Part II: Algorithm Design
General Strategies

- There is no general solution to all problems
- But there are various \textit{general techniques} that can be applied successfully to solve many problems
  - Some of them we covered in our sorting algorithms but didn't stop to identify them
  - Here we will look at a few important techniques that you might find useful when you need to design algorithms (for computers or otherwise)
Coin Changing Problem

How can you make a value $V$ using the fewest coins for some set of coin denominations?
Make 77 from British coin denominations

- (1, 2, 5, 10, 20, 50, 100, 200)

\[ \begin{align*}
0 & \quad \text{50p} \\
4 & \quad 0 \\
5 & \quad 0 \\
7 & \quad 2
\end{align*} \]

4 coins
Greedy Algorithms

- Always perform whatever operation contributes as much as possible in a single step
- Simple to implement and understand
- But it doesn't always optimize fully...

\[200, 100, 50, 20, 2, 1\]

\[
\begin{align*}
60 & < 60 \\
10 & \quad \rightarrow 0
\end{align*}
\]

\[
\begin{align*}
50 & \\
2 & \\
2 & \\
2 & \\
2 & \\
2 & \\
\end{align*}
\]

6 coins

\[
\begin{align*}
20 & \\
20 & \\
20 & \\
20 & \\
\end{align*}
\]

3 coins

\[
\frac{60}{60}
\]
Another Way...

- Let $C[i]$ be the minimum number of coins needed to make value $i$.
- Let there be denominations $\{d_1, d_2, ..., d_k\}$ available.
- Imagine that we knew $d_j$ was part of the best solution for $i$.
- Then

$$C[77] = C[27] + 1$$

$$C[i] = C[i-d_j] + 1$$
Another way...

- We could now define the optimal solution recursively:

\[
C[i] = \begin{cases} 
\infty & i < 0 \\
0 & i = 0 \\
1 + \min_{j=1\text{ to }k} \{ C[i-d_j] \} & i \geq 1
\end{cases}
\]

\[
C[5] = 1 + \min \begin{cases} 
C[4] = 1 \\
C[3] = 1 \\
C[2] = 0 \end{cases}
\]

\[
C[2] = 1 + \min \begin{cases} 
C[1] = 1 \\
C[2-2] = 0 \end{cases} = 1
\]

\[
C[5] = 1 + \min \begin{cases} 
C[5-1] = 1 \\
C[5-2] = 0 \end{cases} = 1
\]

\[
C[6] = 1 + \min \begin{cases} 
C[5] = 1 \\
C[6] = 2 \\
C[5-2] = 2 \end{cases} = 2
\]

\[
C[7] = 2
\]
This is Dynamic Programming

- Don't try to understand the name – it's there for historical reasons
- It's like D&C but...
  - D&C splits the problem into a series of small, independent problems
  - DP splits the problem into a series of small, dependent problems (i.e. the subproblems overlap)
  - DP assumes that a solution it needs for some subproblem is applicable for other subproblems so it 'saves' results
Dynamic Programming

- Ideal when we have lots of possible solutions and we need to find an optimal one (i.e. optimization problems)

- Steps:
  - Characterise the structure of an optimal solution
  - Recursively define the optimal solution
  - Compute the value of the optimal solution bottom-up (DP) or top-down (memoized DP)
  - [Figure out the optimal solution]
Another Example: Fibonacci Numbers

- F(0) = 0 \textit{(different in notes)}
  \[ F(1) = 1 \]
  \[ F(n) = F(n-2) + F(n-1) \quad n > 1 \]

- Recursive:

```c
int fib (int a) {
    if (a==0) return 0;
    if (a==1) return 1;
    else return fib(a-1) + fib(a-2);
}
```

O\textit{(exponential)}
Example: Fibonacci Numbers

- Top-down (memoized DP)

```java
map saved = { (0,0), (1,1) }
int fib (int a) {
    if (saved contains a) return saved[a];
    else saved[a] = fib(a-1) + fib(a-2);
    return saved[a];
}
```

**Diagram:**
- $f(0) = 0$
- $f(1) = 1$
- $f(2) = 1$
- $f(3) = 2$
- $f(4) = 3$
- $f(5) = 5$

Space: $O(n)$

Time: $O(n)$
Example: Fibonacci Numbers

- **Bottom-up (normal DP)**

```plaintext
int fib (int a) {
    if (a == 0) return 0;
    if (a == 1) return 1;
    prev_fib = 0;  <---- f[a-2]
    current_fib = 1;  <---- f[a-1]
    for (i = 1 to (a-1)) {
        this_fib = prev_fib + current_fib;
        prev_fib = current_fib;
        current_fib = this_fib;
    }
}
```

Computed

- \( f(2) = f(1) + f(0) \)
- \( f(3) = f(2) + f(1) \)
- \( f(4) = f(3) + f(2) \)
- \( f(5) = f(4) + f(3) \)

**0(n) performance**

**O(1) storage**
Dynamic Prog: When To Use

- Use when all apply:
  - You have many choices, each with a score and you need to find a max or min.
  - Brute forcing is exponentially tough
  - The optimal solution is composed of optimal solutions to smaller problems
  - The optimal solutions top the smaller problems crop up multiple times when trying to solve the bigger problems (“overlap” of solutions)
Dynamic Programming

- DP has turned out to be really important for many scientific problems
  - Science is often about optimizing very large quantities of data

- The notes and CLRS will give you some more examples (it's best understood through example)
  - Matrix multiplication
  - String matching
Brute Force

- Consider a maze where you need to find the path S to E

Brute Force

- Generate every possible path from S to E ignoring the blocked squares
- Now go through each path until you find one that does not pass through a blocked square
Backtracking

- Now we move one square at a time, recursively exploring each direction.
- If we hit an end, we **backtrack** to where the last split was and continue from there.
  - (Yes, this is basically what you would do naturally.)
Divide and Conquer

- This is a familiar strategy now
  - Quicksort, mergesort

- **Divide**: cut the problem into parts (almost always two)
- **Conquer**: recursively solve the parts
- **Combine**: Use the part solutions to form a full solution
Example: Number Multiplication

- Take two n-bit numbers, a and b
  - Multiplying them together is $O(n^2)$
  - Consider splitting them into two $n/2$ bit numbers

\[
a \times b = a_1 b_1 2^n + a_2 b_2 2^{n/2} + a_1 b_2 2^{n/2} + a_2 b_1 2^{n/2}
\]
Example: Number Multiplication

\[ f(\sqrt{2}) = \frac{1}{2} f(1) + k \frac{1}{2} \]

\[ f(2^m) = 4^n f(2^n) + k2^n \]

\[ f(2^m) = \sum_{i=m}^{\infty} 2^i k2^{m-i} \]
Example: Number Multiplication

\[ a b = a_1 b_1 2^m + a_2 b_2 + a_1 b_2 2^{\frac{m}{2}} + a_2 b_1 2^{\frac{m}{2}} \]

\[ A = a_1 b_1 \]
\[ B = a_2 b_2 \]
\[ C = (a_1 a_2)(b_1 + b_2) = a_1 b_1 + a_2 b_2 + a_2 b_1 + a_1 b_2 \]
\[ a b = 2^A + B + (C - B - A)2^{\frac{m}{2}} \]
Example: Number Multiplication

\[
f(n) = 3f(\frac{n}{2}) + kn
\]

Let \( n = 2^m \) \( \Rightarrow \)

\[
f(2^m) = 3f(2^{m-1}) + k\cdot 2^m
\]

\[
= 3^2f(2^{m-2}) + k\cdot 2^{m-1} + k\cdot 2^m
\]

\[
= 3^3f(2^{m-3}) + k\cdot 2^{m-2} + k\cdot 2^m + k\cdot 2^m
\]

\[
= \cdots \]

\[
= 3^m f(2^0) + k\sum_{i=0}^{m-1} 2^i = 3^m f(1) + k\sum_{i=0}^{m-1} 2^i
\]

\[
= 3^m f(1) + k\cdot \frac{2^m - 1}{2 - 1} = 3^m f(1) + k\cdot (2^m - 1)
\]

\[
f(2^n) = 3^n f(1) + kn\cdot 2^n
\]

\[
0(3) = 0(n \log_2 3)
\]

\[
0(3) = 0(n \log_2 3) = 0(c) = 0(n)
\]