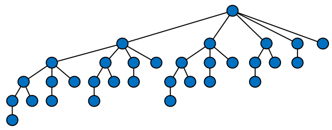
Lecture 1. Course Overview, Motivation, Fundamental Concepts

CpSc 8400: Algorithms and Data Structures
Brian C. Dean



School of Computing
Clemson University
Spring, 2016

Introductions

- **Instructor:** Dr. Brian C. Dean
  - Ph.D. from MIT in 2005.
  - **Email:** bcdean@clemson.edu
  - **Office:** McAdams 205
  - **Office Hours:** See syllabus.
  - **Research Interests:** Algorithms (optimization, data mining and analytics, randomization, data structures, heuristics, matching, geometry) and their applications (medical informatics, networking, scheduling).
  - **Educational Interests** in algorithmic CS education and problem-solving at the high-school level; director of USA Computing Olympiad (usaco.org).

2

Course Overview

- This course provides a fun, fast-paced, modern theory-oriented study of algorithms.
- We will learn:
  - Algorithm design techniques
    Divide and conquer, greedy algorithms, dynamic programming, iterative refinement, randomized incremental construction, …
  - Common algorithms for common algorithmic problems
    Sorting and selection, graph problems, FFTs, optimization, …
  - Mathematical tools for algorithm analysis
    Probability theory, amortized analysis, recurrences, …
  - Interesting algorithmic subfields in which you may want to pursue further research/study.
    Computational geometry, cryptography, approximation and randomized algorithms, data structures, bioinformatics, …

Why Learn Algorithms?...

- Algorithms are the heart and sole of computing!
- Algorithmic computing now plays a key role in nearly everything – proficiency opens many doors.

- Algorithmic proficiency differentiates a true “computer scientist” from a run-of-the-mill “programmer”, and provides the foundation for a long-term career in computing that can thrive as technology changes.
- Theory meets practice. Algorithms have motivated the development of some truly elegant theoretical results in mathematics, for those who appreciate mathematics.
- Algorithms are fun!

Programming ≠ Algorithmic Problem Solving

- "Computer Science is no more about computers than astronomy is about telescopes" – fokelore, sometimes attributed to E. Dijkstra.
- Programming and software engineering are certainly an important part of most computing projects and careers.
- Problem-solving skills and programming skills are different, but re-inforce each-other.
- Both are crucial for success as a computing expert.

5

Course Details

- **Prerequisites**
  - A reasonable amount of mathematical maturity.
  - Enthusiasm, willingness to challenge yourself and ask questions, and a good work ethic!
- **Course Materials**
  - *Algorithms Explained*, currently being written by the instructor – portions may be made available electronically on Blackboard (don't redistribute!)
  - Lecture slides and videos to appear on blackboard.
  - I can suggest supplemental reading, if interested.
  - I may also post prominent research papers relevant to the course content.

6

Assignments and Grades

- **Homework (35%)**
  - Typically focused on mathematical analysis of algorithms, not implementation.
  - Solutions must be typeset and submitted as a PDF file using handin.cs.clemson.edu by email *before the start of class on the day they are due.*
- **2 Quizzes (2 x 20%) and a Final (25%)**
  - All quizzes/exams are cumulative.
- Appropriate letter grade cutoffs set by instructor at end of semester.

7

Course Conduct

- **Academic Integrity**:
  - Do not cheat.
  - Do not plagiarize (pass off the work of others as your own without appropriate attribution).
- **Collaboration:**
  - Highly encouraged, but each student should write up final solutions independently. Please list your collaborators.
  - Do not consult homework solutions from previous semesters, and do not use the web for anything but general reference.
- **Feedback**:
  - Please feel welcome to ask for feedback at any time. The instructor always appreciates constructive feedback.

8

A Good Algorithm (or Data Structure)…

- Always terminates and produces **correct** output.
  - A "close enough" answer is sometimes fine.
  - Some types of randomized algorithms can fail, but only with miniscule probability.
- Makes **efficient** use of computational resources.
  - Minimizes running time, memory usage, processors, bandwidth, power consumed, heat produced.
- Is **simple** to describe, understand, analyze, implement, and debug.

9

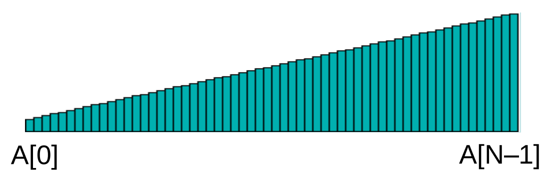
Example: Searching an Array

- **Linear** search: runs in N "steps" in the worst case.

```
for i = 0..N-1:
   if target = A[i], found it!
```

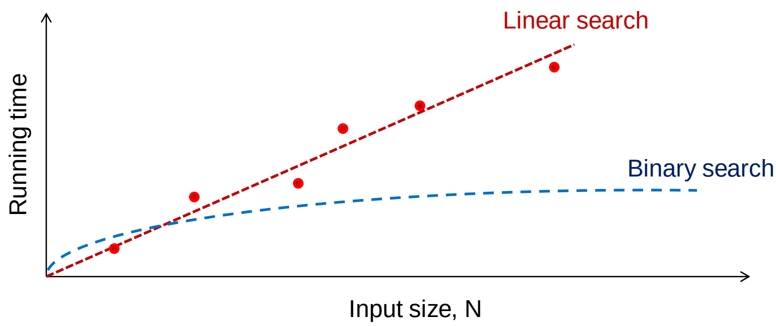
- **Binary** search: $\leq \log_2 N$ "steps" in worst case. (requires our array to be sorted).

```
if target = middle element, found it!
else recursively search first or second half
  array, as appropriate.
```



10

Empirical Performance Testing



- Choose inputs carefully, since often some inputs are much easier than others.
- Do you want to measure "average case" or "worst-case" performance…?

Asymptotic Analysis

- **Linear** search: O(N) time.
- **Binary** search: O(log N) time.

- O(f(N)) means "upper-bounded by a constant times f(N) as N grows large".
- Provides an asymptotic upper bound on running time, where constant factors and lower-order terms don't matter.
- Captures what usually matters most about algorithm performance: how worst-case running time scales with input size.
- However, this can lose important information…
  (e.g., sequential vs. non-sequential linear search)

Asymptotic Notation

- O() provides an asymptotic upper bound.
- Ω() provides an asymptotic lower bound.
- Θ() means both a lower and upper bound.

(think of these as "≤", "≥", and "=")

Usage examples:

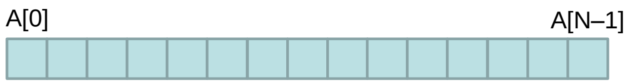
- "The running time of our algorithm is $O(n^2)$."
- "The worst-case running time of our algorithm is $\Theta(n^2)$."
- "This algorithm uses $\Omega(n^2)$ memory".
- "$17n^2 - 5n + 200 = \Theta(n^2)$"
- "Consider the polynomial $5x^{10} - 3x^9 + O(x^8)$."

Running Times

- We almost always focus on **worst case** running times. Why?
- Common running times:
  - Constant: $O(1)$
  - Logarithmic: $O(\log n)$
  - Linear: $O(n)$
  - Polynomial: $O(n \log n)$, $O(n^2)$, $O(n^3)$, $O(n^{100})$, …
  - Exponential: $O(2^n)$, $O(3^n)$, …
  - Worse than exponential: $O(n!)$, $O(n^n)$.
- Logs: base doesn't matter in O(), as long as not in an exponent. By log n we usually mean $\log_2 n$.

Fundamental Data Structures

Arrays



- Retrieve or modify any element in O(1) time.
- Insert or delete in middle of list: O(N) time. ☹
- Insert or delete from ends: O(1) time
  - Be careful not to run over end of allocated memory

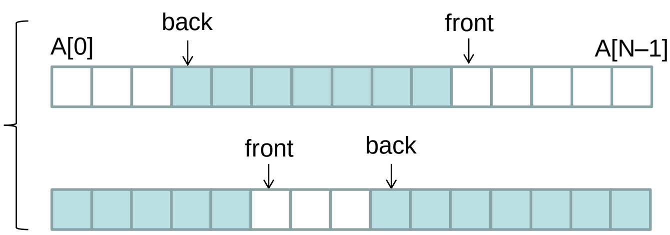
Linked Lists

(sometimes doubly linked, or ending with a sentinel instead of NULL)

- Seek to any position in list: O(N) time. ☹
- Then insert or delete element: O(1) time.
- Insert or delete from ends: O(1) time.

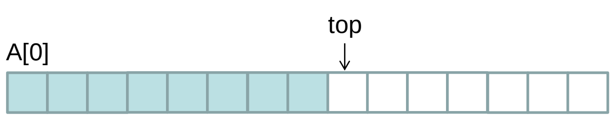
Fundamental Data Structures

Queues



- First-In, First-Out (FIFO).
- O(1) enqueue & dequeue, implemented using arrays or linked lists (often implemented using <u>circular</u> arrays).

Stacks



- Last-In, First-Out (LIFO).
- O(1) push & pop, implemented using arrays or linked lists.

An Important Distinction…

Specification of a data structure in terms of the operations it needs to support.

(sometimes called an *abstract data type)*

A concrete approach for **implementation** of the data structure that fulfills these requirements.

17

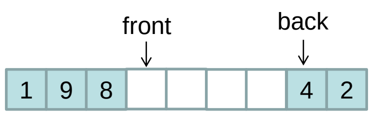
Example: Queues

Abstract data type: queue

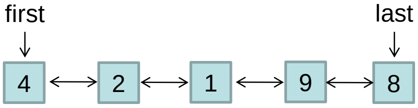
Must support these operations:

- *Insert*(k) a new key *k* into the structure.
- *Remove* the least-recently-inserted key from the structure.
  (so FIFO behavior)

Choices for concrete implementation:



Circular array



(Doubly) linked list

18

Enforcing Abstraction in Code

Abstract data type: queue

Concrete implementation: queue.cpp

```
queue.h:

class Queue {
  private:
     int *A;
     int front, back, N;

  public:
     Queue();
     ~Queue();
     void insert(int key);
     int remove(void);
};
```

19

Enforcing Abstraction in Code

Abstract data type: queue

Concrete implementation: queue.cpp

```
Queue q;
q.insert(6);
x = q.remove();
```

```
queue.h:

class Queue {
  private:
     int *A;
     int front, back, N;

  public:
     Queue();
     ~Queue();
     void insert(i
     int remove(vo
};
```

```
int Queue::remove(void)
{
   int result = A[back];
   back = (back+1) % N;
   return result;
}
```

20