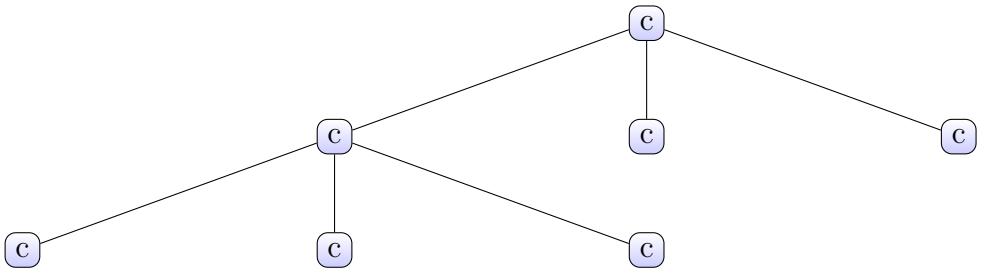
1. $f(n) = c$, где $c = 2021$

При малых n, как было оговорено, будем считать, что $T(n) = \Theta(1)$, а при больших n, $n > 2020$ построим рекурсивное дерево.



Если строго, то на каждом уровне константа делится на 4, но для оценки асимптотики это неважно.

Как будет показано ниже, от округления асимптотика не зависит, так что можно его опустить.

На i-ом шаге узлов: $3^i c$

Высота дерева: $h = \log_4 (n - 2020)$

$T(n) = \sum\limits_{i=0}^{h} 3^i c = c\frac{3^{h+1}-1}{3-1} = \frac{c}{2}(3(n-2020)^{\log_4 3} - 1) = O(n^{\log_4 3})$

Но на любом начальном уровне, то есть $n < 2020$, можно игнорировать количество дополнительных операций, ведь это приведет только к добавлению константы, что не повлияет на асимптотику, так что $T(n) = \Omega(n^{\log_4 3})$

Следовательно, $\mathbf{T(n) = \Theta(n^{\log_4 3})}$

2. Заметим, что $a \cdot b = \frac{(a+b)^2 - a^2 - b^2}{2}$

Операции сложения и вычитания требуют $\log_2 (a + b) = O(n)$, деление на 2 – это побитовый сдвиг: $O(1)$, возведение в квадрат по условию $O(n)$. Следовательно, итоговая сложность алгоритма: $O(n)$.

3.

а) $a = 36, b = 6; d = \log_6 36 = 2$

$f(n^2) = n^2 = \Theta(n^2)$

Следовательно, по основной теореме о рекурсии (случай 2): $\mathbf{T(n) = \Theta(n^2 \log n)}$

б) $a = 3, b = 3; d = \log_3 3 = 1$

$f(n) = n^2 = \Omega(n^{1+\epsilon})$, например, для $\epsilon = 1/2$

А также проверим выполнение условия регулярности: $3(\frac{n}{3})^2 = \frac{n^2}{3} = \frac{f(n)}{3} = cf(n), c = \frac{1}{3}$

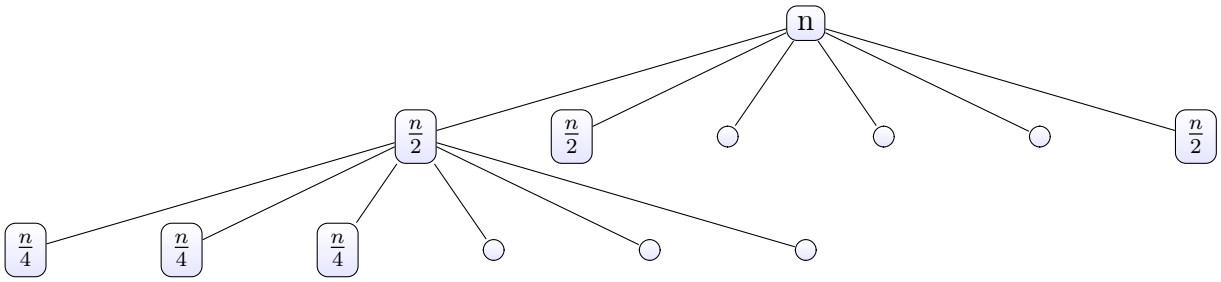
Следовательно, по основной теореме о рекурсии (случай 3): $\mathbf{T(n) = \Theta(n^2)}$

в) $a = 4, b = 2; d = \log_2 4 = 2$

$f(n) = \frac{n}{\log n} = O(n^{2-\epsilon})$, например, для $\epsilon = \frac{1}{2}$

Следовательно, по основной теореме о рекурсии (случай 1): $\mathbf{T(n) = \Theta(n^2)}$

4. 1.



На первом уровне подзадач n, на втором – $\frac{n}{2}n$ и т.д.
Таким образом, на i-ом уровне $\frac{n}{2^{i-1}}...\frac{n}{2}n$ подзадач размера $\frac{n}{2^i}$.
Высота дерева: $h = \log_2 n$
$T(n) = \sum_{i=0}^{h} \frac{n}{2^{i-1}}...\frac{n}{2}n\frac{n}{2^i} \leq n\sum_{i=0}^{h}(\frac{n}{2})^i = n\frac{(\frac{n}{2})^{h+1}-1}{\frac{n}{2}-1} \leq nn^{\log_2 n} = \Theta(n^{\log n})$

2. Если n – произвольное, то $\lceil\frac{n}{2}\rceil$ будет давать округления с избытком. То есть, будет последовательность, $n_0 = n, n_i = \lceil\frac{n_{i-1}}{b}\rceil$. Покажем, что после $\log_b n$ итераций получится число, ограниченное не зависящей от n константой. В данной задаче $b = 2$.
$\lceil x\rceil \leq x + 1$
$n_0 \leq n$
$n_1 \leq \frac{n}{b} + 1$
$n_2 \leq \frac{n}{b^2}\frac{n}{b} + 1$
и т.д.
$1 + \frac{1}{b} + \frac{1}{b^2} + \frac{1}{b^3} + ... \leq \frac{1}{b-1}$
$n_i \leq \frac{n}{b^i} + \frac{b}{b-1} \leq b + \frac{b}{b-1} = O(1)$, , $i = \lfloor\log_b n\rfloor$
Следовательно, $T(n) = f(n_0)+aT(n_1) \leq f(n_0)+af(n_1)+a^2f(n_2)+...+a^{\lfloor\log_b n\rfloor}T(n_{\lfloor\log_b n\rfloor}) = \Theta(n^{\log_b a}) + \sum\limits_{i=0}^{\lfloor\log_b n\rfloor-1} a^i f(n_i)$
$f(n_i) \leq c(\frac{n}{b_i} + \frac{b}{b-1})^{\log_b a} \leq c(\frac{n^{\log_b a}}{a^i})(1 + \frac{b}{b-1})^{\log_b a} = O(\frac{n^{\log_b a}}{a^i})$
То есть, эта добавка никак не портит асимптотику алгоритма. Аналогичное будет и при округлении с недостатком.

Таким образом, округление не влияет на асимптотику, и ответ будет тем же. $T(n) = \Theta(n^{\log n})$

5. Описание алгоритма:

Пусть на вход поступили два числа: a и b. Заведем переменную c, которой присвоим начальное значение 0 и построим рекурсивный алгоритм, в котором будем проверять делится ли предполагаемый НОК$(a, b) = c$ на a и b, если да – то выдаем c, иначе увеличим c на $\max(a, b)$, который вычислим до входа в функцию и будем считать, что он хранится в переменной a. Значит, при наращивании c она гарантировано делится на a и останется проверить делимость на b.

$c = 0$
`если` (a < b) {
  `поменять местами` a `и` b
}
НОК(a, b) {
  $c = c + a$

```
    если c делится на b {
      вернуть c
    }
иначе {
      НОК(a, b)
    }
}
```

Корректность:

По определению $\text{НОК}(a, b)$ – это наименьшее натуральное число, которое кратно a и b одновременно.

Заметим, что $\text{НОК}(a, b) \in [\max(a, b); a \cdot b]$. Действительно, меньше макисмума из двух чисел быть не может, иначе бы НОК не был бы кратен максимуму из двух чисел и гарантировано меньше произведения, так как оно уж точно делится на каждое число. Таким образом, стартуя с $\max(a, b)$ и увеличивая значения на $\max(a, b)$, ведь $\text{НОК}(a, b)$ должен обязательно делится на каждое, так что прибавить меньше мы не можем, проверяем делимость на меньшее, в конце концов приходя к произведению в худшем случае.

Сложность:

Операция сложения: $O(n)$, деления $O(n^2)$ (предыдущее дз). То есть, в каждом вызове будет $O(n^2)$.

В худшем случае, если $\text{НОК}(a, b) = a \cdot b$, вызовов функции будет $b = O(n)$.

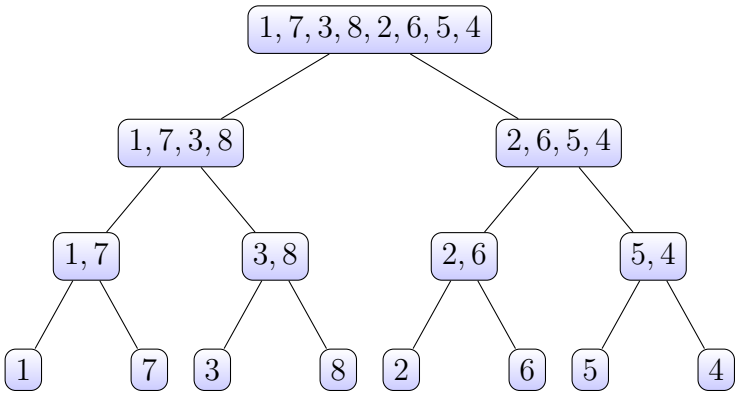
Таким образом, итоговая сложность $O(n^3)$.

6. Для начала разделим массив на элементы точно так же, как в сортировке MergeSort, а затем при склеивании элементов заведем переменную *count*, которая будет инкрементироваться при добавлении новой инверсии.

Число инверсий на каждом уровне склеивания для каждого элемента – это длина текущей левой части минус индекс текущего элемента, с которым мы проводим сравнение.

Действительно, так как элементы в каждой части склеивания упорядочены, то, если элемент из правой части оказался меньше текущего, то он уж точно будет меньше всех элементов в левой части на данном шаге, стоящих правее его.

Продемонстрируем алгоритм:



1 и 7: инверсий нет; 3 и 8: инверсий нет; 2 и 6: инверсий нет; 5 и 4: инверсия есть переменную-счетчик *count* увеличиваем на $1-0=1$, где 1 – длина текущей левой части, 0 – индекс текущего сравниваемого элемента;

Склеиваем $1,7$ и $3,8$:

1 и 3: инверсий нет; 7 и 3: инверсия есть: *count* увеличиваем на $2-1=1$, так как длина левой части уже 2, а индекс текущего элемента 1; 7 и 8: инверсий нет;

Аналогично делаем для правой подобласти и т.д. склеиваем вновь полученные подмассивы.

Итоговое число инверсий, полученное для данного примера алгоритмом: 13, что также совпадает с теоретическим прямым подсчетом.

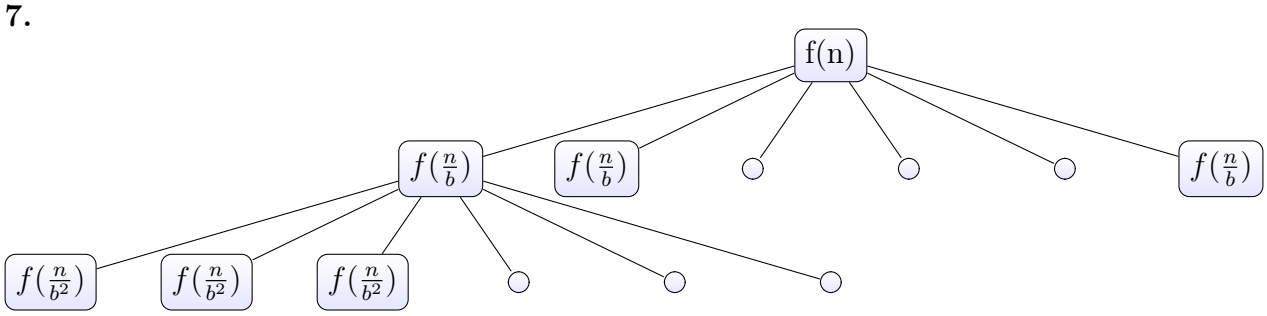
Корректность:

На каждом шаге мы гарантировано находим все инверсии в данном подмассиве (корректность формулы для счетчика была описана выше), склеивая их вновь и вновь мы проходим весь массив и на выходе получаем отсортированный, то есть, мы отслеживаем все инверсии, получая на выходе неинвертированный массив.

Сложность:

Такая же, как у сортировки MergeSort, ведь мы дополнительно не проделываем никаких операций, кроме наращивания счетчика *count*. Следовательно, $O(n\log n)$ (доказано на лекции).

7.



На i-ом уровне a^i подзадач размера $f(\frac{n}{b^i})$, высота дерева $h=\log_b n$.

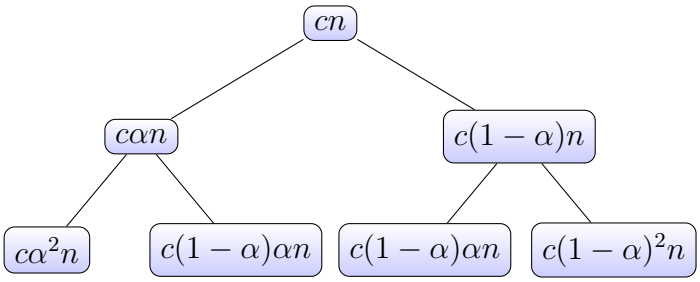
То есть, $T_1(n)=\sum\limits_{i=0}^{h} f(\frac{n}{b^i})a^i$, $T_2(n)=\sum\limits_{i=0}^{h} g(\frac{n}{b^i})a^i$. Так как по условию $f(n)=\Theta(g(n))$, то $C_1g(n)\le f(n)\le C_2g(n)$ и, следовательно, $C_1g(\frac{n}{b^i})\le f(\frac{n}{b^i})\le C_2g(\frac{n}{b^i})$ для любого i в силу монотонности.

Следовательно, $C_1\sum\limits_{i=0}^{h} g(\frac{n}{b^i})a^i\le\sum\limits_{i=0}^{h} f(\frac{n}{b^i})a^i\le C_2\sum\limits_{i=0}^{h} g(\frac{n}{b^i})a^i$. То есть, $T_1(n)=\Theta(T_2(n))$.

8.

а)

Округдения писать в формулах не будем, как было показано выше, это не влияет на асимптотику алгоритма.



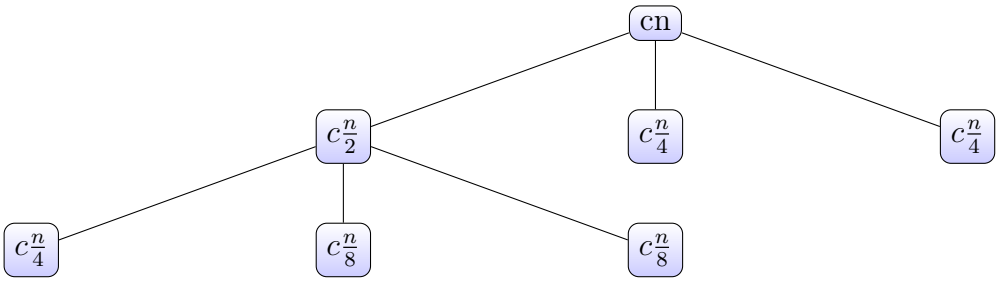
На i-ом уровне: $cn(\alpha^i + 2(1-\alpha)\alpha^{i-1} + ... + (1-\alpha)^i) = cn(\alpha + (1-\alpha))^i = cn \cdot 1$ (бином Ньютона)

Высота дерева: $h = \log_2 n$

Таким образом, $T(n) = \sum\limits_{i=0}^{h} cn = cn(\log n + 1) = \Theta(n \log n)$

$\mathbf{T(n) = \Theta(n \log n)}$

б)



Дерево будет неравномерной длины: $h_1 = \log_2 n, :\ h_2 = \log_4 n$

Самое глубокое: $h_1 = \log_2 n$

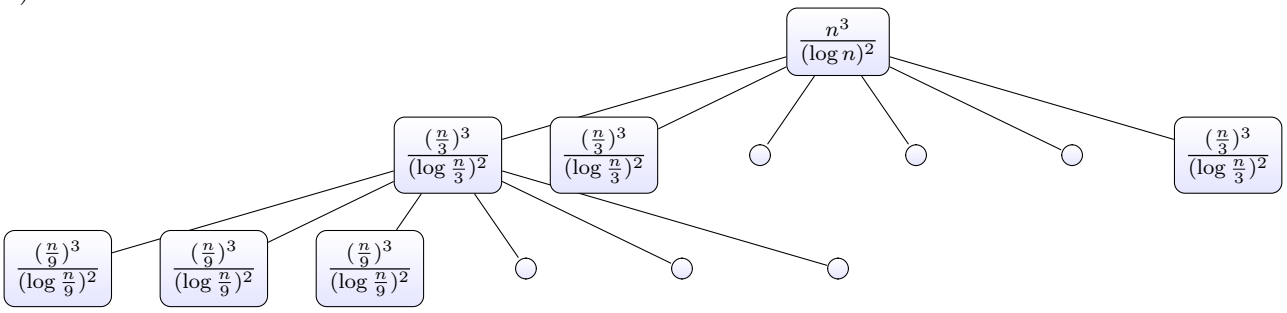
Как было показано выше, от округления асимптотика не меняется.

На каждом уровне сумма всех подзадач: cn

То есть, $T(n) = \sum\limits_{i=0}^{h} cn = cn(\log n + 1) = \Theta(n \log n)$

$\mathbf{T(n) = \Theta(n \log n)}$

в)



На i-ом уровне: 27^i подзадач $\frac{(\frac{n}{3^i})^3}{(\log \frac{n}{3^i})^2}$.

Высота дерева: $h = \log_3 n$

Следовательно, $T(n) = \sum_{i=0}^{h} 27^i \frac{(\frac{n}{3^i})^3}{(\log \frac{n}{3^i})^2} = n^3 \sum_{i=0}^{h} \frac{1}{\log(\frac{n}{3^i})^2} = n^3 \sum_{i=0}^{h} \frac{1}{(\log n - i)^2} = \frac{n^3}{(\log n)^2} \sum_{i=0}^{h} \frac{1}{(1-\frac{i}{\log n})^2} \leq$
$\frac{n^3}{(\log n)^2} \sum_{i=0}^{h} (1 + 2\frac{i}{\log n}) = \frac{n^3}{(\log n)^2}(\log n + 1 + \frac{2}{\log n} \frac{\log n(\log n+1)}{2}) = \Theta(\frac{n^3}{\log n})$

$\mathbf{T(n) = \Theta(\frac{n^3}{\log n})}$

9. Построим рекурсивный алгоритм нахождения заданного максимума.

$max\_val = \max(len \cdot min)$, где len – длина текущего подмассива, min – минимум текущего подмассива. a – заданный исходный массив.

Первым действием установим начальные значения $len =$ длина(a), $min =$ минимум (a) и удалим это минимальное значение из массива. Если минимальное значение дублируется (есть неуникальные элементы), удаляем их все. Ведь иначе на следующем шаге мы опять найдем этот минимум, но длина будет уже меньше, так что max_val уж точно не может стать больше.

Запустим рекурсивную функцию, которая будет считать минимум в обрезанном массиве и искать max_val, если max_val оказывается больше предыдущего значения – его необходимо обновить. Удаляем из массива текущий минимум и все его дубликаты. Если длина массива стала равной 0 – возвращаем max_val, иначе запускаем рекурсивный алгоритм вновь.

$len =$ длина(a), $min =$ минимум (a)
`удаляем` min `и все его дубликаты`
$max\_val = len \cdot min$

$search(a, max\_val)$ {
  $min =$ минимум (a)
  $len =$ длина(a)
  `удаляем` min `и все его дубликаты`
  $max\_val\_new = min \cdot len$
  `если` $max\_val\_new > max\_val$ {
    $max\_val = max\_val\_new$
  }
  `если` $len == 0$ {
    `вернуть` max_val
  }
  `иначе` $search(a, max\_val)$
}

Корректность:

Алгоритм детерминированный. Проходим в любом случае всевозможные подмассивы данного массива. Удаление минимума и его дубликатов на каждом шагу не вредит результату, так как при удалении уменьшается длина, а это влечет только к уменьшению искомого значения с данным минимумом подмассива. Таким образом, за конечное число рекурсивных вызовов данный алгоритм всегда найдет искомое значение.

Сложность:

Пусть n – длина входного массива. Поиск минимума в массиве – $O(n)$, на каждом шаге рекурсии массив уменьшается минимум на одно значение, число рекурсивных вызовов в худшем случае – $O(n)$. Следовательно, итоговая сложность – $O(n^2)$.