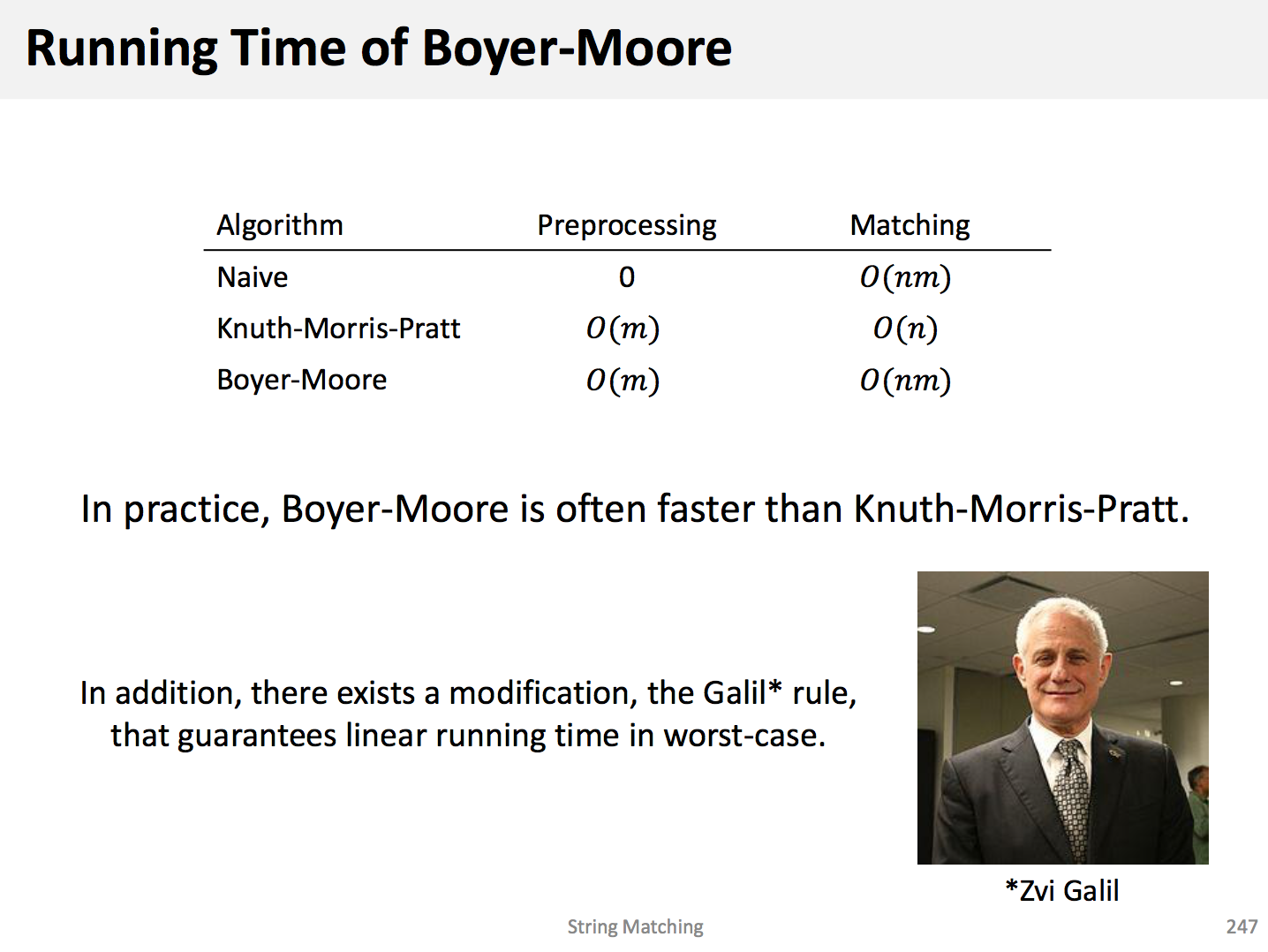
## 1.a.

1. Lowest asymptotic boundary of an algorithm
2. ~~True, because size of the string that you search in is larger than the pattern and hence complexity of n^2 is larger than nm~~

~~~~

1. The next shift value is calculated by s’ = s + (q - π[q]), where s is the old shift value, q is the amount we’ve matched so far and π[q] is the value of q in the prefix table. So the amount we shift by is **(q - π[q])**. q in this case is 5, as we’ve matched 5 characters (s-e-n-s-e-s). π[q] is defined as the longest prefix of P (the whole pattern), that is also a suffix of P5 (what we’ve matched so far). So π[5] is 2, as the longest **prefix** of P (*senses*) that is also a **suffix** of P5 (*sense*) is “*se*”. So the total amount we shift by is **q - π[q] = 5 - 2 = 3**.

## 1.b.

1. Passing in a sorted array to standard quicksort means the maximum/minimum is always selected as the pivot (standard quicksort just picks the last item in the array). This splits the array into two arrays, of length **n - 1** and **1**, which is the worst input case that leads to an O(n2) runtime.
2. Randomising the selection of the pivot means it is impossible to *force* quicksort to choose the minimum/maximum each time. Hence we cannot force the worst case runtime.
3. If the hash function that the hash table uses is known, it may be possible to find hash collisions (two *different* input values that *hash* to the same value). This means they will be stored in the same bucket, and the runtime for retrieval will degenerate to O(n) or O(logn), depending on how the buckets handle collisions (linked list vs. balanced binary tree).
4. The randomised layer distribution of skip lists means it is not possible to force a worse runtime.
5. Could use (not in tree)
6. Could use (not in tree)

## 3.c.

1. Iterate over the array and find the lowest number that is greater than x:

def succ(arr, x):

cur\_succ = float('inf')

for cand in arr:

if cand > x and cand < cur\_succ:

cur\_succ = cand

return cur\_succ

2. Iterate over the array and find the smallest number in the entire array. Then you can use the successor table iteratively to construct the sorted array.

**Example**: [4,3,1,8,9,5]

Find the **minimum** of the array and add it to our new “sorted” array: [1]

Lookup the successor of 1 and add i t to our new array: [1,3]

Lookup the successor of 3 and add it to our new array: [1,3,4]

And so on… until you reach the maximum, at which point you’ll have a fully sorted array.

## 1.d.

Optional.

## 2.

## 



b)

{s,α,δ,γ,t}

2

c)



d)

Considering augmentable vertices backwards from t.

We see we could only change (γ,t), since (δ,t) is at capacity.

Now we have two choices (β,γ) or (δ,γ)

On looking into (β,γ), we see it could never make an augmenting forward path, since its capacities are full.

Therefore our path is now {..., δ,γ,t}.

Then only could be α and of course s. So we have path: {s,α,β,γ,t}

Unique by determinacy of choice (we had none).

e)



By inspection, no non-trivial simple cycles.