Data Structures and Algorithms

Lecture 3: Sorting

Heikki Peura

h.peura@imperial.ac.uk

Last time

Searching and complexity

- Big-O notation
- Binary search vs linear search

Plan for today:

- **Sorting** algorithms
- Selection sort and merge sort

Asymptotic analysis

Principle 0: measure number of basic operations as function of input size

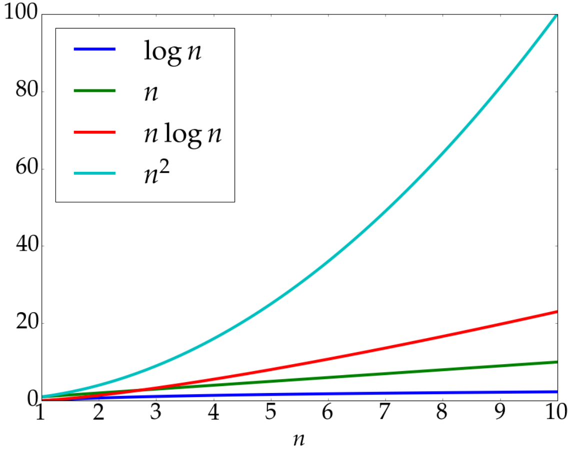
Principle 1: focus on worst-case analysis

Principle 2: ignore constant factors and lower-order terms

Principle 3: 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 (eg `x*y`) (for not too big numbers)
- Comparisons (eg `x > 0`)
- Assign a variable (eg `x = 2`), read/write memory

Basic operations

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

- Arithmetic operations (eg `x*y`) (for not too big numbers)
- Comparisons (eg `x > 0`)
- Assign a variable (eg `x = 2`), read/write memory

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 Lecture 4... 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 $sort(n) + \log(n) < n$?
- 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
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

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

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:

```python
def selection_sort(L):
    n = len(L)
    for index in range(n):
        min_index = find_min_index(L, index) # index with smallest element
        L[index], L[min_index] = L[min_index], L[index] # swap positions
    return L
```

Let's assume the function is implemented. What is its complexity?

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...

How to merge two sorted lists?

x =

24	32	56

y =

19	57	61

z =

How to merge two sorted lists?

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

How to merge two sorted lists?

x = | 24 | 32 | 56 | i1 = 0

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

z = | 19 |

How to merge two sorted lists?

x = | 24 | 32 | 56 | i1 = 1

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

z = | 19 | 24 |

x =	24	32	56
y =	19	57	61
z =	19	24	

How to merge two sorted lists?

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

How to merge two sorted lists?

x =	24	32	56	i1 = 3
y =	19	57	61	i2 = 1
z =	19	24	32	56

How to merge two sorted lists?

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

How to merge two sorted lists?

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?

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
- Comparisons $O(\max\{n_1, n_2\})$
- Two lists of lengths n_1 and n_2: $O(n_1 + n_2)$ copy operations (need to copy each item)

Sidebar: recursion

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

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

Sidebar: recursion

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

- As a function: $\text{fact}(n) = n * (n-1) * (n-2) * \cdots * 2 * 1$
- 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))
```

Sidebar: recursion

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

- As a function: $\text{fact}(n) = n * (n-1) * (n-2) * \cdots * 2 * 1$
- 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))
```

But we can also write the factorial as follows:

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

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

Sidebar: recursion

We can also write the factorial as follows:

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

Factorial can be expressed as a smaller version of itself:

Sidebar: recursion

We can also write the factorial as follows:

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

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

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 -> 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

Merge sort

Dividing

56	24	99	32	9	61	57	79

Merge sort

Dividing

56	24	99	32	9	61	57	79

56	24	99	32

9	61	57	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

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

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]

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:

- Comparisons $O(\max\{n_1, n_2\})$

Merge sort complexity

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

- Comparisons $O(\max\{n_1, n_2\})$
- 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:

- Comparisons $O(\max\{n_1, n_2\})$
- Two lists of lengths n_1 and n_2: $O(n_1 + n_2)$ copy operations (need to copy each item)
- 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:

- Comparisons $O(\max\{n_1, n_2\})$
- Two lists of lengths n_1 and n_2: $O(n_1 + n_2)$ copy operations (need to copy each item)
- If original list length is n, total $O(n)$ work for each round of merging

Merge sort complexity = merging * # number of divisions

Merge sort complexity

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

- Comparisons $O(\max\{n_1, n_2\})$
- Two lists of lengths n_1 and n_2: $O(n_1 + n_2)$ copy operations (need to copy each item)
- If original list length is n, total $O(n)$ work for each round of merging

Merge sort complexity = merging * # 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:

- Comparisons $O(\max\{n_1, n_2\})$
- Two lists of lengths n_1 and n_2: $O(n_1 + n_2)$ copy operations (need to copy each item)
- If original list length is n, total $O(n)$ work for each round of merging

Merge sort complexity = merging * # 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:

- Comparisons $O(\max\{n_1, n_2\})$
- Two lists of lengths n_1 and n_2: $O(n_1 + n_2)$ copy operations (need to copy each item)
- If original list length is n, total $O(n)$ work for each round of merging

Merge sort complexity = merging * # 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:

- Comparisons $O(\max\{n_1, n_2\})$
- Two lists of lengths n_1 and n_2: $O(n_1 + n_2)$ copy operations (need to copy each item)
- If original list length is n, total $O(n)$ work for each round of merging

Merge sort complexity = merging * # 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

Workshop after the break

- Implement sorting
- More looping and function practice

Workshop

Workshop zip file on the Hub

- HTML instructions
- At some point, you'll need the `.py`-file with skeleton code (open in Spyder)