北京航空航天大學
BEIHANG UNIVERSITY

Introduction to Computer Science and Programming

Lecture 7

Sebastian Wandelt (小塞)

Beihang University

Outline

- Recap
- More (and better) sorting: Mergesort
- Algorithm complexity analysis
- Debugging Python programs
- Review: Typical mistakes during lab classes

Recap

- What do you remember about last week's lecture?

The formal problem of sorting

- Sorting is a fundamental operation in CS

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Insertion sort: Python code

```python
def insertion_sort(A):
    for i in range(0,len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j = j - 1
        A[j+1] = cur_value
```

Insertion sort: Python code

```
def insertion_sort(A):
    for i in range(0,len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j = j - 1
        A[j+1] = cur_value
```

Can we also do InsertSort to the right?

Big-O Notation

Formally speaking, let f, g: $N \rightarrow N$.

Then $\mathbf{f(n) = O(g(n))}$ **iff**

$$\exists n_0 \in N, c \in R.$$
$$\forall n \in N.$$
$$(n \geq n_0 \rightarrow f(n) \leq c \cdot g(n))$$

Intuitively, this means that f(n) is upper-bounded by g(n) aka f(n) is “at most as big as” g(n).

Bubble sort

- Compare all elements pairwise
  - Switch them if they are wrongly ordered

4	8	1	5	3	2	6	7

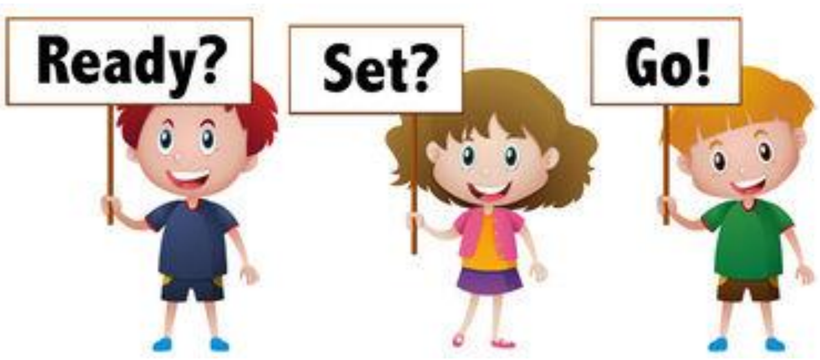
Bubble sort

- Compare all elements pairwise
  - Switch them if they are wrongly ordered

4	8	1	5	3	2	6	7

Let’s implement this in Python!

Best case vs. Worst case



1	2	3	4	5	...	n

n	...	5	4	3	2	1

9	7	n	...	1	4	2

Total work: $O(n)$ or $O(n^2)$ or $\Omega(n)$ or $\Omega(n^2)$?

Best case vs. Worst case

The worst-case runtime of insertion sort is $\Theta(n^2)$.

The best-case runtime of insertion sort is $\Theta(n)$.

Usually, we care more about the worst-case time.

Why?

Best case vs. Worst case

The worst-case runtime of insertion sort is $\Theta(n^2)$.

The best-case runtime of insertion sort is $\Theta(n)$.

Usually, we care more about the worst-case time.

We do not know the user's input at runtime, so we need to expect the worst-case.

It's acceptable, albeit not entirely precise, to say the runtime of insertion sort is $\Theta(n^2)$.

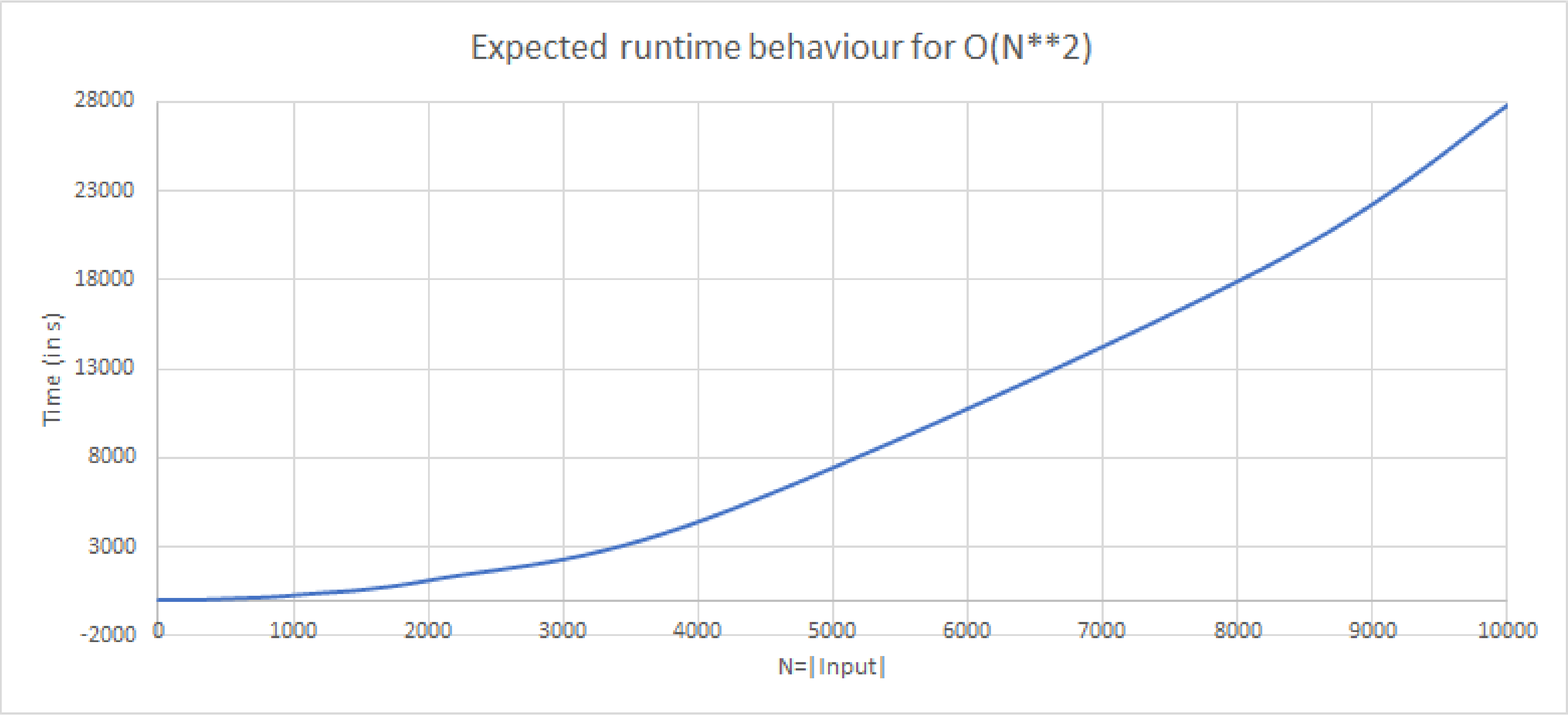
Best case vs. Worst case

The worst-case runtime of bubble sort is $\Theta(n^2)$.

The best-case runtime of bubble sort is $\Theta(n^2)$.

Is there any obvious problem with sorting so far from your side?

Is there any obvious problem with sorting so far from your side?



Is there any obvious problem with sorting so far from your side?

- All algorithms we know have a worst-case time complexity of O(N**2)
  - This is not good for large N …
    - If you double the size of the input, the runtime goes up by a factor of roughly four!
    - If you increase the size of the input by ten, the runtime goes up by a factor of roughly 100!
- This algorithm **does not scale** well for our problem

Can we do (significantly) better?

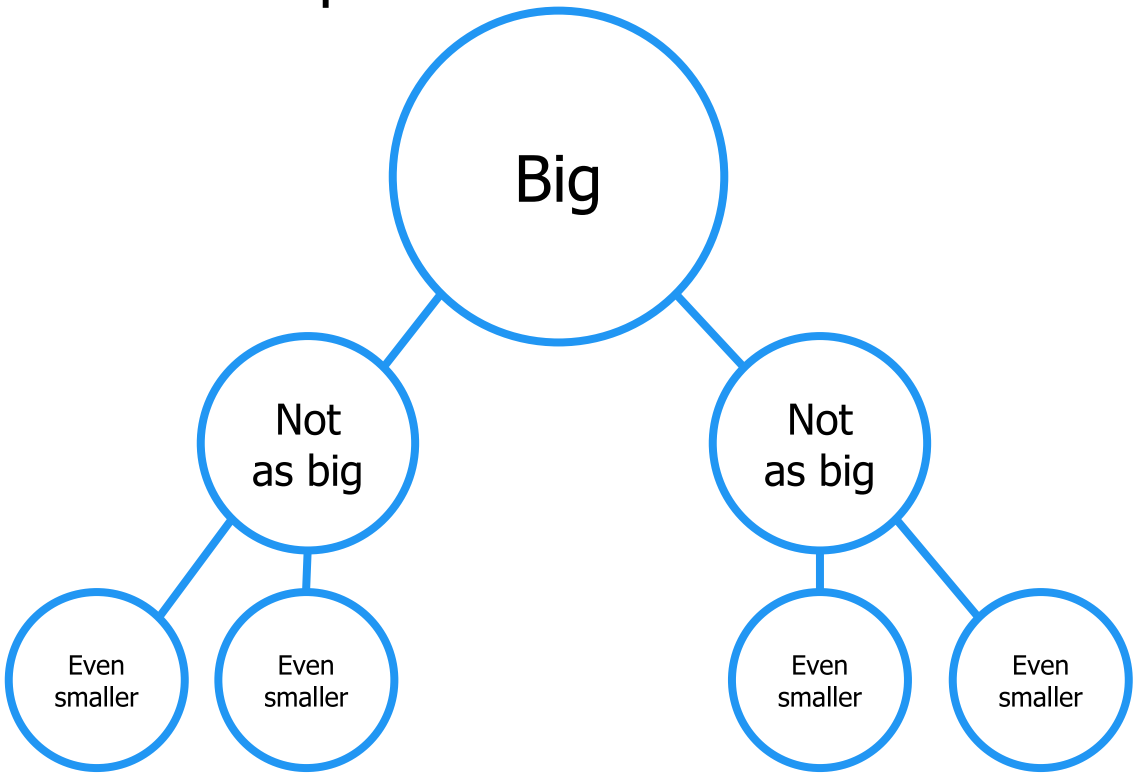
4	8	1	5	3	2	6	7

Divide and Conquer

Divide and Conquer

Divide: break current problem into smaller problems.

Conquer: solve the smaller problems and collate the results to solve the current problem.



Mergesort

Let’s use divide and conquer to improve upon insertion sort!

4	8	1	5	3	2	6	7

Let’s sort an unsorted list of numbers **`A`**.

1	4	5	8	2	3	6	7

Recursively sort each half, **`A[0:3]`** and **`A[4:7]`**, separately.

1	2	3	4	5	6	7	8

Merge the results from each half together.

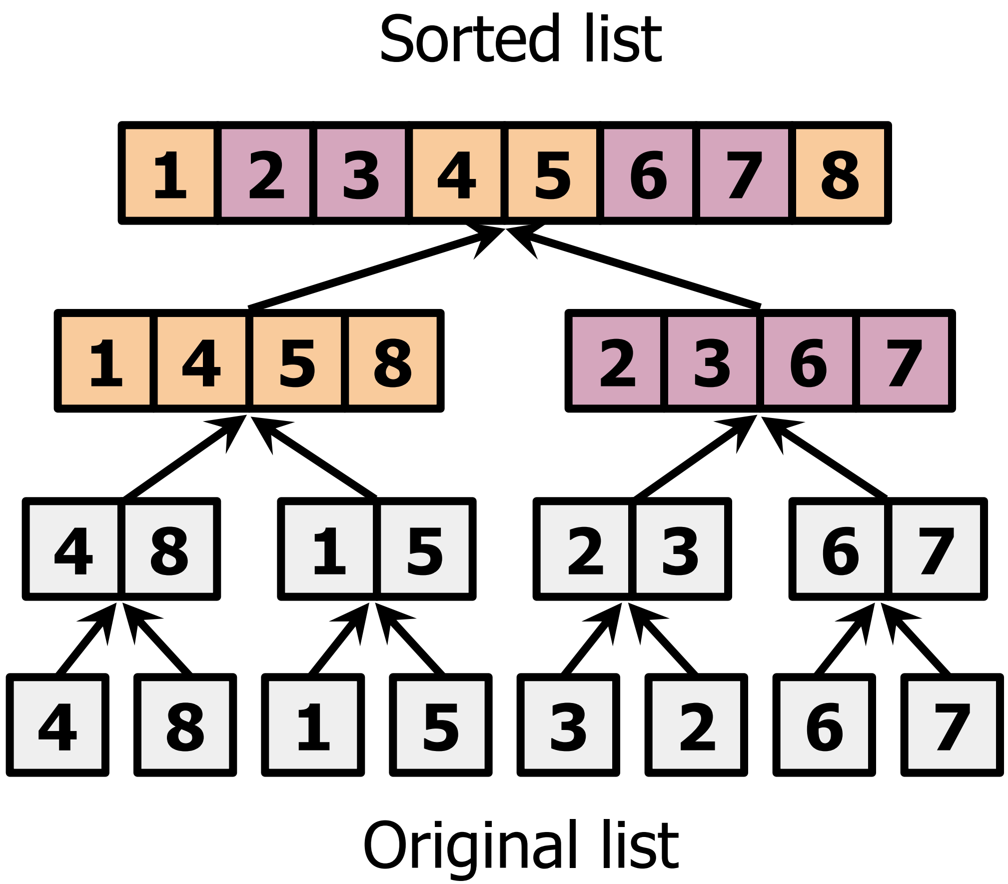
Mergesort: Pseudocode

```
algorithm mergesort(list A):
  if length(A) ≤ 1:
    return A
  let left = first half of A
  let right = second half of A
  return merge(
    mergesort(left),
    mergesort(right)
  )
```

Total work: O(???)

Mergesort

Tracing the recursive calls …



Mergesort

Question 1 How do we prove this algorithm always sorts the input list?

Question 2 How efficiently does this algorithm sort the input list?

Proving Correctness

- Consider a list of length k.
  - If k is 0 or 1, `mergesort` correctly sorts the list since an empty or single-element list is already sorted (base case).
  - Now suppose `mergesort` correctly sorts lists of length 1 to k-1. Since `left` and `right` must have lengths 1 to k-1, `mergesort` correctly sorts these lists.
- By construction, `merge` joins the elements from the two sorted lists into a single sorted list of length k, which it returns.
  - Thus, `mergesort` returns the elements of the original list, but in sorted order (inductive case).
- In the top recursive call, `mergesort` sorts the original array of length n (conclusion).

Analyzing Runtime

Let $T(n)$ represent the runtime of `mergesort` on a list of length n.

$T(n/2)$ is the runtime of `mergesort` on a list of length n/2.

$T(6881441)$ is the runtime of `mergesort` on a list of length 6,881,441.
$T(\lceil n/17 \rceil)$ is the runtime of`mergesort` on a list of length $\lceil n/17 \rceil$.

Recall that `mergesort` on a list of length n calls `mergesort` once for `left` and once for `right`, which costs $\mathbf{T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)}$.

After that, it calls `merge` on the two sublists, which costs $\mathbf{\Theta(n)}$.

Here's our first **recurrence relation**:

$$T(0) = \Theta(1)$$

$$T(1) = \Theta(1)$$

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$$

Analyzing Runtime

- A **recurrence relation** is a function or sequence whose values are defined in terms of earlier values.
- Here, we've written a recurrence relation for the runtime of mergesort. But we could have just as easily written one to describe something else recursive.
- For instance, the Fibonacci sequence can be defined by its recurrence relation $T(n) = T(n-1) + T(n-2)$, where $T(n)$ represents the n^{th} element of the sequence.

Analyzing Runtime

How do we solve our recurrence relation?

Assumption 1: First, it's helpful to assume that n is a power of two.

$$\cancel{T(0) = \Theta(1)}$$

$$T(1) = \Theta(1) = c_1$$

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$$
$$= 2T(n/2) + c_2 n$$

Assumption 2: Let $c = \max\{c_1, c_2\}$

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

Analyzing Runtime

How do we solve our new recurrence relation?

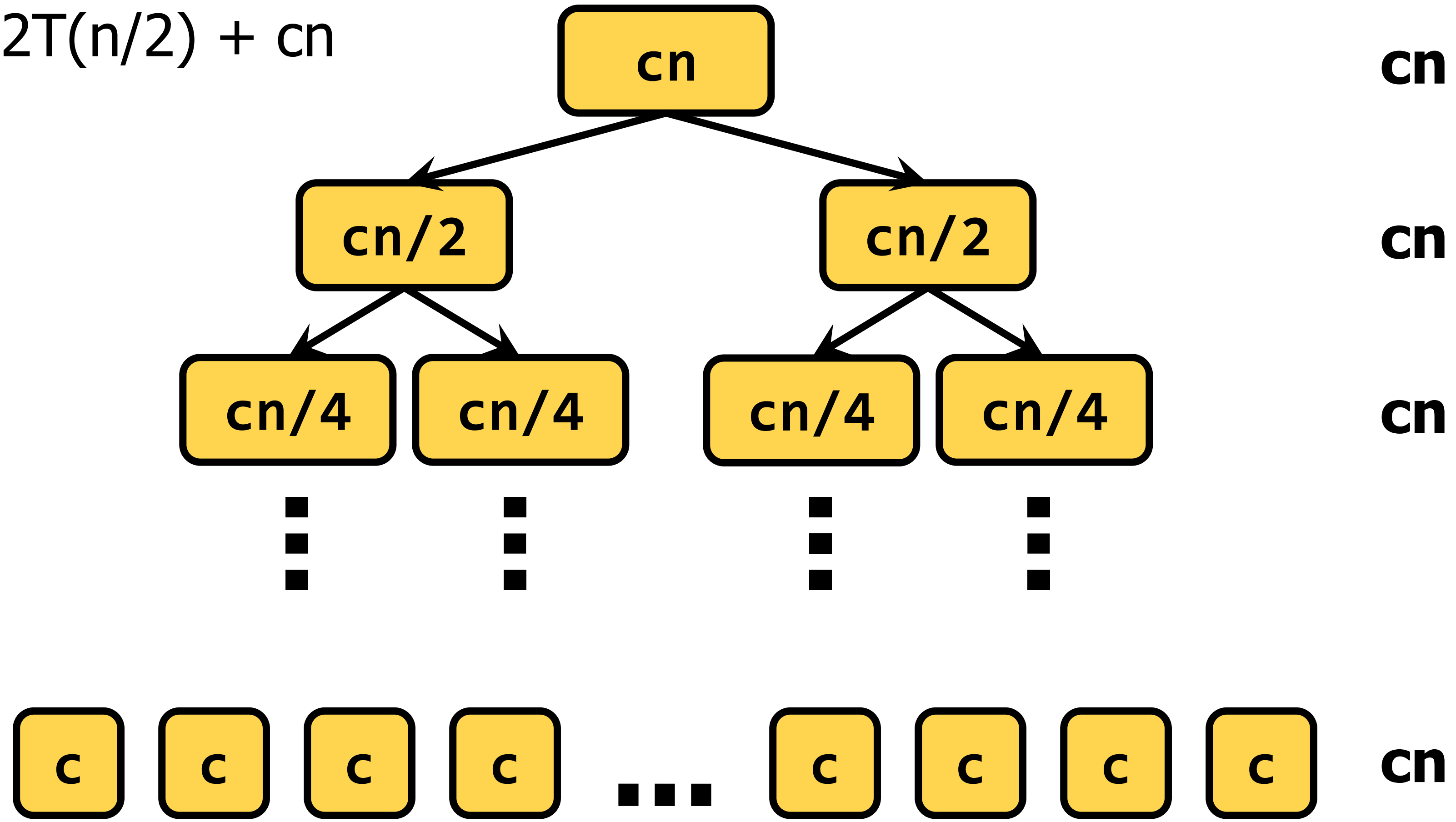
$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

Recursion Tree Method

$T(1) \leq c$

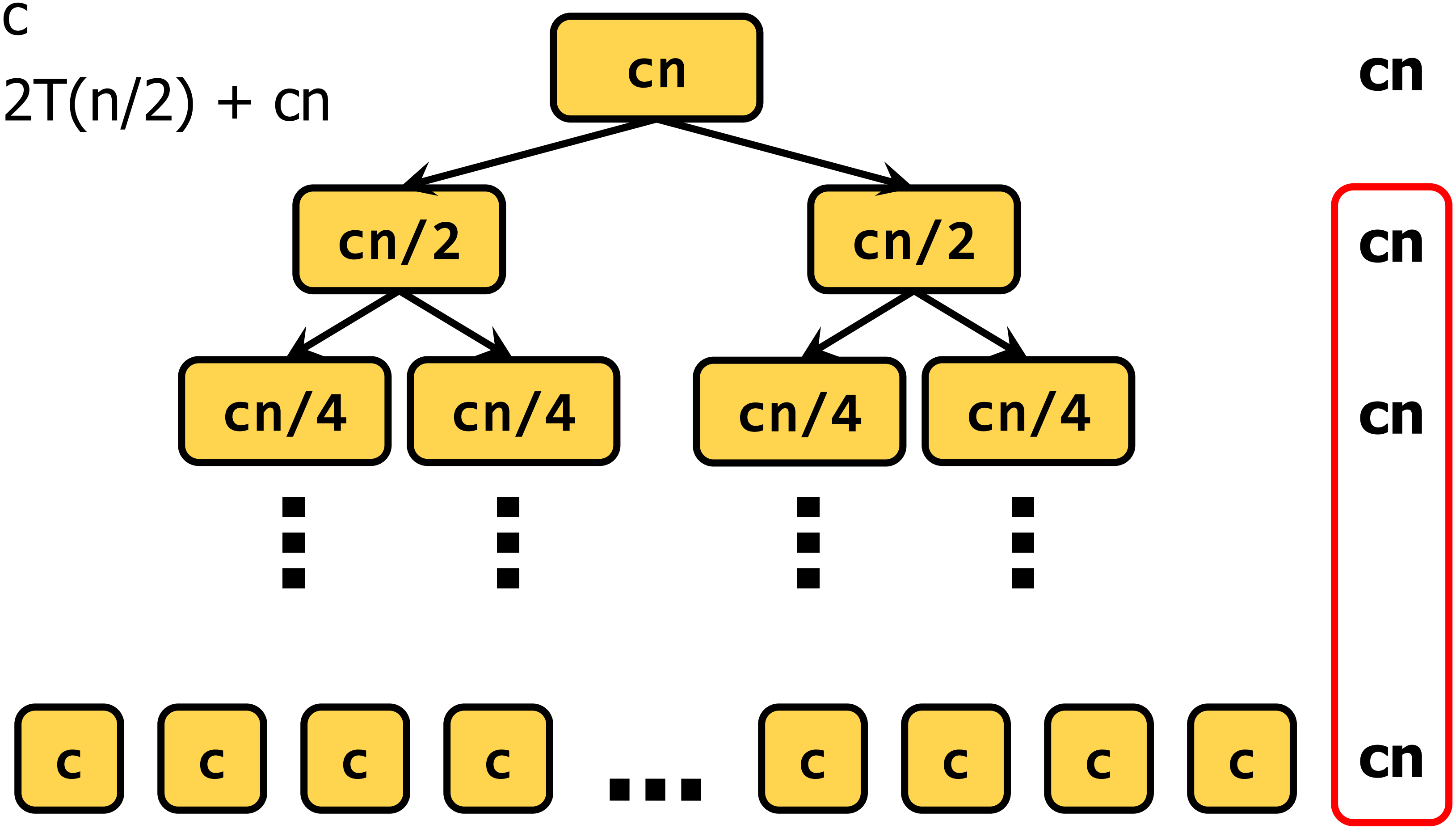
$T(n) \leq 2T(n/2) + cn$



Recursion Tree Method

$T(1) \leq c$

$T(n) \leq 2T(n/2) + cn$



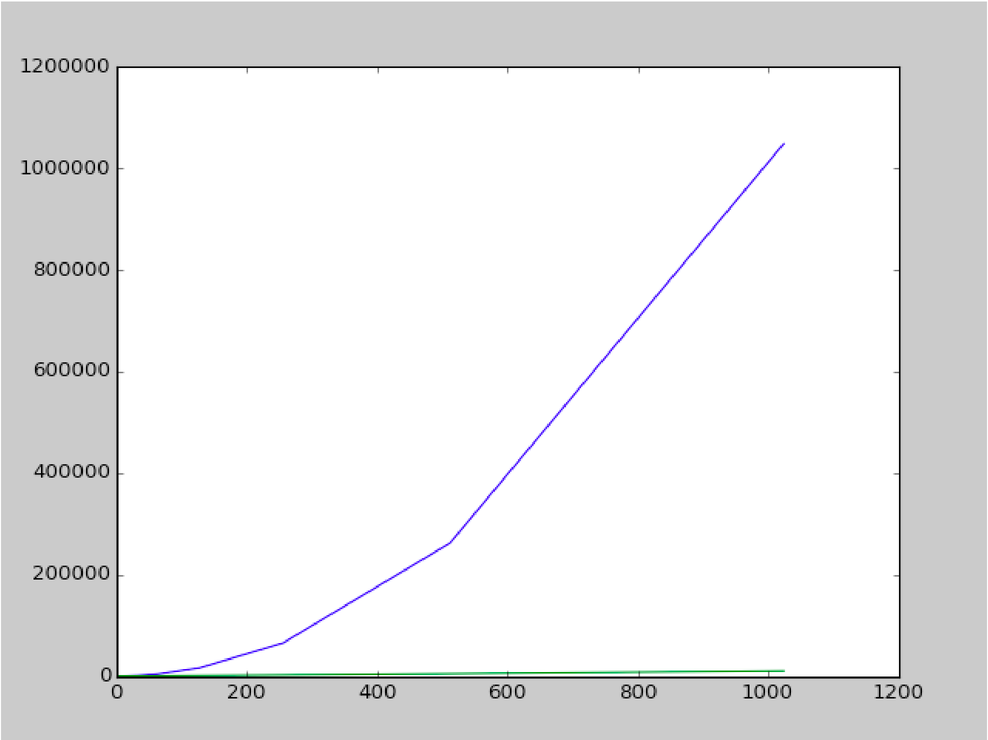
Total work: $cn \log_2 n + cn$

Analyzing runtime

The best and worst-case runtime of `mergesort` is $\Theta(n \log n)$.

The worst-case runtime of `insertion_sort` was $\Theta(n^2)$.

THIS IS A SIGNIFICANT (HUGE) IMPROVEMENT!!



Debugging
Python programs

Program for prime factors

- Break a number n into its prime factors
  - Iterate over each candidate p increasingly, and try to divide the current n (=x) by p

```python
def getPrimeFactors(n):
    result=[]
    x=n
    p=2
    while x!=1:
        if x%p==0:
            result.append(p)
            x=x/p
        else:
            p=p+1
    return(result)



L=getPrimeFactors(20)
print(L)
```

What is the shortest code for this task?

- Break a number n into its prime factors
- Use at most FIVE lines of code …

```
n=20
for x in range(2,n):
    while n%x==0:
        n=int(n/x)
        print(x)
```

Which version is better?

- Major goal is clarity of the code
  - Only afterwards, once you obtain working code, you start to *optimize*!
    - Speed, length, etc.
- Remember
  - “Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.”

Donald Knuth

Overview

- Types of Errors:
  - Syntax errors
  - Runtime Errors
  - Logical Errors

- The first two types lead to stack traces!

Understanding stack traces

- They tell you:
  - Where is the syntax/runtime error?
- Do not help for logical errors ☹

```
1  def getPrimeFactors(n):
2      result=[]
3      x=n
4      p=2
5      while x!=1:
6          if x%p==0:
7              result.add(p)
8              x=x/p
9          else:
10             p=p+1
11     return(result)
12
13
14
15 L=getPrimeFactors(20)
16 print(L)
```

```
In [9]: runfile('C:/Users/basti/Downloads/WinPython-64bit-3.6.2.0Qt5/settings/test.py', wdir='C:/Users/basti/Downloads/WinPython-64bit-3.6.2.0Qt5/settings')
Traceback (most recent call last):

  File "<ipython-input-9-181b47aba674>", line 1, in <module>
    runfile('C:/Users/basti/Downloads/WinPython-64bit-3.6.2.0Qt5/settings/test.py', wdir='C:/Users/basti/Downloads/WinPython-64bit-3.6.2.0Qt5/settings')

  File "C:\Users\basti\Downloads\WinPython-64bit-3.6.2.0Qt5\python-3.6.2.amd64\lib\site-packages\spyder\utils\site\sitecustomize.py", line 688, in runfile
    execfile(filename, namespace)

  File "C:\Users\basti\Downloads\WinPython-64bit-3.6.2.0Qt5\python-3.6.2.amd64\lib\site-packages\spyder\utils\site\sitecustomize.py", line 101, in execfile
    exec(compile(f.read(), filename, 'exec'), namespace)

  File "C:/Users/basti/Downloads/WinPython-64bit-3.6.2.0Qt5/settings/test.py", line 15, in <module>
    L=getPrimeFactors(20)

  File "C:/Users/basti/Downloads/WinPython-64bit-3.6.2.0Qt5/settings/test.py", line 7, in getPrimeFactors
    result.add(p)

AttributeError: 'list' object has no attribute 'add'
```

Understanding stack traces

- A stack trace is a nested list of function calls plus an error message
  - Usually, the inner/lowest function call causes the actual error
    - Note that you even get the line number!
  - The type of error is indicated by the error message

```
In [9]: runfile('C:/Users/basti/Downloads/WinPython-64bit-3.6.2.0Qt5/settings/test.py', wdir='C:/Users/basti/Downloads/WinPython-64bit-3.6.2.0Qt5/settings')
Traceback (most recent call last):

  File "<ipython-input-9-181b47aba674>", line 1, in <module>
    runfile('C:/Users/basti/Downloads/WinPython-64bit-3.6.2.0Qt5/settings/test.py', wdir='C:/Users/basti/Downloads/WinPython-64bit-3.6.2.0Qt5/settings')

  File "C:\Users\basti\Downloads\WinPython-64bit-3.6.2.0Qt5\python-3.6.2.amd64\lib\site-packages\spyder\utils\site\sitecustomize.py", line 688, in runfile
    execfile(filename, namespace)

  File "C:\Users\basti\Downloads\WinPython-64bit-3.6.2.0Qt5\python-3.6.2.amd64\lib\site-packages\spyder\utils\site\sitecustomize.py", line 101, in execfile
    exec(compile(f.read(), filename, 'exec'), namespace)

  File "C:/Users/basti/Downloads/WinPython-64bit-3.6.2.0Qt5/settings/test.py", line 15, in <module>
    L=getPrimeFactors(20)

  File "C:/Users/basti/Downloads/WinPython-64bit-3.6.2.0Qt5/settings/test.py", line 7, in getPrimeFactors
    result.add(p)

AttributeError: 'list' object has no attribute 'add'
```

What errors/exceptions exist?

List of all exceptions (errors):

http://docs.python.org/3/library/exceptions.html#bltin-exceptions

Two other resources, with more details about a few of the errors:

http://inventwithpython.com/appendixd.html

http://www.cs.arizona.edu/people/mccann/errors-python

Common errors

- IndexError
  - Raised when a sequence subscript is out of range.
- KeyError
  - Raised when a mapping (dictionary) key is not found in the set of existing keys.
- NameError
  - Raised when a local or global name is not found.
- SyntaxError
  - Raised when the parser encounters a syntax error.
- IndentationError
  - Base class for syntax errors related to incorrect indentation.
- TypeError
  - Raised when an operation or function is applied to an object of inappropriate type.
- KeyboardInterrupt
  - Raised when the user hits the interrupt key (normally Control-C or Delete).

Where is the bug? Divide and conquer

Where is the defect (or “bug”)?

- Your goal is to find the one place that it is
- Finding a defect is often harder than fixing it

- Initially, the defect might be anywhere in your program
  - It is impractical to find it if you have to look everywhere
- Idea: **bit by bit reduce the scope of your search**
- Eventually, the defect is localized to a few lines or one line
  - Then you can understand and fix it

- Four ways to divide and conquer:
  1. In the program code
  2. During the program execution
  3. In test cases
  4. During the development history

Divide and conquer in the program code

- Localize the defect to part of the program
- Code that isn't executed cannot contain the defect

Three approaches:

- Use print statements to see which parts are still executed
- Test one function at a time
- Split complex expressions into simpler ones

Example: Failure in

```
result = print(f(list(g(input()))))
```

Change it to

```
user_input=input()
gresult=g(user_input)
print("Result of g:",gresult)
L=list(gresult)
fresult=f(L)
print(fresult)
```

Divide and conquer in the program code

- Localize the defect to part of the program
- Code that isn't executed cannot contain the defect

Three approaches:

- Use print statements to see which parts are still executed
- Test one function at a time
-

Important hint for many of you:

Deactivate user input during debugging!

Change it to

```
user_input=60 #input()
gresult=g(user_input)
print("Result of g:",gresult)
L=list(gresult)
fresult=f(L)
print(fresult)
```

Divide and conquer in execution time via print (or "logging") statements

- A sequence of **`print`** statements is a record of the execution of your program
- The **`print`** statements let you see and search multiple moments in time
- Print statements are a useful technique, in moderation
- Be disciplined
  - Too much output is overwhelming rather than informative
  - Remember the scientific method: have a reason (a hypothesis to be tested) for each print statement
  - Don't *only* use print statements

Divide and conquer in test cases

- Your program fails when run on some large input
  - It's hard to comprehend the error message
  - The log of print statement output is overwhelming
- Try a smaller input
  - Choose an input with some but not all characteristics of the large input
  - Example: duplicates, zeroes in data, …
  - Find a Minimum Working Example (MWE)
    - Everything needed to reproduce the bug should be there, but nothing else

Divide and conquer in development history

- The code used to work (for some test case)
- The code now fails
- The defect is related to some line you changed

- This is useful only if you kept a version of the code that worked (use good names!)
- This is most useful if you have made few changes
- Moral: test often!
  - Fewer lines to compare
  - You remember what you were thinking/doing recently

A metaphor about debugging

If your code doesn't work as expected, then by definition you don't understand what is going on.

- You're lost in the woods.
- You're behind enemy lines.
- All bets are off.
- Don't trust anyone or anything.

Don't press on into unexplored territory -- go back the way you came!
(and leave breadcrumbs!)

Time-Saving Trick: Make Sure you're Debugging the Right Problem

- The game is to go from "working to working"
- When something doesn't work, STOP!
  - It's wild out there!
- FIRST: go back to the last situation that worked properly.
  - Rollback your recent changes and verify that everything still works as expected.
  - Don't make assumptions – by definition, you don't understand the code when something goes wrong, so you can't trust your assumptions.
  - You may find that even what previously worked now doesn't
  - Perhaps you forgot to consider some "innocent" or unintentional change, and now even tested code is broken

A bad timeline

- A works, so celebrate a little
- Now try B
- B doesn’t work
- Change B and try again
- Change B and try again
- Change B and try again

…

A better timeline

- A works, so celebrate a little
- Now try B
- B doesn't work
- *Rollback to A*
- Does A still work?
  - Yes: Find A' that is somewhere between A and B
  - No: You have *unintentionally changed something else*, and there's no point futzing with B at all!

These "innocent" and unnoticed changes happen more than you would think!

- You add a comment, and the indentation changes.
- You add a print statement, and a function is evaluated twice.
- You move a file, and the wrong one is being read
- You're on a different computer, and the library is a different version

Thank you very much!