McGill University COMP251: Assignment 2 Solution

Question 1 The partition procedure on a sorted array of length n always gives an empty subarray and an subarray of length $n-1$. So in this case the running time $T(n)$ of Quicksort satisfies:

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

So we have

$$\begin{aligned}
T(n) &= T(n-1) + \Theta(n) \\
&= T(n-2) + \Theta(n-1) + \Theta(n) \\
&= T(n-3) + \Theta(n-2) + \Theta(n-1) + \Theta(n) \\
&\dots \\
&= T(1) + \Theta(2) + \Theta(3) + \ldots + \Theta(n-1) + \Theta(n) \\
&= \Theta(n^2)
\end{aligned}$$

As a result, $T(n) = \Omega(n^2)$.

Question 2 (a) For an array A that is sorted in increasing order, the pairs (i, j) satisfying the given condition are

$$(1,2), (1,3), \ldots, (1,n),\ (2,3), (2,4), \ldots, (2,n),\ \ldots, (n-1, n)$$

The number of such pairs is

$$(n-1) + (n-2) + \ldots + 1 = \frac{(n-1)n}{2}$$

(b) The idea for a divide-and-conquer algorithm is as follows. Given a subarray $A[\ell \ \ldots \ r]$, we count the number of pairs (i, j) that satisfy the given condition (i.e., $i < j$ and $A[i] < A[j]$) by dividing $A[\ell \ \ldots \ r]$ into two halves $A[\ell \ \ldots \ m]$ and $A[(m+1) \ \ldots r]$, and summing up the following numbers:

1. the number of such pairs where $\ell \leq i < j \leq m$, and
2. the number of such pairs where $m+1 \leq i < j \leq r$, and
3. the number of such pairs where $i \leq m$ and $m+1 \leq j$

The first two quantities are computed recursively, and we are looking for a way to compute the last quantity in time $\mathcal{O}(r - \ell)$, so that the total running time will satisfy

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

and the Master Theorem gives $T(n) = \mathcal{O}(n \ln n)$.

(Note that a brute-force way of computing the last quantity above–by comparing all number in the first half with all number in the second half–requires $(\frac{r-\ell}{2})^2$ comparisons, so the running time would satisfy

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

and we would get $T(n) = \Theta(n^2)$, which is not good.)

The number of pairs in (3) above can be computed without performing $(\frac{r-\ell}{2})^2$ comparisons if the two halves $A[\ell \ \ldots \ m]$ and $A[(m+1) \ \ldots r]$ are already sorted in increasing order. For example, suppose that they are already sorted, and suppose that $A[\ell] < A[m+1]$, then we also have $A[\ell] < A[j]$ for all j in the second half. So we know that ℓ is present in exactly $(r-m)$ pairs

$$(\ell, m+1), (\ell, m+1), \ldots, (\ell, r)$$

The following procedure, Combine-and-count, is obtained by modifying the Combine procedure given in lecture. Combine-and-count(A, ℓ, m, r) assumes that $\ell \leq m < r$ and that the two subarray $A[\ell \ \ldots \ m]$ and $A[(m+1) \ \ldots \ r]$ are already sorted in increasing order. It will sort the subarray $A[\ell \ \ldots \ r]$ into increasing order and output the number of pairs (i, j) such that $i \leq m$ and $m+1 \leq j$ and $A[i] < A[j]$ (as in (3) above).

Combine-and-count(A, ℓ, m, r):

% first copy $A[\ell \ \ldots \ m]$ into a separate array $B[1 \ \ldots \ (m-\ell+1)]$

$j \leftarrow 1$

for i from ℓ to m do

    $B[j] \leftarrow A[i]$

    $j \leftarrow j+1$

end for

% now merge $B[1 \ \ldots \ (m-\ell+1)]$ and $A[(m+1) \ \ldots \ r]$ in to $A[\ell \ \ldots \ r]$
at the same time count the number of pairs (i, j) such that $\ell \leq i \leq m$, $m+1 \leq j \leq r$ and $A[i] < A[j]$

$i \leftarrow 1$ % current index in B

$j \leftarrow m+1$ % current index in $A[(m+1) \ \ldots \ r]$

$k \leftarrow \ell$ % current index in the final subarray

count $\leftarrow 0$ % the number of pairs to be output

while $i \leq m-l+1$ do % loop while there are still elements in B

    if $j > r$ % if we have gone through $A[(m+1) \ \ldots \ r]$:

        $A[k] \leftarrow B[i]$ % simply copy the remaining elements in B into A, no more pair to count

        $i \leftarrow i+1$, $k \leftarrow k+1$

    else do % compare $B[i]$ and $A[j]$

        if $A[j] < B[i]$ do % j does not contribute to the count

            $A[k] \leftarrow A[j]$ % copy $A[j]$ to its proper location

$j \leftarrow j+1$, $k \leftarrow k+1$
else do
$A[k] \leftarrow B[i]$
$count \leftarrow count + (r-j+1)$ % i contributes $(r-j+1)$ pairs
$i \leftarrow i+1$, $k \leftarrow k+1$
end if
end if
end while
output *count*

The algorithm that solve the given algorithm is Sort-and-count given below.
Sort-and-count(A, ℓ, r):

if $\ell = r$ return 0 % there is only one element
else if $\ell + 1 = r$ % there are two elements
if $A[\ell] < A[r]$ output 1 % there is only one pair
else
swap $A[\ell] \leftrightarrow A[r]$ and output 0
end if
end if
$m \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$ % mid-point
$c_1 \leftarrow$ Sort-and-count(A, ℓ, m)
$c_2 \leftarrow$ Sort-and-count$(A, m+1, r)$
$c \leftarrow$ Combine-and-count(A, ℓ, m, r)
return $c_1 + c_2 + c$

(c) The Combine-and-count procedure goes through all elements in the subarray $A[\ell \ \ldots \ r]$ at most once, so it runs in linear time. Therefore the running time $T(n)$ of Sort-and-count on input array of length n satisfies

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

(There are two recursive calls to subproblem of length $n/2$ each.) Apply the Master Theorem for $a = b = 2$, $d = 1$ we obtain

$$T(n) = \mathcal{O}(n \ln n)$$

Question 3 (a) An array A represents a ternary heap as follows: $A[1]$ is the root, its children are $A[2], A[3], A[4]$. In general, the children of $A[i]$ are $A[3i-1]$, $A[3i]$, $A[3i+1]$. The parent node of $A[i]$, for $i > 1$, is $A[\lfloor (i+1)/3 \rfloor]$. As for binary heap, there is a heap size heapsize(A) which is at most as large as length(A).

(b) A full ternary tree of height h has

$$1 + 3 + 3^2 + \ldots + 3^h = \frac{3^{h+1} - 1}{2}$$

Since the ternary heap is a near complete ternary tree with all level complete except possibly the last, the height h of a ternary heap with n elements satisfies

$$\frac{3^h - 1}{2} < n \leq \frac{3^{h+1} - 1}{2}$$

Thus

$$3^h < 2n + 1 \leq 3^{h+1}$$

So

$$h < \log_3(2n+1) \leq h + 1$$

Therefore $h = \lceil \log_3(2n+1) \rceil - 1$.

(c) The Heapify3 procedure is a modification of Max-Heapify given in lecture. It assumes that the subtrees at $A[3i-1]$, $A[3i]$, and $A[3i+1]$ are already ternary heaps, but $A[i]$ might be smaller than one of its children and thus violating the max-heap property. It will float $A[i]$ down the subtree of its largest children.

Heapify3(A,i):

% first get the index of the largest element among $A[i]$, $A[3i-1]$, $A[3i]$, $A[3i+1]$
if $3i - 1 \leq heapsize(A)$ and $A[3i-1] > A[i]$ do
    $largest \leftarrow 3i - 1$
else $largest \leftarrow i$
if $3i \leq heapsize(A)$ and $A[3i] > A[largest]$ do
    $largest \leftarrow 3i$
end if
if $3i + 1 \leq heapsize(A)$ and $A[3i+1] > A[largest]$ do
    $largest \leftarrow 3i + 1$
end if
% now $A[largest]$ is the largest element among $A[i]$, $A[3i-1]$, $A[3i]$, $A[3i+1]$
if $largest \neq i$ do
    swap $A[i] \leftrightarrow A[largest]$

14. Heapify3(A,largest)

15. end if

(d) Heapsort3 works in the same way as the algorithm Heapsort given in class. It uses the following Build-max-heap3 procedure, which constructs a ternary heap from the given array A:

Build-max-heap3(A)

1. $heapsize(A) \leftarrow length(A)$
2. for i from $\lfloor length(A)/2 \rfloor$ down to 1 do
3. Heapify3(A,i) % turn the ternary subtree at $A[i]$ into a ternary heap
4. end for

Heapsort3(A)

1. Build-max-heap3(A)
2. for i from length(A) down to 2 do
3. swap $A[1] \leftrightarrow A[i]$
4. $heapsize(A) \leftarrow heapsize(A) - 1$
5. Heapify3(A,1)
6. end for

(e) The running time of Heapify3 on a subtree of height h is $\mathcal{O}(h)$, because we perform at most a constant number of operation on each level of the tree. Therefore the running time of Build-max-heap3 is at most

$$\frac{n}{3}\mathcal{O}(\ln n) = \mathcal{O}(n \ln n)$$

(because from (b) the height of the ternary heap is $\Theta(lnn)$).

The for-loop in Heapsort3 has $\frac{n}{3}$ iterations, each iteration takes time at most $\mathcal{O}(lnn)$. Therefore the total time of Heapsort3 is

$$\mathcal{O}(n \ln n) + \mathcal{O}(n \ln n) = \mathcal{O}(n \ln n)$$

Question 4 The idea is to use the "counting array" C from the counting sort algorithm given in lecture. We want the array C (with indices from 0 to k) so that $C[x]$ is the number of elements $A[i]$ such that $A[i] \leq x$.

The preprocessing procedure will compute such a C. Then to answer the query of how many $A[i]$ such that $a \leq A[i] \leq b$ there are, simply give

$$C[b] - C[a-1] \qquad \text{if } a \geq 1$$

(if $a = 0$ then take $C[b]$).

The pseudo-code for the preprocessing procedure is as follows:

% the following for-loop initializes counting array C:

for x from 0 to k do

    $C[x] \leftarrow 0$

end for

% the next for-loop makes each $C[x]$ be the number of i such that $A[i] = x$:

for i from 1 to length(A) do

    $C[A[i]] \leftarrow C[A[i]] + 1$

end for

% sum up: each $C[x]$ will be the number of i such that $A[i] \leq x$:

for x from 1 to k do

    $C[x] \leftarrow C[x] + C[x-1]$

end for