Analysis of algorithms

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 2

Measuring performance

- Example of validating SIM cards against Aadhaar data
  - Naive approach takes thousands of years
  - Smarter solution takes a few minutes

Measuring performance

- Example of validating SIM cards against Aadhaar data
  - Naive approach takes thousands of years
  - Smarter solution takes a few minutes
- Two main resources of interest
  - Running time — how long the algorithm takes
  - Space — memory requirement

Measuring performance

- Example of validating SIM cards against Aadhaar data
  - Naive approach takes thousands of years
  - Smarter solution takes a few minutes
- Two main resources of interest
  - Running time — how long the algorithm takes
  - Space — memory requirement
- Time depends on processing power
  - Impossible to change for given hardware
  - Enhancing hardware has only a limited impact at a practical level

Measuring performance

- Example of validating SIM cards against Aadhaar data
  - Naive approach takes thousands of years
  - Smarter solution takes a few minutes
- Two main resources of interest
  - Running time — how long the algorithm takes
  - Space — memory requirement
- Time depends on processing power
  - Impossible to change for given hardware
  - Enhancing hardware has only a limited impact at a practical level
- Storage is limited by available memory
  - Easier to configure, augment

Measuring performance

- Example of validating SIM cards against Aadhaar data
  - Naive approach takes thousands of years
  - Smarter solution takes a few minutes
- Two main resources of interest
  - Running time — how long the algorithm takes
  - Space — memory requirement
- Time depends on processing power
  - Impossible to change for given hardware
  - Enhancing hardware has only a limited impact at a practical level
- Storage is limited by available memory
  - Easier to configure, augment
- Typically, we focus on time rather than space

Input size

- Running time depends on input size
  - Larger arrays will take longer to sort

Input size

- Running time depends on input size
  - Larger arrays will take longer to sort
- Measure time efficiency as function of input size
  - Input size n
  - Running time $t(n)$

Input size

- Running time depends on input size
  - Larger arrays will take longer to sort
- Measure time efficiency as function of input size
  - Input size n
  - Running time $t(n)$
- Different inputs of size n may take different amounts of time
  - We will return to this point later

Input size

- Running time depends on input size
  - Larger arrays will take longer to sort
- Measure time efficiency as function of input size
  - Input size n
  - Running time $t(n)$
- Different inputs of size n may take different amounts of time
  - We will return to this point later

Example 1 SIM cards vs Aadhaar cards

- $n \approx 10^9$ — number of cards

Input size

- Running time depends on input size
  - Larger arrays will take longer to sort
- Measure time efficiency as function of input size
  - Input size n
  - Running time $t(n)$
- Different inputs of size n may take different amounts of time
  - We will return to this point later

Example 1 SIM cards vs Aadhaar cards

- $n \approx 10^9$ — number of cards
- Naive algorithm: $t(n) \approx n^2$

Input size

- Running time depends on input size
  - Larger arrays will take longer to sort
- Measure time efficiency as function of input size
  - Input size n
  - Running time $t(n)$
- Different inputs of size n may take different amounts of time
  - We will return to this point later

Example 1 SIM cards vs Aadhaar cards

- $n \approx 10^9$ — number of cards
- Naive algorithm: $t(n) \approx n^2$
- Clever algorithm: $t(n) \approx n \log_2 n$
  - $\log_2 n$ — number of times you need to divide n by 2 to reach 1
  - $\log_2(n) = k \Rightarrow n = 2^k$

Input size . . .

Example 2 Video game

- Several objects on screen

Input size . . .

Example 2 Video game

- Several objects on screen
- Basic step: find closest pair of objects

Input size . . .

Example 2 Video game

- Several objects on screen
- Basic step: find closest pair of objects
- n objects — naive algorithm is n^2
  - For each pair of objects, compute their distance
  - Report minimum distance across all pairs

Input size . . .

Example 2 Video game

- Several objects on screen
- Basic step: find closest pair of objects
- n objects — naive algorithm is n^2
  - For each pair of objects, compute their distance
  - Report minimum distance across all pairs
- There is a clever algorithm that takes sime $n \log_2 n$

Input size . . .

Example 2 Video game

- Several objects on screen
- Basic step: find closest pair of objects
- n objects — naive algorithm is n^2
  - For each pair of objects, compute their distance
  - Report minimum distance across all pairs
- There is a clever algorithm that takes sime $n \log_2 n$

- High resolution gaming consle may have $4000x2000$ pixels
  - 8×10^6 points — 8 million

Input size ...

Example 2 Video game

- Several objects on screen
- Basic step: find closest pair of objects
- n objects — naive algorithm is n^2
  - For each pair of objects, compute their distance
  - Report minimum distance across all pairs
- There is a clever algorithm that takes sime $n \log_2 n$

- High resolution gaming consle may have $4000x2000$ pixels
  - 8×10^6 points — 8 million
- Suppose we have $100,000 = 1 \times 10^5$ objects

Input size . . .

Example 2 Video game

- Several objects on screen
- Basic step: find closest pair of objects
- n objects — naive algorithm is n^2
  - For each pair of objects, compute their distance
  - Report minimum distance across all pairs
- There is a clever algorithm that takes sime $n \log_2 n$

- High resolution gaming consle may have $4000x2000$ pixels
  - 8×10^6 points — 8 million
- Suppose we have $100,000 = 1 \times 10^5$ objects
- Naive algorithm takes 10^{10} steps
  - 1000 seconds, or 16.7 minutes in Python
  - Unacceptable response time!

Input size . . .

Example 2 Video game

- Several objects on screen
- Basic step: find closest pair of objects
- n objects — naive algorithm is n^2
  - For each pair of objects, compute their distance
  - Report minimum distance across all pairs
- There is a clever algorithm that takes sime $n \log_2 n$

- High resolution gaming consle may have $4000x2000$ pixels
  - 8×10^6 points — 8 million
- Suppose we have $100,000 = 1 \times 10^5$ objects
- Naive algorithm takes 10^{10} steps
  - 1000 seconds, or 16.7 minutes in Python
  - Unacceptable response time!
- $\log_2 100,000$ is under 20, so $n \log_2 n$ takes a fraction of a second

Orders of magnitude

- When comparing $t(n)$, focus on orders of magnitude
  - Ignore constant factors

Orders of magnitude

- When comparing $t(n)$, focus on orders of magnitude
  - Ignore constant factors
- $f(n) = n^3$ eventually grows faster than $g(n) = 5000n^2$
  - For small values of n, $f(n) < g(n)$
  - After $n = 5000$, $f(n)$ overtakes $g(n)$

Orders of magnitude

- When comparing $t(n)$, focus on orders of magnitude
  - Ignore constant factors
- $f(n) = n^3$ eventually grows faster than $g(n) = 5000n^2$
  - For small values of n, $f(n) < g(n)$
  - After $n = 5000$, $f(n)$ overtakes $g(n)$
- Asymptotic complexity
  - What happens in the limit, as n becomes large

Orders of magnitude

- When comparing $t(n)$, focus on orders of magnitude
  - Ignore constant factors
- $f(n) = n^3$ eventually grows faster than $g(n) = 5000n^2$
  - For small values of n, $f(n) < g(n)$
  - After $n = 5000$, $f(n)$ overtakes $g(n)$
- Asymptotic complexity
  - What happens in the limit, as n becomes large
- Typical growth functions
  - Is $t(n)$ proportional to $\log n, \ldots, n^2, n^3, \ldots, 2^n$?
    - Note: $\log n$ means $\log_2 n$ by default
  - Logarithmic, polynomial, exponential, ...

Orders of magnitude

Input size	Values of $t(n)$					
	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	3.3	10	33	100	1000	1000	10^6
100	6.6	100	66	10^4	10^6	10^{30}	10^{157}
1000	10	1000	10^4	10^6	10^9		
10^4	13	10^4	10^5	10^8	10^{12}		
10^5	17	10^5	10^6	10^{10}			
10^6	20	10^6	10^7	10^{12}			
10^7	23	10^7	10^8				
10^8	27	10^8	10^9				
10^9	30	10^9	10^{10}				
10^{10}	33	10^{10}	10^{11}				

Measuring running time

- Analysis should be independent of the underlying hardware
  - Don't use actual time
  - Measure in terms of basic operations

Measuring running time

- Analysis should be independent of the underlying hardware
  - Don't use actual time
  - Measure in terms of basic operations
- Typical basic operations
  - Compare two values
  - Assign a value to a variable

Measuring running time

- Analysis should be independent of the underlying hardware
  - Don't use actual time
  - Measure in terms of basic operations
- Typical basic operations
  - Compare two values
  - Assign a value to a variable
- Exchange a pair of values?

```
(x,y) = (y,x)        t = x
                     x = y
                     y = t
```

  - If we ignore constants, focus on orders of magnitude, both are within a factor of 3
  - Need not be very precise about defining basic operations

What is the input size

- Typically a natural parameter
  - Size of a list/array that we want to search or sort
  - Number of objects we want to rearrange
  - Number of vertices and number edges in a graph
    - We shall see why these are separate parameters

What is the input size

- Typically a natural parameter
  - Size of a list/array that we want to search or sort
  - Number of objects we want to rearrange
  - Number of vertices and number edges in a graph
    - We shall see why these are separate parameters
- What about numeric problems? Is n a prime?

What is the input size

- Typically a natural parameter
  - Size of a list/array that we want to search or sort
  - Number of objects we want to rearrange
  - Number of vertices and number edges in a graph
    - We shall see why these are separate parameters
- What about numeric problems? Is n a prime?
  - Magnitude of n is not the correct measure

What is the input size

- Typically a natural parameter
  - Size of a list/array that we want to search or sort
  - Number of objects we want to rearrange
  - Number of vertices and number edges in a graph
    - We shall see why these are separate parameters
- What about numeric problems? Is n a prime?
  - Magnitude of n is not the correct measure
  - Arithmetic operations are performed digit by digit
    - Addition with carry, subtraction with borrow, multiplication, long division ...

What is the input size

- Typically a natural parameter
  - Size of a list/array that we want to search or sort
  - Number of objects we want to rearrange
  - Number of vertices and number edges in a graph
    - We shall see why these are separate parameters
- What about numeric problems? Is n a prime?
  - Magnitude of n is not the correct measure
  - Arithmetic operations are performed digit by digit
    - Addition with carry, subtraction with borrow, multiplication, long division ...
  - Number of digits is a natural measure of input size
    - Same as $\log_b n$, when we write n in base b

Which inputs should we consider?

- Performance varies across input instances
  - By luck, the value we are searching for is the first element we examine in an array

Which inputs should we consider?

- Performance varies across input instances
  - By luck, the value we are searching for is the first element we examine in an array
- Ideally, want the "average" behaviour
  - Difficult to compute
  - Average over what? Are all inputs equally likely?
  - Need a probability distribution over inputs

Which inputs should we consider?

- Performance varies across input instances
  - By luck, the value we are searching for is the first element we examine in an array
- Ideally, want the "average" behaviour
  - Difficult to compute
  - Average over what? Are all inputs equally likely?
  - Need a probability distribution over inputs
- Instead, worst case input
  - Input that forces algorithm to take longest possible time
    - Search for a value that is not present in an unsorted list
    - Must scan all elements

Which inputs should we consider?

- Performance varies across input instances
  - By luck, the value we are searching for is the first element we examine in an array
- Ideally, want the "average" behaviour
  - Difficult to compute
  - Average over what? Are all inputs equally likely?
  - Need a probability distribution over inputs
- Instead, worst case input
  - Input that forces algorithm to take longest possible time
    - Search for a value that is not present in an unsorted list
    - Must scan all elements
  - Pessimistic — worst case may be rare

Which inputs should we consider?

- Performance varies across input instances
  - By luck, the value we are searching for is the first element we examine in an array
- Ideally, want the "average" behaviour
  - Difficult to compute
  - Average over what? Are all inputs equally likely?
  - Need a probability distribution over inputs
- Instead, worst case input
  - Input that forces algorithm to take longest possible time
    - Search for a value that is not present in an unsorted list
    - Must scan all elements
  - Pessimistic — worst case may be rare
  - Upper bound for worst case guarantees good performance

Summary

- Two important parameters when measuring algorithm performance
  - Running time, memory requirement (space)
  - We mainly focus on time

Summary

- Two important parameters when measuring algorithm performance
  - Running time, memory requirement (space)
  - We mainly focus on time
- Running time $t(n)$ is a function of input size n
  - Interested in orders of magnitude
  - Asymptotic complexity, as n becomes large

Summary

- Two important parameters when measuring algorithm performance
  - Running time, memory requirement (space)
  - We mainly focus on time
- Running time $t(n)$ is a function of input size n
  - Interested in orders of magnitude
  - Asymptotic complexity, as n becomes large
- From running time, we can estimate feasible input sizes

Summary

- Two important parameters when measuring algorithm performance
  - Running time, memory requirement (space)
  - We mainly focus on time
- Running time $t(n)$ is a function of input size n
  - Interested in orders of magnitude
  - Asymptotic complexity, as n becomes large
- From running time, we can estimate feasible input sizes
- We focus on worst case inputs
  - Pessimistic, but easier to calculate than average case
  - Upper bound on worst case gives us an overall guarantee on performance

Comparing orders of magnitude

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 2

Orders of magnitude

- When comparing $t(n)$, focus on orders of magnitude
  - Ignore constant factors
- $f(n) = n^3$ eventually grows faster than $g(n) = 5000n^2$

Orders of magnitude

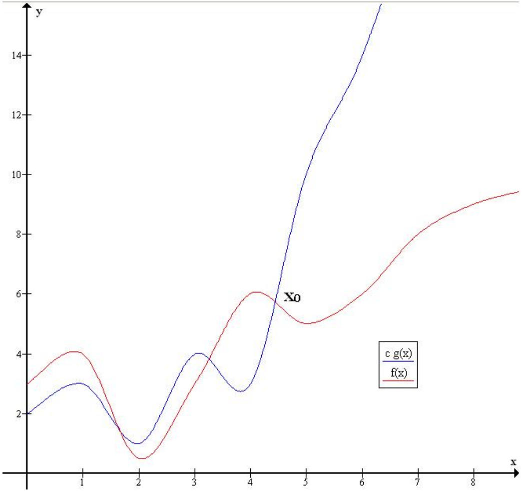
- When comparing $t(n)$, focus on orders of magnitude
  - Ignore constant factors
- $f(n) = n^3$ eventually grows faster than $g(n) = 5000n^2$
- How do we compare functions with respect to orders of magnitude?

Upper bounds

- $f(x)$ is said to be $O(g(x))$ if we can find constants c and x_0 such that $c \cdot g(x)$ is an upper bound for $f(x)$ for x beyond x_0

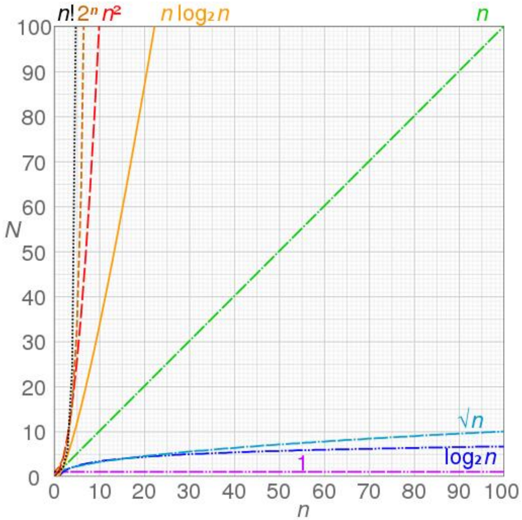
Upper bounds

- $f(x)$ is said to be $O(g(x))$ if we can find constants c and x_0 such that $c \cdot g(x)$ is an upper bound for $f(x)$ for x beyond x_0
- $f(x) \le cg(x)$ for every $x \ge x_0$



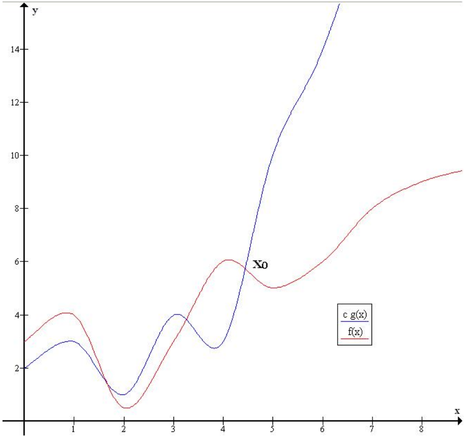
Upper bounds

- $f(x)$ is said to be $O(g(x))$ if we can find constants c and x_0 such that $c \cdot g(x)$ is an upper bound for $f(x)$ for x beyond x_0
- $f(x) \leq cg(x)$ for every $x \geq x_0$
- Graphs of typical functions we have seen



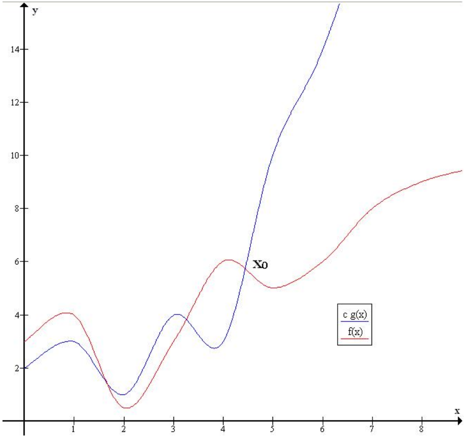
Examples

- $100n + 5$ is $O(n^2)$
  - $100n + 5 \leq 100n + n = 101n$, for $n \geq 5$
  - $101n \leq 101n^2$
  - Choose $n_0 = 5$, $c = 101$



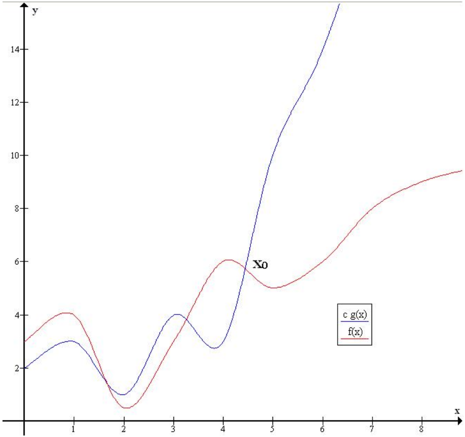
Examples

- $100n + 5$ is $O(n^2)$
  - $100n + 5 \leq 100n + n = 101n$, for $n \geq 5$
  - $101n \leq 101n^2$
  - Choose $n_0 = 5$, $c = 101$
- Alternatively
  - $100n + 5 \leq 100n + 5n = 105n$, for $n \geq 1$
  - $105n \leq 105n^2$
  - Choose $n_0 = 1$, $c = 105$



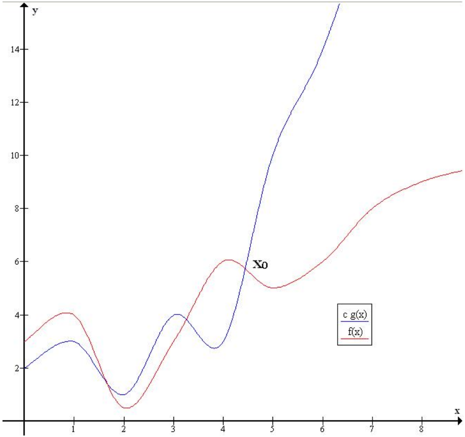
Examples

- $100n + 5$ is $O(n^2)$
  - $100n + 5 \leq 100n + n = 101n$, for $n \geq 5$
  - $101n \leq 101n^2$
  - Choose $n_0 = 5$, $c = 101$
- Alternatively
  - $100n + 5 \leq 100n + 5n = 105n$, for $n \geq 1$
  - $105n \leq 105n^2$
  - Choose $n_0 = 1$, $c = 105$
- Choice of n_0, c not unique



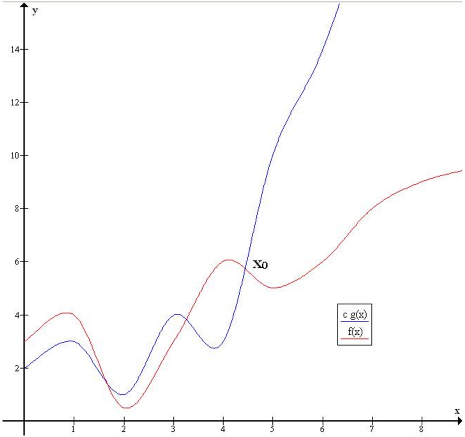
Examples . . .

- $100n^2 + 20n + 5$ is $O(n^2)$
  - $100n^2 + 20n + 5 \leq 100n^2 + 20n^2 + 5n^2$, for $n \geq 1$
  - $100n^2 + 20n + 5 \leq 125n^2$, for $n \geq 1$
  - Choose $n_0 = 1$, $c = 125$



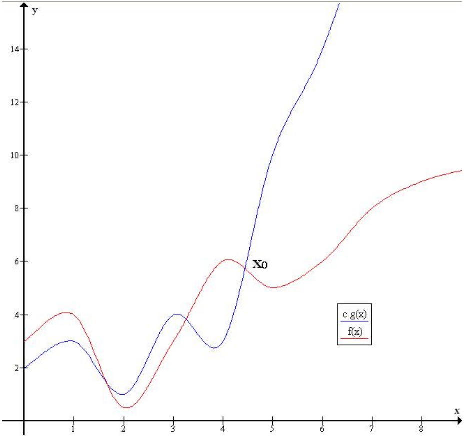
Examples . . .

- $100n^2 + 20n + 5$ is $O(n^2)$
  - $100n^2 + 20n + 5 \leq 100n^2 + 20n^2 + 5n^2$, for $n \geq 1$
  - $100n^2 + 20n + 5 \leq 125n^2$, for $n \geq 1$
  - Choose $n_0 = 1$, $c = 125$
- What matters is the highest term
  - $20n + 5$ is dominated by $100n^2$



Examples . . .

- $100n^2 + 20n + 5$ is $O(n^2)$
  - $100n^2 + 20n + 5 \leq 100n^2 + 20n^2 + 5n^2$, for $n \geq 1$
  - $100n^2 + 20n + 5 \leq 125n^2$, for $n \geq 1$
  - Choose $n_0 = 1$, $c = 125$
- What matters is the highest term
  - $20n + 5$ is dominated by $100n^2$
- n^3 is not $O(n^2)$
  - No matter what c we choose, cn^2 will be dominated by n^3 for $n \geq c$



Useful properties

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$

Useful properties

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$
- Proof
  - $f_1(n) \leq c_1 g_1(n)$ for $n > n_1$
  - $f_2(n) \leq c_2 g_2(n)$ for $n > n_2$

Useful properties

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$
- Proof
  - $f_1(n) \leq c_1 g_1(n)$ for $n > n_1$
  - $f_2(n) \leq c_2 g_2(n)$ for $n > n_2$
  - Let $c_3 = \max(c_1, c_2)$, $n_3 = \max(n_1, n_2)$

Useful properties

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$
- Proof
  - $f_1(n) \leq c_1 g_1(n)$ for $n > n_1$
  - $f_2(n) \leq c_2 g_2(n)$ for $n > n_2$
  - Let $c_3 = \max(c_1, c_2)$, $n_3 = \max(n_1, n_2)$
  - For $n \geq n_3$, $f_1(n) + f_2(n)$
    $\leq c_1 g_1(n) + c_2 g_2(n)$

Useful properties

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$
- Proof
  - $f_1(n) \leq c_1 g_1(n)$ for $n > n_1$
  - $f_2(n) \leq c_2 g_2(n)$ for $n > n_2$
  - Let $c_3 = \max(c_1, c_2)$, $n_3 = \max(n_1, n_2)$
  - For $n \geq n_3$, $f_1(n) + f_2(n)$
    $\leq c_1 g_1(n) + c_2 g_2(n)$
    $\leq c_3(g_1(n) + g_2(n))$

Useful properties

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$
- Proof
  - $f_1(n) \leq c_1 g_1(n)$ for $n > n_1$
  - $f_2(n) \leq c_2 g_2(n)$ for $n > n_2$
  - Let $c_3 = \max(c_1, c_2)$, $n_3 = \max(n_1, n_2)$
  - For $n \geq n_3$, $f_1(n) + f_2(n)$

    $\leq c_1 g_1(n) + c_2 g_2(n)$

    $\leq c_3(g_1(n) + g_2(n))$

    $\leq 2c_3(\max(g_1(n), g_2(n)))$

Useful properties

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$
- Proof
  - $f_1(n) \leq c_1 g_1(n)$ for $n > n_1$
  - $f_2(n) \leq c_2 g_2(n)$ for $n > n_2$
  - Let $c_3 = \max(c_1, c_2)$, $n_3 = \max(n_1, n_2)$
  - For $n \geq n_3$, $f_1(n) + f_2(n)$
    $\leq c_1 g_1(n) + c_2 g_2(n)$
    $\leq c_3(g_1(n) + g_2(n))$
    $\leq 2c_3(\max(g_1(n), g_2(n)))$

- Algorithm has two phases
  - Phase A takes time $O(g_A(n))$
  - Phase B takes time $O(g_B(n))$

Useful properties

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$
- Proof
  - $f_1(n) \leq c_1 g_1(n)$ for $n > n_1$
  - $f_2(n) \leq c_2 g_2(n)$ for $n > n_2$
  - Let $c_3 = \max(c_1, c_2)$, $n_3 = \max(n_1, n_2)$
  - For $n \geq n_3$, $f_1(n) + f_2(n)$

    $\leq c_1 g_1(n) + c_2 g_2(n)$

    $\leq c_3(g_1(n) + g_2(n))$

    $\leq 2c_3(\max(g_1(n), g_2(n)))$

- Algorithm has two phases
  - Phase A takes time $O(g_A(n))$
  - Phase B takes time $O(g_B(n))$
- Algorithm as a whole takes time $\max(O(g_A(n), g_B(n)))$

Useful properties

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$
- Proof
  - $f_1(n) \leq c_1 g_1(n)$ for $n > n_1$
  - $f_2(n) \leq c_2 g_2(n)$ for $n > n_2$
  - Let $c_3 = \max(c_1, c_2)$, $n_3 = \max(n_1, n_2)$
  - For $n \geq n_3$, $f_1(n) + f_2(n)$

    $\leq c_1 g_1(n) + c_2 g_2(n)$

    $\leq c_3(g_1(n) + g_2(n))$

    $\leq 2c_3(\max(g_1(n), g_2(n)))$

- Algorithm has two phases
  - Phase A takes time $O(g_A(n))$
  - Phase B takes time $O(g_B(n))$
- Algorithm as a whole takes time $\max(O(g_A(n), g_B(n)))$
- Least efficient phase is the upper bound for the whole algorithm

Lower bounds

- $f(x)$ is said to be $\Omega(g(x))$ if we can find constants c and x_0 such that $cg(x)$ is a lower bound for $f(x)$ for x beyond x_0
  - $f(x) \geq cg(x)$ for every $x \geq x_0$

Lower bounds

- $f(x)$ is said to be $\Omega(g(x))$ if we can find constants c and x_0 such that $cg(x)$ is a lower bound for $f(x)$ for x beyond x_0
  - $f(x) \geq cg(x)$ for every $x \geq x_0$
- n^3 is $\Omega(n^2)$
  - $n^3 > n^2$ for all n, so $n_0 = 1$, $c = 1$

Lower bounds

- $f(x)$ is said to be $\Omega(g(x))$ if we can find constants c and x_0 such that $cg(x)$ is a lower bound for $f(x)$ for x beyond x_0
  - $f(x) \geq cg(x)$ for every $x \geq x_0$
- n^3 is $\Omega(n^2)$
  - $n^3 > n^2$ for all n, so $n_0 = 1$, $c = 1$
- Typically we establish lower bounds for a problem rather than an individual algorithm
  - If we sort a list by comparing elements and swapping them, we require $\Omega(n \log n)$ comparisons
  - This is independent of the algorithm we use for sorting

Tight bounds

- $f(x)$ is said to be $\Theta(g(x))$ if it is both $O(g(x))$ and $\Omega(g(x))$
  - Find constants c_1, c_2, x_0 such that $c_1 g(x) \leq f(x) \leq c_2 g(x)$ for every $x \geq x_0$

Tight bounds

- $f(x)$ is said to be $\Theta(g(x))$ if it is both $O(g(x))$ and $\Omega(g(x))$
  - Find constants c_1, c_2, x_0 such that $c_1 g(x) \leq f(x) \leq c_2 g(x)$ for every $x \geq x_0$
- $n(n-1)/2$ is $\Theta(n^2)$

Tight bounds

- $f(x)$ is said to be $\Theta(g(x))$ if it is both $O(g(x))$ and $\Omega(g(x))$
  - Find constants c_1, c_2, x_0 such that $c_1 g(x) \leq f(x) \leq c_2 g(x)$ for every $x \geq x_0$
- $n(n-1)/2$ is $\Theta(n^2)$
  - Upper bound
    - $n(n-1)/2 = n^2/2 - n/2 \leq n^2/2$ for all $n \geq 0$

Tight bounds

- $f(x)$ is said to be $\Theta(g(x))$ if it is both $O(g(x))$ and $\Omega(g(x))$
  - Find constants c_1, c_2, x_0 such that $c_1 g(x) \leq f(x) \leq c_2 g(x)$ for every $x \geq x_0$
- $n(n-1)/2$ is $\Theta(n^2)$
  - Upper bound
    - $n(n-1)/2 = n^2/2 - n/2 \leq n^2/2$ for all $n \geq 0$
  - Lower bound
    - $n(n-1)/2 = n^2/2 - n/2 \geq n^2/2 - (n/2 \times n/2) \geq n^2/4$ for $n \geq 2$

Tight bounds

- $f(x)$ is said to be $\Theta(g(x))$ if it is both $O(g(x))$ and $\Omega(g(x))$
  - Find constants c_1, c_2, x_0 such that $c_1 g(x) \leq f(x) \leq c_2 g(x)$ for every $x \geq x_0$
- $n(n-1)/2$ is $\Theta(n^2)$
  - Upper bound
    - $n(n-1)/2 = n^2/2 - n/2 \leq n^2/2$ for all $n \geq 0$
  - Lower bound
    - $n(n-1)/2 = n^2/2 - n/2 \geq n^2/2 - (n/2 \times n/2) \geq n^2/4$ for $n \geq 2$
  - Choose $n_0 = 2$, $c_1 = 1/4$, $c_2 = 1/2$

Summary

- $f(n)$ is $O(g(n))$ means $g(n)$ is an upper bound for $f(n)$
  - Useful to describe asymptotic worst case running time

Summary

- $f(n)$ is $O(g(n))$ means $g(n)$ is an upper bound for $f(n)$
  - Useful to describe asymptotic worst case running time
- $f(n)$ is $\Omega(g(n))$ means $g(n)$ is a lower bound for $f(n)$
  - Typically used for a problem as a whole, rather than an individual algorihm

Summary

- $f(n)$ is $O(g(n))$ means $g(n)$ is an upper bound for $f(n)$
  - Useful to describe asymptotic worst case running time
- $f(n)$ is $\Omega(g(n))$ means $g(n)$ is a lower bound for $f(n)$
  - Typically used for a problem as a whole, rather than an individual algorihm
- $f(n)$ is $\Theta(g(n))$: matching upper and lower bounds
  - We have found an optimal algorithm for a problem

Calculating complexity — Examples

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 2

Calculating complexity

- Iterative programs
- Recursive programs

Example 1

Find the maximum element in a list

- Input size is length of the list
- Single loop scans all elements
- Always takes n steps
- Overall time is $O(n)$

```
def maxElement(L):
  maxval = L[0]
  for i in range(len(L)):
    if L[i] > maxval:
      maxval = L[i]
return(maxval)
```

Example 2

Check whether a list contains duplicates

- Input size is length of the list
- Nested loop scans all pairs of elements
- A duplicate may be found in the very first iteration
- Worst case — no duplicates, both loops run fully
- Time is $(n-1) + (n-2) + \ldots + 1 = n(n-1)/2$
- Overall time is $O(n^2)$

```python
def noDuplicates(L):
  for i in range(len(L)):
    for j in range(i+1,len(L)):
      if L[i] == L[j]:
        return(False)
  return(True)
```

Example 3

Matrix multiplication

- Matrix is represented as list of lists
  - $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$
  - `[[1,2,3],[4,5,6]]`
- Input matrices have size $m \times n$, $n \times p$
- Output matrix is $m \times p$
- Three nested loops
- Overall time is $O(mnp)$ — $O(n^3)$ if both are $n \times n$

```
def matrixMultiply(A,B):
  (m,n,p) = (len(A),len(B),len(B[0]))

  C = [[ 0 for i in range(p) ]
           for j in range(m) ]

  for i in range(m):
    for j in range(p):
      for k in range(n):
        C[i][j] = C[i][j] + A[i][k]*B[k][j]

  return(C)
```

Example 4

Number of bits in binary representation of n

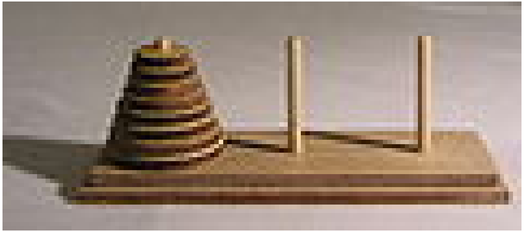
- $\log n$ steps for n to reach 1
- For number theoretic problems, input size is number of digits
- This algorithm is linear in input size

```
def numberOfBits(n):

  count = 1

  while n > 1:
    count = count + 1
    n = n // 2

  return(count)
```

Example 5

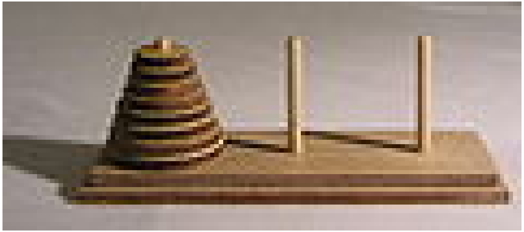
Towers of Hanoi

- Three pegs A,B,C
- Move n disks from A to B, use C as transit peg
- Never put a larger disk on a smaller one



Example 5

Towers of Hanoi



- Three pegs A,B,C
- Move n disks from A to B, use C as transit peg
- Never put a larger disk on a smaller one

Recursive solution

- Move $n-1$ disks from A to C, use B as transit peg
- Move larges disk from A to B
- Move $n-1$ disks from C to B, use A as transit peg

Example 5

Recurrence

- $M(n)$ — number of moves to transfer n disks
- $M(1) = 1$
- $M(n) = M(n-1) + 1 + M(n-1) = 2M(n-1) + 1$

Example 5

Recurrence

- $M(n)$ — number of moves to transfer n disks
- $M(1) = 1$
- $M(n) = M(n-1) + 1 + M(n-1) = 2M(n-1) + 1$

Unwind and solve

$$M(n) \;=\; 2M(n-1) + 1$$

Example 5

Recurrence

- $M(n)$ — number of moves to transfer n disks
- $M(1) = 1$
- $M(n) = M(n-1) + 1 + M(n-1) = 2M(n-1) + 1$

Unwind and solve

$$\begin{aligned} M(n) &= 2M(n-1)+1 \\ &= 2(2M(n-2)+1)+1 = 2^2M(n-2)+(2+1) \end{aligned}$$

Example 5

Recurrence

- $M(n)$ — number of moves to transfer n disks
- $M(1) = 1$
- $M(n) = M(n-1) + 1 + M(n-1) = 2M(n-1) + 1$

Unwind and solve

$$\begin{aligned}
M(n) &= 2M(n-1)+1 \\
&= 2(2M(n-2)+1)+1 = 2^2M(n-2)+(2+1) \\
&= 2^2(2M(n-3)+1)+(2+1) = 2^3M(n-3)+(4+2+1)
\end{aligned}$$

Example 5

Recurrence

- $M(n)$ — number of moves to transfer n disks
- $M(1) = 1$
- $M(n) = M(n-1) + 1 + M(n-1) = 2M(n-1) + 1$

Unwind and solve

$$\begin{aligned}
M(n) &= 2M(n-1) + 1 \\
&= 2(2M(n-2) + 1) + 1 = 2^2 M(n-2) + (2+1) \\
&= 2^2(2M(n-3) + 1) + (2+1) = 2^3 M(n-3) + (4+2+1) \\
&\quad \cdots \\
&= 2^k M(n-k) + (2^k - 1))
\end{aligned}$$

Example 5

Recurrence

- $M(n)$ — number of moves to transfer n disks
- $M(1) = 1$
- $M(n) = M(n-1) + 1 + M(n-1) = 2M(n-1) + 1$

Unwind and solve

$$
\begin{aligned}
M(n) &= 2M(n-1) + 1 \\
&= 2(2M(n-2)+1) + 1 = 2^2 M(n-2) + (2+1) \\
&= 2^2(2M(n-3)+1) + (2+1) = 2^3 M(n-3) + (4+2+1) \\
&\quad \ldots \\
&= 2^k M(n-k) + (2^k - 1)) \\
&\quad \ldots \\
&= 2^{n-1} M(1) + (2^{n-1} - 1)
\end{aligned}
$$

Example 5

Recurrence

- $M(n)$ — number of moves to transfer n disks
- $M(1) = 1$
- $M(n) = M(n-1) + 1 + M(n-1) = 2M(n-1) + 1$

Unwind and solve

$$
\begin{aligned}
M(n) &= 2M(n-1)+1 \\
&= 2(2M(n-2)+1)+1 = 2^2M(n-2)+(2+1) \\
&= 2^2(2M(n-3)+1)+(2+1) = 2^3M(n-3)+(4+2+1) \\
&\quad \cdots \\
&= 2^kM(n-k)+(2^k-1)) \\
&\quad \cdots \\
&= 2^{n-1}M(1)+(2^{n-1}-1) \\
&= 2^{n-1}+2^{n-1}-1 = 2^n-1
\end{aligned}
$$

Summary

- Iterative programs
  - Focus on loops

Summary

- Iterative programs
  - Focus on loops
- Recursive programs
  - Write and solve a recurrence

Summary

- Iterative programs
  - Focus on loops
- Recursive programs
  - Write and solve a recurrence
- Need to be clear about accounting for “basic” operations

Searching in a List

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 2

Search problem

- Is value `v` present in list `l`?

Search problem

- Is value `v` present in list `l`?
- Naive solution scans the list

```
def naivesearch(v,l):
  for x in l:
    if v == x:
      return(True)
  return(False)
```

Search problem

- Is value `v` present in list `l`?
- Naive solution scans the list
- Input size n, the length of the list

```
def naivesearch(v,l):
  for x in l:
    if v == x:
      return(True)
  return(False)
```

Search problem

- Is value `v` present in list `l`?
- Naive solution scans the list
- Input size n, the length of the list
- Worst case is when `v` is not present in `l`

```
def naivesearch(v,l):
  for x in l:
    if v == x:
      return(True)
  return(False)
```

Search problem

- Is value `v` present in list `l`?
- Naive solution scans the list
- Input size n, the length of the list
- Worst case is when `v` is not present in `l`
- Worst case complexity is $O(n)$

```
def naivesearch(v,l):
  for x in l:
    if v == x:
      return(True)
  return(False)
```

Searching a sorted list

- What if `l` is sorted in ascending order?

Searching a sorted list

- What if `l` is sorted in ascending order?
- Compare `v` with the midpoint of `l`

Searching a sorted list

- What if `l` is sorted in ascending order?
- Compare `v` with the midpoint of `l`
  - If midpoint is `v`, the value is found
  - If `v` less than midpoint, search the first half
  - If `v` greater than midpoint, search the second half
  - Stop when the interval to search becomes empty

```
def binarysearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(binarysearch(v,l[:m]))
  else:
    return(binarysearch(v,l[m+1:]))
```

Searching a sorted list

- What if l is sorted in ascending order?
- Compare v with the midpoint of l
  - If midpoint is v, the value is found
  - If v less than midpoint, search the first half
  - If v greater than midpoint, search the second half
  - Stop when the interval to search becomes empty
- Binary search

```python
def binarysearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(binarysearch(v,l[:m]))
  else:
    return(binarysearch(v,l[m+1:]))
```

Binary search

- How long does this take?

```
def binarysearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(binarysearch(v,l[:m]))
  else:
    return(binarysearch(v,l[m+1:]))
```

Binary search

- How long does this take?
  - Each call halves the interval to search
  - Stop when the interval become empty
- $\log n$ — number of times to divide n by 2 to reach 1
  - $1 // 2 = 0$, so next call reaches empty interval

```
def binarysearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(binarysearch(v,l[:m]))
  else:
    return(binarysearch(v,l[m+1:]))
```

Binary search

- How long does this take?
  - Each call halves the interval to search
  - Stop when the interval become empty
- $\log n$ — number of times to divide n by 2 to reach 1
  - $1 // 2 = 0$, so next call reaches empty interval
- $O(\log n)$ steps

```
def binarysearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(binarysearch(v,l[:m]))
  else:
    return(binarysearch(v,l[m+1:]))
```

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n // 2) + 1$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(bsearch(v,l[:m]))
  else:
    return(bsearch(v,l[m+1:]))
```

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n \,//\, 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n \,//\, 2) + 1$, $n > 0$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(bsearch(v,l[:m]))
  else:
    return(bsearch(v,l[m+1:]))
```

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n /\!/ 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n /\!/ 2) + 1, \; n > 0$
- Solve by "unwinding"

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(bsearch(v,l[:m]))
  else:
    return(bsearch(v,l[m+1:]))
```

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n \,//\, 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n \,//\, 2) + 1$, $n > 0$
- Solve by "unwinding"
- $T(n) \quad = T(n \,//\, 2) + 1$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(bsearch(v,l[:m]))
  else:
    return(bsearch(v,l[m+1:]))
```

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n /\!/ 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n /\!/ 2) + 1$, $n > 0$
- Solve by "unwinding"
- $T(n) \;\; = T(n /\!/ 2) + 1$
  $\qquad\quad = (T(n /\!/ 4) + 1) + 1$

```python
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(bsearch(v,l[:m]))
  else:
    return(bsearch(v,l[m+1:]))
```

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n \,//\, 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n \,//\, 2) + 1$, $n > 0$
- Solve by "unwinding"
- $$\begin{aligned} T(n) &= T(n \,//\, 2) + 1 \\ &= (T(n \,//\, 4) + 1) + 1 = T(n \,//\, 2^2) + \underbrace{1 + 1}_{2} \end{aligned}$$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(bsearch(v,l[:m]))
  else:
    return(bsearch(v,l[m+1:]))
```

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n \,//\, 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n \,//\, 2) + 1$, $n > 0$
- Solve by "unwinding"
- $$\begin{aligned} T(n) &= T(n \,//\, 2) + 1 \\ &= (T(n \,//\, 4) + 1) + 1 = T(n \,//\, 2^2) + \underbrace{1 + 1}_{2} \\ &= \cdots \\ &= T(n \,//\, 2^k) + \underbrace{1 + \cdots + 1}_{k} \end{aligned}$$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(bsearch(v,l[:m]))
  else:
    return(bsearch(v,l[m+1:]))
```

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n \,//\, 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n \,//\, 2) + 1$, $n > 0$
- Solve by "unwinding"
- $$\begin{aligned} T(n) &= T(n \,//\, 2) + 1 \\ &= (T(n \,//\, 4) + 1) + 1 = T(n \,//\, 2^2) + \underbrace{1 + 1}_{2} \\ &= \cdots \\ &= T(n \,//\, 2^k) + \underbrace{1 + \cdots + 1}_{k} \\ &= T(1) + k, \text{ for } k = \log n \end{aligned}$$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(bsearch(v,l[:m]))
  else:
    return(bsearch(v,l[m+1:]))
```

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n \,//\, 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n \,//\, 2) + 1$, $n > 0$
- Solve by "unwinding"
- $$\begin{aligned} T(n) &= T(n \,//\, 2) + 1 \\ &= (T(n \,//\, 4) + 1) + 1 = T(n \,//\, 2^2) + \underbrace{1 + 1}_{2} \\ &= \cdots \\ &= T(n \,//\, 2^k) + \underbrace{1 + \cdots + 1}_{k} \\ &= T(1) + k, \text{ for } k = \log n \\ &= (T(0) + 1) + \log n = 2 + \log n \end{aligned}$$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(bsearch(v,l[:m]))
  else:
    return(bsearch(v,l[m+1:]))
```

Summary

- Search in an unsorted list takes time $O(n)$
  - Need to scan the entire list
  - Worst case is when the value is not present in the list

Summary

- Search in an unsorted list takes time $O(n)$
  - Need to scan the entire list
  - Worst case is when the value is not present in the list
- For a sorted list, binary search takes time $O(\log n)$
  - Halve the interval to search each time

Summary

- Search in an unsorted list takes time $O(n)$
  - Need to scan the entire list
  - Worst case is when the value is not present in the list
- For a sorted list, binary search takes time $O(\log n)$
  - Halve the interval to search each time
- In a sorted list, we can determine that v is absent by examining just $\log n$ values!

Selection Sort

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 2

Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values

Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values
- How do we sort a list?

Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values
- How do we sort a list?
- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values
- How do we sort a list?
- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 1

- Scan the entire pile and find the paper with minimum marks

Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values
- How do we sort a list?
- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 1

- Scan the entire pile and find the paper with minimum marks
- Move this paper to a new pile

Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values
- How do we sort a list?
- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 1

- Scan the entire pile and find the paper with minimum marks
- Move this paper to a new pile
- Repeat with the remaining papers
  - Add the paper with next minimum marks to the second pile each time

Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values
- How do we sort a list?
- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 1

- Scan the entire pile and find the paper with minimum marks
- Move this paper to a new pile
- Repeat with the remaining papers
  - Add the paper with next minimum marks to the second pile each time
- Eventually, the new pile is sorted in descending order

Sorting a list

74	32	89	55	21	64

Sorting a list

74	32	89	55	~~21~~	64

21

Sorting a list

74	~~32~~	89	55	~~21~~	64

21	32

Sorting a list

74	~~32~~	89	~~55~~	~~21~~	64
21	32	55			

Sorting a list

74 ~~32~~ 89 ~~55~~ ~~21~~ ~~64~~

21 32 55 64

Sorting a list

~~74~~	~~32~~	89	~~55~~	~~21~~	~~64~~
21	32	55	64	74	

Sorting a list

~~74~~ ~~32~~ ~~89~~ ~~55~~ ~~21~~ ~~64~~

21 32 55 64 74 89

Selection sort

- Select the next element in sorted order

Selection sort

- Select the next element in sorted order
- Append it to the final sorted list

Selection sort

- Select the next element in sorted order
- Append it to the final sorted list
- Avoid using a second list
  - Swap the minimum element into the first position
  - Swap the second minimum element into the second position
  - ...

Selection sort

- Select the next element in sorted order
- Append it to the final sorted list
- Avoid using a second list
  - Swap the minimum element into the first position
  - Swap the second minimum element into the second position
  - ...
- Eventually the list is rearranged in place in ascending order

Selection sort

- Select the next element in sorted order
- Append it to the final sorted list
- Avoid using a second list
  - Swap the minimum element into the first position
  - Swap the second minimum element into the second position
  - ...
- Eventually the list is rearranged in place in ascending order

```
def SelectionSort(L):
  n = len(L)
  if n < 1:
    return(L)
  for i in range(n):
    # Assume L[:i] is sorted
    mpos = i
    # mpos: position of minimum in L[i:]
    for j in range(i+1,n):
      if L[j] < L[mpos]:
        mpos = j
    # L[mpos] : smallest value in L[i:]
    # Exchange L[mpos] and L[i]
    (L[i],L[mpos]) = (L[mpos],L[i])
    # Now L[:i+1] is sorted
  return(L)
```

Analysis of selection sort

- Correctness follows from the invariant

```
def SelectionSort(L):
  n = len(L)
  if n < 1:
    return(L)
  for i in range(n):
    # Assume L[:i] is sorted
    mpos = i
    # mpos: position of minimum in L[i:]
    for j in range(i+1,n):
      if L[j] < L[mpos]:
        mpos = j
    # L[mpos] : smallest value in L[i:]
    # Exchange L[mpos] and L[i]
    (L[i],L[mpos]) = (L[mpos],L[i])
    # Now L[:i+1] is sorted
  return(L)
```

Analysis of selection sort

- Correctness follows from the invariant
- Efficiency

```
def SelectionSort(L):
  n = len(L)
  if n < 1:
    return(L)
  for i in range(n):
    # Assume L[:i] is sorted
    mpos = i
    # mpos: position of minimum in L[i:]
    for j in range(i+1,n):
      if L[j] < L[mpos]:
        mpos = j
    # L[mpos] : smallest value in L[i:]
    # Exchange L[mpos] and L[i]
    (L[i],L[mpos]) = (L[mpos],L[i])
    # Now L[:i+1] is sorted
  return(L)
```

Analysis of selection sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times

```
def SelectionSort(L):
  n = len(L)
  if n < 1:
    return(L)
  for i in range(n):
    # Assume L[:i] is sorted
    mpos = i
    # mpos: position of minimum in L[i:]
    for j in range(i+1,n):
      if L[j] < L[mpos]:
        mpos = j
    # L[mpos] : smallest value in L[i:]
    # Exchange L[mpos] and L[i]
    (L[i],L[mpos]) = (L[mpos],L[i])
    # Now L[:i+1] is sorted
  return(L)
```

Analysis of selection sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times
  - Inner loop: $n - i$ steps to find minimum in `L[i:]`

```
def SelectionSort(L):
    n = len(L)
    if n < 1:
        return(L)
    for i in range(n):
        # Assume L[:i] is sorted
        mpos = i
        # mpos: position of minimum in L[i:]
        for j in range(i+1,n):
          if L[j] < L[mpos]:
              mpos = j
        # L[mpos] : smallest value in L[i:]
        # Exchange L[mpos] and L[i]
        (L[i],L[mpos]) = (L[mpos],L[i])
        # Now L[:i+1] is sorted
    return(L)
```

Analysis of selection sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times
  - Inner loop: $n - i$ steps to find minimum in `L[i:]`
  - $T(n) = n + (n-1) + \cdots + 1$

```
def SelectionSort(L):
  n = len(L)
  if n < 1:
    return(L)
  for i in range(n):
    # Assume L[:i] is sorted
    mpos = i
    # mpos: position of minimum in L[i:]
    for j in range(i+1,n):
      if L[j] < L[mpos]:
        mpos = j
    # L[mpos] : smallest value in L[i:]
    # Exchange L[mpos] and L[i]
    (L[i],L[mpos]) = (L[mpos],L[i])
    # Now L[:i+1] is sorted
  return(L)
```

Analysis of selection sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times
  - Inner loop: $n - i$ steps to find minimum in `L[i:]`
  - $T(n) = n + (n-1) + \cdots + 1$
  - $T(n) = n(n+1)/2$

```
def SelectionSort(L):
  n = len(L)
  if n < 1:
    return(L)
  for i in range(n):
    # Assume L[:i] is sorted
    mpos = i
    # mpos: position of minimum in L[i:]
    for j in range(i+1,n):
      if L[j] < L[mpos]:
        mpos = j
    # L[mpos] : smallest value in L[i:]
    # Exchange L[mpos] and L[i]
    (L[i],L[mpos]) = (L[mpos],L[i])
    # Now L[:i+1] is sorted
  return(L)
```

Analysis of selection sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times
  - Inner loop: $n - i$ steps to find minimum in `L[i:]`
  - $T(n) = n + (n-1) + \cdots + 1$
  - $T(n) = n(n+1)/2$
- $T(n)$ is $O(n^2)$

```
def SelectionSort(L):
  n = len(L)
  if n < 1:
    return(L)
  for i in range(n):
    # Assume L[:i] is sorted
    mpos = i
    # mpos: position of minimum in L[i:]
    for j in range(i+1,n):
      if L[j] < L[mpos]:
        mpos = j
    # L[mpos] : smallest value in L[i:]
    # Exchange L[mpos] and L[i]
    (L[i],L[mpos]) = (L[mpos],L[i])
    # Now L[:i+1] is sorted
  return(L)
```

Summary

- Selection sort is an intuitive algorithm to sort a list

Summary

- Selection sort is an intuitive algorithm to sort a list
- Repeatedly find the minimum (or maximum) and append to sorted list

Summary

- Selection sort is an intuitive algorithm to sort a list
- Repeatedly find the minimum (or maximum) and append to sorted list
- Worst case complexity is $O(n^2)$
  - Every input takes this much time
  - No advantage even if list is arranged carefully before sorting

Insertion Sort

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 2

Sorting a list

- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Sorting a list

- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 2

Sorting a list

- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 2

- Move the first paper to a new pile

Sorting a list

- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 2

- Move the first paper to a new pile
- Second paper
  - Lower marks than first paper? Place below first paper in new pile
  - Higher marks than first paper? Place above first paper in new pile

Sorting a list

- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 2

- Move the first paper to a new pile
- Second paper
  - Lower marks than first paper? Place below first paper in new pile
  - Higher marks than first paper? Place above first paper in new pile
- Third paper
  - Insert into correct position with respect to first two

Sorting a list

- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 2

- Move the first paper to a new pile
- Second paper
  - Lower marks than first paper? Place below first paper in new pile
  - Higher marks than first paper? Place above first paper in new pile
- Third paper
  - Insert into correct position with respect to first two
- Do this for the remaining papers
  - Insert each one into correct position in the second pile

Sorting a list

74	32	89	55	21	64

Sorting a list

~~74~~ 32 89 55 21 64

74

Sorting a list

~~74~~	~~32~~	89	55	21	64
32	74				

Sorting a list

~~74~~	~~32~~	~~89~~	55	21	64

32	74	89

Sorting a list

~~74~~ ~~32~~ ~~89~~ ~~55~~ 21 64

32 55 74 89

Sorting a list

~~74~~ ~~32~~ ~~89~~ ~~55~~ ~~21~~ 64

21 32 55 74 89

Sorting a list

~~74~~ ~~32~~ ~~89~~ ~~55~~ ~~21~~ ~~64~~

21 32 55 64 74 89

Insertion sort

- Start building a new sorted list

Insertion sort

- Start building a new sorted list
- Pick next element and insert it into the sorted list

Insertion sort

- Start building a new sorted list
- Pick next element and insert it into the sorted list
- An iterative formulation
  - Assume `L[:i]` is sorted
  - Insert `L[i]` in `L[:i]`

Insertion sort

- Start building a new sorted list
- Pick next element and insert it into the sorted list
- An iterative formulation
  - Assume `L[:i]` is sorted
  - Insert `L[i]` in `L[:i]`

```
def InsertionSort(L):
  n = len(L)
  if n < 1:
    return(L)
  for i in range(n):
    # Assume L[:i] is sorted
    # Move L[i] to correct position in L
    j = i
    while(j > 0 and L[j] < L[j-1]):
      (L[j],L[j-1]) = (L[j-1],L[j])
      j = j-1
    # Now L[:i+1] is sorted
  return(L)
```

Insertion sort

- Start building a new sorted list
- Pick next element and insert it into the sorted list
- An iterative formulation
  - Assume `L[:i]` is sorted
  - Insert `L[i]` in `L[:i]`
- A recursive formulation
  - Inductively sort `L[:i]`
  - Insert `L[i]` in `L[:i]`

```
def InsertionSort(L):
  n = len(L)
  if n < 1:
    return(L)
  for i in range(n):
    # Assume L[:i] is sorted
    # Move L[i] to correct position in L
    j = i
    while(j > 0 and L[j] < L[j-1]):
      (L[j],L[j-1]) = (L[j-1],L[j])
      j = j-1
    # Now L[:i+1] is sorted
  return(L)
```

Insertion sort

- Start building a new sorted list
- Pick next element and insert it into the sorted list
- An iterative formulation
  - Assume `L[:i]` is sorted
  - Insert `L[i]` in `L[:i]`
- A recursive formulation
  - Inductively sort `L[:i]`
  - Insert `L[i]` in `L[:i]`

```
def Insert(L,v):
    n = len(L)
    if n == 0:
      return([v])
    if v >= L[-1]:
      return(L+[v])
    else:
      return(Insert(L[:-1],v)+L[-1:])

def ISort(L):
    n = len(L)
    if n < 1:
       return(L)
    L = Insert(ISort(L[:-1]),L[-1])
    return(L)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant

```
def InsertionSort(L):
    n = len(L)
    if n < 1:
       return(L)
    for i in range(n):
       # Assume L[:i] is sorted
       # Move L[i] to correct position in L
       j = i
       while(L[j] < L[j-1]):
         (L[j],L[j-1]) = (L[j-1],L[j])
         j = j-1
       # Now L[:i+1] is sorted
    return(L)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency

```
def InsertionSort(L):
    n = len(L)
    if n < 1:
        return(L)
    for i in range(n):
        # Assume L[:i] is sorted
        # Move L[i] to correct position in L
        j = i
        while(L[j] < L[j-1]):
          (L[j],L[j-1]) = (L[j-1],L[j])
          j = j-1
        # Now L[:i+1] is sorted
    return(L)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times

```
def InsertionSort(L):
    n = len(L)
    if n < 1:
        return(L)
    for i in range(n):
        # Assume L[:i] is sorted
        # Move L[i] to correct position in L
        j = i
        while(L[j] < L[j-1]):
          (L[j],L[j-1]) = (L[j-1],L[j])
          j = j-1
        # Now L[:i+1] is sorted
    return(L)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times
  - Inner loop: i steps to insert `L[i]` in `L[:i]`

```
def InsertionSort(L):
  n = len(L)
  if n < 1:
    return(L)
  for i in range(n):
    # Assume L[:i] is sorted
    # Move L[i] to correct position in L
    j = i
    while(L[j] < L[j-1]):
      (L[j],L[j-1]) = (L[j-1],L[j])
      j = j-1
    # Now L[:i+1] is sorted
  return(L)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times
  - Inner loop: i steps to insert `L[i]` in `L[:i]`
  - $T(n) = 0 + 1 + \cdots + (n-1)$

```
def InsertionSort(L):
  n = len(L)
  if n < 1:
    return(L)
  for i in range(n):
    # Assume L[:i] is sorted
    # Move L[i] to correct position in L
    j = i
    while(L[j] < L[j-1]):
      (L[j],L[j-1]) = (L[j-1],L[j])
      j = j-1
    # Now L[:i+1] is sorted
  return(L)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times
  - Inner loop: i steps to insert `L[i]` in `L[:i]`
  - $T(n) = 0 + 1 + \cdots + (n-1)$
  - $T(n) = n(n-1)/2$

```
def InsertionSort(L):
   n = len(L)
   if n < 1:
      return(L)
   for i in range(n):
      # Assume L[:i] is sorted
      # Move L[i] to correct position in L
      j = i
      while(L[j] < L[j-1]):
        (L[j],L[j-1]) = (L[j-1],L[j])
        j = j-1
      # Now L[:i+1] is sorted
   return(L)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times
  - Inner loop: i steps to insert `L[i]` in `L[:i]`
  - $T(n) = 0 + 1 + \cdots + (n-1)$
  - $T(n) = n(n-1)/2$
- $T(n)$ is $O(n^2)$

```
def InsertionSort(L):
    n = len(L)
    if n < 1:
       return(L)
    for i in range(n):
       # Assume L[:i] is sorted
       # Move L[i] to correct position in L
       j = i
       while(L[j] < L[j-1]):
         (L[j],L[j-1]) = (L[j-1],L[j])
         j = j-1
       # Now L[:i+1] is sorted
    return(L)
```

Analysis of recursive insertion sort

- For input of size n, let
  - $TI(n)$ be the time taken by `Insert`
  - $TS(n)$ be the time taken by `ISort`

```
def Insert(L,v):
  n = len(L)
  if n == 0:
    return([v])
  if v >= L[-1]:
    return(L+[v])
  else
    return(Insert(L[:-1],v)+l[-1:])

def ISort(L):
  n = len(L)
  if n < 1:
     return(L)
  L = Insert(ISort(L[:-1]),L[-1])
  return(L)
```

Analysis of recursive insertion sort

- For input of size n, let
  - $TI(n)$ be the time taken by `Insert`
  - $TS(n)$ be the time taken by `ISort`
- First calculate $TI(n)$ for `Insert`
  - $TI(0) = 1$
  - $TI(n) = TI(n-1) + 1$

```
def Insert(L,v):
  n = len(L)
  if n == 0:
    return([v])
  if v >= L[-1]:
    return(L+[v])
  else
    return(Insert(L[:-1],v)+l[-1:])

def ISort(L):
  n = len(L)
  if n < 1:
     return(L)
  L = Insert(ISort(L[:-1]),L[-1])
  return(L)
```

Analysis of recursive insertion sort

- For input of size n, let
  - $TI(n)$ be the time taken by `Insert`
  - $TS(n)$ be the time taken by `ISort`
- First calculate $TI(n)$ for `Insert`
  - $TI(0) = 1$
  - $TI(n) = TI(n-1) + 1$
  - Unwind to get $TI(n) = n$

```
def Insert(L,v):
  n = len(L)
  if n == 0:
    return([v])
  if v >= L[-1]:
    return(L+[v])
  else
    return(Insert(L[:-1],v)+l[-1:])

def ISort(L):
  n = len(L)
  if n < 1:
     return(L)
  L = Insert(ISort(L[:-1]),L[-1])
  return(L)
```

Analysis of recursive insertion sort

- For input of size n, let
  - $TI(n)$ be the time taken by `Insert`
  - $TS(n)$ be the time taken by `ISort`
- First calculate $TI(n)$ for `Insert`
  - $TI(0) = 1$
  - $TI(n) = TI(n-1) + 1$
  - Unwind to get $TI(n) = n$
- Set up a recurrence for $TS(n)$
  - $TS(0) = 1$
  - $TS(n) = TS(n-1) + TI(n-1)$

```
def Insert(L,v):
  n = len(L)
  if n == 0:
    return([v])
  if v >= L[-1]:
    return(L+[v])
  else
    return(Insert(L[:-1],v)+l[-1:])

def ISort(L):
  n = len(L)
  if n < 1:
     return(L)
  L = Insert(ISort(L[:-1]),L[-1])
  return(L)
```

Analysis of recursive insertion sort

- For input of size n, let
  - $TI(n)$ be the time taken by `Insert`
  - $TS(n)$ be the time taken by `ISort`
- First calculate $TI(n)$ for `Insert`
  - $TI(0) = 1$
  - $TI(n) = TI(n-1) + 1$
  - Unwind to get $TI(n) = n$
- Set up a recurrence for $TS(n)$
  - $TS(0) = 1$
  - $TS(n) = TS(n-1) + TI(n-1)$
- Unwind to get $1 + 2 + \cdots + n - 1$

```
def Insert(L,v):
   n = len(L)
   if n == 0:
     return([v])
   if v >= L[-1]:
     return(L+[v])
   else
     return(Insert(L[:-1],v)+l[-1:])

def ISort(L):
   n = len(L)
   if n < 1:
      return(L)
   L = Insert(ISort(L[:-1]),L[-1])
   return(L)
```

Summary

- Insertion sort is another intuitive algorithm to sort a list

Summary

- Insertion sort is another intuitive algorithm to sort a list
- Create a new sorted list
- Repeatedly insert elements into the sorted list

Summary

- Insertion sort is another intuitive algorithm to sort a list
- Create a new sorted list
- Repeatedly insert elements into the sorted list
- Worst case complexity is $O(n^2)$
  - Unlike selection sort, not all cases take time n^2
  - If list is already sorted, `Insert` stops in 1 step
  - Overall time can be close to $O(n)$

Merge Sort

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 2

Beating the $O(n^2)$ barrier

- Both selection sort and insertion sort take time $O(n^2)$
- This is infeasible for $n > 10000$

Beating the $O(n^2)$ barrier

- Both selection sort and insertion sort take time $O(n^2)$
- This is infeasible for $n > 10000$
- How can we bring the complexity below $O(n^2)$?

Beating the $O(n^2)$ barrier

- Both selection sort and insertion sort take time $O(n^2)$
- This is infeasible for $n > 10000$
- How can we bring the complexity below $O(n^2)$?

Strategy 3

- Divide the list into two halves

Beating the $O(n^2)$ barrier

- Both selection sort and insertion sort take time $O(n^2)$
- This is infeasible for $n > 10000$
- How can we bring the complexity below $O(n^2)$?

Strategy 3

- Divide the list into two halves
- Separately sort the left and right half

Beating the $O(n^2)$ barrier

- Both selection sort and insertion sort take time $O(n^2)$
- This is infeasible for $n > 10000$
- How can we bring the complexity below $O(n^2)$?

Strategy 3

- Divide the list into two halves
- Separately sort the left and right half
- Combine the two sorted halves to get a fully sorted list

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`
  - Compare first elements of `A` and `B`

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`
    - Compare first elements of `A` and `B`
    - Move the smaller of the two to `C`

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`
  - Compare first elements of `A` and `B`
  - Move the smaller of the two to `C`
  - Repeat till you exhaust `A` and `B`

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`
  - Compare first elements of `A` and `B`
  - Move the smaller of the two to `C`
  - Repeat till you exhaust `A` and `B`

32	74	89
21	55	64

Combining two sorted lists

- Combine two sorted lists A and B into a single sorted list C
  - Compare first elements of A and B
  - Move the smaller of the two to C
  - Repeat till you exhaust A and B

32	74	89
~~21~~	55	64

21

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`
  - Compare first elements of `A` and `B`
  - Move the smaller of the two to `C`
  - Repeat till you exhaust `A` and `B`

~~32~~	74	89
~~21~~	55	64

21	32

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`
  - Compare first elements of `A` and `B`
  - Move the smaller of the two to `C`
  - Repeat till you exhaust `A` and `B`

~~32~~ 74 89

~~21~~ ~~55~~ 64

21 32 55

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`
  - Compare first elements of `A` and `B`
  - Move the smaller of the two to `C`
  - Repeat till you exhaust `A` and `B`

~~32~~ 74 89

~~21~~ ~~55~~ ~~64~~

21 32 55 64

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`
  - Compare first elements of `A` and `B`
  - Move the smaller of the two to `C`
  - Repeat till you exhaust `A` and `B`

~~32~~	~~74~~	89
~~21~~	~~55~~	~~64~~

21	32	55	64	74

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`
  - Compare first elements of `A` and `B`
  - Move the smaller of the two to `C`
  - Repeat till you exhaust `A` and `B`

~~32~~ ~~74~~ ~~89~~

~~21~~ ~~55~~ ~~64~~

21 32 55 64 74 89

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`
  - Compare first elements of `A` and `B`
  - Move the smaller of the two to `C`
  - Repeat till you exhaust `A` and `B`
- Merging `A` and `B`

~~32~~	~~74~~	~~89~~
~~21~~	~~55~~	~~64~~

21 32 55 64 74 89

Merge sort

- Let `n` be the length of L

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

43	32	22	78	63	57	91	13

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

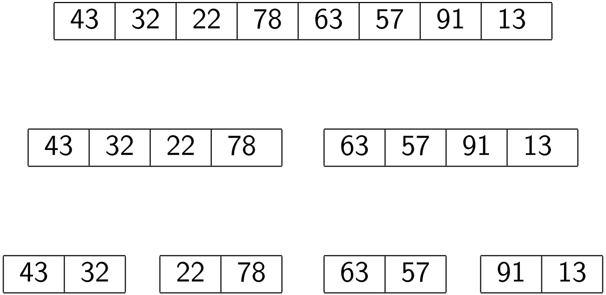
43	32	22	78	63	57	91	13

43	32	22	78

63	57	91	13

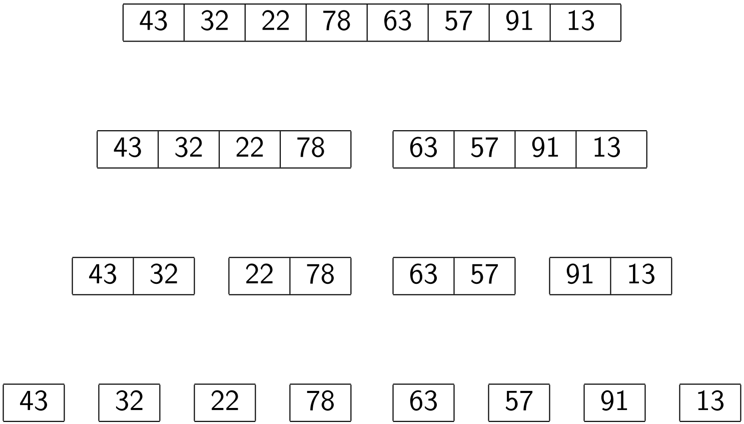
Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!



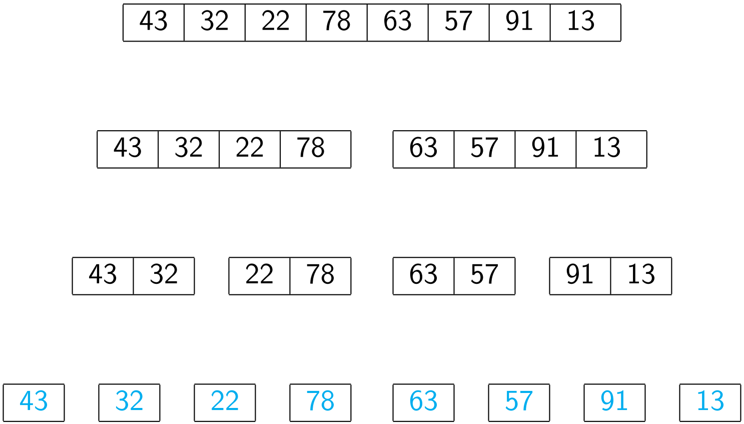
Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!



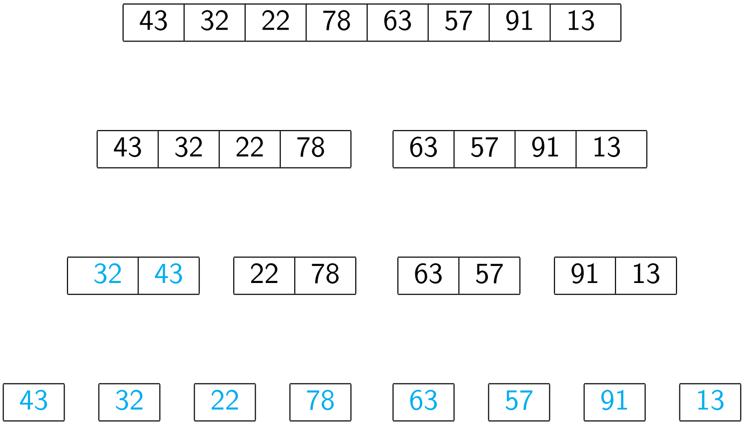
Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!



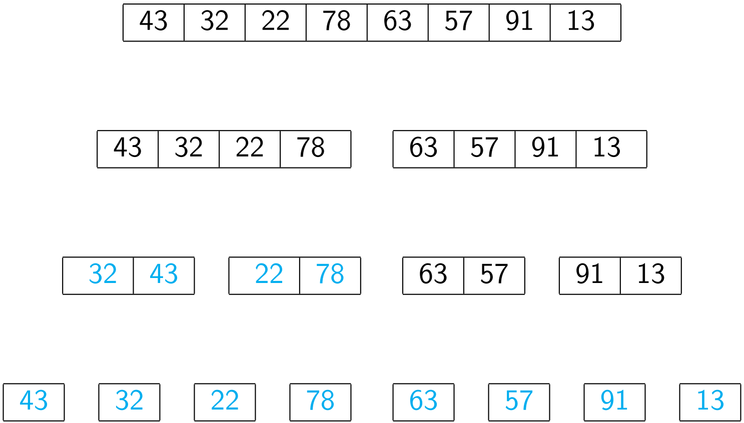
Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!



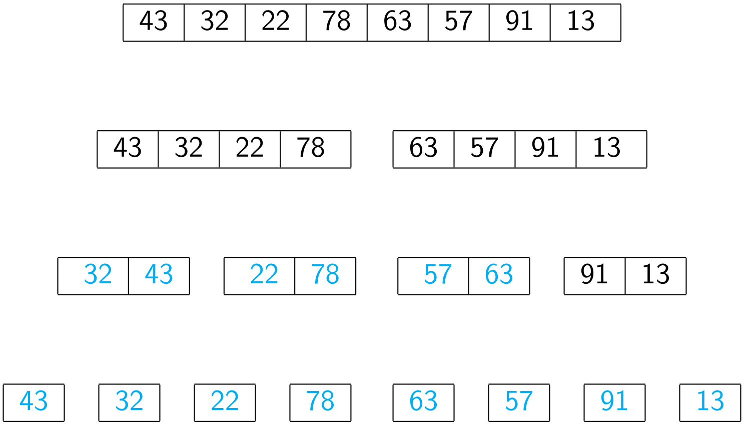
Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!



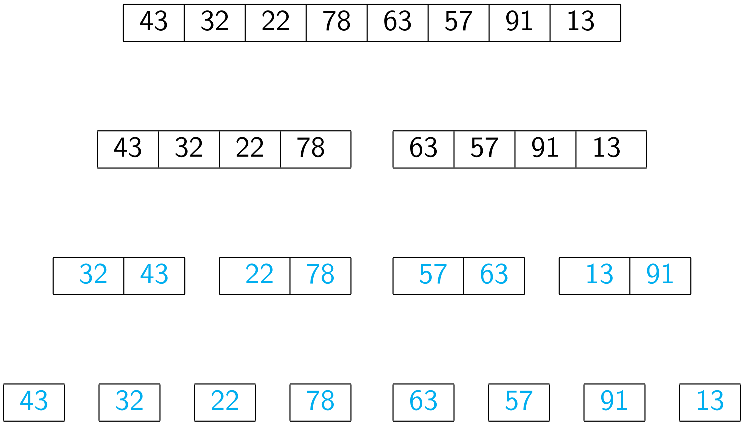
Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!



Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!



Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

43	32	22	78	63	57	91	13

43	32	22	78

63	57	91	13

32	43

22	78

57	63

13	91

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

43	32	22	78	63	57	91	13

22	32	43	78

63	57	91	13

32	43

22	78

57	63

13	91

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

43	32	22	78	63	57	91	13

22	32	43	78

13	57	63	91

32	43

22	78

57	63

13	91

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

43	32	22	78	63	57	91	13

22	32	43	78

13	57	63	91

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

13	22	32	43	57	63	78	91

22	32	43	78

13	57	63	91

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

13	22	32	43	57	63	78	91

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

Divide and Conquer

- Break up the problem into disjoint parts
- Solve each part separately
- Combine the solutions efficiently

Merging sorted lists

- Combine two sorted lists `A` and `B` into `C`

Merging sorted lists

- Combine two sorted lists `A` and `B` into `C`
  - If `A` is empty, copy `B` into `C`

Merging sorted lists

- Combine two sorted lists `A` and `B` into `C`
  - If `A` is empty, copy `B` into `C`
  - If `B` is empty, copy `A` into `C`

Merging sorted lists

- Combine two sorted lists `A` and `B` into `C`
  - If `A` is empty, copy `B` into `C`
  - If `B` is empty, copy `A` into `C`
  - Otherwise, compare first elements of `A` and `B`
    - Move the smaller of the two to `C`

Merging sorted lists

- Combine two sorted lists `A` and `B` into `C`
  - If `A` is empty, copy `B` into `C`
  - If `B` is empty, copy `A` into `C`
  - Otherwise, compare first elements of `A` and `B`
    - Move the smaller of the two to `C`
  - Repeat till all elements of `A` and `B` have been moved

Merging sorted lists

- Combine two sorted lists A and B into C
  - If A is empty, copy B into C
  - If B is empty, copy A into C
  - Otherwise, compare first elements of A and B
    - Move the smaller of the two to C
  - Repeat till all elements of A and B have been moved

```
def merge(A,B):
  (m,n) = (len(A),len(B))
  (C,i,j,k) = ([],0,0,0)
  while k < m+n:
    if i == m:
      C.extend(B[j:])
      k = k + (n-j)
    elif j == n:
      C.extend(A[i:])
      k = k + (m-i)
    elif A[i] < B[j]:
      C.append(A[i])
      (i,k) = (i+1,k+1)
    else:
      C.append(B[j])
      (j,k) = (j+1,k+1)
  return(C)
```

Merge sort

- To sort `A` into `B`, both of length n

Merge sort

- To sort `A` into `B`, both of length n
- If $n \leq 1$, nothing to be done

Merge sort

- To sort `A` into `B`, both of length n
- If $n \leq 1$, nothing to be done
- Otherwise

Merge sort

- To sort `A` into `B`, both of length n
- If $n \leq 1$, nothing to be done
- Otherwise
  - Sort `A[:n//2]` into `L`

Merge sort

- To sort `A` into `B`, both of length n
- If $n \leq 1$, nothing to be done
- Otherwise
  - Sort `A[:n//2]` into `L`
  - Sort `A[n//2:]` into `R`

Merge sort

- To sort `A` into `B`, both of length n
- If $n \leq 1$, nothing to be done
- Otherwise
  - Sort `A[:n//2]` into `L`
  - Sort `A[n//2:]` into `R`
  - Merge `L` and `R` into `B`

Merge sort

- To sort `A` into `B`, both of length n
- If $n \leq 1$, nothing to be done
- Otherwise
  - Sort `A[:n//2]` into `L`
  - Sort `A[n//2:]` into `R`
  - Merge `L` and `R` into `B`

```
def mergesort(A):
  n = len(A)

  if n <= 1:
     return(A)

  L = mergesort(A[:n//2])
  R = mergesort(A[n//2:])

  B = merge(L,R)

  return(B)
```

Summary

- Merge sort using divide and conquer to sort a list

Summary

- Merge sort using divide and conquer to sort a list
- Divide the list into two halves

Summary

- Merge sort using divide and conquer to sort a list
- Divide the list into two halves
- Sort each half

Summary

- Merge sort using divide and conquer to sort a list
- Divide the list into two halves
- Sort each half
- Merge the sorted halves

Summary

- Merge sort using divide and conquer to sort a list
- Divide the list into two halves
- Sort each half
- Merge the sorted halves
- Next, we have to check that the complexity is less than $O(n^2)$

Analysis of Merge Sort

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 2

Merge sort

- To sort `A` into `B`, both of length n
- If $n \leq 1$, nothing to be done
- Otherwise
  - Sort `A[:n//2]` into `L`
  - Sort `A[n//2:]` into `R`
  - Merge `L` and `R` into `B`

Merging two sorted lists `A` and `B` into `C`

- If `A` is empty, copy `B` into `C`
- If `B` is empty, copy `A` into `C`
- Otherwise, compare first elements of `A` and `B`
  - Move the smaller of the two to `C`
- Repeat till all elements of `A` and `B` have been moved

Analysing merge

- Merge A of length m, B of length n

```
def merge(A,B):
  (m,n) = (len(A),len(B))
  (C,i,j,k) = ([],0,0,0)
  while k < m+n:
    if i == m:
      C.extend(B[j:])
      k = k + (n-j)
    elif j == n:
      C.extend(A[i:])
      k = k + (m-i)
    elif A[i] < B[j]:
      C.append(A[i])
      (i,k) = (i+1,k+1)
    else:
      C.append(B[j])
      (j,k) = (j+1,k+1)
  return(C)
```

Analysing merge

- Merge `A` of length m, `B` of length n
- Output list `C` has length $m + n$

```
def merge(A,B):
  (m,n) = (len(A),len(B))
  (C,i,j,k) = ([],0,0,0)
  while k < m+n:
    if i == m:
      C.extend(B[j:])
      k = k + (n-j)
    elif j == n:
      C.extend(A[i:])
      k = k + (m-i)
    elif A[i] < B[j]:
      C.append(A[i])
      (i,k) = (i+1,k+1)
    else:
      C.append(B[j])
      (j,k) = (j+1,k+1)
  return(C)
```

Analysing merge

- Merge A of length m, B of length n
- Output list C has length $m + n$
- In each iteration we add (at least) one element to C

```
def merge(A,B):
  (m,n) = (len(A),len(B))
  (C,i,j,k) = ([],0,0,0)
  while k < m+n:
    if i == m:
      C.extend(B[j:])
      k = k + (n-j)
    elif j == n:
      C.extend(A[i:])
      k = k + (m-i)
    elif A[i] < B[j]:
      C.append(A[i])
      (i,k) = (i+1,k+1)
    else:
      C.append(B[j])
      (j,k) = (j+1,k+1)
  return(C)
```

Analysing merge

- Merge `A` of length m, `B` of length n
- Output list `C` has length $m + n$
- In each iteration we add (at least) one element to `C`
- Hence `merge` takes time $O(m + n)$

```
def merge(A,B):
  (m,n) = (len(A),len(B))
  (C,i,j,k) = ([],0,0,0)
  while k < m+n:
    if i == m:
      C.extend(B[j:])
      k = k + (n-j)
    elif j == n:
      C.extend(A[i:])
      k = k + (m-i)
    elif A[i] < B[j]:
      C.append(A[i])
      (i,k) = (i+1,k+1)
    else:
      C.append(B[j])
      (j,k) = (j+1,k+1)
  return(C)
```

Analysing merge

- Merge `A` of length m, `B` of length n
- Output list `C` has length $m + n$
- In each iteration we add (at least) one element to `C`
- Hence `merge` takes time $O(m + n)$
- Recall that $m + n \leq 2(\max(m, n))$

```
def merge(A,B):
  (m,n) = (len(A),len(B))
  (C,i,j,k) = ([],0,0,0)
  while k < m+n:
    if i == m:
      C.extend(B[j:])
      k = k + (n-j)
    elif j == n:
      C.extend(A[i:])
      k = k + (m-i)
    elif A[i] < B[j]:
      C.append(A[i])
      (i,k) = (i+1,k+1)
    else:
      C.append(B[j])
      (j,k) = (j+1,k+1)
  return(C)
```

Analysing merge

- Merge `A` of length m, `B` of length n
- Output list `C` has length $m + n$
- In each iteration we add (at least) one element to `C`
- Hence `merge` takes time $O(m + n)$
- Recall that $m + n \leq 2(\max(m, n))$
- If $m \approx n$, `merge` take time $O(n)$

```
def merge(A,B):
  (m,n) = (len(A),len(B))
  (C,i,j,k) = ([],0,0,0)
  while k < m+n:
    if i == m:
      C.extend(B[j:])
      k = k + (n-j)
    elif j == n:
      C.extend(A[i:])
      k = k + (m-i)
    elif A[i] < B[j]:
      C.append(A[i])
      (i,k) = (i+1,k+1)
    else:
      C.append(B[j])
      (j,k) = (j+1,k+1)
  return(C)
```

Analysing mergesort

- Let $T(n)$ be the time taken for input of size n
  - For simplicity, assume $n = 2^k$ for some k

```
def mergesort(A):
  n = len(A)

  if n <= 1:
     return(A)

  L = mergesort(A[:n//2])
  R = mergesort(A[n//2:])

  B = merge(L,R)

  return(B)
```

Analysing mergesort

- Let $T(n)$ be the time taken for input of size n
  - For simplicity, assume $n = 2^k$ for some k
- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
    - Solve two subproblems of size $n/2$
    - Merge the solutions in time $n/2 + n/2 = n$

```
def mergesort(A):
  n = len(A)

  if n <= 1:
     return(A)

  L = mergesort(A[:n//2])
  R = mergesort(A[n//2:])

  B = merge(L,R)

  return(B)
```

Analysing mergesort

- Let $T(n)$ be the time taken for input of size n
  - For simplicity, assume $n = 2^k$ for some k
- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
    - Solve two subproblems of size $n/2$
    - Merge the solutions in time $n/2 + n/2 = n$
- Unwind the recurrence to solve

```
def mergesort(A):
  n = len(A)

  if n <= 1:
     return(A)

  L = mergesort(A[:n//2])
  R = mergesort(A[n//2:])

  B = merge(L,R)

  return(B)
```

Analysing mergesort

- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$

```
def mergesort(A):
  n = len(A)

  if n <= 1:
     return(A)

  L = mergesort(A[:n//2])
  R = mergesort(A[n//2:])

  B = merge(L,R)

  return(B)
```

Analysing mergesort

- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
- $T(n) \quad = 2T(n/2) + n$

```
def mergesort(A):
  n = len(A)

  if n <= 1:
     return(A)

  L = mergesort(A[:n//2])
  R = mergesort(A[n//2:])

  B = merge(L,R)

  return(B)
```

Analysing mergesort

- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
- $$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\,[2T(n/4) + n/2] + n \end{aligned}$$

```
def mergesort(A):
  n = len(A)

  if n <= 1:
     return(A)

  L = mergesort(A[:n//2])
  R = mergesort(A[n//2:])

  B = merge(L,R)

  return(B)
```

Analysing mergesort

- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
- $T(n) \;\; = 2T(n/2) + n$

  $= 2\left[2T(n/4) + n/2\right] + n = 2^2 T(n/2^2) + 2n$

```
def mergesort(A):
  n = len(A)

  if n <= 1:
     return(A)

  L = mergesort(A[:n//2])
  R = mergesort(A[n//2:])

  B = merge(L,R)

  return(B)
```

Analysing mergesort

- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
- $$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\,[2T(n/4) + n/2] + n = 2^2\,T(n/2^2) + 2n \\ &= 2^2\left[2T(n/2^3) + n/2^2\right] + 2n = 2^3\,T(n/2^3) + 3n \end{aligned}$$

```
def mergesort(A):
  n = len(A)

  if n <= 1:
     return(A)

  L = mergesort(A[:n//2])
  R = mergesort(A[n//2:])

  B = merge(L,R)

  return(B)
```

Analysing mergesort

- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$

- $$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\left[2T(n/4) + n/2\right] + n = 2^2 T(n/2^2) + 2n \\ &= 2^2\left[2T(n/2^3) + n/2^2\right] + 2n = 2^3 T(n/2^3) + 3n \\ &\ \ \vdots \\ &= 2^k T(n/2^k) + kn \end{aligned}$$

```
def mergesort(A):
  n = len(A)

  if n <= 1:
     return(A)

  L = mergesort(A[:n//2])
  R = mergesort(A[n//2:])

  B = merge(L,R)

  return(B)
```

Analysing mergesort

- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
- $$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\,[2T(n/4) + n/2] + n = 2^2\,T(n/2^2) + 2n \\ &= 2^2\,\left[2T(n/2^3) + n/2^2\right] + 2n = 2^3\,T(n/2^3) + 3n \\ &\;\;\vdots \\ &= 2^k\,T(n/2^k) + kn \end{aligned}$$
- When $k = \log n$, $T(n/2^k) = T(1) = 1$

```
def mergesort(A):
  n = len(A)

  if n <= 1:
     return(A)

  L = mergesort(A[:n//2])
  R = mergesort(A[n//2:])

  B = merge(L,R)

  return(B)
```

Analysing mergesort

- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
- $$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\,[2T(n/4) + n/2] + n = 2^2\,T(n/2^2) + 2n \\ &= 2^2\left[2T(n/2^3) + n/2^2\right] + 2n = 2^3\,T(n/2^3) + 3n \\ &\;\;\vdots \\ &= 2^k\,T(n/2^k) + kn \end{aligned}$$
- When $k = \log n$, $T(n/2^k) = T(1) = 1$
- $T(n) = 2^{\log n}\,T(1) + (\log n)n = n + n\log n$

```
def mergesort(A):
  n = len(A)

  if n <= 1:
     return(A)

  L = mergesort(A[:n//2])
  R = mergesort(A[n//2:])

  B = merge(L,R)

  return(B)
```

Analysing mergesort

- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$

- $$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\,[2T(n/4) + n/2] + n = 2^2\,T(n/2^2) + 2n \\ &= 2^2\,\left[2T(n/2^3) + n/2^2\right] + 2n = 2^3\,T(n/2^3) + 3n \\ &\;\;\vdots \\ &= 2^k\,T(n/2^k) + kn \end{aligned}$$

- When $k = \log n$, $T(n/2^k) = T(1) = 1$
- $T(n) = 2^{\log n}\,T(1) + (\log n)n = n + n\log n$
- Hence $T(n)$ is $O(n \log n)$

```
def mergesort(A):
  n = len(A)

  if n <= 1:
     return(A)

  L = mergesort(A[:n//2])
  R = mergesort(A[n//2:])

  B = merge(L,R)

  return(B)
```

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs
- Variations on merge are possible

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs
- Variations on merge are possible
  - Union of two sorted lists — discard duplicates, if `A[i] == B[j]` move just one copy to `C` and increment both `i` and `j`

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs
- Variations on merge are possible
  - Union of two sorted lists — discard duplicates, if `A[i] == B[j]` move just one copy to `C` and increment both `i` and `j`
  - Intersection of two sorted lists — when `A[i] == B[j]`, move one copy to `C`, otherwise discard the smaller of `A[i]`, `B[j]`

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs
- Variations on merge are possible
  - Union of two sorted lists — discard duplicates, if `A[i] == B[j]` move just one copy to `C` and increment both `i` and `j`
  - Intersection of two sorted lists — when `A[i] == B[j]`, move one copy to `C`, otherwise discard the smaller of `A[i]`, `B[j]`
  - List difference — elements in `A` but not in `B`

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs
- Variations on merge are possible
  - Union of two sorted lists — discard duplicates, if `A[i] == B[j]` move just one copy to `C` and increment both `i` and `j`
  - Intersection of two sorted lists — when `A[i] == B[j]`, move one copy to `C`, otherwise discard the smaller of `A[i]`, `B[j]`
  - List difference — elements in `A` but not in `B`
- Merge needs to create a new list to hold the merged elements
  - No obvious way to efficiently merge two lists in place
  - Extra storage can be costly

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs
- Variations on merge are possible
  - Union of two sorted lists — discard duplicates, if `A[i] == B[j]` move just one copy to `C` and increment both `i` and `j`
  - Intersection of two sorted lists — when `A[i] == B[j]`, move one copy to `C`, otherwise discard the smaller of `A[i]`, `B[j]`
  - List difference — elements in `A` but not in `B`
- Merge needs to create a new list to hold the merged elements
  - No obvious way to efficiently merge two lists in place
  - Extra storage can be costly
- Inherently recursive
  - Recursive calls and returns are expensive