Priorité	Opérateur	Description	Exemple
0	**( )**	appel de fonction, associativité	foo() ; a = (b + c) * d ;
	[ ]	indexation	int tab[3] ; tab[0] = tab[1] = tab[2] = 0 ;
	.	nommage d'un champ	obj.cdr = NULL ;
	− >	nommage indirect de champ	pt− >cdr = NULL ;
1	**!**	négation	$(!a)$ est vraie si a est fausse
	˜	complément à 1	a & (˜a) = 0
	-	opposé	
	++	incrémentation	i++ ; ++i ;
	- -	décrémentation	i- - ; - -i ;
	&	adresse	int i, *pt ; pt = &i ;
	*****	valeur indirecte	*pt = 0 ; /* donne i = 0 */
	(type_de_donnée)	force le type (cast)	int i = (int)1.5 ;
	sizeof	taille en octets	i = sizeof i ;
2	*	Multiplication	
	/	Division	
	%	Modulo	
3	+	Addition	
	-	Soustraction	
4	<<	Décalage à gauche	
	>>	Décalage à droite	
5	<	Strictement inférieur	
	<=	Inférieur ou égal	
	>	Strictement supérieur	
	>=	Supérieur ou égal	
6	==	Egal	
	! =	Différent	
7	**&**	"et" binaire	
8	**ˆ**	"ou" exclusif binaire	
9	|	"ou" binaire	
10	**&&**	"et" logique	
11	||	"ou" logique	
12	**? :**	conditionnelle	`c = (a < b) ? a : b;`
13	= *= /= %= += -= ˆ= &= <<= >>= —=	Affectations	
14	**,**	Séquence	

TABLE 1 – Table des priorités des opérateurs.

```
1  #include "letemps.h"
2  #include <time.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #define N 10000
6  typedef struct fonction_t {
7    void (* fct)(int *, int);
8    char name[20];
9  } fonction_t;
10 void init(int * t, int n) {
11   int i;
12   srand(time(NULL));
13   for(i = 0; i < n; i++) t[i] = n * (rand() / (RAND_MAX + 1.));
14 }
15 void triBulles(int * t, int n) {
16   int i, j;
17   for(i = n - 1; i >= 0; i--)
18     for(j = 1; j <= i; j++)
19       if(t[j - 1] > t[j]) {
20         t[j] ^= t[j - 1]; t[j - 1] ^= t[j]; t[j] ^= t[j - 1];
21       }
22 }
23 void triSelection(int * t, int n) {
24   int i, j, min;
25   for(i = 0; i < n - 1; i++) {
26     min = i;
27     for(j = i + 1; j < n; j++)
28       if(t[j] < t[min]) min = j;
29     if(i != min) {t[i] ^= t[min]; t[min] ^= t[i]; t[i] ^= t[min];}
30   }
31 }
32 void triInsertion(int * t, int n) {
33   int i, j, v;
34   for(i = 1; i < n; i++) {
35     v = t[(j = i)];
36     while(j > 0 && t[j - 1] > v) {
37       t[j] = t[j - 1];
38       j--;
39     }
40     t[j] = v;
41   }
42 }
43 int s[N];
44 void subTriFusion(int * t, int g, int d) {
45   int i, j, k, m;
46   if(d > g) {
47     m = (d + g) >> 1;
48     subTriFusion(t, g, m);
49     subTriFusion(t, m + 1, d);
50     for(i = m; i >= g; i--)
51       s[i] = t[i];
52     for(j = m; j < d; j++)
53       s[d + m - j] = t[j + 1];
54     for(k = g, i = g, j = d; k <= d; k++)
55       t[k] = (s[i] < s[j]) ? s[i++] : s[j--];
56   }
57 }
58 void triFusion(int * t, int n) {
59   subTriFusion(t, 0, n - 1);
60 }
61 void test(int * t, int n, fonction_t * algos) {
62   while(algos->fct) {
63     init(t, n);
64     initTemps();
65     algos->fct(t, n);
66     fprintf(stderr, "%s de %d elements en \t%f secondes.\n",
67             algos->name, N, getTemps());
68     algos++;
69   }
70 }
71 int main(void) {
72   int t[N];
73   fonction_t algos[] = { {triBulles, "Tri a bulles"},
74                          {triSelection, "Tri par selection"},
75                          {triInsertion, "Tri par insertion"},
76                          {triFusion, "Tri par fusion"},
77                          {NULL, ""}};
78   test(t, N, algos);
79   return 0;
80 }
```

FIGURE 1 – Comparatif des temps d'exécution de quatre méthodes de tri.

```
1  #include "letemps.h"
2  #include <time.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #define N 10000
6  void init(int t[], int n) {
7    int i;
8    srand(time(NULL));
9    for(i = 0; i < n; i++) t[i] = n * (rand() / (RAND_MAX + 1.));
10 }
11 inline void swap(int * a, int * b) {
12   *a ^= *b;
13   *b ^= *a;
14   *a ^= *b;
15 }
16 void quickSort(int t[], int g, int d) {
17   int i, j, vpivot;
18   if( d > g ) {
19     vpivot = t[d]; i = g - 1; j = d;
20     for(;;) {
21       while(t[++i] < vpivot);
22       while(t[--j] > vpivot) if(j == i) break;
23       if(i >= j) break;
24       swap(&t[i], &t[j]);
25     }
26     if (i == d)
27       return quickSort(t, g, i - 1);
28     swap(&t[i], &t[d]);
29     quickSort(t, g, i - 1);
30     quickSort(t, i + 1, d);
31   }
32 }
33
34 int main(void) {
35   int t[N];
36   init(t, N);
37   initTemps();
38   quickSort(t, 0, N - 1);
39   fprintf(stderr, "Tri \"QuickSort\" de %d elements en %f secondes.\n", N, getTemps());
40   /* et après ???
41      void qsort(void *base, size_t nel, size_t width, int (*compar)(const void *, const void *));
42   */
43   return 0;
44 }
```

FIGURE 2 – Le Quicksort.

Résultat obtenu après exécution :

```
Tri a bulles de 10000 elements en 0.192286 secondes.
Tri par selection de 10000 elements en 0.164132 secondes.
Tri par insertion de 10000 elements en 0.016740 secondes.
Tri par fusion de 10000 elements en 0.000699 secondes.
Tri Quicksort de 10000 elements en 0.000823 secondes.
```

Listes chaînées :

- Une liste chaînée d'entiers vs Un tableau d'entiers :
  - insérer des éléments au début, à la fin, n'importe où ... ;
  - rechercher un élément ;
  - trier la liste / le tableau ;
  - supprimer un élément.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <assert.h>
5  typedef struct Noeud Noeud;
6  struct Noeud {
7    int id;
8    struct Noeud * suivant;
9  };
10 Noeud ** ajouterNoeud(int id, Noeud ** ici, Noeud ** pdebut) {
11   *ici = malloc(sizeof ** ici);
12   assert(*ici);
13   (*ici)->id      = id;
14   (*ici)->suivant = *pdebut;
15   return &((*ici)->suivant);
16 }
17 Noeud ** supprimerNoeud(Noeud ** apartir, int offset) {
18   Noeud * asupprimer;
19   while(offset > 0) { apartir = &((*apartir)->suivant); offset--; }
20   asupprimer = *apartir;
21   if(*apartir != (*apartir)->suivant) {
22     *apartir = (*apartir)->suivant;
23     free(asupprimer);
24   }
25   return apartir;
26 }
27 void afficherListe(Noeud ** pdebut) {
28   Noeud * ptr1 = *pdebut, * ptr2;
29   if(!*pdebut) return;
30   do {
31     ptr2 = ptr1->suivant;
32     printf("%d ", ptr1->id);
33   } while((ptr1 = ptr2) != *pdebut);
34   putchar('\n');
35   fflush(stdout);
36 }
37 void libererListe(Noeud ** pdebut) {
38   Noeud * ptr1 = *pdebut, * ptr2;
39   if(!*pdebut) return;
40   do {
41     ptr2 = ptr1->suivant;
42     free(ptr1);
43   } while((ptr1 = ptr2) != *pdebut);
44   *pdebut = NULL;
45 }
46 int main(void) {
47   int id, n = 35, m = 3;
48   Noeud * debut = NULL, ** ptr = &debut;
49   for(id = 1; id <= n; id++)
50     ptr = ajouterNoeud(id, ptr, &debut);
51   afficherListe((ptr = &debut));
52   while((ptr = supprimerNoeud(ptr, m)) && (*ptr != (*ptr)->suivant))
53     afficherListe(ptr);
54   debut = *ptr;
55   afficherListe(&debut);
56   libererListe(&debut);
57   return 0;
58 }
```

FIGURE 3 – Le problème de Josephus.

```
 1  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
 2  5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 1 2 3
 3  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 1 2 3 5 6 7
 4  13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 1 2 3 5 6 7 9 10 11
 5  17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 1 2 3 5 6 7 9 10 11 13 14 15
 6  21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 1 2 3 5 6 7 9 10 11 13 14 15 17 18 19
 7  25 26 27 28 29 30 31 32 33 34 35 1 2 3 5 6 7 9 10 11 13 14 15 17 18 19 21 22 23
 8  29 30 31 32 33 34 35 1 2 3 5 6 7 9 10 11 13 14 15 17 18 19 21 22 23 25 26 27
 9  33 34 35 1 2 3 5 6 7 9 10 11 13 14 15 17 18 19 21 22 23 25 26 27 29 30 31
10  2 3 5 6 7 9 10 11 13 14 15 17 18 19 21 22 23 25 26 27 29 30 31 33 34 35
11  7 9 10 11 13 14 15 17 18 19 21 22 23 25 26 27 29 30 31 33 34 35 2 3 5
12  13 14 15 17 18 19 21 22 23 25 26 27 29 30 31 33 34 35 2 3 5 7 9 10
13  18 19 21 22 23 25 26 27 29 30 31 33 34 35 2 3 5 7 9 10 13 14 15
14  23 25 26 27 29 30 31 33 34 35 2 3 5 7 9 10 13 14 15 18 19 21
15  29 30 31 33 34 35 2 3 5 7 9 10 13 14 15 18 19 21 23 25 26
16  34 35 2 3 5 7 9 10 13 14 15 18 19 21 23 25 26 29 30 31
17  5 7 9 10 13 14 15 18 19 21 23 25 26 29 30 31 34 35 2
18  13 14 15 18 19 21 23 25 26 29 30 31 34 35 2 5 7 9
19  19 21 23 25 26 29 30 31 34 35 2 5 7 9 13 14 15
20  26 29 30 31 34 35 2 5 7 9 13 14 15 19 21 23
21  34 35 2 5 7 9 13 14 15 19 21 23 26 29 30
22  7 9 13 14 15 19 21 23 26 29 30 34 35 2
23  15 19 21 23 26 29 30 34 35 2 7 9 13
24  26 29 30 34 35 2 7 9 13 15 19 21
25  35 2 7 9 13 15 19 21 26 29 30
26  13 15 19 21 26 29 30 35 2 7
27  26 29 30 35 2 7 13 15 19
28  2 7 13 15 19 26 29 30
29  19 26 29 30 2 7 13
30  2 7 13 19 26 29
31  26 29 2 7 13
32  13 26 29 2
33  13 26 29
34  26 29
35  26
```

FIGURE 4 – Sa solution.