Lecture 6: Algorithms

Beihang University

Algorithm Analysis

Chapter I.2, I.3, and I.4 of „Introduction to Algorithms“

Note

- In the following we will often use pseudocode!
- What is pseudocode?
  - An abstraction from software engineering details
  - High similarities with C/C++/Python
  - Does not deal with modularity, error handling, etc.
  - Sometimes uses shortcuts, which can cannot be executed directly, e.g., English sentences
  - Covers the **essence** of an algorithm

The formal problem of sorting

- Sorting is a fundamental operation in computer science

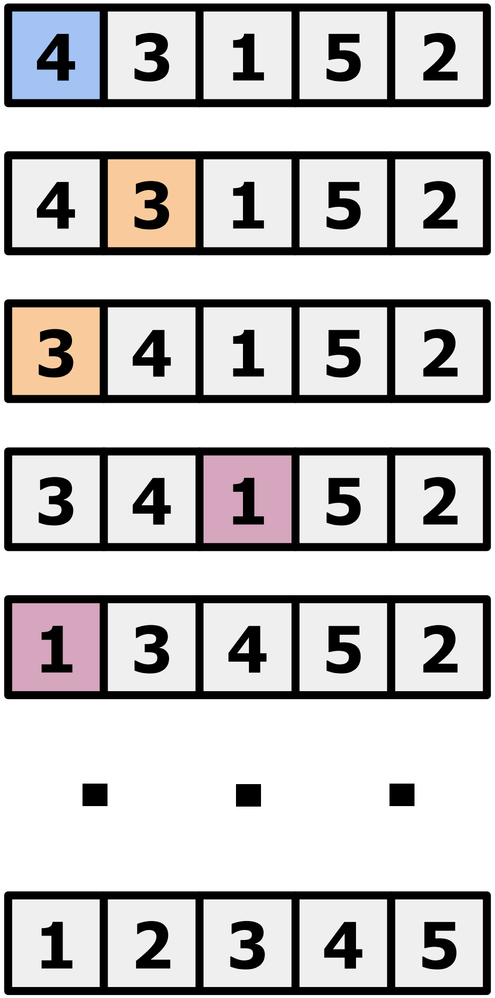
Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

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

The formal problem of sorting

- How to solve this problem?

Insertion sort



Let's sort an unsorted list of numbers **`A`**. The sublist **`A[0:0]`** is trivially sorted.

Look at the second element, **`A[1]`**.

Insert the element into a new position such that the sublist **`A[0:1]`** is sorted.

Now look at the third element, **`A[2]`**.

Insert it such that the sublist **`A[0:2]`** is sorted.

The entire array **`A[0:4]`** is sorted.

Insertion sort: Pseudocode

```
algorithm insertion_sort(list A):
  for i = 0 to length(A)-1:
    let cur_value = A[i]
    let 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

```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
```

Major questions regarding the algorithm

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

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

Short recap: Proofs

- How to prove the following statement?
  - For each natural number **n, we have that**

$$\mathbf{1 + 2 + \ldots + n = n(\ n + 1\ )/2}$$

Short recap: Proofs

- Proof by induction:
- Base case: $n=1$
  - $1 = 1(1+1)/2$
- Inductive case: $n \rightarrow n+1$
  - To show: $1 + 2 + \dots + n + (n + 1) = (n + 1)((n + 1) + 1)/2$
  - **IMPORTANT**: We can use $1 + 2 + \dots + n = n(n + 1)/2$!
  - Steps:

    $1 + 2 + \dots + n + (n + 1)$

    $= n(n + 1)/2 + (n + 1)$

    $= (n + 1) * (n + 2)/2$

    $= (n + 1) * ((n + 1) + 1)/2$

Proving Correctness

Algorithms often initialize, modify, or delete new data.

- In the case of insertion sort, it might be challenging for an untrained observer to formalize the notion of correctness since the manner in which the algorithm behaves depends on the input list.
- Is there a way to prove the algorithm works, without checking it for all (infinitely many) input lists?
- **To reason about the behavior of algorithms, it often helps to look for things that *don't* change.**
  - Notice that insertion sort maintains a sorted sublist, the length of which grows each iteration.
- This unchanging property is called an **invariant**.

Where is an invariant here?

```
algorithm insertion_sort(list A):
  for i = 0 to length(A)-1:
    let cur_value = A[i]
    let j = i - 1
    while j ≥ 0 and A[j] > cur_value:
      A[j+1] = A[j]
      j = j - 1
    A[j+1] = cur_value
```

Proving Correctness

For example, an **invariant** of the outer for-loop of insertion sort: At the start of iteration i of the outer for-loop, the first i elements of the list are sorted.

Sanity checks:

At the start of the third iteration (i.e. the iteration when i = 2), the first two elements of the list are sorted. True.

3	4	1	5	2

At the start of the fifth iteration (i.e. the iteration when i = 4), the first four elements of the list are sorted. True.

1	3	4	5	2

Proving Correctness

Less formally...

1. At the start of the first iteration, the first element of the array is sorted.
2. By construction, the i^{th} iteration puts element **`A[i]`** in the right place.
3. At the start of the $i = \text{length}(A)^{th}$ iteration (aka the end of the algorithm), the first length(A) elements are sorted.

Proving Correctness

More formally (rigorously) …

Outer invariant (for-loop): At the start of iteration i of the outer for-loop, the first i elements of the list are sorted.

Inner invariant (while-loop): At the start of iteration j of the inner while-loop, `A[0:j,j+2:i]` contains the same elements as the original sublist `A[0:i-1]`, still sorted, such that all of the values in the right sublist `A[j+2:i]` are greater than `cur_value`.

Proving Correctness

The theorem follows a consistent format:

Initialization:
The loop invariant starts out as true.

Maintenance:
If the loop invariant is true at step i, then it's true at step i+1.

Termination:
If the loop invariant is true at the end of the algorithm, this tells you something about what you're trying to prove.

Insertion sort

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

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

Analyzing runtime

```
algorithm insertion_sort(list A):
  for i = 1 to length(A):
    let cur_value = A[i]
    let j = i - 1
    while j > 0 and A[j] > cur_value:
      A[j+1] = A[j]
      j = j - 1
    A[j+1] = cur_value
```

O(n) work per iteration

O(n) iterations

Total work: $O(n^2)$

The Big-O notation

Big-O Notation

Big-O notation is a mathematical notation for upper-bounding a function's rate of growth. Informally, it can be determined by <u>ignoring constants and non-dominant growth terms</u>.

Examples

$n + 137 = O(n)$

$3n + 42 = O(n)$

$n^2 + 3n - 2 = O(n^2)$

$n^3 + 10n^2 \log n - 15n = ??$

$2^n + n^2 = ??$

Big-O Notation

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

Then **f(n) = O(g(n)) iff**

$$\exists n_0 \in \mathbf{N}, c \in \mathbf{R}.$$

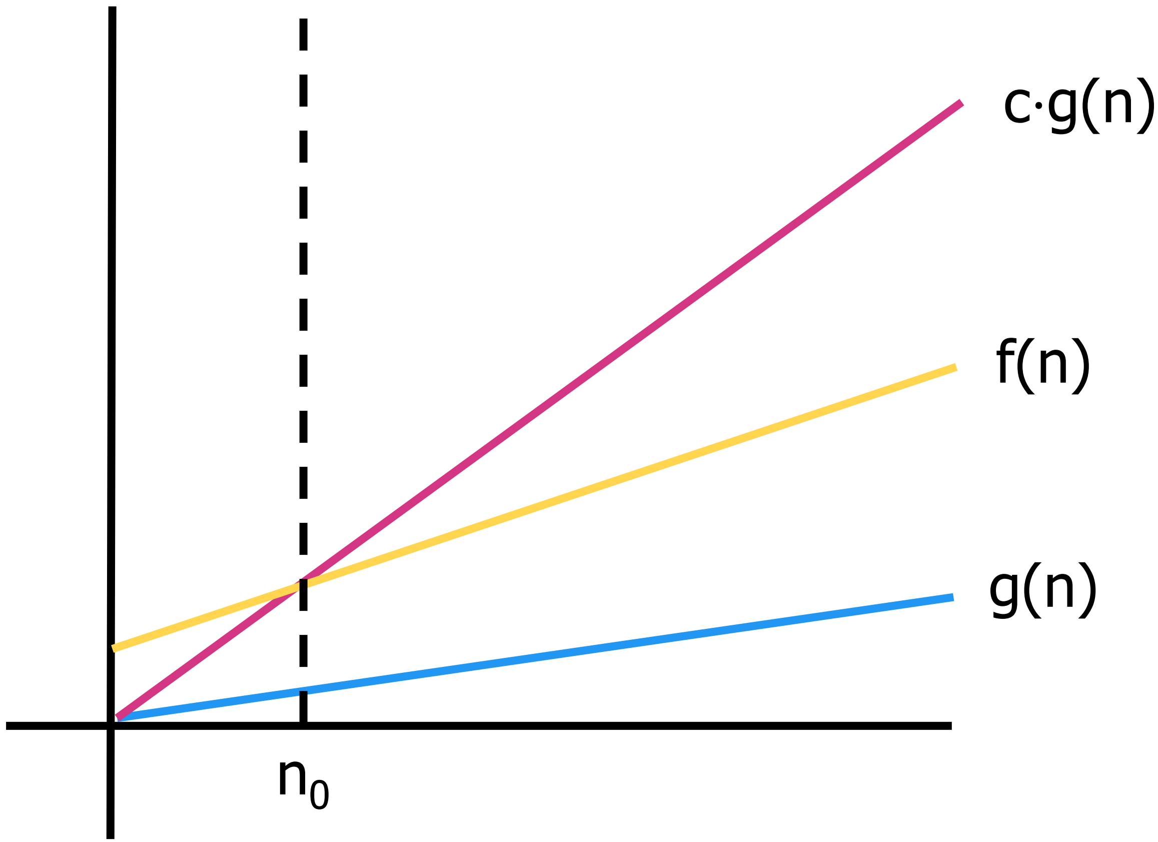
$$\forall n \in \mathbf{N}.$$

$$(n \geq n_0 \to f(n) \leq c \cdot g(n))$$

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

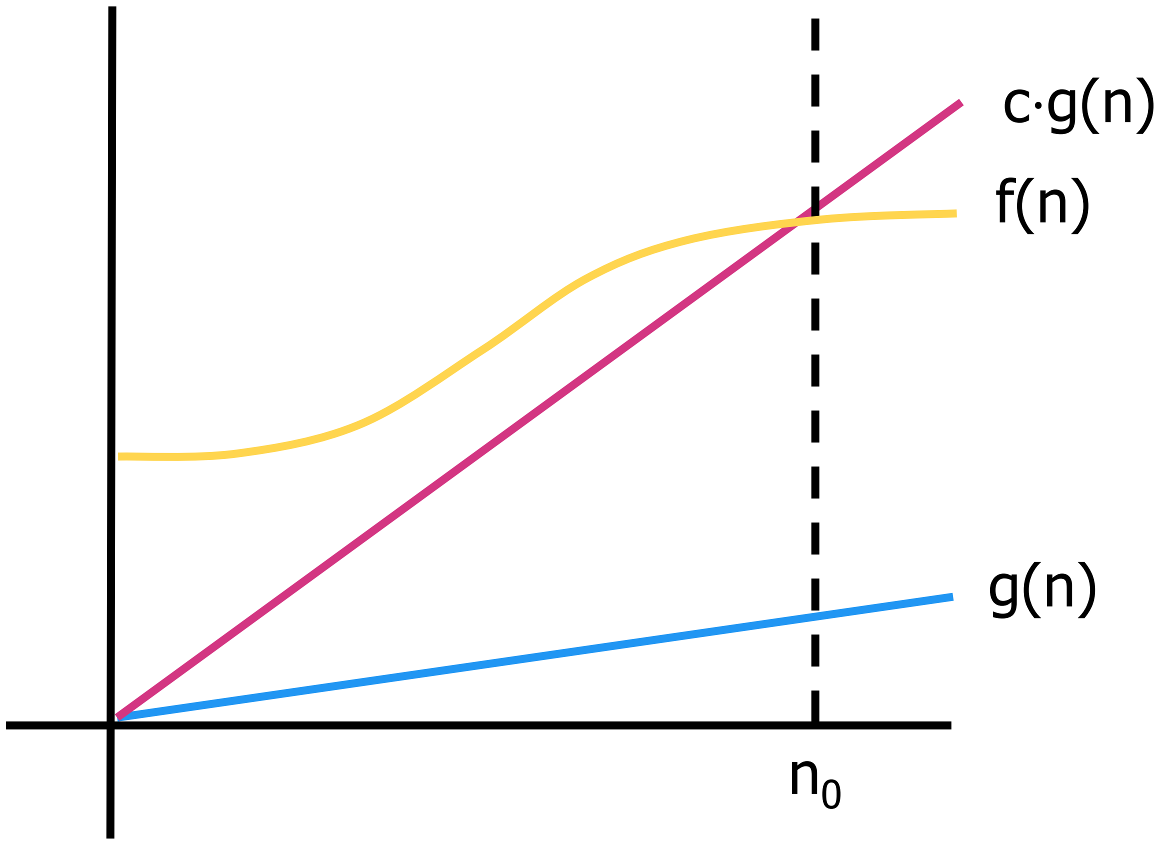
Big-O Notation

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



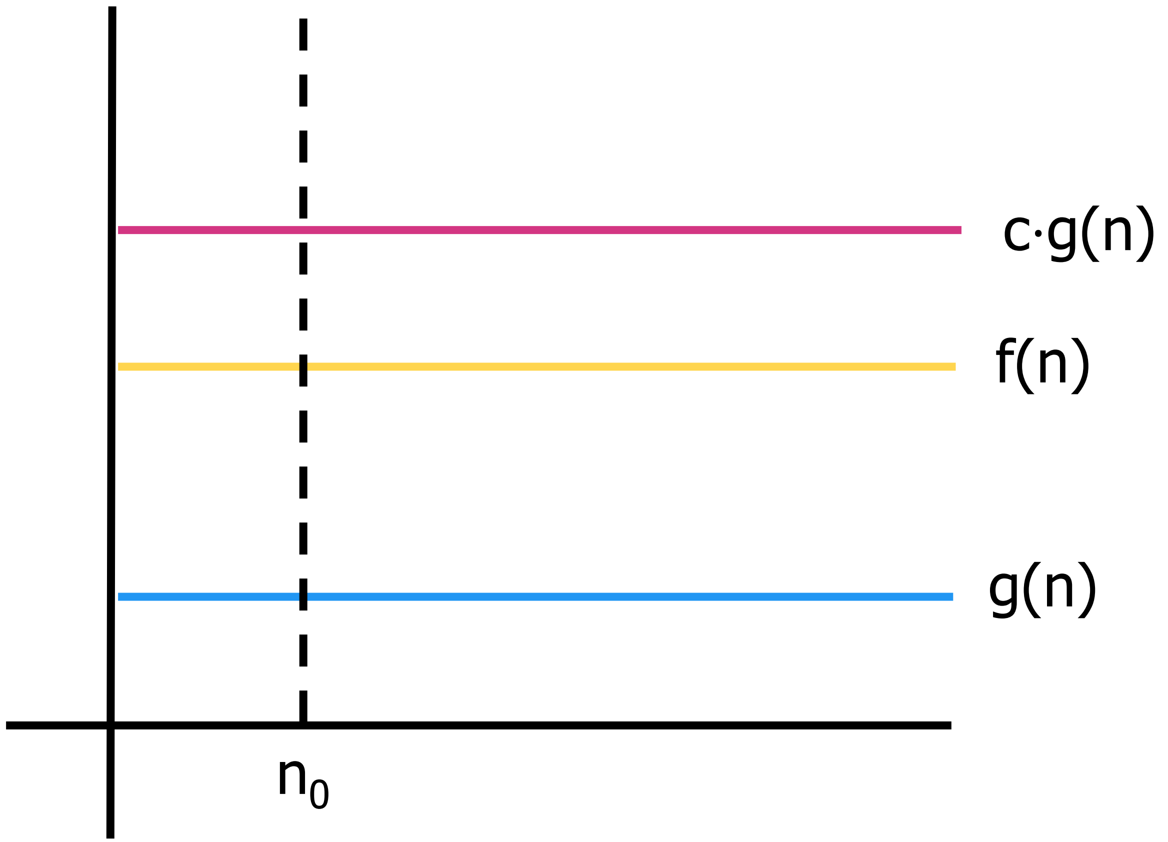
Big-O Notation

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



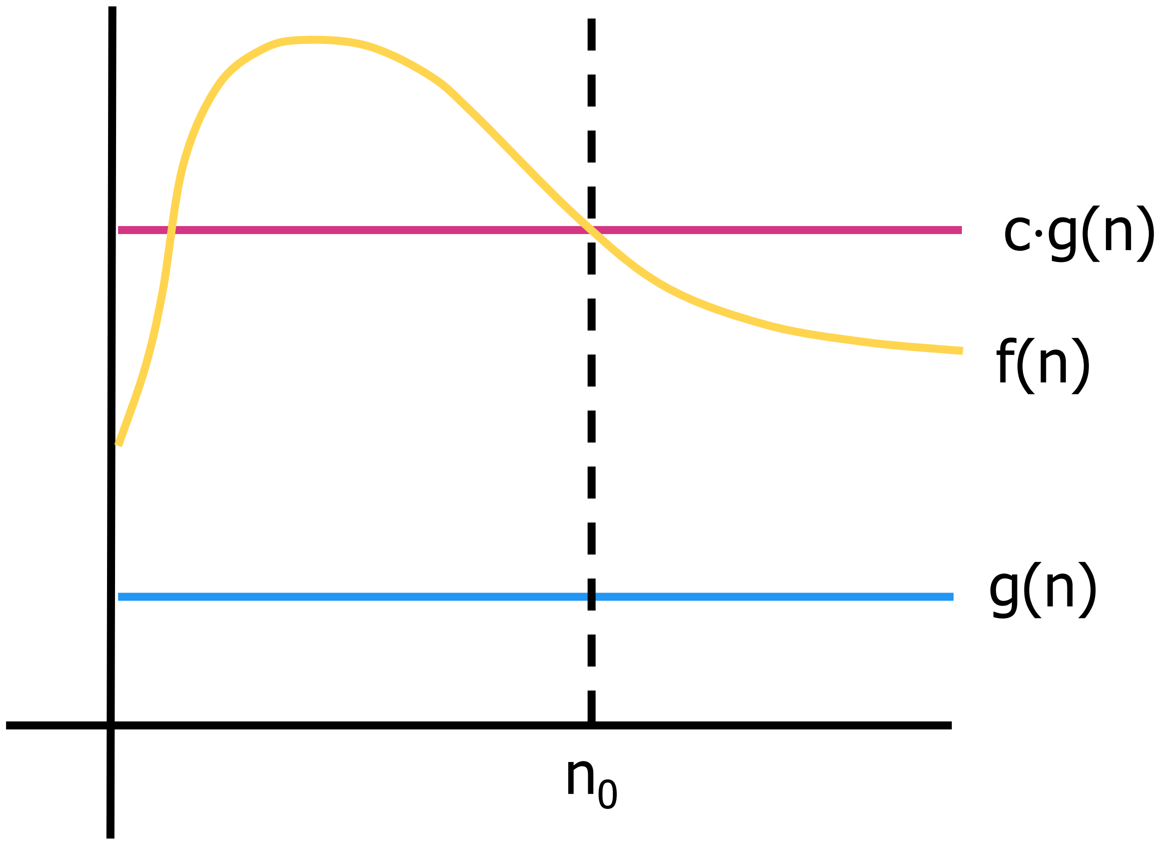
Big-O Notation

$$f(n) = O(g(n)) \quad \text{iff} \quad \exists n_0 \in N, c \in R. \quad \forall n \in N. \quad (n \geq n_0 \rightarrow f(n) \leq c \cdot g(n))$$



Big-O Notation

$$f(n) = O(g(n)) \quad \text{iff} \quad \exists n_0 \in N, c \in R. \quad \forall n \in N. \quad (n \geq n_0 \rightarrow f(n) \leq c \cdot g(n))$$



Big-O Notation

To prove $f(n) = O(g(n))$, show that there exists a c and n_0 that satisfies the definition.

Suppose $f(n) = n$ and $g(n) = n \log n$. We prove that $f(n) = O(g(n))$.

Consider the values $c = 1$ and $n_0 = 2$. We have $n \leq cn \log n$ for $n \geq n_0$ since n is positive and $1 \leq \log n$ for $n \geq 2$.

To prove $f(n) \neq O(g(n))$, proceed by contradiction.

Suppose $f(n) = n^2$ and $g(n) = n$. We prove that $f(n) \neq O(g(n))$.

Suppose there exists some c and n_0 such that for all $n \geq n_0$, $n^2 \leq cn$. Consider $n = \max\{c, n_0\} + 1$. Then $n \geq n_0$, but we have $n > c$, which implies that $n^2 > cn$. Contradiction!

The Big-Ω notation

Big-Ω Notation

Let f, g: $N \rightarrow N$.

Then $f(n) = \Omega(g(n))$ iff

$\exists n_0 \in N, c \in R.$

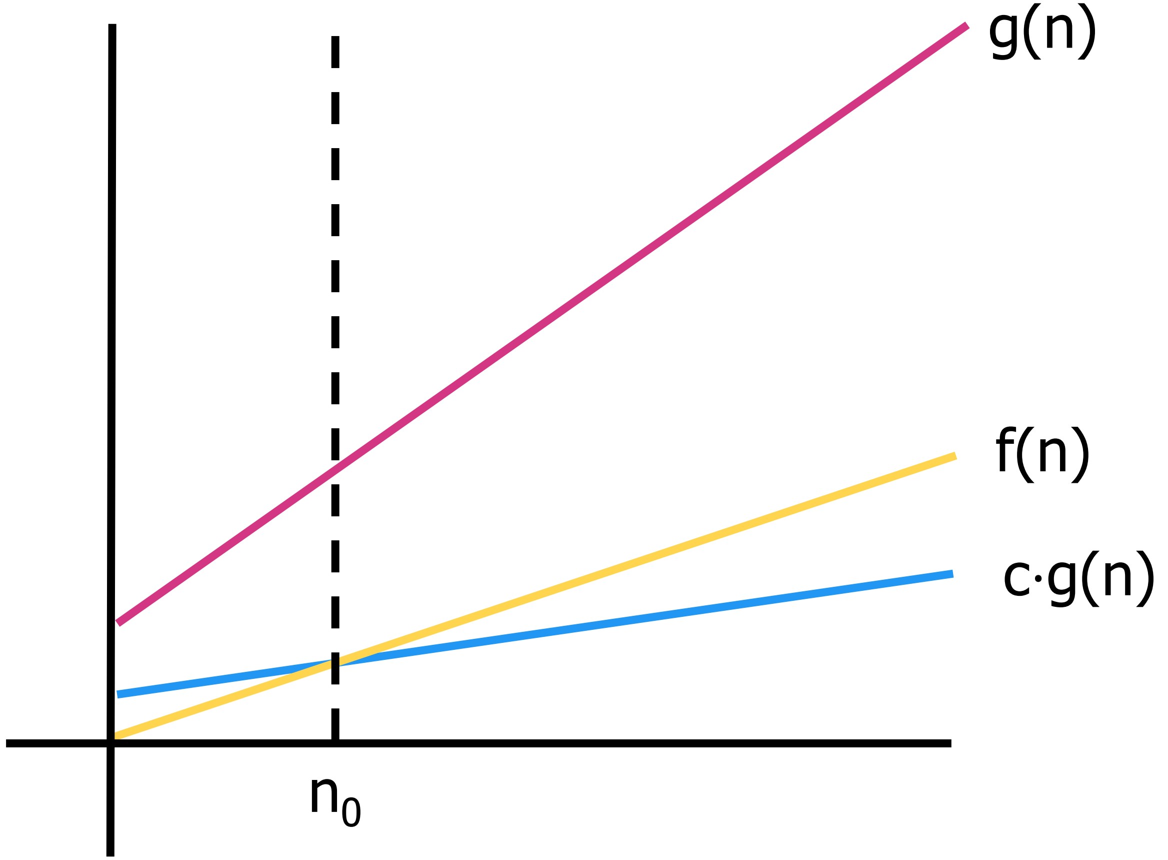
$\forall n \in N.$

$(n \geq n_0 \rightarrow f(n) \geq c \cdot g(n))$

Intuitively, this means that f(n) is lower-bounded by g(n) aka f(n) is "at least as big as" g(n).

Big-Ω Notation

$f(n) = \Omega(g(n))$ iff $\exists n_0 \in N, c \in R. \quad \forall n \in N. \quad (n \geq n_0 \rightarrow f(n) \geq c \cdot g(n))$



The Big-Θ notation

Big-Θ Notation

$f(n) = \Theta(g(n))$ iff both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

More verbosely, let $f, g: N \to N$.

Then $f(n) = \Theta(g(n))$ iff

$\exists n_0 \in N$, c_1 and $c_2 \in R$.

$\forall n \in N$.

$(n \geq n_0 \to c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n))$

Intuitively, this means that $f(n)$ is lower and upper-bounded by $g(n)$ aka $f(n)$ is "the same as" $g(n)$.

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 $\Theta(n^2)$.

Divide and Conquer

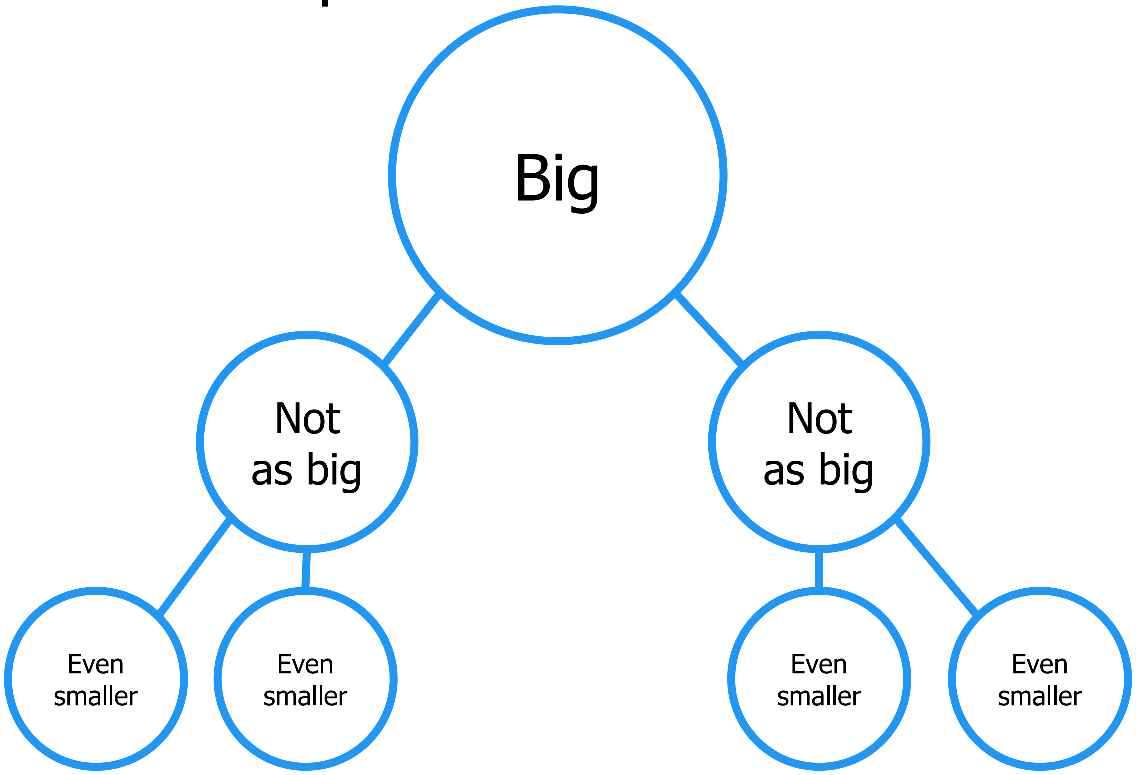
Recap: Sorting

- What is the best/worst case runtime complexity for bubble sort and insertion sort?

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(???)

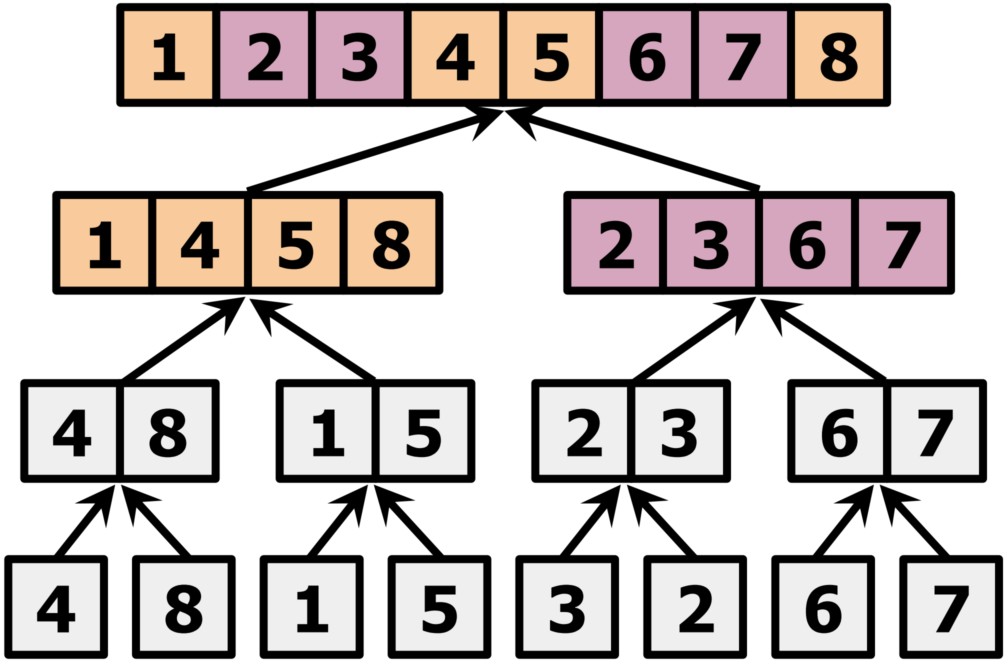
```
algorithm merge(list A, list B):
  let result = []
  while both A and B are nonempty:
    if head(A) < head(B):
      append head(A) to result
      pop head(A) from A
    else:
      append head(B) to result
      pop head(B) from B
  append remaining elements in A to result
  append remaining elements in B to result
  return result
```

Total work: $\Theta(a+b)$, where a and b are the lengths of lists A and B.

Mergesort

Tracing the recursive calls …

Sorted list



Original list

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

Less formally (explain it to your co-worker) ...

Consider a list of length k. If n 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?

$$T(1) \le c$$

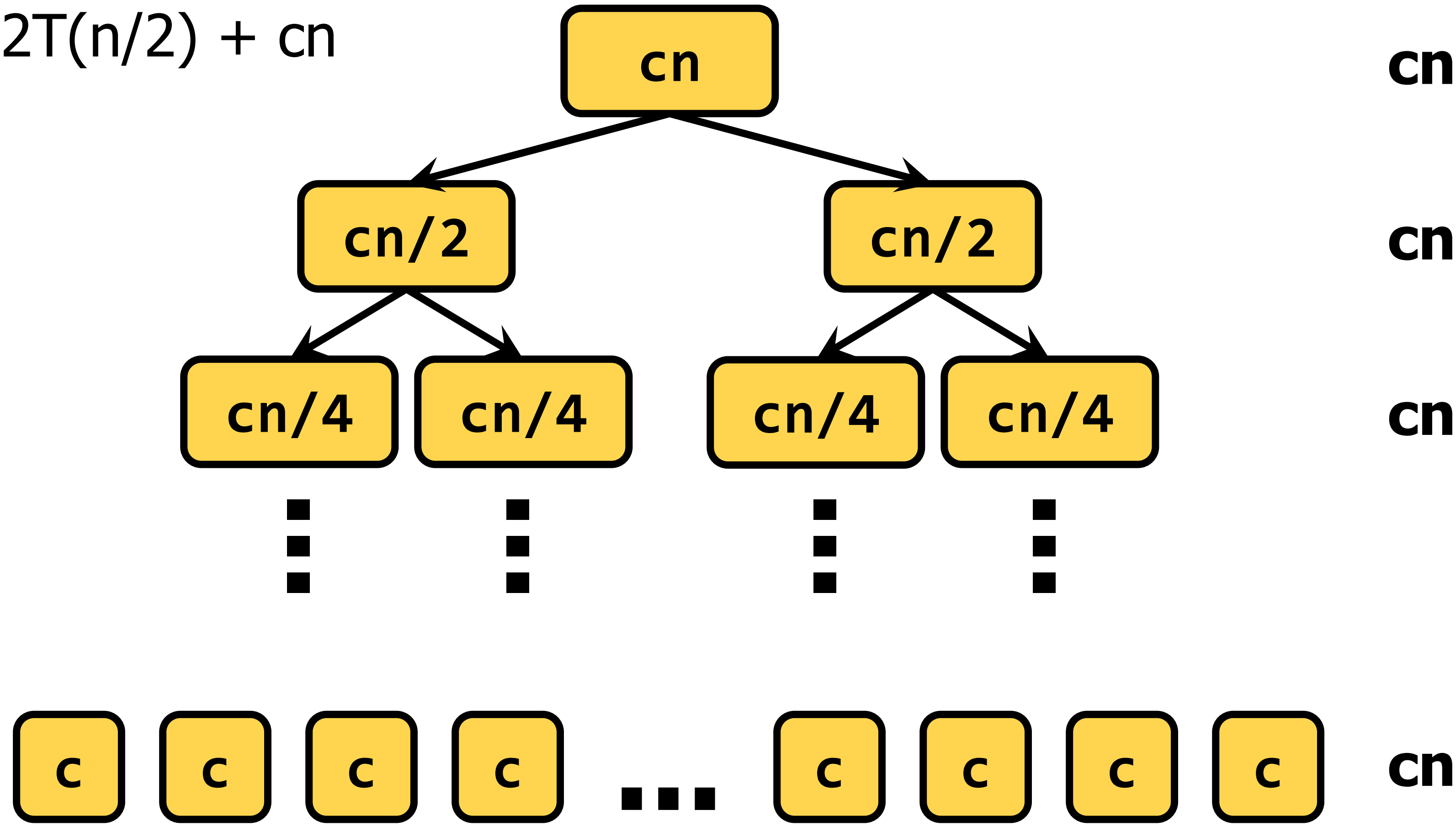
$$T(n) \le 2T(n/2) + cn$$

We'll use the **recursion tree method**.

Recursion Tree Method

$T(1) \leq c$

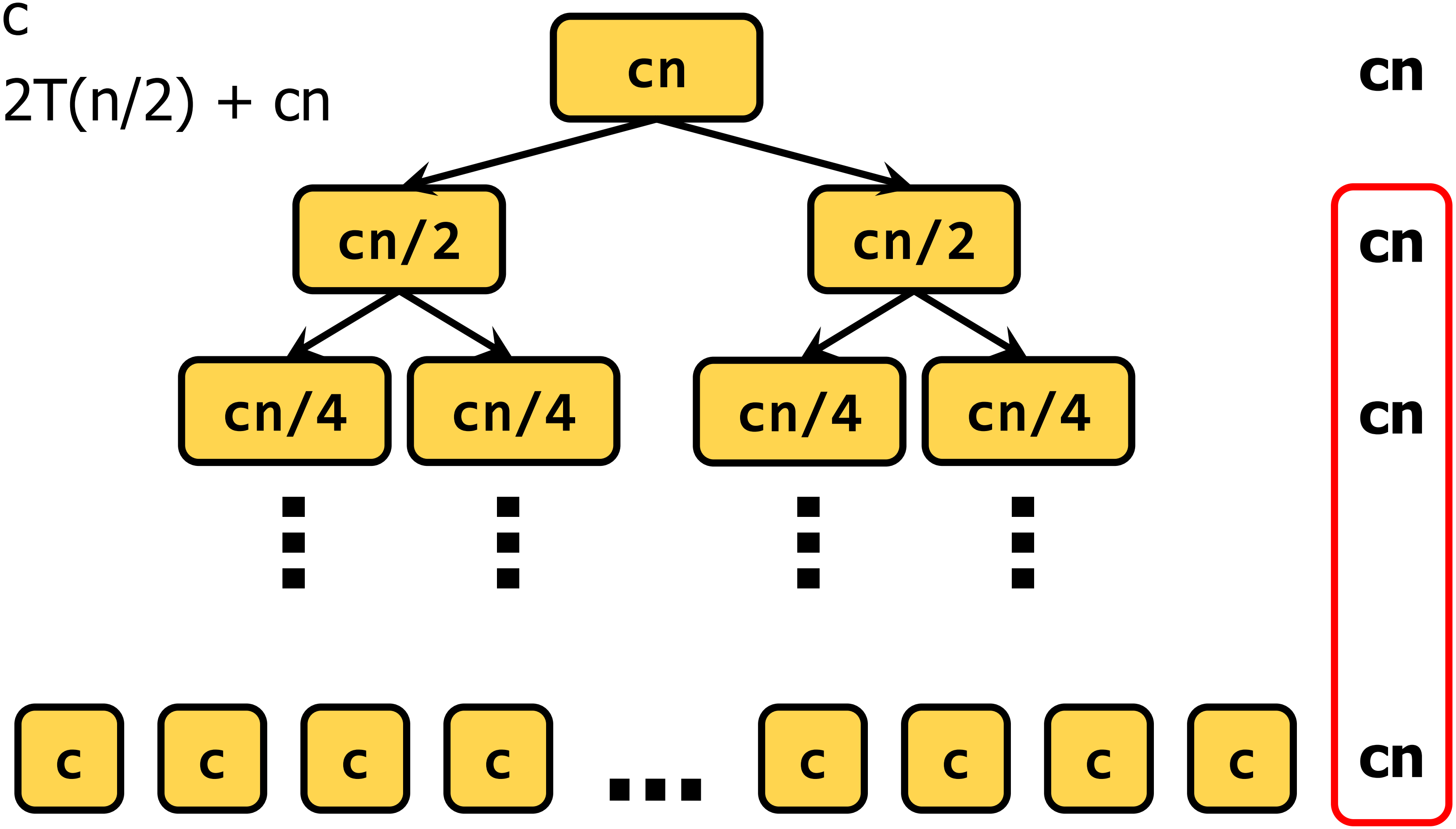
$T(n) \leq 2T(n/2) + cn$



Recursion Tree Method

$T(1) \le c$

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



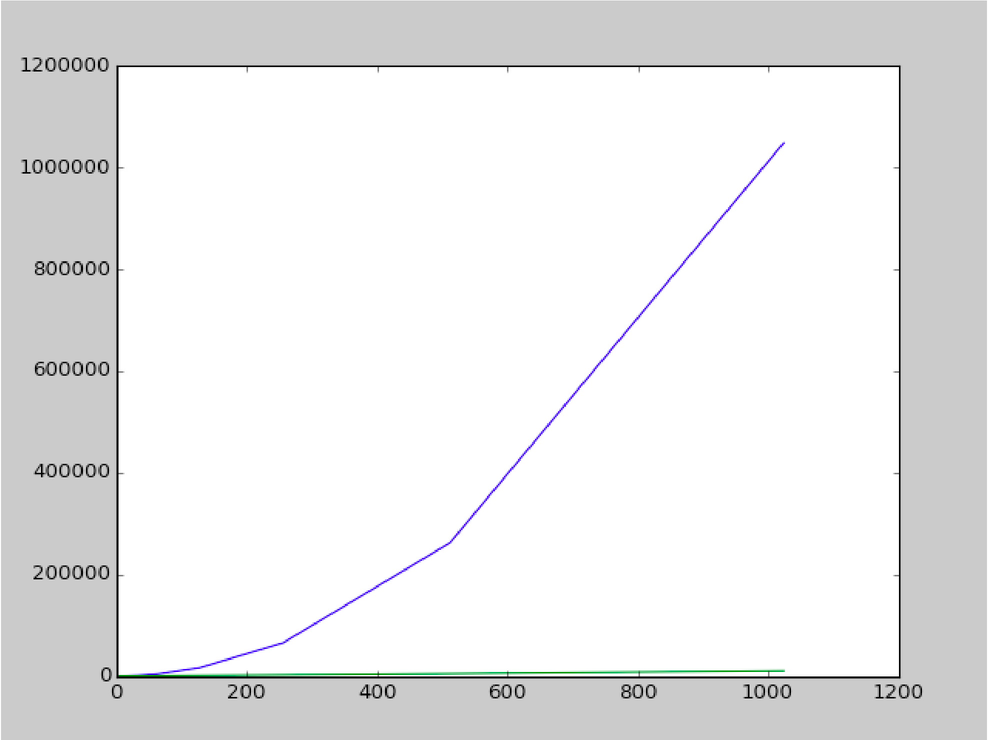
Total work: $cn \log_2 n + cn$

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 HUGE IMPROVEMENT!!



Thank you very much!