Data Structures and Algorithms

Live Class 5

Heikki Peura

h.peura@imperial.ac.uk

Announcements

Homework 1 deadline tomorrow:

- Your functions should almost always `return` rather than `print`.
- Make sure not to `print` anything unless explicitly requested.

I will post a **mock exam** next week

Today

1. Recap
2. Sorting algorithms:
   - Selection sort
   - Merge sort

Go to menti.com

What is the output?

```python
a = 'hiphop'
b = a[2]
c = a[:len(a) - 1]
print(b, c)
```

A. i hipho

B. p hipho

C. i hiph

D. p hiph

E. I don't know

What is the output?

```python
a = 'hip'
for i in range(1, len(a)):
    print(a[i] + a[i - 1])
```

A. An error

B. hp, ih, pi

C. ih, pi

D. hi, ip

E. I don’t know

This algorithm is...?

```python
def fun_function(s):
    # s is a list of length n
    new_list = []
    for v1 in s:
        for v2 in s:
            new_list.append(v1 + v2)
    return new_list
```

A. Constant time - $O(1)$

B. Linear time - $O(n)$

C. Quadratic time - $O(n^2)$

D. None of the above

E. I don’t know

Which of these reverses the string?

A:

```
s = 'hiphop'
new_s = s[0:len(s):1]
```

B:

```
s = 'hiphop'
new_s = s[::-1]
```

C:

```
s = 'hiphop'
new_s = ''
for char in s:
    new_s = new_s + char
```

D:

```
s = 'hiphop'
new_s = ''
for i in range(len(s)):
    new_s += s[len(s) - i - 1]
```

E: More than one of the above

Asymptotic analysis

1: measure number of basic operations as function of input size

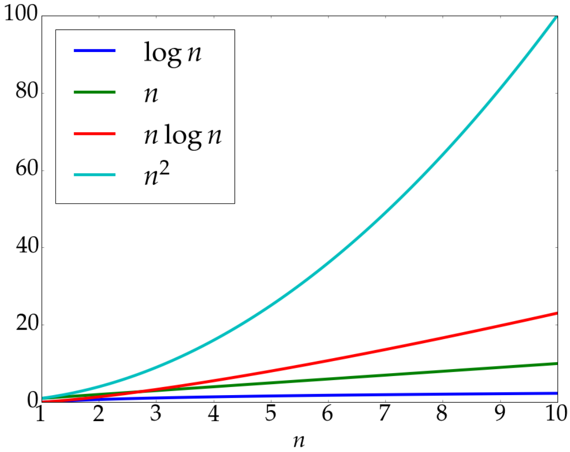
2: focus on worst-case analysis

3: ignore constant factors and lower-order terms

4: only care about large inputs

Formal way to describe this approach:

- Big-O notation: upper bound on worst-case running time



Big O: for **large enough inputs**, an $O(n)$ algorithm will be slower than $O(\log(n))$

Basic operations

Operations that a **computer can perform "quickly"** (constant time $O(1)$ for any input)

- Arithmetic operations
- Comparisons
- Variable assignment
- Memory access

Basic operations

Operations that a **computer can perform "quickly"** (constant time $O(1)$ for any input)

- Arithmetic operations
- Comparisons
- Variable assignment
- Memory access

What if the data structure is more complicated?

- For example, if `L` is a `list`: `L.append(), L[5]` ?
- Are these basic constant-time operations?
- Wait for it... assume for now that list operations are constant time

Complexity classes

Fast algorithm: worst-case running time grows slowly with input size

- $O(1)$: constant running time — basic operations
- $O(\log n)$: logarithmic running time — binary search
- $O(n)$: linear running time — linear search
- $O(n \log n)$: log-linear running time — ??
- $O(n^c)$: polynomial running time — ??
- $O(c^n)$: exponential running time — ??

Sorting algorithms

So if we have an unsorted list, should we sort it first?

- Suppose complexity $O(sort(n))$
- Is it less work to sort and then do binary search than to do linear search?
- Is $sort(n) + \log(n) < n$?
- No...

But what if we need to search repeatedly, say k times?

- Is $sort(n) + k \log(n) < kn$?
- Depends on k...

How would you intuitively sort a list?

56	24	99	32	9	61	57	79

How would you intuitively sort a list?

56	24	99	32	9	61	57	79

9		24	99	32	56	61	57	79

How would you intuitively sort a list?

56	24	99	32	9	61	57	79

9		24	99	32	56	61	57	79
9	24		99	32	56	61	57	79

How would you intuitively sort a list?

56	24	99	32	9	61	57	79

9		24	99	32	56	61	57	79

9	24		99	32	56	61	57	79

9	24	32		99	56	61	57	79

How would you intuitively sort a list?

56	24	99	32	9	61	57	79

9	24	99	32	56	61	57	79
9	24	99	32	56	61	57	79
9	24	32	99	56	61	57	79
9	24	32	56	99	61	57	79

How would you intuitively sort a list?

56	24	99	32	9	61	57	79
9	24	99	32	56	61	57	79
9	24	99	32	56	61	57	79
9	24	32	99	56	61	57	79
9	24	32	56	99	61	57	79
9	24	32	56	57	61	99	79

How would you intuitively sort a list?

56	24	99	32	9	61	57	79
9	24	99	32	56	61	57	79
9	24	99	32	56	61	57	79
9	24	32	99	56	61	57	79
9	24	32	56	99	61	57	79
9	24	32	56	57	61	99	79
9	24	32	56	57	61	99	79

How would you intuitively sort a list?

56	24	99	32	9	61	57	79
9	24	99	32	56	61	57	79
9	24	99	32	56	61	57	79
9	24	32	99	56	61	57	79
9	24	32	56	99	61	57	79
9	24	32	56	57	61	99	79
9	24	32	56	57	61	99	79
9	24	32	56	57	61	79	99

How would you intuitively sort a list?

56	24	99	32	9	61	57	79	n-1
9	24	99	32	56	61	57	79	n-2
9	24	99	32	56	61	57	79	n-3
9	24	32	99	56	61	57	79	...
9	24	32	56	99	61	57	79	
9	24	32	56	57	61	99	79	2
9	24	32	56	57	61	99	79	1
9	24	32	56	57	61	79	99	

In words: Find smallest item and move to front (swap with first unsorted item). Repeat with remaining unsorted items.

n-1+n-2+...+2+1 = n(n-1)/2 comparisons -> O(n^2)

Selection sort algorithm

Selection sort list L of length n:

Selection sort algorithm

Selection sort list L of length n:

- Repeat n times:
  - Find smallest unsorted element
  - Swap its position with the first unsorted element

Selection sort algorithm

Selection sort list *L* of length *n*:

- Repeat *n* times:
  - Find smallest unsorted element
  - Swap its position with the first unsorted element

Python:

```
def selection_sort(L):
    M = L[:] # make a copy to preserve original list
    n = len(M)
    for index in range(n):
        min_index = find_min_index(M, index) # index with smallest element
        M[index], M[min_index] = M[min_index], M[index] # swap positions
    return M
```

Selection sort complexity

Correctness (for those into math): can be proved by induction

Selection sort complexity

Correctness (for those into math): can be proved by induction

Complexity:

- $O(n)$ passes of main loop
- Each pass: search for the smallest element in $O(n)$
- Total $O(n^2)$

Selection sort complexity

Correctness (for those into math): can be proved by induction

Complexity:

- $O(n)$ passes of main loop
- Each pass: search for the smallest element in $O(n)$
- Total $O(n^2)$

Can we do better?

Selection sort complexity

Correctness (for those into math): can be proved by induction

Complexity:

- $O(n)$ passes of main loop
- Each pass: search for the smallest element in $O(n)$
- Total $O(n^2)$

Can we do better?

- Yes! **Merge sort** is $O(n \log n)$
- But you can't do any better than that...

Sidebar: recursion

The factorial of n is the product of integers $1, ..., n$.

- As a function: $\text{fact}(n) = n \times (n - 1) \times (n - 2) \times \cdots \times 2 \times 1$
- By convention, $\text{fact}(0) = 1$

Sidebar: recursion

The factorial of n is the product of integers $1, ..., n$.

- As a function: $\text{fact}(n) = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$
- By convention, $\text{fact}(0) = 1$

```python
def fact(n):
    result = 1
    for i in range(1, n+1):
        result = result * i
    return result
print(fact(4))
```

Sidebar: recursion

The factorial of n is the product of integers $1, \ldots, n$.

- As a function: $\text{fact}(n) = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$
- By convention, $\text{fact}(0) = 1$

```
def fact(n):
    result = 1
    for i in range(1, n+1):
        result = result * i
    return result
print(fact(4))
```

But we can also write the factorial as follows:

$$\text{fact}(n) = 1, \text{ for } n = 0$$

$$\text{fact}(n) = n \times \text{fact}(n-1), \text{ for } n > 0$$

Sidebar: recursion

We can also write the factorial as follows:

$$\text{fact}(n) = 1, \text{ for } n = 0$$
$$\text{fact}(n) = n \times \text{fact}(n-1), \text{ for } n > 0$$

Factorial can be expressed as a smaller version of itself:

Sidebar: recursion

We can also write the factorial as follows:

$$\text{fact}(n) = 1, \text{ for } n = 0$$

$$\text{fact}(n) = n \times \text{fact}(n-1), \text{ for } n > 0$$

Factorial can be expressed as a smaller version of itself:

```python
def fact_rec(n):
    if n == 0:
        return 1
    else:
        return n*fact_rec(n-1)
print(fact_rec(4))
```

This is called **recursion**

- Function calls itself
- Can make some problems easier to define $\rightarrow$ merge sort!

Merge sort idea

Divide and conquer:

- Identify smallest possible "base case" subproblems that are easy to solve
- Divide large problem and solve smaller subproblems
- Find a way to combine subproblem solutions to solve larger problems

Merge sort idea

Divide and conquer:

- Identify smallest possible “base case” subproblems that are easy to solve
- Divide large problem and solve smaller subproblems
- Find a way to combine subproblem solutions to solve larger problems

Merge sort:

Merge sort idea

Divide and conquer:

- Identify smallest possible “base case” subproblems that are easy to solve
- Divide large problem and solve smaller subproblems
- Find a way to combine subproblem solutions to solve larger problems

Merge sort:

- Base case: if list length $n < 2$, the list is sorted

Merge sort idea

Divide and conquer:

- Identify smallest possible "base case" subproblems that are easy to solve
- Divide large problem and solve smaller subproblems
- Find a way to combine subproblem solutions to solve larger problems

Merge sort:

- Base case: if list length $n < 2$, the list is sorted
- Divide: if list length $n \geq 2$, split into two lists and merge sort each

Merge sort idea

Divide and conquer:

- Identify smallest possible "base case" subproblems that are easy to solve
- Divide large problem and solve smaller subproblems
- Find a way to combine subproblem solutions to solve larger problems

Merge sort:

- Base case: if list length $n < 2$, the list is sorted
- Divide: if list length $n \geq 2$, split into two lists and merge sort each
- Combine (merge) the results of the two smaller merge sorts

How to merge two sorted lists into one?

Loop through both lists simultaneously, copy smaller item to new list z

- Compare items at indices i1 = i2 = 0, update with every copy operation

x =	24	32	56

y =	19	57	61

z =	

How to merge two sorted lists into one?

Loop through both lists simultaneously, copy smaller item to new list z

- Compare items at indices i1 = i2 = 0, update with every copy operation

x =	24	32	56	i1 = 0
y =	19	57	61	i2 = 0
z =				

How to merge two sorted lists into one?

Loop through both lists simultaneously, copy smaller item to new list z

- Compare items at indices i1 = i2 = 0, update with every copy operation

x =

24	32	56

i1 = 0

y =

19	57	61

i2 = 1

z =

19

How to merge two sorted lists into one?

Loop through both lists simultaneously, copy smaller item to new list z

- Compare items at indices i1 = i2 = 0, update with every copy operation

x =	24	32	56	i1 = 1
y =	19	57	61	i2 = 1
z =	19	24		

How to merge two sorted lists into one?

Loop through both lists simultaneously, copy smaller item to new list z

- Compare items at indices i1 = i2 = 0, update with every copy operation

x =	24	32	56	i1 = 2
y =	19	57	61	i2 = 1
z =	19	24	32	

How to merge two sorted lists into one?

Loop through both lists simultaneously, copy smaller item to new list z

- Compare items at indices i1 = i2 = 0, update with every copy operation

x = | 24 | 32 | 56 | i1 = 3

y = | 19 | 57 | 61 | i2 = 1

z = | 19 | 24 | 32 | 56 |

How to merge two sorted lists into one?

Loop through both lists simultaneously, copy smaller item to new list z

- Compare items at indices i1 = i2 = 0, update with every copy operation

x =	24	32	56	i1 = 3
y =	19	57	61	i2 = 2

z =	19	24	32	56	57

How to merge two sorted lists into one?

Loop through both lists simultaneously, copy smaller item to new list z

- Compare items at indices i1 = i2 = 0, update with every copy operation

x = | 24 | 32 | 56 | i1 = 3

y = | 19 | 57 | 61 | i2 = 3

z = | 19 | 24 | 32 | 56 | 57 | 61 |

How to merge two sorted lists into one?

Loop through both lists simultaneously, copy smaller item to new list z

- Compare items at indices i1 = i2 = 0, update with every copy operation

x =	24	32	56	i1 = 3
y =	19	57	61	i2 = 3

z =	19	24	32	56	57	61

What is the complexity of this operation?

- Lengths of lists are n_1 and n_2
- Two lists of lengths n_1 and n_2: $O(n_1 + n_2)$ copy operations (need to copy each item)
- No more comparisons than copy operations

Merge sort

Dividing

56	24	99	32	9	61	57	79

Merging

Merge sort

Dividing

56	24	99	32	9	61	57	79

m

56	24	99	32

9	61	57	79

Merging

Merge sort

Dividing

56	24	99	32	9	61	57	79

56	24	99	32

9	61	57	79

56	24

99	32

Merging

Merge sort

Dividing

56	24	99	32	9	61	57	79

56	24	99	32

9	61	57	79

56	24

99	32

56

24

Merging

Merge sort

Dividing

56	24	99	32	9	61	57	79

56	24	99	32

9	61	57	79

56	24

99	32

56

24

Merging

24	56

Merge sort

Dividing

56	24	99	32	9	61	57	79

56	24	99	32

9	61	57	79

56	24

99	32

56   24   99   32

Merging

24	56

Merge sort

Dividing

56	24	99	32	9	61	57	79

56	24	99	32

9	61	57	79

56	24

99	32

56

24

99

32

Merging

24	56

32	99

Merge sort

Dividing

56 | 24 | 99 | 32 | 9 | 61 | 57 | 79

56 | 24 | 99 | 32   9 | 61 | 57 | 79

56 | 24   99 | 32

56   24   99   32

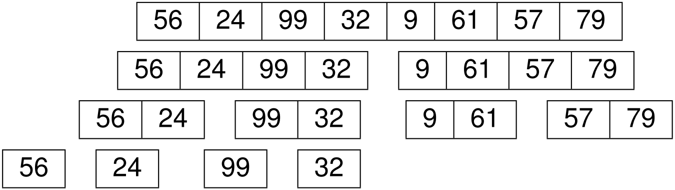
Merging

24 | 56   32 | 99

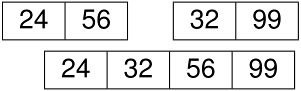
24 | 32 | 56 | 99

Merge sort

Dividing



Merging



Merge sort

Dividing

[56 | 24 | 99 | 32 | 9 | 61 | 57 | 79]

[56 | 24 | 99 | 32] [9 | 61 | 57 | 79]

[56 | 24] [99 | 32] [9 | 61] [57 | 79]

[56] [24] [99] [32] [9] [61]

Merging

[24 | 56] [32 | 99]

[24 | 32 | 56 | 99]

Merge sort

Dividing

| 56 | 24 | 99 | 32 | 9 | 61 | 57 | 79 |

| 56 | 24 | 99 | 32 | | 9 | 61 | 57 | 79 |

| 56 | 24 | | 99 | 32 | | 9 | 61 | | 57 | 79 |

| 56 | | 24 | | 99 | | 32 | | 9 | | 61 |

Merging

| 24 | 56 | | 32 | 99 | | 9 | 61 |

| 24 | 32 | 56 | 99 |

Merge sort

Dividing

[56 | 24 | 99 | 32 | 9 | 61 | 57 | 79]

[56 | 24 | 99 | 32] [9 | 61 | 57 | 79]

[56 | 24] [99 | 32] [9 | 61] [57 | 79]

[56] [24] [99] [32] [9] [61] [57] [79]

Merging

[24 | 56] [32 | 99] [9 | 61]

[24 | 32 | 56 | 99]

Merge sort

Dividing

56	24	99	32	9	61	57	79

[56, 24, 99, 32] [9, 61, 57, 79]

[56, 24] [99, 32] [9, 61] [57, 79]

[56] [24] [99] [32] [9] [61] [57] [79]

Merging

[24, 56] [32, 99] [9, 61] [57, 79]

[24, 32, 56, 99]

Merge sort

Dividing

56	24	99	32	9	61	57	79

56	24	99	32		9	61	57	79

56	24		99	32		9	61		57	79

56	24	99	32	9	61	57	79

Merging

24	56		32	99		9	61		57	79

24	32	56	99		9	57	61	79

Merge sort

Dividing

| 56 | 24 | 99 | 32 | 9 | 61 | 57 | 79 |

| 56 | 24 | 99 | 32 | | 9 | 61 | 57 | 79 |

| 56 | 24 | | 99 | 32 | | 9 | 61 | | 57 | 79 |

| 56 | | 24 | | 99 | | 32 | | 9 | | 61 | | 57 | | 79 |

Merging

| 24 | 56 | | 32 | 99 | | 9 | 61 | | 57 | 79 |

| 24 | 32 | 56 | 99 | | 9 | 57 | 61 | 79 |

| 9 | 24 | 32 | 56 | 57 | 61 | 79 | 99 |

Merge sort complexity

What is the complexity of merge? Two lists of lengths n_1, n_2:

Merge sort complexity

What is the complexity of merge? Two lists of lengths n_1, n_2:

- Two lists of lengths n_1 and n_2: $O(n_1 + n_2)$ copy operations (need to copy each item)

Merge sort complexity

What is the complexity of merge? Two lists of lengths n_1, n_2:

- Two lists of lengths n_1 and n_2: $O(n_1 + n_2)$ copy operations (need to copy each item)
- No more comparisons than copy operations

Merge sort complexity

What is the complexity of merge? Two lists of lengths n_1, n_2:

- Two lists of lengths n_1 and n_2: $O(n_1 + n_2)$ copy operations (need to copy each item)
- No more comparisons than copy operations
- If original list length is n, total $O(n)$ work for each round of merging

Merge sort complexity

What is the complexity of merge? Two lists of lengths n_1, n_2:

- Two lists of lengths n_1 and n_2: $O(n_1 + n_2)$ copy operations (need to copy each item)
- No more comparisons than copy operations
- If original list length is n, total $O(n)$ work for each round of merging

Merge sort complexity = merging $\times$ # number of divisions

Merge sort complexity

What is the complexity of merge? Two lists of lengths n_1, n_2:

- Two lists of lengths n_1 and n_2: $O(n_1 + n_2)$ copy operations (need to copy each item)
- No more comparisons than copy operations
- If original list length is n, total $O(n)$ work for each round of merging

Merge sort complexity = merging $\times$ # number of divisions

- Number of division levels $O(\log n)$ (like binary search)

Merge sort complexity

What is the complexity of merge? Two lists of lengths n_1, n_2:

- Two lists of lengths n_1 and n_2: $O(n_1 + n_2)$ copy operations (need to copy each item)
- No more comparisons than copy operations
- If original list length is n, total $O(n)$ work for each round of merging

Merge sort complexity = merging $\times$ # number of divisions

- Number of division levels $O(\log n)$ (like binary search)
- Log-linear: $O(n \log n)$

Merge sort complexity

What is the complexity of merge? Two lists of lengths n_1, n_2:

- Two lists of lengths n_1 and n_2: $O(n_1 + n_2)$ copy operations (need to copy each item)
- No more comparisons than copy operations
- If original list length is n, total $O(n)$ work for each round of merging

Merge sort complexity = merging $\times$ # number of divisions

- Number of division levels $O(\log n)$ (like binary search)
- Log-linear: $O(n \log n)$
- Big improvement over selection sort!

Merge sort complexity

What is the complexity of merge? Two lists of lengths n_1, n_2:

- Two lists of lengths n_1 and n_2: $O(n_1 + n_2)$ copy operations (need to copy each item)
- No more comparisons than copy operations
- If original list length is n, total $O(n)$ work for each round of merging

Merge sort complexity = merging $\times$ # number of divisions

- Number of division levels $O(\log n)$ (like binary search)
- Log-linear: $O(n \log n)$
- Big improvement over selection sort!
- Does need some more space due to copying lists

Complexity classes

Fast algorithm: worst-case running time grows slowly with input size

- $O(1)$: constant running time — primitive operations
- $O(\log n)$: logarithmic running time — binary search
- $O(n)$: linear running time — linear search
- $O(n \log n)$: log-linear time — merge sort
- $O(n^c)$: polynomial running time — selection sort
- $O(c^n)$: exponential running time — ??

Sorting is a canonical algorithms problem

Many algorithms exist: bubble sort, insertion sort, quick sort, radix sort, heap sort, ...

- Useful for developing algorithmic thinking – eg randomized algorithms

Sorting is a canonical algorithms problem

Many algorithms exist: bubble sort, insertion sort, quick sort, radix sort, heap sort, ...

- Useful for developing algorithmic thinking – eg randomized algorithms

Theoretical bound for worst-case performance is $O(n \log n)$ – **we can't do better than merge sort**

Sorting is a canonical algorithms problem

Many algorithms exist: bubble sort, insertion sort, quick sort, radix sort, heap sort, ...

- Useful for developing algorithmic thinking – eg randomized algorithms

Theoretical bound for worst-case performance is $O(n \log n)$ – **we can't do better than merge sort**

But other algorithms are **better on average**

- Python uses **timsort** (In 2002, a Dutch guy called Tim got frustrated with existing algorithms)
- Exploit the fact that lists tend to be partly sorted already

Sorting is a canonical algorithms problem

Many algorithms exist: bubble sort, insertion sort, quick sort, radix sort, heap sort, ...

- Useful for developing algorithmic thinking – eg randomized algorithms

Theoretical bound for worst-case performance is $O(n \log n)$ – **we can't do better than merge sort**

But other algorithms are **better on average**

- Python uses **timsort** (In 2002, a Dutch guy called Tim got frustrated with existing algorithms)
- Exploit the fact that lists tend to be partly sorted already

Review

Sorting is a canonical computer science problem

- We've looked at two (of many) algorithms
- Selection sort involves repeatedly finding minimum element – intuitive but slow
- Merge sort is blazingly fast and has a neat recursive structure

Review exercises

- Implement sorting
- More looping and function practice