Complexité d'un algorithme

I Généralités

La question de la performance est centrale en informatique. Si un problème est traité en un temps raisonnable, l'utilisateur est satisfait. On s'attache en général peu à une mesure précise du temps d'exécution, mais uniquement à une approximation de ce temps. Si le temps obtenu est perçu comme relativement long, se posera la question d'une solution plus efficace. La machine étant fixée, l'amélioration portera alors sur la façon dont sont conçus les algorithmes en fonction du volume n de données à traiter. Par exemple, si en doublant le volume de données, un algorithme renvoie un résultat en multipliant le temps d'exécution par 2 et un autre le renvoie en multipliant le temps d'exécution par $n > 2$, on optera évidemment pour le premier : on dit que **sa complexité temporelle** est meilleure. Ajoutons que l'on peut aussi définir **sa complexité spatiale** afin de prendre en compte l'espace mémoire occupé au cours de l'exécution d'un algorithme. Plus sa complexité est grande, plus le programme mettant en oeuvre l'algorithme aura besoin de zones mémoires pour stocker les données.

I.1 Complexité temporelle et spatiale d'un algorithme

Définition : **La complexité**[1] **temporelle d'un algorithme** est une information[2] sur son temps d'exécution liée au volume n de données à traiter. **La complexité temporelle d'un programme** est la complexité temporelle de l'algorithme associé.

Remarques :

- La complexité temporelle ne dépend pas :
  - des performances du processeur de la machine sur laquelle le programme est exécuté,
  - des modes d'accès à la mémoire vive et des modes de stockage,
  - du langage de programmation dans lequel l'algorithme est traduit,
  - du compilateur/interpréteur exécuté.

  On se place sur une machine idéalisée dont la mémoire est considérée comme infinie et l'accès aux données se fait en temps constant.
- Le volume de données (on dit aussi *la taille* des données) dépend du problème étudié :
  - il peut s'agir du nombre d'éléments constituant les paramètres de l'algorithme (par exemple le nombre d'éléments d'un tableau),
  - dans le cas d'algorithmes de nature arithmétique (le calcul de $n!$ par exemple), il peut s'agir d'un entier passé en argument,
  - dans le cas d'algorithmes avec des chaînes de caractères, il peut s'agir de la longueur des chaînes de caractères.

  Noter que le volume de données est un entier naturel caractérisant le nombre de données et non son occupation mémoire en octets (ou en bits).

Définition : **La complexité spatiale d'un algorithme** est une estimation de l'espace mémoire occupé au cours de l'exécution d'un programme en fonction du volume n de données à traiter. **La complexité spatiale d'un programme** est la complexité spatiale de l'algorithme associé.

Le programme de CPGE met l'accent sur la complexité temporelle et c'est cette notion que nous allons développer.

1. On dit aussi le « coût ».
2. Ce n'est pas le temps d'exécution réel sur une machine donnée, avec des logiciels donnés et des processus en cours d'exécution.

I.2 Mesure de la complexité temporelle

La mesure de la complexité temporelle nécessite **un modèle**. Dans celui-ci :

— on précise **les opérations élémentaires** qui, par hypothèse, **se déroulent en temps borné par une constante**,

— on suppose que **la complexité temporelle d'un algorithme est proportionnelle au nombre d'opérations élémentaires**.

Sans être exhaustif, les opérations génériques suivantes sont considérées comme **élémentaires** :

— les opérations arithmétiques (`+`,`-`,`*`,`/`,`//`,`%`),

— la comparaison de données : *relation d'égalité* (`==`), *d'ordre* (`<`,`>`,...),

— le transfert de données : *lecture* (`input`, `open`, `readline`,...) et *écriture dans un emplacement mémoire* (`print`, `close`, `writeline`,...),

— les instructions de contrôle (`if`,`elif`,`else`, `while`,`for`),

— l'affectation en général (sans tenir compte de la complexité liée à l'exécution de l'expression à droite de celle-ci), càd. `a =`; par exemple `a = 0`.

Remarques : Les hypothèses ci-dessus sont acceptées, mais simplificatrices. En effet, elles dépendent partiellement du langage de programmation. Par exemple,

— Les entiers python sont de type `long` (i.e. 8 octets) (à partir de la version 3.0) : les opérations arithmétiques ne sont donc pas toujours élémentaires. On supposera cependant leur temps d'exécution borné.

— La comparaison entre chaînes de caractères n'est pas élémentaire. Seule la comparaison entre **deux** caractères sera supposée élémentaire.

— La recopie d'un tableau entier n'est pas élémentaire. Seule **la copie d'un élément** d'un tableau sera supposée élémentaire.

Par ailleurs, il existe des opérations liées à des structures de données, appelées des types dans Python. Certaines opérations **se déroulent en temps constant** (ou *en temps amorti constant*[3]) et on peut les considérer comme des opérations **élémentaires** :

— l'ajout dans un tableau `t` de type `list`, càd. `t.append(item)` ou `t[i] = item`

— l'ajout dans un dictionnaire `d` via `d[i]=item`

— l'accès à un élément via son index dans un itérable (tableau `t` de type `list`, dictionnaire `d`, chaîne de caractères `s`,...) càd. `t[i]`, `d[i]`, `s[i]`, ...

— l'accès à la longueur d'un itérable (tableau `t` de type `list`, dictionnaire `d`, chaîne de caractères `s`,...) càd. `len(t)`, `len(d)`, `len(s)`,...

— la suppression d'un élément dans un dictionnaire càd. `del dict[key]` (alors que ce n'est pas en temps constant pour un tableau de type `list`)

— la récupération des clés d'un dictionnaire càd. `dict.keys()`

— le test d'existence d'un élément dans un dictionnaire (type `dict`) ou un ensemble (type `set`) via `e in s` (ou `e not in s`). **Attention ! Ce test d'existence avec les tableaux de type `list` n'est absolument pas une opération élémentaire**.

Parmi les opérations non élémentaires, on peut citer sans être exhaustif : la recherche d'un élément dans un tableau de type `list`, la suppression d'un élément dans un tableau de type `list`, les tests d'égalité entre listes (`t1 == t2`), le slicing de listes (`t[a:b]`).

3. Cela signifie que le temps est constant dans la grande majorité des cas, mais qu'il peut y avoir des exceptions.

I.3 La notation $\mathcal{O}$

Notation de Landau : on dit qu'une suite numérique (u_n) est **dominée** par la suite (v_n), et on note $u_n = \mathcal{O}(v_n)$, quand il existe $n_0 \in \mathbb{N}$ et un réel $k > 0$ tel que :

$$\forall n \geq n_0, \quad |u_n| \leq k|v_n|$$

Exemple : Considérons le programme de détermination du nombre d'occurrences dans un tableau de type `list`. On veut déterminer la complexité de la fonction `nbre_occurrences`.

```
2 def nbre_occurrences(x:object, t:list) -> int:
3     c = 0
4     for i in range(len(t)):
5         if t[i] == x:
6             c = c + 1
7     return c
```

Complexité :

- L'affectation `c = 0` compte pour une opération élémentaire (on dit qu'elle est en $\mathcal{O}(1)$, car majorée par une constante).
- La boucle bornée `for i in range(len(t))` se déroule `len(t)` fois, où `len(t)` est une opération élémentaire.
- L'instruction de contrôle `if`, l'accès à `t[i]`, ainsi que le test d'égalité `t[i] == x` se déroulent en temps constant (3 opérations).
- Le calcul de l'expression `c+1` ainsi que l'affectation `c = c+1` se déroulent en temps constant (2 opérations).
- `return c` est une opération élémentaire.

Conclusion : Si `n = len(t)` (*taille de l'entrée*), le traitement se réalise en un maximum de $5n+2$ opérations élémentaires. Or, $5n + 2 \leq 6n$, donc que **la complexité temporelle dans le pire des cas** de la fonction `nbre_occurrences` est $\mathcal{O}(n)$. On dit aussi que la complexité temporelle est **linéaire**.

Une fois le travail d'analyse détaillée ci-dessus fait, on se rend compte que l'on aurait pu le simplifier comme suit grâce à la notation de Landau :

La longueur du tableau `n = len(t)` est une mesure de la taille du problème considéré.

- L'affectation `c = 0` se déroule en $\mathcal{O}(1)$.
- La boucle bornée `for i in range(len(t))` se déroule `n` fois. De plus, `len(t)` se déroule en $\mathcal{O}(1)$.
  - Les instructions `if t[i] == x` et `c = c + 1` se déroulent en $\mathcal{O}(1)$. Donc dans le pire des cas, la boucle `for` et son bloc de code se déroulent en $\mathcal{O}(n)$.
- `return c` se déroule en $\mathcal{O}(1)$.

Conclusion : par somme, **la complexité temporelle dans le pire des cas** de la fonction `nbre_occurrences` est en $\mathcal{O}(n)$.

I.4 Le pire et le meilleur des cas.

Reprenons la fonction `nbre_occurrences`.

— Dans le meilleur des cas, `t[0] == x` : la boucle s'arrête après 1 passage. On dit que **la complexité temporelle dans le meilleur des cas** est en $\mathcal{O}(1)$.

— Dans le pire des cas, `x` n'est pas dans `t` : la boucle s'arrête après n passage(s). On dit que **la complexité temporelle dans le pire des cas** est en $\mathcal{O}(n)$.

Par défaut, on déterminera la complexité temporelle dans le pire des cas.

I.5 Classes de complexité asymptotique

Définition : La **complexité asymptotique** est le comportement de la complexité d'un algorithme lorsque la taille de son entrée est asymptotiquement grande.

Il peut s'agir a priori de complexité asymptotique moyenne ou dans le meilleur des cas ou dans le pire des cas. *En CPGE, lorsque l'on demande la « complexité » (ou le « coût ») d'une fonction, il s'agira toujours de « la complexité asymptotique dans le pire des cas » de cette fonction.*

Rappelons que la détermination de la complexité algorithmique ne permet pas d'en déduire le temps d'exécution mais seulement de comparer entre eux deux algorithmes résolvant le même problème. Cependant, il importe de prendre conscience des différences d'échelle considérables qui existent entre les ordres de grandeurs que l'on rencontre usuellement. Considérons un ordinateur personnel standard capable de réaliser 10^{11} opérations numériques par seconde.

Quel est le temps d'exécution $t(n)$ d'un algorithme avec $n = 10^4$ entrées pour plusieurs fonctions $f(n)$?

$f(n)$	$^4log\ n$	n	$n\ log\ n$	n^2	n^3	2^n
$t(n)$	0.13 ns	0.01 μs	0.1 μs	1 ms	10 s	10^{2992} années

Ce tableau est édifiant ! On comprend que les algorithmes ayant une complexité supérieure à une complexité quadratique soient en général considérés comme inutilisables en pratique (sauf pour de petites, voire très petites valeurs de n).

Définition : Soit n un entier naturel non nul. On dit qu'un algorithme de complexité temporelle $T(n)$ s'exécute dans le pire des cas :

- en temps **constant** quand $T(n) = \mathcal{O}(1)$
- en temps **logarithmique** quand $T(n) = \mathcal{O}(log\ n)$
- en temps **linéaire** quand $T(n) = \mathcal{O}(n)$
- en temps **quasi-linéaire** quand $T(n) = \mathcal{O}(n\ log\ n)$
- en temps **quadratique** quand $T(n) = \mathcal{O}(n^2)$
- en temps **cubique** quand $T(n) = \mathcal{O}(n^3)$
- en temps **polynomial** quand il existe $p \geq 2$ tel que $T(n) = \mathcal{O}(n^p)$
- en temps **exponentiel** quand il existe $a > 1$ tel que $T(n) = \mathcal{O}(a^n)$

4. sauf mention autre explicite, en informatique, *log* signifie toujours log_2 (logarithme en base 2). Il est définie, pour tout réel x par $log_2(x) = \frac{ln(x)}{ln(2)}$

II Exemples

II.1 Exemple de boucles imbriquées

```
10 def est_element(x:object,t:list):
11     '''renvoie True si x est dans t, tableau à 2 dimensions; False sinon'''
12     n, p = len(t), len(t[0])
13     for i in range(n):
14         for j in range(p):
15             if t[i][j] == x:
16                 return True
17     return False
```

Complexité : Le nombre n de lignes et p de colonnes du tableau `t` sont des mesures appropriées de la taille du problème considéré.

- Les affectations `n, p = len(t), len(t[0])` se déroulent en $\mathcal{O}(1)$ car `len(t)` et `len(t)[0]` se déroulent en $\mathcal{O}(1)$.
- Dans le pire des cas, pour chacun des np couples d'indices (i, j), on réalise en $\mathcal{O}(1)$ le test `if t[i] == x`. La complexité de la boucle imbriquée est donc en $\mathcal{O}(np)$.
- Alors, l'instruction `return False` est exécutée. Celle-ci se déroule en $\mathcal{O}(1)$.

Par somme, **la complexité temporelle dans le pire des cas** de la fonction `est_element` est donc en $\mathcal{O}(np)$.

II.2 Exemple d'appel à une fonction auxiliaire

```
20 def mystere(t:list, T:list)-> list:
21     L = []
22     for i in range(len(t)):
23         if nb_occurrences(t[i],T) >= 1:
24             L.append(t[i])
25     return L
```

Cette fonction renvoie la liste des éléments de `t` qui sont dans `T`, dans l'ordre dans lequel ils apparaissent dans `T`, et répétés autant de fois qu'ils sont dans `t`.

Complexité : **Il faut tenir compte de la complexité de la fonction `nb_occurrences`.** Posons `n = len(T)` et `p = len(t)`. Le nombre n d'éléments du tableau T et p d'éléments du tableau `t` sont des mesures appropriées de la taille du problème considéré.

— L'affectation `L = []`, se déroule en $\mathcal{O}(1)$.

— Dans tous les cas, l'instruction `if nb_occurrences(t[i],T) >= 1` est exécutée pour les p éléments de `t`. Dans le pire des cas, la fonction `nb_occurrences` s'exécute en $\mathcal{O}(n)$. Le pire des cas correspond à `nb_occurrences(t[i],T) == 0`. Finalement, la complexité temporelle liée à cette boucle est en $\mathcal{O}(np)$.

La complexité temporelle dans le pire des cas de la fonction `mystere`, somme d'une opération en $\mathcal{O}(1)$ et d'une boucle en $\mathcal{O}(np)$, est donc en $\mathcal{O}(np)$.

II.3 Recherche dichotomique dans un tableau trié

On se donne un tableau `t` de type `list` **trié dans l'ordre croissant** et un nombre `x`.
On veut profiter de l'ordre de `t` afin d'accélérer la recherche de `x` dans `t`.
Le principe de base est simple : on compare la valeur cherchée `x` avec celle au milieu du tableau ; si celle de `x` est plus petite, on cherche à gauche du milieu du tableau, sinon on cherche à sa droite ; ensuite, on recommence.

```
28 from math import inf
29
30 def recherche_dichotomique(x:object, t:list)->int:
31     '''renvoie la position de x dans le tableau t supposé trié et inf si x ne
       s'y trouve pas'''
32     assert t != [] , "t est un tableau vide !"
33     ind_g, ind_d = 0, len(t)-1 #indice de gauche et indice de droite
34
35     while ind_g <= ind_d:
36         # invariant : 0 <= ind_g et d <= len(t)-1
37         # invariant : x ne peut se trouver que dans t[ind_g..ind_d], ind_g et
       ind_d inclus
38         ind_m = (ind_g + ind_d)//2 # indice du milieu du tableau (à une unité
       près vers le bas)
39         if x > t[ind_m]:
40             ind_g = ind_m + 1 # x est strictement à droite du milieu
41         elif x < t[ind_m]:
42             ind_d = ind_m - 1 # x est strictement à gauche du milieu
43         else:
44             return ind_m
45     return inf # x n'est pas dans le tableau
```

Nous reviendrons plus tard dans l'année sur la notion d'**invariant de boucle**, mais on peut noter que si $x \in t[g..d]$, avec g et d inclus, et que l'on constate ensuite que x est, par exemple, strictement à droite du milieu m du tableau, on a forcément $x \in t[m+1..d]$. Par ailleurs, il est indispensable qu'à chaque boucle, la différence `ind_d-ind_g` décroisse strictement afin que la boucle **`while`** s'arrête.

Complexité : Soit p le nombre maximal d'itérations de la boucle **`while`**.
Comme toutes les instructions en dehors et dans la boucle sont en $\mathcal{O}(1)$, la fonction a donc une complexité temporelle dans le pire des cas en $\mathcal{O}(p)$.

Déterminons un majorant du nombre maximal d'itérations p en fonction de la longueur n du tableau t.

Considérons la variable `ind_d-ind_g`.

- au début : `ind_d-ind_g` $= n-1$
- après une itération : `ind_d-ind_g` $= \lfloor \frac{n-1}{2} \rfloor \leq \frac{n}{2}$
- après deux itérations : `ind_d-ind_g` $= \lfloor \frac{\lfloor \frac{n-1}{2} \rfloor}{2} \rfloor \leq \frac{n}{4}$
  ...
- après k itérations : `ind_d-ind_g` $\leq \frac{n}{2^k}$

Or, $\frac{n}{2^k} < 1 \iff n < 2^k \iff ln\ n < k \times ln\ 2 \iff k > \frac{ln\ n}{ln\ 2} \iff k > log\ n$ (en base 2).
Ainsi, dès que $k = \lceil log\ n \rceil$, on a forcément `ind_d == ind_g`. On itère une dernière fois et on sort de la boucle, donc $p = \lceil log\ n \rceil + 1$ convient. En particulier, pour $n \geq 2$, on a $p \leq 2 \log n$ càd. $p = \mathcal{O}(log\ n)$.

La complexité temporelle dans le pire des cas de la fonction `recherche_dichotomique`, somme d'opérations en $\mathcal{O}(1)$ et d'une boucle en $\mathcal{O}(log\ n)$, est donc en $\mathcal{O}(log\ n)$.

III Exercices

Q1 Donner la complexité temporelle dans le pire des cas de la fonction suivante :

```
48 def f1(n):
49     x = 0
50     for i in range(n):
51         for j in range(n):
52             x += 1
53     return x
```

Q2 Donner la complexité temporelle dans le pire des cas de la fonction suivante :

```
56 def f2(n):
57     x = 0
58     for i in range(n):
59         for j in range(i):
60             x += 1
61     return x
```

Q3 Donner la complexité temporelle dans le pire des cas de la fonction suivante :

```
64 def f3(n):
65     x, i = 0, n
66     while i > 1:
67         x += 1
68         i //= 2
69     return x
```

Q4 Donner la complexité temporelle dans le pire des cas de la fonction suivante :

```
72 def f4(n):
73     x, i = 0, n
74     while i > 1:
75         for j in range(n):
76             x += 1
77         i //= 2
78     return x
```

Q5 Donner la complexité temporelle dans le pire des cas de la fonction suivante :

```
81 def f5(n):
82     x, i = 0, n
83     while i > 1:
84         for j in range(i):
85             x += 1
86         i //= 2
87     return x
```

Q6 Donner la complexité temporelle dans le pire des cas de la fonction suivante :

```
90 def f6(n):
91     x = 0
92     for i in range(n):
93         j = 0
94         while j * j < i:
95             x += 1
96             j += 1
97     return x
```

Corrigé

Q1 Le nombre n est une mesure de la taille du problème considéré.

L'affectation `x = 0` est en $\mathcal{O}(1)$.

Dans le pire des cas, pour chacun des n^2 couples d'indices (i,j), on réalise en $\mathcal{O}(1)$ l'instruction `x += 1`. La complexité temporelle des boucles imbriquées est donc en $\mathcal{O}(n^2)$.

Par somme, **la complexité temporelle dans le pire des cas** de la fonction `f1` est en $\mathcal{O}(n^2)$.

Q2 Le nombre n est une mesure de la taille du problème considéré.

L'affectation `x = 0` est en $\mathcal{O}(1)$.

Dans le pire des cas, déterminons le nombre de couples d'indices (i,j) pour lesquels on réalise en $\mathcal{O}(1)$ l'instruction `x += 1` :

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

Comme $\frac{(n-1)n}{2} = \mathcal{O}(n^2)$, **la complexité temporelle dans le pire des cas** de la fonction `f2` est en $\mathcal{O}(n^2)$.

Q3 Le nombre n est une mesure de la taille du problème considéré.

Les affectations `x,i = 0,n` sont réalisées en $\mathcal{O}(1)$.

Pour chaque boucle sur i, on réalise en $\mathcal{O}(1)$ les instructions `x +=1; i //= 2`. Ainsi, si p est le nombre de boucles sur i, la complexité temporelle dans le pire des cas est en $\mathcal{O}(p)$.

Déterminons une majoration de p.
On sort de la boucle `while` quand $i \leq 1$ càd. $2^{p-1} < n \leq 2^p$, donc $log(n) \leq p < log(n)+1$ (en base 2). Ainsi, $p = \mathcal{O}(log\ n)$.
Finalement, **la complexité temporelle dans le pire des cas** de la fonction `f3` est en $\mathcal{O}(log\ n)$.

Q4 Le nombre n est une mesure de la taille du problème considéré.

Les affectations `x,i = 0,n` sont réalisées en $\mathcal{O}(1)$.

Pour chaque valeur de i, on réalise n boucles sur j.

Pour chacune de ces boucles, on réalise en $\mathcal{O}(1)$ l'instruction `x += 1`. Ainsi, la boucle sur j a une complexité temporelle en $\mathcal{O}(n)$.

Pour chaque boucle sur i, on réalise en $\mathcal{O}(1)$ l'instruction `i //= 2`. Ainsi, si p est le nombre de boucles sur i, la complexité temporelle dans le pire des cas des boucles imbriquées est en $\mathcal{O}(np)$.

On montre, comme à la question précédente, que $p = \mathcal{O}(log\ n)$.

Finalement, **la complexité temporelle dans le pire des cas** de la fonction `f4` est en $\mathcal{O}(n \log n)$.

Q5 Le nombre n est une mesure de la taille du problème considéré.

Les affectations `x,i = 0,n` sont réalisées en $\mathcal{O}(1)$.

Pour chaque valeur de $i \in [\![0,n]\!]$, on réalise i boucles sur j. Quand, pour une valeur de i donnée, toutes les boucles sur j sont exécutées, l'instruction en `i //= 2`, en $\mathcal{O}(1)$, divise i par 2, et donc le nombre de boucles sur j par 2. Ainsi,

— Pour $i == n$, on a n boucles sur j

— Pour $i == n//2$, on a $n//2$ boucles sur j, nombre majoré par $n/2$
— Pour $i == (n//2)//2$, on a $(n//2)//2$ boucles sur j, nombre majoré par $n/2^2$
...
— Pour $i == 1$, le nombre de boucles sur j est majoré par $n/2^p$ avec $p = log\ n$. A ce moment-là, on est sorti de la boucle sur i.

Par conséquent, le nombre total S de boucles sur (i, j) est majoré par

$$n + \frac{n}{2} + \frac{n}{2^2} + \cdots + \frac{n}{2^{p-1}} + \frac{n}{2^p}$$

Comme $\frac{n}{2^p} \leq 1$, on obtient $S \leq n(1 + \frac{n}{2} + \frac{n}{2^2} + \cdots + \frac{n}{2^{p-1}})$
càd. $S \leq n\frac{1-\frac{1}{2^p}}{1-\frac{1}{2}}$
càd. $S \leq 2n(1 - \frac{1}{n}) \leq 2n$
Comme l'instruction `x += 1` de la boucle en j est réalisée en $\mathcal{O}(1)$, la complexité temporelle dans le pire des cas des boucles imbriquées sur i et j est en $\mathcal{O}(n)$.

Finalement, **la complexité temporelle dans le pire des cas** de la fonction `f5` est en $\mathcal{O}(n)$.

Q6 Le nombre n est une mesure de la taille du problème considéré.

L'affectation `x = 0` est en $\mathcal{O}(1)$.

Dans le pire des cas, déterminons le nombre de couples d'indices (i, j), donc le nombre total de boucles, pour lesquels on réalise en $\mathcal{O}(1)$ les instructions `j = 0, x += 1` et `j += 1`.
On suppose $n \geq 2$. On a le tableau de valeurs suivant :

i	0	1	2	3	4	5	6	7	8	9	10	...	n-1
nbre. de j		0	1	1	1	2	2	2	2	2	3	...	$\lfloor\sqrt{n-2}\rfloor$

Notons n_k le nombre de fois qu'apparaît $k \in [\![1, \lfloor\sqrt{n-2}\rfloor]\!]$ dans la ligne du nombre de j (on lit dans le tableau $n_1 = 3$ et $n_2 = 5$).
Le nombre S total de boucles est donné par la somme du nombre d'indices j :

$$S = 1 \times n_1 + 2 \times n_2 + \cdots + \lfloor\sqrt{n-2}\rfloor \times n_{\lfloor\sqrt{n-2}\rfloor}$$

On peut majorer S en majorant les n_k.
Pour chaque $k \in [\![1, \lfloor\sqrt{n-2}\rfloor]\!]$, considérons i_k tel que $k^2 = i_k$ et i_{k+1} tel que $(k+1)^2 = i_{k+1}$.
Alors, $n_k = i_{k+1} - i_k = (k+1)^2 - k^2 = 2k = \sqrt{i_k} < \sqrt{n}$.

Ainsi, $S < (1 + 2 + \cdots + \lfloor\sqrt{n-2}\rfloor)\sqrt{n} < (1 + 2 + \cdots + \sqrt{n})\sqrt{n}$.
Donc, $S < \frac{\sqrt{n}(\sqrt{n}+1)}{2}\sqrt{n} < \frac{\sqrt{n}(\sqrt{n}+\sqrt{n})}{2}\sqrt{n}$. Donc, $S < n\sqrt{n}$.

Ainsi, **la complexité temporelle dans le pire des cas** de la fonction `f6` est en $\mathcal{O}(n\sqrt{n})$.

Remarque :
On peut aussi minorer S en minorant les n_k par 1 :
Alors, $S \geq 1 + 2 + ... + \lfloor\sqrt{n-2}\rfloor$.
On peut montrer par récurrence que : $\forall n \in \mathbb{N}, \quad 1 + 2 + ... + \sqrt{n} \geq \frac{n\sqrt{4n+5}}{3}$.
Or, $\frac{n\sqrt{4n+5}}{3} \geq \frac{2}{3}n\sqrt{n}$. Donc, $S \geq 1 + 2 + ... + \lfloor\sqrt{n-2}\rfloor \geq \frac{2}{3}\lfloor(n-2)\rfloor\lfloor\sqrt{n-2}\rfloor$
Ainsi, de façon asymptotique, S est minorée par une fonction de la forme $kn\sqrt{n}$ avec k réel. On note : $S = \Omega(n\sqrt{n})$.
Ceci signifie que, quelque soit l'entrée de taille n, le temps d'exécution pour cette entrée est au moins égal à un multiple constant de $n\sqrt{n}$.
Quand le temps d'exécution est asymptotiquement à la fois minoré et majoré par un multiple constant d'une fonction de n, ici $n\sqrt{n}$, on dit que **la complexité temporelle de la fonction `f6` est en** $\Theta(n\sqrt{n})$.