Algorithms and Data Structures (ECS529)

Nikos Tzevelekos

Lecture 2

Sorting, Running Time and Complexity

Quiz time (get ready!)

`go to http:://kahoot.it`

Quiz time (get ready!)

About Python:

- I am fluent in it
- I can decently code in it
- I have only heard of it
- all I only know is there is a big reptile with that name

Suppose array A is sorted and has size 16. If we do **binary** search on it, how many of its elements do we need to compare with our k in the worst case?

- 0
- 1
- 5
- 16

Suppose array A is sorted and has size 16. If we do **linear** search on it, how many of its elements do we need to compare with our k in the worst case?

- 0
- 1
- 5
- 16

What is a better algorithm for putting a deck of cards in increasing order?

- we keep randomly mixing the cards and hope for the best
- we find the smallest card, start a new pile with it, then the second smallest, etc.
- we pick the first card and start a new pile with it, then add each other card in the pile in its ordered position
- we buy a new deck of cards

Sorting algorithms

Every algorithms course continues with sorting. *– Anonymous*

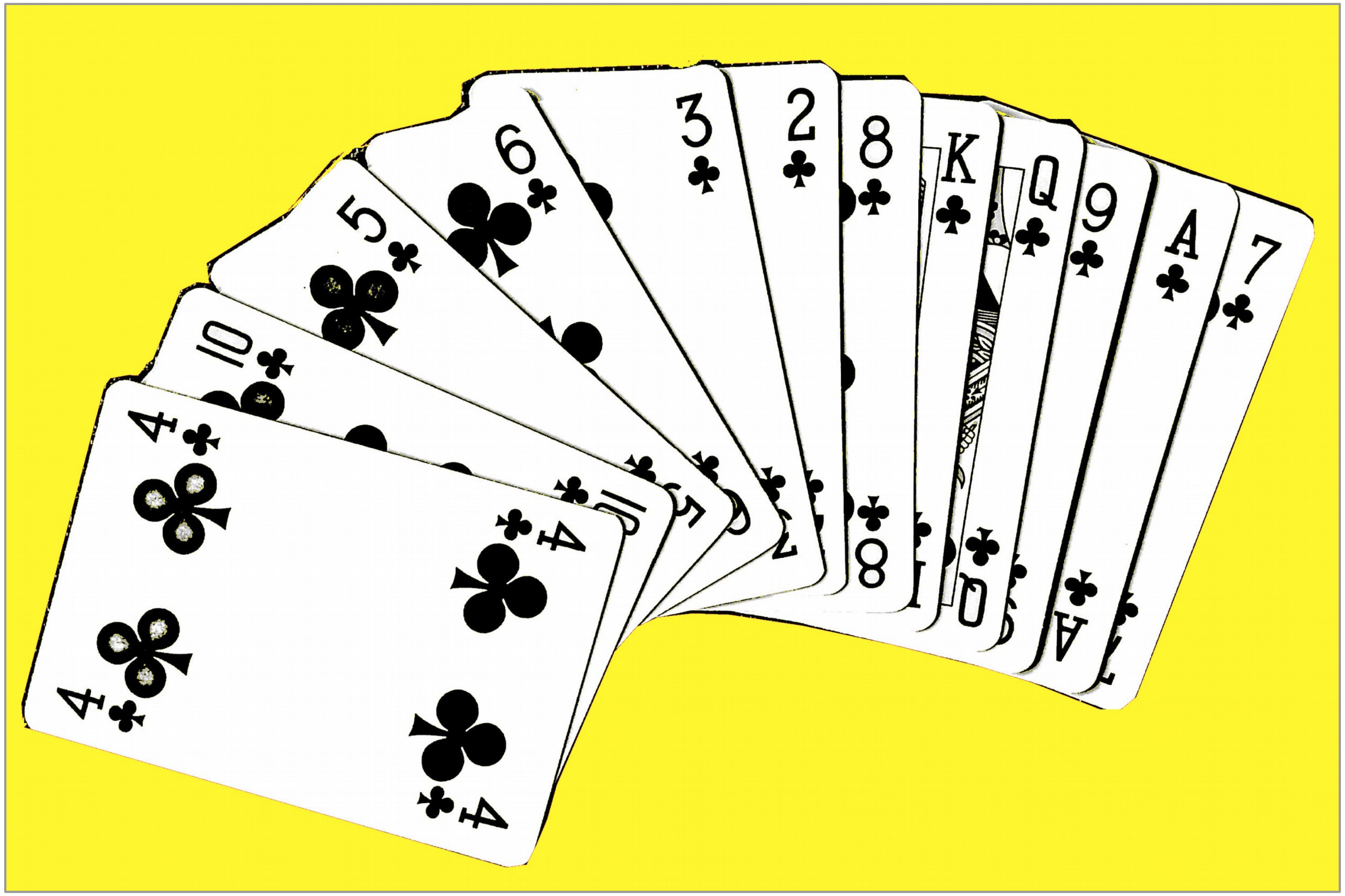
SORTING is the following problem:

- given an array A of integers
- put the elements of A in increasing order

Notes:

- put in order means: re-arrange the elements of A so that they are in increasing order
- the array A can be arbitrarily large
- in general, SORTING is about arrays of elements of any type (not necessarily integers), so long as they can be ordered

Sort a stack cards



How would you put a stack of cards in order? (in a *systematic* way)

Selection sort

To sort array A:

- go through all its elements, find the smallest one, and put it in first position
- go through all its remaining elements, find the smallest one, and put it in second position
- and so on

Some notes:

- Suppose the smallest element in A is $A[42]$. To bring it in first position, we simply **swap** $A[0]$ and $A[42]$.
- At the end of the i-th step of our algorithm, the smallest i elements of A are in the beginning of A, and they are in order.

Selection sort formally

```
def selectionSort(A):
    for i in range(len(A)):
        imin = findMin(i,A)
        swap(i,imin,A)


def findMin(i, A):
    imin = i
    for j in range(i+1,len(A)):
        if A[j] < A[imin]:
            imin = j
    return imin


def swap(i, j, A):
    tmp = A[i]
    A[i] = A[j]
    A[j] = tmp
```

Selection sort formally

```
def selectionSort(A):
    for i in range(len(A)):
        imin = findMin(i,A)
        swap(i,imin,A)


def findMin(i, A):
    imin = i
    for j in range(i+1,len(A)):
        if A[j] < A[imin]:
            imin = j
    return imin


def swap(i, j, A):
    tmp = A[i]
    A[i] = A[j]
    A[j] = tmp
```

We let `i` range over all positions in `A` (i.e. from `0` to `len(A)-1`)

- in each repetition of the loop, we find the index `imin` that contains the smallest element in `A[i],A[i+1],...`
- we then bring `A[imin]` in the `i`-th position by swapping `A[i]` and `A[imin]`

`findMin(i,A)` finds the smallest element in `A[i],A[i+1],...`

it goes through all of them and stores in `imin` the position of the smallest seen so far

Insertion sort

To sort array A:

- pick $A[1]$ and insert it before/after $A[0]$
- pick $A[2]$ and insert it in its right position in $A[0]...A[2]$
- pick $A[3]$ and insert it in its right position in $A[0]...A[3]$
- and so on

At each step, we pick an element $A[i]$ of A and we insert it in its place in the initial part $A[0]...A[i]$ of A, that is already sorted. For that:

- starting from $A[i\text{-}1]$, we move the elements of A to the right
- until we find the position where $A[i]$ needs to be inserted

Insertion sort formally

```
def insertionSort(A):
    for i in range(1,len(A)):
        insert(A[i],A,i)


def insert(k, A, hi):
    for i in range (hi,0,-1):
        if k >= A[i-1]:
            A[i] = k
            return
        A[i] = A[i-1]
    A[0] = k
```

Insertion sort formally

```
def insertionSort(A):
    for i in range(1,len(A)):
        insert(A[i],A,i)

def insert(k, A, hi):
    for i in range (hi,0,-1):
        if k >= A[i-1]:
            A[i] = k
            return
        A[i] = A[i-1]
    A[0] = k
```

We go through each element `A[i]` of `A`, starting from `A[1]`, and call `insert(A[i],A,i)` to insert it in its *ordered position* in `A[0]`…`A[i]`.

In `insert(k,A,hi)`:

- we pick each element `A[i-1]` of `A[0]`…`A[hi]`, and move it to the right, by setting `A[i]` to `A[i-1]`
- until we find the first `i` such that `k >= A[i-1]`, which means that the correct position for `k` is `i`
- we then set `A[i]` to `k`

- if we don't find such an `i`, then this means that `k` is smaller than all elements in `A[0]`…`A[i]`, so we put it in position `0`

Efficiency

Different algorithms can be designed for the same problem.

One way to distinguish “good algorithms” is based on their efficiency:

- between two algorithms for the same problem, we call **more efficient** the one that *on the same input* solves the problem faster
- e.g. a more efficient algorithm is one that does not make unnecessary computational steps

This is called **time complexity analysis**: it tells us whether an algorithm is efficient in time (high complexity = low efficiency)

Time Complexity

How can we systematically say how fast is an algorithm?

Instead of running experiments and timing the results, we can try to figure out how long our algorithm runs **for any given input**:

1. we do this by expressing the running time of the algorithm as a function of the **size of the input**
   - e.g. if the input has size n, the running time is some $f(n)$
2. we focus on **large values** of n
3. in the running time we count the number of **steps** that the algorithm performs

Time Complexity comments

1. why look at the running time as a function of the size of the input?
    - typically, the larger the input, the longer the running time
2. why focus on large values of n?
    - efficiency matters when n is large $-$ for small n, most problems are solved in a split second on any computer
3. what counts as a step?
    - we count basic operations like number comparisons, additions, divisions, etc., and assume that all take the same time.

      The reason is that if you run your algorithm on very large inputs, what really matters is not how long each operation takes, but how many times you perform this operation!

Example: search an ordered array

SORTED-SEARCH is the following problem:

- given a **sorted** array of integers A and an integer k
- if k is in A then return its position, otherwise return -1

For example, if the array is [8,12,16,20,25,28,30,47,63,99]

and the searched integer is 12, the answer should be 1.

There are two standard algorithms:

- Linear search
- Binary search

Linear (sorted) search

The simplest solution is:

- scan the array A from left to right
- until you find k (return the current position),
- or you find an element greater than k (return -1),
- or you reach the end of the array (return -1).

```
def sortedSearch(A, k):
   for i in range(len(A)):
      if A[i] == k:
         return i
      if A[i] > k:
         return -1
   return -1
```

Worst case running time

The only basic operations here are the comparisons (==, >).

If the array has 10 elements, how many comparisons do we need to make? (or, how many times do we do the for loop?)

- Best case: we find k straight away (it is first in the array), so we only need one comparison
- Worst case: we go on and search the whole array (k is not in the array), so we need 20 comparisons

We will mostly **look at the worst case** (optimistic, are we): we would need to make 20 comparisons.

In general, if the array A contains n many elements (for some number n), linear search can make $2n$ comparisons.

We say that the **running time** is $2n$ in the worst case.

Optimised linear (sorted) search

What about the following linear search algorithm:

```python
def sortedSearch2(A, k):
    for i in range(len(A)):
        if A[i] >= k:
            break
    if A != [] and A[i] == k:
        return i
    return -1
```

This is faster than our previous algorithm:

If array A contains n elements, the algorithm makes n+1 comparisons in the worst case.

So, running time is n+1 in the worst case.

But there is a way to do way better...

Binary search

We can look for k in array A in a more efficient way:

- We look at the middle element of A.
- If it is equal to k, we stop. Otherwise:
  - if it is greater than k, we repeat the search only on the elements before the middle one;
  - if it is less than k, we repeat the search only on the items above it.
- Return -1 when the range we search is of size 0.

This is called binary search.

Binary search code and running time

```
def binSearch(A, k):
    lo = 0
    hi = len(A)-1
    while (lo <= hi):
        mid = (lo+hi)//2
        if A[mid] == k:
            return mid
        else:
            if A[mid] < k:
                lo = mid+1
            else:
                hi = mid-1
    return -1
```

The basic operations (steps) here are arithmetic operations and comparisons (in yellow).

Suppose that, for some input, we take the while loop x times.

How many steps do we make in the worst case?

Well, in the worst case, we do 2 comparisons and 2 arithmetic operations in each loop. This gives a total of $4x$ steps.

Also, we check the guard of the while loop $x+1$ times, which gives $x+1$ steps.

There is also 1 step to count for `hi = len(A)-1`.

So, in total we make $5x+2$ steps.

Binary search worst case iterations

```
def binSearch(A, k):
    lo = 0
    hi = len(A)-1
    while (lo <= hi):
        mid = (lo+hi)//2
        if A[mid] == k:
            return mid
        else:
            if A[mid] < k:
                lo = mid+1
            else:
                hi = mid-1
    return -1
```

If the array has 16 elements, how many while-iterations do we make in the worst case?

- 1st time: lo=0, hi=15, mid=7
- 2nd : lo=8, hi=15, mid=11
- 3rd : lo=12, hi=15, mid=13
- 4th : lo=14, hi=15, mid=14
- 5th : lo=15, hi=15, mid=15
- 6th : lo=16, hi=15, STOP

So, we loop 5 times (the 6th time we skip the loop).

What if the input array had length n?

How many times we can divide by 2?

In each while iteration we divide the part of the array that we search in by 2, using //2:

- we start from [lo ... hi], and we end up with [lo ... mid] or [mid ... hi]

Until we reach the case where the search part as length 1, i.e. lo = hi. We then make an additional and final iteration and get lo > hi.

We can show that any number n can be divide it be //2 **exactly** $\log_2 n$ **times** until we reach 1 (where $\log_2 n$ is rounded down).

So, the total number of iterations in the worst case is $\log_2 n + 1$.

Remember logarithms: $\log_2 n = x$ means $n = 2^x$

From now on, when we write $\log n$, we mean $\log_2 n$.

Binary search running time

```
def binSearch(A, k):
    lo = 0
    hi = len(A)-1
    while (lo <= hi):
        mid = (lo+hi)//2
        if A[mid] == k:
            return mid
        else:
            if A[mid] < k:
                lo = mid+1
            else:
                hi = mid-1
    return -1
```

If the array has n elements, we will loop $\log n + 1$ times.

Before, we saw that if we loop x times we have so many steps:

$$5x + 2$$

Replacing $\log n + 1$ for x we get:

$$5(\log n + 1) + 2$$

So, in total we make $5\log n + 7$ steps.

Binary search: worst case

If array A contains n elements, how many times do we go through the while loop?

- in each while iteration, we cut the search range in half
    - e.g. if range has length 11 → new range has 11//2 = 5
- in the worst case, we iterate as many times we can until we get an empty range
    - i.e. we repeat as many times as we can cut n in half,
    - until we get a range of length 1, and we do a last iteration.

Altogether, these are $\log_2 n + 1$ iterations (rounded down).

Linear vs binary search: verdict

We can now answer the question of which search algorithm is more efficient.

If the input array has n elements:

- linear search will at worst make $n+1$ steps
- binary search will at worst do $5\log n + 7$ steps

Which one is more efficient?

size n	$n+1$	$5\log n + 7$
10	11	22
100	101	37
1000	1001	52
10000	10001	72
100000	100001	87
1000000	1000001	102

Note that we round down $\log n$ to the previous integer

Linear vs binary search: verdict

We can now answer the question of which search algorithm is more efficient.

If the input array has n elements:

- linear search will at worst make $n+1$ steps
- binary search will at worst do $5 \log n + 7$ steps

Which one is more efficient?

Binary search is **way more efficient**
(for very large values of n)

size n	$n+1$	$5 \log n + 7$
10	11	22
100	101	37
1000	1001	52
10000	10001	72
100000	100001	87
1000000	1000001	102

Note that we round down $\log n$ to the previous integer

Comparing running times

Complexity analysis is about examining the running time of algorithms with respect to the size of their input.

- I.e. assuming that the input is of size n, we try to express the running time as a function $f(n)$.

Different algorithms for a PROBLEM can have different running times:

- Algorithm A runs in 4242 steps
- Algorithm B runs in $n^2 + 1$ steps

Which one is more efficient? It depends on the value of n:

- for $n = 10$: $4242 > n^2 + 1 = 101$
- for $n = 10^{100}$: $4242 < n^2 + 1 \sim 10^{200}$

So, for large values of n (the ones we care about), 4242 is more efficient

big-Θ notation

We compared these expressions for large n:

- $n+1$ vs $5\log n + 7$
- 4242 vs $n^2 + 1$

In such expressions, we can simply focus on the components that describe their **order of growth** as n becomes large (and forget the rest):

- in $n+1$, we can focus on n: we say that $n+1$ is $\Theta(n)$
- in $5\log n + 7$, we focus on $\log n$: we say that $5\log n + 7$ is $\Theta(\log n)$
- in $n^2 + 1$, we focus on n^2: we say that $n^2 + 1$ is $\Theta(n^2)$
- in 4242 there is no change as n grows : we say that 4242 is $\Theta(1)$

This is called **big-Θ notation** (read: *big theta*) and is a standard measure of complexity of algorithms.

big-Θ notation

Using big-Θ notation:

- $n+1$ is $\Theta(n)$
- $5\log n+7$ is $\Theta(\log n)$
- n^2+1 is $\Theta(n^2)$
- 4242 is $\Theta(1)$

In other words, as n grows very large we can assume that:

- $n+1$ grows proportionally to n
- $5\log n+7$ grows prop. to $\log n$
- n^2+1 grows prop. to n^2
- 4242 remains constant

For large n:

1 **smaller than** $\log n$ **smaller than** n **smaller than** n^2

So, we can say that:

4242 **grows slower / is more efficient than** $5\log n+7$ **grows slower / is more efficient than** $n+1$ **grows slower / is more efficient than** n^2+1

These big-Θ simplifications are great: they allow us to compare different running times in a simple way that looks at what happens for large n

big-Θ notation usefulness

Big-Θ notation allows us to:

- take a potentially complicated $f(n)$ and replace it with a simpler one that has essentially the same behaviour for large n
- compute the running time of an algorithm without caring about constant multiplying factors and low-significance terms
  - e.g. it is OK to count all basic operations as 1 step each
  - if a loop has constant running time, it is OK to count the number of times it is repeated, rather than its total number of steps

For example, all of the following expressions are $\Theta(n)$:

- $n,\ n+1,\ 5000n+1500,\ 12n+1500\log n+2300$

all of the following expressions are $\Theta(n^2)$:

- $n^2,\ n^2+1,\ 5000n^2+1500,\ 12n^2+1500n\log n+n+1500\log n+230$

and so on.

Complexity analysis of algorithms

Complexity analysis is about examining the running time of algorithms with respect to the size of their input:

- i.e. assuming that the input is of size n, we try to express the running time as a function $f(n)$

For example, for sorted arrays:

- linear search has running time $\Theta(1)$ in the best case (i.e. if we find our element straight away), and in the worst case $\Theta(n)$
- binary search has running time $\Theta(1)$ in the best case (i.e. if we find our element straight away), and in the worst case $\Theta(\log n)$

We focus on the worst case:

> the running time of an algorithm on inputs of size n is simply its running time **in the worst case** for inputs of that size

Big-Θ and big-O

To describe the worst-case running time we use **big-O notation** (read: *big oh*), which is the other standard measure of complexity of algorithms that we study.

We say that an algorithm is $O(f(n))$ if it cannot get worse (i.e. larger) than $\Theta(f(n))$ in the worst case.

For sorted arrays:

- linear search has worst-case running time $\Theta(n)$: we say that linear search is $O(n)$
- binary search has worst-case running time $\Theta(\log n)$: we say that linear search is $O(\log n)$

As $\log n$ is smaller than n for large n, binary search is more efficient.

Note that big-Θ describes the order of growth (of a function of n), whereas big-O gives a limit to the worst-case order of growth.

Big-Θ and big-O formally

Formally speaking, big-O and big-Θ are ways to say that a function is bounded in specific ways as n becomes very large.

- Big-O specifies upper bounds:

  $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and some (large) number N such that, for all $n > N$: $f(n) < c \cdot g(n)$

- Big-Θ specifies upper and lower bounds:
  - $f(n)$ is $\Theta(g(n))$ if there is are constants $c, d > 0$ and some (large) number N such that, for all $n > N$: $c \cdot g(n) < f(n) < d \cdot g(n)$

For example:

- $n+1$ is $\Theta(n)$ because, for all $n > 1$: $n < n+1 < 2n$
- $12n+1500 \log n + 2300$ is $\Theta(n)$ because, for all $n > 24138$:

$$n < 12n+1500 \log n + 2300 < 13n$$

Complexity analysis of selection sort

The number of steps made by selection sort can be computed by counting comparisons. Assuming the input array has size n:

- $n - 1$ comparisons for finding the smallest element of the array
- $n - 2$ for finding the second smallest element of the array
- $n - 3$ for finding the second third element of the array

  $\dots$

- 1 for finding the ($n-1$)-th smallest element of the array

Thus, in the every case, we have so many steps:

$$(n-1) + (n-2) + \dots + 2 + 1$$

which is equal to: $n(n-1)/2 = n^2/2 - n/2$ which is $\Theta(n^2)$.

Thus, selection sort is $O(n^2)$.

Complexity analysis of insertion sort

The number of steps made by insertion sort can be computed by counting comparisons. Assuming the input array has size n:

- 1 comparison for inserting second item into array part with 1 item
- 1 to 2 for inserting third item into array part with 2 items
- 1 to 3 for inserting fourth item into array part with 3 items
  …
- 1 to $(n-1)$ for inserting n-th item into array part of $n-1$ items.

Thus, in the worst case, we have so many steps:

$$1 + 2 + \dots + (n-2) + (n-1)$$

which is equal to: $n(n-1)/2 = n^2/2 - n/2$ which is $\Theta(n^2)$.

Thus, insertion sort is $O(n^2)$.

Summary

This week we started studying sorting algorithms. We looked at simple algorithms: selection sort and insertion sort.

We took a detour to efficiency/complexity analysis of algorithms:

- we introduced the big-Θ notation: this lets us express the order of growth of the running time of an algorithm in relation to the size of the input
- the (time) complexity of an algorithm is typically measured by looking at its running time in the worst case – for this, we use the big-O notation

In short, an algorithm is $O(n)$ if:

on inputs of size n, the running time is no worse than $\Theta(n)$

We then used big-O and big-Θ to analyse the complexity of search and sorting algorithms.

Exercises

1. Here are some complexity classes in big-Θ notation:

$$\Theta((\log n)^2),\ \Theta(\log n),\ \Theta(\log n^2),$$
$$\Theta(n\log n), \Theta(n^2),\ \Theta(n),\ \Theta(n^{42}),\ \Theta(2^n)$$

   Put them in order, starting from the smallest complexity (i.e. the one for which algorithms run faster)

2. Compute the worst-case running time of binary search when, apart from comparisons and integer operations, we also count as steps all variable assignments.

```
def binSearch(A, k):
    lo = 0
    hi = len(A)-1
    while (lo <= hi):
        mid = (lo+hi)//2
        if A[mid] == k:
            return mid
        else:
            if A[mid] < k:
                lo = mid+1
            else:
                hi = mid-1
    return -1
```

   Is it still $\Theta(\log n)$?

3. Define a function `swap(i,j,A)` that swaps `A[i]` and `A[j]` inside `A` without using any auxiliary variables, where `A` is an array of integers.

4. Recall these array functions from last week:

```
def append(A, k):
    B = [0 for i in range(len(A)+1)]
    for i in range(len(A)):
        B[i] = A[i]
    B[len(A)] = k
    return B
```

```
def searchLast(A, k):
    for i in range(len(A)-1,-1,-1):
        if A[i] == k:
            return i
    return -1
```

   For each of them, find the worst-case running time with in relation to the length of the array `A`.

5. Spot the error in the following code:

```
def insertionSort(A):
    B = []
    for i in range(len(A)):
        insert(A[i],B)
    A = B
```

6. Using any algorithm, what is the least possible number of comparisons we need to make in order to find the minimum in an int array of length n?

Exercises – Solutions

1. Here they are in order:

$$\Theta(\log n) = \Theta(\log n^2) < \Theta((\log n)^2) < \Theta(n)$$
$$< \Theta(n \log n) < \Theta(n^2) < \Theta(n^{42}) < \Theta(2^n)$$

2. Suppose the number of while iterations is x. We have:
   - we make a total of $2x$ comparisons and $2x$ integer operations within the loop: total $4x$
   - we also check $x+1$ times whether `lo <= hi`
   - we make 1 integer operation at the start
   - we make 2 assignments in each loop iteration, which gives a total of $2x$ assignments
   - we make 2 assignments at the start

   So, summing up, we get $7x+4$ steps. Now, the number of iterations is still $\log n+1$, so replacing $\log n+1$ for x in $7x+4$ the we get $7\log n+11$.

   This is still $\Theta(\log n)$. The moral is that in complexity analysis we do not need to be precise about the number of steps in a given part of our code, so long as that number is constant. What is important is how many times are these steps repeated.

3. Here is such an implementation of `swap`:

```
def swap(i,j,A):
    A[i] = A[i]+A[j]
    A[j] = A[i]-A[j]
    A[i] = A[i]-A[j]
```

4. For `append`, assuming `A` has length n, we have:
   - n steps for creating `B`
   - we loop n times and make 1 assignment per time
   - one step for `B[len(A)] = k`

   which sums up to $2n+1$.

   For `searchLast`, we loop through the elements of `A` until we find `k` or we reach the end of `A`. So, in the worst case, we loop through all elements, i.e. n times. Each time we make one comparison (`A[i] == k`). Thus, overall we have n steps in the worst case.

   In terms of big-Θ, both functions are $\Theta(n)$.

5. There is a subtle bug, which is easier to see if we run:

```
A = [2,1]
insertionSort(A)
print(A) # prints [2,1]
```

   This is because the assignment `A=B` inside the body of `insertionSort` simply updates the *local variable* `A` to `B`, it does not update the values of the outside `A`

6. Each comparison reduces by 1 the elements that could be the minimum. In any algorithm, starting from n possible minimums we need to reach just 1 at the end. So, we need at least n-1 comparisons overall.