Instituto Politécnico Nacional

Escuela Superior de Cómputo

Complejidad espacial y temporal

Unidad de aprendizaje: Analisis de algoritmos

Grupo: 3CM3

Alumnos:
López Manríquez Ángel

M. en C.:
Franco Martinez Edgardo Adrian

22 de junio de 2019

Índice

1. Eficiencia de un algoritmo **2**
1.1. Eficiencia en los bucles . . . 2
1.1.1. bucles lineales . . . 2
1.1.2. Bucle logaritmico . . . 3
1.1.3. Bucles anidados . . . 3

2. Complejidad temporal y espacial **5**
2.1. Algoritmo 1 . . . 5
2.2. Algoritmo 2 . . . 5
2.3. Algoritmo 3 . . . 5
2.4. Algoritmo 4 . . . 6
2.5. Algoritmo 5 . . . 6

3. Aproximación de una función que cuente el número de "prints" **7**
3.1. Algoritmo 6 . . . 7
3.2. Algoritmo 7 . . . 9
3.3. Algoritmo 8 . . . 10

4. Analisis de casos **11**
4.1. Algoritmo 9 . . . 11
4.2. Algoritmo 10 . . . 12
4.3. Algoritmo 11 . . . 13
4.4. Algoritmo 12 . . . 13
4.5. Algoritmo 13 . . . 14
4.6. Algoritmo 14 . . . 15
4.7. Algoritmo 15 . . . 16

Complejidad temporal y analisis de casos

López Manríquez Ángel
3CM3

22 de junio de 2019

1. Eficiencia de un algoritmo

Raramente existe un único algoritmo para resolver un problema determinado. Cuando se comparan dos algoritmos diferentes que resuelven el mismo problema, normalmente, se encontrará que un algoritmo es un orden de magnitud más eficiente que el otro. En este sentido lo importante, es que el programador sea capaz de reconocer y elegir el algoritmo más eficiente.

¿Entonces, qué es eficiencia? La eficiencia de un algoritmo es la propiedad mediante la cual un algoritmo debe alcanzar la solución al problema en el tiempo más corto posible y/o utilizando la cantidad más pequeña posible de recursos físicos, y que sea compatible con su exactitud o corrección. Un buen programador buscará el algoritmo más eficiente dentro del conjunto de aquellos que resuelven con exactitud un problema dado.

1.1. Eficiencia en los bucles

En general, el formato de la eficiencia se puede expresar mediante una función:

$$f(n) = eficiencia$$

Es decir, la eficiencia del algoritmo se examina como una función del número de elementos a ser procesados.

1.1.1. bucles lineales

En los bucles se repiten las sentencias del cuerpo del bucle un número determinado de veces, que determina la eficiencia del mismo. Normalmente, en los algoritmos los bucles son el término dominante en cuanto a la eficiencia del mismo.

Consideremos el siguiente lazo `for`, donde $a, b \in \mathbb{Z}$, $a \leq b$ y $s \in \mathbb{N}$:

Algorithm 1 linear for

```
1: for i = a to b with step s  do
2:     instruction();
3: end for
```

Sea x el número de veces que se ejecuta `instruccion()`. Vemos que la sucesión de valores que i va tomando es una progresión aritmética con primer término a y diferencia común s. Entonces:

$$i = a + s(k-1)$$

donde $k \in \mathbb{N}$. Por definición, se tiene que cumplir que $a \leq i \leq b$, es decir:

$$a \leq a + s(k-1) \leq b$$

De esa forma, x es simplemente el número de valores que puede tomar k. Simplificando ambos lados:

$$1 \leq k \leq \frac{b-a}{s} + 1$$

Pero como k es un entero positivo, entonces:

$$1 \leq k \leq \left\lfloor \frac{b-a}{s} + 1 \right\rfloor$$

De esa forma, $x = \left\lfloor \frac{b-a}{s} + 1 \right\rfloor = \left\lfloor \frac{b-a+1}{s} + \frac{s-1}{s} \right\rfloor = \left\lceil \frac{b-a+1}{s} \right\rceil$.

1.1.2. Bucle logaritmico

Ahora veamos qué pasa en el siguiente ciclo `for`, donde donde $a, b \in \mathbb{N}$, $a \leq b$ y $r \in \mathbb{N}$:

Algorithm 2 exponential for

```
1: for i = a to b at the end i = i * r  do
2:     instruction();
3: end for
```

Ahora i se comporta como una progresión geométrica con primer término a y razón r, es decir:

$$i = ar^{k-1}$$

donde $k \in \mathbb{N}$. De forma similar que en el primer caso, se tiene que cumplir que $a \leq i \leq b$, es decir, $a \leq ar^{k-1} \leq b$. Simplificamos y obtenemos $1 \leq k \leq \log_r \left(\frac{b}{a}\right) + 1$.

De esa forma, $x = \left\lfloor \log_r \left(\frac{b}{a}\right) + 1 \right\rfloor$ es el número de veces que se ejecutará `instruction()`.

1.1.3. Bucles anidados

El total de iteraciones de bucles anidados (bucles que contienen a otros bucles) se determina multiplicando el número de iteraciones del bucle interno por el número de iteraciones del bucle externo.

iteraciones = iteraciones del bucle externo* × *iteraciones bucle interno

Existen tres tipos de bucles anidados: lineal logarítmico, cuadráticos dependientes y cuadráticos que con análisis similares a los anteriores nos conducen a ecuaciones de eficiencia contempladas en la Tabla.

Linealmente logaritmica	$f(n) = [n \log_b n]$
Dependiente cuadratica	$f(n) = n\frac{n+1}{2}$
Cuadratica	$f(n) = n^2$

2. Complejidad temporal y espacial

Para los siguientes 5 algoritmos determine la función de complejidad temporal y espacial en términos de n. Considere las operaciones de: asignación, aritméticas, condicionales y saltos implícitos.

Supongamos los siguientes costos: c_1, c_2, c_3, c_4 para la asignacion, comparacion, operacion aritmetica y salto del contador del programa, respectivamente.

2.1. Algoritmo 1

para el primer y segundo `for` tenemos que $l(n) = (n-1)(c_1 + c_2 + c_4) + (n-2)(c_3 + instrucciones\ internas)$. por tanto

```
2  for (int i = 1; i < n; i++)                  // (n - 1)(c1 + c2 + c4) + c3 (n - 2)
3      for (int j = 0; j < n - 1; j++) {        // (n - 2)((n - 1)(c1 + c2 + c4) + c3 (n -
       ↪  2))
4          tmp = a[j];                          // c1 (n - 2) ^ 2
5          a[j] = a[j + 1];                    // (c1 + c3)(n - 2) ^ 2
6          a[j + 1] = tmp;                      // (c1 + c3)(n - 2) ^ 2
7      }
8
```

suponiendo que $c_1 = c_2 = c_3 = c_4 = 1$ tenemos

$$\begin{aligned} f_t(n) &= 3(n-1) + (n-2) + 2(n-2)n - 1) + 5(n-2)^2 \\ &= 3(n-1) + (n-2)(1 + 3n - 3 + 5n - 12) \\ &= 9n^2 - 29n + 25 \text{si n} > 1 \end{aligned}$$

cuando $n \leq 1$ solo tenemos dos costos, la asignacion $i = 1$ y su comparacion $i < n$.

La complejidad espacial es $s(n) = n + 4$, pues usamos el arreglo A de tamaño n, una variable extra `temp` y los dos iteradores.

2.2. Algoritmo 2

Aqui se nos pide evaluar la regla de Horner para la evaluacion del un polinomio

```
10 polynomial = 0;                              // c1
11 for (int i = 0; i <= n; i++)                 // (c1 + c2 + c4)(n + 1) + c3 n
12    polynomial = polynomial * z + a[n - 1]; // n (c1 + 2c3)
13
```

Tenemos $T_n = 2(3n + 2)$ y una complejidad espacial $s(n) = 4 + n$

2.3. Algoritmo 3

Para la evaluacion de un producto de matrices cuadradas de orden n, procedemos de manera analoga.

```
for (int i = 1; i <= n; i++)                    // 3(n + 1) + n
   for (int j = 1; j <= n; j++) {                 // n(4n + 3)
      c[i][j] = 0;                              // n ^ 2
      for (int k = 1; k <= n; k++)               // (4n + 3) n ^ 2
         c[i][j] = c[i][j] + a[i][k] * b[k][j];   // 3 n ^ 3
   }

```

Asi: $T_n = n(4n+3) + n^2 + (4n+3)n^2 + 3n^3 = 7n^3 + 8n^2 + 5n + 6$ y $S_n = 4 + 3n^2$.

2.4. Algoritmo 4

```
last = 1;                // c1
current = 1;             // c1
while (n > 2) {             // (n - 1)(c2 + c4)
   aux = last + current;     // (n - 2)(c1 + c3)
   last = current;           // (n - 2) c1
   current = aux;           // (n - 2) c1
   --n;                // (n - 2) (c1 + c3)
}

```

Asi, para $n > 2$: $T_n = 2 + 6n - 12 + 2n - 2 = 8n - 12$, 4 para $n \leq 2$ y $S_n = 4$.

2.5. Algoritmo 5

```
j = 0; // c1
for (i = n - 1; i >= 0; i--) // c3 (n - 1) + n (c1 + c2 + c4)
   s2[j] = s[i]; // c1(n - 1)
   j++; // (c1 + c3) (n - 1)
for (i = 0; i < n; i++) // c3(n - 1) + n(c1 + c2 + c4)
   s[i] = s2[i]; // c1(n - 1)
```

Asi, para $n \geq 0$: $T_n = 12n - 5$, 5 en caso contrario y $S_n = 2n + 3$.

3. Aproximación de una función que cuente el número de "prints"

3.1. Algoritmo 6

```cpp
#include <iostream>
#include <vector>

#include <cstdio>

using namespace std;

int six(int n) {
    int count = 0;
    for (int i = 10; i < 5 * n; i *= 2) {
        // cout << "\"Algoritmos\"\n";
        ++count;
    }
    return count;
}

int seven(int n) {
    int count = 0;
    for (int j = n; j > 1; j /= 2) {
        if (j < n / 2) {
            for (int i = 0; i < n; i += 2) {
                // cout << "\"Algoritmos\"\n";
                ++count;
            }
        }
    }
    return count;
}

int eight(int n) {
    int i = n;
    int count = 0;
    while (i >= 0) {
        for (int j = n; i < j; i -= 2, j /= 2) {
            // cout << "\"Algoritmos\"\n";
            printf("i: %d \t j: %d \n", i, j);
            ++count;
        }
    }
    return count;
}

vector<int> fillVector(int n) {
    vector<int> v;
```

```
45      for (int i = 1; i <= n; i++) v.push_back(i);
46      return v;
47  }
48
49  int main(void) {
50      int sel;
51      int (*fun)(int) ;
52      vector<int> values = { 10, 100, 1000, 5000, 100000 };
53
54      cout << endl << "select (6, 7, 8): ";
55      cin >> sel;
56
57      switch (sel) {
58          case 6: fun = six; break;
59          case 7: fun = seven; break;
60          case 8: fun = eight; break;
61          default: cout << "Invalid option" << endl; return 1;
62      }
63
64      printf("n\t|\tf(n)\n");
65      puts("======================");
66      for (int n: values) {
67          printf("%d\t|\t%d\n", n, fun(n));
68      }
69
70      return 0;
71  }
```

Tenemos la variante geométrica 1 del `for` con parámetros $a = 10$, $b = 5n - 1$ y $r = 2$. Por lo tanto, la función que indica el número de impresiones de la palabra "Algoritmos" es:

$$f(n) = \left\lfloor \log_2 \left(\frac{5n-1}{10} \right) + 1 \right\rfloor = \left\lceil \lg \left(\frac{n}{2} \right) \right\rceil$$

La formula funciona para $n > 2$ pues en caso contrario es 0. Comprobación :

```
C:\Users\ANGEL\Documents\codes\git\algorithmAnalysis\ej2 (master)
λ a

select (6, 7, 8): 6
n       |       f(n)
======================
10      |       3
100     |       6
1000    |       9
5000    |       12
100000  |       16
```

n	Número de impresiones reales	Número de impresiones según $f(n)$
10	3	$\left\lceil \lg\left(\frac{10}{2}\right)\right\rceil = 3$
100	6	$\left\lceil \lg\left(\frac{100}{2}\right)\right\rceil = 6$
1000	9	$\left\lceil \lg\left(\frac{1000}{2}\right)\right\rceil = 9$
5000	12	$\left\lceil \lg\left(\frac{5000}{2}\right)\right\rceil = 12$

3.2. Algoritmo 7

Si nos enfocamos unicamente en los bucles *for* tenemos que el primer lazo y segundo lazo, respectivamente son

$$l_1(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ \lfloor \lg n \rfloor & \text{en otro caso} \end{cases}$$

$$l_2(n) = \begin{cases} 0 & \text{si } n < 0 \\ \left\lceil \frac{n}{2} \right\rceil & \text{en otro caso} \end{cases}$$

Mas sin embargo tenemos una sentencia `if` $(j < \lfloor n/2 \rfloor)$ que complica el obtener el numero de impresiones. Para evadir este condicional podriamos empezar a l_1 en $\lfloor n/2 \rfloor$ y multiplicar las funciones para obtener a

$$f(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ \left\lceil \frac{n}{2} \right\rceil \left\lfloor \lg \left\lfloor \frac{n}{2} \right\rfloor \right\rfloor & \text{en otro caso} \end{cases}$$

Pero parece que nos equivocamos en una unidad con el factor del logaritmo:
Comprobación:

```
C:\Users\ANGEL\Documents\codes\git\algorithmAnalysis\ej2 (master)
λ a

select (6, 7, 8): 7
n       |       f(n)
========================
1       |       0
10      |       5
100     |       200
1000    |       3500
5000    |       25000
100000  |       700000
```

n	Número de impresiones reales	Número de impresiones según $f(n)$
10	5	$\lceil \frac{10}{2} \rceil \lfloor \lg \lfloor \frac{10}{2} \rfloor \rfloor \approx 5 \lfloor 2{,}32 \rfloor = 10$
100	200	$\lceil \frac{100}{2} \rceil \lfloor \lg \lfloor \frac{100}{2} \rfloor \rfloor \approx 50 \lfloor 5{,}64 \rfloor = 250$
1000	3500	$\lceil \frac{1000}{2} \rceil \lfloor \lg \lfloor \frac{1000}{2} \rfloor \rfloor \approx 500 \lfloor 8{,}97 \rfloor = 4000$
5000	25000	$\lceil \frac{5000}{2} \rceil \lfloor \lg \lfloor \frac{5000}{2} \rfloor \rfloor \approx 2500 \lfloor 11{,}29 \rfloor = 27500$

3.3. Algoritmo 8

Para este algoritmo el numero de impresiones es 0 $\forall n \in \mathbb{Z}$. Es importante destacar que el algoritmo se cicla infinitamente para $n \in \mathbb{Z}^+$ pues al ser mayor o igual a 0 se entra en el bucle `while` con $i = n$, luego en el lazo $j = n$ y se pregunta si $i < j$ es una contradiccion y nunca entramos al cuerpo del bucle, luego salimos de este pero como no decrementamos a i seguimos en el bucle, hacemos lo mismo anteriormente y volvemos a este punto y asi por siempre.

4. Analisis de casos

En este apartado usaremos como operaciones basica: asignacion y comparacion y asumiremos que el costo de calcularlas es la unidad.

4.1. Algoritmo 9

```
1  #include <vector>
2  using std::vector;
3
4  // |A| >= 2
5  double productOfTheTwoBiggest(vector<double> A, int n) {
6      double biggest1, biggest2; // 0, biggest1 > biggest2
7
8      // 1 comparacion y 2 asignaciones
9      if (A[1] > A[2]) {
10         biggest1 = A[1];
11       biggest2 = A[2];
12     } else {
13       biggest1 = A[2];
14       biggest2 = A[1];
15     }
16
17     int i = 3; // 1 asignacion
18
19     // si n == 2 solo habra una comparacion
20     while (i <= n) { // n - 2 comparaciones, las operaciones del cuerpo se ejecutan n -
       ↪  3 veces
21
22       if(A[i] > biggest1) { // una comparacion
23          biggest2 = biggest1; // una asignacion
24          biggest1 = A[i]; // una asignacion
25        } else if (A[i] > biggest2) // una comparacion
26          biggest2 = A[i]; // una asignacion
27
28       i = i + 1; // 1 asignacion
29     }
30     return biggest1 * biggest2; // 0 (no consideramos operaciones aritmeticas)
31 }
```

- **Mejor caso**
  El mejor caso en el trivial, cuando $n = 2$ tenemos $T_n = 5$.

- **Peor caso**
  Se da cuando A_{n-1}, A_{n-2} son los valores mas grandes del vector, el `if` de la linea 22 costaria 3 pues A_i sera mayor que el actual mas grande, lo que implica que se hacen dos asignaciones mas. asi:

$$T_n = 5 + (n-2) + 4n = 5n + 3$$

- **Caso medio** Suponiendo que en los condicionales `if` son equiprobables tenemos los siguientes costos:
  - $A_i > B_1$: 3
  - $A_i > B_2$: 3
  - $A_i < B_2$: 2

  Por tanto:

$$T_n = 5 + (n-2) + \frac{n}{3}(3+3+2) + n = \frac{14}{3}n + 2$$

4.2. Algoritmo 10

```
template<typename T>
void swapSort(vector<T> a) {
    int n = a.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++)
            if (a[j] < a[i]) // c1
                swap(a[j], a[i]); // c2
    }
}
```

- **Peor caso**

  Tenemos:

$$\begin{aligned} T_n &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} (c_1 + c_2) \\ &= (c_1 + c_2) \sum_{i=0}^{n-2} (n-1-i) \\ &= (c_1 + c_2) \left(n(n-1) - (n-1) - \frac{(n-2)(n-1)}{2} \right) \\ &= \frac{c_1 + c_2}{2} n(n-1) \end{aligned}$$

- **Mejor caso**
  El mejor caso es el mismo que el peor con $c_2 = 0$. **Caso medio**

- Para el caso medio supondremos que la condicion y su complemento son equiprobables, por lo que obtenemos el promedio entre el mejor y el peor caso:

$$\begin{aligned} T_n &= \frac{1}{2} \left[\frac{c_1 + c_2}{2} n(n-1) + \frac{c_1}{2} n(n-1) \right] \\ &= \frac{n(n-1)}{4} (2c_1 + c_2) \end{aligned}$$

4.3. Algoritmo 11

```
1 def gcd(m, n):
2    a = max(n, m)
3    b = min(n, m)
4    r = 1
5    while b > 0:
6       r = a % b
7       a = b
8       b = r
9    return a
```

Usaremos como operación básica el módulo de `a` y `b`. Sin pérdida de generalidad asumamos que $m \geq n$, pues si $m < n$, se intercambian en las líneas 2 y 3.

- **Mejor caso** Se da cuando $n = 0$, así nunca entramos al `while`, por lo que la complejidad del mejor caso es $f_t(n) = 0$ $\square$.
- **Peor caso** Se da cuando m y n son dos números consecutivos de la serie de Fibonacci, es decir, $m = F_{k+1}$ y $n = F_k$ para alguna $k \in \mathbb{N}$.

  Por definición sabemos que $F_{k+1} = F_k + F_{k-1}$ y $0 \leq F_{k-1} < F_k$, de esa forma el residuo de dividir F_{k+1} entre F_k será F_{k-1} y el cociente será 1 en **cada** iteración. Entonces, estamos transformando $(F_{k+1}, F_k) \to (F_k, F_{k-1})$, los cuales siguen siendo números de Fibonacci consecutivos.

  Como el cociente es al menos 1 para cualquier entrada y con esta entrada siempre obtenemos 1, estamos reduciendo al mínimo los dos números en cada iteración, obteniendo el peor caso. Así, nos tardaremos k divisiones llegar a que $F_k = 0$, y como $F_k \approx \phi^k$, la complejidad temporal aproximada del peor caso será $f_t(n) \approx \log_\phi(n)$, donde $\phi = \frac{1+\sqrt{5}}{2}$. $\square$
- **Caso medio** Si m y n no son números consecutivos de Fibonacci, al menos uno de los cocientes obtenidos será mayor a uno. Informalmente, podemos aumentar la base del logaritmo, de ϕ a 2, y decir que la complejidad temporal del caso medio es $f_t(n) \approx \log_2(n)$. $\square$

4.4. Algoritmo 12

```
1  template<typename T>
2  void bubbleSortOptimized(std::vector<T> a, int n) {
3      bool change = false;
4      int i = 0;
5      while (i < n - 1 && change != false) {
6          change = false;
7          for (int j = 0; j <= n - 2 - i)
8              if (a[i] < a[j]) { // c1
9                  swap(a[i], a[j]); // c2
10                 change = true; // 1
11             }
12         ++i;
13     }
14 }
```

- **Mejor caso**
  Se da cuando todos los elementos estan arreglado, a diferencia de otras versiones del bubblesort aqui se introduce una variable booleana que determina si la cantidad de elementos esta ordenada para algun k, si no se hizo ningun cambio salimos del bucle y no seguimos analizando el arreglo de manera cuadratica, asi, si los elementos estan ordenados ascendentemente tenemos que el algoritmo se ejecuta $n-1$ veces.

- **Peor caso**
  El arreglo esta ordenado de manera descendente, por lo que la variable `change` nunca se modifica en el `for` por tanto:

$$\begin{aligned} T_n &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} (c_1 + c_2) \\ &= (c_1 + c_2) \sum_{i=0}^{n-2} (n-2-i) \\ &= (c_1 + c_2) \left[n(n-1) - 2(n-1) - \frac{(n-2)(n-1)}{2} \right] \\ &= \frac{c_1 + c_2}{2} (n-1)(n-2) \end{aligned}$$

- **Caso medio**
  De manera similar, obtenemos el promedio de el peor y mejor caso.

4.5. Algoritmo 13

```cpp
#include <vector>
#include <iostream>

using std::vector;
using std::cout;

template <typename T>
void simpleBubble(vector<T> &a) {
    int n = a.size();
    for (int i = 0; i <= n - 2; i++)
        for (int j = 0; j <= n - 2 - i; j++) {
            if (a[j + 1] < a[j]) // c1
                swap(a[j], a[j + 1]); // c2
        }
}
```

- **Peor caso**
  Para el peor caso tenemos que c_2 siempre se ejecuta, por tanto

$$\begin{aligned} T_n &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} (c_1 + c_2) \\ &= (c_1 + c_2) \sum_{i=0}^{n-2} (n-2-i)) \\ &= (c_1 + c_2) \left[n(n-1) - 2(n-1) - \frac{(n-2)(n-1)}{2} \right] \\ &= \frac{c_1 + c_2}{2}(n-1)(n-2) \end{aligned}$$

- **Mejor caso**
  El mejor caso es similar al peor caso, solo que c_2, el intercambio, nunca se cumple, por tanto:

$$\begin{aligned} T_n &= \frac{c_1 + 0}{2}(n-1)(n-2) \\ &= \frac{c_1}{2}(n-1)(n-2) \end{aligned}$$

- **Caso medio**
  Formalmente el caso medio seria $\sum_{i=1}^{n} p(i)I(i)$, aqui suponemos que tanto la probabilidad de obtener $A_j > A_{j+1}$ como su complemento son equiprobables, por tanto

$$\begin{aligned} T_n &= \frac{1}{2}\left[\frac{c_1 + c_2}{2}(n-1)(n-2) + \frac{c_1}{2}(n-1)(n-2) \right] \\ &= \frac{(n-1)(n-2)}{4}(2c_1 + c_2) \end{aligned}$$

4.6. Algoritmo 14

```
1  def sort(a, b, c):
2     if a > b:
3        if a > c:
4           if b > c:
5              output(a, b, c) # 3 pasos: a > b, a > c, b > c
6           else:
7              output(a, c, b)   # 3 pasos: a > b, a > c, b <= c
8        else:
9           output(c, a, b)      # 2 pasos: a > b, a <= c
10    else:
11       if b > c:
12          if a > c:
13             output(b, a, c)   # 3 pasos: a <= b, b > c, a > c
14          else:
```

```
15              output(b, c, a)    # 3 pasos: a <= b, b > c, a <= c
16          else:
17              output(c, b, a)    # 2 pasos: a <= b, b <= c
```

Usaremos como operaciones básicas las comparaciones entre a, b y c.

- **Mejor caso** Se da cuando:
  - $a > b$ y $a \leq c$
  - $a \leq b$ y $b \leq c$

  Dando como complejidad $f_t(n) = 2$. □
- **Peor caso** Se da cuando:
  - $a > b$, $a > c$ y $b > c$
  - $a > b$, $a > c$ y $b \leq c$
  - $a \leq b$, $b > c$ y $a > c$
  - $a \leq b$, $b > c$ y $a \leq c$

  Dando como complejidad $f_t(n) = 3$. □
- **Caso medio** Como todas las salidas de la función tienen la misma probabilidad, $\frac{1}{6}$, la complejidad del caso medio es: $f_t(n) = \dfrac{3 \times 4 + 2 \times 2}{6} = \dfrac{8}{3}$. □

4.7. Algoritmo 15

```
template<typename T>
void selection(vector<T> a, int n) {
    for (int i = 0; i <= n - 2; i++) {
        int minpos = i; // 1
        for (int j = i + 1; j <= n -1; j++)
            if (a[j] < a[minpos]) p = j; // c1
        swap(a[p], a[i]); // c2
    }
}
```

Este algoritmo no varia mucho en casos, mas sin embargo reduce el no. de intercambio de elementos, lo cual es muy conveniente si lo que se va a intercambiar es costoso en si.

- **Peor caso**

Tenemos:

$$T_n = \sum_{i=0}^{n-2} \left(\sum_{j=i+1}^{n-1} c_1 + 1 + c_2 \right)$$
$$T_n = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} c_1 + (n-1)(c_2+1)$$
$$T_n = c_1 \sum_{i=0}^{n-2} (n-2-i-1+1) + (n-1)(c_2+1)$$
$$T_n = c_1 \frac{(n-2)(n-1)}{2} + (n-1)(c_2+1)$$
$$T_n = \frac{c_1}{2} n^2 + (c_2 - \frac{3}{2} c_1 + 1)n + c_1 - c_2 - 1$$

- **Mejor caso**
  El mejor caso a decir verdad es muy similar al peor caso en cuestion de costo si hacemos $c_2 \leftarrow c_2 - 1$, dando por sentado que una asignacion de enteros cuesta la unidad.

- **Caso medio**
  De la misma manera, obtenemos el promedio de los casos anteriores, sin embargo las variaciones son despreciables.