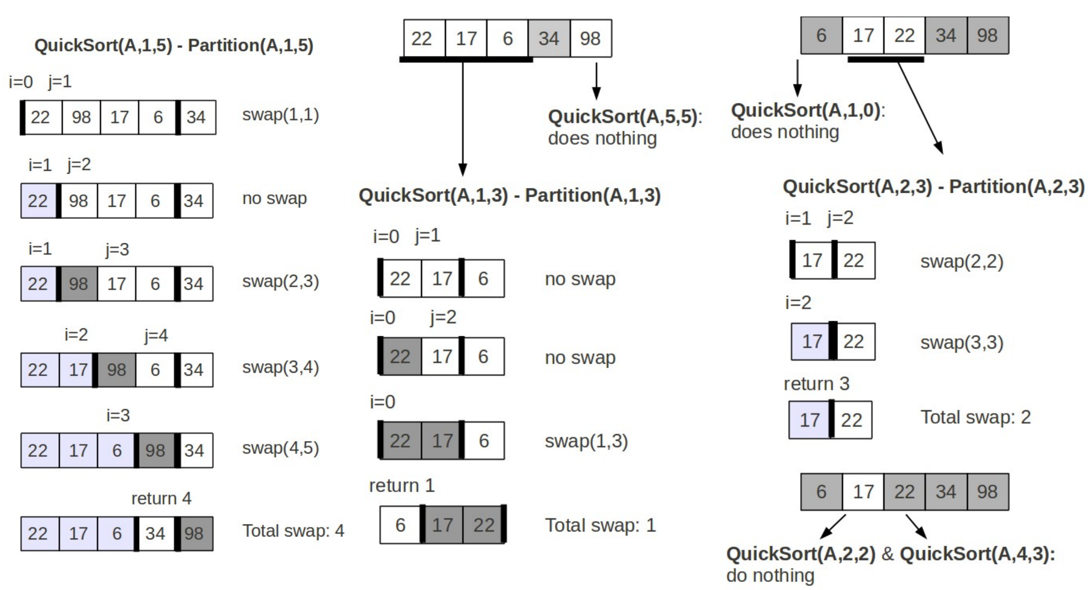
Analysis of Algorithms 91.404, Fall, 2012
Instructor: Prof. Benyuan Liu / bliu@cs.uml.edu
TA: Jian Lu / jlu1@cs.uml.edu
Homework #6

1. (20 points) **QuickSort: array A=<22, 98, 17, 6, 34>.**
(a) (10 points) **Illustrate the operation of QuickSort using Fig. 7.1 on p. 172 as a model for the operation of Partition.**
(b) (10 points) **How many swaps are performed by QuickSort to sort array A? How does this compare with the number of swaps used by HeadSort for this same array.**

Ans:
(a) Firstly, apply QuickSort(A, 1, 5) on A, there Partition(A,1,5) performs 4 swaps and return 4. Then, continue to apply QuickSort on the partitions, QuickSort(A,1,3) will execute Partition(A,1,3) which performs 1 swap and return 1, and QuickSort(A,5,5) will do nothing. After that, QuickSort(A,1,0) and QuickSort(A,2,3) will be executed. QuickSort(A,2,3) will execute Partition(A,2,3) which takes 2 swaps and return 3, while QuickSort(A,1,0) does nothing. The QuickSort(A,1,5) will finish after QuickSort(A,2,2) and QuickSort(A,4,3) finished with no effect.



(b) The total swap number of QuickSort(A,1,5) is 4+1+2 = 7. Compared to HeapSort on A which has 8 swaps, the performance of QuickSort is better than that of HeapSort.

2. (20 points) **QuickSort Analysis: Textbook Exercise 7.4-2 on p. 184.**

Ans:

Method1:

The best-case time for procedure QUICKSORT on an input of size n is:

$$T(n) = \min_{0 \le q \le n-1} (T(q) + T(n-q-q)) + \Theta(n)$$

We guess $T(n) \ge cnlgn$ for some constant c, we obtain:

$$T(n) \ge \min_{0 \le q \le n-1} (cqlgq + c(n-q-1)lg(n-q-1)) + \Theta(n)$$

Let $F(q) = cqlgq + c(n-q-1)lg(n-q-1)$, we have:

$$\begin{aligned} F'(q) &= lgq + q\frac{1}{qln2} + [lg(n-q-1) + (n-q-1)\frac{1}{(n-q-1)ln2}](-1) \\ &= lgq - lg(n-q-1) \end{aligned}$$

Let $F'(q)=0$, we know that $F(q)$ will get the minimum value when $q=(n-1)/2$. Therefore,

$$\begin{aligned} T(n) &\ge c \cdot F(\frac{n-1}{2}) + \Theta(n) \\ &= c[\frac{n-1}{2} lg\frac{n-1}{2} + \frac{n-1}{2} lg\frac{n-1}{2}] + \Theta(n) \\ &= cnlg\frac{n-1}{2} - clg\frac{n-1}{2} + \Theta(n) \\ &= cnlg(n-1) - c(nlg2 + lg\frac{n-1}{2}) + \Theta(n) \\ &\ge cnlg(\frac{n}{2}) - c(n\,lg2 + lg\frac{n-1}{2}) + \Theta(n) (\text{since}\, n-1 \ge \frac{n}{2}\, \forall\, n \ge 2) \\ &\ge cnlgn - c((2\text{lg}2)n + lg\frac{n-1}{2}) + \Theta(n) \\ &\ge cnlgn \end{aligned}$$

since we can pick the constant c small enough so that the term $c(2\text{n}\,lg2 + lg\frac{n-1}{2})$ will be dominated by the $\Theta(n)$ term. Thus, $T(n) = \Omega(n)$.

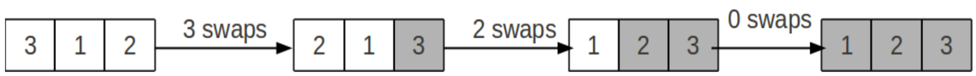
3. (20 points) **Suppose that the PARTITION procedure of QUICKSORT is modified to always use the first element as the pivot:**

(a) Provide a worst-case example for QUICKSORT.

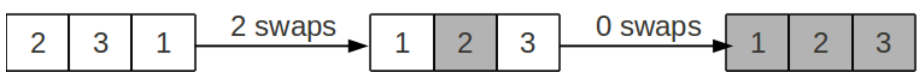
(b) Provide a best-case example for QUICKSORT.

Ans:

(a) Worst-case: <3,1,2>



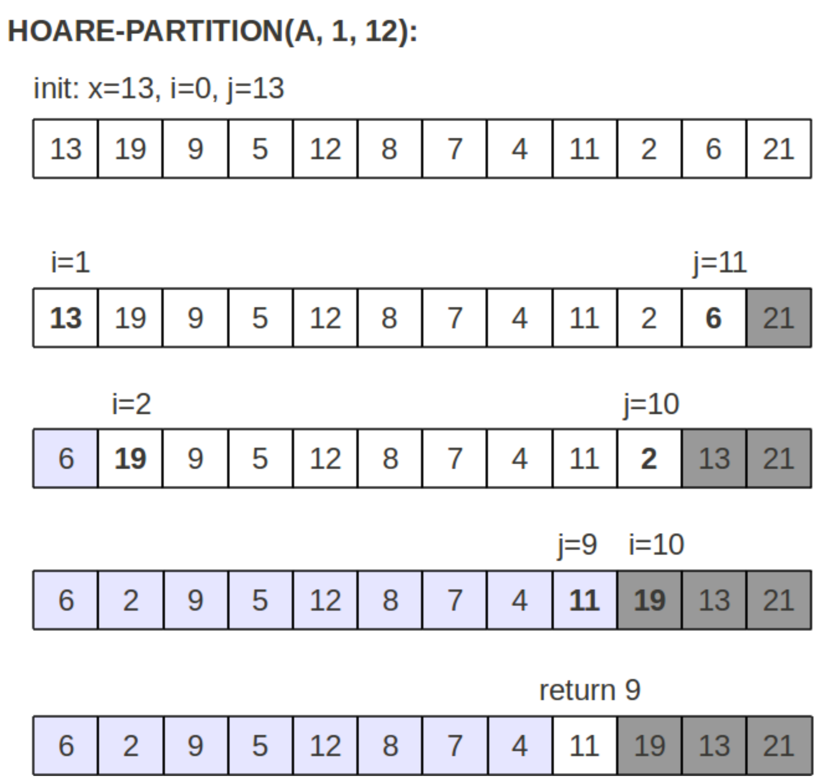
(b) Best-case: <2, 3, 1>



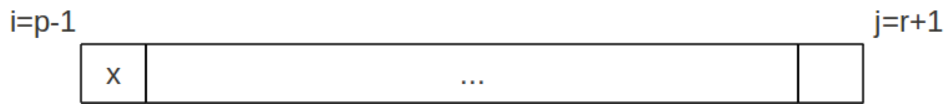
4. (40 points) **Hoare Partition:** Textbook Prob. 7-1 on p. 185-186, parts(a)-(d)

Ans:

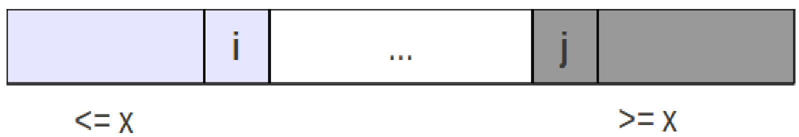
(a) Here, we only show the illustration of HOARE-PARTITION(A,1,12), the procedure has 3 iterations of the while loop and breaks the loop at i=10 and j=9, finally returns 9:



(b) Assume we apply HOARE-PARTITION on the array $A[p..r]$ where $r > p$ since A[p..r] contains at least two elements, and $x=A[p]$ is the pivot value.
The initial state of the array should be as follows:



According to the code, we know that j decreases from $j=r+1$, and this decreasing at least terminates at $j=p$ since $A[p]=x$. Similarly, the increasing of i terminates after increasing 1. After the initial loop, the starting state of the array in any loop can be of the following form:



Note that the light-shadowed part is the smaller part in where all elements are smaller or equal to x, correspondingly the heavy-shadowed part is the larger part in where all elements are greater or equal to x. This is satisfied because of line 5-10.
Therefore, for each iteration of the while loop, it guarantees that the decreasing of j (line 5-7) will stop at the rightmost element of the smaller part even if the all elements in the middle are greater than x. Similarly, i cannot exceed the leftmost element of the larger part. And we know that the size of each parts is at least 1 since $r > p$, so we can conclude that the decreasing j can not be smaller than p, and the increasing i can not be greater than r for $r > p$.

(c) We already prove that $j >= p$ in problem (b). Now we only need to prove $j<r$:
We find j will decrease if $A[j]>x$, there is two cases in the first loop:

1) If $A[j=r] > x$: j will decrease which means $j<r$.
2) If $A[j=r] <= x$: j will not decrease in the first iteration, then $j=r$. Due to line 8-10, i will stop at i=p since $A[p]=x>=x$, then $i=p$. Therefore, we know that $i=p<r=j$. According to line 11-13, the while loop will continue, which means in the second iteration j will decrease at least 1 anyway. So, j becomes smaller than r in the second iteration.

(d) As shown in (b), when the procedure terminates there are only two parts of the array: the smaller part ($A[k]$, $k=p,...,i$) and the larger part ($A[k]$, $k=j,...,r$), where $i>=j$ (due to line 11-13). We know that elements in larger part should be greater or equal to x and elements in smaller part should be smaller or equal to x, that is, elements of $A[p..j]$ (subset of the smaller part) are smaller or equal to elements of $A[j+1..r]$ (subset of the larger part).