UCLA Samueli Computer Science

CS32: Introduction to Computer Science

Discussion Week 7

Junheng Hao, Ramya Satish

Feb 22, 2019

Announcement

UCLA Samueli Computer Science

- Midterm Part 2 is due on Tuesday, Feb 26.
- Project 3 Part 2 is due on Thursday, February 28

Outline

UCLA Samueli Computer Science

- STL (continued)
- Algorithmic Efficiency (Big-O notation)
- Sorting (1)

STL Table list

Check out when you need

- C++ Containers library
  - Sequence containers: `array, vector, deque, list, forward_list`
  - Container adaptors: `set, map, multiset, multimap`
  - Associate containers: `unordered_set, unordered_map`
  - Unordered associative containers: `stack, queue, priority_queue`
- Check link
  - https://en.cppreference.com/w/cpp/container#Sequence_containers

Project 3 warmup

Why fail on test 3?

- Pitfalls in modifying STL containers while you're traversing them, given the various iterator invalidation rules.
- Another reminder: For a container of raw pointers to dynamically allocated objects, the container operations know nothing about that: erasing an item does **NOT** call delete on the pointer.

STL

Iterator invalidation

<table>
<tr><th rowspan="2">Category</th><th rowspan="2">Container</th><th colspan="2">After insertion, are...</th><th colspan="2">After erasure, are...</th><th rowspan="2">Conditionally</th></tr>
<tr><th>iterators valid?</th><th>references valid?</th><th>iterators valid?</th><th>references valid?</th></tr>
<tr><td rowspan="8">Sequence containers</td><td><code>array</code></td><td colspan="2">N/A</td><td colspan="2">N/A</td><td></td></tr>
<tr><td rowspan="3"><code>vector</code></td><td colspan="2">No</td><td colspan="2">N/A</td><td>Insertion changed capacity</td></tr>
<tr><td colspan="2">Yes</td><td colspan="2">Yes</td><td>Before modified element(s)</td></tr>
<tr><td colspan="2">No</td><td colspan="2">No</td><td>At or after modified element(s)</td></tr>
<tr><td rowspan="2"><code>deque</code></td><td rowspan="2">No</td><td>Yes</td><td colspan="2">Yes, except erased element(s)</td><td>Modified first or last element</td></tr>
<tr><td>No</td><td colspan="2">No</td><td>Modified middle only</td></tr>
<tr><td><code>list</code></td><td colspan="2">Yes</td><td colspan="2">Yes, except erased element(s)</td><td></td></tr>
<tr><td><code>forward_list</code></td><td colspan="2">Yes</td><td colspan="2">Yes, except erased element(s)</td><td></td></tr>
<tr><td>Associative containers</td><td><code>set</code>
<code>multiset</code>
<code>map</code>
<code>multimap</code></td><td colspan="2">Yes</td><td colspan="2">Yes, except erased element(s)</td><td></td></tr>
<tr><td rowspan="2">Unordered associative containers</td><td rowspan="2"><code>unordered_set</code>
<code>unordered_multiset</code>
<code>unordered_map</code>
<code>unordered_multimap</code></td><td>No</td><td rowspan="2">Yes</td><td colspan="2">N/A</td><td>Insertion caused rehash</td></tr>
<tr><td>Yes</td><td colspan="2">Yes, except erased element(s)</td><td>No rehash</td></tr>
</table>

STL Table list

Member function table

		Sequence containers					Associative containers				Unordered associative containers				Container adaptors		
Header		<array>	<vector>	<deque>	<forward_list>	<list>	<set>		<map>		<unordered_set>		<unordered_map>		<stack>	<queue>	
Container		array	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
	(constructor)	(implicit)	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
	(destructor)	(implicit)	~vector	~deque	~forward_list	~list	~set	~multiset	~map	~multimap	~unordered_set	~unordered_multiset	~unordered_map	~unordered_multimap	~stack	~queue	~priority_queue
	operator=	(implicit)	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=
	assign		assign	assign	assign	assign											
Iterators	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin			
	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin			
	end	end	end	end	end	end	end	end	end	end	end	end	end	end			
	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend			
	rbegin	rbegin	rbegin	rbegin		rbegin	rbegin	rbegin	rbegin	rbegin							
	crbegin	crbegin	crbegin	crbegin		crbegin	crbegin	crbegin	crbegin	crbegin							
	rend	rend	rend	rend		rend	rend	rend	rend	rend							
	crend	crend	crend	crend		crend	crend	crend	crend	crend							
Element access	at	at	at	at					at				at				
	operator[]	operator[]	operator[]	operator[]					operator[]				operator[]				
	data	data	data														
	front	front	front	front	front	front										front	top
	back	back	back	back		back									top	back	
Capacity	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty
	size	size	size	size		size	size	size	size	size	size	size	size	size	size	size	size
	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size			
	resize		resize	resize	resize	resize											
	capacity		capacity								bucket_count	bucket_count	bucket_count	bucket_count			
	reserve		reserve								reserve	reserve	reserve	reserve			
	shrink_to_fit		shrink_to_fit	shrink_to_fit													
Modifiers	clear		clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear			
	insert		insert	insert	insert_after	insert	insert	insert	insert	insert	insert	insert	insert	insert			
	insert_or_assign								insert_or_assign				insert_or_assign				
	emplace		emplace	emplace	emplace_after	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace			
	emplace_hint						emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint			
	try_emplace								try_emplace				try_emplace				
	erase		erase	erase	erase_after	erase	erase	erase	erase	erase	erase	erase	erase	erase			
	push_front			push_front	push_front	push_front											
	emplace_front			emplace_front	emplace_front	emplace_front											
	pop_front			pop_front	pop_front	pop_front										pop	pop
	push_back		push_back	push_back		push_back									push	push	push
	emplace_back		emplace_back	emplace_back		emplace_back									emplace	emplace	emplace
	pop_back		pop_back	pop_back		pop_back									pop		
	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap
	merge				merge	merge	merge	merge	merge	merge	merge	merge	merge	merge			
	extract						extract	extract	extract	extract	extract	extract	extract	extract			
List operations	splice				splice_after	splice											
	remove				remove	remove											
	remove_if				remove_if	remove_if											
	reverse				reverse	reverse											
	unique				unique	unique											
	sort				sort	sort											
Lookup	count						count	count	count	count	count	count	count	count			
	find						find	find	find	find	find	find	find	find			
	contains						contains	contains	contains	contains	contains	contains	contains	contains			
	lower_bound						lower_bound	lower_bound	lower_bound	lower_bound							
	upper_bound						upper_bound	upper_bound	upper_bound	upper_bound							
	equal_range						equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range			
Observers	key_comp						key_comp	key_comp	key_comp	key_comp							
	value_comp						value_comp	value_comp	value_comp	value_comp							
	hash_function										hash_function	hash_function	hash_function	hash_function			
	key_eq										key_eq	key_eq	key_eq	key_eq			
Allocator	get_allocator		get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator			
Container		array	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
		Sequence containers					Associative containers				Unordered associative containers				Container adaptors		

*Smart Pointer

A good tool in modern C++

- A smart pointer is an abstract data type that simulates a pointer while providing added features, such as automatic memory management or bounds checking.
- C++ libraries provide implementations of smart pointers in the form of `unique_ptr`, `shared_ptr` and `weak_ptr`
- Trade-off by using smart pointers: may increase memory usage (for example in `list`)
- More info: [Smart pointer tutorial]

```
// normal pointers
void UseNormalPointer{
  MyClass *ptr = new MyClass();
  ptr->doSomething();
}
// We must delete ptr to avoid memory leak!
```

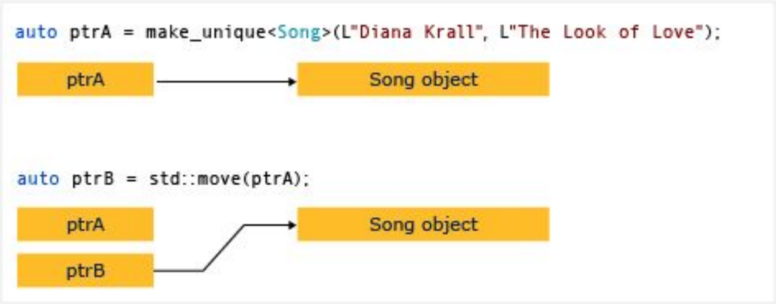
```
// smart pointers, defined in std
void UseSmartPointer{
  unique_ptr<MyClass> ptr(new MyClass());
  ptr->doSomething();
}
// ptr is deleted automatically here!
// unique_ptr:encapsulated pointer as only data member
```

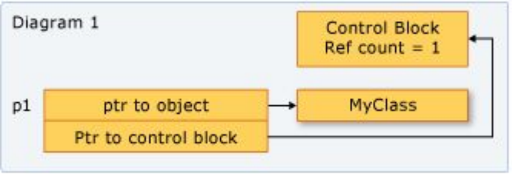
*Smart Pointer

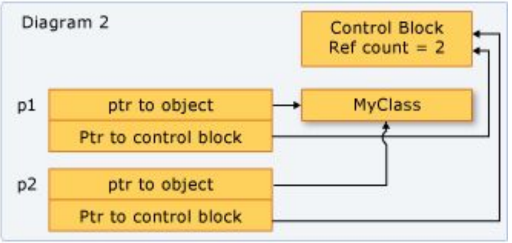
`unique_ptr` and `shared_ptr`

UCLA Samueli Computer Science

- `unique_ptr`
  - Allows exactly one owner of the underlying pointer.
  - Can be moved to a new owner, but not copied or shared.
  - Small and efficient (the size is one pointer as data member)
  - More about `unique_ptr`: [unique_ptr tutorial]
- `shared_ptr`
  - Reference-counted smart pointer. Use when you want to assign one raw pointer to multiple owners.
  - The size is two pointers; one for the object and one for the shared control block that contains the reference count.
  - More about `shared_ptr`: [shared_ptr tutorial]

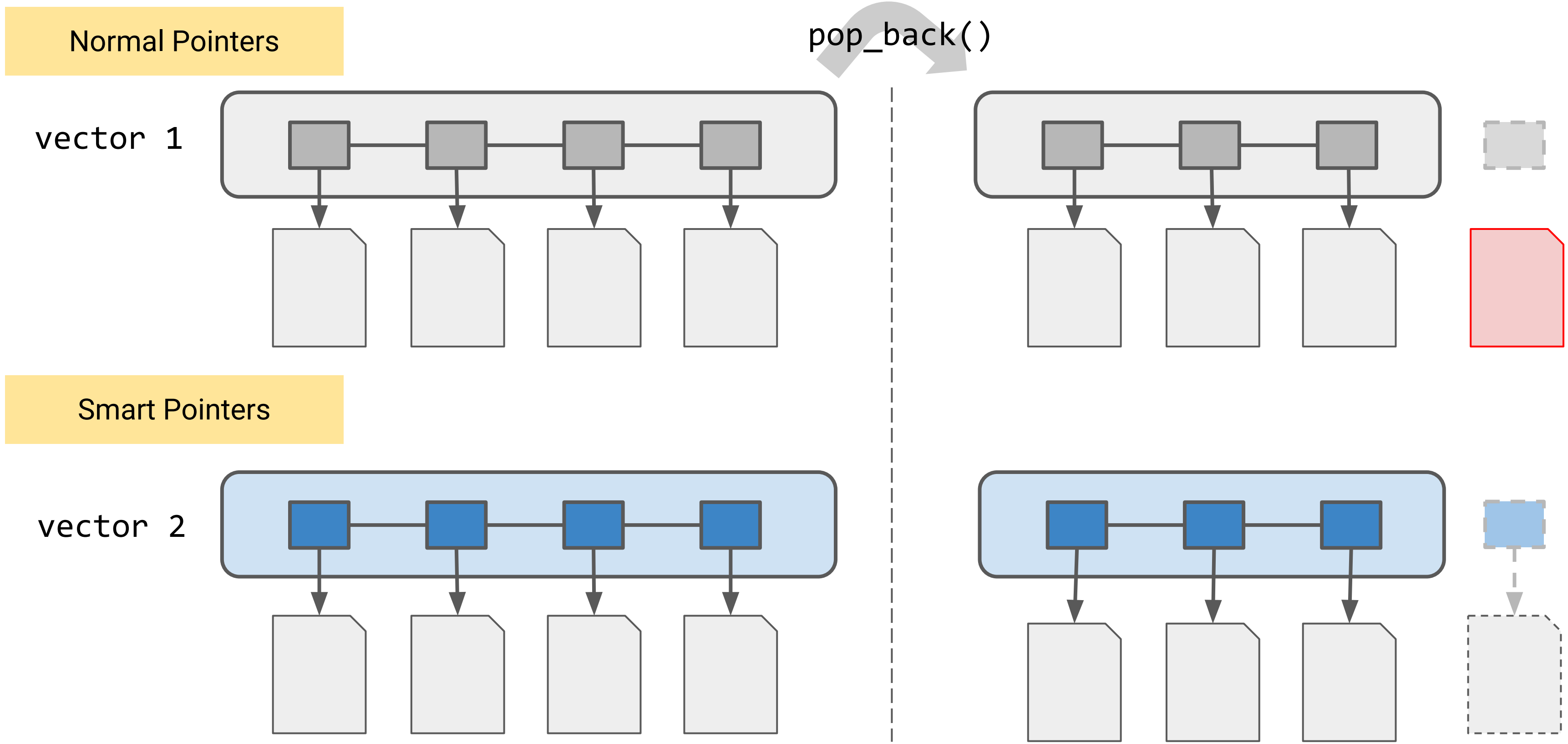






Pointers vs Smart Pointers

Example: Container of pointers



Algorithm Efficiency

Note: Complexity of a program

UCLA Samueli Computer Science

- Quantify the efficiency of a program.
- The magnitude of time and space cost for an algorithm given certain size of input.
  - Time complexity: quantifies the run time.
  - Space complexity: quantifies the usage of the memory (or sometimes hard disk drives, cloud disk drives, etc.).
- Naturally, the size of input determines how long a program runs.
  - Often, the larger the size of input, the longer the run time. But not always that case.
  - Consider: sort an array of 1,000 items and 1,000,000 items vs get size of an array of 1,000 items and 1,000,000 items
- Big-O notation

Big-O Notation

Formal definition

If you are interested in formal definition, check here.

Well, you can simply understand as how many operations given input size of `n` regardless of the constant.

No need to memorize definitions.

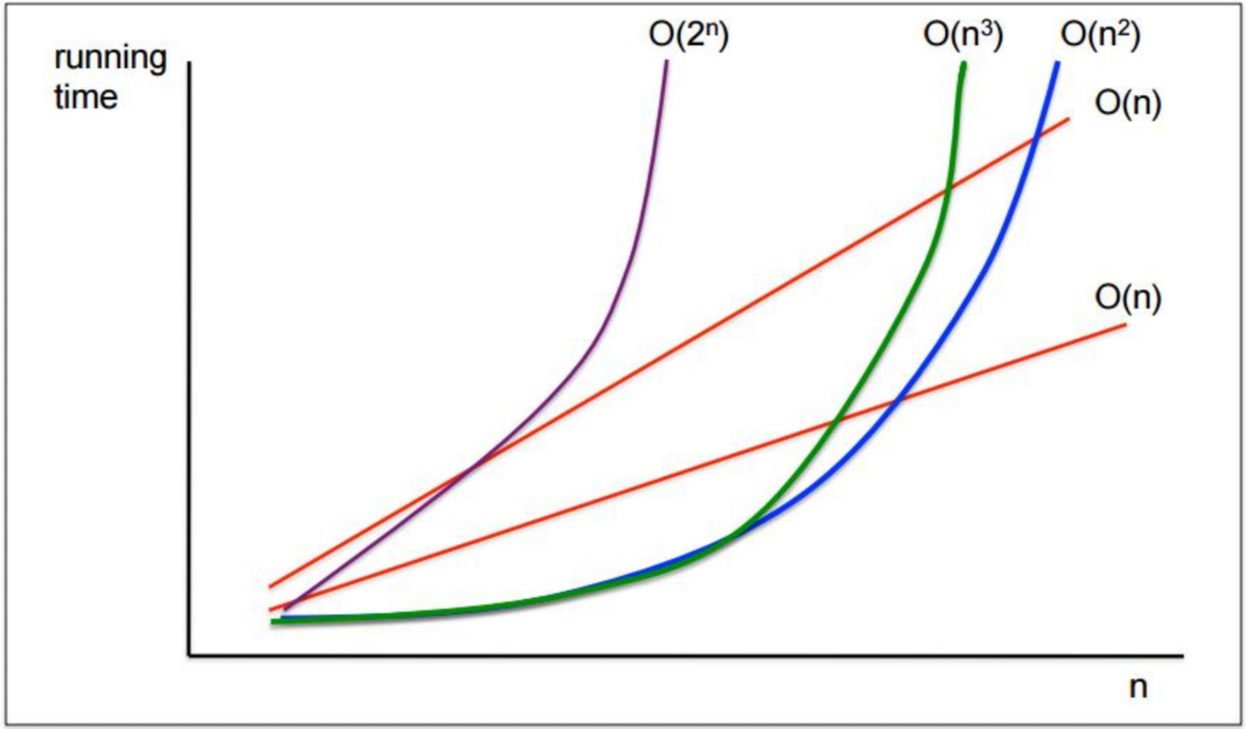
Example: if your program takes,

- about `n` steps → `O(n)`
- about `2n` steps → `O(n)`
- about `n^2` steps → `O(n^2)`
- about `3n^2+10n` steps → `O(n^2)`
- about `2^n` steps → `O(2^n)`

Question: What is the speed of growth for typical function?

`f(n) = log(n) / n / n^2 / 2^n / n!`

Big-O Notation
Growth speed



Big-O Arithmetic

How to determine the entire program?

Generally,

- If things happen sequentially, we add Big-Os;
- If one thing happen with another, then we multiply Big-Os.
- Watch the LOOPS in your programs!

Rules:

$$O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$$

$$O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$$

Efficiency Analysis

Example 1: Linear Search

- Linear search: Look for one item in an unsorted array
- Best cases? Average cases? Worst cases?
- What if the array is ordered?

```
int linear_search(array arr, size n, value v)
{
  for (int i=0; i<n; i++)
  {
    if (arr[i] == v)
      return i;
  }
  return -1;
}
```

Efficiency Analysis

Example 2: Enumerate all pairs

UCLA Samueli Computer Science

- Task: Find all pairs from one array (Note: [1,2] and [2,1] are considered different pairs)

```
int all_pairs(array arr, size n, value v)
{
  for (int i=0; i<n; i++)
  {
    for (int j=0; i<n; j++)
    {
      if (i != j)
        cout << “Pair:” << arr[i] << “and” << arr[j] <<endl;
    }
  }
  return -1;
}
```

Efficiency Analysis

Example 3: Binary search

UCLA Samueli Computer Science

- Task: Look for one item in a sorted array

```
// this is pseudo code
int binary_search(array arr, value v, start_index s, end_index e)
{
  if (s > e) return -1
  find the middle point i=(s+e)/2
  if (arr[i] == v) return i
  else if (arr[i] < v) return binary_search(arr, v, i+1, e)
  else return binary_search(arr, v, s, i-1)
}
```

Big-O and Complexity

UCLA Samueli Computer Science

Big O	Name	n = 128
$O(1)$	constant	1
$O(\log n)$	logarithmic	7
$O(n)$	linear	128
$O(n \log n)$	"n log n"	896
$O(n^2)$	quadratic	16192
$O(n^k)$, $k >= 1$	polynomial	
$O(2^n)$	exponential	10^{40}
$O(n!)$	factorial	10^{214}

Question: Can you find an algorithms with `O(n!)` complexity?

Sorting
Introduction

Most important algorithm ever!

Methods:

- Selection sort
- Bubble sort
- Insertion sort
- Merge sort
- Quick sort

Focus on:

1. Steps for each sorting algorithm
2. Runtime complexity for worst cases, best cases and average cases
3. Space complexity
4. How about additional assumptions, such as the array is “almost sorted” / “reversed” arrays

Sorting

Selection sort

Steps:

4	3	1	5	2
1	3	4	5	2
1	2	4	5	3
1	2	3	5	4
1	2	3	4	5

Idea: Find the smallest item in the unsorted portion and place it in the front.

Runtime complexity:

Average: $O(n^2)$

Worst: $O(n^2)$

Best: $O(n^2)$

Space complexity: $O(1)$

Sorting

Insertion sort

Steps:

4	3	1	5	2
3	4	1	5	2
1	3	4	5	2
1	3	4	5	2
1	2	3	4	5

Idea: Pick one from the unsorted part and place it in the right position.

Runtime complexity:

Average: $O(n^2)$

Worst: $O(n^2)$

Best: $O(n)$

Space complexity: $O(1)$

Sorting

Bubble sort

Steps:

4	3	1	5	2
3	4	1	5	2
3	1	4	5	2
3	1	4	2	5
1	3	2	4	5
1	2	3	4	5

Idea: Well, just "bubble" as its name

Runtime complexity:

Average: $O(n^2)$

Worst: $O(n^2)$

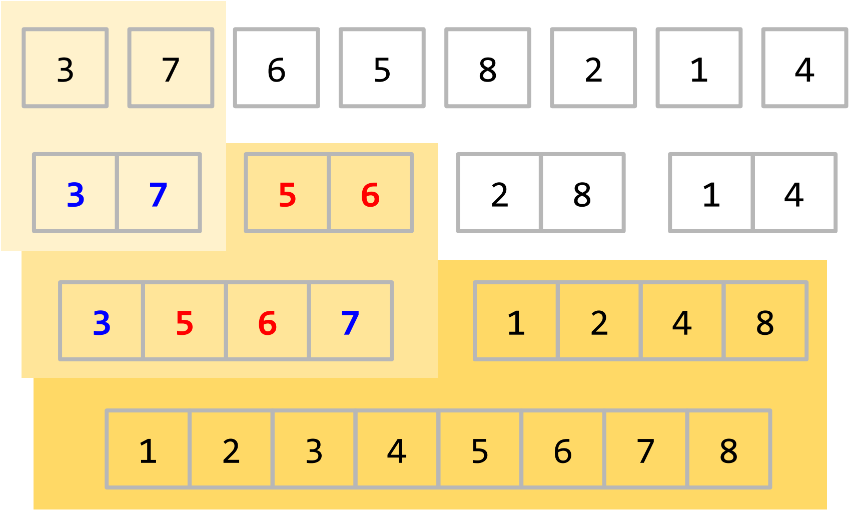
Best: $O(n)$

Space complexity: $O(1)$

Sorting

Merge sort

Steps:



Idea: Divide and conquer

Runtime complexity:

Average: $O(n \log n)$

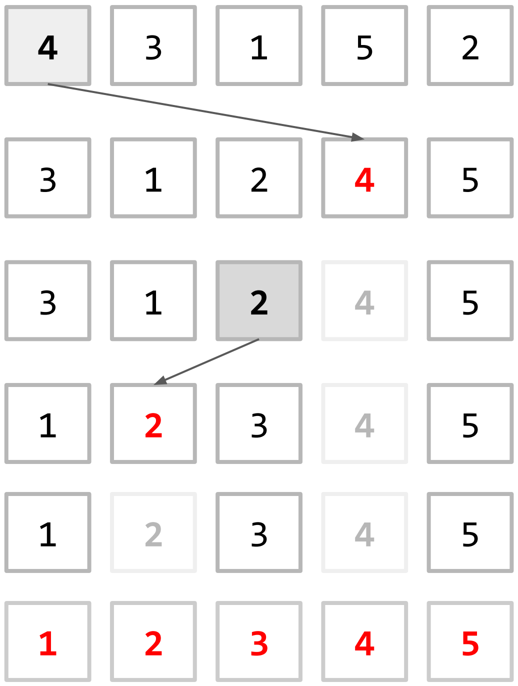
Worst: $O(n \log n)$

Best: $O(n \log n)$

Space complexity: $O(n)$

Sorting
Quicksort

Steps:



Idea: Set a pivot. Numbers less then pivot are placed to front while other to end.

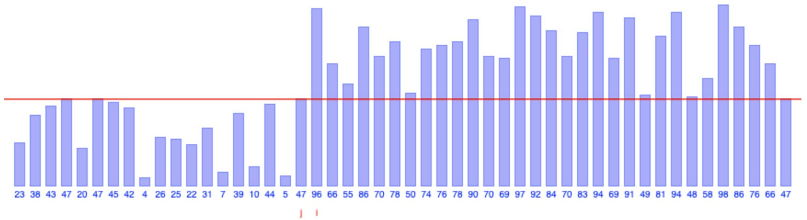
Runtime complexity:

Average: $O(n \log n)$

Worst: $O(n^2)$

Best: $O(n \log n)$

Space complexity: $O(\log n)$

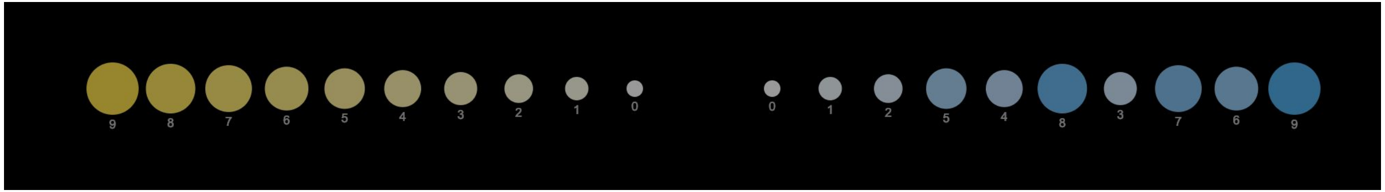


Sorting

Other methods and complexity?

- O(n log n) is faster than O(n^2) → Merge sort is more efficient than selection, insertion and bubble sort in runtime.
- O(n log n) is best average complexity that a general sorting algorithm can achieve.
- With more information about the data provided, you can sometimes sort things almost linearly.

Question: What is the complexity of these sorting algorithms if you know the array is **reversed**? What if the array is **almost already sorted**?

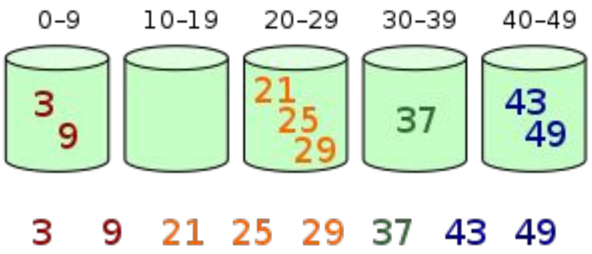


Sorting

Other methods and complexity?

There are many other sorting methods:

- Shell sort (shell 1959, Knuth 1973, Ciura 2001)
- Quicksort 3-way
- Heap sort
- Bucket sort



Sorting

Why sorting is important?

Sorting is the most important and basic algorithm. Many other real-world problems are somewhat based on sorting, including:

Sorting Algorithms Animations: https://www.toptal.com/developers/sorting-algorithms
Other good demos:
https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html
http://sorting.at/

Sorting

Variant sorting problems

Question: How about get the *K-th* largest numbers in one array?

Leetcode question #215

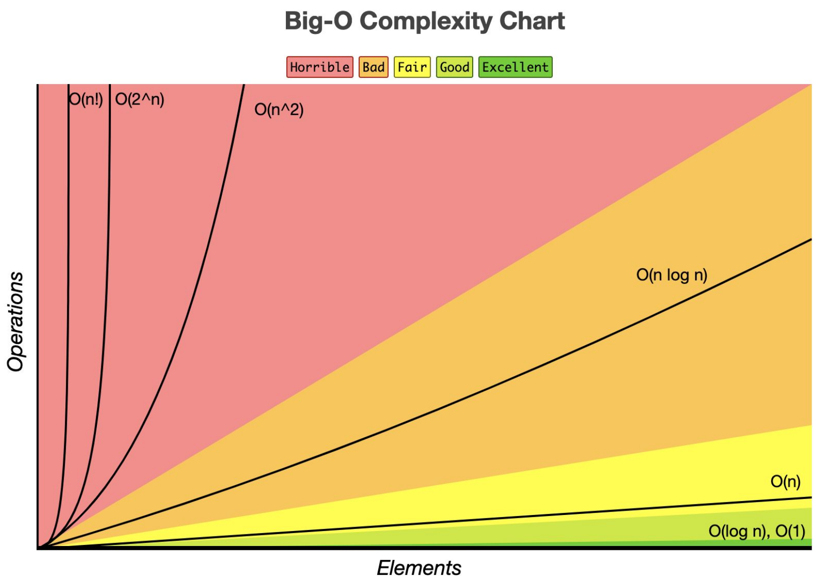
Hint:

1. How to find the k-th largest numbers by merge sort and quicksort (or other sort methods)? What are the average and worst complexity?
2. What data structures is good to use?

Big-O Notation

Big-O Complexity Chart

UCLA Samueli Computer Science



Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

UCLA Samueli Computer Science

Thank you!

Q & A