Algorithms and Data Structures SoSe 25 Solutions

Igor Dimitrov

2024-12-18

Table of contents

Preface

1 Sheet 1

1.1 Friday the 13th

Every now and then friday coincides with the 13th day of a month. How often does this happen?

First we define a function to determine whether a year is a leap year:

```
def is_leap(year) -> bool :
    """output true iff year is a leap year
    precondition: year >= 0

    >>> is_leap(1999)
    False
    >>> is_leap(2000)
    True
    >>> is_leap(1900)
    False
    """
    return year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)

print(
    is_leap(2000),
    is_leap(1999),
    is_leap(1900))
```

```
True False False
```

Next we write a function that converts the day of a year to the day of the month that this day would fall on. This function takes leap years into account:

```
def convert_day(year_day, isLeap = False) :
    """convert a year day to a month day
```

```
    precondition: year_day in [1...365] for default call
    precondition: year_day in [1...366] for leap year call, with isLeap = True
    for a given year day year_day returns the day of the month the day falls on
    taking into account if a year is a leap year with the isLeap parameter

    >>> convert_day(1)
    1 # 1st day of a year is the 1st day of January
    >>> convert_day(255)
    12 # 255th day of a year is the 12th day of August
    >>> convert_day(60, True)
    29 # 60th day of a leap year is the 29th day of February
    >>> convert_day(366, True)
    31 # the last of a leap year is the 31th December
    """

    # Days in each month (non-leap year)
    months = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    if isLeap : months[1] = 29

    i = 0 # month number starting with 0
    s = 0 # accumulates the sum of the days in months
    # invariant: s == sum(months[0..i-1]) and day > s
    while year_day > s + months[i] :
        s += months[i]
        i += 1
    # sum(months[0..i-1]) == s < day <= s + months[i] == sum(months[0..i])
    return year_day - s
```

We test the function for a few days, including

- 1st day of a non-leap year: January 1st
- 255th day of a non-leap year: September 12th
- 366th day of a leap year: December 31st
- 60th day of a leap year: February 29th

```
print(
    convert_day(1),
```

```
    convert_day(255),
    convert_day(366, True),
    convert_day(60, True), sep = '\n'
)
```

```
1
12
31
29
```

Next we compute the days of the year that fall on the 13th day of a month, both for a leap and a non-leap year.
We will use this information in the next function.

```
thirteenth_days = []
for i in range(1, 366) :
    if convert_day(i) == 13 : thirteenth_days.append(i)

thirteenth_days

thirteenth_days_leap = []
for i in range(1, 367) :
    if convert_day(i, True) == 13 : thirteenth_days_leap.append(i)

print(thirteenth_days,
    thirteenth_days_leap, sep = '\n')
```

```
[13, 44, 72, 103, 133, 164, 194, 225, 256, 286, 317, 347]
[13, 44, 73, 104, 134, 165, 195, 226, 257, 287, 318, 348]
```

Now we define a function that accepts a parameter that stands for the day of the week, ranging from 0 to 6, representing the days Mo, …, Sun.

This input parameter is the day of the week that the first 13th day of the month (i.e. January 13th) fell on.
E.g. if January 13th was Wednesday that year, then the input parameter is 2. If the year is a leap year a second default boolean parameter is provided as `True`.

```
def days_of_the_week_that_are_thirteenth(i, isLeap = False) :
    """return an array that contains the days of the week
    that are the thirteenth day of the month

    days are numbered from 0 to 6 corresponding to days Mon .. Sun
    input parameter i is the number of the first 13th day. It can
    be any day of the week. E.g. if in a given year Januaray 13th was Monday
    then i is 0.

    """

    thirteenth_days = [13, 44, 72, 103, 133, 164, 194, 225, 256, 286, 317, 347]
    if isLeap :
        thirteenth_days = [13, 44, 73, 104, 134, 165, 195, 226, 257, 287, 318, 348]



    b = [None] * len(thirteenth_days)
    b[0] = i
    for j in range(1, len(thirteenth_days)) :
        b[j] = (b[j - 1] + (thirteenth_days[j] - thirteenth_days[j - 1])) % 7
    return b


print(days_of_the_week_that_are_thirteenth(1),
    days_of_the_week_that_are_thirteenth(3, True), sep = '\n')
```

```
[1, 4, 4, 0, 2, 5, 0, 3, 6, 1, 4, 6]
[3, 6, 0, 3, 5, 1, 3, 6, 2, 4, 0, 2]
```

Note that January 13th can be any day of the week from Monday to Sunday for a given year. This function computes the days of the week all the 13th days of a month fall on given the day of the week 13th January falls on that year. E.g. for the year 1993, January 13th was a Wednesday. So the input parameter is simply the number 2.

All the other 13th days of other months are computed that as follows with this function:

```
print(days_of_the_week_that_are_thirteenth(2, is_leap(1993)))
```

```
[2, 5, 5, 1, 3, 6, 1, 4, 0, 2, 5, 0]
```

From this result we see that the year 1993 had only one Friday the 13th - on August.

Next we define another function that computes the first day of any year after (and including) 1 A.D.

It is assumed that the first day of the first year was a Monday. The first years of all other years are computed accordingly and taking leap years into account:

```
def first_day_of_year(year) :
    """return the first day of a year given as input parameter
    where days are numbered 0..6 starting from monday
    precondition: year >= 1


    >>> first_day_of_year(1999)

    """
    days_past = 0
    for i in range(1, year) :
        if is_leap(i) : days_past += 366
        else : days_past += 365
    return days_past % 7
```

Let's test this function to find out the what day of the week the first days of the first 5 years fell on:

```
for i in range(1, 6) : print(first_day_of_year(i))
```

```
0
1
2
3
5
```

First day of year 1993:

```
first_day_of_year(1993)
```

```
4
```

Now we are ready to generate all the 13th days of all months of years ranging from 1993 to 2025.

```
thirteenth_days = []
for y in range(1993, 2026) :
    first_day = first_day_of_year(y)
    first_thirteenth_day = (first_day + 12) % 7
    thirteenth_days.append(days_of_the_week_that_are_thirteenth(first_thirteenth_day, is_lea
```

`thirteenth_days` holds 33 arrays, each holding 12 values, that contains the days of the week that the 13th of the corresponding month fell on:

```
print(thirteenth_days)
```

```
[[2, 5, 5, 1, 3, 6, 1, 4, 0, 2, 5, 0], [3, 6, 6, 2, 4, 0, 2, 5, 1, 3, 6, 1], [4, 0, 0, 3, 5,
```

To find out how many times 'Friday the 13th' was experienced since 1993, we simply have to count and add iup each occurrence of '4' (Friday) in each array:

```
count = 0
for days in thirteenth_days :
    for day in days :
        if day == 4 : count += 1

print(count)
```

```
55
```

So Friday the 13th was experienced 55 times in total since year 1993. Finally, we can encapsulate all of this computation in a single function:

```
def count_friday_13th_since(year) :
    thirteenth_days = []
    for y in range(year, 2026) :
        first_day = first_day_of_year(y)
        first_thirteenth_day = (first_day + 12) % 7
        thirteenth_days.append(
            days_of_the_week_that_are_thirteenth(first_thirteenth_day, is_leap(y)))
    count = 0
    for days in thirteenth_days :
```

```
        for day in days :
            if day == 4 : count += 1
    return count


count_friday_13th_since(2000)
```

```
44
```

1.2 Stable Glasses

a) *Linear search*: **30** tries in worst case, corresponding to the situation when the glass breaks at the highest platform. We start at the bottom platfrom and move to the next higher one if the glass doesn't break. As soon as the glass breaks we know that the platform just below is the maximum height the glass can withstand..

b) *Binary search*: **5** tries always. In worst case each time we try the glass breaks. Therefore, maximum amount of broken glasses is **5**. This corresponds to the situation where no height is safe for the glass.

Justification: The general strategy is the same as with binary search - we first try the middle platform, i.e. 15th platform. In case the glass doesn't break we know that we must search only in the upper half. Otherwise, we must search only in the lower half. Since in each step we half the amount of platforms the procedure is guaranteed to terminate in approximately $\log_2 30$ steps. How many exactly can be seen with the following sequence: 15, 8, 4, 2, 1. Explanation of this sequence is that the middle platform where we have to perform the experiment is determined by the formula: `(total platforms left + 1) // 2`. And the total platforms left in the next step is always the number of the middle platform from the previous step. So in total there will be 5 steps and each time a glass will break.

c) In this case we can determine the highest platform in **11** steps at worst case with the following procedure (by dividing the stand in 3 equal parts)

1. Test first glass in the highest platform in the lower third, i.e. the 11th platform. If it breaks then we must test the other glass in the remaining 10 platforms in the lower third linearly until it breaks the first time: $10 + 1 = \mathbf{11}$ tries at worst.
2. if the first glass doesn't break, then we try the lowest platform in the upper third, i.e. the 21st platform with this glass. If the glass breaks here, then we search the middle third, i.e. we test the platforms 12..20 with the other glass linearly, until it breaks. This totals to $1 + 1 + 9 = \mathbf{11}$ tries in worst case.

3. If first glass doesn't break in the 21st platform, then we continue to search the upper third platform with it, namely the platforms 22...30, linearly until the glass breaks or we reach the end. This totals to $1 + 1 + 9 = \mathbf{11}$ tries in worst case.

1.3 Python Documentation

Chapters 3, 4.1 - 4.9.2, 8.1 - 8.4, 9.3, and 10.6

```
def reverse(s) :
    if len(s) == 0 : return s
    return s[-1] + reverse(s[: len(s) - 1])
```

```
reverse('he')
```

```
'eh'
```

```
def palindrom(s) -> bool :
    return s == reverse(s)
```

```
palindrom('ccababaccd')
```

```
False
```

```
def fib(limit):
    a, b = 0, 1
    #
    # there exist i: a == fib(i), b == fib(i + 1)
    while (a < limit) :
        a, b = b, a + b
    # a >= limit
    return(a)
```

```
fib(144)
```

```
144
```

```
d = {"a" : "active", "b" : "inactive", "c" : "active"}

for i, j in d.items() :
    print(i, j)

for i, j in d.copy().items() :
    if j == 'inactive' : del d[i]

d2 = {}
d2["igor"] = 31
d2["igor"] += 1
d2
```

```
a active
b inactive
c active

{'igor': 32}
```

```
list(range(5, 10))
list(range(0, 10, 3))
list(range(-10, -100, -30))

a = ['mary', 'had', 'a', 'little', 'lamb']
b = []
for i in range(len(a)):
    b.append((i, a[i]))

b
d = dict(b)
d

for i, word in enumerate(a) :
    print(f"{i}: {a[i]}" + i**2*"!")
```

```
0: mary
1: had!
2: a!!!!
3: little!!!!!!!!!
4: lamb!!!!!!!!!!!!!!!!
```

```
dict(enumerate(list(range(5))))
```

```
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4}
```

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(f"{n} equals {x} * {n // x}")
            break
    else: # loop fell through withouit finding a factor
        print(n, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

```
for num in range(2, 10):
    if num % 2 == 0:
        print(f"Found an even number {num}")
        continue
    print(f"Found an odd number {num}")
```

```
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

```
def http_error(status):
    match status:
        case 400: return "Bad Request"
        case 404: return "Not found"
        case 418: return "I'm a teapot"
        case 401 | 403 | 402: return "Not allowed"
        case _: return "Something's wrong"

for i in [400, 404, 418, 134] : print(http_error(i))

for i in ['a', 'b']: print(i)

print(http_error(401))
```

```
Bad Request
Not found
I'm a teapot
Something's wrong
a
b
Not allowed
```

```
nums = [1, 2, 3]
a, b, c = nums
print(a, b, c)

def add(a, b, c) :
    return a + b + c

add(1, 2, 3)
add(*nums)

a, *b, c = [1, 2, 3, 4]
print(a, b, c)
a, *b, c = 1, 2, 3, 4
print(a, b, c)
```

```
1 2 3
1 [2, 3] 4
1 [2, 3] 4
```

```
x, y = 1, 2
print(x, y)
x, y = [1, 2]
print(x, y)
x, y = (1, 2)
print(x, y)

*head, last = 1, 2, 3, 4
print(head, last)
*head, last = [1, 2, 3, 4]
print(head, last)
first, *middle, last = (1, 2, 3, 4, 5)
print(first, middle, last)
x,*_, z = 1, 2, 3, 4
print(x, z)
a, *b, c, d = range(10)
print(a, b, c, d)
```

```
1 2
1 2
1 2
[1, 2, 3] 4
[1, 2, 3] 4
1 [2, 3, 4] 5
1 4
0 [1, 2, 3, 4, 5, 6, 7] 8 9
```

** operator works with dictionaries to unpack keyword arguments

```
def greet(name, greeting):
    print(f"{greeting}, {name}!")

params = {"name": "Alice", "greeting": "Hello"}

greet(**params)
```

```
Hello, Alice!
```

```python
a, *b, c = 1,2
# print(a, b, c)

def foo(a, *args, **kwargs):
    print(a)
    print(args)
    print(kwargs)

foo(1, 2, 3, 4, x = 10, y = 20, z = 30)
```

```
1
(2, 3, 4)
{'x': 10, 'y': 20, 'z': 30}
```

```python
def bar(*args):
    print(args)


l = [1, 2, 3]

bar(l, *l)
```

```
([1, 2, 3], 1, 2, 3)
```

2 Sheet 2

```
import random
```

2.1 Properties of Sorting Algorithms

a) `:=` means 'defined as':

- `a < b := not ( b <= a)`
- `a > b := b < a` (utilize the definition above)
- `a >= b := b <= a`
- `a == b := (a <= b) and (b <= a)`
- `a != b := not (a == b)` (again utlizing the previous definiton)

b) The following $N-1$ tests are sufficient:

```
1) a[0] <= a[1],
2) a[1] <= a[2],
...
N-1) a[N-2] <= a[N-1]
```

Proof: assume the previous $N-1$ have passed. Let $i \in [0..N-2]$ and $j \in [i+1..N-1]$ be arbitrary indices.

Since the ordering is total, transitivity holds. Specifically having `a[i] <= a[i + 1]` and `a[i + 1] <= a[i + 2]` we can deduce by transitivity that `a[i] <= a[i + 2]`. Now having this result and the condition that `a[i+2] <= a[i+3]` we can again deduce by transitivity that `a[i] <= a[i + 3]`

Applying such a transitivity chain $j-i$ times we will obtain `a[i] <= a[j]`. Since i and j were arbitrary, s.t. $i < j$, this concludes our proof.

c) The usual versions of `insertion_sort()` and `quick_sort()`, without counting are given as:

```
def insertion_sort(a) :
   N = len(a)
   # i = 0
   # sorted(a[0..i])
   for i in range(N):
      k = i + 1
      while k > 0 :
         if a[k] < a[k-1]: a[k], a[k-1] = a[k-1], a[k]
         else: break
         k -= 1

# partition function for quicksort
def partition(a, l, r):  # l, r are left and right boundaries (inclusive) of a
   p = random.randint(l, r) # choose a random index
   a[p], a[r] = a[r], a[p] # exchange the right-most element with the one at random index
   pivot = a[r]
   i = l
   k = r - 1
   while True:
      while i < r and a[i] <= pivot: i += 1 # increment until found a misplaced element
      while k > l and a[k] >= pivot: k -= 1 # decrement until found a misplaced element
      if i < k : a[i], a[k] = a[k], a[i]
      else : break
   a[i], a[r] = a[r], a[i] # bring pivot to the correct position
   return i # so that recursive calls now where the partitions are

# implementation of quicksort with arbitrary array boundaries
def quick_sort_impl(a, l, r):
   if r <= l: return # no return argument since in-place
   i = partition(a, l, r)
   quick_sort_impl(a, l, i-1)
   quick_sort_impl(a, i+1, r)

# the clean user interface
def quick_sort(a) :
   quick_sort_impl(a, 0, len(a) - 1)
```

We modify them slightly to return the number of comparisons:

```
def insertion_sort(a) :
    N = len(a)
```

```
    count = 0
    # i = 1
    # sorted(a[0..i-1])
    for i in range(1, N):
        k = i
        while k > 0 :
            if a[k] < a[k-1]:
                a[k], a[k-1] = a[k-1], a[k]
                count += 1
            else:
                count += 1
                break
            k -= 1
    return count
```

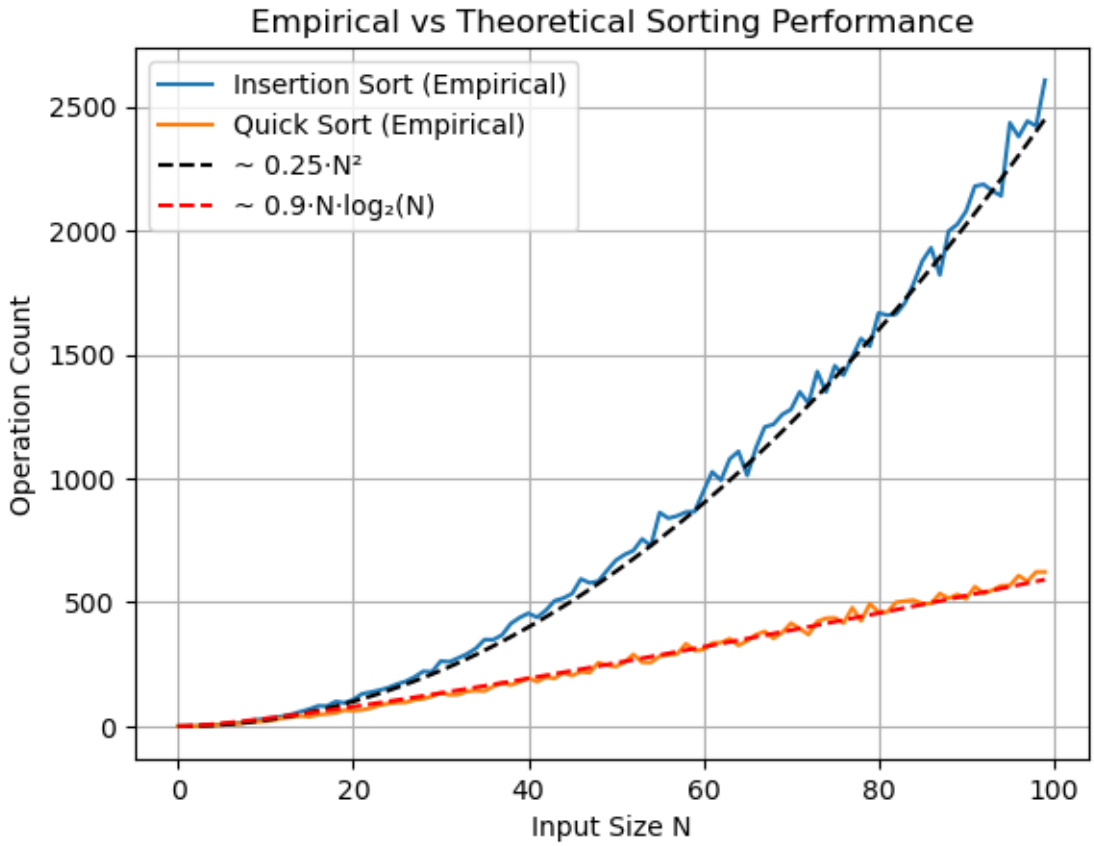
```
def partition(a, l, r):  # l, r are left and right boundaries (inclusive) of a
    count = 0
    p = random.randint(l, r)
    a[p], a[r] = a[r], a[p]
    pivot = a[r]
    i = l
    k = r - 1
    while True:
        while i < r and a[i] <= pivot:
            i += 1 # increment until found a misplaced element
            count += 1
        while k > l and a[k] >= pivot:
            k -= 1 # decrement until found a misplaced element
            count += 1
        if i < k : a[i], a[k] = a[k], a[i]
        else : break
    a[i], a[r] = a[r], a[i] # bring pivot to the correct position
    return i, count # so that recursive calls now where the partitions are

# implementation of quicksort with arbitrary array boundaries
def quick_sort_impl(a, l, r):
    if r <= l: return 0# no return argument since in-place
    i, count = partition(a, l, r)
    count += quick_sort_impl(a, l, i-1)
    count += quick_sort_impl(a, i+1, r)
    return count
```

```
# the clean user interface
def quick_sort(a) :
    return quick_sort_impl(a, 0, len(a) - 1)
```

Now we write a simple script that aggregates the counts of sorts in lists, and subsequently plot the lists:

```
import matplotlib.pyplot as plt
import numpy as np

def create_counts(sorting_function, shuffle = True, MAX = 100):
    counts_array = []
    for N in range(MAX):
        count = 0
        for i in range(5):
            a = list(range(N))
            if shuffle :
                random.shuffle(a)
            count += sorting_function(a)
        count /= 5
        counts_array.append(count)
    return counts_array

insertion_counts = create_counts(insertion_sort)

quick_counts = create_counts(quick_sort)

# insertion_sorted_counts = create_counts(insertion_sort, False)

# --- Prepare input sizes ---
N_values = np.arange(100)

# --- Theoretical curves with estimated constants ---
d1, e1, f1 = 0.25, 0, 0  # Insertion Sort: Quadratic
d2, e2, f2 = 0.9, 0, 0   # Quick Sort: N log N

insertion_theoretical = d1 * N_values**2 + e1 * N_values + f1
quick_theoretical = d2 * N_values * np.log2(N_values + 1) + e2 * N_values + f2

# --- Plot everything ---
plt.plot(insertion_counts, label="Insertion Sort (Empirical)")
plt.plot(quick_counts, label="Quick Sort (Empirical)")
```

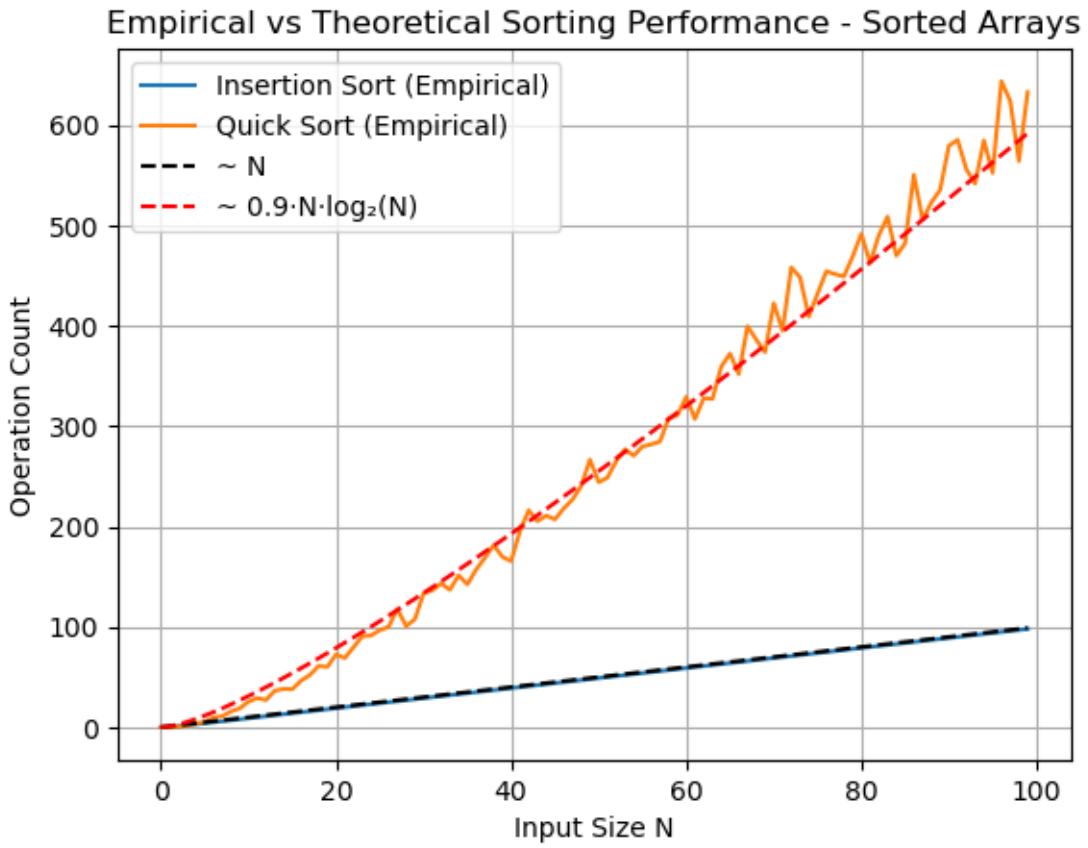
```
# plt.plot(insertion_sorted_counts, label="Insertion Sort (Sorted Input)")

plt.plot(N_values, insertion_theoretical, 'k--', label="~ 0.25·N²")  # Dashed black
plt.plot(N_values, quick_theoretical, 'r--', label="~ 0.9·N·log(N)")  # Dashed red

plt.xlabel("Input Size N")
plt.ylabel("Operation Count")
plt.title("Empirical vs Theoretical Sorting Performance")
plt.legend()
plt.grid(True)
plt.show()
```



We see that insertion sort is well approximated by the function $0.25 \cdot N^2$ and quick sort by $0.9 \cdot N \cdot \log_2 N$, confirming our theoretical expectations.

Next we do the same for already sorted arrays:

```
insertion_counts = create_counts(insertion_sort, False)
quick_counts = create_counts(quick_sort, False)

# --- Prepare input sizes ---
N_values = np.arange(100)

# --- Theoretical curves with estimated constants ---
d1, e1 = 1, 0  # Insertion Sort: Quadratic
d2, e2, f2 = 0.9, 0, 0   # Quick Sort: N log N

# insertion_theoretical = d1 * N_values**2 + e1 * N_values + f1
insertion_theoretical = d1 * N_values + e1
quick_theoretical = d2 * N_values * np.log2(N_values + 1) + e2 * N_values + f2

# --- Plot everything ---
plt.plot(insertion_counts, label="Insertion Sort (Empirical)")
plt.plot(quick_counts, label="Quick Sort (Empirical)")
# plt.plot(insertion_sorted_counts, label="Insertion Sort (Sorted Input)")

plt.plot(N_values, insertion_theoretical, 'k--', label="~ N")  # Dashed black
plt.plot(N_values, quick_theoretical, 'r--', label="~ 0.9·N·log(N)")  # Dashed red

plt.xlabel("Input Size N")
plt.ylabel("Operation Count")
plt.title("Empirical vs Theoretical Sorting Performance - Sorted Arrays")
plt.legend()
plt.grid(True)
plt.show()
```



Here we see that insertion sort is linear while quicksort is still $\mathcal{O}N\log(N)$, approximated by the same function from previous output. Thus insertion sort outperforms quicksort for already sorted arrays.

Next we compare how quicksort works on already sorted arrays, if the pivot is not randomly chosen, but always chosen as the right-most element - the naive version.

We first define this naive quick sort version:

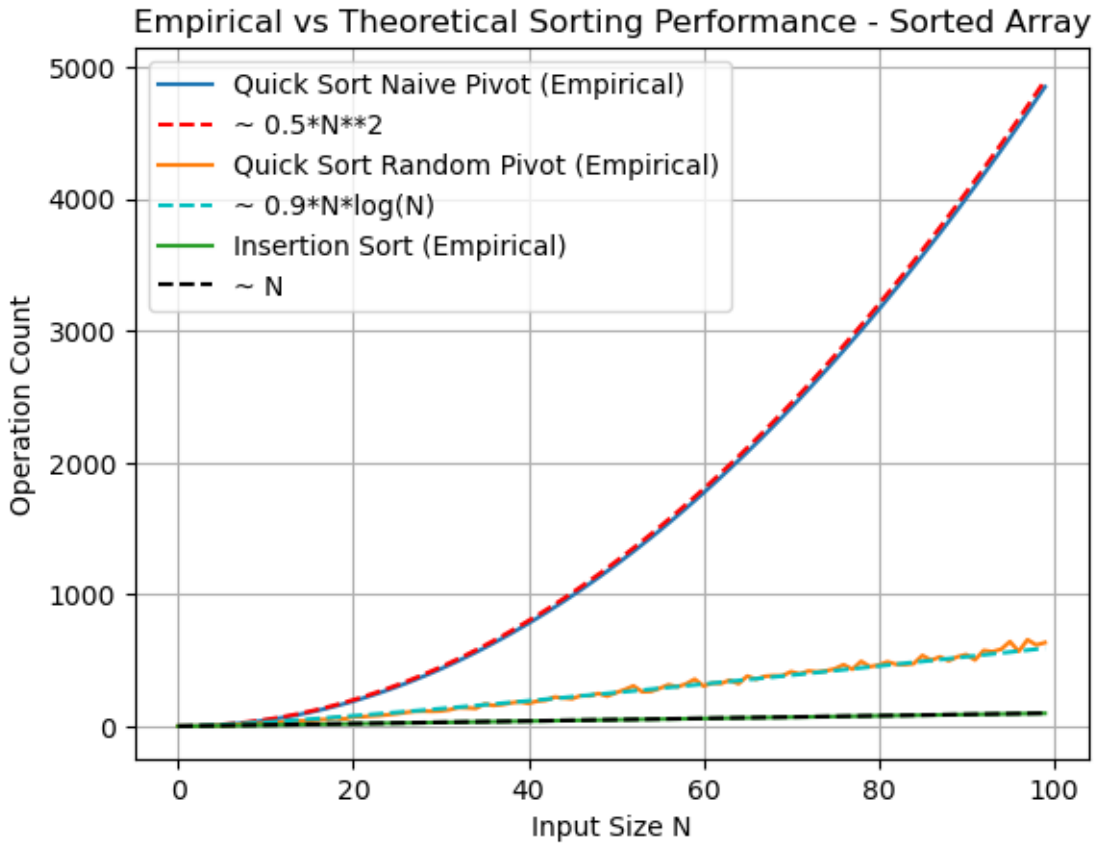
```
def partition2(a, l, r):  # l, r are left and right boundaries (inclusive) of a
    count = 0
    pivot = a[r]
    i = l
    k = r - 1
    while True:
        while i < r and a[i] <= pivot:
            i += 1 # increment until found a misplaced element
            count += 1
        while k > l and a[k] >= pivot:
```

```
            k -= 1 # decrement until found a misplaced element
            count += 1
        if i < k : a[i], a[k] = a[k], a[i]
        else : break
    a[i], a[r] = a[r], a[i] # bring pivot to the correct position
    return i, count # so that recursive calls now where the partitions are

# implementation of quicksort with arbitrary array boundaries
def quick_sort_impl2(a, l, r):
    if r <= l: return 0# no return argument since in-place
    i, count = partition2(a, l, r)
    count += quick_sort_impl2(a, l, i-1)
    count += quick_sort_impl2(a, i+1, r)
    return count

# the clean user interface
def quick_sort2(a) :
    return quick_sort_impl2(a, 0, len(a) - 1)
```

Next we perform the same experiments from before and plot the results:

```
insertion_counts = create_counts(insertion_sort, False)

quick2_counts = create_counts(quick_sort2, False)

quick_counts = create_counts(quick_sort, False)

# --- Prepare input sizes ---
N_values = np.arange(100)

# --- Theoretical curves with estimated constants ---
d1, e1, f1 = 0.5, 0, 0  # Quick Sort Naive Pivot: Quadratic
d2, e2, f2 = 0.9, 0, 0   # Quick Sort: N log N
d3, e3 = 1, 0 # Insetion sort: linear

quick2_theoretical = d1 * N_values**2 + e1 * N_values + f1
quick_theoretical = d2 * N_values * np.log2(N_values + 1) + e2 * N_values + f2
insertion_theoretical = d3 * N_values + e3

# --- Plot everything ---
plt.plot(quick2_counts, label="Quick Sort Naive Pivot (Empirical)")
plt.plot(N_values, quick2_theoretical, 'r--', label="~ 0.5*N**2")  # Dashed red
```

```
plt.plot(quick_counts, label="Quick Sort Random Pivot (Empirical)")
plt.plot(N_values, quick_theoretical, 'c--', label="~ 0.9*N*log(N)")  # Dashed red

plt.plot(insertion_counts, label="Insertion Sort (Empirical)")
plt.plot(N_values, insertion_theoretical, 'k--', label="~ N")  # Dashed black

plt.xlabel("Input Size N")
plt.ylabel("Operation Count")
plt.title("Empirical vs Theoretical Sorting Performance - Sorted Array")
plt.legend()
plt.grid(True)
plt.show()
```



Here we see that quicksort with naive pivot choice runs quadratically for already sorted arrays, while insetion sort is linear. Thus insertion sort outperforms naive quicksort in this case. (random pivot quicksort is still $\mathcal{O}(N \log N)$)

d)

3 Blatt 03

3.1 Problem 1

We use a slightly different, more convenient method based on Hoare calculus and the program derivation methodoloy of Dijkstra. In this methodology pre- and postconditions, as well as loop invariants are provided directly within the program text as comments.

A commment before a command is a logical statement about the state-space of the program before the execution of the command, i.e. the precondition, and the comment after is the postcondition.

The comment before a while or a for loop is specifically labeled as 'invariant' and is the invariant statement that stays true after the execution of the loop. The general form of such specification is:

```
# P
C
# Q
```

This expression means: if P holds before the execution of `C` then Q will hold after the execution of C. For example:

```
# x == 0
x += 1
# x == 1
```

or

```
# y > 0
y -= 1
# y >= 0
```

or

```
# x == X and y == Y
x += y
# x == X + Y and y == Y
y = x - y
# x == X + Y and Y == X
x -= y
# x == X and Y == X
```

or more compactly

```
# x == X and y == Y
x += y
y =  x - y
x = x - y
# x == Y and y == X
```

In the above program we exchnaged the values of variables `x`, and `y` without using a temporary varible `t`. But as a furhter example of pre- and postcondition specification let's do that:

```
# x == X and y == Y
t = x
# t == X
x = y
# x == Y
y = t
# y == X
```

The semantics of while loops is given as follows:

if

```
# P and Q
C
# P
```

Where C is an arbitrary command, or a sequence of commands.

Then:

```
# invariant: P
while Q :
    C
# P and !Q
```

So what we have to show is that:

1) P holds before the execution of the loop, before C is executed (induction base)
2) if Q holds additionally then P still holds after C is executed (inductive step)

Having shown 1) and 2) we can conclude:

1) After exiting the loop (P and !Q) holds

A simple example: incrementing a variable i up to a value `N`, where it is assumed that N >= 0

```
i = 0
# invariant: i <= N (since we know that 0 <= N)
while i < N:
    i += 1
# i >= N
```

1) **IB**: Before the loop executes: `i == 0 <= N`, by assumption.
2) **IS**: if `i <= N and i < N` simply implies `i < N`. if `i < N` holds before incrementation, then `i <= N` will hold after the incrementation, by simple arithmetic rules, which proves the invariant.

So, after exiting the loop the following holds:

3) `i <= N and !(i < N)`. which is exactly equivalent to `i == N`, what

Let's apply this scheme to the division example:

```
def div(x, y):
    # precondition: x >= 0, y > 0
    q = 0
    r = x
    # invariant: x == r + q * y and r >= 0
    while r >= y:
        r -= y
        q += 1
    # r < y
    return (q, r)

print(f"31/9 == {div(31, 9)[1]} + 9 * {div(31,9)[0]}")
```

```
31/9 == 4 + 9 * 3
```

Proof:

1) **IB**: before the loop executes we have; `q == 0 and 0 <= x == r` therefore `x == r + q * y and r >= 0` holds trivially.

2) **IS**: assume the invariant holds before some arbitrary execution of the loop. Assume that `r >= y` additionally holds, so we are in the body of the loop. Now since the invariant holds we have

   ```
   x == r + q * y and r >= 0 and r >= y
   ```

   we can manipulate `r + q * y` as follows:

   ```
   r + q * y == (r - y) + (q + 1) * y
   ```

   So after executing the body of the loop the `r` holds what previously was `r - y` and `q` holds what previously was `q + 1`. But as shown above this leaves the value of expression unchanged, therefore after updating the variables `r` and `q` in the body of the loop the equality `x == r + q * y` still holds.

   Now we turn our attention to the second conjunct of the invariant: `r >= 0`. Since `r >= y` holds in the body of the loop, `r >= 0` still holds after the update to r - `r -= y`.

   These two observations conclude our proof

Now after exiting the loop in addition to the invariant the negation of the loop condition holds. Thus we have:

3) `x == r + q * r and r >= 0 and !(r >= y)` iff `x == r + q * r and 0 <= r < y`

Which is exactly satisfies the condition of whole number division.

Below we give the trace of the execution of the algorith for `x == 31` and `y == 9` with a short explanation of Hoare-style secification:

Hoare Logic Specification for Integer Division

Problem: Implement integer division with remainder using repeated subtraction.

Function:

```
def div(x, y):
    # Precondition: x   0   y > 0
    q = 0
    r = x
    # Invariant: x == r + q * y   r   0
    while r >= y:
        r = r - y
        q = q + 1
    # Postcondition: x == q * y + r   0   r < y
    return (q, r)
```

Hoare Triple:

$$\{\, x \geq 0 \land y > 0 \,\}\texttt{div(x, y)}\{\, x = q \cdot y + r \land 0 \leq r < y \,\}$$

- **Invariant**: $x = r + q \cdot y \quad \land \quad r \geq 0$
- **Termination**: Each iteration decreases `r` by `y` while keeping `r  0` therefore the loop will eventually terminate.

Execution Trace: `div(31, 9)`

Initial values: `x = 31`, `y = 9`

q	r	Invariant: `r + y * q == x  r  0`	Loop condition (`r  y`)
0	31	$31 + 9 * 0 = 31 \quad 31 \quad 0 \rightarrow$ true	$31 \quad 9 \rightarrow$ true
1	22	$22 + 9 * 1 = 31 \quad 22 \quad 0 \rightarrow$ true	$22 \quad 9 \rightarrow$ true
2	13	$13 + 9 * 2 = 31 \quad 13 \quad 0 \rightarrow$ true	$13 \quad 9 \rightarrow$ true
3	4	$4 + 9 * 3 = 31 \quad 4 \quad 0 \rightarrow$ true	$4 \quad 9 \rightarrow$ false

- The loop terminates when `r < y`, here `r = 4 < 9`.
- Final return: `(q, r) = (3, 4)`
- Confirming: $31 = 9 \cdot 3 + 4$

This demonstrates **partial correctness** (postcondition holds if it terminates), and the **termination argument** ensures **total correctness**.