CS310 - Advanced Data Structures and Algorithms

Fall 2016 – Algorithmic Techniques

October 2, 2016
Common techniques to solve various problems.
- Divide and conquer
- Backtracking
- Greedy algorithms
- Dynamic programming
- We will use several examples, some problems you have seen before (sorting), to demonstrate the use of such techniques.
- We will work with a software package that programs simple board games.
A method that is partially defined in terms of itself is called \textit{recursive}.

- Mathematical induction
- Numerical applications
- Divide and conquer
- Dynamic programming
- Backtracking
Basic Rules of Recursion

- **Base case:** Always have at least one case that can be solved without using recursion.
- **Make progress:** Any recursive call must progress toward a base case.
- **For efficient runtime, observe the compound interest rule:** Never duplicate work by solving the same instance of a problem in separate recursive calls.
One of the most fundamental problems in CS.

- Problem definition: Given a series of elements with a well-defined order, return a series of the elements sorted according to this order.
- Simple (insertion) Sort – runs in quadratic time
- BubbleSort – runs in quadratic time
- Shellsort – runs in sub-quadratic time
- Mergesort – runs in $O(N\log N)$ time
- Quicksort – runs in average $O(N\log N)$ time
3 steps
1. Return if the number of items to sort is 0 or 1
2. Recursively Mergesort the first and second halves separately
3. Merge the two sorted halves into a sorted group

This approach is called "divide and conquer".

Divide the problem into sub-problems, "conquer" (solve) them separately and merge the results.

Mergesort is an O(N*logN) algorithm
public static <AnyType extends Comparable<? super AnyType>>
    void mergeSort(AnyType [] a)
{
    AnyType [] tmpArray = (AnyType []) new
        Comparable[a.length];
    mergeSort(a, tmpArray, 0, a.length - 1);
}

// Internal method that makes recursive calls.
private static <AnyType extends Comparable<? super AnyType>>
    void mergeSort(AnyType[ ] a, AnyType[ ] tmpArray,
        int left, int right)
{
    if( left < right ) {
        int center = ( left + right ) / 2;
        mergeSort( a, tmpArray, left, center );
        mergeSort( a, tmpArray, center + 1, right );
        merge( a, tmpArray, left, center + 1, right );
    }
}
import java.util.Arrays;

public class InternalMergeMethod {
    private static <AnyType extends Comparable<? super AnyType>>
    void merge(AnyType[] a, AnyType[] tmpArray, int leftPos, int rightPos, int rightEnd) {
        int leftEnd = rightPos - 1;
        int tmpPos = leftPos;
        int numElements = rightEnd - leftPos + 1;

        // Main loop
        while (leftPos <= leftEnd && rightPos <= rightEnd) {
            if (a[leftPos].compareTo(a[rightPos]) <= 0) {
                tmpArray[tmpPos++] = a[leftPos++];
            } else {
                tmpArray[tmpPos++] = a[rightPos++];
            }
        }

        // Copy rest of first half
        while (leftPos <= leftEnd) {
            tmpArray[tmpPos++] = a[leftPos++];
        }

        // Copy rest of right half
        while (rightPos <= rightEnd) {
            tmpArray[tmpPos++] = a[rightPos++];
        }

        // Copy tmpArray back
        for (int i = 0; i < numElements; i++, rightEnd--) {
            a[rightEnd] = tmpArray[rightEnd];
        }
    }
}

Nurit Haspel
CS310 - Advanced Data Structures and Algorithms
Linear-time Merging of Sorted Arrays

1 13 24 26
1 13 24 26
1 13 24 26
1 13 24 26

2 15 27 38
2 15 27 38
2 15 27 38
2 15 27 38

1
1 2
1 2 13
1 2 13 15

1 13 24 26
1 13 24 26
1 13 24 26
1 13 24 26

2 15 27 38
2 15 27 38
2 15 27 38
2 15 27 38

1 2 13 15 24 26 27 38
MergeSort Performance

\[ T(N) = 2 \times T(N/2) + O(N) \]

\[ = 2 \times (2 \times T(N/4) + O(N/2)) + O(N) \]

\[ = 4 \times T(N/4) + O(N) + O(N) \]

\[ = 4 \times (2 \times T(N/8) + O(N/4)) + O(N) + O(N) \]

\[ = 8 \times T(N/8) + O(N) + O(N) + O(N) \]

\[ = \ldots \ldots = 2 \log N \times T(1) + O(N) + O(N) + \ldots + O(N) \]

\[ = N \times O(1) + O(N) + O(N) + \ldots + O(N). \]

The terms are expanded logN times, each produces an \( O(N) \). log N terms of \( O(N) = O(N \log N) \).
4 steps:

1. Return if the number of elements in $S$ is 0 or 1
2. Pick a “pivot” – element $v$ in $S$
3. Partition $S - \{v\}$ into 2 disjoint sets:
   \[
   L = \{x \in S - \{v\} | x < v\},
   R = \{x \in S - \{v\} | x > v\}
   \]
4. Return the result of Quicksort($L$) followed by $v$ followed by Quicksort($R$)

Notice that after each partition the pivot is in its final sorted position.
Quick sort Algorithm

Select pivot

Partition

Quick sort small items

Quick sort large items
$$T(N) = O(N) + T(|L|) + T(|R|)$$

- The first term refers to the partition, which is linear in $N$.
- The second and third are recursive calls to subarrays of size $L$ and $R$, respectively.
- Similar to mergesort analysis, so should be $O(N \log N)$... or is it?
- The result depends on the size of $L$ and $R$. If roughly the same – yes. Otherwise – if one partition is $O(1)$ and the other $O(N)$, may be quadratic!
A wrong way
- Pick the first element or the larger of the first two elements
- If the input has been presorted or is reverse order, this is a poor choice

A safe choice
- Pick the middle element

Median-of-three
- Pivot equal to the median of the first, middle and last elements
- Nothing guarantees asymptotic $O(N \times \log N)$, but it can be shown that mostly this is the case.
Binary Search

- Definition: Search for an element in a sorted array.
- Return array index where element is found or a negative value if not found.
- Implemented in Java as part of the Collections API.
- Idea from the book: start in the middle of the array.
- If the element is smaller than that, search in the smaller half. Otherwise – search in the larger half.
static <T> int binarySearch(T[] a, T key, Comparator<? super T> c)
static int binarySearch(Object[] a, Object key)

- The version without the Comparator uses “natural order” of the array elements, i.e., calls compareTo of the element type to compare elements.
- Thus the elements need to be Comparable – the element type implements Comparable<ElementType> in the generics setup.
- Or the old Comparable works here too.
// Hidden recursive routine.
private static <AnyType extends Comparable<? super AnyType>>
int binarySearch( AnyType [] a, AnyType x, int low, int high )
{
    if( low > high )
        return NOT_FOUND;

    int mid = ( low + high ) / 2;

    if( a[ mid ].compareTo( x ) < 0 )
        return binarySearch( a, x, mid + 1, high );
    else if( a[ mid ].compareTo( x ) > 0 )
        return binarySearch( a, x, low, mid - 1 );
    else
        return mid;
}
What is that <\texttt{superT}> clause?

The \textit{Comparable} <\texttt{superT}> specifies that T ISA \textit{Comparable} <\texttt{Y}>, where \texttt{Y} is T or any superclass of it.

This allows the use of a \texttt{compareTo} implemented at the top of an inheritance hierarchy (i.e., in the base class) to compare elements of an array of subclass elements.

For example, we commonly use a unique id for equals, \texttt{hashCode} and \texttt{compareTo} across a hierarchy, and only want to implement it once in the base class.
You should be able to figure this one out by now (I hope):
\[ T(N) = T(N/2) + O(1) \]
\[ T(N) = O(\log N) \]
 Often we use a tree to represent sorted data.

The tree is not always balanced (so we don’t always cut it in half when we search) but we can show that often the tree is balanced *enough* to give a logarithmic performance.

It’s beyond the scope of this course, but the reasons are very similar to quick sort being very often $O(N \log N)$. As a matter of fact, these are very closely related problems.
static void sort(Object[] a)

static <T> void sort(T[] a, Comparator<? super T> c)

Default – natural order of elements from small to large. Possible to define another Comparator.
• It can be shown that in the general case (comparison based sorting) we can’t do better than $O(N \times \log N)$ in the worst case.

• When assumptions can be made on the input – linear sorting is possible.

• Example – $N$ integers all between 1 and $O(N)$. 
Recursion – Numerical Applications

- Modular arithmetic
- Modular exponentiation
- GCD and multiplicative inverse
- The RSA cryptosystem
An arithmetic system where the count “wraps around” a certain number, called the modulo.

Common example – the 12 (or 24) hour clock.

For any positive integer $n$, two numbers $A$ and $B$ are congruent modulo $n$, written $A \equiv B \pmod{n}$ if $a - b$ is an integer multiple of $n$.

Equivalently – $a$ and $b$ have the same remainder when divided by $n$.

$a$ and $b$ can also be negative...

For example – $38 \equiv 14 \pmod{12}$
Modular Arithmetic

Theorems

1. If \( A \equiv B \pmod{N} \), then for any \( C \), \( A + C \equiv B + C \pmod{N} \)
2. If \( A \equiv B \pmod{N} \), then for any \( D \), \( AD \equiv BD \pmod{N} \)
3. If \( A \equiv B \pmod{N} \), then for any positive \( P \), \( A^P \equiv B^P \pmod{N} \)

What is the last digit in \( 3333^{5555} \)?

There are more than 15,000 digits, too prohibitive to compute directly

Wanted: \( 3333^{5555} \pmod{10} \)

\( 3333 \equiv 3 \pmod{10} \), thus we only need \( 3^{5555} \pmod{10} \)

\( 3^4 = 81, \ 3^4 \equiv 1 \pmod{10} \)

\( (3^4)^{1388} = 3^{5552} \equiv 1 \pmod{10} \)

\( 3^3 \ast 3^{5552} \equiv 3^3 \ast 1 \pmod{10} = 27 \pmod{10} = 7 \)
Modular Exponentiation

- How to compute $x^n \pmod{p}$ when $n$ is huge?
- Take $\pmod{p}$ for intermediate results – keep the numbers small
- If $n$ is even, $x^n = (x \cdot x)^{\lfloor n/2 \rfloor}$
- If $n$ is odd, $x^n = x \cdot (x \cdot x)^{\lfloor n/2 \rfloor}$
- Let $M(n)$ be the number of multiplications used by power
- $M(n) \leq M(\lfloor n/2 \rfloor) + 2$
- $M(n) < 2 \log n$
- On average, $M(n)$ is about $(3/2) \log n$

```java
// Return $x^n \pmod{p}$
// Assumes $x, n \geq 0$, $p > 0$, $x < p$, 0^0 = 1
// Overflow may occur if $p > 31$ bits.
public static long power(long x, long n, long p) {
    if (n == 0)
        return 1;
    long tmp = power( (x*x)%p, n/2, p );
    if (n % 2 != 0)
        tmp = (tmp*x) % p;
    return tmp;
}
```

Nurit Haspel
GCD, Euclid’s Algorithm

- Assume w.l.o.g $a > b$ (you can always switch places)
- $\gcd(a, b) \equiv \gcd(a - b, b)$
- Repeat as necessary...
- $\gcd(a, b) \equiv \gcd(b, a \pmod{b})$
- $\gcd(n, m) = O(\log n)$

```java
// Return greatest common divisor
public static long gcd( long a, long b )
{
    if (b == 0)
        return a;
    else
        return gcd( b, a % b);
}
```
Assume $1 \leq a < n$

The solution $1 \leq x < n$ to the equation $ax \equiv 1 \pmod{n}$ is called **multiplicative inverse** of $a \pmod{n}$

Think of it as the inverse number.

Example:

- What is $i$ such that $3i \equiv 7 \pmod{13}$?
- The multiplicative inverse of $3 \pmod{13}$ is $9$
- Multiply both sides of $3i \equiv 7 \pmod{13}$ by $9$ to “eliminate” the $3$.
- $i \equiv 63 \pmod{13}$, so $i = 11$

Notice that a multiplicative inverse for $a \pmod{N}$ exists iff $a$ and $N$ are co-prime.
We will use the extended Euclidean Algorithm.

An extension of Euclid’s algorithm that, given $0 < |b| < |a|$, finds $x$ and $y$ such that $ax + by = gcd(a, b)$.

Notice that $x$ and $y$ are guaranteed to exist and obviously, at least one of them is usually negative.

It does so by keeping track of the quotients, not only the remainders, while running Euclid’s algorithm.

Finding the multiplicative inverse is a special case of this algorithm.
Computing Multiplicative Inverse

- Given a number $a$, its multiplicative inverse $x$, if one exists, has the property of $ax \equiv 1 \pmod{n}$
- If $x$ exists then $a$ and $n$ are co-prime, so $\gcd(a, n) = 1$
- Notice that if for some $ax \equiv 1 \pmod{n}$ then for any $y$, $ax + ny \equiv 1 \pmod{n}$
- In other words, we can ignore the $yn$ part and apply the extended algorithm to find $x$
// Internal variables for fullGcd
private static long x, y;

// Find x and y such that if gcd(a,b) = 1, ax + by = 1.
private static void fullGcd(long a, long b) {
    long x1, y1;
    if( b == 0 ) {
        x = 1; y = 0;
    }
    else {
        fullGcd( b, a % b );
        x1 = x; y1 = y; x = y1;
        y = x1 - ( a / b ) * y1;
    }
}

public static long inverse(long a, long n) {
    fullGcd( a, n );
    return x > 0 ? x : x + n;
}
RSA Cryptosystem

Hello Bob

Public Key

RSA

Private Key

Hello

Bob

Eve

x0Ak3o$2Rj

RSA

Hello

Bob

Alice

Bob

Nurit Haspel

CS310 - Advanced Data Structures and Algorithms
RSA Cryptosystem

- Pick two large prime numbers, \( p \) and \( q \), each having 100 digits or more.
- Compute \( N = pq \) and \( N' = (p-1)(q-1) \).
- Choose a number \( e \) such that \( \gcd(e, N') = 1 \), relatively prime.
- Compute \( d \), the multiplicative inverse of \( e \) (mod \( N' \)).
- Destroy \( p \), \( q \), and \( N' \).
- Publish \( e \) and \( N \), and keep \( d \) a secret.

- To encrypt a message \( M \), compute \((M^e \text{ mod } N)\) and send it.
- To decrypt a received message \( R \), compute \((R^d \text{ mod } N)\).
- \( M^{ed} = M \) (mod \( N \)).
- This is called public key cryptography, whereas DES and AES are symmetric key cryptography.
- Public key cryptography is slow – AES is fast.
- Use RSA to exchange the AES key.
Choose \( p = 3 \) and \( q = 11 \)

2. Compute \( N = p \times q = 3 \times 11 = 33 \)

3. Compute \( N' = (p - 1) \times (q - 1) = 2 \times 10 = 20 \)

4. Choose \( e \) such that \( 1 < e < N' \) and \( e \) and \( N \) are co-prime. Let \( e = 7 \)

5. Compute a value for \( d \) such that \( (d \times e) \mod N' = 1 \). One solution is \( d = 3 \) because \( (3 \times 7) \mod 20 = 1 \)

6. Public key is \( (e, N) \Rightarrow (7, 33) \)

7. Private key is \( (d, N) \Rightarrow (3, 33) \)

8. The encryption of \( M = 2 \) is \( c = 2^7 \mod 33 = 29 \)

9. The decryption of \( c = 29 \) is \( M = 29^3 \mod 33 = 2 \)
A very useful proof technique

1. Establish the basis – usually a very simple case.
2. Assume the hypothesis for $1 \geq k < n$
3. Demonstrate the induction for $n$
Prove These by Math Induction

\[
\sum_{i=1}^{n} i = \frac{n(n+1)}{2}
\]

\[
\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}
\]

\[
\sum_{i=1}^{n} i^3 = \left( \frac{n(n+1)}{2} \right)^2
\]

\[
\sum_{i=0}^{n} 2^i = 2^{n+1} - 1
\]
First example

Prove that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

- Base case for $n = 1$: $\sum_{i=1}^{1} i = \frac{1*2}{2} = 1$ (told you it was easy...)

- Induction hypothesis: Suppose the equation is true for $1 \geq k < n$

- Proof:

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n-1} i + n$$

$$= \frac{(n-1)n}{2} + n \text{ (By inductive hypothesis)}$$

$$= \frac{(n-1)n}{2} + \frac{2n}{2} = \frac{n^2 - n + 2n}{2}$$

$$= \frac{n^2 + n}{2} = \frac{n(n+1)}{2}$$
Proof by Induction – Some Tips

- Base case is usually the smallest non-trivial example and should be immediate.
- In the proof stage you **must** use the inductive hypothesis. If not – something is wrong.
- Try the other equations! All you need is a bit of calc 1 level math.
- Induction and recursion share a lot of similarities, even though it looks like they are "opposite".

Nurit Haspel
CS310 - Advanced Data Structures and Algorithms
Fibonacci Numbers

- Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, …
- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n - 1) + F(n - 2)$, for $n \geq 2$
- Related to golden ratio
- Closed form of Fibonacci numbers

\[
\begin{align*}
\alpha &= \frac{1 + \sqrt{5}}{2} \\
\beta &= \frac{1 - \sqrt{5}}{2} \\
F_n &= \frac{\alpha^n - \beta^n}{\sqrt{5}}
\end{align*}
\]

- Fibonacci numbers are exponential in $n$
Remember Too Much Recursion?

Compute the n-th Fibonacci number

// Bad algorithm
public static void fib(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}

Let $C(n)$ be the number of calls to fib() made during the evaluation of fib(n)

- $C(0) = C(1) = 1$
- $C(n) = C(n-1) + C(n-2) + 1$
- $C(n) = F(n+2) + F(n-1) - 1$

Prove by?
Remember Too Much Recursion?

Compute the n-th Fibonacci number

// Bad algorithm
public static void fib(int n) {
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}

Let \( C(n) \) be the number of calls to \( \text{fib}() \) made during the evaluation of \( \text{fib}(n) \)

\[
C(0) = C(1) = 1
\]

\[
C(n) = C(n-1) + C(n-2) + 1
\]

\[
C(n) = F(n+2) + F(n-1) - 1
\]

Prove by?

Induction