Part II: Algorithm Design

General Strategies

- There is no general solution to all problems
- But there are various 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?

How Would You Normally Do It?

- Make 77 from British coin denominations
  - (1,2,5,10,20,50,100,200)

77

Biggest $\leq 77 \Rightarrow$ 50p

Biggest $\leq 27 \rightarrow$ 20p

Biggest $\leq 7 \rightarrow 5$

Biggest $\leq 2 \rightarrow 2$

$$\left.\begin{array}{r} 50p \\ 20p \\ 5 \\ 2 \end{array}\right\} \text{4 coins}$$

$$\overline{77}$$

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

60p

60 <60

10 —0

50
2
2
2
2
2

6 coins

20
20
20
60

3 coins

Another Way…

- Let $C[i]$ be the minimum number of coins needed to make value i value

No of coins to make i

- Let there be denominations $\{d_1, d_2, \ldots, d_k\}$ available

100 50 1

- Imagine that we knew d_j was part of the best solution for i

50p

- Then

$$C[77] = \underbrace{C[27]}_{\text{left}} + \underbrace{1}_{\text{50p}}$$

$$C[i] = C[i - d_j] + 1$$

Another way...

- We could now define the optimal solution recursively:

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

$$C[7] = 1 + \min \begin{cases} C[7-1] = C[6] = 2 \\ C[7-2] = C[5] = 1 \\ C[7-5] = C[2] = 1 \end{cases} = \begin{cases} 3 \\ 2 \\ 2 \end{cases}$$

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

$$C[6] = 1 + \min \begin{bmatrix} C[5] = 1 \\ C[4] = 2 \\ C[1] = 1 \end{bmatrix} = 2$$

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

$$\underline{C[7] = 2}$$

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)

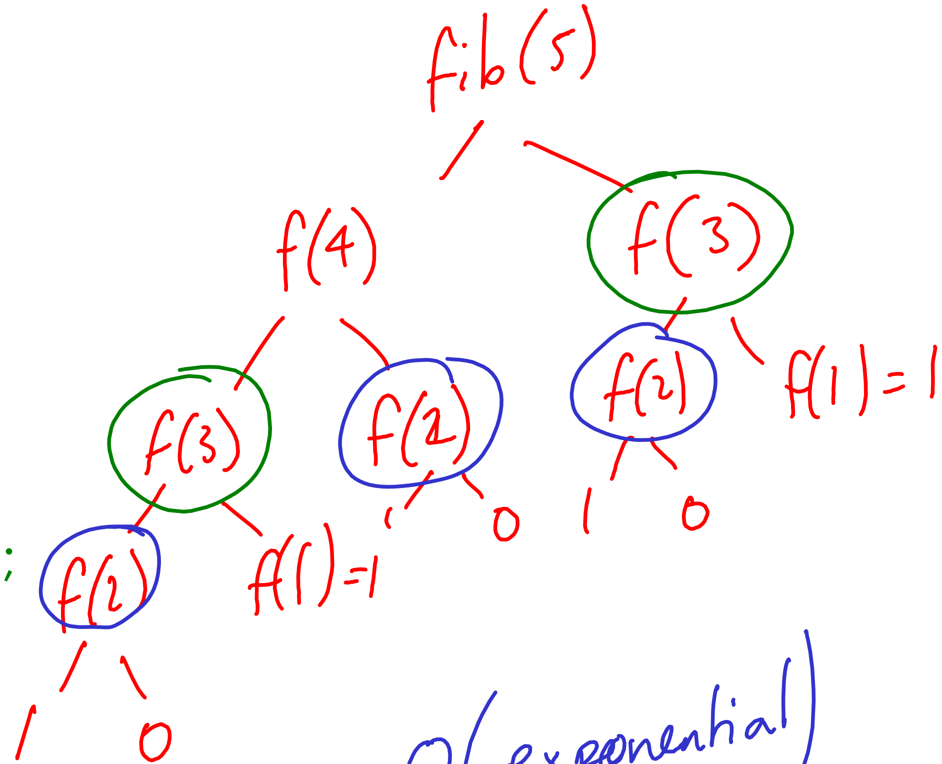
- Steps:
  - Characterise the structure of an optimal solution
  - Recursively define the optimal solution
  - Compute the value of the optimal solution bottom-up (DP) or top-down (memoized DP)
  - [Figure out the optimal solution]

Another Example: Fibonacci Numbers

- F(0) = 0 (different in notes)
  F(1) = 1
  F(n) = F(n-2) + F(n-1)   n>1

- Recursive:

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



O(exponential)

Example: Fibonacci Numbers

- Top-down (memoized DP)

Not valid 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];
}
```

Saved map

$0 \to 0$
$1 \to 1$
$2 \to 1$
$3 \to 2$
$4 \to 3$
$5 \to 5$

$O(n)$ space

$f(5)$
$f(4)$ $f(3) = 2$
$f(3)$ $f(2) = 1$
$f(2)$ $f(1) = 1$
$f(1) = 1$ $f(0) = 0$

$O(n)$

Example: Fibonacci Numbers

- Bottom-up (normal DP)

```
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

Remembered

$f(2) = f(1) + f(0)$

$f(3) = f(2) + f(1)$

$f(4) = f(3) + f(2)$

$f(5) = f(4) + f(3)$

$\boxed{O(n)}$ performance

$\boxed{O(1)}$ storage

Dynamic Prog: When To Use

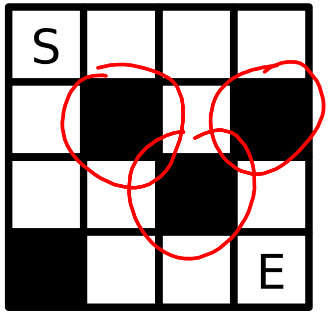
- Use when all apply:
  - You have many choices, each with a score and you need to find a max or min.
  - Brute forcing is exponentially tough
  - The optimal solution is composed of optimal solutions to smaller problems
  - The optimal solutions top the smaller problems crop up multiple times when trying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

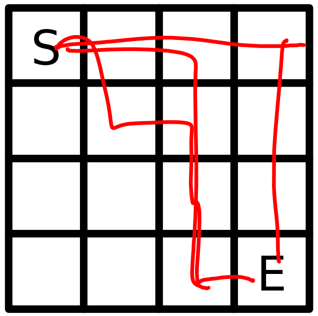
Dynamic Programming

- DP has turned out to be <u>really</u> 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

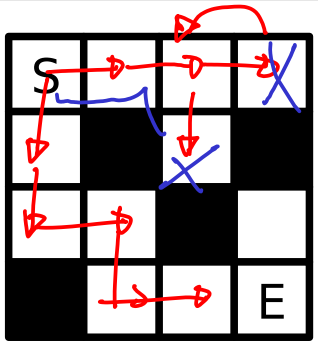
Brute Force





- 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)

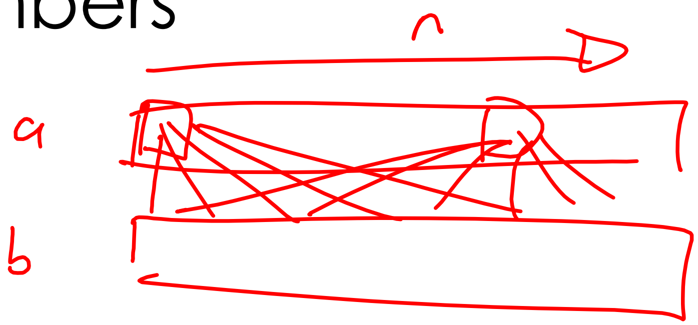
Divide and Conquer

- 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

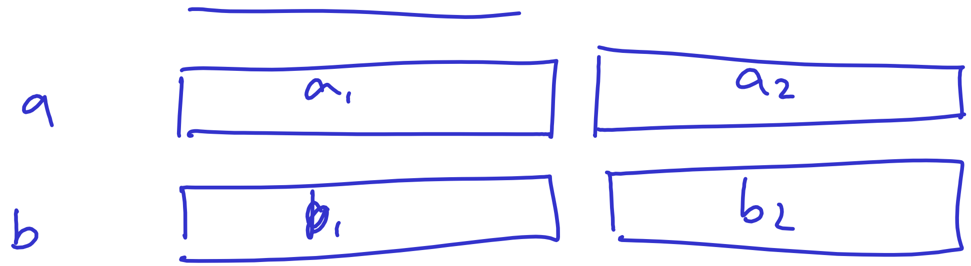
Example: Number Multiplication

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



$$\begin{array}{r} a \times \\ b \\ \hline ab \end{array}$$

$\boxed{O(n^2)_{\text{normal}}}$



$a = a_1 \cdot 2^{n/2} + a_2$

$b = b_1 \, 2^{n/2} + b_2$

$ab = \underline{a_1 b_1} 2^n + \underline{a_2 b_2} + \underline{a_1 b_2} 2^{n/2} + \underline{a_2 b_1} 2^{n/2}$ } 4 mult

Example: Number Multiplication

$$f(n) = \underline{4} f\left(\frac{n}{2}\right) + kn$$

$\div 2 \Rightarrow exp --$

$\times 2^2 \Rightarrow exp += 2$

$f(2^m)$

$$f(2^m) = 4 f(2^{m-1}) + k 2^m$$

$$= 4^2 f(2^{m-2}) + 2^2 k 2^{m-1} + k 2^m$$

$$= 4^n f(2^0) + k 2^m + k 2^{m+1} + k 2^{m+2} + \dots k 2^{2m-1}$$

$$= 4^n f(2^0) + k \sum_{i=m}^{2m-1} 2^i$$

$$= 4^n f(2^0) + k 2^m \sum_{i=0}^{m-1} 2^i$$

$$= 4^n f(2^0) + k\, \underline{2^m (2^m - 1)}$$

$2^m = n$

$\underline{\underline{O(n^2)}}$

Example: Number Multiplication

$$ab = a_1 b_1 2^n + a_2 b_2 + \underline{\underline{a_1 b_2 2^{n/2} + a_2 b_1 2^{n/2}}}$$

$$A = a_1 b_2$$

$$B = a_2 b_2$$

$$C = (a_1 + a_2)(b_1 + b_2) = a_1 b_1 + a_2 b_2 + a_2 b_1 + a_1 b_1$$

$$ab = 2^n A + B + (C - B - A)\underline{\underline{2^{n/2}}} \quad \ll$$

Example: Number Multiplication

$$f(n) = 3f\left(\frac{n}{2}\right) + kn$$

$n = 2^m \Rightarrow$

$$f(2^m) = 3f(2^{m-1}) + k2^m$$

$$= 3^2 f(2^{m-2}) + 3k2^{m-1} + k2^m$$

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

$$= \underline{3^m f(1)} + \underline{k 2^m \sum_{i=0}^{m-1} \left(\frac{3}{2}\right)^i}$$

$$O(3^m) = O(\underline{3^{\log_2 n}})$$

$$= O(n^{\log_2 3})$$

$$= O(n^{1.585})$$

$$\frac{2^m \left[\left(\frac{3}{2}\right)^m \cancel{-1}\right]}{\cancel{\frac{3}{2}-1}}$$ cont, const

Geometric progression S_n

$$2^m \cdot \frac{3^m}{2^3} \quad O(3^m)$$

$$\underline{O(n^{1.585})}$$

$$\log x \log y = \log y \log x$$

$$\log y^{\log x} = \log x^{\log y}$$

$$y^{\log x} = x^{\log y}$$