1 What is the largest-possible number of inversions a 6-element array can have?

1. 15
2. 21
3. 36
4. 64

(1) is correct. The inversions are the largest when the array is arranged in descending order such that if $x_i < x_j \implies A[x_i] > A[x_j]$. We construct an example [6, 5, 4, 3, 2, 1]. We see that the number of inversions are $5+4+3+2+1 = \sum_{i=1}^{5} i = 15$. In general the largest-possible number of inversions for an array of length n is $\sum_{i=1}^{n-1} i = \frac{(n-1)\cdot n}{2}$.

2 Suppose the input array A has no split inversions. What is the relationship between the sorted subarrays C and D?

1. C has the smallest element of A, D the second-smallest, C the third smallest, and so on.
2. All elements of C are less than all elements of D.
3. All elements of C are greater than all elements of D.
4. There is not enough information to answer this question.

(2) Since there is no split inversion there is no i in the first half and j in the second half of the array such that $i \leq \frac{n}{2} < j$ and $A[i] > A[j]$. This implies all elements in the left half array denoted C must be less than all elements of right half denoted D.

3 What is the asymptotic running time of the straightforward algorithm for matrix multiplication, as a function of the matrix dimension n? Assume that the addition or multiplication of two matrix entries is a constant-time operation.

1. $\Theta(n \cdot log_2 n)$
2. $\Theta(n^2)$
3. $\Theta(n^3)$
4. $\Theta(n^4)$

(3) is correct. the work done is $\Theta(n \cdot n \cdot n) = \Theta(n^3)$.

ClosestPair (Preliminary Version)

Input: two copies P_x and P_y of $n \geq 2$ points in the plane, sorted by x- and y-coordinate, respectively.
Output: the pair p_i, p_j of distinct points with smallest Euclidean distance between them.

```
// base case of <= 3 points omitted
1 L_x := first half of P_x, sorted by x-coordinate
2 L_y := first half of P_x, sorted by y-coordinate
3 R_x := second half of P_x, sorted by x-coordinate
4 R_y := second half of P_x, sorted by y-coordinate
5 (l_1, l_2) := ClosestPair(L_x, L_y)     // best left pair
6 (r_1, r_2) := ClosestPair(R_x, R_y)  // best right pair
7 (s_1, s_2) := ClosestSplitPair(P_x, P_y)  // best split
  pair
8 return best of (l_1, l_2), (r_1, r_2), (s_1, s_2)
```

Figure 0.1: ClosestPair

4

Suppose that we correctly implement the $ClosetSplitPair$ subroutine in $O(n)$ time. What will be the overall running time of $ClosetPair$ algorithm?

1. $O(n)$
2. $O(n \cdot log_2 n)$
3. $O(n \cdot (log_2 n)^2)$
4. $O(n^2)$

(3) is the correct answer. The recurrence is of the form $T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$. This recurrence solves to $\Theta(n \cdot log_2 n)$.

5 Consider the following pseduocode for calculating a^b where a and b are positive integers.

Assume for this problem that each multiplication and division can be performed in constant time. What is the asymptotic running time of this algorithm as a function of b?

1. $\Theta(log_2 b)$
2. $\Theta(\sqrt{b})$
3. $\Theta(b)$
4. $\Theta(b \cdot log_2 b)$

```
FastPower

Input: positive integers a and b.
Output: a^b.

---

if b = 1 then
    return a
else
    c := b · b
    ans := FastPower(c, ⌊b/2⌋)
if b is odd then
    return a · ans
else
    return ans
```

Figure 0.2: FastPower

(3) is correct. The recurrence is of the form $T(n) = T(\frac{n}{2}) + O(1)$. We use the *Akra-Bazzi* formula $T(n) = \Theta(n^p \cdot (1 + \int_1^n \frac{g(u)}{u^{p+1}} \cdot du))$ where p satisfies $(\frac{1}{2})^p = 1$. In this case $p = 0$ and $g(u) = 1$. Substituting we obtain

$$\begin{aligned} \Theta(n^0 \cdot (1 + \int_1^n \frac{1}{u} du)) &= \Theta(1 + lnn) \\ &= \Theta(lnn) \\ &= \Theta(log_2 n) \end{aligned}$$

6 You are given a *unimodal* array of n distinct elements, meaning that its entries are in increasing order until its maximum element, after which its elements are in decreasing order. Give an algorithm to compute the maximum element of a unimodal array that runs in $O(log_2 n)$ time.

An brute-force algorithm that is $\Theta(n)$ is to scan through the array and check if every entry is the maximum element. We can develop an $\Theta(log_2 n)$ algorithm by utilizing the divide and conquer principle. we label the indices of the arrays as $1, 2, 3, \ldots \frac{n}{2} - 1, \frac{n}{2}, \frac{n}{2} + 1, \ldots n$. We first look at the $\frac{n}{2}$ element if it is both greater than $\frac{n}{2} - 1$ and the $\frac{n}{2} + 1$ elements then we have found the maximum

element. If the $\frac{n}{2}-1$ element is bigger than the $\frac{n}{2}$ element we recurse on the left half of the array so now the possible candiates are the entries $1, 2, 3, \ldots, \frac{n}{2}-1$. A similar argument follows if the $\frac{n}{2}+1$ element is bigger than the $\frac{n}{2}$ element. The recurrence relation is of the form of $T(n) = T(\frac{n}{2}) + O(1)$. This recurrence solves to $T(n) = \Theta(log_2 n)$.

7 You are given a sorted array A of n distinct integers which can be positive, negative, or zero. You want to decide whether or not there is an index i such that $A[i] = i$. Design the fastest algorithm you can for solving this problem.

A brute-force algorithm that is $\Theta(n)$ is to check every entry i in the array to see if $A[i] = i$. We can develop an $\Theta(log_2 n)$ algorithm by using the divide and conquer principle. We label the indices $0, 1, 2 \ldots, \frac{n}{2}-1, \frac{n}{2}, \frac{n}{2}+1, \ldots, n$. We first check if $A[\frac{n}{2}] = \frac{n}{2}$. If it is we're done. If not two cases exist. If $A[\frac{n}{2}] < \frac{n}{2}$ then it is the case that no element in the left subarray will have an element i such that $A[i] = i$. We recurse on the new subarray $\frac{n}{2}+1, \ldots, n$. A similar argument is developed for when $A[\frac{n}{2}] > \frac{n}{2}$. The recurrence is of the form $T(n) = T(\frac{n}{2}) + O(1)$. This recurrence solves to $T(n) = \Theta(log_2 n)$.

8 You are given an $n-by-n$ grid of distinct nmbers. A number is a *local minimum* if its is smaller than all its neighbors. Use the divide-and-conquer algorithm to compute a local minimum with only $O(n)$ comparison between pairs of numbers.

The brute-force algorithm is to check every element to see if it is a local minimum this algorithm has a complexity of $\Theta(n^2)$. To improve the complexity we develop a divide and conquer algorithm. We start the search on the column $j = \frac{n}{2}$. We search for a local minmum on column j say the the ordered pair is (i, j). We compare to its neighbors $(i, j-1), (i, j+1)$. If $(i, j-1) < (i, j)$ then we recurse where $1, 2, 3, \ldots, j-1$ are the new candiates. If $(i, j) > (i, j+1)$ then we recurse where $j+1, j+2, \ldots, n$ are the new candiates. Each iteration throws away half of the columns. The recurrence is of the form of $T(n, n) = T(n, \frac{n}{2}) + \Theta(n)$. The base case is $T(n, 1) = \Theta(n)$. Hence solving the recurrence

$$\begin{aligned} T(n,n) &= \underbrace{\Theta(n) + \Theta(n) + \ldots + \Theta(n)}_{log_2 n \, times} \\ &= \Theta(n \cdot log_2 n) \end{aligned}$$