Apuntes de Teórico

PROGRAMACIÓN 3

Complejidad

Versión 1.0

Apuntes de Teórico de Programación 3 – Complejidad

Índice

COMPLEJIDAD DE PROBLEMAS - Introducción

Análisis de Algoritmos: estudio de familias de algoritmos y su costo en el PEOR CASO y CASO MEDIO.

Dado un problema hay algoritmos que lo resuelven. Es a través del análisis de algoritmos que determinamos la respuesta a la siguiente pregunta ¿hasta donde se pueden mejorar los algoritmos?

Todo problema tiene una complejidad mínima asociada.
En otras palabras, se debe hallar una función, que se denotará por F(n), que acote inferiormente la cantidad de operaciones para

- Peor Caso: $F_W(n)$
- Caso Medio: $F_A(n)$

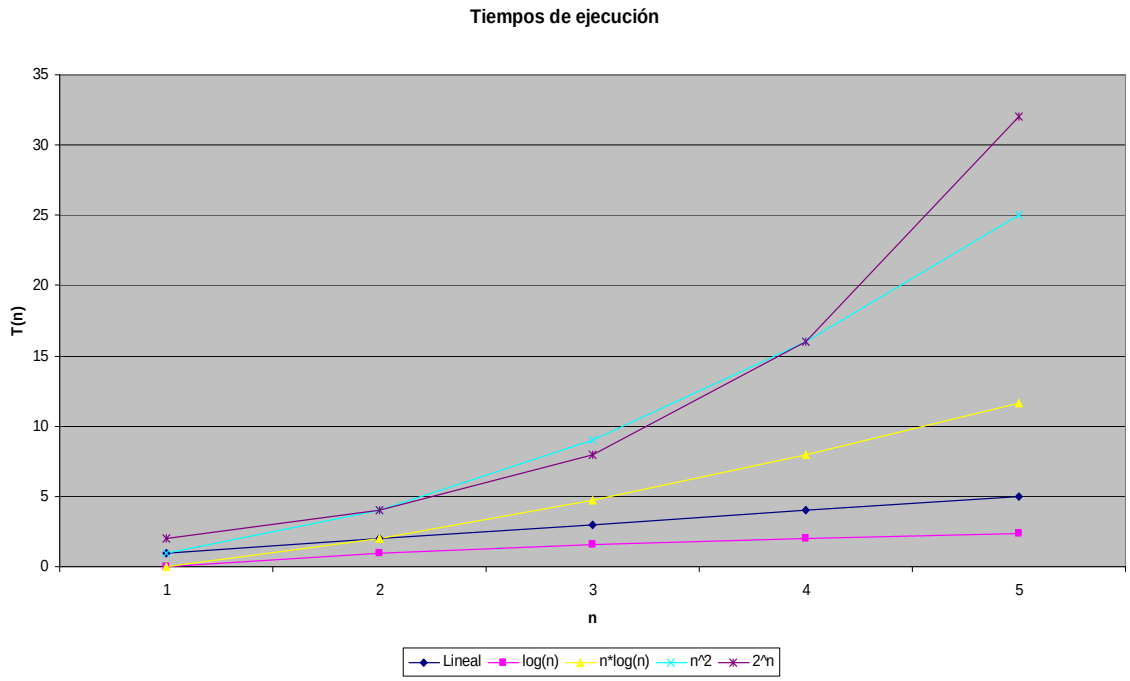
del problema considerado.

Reformulando:
Sea el problema P del que se quiere estudiar la complejidad. Existe un conjunto $A = \{A_1, A_2, \ldots, A_n\}$ de algoritmos que brindan una solución al problema P. Sea T el conjunto de tiempos (en el peor caso) para cada algoritmo, $T = \{T_1, T_2, \ldots, T_n\}$ (siendo T_{wi} el peor caso para el algoritmo A_i) .

Se deben encontrar expresiones $F_W(n)$ tales que $T_{wi} \geq F_W(n)$, $1 \leq i \leq n$.
Lo anterior se debe cumplir para todo algoritmo que solucione el problema, t**anto los conocidos (A_i), como los todavía desconocidos**. Esto sólo puede lograrse con un estudio teórico independiente de alguna implementación particular, en base a las características del problema.

Al buscar la función F, se debe procurar además que sea lo mayor posible (entre todas las posibles cotas inferiores).

De manera grafica: Sea el problema P con algoritmos conocidos $A = \{A_1, A_2, A_3, A_4\}$ y tiempos (en el peor caso) T = {n, n×log(n), n ^2, 2 ^n}. Entonces una cota mínima para todos estos algoritmos podría ser log(n). Si se demuestra además que independientemente de los algoritmos, esta cota se cumple para todo algoritmo que resuelva el problema, hemos hallado la cota que estamos buscando. La demostración de las cotas de tiempo es en base a especificaciones propias del problema que se considera.
Siguiendo con este ejemplo, si por otra vía se determina que otra función mayor que log n también es cota inferior para el problema (p.e. F= log n^2 o F=n), se determina que la mayor será la cota buscada.-



Ejemplo: Encontrar el mínimo de una secuencia de enteros mediante comparaciones dos a dos.

Estudiar la complejidad del problema anterior seria equivalente a responder la pregunta: ¿Cuántas comparaciones deben realizarse como mínimo para encontrar el mínimo en la secuencia de enteros mediante comparaciones dos a dos?

Replanteando el problema anterior: Encontrar el mínimo de una secuencia de enteros mediante comparaciones dos a dos es equivalente a determinar los (n=1) Elementos que no son el mínimo.

Se tiene entonces la siguiente información del problema simétrico:

1. Para “descartar” un elemento, debe figurar al menos **una** comparación (para este elemento)
2. Si un elemento *x* no es comparado **nunca** entonces, este podría ser el mínimo.

De lo anterior se desprende que si cada uno de los n-1 elementos que no son mínimos tienen que ser comparados por lo menos una vez, se concluye que deben realizarse al menos n-1 comparaciones para descartar dichos elementos.

Formalizando:

$$\begin{cases} F_W(n) = n-1 \\ F_A(n) = n-1 \end{cases}$$

¿Cómo se puede determinar si la cota encontrada es suficientemente buena o se puede mejorar?

Una manera es la siguiente: Encontrar un algoritmo A_k que cumpla con la cota. Al encontrar dicho algoritmo, sabemos que no hay otra funcion F' > F que esté por encima y además se cumpla que $T_{wi} \geq F_W(n)$ para todo algoritmo.

Lo anterior se desprende del siguiente razonamiento: supongamos que existe $F'_W(n) \mid F'_W(n) > F_W(n)$, entonces $F'_W(n)$ no es cota mínima, puesto que existe un algoritmo A_k (el que cumple con la cota mínima) y además cumple que $T_{wk}(n) < F'_W(n)$.

A continuación se presenta el caso de estudio de **SORTING** para estudiar la complejidad de los problemas.

Se presentara también una herramienta que permitirá hacer deducciones teóricas sobre la complejidad: el árbol de decisión (de los algoritmos).

Introducción a Árboles de decisión

Para proceder al análisis de un problema, donde se tratará de determinar la complejidad mínima del mismo, se presentará previamente la herramienta teórica a usar con ese fin: el *árbol de decisión*. El *árbol de decisión* será el modelo matemático que representará las diferentes ejecuciones de los algoritmos en función de su entrada.

Se realizará en principio una presentación informal, ejemplificando la herramienta para dos algoritmos conocidos y luego se formalizará.

Los siguientes algoritmos ya son conocidos del práctico 2 del curso y se tratan de la ordenación por Selección (*Selection Sort*) y la ordenación por Inserción (*Insertion Sort*).

Presentación del árbol de decisión para el Selection Sort

Este algoritmo ordena una secuencia de *n* enteros en orden ascendente. Éste localiza el valor más pequeño, lo coloca en el primer lugar, luego localiza el segundo valor más pequeño y lo ubica en el segundo lugar y así sucesivamente.

La especificación queda:

Entrada: Secuencia *S* a ordenar
Salida: Secuencia *R* ordenada

`Proceso`
`  R = Crear()`
`  Mientras NOT Vacia(S) hacer`
`  R = InsBack(R, Minimo(S))`
`  S = DeleteMin(S)`
`  Fin Mientras`
`  Devolver R`

Una implementación

Se utilizará la opción de implementar la secuencia y el resultado ordenado sobre un único arreglo *A* de *n* elementos. El arreglo se considerará dividido en dos partes, la izquierda contendrá la parte ordenada, la derecha contendrá la parte desordenada. En cada paso se buscará el menor elemento de la desordenada y se intercambiará con el primer elemento de ésta, creciendo así la parte ordenada y decreciendo la desordenada (en tamaño).

```
void SelectionSort (int* A, int n){
    int i, j, posmin, tmp;
(1) for (i = 0; i < n-1; i++){
        posmin = i;
(2)     for (j = i+1; j < n; j++){
(3)         if (A[j] < A[posmin])
                posmin = j;
        }
        // Intercambio de elementos
        tmp = A[i];
        A[i] = A[posmin];
        A[posmin] = tmp;
    }
}
```

En esta implementación las secuencias *S* y *R* se implementan sobre el mismo arreglo *A*, el cual contiene al principio la secuencia *S* y al final la *R*.

En el desarrollo de la ejecución se da que:

- La secuencia *R* estará al comienzo del arreglo, empezando en la posición señalada por la variable ***i***.
- Inicialmente *R* es vacía, pero irá creciendo según el ciclo (1).
- En el ciclo (2) se busca el mínimo en *S*, notar que *S* comienza en la posición ***i*+1** de ***A*** y ocupa hasta la posición ***n*-1** (final del arreglo).
- Al finalizar el ciclo (2) queda en ***posmin*** la posición donde se encuentra el mínimo en cada iteración.
- Finalmente se intercambian las posiciones *A*[`i`] con *A*[*posmin*], comenzando otra iteración de (1), creciendo así *R* y achicándose *S*.

Observar que, aprovechando la estrategia seguida, en el ciclo (1) sólo se llega hasta la posición *n*-2, el restante elemento tiene, necesariamente, que ser el mayor y quedar al final de la ejecución en la posición *n*-1.

Para analizar el algoritmo se quiere contar únicamente la cantidad de comparaciones entre los elementos de la secuencia, o sea cuantas veces se evalúa la condición del if (3).

En el ciclo (2) dicha condición se evalúa (*n* - `i` - 1) veces. Éste ciclo se ejecuta (*n*-1*)* veces (de 0 a n-2) debido al ciclo (1): Por tanto, el tiempo de ejecución está dado por la siguiente sumatoria

$$T(n) = \sum_{i=0}^{i=n-2} \left(\sum_{j=i+1}^{j=n-1} (1) \right) = \sum_{i=0}^{i=n-2} (n-i-1) = \sum_{i=0}^{i=n-2} (n-1) - \sum_{i=0}^{i=n-2} (i)$$

$$= (n-1)^2 - \frac{(n-1).(n-2)}{2} = \frac{2n^2 - 4n + 2 - n^2 + 3n - 2}{2} = \frac{n^2 - n}{2}$$

$$\Rightarrow T(n) = \frac{n(n-1)}{2} \Rightarrow T(n) \in \mathrm{O}(n^2)$$

Ambas expresiones obtenidas son válidas según el contexto:

Se tienen $n(n\text{-}1)/2$ comparaciones si lo que se quiere obtener es el conteo exacto. Sin embargo, este valor es $O(n^2)$ si nos alcanza con una cota superior del mismo.

Este valor de $T(n)$ se da para toda entrada debido a que se hace una búsqueda del mínimo en cada iteración y eso hace que las comparaciones se hagan independientemente de los valores que contenga la secuencia. Por lo tanto el $T(n)$ hallado vale para el *peor caso*, *mejor caso* y para el *caso medio*.

Árbol de decisión para Selection Sort

Introducción

La idea es modelar todas las posibles ejecuciones del algoritmo cualquiera sea la secuencia de entrada. Su construcción será como una ejecución "a mano" del algoritmo dibujando un árbol binario.

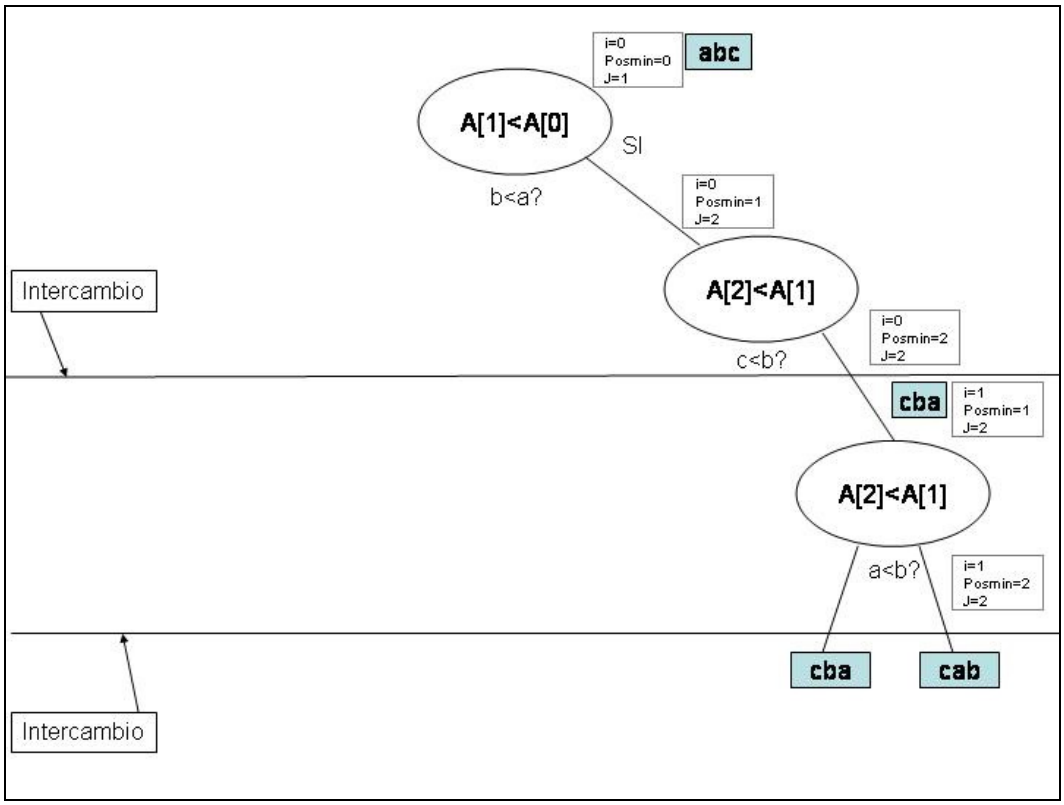
Cada nodo del árbol corresponderá a una comparación de elementos `A[j] < A[posmin];` esto se hará desde el comienzo de la ejecución del algoritmo, con los valores iniciales de `j` y `posmin`. Se tendrán dos resultados posibles según la respuesta a la comparación, en cada caso se adecuará el arreglo según lo que indica el algoritmo y se volverá a realizar lo anterior con los nuevos valores de `j` y `posmin`, agregando los nodos correspondientes a las comparaciones. Al terminar con las comparaciones se colocarán como hojas del árbol (nodos externos) rectángulos con los elementos ordenados.

Construcción

A continuación se verá el *árbol de decisión* para este algoritmo en un ejemplo para ***n***=3.

Inicialmente se tiene que en el arreglo vienen dados 3 elementos con valores *a*, *b* y *c* en las posiciones A_0, A_1 y A_2 respectivamente.

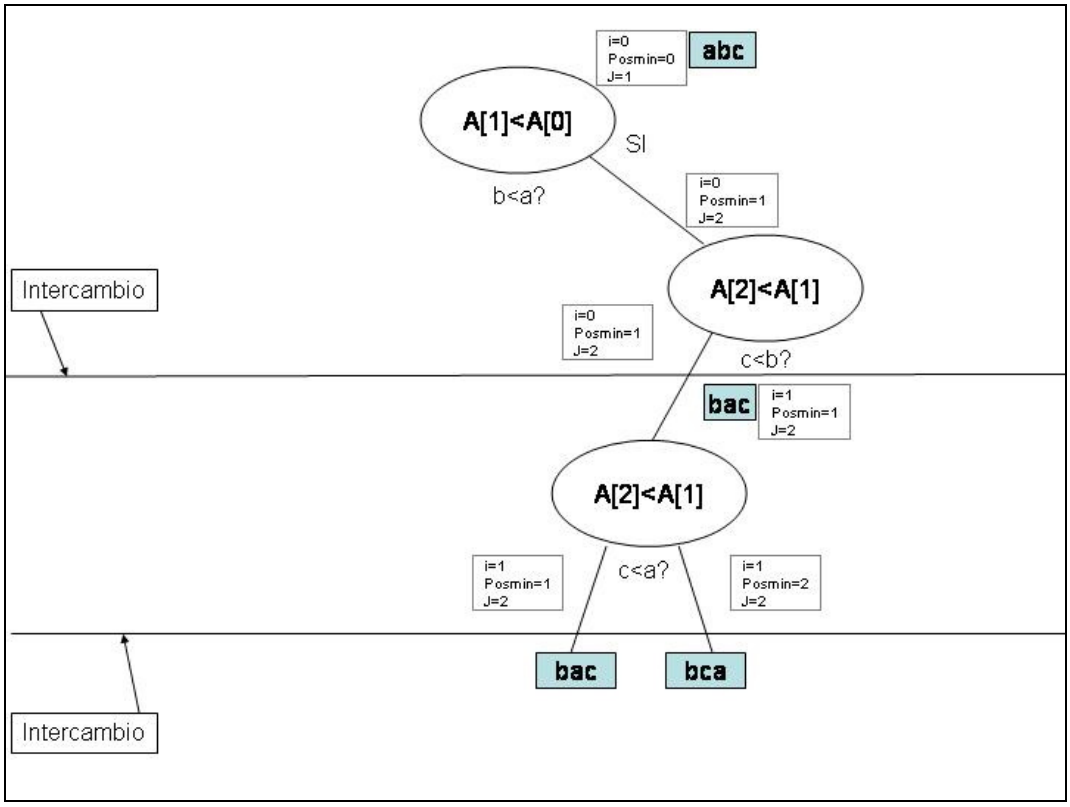
En el árbol los óvalos representan las comparaciones, los rectángulos coloreados son las diferentes variaciones de `A`, agregándose además los valores de `i`, `posmin` y `j` en los distintos pasos.



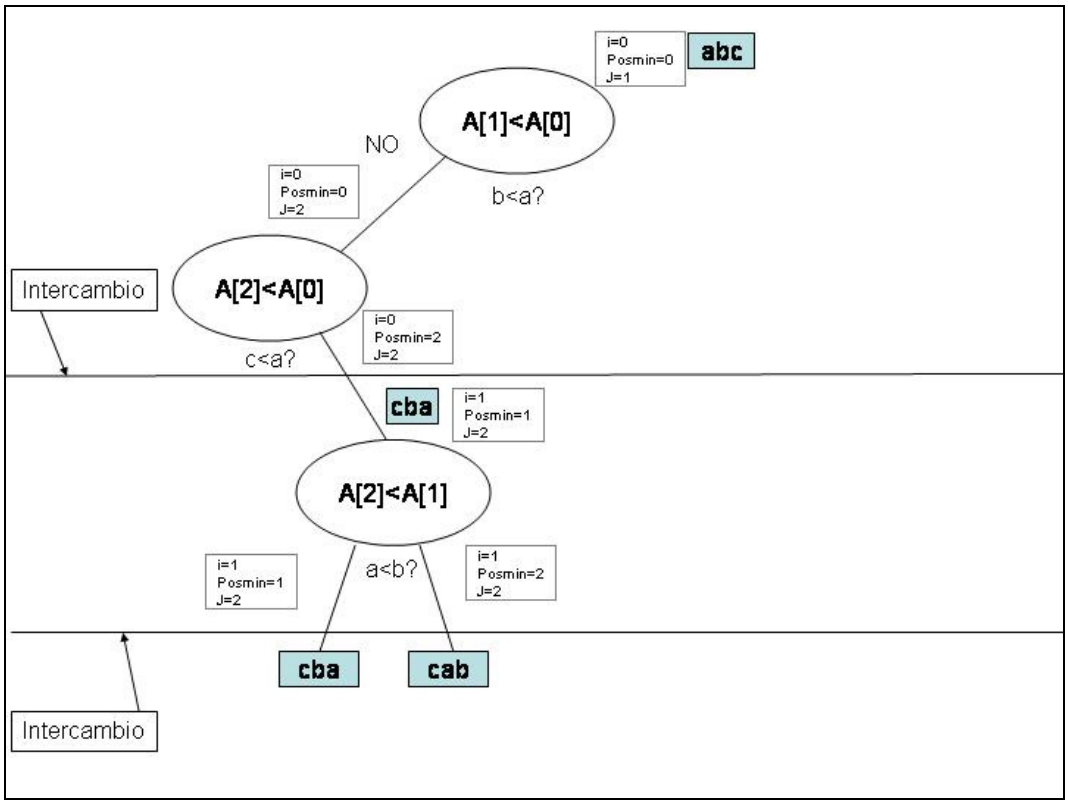
Esta primer figura muestra el árbol parcial con su rama de más a la derecha: inicialmente se tiene que el primer nodo (raiz) realiza la comparación $A[1] < A[0]$, esto se debe a que $j = 1$ y $posmin = i = 0$. Notar que ésta comparación equivale a comparar los elementos de la secuencia *a* y *b:* ***¿*** $a < b$ ***?*** ya que el arreglo inicial es [*abc*].

Si esta comparación da *verdadero*, se toma a la derecha en el árbol, el valor de `posmin` cambia a 1 y el de j a 2 (ver algoritmo). En estas condiciones continúa el ciclo (2), la comparación es entre los elementos 2 y 1 de A (***¿***$c < b$***?***). Si la respuesta es afirmativa, se cambia `posmin` al valor de j que es 2, termina el ciclo (2) y se realiza el intercambio, ya se encontró el mínimo que está en la posición 2, o sea que *c* es el menor elemento y se cambia al primer lugar ($i = 0$). Notar que *c* no se accederá más, es el primer elemento de la parte ordenada del arreglo.

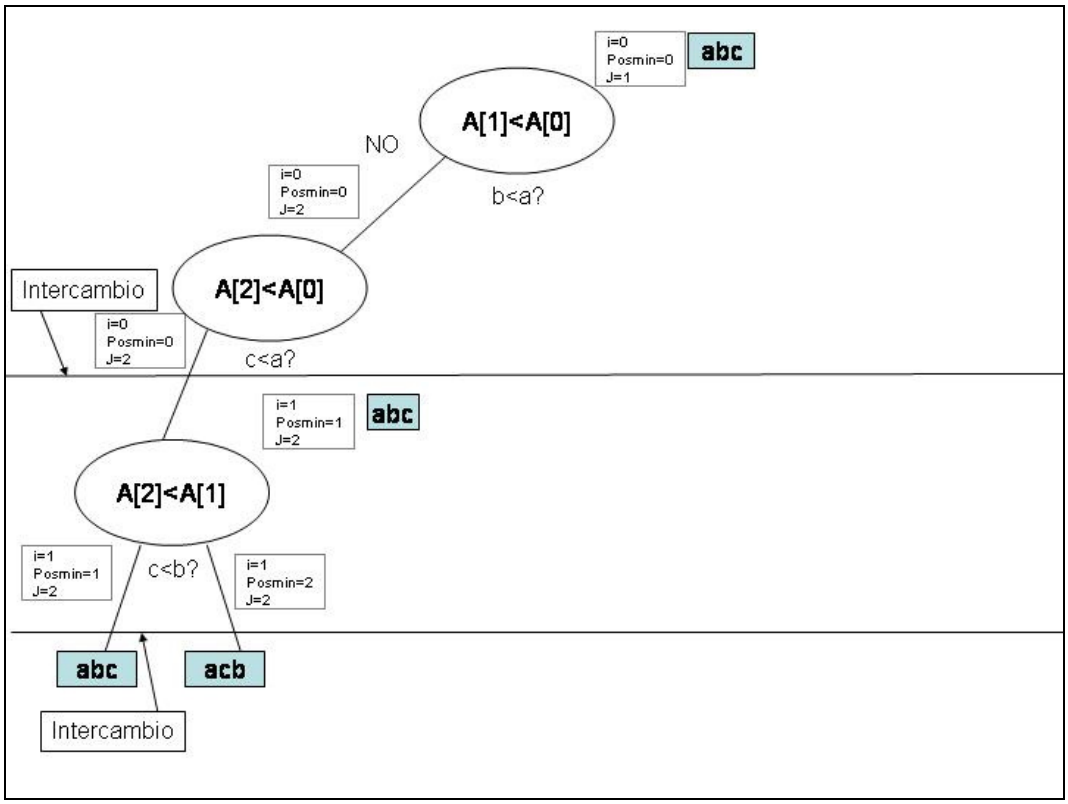
Comienza un nuevo ciclo (1) con $i=1$, entonces $posmin=1$ y $j=2$. La siguiente comparación es entre los elementos 2 y 1 del arreglo (ya modificado) o sea ***¿***$a < b$***?***. Si la comparación da *verdadero*, entonces $posmin = 2$, termina el ciclo (2), se hace el intercambio quedando [*cab*] y el algoritmo termina, [*cab*] será la ordenación final. Si la respuesta es *falso*, el intercambio mantiene el elemento en su lugar y la secuencia final ordenada será [*cba*].



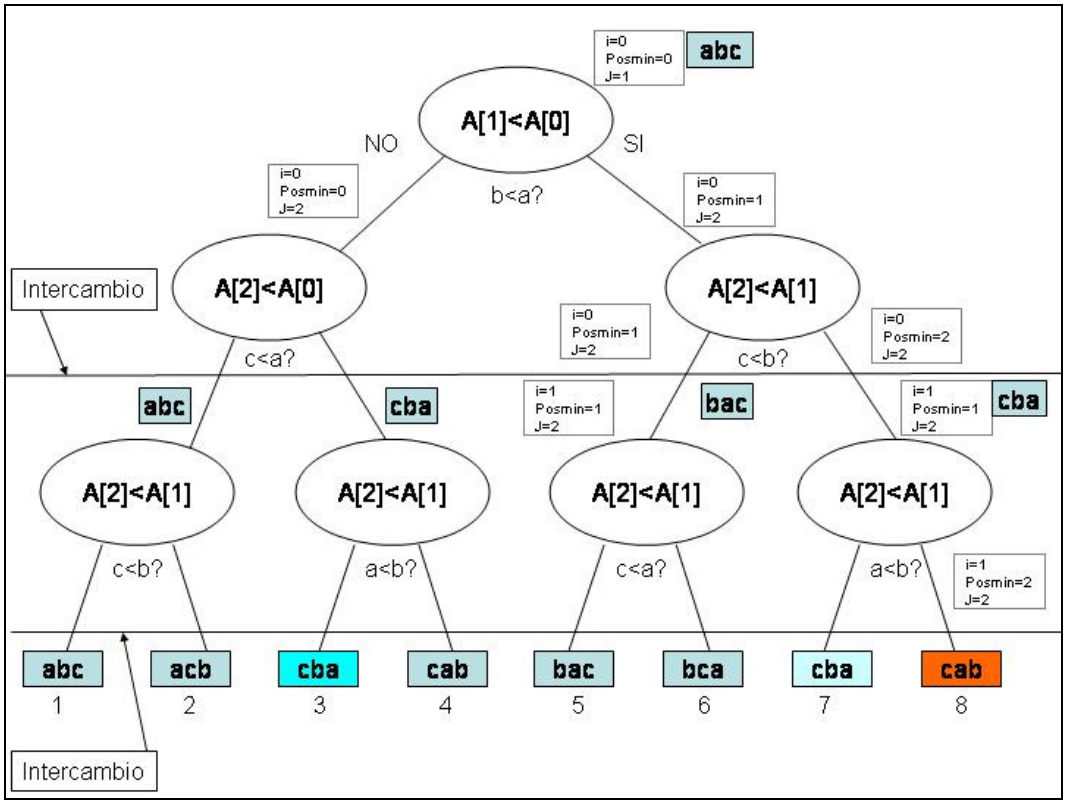
Yendo hacia “atrás” en la ejecución, esta segunda figura muestra el árbol resultado de una evaluación negativa de la segunda comparación inmediata antes del intercambio. Esta respuesta hace que `posmin` no cambie y por lo tanto que el mínimo del arreglo sea el segundo elemento, *A*[1] o sea *b*. Como antes, éste será el primer elemento de la parte ordenada y ya no cambiará, restando definir si le sigue *a* o *c*. Después del intercambio, se inicia otro ciclo (1) (como en el primer caso), se tendrá `posmin`=`i`=1, `j`=2 y el arreglo [*bac*]. Se comparan los elementos 2 y 1; si el resultado es afirmativo, `posmin` pasa a valer 2, se intercambian elementos y termina con la ordenación final [*bca*]. Por el contrario, si la respuesta es negativa, el intercambio se realiza pero no mueve efectivamente elementos, quedando [*bac*].



Volviendo al inicio de la ejecución del algoritmo, se verán ahora los casos en que la respuesta a la primera comparación (raíz) fue negativa. En ese caso no cambia `posmin` y al continuar el ciclo (2) con `j` = 2, compara el elemento 2 con el elemento 0 (*¿ c < a?*). Si da *verdadero*, `posmin` = 2 y termina el ciclo (2), se realiza el intercambio quedando el arreglo [*cba*]: el mínimo es *c* y ya no cambiará. Se inicia un nuevo paso del ciclo (1), `posmin` = `i` = 1, `j` = 2. La siguiente comparación es entre los elementos 2 y 1 (*¿a < b?*), si es *verdadero*, `posmin` = 2, termina el ciclo (2) y se hace el último intercambio quedando [*cab*]. Si el resultado da *negativo*, queda [*cba*] sin cambios.



Esta cuarta figura muestra la rama de más a la izquierda del árbol que corresponde a todas respuestas negativas. Con respecto al caso anterior implica volver un paso atrás y considerar el caso en que la segunda comparación dio negativo. En ese caso no cambia `posmin` (como siempre que la respuesta a la comparación sea negativa). El mínimo entonces es *a*, que se cambiará al primer lugar (en realidad el intercambio lo deja en el mismo lugar). El arreglo queda como el inicial [*abc*]. Comienza otra iteración de (1) `posmin`=`i`=1, `j`=2 y se comparará el elemento 2 con el 1. Si la respuesta es afirmativa, `posmin`=2, se intercambia, termina el algoritmo resultando la secuencia ordenada [acb]. Si la respuesta es negativa, `posmin` no cambia, el intercambio deja el elemento en el mismo lugar quedando la secuencia ordenada [abc], la cual corresponde al caso en que la secuencia inicial ya venía ordenada.



Esta figura muestra el árbol completo. No se muestran todos los valores intermedios por simplicidad. Notar que aquí se representan TODAS las posibles ejecuciones del algoritmo. Para una entrada particular, por ejemplo [3,1,2] el algoritmo recorrerá un único camino de la raíz a una hoja. En el ejemplo terminará en el nodo externo (hoja) 6 que representa el caso en que el menor elemento es el que viene inicialmente en el segundo lugar, el siguiente es el tercero y el último es el primero.

Nota:

El nodo 8 no puede darse nunca. La primer comparación (***¿****b* < *a****?***) resulta afirmativa, entonces no puede resultar afirmativa la última comparación que lleva a que *a* sea menor que *b*.

Los nodos externos 3 y 7 representan [cba] en ambos casos, pero en el nodo 7 se representan los casos donde $b<a$ y el caso del nodo 3 sólo puede representar los casos donde $a=b$ (por construcción). Si no se admiten repetidos, el caso del nodo 3 no puede darse nunca (por las comparaciones anteriores como en el caso del nodo 8).

Presentación del árbol de decisión para el Insertion Sort

Este algoritmo ordena una secuencia de *n* enteros en orden ascendente mediante una estrategia que considera dos secuencias: origen y resultado. En cada paso se toma el primero de la secuencia a ordenar y se busca su ubicación en el resultado de forma que éste se mantenga ordenado.

La especificación queda:

Entrada: Secuencia *S* a ordenar
Salida: Secuencia *R* ordenada

`Proceso`
`  R = Crear()`
`  Mientras NOT Vacia(S) hacer`
`  R = InsEnOrden(R, Primero(S))`
`  S = Resto(S)`
`  Fin Mientras`
`  Devolver R`

Una implementación

Como en el caso de *Selection Sort*, se utilizará un algoritmo que reciba la secuencia en un arreglo de *n* elementos y proceda a ordenar con esta estrategia sobre el mismo arreglo. Para ello se considerará el arreglo dividido conceptualmente en dos partes, una, la de más a la izquierda (o de menores índices) ya ordenada y el resto será la secuencia desordenada. La idea consiste en tomar el primer elemento de la parte desordenada e irlo desplazando a la izquierda hasta insertarlo en su lugar. Para ello se deberá al mismo tiempo desplazar a la derecha los elementos mayores a él que forman la parte ordenada de manera de hacer lugar en el arreglo para ese elemento a insertar.

En el algoritmo siguiente se recibe el arreglo `A`, se usan las variables: *First* para ubicar transitoriamente el primer elemento de la parte desordenada, o sea el elemento a insertar en la parte ordenada, `i` para indicar la posición de comienzo de la parte desordenada y *j* para buscar la ubicación de *First* en la parte ya ordenada (desde `i`-1 a la izquierda).

```
void InsertionSort (int* A, int n){
   int i, j, First;
(1)for (i = 1; i < n; i++){
        First = A[i];
        j = i-1;
(2)   while (j >= 0 && First < A[j]){
                A[j+1] = A[j];
                j--;
       }
     A[j+1] = First;
  }
}
```

Como se comentó, en esta implementación las secuencias *S* y *R* se implementan sobre el mismo arreglo *A*, el cual contiene en su parte izquierda la secuencia *S* y en la derecha la *R*.

En el desarrollo de la ejecución se da que:

- La secuencia *R* estará al comienzo del arreglo, inicialmente *R* es vacía, por lo tanto no ocuparía lugar en el arreglo, sin embargo aprovechando la estrategia, puede considerarse que contiene un elemento desde el comienzo, éste será *A*[0], luego irá creciendo según el ciclo (1). La secuencia *S* empezará en la posición señalada por la variable ***i***, inicialmente con el valor 1, acorde con lo anterior.

- En el ciclo (2) se busca la posición de First en *R*: notar que *S* comienza en la posición ***i*** de ***A*** y ocupa hasta la posición ***n-1*** (final del arreglo), mientras que *R* comienza en 0 y termina en la posición ***i-1***.

- Al finalizar el ciclo (2) ***j*** se ha pasado de la posición donde debe insertarse ***First*** y por eso termina el ciclo (puede ser porque haya encontrado un lugar intermedio o que sea el menor de *R* y tenga que colocarse en la posición 0, por eso se admite que ***j*** llegue a valer -1.

- Finalmente se coloca ***First*** en la posición ***j*+1**, de acuerdo a lo anterior.

Para analizar el algoritmo se quiere contar únicamente la cantidad de comparaciones entre los elementos de la secuencia, o sea cuantas veces se evalúa la segunda parte de la condición del *while* (2): ***First < A[j].***

El Peor Caso viene dado cuando se tiene que colocar al principio del arreglo cada elemento a ordenar; en ese caso se realizarán en cada paso ***i*** comparaciones.

$$T(n) = \sum_{i=1}^{i=n-1} (i) = \frac{n.(n+1)}{2} - n = \frac{n^2 - n}{2} = \frac{n(n-1)}{2}$$

En este algoritmo se tiene también que el peor caso es $O(n^2)$.

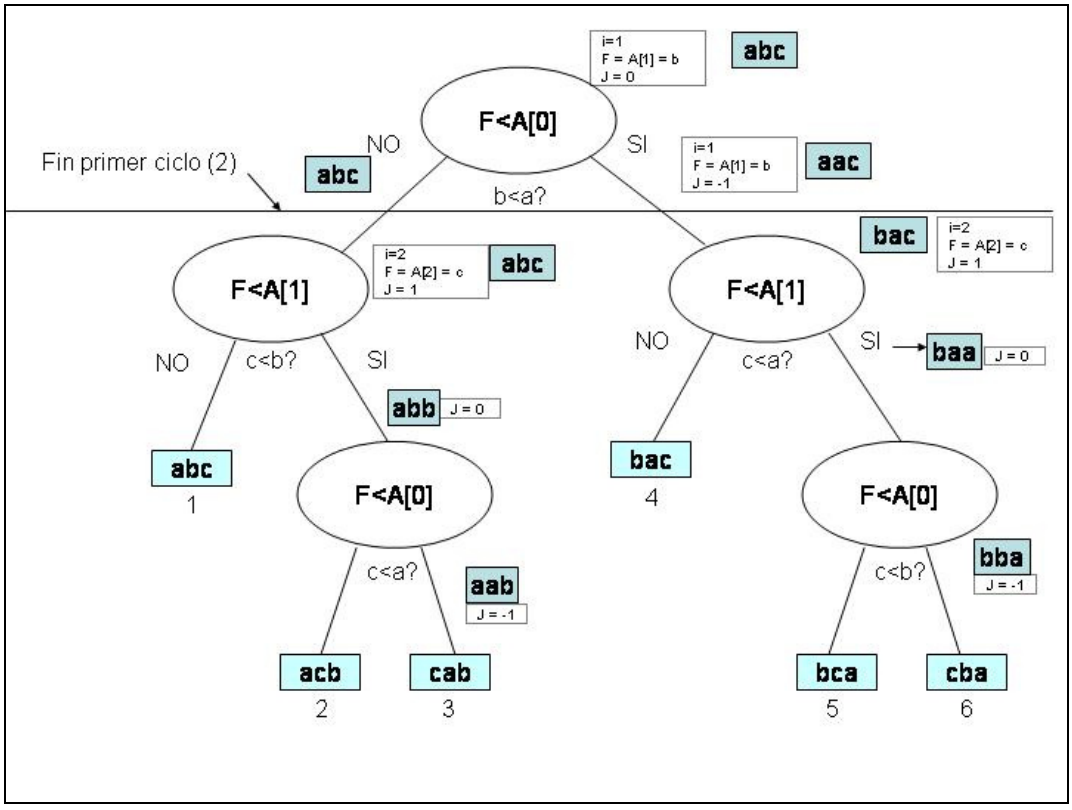
Árbol de decisión para Insertion Sort

A continuación se verá el *árbol de decisión* para este algoritmo. Como antes, la idea es modelar todas las posibles ejecuciones del algoritmo cualquiera sea la secuencia de entrada. Se seguirá el mismo método de construcción antes utilizado.

En este caso, cada nodo interno del árbol corresponderá a una comparación de elementos de la forma `First < A[j];` esto se hará desde el comienzo de la ejecución del algoritmo, con los valores iniciales de `i, j` y `First`. Como antes, de acuerdo al resultado, se actualizarán las variables y se realizarán los cambios especificados en el algoritmo. Notar que se compara con `First`, al que se abreviará como *F* en la figura que almacena un valor fijo mientras varía `j`.

Otra vez se estudiará este ejemplo para *n*=3.

Inicialmente se tiene que en el array vienen dados 3 elementos con valores *a*, *b* y ***c*** en las posiciones A_0, A_1 y A_2 respectivamente.



En este caso no se verá en la forma detallada del caso del *Selection Sort*, en la figura se muestra el árbol final que ilustra o modela todas las ejecuciones posibles del *Insertion Sort*.

Se seguirá paso a paso sólo la rama derecha del árbol:

Inicialmente *i*=1, *First*=*A*[1]=*b*, *j*=0; el arreglo es [*abc*]. Eso hace que la comparación que quede en el nodo raíz sea *F*<*A*[0] o sea *b* < *a*. Por la rama *SI*, se entra al *while* (2), se mueve el primer elemento al segundo lugar quedando el arreglo [*aac*], *j* pasa a valer -1 por lo cual termina la iteración. Se coloca *First* en el lugar 0, entonces queda el arreglo como [*bac*].

Comienza una nueva iteración (1) por lo tanto *i*=2, *First* = *A*[2] = *c*, *j*=1; el arreglo es el que había quedado del paso anterior [*bac*]. La siguiente comparación es *F* < *A*[1] o sea *c* < *a*. Por la rama *SI*, se entra al *while* (2), se mueve el segundo elemento al tercer lugar quedando el arreglo como [*baa*], *j* pasa a valer 0 por lo que continúa la iteración (2). La siguiente comparación es *F* < *A*[0] o sea *c* < *b*. Por la rama *SI*, se vuelve a entrar al ciclo (2), se mueve el primer elemento al segundo lugar quedando el arreglo como [*bba*] y *j* pasa a valer -1 con lo cual termina la iteración (2). Se coloca *First* en el lugar 0, quedando el arreglo como [*cba*]. Se llegó así a un nodo externo (hoja) que representa el final de la ordenación.

De manera similar se procederá con el resto del árbol, analizando las restantes variaciones en la ejecución del algoritmo según el resultado de las comparaciones.

El árbol correspondiente a este algoritmo (mostrado en la figura) tiene 6 nodos externos. Obsérvese que esta cantidad de hojas es la mínima que puede tener un árbol bien construído ya que debe reflejar todas las posibles ejecuciones para todas las secuencias de 3 elementos que se puedan formar con *a*, *b* y *c*, o sea todas las permutaciones de 3 elementos : 3! = 6.

Conclusiones en base a los árboles de decisión construídos

De acuerdo a lo visto, un *Árbol de Decisión* es un árbol binario estricto (cada nodo tiene 2 hijos o ninguno) donde los nodos internos tienen 2 hijos y los externos (hojas) no tienen ninguno. Estos árboles pueden ser usados en general para modelar la ejecución de algoritmos que resuelvan un problema dado. En este marco, los nodos internos representan operaciones y los externos representan el final de la ejecución de un algoritmo.

Como herramienta para modelar algoritmos de ordenación, los nodos internos representan comparaciones entre elementos de la secuencia a ordenar y los nodos externos representan el estado final de la ordenación.

Todo algoritmo de ordenación tendrá asociado su árbol de decisión particular. Una ejecución del algoritmo, para una determinada entrada, está representado por el camino de la raiz a un único nodo externo. El costo, en cantidad de comparaciones para dicha entrada, estará dado por el largo de dicho camino. De esta forma el peor caso corresponderá a la profundidad del árbol (camino más largo de la raiz a un nodo externo).

Otro elemento a notar es que todo árbol de decisión para el problema de ordenamiento tendrá como mínimo n! nodos externos y el máximo se dará cuando el árbol sea completo. Esta propiedad se usará para el problema planteado de determinar la complejidad del problema de ordenación (*Sorting*).

Se adoptará como convención que si la comparación resulta verdadera, se toma a la derecha y de lo contrario a la izquierda.

Comentarios sobre los ejemplos realizados

Puede observarse que en los dos árboles vistos la profundidad es 3 que se corresponde con la fórmula vista T(n) = $n(n-1)/2$ que era el costo en el peor caso en ambos algoritmos.

En *Selection Sort* todas las entradas tienen el mismo costo (el que representa el peor caso), esto se puede verificar gráficamente ya que el árbol correspondiente es completo.

En *Insertion Sort* sólo los nodos externos 2, 3, 4 y 6 representan el peor caso.