CS58000 Algorithm Design, Analysis, And Implementation

Ⓒ Xiyuan Chen

Fall Term, 2023

Information

- Instructor: Mikhail J Atallah, Tamal Krishna Dey
- Course website is here.
- No group project.

1 Background, sorting and searching

1.1 Big-O & Big-Omega

1.1.1 Running time analysis

- **Worst case running time.** Obtain bound on largest possible running time of the algorithm on input of a given size N.
- **Average case running time.** Obtain bound on running time of algorithm on random input as a function of input size N.
  - Hard to accurately model real instances.
  - Algorithm tuned for a certain distribution may perform poorly on other inputs.

1.1.2 Asymptotic order of growth

- **Upper bounds.** $T(n)$ is $\mathcal{O}(f(n))$ if there exist constant $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.
- **Lower bounds.** $T(n)$ is $\Omega(f(n))$ if there exist constant $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.
- **Tight bounds.** $T(n)$ is $\times(f(n))$ if there exist constant $c_1 > 0$, $c_2 > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$.

We use "∈" symbol to state the relationship between running time and upper bound. E.g. $T(n) \in \mathcal{O}(f(n))$.

1.2 Quick Sort

Quick Sort is a divide-and-conquer sorting algorithm that have average time complexity of $\mathcal{O}(nlogn)$ and have decent space complexity since it only involves swapping.

1.2.1 Idea

Pick a pivot element in the target list. Divide the whole list into **two** parts: one with all elements **smaller** than the pivot and the other one with all elements **greater** than the pivot(where to put the pivot is up to the actual implementation). Do this recursively on both lists and we result in getting a sorted list.

this is done by **Partition**, see below.

1.2.2 Partition

Partition(see more on the slide) is the core component of Quick Sort algorithm. It processes a list such that elements that are smaller than the pivot get gathered together and the same for elements larger than the pivot. There are multiple ways to do that, here we introduce one approach:

Assume we have a list A with starting index p and ending index q.

1. Initialize two pointers i and j. Let $i = p - 1$ and $j = q + 1$.
2. Pick a pivot called P(leftmost, rightmost or just random).
3. Keep decrementing j until we find a $A[j]$ such that $A[j] \leq P$, i.e. from right to left, find one element that is smaller or equal to the pivot.
4. Keep incrementing i until we find a $A[i]$ such that $A[i] \leq P$, i.e. from left to right, find one element that is greater or equal to the pivot.

5. If $i < j$, swap $A[i]$ and $A[j]$, i.e. if i and j haven't meet each other, swap the elements they are pointed to. Then go back to **step 3**.

   Otherwise, declare j as the dividing boundary(i.e. elements before index $j + 1$ are all smaller than or equal to P and naturally all elements greater than or equal to P are gathered after index j.)

6. Done. Now we have two partitions which add up to the whole list such that one contains all smaller elements and the other contains all larger elements. Notice that these parts themselves may not be sorted so we need to do recursive calls until there is only one, two or three elements in the input list, then it is guaranteed to be sorted after partition.

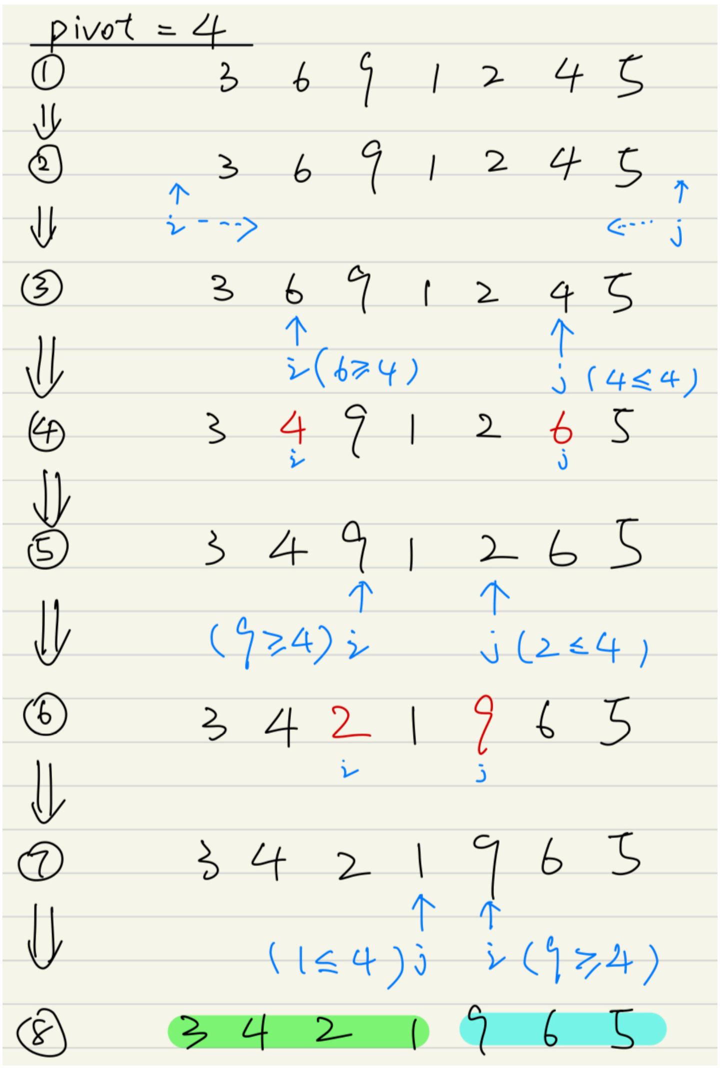

Figure 1: Partition example walk-through.

The intuition is that we want all small numbers grouped on the left and all large numbers grouped on the right. So whenever we have a small number relatively on the right, we swap it with a large number relatively on the left. Notice that when the two pointers meet with each other, by definition, it must be true that numbers after index j are all greater than the pivot and numbers before index i are all smaller than the pivot.

1.3 Heap Sort