1 Обязательная часть

1. ***Сложность сортировок***

   - $..! = o(n \cdot \log n)$

     Предположим, что утверждение неверно. То есть $\forall c > 0\ \exists N$: $\forall n \geq N\ k < c \cdot n \log n$, где k - количество сравнений (см. теорию). Также по условию $k \geq logn! - n \log 100$. Получаем:

     $$\forall c > 0\ \exists N\text{: } \forall n \geq N\ c \cdot n \log n > logn! - n \log 100$$

     Заметим, что $\exists c_1 > 0\ \exists N_1$: $\forall n \geq N_1\ logn! - n \log 100 \geq c_1 \cdot \log n!$. Известно, что $\log n! = \Theta(n \log n)$. То есть

     $$\exists c_2 > 0\ \exists N_2\text{: } \forall n \geq N_2\ logn! - n \log 100 \geq c_2 \cdot n \log n$$

     В итоге получаем, что

     $$c \cdot n \log n > logn! - n \log 100 \geq c_2 \cdot n \log n$$

     Заметим, что если рассмотреть $c < c_2$ то неравенство будет ложным. Получили противоречие.

2. ***Anti-QuickSort Test***

   - *Простой QSort*

     Заведем массив $mass[1 : n]$, в котором на i-ой позиции будет стоять i-ый элемет. Заведем пустой массив a - перестановка, на которой будет работать на $\Omega(n^2)$ Запустим на $mass$ нашу сортировку. Каждый раз, когда возрвращают $j = pivot(l, r)$, мы в ячейку $a[mass[j]]$ ставим значение u - номер шага QuickSort (1 шаг - pivot(1, n), 2 шаг - pivot(1, pivot(1, n)) и тд).

     Всего таких шагов будет n. В конце сортировки мы получим заполненный массив a, который и является искомой перестановкой, ведь его элементы стоят так, что на каждом шаге сортировки QSort мы будем получать элемент, меньший всех других, то есть массив следующего шага будет меньше всего на 1.

     Изначально все элементы $mass$ были отсортированы, то есть QSort отработал за нужное нам время.

   - *QSprt с медианой*

     Делаем все то же самое, что и в первом пункте, но с небольшой поправкой. Для удобвства будет считать, что $mass[i] \leq mass[j] \leq mass[k]$. Тогда на каждом шаге будем в $a[mass[j]]$ класть i-ое нечетное, а в $a[mass[i]]$ - i-ое четное. Тогда опять получится такая перестановка a, что при QSort на следующий шаг мы будем передавать массив всего на один меньший предыдущего и сортировка отработает за $\mathcal{O}(n^2)$.

3. ***Отрезки на прямой***

- *Непересекающиеся* Пусть у нас есть массив пар (начало отрезка, конец отрезка). Отсортируем его с приоритетом по началу. Бежим по получившемуся массиву от 1ой пары, до n-1ой. Если начало отрезка следующей пары не больше конца отрезка нашей, то не берем эт пару в наше множество. Иначе - берем. Работает за время сортировки.
- *Покрывающие*

  На этот раз отсоритруем отрезки с приоритетом по концу. Найдем первый отрезок, чем конец не меньше 0. Дальше набираем наше множество по таким правилам:

  1) Если начало рассматриваемого отрезка больше начала последнего из множества взятых, то удаляем последний отрезок. Делаем так до тех пор, пока начало рассматриваемого не станет больше начала верхнего из множества. В конце добавляем рассматриваемый в наше множество.
  2) Если начало больше предыдущего, но при этом не больше конца предыдущего, то добавляем этот отрезок в множество.
  3) Если начало больше конца предыдущего - игнорируем этот отрезок, он нам не нужен, принесет выгоды меньше, чем один из последующих с большим концом.
  4) Если вдруг оказывается, что конец последнего из множества взятых отрезов не меньше M, то заканчиваем работу алгоритма.

  Работает за время сортировки, так как при наборе множества каждый отрезок будет рассматриваться не более двух раз (удаение и потенциальное добавление). То есть множество наберем за $\mathcal{O}(n)$.
- *Время покрытия*

  При помощи бинпоиска по ответу мы умеем искать время, когда впервые покрылся весь отрезок $[0, M]$. Нижняя граница бинпоиска - 0. Верхняя - время покрытия $[0, M]$ отрезком с минимальным r_i. Проверка, покрылся ли в этот момент времени отрезок написана в предыдущем пункте.

4. ***SiftUp***

Куча построена так, что родитель меньше потомка. Когда мы добавляем в конец кучи новый элемент, мы точно может предсказать, по какому пуйти он пройдет свою проверку ($h[n], h[n/2], h[n/4]$ и тд). Причем элементы в этой последовательности убывают. То есть мы можем искать место, куда вставить наш новый элемент не по шагам, а при помощи бинпоиска. Так как всего элементов в последоватльности - $\log n$ (количество уровней в куче), то бинпоиск отработает за $\log logn$, что и требуется.