UCLA Samueli Computer Science

CS32: Introduction to Computer Science II

Discussion Week 7

Yichao (Joey)

Feb. 21, 2019

Credit: Carey Nachenberg, Junheng Hao

Announcements

- Project 3 part 2 is due 11:00PM Thursday, February 27.
- Homework 4 is due 11:00 PM Tuesday, March 3.
- Midterm 2 is scheduled February 25.

  open book, open notes, no electronic devices,

  emphasizing **stacks and queues, inheritance and polymorphism, and recursion** (not templates, not big-O).

Outline Today

UCLA Samueli Computer Science

- STL Iterator
- Algorithm Efficiency and Big O Notation
- Sorting Algorithms
- Note: We'll be combining the presentation portion and worksheet today!

STL: Standard Template Library

Easy and efficient implementation

UCLA Samueli Computer Science

- A collection of pre-written, tested classes provided by C++.
- All built using templates (adaptive with many data types).
- Provide useful data structures
  - `vector(array), set, list, map, stack, queue`
- Standard functions:
  - Common ones: `.size(), .empty()`
  - For a container that is neither stack or queue: `.insert(), .erase(), swap(), .clear()`
  - For list or vector: `.push_back(), .pop_back()`
  - For set or map: `.find(), .count()`
  - More on stacks and queues…

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

Iterator erase

```
std::list<int> mylist;
std::list<int>::iterator it1,it2;

for (int i=1; i<10; ++i) mylist.push_back(i*10);
it1 = it2 = mylist.begin();
advance (it2,6);
++it1;

it1 = mylist.erase (it1);
it2 = mylist.erase (it2);
++it1;
--it2;
mylist.erase (it1,it2);

std::cout << "mylist contains:";
for (it1=mylist.begin(); it1!=mylist.end(); ++it1)
  std::cout << ' ' << *it1;
std::cout << '\n';
```

```
// 10 20 30 40 50 60 70 80 90
// ^*
// ^                    *
//    ^                 *
// 10 30 40 50 60 70 80 90
//    ^              *
// 10 30 40 50 60 80 90
//    ^              *
//       ^           *
//       ^      *
// 10 30 60 80 90
//       ^
```

STL

removeOdds

```
int a[8] = { 2, 8, 5, 6, 7, 3, 4, 1 };
vector<int> li(a, a+8);

void removeOdds(vector<int>& li)
{
    vector<int>::iterator it = li.begin();
    while (it != li.end())
        if (*it % 2 == 1) {
            it = li.erase(it);
            cout<<*it<<endl;
    }
        else
            it++;
}
```

```
int a[8] = { 2, 8, 5, 6, 7, 3, 4, 1 };
list<int> li(a, a+8);

void removeOdds(list<int>& li)
{
    list<int>::iterator it = li.begin();
    while (it != li.end())
        if (*it % 2 == 1) {
            it = li.erase(it);
            cout<<*it<<endl;
    }
        else
            it++;
}
```

erase() return:
An iterator pointing to the element that followed the last element erased by the function call.

STL

removeBad

```
int a[8] = { 85, 80, 30, 70, 20, 15, 90, 10 };
list<Movie*> li;
for (int k = 0; k < 8; k++)
    li.push_back(new Movie(a[k]));
void removeBad(list<Movie*>& li)
{
    auto it=li.begin();
     while (it!=li.end())
       if ((*it)->rating() < 50) {
          delete *it;
          it = li.erase(it);
    }
       else
          it++;
}
```

* 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, shared_ptr and weak_ptr`

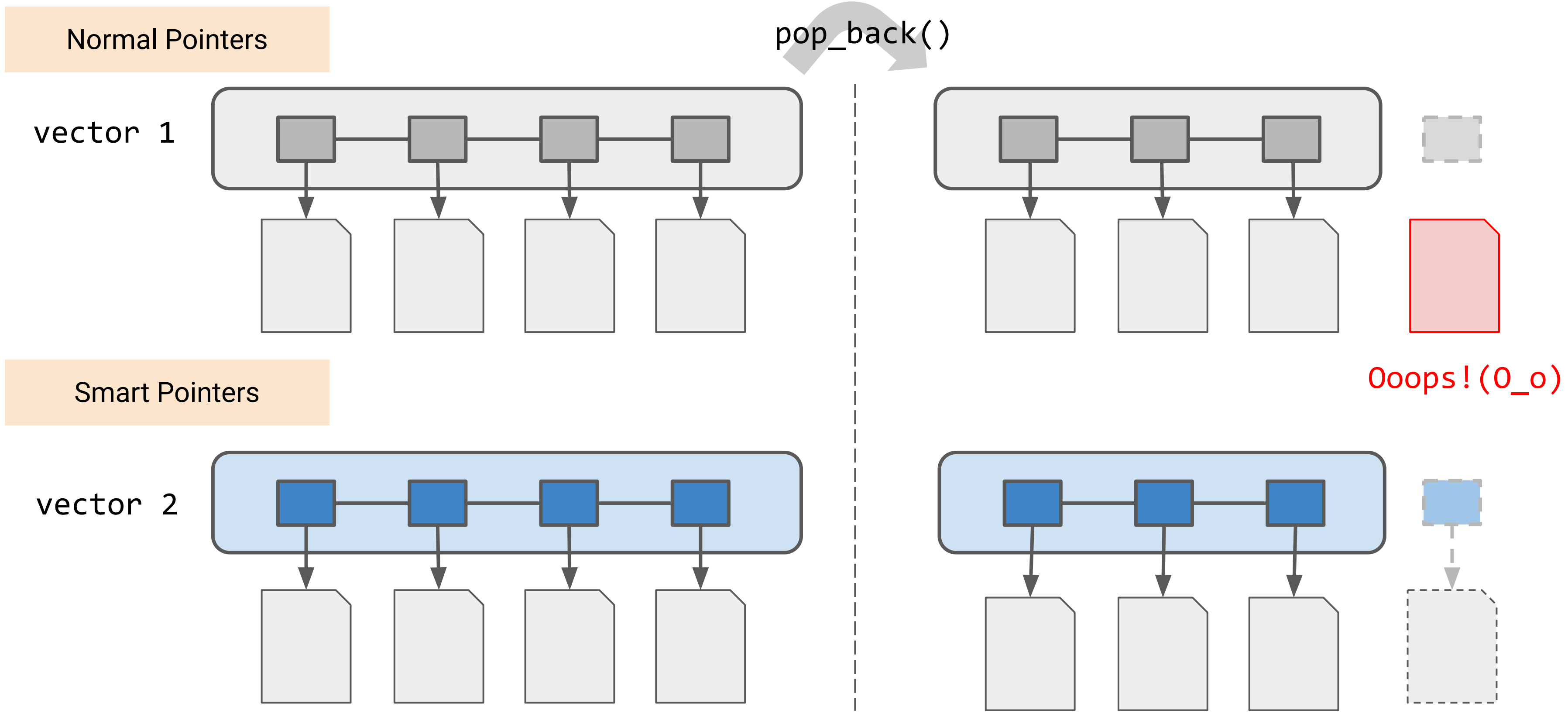
- **What is a smart pointer?**
  It's a type whose values can be used like pointers, but which provides the additional feature of automatic memory management: When a smart pointer is no longer in use, the memory it points to is deallocated

- **When should I use one?**
  In code which **involves tracking the ownership of a piece of memory,** allocating or de-allocating; the smart pointer often saves you the need to do these things explicitly.

- **But which smart pointer should I use in which of those cases?**
  - Use `std::unique_ptr` when you **don't intend to hold multiple references to the same object**. For example, use it for a pointer to memory which gets allocated on entering some scope and de-allocated on exiting the scope.
  - Use `std::shared_ptr` when you **do want to refer to your object from multiple places** - and do not want your object to be de-allocated until all these references are themselves gone.
  - Use `std::weak_ptr` when you **do want to refer to your object from multiple places** - for those references for which it's ok to ignore and deallocate (so they'll just note the object is gone when you try to dereference).

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,

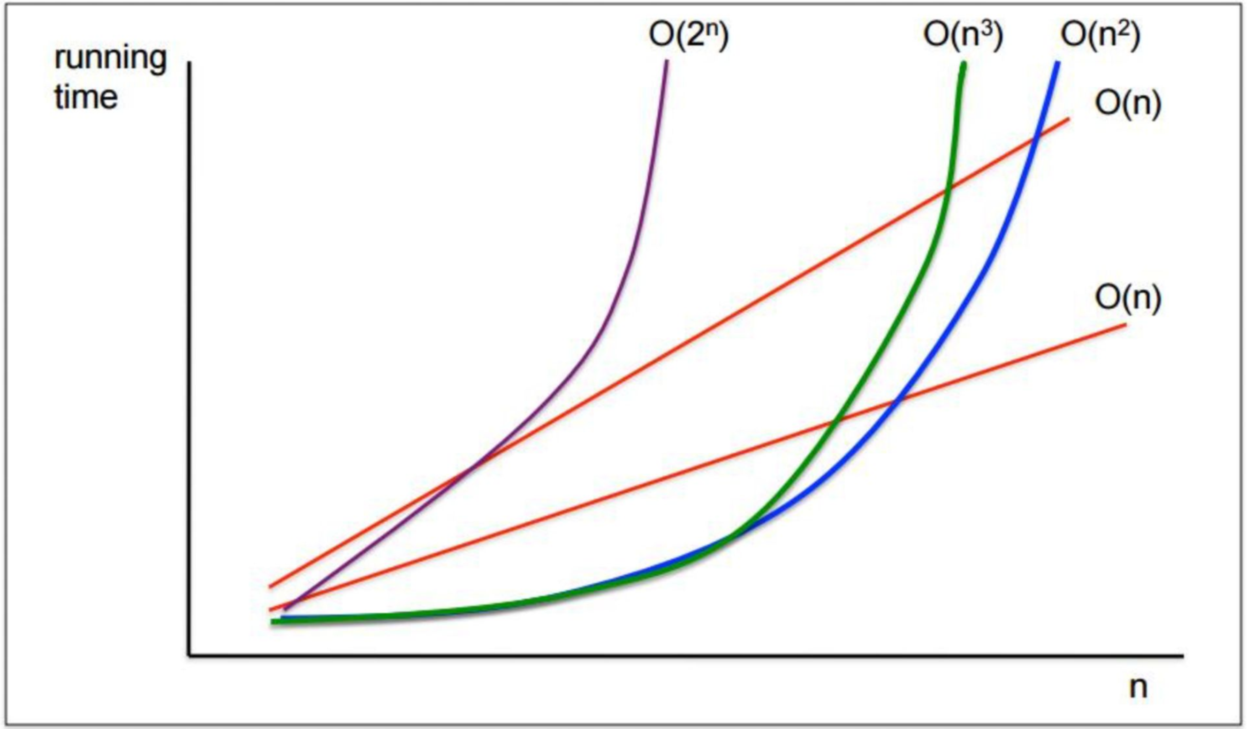
- about n steps → O(n)
- about 2n steps → O(n)
- about n^2 steps → O(n^2)
- about 3n^2+10n steps → O(n^2)
- about 2^n steps → O(2^n)

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 within another, then we **multiply** Big-Os.
- Simple rule: 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

- 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

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?

Big-O Worksheet Questions

Questions: 1,2,3,8

Hints:

1: The loop runs while is less than what value?

2: The innermost for loop is incremented by j each time. This is not a trivial detail!

3: You have to think about how vectors and sets are organized. What is the insertion time for each?

Big-O and Complexity

Worksheet (question 1)

UCLA Samueli Computer Science

What is the time complexity of the following code?

```
bool isPrime(int n) {
  if (n < 2 || n % 2 == 0) return false;
  if (n == 2) return true;
  for (int i = 3; (i * i) <= n; i += 2) {
    if (n % i == 0) return false;
  }
  return true;
}
```

Big-O and Complexity

Worksheet (question 2)

What is the time complexity of the following code?

```
int randomSum(int n) {
  int sum = 0;
  for(int i = 0; i < n; i++) {
    for(int j = 0; j < i; j++) {
      if(rand() % 2 == 1) {
        sum += 1;
      }
      for(int k = 0; k < j*i; k += j) {
        if(rand() % 2 == 2) {
          sum += 1;}
      }}}
  return sum;
}
```

Big-O and Complexity

Worksheet (question 3)

Find the time complexity of the following function.

```
int obfuscate(int a, int b) {
	vector<int> v;
	set<int> s;
	for (int i = 0; i < a; i++) {
		v.push_back(i);
		s.insert(i);
	}
	v.clear();

	int total = 0;
	if (!s.empty()) {
		for (int x = a; x < b; x++) {
			for (int y = b; y > 0; y--) {
				total += (x + y);
			}
		}
	}
	return v.size() + s.size() + total;
}
```

Big-O and Complexity

Worksheet (question 8)

What is the time complexity of the following code?

// assuming vector v is of size N, and head is the head of a linked list of size M

```
void foo(vector<int> v, Node* head) {
    if (!v.empty() && v[0] == 0)
        v.erase(v.begin());
    Node* current = head;
    while (current != NULL) {
        for (vector<int>::iterator itr = v.begin(); itr != v.end();
itr++) {
            if (*itr == current->val) {
                *itr = 0;
                break;                  }
        }
        current = current->next;
    }
}
```

Sorting
Introduction

UCLA Samueli Computer Science

Most important algorithm ever!

Methods:

- Selection sort
- Insertion sort
- Bubble 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

UCLA Samueli Computer Science

Steps:

5 3 4 1 2

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

Selection sort

```
void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}
```

Sorting

Insertion sort

UCLA Samueli Computer Science

Steps:

6 5 3 1 8 7 2 4

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:

6 5 3 1 8 7 2 4

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:

6 5 3 1 8 7 2 4

Idea: Divide and conquer

Runtime complexity:

Average: $O(n \log n)$

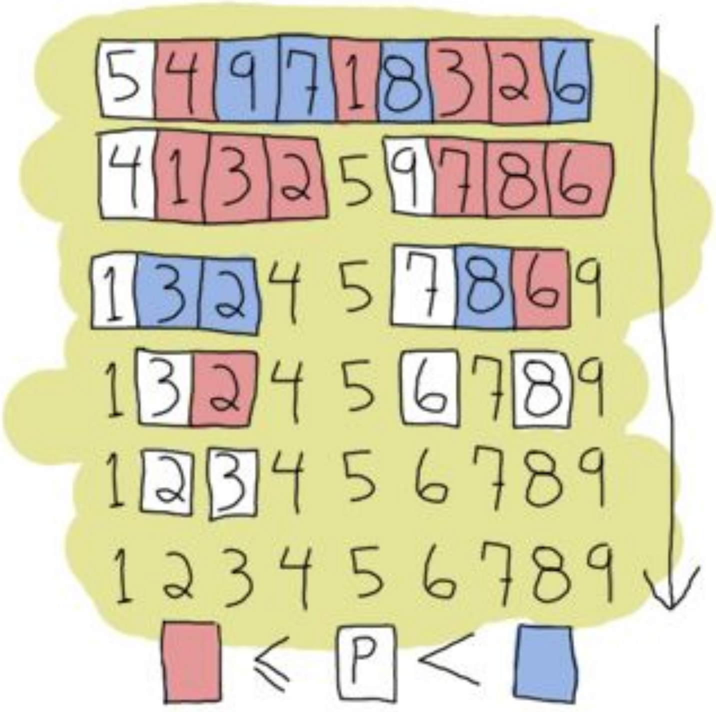
Worst: $O(n \log n)$

Best: $O(n \log n)$

Space complexity: $O(n)$

Sorting
Quicksort

UCLA Samueli Computer Science



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?

UCLA Samueli Computer Science

- 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.

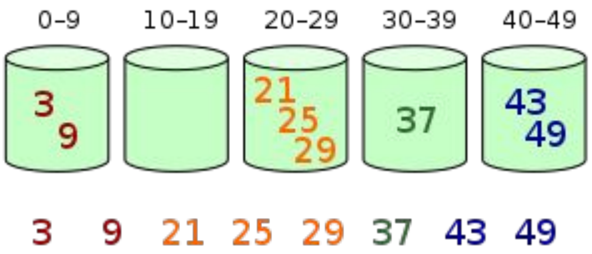
Question: What is the complexity of these sorting algorithms if you know the array is **reversed**? What if the array is **almost already sorted**?

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?

UCLA Samueli Computer Science

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

UCLA Samueli Computer Science

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?

Sorting Worksheet Question

UCLA Samueli Computer Science

- Worksheet questions 4, 6

Sorting

Worksheet (question 4)

UCLA Samueli Computer Science

Here are the elements of an array after each of the first few passes of a sorting algorithm discussed in class. Which sorting algorithm is it?

3 7 4 9 5 2 6 1

3 7 4 9 5 2 6 1

3 **7** 4 9 5 2 6 1

3 **4** 7 9 5 2 6 1

3 4 7 **9** 5 2 6 1

3 4 **5** 7 9 2 6 1

2 3 4 5 7 9 6 1

2 3 4 5 **6** 7 9 1

1 2 3 4 5 6 7 9

a. bubble sort
b. insertion sort
c. quicksort with the pivot always being chosen as the first element
d. quicksort with the pivot always being chosen as the last element

Sorting

Worksheet (question 6)

UCLA Samueli Computer Science

Given an array of *n* integers, where each integer is guaranteed to be between 1 and 100 (inclusive) and duplicates are allowed, write a function to sort the array in O(*n*) time.

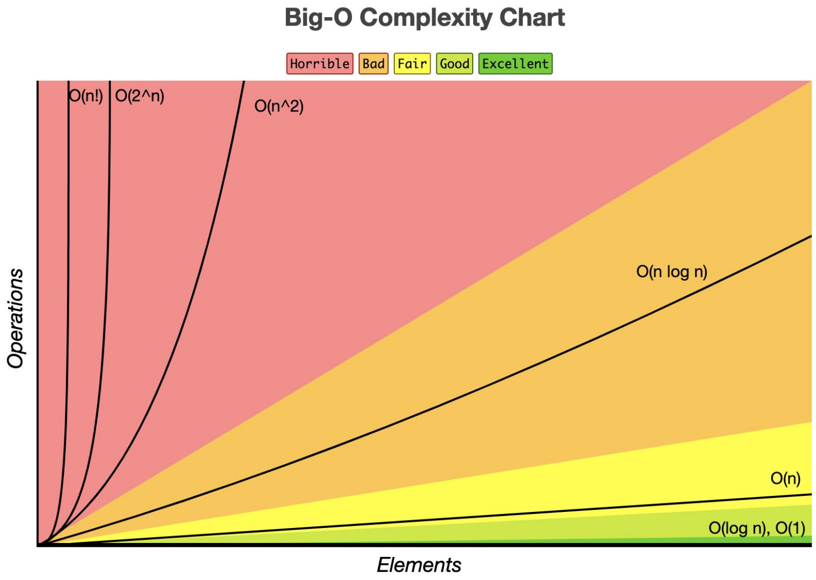
(Hint: the key to getting a sort faster than O(*n log n*) is to avoid directly comparing elements of the array!) (MV)

```
void sort(int a[], int n);
```

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)