① Assuming that the boxes are numbered down to up from 1 to 2n change box 2 and box 2n-1. This makes the first and last pair of boxes alternate in the required pattern and hence reduces the problem to the same problem with 2(n-2) middle boxes If n is even, the number of times this operation needs to be repeated is equal to n/2 ; if n is odd it is equal to (n-1)/2. The formula n/2 provides a closed-form answer for both cases.
Note that this can also be obtained by solving the recurrence $M(n) = M(n-2) + 1$ for $n \geq 2$, $M(2) = 1$ $M(1) = 0$, where M(n) is the number of moves made by the decrease-by-two algorithm described above since any algorithm for this problem must move at least one block box for each of the n/2 nonoverlapping pairs of the changed box, n/2 is the least number of moves needed to solve the problem.
This problem solve similar insertion sort idea so we can make analysis similar to insertion sort

Best case → occurs if the input is already sorted

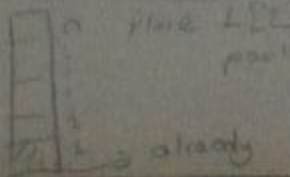
$$B(n) = \sum_{i=2}^{n} 1 = n-1 \in O(n)$$

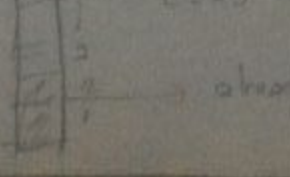
Worst case → For each iteration of the for loop the basic operation is executed maximum num of times

$$W(n) = \sum_{i=2}^{n} (i-1) = \sum_{i=1}^{n-1} = \frac{n(n-1)}{2} \in O(n^2)$$

The situation occurs if the list $L[1..n]$ is already sorted in reverse order

Average case





$\Rightarrow T = T_{1,2} T_2 + \cdots + T_{n-1} = \sum_{i=1}^{n-1} T_i$

$A(n) = E[T] = E\left[\sum_{i=1}^{n-1} T_i\right] = \sum_{i=1}^{n-1} E[T_i] = E[T_1] + E[T_2] + \cdots + E[T_{n-1}]$

- Each of the T_i's are random variables
- Calculate $E[T_i]$

$$E[T_i] = \sum_{j=1}^{\infty} j \cdot P(T=j)$$

$$A(n) = E[T] = \sum_{i=1}^{n-1} E[T_i] = \sum_{i=1}^{n-1} \frac{1}{2} \cdot i = \frac{1}{i+1} = O(n^2)$$

② Among n identical-looking coins one is fake with a balance scale, we can compare any two sets of coins That is tipping to the left, to the right or staying even, the balance scale will tell whether the sets weigh the same or which of the sets is heavier than the other but not by how much. We can make for solving this problem is to divide n coins into two piles of n/2 coins each, leaving one extra coin aside if n is odd and put the two piles on the scale. If the piles weigh the same, the coin put aside must be fake. Otherwise we can proceed in the same manner with the lighter pile, which must be the one with the fake coin. That is we have decrease and conquer strategy for this problem.

→ If n is even we put half the coins on each side of the balance The side which is lightest contains the fake coin.

→ If n is odd the (n-1) is even and we split the (n-1) coins in half and put each half on the balance. If both sides are equal weight, than we are done because the coin we left out is the fake one If the balance is not even then we choose the lightest pile of coins to be the one containing the fake

→ We continue in this manner until we have found the false coin by reducing the problem to weighing one on each side of the balance or found it by being the one we didn't weigh.

Analysis

W(n) be the number of weighings needed in

Worst Case

$W(n) = W(\lfloor n/2 \rfloor) + 1$ for $n \geq 1$, $W(1) = 0$

The solution to the recurrence for the number of weighings is also very similar to the one we had for binary search $W(n) = \log_2 n$

In the fact that the above algorithm is not the most efficient solution It would be more efficient to divide the coins not into two but into three piles of about $n/3$ coins each. After weighing two of the piles, we can reduce the instance size by a factor of three. Accordingly, where $\log_3 n$ is smaller than $\log_2 n$

It is almost identical to the one for the worst-case number of comparisons in binary search both algorithms are based on the same technique of solving an instance size So in the Best case compare is similar binary search A constant number of comparisons are required $O(1)$

Average Case

In average case take the sum over all elements of the product of number of comparisons required to find each element and the probability of searching for that elements To simplify the analysis assume that no item which is not in A will be searched for and that the probabilities of searching for each element are uniform

average case $O(\log n)$

③ Theoretical Analysis

Quicksort → Already sorted and reverse sorted inputs are the worst cases

Insertion Sort ⇒ The best case is the already sorted input and the worst case is the already reverse sorted input. Average case time complexity $O(n^2)$

→ Insertion sort is faster for small n because Quick sort has extra overhead from the recursive function calls

→ Insertion sort is also more stable than Quick sort and requires less memory

→ Best case of Insertion sort is O(n), Best case of quick sort is O(nlogn)

Insertion sort use decrease and conquer approach but quick sort is a divide and conquer approach with recurrence relation.

$T(n) = T(k) + T(n-k-1) + cn$

Worst case when the array is sorted or reverse sorted, the partition algorithm divides the array in two subarrays with 0 and n-1 elements.

$T(n) = T(0) + T(n-1) + cn$ we get $T(n) = O(n^2)$

Best case and Average case : On an average, the partition algorithm divides the array in two subarrays with equal size Therefore

$T(n) = 2T(n/2) + cn$ we get $T(n) = O(n \log n)$

Experimental Analysis

Insertion sort and Quick sort algorithms tested with Python language and finded results. Both of them an array sorted from smallest to biggest. And swaps operations were counted for both of them.

Array List is 19, 12, 1, 8, 15, 22 according to this list finded Quick sort algorithm swap size 11 and insertion sort swap size 6. In this way we proof insertion sort algorithm is faster than quicksort algorithm. And so best case complexity analysis insertion sort O(n) quick sort O(nlogn)

④ We use selection algorithm for finds the median of an unsorted array.

According to selection algorithm

→ Firstly we find the k^{th} smallest element in a list of n numbers

→ This number is called k^{th} order statics (We use order statics for unsorted array)

→ For $k=1$ or $k=n$ we can scan the list in question to find the smallest or biggest element

for $k=n/2$ this middle value called median

Time complexity is for kth smallest element $O(n \log n)$

We can divide elements into two subsets to find k^{th} smallest element

→ Less than or equal to some value p

→ greater than or equal to p

selection Problem solving

s → split position

p → pivot

If $s=k$ → pivot p solves the selection problem

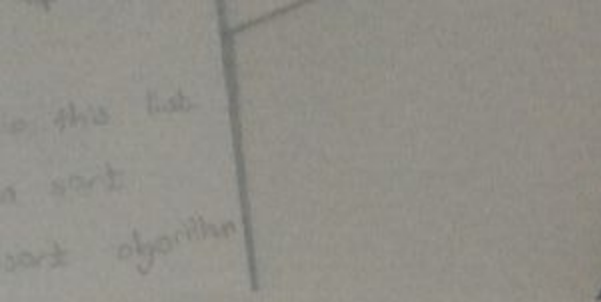
If $s>k$ → check k^{th} smallest element in the left part of the partitioned array

If $s<k$ → proceed by searching for the $(k-s)^{th}$ smallest element in its right part

$$\text{median}[A[1..n]] = \begin{cases} \frac{n}{2} & \text{if } n \text{ is odd} \\ \frac{n}{2} \text{ and } \frac{n}{2}+1 & \text{if } n \text{ is even} \end{cases}$$

In general $\left\lfloor \frac{(n+1)}{2} \right\rfloor$ is called the lower median and $\left\lceil \frac{(n+1)}{2} \right\rceil$ the upper median.

kth smallest element algorithm follows the decrease and conquer approach, as the strategy is to split the problem into subproblems and then select the appropriate subproblem for finding solutions.



Complexity Analysis

Each time when the pivot element partition the array exactly into two halves, the recurrence equation is $T(n) = T(n/2) + n$, where n is required for separating the n elements into two sets. This $T(n) = O(n)$, which is the best case.

However, in the worst case, the pivot element will be either the largest or the smallest element in the set. Thus, the array is partitioned into only one set of size $n-1$. Hence the recurrence equation is, $T(n) = T(n-1) + n$, which is $O(n^2)$

Average case is linear. Algorithm always works in linear time. We have discovered. Partioning based algorithm solves more general problem identifies the k smallest and $n-k$ largest elements of a given list, not just the value of its k^{th} smallest element

(3) We can solve this problem like knapsack problem because this is just similar knapsack problem knapsack can solved with exhaustive search approach

Problem is given n items of known weights $w_1, \ldots w_n$ and values $v_1 \ldots v_n$ and a knapsack of capacity W find the most valuable subset of the items that fit into the knapsack

Given an integer W which represents knapsack capacity, find out the maximum value subset of w[] such that multiplication the weights of this subset is bigger than or equal to $3^{*}(w)$

Optimal substructure

① maximum value obtained by n-1 items and W weight

② Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item

Exhaustive Search approach

→ Consider all the subsets of the set of n items given

→ Compute the multi weight of each subset in order to identify feasible subsets

→ Find a subset of the largest value among them

Analysis

of subsets of an n element set is 2^n

So exhaustive search leads to a $\Omega(2^n)$ algorithm