Correction des exercices du chapitre 6

1. (a) La fonction va afficher `0 1`.
   (b) La fonction va afficher `2 3`.
   (c) La fonction va afficher `42`.
2. La fonction va afficher `0`.
3. La fonction va afficher `1`.
4. La fonction va afficher `-1`.
5. La fonction va afficher `[4]`.
6. (a) Cette fonction retourne le produit de tous les entiers dans le tableau `t` jusqu'à la première valeur 0.
   (b) Cette fonction va avancer de 1 case du tableau à chaque appel de la fonction. Le pire des cas qu'on pourrait avoir pour un tableau de taille n est d'avoir le premier élément nul en tant que dernier élément. Nous devrons alors effectuer $n+1$ appels. La complexité en temps est donc égale à la profondeur de récursion : $\mathcal{O}(n)$.
   (c) On peut réécrire cette fonction de manière itérative de la façon suivante :

```
long f(int *t) {
    long res = 1;
    for (int i = 0; t[i]; i++)
        res *= t[i];

    return res;
}
```

7. Cette fonction retourne une valeur indiquant si le nombre de chiffres contenus dans la chaine de caractères dans `s` est impair ou pas.
8. Cette fonction retourne la longueur du plus grand préfixe commun des deux chaines de caractères, `s` et `t`.
9. Cette fonction compte le nombre d'éléments du tableau `n` qui sont divisibles par `m`.
10. (a) Cette fonction retourne le produit des éléments du tableau.
    (b) On peut réécrire cette fonction de manière itérative de la façon suivante :

```
float f(float t[], unsigned n) {
    float res = 1;

    for (unsigned i = 0; i < n; i++)
        res *= t[i];

    return res;
}
```

11. (a) Cette fonction retourne une valeur booléenne disant si les deux chaines de caractères sont identiques ou pas.
    (b) On peut réécrire cette fonction de manière itérative de la façon suivante :

```
int p(char *s1, char *s2) {
    for (; *s1; s1++, s2++)
        if (*s1 != *s2)
            return 0;
    return 1;
}
```

12. (a) Cette fonction retourne le nombre de caractères de s qui sont égaux au caractère c.

    (b) Nous allons appeler la fonction jusqu'à arriver au caractère terminateur tout en parcourant la chaine de caractère élément par élément. Nous avons donc une profondeur de récursion égale à la taille de la chaine de caractères. La complexité en espace est $\mathcal{O}(n)$.

    (c) On peut réécrire cette fonction de manière itérative de la façon suivante :

```
unsigned f(char *s, char c) {
    unsigned r = 0;
    while(*s) {
        if (*s == c)
            r++;
        s++;
    }

    return r;
}
```

13. La fonction peut s'écrire :

```
int rand_tab(unsigned int n, int min, int max, int **dest) {
    if (min > max)
        return 0;

    dest = malloc(sizeof(int*));
    if (!dest)
        return 0;

    *dest = malloc(sizeof(int)*n);
    if(!*dest) {
        free(dest);
        return 0;
    }

    for (unsigned i = 0; i < n; i++)
        *dest[i] = rand(min, max);

    return 1;
}
```

14. La fonction peut s'écrire :

```
unsigned g(unsigned n) {
    unsigned nb = 0;
    int *x;
    for (int res = f(x);res && nb < n;res = f(x), nb++);
```

```
    return nb;
}
```

15. (a) Pour résoudre ce problème, nous devons parcourir le vecteur et pour chaque élément trouvé, mettre à jour l'élément b correspondant :

```
void p(unsigned char *v, unsigned n, int *b) {
    for (unsigned i - 0; i < 256; i++)
        b[i] = 0;

    for (unsigned i = 0; i < n; i++)
        b[v[i]] = 1;
}
```

    (b) La première boucle effectue toujours 256 itérations, on considérera donc ceci comme une complexité constante. La deuxième boucle va itérer de 0 à $n-1$. Nous avons donc une complexité théorique de $\mathcal{O}(n)$.

16. Il nous suffit d'allouer une nouvelle chaine de caractères de taille égale à la somme des tailles des deux chaines (+1 pour le caractère terminateur) et d'écrire dedans caractère par caractère.

```
char *concatenation(char *s1, chae *s2) {
    unsigned l1 = strlen(s1);
    unsigned l2 = strlen(s2);

    char *s, *c;
    s = malloc(sizeof(char)*(l1+l2+1));
    if (!s)
        return NULL;

    for (c = s; *s1; s1++, p++)
        *p = *s1;

    for (; *s2; s2++, p++)
        *p = *s2;

    *p = '/0';
    return s;
}
```

17. Pour résoudre ce problème, nous pouvons itérer en même temps sur $t1$ et $t2$ et les ajouter dans le nouveau tableau.

```
int *f(int *t1, int *t2, unsigned n) {
    int *t;
    t = malloc(sizeof(int)*(2*n));
    if (!t)
        return NULL;

    for (unsigned i = 0; i < n; i++) {
        t[2*i] = t1[i];
        t[2*i+1] = t2[i];
```

```
    }
    return t;
}
```

18. (a) Comme nous avons chaque fois besoin de la ligne précédente pour calculer l'actuelle, dans un soucis d'optimiser l'espace, nous créerons seulement 2 vecteurs au lieu de créer le triangle en entier et de le garder.

```
unsigned *pascal(unsigned i) {
    unsigned *l1, *l2;
    l1 = malloc(sizeof(unsigned)*(i+1));
    if (!l1)
        return NULL;
    l2 = malloc(sizeof(unsigned)*(i+1));
    if (!l2) {
        free(l1);
        return NULL;
    }

    for (unsigned j = 0; j <= i; j++) {
        l2[0] = 1;
        l2[j] = 1;
        for (unsigned k = 1; k < j; k++)
            l2[k] = l1[k-1] + l1[k];
        for (unsigned k = 0; k <= j; k++)
            l1[k] = l2[k];
    }

    free(l1);

    return l2;
}
```

(b) Nous créons 2 vecteurs de taille $i + 1$, nous avons une complexité en espace correspondant à $2 \times (i + 1)$ ce qui nous donne une complexité théorique de $\mathcal{O}(n)$. Pour la complexité en temps, nous effectuons des boucles imbriquées. La première itère de 0 à i. La 1ère boucle à l'intérieur va itérer de 1 à $j - 1$. Le pire cas pour cette boucle est quand $j = i$. La deuxième boucle intérieure va itérer de 0 à j. Le pire cas arrive également quand $j = i$. Nous avons alors une complexité ressemblant à : $(i + 1) \times ((i - 1) + (i + 1))$. Cette complexité, une fois approximée, nous donne une complexité théorique en temps de $\mathcal{O}(i^2)$.

19. (a) Le type structuré peut s'écrire :

```
typedef struct {
    float x;
    float y;
} point;

typedef struct {
    point origine;
    point destination;
} segment;
```

(b) Cette fonction peut s'écrire :

```
segment *creer(float x1, float y1, float x2, float y2) {
    segment *nouv;
    nouv = malloc(sizeof(segment));
    if (!nouv)
        return NULL;

    nouv->origine.x = x1;
    nouv->origine.y = y1;
    nouv->destination.x = x2;
    nouv->destination.y = y2;

    return nouv;
}
```

(c) Cette fonction peut s'écrire :

```
void liberer(segment *s) {
    free(s);
}
```

20. (a) Ce type structuré peut s'écrire :

```
typedef struct {
    char nom[21];
    char prenom[21];
} personne;
```

(b) Cette fonction peut s'écrire :

```
unsigned egal(char *s1, char *s2) {
    for (;*s1 || *s2; s1++, s2++) {
        if (*s1 == *s2)
            continue;
        if (*s1 >= 'A' && *s1 <= 'Z' &&
            *s1 + 'a' - 'A' == *s2)
            continue;
        if (*s2 >= 'A' && *s2 <= 'Z' &&
            *s1 == *s2 + 'a' - 'A' )
            continue;
        return 0
    }
    return 1;
}

unsigned meme_personne(personne *p1, personne *p2) {
    return (egal(p1->nom, p2->nom) &&
            egal(p1->prenom, p2->prenom));
}
```

21. (a) Pour ce type structuré, nous allons ajouter un champs nous permettant de savoir si la borne est infinie ou pas. Ce type structuré peut alors s'écrire :

```
typedef struct {
    unsigned a_inf, b_inf;
    int a, b;
} intervalle;
```

(b) Cette fonction peut s'écrire :

```
unsigned inclu(intervalle i1, intervalle i2) {
    if (i1.a_inf && !i2.a_inf)
        return 0;

    if (!i2.a_inf && i1.a < i2.a)
        return 0;

    if (i1.b_inf && !i2.b_inf)
        return 0;

    return (i2.b_inf || i1.b <= i2.b);
}
```

22. (a) Ce type structuré peut s'écrire :

```
typedef struct element_t {
    int val;
    struct element_t *suiv;
} element;

typedef struct {
    element *premier;
} liste;
```

(b) Cette fonction peut s'écrire :

```
liste *creer() {
    liste *l;
    l = malloc(sizeof(liste));
    if (!l)
        return NULL;

    l->premier = NULL;

    return l;
}
```

(c) Cette fonction peut s'écrire :

```
void inserer(liste *l, int v) {
    element *e;
    e = malloc(sizeof(element));
    if (!e)
        return;

    e->val = v;
    e->suivant = l->premier;
```

```
    l->premier = e;
}
```

(d) Cette fonction peut s'écrire :

```
int somme(liste *l) {
    element *e;
    int res = 0;

    for (e l->premier; e; e = e->suiv)
        res += e->val;

    return res;
}
```

23. (a) Le type structuré peut s'écrire :

```
typedef struct element_t {
    int val;
    struct element_t suiv;
} element;
```

(b) Cette fonction peut s'écrire :

```
element *tampon(int *t, unsigned n) {
    unsigned i;
    element *e, *courant, *dernier;

    for (i = 0; i < n; i++) {
        e = malloc(sizeof(element));
        if (!e)
            return NULL;

        e->val = t[n - i - 1];

        if (i)
            e->suiv = courant;
        else
            dernier = e;

        courant = e;
    }

    if (i)
        dernier->suiv = e;

    return e;
}
```

(c) Cette fonction peut s'écrire :

```
int somme(element *e) {
    element *courant;
    int res = 0;
```

```
    for (courant = e; ; ){
        res += courant->val;
        courant = courant->suiv;
        if (courant == e)
            return res;
    }
}
```

(d) Cette fonction peut s'écrire :

```
void liberer(element *e) {
    element *courant, *suivant;

    for (courant = e; ; ) {
        suivant = courant->suivant;
        free(courant);
        if (suivant == e)
            return;
        courant = suivant;
    }
}
```

24. (a) Ce type structuré peut s'écrire :

```
typedef struct {
    double x, y, z, r;
} sphere;
```

(b) Cette fonction peut s'écrire :

```
sphere *creer(double x, double y, double z, double r) {
    sphere *s;
    if (r < 0.0)
        return NULL;

    s = malloc(sizeof(sphere))
    if (!s)
        return NULL;

    s->x = x;
    s->y = y;
    s->z = z;
    s->r = r;

    return s;
}
```

(c) Cette fonction peut s'écrire :

```
sphere *plus_petite(sphere *s[], unsigned n) {
    unsigned min = 0;

    for (unsigned i = 1; i < n; i++)
```

```
        if(s[i]->r < s[min]->r)
            min = i;

    return s[min];
}
```

25. (a) Ce type structuré peut s'écrire :

```
typedef struct {
    char c[101];
} chaine;

typedef struct {
    chaine **elements;
    unsigned lignes, colonnes;
} tableau;
```

(b) Cette fonction peut s'écrire :

```
tableau *creer(unsigned n, unsigned m) {
    tableau *t;
    t = malloc(sizeof(tableau));
    if (!t)
        return NULL;

    t->lignes = n;
    t->colonnes = m;

    t->elements = malloc(sizeof(elements*)*n);
    if (!t->elements) {
        free(t);
        return NULL;
    }

    for (unsigned i = 0; i < n; i++) {
        t->elements[i] = malloc(sizeof(element)*m);
        if (!t->elements[i]) {
            for (unsigned j = 0; j < i; j++)
                free(t->elements[j]);
            free(t->elements);
            free(t);
            return NULL;
        }
        for (unsigned j = 0; j < m; j++)
            t->elements[i][j].c[0] = '\0';
    }
    return t;
}
```

(c) Cette fonction peut s'écrire :

```
unsigned egal(tableau *t1, tableau *t2) {
    if (t1->lignes != t2->lignes ||
        t1->colonnes != t2->colonnes)
```

```
        return 0;

    for (unsigned i = 0; i < t1->lignes; i++)
        for (unsigned j = 0; j < t1->colonnes; j++)
            for (unsigned k = 0; k < 101; k++) {
                if (!t1->elements[i][j].c[k] &&
                    !t2->elements[i][j].c[k])
                        break;
                if (t1->elements[i][j].c[k] !=
                    t2->elements[i][j].c[k])
                    return 0;
            }
    return 1;
}
```

26. (a) Ce type structuré peut s'écrire :

```
typedef struct {
    unsigned lignes;
    unsigned *taille_lignes;
    int **elements;
} tableau;
```

(b) Cette fonction peut s'écrire :

```
tableau *creer(unsigned taille[], unsigned n) {
    tableau *t;
    t = malloc(sizeof(tableau));
    if (!t)
        return NULL;

    t->lignes = n;
    t->taille_lignes = malloc(sizeof(n));
    if (!t->taille_lignes) {
        free(t);
        return NULL;
    }

    t->elements = malloc(sizeof(int *)*n);
    if (!t->elements) {
        free(t->taille_lignes);
        free(t);
        return NULL;
    }

    for (unsigned i = 0; i < n; i++) {
        t->taille_lignes[i] = taille[i];
        t->elements[i] = malloc(sizeof(int)*taille[i]);
        if (!t->elements[i]) {
            for (unsigned j = 0; j < i; j++)
                free(t->elements[j]);

            free(t->elements);
```

```
            free(t->taille_lignes);
            free(t);
            return NULL;
        }
        for (unsigned j = 0; j < taille[j]; j++)
            t->elements[i][j] = 0;
    }
    return t;
}
```

(c) Cette fonction peut s'écrire :

```
void liberer(tableau *t) {
    for (unsigned i = 0; i < t->lignes; i++)
        free(t->elements[i]);

    free(t->elements);
    free(t->taille_lignes);
    free(t);
}
```

(d) Ce prototype peut s'écrire :

```
static int meme_forme(tableau *, tableau *);
```

27. (a) Ce type structuré peut s'écrire :

```
typedef struct bloc_t {
    int elements[1000];
    unsigned n;
    struct bloc_t *suiv;
}bloc;
```

(b) Cette fonction peut s'écrire :

```
void liberer(bloc *b) {
    while (b) {
        bloc *suiv = b->suiv;
        free(b);
        b = suiv;
    }
}

bloc *creer(int *v, unsigned n) {
    bloc *b, *liste;
    unsigned start = 0;

    for (liste = NULL; n; ) {
        b = malloc(sizeof(bloc));
        if (!b) {
            liberer(liste);
            return NULL;
        }
        b->n = (n-1)%1000+1;
```

```
            for (unsigned i = 0; i < b->n; i++)
                b->elements[i] = v[i + n - b->n];
            b->suivant = liste;
            liste = b;
            n -= b->n;
        }
        return liste;
    }
```

(c) Cette fonction peut s'écrire :

```
unsigned taille(bloc *b) {
    unsigned taille;

    for (taille = 0; b; taille += b->n, b = b->suivant);

    return taille;
}
```

28. (a) Ce type structuré peut s'écrire :

```
typedef struct element_t {
    char *mot;
    struct element_t *suivant;
} element_ligne;
```

(b) Cette fonction peut s'écrire :

```
unsigned longueur(element_ligne *e) {
    unsigned longueur;

    for (longueur = 0; e; e = e->suivant) {
        char *c = e->mot;
        while (*c) {
            longueur++;
            c++;
        }
        if (e->suivant)
            longueur++;
    }
}
```

(c) Cette fonction peut s'écrire :

```
char *chaine(element_ligne *e) {
    char *s, *p;
    unsigned longueur;

    s = malloc(longueur(e) + 1);
    if (!s)
        return NULL;

    for (p = s; e; e = e->suivant) {
        longueur = 0;
```

```
            char *c = e->mot;
            while (*c) {
                longueur++;
                c++;
            }
            for (unsigned i = 0; i < longueur; i++)
                p[i] = e->mot[i];
            if (!e->suivant)
                p[longueur] = '\0';
            else
                p[longueur] = ' ';
            p += longueur + 1;
        }
        return s;
    }
```

29. (a) Ce type structuré peut s'écrire :

```
    typedef struct seq_t {
        unsigned n_mots;
        char **mots;
    } seq;
```

(b) Cette fonction peut s'écrire :

```
    seq *decomp(char *c) {
        seq *s;
        char *p;
        unsigned n, i;

        s = malloc(sizeof(seq));
        if (!s)
            return NULL;

        for (p = c, s->n_mots = 0, n = 0; *p; ) {
            if (*p == ' ')
                p++;
            else {
                s->n_mots++;
                for (n++, p++; *p && *p != ' '; n++, p++);
            }
        }
        if (!s->n_mots)
            return s;

        s->mots = malloc(sizeof(char *)s->n_mots);
        if (!s->mots) {
            free(s);
            return NULL;
        }

        p = malloc(n + s->n_mots)
        if (!p) {
```

```
            free(s->mots);
            free(s);
            return NULL;
        }

        for (i = 0; *c; ) {
            if (*c == ' ')
                c++;
            else {
                for (s->mots[i++] = p; *c && *c != ' '; c++)
                    p++ = *c;
                *p++ = '\0';
            }
        }
        return s;
    }
```

(c) Cette fonction peut s'écrire :

```
    void liberer(seq *s) {
        if (s->n_mots) {
            free(s->mots[0]);
            free(s->mots);
        }
        free(s);
    }
```

30. (a) Ce type structuré peut s'ecrire :

```
    typedef struct date_t {
        unsigned jour, mois;
        int annee;
    } date;
```

(b) Cette fonction peut s'écrire :

```
    int anterieur(date *d1, date *d2) {
        if (d1->annee < d2->annee)
            return 1;

        if (d1->annee > d2->annee)
            return 0;

        if (d1->mois < d2->mois)
            return 1;

        if (d1->mois > d2->mois)
            return 0;

        if (d1->jour < d2->jour)
            return 1;

        return 0;
    }
```

31. (a) Ce type structuré peut s'écrire :

```
typedef struct seq_t {
    char *chaines[100];
    unsigned n_chaines;
} seq;
```

(b) Cette fonction peut s'écrire :

```
int seq_identiques(seq *s1, seq *s2) {
    if (s1->n_chaines != s2->n_chaines)
        return 0;

    for (unsigned i = 0; i < s1->n_chaines; i++)
        if (strcmp(s1->chaines[i], s2->chaines[i]))
            return 0;

    return 1;
}
```

32. (a) Ce type structuré s'écrit :

```
typedef struct lasso_t {
    char c;
    struct lasso_t *next;
} lasso;
```

(b) Cette fonction peut s'écrire :

```
lasso *nouv_lasso(char *b, char *p) {
    unsigned longueur_b;
    unsigned longueur_p;

    for (longueur_b = 0; b[longueur_b]; longueur_b++);

    for (longueur_p = 0; p[longueur_p]; longueur_p++);

    if (!longueur_b || !longueur_p)
        return NULL;

    lasso *e;
    e = malloc(sizeof(lasso)*(longueur_p + longueur_b))
    if (!e)
        return NULL;

    for (unsigned i = 0; i < longueur_b; i++)
        e[i].c = b[i];

    for (unsigned i = 0; i < longueur_p; i++)
        e[longueur_b + i].c = p[i];

    for (unsigned i = 0; i < longueur_b + longueur_p; i++)
        e[i].next = e + i + 1;
```

```
    e[longueur_b + longueur_p].next = e + longueur_b;

    return e;
}
```

(c) Cette fonction peut s'écrire :

```
void liberer(lasso *e) {
    free(e);
}
```

33. (a) Ce type structuré peut s'écrire :

```
typedef struct lieu_t {
    char *nom;
    struct lieu_t *suivant;
} lieu;
```

(b) Cette fonction peut s'écrire :

```
void liberer(lieu *p) {
    lieu *p_suivant;

    for (; p; p = p_suivant) {
        p_suivant = p->suivant;
        free(p);
    }
}

lieu *creer(char *s) {
    lieu *premier, **q_precedent;

    for (premier = NULL, q_precedent = &premier; *s; s++) {
        lieu *q;
        q = malloc(sizeof(lieu));
        if (!q) {
            liberer(premier);
            return NULL;
        }

        q->nom = s;
        q->suivant = NULL;

        *q_precedent = q;
        q_precedent = &q->suivant;

        while (*s && *s != ' ')
            s++;

        *s = '\0';
    }
    return premier;
}
```

(c) Cette fonction peut s'écrire :

```
unsigned compter(lieu *p) {
    lieu *p_suivant;
    unsigned n;

    for (n = 0; p; p = p_suivant) {
        n++;
        p_suivant = p->suivant;
        free(p);
    }
    return n;
}
```