COMP3711 Assignment 2

TANUWIJAYA, Randy Stefan
(20582731)
rstanuwijaya@connect.ust.hk

Department of Physics - HKUST
Department of Computer Science and Engineering - HKUST

April 11, 2021

Problem 1: Recurrence Relations

Give asymptotic upper bounds for $T(n)$ satisfying the following recurrences. Make your bounds as tight as possible. For example, if $T(n) = \Theta(n^2)$ then $T(n) = O(n^2)$ is a tight upper bound but $T(n) = O(n^2 \log n)$ is not. Your upper bound should be written in the form $T(n) = O(n^\alpha (\log n)^\beta)$ where α, β are appropriate constants.

(a) $T(1) = 1$; $T(n) = 8T(n/2) + n^2\sqrt{n}$ for $n > 1$
Let $c = \log_2 8 = 3$ and $f(n) = n^2\sqrt{n} = O(n^3)$. Since $f(n) = O(n^{c-\epsilon})$ for some $\epsilon > 0$, then by *Master Theorem*, $\boxed{T(n) = \Theta(n^c) = \Theta(n^3) = O(n^3)}$

(b) $T(1) = 1$; $T(n) = 5T(n/2) + n^3 \log_2 n$ for $n > 1$
Let $c = \log_2 5 < 3$ and $f(n) = n^3 \log_2 n = \Omega(n^3)$. Since $f(n) = \Omega(n^{c+\epsilon})$ for some $\epsilon > 0$ and the smoothness condition is satisfied, then by *Master Theorem*, $\boxed{T(n) = \Theta(f(n)) = \Theta(n^3 \log n) = O(n^3 \log n)}$

(c) $T(1) = 1$; $T(n) = 3T(n/4) + \sum_{i=1}^{n} i$ for $n > 1$
Let $c = \log_4 3 < 1$ and $f(n) = (n^2 + n)/2 = \Omega(n)$. Since $f(n) = \Omega(n^{c+\epsilon})$ for some $\epsilon > 0$ and the smoothness condition is satisfied, then by *Master Theorem*, $\boxed{T(n) = \Theta(f(n)) = \Theta(n^2) = O(n^2)}$

(d) $T(1) = 1$; $T(n) = 5T(n/4) + 1$ for $n > 1$
Let $c = \log_4 5 > 1$ and $f(n) = 1 = O(n)$. Since $f(n) = O(n^{c-\epsilon})$ for some $\epsilon > 0$, then by *Master Theorem*, $\boxed{T(n) = \Theta(n^c) = \Theta(n^{\log_4 5}) = O(n^{\log_4 5})}$

(e) $T(1) = 1$; $T(n) = 2T(n/7) + 2^{\log_7 n}$ for $n > 1$
Let $c = \log_7 2 > 1$ and $f(n) = 2^{\log_7 n} = n^{\log_7 2}$. Since $f(n) = \Theta(n^c)$, then by *Master Theorem*, $\boxed{T(n) = \Theta(n^c \log n) = \Theta(n^{\log_7 2} \log n) = O(n^{\log_7 2} \log n)}$

(f) $T(1) = 1$; $T(n) = 49T(n/7) + \log_3(n!)$ for $n > 1$
Let $c = \log_7 49 = 2$ and $f(n) = \log_3(n!) = \Theta(n \log n) = O(n^2)$. Since $f(n) = O(n^{c-\epsilon})$ for some $\epsilon > 0$, then by *Master Theorem*, $\boxed{T(n) = \Theta(n^c) = \Theta(n^2) = O(n^2)}$

Problem 2: Monotone Matrices

An $N \times N$ matrix is *monotone if each of its rows and columns are sorted in nondecreasing order. Given monotone matrix A and value a, the* membership problem is to test whether $a \in A$ or $a \notin A$.

More specifically $MEM(a; m_1; n_1; m_2; n_2)$ will return `TRUE` if there exist i, jsuch that $a = A[i, j]$ where $m_1 \leq i \leq m_2$ and $n_1 \leq j \leq n_2$ and `FALSE` otherwise.

(a) Prove that this algorithm (i) terminates and (ii) correctly solves the membership problem for monotone matrices.

Base Cases: $\Delta m = \{0, 1\} \cap \Delta n = \{0, 1\}$

(i) $\mathbf{a} \in \mathbf{A}[\mathbf{m_1}..\mathbf{m_2}, \mathbf{n_1}..\mathbf{n_2}]$: the algorithm correctly returns `TRUE` and terminates

(ii) $\mathbf{a} \notin \mathbf{A}[\mathbf{m_1}..\mathbf{m_2}, \mathbf{n_1}..\mathbf{n_2}]$: the algorithm correctly returns `FALSE` and terminates

General Cases: $\Delta m > 1 \cup \Delta n > 1$, assume $I(\Delta m', \Delta n')$ is true for all $1 \leq \Delta m' < \Delta m, 1 \leq \Delta n' < \Delta n$. Let $u = \lfloor (m_1 + m_2)/2 \rfloor$ and $v = \lfloor (n_1 + n_2)/2 \rfloor$ as the midpoint of the array for the first and second axes in the current recursive call. First assume if a is in A:

(i) Check if a is in the lower-left submatrix by calling $Mem(a, u, m_2, n_1, v)$. By the induction hypothesis, the checking returns the correct answer. The algorithm correctly returns `TRUE` and terminates if a is in the lower-left.

(ii) Check if a is in the upper-right submatrix by calling $Mem(a, m_1, u, v, n_2)$. By the induction hypothesis, the checking returns the correct answer. The algorithm correctly returns `TRUE` and terminates if a is in the upper-right.

(iii) $\mathbf{a} \leq \mathbf{A}[\mathbf{u}, \mathbf{v}]$: and a is not in the lower-left and upper-right submatrices, then a must be in the upper-left submatrix. Algorithm will call $Mem(a, m_1, n_1, u, v)$. By the induction hypothesis, the checking returns the correct answer. The algorithm correctly returns `TRUE` and terminates.

(iv) $\mathbf{a} > \mathbf{A}[\mathbf{u}, \mathbf{v}]$: and a is not in the lower-left and upper-right submatrices, then a must be in the lower-right submatrix. Algorithm will call $Mem(a, u; b, m_2, n_2)$. By the induction hypothesis, the checking returns the correct answer. The algorithm correctly returns `TRUE` and terminates.

Otherwise, if a is not in A, the algorithm will correctly returns `FALSE` and terminates.

Thus, $MEM(a, m_1, m_2, n_1, n_2)$ always terminates and returns the correct answer. The induction hypothesis $I(\Delta m, \Delta n)$ is true for all $\Delta m, \Delta n \geq 0$.

(b) Show that without monotonicity, the algorithm might not be correct. To do this, give an example of a 5×5 non-monotone matrix A and value a such that $a \in A$ but $MEM(a, 1, 1, 5, 5)$ would return false.

For example, take A to be:

$$A = \begin{pmatrix} 25 & 24 & 23 & 22 & 21 \\ 20 & 19 & 18 & 17 & 16 \\ 15 & 14 & 13 & 12 & 11 \\ 10 & 9 & 8 & 7 & 6 \\ 5 & 4 & 3 & 2 & 1 \end{pmatrix}$$

and $a = 25$. The algorithm will first check the lower-left and upper-right submatrices, and then check the lower-right submatrices (since $25 > 13$), returns `FALSE` and terminates.

(c) Analyze, from scratch (no use of other theorems allowed) the running time of the algorithm when run on an $N \times N$ monotone matrix, i.e., when $MEM(a, 1, 1, N, N)$ is called.

Let T(N) be the number of comparison needed for an array with size of $N \times N$, where $N = 2^k + 1$ The recurrence relation is given by:

$$T(1) = T(2) = 1, \quad T(N) \leq 3T\left(\left\lceil \frac{N}{2} \right\rceil\right) + 1$$

The recurrence relation can be explained as following. The base case is when the array size is either $N \in \{1, 2\}$, in which the algorithm runs a constant number of comparison. In the general case, the worst scenario of the algorithm is when it needs to check three out of four submatrices - where a is not in the lower-left or upper right submatrix. In this case, the algorithm checks lower-left and upper-left submatrices, in addition to either upper-left ($a \leq A[u, v]$) or lower-right ($a > A[u, v]$) submatrix, with a single comparison. The asymptomatic upper bound of the algorithm running time is given by:

$$
\begin{aligned}
T(n) &= 3T(\lceil n/2 \rceil) + 1 \\
&= 3(3T(\lceil n/2^2 \rceil) + 1) + 1 \\
&= 3^2 T(\lceil n/2^2 \rceil) + (1 + 3) \\
&= 3^2(3T(\lceil n/2^3 \rceil) + 1) + (1 + 3) \\
&= 3^3 T(\lceil n/2^3 \rceil) + (1 + 3 + 3^2) \\
&= 3^k T(\lceil n/2^k \rceil) + \sum_{i=0}^{k-1} \left(3^i\right) \\
&\text{Assume } n = 2^k + 1, \text{ then } T(\lceil n/2^k \rceil) = T(2) = 1. \\
&= 3^k + \sum_{i=0}^{k-1} \left(3^i\right) \\
&= 3^k + \left(\frac{3^k - 1}{3 - 1}\right) \\
&\leq 3^k + 3^k \times \frac{1}{2} = \frac{3}{2} 3^k = \frac{3}{2} 3^{\log_2 n} = \frac{3}{2} n^{\log_2 3} \quad \text{for } n > 1
\end{aligned}
$$

$$
\boxed{T(n) = O(n^{\log_2 3}) \approx O(n^{1.585})}
$$

Problem 3: Selection

Recall the Selection problem. *Given an array $A[1 \dots n]$ of n unsorted values and an integer $k \leq n$, return the (index of the) k^{th} smallest item in A*

In class you learned a simple $O(n)$ randomized algorithm for solving Selection. A (more complicated) $O(n)$ time worst-case Selection algorithm also exists.

For this problem, assume that you are given a black-box procedure for implementing this worst-case $O(n)$ selection algorithm. That is, given an array $A[p \dots r]$, $p < r$ and k, $BB(A, p, r, k)$ finds and reports the index of the k^{th} smallest item in $A[p, \dots r]$ in $O(r - p + 1)$ time.

(a) Show how, using Selection as a subroutine, Quicksort can be modified to run in $O(nlogn)$ worst-case time.

(i) Provide documented pseudocode for your $O(n \log n)$ worst-case time Quicksort. Your new Quicksort pseudocode can call the black-box $BB(A, p, r, k)$ procedure as often as it wants.

Algorithm 1: Qsort(A, p, r)

```
if p ≥ r then                                   // base case and invalid parameters
    return
else
    q ← ⌊(p+r)/2⌋ ;                             // get the index of midpoint
    swap A[q] with A[BB(A, 1, n, q)]
    Partition(A, p, r) ;                        // Partition around the midpoint
    Qsort(A, p, q-1) ;                          // Recursively sort the left subarray
    Qsort(A, q+1, r) ;                          // Recursively sort the right subarray
end
```

(ii) Explain why the psuedocode runs in $O(n \log n)$ worst-case time.
It runs on $O(n \log n)$ time since we can always use BB to find the pivot point in the middle of the array, thus the submatrices are always symmetric. The reccurence relation of the worst-case running time of this algorithm would be similar to Merge Sort algorithm, which is $O(n \log n)$.

(b) Find a good algorithm for reporting the k middle items in the array in sorted order. The middle k items are item $\lfloor \frac{n-k}{2} \rfloor + 1$ to item $\lfloor \frac{n+k}{2} \rfloor$. Use the black-box selection procedure to design an algorithm that is never worse (and sometimes better) than $\Theta(n \log n)$. In particular, for all $k \leq \frac{n}{\log_2 n}$, your algorithm should run in $\Theta(n)$ time. Algorithms that do not satisfy this last condition will be considered wrong.

(i) Provide documented pseudocode for your algorithm.

Algorithm 2: k-select(A, n, k)

```
lb ← ⌊(n-k)/2⌋ + 1, ub ← ⌊(n+k)/2⌋ ;        // calculate the lower and upper bounds
id_lb ← BB(A, 1, n, lb) ;                   // get the index of the lower bound-th item
swap A[n] with A[id_lb]
Partition(A, 1, n)
id_ub ← BB(A, 1, n, ub) ;                   // get the index of the upper bound-th item
swap A[n] with A[id_ub]
Partition(A, 1, n)
// the item between lower and upper bounds are the desired item, but
   unsorted - use MergeSort to sort the items
MergeSort(A, lb, ub)
return A[lb ... ub]
```

(ii) State the running time of your new procedure as tightly as possible in terms of both n and k. Explain why the pseudocode runs in that time.

The running time of the algorithm is $O(n + k \log k)$. Finding the lower-bound lb and the upper bound ub takes $O(n)$ time. Calling partition function takes $O(n)$ time, and calling $MergeSort$ to get the sorted array with size of k takes $O(k \log k)$ time. Therefore, the total running time is $O(n + k \log k)$

(c) Show how to use a solution to the Tierce problem to solve the Selection problem.

The Tierce problem is to find the index of the $\frac{1}{3}$'rd item in the array, i.e., the index that would be returned by $BB(A, p, r, \lfloor \frac{r-p+1}{3} \rfloor)$.

Suppose that you lost the code for $BB(A, p, r, k)$ but were given a new black box for solving $Tierce(A, p, r)$ in $O(r - p + 1)$ worst case time. Show how, using $Tierce(A, p, r)$ as a subroutine, you can write a new linear time Selection algorithm.

(i) Provide documented pseudocode for your algorithm.

Algorithm 3: SelectTierce(A, p, r, k)

```
tb ← ⌊(r−p+1)/3⌋ ;                                  // get the tierce bound
id_tb ← Tierce(A, p, r) ;                  // get the index of the 1/3'th item
if tb = k then                                                  // base case
    return tb
else
    swap A[r] with A[id_tb]
    j ← Partition(A, p, r)
    // the array now partitioned around the pivot (tb-th item)
    if j < k then                                              // recursive step
        return SelectTierce(A, p, q, k)
    else
        return SelectTierce(A, q+1, r, k - tb)
    end
end
```

(ii) Explain why your algorithm is correct.

The base case is for $n = 1$ and the algorithm correctly returns the only item in the array. The inductive hypothesis $I(n')$ as the algorithm correctly returns the k'th item for all $n' < n$.

Let the *Tierce Bound* tb as $tb = \lfloor \frac{r-p+1}{3} \rfloor$ and $A[id_{tb}]$ as the $\frac{1}{3}$'th item on the array. For $n = 2$

If $k = tb$, the algorithm correctly returns the index of the k'th item. Otherwise, by calling the partition with pivot around the $A[id_{tb}]$, all the item less than or equal to the $A[id_{tb}]$ will be positioned on the left, and all the item greater than $A[id_{tb}]$ will be positioned on the right. If $k < tb$, then the elements to be selected must be positioned on the left of the pivot, and the algorithm will correctly return the index by the induction hypothesis. Conversely, if $k > tb$, the algorithm also returns the correct answer by the induction hypothesis.

Thus, the algorithm is correct and $I(n)$ holds for all $n \geq 1$.

(iii) Prove that it runs in $O(r - p + 1)$ worst case time.

Note that the worst case is when the the algorithm is used to search the maximum (n'th) item in an array. The recurrence relation is given by:

$$T(1) = 1, \quad T(n) \leq T(2n/3) + O(n) \quad \Rightarrow \quad \boxed{T(n) = O(n)}$$

The last equality holds by applying Master's Theorem for inequalities for $c = 0$ and $f(n) = \Omega(n^{c+\epsilon})$ for some $\epsilon > 0$ (satisfying smoothness condition).

Problem 4: Sorting

(a) (Radix Sort)
You are given a set of 10 decimal integers in the range of 1 to 65535:

$$A = [23825, 29914, 14359, 39515, 62095, 49896, 63678, 37642, 40263, 52107]$$

(i) Please conduct Radix Sort on A using Base 10. Illustrate your result after each step following the worked example on Page 35 in the 08 Linearsort lecture slides.

index	A	$A^{(1)}$	$A^{(2)}$	$A^{(3)}$	$A^{(4)}$	$A^{(5)}$
1	23825	37642	52107	62095	40263	14359
2	29914	40263	29914	52107	62095	23825
3	14359	29914	39515	40263	52107	29914
4	39515	23825	23825	14359	63678	37642
5	62095	39515	37642	39515	23825	39515
6	49896	62095	14359	37642	14359	40263
7	63678	49896	40263	63678	37642	49896
8	37642	52107	63678	23825	39515	52107
9	40263	63678	62095	49896	49896	62095
10	52107	14359	49896	29914	29914	63678

Table 1: Radix Sort of A_{10}

(ii) Now convert these decimal integers to hexadecimal and conduct Radix sort again, this time using Base 16. Illustrate your result after each step following the worked example on Page 35 in the 08 Linearsort lecture slides.

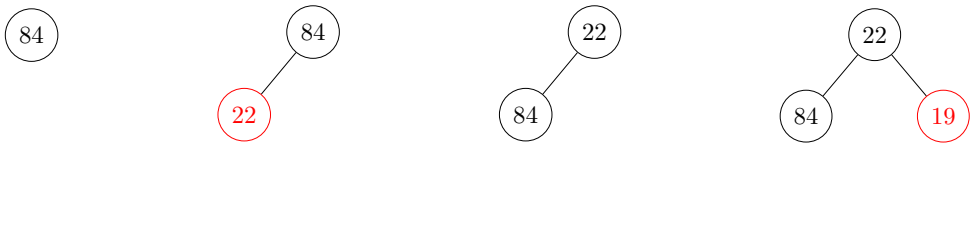
$$A_{16} = [\texttt{5D11}, \texttt{74DA}, \texttt{3817}, \texttt{9A5B}, \texttt{F28F}, \texttt{C2E8}, \texttt{F8BE}, \texttt{930A}, \texttt{9D47}, \texttt{CB8B}]$$

index	A	$A^{(1)}$	$A^{(2)}$	$A^{(3)}$	$A^{(4)}$
1	5D11	5D11	930A	F28F	3817
2	74DA	3817	5D11	C2E8	5D11
3	3817	9D47	3817	930A	74DA
4	9A5B	C2E8	9D47	74DA	930A
5	F28F	74DA	9A5B	3817	9A5B
6	C2E8	930A	CB8B	F8BE	9D47
7	F8BE	9A5B	F28F	9A5B	C2E8
8	930A	CB8B	F8BE	CB8B	CB8B
9	9D47	F8BE	74DA	5D11	F28F
10	CB8B	F28F	C2E8	9D47	F8BE

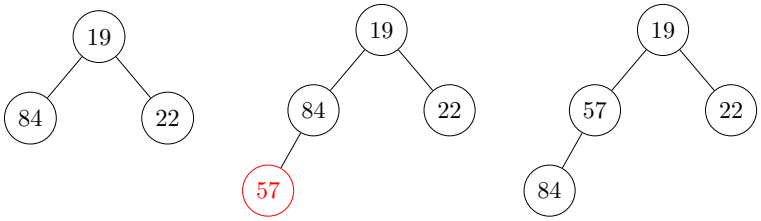
Table 2: Radix Sort of A_{16}

(b) (Heapsort) Apply Heapsort on an initial input array $A = [84, 22, 19, 57]$. Illustrate the heap after each step following the worked example in 07a_Example_Heapsort slides.

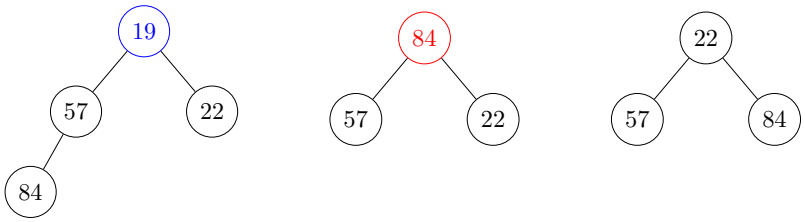
The output of the heapsort algorithm A' is given by: $A' = [19, 22, 57, 84]$. Figure 1 illustrates the insertion and extraction methods call on the temporary heap.



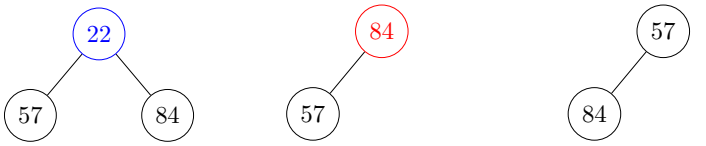
(a) Insert 84 (b) Insert 22 (c) Insert 22 (d) Insert 19



(e) Insert 19 (f) Insert 57 (g) Insert 57



(h) Extract-Min 19 (i) Extract-Min 19 (j) Extract-Min 19



(k) Extract-Min 22 (l) Extract-Min 22 (m) Extract-Min 22

(n) Extract-Min 57 (o) Extract-Min 57 (p) Extract-Min 84

Figure 1: Heapsort Illustration
$A = [84, 22, 19, 57],\ A' = [19, 22, 57, 84]$

Problem 5: Decision Trees

Recall that a comparison-based sorting algorithm can be represented in the (binary) decision tree model.

Following the worked example of Tutorial SS13, expand the subtree T2 of the decision tree for sorting a list of four items $a_1, a_2.a_3.a_4$ using Quicksort with the last item as pivot.

From the parents of T_2 node in the decision tree, we know that there are two sets of sorted lists in the array, which are $\{a_1, a_4\}$ and $\{a_4, a_2\}$. These two expressions also imply $\{a_1, a_4, a_2\}$. Now, the decision tree T_2 just need to merge the sorted list $\{a_1, a_4, a_2\}$ with a_3. Figure 2 illustrates the decision tree of T_2

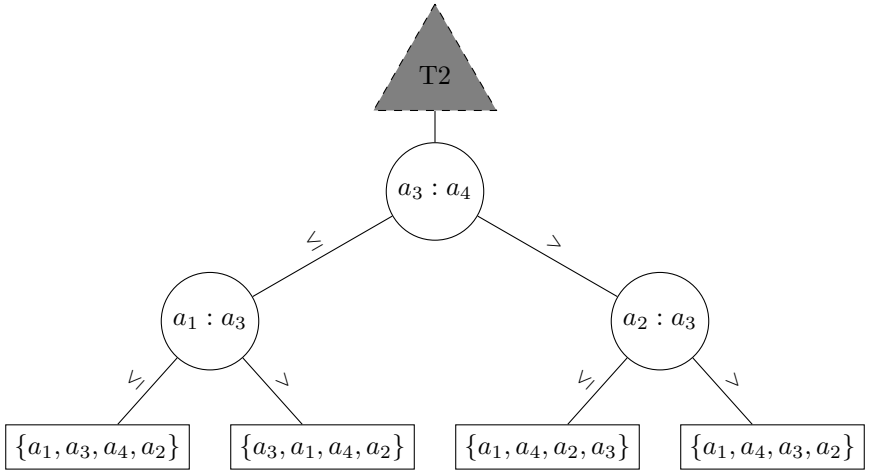


Figure 2: Decision Tree of T_2

Problem 6: Randomization

Let $A[1..n]$ be an array of n distinct numbers. In class we said that if $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an inversion of A. In Tutorial R5 we used indicator random variables to show that, if the elements of A form a uniform random permutation of $\langle 1, 2, \ldots, n\rangle$, then the expected number of inversions in A is $\frac{n(n-1)}{4}$

For this problem let $n \geq 2$ be an even number and assume that $A[1 \ldots n]$ is initialized with $A[i] = i$.

(a) Now run the following random procedure on A:

For every i, $1 \leq i \leq n/2$, swap the values in $A[i]$ and $A[\frac{n}{2} + i]$ with probability $\frac{1}{2}$ (and leave them unswapped with probability $\frac{1}{2}$.)

Answer the following question: what is the expected number of inversions in A after running this swapping procedure? Show how you mathematically derived this answer.

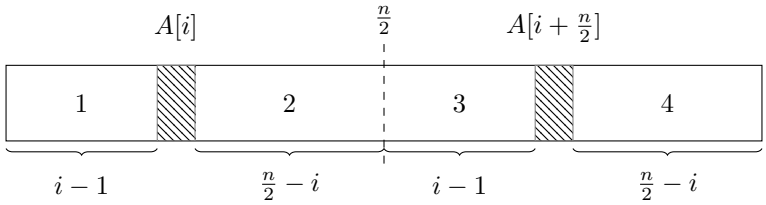


Figure 3: Partitions of the array

Consider Figure 3, and note that there is equal probability of $A[i]$ being switched with $A[i + \frac{n}{2}]$ or not. For each case, there is equal probability of each pair of partitions ($(1, 3)$ and $(2, 4)$) will be switched or not. For simplicity, define $j = i + \frac{n}{2}$. Let $I(k, p)$ denote the number of inversions of $A[k]$ with the partition p, and $J(i, j) = 1$ if i and j is inverted, 0 otherwise.

- $A[i]$ **is not switched**:
  - **Partitions are not switched**:

$$I(i,1) = 0 \quad I(i,2) = 0 \quad I(i,3) = 0 \quad I(i,4) = 0 \quad J(i,j) = 0$$
$$I(j,1) = 0 \quad I(j,2) = 0 \quad I(j,3) = 0 \quad I(j,4) = 0 \quad J(j,i) = 0$$

  - **Partitions are switched**:

$$I(i,1) = i-1 \quad I(i,2) = 0 \quad I(i,3) = i-1 \quad I(i,4) = 0 \quad J(i,j) = 0$$
$$I(j,1) = 0 \quad I(j,2) = \frac{n}{2} - i \quad I(j,3) = 0 \quad I(j,4) = \frac{n}{2} - i \quad J(j,i) = 0$$

- $A[i]$ **is switched**:
  - **Partitions are not switched**:

$$I(i,1) = 0 \quad I(i,2) = \frac{n}{2} - i \quad I(i,3) = i-1 \quad I(i,4) = 0 \quad J(i,j) = 1$$
$$I(j,1) = 0 \quad I(j,2) = \frac{n}{2} - i \quad I(j,3) = i-1 \quad I(j,4) = 0 \quad J(j,i) = 1$$

  - **Partitions are switched**:

$$I(i,1) = 0 \quad I(i,2) = 0 \quad I(i,3) = i-1 \quad I(i,4) = \frac{n}{2} - i \quad J(i,j) = 1$$
$$I(j,1) = i-1 \quad I(j,2) = \frac{n}{2} - i \quad I(j,3) = 0 \quad I(j,4) = 0 \quad J(j,i) = 1$$

All cases have equal probability of $\frac{1}{4}$. Let $S = 3n - 2$ as the sum of all the inversions calculated above. Then, by the linearity of expectation, the expected inversion number is given by:

$$E[\text{Inv}] = \frac{S}{4} \times \frac{n}{2} \times \frac{1}{2} = \boxed{\frac{n}{4}\left(\frac{3}{4}n - \frac{1}{2}\right)}$$

The factor $1/4$ comes from the probability of each cases, $\frac{n}{2}$ comes from the loop condition, and $\frac{1}{2}$ comes from the fact that the inversion number obtained is double counted.

(b) Now assume again that $A[1 \dots n]$ is initialized with $A[i] = i$ and run the following different random procedure on A:

For every i, $1 \le i \le n/2$; swap the values in $A[i]$ and $A[n - i + 1]$ with probability $\frac{1}{2}$ (and leave them unswapped with probability $\frac{1}{2}$.)

Answer the following question: what is the expected number of inversions in A after running this swapping procedure? Show how you mathematically derived this answer.

Similar to the previous question, we can separate it into four equiprobable cases:

- $A[i]$ **is not switched**:
  - **Partitions are not switched**:

$$\begin{array}{lllll} I(i,1) = 0 & I(i,2) = 0 & I(i,3) = 0 & I(i,4) = 0 & J(i,j) = 0 \\ I(j,1) = 0 & I(j,2) = 0 & I(j,3) = 0 & I(j,4) = 0 & J(j,1) = 0 \end{array}$$

  - **Partitions are switched**:

$$\begin{array}{lllll} I(i,1) = i-1 & I(i,2) = 0 & I(i,3) = 0 & I(i,4) = i-1 & J(i,j) = 0 \\ I(j,1) = i-1 & I(j,2) = 0 & I(j,3) = 0 & I(j,4) = i-1 & J(j,1) = 0 \end{array}$$

- $A[i]$ **is switched**:
  - **Partitions are not switched**:

$$\begin{array}{lllll} I(i,1) = 0 & I(i,2) = \frac{n}{2} - i & I(i,3) = \frac{n}{2} - i & I(i,4) = 0 & J(i,j) = 1 \\ I(j,1) = 0 & I(j,2) = \frac{n}{2} - i & I(j,3) = \frac{n}{2} - i & I(j,4) = 0 & J(j,i) = 1 \end{array}$$

  - **Partitions are switched**:

$$\begin{array}{lllll} I(i,1) = i-1 & I(i,2) = \frac{n}{2} - i & I(i,3) = \frac{n}{2} - i & I(i,4) = i-1 & J(i,j) = 1 \\ I(j,1) = i-1 & I(j,2) = \frac{n}{2} - i & I(j,3) = \frac{n}{2} - i & I(j,4) = i-1 & J(j,i) = 1 \end{array}$$

All cases have equal probability of $\frac{1}{4}$. The sum of inversions obtained from the calculation above is $S = 4n - 4$. By the linearity of expectation, the expected inversion number is given by:

$$E[\text{Inv}] = \frac{S}{4} \times \frac{n}{2} \times \frac{1}{2} = \boxed{\frac{n}{4}(n-1)}$$

Where the factor $1/4$ comes from the probability of each cases, $\frac{n}{2}$ comes from the loop condition, and $\frac{1}{2}$ comes from the fact that the inversion number obtained is double counted.