Recursive Functions & Algorithm Analysis

CIS2520

Lab5

Tao Xu

Outline

1. Recursion
2. Basic algorithm analysis
3. Common programming mistakes

Recursive Calls

- Call itself (directly or indirectly)
- Terminating condition
- Moves ‘closer’ to terminating condition
- May use stack (memory) to keep intermediate results (return values)

Recursion vs. Iteration

- Readability
- Efficiency in time and memory consumption
- Transformable to each other

Length of a list in C

```
/* Iteration */
int length_I(node* p)
{
   int countNodes = 0;
   while (p)
   {
      countNodes++;
      p = p->next;
   }
   return countNodes;
}
```

```
/* Recursion */
int length_R(node* p)
{
    if (!p)
         return 0;
    return (1 + length_R(p->next));
}
```


When Recursion Is NOT Good

Fibonacci Numbers:

$fib_0 = 0$

$fib_1 = 1$

$fib_n = fib_{n-1} + fib_{n-2}$ for $n > 1$

```
int fib (int n) {
    if (n <= 1) return n;
    return  (fib(n – 1) + fib (n – 2)); /* doubly recursive */
}
```

inefficient: values repeatedly calculated, then 'forgotten'

When Iterative Is Better

```
int fib (int n) {
    int i, x, y, z;
    if (n <= 1) return n;
    i = 1; x = 1; y = 0;
    while (i != n) {
        z = x;
        i++;
        x = x + y;
        y = z;
    }
    return x;
}
```


When Iteration is NOT Good

Iterative forward traversing a (singly) linked list is easy:

```
void traverse (node* p) {
   while (p) {
         process(p->data); /* assume a process function */
         p = p->next;
   }
}
```

BUT

Iterative backward traversing is **hard**
(no pointers, so need to *stack* return pointers)

When Recursion Is Better

Recursive backward traversing a (singly) linked list is easy:

```
void reverseTraverse (node* p) {
   if (p) {
         reverseTraverse(p->next);
         process(p->data);
   }
}
```


Algorithm Analysis

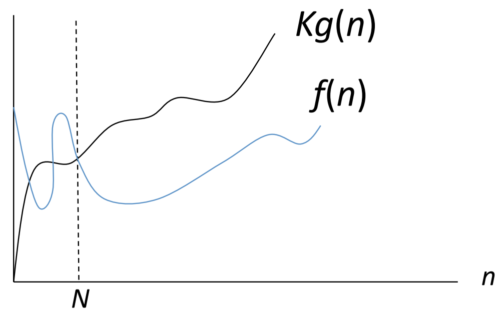
- Time complexity
- Space complexity
- Theoretical rather than empirical
- Notation
  - n (problem size or input size)
  - $f(n)$ (worst time or number of steps to solve the problem)

Big Oh Notation

Let *f* and *g* be two functions. We say that

f is O(g)

if there are positive real number K and positive integer N such that $f(n) <= Kg(n)$ for all integers $n >= N$.



f is O(g)

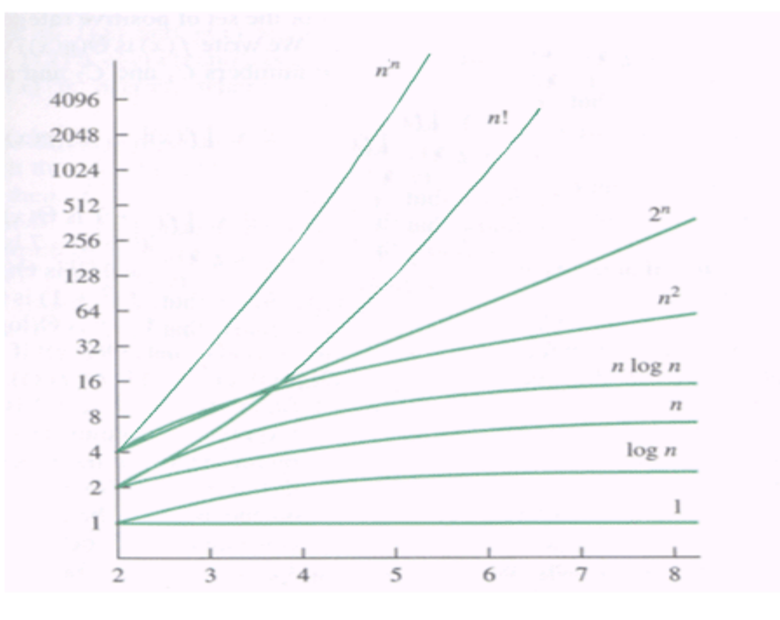
Big Oh Notation (cont'd)

- Capture the behavior of functions for large values of n.
- Also called ***asymptotic upper bound***.
- The tightest upper bound preferred

Big-Oh Rules

- If f is a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
    1. Drop lower-order terms
    2. Drop constant factors
- Use the tightest possible class of functions
    – Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
    – Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Growth Rate of Functions



Examples

- 7n-2 is O(n)

  Proof: If K = 7 and N = 1 then

  7n-2 <= Kn for any n>= N

- $3n^3 + 20n^2 + 5$ is $O(n^3)$

  Proof: If K = 4 and N = 21 then

  $3n^3 + 20n^2 + 5 <= Kn^3$ for any n>=N

- 3 log n + 5 is O(log n)

  Proof: If K = 8 and N = 2 then

  3 log n + 5 <= K log n for any n>=N

Examples (cont'd)

- log(n+1) is O(logn)

  Proof: log(n+1)<=Klogn equivalent to $n+1 <= n^K$;

  If K=2 and N=2 then

  $n^K >= 2n >= n+1$ for any n>=N

Makefiles for Debug & Release

– Use variables (Macros)

make options=y

makefile

…

CFLAGS = <common settings>

ifdef options

CFLAGS += -g -DDEBUG

endif

…gcc $(CFLAGS) …

– if, ifndef, ifeq…

Common Issues

- THINK before coding!
  - Understand and follow the specifications (interfaces)
    - E.g., FreeStudent(), Destroy(),…
  - Figure out potential traps
- Programming skills
  - Stack vs. heap memory
    - E.g., NameOfStudent()
  - The mechanism of making function calls
  - The use of directives for compilers
    - E.g., #ifdef, …

Common Issues (cont'd)

- Useful tips
  - Pass pointers as arguments rather than structures
  - Keep the logic as simple as possible
  - Sacrifice efficiency for readability (for beginners)
  - ......