Introduction to Algorithms

For DS 110

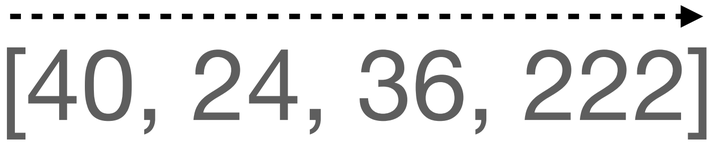
klgold

What is an algorithm?

- An algorithm is “any **well-defined computational procedure** that takes some value or set of values as input and produces some value or set of values as output” (Cormen, Leiserson, Rivest, & Stein)
- A *good* algorithm produces the **best possible output** and **avoids excess computation or use of memory**
- As the input gets big, good algorithms are needed to run in reasonable time

Algorithm: Linear Search

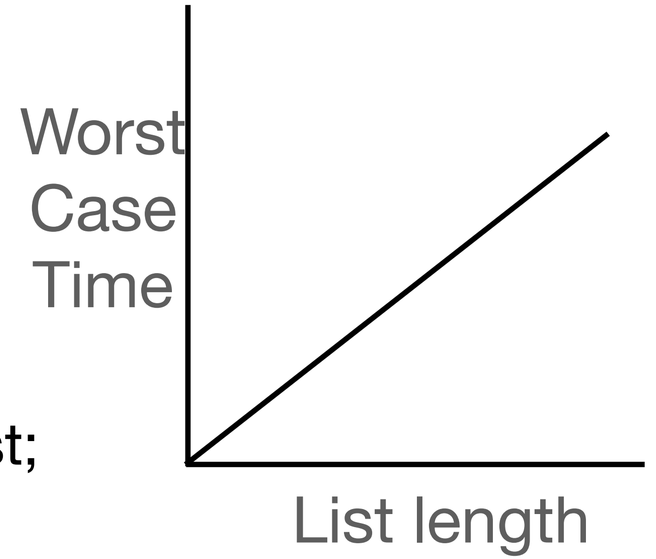
- Goal: Find index of a number in a list of numbers
- General strategy: Go down the list until the goal item is found



- Pseudocode:

```
LinearSearch(goal, list):
    for idx in 0 to (length of list - 1):
        if list[idx] == goal:
            return idx
    return NOT FOUND
```

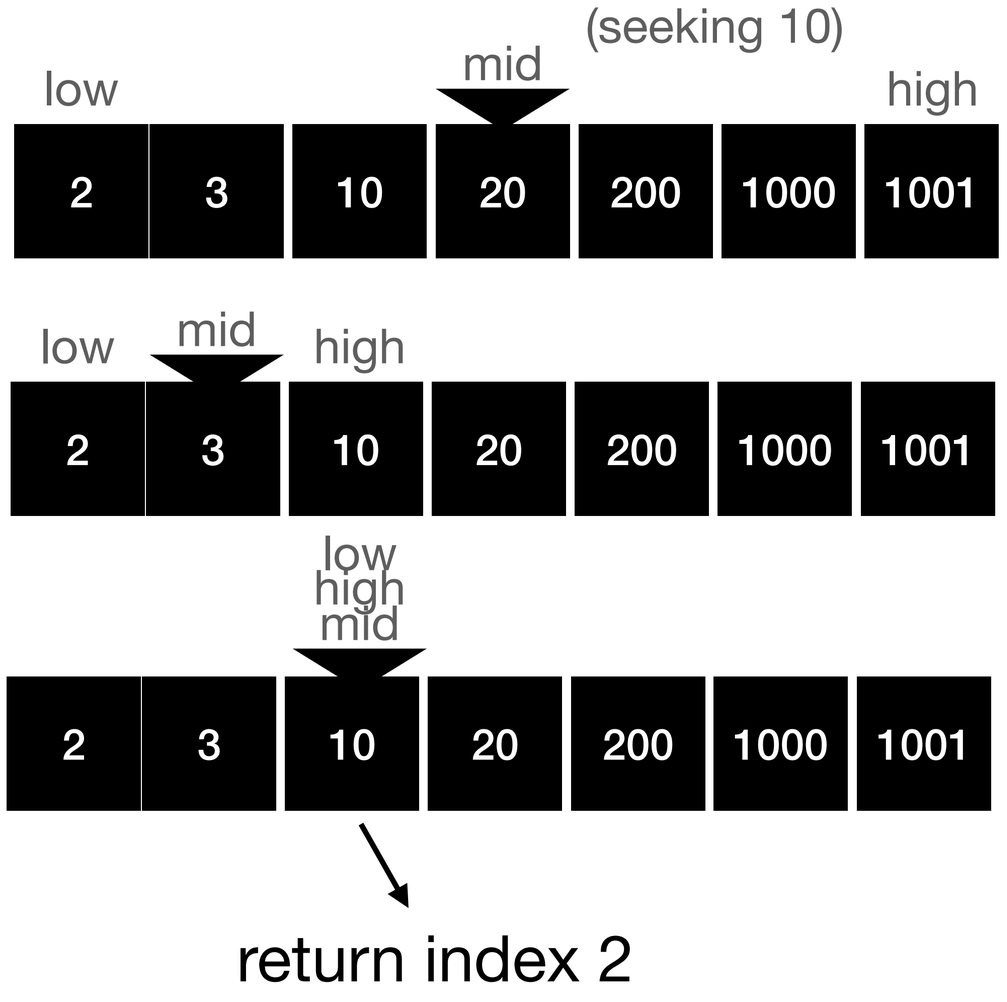
- If we fail to find the number, we took n steps down the list; **worst case running time is linear** in the size of the list



Algorithm: Binary Search

- Goal: Find index of a number in a *sorted* Python list or array of numbers
- General strategy: With each step, jump to the middle of the "still good" pile, determine which half is irrelevant, make the other half the "still good" pile
  - Pseudocode:

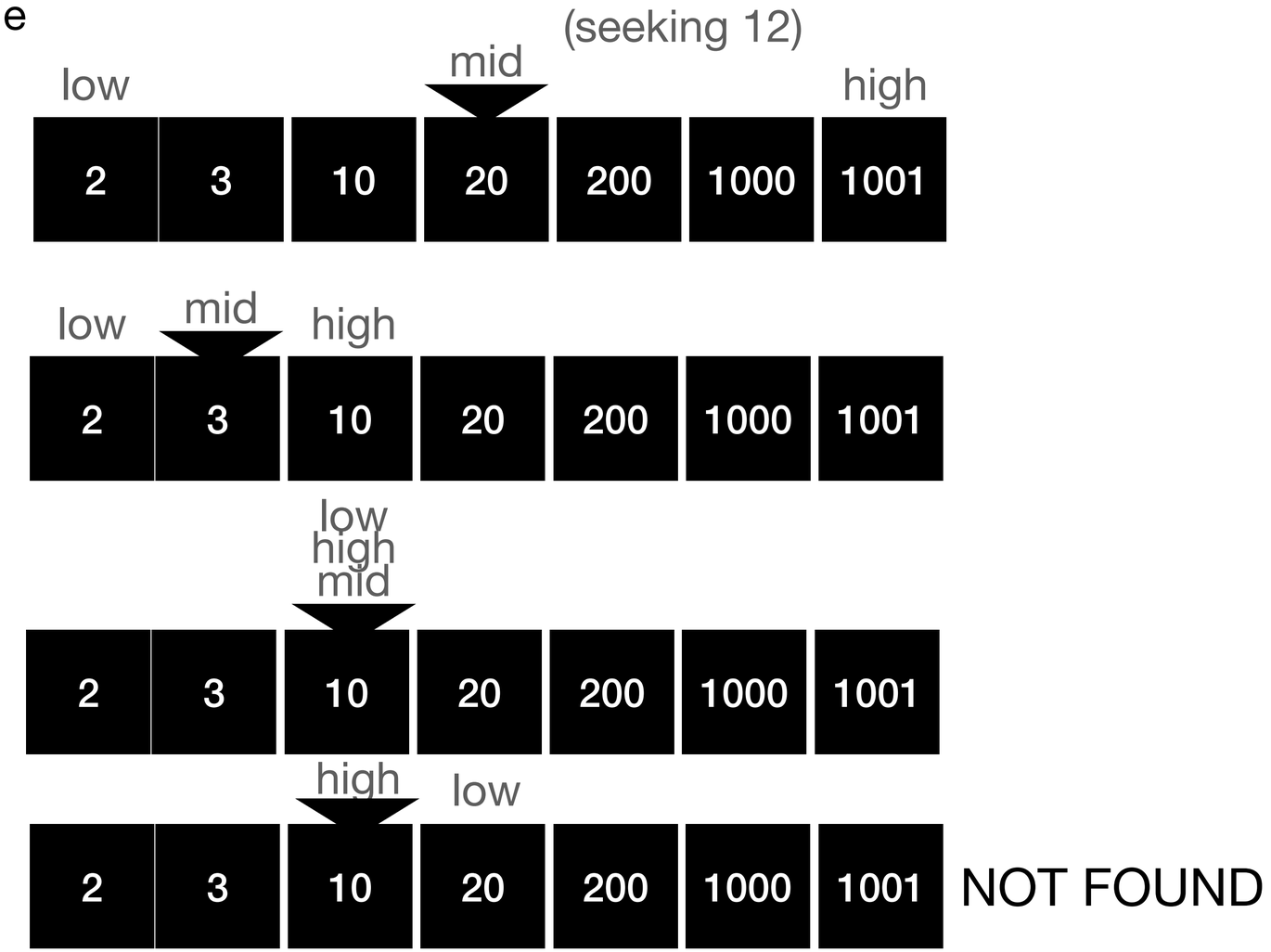
```
BinarySearch(goal, list):
    low = 0
    high = length of list - 1
    while True:
        if low > high return NOT FOUND
        midpoint = floor((low+high)/2)
        if list[midpoint] == goal:
            return midpoint
        else if list[midpoint] < goal:
            low = midpoint + 1
        else:  // list[midpoint] > goal
            high = midpoint - 1
```



Algorithm: Binary Search

- Goal: Find index of a number in a *sorted* Python list or array of numbers
- General strategy: With each step, jump to the middle of the "still good" pile, determine which half is irrelevant, make the other half the "still good" pile
  - Pseudocode:

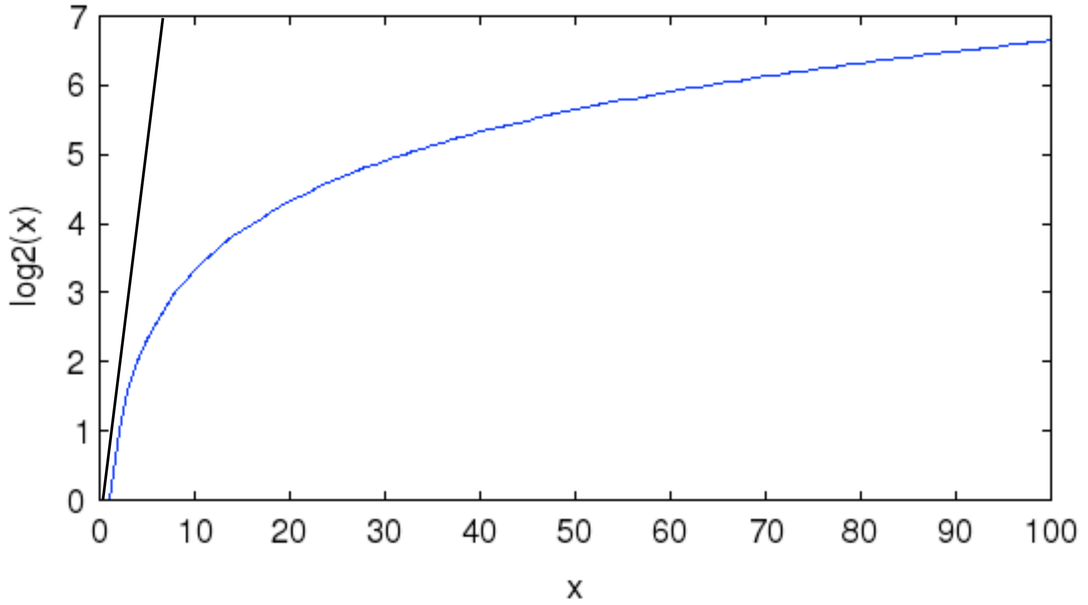
```
BinarySearch(goal, list):
    low = 0
    high = length of list - 1
    while True:
        if low > high return NOT FOUND
        midpoint = floor((low+high)/2)
        if list[midpoint] == goal:
            return midpoint
        else if list[midpoint] < goal:
            low = midpoint + 1
        else:  // list[midpoint] > goal
            high = midpoint - 1
```



Binary Search: Analysis of Running time

- Typically analyze the worst case
  - This is when the value is not found
- Half the array is "thrown out" each time - the number of elements to consider is halved until there is 1 left
- The number of times N must be divided in half to reach 1 is the **logarithm base 2** of N
  E.g. 64 ... 32 ... 16 ... 8 ... 4 ... 2 ... 1 6 divisions by 2, and $\log_2 64 = 6$
- So the running time is logarithmic

Logarithmic running times scale better than linear times



- Logarithms grow more slowly than linear.
- A logarithmic running time means many fewer operations as input gets large
  - E.g. $\log_2(1024) = 10$, which is much smaller than 1024
  - So binary search is the superior algorithm to linear search

Broad categories of running times

- **Constant time**: Takes the same amount of time regardless of input size
  Example: Indexing into an array

- **Logarithmic time**: Number of operations grows very slowly with size of input; algorithm is quick
  Example: Binary search

- **Linear time**: Operations grow proportionally to size of input; reasonably quick
  Example: Linear search

- **Quadratic time**: Time is input size squared; every new item slows the algorithm down more
  Example: Iterate through all possible 2-player matchups

Broader categories of running times

- **Polynomial time**: Operations grow like some polynomial function of the input; varies from quick to somewhat sluggish
  Examples: All categories on previous slide

- **Exponential time**: Operations grow exponentially with input size, algorithm gets unreasonably slow
  Example: Try all possible moves in a Sudoku puzzle to brute force it

Criteria for Evaluating Algorithms

- Main criteria we'll discuss today:
  - Is the algorithm **correct** (always produces a valid output)?
  - Is the algorithm **asymptotically fast** (running time scales well with input size)?
- Other questions one could ask:
  - Is the algorithm **optimal** (produces the best solution)?
    - Some famous algorithms aren't optimal (e.g. decision trees), but they produce "good enough" solutions due to time constraints
  - Does the algorithm's **memory use** grow reasonably with the size of the input?

The Joy of Algorithms

- A good algorithm
  - often has a clever idea at its heart
  - which may be complex to execute, but
  - ultimately saves time

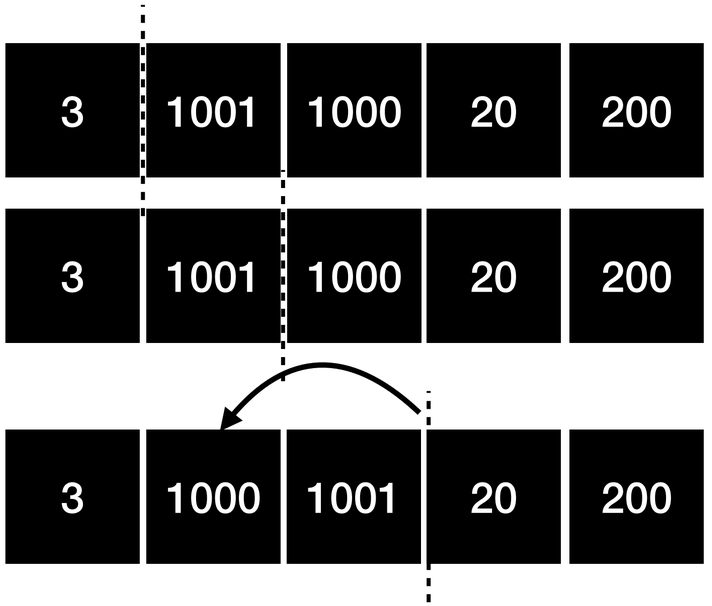
Example Algorithm to Analyze: Insertion Sort

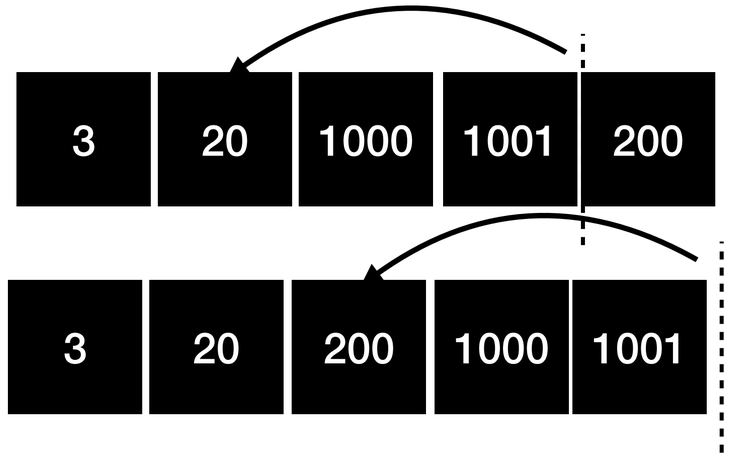
- Goal: Sort an array.

- Strategy: One at a time from left to right, take an item and move it left until the item to its left is smaller or tied. This creates a growing "sorted region" on the left-hand-side that eventually covers the whole array.

Example Algorithm to Analyze: Insertion Sort

- Strategy: One at a time from left to right, take an item and move it left until the item to its left is smaller or tied. This creates a growing "sorted region" on the left-hand-side that eventually covers the whole array.



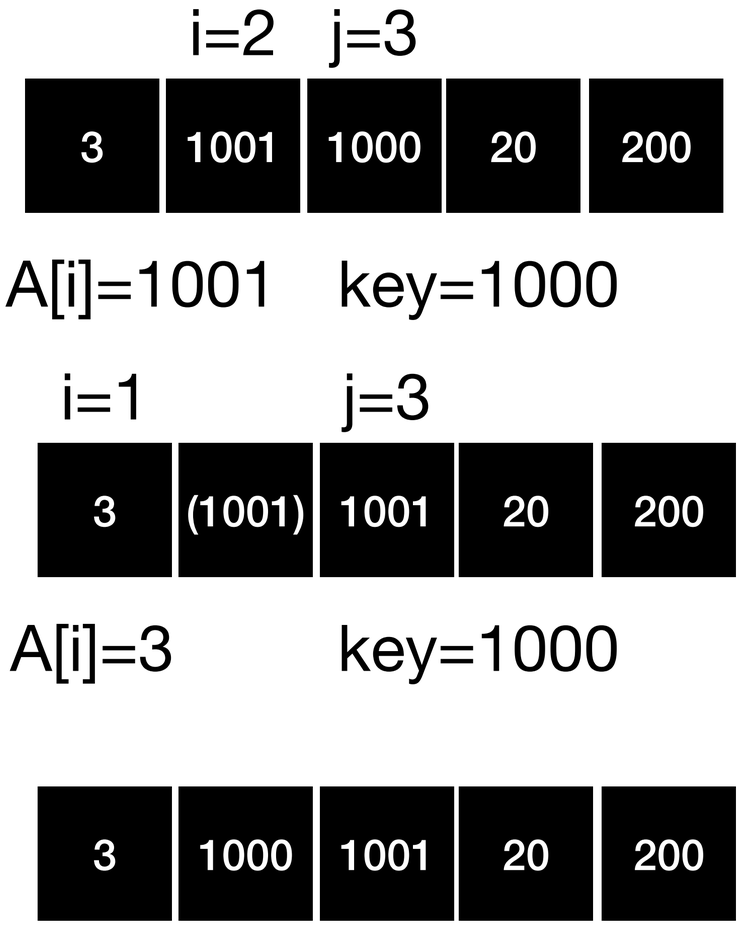


: sorted to the left of here

Insertion Sort Pseudocode

from Cormen, Leiserson, Rivest, & Stein

```
// Note:  one-indexing for clarity since we start from 2nd item
INSERTION-SORT(A)
  for j = 2 to length(A):     // Index of the first unsorted element
      key = A[j]
      // Work backwards from the right to find where key fits
      i = j - 1
      while i > 0 and A[i] > key:
          A[i+1] = A[i]         // Scoot this entry right
          i = i-1
      A[i+1] = key
```



Insertion Sort Correctness Proof Sketch

- At the beginning, the element on the far left of the array is a sorted region of length 1.
- Every iteration of the outer loop takes the element to the right of the sorted region and pushes it left through the sorted region until it is in the proper place in the sorted order.
    - This grows the sorted region by 1 element while maintaining its sortedness.
- So the sorted region grows by 1 element each time until the end of the last iteration, when it includes the whole array.
- So Insertion Sort sorts the whole array.

Insertion Sort Running Time Analysis: Best Case

- The **best case** is that the array is already sorted.
  - In that case, the inner loop performs only a constant number of operations per item, checking that it is indeed where it is supposed to be.
  - This results in overall **linear** time for the algorithm.

Insertion Sort Running Time Analysis: Worst Case

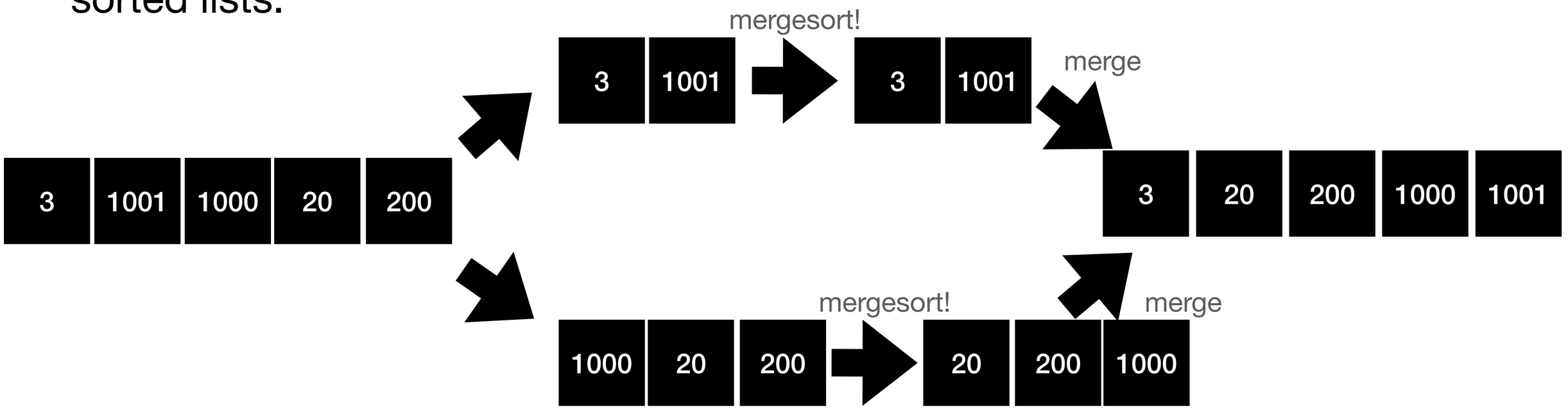
- The **worst case** is that the array is in reverse order.
  - Then every new element is less than all the elements preceding it, and needs to be shifted past all of them.
  - This results in 1 + 2 + ... + n-1 shift operations, which is $n^2/2$.
  - This results in overall **quadratic** ($\Theta(n^2)$) worst case time for the algorithm.

Why study the worst case?

- The **best case** for an algorithm often doesn't come up - it's a rare alignment of everything going right for the algorithm (like already being sorted)
- The **worst case** gives us a bound on how bad our running time could be, which matters more practically than how good it could possibly be
- The worst case is also easier to compute than the **average case** over all inputs, while being often asymptotically the same as the average case
    - An average case computation also needs to make assumptions like "all inputs are equally likely" which may not be true in practice

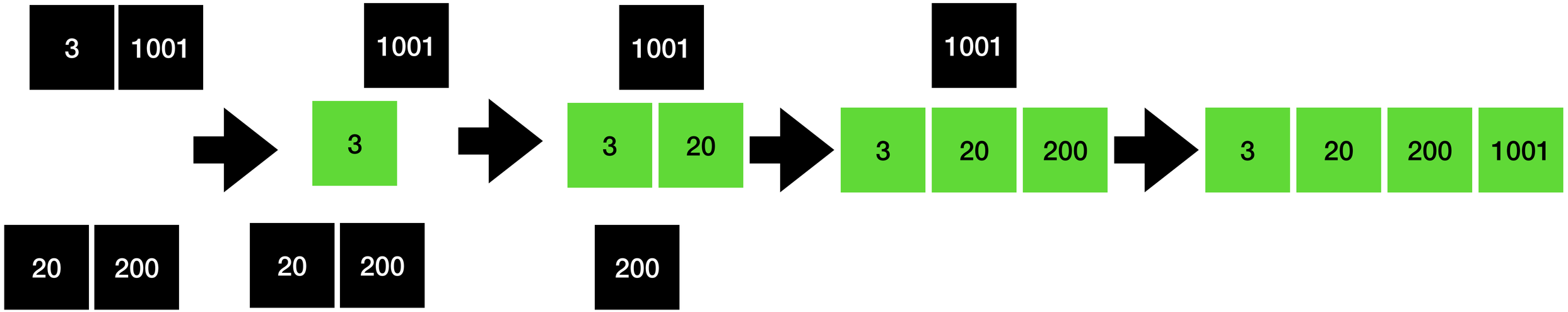
Mergesort

- Goal: Sort an array.
- Strategy: **Divide** the array in two, **sort** each half **recursively**, then **merge** the sorted lists.



Merge: The Basic idea

- Only works with 2 **sorted** lists as input.
- Compare the 2 items at the front of their lists, remove the smaller item from its list, and add it to the result list.
- Repeat to add, one-by-one, the smallest remaining element of either list to the result

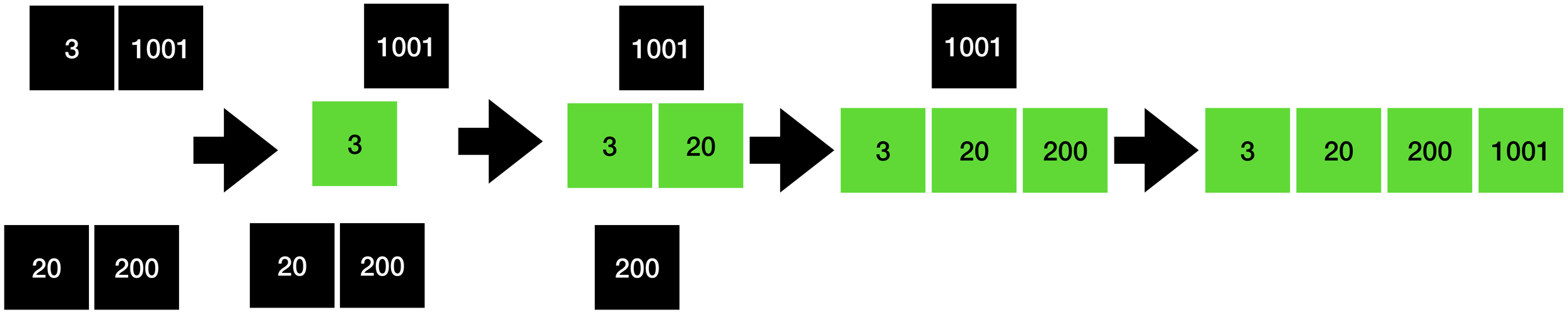


Merge Pseudocode

```
MERGE(L1, L2):      // L1, L2 are lists; "head" refers to the first element of a list
    result = empty list
    while(True):
        if L1 is empty:
            return result concatenated with L2
        if L2 is empty:
            return result concatenated with L1
        if the head of L1 is less than the head of L2:
            remove the head of L1, append it to result
        else:
            remove the head of L2, append it to result
```

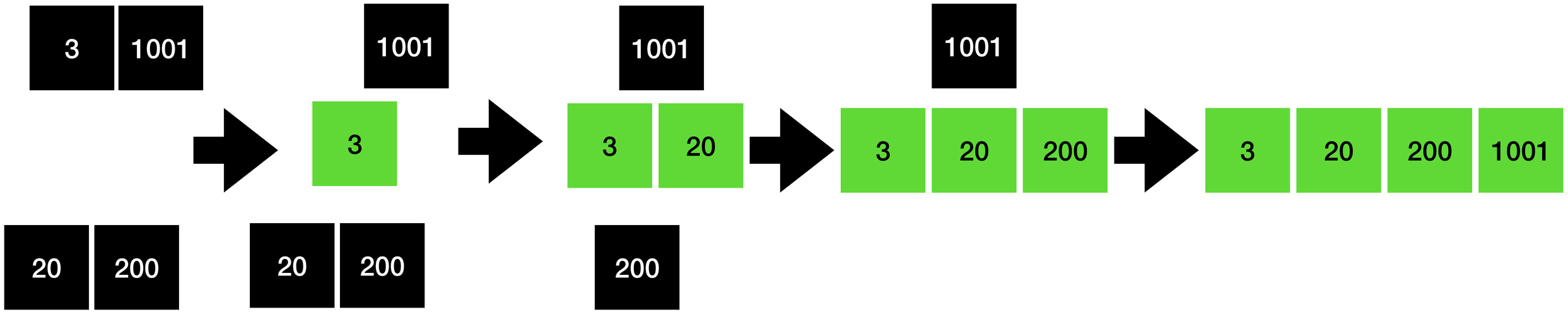
Merge: Correctness Analysis Sketch

- The smallest item over both sorted lists must always be either the smallest item of list 1 or the smallest item of list 2
- Since the lists are sorted, this smallest item must be at the head of list 1 or the head of list 2
- It is therefore a correct first move to place this item at the beginning of the output list
- Every round after that, the smallest item of what's left is what should be placed next in the output, and it's always to be found either at the head of list 1 or the head of list 2



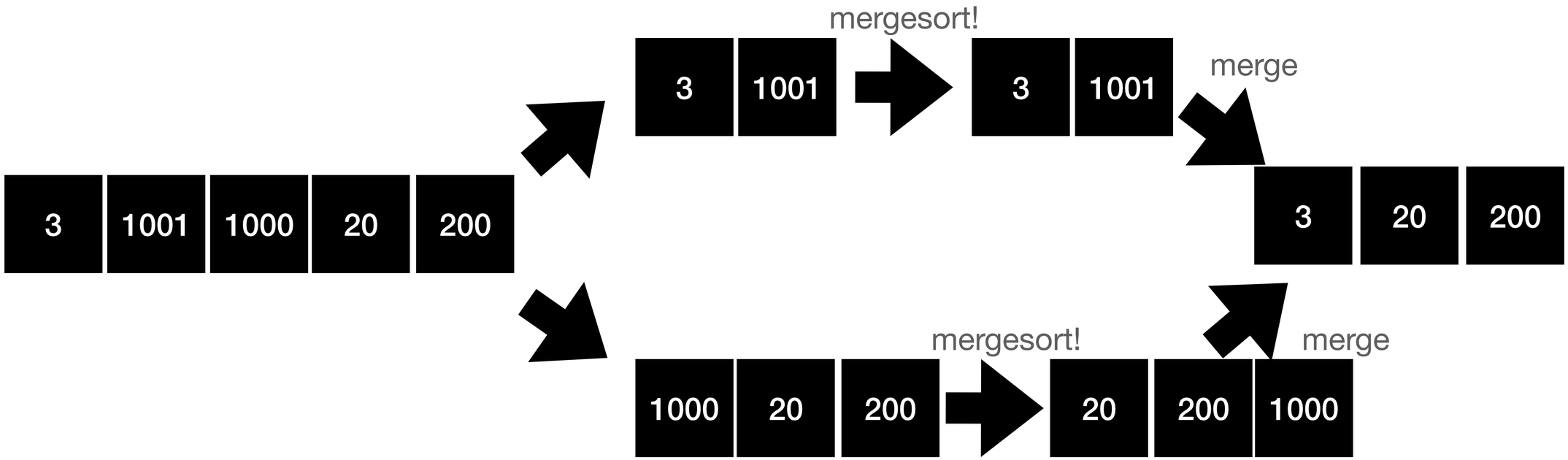
Merge: Running Time Analysis

- Every round we do a constant amount *c* of work (one comparison, one removal from a list, one placement in a new list)
- There is one round per item added to the new list
- There are N of these, where N is the total number of items in the two lists
- So this is a **linear** time operation, taking *c*N time



Mergesort Pseudocode

MERGESORT(L): // L is a linked list of data
    If |L| = 1 or 0, return L
    Copy the first ⌊|L|/2⌋ elements of the list to list “left” and the rest to list “right”
    return MERGE(MERGESORT(left), MERGESORT(right))

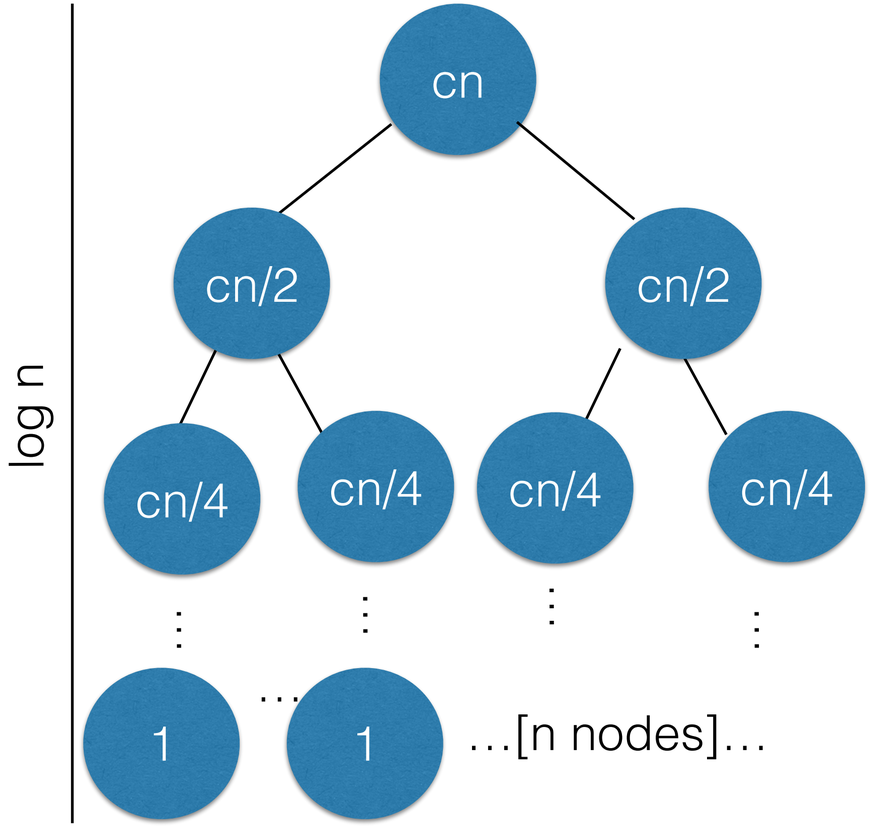


Mergesort Correctness Proof Sketch

- We know mergesort correctly sorts 0 or 1 elements.
- If mergesort correctly sorts up to *k* elements, it should also correctly sort up to *2k* elements, because each up to *k*-element mergesort works, and the merge correctly takes the 2 sorted lists that result and produces 1 sorted list.
- We could repeat that argument to argue the correctness of Mergesort for any value of *2k*
  - If it works for 1, it works for 2. If it works for up to 2, it works for 3 or 4. Etc....

Mergesort Running Time Analysis

- All the significant work of the algorithm happens in the merges
- We can draw a tree where each node gives the running time of a merge in the recursion
- This tree has $\log_2 N$ levels, because the node sizes go cn, cn/2, cn/4, cn/8, ... 1
- And each level of the tree sums to the same value, cN
- So the running time is $cN \log_2 N$

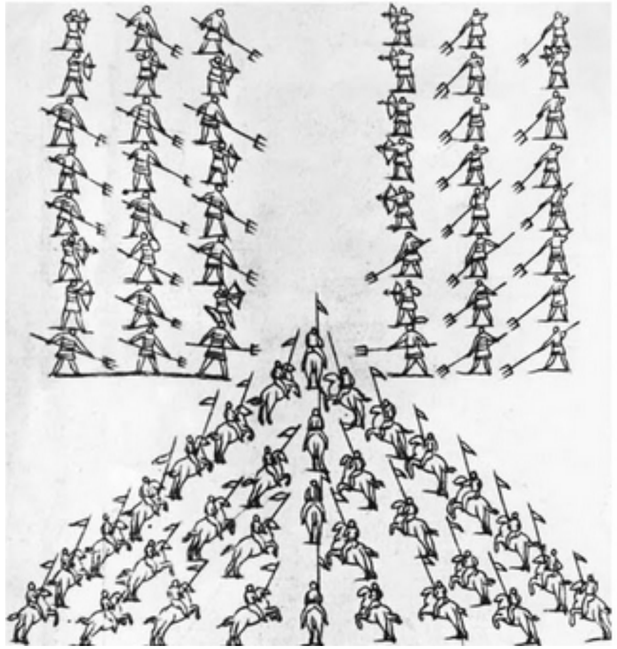


So mergesort is usually better than insertion sort

- The worst case time of Θ(N log N) is better than the worst case time of $\Theta(N^2)$.
  - N log N happens to be the best you can do for sorting in the general case!
- But with a best case of linear time when the input is sorted, insertion sort may be faster if the input is already almost sorted.

Mergesort is a "Divide and Conquer" algorithm

- Divide and conquer: split the problem into parts, solve each part with a recursive call, then combine the parts to solve the problem
- Fast divide and conquer algorithms also exist for
  - Integer multiplication
  - Matrix multiplication
  - Finding the 2 closest points in a plane
  - And more



Takeaways

- Two main things to analyze about an algorithm:
  - Is it always correct?
  - How does its running time scale with the input?
- We typically analyze the worst case for running time
- Broad running time categories include constant time, logarithmic time, linear time, quadratic time, polynomial time, exponential time
- We saw two searches (Linear Search and Binary Search) and two sorts (Insertion Sort and Mergesort) get analyzed and compared for running time