Advanced Programming

Search and Complexity

Artur Kulmizev

2021-02-05

(slides used with permission from Sara Stymne)

Module 2

- Data structures, algorithms, complexity
- Activities:
  - 3 lectures
  - 1 lab package (intructions in Studium)
  - 1 lab session (we can add another as needed)
- Reading list in Studium

Today

- Search
- Analysis of algorithms – complexity
- Hash tables (maybe)

What is search?

- To find a specific value in a collection of values
- We will focus on finding values in a list of integers
- Return values:
  - If the value exists: the index where we find the value
  - If the value does not exist: -1

What is search?

- To find a specific value in a collection of values
- We will focus on finding values in a list of integers
- Return values:
  - If the value exists: the index where we find the value
  - If the value does not exist: -1 (cannot be used as an index)

What is search?

- To find a specific value in a collection of values
- We will focus on finding values in a list of integers
- Return values:
  - If the value exists: the index where we find the value
  - If the value does not exist: -1 (cannot be used as an index)
- Other common formulation (e.g. in PS book)
  - If the value exists: True
  - If the value does not exist: False

How to search?

- To find a specific value in a collection of values
- We will focus on finding values in a list of integers
- Return values:
  - If the value exists: the index where we find the value
  - If the value does not exist: -1
- How we do this depends on the list:
  - Unsorted
  - Sorted

How to search?

- To find a specific value in a collection of values
- We will focus on finding values in a list of integers
- Return values:
  - If the value exists: the index where we find the value
  - If the value does not exist: -1
- How we do this depends on the list:
  - Unsorted – Linear search
  - Sorted – Binary search

Linear search

- Find a value in an unsorted list
- Solution: Loop through the list until we find the value, or until we have looked through the list without finding it

Linear search

- Find a value in an unsorted list
- Solution: Loop through the list until we find the value, or until we have looked through the list without finding it

```
def sequentialSearch(alist, item):
    pos = 0
    while pos < len(alist):
        if alist[pos] == item:
            return pos
        else:
            pos = pos+1

    return -1
```

How long time does linear search take?

- Count the time in the number of comparisons made
- Based on the size of the list: n
- How long time does the search take?
  - In the best case?
  - In the worst case?
  - On average?

How long time does linear search take?

- Count the time in the number of comparisons made
- Based on the size of the list: n
- How long time does the search take?
  - In the best case?
    - 1 – the value we're looking for is first
  - In the worst case?
  - On average?

How long time does linear search take?

- Count the time in the number of comparisons made
- Based on the size of the list: n
- How long time does the search take?
  - In the best case?
    - 1 – the value we're looking for is first
  - In the worst case?
    - n – The value does not exist, have to look at the full list
  - On average?

How long time does linear search take?

- Count the time in the number of comparisons made
- Based on the size of the list: n
- How long time does the search take?
  - In the best case?
    - 1 – the value we're looking for is first
  - In the worst case?
    - n – The value does not exist, have to look at the full list
  - On average?
    - $n/2$ – If we're only looking for exisiting values

How long time does linear search take?

- Count the time in the number of comparisons made
- Based on the size of the list: n
- How long time does the search take?
  - In the best case?
    - 1 – the value we're looking for is first
  - In the worst case?
    - n – The value does not exist, have to look at the full list
  - On average?
    - $n/2$ – If we're only looking for exisiting values
    - Also dependent on the distributuion of values we're searching for

Binary search

- If the list is sorted, we can take advantage of that for faster search

Binary search

- If the list is sorted, we can take advantage of that for faster search
- Look in the middle first
  - If correct, return this index
  - If the value is smaller, search in lower half of list
  - If the value is larger, search in upper half of list
- Repeat!

Binary search – iterative

```
def binarySearch(alist, item):
    first = 0
    last = len(alist)-1

    while first<=last:
        midpoint = (first + last)//2
        if alist[midpoint] == item:
            return midpoint
        else:
            if item < alist[midpoint]:
                last = midpoint-1
            else:
                first = midpoint+1

    return -1
```

Recursion

- Recursive function – a function that calls itself
- A recursive function has two types of cases:
  - Base case – cases where the solution is trivial
  - Recursive cases – cases where the method calls itself

Recursion

- Recursive function – a function that calls itself
- A recursive function has two types of cases:
  - Base case – cases where the solution is trivial
    - Binary search: -1 if the answer does not exist, the correct index if we find the value
  - Recursive cases – cases where the method calls itself
    - Binary search: search in one half by calling itself

Binary search – recursive

```
def binarySearch(alist, item):
    return binSearch(alist, item, 0, len(alist)-1)

def binSearch(alist, item, first, last):
    if last < first:
        return -1
    midpoint = (first + last)//2
    if alist[midpoint] == item:
        return midpoint
    if item < alist[midpoint]:
        return binSearch(alist, item, first, midpoint-1)
    else:
        return binSearch(alist, item, midpoint+1, last)
```

How long time does binary search take?

- Count the time in the number of comparisons made
- Based on the size of the list: n
- How long time does the search take?
  - In the best case:
  - In the worst case:
  - On average

How long time does binary search take?

- Count the time in the number of comparisons made
- Based on the size of the list: n
- How long time does the search take?
  - In the best case:
    - 1 – the value is in the middle of the array
  - In the worst case:
    - $\log n$ – the value does not exist
  - On average
    - $1 \leq t \leq \log n$

Search – comparison

	Linear	**Binary**
Can be used	always	for sorted list
For linked list	yes	not a good fit
Time (worst)	n	$\log n$
Implementation	extremely easy	somewhat harder

Search – alternatives

- Linear and binary search works well for lists
- There are other data structures that can facilitate search:
  - Binary search trees
  - Hash tables

Algorithms and data structures

- Algorithm
  - "a procedure for solving a mathematical problem . . . in a finite number of steps that frequently involves repetition of an operation; broadly : a step-by-step procedure for solving a problem or accomplishing some end especially by a computer" (Merriam-Webster)
  - A description of how to solve a problem
- Data structure
  - A structure used for storing and manipulating data, for instance, array, linked list, hash table

Algorithms and data structures

- Algorithm
  - ''a procedure for solving a mathematical problem ...in a finite number of steps that frequently involves repetition of an operation; broadly : a step-by-step procedure for solving a problem or accomplishing some end especially by a computer'' (Merriam-Webster)
  - A description of how to solve a problem
- Data structure
  - A structure used for storing and manipulating data, for instance, array, linked list, hash table
- Which algorithms that are applicable for a problem is partly dependent on which data structures that is used

Analysis of algorithms

- Important questions to ask with respect to a program or an algorithm
  - Does it always give a correct answer?
  - Does it always give an answer? (cannot get stuck in a loop, for instance)
  - How fast is it?
  - How much memory does it require?

Analysis of algorithms

- Important questions to ask with respect to a program or an algorithm
  - Does it always give a correct answer?
  - Does it always give an answer? (cannot get stuck in a loop, for instance)
  - **How fast is it? – time complexity**
  - How much memory does it require?

Time Complexity

- How can we decide which algorithm is the fastest one for solving a specific problem?

Time Complexity

- How can we decide which algorithm is the fastest one for solving a specific problem?
- Measuring actual time is often impractical – different machines have different speed, for instance

Time Complexity

- How can we decide which algorithm is the fastest one for solving a specific problem?
- Measuring actual time is often impractical – different machines have different speed, for instance
- Use **asymptotic analysis** – the tendency over time

Asymptotic analyses

- The analysis of the time for running an algorithm, when the size of the input grows
- Size?
  - List: number of elements
  - String: number of characters
  - General case: often number of bytes
  - NLP: often words per sentence/document
- We can discuss different cases
  - Worst
  - Average
  - Best

Asymptotic analyses

- The analysis of the time for running an algorithm, when the size of the input grows
- Size?
  - List: number of elements
  - String: number of characters
  - General case: often number of bytes
  - NLP: often words per sentence/document
- We can discuss different cases
  - Worst – common analysis
  - Average – sometimes done, but harder
  - Best – often not so meaningful

Asymptotic analyses

- The analysis of the time for running an algorithm, when the size of the input grows
- Size?
  - List: number of elements
  - String: number of characters
  - General case: often number of bytes
  - NLP: often words per sentence/document
- We can discuss different cases
  - Worst – common analysis
  - Average – sometimes done, but harder
  - Best – often not so meaningful
  - Amortized analysis – the analysis of a series of operations, often better specified than average analysis

Asymptotic analysis

- Basic requirements: An algorithm should work for input of an arbitrary size (n)
- Estimate the running time as a function of the size of the input $T(n)$
- Ignore constant factors
- Focus on dominant factors with large input

Asymptotic analysis

- Basic requirements: An algorithm should work for input of an arbitrary size (n)
- Estimate the running time as a function of the size of the input $T(n)$
- Ignore constant factors
- Focus on dominant factors with large input
- The analysis should not be dependent on the machine/computer
- Powerful computers raise the speed by a constant

Primitive operations

- A primitive operation is assumed to take constant time
  - Assignment, e.g. x = y;
  - Arithmetic operations, e.g. x+5;
  - Comparisons, e.g. x < 5;
  - Array indexation, e.g. myArray[5];
  - Return statements, e.g. return 5;
  - ...
- Let $T(n)$ be the number of primitive operations as a function of "the size of the input"

Example – multiply the numbers in a list

```
def multiply(numbers):
  res = 1
  index = 0
  while index < len(numbers):
     res = res * numbers[index]
     index = index + 1

  return res
```

Example – multiply the numbers in a list

```
def multiply(numbers):
  res = 1                          1
  index = 0                        1
  while index < len(numbers):      1+1 (*n)
     res = res * numbers[index]    1+1+1 (*n)
     index = index + 1             1+1 (*n)

  return res                       1
```

Example – multiply the numbers in a list

```
def multiply(numbers):
  res = 1                            1
  index = 0                          1
  while index < len(numbers):        1+1 (*n)
     res = res * numbers[index]      1+1+1 (*n)
     index = index + 1               1+1 (*n)

  return res                         1
```

$T(n) = 7 * n + 3$

Big O

- O – Big O (or Ordo) is an upper bound for how the time for an algorithm grows
- Definition:
  $T(n)$ is a non-negative function
  $T(n) \in O(f(n))$ (i.e. $T(n)$ belongs to the set $O(f(n))$)
  if there are positive cosntants c and n_0 such that
  $T(n) \leq c * f(n)$ for $n \geq n_0$

Time Complexity – cases

- Time Complexity is for a given case:
  - Worst
  - Best
  - Average
- For multiplication in a list, it does not matter, but in other cases it often does
- Important to be clear about what you mean!

Time Complexity linear search

- Worst case: $O(n)$
- Best case: $O(1)$ – 1 is used for anything that takes constant time

Time Complexity linear search

- Worst case: $O(n)$
- Best case: $O(1)$ – 1 is used for anything that takes constant time
- Average case: $O(n)$
  - Assume that average case takes the time $T(n/2)$
  - $n/2 = n * 1/2$

    $1/2$ is a constant that we can ignore

Time Complexity binary search

- Worst case: $O(\log n)$
- Best case: $O(1)$
- Average case: $O(\log n)$

Time Complexity – estimation

- When you give the time complexity, you ignore constants and temrs of lower order
- Examples:
  - $T(5 + n) \in O(n)$
  - $T(5 + 10n) \in O(n)$
  - $T(500000 + 100000n) \in O(n)$

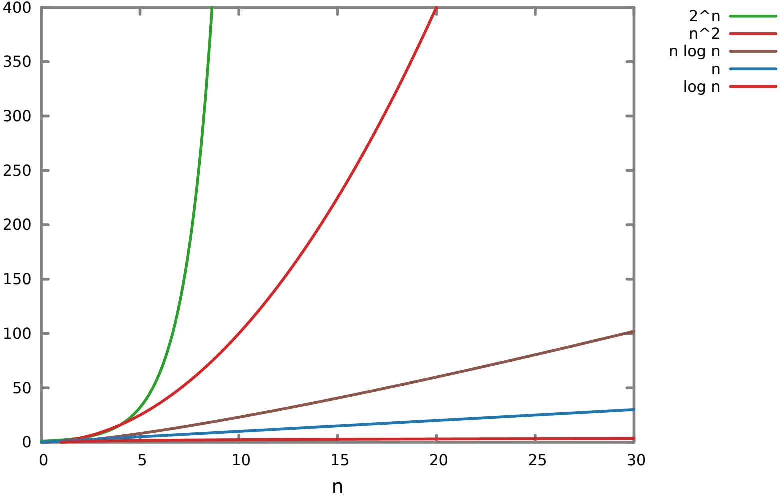
Time Complexity – estimation

- When you give the time complexity, you ignore constants and temrs of lower order
- Examples:
  - $T(5+n) \in O(n)$
  - $T(5+10n) \in O(n)$
  - $T(500000+100000n) \in O(n)$
  - $T(n^2+n) \in O(n^2)$
  - $T(15n^2+5/8*n) \in O(n^2)$
  - $T(n+\text{log } n) \in O(n)$
  - $T(5n \text{ log } n) \in O(n \text{ log } n)$
  - $T(25n \text{ log } n+800) \in O(n \text{ log } n)$

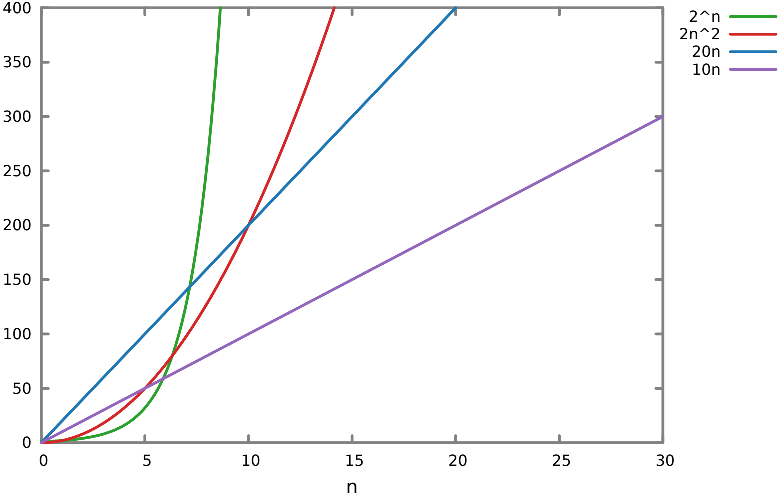
Ordo classes

Ordo	Class
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(n^x)$	Polynomial (for $x > 1$)
$O(2^n)$	Exponential
$O(n!)$	Factorial

Asymptotic size matters



Asymptotic size matters despite constants



Omega and Theta

- O – upper bound
- There are other variants:
  - Ω – lower bound
  - Θ –
    $T(n) \in \Theta(n)$ iff $T \in O(n)$ and $T \in \Omega(n)$

Asymptotic analysis – discussion

- Asymptotic analysis is a good tool for discussing algorithms
- For large input, an algorithm with lower complexity is always better
- What "large" is might vary, though
- For very small input, an algorithm with higher complexity might do better
  - Example: for very small lists, linear search can be faster than binary search, since we do not have to calculate averages.
  - However, for so small lists, time often does not really matter

Asymptotic analysis – discussion

- Asymptotic analysis is a good tool for discussing algorithms
- For large input, an algorithm with lower complexity is always better
- What "large" is might vary, though
- For very small input, an algorithm with higher complexity might do better
  - Example: for very small lists, linear search can be faster than binary search, since we do not have to calculate averages.
  - However, for so small lists, time often does not really matter
- Memory complexity can be discussed in the same way as time complexity

Analysis of algorithms – What do you need to know?

- Basic discussions of time complexity for the upper bound of a program/algorithm (Ordo)
- Especially for:
  - The search and sorting algorithms we discuss in the course
  - Common operations for the data structures we discuss in the course
- Based on simple code, be able to reason about the time complexity (like for the multiplication example)

Maps

- Arrays / lists
    - Mapping från integers (0–n) to values

Maps

- Arrays / lists
  - Mapping från integers (0–n) to values
- Hash tables
  - Mapping from one type to another
  - Python dictionaries is a hash table
  - Example: freqList = {'and': 375, 'run': 27, 'missing': 2}

Hash Tables – terminology

- Mapping from a **key** to a **value**
- Examples:
  - Frequency word list
    - Key: word (string)
    - Value: freqeency (integer)
  - Map from word form to lemma
    - Key: word form (string)
    - Value: lemma (string)
- Python allows dictionaries with mixed types:
  dict = {'Name': 'Max', 'Age': 37}

Hash Tables – types

- Dictionary keys must be **immutable**
  - Mutable: values can be altered, e.g. list, dict, user-defined classes (unless explicitly made mutable)
  - Immutable: values cannot be altered: e.g. int, float, string, tuple
- Dictionary keys:
  - must be immutable!
  - must be unique!
- Dictionary values
  - Any type of object
  - Does not need to be unique

Lab package 2

- Searching
  - Implement and try different methods for search (in lists, deque, hash table)
  - Time them, and compare to theoretical complexity
- Linked lists
  - Implement a linked list for a sorted word frequency list
- Sorting
  - Implement two sorting algorithms using a given API
  - Compare the theoretical complexity with the number of operations used in the different algorithms

Coming up

- Lab session this afternoon
- Lecture tomorrow: data structures
- Next week:
  - Lecture: sorting
  - Lab session
- Own work on lab
- Soft deadline lab package 2: February 19