CS 409: Data Structures and Algorithms

Instructor: Joe Halpern

There’s a handout that tells you everything you need to know for now about the course structure:

- TAs and consultants
- Office hours
- Grading
- Text
- How to find out more
  - Check the course web site and newsgroup!
What’s It All About?

In a nutshell:

• designing and analyzing algorithms for solving computational problems.

Such algorithms deal with data.

• That means we need good data structures
  ◦ ways of storing and accessing the data to make the algorithm efficient

We consider some key data structures and efficient ways of implementing them:

• stacks, queues, linked lists, hash tables, binary search trees, binomial heaps, . . .

Data structures are the building blocks for many algorithms, so it’s worth optimizing them.

We apply data structures to important problems (graph algorithms, sorting, prioritizing, string matching, . . .)

Programming is an important component of this course!
Required Background

I will assume you know basic properties of:

- Basic functions ($|x|$, $\lg n$, $2^n$, $n!$) (Ch. 2.2)
- Summations (Ch. 3)
  - Summation notation: $\sum_{k=1}^{n} k$
  - Technique for bounding sums: $\sum_{k=1}^{n} k < n^2$.
    * including approximation by integrals
- Sets, relations, functions, graphs, trees (Ch. 5)
  - manipulating intersection, union, complement
  - reflexive, symmetric, transitive relations
  - injection, bijection, one-to-one correspondence
  - degree, connected component, (un)directed graph
  - binary trees
- Counting and Probability (Chapter 6.1-6.4)
  - choosing $k$ out of $n$
  - axioms of probability
  - conditional probability and independence
  - expected value
  - binomial distribution
A word to the wise:

- Review this material NOW!
- Make sure you can do all the review problems
- See me or someone else on staff if you can’t

CS211 is a prerequisite for the course.

- There is some overlap in topics covered
  - I will cover some topics in greater depth (e.g., hashing), and briefly review others (e.g., breadth-first and depth-first search)

CS280 is also a prerequisite:

- You need to know how to do induction
- We will cover Minimum Spanning Trees and Dijkstra’s algorithm (sometimes done in CS280)
An Example: Sorting

Given: A sequence of \( n \) numbers \( \langle a_1, \ldots, a_n \rangle \)

Output: A permutation (reordering) \( \langle a'_1, \ldots, a'_n \rangle \)
such that \( a'_1 \leq \ldots \leq a'_n \).

A naive (but common) approach: Insertion Sort.

- Assume we’ve sorted the first \( k \) elements; put the
  \((k + 1)\)st element into the right place by comparing it until we find the right place for it.
Suppose $A = \langle a_1, \ldots, a_n \rangle$ is an array to be sorted.

- $A[i] = a_i$

**INSERTION-SORT($A$)**

1. for $j \leftarrow 2$ to $\text{length}[A]$
2. do $key \leftarrow A[j]$
4. $i \leftarrow j - 1$
5. while $i > 0$ and $A[i] > key$
6. do $A[i+1] \leftarrow A[i]$
7. $i \leftarrow i - 1$
8. $A[i+1] \leftarrow key$

Suppose $A = \langle 5, 2, 4, 6, 1 \rangle$.

2 5 4 6 1 $j = 2; key = 2$

2 5 4 6 1 $j = 3; key = 4$

2 4 5 6 1 $j = 4; key = 6$

2 4 5 6 1 $j = 5; key = 1$

1 2 4 5 6
Running Time of Insertion Sort

Assume step $i$ in the algorithm “costs” $c_i$
Let $t_j$ be number of times $j$th inner loop is executed

- $t_j$ depends on the input $A$
- best case: $t_j = 1$
- worst case: $t_j = j$

1. for $j \leftarrow 2$ to length[$A$]  
   do $key \leftarrow A[j]$  
   ▷ Insert $A[j]$ into sorted  
   sequence $A[1..j-1]$.  
   i $\leftarrow j - 1$  
   while $i > 0$ and $A[i] > key$  
   do $A[i+1] \leftarrow A[i]$  
   i $\leftarrow i - 1$  
   $A[i+1] \leftarrow key$

Let $T(A)$ be the running time on input $A$:

$$T(A) = c_1 n + (c_2 + c_4 + c_8 - c_6 - c_7)(n-1) + (c_5 + c_6 + c_7) \sum_{j=1}^{n-1} t_j$$
\[ T(A) = c_1 n + (c_2 + c_4 + c_5 + c_7)(n - 1) + (c_5 + c_6 + c_7) \sum_{j=1}^{n-1} t_j \]

Best case: \( t_j = 1 \)

\[
T(A) = c_1 n + (c_2 + c_4 + c_5 + c_8)(n - 1) \\
= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)
\]

Worst case: \( t_j = j \)

\[
T(A) = (c_5 + c_6 + c_7) \sum_{j=2}^{n-1} j + \cdots \\
= (c_5 + c_6 + c_7)\left(\frac{n(n+1)}{2} - 1\right) + \cdots
\]

- Quadratic is OK if \( n \) is 5, 10, 100.
- But what if \( n = 1,000,000 \)?

Average case:

- When we insert \( A[j] \), roughly half the elements in \( A[1..j-1] \) will be greater than \( A[j] \), and half will be less.
  
  \( \circ t_j \sim j/2 \)

- Average case is still quadratic
Designing Algorithms

Insertion sort uses an *incremental* approach:

- We insert a single element into $A[1..j]$.

We can do better using “divide-and-conquer”

- *Divide* each problem into smaller subproblems (typically about half the size of the original)
- *Conquer* each subproblem
- *Combine* the solutions
Merge Sort

Merge sort is a sorting algorithm that uses divide and conquer.

- **Divide** the sequence to be sorted into two subsequences of size \( n/2 \)
- **Conquer**: sort the two subsequence (recursively)
- **Combine**: merge the resulting sequences

Suppose \( A \) is an array of length \( n \), \( 1 \leq p \leq r \leq n \):

\[
\text{MERGE-SORT}(A, p, r)
\]

1. **if** \( p < r \)
2. **then** \( q \leftarrow \lfloor (p + r)/2 \rfloor \)
3. \( \text{MERGE-SORT}(A, p, q) \)
4. \( \text{MERGE-SORT}(A, q + 1, r) \)
5. \( \text{MERGE}(A, p, q, r) \)
Analysis of Merge-Sort

\textsc{merge-sort}(A, p, r)

1 \textbf{if} \ p < r \\
2 \textbf{then} \ q \leftarrow \lfloor (p + r)/2 \rfloor \\
3 \textsc{merge-sort}(A, p, q) \\
4 \textsc{merge-sort}(A, q + 1, r) \\
5 \textsc{merge}(A, p, q, r)

If \( m = r - p + 1 \), define

\begin{itemize}
  \item \( T(m) \) = the worst-case time for \textsc{merge-sort}(A, p, r) \\
  \item \( U(m) \) = be the worst-case time for \textsc{merge}(A, p, q, r)
\end{itemize}

\[
T(m) = \begin{cases} 
  c_1 & \text{if } m = 1 \\
  2T(\lfloor m/2 \rfloor) + U(m) & \text{if } m > 1
\end{cases}
\]

Not hard to show that \( U(m) = \Theta(m) \)

It follows that \( T(n) = \Theta(n \lg n) \) (\( \lg = \log_2 \))

\begin{itemize}
  \item This is much better than \( \Theta(n^2) \) for large \( n \)!
\end{itemize}
What We’ll Cover

• Data structures (Chapters 7, 11–14, 22)
  ○ Stacks, queues, linked lists
  ○ Hashing
  ○ Binary search trees, red-black trees (maybe)
  ○ Heaps

• Algorithm design techniques (Chapters 16–18)
  ○ dynamic programming
  ○ greedy algorithms
  ○ amortized analysis

• Graph algorithms (Chapters 23–25, 27)
  ○ Strongly connected components
  ○ Minimum spanning tree
  ○ Shortest paths (Dijkstra’s algorithm)
  ○ Maximum flow
• NP-completeness (Chapter 36)
• If there’s time:
  o String Matching (Chapter 34)
  o The RSA cryptosystem (Chapter 33.7)

This week:
• Technical background
  o Asymptotic growth (big O,\Theta,\Omega) (Chapter 2.1)
  o Recurrences (Chapter 4.1, 4.3)
  o A little probability
Asymptotic Notation

We measure the efficiency of an algorithm as a function of the input size.

- Want to describe the efficiency succinctly

Some useful notation:

- $T(n) = O(g(n))/\Omega(g(n))/\Theta(g(n))$ if there is a constant $c$ such that $cg(n)$ is asymptotically an upper/lower/tight bound for $T(n)$
  - $T(n) = \Theta(g(n))$ iff $T(n) = O(g(n))$ and $T(n) = \Omega(g(n))$.

We won’t cover $o(g(n))$, $\omega(g(n))$.

Formally, $\Theta(g(n))$, $O(g(n))$, and $\Omega(g(n))$ are sets of functions:

- $\Theta(g(n)) = \{f(n) : \exists c_1, c_2 > 0, n_0 \quad (c_1g(n) \leq f(n) \leq c_2g(n) \text{ for } n \geq n_0)\}$
- $O(g(n)) = \{f(n) : \exists c_2 > 0, n_0 (f(n) \leq c_2g(n) \text{ for } n \geq n_0)\}$
- $\Omega(g(n)) = \{f(n) : \exists c_1 > 0, n_0 (c_1g(n) \leq f(n) \text{ for } n \geq n_0)\}$
Example: $2n^2 + 3n + 1 = \Theta(n^2) = 2n^2 + \Theta(n)$.

- Clearly $2n^2 \leq 2n^2 + 3n + 1 \leq 3n^2$ for $n \geq 4$
  - Since $n^2 \geq 3n + 1$ if $n \geq 4$
- Also $2n^2 + 3n \leq 2n^2 + 3n + 1 \leq 2n^2 + 4n$

More generally, if $a > 0$,

$$an^2 + bn + c = \Theta(n^2) = an^2 + \Theta(n)$$

Example: $6n^3 \neq \Theta(n^2)$; $6n^3 = \Omega(n^2)$.

- Really should say $6n^3 \notin \Theta(n^2)$; $6n^3 \in \Omega(n^2)$. 
The $O$, $\Theta$, $\Omega$ notation ignores constants.

- The constants depend on the machine, details of implementation
- Improving the constants is good but ...
- Improving $\Theta(\ldots)$ is better
  - It gives us a better indication of how the problem scales
  - $\Theta(\lg n) < \Theta(\lg^2 n) < \Theta(n) < \Theta(n \lg n) < \Theta(n^2) < \Theta(2^n)$
Recurrences

A *recurrence* is a relation that describes a function in terms of its value on smaller inputs.

\[
T(m) = \begin{cases} 
  c_1 & \text{if } m = 1 \\
  2T(\lfloor m/2 \rfloor) + c_2m & \text{if } m > 1
\end{cases}
\]

Recurrences arise frequently when computing the running time of a recursive algorithm.

- Often stated without \( \lfloor \rfloor, \lceil \rceil \)

How do we solve them?

- Guess and verify by induction (substitution method)
- Apply master theorem

We won’t cover iteration method.
The Substitution Method

\[ T(n) = \begin{cases} 
  c_1 & \text{if } m = 1 \\
  2T(n/2) + n & \text{if } m > 1 
\end{cases} \]

Guess \( T(n) \leq cn \lg(n) \) (for some \( c \):)
Verify:

\[
T(n) = 2T(n/2) + n \\
\leq 2c(n/2 \lg n/2) + n \\
= cn \lg(n/2) + n \\
= cn \lg n + n - cn \lg 2 \\
= cn \lg n + (1 - c)n \\
\leq cn \lg n \quad (\text{if } c \geq 1)
\]

What about \( T(1) \)?

- \( c \lg 1 = 0 \)

All we need is \( T(n) \leq cn \lg n \) for \( n \) sufficiently large.
- E.g. \( T(2) = 2c_1 + 2 \). Choose \( c = c_1 + 1 \).

- Then \( T(2) \leq 2c \lg 2 = 2c_1 + 2 \)

A formal proof that \( T(n) \leq cn \lg n \) for \( n \geq 2 \) proceeds by induction.

- YOU ALL SHOULD KNOW HOW TO DO INDUCTION PROOFS!
The Master Method

**Theorem:** Let $a, b \geq 1$ and suppose

$$T(n) = aT(n/b) + f(n).$$

- Can replace $n/b$ by $\lceil n/b \rceil$ or $\lfloor n/b \rfloor$.

1. If $f(n) = O(n^{\log_b(a)-\epsilon})$ for some $\epsilon > 0$
   then $T(n) = \Theta(n^{\log_b(a)})$.

2. If $f(n) = \Theta(n^{\log_b(a)})$ then $T(n) = \Theta(n^{\log_b(a)} \lg n)$.

3. If $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ for some $\epsilon > 0$ and
   \[af(n/b) \leq cf(n)\text{ for some } c > 1\]
   then $T(n) = \Theta(f(n))$.

In all three cases we compare $f(n)$ with $n^{\log_b(a)}$.

- The larger function dominates

Roughly:

- if $f(n) \ll n^{\log_b(a)}$, then $T(n) = \Theta(n^{\log_b(a)})$
- if $f(n) \sim n^{\log_b(a)}$, then $T(n) = \Theta(n^{\log_b(a)} \lg n)$
- if $f(n) \gg n^{\log_b(a)}$, then $T(n) = \Theta(f(n))$
\[ T(n) = aT(n/b) + f(n): \]

1. If \( f(n) = O(n^{\log_b(a) - \epsilon}) \) for some \( \epsilon > 0 \)
   then \( T(n) = \Theta(n^{\log_b(a)}) \).

2. If \( f(n) = \Theta(n^{\log_b(a)}) \) then \( T(n) = \Theta(n^{\log_b(a)} \lg n) \).

3. If \( f(n) = \Omega(n^{\log_b(a) + \epsilon}) \) for some \( \epsilon > 0 \) and
   \( af(n/b) \leq cf(n) \) for some \( c > 1 \)
   then \( T(n) = \Theta(f(n)) \).

Comments:

- \( f(n) \ll n^{\log_b(a)} \) means there is some polynomial \( n^\epsilon \) such that \( f(n) \leq cn^{\log_b(a)}/n^\epsilon \)
- Third case has a regularity condition: \( af(n/b) \leq cf(n) \)
- Not all cases covered by theorem—but it’s still very useful
\[ T(n) = aT(n/b) + f(n): \]

1. If \( f(n) = O(n^{\log_b(a) - \epsilon}) \) for some \( \epsilon > 0 \)
then \( T(n) = \Theta(n^{\log_b(a)}) \).

2. If \( f(n) = \Theta(n^{\log_b(a)}) \) then \( T(n) = \Theta(n^{\log_b(a) \lg n}) \).

3. If \( f(n) = \Omega(n^{\log_b(a) + \epsilon}) \) for some \( \epsilon > 0 \) and
\[ af(n/b) \leq cf(n) \text{ for some } c > 1 \]
then \( T(n) = \Theta(f(n)) \).

Examples:

- \( T(n) = 9T(n/3) + n \)
  - \( a = 9, b = 3, f(n) = n \)
  - \( n^{\log_3(9)} = n^2 \), so \( f(n) = O(n^{\log_3(9) - \epsilon}) \)
  - \( T(n) = \Theta(n^2) \)

- \( T(n) = T(2n/3) + 1 \)
  - \( a = 1, b = 3/2, f(n) = 1 = n^{\log_{3/2}(1)} \)
  - \( T(n) = \Theta(\lg n) \)

- \( T(n) = 3T(n/4) + n \lg n \)
  - \( a = 3, b = 4, f(n) = n \lg n \)
  - \( n^{\log_4 3} \sim n^{0.793}, f(n) = \Omega(n^{0.793 + \epsilon}) \)
  - \( af(n/b) = 3(n/4) \lg(n/4) \leq (3/4)n \lg n = 3/4f(n) \)
  - \( T(n) = \Theta(n \lg n) \)
\[ T(n) = aT(n/b) + f(n): \]

1. If \( f(n) = O(n^{\log_b(a) - \epsilon}) \) for some \( \epsilon > 0 \) then \( T(n) = \Theta(n^{\log_b(a)}) \).

2. If \( f(n) = \Theta(n^{\log_b(a)}) \) then \( T(n) = \Theta(n^{\log_b(a)} \lg n) \).

3. If \( f(n) = \Omega(n^{\log_b(a) + \epsilon}) \) for some \( \epsilon > 0 \) and \( af(n/b) \leq cf(n) \) for some \( c > 1 \) then \( T(n) = \Theta(f(n)) \).

- \( T(n) = 2T(n/2) + n \lg n \)
  - \( a = b = 2, f(n) = n \lg n \)
  - \( n^{\log_2(2)} = n^1; \)
    - \( n \lg n \neq O(n^{1-\epsilon}) \)
    - \( n \lg n \neq \Theta(n) \)
    - \( n \lg n \neq \Omega(n^{1+\epsilon}) \)
- Theorem does not apply!
Random Variables and Expectation: A Review

Suppose $S$ is a sample space with a probability $\Pr$.

Remember: a random variable $X$ on $S$ is a function from $S$ to the real numbers.

- $\Pr(X = x) = \Pr(\{s \in S : X(s) = x\})$
- Example: toss a pair of fair dice.
  - Let $S$ be the set of 36 outcomes: $(1, 1), (1, 2), \ldots$
  - Let $X(a, b) = a + b$
  - $\Pr(X = 4) = \Pr(\{(1, 3), (2, 2), (3, 1)\}) = 1/12$

The expected value of $X$ is

$$E(X) = \sum_x x \Pr(X = x).$$

- For $X(a, b) = a + b$
  $$E(X) = 2(1/36) + 3(2/36) + 3(3/36) + \cdots + 7(6/36) + \cdots + 12(1/36)$$

When we talk about the average-case running time of an algorithm, we mean the expectation.
Dynamic Sets

- A *dynamic set* is one whose membership changes over time.
- Sometimes the elements of a dynamic set have an associated *key*
  - In that case, we write $\text{key}[x] = k$
- Sometimes keys come from a totally ordered set
  - this means $\text{key}[x] > \text{key}[x']$, $\text{key}[x] < \text{key}[x']$ or $\text{key}[x] = \text{key}[x']$
Dynamic Set Operations

We want to be able to manipulate dynamic sets.

Typical operations:

- **SEARCH(S, k)**: returns \( x \in S \) such that \( key[x] = k \) if there is one; **NIL** otherwise
  - typically \( x \) is a pointer to an element in \( S \), not the element itself
- **INSERT(S, x)**
- **DELETE(S, x)**
- **MINIMUM(S)**: returns element with smallest key
- **MAXIMUM(S)**: returns element with largest key
- **SUCCESSOR(S, x)**
- **PREDECESSOR(S, x)**
  - **MINIMUM**, **MAXIMUM**, **PREDECESSOR**, and **SUCCESSOR** make sense only if the keys are totally ordered
We do not necessarily want or need to implement all these operations.

- Different data types implement different subsets
- A dictionary allows insert, delete, and search
- A priority queue allows insert, delete, maximum
- There are typically tradeoffs between implementations