目录

Complexity continued

Subsets

The subsets of the set *{x,y,z}*, where *x*, *y* and *z* are numbers, are

```
   a[0] [1] [2]             alternatively
1.   x   y   z              x y z
2.       y   z              x y
3.   x       z              x z
4.           z              y z
5.   x   y                  x
6.       y                  y
7.   x                      z
8.
```

The left and right columns have the same subsets, but the order is different.

- the right is the 'natural' way of doing it
- the left is another way of 'thinking' about it

Now focus on the left ordering. It is based on the algorithm:

> *subset = {}*
> *for a_2 in subset && a_2 !in subset {*
>   *for a_1 in subset && a_1 !in subset {*
>   *for a_0 in subset && a_0 !in subset {*
>   *print subset*
>   *}*
>   *}*
> *}*

We can achieve this recursively by calling a function with decreasing *n*

This algorithm generates:

$\{a_2,a_1,a_0\}$ $\{a_2,a_1\}$ $\{a_2,a_0\}$ $\{a_2\}$ $\{a_1,a_0\}$ $\{a_1\}$ $\{a_0\}$ $\{\}$

How do we implement this?

- define an array *in[]* of Booleans to represent whether a number is in the subset
    - *in[1,1,1]* means the subset is $\{a_0,a_1,a_2\}$
    - *in[0,0,1]* means the subset is $\{a_2\}$

How many subsets are there?

- 2^3 if the number of elements is 3

Here is a program that computes all the subsets of an array of elements

切换行号显示

```
// sub.c
// compute the subsets of a hardcoded array

#include <stdio.h>
#include <stdbool.h>

void printSubset(int a[], bool in[], int len) {
    printf("Subset: ");
    for (int i=0; i<len; i++) {
        if (in[i]) {
            printf("%d ", a[i]); // print out the path elements
        }
    }
    putchar('\n');
    return;
}

void subset(int a[], int n, bool in[], int len) {
   // n is decremented, len is constant
   printf("n = %d\n", n);  // this is just debug

   if (n == 0) {
      printSubset(a, in, len);
   }
   else { // n>=1 here
      in[n-1] = true; // element is in subset
      subset(a, n-1, in, len);
      in[n-1] = false; // element is not in subset
      subset(a, n-1, in, len);
   }
   return;
}

int main() {
   int a[] = {5,6,7};  // for demo purposes

   int n = sizeof(a)/sizeof(a[0]); // n is number of elements
   bool in[n]; // array of Booleans, in/out of subset
               // initialised in subset()
   subset(a, n, in, n);

   return 0;
}
```

All the work is done by *subset()*

- it calls itself with *n=n-1* twice
  - corresponding to the left and right branches
    - the left branch with *in[n-1]==true*
    - the right branch with *in[n-1]==false*
- at *0*
  - *in[]* is complete
  - it prints those elements that have *in[] = true*

Output:

```
prompt$ dcc sub.c
prompt$ ./a.out
```

```
n = 3
n = 2
n = 1
n = 0
Subset: 5 6 7
n = 0
Subset: 6 7
n = 1
n = 0
Subset: 5 7
n = 0
Subset: 7
n = 2
n = 1
n = 0
Subset: 5 6
n = 0
Subset: 6
n = 1
n = 0
Subset: 5
n = 0
Subset:
```

The recursive calls to *subset()* are:

```
                n=3
                 /\
                /  \
         7   +/      \-
            /          \
          /              \
        n=2            n=2
         /\              /\
        /  \            /  \
   6  +/    \-        +/    \-
      /      \        /      \
    n=1    n=1    n=1    n==1
     /\      /\      /\      /\
5  +/  \-  +/  \-  +/  \-  +/  \-
   /    \  /    \  /    \  /    \
 n=0  n=0 n=0 n=0 n=0 n=0 n=0  n=0
 567   67  57   7  56   6 5
```

Notice the number of paths = number of leaves = number of subsets = $2^3 = 8$

- makes sense: 3 Booleans, each can be 0/1

How many recursive calls will it make?

- 1 + 2 + 4 + 8 = 15

How many recursive calls will it make for a set of n numbers?

- $1 + 2 + 4 + 8 + ... + 2^n = 2^{n+1} - 1$
  - $2^0 + 2^1 + 2^2 + 2^3 + ... + 2^n = 2^{n+1} - 1$ is a well-known property of geometric series

For example:

- the number of subsets of a set of 20 numbers is $2^{21} - 1 = 2{,}097{,}152 - 1$
- e.g. for 40 numbers 2,199,023,255,552 - 1 (i.e about 2 trillion)

What is its complexity?

Subset sum problem

A very important problem in computer science:

> Given a set of positive integers, and an integer *k*, is there a non-empty subset whose sum is equal to *k*

Example

- The set {2, 4, 6, 8, 10} and *k*=11. *Does any subset sum to k?*
  - The answer is *no*.
- The set {2, 5, 7, 9} and *k*=14.
  - The answer is *yes*: subsets {2,5,7} and {5,9} both sum to 14

The subset algorithm we have used actually generates the subsets of a given set.

- so we won't answer yes/no, we will generate the sets that sum to *k*
- we modify *sub.c* in the following way:
  1. we add a *sum* parameter to *subset()*
     - initialised in *main()* with the value of *k*
  2. we recurse from *n* down to 0
     - crucially, we decrease the sum when we call *subset()*:
       - ***if in[n-1]*==true*, then sum = sum-a[n-1]***
       - ***if in[n-1]*==false*, then sum does not change***
  3. when we reach *n==0* (at the leaf node), its a solution <u>only</u> if *sum==0*

These changes can be found in the following program (comments in upper case).

切换行号显示

```
// subSum.c
// generate all subsets of a hardcoded array that sum to variable 'k'
#include <stdio.h>
#include <stdbool.h>

void printSubset(int a[], bool in[], int len) {
    printf("Subset: ");
    for (int i=0; i<len; i++) {
        if (in[i]) {
            printf("%d ", a[i]); // print out the path elements
        }
    }
    putchar('\n');
    return;
}

void subset(int a[], int n, bool in[], int len, int sum) {
   // n is decremented, len is constant
   // printf("n = %d, sum = %d\n", n, sum); // TOO MUCH OUTPUT GENERATED

   if (n == 0) {
      if (sum == 0) {     // ONLY WANT SUBSETS THAT SUM TO SUM
         printSubset(a, in, len);
      }
   }
   else { // n>=1 here
      in[n-1] = true; // element is in subset
      subset(a, n-1, in, len, sum-a[n-1]); // IF IN, SUBTRACT FROM SUM
      in[n-1] = false; // element is not in subset
```

```
      subset(a, n-1, in, len, sum);        // NOT IN, SO NO CHANGE
   }
   return;
}

int main() {
   int a[] = {2,5,7,9};  // FOR DEMO PURPOSES
   int k = 14;           // THIS IS THE SUM WE WANT

   int n = sizeof(a)/sizeof(a[0]); // n is number of elements
   bool in[n]; // array of Booleans, in/out of subset
               // initialised in subset()
   subset(a, n, in, n, k);  // PASS K TO THE FUNCTION

   return 0;
}
```

Output:

```
prompt$ dcc subSum.c
prompt$ ./a.out
Subset: 5 9
Subset: 2 5 7
```

```
                                sum=14
                                  /\
                              /       \
                     9    +/             \-
                       /                    \
                    /                          \
                sum=5                          sum=14
                 /\                              /\
                /  \                            /  \
         7   +/      \-                      +/      \-
            /          \                    /          \
          /              \                /              \
       sum=-2          sum=5           sum=7           sum=14
         /\              /\              /\              /\
        /  \            /  \            /  \            /  \
  5   +/    \-        +/    \-        +/    \-        +/    \-
      /      \        /      \        /      \        /      \
  sum=-7     -2      0        5      2        7      9        14
     /\      /\      /\      /\      /\      /\      /\      /\
2  +/  \-  +/  \-  +/  \-  +/  \-  +/  \-  +/  \-  +/  \-  +/  \-
   /    \  /    \  /    \  /    \  /    \  /    \  /    \  /    \
  -9   -7 -4   -2  -2   0  3    5  0    2  5    7  7    9  12   14
                        ^          ^
                        |          |
```

Note:

- *sum=0* at an internal node is a subset computation not yet complete
- *sum=0* at a leaf node is a solution
    - two solutions: reading from top-to-bottom
        1. 9 5
        2. 7 5 2

But the highlighted *subset sum problem* statement only wants a yes/no answer

- that's easier: we do not need to know the route to the leaf
    - so remove all references to *in[]*
    - change *subset()* into a Boolean function

切换行号显示

```
// subSumYes.c
// for a hardcoded array and sum k, returns 'yes' if some subset sums to k

#include <stdio.h>
#include <stdbool.h>

bool subset(int a[], int n, int sum) {
   bool retval = false;

   // printf("n = %d, sum = %d\n", n, sum);

   if (n == 0) {
      if (sum == 0) {
         retval = true;
      }
   }
   else { // n>=1 here
      bool with = subset(a, n-1, sum-a[n-1]);
      bool wout = subset(a, n-1, sum);
      retval = with || wout;
   }
   return retval;
}

int main() {
   int a[] = {2,5,7,9};  // FOR DEMO PURPOSES
   int k = 14;          // THIS IS THE SUM WE WANT

   int n = sizeof(a)/sizeof(a[0]); // n is number of elements
   if (subset(a, n, k)) {
      printf("Yes\n");
   }
   else {
      printf("No\n");
   }
   return 0;
}
```

Output:

```
prompt$ dcc subSumYes.c
prompt$ ./a.out
Yes
```

What is the complexity of this algorithm?

- the algorithm to generate all the subsets was $O(2^n)$
- with the additional work of computing sums, which is linear, this algorithm is $O(n*2^n)$

Towers of Hanoi

Described on site Wikipedia: Towers of Hanoi

The number of moves to solve the Towers of Hanoi is given by the formula (2^n -1) where *n* is the number of disks

- For example:
  - to move 2 disks takes **3** moves
  - to move 100 disks takes **1,267,650,600,228,229,401,496,703,205,375** moves (i.e. 10^{30} moves)

To solve the problem, you need to express the *n-disk* problem in terms of the *n-1* disk problem

- called the *inductive* case

then define a base case

- from the base case, the inductive case can be used to compute a solution

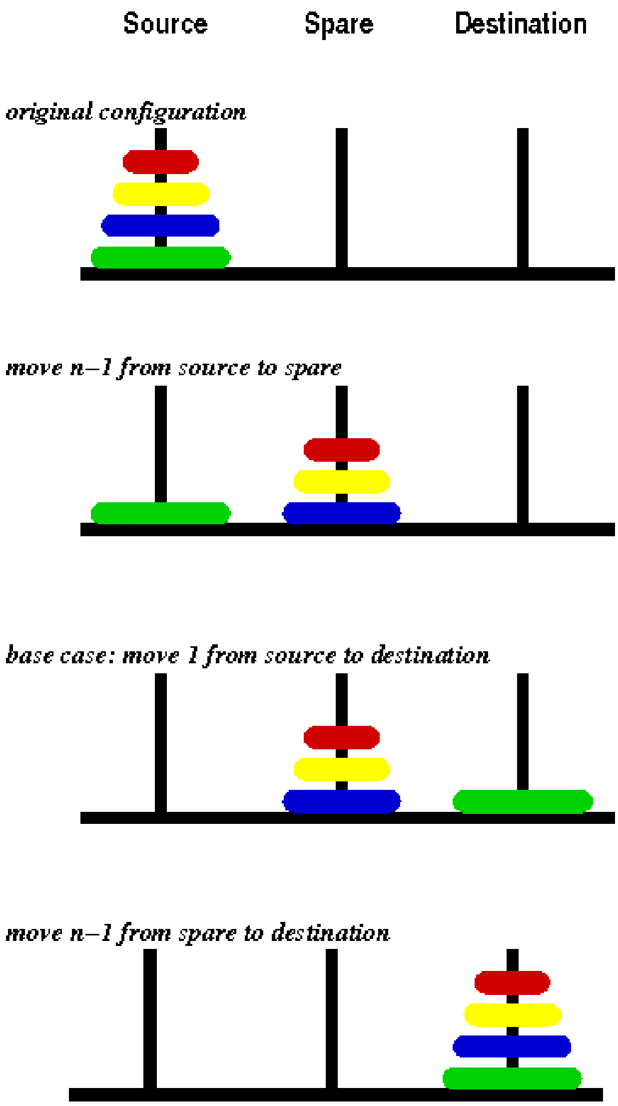
So how do you move n *disks from the source pin A to the destination pin C, with only pin B as spare?*

1. move *n-1* disks to the spare pin
   - (this leaves just one disk remaining on the source pin)
2. move that remaining disk to the destination pin
3. move the *n-1* disks on the spare pin onto the destination pin

This is 'divide and conquer':

- translate the larger problem into smaller problems that are solvable
- the smaller solutions are then combined to form a solution of the original problem

See how it works if *n=4*



Expressed as a recursive algorithm:

- to move *n* disks from pin A to pin C using pin B as spare:
  - if n is 1, just do it (this is the base case)
  - otherwise the recursive case
    - move *n-1* disks from A to B using C as spare

        - move *1* disk from A to C using B as spare
        - move *n-1* disks from B to C using A as spare
- to program, you do not need any data structures
    - you only want to know the moves

Algorithmic complexity

Let $T(n)$ be the minimum number of moves needed to solve the puzzle with n disks.

- We can see for example:
    - $T(0) = 0$
    - $T(1) = 1$
    - $T(2) = 3$
    - $T(3) = 7$
    - $T(4) = 15$

Let's try to derive a general formula to express this.

- The algorithm above:
    - twice moves $(n-1)$ disks from one pin to another and
    - makes 1 additional move
- We therefore have the *recurrence relation*:
    - $T(n) = T(n-1) + 1 + T(n-1) = 2*T(n-1) + 1$

Let's check this recurrence relation works (given $T(0) = 0$):

- $T(1) = 2*T(0) + 1 = 1$
- $T(2) = 2*T(1) + 1 = 3$
- $T(3) = 2*T(2) + 1 = 7$
- and so on, agreeing with the numbers above.

Let's simplify and write $S(n) = T(n) + 1$

- we can express the recurrence relation in terms of $S(n)$
    - $S(n) = T(n) + 1 = (2*T(n-1) + 1) + 1 = 2*T(n-1) + 2 = 2*S(n-1)$

We can solve this by *unrolling*:

- $S(n) = 2*S(n-1) = 2*2*S(n-2) = 2*2*2*S(n-3) = ... = 2^n*S(n-n) = 2^n$
    - note $S(0) = T(0) + 1 = 1$

Substituting that value of $S(n)$ into $S(n) = T(n) + 1$

- $T(n) = S(n) - 1 = 2^n - 1$ for $n >= 0$

Hence the complexity of the Hanoi algorithm is exponential, $O(2^n)$

Ackermann's function

Originally conceived in 1928.

```
A(0, n):= n+1 for n>=0
A(m, 0):= A(m-1, 1) for m>0
A(m, n):= A(m-1, A(m, n - 1)) for m>0, n>0
```

Often used as a 'mad' compiler benchmark

- will always overflow or burn out something

Performance

- uses addition and subtraction only
- grows faster than an exponential or even multiple exponential function
- recursion is almost unlimited

For example:

```
The expansion of A(4,3) cannot be recorded in the known physical
universe.
```

$$\begin{aligned}
A(4,3) &= A(3,A(4,2)) \\
&= A(3,A(3,A(4,1))) \\
&= A(3,A(3,A(3,A(4,0)))) \\
&= A(3,A(3,A(3,A(3,1)))) \\
&= A(3,A(3,A(3,A(2,A(3,0))))) \\
&= A(3,A(3,A(3,A(2,A(2,1))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(2,0)))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(1,1)))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(0,A(1,0))))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(0,A(0,1))))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(0,2)))))) \\
&= A(3,A(3,A(3,A(2,A(1,3))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(1,2)))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,A(1,1))))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,A(0,A(1,0)))))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,A(0,A(0,1)))))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,A(0,2))))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,3)))))) \\
&= A(3,A(3,A(3,A(2,A(0,4))))) \\
&= A(3,A(3,A(3,A(2,5)))) \\
&= \ldots \\
&= A(3,A(3,A(3,13))) \\
&= \ldots \\
&= A(3,A(3,65533)) \\
&= \ldots \\
&= A(3,2^{65536}-3) \\
&= \ldots \\
&= 2^{2^{65536}}-3.
\end{aligned}$$

A(m,n)	**n = 0**	**n = 1**	**n = 2**	**n = 3**	**n = 4**	**n = 5**
m = 0	1	2	3	4	5	6
m = 1	2	3	4	5	6	7
m = 2	3	5	7	9	11	13

m = 3	5	13	29	61	125	253
m = 4	2**2**2 -3	2**2**2**2 -3	2**2**2**2**2 -3	2**2**2**2**2**2 -3	2**2**2**2**2**2**2 -3	...

- A(4,0) = 2**2**2 -3 = 2^4 -3 = 13
- A(4,1) = 2**2**2**2 -3 = 2^{16} -3 = 65533
- A(4,2) = 2**2**2**2**2 -3 = 2^{65536} -3 = $10^{19727.78}$
- A(4,3) = 2**2**2**2**2**2 -3 = 2**(2^{65536}) -3 = ...

Curiously:

- rows *m=0*, *m=1* and *m=2* all grow ?
- row *m=3* grows ?
- row *m=4* grows ?

Complexity2 (2019-07-04 10:05:58由AlbertNymeyer编辑)