Correction du problème "élimination des doublons"

1.1. On trouve la longueur de la chaîne en examinant successivement chaque caractère, et en le comparant à \0

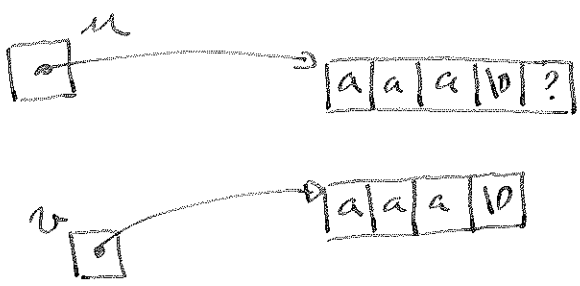
```
int strlen(char* s) {
    int i = 0;
    while (s[i] != '\0')  i++;
    return i;
}
```

1.2 Pour copier la chaîne, il faut réserver un espace mémoire, puis y copier chaque caractère à l'aide d'une boucle.

```
char* strcpy(char* s) {

    int n = strlen(s); int i;
    char* r = (char*) malloc((n+1) * sizeof(char));
    for(i = 0; i < n; i++)
        r[i] = s[i];
    r[n] = '\0';
    return r;
}
```

Note: l'espace à réserver est de taille n+1 si n est la longueur de la chaîne : il faut l'espace pour stocker le \0 terminal.



Ci-contre, l'état de la mémoire après exécution des 2 instructions.

1.3 Pour comparer les fonctions selon l'ordre lexicographique, on parcourt les 2 chaînes de ~~droite~~ gauche à droite, en comparant leurs caractères.

Version récursive

```
int strcmp(char* u, char* v) {
    if(u[0] == '\0') {
        if(v[0] == '\0') return 0;
        else return -1;
    }
    else if(v[0] == '\0') return 1;
    else if(u[0] < v[0]) return -1;
    else if(u[0] > v[0]) return 1;
    else return strcpy(u+1, v+1);
}
```

La version itérative est assez semblable. Le corps de la fonction est constitué d'une boucle while dont les seules sorties éventuelles sont les instructions return. Celles-ci sont évitées seulement si u[i] = v[i] et u[i] ≠ '\0'. Or u et v étant de longueur finie et terminées par '\0', ceci arrive nécessairement pour un nombre suffisant d'itérations. Donc la fonction termine.

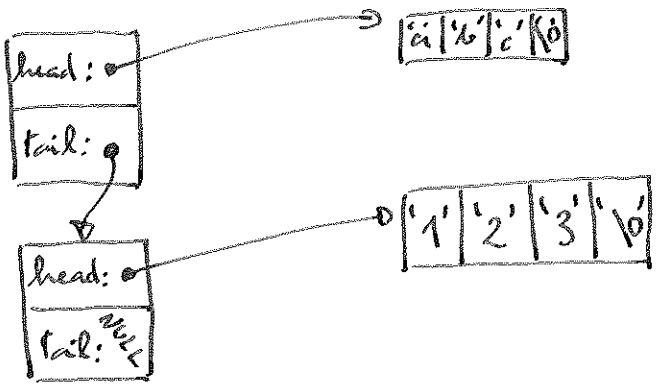
```
int strcmp(char* u, char* v) {
int i = 0;
while(1) {
    if(u[i] == '\0') {
        if(v[i] == '\0') return 0;
        else return -1;
    }
    else if(v[0] == '\0') return 1;
    else if(u[0] < v[0]) return -1;
    else if(u[0] > v[0]) return 1;
    i++
}
}
```

On pourrait faire un commentaire analogue pour la version récursive.

2.1 On suit la structure des listes chaînées vue en cours. La liste vide est conventionnellement représentée par NULL, sinon :

```
struct list_t {
    char* head;
    struct list_t * tail;
};
typedef struct list_t* list;
```

2.2 La liste "abc" :: "123" :: ∅ sera représentée en mémoire par



2.3 Le constructeur s'écrit :

```
list cons(char* h, list tail) {
    list r = (list) malloc(sizeof(struct list_t));
    r->head = strcpy(h);
    r->tail = tail;
    return r;
}
```

2.4 Ce calcul s'effectue récursivement :

```
int total_length (list l) {
    if (l == NULL) return 0;
    else {
        int shead = strlen (l->head);
        int stail = total_length (l->tail);
        return shead + stail;
    }
}
```

2.5 Il faut penser à libérer (dans le bon ordre) la chaîne et la structure :

```
void free_list (list l) {
    if (l != NULL) {
        free_list (l->tail);
        free (l->head);
        free (l);
    }
}
```

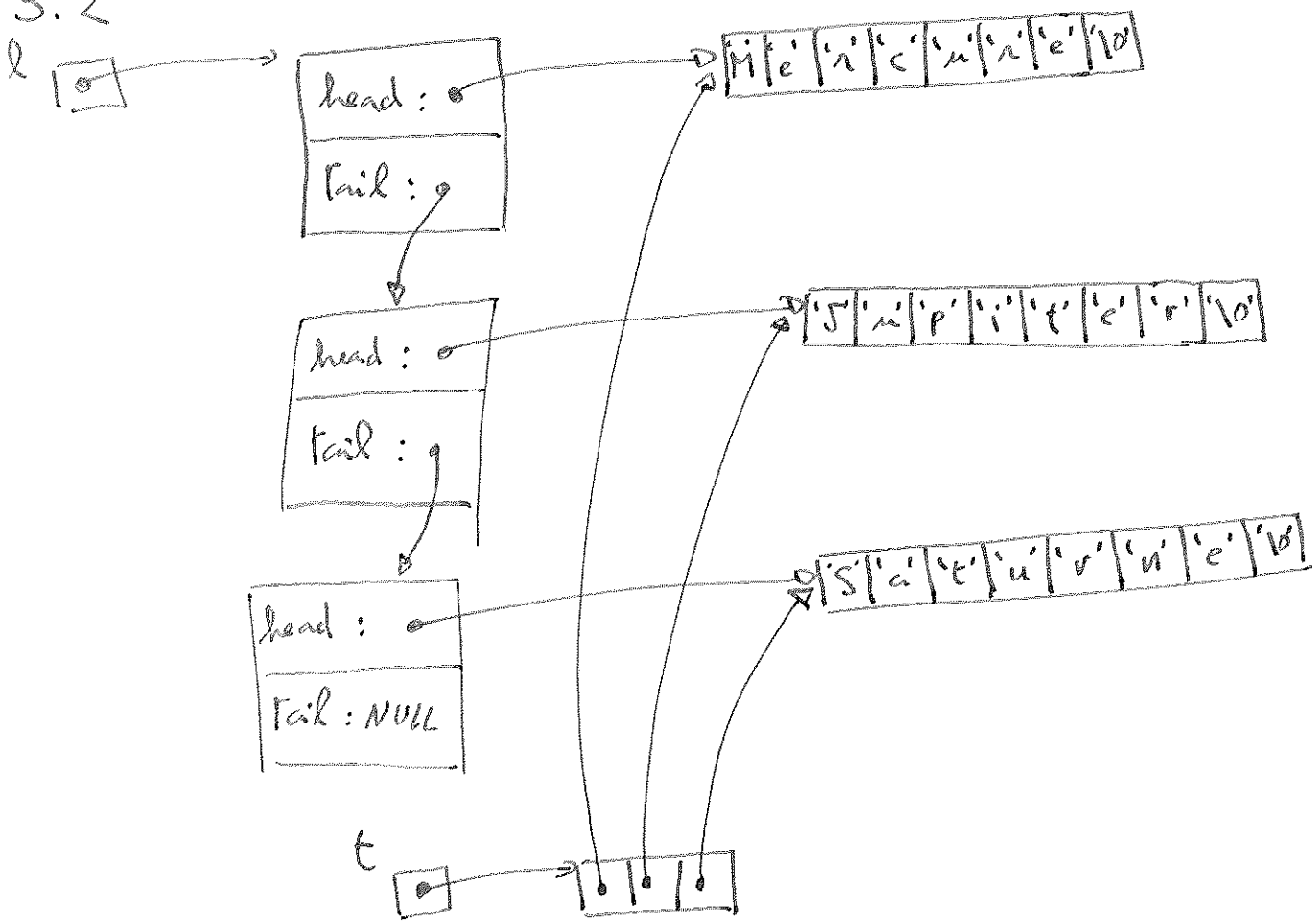
3.1

```
char ** array_from_list (list l) {
    int n = length (l); // définie plus bas
    char** r = (char**) malloc (n * sizeof (char*));
    int i; list p = l;
    for (i=0; i<n; i++) {
        r[i] = p->head;
        p = p->tail;
    }
    return r;
}
```

```
int length(list l) {
    if (l == NULL) return 0;
    else return 1 + length(l->tail);
}
```

Remarque: pour la fonction array_from_list, le style itératif convient mieux, puisqu'il faut remplir un tableau.

3.2



3.3

```
void sort(char** t, int n) {
    int i, j, k;
    for(i=0; i<n-1; i++) {
        k = i;
        for(j=i+1; j<n; j++)
            if (strcmp(t[j], t[k]) < 0) k = j;
        char* temp = t[k];
        t[k] = t[i];
        t[i] = temp;
    }
}
```

L'appel à strcmp est imbriqué dans deux boucles. A la i^e itération de la boucle englobante, strcmp est appelée $n-i-1$ fois dans la boucle interne. Le nombre total d'appel est donc $\sum_{i=0}^{n-2}$

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

$$= n(n-1) - \frac{n(n-1)}{2}$$

$$= \frac{n(n-1)}{2}$$

$$= \mathcal{O}(n^2)$$

La complexité est quadratique.

3.4 On adopte encore un style itératif, pour faciliter le parcours du tableau:

```
list list_from_array(char** t, int n) {
    list r = NULL; int i;
    for(i=n-1; i>=0; i--)
      {r = cons(t[i], r);}
    return r;
}
```

Note : il faut prendre garde à parcourir le tableau dans le bon sens, sinon on inverse l'ordre de la liste.

4.1
```
int myst(list l) {
    return (l!=NULL)&&(l->tail!=NULL)
                    &&(strcmp(l->head, l->tail->head)==0);
}
```

C'est une forme un peu compacte, on aurait pu utiliser des 'if'.

par exemple :

```
int mystr(list l) {
    if (l != NULL) {
        if (l->tail != NULL) {
            if (strcmp(l->head, l->tail->head)==0)
                return 1;
            else return 0;
        }
        else return 0;
    }
    else return 0;
}
```

La fonction pourrait s'appeler "commence_par_doublon" puisqu'elle permet de détecter si le premier et le deuxième élément sont identiques.

4.2 Comme en TD, mais attention à bien libérer toute la mémoire :

```
list drop(list l) {
    if (l == NULL) return NULL;
    else {
        list r = l->tail;
        free(l->head);
        free(l);
        return r;
    }
}
```

4.3 On procède récursivement

```
list supprime_doublons(list l) {
    if (l == NULL) return l;
    else {
        if(commence_par_doublon(l))
            return supprime_doublons(chop(l));
        else {
            list tail = supprime_doublons(l->tail);
            l->tail = tail;
            return l;
        }
    }
}
```

Conclusion

```
list uniq(list l) {
    char** t = array_from_list(l);
    int n = length(l);
    sort(t, n);
    list r = list_from_array(t, n);
    free(t)
    return supprime_doublons(r);
}
```

Complexité :

1 : $O(1)$
2 : $O(1)$
3 : $O(n^2)$ appels à strcmp
4 : $O(n)$ appels à strcpy et à strlen
5 : $O(1)$
6 : $O(n)$ appels à strcmp au pire

Soit une complexité quadratique pour l'ensemble.

On aurait obtenu une complexité identique avec un algorithme beaucoup plus simple : créer une nouvelle liste en ajoutant un à un les éléments de la liste de départ, en ayant au préalable vérifié qu'ils ne se trouvaient pas déjà dans la liste en cours de construction.

En revanche, si l'on remplace le tri par insertion par un tri rapide (en $O(n \log n)$), l'approche proposée dans le devoir est asymptotiquement plus rapide.