

Analysis of Algorithms

Data Structures and Algorithms for Computational Linguistics III
(ISCL-BA-07)

Çağrı Çöltekin

`ccoltekin@sfs.uni-tuebingen.de`

University of Tübingen
Seminar für Sprachwissenschaft

Winter Semester 2023/24

What are we analyzing?

- So far, we frequently asked: ‘can we do better?’

What are we analyzing?

- So far, we frequently asked: ‘can we do better?’
- Now, we turn to the questions of
 - what is better?
 - how do we know an algorithm is better than the other?

What are we analyzing?

- So far, we frequently asked: ‘can we do better?’
- Now, we turn to the questions of
 - what is better?
 - how do we know an algorithm is better than the other?
- There are many properties that we may want to improve
 - correctness
 - robustness
 - simplicity
 - ...
 - In this lecture, *efficiency* will be our focus
 - in particular time efficiency/complexity

How to determine running time of an algorithm?

write the code, experiment

- A possible approach:
 - Implement the algorithm
 - Test with varying input
 - Analyze the results

How to determine running time of an algorithm?

write the code, experiment

- A possible approach:
 - Implement the algorithm
 - Test with varying input
 - Analyze the results
- A few issues with this approach:
 - Implementing something that does not work is not productive (or fun)
 - It is often not possible to cover all potential inputs
 - If your version takes 10 seconds less than a version reported 10 years ago, do you really have an improvement?

How to determine running time of an algorithm?

write the code, experiment

- A possible approach:
 - Implement the algorithm
 - Test with varying input
 - Analyze the results
- A few issues with this approach:
 - Implementing something that does not work is not productive (or fun)
 - It is often not possible to cover all potential inputs
 - If your version takes 10 seconds less than a version reported 10 years ago, do you really have an improvement?
- A formal approach offers some help here

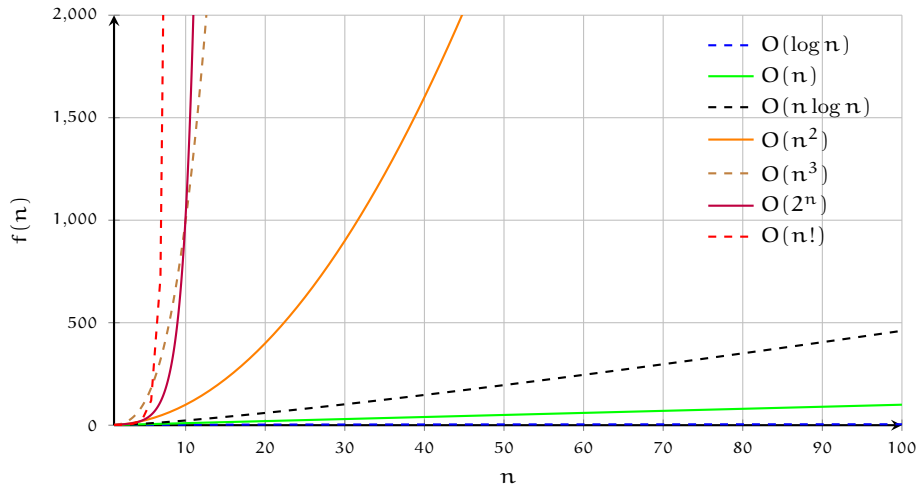
Some functions to know about

Family	Definition
Constant	$f(n) = c$
Logarithmic	$f(n) = \log_b n$
Linear	$f(n) = n$
$N \log N$	$f(n) = n \log n$
Quadratic	$f(n) = n^2$
Cubic	$f(n) = n^3$
Other polynomials	$f(n) = n^k, \text{ for } k > 3$
Exponential	$f(n) = b^n, \text{ for } b > 1$
Factorial	$f(n) = n!$

- We will use these functions to characterize running times of algorithms

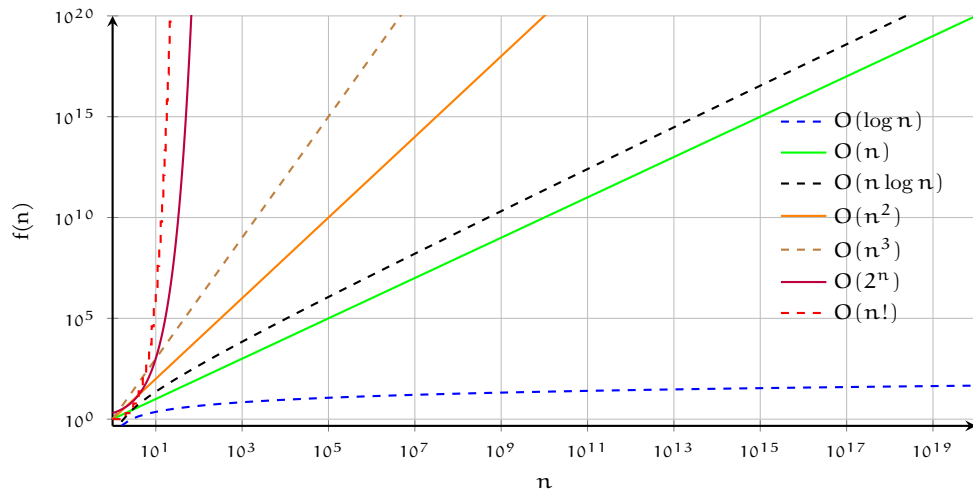
Some functions to know about

the picture - why we care about their difference



Some functions to know about

the bigger picture



A few facts about logarithms

- Logarithm is the inverse of exponentiation:

$$x = \log_b n \iff b^x = n$$

- We will mostly use base-2 logarithms. For us, no-base means base-2
- Additional properties:

$$\log xy = \log x + \log y$$

$$\log \frac{x}{y} = \log x - \log y$$

$$\log x^a = a \log x$$

$$\log_b x = \frac{\log_k x}{\log_k b}$$

- Logarithmic functions grow (much) slower than linear functions

Polynomials

- A degree-0 polynomial is a constant function ($f(n) = c$)
- Degree-1 is linear ($f(n) = n + c$)
- Degree-2 is quadratic ($f(n) = n^2 + n + c$)
- ...
- We generally drop the lower order terms (soon we'll see why)
- Sometimes it will be useful to remember that

$$1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$$

Combinations and permutations

- $n! = n \times (n - 1) \times \dots \times 2 \times 1$
- Permutations:

$$P(n, k) = n \times (n - 1) \times \dots \times (n - k + 1) = \frac{n!}{(n - k)!}$$

- Combinations 'n choose k':

$$C(n, k) = \binom{n}{k} = \frac{P(n, k)}{P(k, k)} = \frac{n!}{(n - k)! \times k!}$$

Proof by induction

- Induction is an important proof technique
- It is often used for both proving the correctness and running times of algorithms
- It works if we can enumerate the steps of an algorithm (loops, recursion)
 - Show that base case holds
 - Assume the result is correct for n , show that it also holds for $n + 1$

Proof by induction

Example: show that $1 + 2 + 3 + \dots + n = n(n + 1)/2$

- Base case, for $n=1$

$$(1 \times 2)/2 = 1$$

- Assuming

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

we need to show that

$$\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$$

Proof by induction

Example: show that $1 + 2 + 3 + \dots + n = n(n + 1)/2$

- Base case, for $n=1$

$$(1 \times 2)/2 = 1$$

- Assuming

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

we need to show that

$$\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$$

$$\frac{n(n+1)}{2} + (n+1)$$

Proof by induction

Example: show that $1 + 2 + 3 + \dots + n = n(n + 1)/2$

- Base case, for $n=1$

$$(1 \times 2)/2 = 1$$

- Assuming

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

we need to show that

$$\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$$

$$\frac{n(n+1)}{2} + (n+1) = \frac{n(n+1) + 2(n+1)}{2}$$

Proof by induction

Example: show that $1 + 2 + 3 + \dots + n = n(n + 1)/2$

- Base case, for $n=1$

$$(1 \times 2)/2 = 1$$

- Assuming

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

we need to show that

$$\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$$

$$\frac{n(n+1)}{2} + (n+1) = \frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)(n+2)}{2}$$

Formal analysis of running time of algorithms

- We are focusing on characterizing running time of algorithms
- The running time is characterized as a function of input size
- We are aiming for an analysis method
 - independent of hardware / software environment
 - does not require implementation before analysis
 - considers all possible inputs

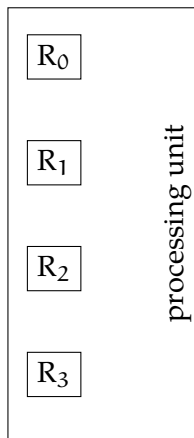
How much hardware independence?

How much hardware independence?

quite, but not completely: we assume a RAM model of computing

- Characterized by random access memory (RAM) (e.g., in comparison to a sequential memory, like a tape)
- We assume the system can perform some primitive operations (addition, comparison) in constant time
- The data and the instructions are stored in the RAM
- The processor fetches them as needed, and executes following the instructions
- This is largely true for any computing system we use in practice

RAM model: an example



10	...
11	load $R_0, 20$
12	add $R_0, 1$
13	compare R_0, R_1
14	jumpeq 18
15	...
16	
17	
18	
19	
20	
21	
22	
23	

- Processing unit performs basic operations in constant time
- Any memory cell with an address can be accessed in equal (constant) time
- The instructions as well as the data is kept in the memory
- There may be other, specialized registers
- Modern processing units also employ a 'cache'

Formal analysis of running time

- Simply count the number of *primitive operations*
- Primitive operations include:
 - Assignment
 - Arithmetic operations
 - Comparing primitive data types (e.g., numbers)
 - Accessing a single memory location
 - Function calls, return from functions
- **Not** primitive operations:
 - loops, recursion
 - comparing sequences

Focus on the worst case

- Algorithms are generally faster on certain input than others
- In most cases, we are interested in the *worst case* analysis
 - Guaranteeing worst case is important
 - It is also relatively easier: we need to identify the worst-case input
- Average case analysis is also useful, but
 - requires defining a distribution over possible inputs
 - often more challenging

Counting primitive operations

example: nearest points, the naive algorithm

```
def shortest_distance(points):
    n = len(points)                # 2 (constant?)
    min = 0                        # 1 (constant)
    for i in range(n):            # n times
        for j in range(i):        # i times
            d = distance(points[i], points[j]) # 2? (constant)
            if min > d:            # 1 (constant)
                min = d           # 1 (constant)
    return min                     # 1 (constant)
```

$$\begin{aligned}
 T(n) &= 3 + (1 + 2 + 3 + \dots + n - 1) \times 4 + 1 \\
 &= 4 \times \frac{(n-1)n}{2} + 4
 \end{aligned}$$

Big-O notation

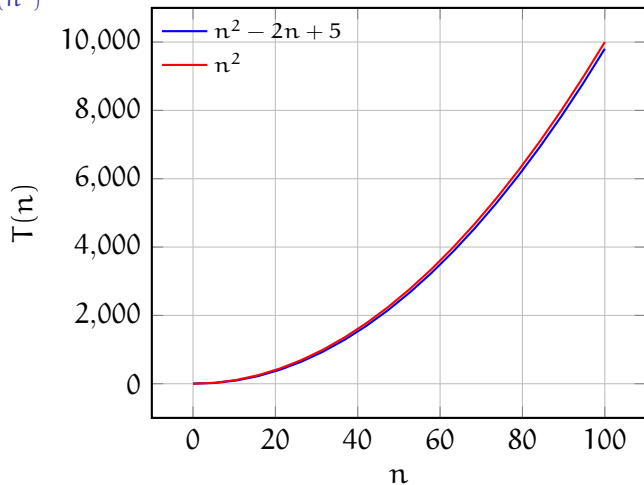
- Big-O notation is used for indicating an upper bound on running time of an algorithm as a function of running time
- If running time of an algorithm is $O(f(n))$, its running time grows proportional to $f(n)$ as the input size n grows
- More formally, given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and integer $n_0 \geq 1$ such that

$$f(n) \leq c \times g(n) \text{ for } n \geq n_0$$

- Sometimes the notation $f(n) = O(g(n))$ is also used, but beware: this equal sign is not symmetric

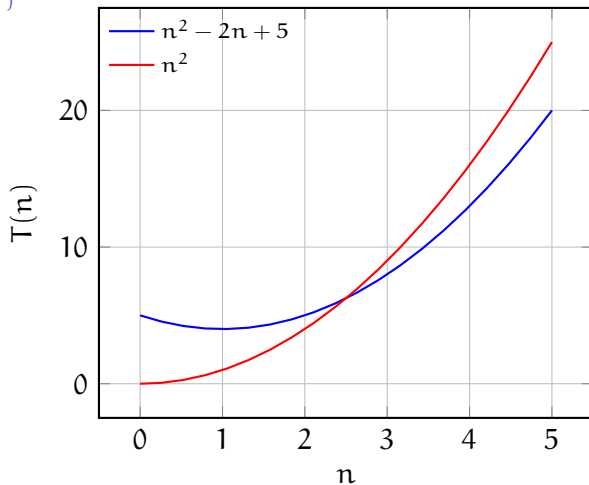
Big-O example

$T(n) = n^2 - 2n + 5$ is $O(n^2)$



Big-O example

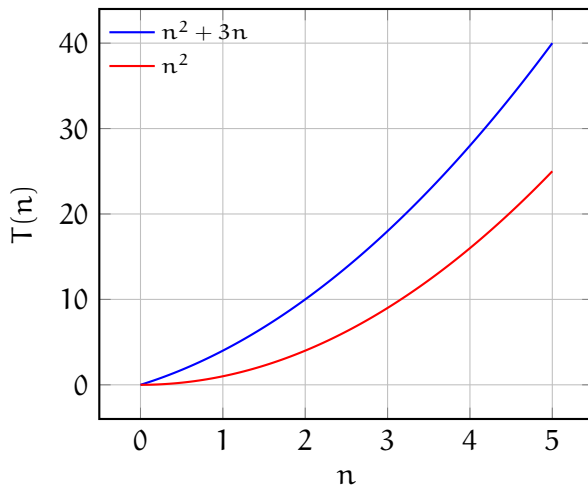
$T(n) = n^2 - 2n + 5$ is $O(n^2)$



Not surprising: $T(n) < n^2$ for $n \geq 3$

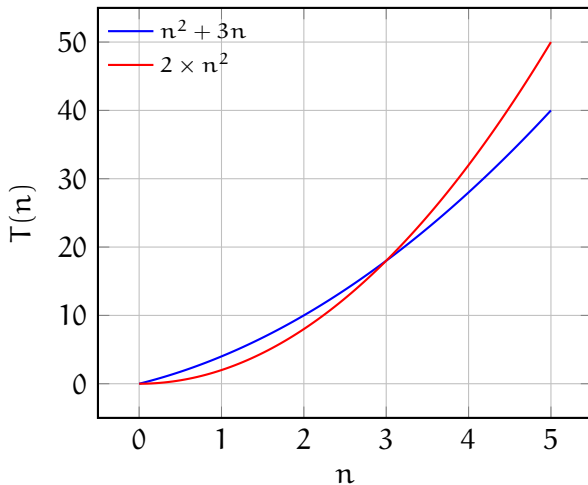
Big-O, another example

$T(n) = n^2 + 3n$ is $O(n^2)$



Big-O, another example

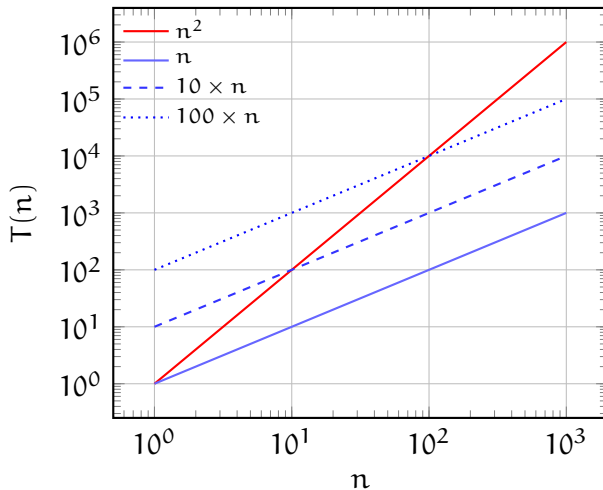
$T(n) = n^2 + 3n$ is $O(n^2)$



$$T(n) < 2 \times n^2 \text{ for } n \geq 4$$

Big-O, yet another example

but n^2 is not $O(n)$ – proof by picture



Back to the function classes

Family	Definition
Constant	$f(n) = c$
Logarithmic	$f(n) = \log_b n$
Linear	$f(n) = n$
$N \log N$	$f(n) = n \log n$
Quadratic	$f(n) = n^2$
Cubic	$f(n) = n^3$
Other polynomials	$f(n) = n^k$, for $k > 3$
Exponential	$f(n) = b^n$, for $b > 1$
Factorial	$f(n) = n!$

- None of these functions can be expressed as a constant factor of another

Rules of thumb

Drop the lower order terms

- In the big-O notation, we drop the constants and lower order terms
 - Any polynomial degree d is $O(n^d)$
 $10n^3 + 4n^2 + n + 100$ is $O(n^3)$
 - Drop any lower order terms:
 $2^n + 10n^3$ is $O(2^n)$
- Use the simplest expression:
 - $5n + 100$ is $O(5n)$, but we prefer $O(n)$
 - $4n^2 + n + 100$ is $O(n^3)$, but we prefer $O(n^2)$
- Transitivity: if $f(n) = O(g(n))$, and $g(n) = O(h(n))$, then $f(n) = O(h(n))$
- Additivity: if both $f(n)$ and $g(n)$ are $O(h(n))$ $f(n) + g(n)$ is $O(h(n))$

Rules of thumb

examples

$$\frac{f(n) \quad O(f(n))}{7n - 2}$$

Rules of thumb

examples

$f(n)$	$O(f(n))$
$7n - 2$	n
$3n^3 - 2n^2 + 5$	

Rules of thumb

examples

$f(n)$	$O(f(n))$
$7n - 2$	n
$3n^3 - 2n^2 + 5$	n^3
$3 \log n + 5$	

Rules of thumb

examples

$f(n)$	$O(f(n))$
$7n - 2$	n
$3n^3 - 2n^2 + 5$	n^3
$3 \log n + 5$	$\log n$
$\log n + 2^n$	

Rules of thumb

examples

$f(n)$	$O(f(n))$
$7n - 2$	n
$3n^3 - 2n^2 + 5$	n^3
$3 \log n + 5$	$\log n$
$\log n + 2^n$	2^n
$10n^5 + 2^n$	

Rules of thumb

examples

$f(n)$	$O(f(n))$
$7n - 2$	n
$3n^3 - 2n^2 + 5$	n^3
$3 \log n + 5$	$\log n$
$\log n + 2^n$	2^n
$10n^5 + 2^n$	2^n
$\log 2^n$	

Rules of thumb

examples

$f(n)$	$O(f(n))$
$7n - 2$	n
$3n^3 - 2n^2 + 5$	n^3
$3 \log n + 5$	$\log n$
$\log n + 2^n$	2^n
$10n^5 + 2^n$	2^n
$\log 2^n$	n
$2^n + 4^n$	

Rules of thumb

examples

$f(n)$	$O(f(n))$
$7n - 2$	n
$3n^3 - 2n^2 + 5$	n^3
$3 \log n + 5$	$\log n$
$\log n + 2^n$	2^n
$10n^5 + 2^n$	2^n
$\log 2^n$	n
$2^n + 4^n$	4^n
100×2^n	

Rules of thumb

examples

$f(n)$	$O(f(n))$
$7n - 2$	n
$3n^3 - 2n^2 + 5$	n^3
$3 \log n + 5$	$\log n$
$\log n + 2^n$	2^n
$10n^5 + 2^n$	2^n
$\log 2^n$	n
$2^n + 4^n$	4^n
100×2^n	2^n
$n2^n$	

Rules of thumb

examples

$f(n)$	$O(f(n))$
$7n - 2$	n
$3n^3 - 2n^2 + 5$	n^3
$3 \log n + 5$	$\log n$
$\log n + 2^n$	2^n
$10n^5 + 2^n$	2^n
$\log 2^n$	n
$2^n + 4^n$	4^n
100×2^n	2^n
$n2^n$	$n2^n$
$\log n!$	

Rules of thumb

examples

$f(n)$	$O(f(n))$
$7n - 2$	n
$3n^3 - 2n^2 + 5$	n^3
$3 \log n + 5$	$\log n$
$\log n + 2^n$	2^n
$10n^5 + 2^n$	2^n
$\log 2^n$	n
$2^n + 4^n$	4^n
100×2^n	2^n
$n2^n$	$n2^n$
$\log n!$	$n \log n$

Big-O: back to nearest points

```
def shortest_distance(points):
    n = len(points)                # 2 (constant?)
    min = 0                        # 1 (constant)
    for i in range(n):            # n times
        for j in range(i):        # i times
            d = distance(points[i], points[j]) # 2? (constant)
            if min > d:            # 1 (constant)
                min = d           # 1 (constant)
    return min                     # 1 (constant)
```

$$\begin{aligned}
 T(n) &= 3 + (1 + 2 + 3 + \dots + n - 1) \times 4 + 1 \\
 &= 4 \times \frac{(n-1)n}{2} + 4 = 2n^2 - 2n + 4
 \end{aligned}$$

Big-O: back to nearest points

```
def shortest_distance(points):
    n = len(points)                # 2 (constant?)
    min = 0                        # 1 (constant)
    for i in range(n):            # n times
        for j in range(i):        # i times
            d = distance(points[i], points[j]) # 2? (constant)
            if min > d:            # 1 (constant)
                min = d           # 1 (constant)
    return min                     # 1 (constant)
```

$$\begin{aligned}
 T(n) &= 3 + (1 + 2 + 3 + \dots + n - 1) \times 4 + 1 \\
 &= 4 \times \frac{(n-1)n}{2} + 4 = 2n^2 - 2n + 4 \\
 &= O(n^2)
 \end{aligned}$$

Big-O examples

linear search

- What is the worst-case running time?

```
1 def linear_search(seq, val):
2     i, n = 0, len(seq)
3     while i < n:
4         if seq[i] == val:
5             return i
6         i += 1
7     return None
```

Big-O examples

linear search

```
1 def linear_search(seq, val):
2     i, n = 0, len(seq)
3     while i < n:
4         if seq[i] == val:
5             return i
6         i += 1
7     return None
```

- What is the worst-case running time?
 2. 2 assignments
 3. $2n$ comparisons, n increment
 7. 1 return statement

Big-O examples

linear search

```
1 def linear_search(seq, val):
2     i, n = 0, len(seq)
3     while i < n:
4         if seq[i] == val:
5             return i
6         i += 1
7     return None
```

- What is the worst-case running time?

2. 2 assignments

3. $2n$ comparisons, n increment

7. 1 return statement

$$T(n) = 3n + 3 = O(n)$$

- What is the average-case running time?

Big-O examples

linear search

```

1 def linear_search(seq, val):
2     i, n = 0, len(seq)
3     while i < n:
4         if seq[i] == val:
5             return i
6         i += 1
7     return None

```

- What is the worst-case running time?

2. 2 assignments
3. $2n$ comparisons, n increment
7. 1 return statement

$$T(n) = 3n + 3 = O(n)$$

- What is the average-case running time?

2. 2 assignments
3. $2(n/2)$ comparisons, $n/2$ increment, 1 return

Big-O examples

linear search

```

1 def linear_search(seq, val):
2     i, n = 0, len(seq)
3     while i < n:
4         if seq[i] == val:
5             return i
6         i += 1
7     return None

```

- What is the worst-case running time?

2. 2 assignments
3. $2n$ comparisons, n increment
7. 1 return statement

$$T(n) = 3n + 3 = O(n)$$

- What is the average-case running time?

2. 2 assignments
3. $2(n/2)$ comparisons, $n/2$ increment, 1 return

$$T(n) = 3/2n + 3 = O(n)$$

- What about best case?

Big-O examples

linear search

```

1 def linear_search(seq, val):
2     i, n = 0, len(seq)
3     while i < n:
4         if seq[i] == val:
5             return i
6         i += 1
7     return None

```

- What is the worst-case running time?

2. 2 assignments
3. $2n$ comparisons, n increment
7. 1 return statement

$$T(n) = 3n + 3 = O(n)$$

- What is the average-case running time?

2. 2 assignments
3. $2(n/2)$ comparisons, $n/2$ increment, 1 return

$$T(n) = 3/2n + 3 = O(n)$$

- What about best case? $O(1)$

Big-O examples

linear search

```

1 def linear_search(seq, val):
2     i, n = 0, len(seq)
3     while i < n:
4         if seq[i] == val:
5             return i
6         i += 1
7     return None

```

- What is the worst-case running time?

2. 2 assignments
3. $2n$ comparisons, n increment
7. 1 return statement

$$T(n) = 3n + 3 = O(n)$$

- What is the average-case running time?

2. 2 assignments
3. $2(n/2)$ comparisons, $n/2$ increment, 1 return

$$T(n) = 3/2n + 3 = O(n)$$

- What about best case? $O(1)$

Note: do not confuse the big-O with the worst case analysis.

Recursive example

Recursive binary search

```
1 def rbs(a, x, L=0, R=n):
2     if L >= R:
3         return None
4     M = (L + R) // 2
5     if a[M] == x:
6         return M
7     if a[M] > x:
8         return rbs(a, x, L,
9                     ↪ M - 1)
9     else:
10        return rbs(a, x, M +
11                  ↪ 1, R)
```

- Counting is not easy, but realize that
 $T(n) = c + T(n/2)$

Recursive example

Recursive binary search

```

1 def rbs(a, x, L=0, R=n):
2     if L >= R:
3         return None
4     M = (L + R) // 2
5     if a[M] == x:
6         return M
7     if a[M] > x:
8         return rbs(a, x, L,
9                     ↪ M - 1)
10    else:
11        return rbs(a, x, M +
12                  ↪ 1, R)

```

- Counting is not easy, but realize that $T(n) = c + T(n/2)$
- This is a recursive formula, it means $T(n/2) = c + T(n/4)$,
 $T(n/4) = c + T(n/8), \dots$

Recursive example

Recursive binary search

```

1 def rbs(a, x, L=0, R=n):
2     if L >= R:
3         return None
4     M = (L + R) // 2
5     if a[M] == x:
6         return M
7     if a[M] > x:
8         return rbs(a, x, L,
9                     ↪ M - 1)
10    else:
11        return rbs(a, x, M +
12                  ↪ 1, R)

```

- Counting is not easy, but realize that $T(n) = c + T(n/2)$
- This is a recursive formula, it means $T(n/2) = c + T(n/4)$,
 $T(n/4) = c + T(n/8), \dots$
- So, $T(n) = 2c + T(n/4) = 3c + T(n/8)$

Recursive example

Recursive binary search

```

1 def rbs(a, x, L=0, R=n):
2     if L >= R:
3         return None
4     M = (L + R) // 2
5     if a[M] == x:
6         return M
7     if a[M] > x:
8         return rbs(a, x, L,
9                     ↪ M - 1)
10    else:
11        return rbs(a, x, M +
12                  ↪ 1, R)

```

- Counting is not easy, but realize that $T(n) = c + T(n/2)$
- This is a recursive formula, it means $T(n/2) = c + T(n/4)$,
 $T(n/4) = c + T(n/8), \dots$
- So, $T(n) = 2c + T(n/4) = 3c + T(n/8)$
- More generally, $T(n) = ic + T(n/2^i)$

Recursive example

Recursive binary search

```

1 def rbs(a, x, L=0, R=n):
2     if L >= R:
3         return None
4     M = (L + R) // 2
5     if a[M] == x:
6         return M
7     if a[M] > x:
8         return rbs(a, x, L,
9                     ↪ M - 1)
10    else:
11        return rbs(a, x, M +
12                  ↪ 1, R)

```

- Counting is not easy, but realize that $T(n) = c + T(n/2)$
- This is a recursive formula, it means $T(n/2) = c + T(n/4)$,
 $T(n/4) = c + T(n/8), \dots$
- So, $T(n) = 2c + T(n/4) = 3c + T(n/8)$
- More generally, $T(n) = ic + T(n/2^i)$
- Recursion terminates when $n/2^i = 1$, or $n = 2^i$,
the good news: $i = \log n$

Recursive example

Recursive binary search

```

1 def rbs(a, x, L=0, R=n):
2     if L >= R:
3         return None
4     M = (L + R) // 2
5     if a[M] == x:
6         return M
7     if a[M] > x:
8         return rbs(a, x, L,
9                     ↪ M - 1)
10    else:
11        return rbs(a, x, M +
12                    ↪ 1, R)

```

- Counting is not easy, but realize that $T(n) = c + T(n/2)$
- This is a recursive formula, it means $T(n/2) = c + T(n/4)$,
 $T(n/4) = c + T(n/8), \dots$
- So, $T(n) = 2c + T(n/4) = 3c + T(n/8)$
- More generally, $T(n) = ic + T(n/2^i)$
- Recursion terminates when $n/2^i = 1$, or $n = 2^i$,
the good news: $i = \log n$
- $T(n) = c \log n + T(1) = O(\log n)$

Recursive example

Recursive binary search

```

1 def rbs(a, x, L=0, R=n):
2     if L >= R:
3         return None
4     M = (L + R) // 2
5     if a[M] == x:
6         return M
7     if a[M] > x:
8         return rbs(a, x, L,
9                     ↪ M - 1)
10    else:
11        return rbs(a, x, M +
12                    ↪ 1, R)

```

- Counting is not easy, but realize that $T(n) = c + T(n/2)$
- This is a recursive formula, it means $T(n/2) = c + T(n/4)$,
 $T(n/4) = c + T(n/8), \dots$
- So, $T(n) = 2c + T(n/4) = 3c + T(n/8)$
- More generally, $T(n) = ic + T(n/2^i)$
- Recursion terminates when $n/2^i = 1$, or $n = 2^i$,
the good news: $i = \log n$
- $T(n) = c \log n + T(1) = O(\log n)$

Recursive example

Recursive binary search

```

1 def rbs(a, x, L=0, R=n):
2     if L >= R:
3         return None
4     M = (L + R) // 2
5     if a[M] == x:
6         return M
7     if a[M] > x:
8         return rbs(a, x, L,
9                     ↪ M - 1)
10    else:
11        return rbs(a, x, M +
12                  ↪ 1, R)

```

- Counting is not easy, but realize that $T(n) = c + T(n/2)$
- This is a recursive formula, it means $T(n/2) = c + T(n/4)$,
 $T(n/4) = c + T(n/8), \dots$
- So, $T(n) = 2c + T(n/4) = 3c + T(n/8)$
- More generally, $T(n) = ic + T(n/2^i)$
- Recursion terminates when $n/2^i = 1$, or $n = 2^i$,
the good news: $i = \log n$
- $T(n) = c \log n + T(1) = O(\log n)$

You do not always need to prove: for most recurrence relations, there is a way to obtain quick solutions (we are not going to cover it further, see [Appendix](#))

Why asymptotic analysis is important?

'maximum problem size'

- Assume we can solve a problem of size m in a given time on current hardware
- We get a better computer, which runs 1024 times faster
- New problem size we can solve in the same time

Complexity	new problem size
Linear (n)	$1024m$
Quadratic (n^2)	$32m$
Exponential (2^n)	$m + 10$

- This also demonstrates the gap between polynomial and exponential algorithms:
 - with a exponential algorithm fast hardware does not help
 - problem size for exponential algorithms does not scale with faster computers

Worst case and asymptotic analysis

pros and cons

- We typically compare algorithms based on their worst-case performance

Worst case and asymptotic analysis

pros and cons

- We typically compare algorithms based on their worst-case performance
pro it is easier, and we get a (very) strong guarantee: we know that the algorithm won't perform worse than the bound

Worst case and asymptotic analysis

pros and cons

- We typically compare algorithms based on their worst-case performance
 - pro it is easier, and we get a (very) strong guarantee: we know that the algorithm won't perform worse than the bound
 - con a (very) strong guarantee: in some (many?) problems, worst case examples are rare

Worst case and asymptotic analysis

pros and cons

- We typically compare algorithms based on their worst-case performance
 - pro it is easier, and we get a (very) strong guarantee: we know that the algorithm won't perform worse than the bound
 - con a (very) strong guarantee: in some (many?) problems, worst case examples are rare
 - In practice you may prefer an algorithm that does better on average (we'll see examples from sorting)

Worst case and asymptotic analysis

pros and cons

- We typically compare algorithms based on their worst-case performance
 - pro it is easier, and we get a (very) strong guarantee: we know that the algorithm won't perform worse than the bound
 - con a (very) strong guarantee: in some (many?) problems, worst case examples are rare
 - In practice you may prefer an algorithm that does better on average (we'll see examples from sorting)
- Our analyses are based on asymptotic behavior

Worst case and asymptotic analysis

pros and cons

- We typically compare algorithms based on their worst-case performance
 - pro it is easier, and we get a (very) strong guarantee: we know that the algorithm won't perform worse than the bound
 - con a (very) strong guarantee: in some (many?) problems, worst case examples are rare
 - In practice you may prefer an algorithm that does better on average (we'll see examples from sorting)
- Our analyses are based on asymptotic behavior
 - pro for a 'large enough' input, asymptotic analysis is correct

Worst case and asymptotic analysis

pros and cons

- We typically compare algorithms based on their worst-case performance
 - pro it is easier, and we get a (very) strong guarantee: we know that the algorithm won't perform worse than the bound
 - con a (very) strong guarantee: in some (many?) problems, worst case examples are rare
 - In practice you may prefer an algorithm that does better on average (we'll see examples from sorting)
- Our analyses are based on asymptotic behavior
 - pro for a 'large enough' input, asymptotic analysis is correct
 - con constant or lower order factors are not always unimportant

Worst case and asymptotic analysis

pros and cons

- We typically compare algorithms based on their worst-case performance
 - pro it is easier, and we get a (very) strong guarantee: we know that the algorithm won't perform worse than the bound
 - con a (very) strong guarantee: in some (many?) problems, worst case examples are rare
 - In practice you may prefer an algorithm that does better on average (we'll see examples from sorting)
- Our analyses are based on asymptotic behavior
 - pro for a 'large enough' input, asymptotic analysis is correct
 - con constant or lower order factors are not always unimportant
 - A constant factor of 100^{100} should probably not be ignored

Big-O relatives

- Big-O (upper bound): $f(n)$ is $O(g(n))$
if $f(n)$ is asymptotically *less than or equal to* $g(n)$

$$f(n) \leq cg(n) \text{ for } n > n_0$$

Big-O relatives

- Big-O (upper bound): $f(n)$ is $O(g(n))$
if $f(n)$ is asymptotically *less than or equal to* $g(n)$

$$f(n) \leq cg(n) \text{ for } n > n_0$$

- Big-Omega (lower bound): $f(n)$ is $\Omega(g(n))$
if $f(n)$ is asymptotically *greater than or equal to* $g(n)$

$$f(n) \geq cg(n) \text{ for } n > n_0$$

Big-O relatives

- Big-O (upper bound): $f(n)$ is $O(g(n))$
if $f(n)$ is asymptotically *less than or equal to* $g(n)$

$$f(n) \leq cg(n) \text{ for } n > n_0$$

- Big-Omega (lower bound): $f(n)$ is $\Omega(g(n))$
if $f(n)$ is asymptotically *greater than or equal to* $g(n)$

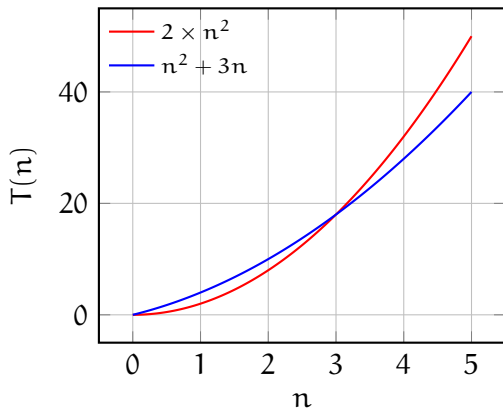
$$f(n) \geq cg(n) \text{ for } n > n_0$$

- Big-Theta (upper/lower bound): $f(n)$ is $\Theta(g(n))$
if $f(n)$ is asymptotically *equal to* $g(n)$

$$f(n) \text{ is } O(g(n)) \text{ and } f(n) \text{ is } \Omega(g(n))$$

Big-O, Big-Ω, Big-Θ: an example

$T(n) = n^2 + 3n$ is $O(n^2)$

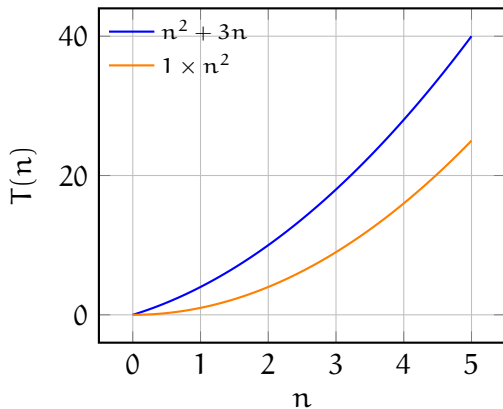


O for $c = 2$ and $n_0 = 3$

$$T(n) \leq c g(n) \text{ for } n > n_0$$

Big-O, Big-Ω, Big-Θ: an example

$$T(n) = n^2 + 3n \text{ is } \Omega(n^2)$$



O for $c = 2$ and $n_0 = 3$

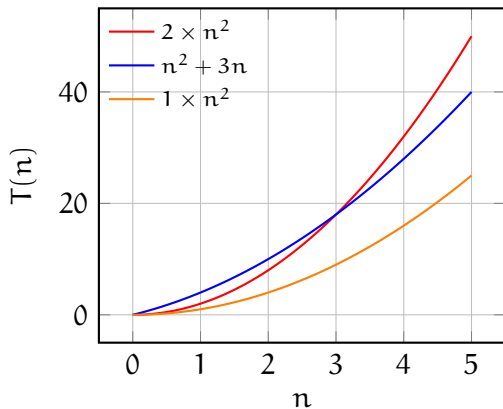
$$T(n) \leq c g(n) \text{ for } n > n_0$$

Ω for $c = 1$ and $n_0 = 0$

$$T(n) \geq c g(n) \text{ for } n > n_0$$

Big-O, Big-Ω, Big-Θ: an example

$T(n) = n^2 + 3n$ is $\Theta(n^2)$



\mathcal{O} for $c = 2$ and $n_0 = 3$

$$T(n) \leq c g(n) \text{ for } n > n_0$$

Ω for $c = 1$ and $n_0 = 0$

$$T(n) \geq c g(n) \text{ for } n > n_0$$

Θ for $c = 2, n_0 = 3, c' = 1$ and $n'_1 = 0$

$$T(n) \leq c g(n) \text{ for } n > n_0 \quad \text{and}$$

$$T(n) \geq c' g(n) \text{ for } n > n'_0$$

Summary

- Algorithmic analysis mainly focuses on worst-case asymptotic running times
- *Sublinear* (e.g., *logarithmic*), *Linear* and $n \log n$ algorithms are good
- *Polynomial* algorithms may be acceptable in many cases
- *Exponential* algorithms are bad
- We will return to concepts from this lecture while studying various algorithms
- Reading for this lecture: Goodrich, Tamassia, and Goldwasser (2013, chapter 3)

Summary

- Algorithmic analysis mainly focuses on worst-case asymptotic running times
- *Sublinear* (e.g., *logarithmic*), *Linear* and $n \log n$ algorithms are good
- *Polynomial* algorithms may be acceptable in many cases
- *Exponential* algorithms are bad
- We will return to concepts from this lecture while studying various algorithms
- Reading for this lecture: Goodrich, Tamassia, and Goldwasser (2013, chapter 3)

Next:

- Common patterns in algorithms
- Sorting algorithms
- Reading: Goodrich, Tamassia, and Goldwasser (2013, chapter 12) – up to 12.7

Acknowledgments, credits, references

- Some of the slides are based on the previous year's course by Corina Dima.



Goodrich, Michael T., Roberto Tamassia, and Michael H. Goldwasser (2013).
Data Structures and Algorithms in Python. John Wiley & Sons, Incorporated. ISBN:
9781118476734.

A(nother) view of computational complexity

P, NP, NP-complete and all that

- A major division of complexity classes according to Big-O notation is between
 - P polynomial time algorithms
 - NP non-deterministic polynomial time algorithms
- A big question in computing is whether $P = NP$
- All problems in NP can be reduced in polynomial time to a problem in a subclass of NP (*NP-complete*)
 - Solving an NP complete problem in P would mean proving

$$P = NP$$

Video from <https://www.youtube.com/watch?v=YX40hbAHx3s>

Exercise

Sort the functions based on asymptotic order of growth

$$\log n^{1000}$$

$$n \log(n)$$

$$5^n$$

$$\log n$$

$$\log n^{1/\log n}$$

$$\log n$$

$$\log 2^n/n$$

$$\log n!$$

$$\log 2^n$$

$$\log 5^n$$

$$\binom{n}{n/2}$$

$$\log \log n!$$

$$\sqrt{n}$$

$$n^2$$

$$2^n$$

$$\binom{n}{2}$$

Recurrence relations

the master theorem

- Given a recurrence relation:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

a number of sub-problems

b reduction factor or the input

f(n) amount of work for creating and combining sub-problems

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } f(n) \text{ is } O(n^{\log_b a - \epsilon}) \\ \Theta(n^{\log_b a} \log n) & \text{if } f(n) \text{ is } \Theta(n^{\log_b a}) \\ \Theta(f(n)) & \text{if } f(n) \text{ is } \Omega(n^{\log_b a + \epsilon}) \text{ and } af(n/b) \leq cf(n) \text{ for some } c < 1 \end{cases}$$

- In many practical cases $a = b$ (simplifies the expressions above)
- But the theorem is not general for all recurrences: it requires equal splits

