۱.

الف و ب) Insertion و Bubble بدترین حالت $O(n^2)$ و حالت میانگین $\theta(n^2)$ و بهترین حالت $\Omega(n)$

ج) Merge بدترین حالت بهترین حالت و بدترین حالت و حالت میانگین $O(n \log n)$

د) Quick بدترین حالت $O(n^2)$ و بهترین حالت و حالت میانگین $O(n \log n)$

ه) Heap بدترین حالت بهترین حالت و حالت میانگین $O(n \log n)$

چون آرایه مرتب شده است ، بهتر است از Insertion یا Bubble استفاده شود.

۲. حداکثر $r + p - 1$ مقایسه لازم است. می توان اینطور در نظر گرفت که تمام اعضای A غیر از آخرین عضو با عضو اول B مقایسه می شوند و سپس تمام اعضای B با آخرین عضو A مقایسه و برای عضو پایانی A نیازی به مقایسه نیست.

۳.

الف) **نادرست** ؛ در حالتی که آرایه به صورت نزولی باشد ، Quick زمان بیشتری می برد.

ب) **درست** ؛ Binary Insertion در بدترین حالت $O(n \log n)$ ولی quick $O(n^2)$ است.

ج) **نادرست** ؛ quick در بدترین حالت $O(n^2)$ و Heap $O(n \log n)$ است.

د) **درست** ؛ زیرا insertion $O(n^2)$ است.

ه) **درست**

۴. ؟

۵. می توان از ساختمان داده های دیگر برای نگه داری عناصر داری collision استفاده کرد مانند BST (Red Black) ، که زمان اضافه و حذف و جست و جو را $O(\log n)$ می کند یا لیست پیوندی دو طرفه ، که زمان اضافه و حذف را $O(1)$ و زمان جست جو را $O(n)$می کند یا حتی یک HashMap دیگر ؛ تا زمان دسترسی به عناصر و جست و جو در بدترین حالت که همه ی عناصر به یک خانه از جدول مپ شده اند کاهش یابد.

۶.

تقسیم : در هر مرحله ، اگر اندیس شروع آرایه i باشد ، مسئله را برای اندیس 2i و 2i+1 مادامی که به انتهای آرایه نرسند حل می کنیم و سپس عنصر i ام را با مینیمم عنصر های 2i+1 , 2i , i جابجا می کنیم.

غلبه : وقتی به مرحله ای رسیدیم که i بزرگتر یا مساوی انتهای آرایه است ، پاسخ همان i است و باز میگردیم.

به صورت کد :

```
void heapify(int i,int arr[]){
    if(i >= arr.size)    //If end of array is reached
        return;
    else{
        heapify(2*i,arr);
        heapify(2*i+1,arr);
        swap(arr[i],__min(arr[i],arr[2*i],arr[2*i+1])); //Check for array index out of bounds is not
mentioned
    }
}
```

۷.

روش ۱:

تقسیم : آرایه را به بزرگترین زیر آرایه هایی که تمام اعضای هر کدام از آن ها مثبت اند یا منفی تقسیم می کنیم. سپس به ازای هر دو آرایه ی مثبت ، که یک منفی هم بین آن ها هست ، جواب ، ماکسیممِ { پاسخ هر کدام و مجموع اندازه هر سه} است. (وقتی به کمتر از سه زیر آرایه رسیدیم ، پاسخ ماکسیمم جواب ها است.

غلبه : اگر تمام اعضای زیر آرایه ها منفی بودند ، پاسخ ماکسیمم اعضای آن و اگر مثبت بودند پاسخ مجموع اعضای آن است.

روش ۲:

تقسیم: آرایه را از وسط به دو قسمت تقسیم کرده و پاسخ خواهد بود ماکسیممِ {پاسخ چپ و پاسخ راست و بزرگترین زیر آرایه با شروع از چپ و پایان در راست}

غلبه: اگر به یک عضو رسیدیم پاسخ همان عضو است.

به صورت کد:

```
int find_max_cross_subarray(int left,int middle,int right){
    int sum = 0;     //Sum of elements in sub array
    int left_max = -1*infinity; //Maximum sum in left of middle
    int right_max = -1*infinity;     //Maximum sum in right of middle
    for(i = middle , i >= left ; i--){
        sum += arr[i];
        left_max = __max(left_max,sum);
        }
        sum = 0;
    for(i = middle , i <= right ; i++){
        sum += arr[i];
        right_max = __max(right_max,sum);
        }
        return right_max+left_max;
}

int find_max_subarray(i,j){
    if(i==j)     //If there's one element in sub array
        return arr[i];
    else
        return __max(find_max_subarray(i,(i+j)/2)
        ,find_max_subarray((i+j)/2,j)
        ,find_max_cross_subarray(i,(i+j)/2,j));
}
```

۸. با استفاده از DP :

روش ۱ :

تعریف : dp[i][j] بلندترین زیردنباله خودوارون از اندیس i تا اندیس j از ورودی است.

رابطه بازگشتی : برای dp[i][j] دو حالت وجود دارد ، اگر مقدار ورودی در اندیس های i , j برابر باشند ، مقدار dp[i+1][j-1]+2 خواهد بود و در غیر این صورت ، بیشترین بین dp[i][j-1] و dp[i+1][j] است.

$$dp[i][j] = \begin{cases} dp[i+1][j-1] + 2 \; if \; input[i] = input[j] \\ max\{dp[i][j-1], dp[i+1][j]\} \; otherwise \end{cases}$$

حالت پایه : هر زیردنباله ی تک عضوی ، بلندترین طول ۱ را دارد که یعنی *dp[i][i] = 1*

به صورت کد :

```
//Final Answer is max_symmetric_subsequent(input,1,input.length)
int max_symmetric_subsequent(input,i,j){
    if(i==j)
        return 1;
    if(input[i]==input[j])
        return max_symmetric_subsequent(input,i+1,j-1)+2;
    else return __max(max_symmetric_subsequent(input,i,j-1)
                  ,max_symmetric_subsequent(input,i+1,j));
}
```

پیچیدگی زمانی $O(n^2)$ است زیرا برنامه مانندو حلقه یکی روی اندیس اول و دیگری روی اندیس دوم پیمایش می کند عمل می کند .

روش ۲ : می توان آرایه استفاده شده به صورت *dp[i]* که بزرگترین زیردنباله متقارن از اندس *0* تا *i* را نگه می دارد در نظر گرفت و در این صورت با بررسی *i,j* های مختلف مشابه روش اول عمل می کنیم ، فقط در حالتی که مقدار ورودی در اندیس های i , j برابر است ، dp[j]=dp[j-1]+2 را در نظر میگیریم.

۹. تابع هش $h(n) = \frac{n}{2}$ را در نظر میگیریم در این صورت هر یک از اعداد ورودی ، در بازه ی $[0, n]$ مپ می شوند. به طور مثال اعداد ۰و۱ به اندیس ۰ ، اعداد ۲و۳ به اندیس ۱ و همین طور.حین عمل مپ اگر collision رخ داد ، با لیست پیوندی عناصر را پشت سر هم می گذاریم به طوری که اگر عددی که اضافه شده کوچکتر بود به ابتدا ی لیست و در غیر این صورت به انتهای لیست اضافه می شود. با توجه به بازه ی داده شده ، حداکثر تعداد اعداد متفاوت در یک خانه ی جدول هش مپ می شوند ، ۲ است.

در پایان با پیمایش کل hash table ، خروجی مرتب شده بدست می آید. پیچیدگی زمانی نیز $O(2n) = O(n)$ است.

۱۰.

الف) غلط است. برای مثال اگر موقعیت نقاط به ترتیب زیر باشد :

$$0 , 0.9 , 1 , 1.1 , 1.2 , 1.3 , 2.1$$

الگوریتم در مرحله ی اول بازه ی $[0.9,1.9]$ را انتخاب می کند که شامل ۵ نقطه می شود . نقاط باقی مانده :

$$0 , 2.1$$

حال باید ۲ پاره خط دیگر برای این دو نقطه اضافه کند که در نتیجه با ۳ پاره خط این کار را انجام می دهد.

امّا میشد با دو پاره خط در بازه های $[0 - 1], [1.1 - 2.1]$ این کار را انجام داد پس الگوریتم پاسخ بهینه نمی دهد.

ب) درست است. زیرا بهترین حالت برای پوشاندن سمت چپ ترین نقطه آن است که در ابتدای آن باشد که در این صورت بیشترین تعداد نقطه با حداقل پاره خط پوشانده می شوند.

در واقع پاسخ بهتر ، در صورت وجود ، منجر به خارج شدن قسمتی از سمت چپ ترین پاره خط از بازه ای که نقاط در آن هستند می شود.

۱۱.ابتدا تمام ستون ها را دو به دو از ابتدا و انتها جابجا می کنیم؛یعنی ستون ۱ با ستون n ، ستون ۲ با ستون n-1 و همین طور.(اگر تعداد ستون ها فرد باشد ، ستون وسط تغییری نمی کند.) سپس همین کار را با تمام سطر ها انجام می دهیم.

ماتریس حاصل ، پاسخ است ؛ یعنی ترانهاده ی ماتریس ورودی. دلیل آن این است در مرحله ی اول با تعویض ستون ها خواهیم داشت :

$$t'_{i,j} = t_{i,n-j}$$

و در مرحله ی بعد با تعویض سطر ها :

$$t''_{i,j} = t'_{n-i,j} = t_{n-i,n-j}$$

و با توجه به اینکه ماتریس پاد مقارن بوده و فقط متشکل از ۱ و ۱- است ، خواهیم داشت :

$$t''_{i,j} = t_{n-i,n-j} = -t_{n-j,n-i} = t_{j,i}$$

به صورت کد :

```
void tranpose_matrix(matrix){
    for(int j = 1 ; j < n/2 ; j++)
        matrix.swap_column(j,n-j);
    for(int i = 1 ; i < n/2 ; i++)
        matrix.swap_row(i,n-i);
    return matrix;
}
```

پیچیدگی با در نظر گرفتن پیچیدگی n برای هر جابجایی ، $O(n^2)$ است.

۱۲.مسئله را با گراف جهت دار مدل سازی می کنیم به این صورت که هر کارمند یه رأس گراف است و کارمند آ با کارمند ب رابطه دارد اگر آ مغلوب ب باشد. توجه می کنیم که رابطه مغلوب بودن ، خاصیت تعدی دارد و خاصیت تقارن ندارد ،پس فاقد دور خواهد بود . البته گراف حاصل می تواند نا همبند باشد.

حال پاسخ مسئله طول بلندترین مسیر مؤلفه های همبندی این گراف جهت دار فاقد دور (DAG) است.

برای حل ،مادامی که تمام رئوس پیمایش نشده اند ، DFS از رأس دلخواه میزنیم و هر وقت به رأسی رسیدیم که تمام همسایه هایش پیمایش شده بودند ، یا همسایه ای نداشت ، آن را داخل یک stack میریزیم. در این صورت ، در پایان ، بالاترین عضو پشته ، عضوی است که کسی مغلوب آن نیست.

حال با شروع از همین عضو ، بلندترین مسیر در مؤلفه هم بندی آن بدست می آید به این صورت که اگر dis[i] بلندترین مسیر از رأس i باشد ، مقدار آن خواهد بود :

$$dis[i] = \begin{cases} 0 \ if \ no \ adjacent \ vertex \ exists \\ 1 + \max\{dis[a_i]\} \ when \ a_i \ are \ adjacent \ vertices \end{cases}$$

مادامی که تمام اعضای پشته پیمایش نشده اند ، این عمل را تکرار می کنیم.

در پایان بیشترین مقدار *dis* ، بیشترین تعداد اعضای مهمانی است.

به صورت کد :

```
struct employee {
    p;//Power of employee
    c;//Charm of employee
};

int max_party_size(staff){
    for(employee e : staff)
        if(!e.visited)
            DFS(e,stack);
    int result = 0;
    while(!stack.empty){
        while(stack.top.visited) stack.pop();
        find_longest_path(stack.top);
        result = __max(stack.top.max,result)
    }
```

```
    return result;
}

void DFS(employee,stack){
    for(employee e : employee.adjacents)//adjacents = each employee x such that x.p > e.p & x.c > e.c
        if(!e.visited)
            DFS(e,stack);
    employee.visited = true;
    stack.push(e);
}

void find_longest_path(employee){
    int max = 0;
    for(employee e : employee.adjacents){
        if(!e.visited)
            find_longest_path(e);
        if(e.max+1 > max)
            max = e.max+1;
    }
    employee.visited = true;
    employee.max = max; //max = maximum length of path starting from employee
}
```