Programming Tutorial

Khalid Hourani

University of Houston

October 9, 2020

Table of Contents

Table of Contents

Things to Consider

Good code depends on

Things to Consider

Good code depends on

- Correctness

Things to Consider

Good code depends on

- Correctness
- Readability

Things to Consider

Good code depends on

- Correctness
- Readability
- Performance

Correctness

Correctness

- Code should be correct

Correctness

- Code should be correct
- If code is incorrect then doesn't matter

Correctness

- Code should be correct
- If code is incorrect then doesn't matter
  - how performant it is

Correctness

- Code should be correct
- If code is incorrect then doesn't matter
    - how performant it is
    - how readable it is

Correctness

- Code should be correct
- If code is incorrect then doesn't matter
    - how performant it is
    - how readable it is

Correctness

- Code should be correct
- If code is incorrect then doesn't matter
  - how performant it is
  - how readable it is

Boolean Satisfiability

```
def bool_sat(formula):
  return True
```

Correctness

- Code should be correct
- If code is incorrect then doesn't matter
  - how performant it is
  - how readable it is

Boolean Satisfiability

```
def bool_sat(formula):
  return True
```

- Above code is $\mathcal{O}(1)$ solution to an NP-Complete Problem

Correctness

- Code should be correct
- If code is incorrect then doesn't matter
  - how performant it is
  - how readable it is

Boolean Satisfiability

```python
def bool_sat(formula):
  return True
```

- Above code is $\mathcal{O}(1)$ solution to an NP-Complete Problem
- But obviously not a correct solution

Testing

Testing

- You must test your code

Testing

- You must test your code
- Easiest way to catch bugs

Testing

- You must test your code
- Easiest way to catch bugs
- Example

Testing

- You must test your code
- Easiest way to catch bugs
- Example

Testing

- You must test your code
- Easiest way to catch bugs
- Example

Primality Checker

```python
def is_prime(n):
  if n in {0, 1}:
    return False
  for i in range(2, int(n ** 0.5)):
    if n % i == 0:
      return False
  return True
```

Testing

- You must test your code
- Easiest way to catch bugs
- Example

Primality Checker

```python
def is_prime(n):
  if n in {0, 1}:
    return False
  for i in range(2, int(n ** 0.5)):
    if n % i == 0:
      return False
  return True
```

- What is wrong with above code?

Testing

- You must test your code
- Easiest way to catch bugs
- Example

Primality Checker

```python
def is_prime(n):
  if n in {0, 1}:
    return False
  for i in range(2, int(n ** 0.5)):
    if n % i == 0:
      return False
  return True
```

- What is wrong with above code?
- Might not be obvious, but easy to see with testing

Test is_prime

```
for i in range(2, 11):
  print(i, is_prime(i))
```

Test `is_prime`

```python
for i in range(2, 11):
  print(i, is_prime(i))
```

i	`is_prime(i)`	Correct?
2	True	✓
3	True	✓
4	True	✗
5	True	✓
6	True	✗
7	True	✓
8	True	✗
9	True	✗
10	False	✓

Test is_prime

```python
for i in range(2, 11):
  print(i, is_prime(i))
```

i	is_prime(i)	Correct?
2	True	✓
3	True	✓
4	True	✗
5	True	✓
6	True	✗
7	True	✓
8	True	✗
9	True	✗
10	False	✓

- Incorrect for composite values 4, 6, 8, and 9

Test is_prime

```python
for i in range(2, 11):
  print(i, is_prime(i))
```

i	is_prime(i)	Correct?
2	True	✓
3	True	✓
4	True	✗
5	True	✓
6	True	✗
7	True	✓
8	True	✗
9	True	✗
10	False	✓

- Incorrect for composite values 4, 6, 8, and 9
- For these values `int(n ** 0.5)` $= \lfloor \sqrt{n} \rfloor = 2$

Test is_prime

```python
for i in range(2, 11):
  print(i, is_prime(i))
```

i	is_prime(i)	Correct?
2	True	✓
3	True	✓
4	True	✗
5	True	✓
6	True	✗
7	True	✓
8	True	✗
9	True	✗
10	False	✓

- Incorrect for composite values 4, 6, 8, and 9
- For these values `int(n ** 0.5)` $= \lfloor \sqrt{n} \rfloor = 2$
- Off-by-one error – add 1 to range: `int(n ** 0.5) + 1`

Readability

Readability

- Usually the most important thing after correctness

Readability

- Usually the most important thing after correctness
- Code will be read far more often than it is written

Readability

- Usually the most important thing after correctness
- Code will be read far more often than it is written
- Maintenance of code is often the highest expense

Syntax

- Follow a set of best-practices for your language. For example

Syntax

- Follow a set of best-practices for your language. For example
  - In Python, there is PEP 8

Syntax

- Follow a set of best-practices for your language. For example
  - In Python, there is PEP 8
  - In C, there is the Linux Kernel Style

Syntax

- Follow a set of best-practices for your language. For example
  - In Python, there is PEP 8
  - In C, there is the Linux Kernel Style
- The style chosen is less important than that you follow a consistent style

Syntax

- Follow a set of best-practices for your language. For example
  - In Python, there is PEP 8
  - In C, there is the Linux Kernel Style
- The style chosen is less important than that you follow a consistent style

Syntax

- Follow a set of best-practices for your language. For example
  - In Python, there is PEP 8
  - In C, there is the Linux Kernel Style
- The style chosen is less important than that you follow a consistent style

Ugly

```
c=(a+b)**0.5
L=[1,2,7]
```

Syntax

- Follow a set of best-practices for your language. For example
  - In Python, there is PEP 8
  - In C, there is the Linux Kernel Style
- The style chosen is less important than that you follow a consistent style

Ugly

```
c=(a+b)**0.5
L=[1,2,7]
```

Readable

```
c = (a + b) ** 0.5
L = [1, 2, 7]
```

- Easiest way to be consistent is to use a formatter

Syntax

- Follow a set of best-practices for your language. For example
  - In Python, there is PEP 8
  - In C, there is the Linux Kernel Style
- The style chosen is less important than that you follow a consistent style

Ugly

```
c=(a+b)**0.5
L=[1,2,7]
```

Readable

```
c = (a + b) ** 0.5
L = [1, 2, 7]
```

- Easiest way to be consistent is to use a formatter
- Automatically formats your code according to some style guide

Syntax

- Follow a set of best-practices for your language. For example
  - In Python, there is PEP 8
  - In C, there is the Linux Kernel Style
- The style chosen is less important than that you follow a consistent style

Ugly

```
c=(a+b)**0.5
L=[1,2,7]
```

Readable

```
c = (a + b) ** 0.5
L = [1, 2, 7]
```

- Easiest way to be consistent is to use a formatter
- Automatically formats your code according to some style guide
  - Black, yapf, autopep8, etc., for Python

Syntax

- Follow a set of best-practices for your language. For example
  - In Python, there is PEP 8
  - In C, there is the Linux Kernel Style
- The style chosen is less important than that you follow a consistent style

Ugly

```
c=(a+b)**0.5
L=[1,2,7]
```

Readable

```
c = (a + b) ** 0.5
L = [1, 2, 7]
```

- Easiest way to be consistent is to use a formatter
- Automatically formats your code according to some style guide
  - Black, yapf, autopep8, etc., for Python
  - clang-format for C/C++

Structure

- Syntactic readability (style) is not enough

Structure

- Syntactic readability (style) is not enough
- Code should be structured for readability

Structure

- Syntactic readability (style) is not enough
- Code should be structured for readability
- Consider following problem

Structure

- Syntactic readability (style) is not enough
- Code should be structured for readability
- Consider following problem

Structure

- Syntactic readability (style) is not enough
- Code should be structured for readability
- Consider following problem

Problem

Find the largest product of two 3-digit numbers that is a palindrome.

Structure

- Syntactic readability (style) is not enough
- Code should be structured for readability
- Consider following problem

Problem

Find the largest product of two 3-digit numbers that is a palindrome.

Brute-Force Solution

Iterate through all 10^6 pairs of products and check if they are palindromic while keeping track of max.

Example of Bad Structure

Example of Bad Structure

Poorly Structured

```python
biggest = 0
for a in range(100, 1000):
  for b in range(100, 1000):
    prod = a * b
    is_palindrome = True
    s = str(prod)
    n = len(s)
    # check if number is palindrome
    for i in range(n):
      if s[i] != s[n - i - 1]:
        is_palindrome = False
        break
    if not is_palindrome:
      continue
    else:
      if prod > biggest:
        biggest = prod
print(biggest)
```

Example of Bad Structure

Poorly Structured

```
biggest = 0
for a in range(100, 1000):
  for b in range(100, 1000):
    prod = a * b
    is_palindrome = True
    s = str(prod)
    n = len(s)
    # check if number is palindrome
    for i in range(n):
      if s[i] != s[n - i - 1]:
        is_palindrome = False
        break
    if not is_palindrome:
      continue
    else:
      if prod > biggest:
        biggest = prod
print(biggest)
```

- code is hard to follow

Example of Bad Structure

Poorly Structured

```python
biggest = 0
for a in range(100, 1000):
  for b in range(100, 1000):
    prod = a * b
    is_palindrome = True
    s = str(prod)
    n = len(s)
    # check if number is palindrome
    for i in range(n):
      if s[i] != s[n - i - 1]:
        is_palindrome = False
        break
    if not is_palindrome:
      continue
    else:
      if prod > biggest:
        biggest = prod
print(biggest)
```

- code is hard to follow
- unnecessarily complex — `is_palindrome` is essentially a sentinel value

Example of Bad Structure

Poorly Structured

```python
biggest = 0
for a in range(100, 1000):
  for b in range(100, 1000):
    prod = a * b
    is_palindrome = True
    s = str(prod)
    n = len(s)
    # check if number is palindrome
    for i in range(n):
      if s[i] != s[n - i - 1]:
        is_palindrome = False
        break
    if not is_palindrome:
      continue
    else:
      if prod > biggest:
        biggest = prod
print(biggest)
```

- code is hard to follow
- unnecessarily complex — `is_palindrome` is essentially a sentinel value
- code is imperative, not declarative

Example of Better Structure

Example of Better Structure

Better Structured

```
def is_palindrome(num):
  s = str(num)
  n = len(s)
  for i in range(n):
    if s[i] != s[n - i - 1]:
      return False
  return True


biggest = 0
for a in range(100, 1000):
  for b in range(100, 1000):
    if is_palindrome(a * b):
      biggest = max(biggest, a * b)
print(biggest)
```

Example of Better Structure

Better Structured

```python
def is_palindrome(num):
  s = str(num)
  n = len(s)
  for i in range(n):
    if s[i] != s[n - i - 1]:
      return False
  return True


biggest = 0
for a in range(100, 1000):
  for b in range(100, 1000):
    if is_palindrome(a * b):
      biggest = max(biggest, a * b)
print(biggest)
```

- code is much easier to follow

Example of Better Structure

Better Structured

```
def is_palindrome(num):
  s = str(num)
  n = len(s)
  for i in range(n):
    if s[i] != s[n - i - 1]:
      return False
  return True


biggest = 0
for a in range(100, 1000):
  for b in range(100, 1000):
    if is_palindrome(a * b):
      biggest = max(biggest, a * b)
print(biggest)
```

- code is much easier to follow
- function `is_palindrome` clearly conveys intent

Example of Better Structure

Better Structured

```
def is_palindrome(num):
  s = str(num)
  n = len(s)
  for i in range(n):
    if s[i] != s[n - i - 1]:
      return False
  return True


biggest = 0
for a in range(100, 1000):
  for b in range(100, 1000):
    if is_palindrome(a * b):
      biggest = max(biggest, a * b)
print(biggest)
```

- code is much easier to follow
- function `is_palindrome` clearly conveys intent
- code is declarative

Example of Good Structure

Example of Good Structure

Well Structured

```python
def is_palindrome(num):
  s = str(num)
  n = len(s)
  for i in range(n):
    if s[i] != s[n - i - 1]:
      return False
  return True

def main():
  biggest = 0
  for a in range(100, 1000):
    for b in range(100, 1000):
      if is_palindrome(a * b):
        biggest = max(biggest, a * b)
  print(biggest)

if __name__ == "__main__":
    main()
```

Example of Good Structure

Well Structured

```
def is_palindrome(num):
  s = str(num)
  n = len(s)
  for i in range(n):
    if s[i] != s[n - i - 1]:
      return False
  return True

def main():
  biggest = 0
  for a in range(100, 1000):
    for b in range(100, 1000):
      if is_palindrome(a * b):
        biggest = max(biggest, a * b)
  print(biggest)

if __name__ == "__main__":
    main()
```

- basically same code

Example of Good Structure

Well Structured

```
def is_palindrome(num):
  s = str(num)
  n = len(s)
  for i in range(n):
    if s[i] != s[n - i - 1]:
      return False
  return True

def main():
  biggest = 0
  for a in range(100, 1000):
    for b in range(100, 1000):
      if is_palindrome(a * b):
        biggest = max(biggest, a * b)
  print(biggest)

if __name__ == "__main__":
    main()
```

- basically same code
- clearly describes what the program is doing overall

Comments

Comments

- Code should be commented

Comments

- Code should be commented
- but not over commented

Comments

- Code should be commented
- but not over commented

Comments

- Code should be commented
- but not over commented

Bad Comment

```
def i_sqrt(n):
  i = 0
  while i ** 2 < n:
    i += 1  # increment i
  return i
```

Comments

- Code should be commented
- but not over commented

Bad Comment

```
def i_sqrt(n):
  i = 0
  while i ** 2 < n:
    i += 1  # increment i
  return i
```

- it is clear that `i += 1` increments `i`

Comments

- Code should be commented
- but not over commented

Bad Comment

```python
def i_sqrt(n):
  i = 0
  while i ** 2 < n:
    i += 1  # increment i
  return i
```

- it is clear that `i += 1` increments `i`
- comment is superfluous and distracting

Good Comments

Good Comments

- As a rule of thumb

Good Comments

- As a rule of thumb
  - good code describes what and how

Good Comments

- As a rule of thumb
  - good code describes what and how
  - good comments describe why

Good Comments

- As a rule of thumb
  - good code describes what and how
  - good comments describe why

Good Comments

- As a rule of thumb
  - good code describes what and how
  - good comments describe why

Good Comment

```python
def is_prime(n):
  if n == 1:
    return False
  elif n in {2, 3}:
    return True
  elif n % 2 == 0:
    return False
  else:
    # we need only check for odd factors
    # up to sqrt(n)
    for i in range(3, int(n ** 0.5) + 1, 2):
      if n % i == 0:
        return False
    return True
```

Good Comments

- As a rule of thumb
  - good code describes what and how
  - good comments describe why

Good Comment

```python
def is_prime(n):
  if n == 1:
    return False
  elif n in {2, 3}:
    return True
  elif n % 2 == 0:
    return False
  else:
    # we need only check for odd factors
    # up to sqrt(n)
    for i in range(3, int(n ** 0.5) + 1, 2):
      if n % i == 0:
        return False
    return True
```

- this comment explains why the code is iterating from `1` to `int(n ** 0.5) + 1`

Good Comments

- As a rule of thumb
  - good code describes what and how
  - good comments describe why

Good Comment

```python
def is_prime(n):
  if n == 1:
    return False
  elif n in {2, 3}:
    return True
  elif n % 2 == 0:
    return False
  else:
    # we need only check for odd factors
    # up to sqrt(n)
    for i in range(3, int(n ** 0.5) + 1, 2):
      if n % i == 0:
        return False
    return True
```

- this comment explains why the code is iterating from 1 to `int(n ** 0.5) + 1`
- without this comment, reader would have to determine for themselves

Good Code

- If your code is well-written, it will improve readability

Good Code

- If your code is well-written, it will improve readability
- Not just in terms of structure, but things like variable and function names

Good Code

- If your code is well-written, it will improve readability
- Not just in terms of structure, but things like variable and function names
- Well-written code often makes many comments unnecessary

Good Code

- If your code is well-written, it will improve readability
- Not just in terms of structure, but things like variable and function names
- Well-written code often makes many comments unnecessary
- Often expressed as

Good Code

- If your code is well-written, it will improve readability
- Not just in terms of structure, but things like variable and function names
- Well-written code often makes many comments unnecessary
- Often expressed as

Good Code is its Own Best Documentation

Good Code

- If your code is well-written, it will improve readability
- Not just in terms of structure, but things like variable and function names
- Well-written code often makes many comments unnecessary
- Often expressed as

Good Code is its Own Best Documentation

- Not an excuse to avoid comments

Good Naming

Bad Names

```python
def qs(a):
  if len(a) <= 1:
    return a
  else:
    x, z = [], []
    y = a[0]
    for p in a:
      if p < y:
        x.append(p)
      else:
        z.append(p)
    return qs(x) + qs(z)
```

Good Names

```python
def quick_sort(array):
  if len(array) <= 1:
    return array
  else:
    left, right = [], []
    pivot = array[0]
    for ele in array:
      if ele < pivot:
        left.append(ele)
      else:
        right.append(ele)
    return quick_sort(left) + quick_sort(right)
```

- these names convey

Good Naming

Bad Names

```python
def qs(a):
  if len(a) <= 1:
    return a
  else:
    x, z = [], []
    y = a[0]
    for p in a:
      if p < y:
        x.append(p)
      else:
        z.append(p)
    return qs(x) + qs(z)
```

Good Names

```python
def quick_sort(array):
  if len(array) <= 1:
    return array
  else:
    left, right = [], []
    pivot = array[0]
    for ele in array:
      if ele < pivot:
        left.append(ele)
      else:
        right.append(ele)
    return quick_sort(left) + quick_sort(right)
```

- these names convey
  - basically nothing

Good Naming

Bad Names

```
def qs(a):
  if len(a) <= 1:
    return a
  else:
    x, z = [], []
    y = a[0]
    for p in a:
      if p < y:
        x.append(p)
      else:
        z.append(p)
    return qs(x) + qs(z)
```

Good Names

```
def quick_sort(array):
  if len(array) <= 1:
    return array
  else:
    left, right = [], []
    pivot = array[0]
    for ele in array:
      if ele < pivot:
        left.append(ele)
      else:
        right.append(ele)
    return quick_sort(left) + quick_sort(right)
```

- these names convey
  - basically nothing

- these names convey

Good Naming

Bad Names

```python
def qs(a):
  if len(a) <= 1:
    return a
  else:
    x, z = [], []
    y = a[0]
    for p in a:
      if p < y:
        x.append(p)
      else:
        z.append(p)
    return qs(x) + qs(z)
```

Good Names

```python
def quick_sort(array):
  if len(array) <= 1:
    return array
  else:
    left, right = [], []
    pivot = array[0]
    for ele in array:
      if ele < pivot:
        left.append(ele)
      else:
        right.append(ele)
    return quick_sort(left) + quick_sort(right)
```

- these names convey
  - basically nothing

- these names convey
  - function is quicksort

Good Naming

Bad Names

```python
def qs(a):
  if len(a) <= 1:
    return a
  else:
    x, z = [], []
    y = a[0]
    for p in a:
      if p < y:
        x.append(p)
      else:
        z.append(p)
    return qs(x) + qs(z)
```

Good Names

```python
def quick_sort(array):
  if len(array) <= 1:
    return array
  else:
    left, right = [], []
    pivot = array[0]
    for ele in array:
      if ele < pivot:
        left.append(ele)
      else:
        right.append(ele)
    return quick_sort(left) + quick_sort(right)
```

- these names convey
  - basically nothing

- these names convey
  - function is quicksort
  - input is an array

Good Naming

Bad Names

```python
def qs(a):
  if len(a) <= 1:
    return a
  else:
    x, z = [], []
    y = a[0]
    for p in a:
      if p < y:
        x.append(p)
      else:
        z.append(p)
    return qs(x) + qs(z)
```

Good Names

```python
def quick_sort(array):
  if len(array) <= 1:
    return array
  else:
    left, right = [], []
    pivot = array[0]
    for ele in array:
      if ele < pivot:
        left.append(ele)
      else:
        right.append(ele)
    return quick_sort(left) + quick_sort(right)
```

- these names convey
  - basically nothing

- these names convey
  - function is quicksort
  - input is an array
  - `left` and `right` are partitions around `pivot = array[0]`

Good Naming

Bad Names

```
def qs(a):
  if len(a) <= 1:
    return a
  else:
    x, z = [], []
    y = a[0]
    for p in a:
      if p < y:
        x.append(p)
      else:
        z.append(p)
    return qs(x) + qs(z)
```

Good Names

```
def quick_sort(array):
  if len(array) <= 1:
    return array
  else:
    left, right = [], []
    pivot = array[0]
    for ele in array:
      if ele < pivot:
        left.append(ele)
      else:
        right.append(ele)
    return quick_sort(left) + quick_sort(right)
```

- these names convey
  - basically nothing

- these names convey
  - function is quicksort
  - input is an array
  - `left` and `right` are partitions around `pivot = array[0]`
  - `ele` iterates through values of `array`

Miscellaneous

Miscellaneous

- Impractical to create exhaustive list of best practices

Miscellaneous

- Impractical to create exhaustive list of best practices
    - and some practices are debateable

Miscellaneous

- Impractical to create exhaustive list of best practices
  - and some practices are debateable
- Good idea to look at resources like

Miscellaneous

- Impractical to create exhaustive list of best practices
  - and some practices are debateable
- Good idea to look at resources like
  - The Little Book of Python Anti-Patterns

Miscellaneous

- Impractical to create exhaustive list of best practices
  - and some practices are debateable
- Good idea to look at resources like
  - The Little Book of Python Anti-Patterns
  - The C++ Core Guidelines

Miscellaneous

- Impractical to create exhaustive list of best practices
  - and some practices are debateable
- Good idea to look at resources like
  - The Little Book of Python Anti-Patterns
  - The C++ Core Guidelines
- And to use a linter — a static code analysis tool to flag bugs, style errors, etc, such as

Miscellaneous

- Impractical to create exhaustive list of best practices
  - and some practices are debateable
- Good idea to look at resources like
  - The Little Book of Python Anti-Patterns
  - The C++ Core Guidelines
- And to use a linter — a static code analysis tool to flag bugs, style errors, etc, such as
  - Python – flake8

Miscellaneous

- Impractical to create exhaustive list of best practices
  - and some practices are debateable
- Good idea to look at resources like
  - The Little Book of Python Anti-Patterns
  - The C++ Core Guidelines
- And to use a linter — a static code analysis tool to flag bugs, style errors, etc, such as
  - Python – flake8
  - C++ – clang-tidy

Performance

Performance

- Highly case-by-case

Performance

- Highly case-by-case
  - specific use case may allow for less performant code

Performance

- Highly case-by-case
  - specific use case may allow for less performant code
    - e.g. some one-time scientific calculation

Performance

- Highly case-by-case
  - specific use case may allow for less performant code
    - e.g. some one-time scientific calculation
  - or may require more performant code

Performance

- Highly case-by-case
  - specific use case may allow for less performant code
    - e.g. some one-time scientific calculation
  - or may require more performant code
    - e.g. real-time financial analysis

Performance

- Highly case-by-case
  - specific use case may allow for less performant code
    - e.g. some one-time scientific calculation
  - or may require more performant code
    - e.g. real-time financial analysis
- Usually comes down to correct choice of algorithm and data structure

Performance

- Highly case-by-case
  - specific use case may allow for less performant code
    - e.g. some one-time scientific calculation
  - or may require more performant code
    - e.g. real-time financial analysis
- Usually comes down to correct choice of algorithm and data structure
  - e.g., if checking existence of an element, an array gives $\mathcal{O}(n)$ but a hash table gives $\mathcal{O}(1)$

Performance

- Highly case-by-case
  - specific use case may allow for less performant code
    - e.g. some one-time scientific calculation
  - or may require more performant code
    - e.g. real-time financial analysis
- Usually comes down to correct choice of algorithm and data structure
  - e.g., if checking existence of an element, an array gives $\mathcal{O}(n)$ but a hash table gives $\mathcal{O}(1)$
    - which might make the difference between $\mathcal{O}\left(n^2\right)$ and $\mathcal{O}(n)$ overall

Performance

- Highly case-by-case
  - specific use case may allow for less performant code
    - e.g. some one-time scientific calculation
  - or may require more performant code
    - e.g. real-time financial analysis
- Usually comes down to correct choice of algorithm and data structure
  - e.g., if checking existence of an element, an array gives $\mathcal{O}(n)$ but a hash table gives $\mathcal{O}(1)$
    - which might make the difference between $\mathcal{O}\left(n^2\right)$ and $\mathcal{O}(n)$ overall
- While it does depend on use case, it is still usually best to focus on readability over performance

Performance

- Highly case-by-case
  - specific use case may allow for less performant code
    - e.g. some one-time scientific calculation
  - or may require more performant code
    - e.g. real-time financial analysis
- Usually comes down to correct choice of algorithm and data structure
  - e.g., if checking existence of an element, an array gives $\mathcal{O}(n)$ but a hash table gives $\mathcal{O}(1)$
    - which might make the difference between $\mathcal{O}\left(n^2\right)$ and $\mathcal{O}(n)$ overall
- While it does depend on use case, it is still usually best to focus on readability over performance
  - it's easier to make slow code fast than to make confusing code understandable

Table of Contents

Problem

Problem

Problem

Given an array of integers `nums` and a positive integer k, find whether it is possible to divide `nums` into sets of k consecutive numbers.

Where to Start

Where to Start

- Look at examples

Where to Start

- Look at examples
  1. $\texttt{nums} = [1, 2, 3, 3, 4, 4, 5, 6]$, $k = 4$

Where to Start

- Look at examples
  1. $\texttt{nums} = [1, 2, 3, 3, 4, 4, 5, 6]$, $k = 4$
     - $[1, 2, 3, 4]$, $[3, 4, 5, 6]$

Where to Start

- Look at examples
  1. $\texttt{nums} = [1, 2, 3, 3, 4, 4, 5, 6]$, $k = 4$
     - $[1, 2, 3, 4]$, $[3, 4, 5, 6]$
  2. $\texttt{nums} = [3, 2, 1, 2, 3, 4, 3, 4, 5, 9, 10, 11]$, $k = 3$

Where to Start

- Look at examples
  1. $\texttt{nums} = [1, 2, 3, 3, 4, 4, 5, 6]$, $k = 4$
     - $[1, 2, 3, 4]$, $[3, 4, 5, 6]$
  2. $\texttt{nums} = [3, 2, 1, 2, 3, 4, 3, 4, 5, 9, 10, 11]$, $k = 3$
     - $[1, 2, 3]$, $[2, 3, 4]$, $[3, 4, 5]$, $[9, 10, 11]$

Where to Start

- Look at examples
  1. $\texttt{nums} = [1, 2, 3, 3, 4, 4, 5, 6]$, $k = 4$
     - $[1, 2, 3, 4]$, $[3, 4, 5, 6]$
  2. $\texttt{nums} = [3, 2, 1, 2, 3, 4, 3, 4, 5, 9, 10, 11]$, $k = 3$
     - $[1, 2, 3]$, $[2, 3, 4]$, $[3, 4, 5]$, $[9, 10, 11]$
- Hopefully notice a pattern – if $x = \min(A)$

Where to Start

- Look at examples
  1. $\texttt{nums} = [1, 2, 3, 3, 4, 4, 5, 6]$, $k = 4$
     - $[1, 2, 3, 4]$, $[3, 4, 5, 6]$
  2. $\texttt{nums} = [3, 2, 1, 2, 3, 4, 3, 4, 5, 9, 10, 11]$, $k = 3$
     - $[1, 2, 3]$, $[2, 3, 4]$, $[3, 4, 5]$, $[9, 10, 11]$
- Hopefully notice a pattern – if $x = \min(A)$
  - then x, $x + 1$, $\ldots$, $x + k - 1$ form one subarray

Where to Start

- Look at examples
  1. $\texttt{nums} = [1, 2, 3, 3, 4, 4, 5, 6]$, $k = 4$
     - $[1, 2, 3, 4]$, $[3, 4, 5, 6]$
  2. $\texttt{nums} = [3, 2, 1, 2, 3, 4, 3, 4, 5, 9, 10, 11]$, $k = 3$
     - $[1, 2, 3]$, $[2, 3, 4]$, $[3, 4, 5]$, $[9, 10, 11]$
- Hopefully notice a pattern – if $x = \min(A)$
  - then x, $x + 1$, $\ldots$, $x + k - 1$ form one subarray
- Naturally lends itself to a recursive solution:

Where to Start

- Look at examples
  1. $\texttt{nums} = [1, 2, 3, 3, 4, 4, 5, 6]$, $k = 4$
     - $[1, 2, 3, 4]$, $[3, 4, 5, 6]$
  2. $\texttt{nums} = [3, 2, 1, 2, 3, 4, 3, 4, 5, 9, 10, 11]$, $k = 3$
     - $[1, 2, 3]$, $[2, 3, 4]$, $[3, 4, 5]$, $[9, 10, 11]$
- Hopefully notice a pattern – if $x = \min(A)$
  - then x, $x + 1$, $\ldots$, $x + k - 1$ form one subarray
- Naturally lends itself to a recursive solution:
  - remove x, $x + 1$, $\ldots$, $x + k - 1$, then repeat

Where to Start

- Look at examples
  1. $\texttt{nums} = [1, 2, 3, 3, 4, 4, 5, 6]$, $k = 4$
     - $[1, 2, 3, 4]$, $[3, 4, 5, 6]$
  2. $\texttt{nums} = [3, 2, 1, 2, 3, 4, 3, 4, 5, 9, 10, 11]$, $k = 3$
     - $[1, 2, 3]$, $[2, 3, 4]$, $[3, 4, 5]$, $[9, 10, 11]$
- Hopefully notice a pattern – if $x = \min(A)$
  - then x, $x + 1$, $\ldots$, $x + k - 1$ form one subarray
- Naturally lends itself to a recursive solution:
  - remove x, $x + 1$, $\ldots$, $x + k - 1$, then repeat
  - if run out of elements before removing all k, then no solution

Writing a Solution

Writing a Solution

- two observations

Writing a Solution

- two observations
  1. we are removing the k smallest elements

Writing a Solution

- two observations
  1. we are removing the k smallest elements
  2. if we know some element x, we know the next element is $x + 1$

Writing a Solution

- two observations
  1. we are removing the k smallest elements
  2. if we know some element x, we know the next element is $x + 1$

Writing a Solution

- two observations
  1. we are removing the k smallest elements
  2. if we know some element x, we know the next element is $x + 1$
- 1. suggests the use of a heap

Writing a Solution

- two observations
  1. we are removing the k smallest elements
  2. if we know some element x, we know the next element is $x + 1$
- 1. suggests the use of a heap

Writing a Solution

- two observations
  1. we are removing the k smallest elements
  2. if we know some element x, we know the next element is $x + 1$
- (1) suggests the use of a heap
- (2) suggests the use of a dictionary

Writing a Solution

- two observations
  1. we are removing the k smallest elements
  2. if we know some element x, we know the next element is $x + 1$
- 1. suggests the use of a heap
- 2. suggests the use of a dictionary
- both solutions are valid

Heap Solution

Heap Solution

- Heap Pop removes the smallest element

Heap Solution

- Heap Pop removes the smallest element
- Need to account for removing the k smallest distinct elements

Heap Solution

- Heap Pop removes the smallest element
- Need to account for removing the k smallest distinct elements
- Thus, set up `(value, count)` pairs

Heap Solution

- Heap Pop removes the smallest element
- Need to account for removing the k smallest distinct elements
- Thus, set up `(value, count)` pairs
- Then, pop heap — say smallest value is (x, c)

Heap Solution

- Heap Pop removes the smallest element
- Need to account for removing the k smallest distinct elements
- Thus, set up `(value, count)` pairs
- Then, pop heap — say smallest value is (x, c)
  - effectively removing c copies of x

Heap Solution

- Heap Pop removes the smallest element
- Need to account for removing the k smallest distinct elements
- Thus, set up `(value, count)` pairs
- Then, pop heap — say smallest value is (x, c)
  - effectively removing c copies of x
  - then remove c copies of $x + 1, \ldots, x + k - 1$ if possible

Heap Solution

- Heap Pop removes the smallest element
- Need to account for removing the k smallest distinct elements
- Thus, set up `(value, count)` pairs
- Then, pop heap — say smallest value is (x, c)
  - effectively removing c copies of x
  - then remove c copies of $x + 1, \ldots, x + k - 1$ if possible
  - if not possible, return `False`

Heap Solution

- Heap Pop removes the smallest element
- Need to account for removing the k smallest distinct elements
- Thus, set up `(value, count)` pairs
- Then, pop heap — say smallest value is (x, c)
  - effectively removing c copies of x
  - then remove c copies of $x + 1, \ldots, x + k - 1$ if possible
  - if not possible, return `False`
- After popping, need to return remaining elements to heap

Heap Solution

- Heap Pop removes the smallest element
- Need to account for removing the k smallest distinct elements
- Thus, set up `(value, count)` pairs
- Then, pop heap — say smallest value is (x, c)
  - effectively removing c copies of x
  - then remove c copies of $x + 1, \ldots, x + k - 1$ if possible
  - if not possible, return `False`
- After popping, need to return remaining elements to heap
- Keep track of remaining elements in array

Python Implementation

Python Implementation

```
popped = []  # store popped heap values
prev, min_count = heappop(heap)
for i in range(k - 1):
  if not heap:  # ran out of elements
    return False
  else:
      value, count = heappop(heap)
      if value != prev + 1:  # not consecutive
        return False
      else:
        count -= min_count
        prev = value
        if count > 0:
          popped.append((value, count))
for val in popped:
    heappush(heap, val)
```

Dictionary Solution

Dictionary Solution

- Dictionary holds the count for each element

Dictionary Solution

- Dictionary holds the count for each element
- if `x` is min and `D[x] = count`

Dictionary Solution

- Dictionary holds the count for each element
- if `x` is min and `D[x] = count`
  - must have `D[x + i] >= count` for $i \in \{x, x+1, \ldots, x+k-1\}$

Dictionary Solution

- Dictionary holds the count for each element
- if `x` is min and `D[x] = count`
    - must have `D[x + i] >= count` for $i \in \{x, x+1, \ldots, x+k-1\}$
    - else return `False`

Dictionary Solution

- Dictionary holds the count for each element
- if `x` is min and `D[x] = count`
  - must have `D[x + i] >= count` for $i \in \{x, x+1, \ldots, x+k-1\}$
  - else return `False`
- Decrease each count for $i \in \{x, x+1, \ldots, x+k-1\}$ by `count`

Dictionary Solution

- Dictionary holds the count for each element
- if `x` is min and `D[x] = count`
  - must have `D[x + i] >= count` for $i \in \{x, x+1, \ldots, x+k-1\}$
  - else return `False`
- Decrease each count for $i \in \{x, x+1, \ldots, x+k-1\}$ by `count`
- Remove entry if it becomes 0

Python Implementation

Python Implementation

```python
while counts:  # dictionary to store counts
  x = min(counts)
  min_count = counts[x]
  del counts[x]  # remove smallest
  for i in range(1, k):
      if counts[x + i] < min_count:
        return False
      else:
        counts[x + i] -= min_count
        if counts[x + i] == 0:
          del counts[x + i]
```