情報科学演習及び実験1（Ocamlの基礎2）

IS2 6316047 出納 光

平成29年6月1日

1　No.7

7-1．1回で1段か2段登ることが出来る人が、0段目からn段目までの階段を登る方法を全通り求めるプログラムを作成しなさい。
7-2．アルゴリズムについて考察しなさい。

7-1．添付したmlファイル上、関数stairsで確認できる。
7-2．本問は、フィボナッチ数列の各値と等しいこと予測できる。
まず、次のようなフィボナッチ数列を求める関数fibを条件分岐により定義する。
1，1，2，3，5，8，13，21 ...
しかし、上記のフィボナッチ数列は2番目以降だけが関係しており、
このままでは、階段の各段数から得られるパターン数とフィボナッチ数列の値が一致しない為、
階段の段数を引数nに取る関数stairsを定義する。
これは、階段の段数に+1した値をフィボナッチ数列の関数に入力したものである。
これにより、関数stairsに階段の段数を入力することで、求めたい値が得られる。

負の値が入力された場合を考慮して、例外処理を実装することでプログラム上のエラーを回避した。

また、同様の処理を、パターンマッチングでも実装できた為、
参考までにソースコードを添付する。
試しに、それぞれの処理完了までにかかる実行時間を計測してみた。

条件分岐
real 0m0.005s
user 0m0.002s

```
sys 0m0.001s
```

パターンマッチング

```
real 0m0.004s
user 0m0.002s
sys 0m0.001s
```

上記より、条件分岐の場合でもパターンマッチングの場合でも、実行時間にほとんど大差はないことが定量的に示された。

これ以外にも、コンビネーションを用いたアルゴリズムなども多々思いつき、様々な手法により、実装可能であることが分かった。

2 No.1

1-1．ソーティングアルゴリズムの種類について調査し、適切に引用して、まとめなさい．

1-1．ソーティングアルゴリズムの中でも、代表的な次の 4 つが挙げられる。

バブルソート
隣り合う要素の大小を比較しながら整列させること。
最悪計算時間が O(n2) と遅いが、アルゴリズムが単純で実装が容易なため、また並列処理との親和性が高いことから、しばしば用いられる。
安定な内部ソート。基本交換法、隣接交換法ともいう。（単に交換法と言う場合もある）

Wikipedia（https://ja.wikipedia.org/wiki/バブルソート）より、引用

選択ソート
配列された要素から、最大値やまたは最小値を探索し配列最後の要素と入れ替えをおこなうこと。
最悪計算時間が O(n2) と遅いが、アルゴリズムが単純で実装が容易なため、しばしば用いられる。
内部ソート。安定ソートではない。

Wikipedia（https://ja.wikipedia.org/wiki/選択ソート）より、引用

マージソート
既に整列してある複数個の列を 1 個の列にマージする際に、小さいものから先に新しい列に並べれば、新しい列も整列されている、
というボトムアップの分割統治法によるもの。
大きい列を多数の列に分割し、そのそれぞれをマージする作業は並列化できる。
n 個のデータを含む配列をソートする場合、最悪計算量 O(n log n) である。

分割と統合の実装にもよるが、一般に安定なソートを実装できる。
インプレースなソートも提案されているが、通常 O(n) の外部記憶を必要とする。

Wikipedia（https://ja.wikipedia.org/wiki/マージソート）より、引用

クイックソート

1960 年にアントニー・ホーアが開発したソートのアルゴリズム。分割統治法の一種。
最良計算量および平均計算量は O(n log n) である。
他のソート法と比べて、一般的に最も高速だといわれているが、
対象のデータの並びやデータの数によっては必ずしも速いわけではなく、
最悪の計算量は O(n^2) である。また数々の変種がある。
安定ソートではない。

Wikipedia（https://ja.wikipedia.org/wiki/クイックソート）より、引用

ソーティングアルゴリズムは、速さ精度などを考慮した上で、用途によっての使い分けが重要である。

3 No.3

3-1．台形の面積を求める関数を定義しなさい。
3-2．次の要件を満たす関数を定義しなさい。
・入力: 関数 f、実数 a,b（ただし、a<b とする。）
・出力: 範囲 a から b までの定積分
3-3．実装したプログラムの精度に関する考察を定量的に行いなさい．

3-1．台形の公式、「(上底 a + 下底 b) * 高さ h / 2」を用いた。
添付した ml ファイル上、関数 trpez で確認できる。
このとき、float 型として扱うことに留意した。
3-2．添付した ml ファイル上、関数 intel で確認できる。
できうる限り小さな帯域幅 h で与えられた関数の範囲を分割し、
それぞれの台形の面積の和を求める。
いわゆる、区分求積法と同様の考え方である。
台形の面積の導出には、3-1 を用い、
その上底と下底は、範囲と比べ、非常に小さい値であるため、ほぼ近似できるとした。
3-3．h の値を変化させることで、定量的な観察ができる。
これは、台形の上底と下底の値に h を使用しており、分割単位の基準にもなっているからである。
具体的には、h の値を 0 に近づけるほど、より精度の高い結果が得られる。
次の結果は、h の値がそれぞれ、0.01 と 0.001 の場合である。

精度
0.01

```
# intel ((fun a -> a*.a), -1.0, 1.0);;
- : float = 0.656700000000000395
```

0.001

```
# intel ((fun a -> a*.a), -1.0, 1.0);;
- : float = 0.665666999999999898
```

これらを比較しても、0.001 の方がより誤差の少ない値となっている。

4 No.2

2-1．次の要件を満たす関数を定義しなさい。
・入力: 関数 f、実数 x
・出力: x における f の導関数の値。
2-2．関数 f の極値を 1 つ求める関数 ext を定義しなさい。
2-3．関数 ext に関して、プログラムの実行速度と精度の 2 点について定量的に考察しなさい。

2-1．添付した ml ファイル上、関数 diff で確認できる。
これは導関数の定義に従って、実装した。
2-2．添付した ml ファイル上、関数 ext で確認できる。
このとき、極値を探す範囲を引数 a,b を用いて、任意で指定できるようにした。
正負の両方向に絶対値 0.01 の帯域幅に入る、導関数の値を求めることで極値の存在する x 座標を求めた。
その後、元の関数にそのときの座標を代入することで、求める極値の値を導出した。
2-3．この関数において、h は極値を探すための範囲を指定する役割を果たしており、
結果の精度、実行時間に大きく影響してくると予想できる。
したがって、h が 0.01 と 0.001 のときの実行速度と精度を計測した。
以下は、実行結果である。

実行速度
0.01
```
real 0m0.012s
user 0m0.002s
sys 0m0.002s
```
0.001
```
real 0m0.005s
user 0m0.002s
sys 0m0.001s
```

精度
0.01
```
# ext ((fun i -> i*.i*.i), -10.0, 10.0);;
- : float = -0.00021600000000001824081
```
0.001
```
# ext ((fun i -> i*.i*.i), -10.0, 10.0);;
```

```
- : float = -0.000195112000001035791
```

実行速度に関して、定性的に考えれば、
h の帯域は狭いほど、再帰の回数が増え、実行時間が長くなると予想される。
しかしながら、定量的な観察の上、
h の幅を狭くするほど、実行時間が短いという結果が得られた。
これは、そのときのマシンの状態に起因することが考えられる。

また、精度に関して、
h の値を 0 に近づけるほど、より精度の高い結果が得られる。
また、極値をとる条件である導関数の値を 0 に近づけるため、
その許容帯域を狭くするほど、より正確と言えるだろう。
今回は、扱う値の帯域やそれ自体を細かくしすぎると、
"Stack Overflow"となったり、（おそらく）進数の異なる値が出力されるなどの問題に悩まされたが、
調査の結果、使用する計算機がチューリング完全な場合でも、言語によって CPU やメモリの扱い方が異なるため、
すなわち、用いる言語の限界により、求めたい結果が出力できないなどの問題も十分に有り得ると予測している。

5 No.6

6-1．次の要件を満たす関数を定義しなさい．
・入力: 要素が char 型のリスト l,m
・出力: l の部分リストが m に一致するとき、その最初の文字の添え字。ただし、一致しない場合は-1 を返すこと。
6-2．実装したプログラムの実行時間に関する定量的な考察をしなさい。

6-1．添付した ml ファイル上、関数 search で確認できる。
パターンマッチングにより、リストを分割していき、
まず、探したいリストの先頭が、対象のリストに存在するかどうかを判別する。
その後、探したいリストの残りが、対象のリストの続きに存在するかどうかを判別する。
リストの分割と同時に、カウントしておくことで、その目的の添え字を結果として得られる。
もし、途中で一致しなくなった場合、探したいリストの先頭に再帰させることで、リストの全要素を検索できる。
対象に探したいリストが存在しなければ、-1 を返す。
6-2．以下は、実行結果である。

実行時間
```
real 0m0.004s
user 0m0.002s
sys 0m0.001s
```

他の問いと違い、ある一定の帯域幅を確保したり、できる限り小さな値を用いなければならないという関数ではないため、
実行時間をこれ以上に速くすることは容易ではない。
値に因らないということは、アルゴリズム自体を変更する必要があると考えられる。