https://www.cs.rochester.edu/~gildea/csc282/slides/C07-quicksort.pdf`

Partition Example 2[8]



[8]https://www.cs.rochester.edu/~gildea/csc282/slides/C07-quicksort.pdf

Partition Example 2[9]

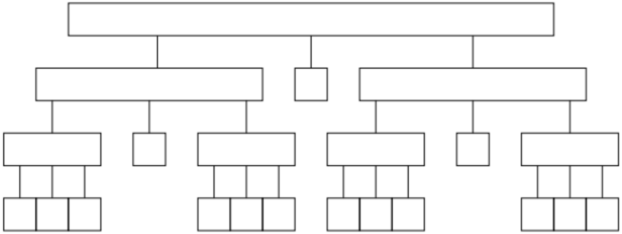


[9] https://www.cs.rochester.edu/~gildea/csc282/slides/C07-quicksort.pdf

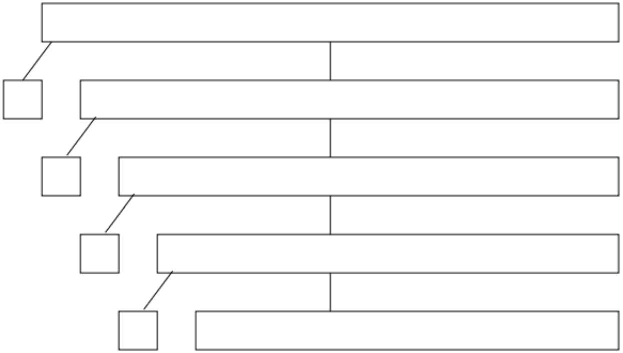
Partition Subroutine

- Time Complexity: $O(n)$ - why?
- Correctness: Why does it work?

Quicksort: Best Case Scenario



Quicksort: Worst Case Scenario

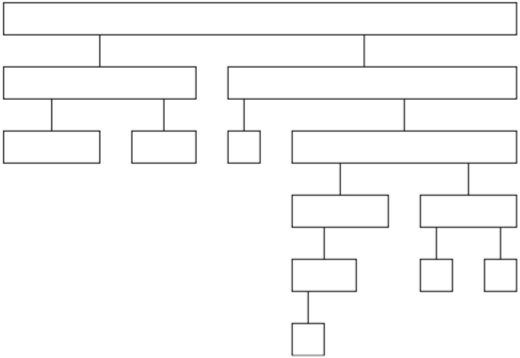


Quicksort: Analysis

Recurrence Relation for Quicksort:

- Best Case: $T(n) = 2T(\frac{n}{2}) + n$
  - Same as Merge sort: $\Theta(n \log n)$
- Worst Case: $T(n) = T(n-1) + n$
  - Same as Bubble sort: $O(n^2)$
- Average Case: $T(n) = \sum_{p=1}^{n} \frac{1}{n} (T(p-1) + T(n-p)) + n$
  - Analysis is very tricky, but returns $O(n \log n)$
  - Intuition: Even an uneven split is okay (as long as it is between 25:75 to 75:25)
  - When looking at all possible arrays of size n, we expect (on average) such a split to happen half the time

Quicksort: Average Case Scenario



Strategies to Pick Pivot

- Pick first, middle or last element as pivot
- Pick median-of-3 as pivot (i.e. median of first, middle and last element)
- **Bad News:** All of these strategies work well in practice, but has worst case time complexity of $O(n^2)$
- Pick the median as pivot

Randomized Quicksort

```
Randomized-Partition(A, p, r):
    i = Random(p,r)
    Exchange A[r] with A[i]
    return Partition(A, p, r)

Randomized-QuickSort(A, p, r):
    if p < r:
        q = Randomized-Partition(A, p, r)
        Randomized-QuickSort(A, p, q-1)
        Randomized-QuickSort(A, q+1, r)
```

Randomized Quicksort

- Adversarial analysis
- It is easy to construct a worst case input for every deterministic pivot picking strategy
- Harder to do for randomized strategy
- Idea: Pick a pivot randomly or shuffle data and use a deterministic strategy
- *Expected* time complexity is $O(n \log n)$

Sorting in the Real World

- Sorting is a fundamental problem with intense ongoing research
- No single best algorithm - Merge, Quick, Heap, Insertion sort all excel in some scenarions
- Most programming languages implement sorting via tuned QuickSort (e.g. Java 6 or below) or a combination of Merge Sort and Insertion sort (Python, Java 7, Perl etc).

Summary

Major Concepts:

- D&C - Divide, Conquer and Combine
- Merge and Quick sort
- Randomization as a strategy