Other Asymptotic Notations and Recursion

Subhabrata Samajder

IIIT, Delhi

Winter Semester,
10th March, 2023

Recap: Algorithms

- **Algorithm:** It is a finite sequence of elementary operations with the objective of performing some (computational) task.

Recap: Algorithms

- **Algorithm:** It is a finite sequence of elementary operations with the objective of performing some (computational) task.
    - **Elementary operations:** Arithmetic and logical operations.
    - **Finiteness:** It must stop.

Recap: Algorithms

- **Algorithm:** It is a finite sequence of elementary operations with the objective of performing some (computational) task.

- **Input and Output:** Can take *several* inputs but produces a *single* output.

- **Efficiency:** Requiring little 'resources'.
  - **Resources:** Time and space.

Recap: Algorithms

- **Algorithm:** It is a finite sequence of elementary operations with the objective of performing some (computational) task.

- **Input and Output:** Can take *several* inputs but produces a *single* output.

- **Efficiency:** Requiring little 'resources'.
  - **Resources:** Time and space.
  - Time $\propto$ Size of Inputs.

Recap: Algorithms

- **Algorithm:** It is a finite sequence of elementary operations with the objective of performing some (computational) task.

- **Input and Output:** Can take *several* inputs but produces a *single* output.

- **Efficiency:** Requiring little 'resources'.
    - **Resources:** Time and space.
    - Time $\propto$ Size of Inputs.
    - **Size of Inputs:** Function from set of all possible inputs to $\mathbb{N} \cup \{0\}$.
    - But for the same input size n, $t(n)$ varies across different inputs!

Recap: Algorithms

- **Two Possible Way Outs:**
  - Worst-case Complexity
  - Average-case Complexity

Recap: Algorithms

- **Two Possible Way Outs:**
  - Worst-case Complexity
  - Average-case Complexity

- **Asymptotic Comparison:** Big-oh notation.

Other Asymptotic Notations

Ω Notation

Definition

If there exist constants c and N, such that for all $n > N$ the number of steps $T(n)$ required to solve the problem for input size n is at least $cg(n)$, i.e.,

$$T(n) \geq cg(n),$$

then we say that $T(n) = \Omega(g(n))$.

Ω Notation

Definition

If there exist constants c and N, such that for all $n > N$ the number of steps $T(n)$ required to solve the problem for input size n is at least $cg(n)$, i.e.,

$$T(n) \geq cg(n),$$

then we say that $T(n) = \Omega(g(n))$.

- **Example:** $n^2 = \Omega(n^2 - 100)$, $n = \Omega(n^{0.9})$.

Ω Notation

Definition

If there exist constants c and N, such that for all $n > N$ the number of steps $T(n)$ required to solve the problem for input size n is at least $cg(n)$, i.e.,

$$T(n) \geq cg(n),$$

then we say that $T(n) = \Omega(g(n))$.

- **Example:** $n^2 = \Omega(n^2 - 100)$, $n = \Omega(n^{0.9})$.
- The Ω notation thus correspond to the "$\geq$" relation.

Θ Notation

Definition

If a certain function $f(n)$ satisfies both $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$, then we say that $f(n) = \Theta(g(n))$.

- **Example:** $5n \log_2 n - 10 = \Theta(n \log n)$.

Θ Notation

Definition

If a certain function $f(n)$ satisfies both $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$, then we say that $f(n) = \Theta(g(n))$.

- **Example:** $5n \log_2 n - 10 = \Theta(n \log n)$.
- The constants used to prove the $\mathcal{O}$ part and the Ω part need not be the same.

Small-oh or Little-oh Notation

- The $\mathcal{O}, \Omega$ and Θ correspond (loosely) to "$\leq$", "$\geq$", and "$=$".
- Sometimes we need notation corresponding to "$<$" and "$>$".

Definition

We say that $f(n) = o(g(n))$ (pronounced "$f(n)$ is little oh of $g(n)$") if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Small-oh or Little-oh Notation

- The $\mathcal{O}, \Omega$ and Θ correspond (loosely) to "$\leq$", "$\geq$", and "$=$".
- Sometimes we need notation corresponding to "$<$" and "$>$".

Definition

We say that $f(n) = o(g(n))$ (pronounced "$f(n)$ is little oh of $g(n)$") if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

Example: $n/\log_2 n = o(n)$, but $n/10 \neq o(n)$.

ω Notation

Definition

Similarly, we say that $f(n) = \omega(g(n))$ (small omega) if

$$g(n) = o(f(n)).$$

In other words, $f(n) = \omega(g(n))$ means that for any positive constant c, there exists a constant N, such that

$$0 \leq cg(n) < f(n)$$

for all $n \geq N$. The value of N must not depend on n, but may depend on c.

Recursion: A Recap

Recursion

Definition

A “function” is said to be recursive if it calls itself, either directly or indirectly.

Recursion

Definition

A "function" is said to be recursive if it calls itself, either directly or indirectly.

- In its simplest form, the idea of recursion is straight-forward.

Recursion

Definition

A "function" is said to be recursive if it calls itself, either directly or indirectly.

- In its simplest form, the idea of recursion is straight-forward.

Example:

```
#include <stdio.h>

int main(void) {
   printf(" The universe is never ending! ");
   main();
   return 0; }
```

Recursion

Definition

A "function" is said to be recursive if it calls itself, either directly or indirectly.

- In its simplest form, the idea of recursion is straight-forward.

Example:

```
int sum(int n) {
  if (n <= 1)
    return n;
  else
    return (n + sum(n - 1)); }
```

Example: sum(4)

Example: sum(4)

Function call	Value returned		
sum(1)	1		
sum(2)	2 + sum(1)	or	2 + 1
sum(3)	3 + sum(2)	or	3 + 2 + 1
sum(4)	4 + sum(3)	or	4 + 3 + 2 + 1

- The base case is considered,

Example: sum(4)

Function call	Value returned		
sum(1)	1		
sum(2)	2 + sum(1)	or	2 + 1
sum(3)	3 + sum(2)	or	3 + 2 + 1
sum(4)	4 + sum(3)	or	4 + 3 + 2 + 1

- The base case is considered,
- then working out from the base case, the other cases are considered.

Recursion (Cont.)

Simple recursive routines follow a standard pattern!

Recursion (Cont.)

Simple recursive routines follow a standard pattern!

- The problem is broken into sub-problem(s).

Recursion (Cont.)

Simple recursive routines follow a standard pattern!

- The problem is broken into sub-problem(s).
- Typically, there is a base case (or cases).

Recursion (Cont.)

Simple recursive routines follow a standard pattern!

- The problem is broken into sub-problem(s).
- Typically, there is a base case (or cases).
- The base case is tested for upon entry to the function.

Recursion (Cont.)

Simple recursive routines follow a standard pattern!

- The problem is broken into sub-problem(s).
- Typically, there is a base case (or cases).
- The base case is tested for upon entry to the function.
- Then there is a general recursive case in which one of the variables, often an integer, is passed as an argument in such a way as to ultimately lead to the base case.

Recursion (Cont.)

Simple recursive routines follow a standard pattern!

- The problem is broken into sub-problem(s).
- Typically, there is a base case (or cases).
- The base case is tested for upon entry to the function.
- Then there is a general recursive case in which one of the variables, often an integer, is passed as an argument in such a way as to ultimately lead to the base case.

Example: $\mathrm{sum}()$

Recursion (Cont.)

Simple recursive routines follow a standard pattern!

- The problem is broken into sub-problem(s).
- Typically, there is a base case (or cases).
- The base case is tested for upon entry to the function.
- Then there is a general recursive case in which one of the variables, often an integer, is passed as an argument in such a way as to ultimately lead to the base case.

Example: $\mathrm{sum}()$

- $\mathrm{sum}(\mathrm{n}) = n + (n-1) + \cdots + 1 = n + \mathrm{sum}(\mathrm{n}-1)$.

Recursion (Cont.)

Simple recursive routines follow a standard pattern!

- The problem is broken into sub-problem(s).
- Typically, there is a base case (or cases).
- The base case is tested for upon entry to the function.
- Then there is a general recursive case in which one of the variables, often an integer, is passed as an argument in such a way as to ultimately lead to the base case.

Example: $\mathrm{sum}()$

- $\mathrm{sum(n)} = n + (n-1) + \cdots + 1 = n + \mathrm{sum(n-1)}$.
- The variable n is reduced by 1 each time until
- the base case with $n = 1$ is reached.

Examples: Factorial

$$0\,! = 1, \quad n\,! = n(n-1)\cdots 3\cdot 2\cdot 1 \quad \text{for } n > 0$$

or equivalently,

$$0\,! = 1, \quad n\,! = n\cdot((n-1)\,!) \quad \text{for } n > 0$$

Examples: Factorial

$$0\,! = 1, \quad n\,! = n(n-1)\cdots 3\cdot 2\cdot 1 \quad \text{for } n > 0$$

or equivalently,

$$0\,! = 1, \quad n\,! = n\cdot((n-1)\,!) \quad \text{for } n > 0$$

For example: $5\,! = 5\cdot 4\cdot 3\cdot 2\cdot 1 = 120.$

Examples: Factorial

$$0\,! = 1, \quad n\,! = n(n-1)\cdots 3\cdot 2\cdot 1 \quad \text{for } n > 0$$

or equivalently,

$$0\,! = 1, \quad n\,! = n\cdot((n-1)\,!) \quad \text{for } n > 0$$

For example: $5\,! = 5\cdot 4\cdot 3\cdot 2\cdot 1 = 120.$

- **Base Case:** $0! = 1$ and $1! = 1$.

Examples: Factorial

$$0\,! = 1, \quad n\,! = n(n-1)\cdots 3\cdot 2\cdot 1 \quad \text{for } n > 0$$

or equivalently,

$$0\,! = 1, \quad n\,! = n\cdot((n-1)\,!) \quad \text{for } n > 0$$

For example: $5\,! = 5\cdot 4\cdot 3\cdot 2\cdot 1 = 120.$

- **Base Case:** $0! = 1$ and $1! = 1$.
- **Recursive Case:** $n! = n\cdot(n-1)!$.

Factorial: Recursive Version

```
int RecFactorial (int n) {        /* recursive version */
   if (n <= 1)
      return 1;
   else
      return (n * RecFactorial (n - 1)); }
```

Factorial: Recursive Version

```
int RecFactorial (int n) {        /* recursive version */
  if (n <= 1)
    return 1;
  else
    return (n * RecFactorial (n - 1)); }
```

Note:

- Works properly within the limits of integer precision.

Factorial: Recursive Version

```
int RecFactorial (int n) {        /* recursive version */
  if (n <= 1)
    return 1;
  else
    return (n * RecFactorial (n - 1)); }
```

Note:

- Works properly within the limits of integer precision.
- Factorial function grows very rapidly!

Factorial: Recursive Version

```
int RecFactorial (int n) {        /* recursive version */
   if (n <= 1)
      return 1;
   else
      return (n * RecFactorial (n - 1)); }
```

Note:

- Works properly within the limits of integer precision.
- Factorial function grows very rapidly!
- RecFactorial(n) runs only a few values of n (upto $n = 12$!!).

Factorial: Recursive Version

```
int RecFactorial (int n) {        /* recursive version */
   if (n <= 1)
      return 1;
   else
      return (n * RecFactorial (n - 1)); }
```

Note:

- Works properly within the limits of integer precision.
- Factorial function grows very rapidly!
- RecFactorial(n) runs only a few values of n (upto $n = 12!!$).
- For $n > 12$, incorrect values are returned.

Factorial: Recursive Version

```
int RecFactorial (int n) {        /* recursive version */
   if (n <= 1)
      return 1;
   else
      return (n * RecFactorial (n - 1)); }
```

Note:

- Works properly within the limits of integer precision.
- Factorial function grows very rapidly!
- RecFactorial(n) runs only a few values of n (upto $n = 12$!!).
- For $n > 12$, incorrect values are returned.
- This type of programming error is common!!

Factorial: Recursive Version

```
int RecFactorial (int n) {        /* recursive version */
  if (n <= 1)
    return 1;
  else
    return (n * RecFactorial (n - 1)); }
```

Take Away: Functions that are logically correct can return incorrect values if the logical operations in the body of the function are beyond the integer precision available to the system!!

Factorial: Iterative Version

Note:

- As in sum(), RecFactorial() activates n nested copies of the function before returning level by level to the original call.

Factorial: Iterative Version

Note:

- As in sum(), RecFactorial() activates n nested copies of the function before returning level by level to the original call.
- Thus n function calls are used for this computation.
- This is "costly"!!

Factorial: Iterative Version

Note:

- As in sum(), RecFactorial() activates n nested copies of the function before returning level by level to the original call.
- Thus n function calls are used for this computation.
- This is "costly"!!

"What to do?"

Factorial: Iterative Version

Note:

- As in sum(), RecFactorial() activates n nested copies of the function before returning level by level to the original call.
- Thus n function calls are used for this computation.
- This is "costly"!!

"What to do?"

Way out: Rewrite them as iterative functions.

Factorial: Iterative Version

Note:

- As in sum(), RecFactorial() activates n nested copies of the function before returning level by level to the original call.
- Thus n function calls are used for this computation.
- This is "costly"!!

"What to do?"

Way out: Rewrite them as iterative functions.

```
int IterFactorial (int n) {        /* iterative version */
    int product = 1;

    for ( ; n > 1; --n)
        product *= n;
    return product; }
```

Factorial: Iterative Version

Note:

- As in sum(), RecFactorial() activates n nested copies of the function before returning level by level to the original call.
- Thus *n* function calls are used for this computation.
- This is "costly"!!

"What to do?"

Way out: Rewrite them as iterative functions.

```
int IterFactorial (int n) {        /* iterative version */
    int product = 1;
    for ( ; n > 1; - -n)
        product *= n;
    return product; }
```

IterFactorial(n): Takes only 1 function call.

Efficiency Considerations

Many algorithms have both iterative and recursive formulations.

Efficiency Considerations

Many algorithms have both iterative and recursive formulations.

“Then why bother?”

Efficiency Considerations

Many algorithms have both iterative and recursive formulations.

“Then why bother?”

- Recursion is more elegant.

Efficiency Considerations

Many algorithms have both iterative and recursive formulations.

“Then why bother?”

- Recursion is more elegant.
- Requires fewer variables to make the same calculation.

Efficiency Considerations

Many algorithms have both iterative and recursive formulations.

"Then why bother?"

- Recursion is more elegant.
- Requires fewer variables to make the same calculation.
- Takes care of its bookkeeping by stacking arguments and variables for each invocation.

Efficiency Considerations

Many algorithms have both iterative and recursive formulations.

“Then why bother?”

- Recursion is more elegant.
- Requires fewer variables to make the same calculation.
- Takes care of its bookkeeping by stacking arguments and variables for each invocation.
- *This stacking of arguments, while invisible to the user, is still costly in time and space.*

Books Consulted

1. *Introduction to Algorithms: A Creative Approach* by Udi Manber.

2. *Introduction to Algorithms* by Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein.

Thank You for your kind attention!