Efficient Problem Solving:

Using the Right Data Structures and the Right Algorithms

Ira A. Fulton Schools of Engineering
Arizona State University

Part 1: Looking Back

- Why do we study data structures and algorithms?
- Why do we study asymptotic analysis?
- Problem Solving: Good/Better/Best Algorithms.
- How to verify your implementation?
- Implement algorithms the correct/wrong way?

Why Study Data Structures and Algorithms?

- We want to solve a given problem efficiently!

- Example: **Using the right algorithm**

- ***If the goal is to sort an almost sorted array with efficient worst-case time complexity*, use insertion sort.**

- This is decided by the inversion number of the input sequence.

Why Study Data Structures and Algorithms?

- **We want to solve a given problem efficiently!**

- **Example: Using the right algorithm**

- ***If the goal is to sort a random array with efficient <u>worst-case</u> time complexity, <u>use merge-sort</u>.***

- ***If the goal is to sort a random array with efficient <u>average-case</u> time complexity, <u>use quicksort</u>.***

Why Study Data Structures and Algorithms?

- We want to solve a given problem efficiently!
- Example: **Using the right data structure**
- *If the goal is to support insertion/search/deletion with good worst-case time complexity*, <u>use the red-black tree</u>.
- What about a linked list?
- What about a binary search tree?
- What about a hash table?

Why Study Data Structures and Algorithms?

- We want to solve a given problem efficiently!
- Example: **Using the right data structure**
- *If the goal is to support insertion/delete-min/decreasekey with good worst-case time complexity,* use red-black tree or min-heap*.*
- Min-heap is very easy to implement.
- RBT is much harder to implement.

Why Study Asymptotic Analysis?

- It abstracts the running time in a simple way!
- $O(\log(n))$ running time
- $O(n)$ running time
- $O(n \log(n))$ running time
- $O(n^2)$ running time
- $O(n^3)$ running time
- $O(2^n)$ running time

The Goal Is to Solve a Problem Efficiently

- Suppose you have ***several algorithms for solving the same problem***. The algorithms have worst-cast time complexities given in the following:
- A: O(log(n)) running time
- B: O(n) running time
- C: O(n log(n)) running time
- D: O(n^2) running time
- E: O(n^3) running time
- F: O(2^n) running time
- **Which algorithm is more appealing?**

The Goal Is to Solve a Problem Efficiently

- Suppose you have ***several algorithms for solving the same problem***. The algorithms have worst-cast time complexities given in the following:
- A: $O(\log(n))$ running time
- B: $O(n)$ running time
- C: $O(n \log(n))$ running time
- D: $O(n^2)$ running time
- E: $O(n^3)$ running time
- F: $O(2^n)$ running time
- **Algorithm A is more appealing.**

Ira A. Fulton Schools of Engineering

Arizona State University

Part 2: Efficient Problem Solving

- Why do we study data structures and algorithms?
- Why do we study asymptotic analysis?
- Problem Solving: Good/Better/Best Algorithms.
- How to verify your implementation?
- Implement algorithms the correct/wrong way?

Good/Better/Best Algorithms for a Problem

- We are given an unsorted array *A* of size *n*. Let *k* be an integer between 1 and *n*. We need to *find the k-th smallest element of A*.
- What algorithms can you come up with?
- What are the corresponding worst-case time complexities?

Good/Better/Best Algorithms for a Problem

- We are given an unsorted array *A* of size *n*. Let *k* be an integer between 1 and *n*. We need to *find the k-th smallest element of A*.
- What algorithms can you come up with?
- Approach A: Find the smallest element and delete it. Repeat this process *k* times.
- Time complexity is $O(k\,n)$.

Good/Better/Best Algorithms for a Problem

- We are given an unsorted array *A* of size *n*. Let *k* be an integer between 1 and *n*. We need to *find the k-th smallest element of A*.
- What algorithms can you come up with?
- Approach B: Sort *A* using mergesort. Return *A*[*k*].
- Time complexity is $O(n \log n)$.
- This is better than Approach A.

Good/Better/Best Algorithms for a Problem

- We are given an unsorted array *A* of size *n*. Let *k* be an integer between 1 and *n*. We need to *find the k-th smallest element of A*.
- What algorithms can you come up with?
- Approach C: Build a min-heap. Delete-Min *k* times.
- Time complexity is $\mathbf{O}(\mathbf{n}) + \mathbf{O}(\mathbf{k} \log \mathbf{n}) = \mathbf{O}(\mathbf{n} + \mathbf{k} \log \mathbf{n})$.
- This is better than Approach B.

Good/Better/Best Algorithms for a Problem

- We are given an unsorted array *A* of size *n*. Let *k* be an integer between 1 and *n*. We need to *find the k-th smallest element of A*.
- What algorithms can you come up with?
- Approach D: Use linear time selection.
- Time complexity is $O(n)$.
- This is better than Approach C.

Good/Better/Best Algorithms for a Problem

- We are given an unsorted array *A* of size *n*. Let *k* be an integer between 1 and *n*. We need to *find the k-th smallest element of A*.
- What algorithms can you come up with?
- Can we do better than Approach D?
- No. There are n elements in unsorted order. Any correct algorithm needs to check all n elements. This requires a time complexity of $\Omega(n)$.
- D is the best approach (asymptotically).

Good/Better/Best Algorithms for a Problem

- We are given an unsorted array *A* of size *n*. Let *k* be an integer between 1 and *n*. We need to *return the k smallest elements of A in sorted order.*
- What algorithms can you come up with?
- What are the corresponding worst-case time complexities?

Good/Better/Best Algorithms for a Problem

- We are given an unsorted array *A* of size *n*. Let *k* be an integer between 1 and *n*. We need to *return the k smallest elements of A in sorted order.*

- What algorithms can you come up with?

- Approach A: Find the smallest element and delete it. Repeat this process *k* times.

- Time complexity is $O(k\,n)$.

Good/Better/Best Algorithms for a Problem

- We are given an unsorted array *A* of size *n*. Let *k* be an integer between 1 and *n*. We need to *return the k smallest elements of A in sorted order.*
- What algorithms can you come up with?
- Approach B: Sort *A* using mergesort. Return *A*[*1*] through A[*k*].
- Time complexity is $O(n \log n + k)$.
- This is better than Approach A.

Good/Better/Best Algorithms for a Problem

- We are given an unsorted array *A* of size *n*. Let *k* be an integer between 1 and *n*. We need to *return the k smallest elements of A in sorted order.*
- What algorithms can you come up with?
- Approach C: Build a min-heap. Delete-Min *k* times.
- Time complexity is $O(n)+O(k \log (n))=O(n+k \log n)$).
- This is better than Approach B.

Good/Better/Best Algorithms for a Problem

- We are given an unsorted array *A* of size *n*. Let *k* be an integer between 1 and *n*. We need to *return the k smallest elements of A in sorted order.*
- What algorithms can you come up with?
- Approach D: Use linear time selection to return the k smallest elements. Then sort them using mergesort.
- Time complexity is $O(n)+O(k \log (k))=O(n+k \log k)$.
- This is better than Approach C.

Good/Better/Best Algorithms for a Problem

- We are given an unsorted array *A* of size *n*. Let *k* be an integer between 1 and *n*. We need to *return the k smallest elements of A in sorted order.*

- Can we do better than Approach D?

- No. There are n elements in unsorted order. Any correct algorithm needs to check all n elements. This requires a time complexity of $\Omega(n)$. Any algorithm needs to sort at least k elements. This requires a time complexity of $\Omega(k \log k)$. Therefore any algorithm requires $\Omega(n + k \log k)$time.

- D is the best approach (asymptotically).

ASU Ira A. Fulton Schools of Engineering

Arizona State University

Part 3: Algorithm Implementation

- Why do we study data structures and algorithms?
- Why do we study asymptotic analysis?
- Problem Solving: Good/Better/Best Algorithms.
- How to verify your implementation?
- Implement algorithms the correct/wrong way?

How to Verify Your Implementation

- Suppose you are implementing an algorithm for solving a given problem. How do you know that you have implemented your algorithm correctly?

- You can test your algorithm with carefully designed test cases (instances).

- Suppose that you have implemented an algorithm with worst-case time complexity $O(n)$.

- You can run your algorithm on test cases with n=1, 2, 4, 8, 16, 32, 64, …, and measure the running time for each test case.

How to Verify Your Implementation

- You can run your algorithm on test cases with n=1, 2, 4, 8, 16, 32, 64, …, and measure the running time for each test case.
- You can compute the ratio T(2n)/T(n) for each of these values of n, where T(n) is the measured running time for the particular test case.
- What do you expect to see T(2n)/T(n) as n goes to infinity?
- Suppose it is bounded by 2, what do you learn?
- Suppose it approaches 3 what do you learn?

How to Verify Your Implementation

- What happens if the algorithm has worst-case time complexity $O(n^2)$?

- What happens if the algorithm has worst-case time complexity $O(\log(n))$?

- What happens if the algorithm has worst-case time complexity $O(n \log(n))$?

How to Implement Algorithms Wrongly?

- Suppose that you are implementing Dijkstra’s shortest path algorithm using a binary heap. The worst-case running time should be O(m+n)log(n)).
- You can generate test cases with m=5n, and n doubles each time.
- If the running time doubles each time, is it OK?
- If the running time quadruples each time, is it OK?

How to Implement Algorithms Wrongly?

- Suppose that you are implementing Dijkstra's shortest path algorithm using a binary heap. The worst-case running time should be O(m+n)log(n)).
- How can you implement it with a worst-case time complexity O(n(m+n))?
- If the running time quadruples each time you double the size of the instance, what do you learn?
- How can this happen???

More on Data Structures

- **You need to write a program to manipulate polynomials: evaluation, addition, subtraction, multiplication.**
- **Which data structure should you use?**
  - Array
  - Singularly linked list
  - What happens if the polynomial is $x^{103} + 3$?

More on Algorithms

- You are given a sequence of integers. Your task is to find the distinct integers in the sequence, in sorted order.
- Can you use algorithms learned in class to solve this problem?
- $100, 2, 24, 2, 34, 24, 60, 50, 60$

Ira A. Fulton Schools of
Engineering
Arizona State University