UCLA Samueli Computer Science

CS32: Introduction to Computer Science II

Discussion Week 8

Yichao (Joey)

May 22, 2020

Credit: Carey Nachenberg, Junheng Hao

Outline Today

UCLA Samueli Computer Science

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

Announcements

- Project 3 is due 11:00 PM tonight.
- Homework 4 is due 11:00 PM Tuesday, May 26.

Algorithm Efficiency

Note: Complexity of a program

- 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

UCLA Samueli Computer Science

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.

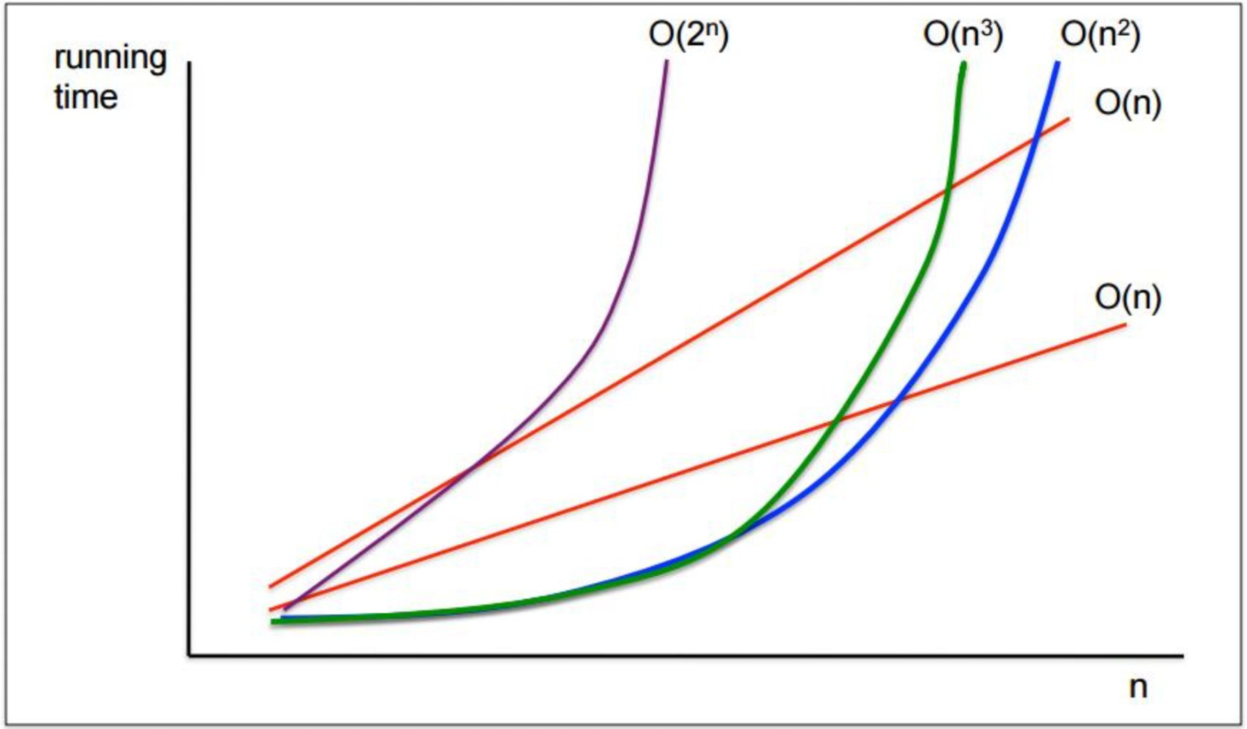
No need to memorize definitions. Example: if your program takes,

- 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

- 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 and Complexity

Worksheet Prob. #1

UCLA Samueli Computer Science

```
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;
}
```

What is the complexity?

Big-O and Complexity
Worksheet Prob. #1

UCLA Samueli Computer Science

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

      for(int k = 0; k < j*i; k+=j) { // 3
        if(rand() % 2 == 2)
            sum += 1;
      }
    }
  }
  return sum;
}
```

Analysis:

1: **O(n)**
2: **O(n)**

3: **O(n)**: i * j, but incrementing j every time

Now multiply all of them:
O(n^3)

Big-O and Complexity

Worksheet Prob. #2

UCLA Samueli Computer Science

```
int operationFoo(int n, int m, int w) {
  int res = 0;
  for (int i = 0; i < n; ++i) {
    for (int j = m; j > 0; j /= 2) {
      for (int jj = 0; jj < 50; jj++) {
        for (int k = w; k > 0; k -= 3) {
          res += i*j + k;
        }
      }
    }
  }
  return res;
}
```

What is the complexity?

Big-O and Complexity

Worksheet Prob. #2

UCLA Samueli Computer Science

```
int operationFoo(int n, int m, int w) {
  int res = 0;
  for (int i = 0; i < n; ++i) {              // 1
    for (int j = m; j > 0; j /= 2) {         // 2
      for (int jj = 0; jj < 50; jj++) {      // 3
        for (int k = w; k > 0; k -= 3) {     // 4
          res += i*j + k;
        }
      }
    }
  }
  return res;
}
```

Analysis:

1: **O(n)**
2: **O(logm)**
3: **O(1)**
4: **O(w)**

Now multiply all of them:
O(n * w * logm)

Big-O and Complexity

Worksheet Prob. #3

UCLA Samueli Computer Science

```
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;
}
```

What is the complexity?

Big-O and Complexity
Worksheet Prob. #3

UCLA Samueli Computer Science

```
int obfuscate(int a, int b) {
      vector<int> v;
      set<int> s;
      for (int i = 0; i < a; i++) {           // 1
            v.push_back(i);                   // 2
            s.insert(i);                      // 3
      }
      v.clear();                              // 4

      int total = 0;
      if (!s.empty()) {
            for (int x = a; x < b; x++) {     // 5
                  for (int y = b; y > 0; y--) { // 6
                        total += (x + y);
                  }
            }
      }
      return v.size() + s.size() + total;
}
```

Analysis:

1: **O(a)**
2: **O(1)**
3: **O(loga)**

4: **O(a)**

5: **O(b)**
6: **O(b)**

Combine:

Multiply 1-3:
O(aloga)
Add:
O(a)
Add:
O(b^2)

= O(aloga + a + b^2)

simplify:

= O(aloga + b^2)

Big-O and Complexity

Worksheet Prob. #6

UCLA Samueli Computer Science

```
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;
}
```

What is the complexity?

Big-O and Complexity

Worksheet Prob. #6

```
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) {  // 1
    if (n % i == 0) return false;
  }
  return true;
}
```

Analysis:

1: **O**(√n)

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

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

UCLA Samueli Computer Science

```
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

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: $O(n \log n)$

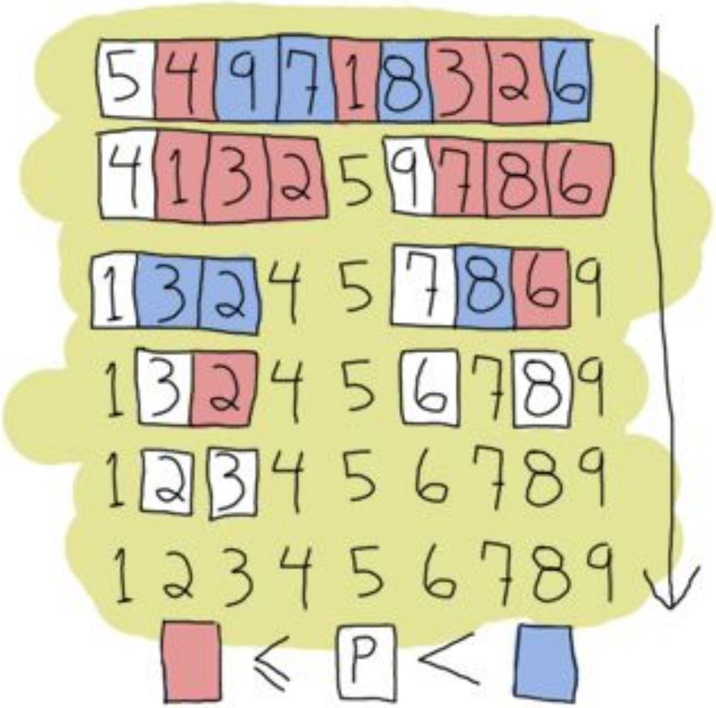
Worst: $O(n \log n)$

Best: $O(n \log n)$

Space complexity: $O(n)$

Sorting

Quicksort



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.

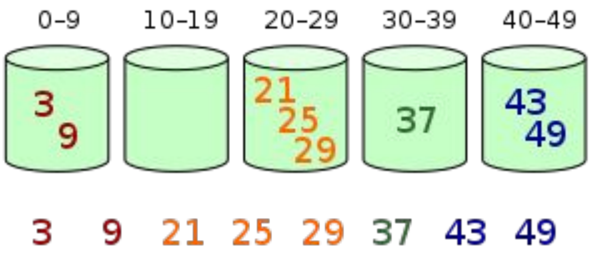
Question: What is the complexity of these sorting algorithms if you know the array is **reversed**? What if the array is **almost already sorted**?

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?

Sorting Worksheet Question: Prob #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: Prob #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: Prob #5

UCLA Samueli Computer Science

Given the following vectors of integers and sorting algorithms, write down what the vector will look like after 3 iterations or steps and whether it has been perfectly sorted.

i. `{45, 3, 21, 6, 8, 10, 12, 15}` insertion sort (1st step starts at comparing a[1])

ii. `{5, 1, 2, 4, 8}` bubble sort (Consider the array after 3 "passes" and after 3 "swaps." Do the results differ? Does the algorithm know when it's "done" in either case?)

iii. `{-4, 19, 8, 2, -44, 3, 1, 0}` quicksort (where pivot is always the last element)

Sorting Worksheet Question: Prob #5

UCLA Samueli Computer Science

`{45, 3, 21, 6, 8, 10, 12, 15}` insertion sort (1st step starts at comparing a[1])

{45, **3**, 21, 6, 8, 10, 12, 15}

{3, 45, **21**, 6, 8, 10, 12, 15}

{3, 21, 45, **6**, 8, 10, 12, 15}

{3, 6, 21, 45, **8**, 10, 12, 15} ...

Sorting Worksheet Question: Prob #5

UCLA Samueli Computer Science

`{5, 1, 2, 4, 8}` bubble sort (Consider the array after 3 “passes” and after 3 “swaps.” Do the results differ? Does the algorithm know when it’s “done” in either case?)

{**5**, **1**, 2, 4, 8}

{1, **5**, **2**, 4, 8}

{1, 2, **5**, **4**, 8}

{1, 2, 4, **5**, **8**}...

Sorting Worksheet Question: Prob #5

UCLA Samueli Computer Science

{-4, 19, 8, 2, -44, 3, 1, 0} quicksort (where pivot is always the last element)

{-4, 19, 8, 2, -44, 3, 1, **0**}

{-4, **-44**, 0, 2, 19, 3, 1, **8**}

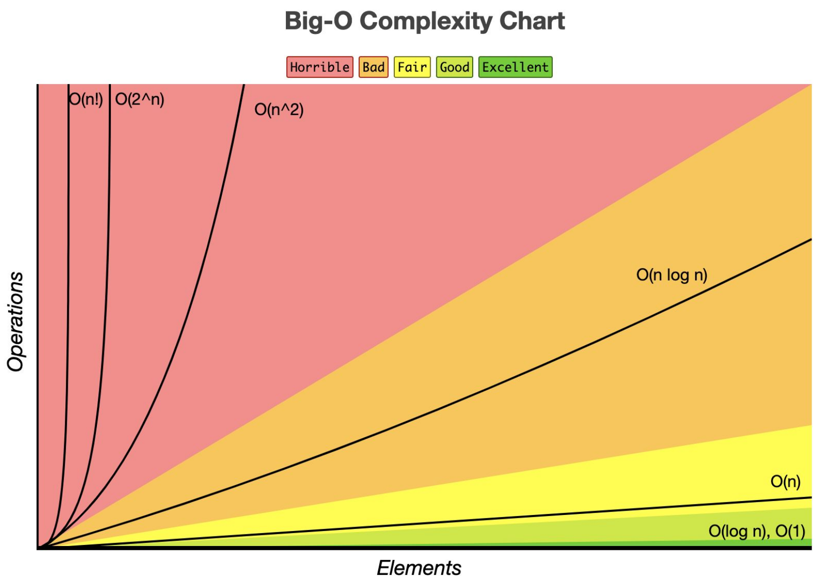
{-44, -4, 0, 2, 3, **1**, 8, **19**}

{-44, -4, 0, 1, **3**, 2, 8, **19**} ...

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)