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.

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: \( t_j = j \)

\[
\begin{align*}
&1 \text{ for } j \gets 2 \text{ to } \text{length}[A] \\
&2 \quad \text{do } key \gets A[j] \\
&3 \quad \triangleright \text{ Insert } A[j] \text{ into sorted} \\
&4 \quad \text{sequence } A[1..j-1]. \\
&5 \quad i \gets j - 1 \\
&6 \quad \text{while } i > 0 \text{ and } A[i] > key \\
&7 \quad \text{do } A[i+1] \gets A[i] \\
&8 \quad i \gets i - 1 \\
&9 \text{ while } i > 0 \text{ and } A[i] > key \\
&10 \quad A[i+1] \gets key
\end{align*}
\]

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

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

Suppose \( A = \langle a_1, \ldots, a_n \rangle \) is an array to be sorted.
- \( A[i] = a_i \)

**INSERTION-SORT(A)**

\[
\begin{align*}
&1 \text{ for } j \gets 2 \text{ to } \text{length}[A] \\
&2 \quad \text{do } key \gets A[j] \\
&3 \quad \triangleright \text{ Insert } A[j] \text{ into sorted sequence } A[1..j-1]. \\
&4 \quad i \gets j - 1 \\
&5 \quad \text{while } i > 0 \text{ and } A[i] > key \\
&6 \quad \text{do } A[i+1] \gets A[i] \\
&7 \quad i \gets i - 1 \\
&8 \quad A[i+1] \gets key
\end{align*}
\]

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

\[
\begin{align*}
&1 \quad 2 \quad 4 \quad 6 \quad 1 \quad j = 2; \text{ key } = 2 \\
&2 \quad 5 \quad 4 \quad 6 \quad 1 \quad j = 3; \text{ key } = 4 \\
&3 \quad 4 \quad 5 \quad 6 \quad 1 \quad j = 4; \text{ key } = 6 \\
&4 \quad 4 \quad 5 \quad 6 \quad 1 \quad j = 5; \text{ key } = 1 \\
&5 \quad 2 \quad 4 \quad 5 \quad 6
\end{align*}
\]

\[
T(A) = c_1 n + (c_2 + c_4 + c_6 - c_5 - c_7)(n-1) + (c_3 + c_6 + c_7) \sum_{j=1}^{n-1} t_j
\]

Best case: \( t_j = 1 \)

\[
T(A) = c_1 n + (c_2 + c_4 + c_6 - c_5 - c_7)(n-1) = (c_1 + c_2 + c_4 + c_5 + c_6)n - (c_2 + c_4 + c_5 + c_6)
\]

Worst case: \( t_j = j \)

\[
T(A) = (c_2 + c_4 + c_6) \sum_{j=1}^{n-1} j + \cdots = (c_2 + c_4 + c_6) \frac{(n+1)n}{2} - 1 + \cdots
\]

- Quadratic is OK if \( n = 5, 10, 100 \).
- 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.
  - \( t_j \sim j/2 \)
- 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 sub-sequences 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. \( \text{then } q \leftarrow \lfloor (p + r)/2 \rfloor \)
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

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

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

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

- \( T(m) \) = the worst-case time for \( \text{MERGE-SORT}(A, p, r) \)
- \( U(m) \) = be the worst-case time for \( \text{MERGE}(A, p, q, r) \)

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

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

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

- This is much better than \( \Theta(n^2) \) for large \( n \)

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, Θ, Ω) (Chapter 2.1)
  o Recurrences (Chapter 4.1, 4.3)
  o A little probability

Example: 2n^2 + 3n + 1 = Θ(n^2) = 2n^2 + Θ(n).
• Clearly 2n^2 ≤ 2n^2 + 3n + 1 ≤ 3n^2 for n ≥ 4
  o Since n^2 ≥ 3n + 1 if n ≥ 4
• Also 2n^2 + 3n ≤ 2n^2 + 3n + 1 ≤ 2n^2 + 4n

More generally, if a > 0,
  an^2 + bn + c = Θ(n^2) = an^2 + Θ(n)

Example: 6n^3 ≠ Θ(n^2); 6n^3 = Ω(n^3).
• Really should say 6n^3 ∉ Θ(n^2); 6n^3 ∈ Ω(n^3).

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)) if there is a constant c such that c Ấ(n) is asymptotically an upper/lower/tight bound for T(n)
  o T(n) = Θ(g(n)) iff
    T(n) = O(g(n)) and T(n) = Ω(g(n)).

We won’t cover o(g(n)), ω(g(n)).

Formally, Θ(g(n)), O(g(n)), and Ω(g(n)) are sets of functions:
• Θ(g(n)) = {f(n) : \exists c1, c2 > 0, n0
  (c1 g(n) ≤ f(n) ≤ c2 g(n) for n ≥ n0)}
• O(g(n)) = {f(n) : \exists c2 > 0, n0
  (f(n) ≤ c2 g(n) for n ≥ n0)}
• Ω(g(n)) = {f(n) : \exists c1 > 0, n0
  (c1 g(n) ≤ f(n) for n ≥ n0)}

The O, Θ, Ω notation ignores constants.
• The constants depend on the machine, details of implementation
• Improving the constants is good but . . .
• Improving Θ(. . .) is better
  o It gives us a better indication of how the problem scales
    o Θ(\log n) < Θ(\log^2 n) < Θ(n) < Θ(n \log n) <
      Θ(n^2) < Θ(n^3)
Recurrences

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

\[
T(m) = \begin{cases} 
  c_1 & \text{if } m = 1 \\
  2T\left(\left\lceil \frac{m}{2} \right\rceil \right) + cm & \text{if } m > 1
\end{cases}
\]

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

- Often stated without \([\ ]\), \([\ ]\)

How do we solve them?

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

We won’t cover iteration method.

The Substitution Method

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

Guess \(T(n) \leq cn\log(n)\) (for some \(c\))

Verify:

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

What about \(T(1)\)?

- \(c\log 1 = 0\)

All we need is \(T(n) \leq cn \log n\) for \(n\) sufficiently large.

- E.g. \(T(2) = 2c_1 + 2\). Choose \(c = c_1 + 1\).
  - Then \(T(2) \leq 2c_1\log 2 = 2c_1 + 2\)

A formal proof that \(T(n) \leq cn\log n\) 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

\[
T(n) = aT\left(\frac{n}{b}\right) + f(n).
\]

- 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 \(T(n) = \Theta(n^{\log_b(a)} \log n)\).
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 \(T(n) = \Theta(f(n))\).

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

**Examples:**
- \( T(n) = 9T(n/3) + n \)
  - \( a = 9, b = 3, f(n) = n \)
  - \( n^{\log_3(3)} = n^2 \), so \( f(n) = O(n^{\log_3(3)-1}) \)
  - \( T(n) = \Theta(n^2) \)
- \( T(n) = T(2n/3) + 1 \)
  - \( a = 1, b = 3/2, f(n) = 1 = n^{\log_3(2)} \)
  - \( T(n) = \Theta(\log n) \)
- \( T(n) = 3T(n/4) + n \log n \)
  - \( a = 3, b = 4, f(n) = n \log n \)
  - \( n^{\log_4(3)} \sim n^{0.793} \), so \( f(n) = \Omega(n^{0.793 + \epsilon}) \)
  - \( af(n/b) = 3(n/4) \log(n/4) \leq (3/4)n \log n = 3/4f(n) \)
  - \( T(n) = \Theta(n \log n) \)

**Random Variables and Expectation: A Review**

Suppose \( S \) is a sample space with a probability \( \text{Pr} \) Remember: a random variable \( X \) on \( S \) is a function from \( S \) to the real numbers.

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

The expected value of \( X \) is \( E(X) = \sum_{x} x \text{Pr}(X = x) \).

- 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; \text{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