ALGORITMOS Y ESTRUCTURAS DE DATOS

ANÁLISIS DE ALGORITMOS

ALGORITMOS

- ¿Qué es un algoritmo?

ALGORITMOS

- ¿Qué es un algoritmo?
  - Un proceso o conjunto de reglas que se siguen en operaciones de calculo y solución de problemas, particularmente en un computador o máquina.

ALGORITMOS

- ¿Qué es un algoritmo?
  - Un proceso o conjunto de reglas que se siguen en operaciones de calculo y solución de problemas, particularmente en un computador o máquina.
- ¿Cómo se demuestra que un algoritmo es correcto y que realiza la tarea para la que fue diseñado?

ALGORITMOS

- ¿Qué es un algoritmo?
  - Un proceso o conjunto de reglas que se siguen en operaciones de calculo y solución de problemas, particularmente en un computador o máquina.
- ¿Cómo se demuestra que un algoritmo es correcto y que realiza la tarea para la que fue diseñado?
- ¿Cómo se calcula el tiempo de computo de un algoritmo?

ALGORITMOS

- ¿Qué es un algoritmo?
  - Un proceso o conjunto de reglas que se siguen en operaciones de calculo y solución de problemas, particularmente en un computador o máquina.
- ¿Cómo se demuestra que un algoritmo es correcto y que realiza la tarea para la que fue diseñado?
- ¿Cómo se calcula el tiempo de computo de un algoritmo?
- ¿Cómo reducir los recursos usados por un algoritmo?

ANÁLISIS DE ALGORITMOS

- Determinar al complejidad computacional de algoritmos
  - Tiempo de ejecución (time complexity)
  - Almacenamiento (space complexity)
  - Otros recursos (bandwidth, throughput, etc.)
- Reducir la complejidad computacional
- Creación de algoritmos eficientes

CRECIMIENTO DE FUNCIONES

- Def. 1: T(N) = O(f(N))
- Def. 2: T(N) = OMEGA(g(N))
- Def. 3: T(N) = THETA(h(N))
- Def. 4: T(N) = o(p(N))
- Def. 5: T(N) = omega(q(N))
- Utiles para establecer un orden entre funciones
- Informan sobre el orden relativo y no absoluto

CRECIMIENTO DE FUNCIONES

- Ejemplo: 1000N vs. N^2
  - Use notación anterior para expresar el crecimiento relativo de estas funciones
    - para N chico 1000N es más grande
    - para N grande N^2 es más grande
    - punto de cambio de régimen: N = 1000

CRECIMIENTO DE FUNCIONES

- Otro ejemplo: N^3 vs. N^2
  - Use todas la definiciones anteriores para expresar el crecimiento relativo de estas funciones

REGLAS SOBRE EL CRECIMIENTO DE FUNCIONES

A. Suma
B. Producto
C. Polinomios
D. Potencias de logaritmos
E. Constantes
F. Suma de órdenes

CASO PROMEDIO Y PEOR CASO

- Necesitamos un modelo para analizar algoritmos (RAM)
- **Modelo de computación**:
  1. Un computador que ejecuta instrucciones secuencialmente
  2. Operaciones básicas: adición, multiplicación, comparación y asignación
  3. Toma una unidad de tiempo hacer cualquiera de estas operaciones

CASO PROMEDIO Y PEOR CASO

- **Modelo de computación**:

   4. Las unidades de información tienen un tamaño fijo (`int`, `float`, `string`, `bool`, `etc.`)

   5. El computador tiene memoria infinita

- Desventajas:

   - No todas las operaciones toman el mismo tiempo

   - Acceso a memoria infinita es lento

CASO PROMEDIO Y PEOR CASO

- Definimos dos funciones
  - La función caso promedio $T_{ave}(N)$, que refleja el comportamiento típico del algoritmo
  - La función peor caso $T_{worst}(N)$, que garantiza el rendimiento (performace) para cualquier input
  - La función mejor caso $T_{best}(N)$, que muestra el rendimiento (performace) óptimo para cualquier input

CASO PROMEDIO Y PEOR CASO

- Típicamente se satisface que
  - $T_{ave}(N) <= T_{worst}(N)$
  - $T_{worst}(N) < T_{best}(N)$
  - $T_{best}(N) <= T_{ave}(N)$
- Ejemplo: *maximum subsequence sum problem*
  - Tarea: analizar los distintos algoritmos para resolver este problema

LIMITACIONES DEL ANÁLISIS DE ALGORITMOS

- Usualmente se have una sobre estimación cuando se trabaja con el peor caso
- En algunos casos puede ocurrir que $T_{ave}(N) << T_{worst}(N)$
  - No hay una conexión formal entre los dos casos
- Para algunos algoritmos complejos $T_{worst}(N)$ se logra solo con inputs malos
- Obtener $T_{ave}(N)$ es muy difícil, solo nos queda $T_{worst}(N)$

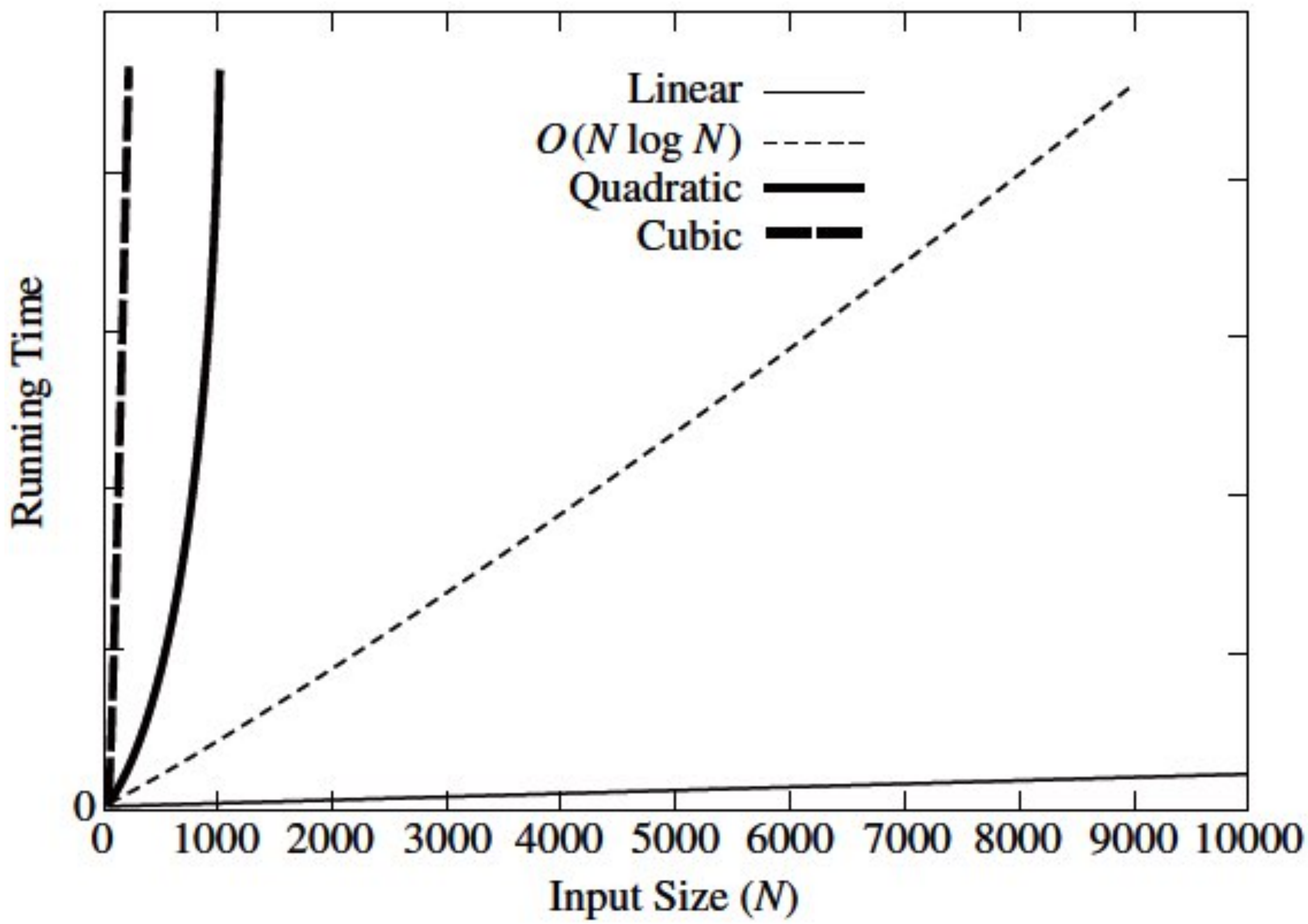
TIEMPO DE EJECUCIÓN (RUNNING TIME)

- Típicamente se analiza el tiempo de ejecución y no la complejidad espacial o demás recursos
- Factores que afectan el tiempo de ejecución
  - El compilador y optimizaciones
  - El computador y su aritmética
  - El algoritmo *per se*
  - ✓ El input (entrada del algoritmo)

MAXIMUM SUBSEQUENCE SUM PROBLEM

	Algorithm Time			
Input Size	1 $O(N^3)$	2 $O(N^2)$	3 $O(N \log N)$	4 $O(N)$
$N = 100$	0.000159	0.000006	0.000005	0.000002
$N = 1{,}000$	0.095857	0.000371	0.000060	0.000022
$N = 10{,}000$	86.67	0.033322	0.000619	0.000222
$N = 100{,}000$	NA	3.33	0.006700	0.002205
$N = 1{,}000{,}000$	NA	NA	0.074870	0.022711

TIEMPO DE EJECUCIÓN PARA VARIOS ALGORITMOS



CÓMO CALCULAR EL TIEMPO DE EJECUCIÓN

1. Empíricamente: ejecutando el código varias veces y sacado promedios de tiempos
2. Usando el modelo de computación y hacerlo de forma analítica
   - No formal ni riguroso (muy complejo)
   - Siguiendo las reglas para el cálculo de tiempos de ejecución

REGLAS PARA CALCULAR TIEMPOS DE EJECUCIÓN

1. <u>For loop</u>: el tiempo de calculo del for es el tiempo de calculo del bloque de instrucciones veces el numero de iteraciones
2. <u>Bucles anidados</u>: el tiempo de ejecución es el tiempo de ejecución del bloque veces el producto de los tamaños de los bucles
3. <u>Instrucciones consecutivas</u>: su tiempo de ejecución se suma
4. <u>If/else</u>: su tiempo de ejecución no es nunca mayor al del test más el bloque de instrucciones más largo

REGLAS PARA CALCULAR TIEMPOS DE EJECUCIÓN

- Ejemplo: Estime el tiempo de ejecución

```
FUNCTION sum:
 INPUT: integer n >= 0
 OUTPUT: integer partialSum
 USAGE: res = sum(n)
BEGIN

 partialSum = 0
 FOR i = 0, n:
  partialSum += i * i * i
  i++

 RETURN partialSum

END // sum
```

REGLAS PARA CALCULAR TIEMPOS DE EJECUCIÓN

- Ejemplo: Estime el tiempo de ejecución

```
int sum(int n)
{
    int partialSum;

    partialSum = 0

    for(int i = 1; i <= n; ++i) {
        partialSum += i * i * i;
    }

    return partialSum;
}
```

REGLAS PARA CALCULAR TIEMPOS DE EJECUCIÓN

- Ejemplo: Estime el tiempo de ejecución

```
FUNCTION sum:
  INPUT: integer n >= 0
  OUTPUT: integer partialSum
  USAGE: res = sum(n)
BEGIN

  partialSum = 0
  FOR i = 0, n:
    partialSum += i * i * i
    i++

  RETURN partialSum

END // sum
```

Pseudocódigo

```cpp
int sum(int n)
{
    int partialSum;

    partialSum = 0

    for(int i = 1; i <= n; ++i) {
        partialSum += i * i * i;
    }

    return partialSum;
}
```

C++ código

SOLUCIÓN 1: MAXIMUM SUBSEQUENCE SUM PROBLEM

```
//Cubic maximum contiguous subsequence sum algorithm
int maxSubSum1(const vector<int> & a)
{
    int maxSum = 0;

    for(int i = 0; i < a.size(); ++i)
        for(int j = i; j < a.size(); ++j)
        {
            int thisSum = 0;

            for(int k = i; k <= j; ++k) thisSum += a[k];

            if(thisSum > maxSum) maxSum = thisSum;
    }
    return maxSum;
}
```

SOLUCIÓN 2: MAXIMUM SUBSEQUENCE SUM PROBLEM

```
//Quadratic maximum contiguous subsequence sum algorithm
int maxSubSum2(const vector<int> & a)
{
    int maxSum = 0;

    for(int i = 0; i < a.size(); ++i) {
        int thisSum = 0;

        for(int j = i; j < a.size(); ++j) {
            thisSum += a[j];

            if(thisSum > maxSum) maxSum = thisSum;
        }
    }
    return maxSum;
}
```

SOLUCIÓN 3: MAXIMUM SUBSEQUENCE SUM PROBLEM

```
//Recursive maximum contiguous subsequence sum algorithm
int maxSumRec(const vector<int> & a, int left, int right)
{
    if(left == right)
        if(a[left] > 0) return a[left];
    else return 0;

    int center = (left + right) / 2;
    int maxLeftSum = maxSumRec(a, left, center);
    int maxRightSum = maxSumRec(a, center + 1, right);

    int maxLeftBorderSum = 0, leftBorderSum = 0;
    for(int i = center; i >= left; --i) {
        leftBorderSum += a[i];
        if(leftBorderSum > maxLeftBorderSum)
            maxLeftBorderSum = leftBorderSum;
    }
```

SOLUCIÓN 3: MAXIMUM SUBSEQUENCE SUM PROBLEM

```
    int maxRightBorderSum = 0, rightBorderSum = 0;
    for(int j = center + 1; j <= right; ++j) {
        rightBorderSum += a[j];
        if(rightBorderSum > maxRightBorderSum)
            maxRightBorderSum = rightBorderSum;
    }

    return max3(maxLeftSum, maxRightSum,
                maxLeftBorderSum + maxRightBorderSum);
}

int maxSubSum3(const vector<int> & a)
{
    return maxSumRec(a, 0, a.size() - 1);
}
```

SOLUCIÓN 4: MAXIMUM SUBSEQUENCE SUM PROBLEM

```
//Linear-time maximum contiguous subsequence sum algorithm
int maxSubSum4(const vector<int> & a)
{
    int maxSum = 0, thisSum = 0;

    for(int j = 0; j < a.size(); ++j) {
        thisSum += a[j];

        if(thisSum > maxSum) maxSum = thisSum;
        else if(thisSum < 0) thisSum = 0;
    }

    return maxSum;
}
```