

---

**WarWithArray.**

---

**PseudoCode:**

```
Input: int k, Array a[]\nFor each string e in a[]:\n    For each string f in a[]:\n        check if the string "e+f" is valid(e+f)
```

```
function valid(s):\n    For each b = (k length substring in s):\n        Loop through our original array a:\n            check if b is in a
```

**Runtime of compute2k():**

The algorithm essentially 4 nested loops, they run n-times, n-times, k-times, and n-times.  
So the run-time is  $O(kn^3)$

---

---

**WarWithBST.**

---

**PseudoCode:**

```
Input: int k, Array a[]\nFor each string e in a[]:\n    For each string f in a[]:\n        check if the string "e+f" is valid(e+f)
```

```
function valid(s):\n    For each b = (k length substring in s):\n        search the BST for the substring b
```

**Runtime of compute2k():**

The algorithm is essentially 4 nested loops, they run n-times, n-times, k-times, and log n-times.  
So the run-time is  $O(kn^2 \log n)$

---

**WarWithHash.**

---

**PseudoCode:**

```
Input: int k, Array a[]\nFor each string e in a[]:\n    For each string f in a[]:\n        check if the string "e+f" is valid(e+f)
```

```
function valid(s):\n    For each b = (k length substring in s):\n        For each character in b\n            hash character and add to b's hash\n            look up b's hash in the HashSet
```

**Runtime of compute2k():**

The algorithm is essentially 4 nested loops, they run  $n$ -times for the most outer loop,  $n$ -times for the second most outer loop,  $k$ -times for each substring of the  $2k$  string, and  $k$ -times for the hash of the  $k$ -length string. So the run-time is  $O(k^2n^2)$

---

**WarWithRollHash.**

---

**PseudoCode:**

```
Input: int k, Array a[]\nFor each string e in a[]: For each string f in a[]: check if the string "e+f" is valid(e+f)\nfunction valid(s): For each b = (k length substring in s): hash b and look it up in the HashSet
```

**Runtime of compute2k():**

The algorithm is very similar to the `compute2K()` method in `WarWithHash`, except that each hash (after the first hash) is computed in  $O(1)$  time, rather than  $O(k)$  time. The rolling hash uses the last hash value to compute the new hash with one addition, one subtraction, and one multiplication, regardless of the values of  $n$  or  $k$ .

The algorithm is essentially 3 nested loops, they run  $n$ -times for the most outer loop,  $n$ -times for the second most outer loop, and  $k$ -times for each substring of the  $2k$  string.

So the runtime is  $O(kn^2)$ .