Build a String

Taylor He, Jacob Manzelmann, Thomas Osterman

I pledge my honor that I have abided by the Stevens Honor System.

BUILD DAT STRANG

Greg wants to build a string, S.

Starting with an empty string, he can perform 2 operations:

1. Add a character to the end of S for A dollars.
2. Copy any substring of S, and then add it to the end of S for B dollars.

Calculate minimum amount of money Greg needs to build S.

Example:

N A B

9 4 5

aabaacaba

Test Case 0:

$S_{initial} =$ "" ; $S_{final} =$ "$aabaacaba$"

Append "a"; $S =$ "a"; cost is 4

Append "a"; $S =$ "aa"; cost is 4

Append "b"; $S =$ "aab"; cost is 4

Copy and append "aa"; $S =$ "$aabaa$"; cost is 5

Append "c"; $S =$ "$aabaac$"; cost is 4

Copy and append "aba"; $S =$ "$aabaacaba$"; cost is 5

Summing each cost, we get $4 + 4 + 4 + 5 + 4 + 5 = 26$, so our output for *Test Case 1* is 26.

When in doubt, DP it up

Dynamic Programming Approach: O(n^3)

Keep track of the lowest costs so far in a cost [ ] as we build up the new string

```python
def build_string_dp(a, b, s):
    """ Finds the lowest cost to build a string using DP """
    cost = [0] + ([2**31 - 1] * len(s))
    min_copy_length = min(1, b/a)   # At what length does it make sense to copy?
    for i in range(1, len(cost)):
        cost[i] = min(cost[i], cost[i-1] + a)
        # j <= i:            You can't copy more than you have already built
        # i + j < len(cost): You can't copy more than length of string
        # s[i:i+j] in s[:i]: The future string appears in the current one
        j = min_copy_length
        while (j <= i and i + j < len(cost) and s[i:i+j] in s[:i]):
            cost[i+j] = min(cost[i+j], cost[i] + b)
            j += 1
    return cost[-1]
```

Dynamic Programming solves all problems!

- ✔ Test Case #0
- ✔ Test Case #1
- ✔ Test Case #2
- ✔ Test Case #3
- ✔ Test Case #4
- ✔ Test Case #5
- ✔ Test Case #6
- ✔ Test Case #7
- ✔ Test Case #8
- ✔ Test Case #9
- ✔ Test Case #10

Dynamic Programming solves all problems!

Terminated due to timeout

Ask your friends for help:

✔ Test Case #0	✔ Test Case #1	✔ Test Case #2
✔ Test Case #3	✔ Test Case #4	✔ Test Case #5
✔ Test Case #6	✔ Test Case #7	✔ Test Case #8
✔ Test Case #9	✔ Test Case #10	🕒 Test Case #11
🕒 Test Case #12	🕒 Test Case #13	🕒 Test Case #14
🕒 Test Case #15	🕒 Test Case #16	🕒 Test Case #17
🕒 Test Case #18	🕒 Test Case #19	🕒 Test Case #20

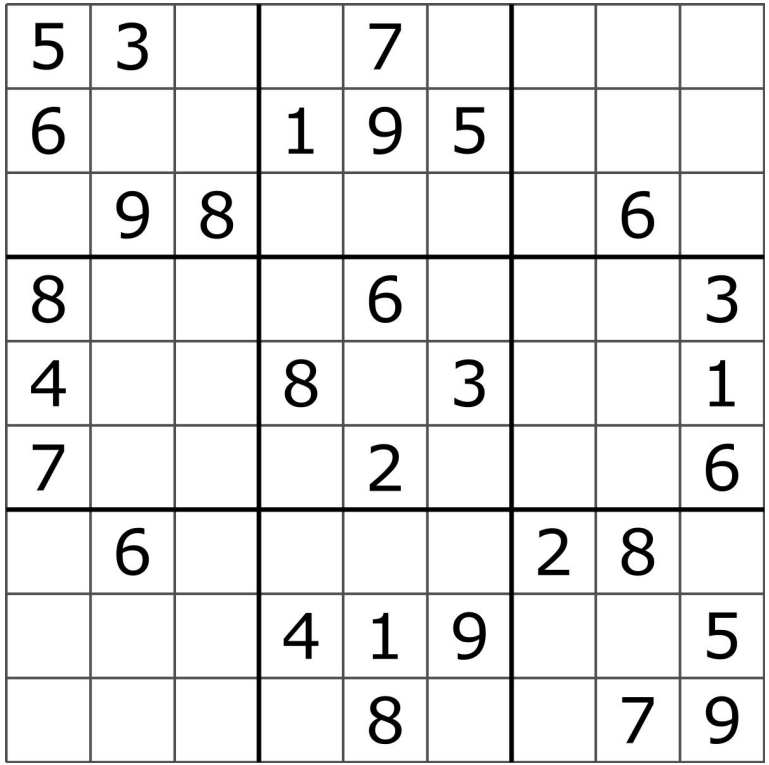
Try Again

How do we improve?

(besides committing sudoku)

Pain Point:

Checking s[i:i+1], s[i:i+2],... in s[:i]



```python
def build_string_dp(a, b, s):
    """ Finds the lowest cost to build a string using DP """
    cost = [0] + ([2**31 - 1] * len(s))
    min_copy_length = min(1, b/a)   # At what length does it make sense to copy?
    for i in range(1, len(cost)):
        cost[i] = min(cost[i], cost[i-1] + a)
        # j <= i:            You can't copy more than you have already built
        # i + j < len(cost): You can't copy more than length of string
        # s[i:i+j] in s[:i]: The future string appears in the current one
        j = min_copy_length
        while (j <= i and i + j < len(cost) and s[i:i+j] in s[:i]):
            cost[i+j] = min(cost[i+j], cost[i] + b)
            j += 1
    return cost[-1]
```

Approaches:

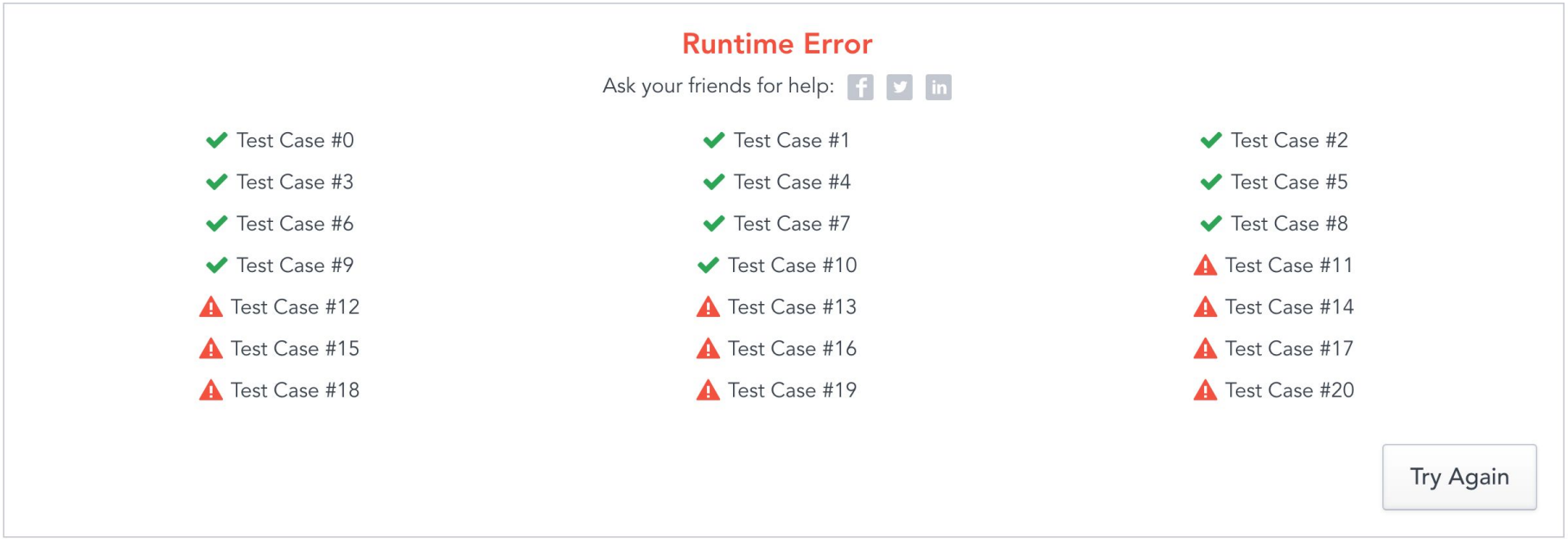
- Memoization
- Greedy
- Pattern Searching
  - Rabin - Karp rolling hash
  - Suffix Array

Caching found substrings

```
def build_string_dp(a, b, s):
    """ Finds the lowest cost to build a string using DP """
    cost = [0] + ([2**31 - 1] * len(s))
    found_substrings = set()        # Cache substrings that have appeared
    min_copy_length = min(1, b/a)   # At what length does it make sense to copy?
    for i in range(1, len(cost)):
        cost[i] = min(cost[i], cost[i-1] + a)
        # j <= i:            You can't copy more than you have already built
        # i + j < len(cost): You can't copy more than length of string
        # s[i:i+j] in s[:i]: The future string appears in the current one
        j = min_copy_length
        while (j <= i and i + j < len(cost)):
            if s[i:i+j] not in found_substrings and s[i:i+j] in s[:i]:
                found_substrings.add(s[i:i+j])
            else:
                break

            cost[i+j] = min(cost[i+j], cost[i] + b)
            j += 1
    return cost[-1]
```

Ahh, runtimes errors, likely too much memory



Greedy Approach

Working backwards

Split last character off of the string into its own string (ex: “abcb” -> “abc”, “b”)

If the right string is a substring of the left string, add one more character to the right string until it isn’t a substring (ex: “abc”, “b” -> “ab”, “cb”)

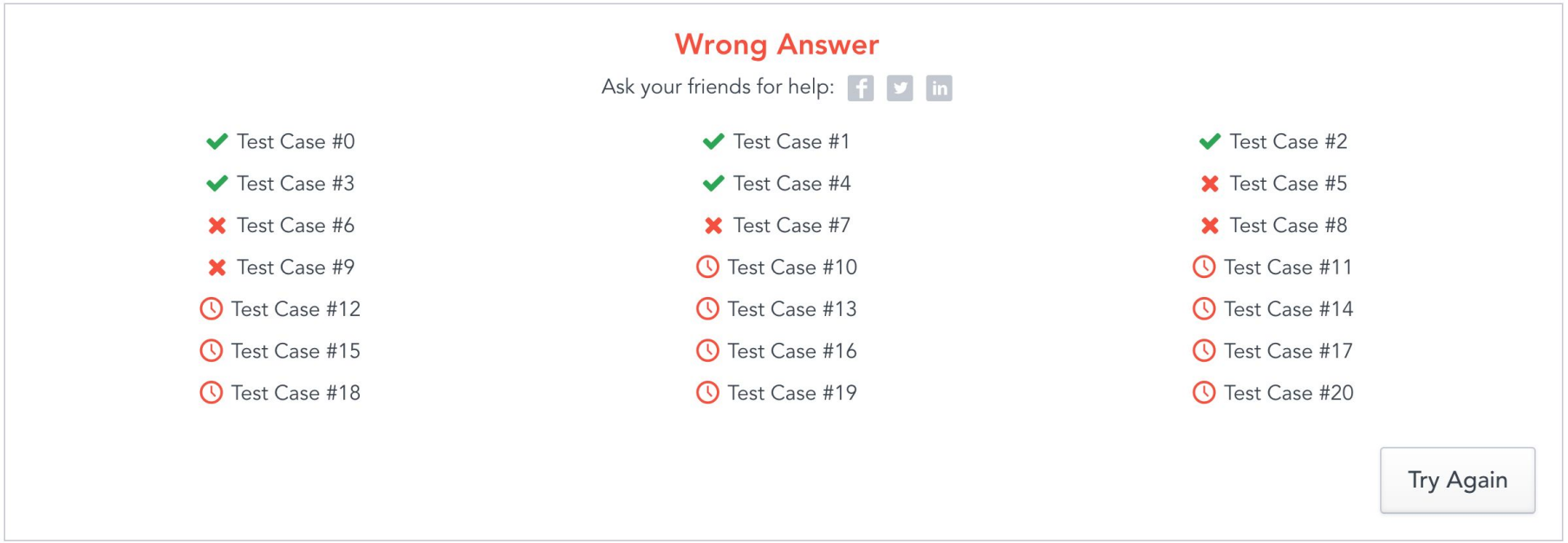
Once a substring isn’t found, throw away the right string, excluding the most recent character (ignored if the length of the right string is 1)

Add the min of the copy cost or the append cost for each character

Repeat until the string is empty

```
def buildString(append, copy, string):
    size = len(string)
    if size == 0:
        return 0
    lcs = 0
    size -= 1
    found = False
    while size != -1:
        temp = longestSubstring(string[:size], string[size:])
        if temp <= lcs:
            if found:
                return min(copy, append * (len(string[size:]) - 1)) + buildString(append,copy,string[:size+1])
            return min(copy, append * (len(string[size:]))) + buildString(append,copy,string[:size])
        lcs = temp
        size -= 1
        found = True
```

But the greedy approach doesn't account for everything



Besides being too slow

It doesn’t account for possible larger substrings earlier on.

Ex: “abccdabcd”, append=2, copy=3

Works like this: “abccd” “ab” “cd” cost = ((5 * 2) + 3 + 3) = 16

Instead of like this: “abccd” “abc” “d” cost = ((5 * 2) + 3 + 2) = 15

Pattern Searching Algorithms

- Rabin Karp

- Suffix Array

Rabin Karp

Naive Approach: Match 1 by 1

Rabin-Karp: Matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters.

Benefit: As the window slides, it only takes `O(1)` time to compute the new hash, and `O(1)` time to compare hashes.

Doing a full string comparison would take up to `O(m)` time every time you slide the window, where `m` is the length of the substring, and is likely to suffer from branch misprediction.

```
    while j <= i and i + j < len(cost) and rabin_karp_search(S[i:i+j], S[:i]):
        cost[i+j] = min(cost[i+j], cost[i] + B)
        j += 1
```

```python
def rabin_karp_search(pattern, s, d=5294212309, q=9743212277):
    """Rolling Hash
    The idea is to not recompute and store the hash every time,
    as we can compute quickly the hash given the previous string
    """
    m, n = len(pattern), len(s)
    # h = pow(d,m-1)%q
    # mod as we go
    h = 1
    for _ in range(m-1):
        h = (d * h)%q
    p, t = 0, 0
    result = []
    for i in range(m):
        p = (d*p+ord(pattern[i]))%q
        t = (d*t+ord(s[i]))%q
    for k in range(n-m+1):
        if p == t: # check character by character
            match = True
            for i in range(m):
                if pattern[i] != s[k+i]:
                    match = False
                    break
            if match:
                result = result + [k]
        if k < n-m:
            t = (t-h*ord(s[k]))%q
            t = (t*d+ord(s[k+m]))%q
            t = (t+q)%q
    return result
```

Terminated due to timeout

Ask your friends for help:

✔ Test Case #0	✔ Test Case #1	✔ Test Case #2
✔ Test Case #3	✔ Test Case #4	✔ Test Case #5
✔ Test Case #6	🕒 Test Case #7	🕒 Test Case #8
🕒 Test Case #9	🕒 Test Case #10	🕒 Test Case #11
🕒 Test Case #12	🕒 Test Case #13	🕒 Test Case #14
🕒 Test Case #15	🕒 Test Case #16	🕒 Test Case #17
🕒 Test Case #18	🕒 Test Case #19	🕒 Test Case #20

Try Again

Suffix Arrays

`Current Approach:`

`s[i:i+j] in s[:i]`

Checks if the next j characters are in the built string

Let’s OPTIMIZE!

Q: How do we find the optimal length j?

A: What are suffix arrays?

Suffix Arrays

Sorted array of all suffixes in a string

All suffixes of the word banana

Time to construct: O(n log n)

Source: wikipedia

Suffix	i
banana$	1
anana$	2
nana$	3
ana$	4
na$	5
a$	6
$	7

Suffix	i
$	7
a$	6
ana$	4
anana$	2
banana$	1
na$	5
nana$	3

Application to Problem

The suffix array of a string can be used as an index to quickly locate every occurrence of a substring pattern P within the string S

```python
def build_string_suffix(a, b, s):
    s2 = s[::-1]
    suffix_array, rank = suff_and_rank(s2)
    cost = [0] * (len(s) + 1)
    for i in range(len(s)):
        # Populate dp cost array
        cost[i+1] = cost[i] + a
        longest_prefix = find_longest_prefix(s2, suffix_array, rank, len(s)-i-1)
        for j in range(longest_prefix):
            cost[i+1] = min(cost[i+1], cost[i-j] + b)
    return cost[-1]
```

```python
def suff_and_rank(s):
    suffix_array = [i for i in range(len(s)+1)]
    rank = [ord(s[i]) for i in range(len(s))] + [-1]
    # print rank, suffix_array, s
    k = 1

    # O(nlg(n))
    while k <= len(s):
        # Sort suffix array by rank
        def suffcmp(a, b):
            if rank[a] != rank[b]:
                return 1 if rank[a] < rank[b] else -1
            ra = rank[a+k] if a + k <= len(s) else -1
            rb = rank[b+k] if b + k <= len(s) else -1
            if ra != rb:
                return 1 if ra < rb else -1
            return 0
            # return 1 if (rank[a], ra) < (rank[b], rb) else -1

        suffix_array.sort(cmp=suffcmp, reverse=True)
        # do the DP
        dp = [0] * (len(s) + 1)
        dp[suffix_array[0]] = 0
        for i in range(len(s)):
            # The next item in the cost array is the previous suffix array index
            # plus 1 if prev rank < i+1 rank, or -1 if prev rank >= i+1 rank
            dp[suffix_array[i+1]] = dp[suffix_array[i]] + suffcmp(suffix_array[i], suffix_array[i+1])
        rank = dp
        k *= 2
    # print suffix_array, rank
    return suffix_array, rank
```

```
def find_longest_prefix(s, suffix_array, rank, start):
    """
    Finds the longest prefix length given the string, a precomputed
    suffix array and its rank, and offset.
    """
    def where_differs(f, s):
        """Returns the index at which 2 strings differ"""
        i = 0
        while i < len(f) and f[i] == s[i]:
            i += 1
        return i

    ret = 0
    # Work down from rank
    i = rank[start]
    # print "i is:", i
    for i in range(rank[start], 0, -1):
        if (s[suffix_array[i]] != s[start]):
            break
        if suffix_array[i] <= start:
            continue
        # print "t:", s, "; offset:", start, print "; t+offset:", s[start]
        # Find the point at which the 2 arrays differ
        first = s[suffix_array[i]:]
        second = s[start:]
        dif = where_differs(s[suffix_array[i]:], s[start:])
        # print "L is:", dif
        if dif <= ret: break
        ret = max(ret, min(suffix_array[i] - start, dif))
    # print "start:", start, print "; s:", s
    for i in range(rank[start], len(suffix_array), 1):
        #print suffix_array[i]
        if s[suffix_array[i]-1] != s[start]: break
        if suffix_array[i] <= start: continue
        first = s[suffix_array[i]:]
        second = s[start:]
        dif = where_differs(s[suffix_array[i]:], s[start:])
        # print "l is:", distance_from_second
        if dif <= ret: break
        ret = max(ret, min(suffix_array[i] - start, dif))
    # print "ans:", ret
    return ret
```

Wrong Answer

Ask your friends for help:

✔ Test Case #0	✔ Test Case #1	✔ Test Case #2
✔ Test Case #3	✔ Test Case #4	✖ Test Case #5
✖ Test Case #6	✖ Test Case #7	✖ Test Case #8
✖ Test Case #9	🕒 Test Case #10	🕒 Test Case #11
🕒 Test Case #12	🕒 Test Case #13	🕒 Test Case #14
🕒 Test Case #15	🕒 Test Case #16	🕒 Test Case #17
🕒 Test Case #18	🕒 Test Case #19	🕒 Test Case #20

Try Again

Thank You!