目录

Program analysis

A program may execute for a small value of some input parameter *n*:

- but what happens to its performance when *n* gets large?
- does it *scale*?

Scalability refers to the:

- space it requiries to execute, or
- time it takes to execute

although conventionally its refers to execution time

Donald Knuth (1932 - .)

- called the "father" of the (complexity) analysis of algorithms
- author of the seminal 3-volume book *The Art of Computer Programming*
- *Professor Emeritus of the Art of Computer Programming* at Stanford University
- also creator of the TeX computer typesetting system and the Computer Modern family of typefaces

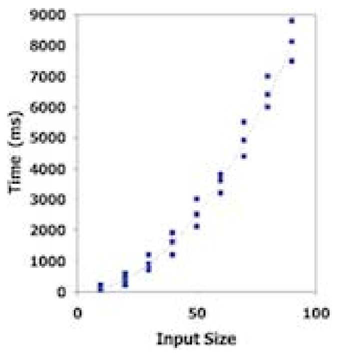
A complexity analysis is part of design.

First get the design right, then concern yourself with optimisation

Donald Knuth (ACM Journal Computing Surveys, 1974)

Empirical complexity analysis

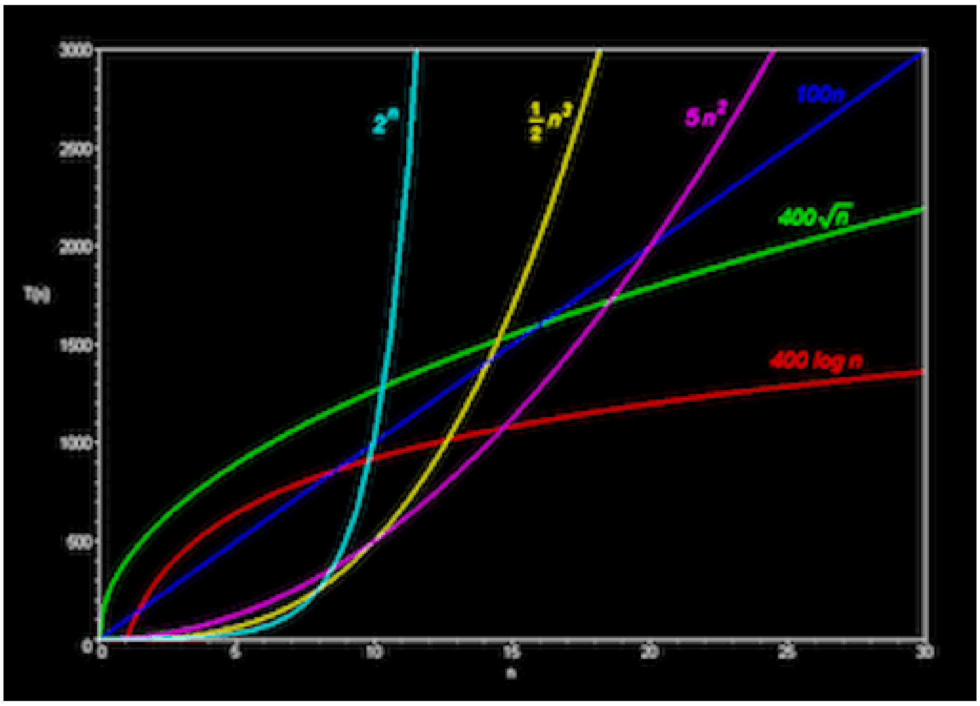
1. Write program that implements an algorithm
2. Run program with inputs of varying size and composition
3. Measure the actual running time
4. Plot the results



Limitations:

1. must code the algorithm, which may be difficult
2. results depends on the input we give it (may be different elsewhere)
3. comparing algorithms/designs requires the same hardware and operating system be used
4. How do you know when the input (*n*) is big enough to tell you what kind of algorithm you have?
    - if a test does not terminate, how long will you need to wait?

Here is a plot of the various complexity classes of algorithms.



Say you don't know how large *n* will get. You test all the algorithms for:

- *n=5*
    - *red* looks bad

- *yellow* looks good
- *turquoise* looks great

Based on these results, which algorithm would you pick?

- *n=10*
    - *yellow* still looks good
    - *red* is still bad (takes twice as long *yellow*)
    - *turquoise* is now even worse than *red*

Based on these new results, which algorithm would you pick?

Knowing what happens asymptotically, ***which algorithm should you have picked?***

- clearly *red* is much much faster than *yellow*, and *turquoise* will take forever

Complexity analysis

Has 3 main features:

1. it is an abstraction of the performance
    - it ignores details that don't affect the performance significantly
2. it is asymptotic
    - it is valid and useful only when the size of the input gets large
3. it classifies algorithms into a hierarchy (based on execution time or space)
    - from fastest to slowest, or smallest to largest

Linear search

Consider the following algorithm

切换行号显示

```
1 int linearSearch(int a[], int n, int key) {
2     int found = -1; // returned value if the 'key' is not found
3     for (int i=0; i<n && (found == -1); i++) {
4       if (key == a[i]) {
5           found = i;
6       }
7     }
8     return found;   // =key if found, =-1 if not found
9 }
```

- searches for a *key* in an array *a[]* of size *n*
- returns *-1* if not found, or the index of the key if found

*What is the **worst-case** behaviour here?*

- the key is not in the list and the search checks all *n* elements

Complexity analysis of linear search

How much work does the linear search algorithm do in the worst case?

- 1 time line 2
- *n+1* times line 3
- *n* times line 4
- 0 times line 5
- 1 time line 8

Hence $total = c_{L2} + c_{L3}*(n+1) + c_{L4}*n + c_{L8}$

$= (c_{L3} + c_{L4}) * n + c_{L2} + c_{L3} + c_{L8}$
$= constant_1 * n + constant_2$

When n gets large, the linear term dominates, and $constant_2$ is not significant

- hence the worst-case execution time is proportional to n
- it is classed as a *linear* algorithm

Being linear, you can predict:

- if the input increases by a factor k, the program will take a factor k longer to execute

Polynomial complexity

Most serious algorithms have computational complexity larger (worse) than linear.

The execution profile of a *quadratic* algorithm can be written

- constant$_1$ * n^2 + constant$_2$ * n + constant$_3$

Algorithms can also be

- *cubic* (goes like n^3)
- *quartic* (like n^4)
- *quintic* (n^5)
- ...

In fact, any algorithm with execution profile n^c, where $c>1$ is a constant is called

- *polynomial*

For example, $n^{1.0000001}$ is polynomial.

The dominant term in the execution profile is always enough to determine the computational complexity

- Consider:

n	$n^2 + n$	n^2
1	2	1
10	110	100
100	10,100	10,000
1,000	1,001,000	1,000,000
10,000	100,010,000	100,000,000
100,000	10,000,100,000	10,000,000,000
...	...	...

Asymptotically, that is as n gets very large, we see that

- $n^2 + n$ converges to n^2

In general

- in linear algorithms we ignore constant factors
- in quadratic algorithms we ignore linear and constant factors
- etc

Linear and polynomial algorithms are generally referred to as 'fast'

- can execute in reasonable time even for large *n*

Exponential and factorial complexity

Exponential algorithms

- the size of the space increases by a multiplicative factor with *n*
    - e.g. compute all the subsets of an alphabet of *n* characters
- execution profile is dominated by the term c^n for a constant $c>1$
- *c* could be the *exponential constant* 2.718, or 2, or 10, it does not matter

Factorial algorithms

- execution profile is dominated by the term *n!*
    - sliding tile puzzle
- these are 'nasty' problems

All these algorithms are *intractible* for large *n*

Logarithmic complexity

The best class of problems.

- essentially *halve the space* for each iteration
    - 'divide and conquer' techniques are often logarithmic
    - e.g. a binary search of a list
- performance is given by *log(n)*
    - often $log_2(n)$ (binary logarithm)
    - or $log_{10}(10)$ (normal logarithm)
    - or $log_e(n)$ (natural logarithm)

It does not matter which logarithm is used in complexity

$\log_e(n)$, where *e* = 2.718281828...	$\log_2(n)$
often denoted ln(n)	denoted lg(n)
is the power to which *e* has to be raised to equal n, so $e^{\ln(n)}=n$	is the power to which 2 has to be raised to equal n, so $2^{\lg(n)}=n$
ln(e) = 1	lg(2) = 1
ln(100) ~= 4.6	lg(100) ~= 6.6
ln(1000000) ~= 14	lg(1000000) ~= 20

Conversion: *lg(n) = lg(e)*ln(n) ~= 1.44*ln(n)*

We saw linear search checks every element in a list in the worst case

- the list may be sorted or unsorted

Binary search works only on a sorted list

- every iteration it halves the list
  - it searches in one of the halves and
    - then re-halves the list and searches again and
      - then re-halves the list again ...
- until there is just 1 element

It is an example of a 'divide and conquer' approach

- controversial
  - the problem is divided into 2 parts, but only one part is solved

Example of binary search

- find a name in a very long alphabetic list

Given a sorted array of *n* elements

切换行号显示

```
int binarySearch(int a[], int n, int key) {
   int found = 0;
   int low = 0;
   int mid;
   int high = n-1;
   while (low <= high && !found) {
      mid = (low+high)/2;
      if (a[mid] == key) {
         found = 1;
      }
      else if (a[mid] < key) {
         low = mid + 1;
      }
      else {
         high = mid - 1;
      }
   }
   return found;
}
```

Comparing linear and binary search:

- Binary search requires the input data to be sorted; linear search does not
- Binary search requires an ordering comparison; linear search only requires equality comparisons
- Binary search requires random access to the data; linear search requires only sequential access (this can be important)

So, binary search is 'fussy': it requires a list of sorted, random-access elements

Why should you use it when you can?

- *The number of comparisons for both methods:*

# of elements	Binary Search	Linear Search
n	*O(log(n))*	*O(n)*
1	1	1
100	8	100
10,000	15	10,000
1,000,000	21	1,000,000
10,000,000,000	35	10,000,000,000

Complexity analysis of binary search

Assume the list is length *n*, and the algorithm takes *k* iterations

- in each iteration the list is divided into 2
  - after 1 iteration, the length is $n/2^1$
  - after 2 iterations, the length is $n/2^2$
  - after *k* iterations, the length is $n/2^k$

In the worst case, the element is not in the list and

- in the k^{th} iteration, the length is 1

So

- $n/2^k = 1$
- simplifying: $n = 2^k$
- take the log of both sides: $log_2(n) = log_2(2^k)$
- simplifying: $log_2(n) = k * log_2(2)$
- because $log_a(a) = 1$ we can write $log_2(n) = k$

Hence

- $k = log_2(n)$
- i.e. the time complexity of binary search is *O(log(n))*

The big picture of computational classes

Let's assume that for input $n = 1$ an algorithm takes *1 nanosecond* (i.e. 10^{-9} seconds).

(Note, one *millisecond* (i.e. 10^{-3}) = one million nanoseconds (i.e. $10^{-3} = 10^6 * 10^{-9}$).

Explanation: if an algorithm is n^2 and it takes 1 ns to execute $n = 1$, then

- $n=10$ takes 10^2ns
- $n=50$ takes 50^2ns
- and so on

Time taken for increasing *n* for algorithms of different complexities:

	log(n)	**linear**	**nlog(n)**	**polynomial**	**exponential**	**factorial**	**astronomical**
n	log(*n*)	*n*	*n*log(*n*)	$n^{c=2}$	$(c=2)^n$	*n!*	n^n or worse
10	3ns	10ns	33ns	100ns	1024ns	3 million ns	10 million ns
..							
50	6ns	50ns	282ns	2,500ns	13 days	10^{40} *AOU*	
..							
100	7ns	100ns	644ns	10,000ns	1000 *AOU*	*gasp*	
..							

1000	10ns	1,000ns	1,000ns	million ns	*gasp*	*gasp*	
..							
10000	13ns	10,000ns	130,000ns	0.1 second	*gasp*	*gasp*	

where *AOU* = Age Of Universe = time since the Big Bang = 14 billion years = 14 * 10^9 years

Observations:

- anything lower than polynomial is good
- polynomial is okay if *n* is not any bigger than millions (say)
- exponential and factorial is impossible for $n > 10$

Note:

- it is rare to find practical exponential algorithms, but they do exist
- the above is the (theoretical) **worst-case behaviour**
- in practice, the input data may rarely be *worst-case*
    - **best-case** data may be much better than *worst-case*

Big oh Notation

Used to bound the worst-case behaviour for given input *n*:

- It is the asymptotic behaviour, i.e. if *n* is large
- It can be used for space (i.e. memory) as well. Time taken is most common.
    - For example: an algorithm may take $O(n)$ time and use $O(n^2)$ memory, or $O(2^n)$ time and $O(1)$ memory, or ...

Running time analysis may be *best-case*, *worst-case*, or *average-case*

- usually different
- e.g. best-case behaviour may be fast and average-case and worst-case behaviour may be poor

Example: if we are sorting a list of *n* numbers, then

- best-case is when the numbers are already sorted: $O(n)$
- worst-case is when the numbers are reverse sorted: $O(n^2)$

Generall, the **complexity** of a problem is

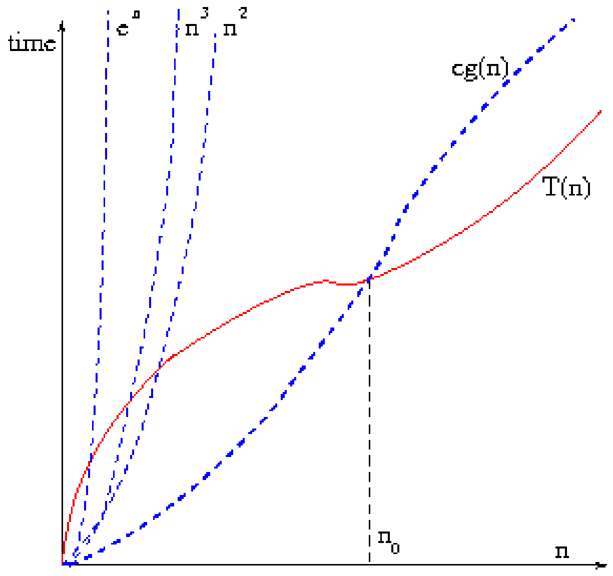
- a bound on the **worst-case runtime** of the best algorithm to solve the problem
- worst-case runtime you get with the worst possible input

Definition

- A function $T(n) \in O(g(n))$ means there are positive constants *c* and n_0

such that $0 \le T(n) \le cg(n)$ for all $n \ge n_0$.

Pictorially:



We can ignore the constants. For example:

- an algorithm with asymptotic runtime $T(n)=100n^2$ will be faster than an algorithm with runtime n^3 for any value of $n>100$
    - ... but of course if you know that $n<<100$ then the cubic algorithm will be (much) faster

Including constants in Big-oh notation is pointless

- there will always be a point n_0 that makes one complexity larger than the other (from that point on)

This definition does not say which function $g(n)$ we should pick.

- $O(g(n))$ is actually a set because
    - there are an infinite number of functions $g(n)$ that can make this true
    - that's why we say $T(n) \in O(g(n))$

Example: consider $T(n) = 2n$

- we could say $g(n)=3n$, i.e. $T(2n) \in O(3n)$
- we could say $g(n)=n^2$ or $g(n)=n^3$ or $g(n)=e^n$ or even $g(n)=n!$
- so $O(g(n))$ comprises all functions $g(n)$ whose asymptotic behaviour is not faster than $T(n)$

Example: consider $T(n) = 2n^2 + 7n -10$

- $T(n) \in O(n^2)$ because $2n^2 + 7n -10 < 3n^2$ for $n>5$
- but technically also $T(n) \in O(n^3)$ because n^2 is a member of $O(n^3)$

We are actually interested in the tightest asymptotic upper bound

- i.e. the member of $O(g(n))$ that performs fastest
- so we don't consider the higher classes

O(1) versus constant time

An algorithm is **O(1)** if its asymptotic runtime is bounded by a constant

- e.g. the algorithm
    - if n is even then $even(n) = n$ else $even(n) = n + n$

- this algorithm sometimes uses 1 operation, sometimes 2 operations
- it is *O(1)* because the runtime is bounded

An algorithm is **constant-time** if its asymptotic runtime is constant for all *n*

- e.g. access an element in an array is a constant-time algorithm
- e.g. *push* and *pop* operations on a stack
- the *even()* function above is not constant-time (the execution time varies)

O(1) is a 'weaker' condition than constant time

- a constant-time is *O(1)*, but an *O(1)* algorithm may not be constant time

Big omega Notation

An algorithm is $T(n) \in \Omega(g(n))$ if there exists constants c and n_0 such that

> **$T(n) >= cg(n)$ for all $n >= n_0$**

T(n) is hence bound underneath by some function *cg(n)* for large *n*

- *cg(n)* is also called the asymptotic lower bound

Not so interesting facts:

- *n* is sandwiched above *log(n)* and below *nlog(n)*: i.e.
  - $n = \Omega(log(n))$
  - $n = O(nlog(n))$
- n^2 is sandwiched above *nlog(n)* and below n^3
  - $n^2 = \Omega(nlog(n))$
  - $n^2 = O(n^3)$
- *n!* is sandwiched above 2^n and below n^n
  - $n! = \Omega(2^n)$
  - $n! = O(n^n)$

Big oh, omega and theta video

http://www.youtube.com/watch?v=6Ol2JbwoJp0

How good are the bounds?

We do not always know how good the upper bound is. For example:

- algorithm A $\in O(n^3)$ steps
- algorithm B $\in O(n^2)$ steps

Algorithm A may take much fewer than n^3 steps

- because it was difficult to analyze, and it is actually linear!
- or because a better estimate is mathematically not possible

Algorithm B may be exactly n^2

- this bound is tight because it is an easy algorithm

So here, although A has a 'slower' Big oh, in reality, A is always faster

```
Big oh gives you an upper bound, but it does not tell you how good that bound
is
```

If you know that a Big oh complexity is an over-estimate, you really need the Big omega as well

- this gives you a lower bound

But the lower bound (Ω) may have the same problem as the upper bound

- the bound is very approximate
    - it could be Ω(1) for example, which does not help at all

```
So, while Big-Omega gives you a lower bound, it does not tell you how good
that bound is
```

Ideally, what you want is the tightest possible bounds:

- e.g. $T(n) \in O(log(n))$ and $T(n) \in \Omega(log(n))$ or
- e.g. $T(n) \in O(n)$ and $T(n) \in \Omega(log(n))$

and certainly not:

- $T(n) \in O(2^n)$ and $T(n) \in \Omega(1)$

which basically says that

- the asymptotic execution time of your program is somewhere between 1 nanosecond and billions of years

Examples of different algorithmic complexities

Complexity	Description
1	Execution time does not depend on the input size
log(n)	Very slow increase in running time as n increases
	Whenever n doubles, the running time increases by a constant
n	Linear algorithm are optimal if you need to process n inputs
	Whenever n doubles, then so does the running time
$n$log(n)	Like linear algorithms, 'linearithmic' algorithms scale to huge problems
	Whenever n doubles, the running time more (but not much more) than doubles
n^2	Quadratic algorithms are practical on relatively small problems
	Whenever n doubles, the running time increases fourfold
n^3	Cubic algorithms are practical on only small problems
	Whenever n doubles, the running time increases eightfold
2^n	Exponential algorithms are generally not practical
	Whenever n doubles, the running time squares!
n!	Factorial algorithms are very bad in general to compute
	Whenever n increases by 1, the running time increases by a factor of n

Examples:

- **constant**: the running time of the statement will not change in relation to *n*

  ```
  make a cup of coffee at breakfast
  ```

  or

  切换行号显示

  ```
  i = a[n]; // note that the value of 'n' makes no difference

  ```

  or even

  切换行号显示

  ```
  for (i=0; i<1000; i++)
     for (j=0; j<5000; j++)
        // perform some statements that run in constant time

  ```

- **linear**: the running time of the loop is directly proportional to *n*. When *n* doubles, so does the running time

  ```
  reading a book
  ```

  or

  切换行号显示

  ```
  for (int i = 0; i < n; i++) {
     statement;
  }
  ```

- **quadratic**: the running time of the two loops is proportional to n^2. When *n* doubles, the running time increases by *n * n*.

  切换行号显示

  ```
  for (int i = 0; i < n; i++) {
     for (int j = 0; j < n; j++) {
        ...
     }
  }
  ```

- **logarithmic**: the running time of the algorithm is proportional to the number of times *n* can be divided by 2. This is because the algorithm divides the working area in half with each iteration.

  切换行号显示

  ```
  while (low <= high) {
     mid = ( low + high ) / 2;

     if (target < list[mid]) {
        high = mid - 1;
     }
     else if (target > list[mid]) {
        low = mid + 1;
     }
     else break; // not my program, do not do this
  }
  ```

- **nlog(n)**: merge sort. The split of the list in two leads to logarithmic steps (*2 * O(log(n))*). The combining of the 2 halves takes *O(n)* time (note the recursion).

切换行号显示

```
if (low < high) {
  mid = (low+high)/2;         //
  call mergesort(low,mid);    // O(log(n)) time
  call mergesort(mid+1,high);// O(log(n)) time
  merge(low,mid,high);        // combine the 2 halves, takes time
O(n)
}
```

- **n!**: compute all the tours of *n* cities

```
- I can start in any of the n cities...
- I can then select any of n-1 cities to visit next...
- Then any of n-2 cities...
- ...
- Finally I am left with only 1 city to visit
```

  The total number of different tours is *n*(n-1)*(n-2)*...*1*, which is *n!*.
  - if *n=10* then there are *3,628,800* possible tours

Recursion and complexity

Problems with using recursion:

- **space**: a disadvantage in using recursion is that extra memory is needed to call a function
  - a *stack-frame* is created in memory for each call
    - it contains the return address, and the function's local variables, and may be quite large.
- **efficiency**: a recursive algorithm may re-do the same computation **many** times
  - and **many** can be a very large number

Fibonacci Numbers

The Fibonacci numbers are given by:

- *fib(n) = fib(n-1) + fib(n-2)*
- *fib(0) = 0* and *fib(1) = 1*

0	1	2	3	4	5	6	7	8	9	10	...
0	1	1	2	3	5	8	13	21	34	55	...

- Wikipedia: Fibonacci numbers
- can be computed iteratively and recursively

Fibonacci using iteration

切换行号显示

```
// fibit.c: compute the Fibonacci number of a command line argument
iteratively
#include <stdio.h>
#include <stdlib.h>
int fibit(int n){
   int prev = -1;
   int curr = 1;
   for (int i=0; i<=n; i++) {
      int sum = prev + curr;
      prev = curr;
      curr = sum;
   }
12     return curr;
13 }
14
15 int main(int argc, char *argv[]) {
16     int n;
17     if ((argc > 1) && (sscanf(argv[1], "%d", &n) == 1)) {
18         printf("Fibonacci number %d is %d\n", n, fibit(n));
19     }
20     return EXIT_SUCCESS;
21 }
```

- program will print the n^{th} Fibonacci number, where *n* is given on the command line:
- *how does it work?*

i	0	1	2	3	4	5	6	7	8	9	10	...
prev	**-1**	1	0	1	1	2	3	5	8	13	21	...
curr	**1**	0	1	1	2	3	5	8	13	21	34	...
sum	0	1	1	2	3	5	8	13	21	34	55	...

- complexity?
  - the loop in *fibit()* is executed *n+1* times to computer *fibit(n)*
    - *fibit(0)*, *fibit(1)*, *fibit(2)*... *fibit(n)* are each calculated just once
  - a complexity analysis of the iterative Fibonacci fragment:

line	number of iterations
5	1
6	1
7	n+2
8	n+1
9	n+1
10	n+1
12	1

    - *Total time = $(C_{L7}+C_{L8}+C_{L9}+C_{L10})*n$ + constant*
    - hence *O(n)*
- *speed?*

```
prompt$ time ./fibit 20
Fibonacci number 10 is 6765
real    0m0.001s
user    0m0.000s
sys     0m0.001s

prompt$ time ./fibit 30
Fibonacci number 20 is 832040
real    0m0.001s
user    0m0.001s
sys     0m0.000s

prompt$ time ./fibit 40
Fibonacci number 30 is 102334155
real    0m0.001s
user    0m0.000s
sys     0m0.000s

prompt$ time ./fibit 44
Fibonacci number 44 is 701408733
real    0m0.001s
user    0m0.000s
sys     0m0.000s
```

- notice that the *user* time is 0 ...
  - this means the time is too short to measure

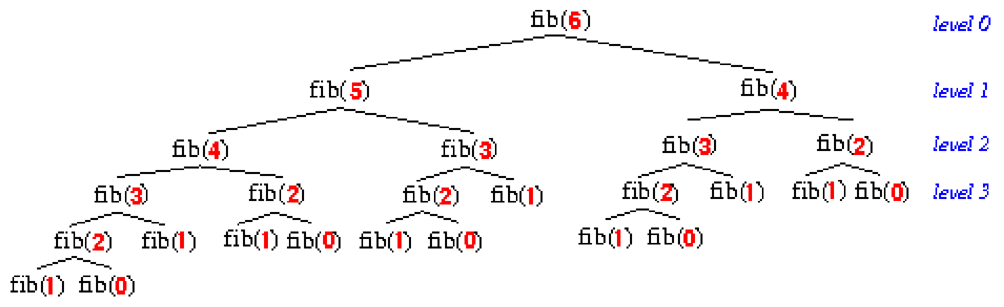
Fibonacci using recursion

切换行号显示

```c
// fibre.c: compute the Fibonaaci number of a command line argument recursively
#include <stdio.h>
#include <stdlib.h>
int fibre(int n){
   if (n == 0) return 0;          // 2 base cases, each with its own 'return'
   if (n == 1) return 1;          // for illustrative purposes only
   return fibre(n-1) + fibre(n-2); // now recurse
}

int main(int argc, char *argv[]) {
   int n;
   if ((argc > 1) && (sscanf(argv[1], "%d", &n) == 1)) {
      printf("Fibonacci number %d is %d\n", n, fibre(n));
   }
   return EXIT_SUCCESS;
}
```

Complexity analysis

- e.g., consider *fibre{6)*



- massive repetition here
  - look at the number of nodes on each level:
    - 2^0 at the root, 2^1 at level 1, 2^2 at level 2, 2^3 at level 3, ...
    - grows by a factor 2 each level, so at level *n*, we have 2^n
    - *gee!*, this is exponential!
      - ***... but the iterative version above was linear ... ?***

How many recursive calls does it actually make?

n	fib(n)	#calls
4	3	9
10	55	177
20	6,765	21,891
30	832,040	2,692,538

- increase *n* by 10, the number of calls increases by roughly 10^2

Recursive Fibonacci is extremely inefficient:

- each call requires a stack-frame to be created
- most calls are a repeat of an earlier call

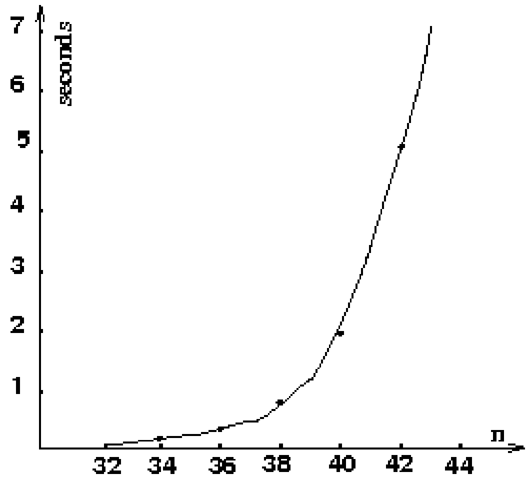
- (note: this program is often used to *stress test* the operating-system stack)

What about speed?

```
prompt$ time ./fibre 20
Fibonacci number 20 is 6765
real    0m0.001s
user    0m0.001s
sys     0m0.000s

prompt$ time ./fibre 30
Fibonacci number 30 is 832040
real    0m0.021s
user    0m0.019s
sys     0m0.000s

prompt$ time ./fibre 40
Fibonacci number 40 is 102334155
real    0m2.044s
user    0m2.038s
sys     0m0.000s

prompt$ time ./fibre 44
Fibonacci number 44 is 701408733
real    0m13.320s
user    0m13.266s
sys     0m0.000s
```

We see

- *fibre(44)* takes 13secs
  - remember *fibit(44)* took ~0secs
- between *n=40* and *n=44* the performance deteriorated rapidly
  - the 'exponential decay' has really started



By the way, a 32bit int only allows the Fibonacci numbers to about 46 to be computed

Fibonacci is bad for recursion, but the problem could be fixed quite simply by adding 'memory' to *fibre()*

Fibonacci using recursion and dynamic programming

Use *dynamic programming* to improve the performance

- the idea is to *trade space for time*
  - place the computed values *fibre(k)* in an array for re-use later
  - this array is called a **cache**
    - the length of the cache is *n*

        - initialize the whole cache to *-1* indicating not defined
    - when a fib number is needed, check the cache
        - if defined, use that number, so no re-computation

Complexity

- when we compute *fibre(n) = fibre(n-1) + fibre(n-2)*
    - both right-hand side terms are available
    - no terms are re-computed
    - each *fibre(n)* can be computed in constant time (like the iterative version)
- to compute *fibre(n)* for all *n* hence requires *O(n)*

切换行号显示

```
// fibrear.c: compute the Fibonacci number of a command line argument
// the number should be between 0 and 46

#include <stdio.h>
#include <stdlib.h>

#define MAXFIB 46

int fibrear(int n, int cache[]) {
   // THIS IS NON-CONFORMING CODE TO MAKE THE BASE CASE CLEAR
   if (cache[n] >= 0) {
      return cache[n];                      // base cases plus more
   }
   if (cache[n-1] < 0) {
      cache[n-1] = fibrear(n-1, cache);
   }
   return cache[n-1] + fibrear(n-2, cache); // recursive case
}


int main(int argc, char *argv[]) {
   int cache[MAXFIB+1];
   int n;
   if ((argc > 1) && (sscanf(argv[1], "%d", &n) == 1) && n>=0 && n<=MAXFIB) {
      cache[0] = 0;                   //---
      cache[1] = 1;                   //   | prepare the cache
      for (int i=2; i<=MAXFIB; i++) { //   |
         cache[i] = -1;               //   |
      }                               //---
      printf("Fibonacci number %d is %d\n", n, fibrear(n, cache));
   }
   if (n < 0 || n > MAXFIB) {
      fprintf(stderr, "Number out of bounds\n");
   }
   return EXIT_SUCCESS;
}
```

The output is shown below, with extra print statements:

- notice each fib number is being called once, and is re-used thereafter

```
prompt$ ./fibrear 6
calling cache[6]
calling cache[5]
calling cache[4]
calling cache[3]
calling cache[2]
calling cache[0]
REUSING cache[0]
calling cache[1]
REUSING cache[1]
calling cache[2]
```

```
REUSING cache[2]
calling cache[3]
REUSING cache[3]
calling cache[4]
REUSING cache[4]
Fibonacci number 6 is 8
```

Speed?

```
prompt$ time ./fibrear 20
Fibonacci number 20 is 6765
real    0m0.001s
user    0m0.000s
sys     0m0.001s

prompt$ time ./fibrear 30
Fibonacci number 30 is 832040
real    0m0.001s
user    0m0.000s
sys     0m0.001s

prompt$ time ./fibrear 40
Fibonacci number 40 is 102334155

real    0m0.001s
user    0m0.000s
sys     0m0.000s
```

The use of dynamic programming has changed the performance from exponential to linear!

.... to be continued

Complexity (2019-07-01 17:44:16由AlbertNymeyer编辑)