Greedy - Example 2 Continued

Start:D Finish:L Greedy will chose D,G,L. Total distance of 330km, optimum

Start:L Finish:D Greedy will chose L,C,D. Total distance of 370km, sub-optimum

The Travelling Salesperson Problem

This is a classic problem in Computer Science, and is essentially a variant on the above: The Salesperson must visit all the cities on the network exactly once, without "retracing his/her steps". Given the distances, calculate the shortest such route.

The Coins Problem

In a certain country, coins are available in denominations of $c_1, c_2, c_3, \ldots, c_d$. Find the least number of coins that are

necesssary to make up some fixed amount n, assuming we have a large number (∞ if you want) of each coin.

Greedy Solution to coins problem

At each step, choose largest coin that does not excede remaining amount.

Example 1 $c_i = 1, 2, 5, 10, 20, 50$ and $n = 28$:

step number	1	2	3	4
coin chosen	20	5	2	1
running total	20	25	27	28

so, a total of 4 coins: this is the best answer in this case.

Example 2 $c_i = 4, 5$ and $n = 20$:

step number	1	2	3	4
coin chosen	5	5	5	5
running total	5	10	15	20

again the greedy strategy gets the best answer.

Greedy Solution to coins problem - Continued

What about the cases

1. $c_i = 4, 5$ and $n = 19$
2. $c_i = 4, 5$ and $n = 18$
3. $c_i = 4, 5$ and $n = 17$

The (0/1) Knapsack Problem

In this problem, we have a bag/container of (weight) capacity K kilograms. We have n distinct items with weights $w(1), w(2), w(3), \ldots w(n)$ (kg) and corresponding values $v(1), v(2), v(3), \ldots v(n)$ (euro). The aim is to put as many items as possible in to our bag, in order to obtain maximum value.
Note:

- There is only one of each item (which is why it is called the 0/1 knapsack problem)

- If $w(1) + w(2) + w(3) + \cdots + w(n) < K$ then the problem is trivial (i.e. easy to solve): just take everything!

Knapsack Example

Suppose our bag capacity is $K = 4$ kg, and we have $n = 4$ items A, B, C, D, with weights $w(i)$ of 2,3,4,1 and associated values $v(i)$ 5,6,7,2.

	A	B	C	D
weight (kg)	2	3	4	1
value (euro)	5	6	7	2

Clearly all the items don't fit in the bag (since 2+3+4+1 > 4), so which items should we choose?

Brute Force Solution to 0/1 Knapsack Problem

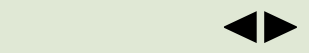
Consider/Enumerate all possibilities:

- Choose just one item: 4 possibilities
- Choose any two items: 6 possibilities
- Choose any three items: 4 possibilities
- Choose four items: 1 possibility

(In actual fact, we don't need to look at all these options: In this example, we can't pick 4 items: We can't even fit any 3 items in the bag.)

For any item, we can either take it or not, i.e. 2 choices. 2 choices each for n items means 2^n possibilities. If for example we had 50 items, 2^{50} is about 1,000,000,000,000,000. Even if we have to consider a fraction of these possibilities, this algorithm would be unworkable for anything but a supercomputer. The Brute Force Solution is $\mathbf{O}(2^n)$.

Greedy Solution to 0/1 Knapsack Problem



Greedy on weight We at each stage pick the lightest item: So we pick D, then A, then terminate with final value of 7 euro in the bag.

Greedy on value We at each stage pick the most valuable item: So we pick C, then terminate with final value of 7 euro in the bag.

Greedy on euro/kg We at each stage pick the most value per kilogram item: So we pick A, then D, then terminate with final value of 7 euro in the bag.

Algorithm Strategies: Recursion

(Sometimes this is viewed as a *methodology* rather than a strategy - i.e. it can be used within other strategies.) Problems which allow a recursive solution have two features:

1. They can be (easily) solved for some small "size" (n) of the problem (typically n is 0 or 1). This is called the base case.
2. The general problem of size n can be broken up (re-written) in terms of smaller problem(s) of size p where $p < n$:
   - Typically p is $n - 1$, or $n/2$, or $n - r$ for some r.
   - The way in which n gets reduced to p must ensure we reach the base case.

   This is called the recursive step.

Recursion - Example 1

Consider the calculation of $n!$ (the factorial of a number):

Base Case Do we know how to solve it for a small value of n? Yes, we know $0! = 1$, or $1! = 1$

Recursive Step Do we know how to write $n!$ in terms of the factorial of a smaller number? Yes - $n!$ is just n multiplied by $(n-1)!$. So we can re-write the problem in terms of one of smaller size p, where in this case p is $n-1$.

We can write the recursive calculation for $n!$ in pseudocode as follows:

```
FACT(i)
IF i=1 THEN FACT ← 1 ELSE FACT ← i*FACT(i-1)
BEGIN
IN(n)
```

```
OUT FACT(n)
END
```

Recursion - Towers of Hanoï

We can denote by `H(i,a,b,c)` our function which will tell us how to move `i` disks from peg `a` to peg `b` (with peg `c` as spare).

Base Case Do we know how to solve it for a small value of n?
Yes, for $n = 1$ we know

$$\texttt{H}(1,\texttt{a},\texttt{b},\texttt{c}) \equiv$$

"Move the top disk from peg a to peg b"

Recursion - Towers of Hanoï Continued

Recursive Step Do we know how to write `H(n,a,b,c)` in terms of some other `H` function(s) `H(p,a,b,c)` with $p < n$? Yes -

$$H(n,a,b,c) \equiv \begin{array}{l} H(n-1,a,c,b) \\ H(1,a,b,c) \\ H(n-1,c,b,a) \end{array}$$

In this recursive step, we re-write the problem of size n in terms of three new problems, of sizes $n-1$, 1, and $n-1$ respectively.

Fibonacci Numbers

The Fibonacci sequence is defined by

$$\begin{array}{rcll} F_n & = & 1 & \text{if } n < 3 \\ & = & F_{n-1} + F_{n-2} & \textit{otherwise} \end{array}$$

which gives us the sequence

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

Note that here, the *definition* of the numbers gives us directly the

base case(s) i.e. $F_1 = 1$ and $F_2 = 1$.

recursive step i.e. $F_n = F_{n-1} + F_{n-2}$. Here we re-write a problem of size n in terms of *two* new problems of size $n - 1$ and $n - 2$ respectively.

Recursion - Fibonacci Numbers

We can denote by `fib(i)` our function which will tell us how to calculate the *i*th Fibonacci Number

Base Case Do we know how to solve it for a small value of *i*? Yes, for $i = 1$ we know $f(1) = 1$. Also for $i = 2$ we know $f(2) = 1$

Recursive Step Do we know how to write `fib(i)` in terms of some other function(s) `fib(p)` with `p < i`? Yes, we can re-write `fib(i)` in terms of two functions `fib(p)` with $p = i - 1$ and $p = i - 2$ respectively:

$$\texttt{fib(i)} = \texttt{fib(i} - 1\texttt{)} + \texttt{fib(i} - 2\texttt{)}$$

Th algorithm in pseudocode for the recursive solution to the Fibonacci problem is:

```
FIB(i)
IF i<3 THEN FIB ← 1
      ELSE FIB ← FIB(i-1) + FIB(i-2)

BEGIN
IN(n)
OUT FIB(n)
END
```

Is this efficient? Let $N(n)$ denote the number of operations (additions) we do in this recursive solution. Then

$$N(n) = 1 + N(n-1) + N(n-2)$$

We can apply this process repeatedly to get

$$
\begin{aligned}
N(n) &= 1 + N(n-1) + N(n-2) \\
&= 1 + [1 + N(n-2) + N(n-3)] + N(n-2) \\
&= 2 + 2N(n-2) + N(n-3) \\
&= 2 + 2[1 + N(n-3) + N(n-4)] + N(n-3) \\
&= 4 + 3N(n-3) + 2N(n-4) \\
&= 4 + 3[1 + N(n-4) + N(n-5)] + 2N(n-4) \\
&= 7 + 5N(n-4) + 3N(n-5) \\
&= 7 + 5[1 + N(n-5) + N(n-6)] + 3N(n-5) \\
&= 12 + 8N(n-5) + 5N(n-6) \\
&= \ldots \\
&= [f(i+2) - 1] + f(i+1)N(n-i) \\
&\qquad + f(i)N(n-i-1) \qquad \text{in the } i\text{th iteration}
\end{aligned}
$$

Note the coefficients here are the fibonacci numbers themselves. When do we terminate (*base case*)? Well, we know N(1) is 0 as is N(2) (no additions required): So we get that this recursive solution is $\mathbf{O}(fib(n))$.
Where does this fit in our list of functions? Consider

1.5^n i.e. $1, 1.5, 2.2, 3.4, 5.1, 7.6, 11.4, 17.1, 25.6, \ldots$

2^n i.e. $1, 2, 4, 8, 16, 32, 64, 128, 256, 512, \ldots$

The fibonacci sequence falls between these two, so we know it is about $\mathbf{O}(a^n)$ where a is some number between 1.5 and 2.

Non- Recursive Fibonacci Solution

```
BEGIN
IN(n)
IF n<3 THEN OUT(1)
      ELSE
      BEGIN
```

```
        a ← 1
        b ← 1
        count ← 2
        WHILE count < n DO
             BEGIN
             f ← a+b
             a ← b
             b ← f
             count ← count+1
             END
        OUT(f)
        END
END
```

Clearly this algorithm is **O**(n) since the while loop runs (approximately) n times.

Divide and Conquer (D&C) Strategy

- Divide a problem of size n into (a number of) smaller parts, of identical type.
- Solve the smaller parts.
- Reconstruct the solution from the parts.
- Often uses recursion (but not necessarily so).

D&C Example 1

The Towers of Hanoï solution uses a divide and conquer strategy (breaking the problem of size n in to two new problems, each of size $n - 1$).

D&C Example 2

In Quicksort, we break a problem of size n in to two new problems, of identical nature, one of size p and the other of size $n - p$.

D&C Example 3

In Binary Search, we break a problem of size n in to one new problem of size $n/2$.

D&C Example 4: The "Fake Coin" Problem

Suppose we have 32 identical coins, of which one is a fake. We know that the fake coin weighs less than the other coins. Given a weighing scales determine which coin is fake, using the least possible number of measurements.

Solution 1 (Brute Force) We just weigh each coin. Or, more specifically, pick one coin which is always on the left side of the balance scales, and put each of the remaining coins, in turn, on the right side.

Solution 2 (D&C)

1. Divide 32 coins in to two piles of 16, put 16 coins on each side of the balance.

2. Pick the pile that is lighter.
3. Divide this pile of 16 in to two piles of 8, and put 8 coins on each side of the balance.
4. Pick the pile that is lighter.
5. Divide this pile of 8 in to two piles of 4, and put 4 coins on each side of the balance.
6. Pick the pile that is lighter.
7. Divide this pile of 4 in to two piles of 2, and put 2 coins on each side of the balance.
8. Pick the pile that is lighter.
9. Divide this pile of 2 in to two piles of 1, and put 1 coin on each side of the balance.
10. Pick the lighter coin.

If we analize these solutions:

Solution 1 (Brute Force) We made 31 measurements. Clearly in the general case this strategy gives $\mathbf{O}(n)$

Solution 2 (D&C) We made 5 measurements. Note that $2^5 = 32$. In the general case, this strategy gives $\mathbf{O}(\log_2 n)$.

Aside: If we had a normal weighing scales

(with a digital readout), the D&C method would necessitate 10 readings in the above example, or in the general case $2 \log_2 n$ readings. But this is still $\mathbf{O}(\log_2 n)$.

D&C Example 5: Fast Multiplication of Large Integers

Consider the multiplication of the numbers 479 and 531. In the "step by step" procedure for calculating the answer (254,349), how many individual multiplications do we need to do? The answer is 9 "single digit" multiplications (+ some extra work with carrying over digits, and adding parts at the end). If we ignore this extra work, the multiplication of two n- digit numbers requires $\mathbf{O}(n^2)$ single- digit multiplications.
Let our 2 n- digit numbers be A and B. We break each number in to two parts by writing

$$A = A_1 10^{n/2} + A_2 \quad B = B_1 10^{n/2} + B_2$$

For example: $A = 1937$, $B = 3832$, then n is 4, A_1 is 19, A_2 is 37, B_1 is 38, B_2 is 32.

D&C Solution 1 We write

$$\begin{aligned} AB &= (A_1 10^{n/2} + A_2)(B_1 10^{n/2} + B_2) \\ &= (A_1 B_1)10^n + (A_1 B_2)10^{n/2} + (A_2 B_1)10^{n/2} + (A_2 B_2) \end{aligned}$$

which means we have to do 4 new multiplications, in each case of two $n/2$ digit numbers (i.e. $A_1 B_1, A_1 B_2, A_2 B_1$ and $A_2 B_2$). Let $N(n)$ denote the number of single digit products we need to calculate to multiply two n-digit numbers. So $N(0) = 0$ and $N(1) = 1$. Using the strategy above, we have

$$\begin{aligned} N(n) &= 4N(n/2) = 4[4N(n/4)] = 16N(n/4) \\ &= 16[4N(n/8)] = 64N(n/8) \\ &= \cdots = 4^k N(n/2^k) \end{aligned}$$

$n/2^k$ is one when $n = 2^k$ or $k = \log_2 n$ so we get the result that

$$N(n) \equiv n^2 = \mathbf{O}(n^2)$$

so this D&C solution is no better than standard multiplication.

D&C Solution 2: The Karatsuba Algorithm Write

$$\begin{aligned} AB &= (A_1 10^{n/2} + A_2)(B_1 10^{n/2} + B_2) \\ &= (A_1 B_1)10^n + (A_2 B_2) \\ &+ [(A_1 + A_2)(B_1 + B_2) - A_1 B_1 - A_2 B_2]10^{n/2} \end{aligned}$$

We have now written the product of two n- digit numbers in terms of 3 products (in blue) of $n/2$ digit numbers. With the assumption that we can

ignore the extra additions / subtractions in the re-writing, this means we have

$$\begin{aligned} N(n) &= 3N(n/2) = 3[3N(n/4)] = 9N(n/4) \\ &= 9[3N(n/8)] = 27N(n/8) \\ &= \cdots = 3^k N(n/2^k) \end{aligned}$$

As before this terminates when $n/2^k = 1$ so we get

$$N(n) \approx 3^{\log_2 n} \approx n^{\log_2 3} \approx \mathbf{O}(n^{1.6})$$

so tha Karatsuba Algorithm beats "standard" multiplication (for large n)

Dynamic Programming Strategy

- The title has nothing to do with (Computer) Programming in the modern sense of the word (Dynamic Programming as a strategy pre- dates computers).

- It is a “bottom up” approach (as opposed to “top down”).
- To solve a problem of size n
  1. First solve an equivalent problem of size 0 or 1, and store the answer in a variable/array/table.
  2. Then solve a problem one step larger, making use of the previous solution
  3. Step by step increase the problem size until we get to the full problem of size n.

Dynamic Programming: Example 1

To calculate e.g. 11!

1. Calculate 0!: 1
2. Use the previous result to calculate 1!: i.e. 1! = (1)0! =1
3. Use the previous result to calculate 2!: i.e. 2! = (2)1! =2
4. Use the previous result to calculate 3!: i.e. 3! = (3)2! =6
5. Use the previous result to calculate 4!: i.e. 4! = (4)3! =24
6. etc.

Dynamic Programming: Example 2: The Knapsack Problem

Recall: We have a bag/container of (weight) capacity K kilograms. We have n distinct items with weights $w(1), w(2), w(3), \ldots w(n)$ (kg) and corresponding values $v(1), v(2), v(3), \ldots v(n)$ (euro). The aim is to put as many items as possible in to our bag, in order to obtain maximum value. Suppose $K = 4$ and

	A	B	C	D
weight (kg)	2	3	4	1
value (euro)	5	6	7	2

The Dynamic Programming approach is firstly to try and solve a much smaller problem, both in terms of the number of items available (in fact we will start assuming zero items available) and in terms of the bag capacity (in fact we will start assuming zero bag capacity), and building up, step by step to larger bag capacity and more items:

no. of items	items	Bag Capacity (kg)				
available	available	0	1	2	3	4
0		0	0	0	0	0
1	A	0	0	5(A)	5(A)	5(A)
2	A,B	0	0	5(A)	6(B)	6(B)
3	A,B,C	0	0	5(A)	6(B)	7(C)
4	A,B,C,D	0	2(D)	5(A)	7(A,D)	8(B,D)

- We work through the table from top left corner to bottom right, going firstly from left to right across each row.
- The numbers in the body of the table are the *euro* values we can get, followed in brackets by the item choices.
- The number at the bottom right corner is the answer (8 euro: choose B,D)

To formalize this: let

PROF(i,j) ≡ Best profit we can obtain from choosing amongst the first i items, with bag capacity j.

With this definition:

- PROF(0,j) = 0 (there are no items to put in our bag).
- PROF(i,0) = 0 (our bag has capacity zero).

When we introduce a new item i (i.e. we go to a new row in the table above), we can either choose the new item or not:

Don't choose the new item i Then PROF(i,j) is just PROF(i-1,j) (i.e. our previous choice for best profit, not including new item i).

Choose the new item i From choosing the item i we get value v(i). But now the current bag capacity j is reduced to j - w(i). With this remaining bag capacity, the best profit we can obtain is $PROF(i-1, j-w(i))$. So our total best profit here is $v(i) + PROF(i-1, j-w(i))$.

Which of these two choices we make depends on which is best, i.e. we should choose the maximum! So we have

$$PROF(i,j) = MAX(PROF(i-1,j), v(i) + PROF(i-1, j-w(i)))$$

Note:

1. The Dynamic Programming Strategy here gives us all the intermediate results (if we wanted them!)
2. The way we order the items is immaterial to the final result, but affects the intermediate results.
3. What if our bag capacity (or item weights) were not integers? Then we need to make more "fine steps", in steps of e.g. 0.1, or 0.01 in the bag size.
4. The total complexity is just the size of the table (i.e. (n)(K)) so the algorithm is **O**(Kn). For example, for 20 items, bag capacity 50, we need to calculate (20)(50) numbers, i.e. 1,000 numbers. A Brute Force solution here requires $2^n = 2^{20} \approx 1,000,000$ calculations.