1. For each of the following problems, give an algorithm that finds the desired numbers within the given amount of time. To keep your answers brief, feel free to use algorithms from the book as subroutines. For the example, S = {6, 13, 19, 3, 8}, 19 − 3 maximizes the difference, while 8 − 6 minimizes the difference.

(a) Let S be an unsorted array of n integers. Give an algorithm that finds the pair x, y ∈ S that maximizes |x − y|. Your algorithm must run in $O(n)$ worst-case time.

(b) Let S be a sorted array of n integers. Give an algorithm that finds the pair x, y ∈ S that maximizes |x − y|. Your algorithm must run in $O(1)$ worst-case time.

(c) Let S be an unsorted array of n integers. Give an algorithm that finds the pair x, y ∈ S that minimizes |x − y|, for x # = y. Your algorithm must run in $O(n\ log\ n)$ worst-case time.

(d) Let S be a sorted array of n integers. Give an algorithm that finds the pair x, y ∈ S that minimizes |x − y|, for x # = y. Your algorithm must run in $O(n)$ worst-case time

آ) کافیست تا یکبار آرایه را پیمایش کنیم $O(n)$ و بزرگترین و کوچکترین عنصر را بیابیم. و تفاضل آنها را بازگردانیم.

ب) کافیست تا تفاضل اولین و آخرین عنصر را بیابیم. $|s[0] - s[n-1]|$

پ) ابتدا اقدام به مرتبط کردن آرایه میکنیم $O(nlogn)$. سپس یکبار آن را پیمایش میکنیم و تفاضل هر دو عنصر مجاور را میبابیم و کمترین آن را باز میگردانیم $O(n)$. پس مرتبطه زمانی الگوریتم برابر میشود با $O(nlogn) + O(n) = O(n \log n)$

ت) مانند مرحله بالا با این تفاوت که مرحله مرتب سازی را نداریم.

2. Given two sets S1 and S2 (each of size n), and a number x, describe an $O(n \log n)$ algorithm for finding whether there exists a pair of elements, one from S1 and one from S2, that add up to x. (For partial credit, give a Θ(n2) algorithm for this problem.)

میتوان الگوریتم زیر را اجرا کرد.

```
sort S2;  // takes O(n logn)
foreach element in S1:
    if binarySearch(S2, x-element) then return (element, x-element);
return Null;
```

در این الگوریتم ابتدا مجموعه S2 را مرتب میکنیم.

سپس برای تمام المنت های این مجموعه x-element را حساب میکنیم. اگر این مقدار در مجموعه دوم موجود بود (چون مرتب شده است میشود با الگوریتم binarySearch در مرتبه زمانی $\log n$ پیدا کرد) آن را باز میگردانیم و در غیر اینصورت نال برمیگردانیم.

$$O(n \log n) + n \times O(\log n) = O(n \log n)$$

3. Devise an algorithm for finding the k smallest elements of an unsorted set of n integers in $O(n + k \log n)$.

روش اول: میتوان یک آرایه با کوچکترین عنصر در ابتدای آن ساخت $O(n)$. سپس k مرتبه باید heapfy کنیم (k عضو را خارج کنیم) که مرتبه آن $O(k \log n)$ است.

روش دوم: ساختن هیپ و خارج کردن k عضو اول آن است.

(ساختن هیپ دارای مرتبه زمانی $\Sigma_{i=1}^{n} \log i = O(n)$ است)

4. Mr. B. C. Dull claims to have developed a new data structure for priority queues that supports the operations Insert, Maximum, and Extract-Max—all in O(1) worst case time. Prove that he is mistaken. (Hint: the argument does not involve a lot of gory details—just think about what this would imply about the $\Omega(n \log n)$ lower bound for sorting.)

حد پایین در مرتب سازی $\Omega(n\log n)$ است. اگر ادعای وی صحیح باشد ، می توان از ساختار داده های او برای مرتب سازی دنباله ای از اعداد n در زمان $O(n)$ استفاده کرد و فقط با وارد کردن تمام اعداد و سپس استخراج حداکثر مقادیر متوالی.

5. Suppose that you are given a sorted sequence of distinct integers {a1, a2, . . . , an}.
Give an O(log n) algorithm to determine whether there exists an i index such as ai = i. For example, in {−10,−3, 3, 5, 7}, a3 = 3. In {2, 3, 4, 5, 6, 7}, there is no such i.

قطعه کد زیر میتواند مساله را حل کند.

```
bool find (int[] s, int low, int high){
    int mid = ( low + high ) / 2 ;
    if (s[mid] == mid) return true;
    else if (s[mid] > mid ) { // possible index is in the left half of the array
        return find (s, low , mid);
    }else return find (s, mid, high); // possible index is in the right half of t
he array
    return false;
}

low = 0;
high = n;
find (s, low, high);
```

ترفند حل این مسئله این است که بدانیم شاخص های آرایه و مقادیر آرایه در شاخص های مربوطه به صورت یکنواخت در حال افزایش هستند. این به این دلیل است که آرایه مرتب شده است.

حال اگر عنصری در برخی از شاخص ها از مقدار شاخص بیشتر باشد ، دیگر نیازی به جستجوی سمت راست آن آرایه نیست و در عوض باید سمت چپ آن جستجو شود (و برعکس نیز).