情報科学実験レポート ocaml2

6321120
横溝尚也

提出日：６月　７日　（火）

ソースコードのファイル名

すべてをプログラムを一つにまとめたファイル：ocaml_report2_all

課題１：ocaml_report2_1.ml

課題２：ocaml_report2_2.ml

課題３：ocaml_report2_3.ml

課題４：ocaml_report2_4.ml

課題５：ocaml_report2_5.ml

課題６：ocaml_report2_6.ml

1　課題１

1.1　ソーティングアルゴリズムの調査

1.1.1　ソーティングアルゴリズムについて

ソーティングアルゴリズムとはデータの羅列を降順又は昇順に並び替え、整理する考え方である。ソーティングアルゴリズムは約１１種類存在する。バブルソート、選択ソート、挿入ソート、マージソート、クイックソート、ヒープソート、ビンソート、分布数え上げソート、基数ソート、シェルソート、ボゴソートである。データの羅列を昇順にすることは人が情報を認識しやすくさせ、自然な形になる。このデータの整理をプログラム上で実行するとき、いかに少ないメモリ量かつ短時間で整列させることができるか研究が続いている。上記１１種類のソーティングアルゴリズムにも、アルゴリズムのわかりやすさ、使用するメモリ量の少なさ、実行時間の短さなどそれぞれの利点がある。

ここでそれぞれのソートの大まかなアルゴリズムについて記述する。

1.1.2　アルゴリズムが簡潔なソーティング

１１種類の中でバブルソート、選択ソート、挿入ソートはアルゴリズムが簡単であることが利点であるアルゴリズムである。

この３つは課題 1-2 でプログラムを作成し、そこでアルゴリズムの考察を行っているため省略する。

1.1.3　短時間での整列ができるソーティング

ソートする時間がより速いアルゴリズムを目指したものがクイックソート、マージソート、ヒープソートである。これらの欠点はアルゴリズムがかなり煩雑なことである。

1.1.4　限定的なソーティング

クイックソートなどは実行時間がある程度早いが限界がある。それらよりも使うことができる条件が厳しかったり欠点は多いがクイックソートよりも早い実行時間を実現することができる。それがビンソート、分布数え上げソート、基数ソートである。

1.1.5　参考文献

・基本的なソートアルゴリズム 11 個まとめ — Shino's Mind Archive
https://www.bing.com/ck/a?!p=415b7901abc9d67dfe7bfc644b8fff1e1b70e1ed2056176676bbe4c9ee4202f5JmltdHM9MTY1NDUyNTA4NSZpZ3VpZD0zYjczMTgyYi0zMWZhLTQzMzctYjAwNy1hMjVmMjVkMzVhZmEmaW5zaWQ9NTIxNQptn=3fclid=7b5df63f-e5a3-11ec-9015-290254a8ff81u=a1aHR0cHM6Ly9zaGlub2FyY2hpdmUuY29tL2NvbnRlbnRzLzE5MTYvntb=1

・基本的なソートアルゴリズムまとめ＋α。
https://www.bing.com/ck/a?!p=bc8c93dddd39c8c4ac0783149a2e80ee2f9c038ea1685af50545329c72efe89fJmltdHM9MTY1NDUyNTA4NSZpZ3VpZD0zYjczMTgyYi0zMWZhLTQzMzctYjAwNy1hMjVmMjVkMzVhZmEmaW5zaWQ9NTMyNQptn=3fclid=7b60002b-e5a3-

11ec-a37e-de23f0673dbbu=a1aHR0cHM6Ly9xaWl0YS5jb2
0vaGlzby9pdGVtcy81YzM2ZjUwYzdkZTYxZmU4NzBhMgntb=1

・ソート - Wikipedia
https://www.bing.com/ck/a?!p=c0eddc623c2ec36def514316b94afc126
daa679332b0bec09511b1aab77186e6JmltdHM9MTY1NDUyNTA4NSZpZ3VpZD0zYjcz
MTgyYi0zMWZhLTQzMzctYjAwNy1hMjVmMjVkMzVhZmEmaW5zaWQ9NTM1Ngptn=3fclid=7b6029e9-
e5a3-11ec-82c1-7dae62287067u=a1aHR0cHM6Ly9qYS53aWtpcG
VkaWEub3JnL3dpa2kvJUUzJTgyJUJEJUUzJTgzJUJDJUUzJTgzJTg4ntb=1

1.2 アルゴリズムの実装

今回自分は挿入ソート、選択ソート、バブルソートの３つのプログラムを作成し、アルゴリズムの説明、実行時間の考察を行う。

1.2.1 挿入ソート：アルゴリズムの説明

1：初めに空リストであるソート済みの新たなリスト lst2 を作成する。

2：match 文を使用して未ソートリスト lst の先頭要素 a を取り出し a をソートする作業を行う。

3：再び mtch 文を使用し、lst2 の先頭要素 x と a を比較しながら、a をどの位置にソートするのかを決める。その際 x の値を変えて比較し続ける必要があるため再帰させる。

4：a をソートできたら、次の要素 a' をソートするために再帰させる

２つの再帰関数を使用する必要があるが順番としては a をソートするために１回目の再帰させ、その後、ソート後に他の要素をソートさせるために２回目の再帰させるので 3 の再帰が小回りで、4 の再帰が大回りにする必要がある。

1.2.2 挿入ソート：プログラムの説明

以下すべてのプログラムの説明では、別で提出してあるソースコードと同じものをこのレポートにも記載し説明の都合上、行数を振ってある。実行する際にはソースコードの方を使用してください。

```
1  let sort_of_insert lst =
2  let lst2 = [] in
3  let rec insert lst lst2 =
4    match lst with
5    | [] -> lst2
6    | a :: rest ->
7        let rec insert2 lst2 =
8         match lst2 with
9         |[] -> a :: lst2
```

```
10        | x :: rest2 ->
11             if a > x then x :: (insert2 rest2)
12             else a :: lst2
13        in insert rest (insert2 lst2)
14   in insert lst lst2
15 ;;

16  sort_of_insert [150;2;14;56;3;4];;
17  sort_of_insert [100;123;3;6;38;394;4441;5;0];;
```

プログラムの説明

2行目：ソート済みのリスト lst2 を作成する。

3から5行目：新たな関数 insert を作り、未ソートの lst が空リストになったとき、すべてがソートされたのでソート済みの lst2 を返す。

6から8行目：未ソート lst にまだ要素が残っている場合、lst の先頭要素 a をソート済み lst のどこに入るのか比較しながらソートしていく。再起関数 insert2 を作り、lst2 を引数とした match 文を使用する。

9から１２行目：ソートしたい要素 a がソート済みの要素 x と比較して a の方が小さい、またはソート済みの lst ２の要素すべてと比較し、その中で a が最大であるときは last2 にソートする。x より a の方が大きければ x を除いた lst2 と a を再び評価するため再帰させる。

１３行目：新たに a をソートした lst2 を insert の lst ２の新たな引数として使用し、再び未ソートの要素をソートするために再帰させる。

１４行目：最後にすべてをソートした lst2 を出力する。

1.2.3 選択ソート：アルゴリズムの説明

１：リストの中から一番小さな要素 a を見つける。

２：a を左から加える。加える先は、リストから a を除いた残りの要素の中で再び一番小さな要素 a' を見つける再帰関数とする。

３：再帰を繰り返すたびに未ソートのリストの要素数がひとつづつ減っていく。

４：すべてをソートし終わった時点でソートしたリストを出力する。

1.2.4 選択ソート：プログラムの説明

(*サブ関数：指定した要素をリストから引き抜く*)

```
1 let rec pull_out n lst =
2   match (n , lst) with
3   | (_ , []) -> []
4   | (n , a :: rest) ->
5      if n = a then (pull_out n rest)
6      else a :: (pull_out n rest)
7 ;;
```

```
(*メイン関数*)
8  let rec sort_of_select lst =
9    match lst with
10    | [] -> []
11    | [a] -> [a]
12    | a :: b :: rest ->
13      let lst2 = lst
14      in let rec pick_min lst =
15           match lst with
16           | [] -> []
17           | [a] -> a :: sort_of_select (pull_out a lst2)
18           | a :: b :: rest ->
19               if a >= b then pick_min (b :: rest)
20               else pick_min (a :: rest)
21      in pick_min lst
22 ;;

23 sort_of_select [150;2;14;56;3;4];;
24 sort_of_select [100;123;3;6;38;394;4441;5;0];;
```

プログラムの説明

1から6行目：まずある要素とリスト lst を引数としてとり、lst から引き抜いたリストを出力するサブ関数を作成する。

１０から１１行目：未ソートリスト lst が最後１個になったとき、それはリストの要素の中で最大の要素であるので一番右に加える。プログラム上 switch 文での分岐がすべての選択肢を満たしていなければ警告が出てしまう。そのために空リストの場合も用意しているが、調べたいリストが空リストでない限り使用しない。

１２から１４行目：未ソートのリストと同じリスト lst2 を作り、新たな関数を宣言する。

１５から２０行目：ここでは最小の要素を見つける作業である。１７行目は最小の要素 a が見つかったときその要素を左から加え、要素 a 世除いたリストで２番目に小さな要素を探すために再帰させる作業をしている。

２１行目：最後にソート済み lst を返す。

1.2.5 バブルソート：アルゴリズムの説明

１：一番左の要素と左から２番目のよその大小を比較する。小さいものが左、大きいものが右に来るように順番を入れ替える。

２：左から２番目と左から３番目を比較し、１と同じような処理をする。

３：１と２の作業を一番右の要素まで行うと、リストの中で最大の要素のソートが完了する。

４：再び１と２を行うと今度はリストの中で２番目に大きな要素のソートが完了する。

５：この操作をくり返し続けると１と２の操作をしてもリストの順番が変化しなくなる。これはソートの完了を意味する。

1.2.6 バブルソート：プログラムの説明

```
1 let rec sort_of_bubble lst=
2   match lst with
3     [] -> []
4    | lst ->
5      let lst2 = lst
6      in let rec bubble lst2 =
7           match lst2 with
8             [] -> []
9            | [a] -> [a]
10           | a :: b :: rest ->
11               if a > b then b :: bubble (a :: rest)
12               else a :: bubble (b :: rest)
13     in let lst3 = (bubble lst2)
14     in if lst3 = lst2 then lst3
15     else sort_of_bubble lst3
16 ;;

17 sort_of_bubble [150;2;14;56;3;4];;
18 sort_of_bubble [100;123;3;6;38;394;4441;5;0];;
```

プログラムの説明

１から６行目：３行目ではリストが空になったときプログラムを終了させるために作り、から出ないときをメインにプログラムする。新たに lst2 を宣言し、再帰関数を新たに作成する。

８から９行目：要素が一つになったとき、最大の要素を右端に移動させる作業が終了させる。８行目はソートしたいリストが空リストでない限り使用することはないが、match 文を使用していると８行目を記述しないと警告が出てしまうので念のため記述した。

１０から１２行目：a が b より大きければ a と b の順番を入れ替え、そうでないときは何もせずに先頭以外の要素で再帰させる。再帰させるごとに lst2 の要素数がひとつづつ減り続け、最後の一つになったときに再帰を終了させる。

１３から１５行目：再帰が終わったら、lst3 に先ほどの関数で得られた値を格納し、もし lst3=lst2 が成り立つ、つまり６～１２行目でリストに何も変化が施されていないときはソートが完了していることを意味するのでプログラムを終了させる。もし lst3=lst2 が成り立たなければまだソートが完了していないので再び１行目から再帰させる

1.3　プログラム実行時間の考察

1.3.1　ソートアルゴリズムの計算量について

ソーティングアルゴリズムについて調べた際に使用した文献をもとに考察を行う。様々なソートの計算量について調べると、以下の通りとなった。

ソートの種類	計算量
挿入ソート	n^2
バブルソート	n^2
選択ソート	n^2
クイックソート	nlogn (n^2)
マージソート	nlogn

表 1　それぞれのソートによる計算量

自分が実装した挿入ソート、バブルソート、選択ソートはソートしたい要素数が n 個であるとき、計算量はすべて n^2である。よって時間的効率はこの 3 つのアルゴリズムすべて同じである。

マージソートの計算量は常に nlogn である。一方でクイックソートに関しては平均計算量は nlogn であるが、ソートしたい対象によっては n^2回計算する必要が出てくる。

ここで n^2と $nlogn$ を比較する。計算サイトで二つの関数のグラフを描いたのが図 1 である。図 1 からわかる通り、すべての自然数において n^2の方が $nlogn$ よりも大きい。このことから、挿入ソート、バブルソート、選択ソートよりもクイックソート、マージソートの方が計算量が少なく、効率的であることがわかる。1.1.3 アルゴリズムの調査でも記述したとおり、クイックソートマージソートは時間的効率がよいことにも一致する。

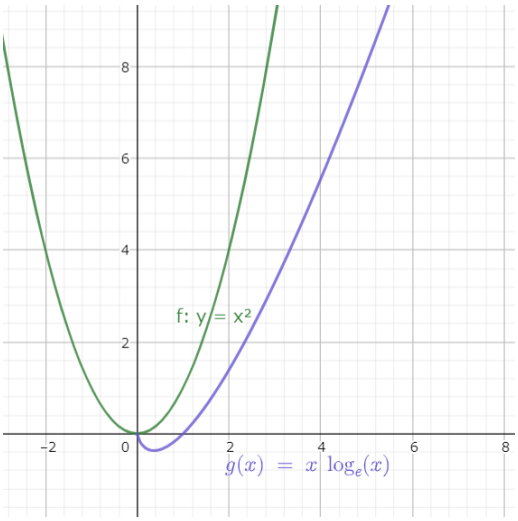

図 1　グラフ

1.3.2　それぞれのソートの実行時間について

表２はソートする実行時間を調べまとめたものである。要素数が少ない１０で実行するとアルゴリズムによる実行時間の差はほとんど見られず、コンピュータの処理速度が速すぎて誤差といえる。また、要素数を３００に増やしたとしても全体的に 0.002 秒ほど増えたりしたが、ほとんど変わらなかった。

これらの調査からわかることは要素数をもっと増やし実行時間の差が如実に現れるくらいの要素数で行うべきであった。自分は３００個の要素を地道に打ち込んだが、乱数などを使用して１万個ほどの要素数を持つリストを作成するプログラムを書けば実現できそうだと考えた。

また、今回の調査ではクイックソートやマージソートのプログラムの実行を行わなかったため実行時間についても調べることができなかった。しかし、先ほどの考察で調査した３つのソートよりも計算量が少ないことが分かったいるため、実行時間が短くなるのではないかと考える。

挿入ソート、選択ソート、バブルソートの計算量は n^2で等しいことがわかっているため要素を比較する回数は等しい。しかし、要素を交換する回数はそれぞれのアルゴリズムが異なるため違うはずである。例えば、バブルソートと選択ソートでは要素を交換する回数は選択ソートの方が少ないため、要素数を大きくした時この３つのソートでも実行時間に差が出てくるのではないかと考えた。

ソートの種類	ソートする要素数	実行時間
挿入ソート	１０	平均 0m0.003s
選択ソート	１０	平均 0m0.002s
バブルソート	１０	平均 0m0.003s
挿入ソート	約３００	平均 0m0.005s
選択ソート	約３００	平均 0m0.003s
バブルソート	約３００	平均 0m0.006s

表２　それぞれのソートの実行時間

2　課題２

2.1　プログラム作成

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` の導関数で求められる微分係数

2.1.1　プログラムの説明

```
1 let differential x f =
2 ( f (x +. 0.00000001) -. f x ) /. 0.00000001
3;;
4
5let f x = x *. x
6in differential 3. f
7;;
```

プログラムの説明

１～３行目：引数として関数 f と実数 x を受け取る。ここで微分の定義について振り返ると、

$$f'(x) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h} \tag{1}$$

である。プログラムでは何かの値に極限まで近づけるという作業をするには様々な方法があるが、今回は h の値を自分で微小な数として設定した。今回は 0.0000001 とした。
６～８行目：関数 f を定義しつつ作成したプログラムの実行を行った。

2.2　プログラム作成２

関数 `f` の極値を `1` つ求める関数 `ext` を定義しなさい．
極値は `f(x)` ですが，`x` を求める関数でも正解とします）

2.2.1　プログラムの説明

```
1  let rec ext x y f =  (*x,y はそれぞれ極値を求める上限、下限の設定*)
2  if x > y then failwith "Error"
3  else if (differential (x -. 0.01) f) *.(differential (x +. 0.01) f) < 0. then
    x
4  else ext (x +. 0.1) y f
5  ;;
```

プログラムの説明

1行目：引数に極値となる x を調べる上限と下限、関数 f をとる。

2行目：上限 y を x が超えてしまった場合、範囲 x,y には極値を満たす x が存在しなかったことからエラーを出力している。

3行目：極値となる x が見つかった場合のプログラムである。調べる x より微小量小さい値と、微小量大きい値をそれぞれ微分した結果の積が負になったときが条件である。この時は x を出力する。

4行目：それ以外の場合、つまり調査を続ける場合は、x を微小量増やし元の関数 ext を再帰させている。

2.3 考察

2.3.1 アルゴリズムについての考察（おまけ）

理想ではすべての実数 x に対して f'(x)=0 を満たすかどうか調べたいが、プログラム上それは不可能に近い。今回のプログラムでは、極値となる x を調べる範囲を設定する必要があった。もちろん、調べる範囲を広げれば広げるほど調査する回数が増え、実行時間が増えた。しかし、今回はプログラム4行目よりずらす値に 0.1 を採用しているため、実行時間はあまり増えなかった。。0.1 の時実行時間は 0.003 秒、一方で、0.00001 にした時、実行時間は 1.248 秒とかなり実行時間が増えた。これより、調べる範囲を増やしても実行時間は増えるが、それよりも極値を探すときに x をずらしていく値が細かい方が試行回数が増えているため実行時間が増えていることが分かった。

初めに思い付いたプログラムがプログラム3行目の部分が

```
3 else if differentioal x = 0 then x
```

という微分し、x に値を代入したとき 0 にあるのかどうかで極値を判断するというアルゴリズムであった。しかしこれには二つの大きな欠点がある。

一つ目は f'(x)=0 となれば必ずしも極値を持つとは限らないという事である。例えば $f(x)=x^3$で考えると、$x = 0$ の時 $f'(0) = 0$ を満たすが極値にはならない。よって極値ではない誤った値が出力される可能性がある。

二つ目は、プログラム上では f'(x) がぴったり0にならない限り極値として認識されないという事である。今回のプログラムでは x の値を 0.1 づつずらしていき、極値となるものが存在するのか調べている。もし仮に x=10 が極値となる x だとしても、コンピュータが勝手に四捨五入して f'(10) が限りなく0に近い0でない数になってしまった場合、極値であるのにプログラムではスルーされ、エラーが出てしまうのである。

これら二つの問題点を解決できたのが今回のプログラムであり、極値はその前後において f'(x) の符号が入れ替わるという性質を採用したアルゴリズムである。よって極値を持つ x の前後では f'(x) 符号が入れ替わる性質を用いるのが最適であると考えた。

2.3.2 プログラムについての考察（おまけ）

このレポートを書いている途中でこのプログラムの欠点が発覚したので考察する。n 次関数の極値の数は最大で n-1 個ある。しかしこのプログラムではある極値が発見したら、再帰を終了し極値を出力するプログラムになっている。複数個極値が存在したとしてもひとつの極値しか出力されないことがこのプログラムの欠点である。

極値を発見した際の処理として、疎の極値を出力すると同時に再起を続ければこの欠点は解決すると考える。（正直プログラムを修正し、提出しなおす気力と時間がありませんでした。）

2.3.3 プログラムの実行速度と精度の考察

今回実行速度と精度に影響する変数は、プログラム１行目の引数である調査範囲 x と、プログラム４行目の x の値を変えて再帰させる際の x に足す微小量（以下ここでは y とする。)、プログラム４行目の調べる x に微小量足した値と引いた値の積が負になるか判定する際の微小量（以下ここでは z とする）の３つであると考える。今回は出力されるべき理想の x の値を 0 として関数を設定しているので出力された x の値そのものが誤差として現れるようにしてある。また、今回の誤差とは極値の誤差ではなく極値となる x の値における誤差として考えた。

x,y,z の値を変えたときの実行速度と、実際に出力される値において本来の極値とどれだけ差があるのかをまとめた表が以下である。

番号	x(調査範囲)	y(次の x へと増やす間隔)	z(極値か判断する際の前後の間隔)	実行時間	x の誤差
1	20	0.1	0.01	0m0.002s	-1.87905246917807744e-14
2	40	0.1	0.01	0m0.002s	1.52933221642115313e-14
3	20	0.001	0.01	0m0.006s	-0.0100000000001025935
4	20	0.0000001	0.01	0m12.397s	-0.0100000038768045113
5	20	0.1	0.001	0m0.002s	-1.87905246917807744e-14
6	20	0.1	0.00001	0m0.004s	-1.87905246917807744e-14
7	20	0.0000001	0.0000001	平均 0m12.470s	-1.03876809637232964e-07
8	40	0.0000001	0.0000001	平均 0m24.917s	-9.35972478269591236e-08
9	20	0.1	0.000000001	0m0.001s	極値として認識されない

表 3　実行時間と精度における考察

表から読み取れることを考察していく。

・１と２、７と８を比較する。変化させた部分は調査範囲であり、誤差にはほぼ変わりはない。しかし実行時間が約２倍になっている。このことから調査範囲と実行時間は比例関係にあると考える。これは再帰させる回数が調査範囲に依存していると考えられる。

・１と３と４を比較する。変化させた値は y であり、y を変えることで再帰回数が増えるので実行時間は増えている。また誤差については y を細かくすればするほど大きくなっている。自分の感覚では細かく調べれば調べるほど時間はかかるが、誤差が小さくなるのがふつうであると感じた。しかし誤差は大きくなっている。この現象が起こる理由は ocaml でおこなうと細かすぎたときコンピュータが勝手に切り捨てを行い、誤差が大きくなってしまっていると考えた。このことから細かすぎればよいというわけでもないと考える。

・５と６を比較する。変化させた値は z であり、z を変えることで極値かどうか判断する条件がきつくなっている。実行時間と誤差にほぼ変わりがない。今回実行時間の誤差は z を変えたことによる変化ではなく、全く同じ値で実行時間を調べた際にも若干異なっているため、ネット環境であったり、コンピュータの性能による誤差であると考えた。

・最後に９番の考察である。本来極値となる x=0 が存在するはずであるが見つからずエラーが出力された。これは y と z の関係に問題がある。試行する回数を少なくして、かつ、極値か判定する厳しさ z をきつくした

とき、値によっては認識されないことが分かった。今回の関数では、９番以外は認識されたが y と z を同じ値にする必要があることが分かった。

3　課題３

3.1　プログラムの作成

台形の面積を求める関数を定義しなさい.

3.1.1　プログラムの説明

```
1 let rec trapezoid a b high =
2  (a +. b) *. high /. 2.
3 ;;
```

台形の面積を求める公式を使用し、引数に上底、下底、高さの３つを引数として設定した。簡単なプログラムであるため説明は省略する。

3.2　プログラムの作成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： $\int_b^a f(x)dx$

3.2.1　プログラムの説明

```
1 let rec integral a b f =
2 if a < b then
3    (f a *. 0.01) +. integral (a +. 0.01) b f
4 else 0.
5 ;;

6 let f x = x *. x
7 in integral 1. 3. f;;
```

プログラムの説明

１行目：引数を積分範囲 a,b と任意の関数 f を引数に設定する。

２～３行目：積分範囲内に該当したとき、設定した幅（今回は 0.01）と f(a) の積で得られる面積を設定した幅の文だけ進めた再起関数に足し合わせる。

４行目：a を少しずつ進めていき積分上限 b を超えたとき、面積の足し合わせを終了させるために０とする。

６，７行目：関数 f と積分範囲を代入して実行している。

3.3 考察

3.3.1 3.1 と 3.2 における精度の考察

今回のアルゴリズムでの積分は微小幅の棒の面積を積分範囲すべて足し合わせて求めている。これは区分求積法といい、これをアルゴリズムとして採用している。本来なら幅を極限まで０に近づけるがプログラム上ではそれを表すのは不可能である。もちろんこのプログラムでは正確な値は出ず、誤差が生まれる。誤差に依存していると考えられる積分範囲と微小幅の設定を変えながら誤差を調べる。

なお、今回使用した関数は $f(x)=x^2$である。

番号	積分上限	積分下限	微小幅	理想の値	出力された値	誤差
1	0	3	0.1	9	8.55500000000000149	0.44499999999999851
2	0	30	0.1	9000	8955.05000000003383	44.94999999996617
3	0	300	0.1	9000000	9004500.49999948591	4500.49999948591
4	0	3	0.01	9	9.04504999999995185	0.04504999999995185
5	0	3	0.001	9	9.00450049999897395	0.00450049999897395
6	0	3	0.0001	9	8.99955000500411728	0.00044999499588272
7	0	3	0.00001	9	測定不可能	測定不可能

表 4　使用したアルゴリズムにおける精度の考察

上の表を用いて考察を行う。

・１と２と３を比較する。微小幅は変えずに 0.1 とし、積分上限を 3,30,300 と変化させている。この時誤差は明らかに増えている。これは微小幅で作られる一つの長方形の棒に少しづつ誤差が存在し、その棒の数が増えていくために誤差が増えていると考えられる。また、今回採用した $f(x)=x^2$では x が０付近よりも３０，３００と増えていくにつれて変化量（傾き）が増えていくため、長方形の棒の誤差は増えていく。よって誤差が増える要因は長方形の棒の数が増えるほかに、一つの棒における誤差の増加である。

・４から６の比較をする。積分範囲を変えずに０から３とし、微小幅のみを変化させた。プログラムから、微小量を細かくすればするほど再帰回数は増えていくのは自明である。また、一つの棒における誤差は棒の幅を狭くすればするほど小さくなるため、誤差も減っていることがわかる。$y=x^2$, 微小幅 0.0001, 積分範囲 0 からのとき誤差は 10^{-4}まで近似できている。

・最後に７の考察である。微小幅を 0.0001 としたとき今回は０から３まで積分するため、再帰させる回数は 300000 回である。このときコンピュータでは扱いきれず”スタックオーバーフロウ”というエラーが発生する。$y=x^2$のとき微小幅 10^{-4}でエラーが出るが、$y = x^3$のときは微小幅 10^{-3}でエラーが出る。なお、６番の結果は $teraterm$ 上で出力された値であるが、これを $tryocaml$ や $cygwin64terminal$ で実行するとエラーが出力され、実行するソフトによっても扱いきれる量が変わってくることがわかる。

3.3.2 ほかの積分法を用いたアルゴリズムの考察

区分求積法以外の積分方法でプログラムできないか考える。例えばニュートン・コーツ求積法、ロンバーグ法がある。ニュートン・コーツ旧積法とは求めたい面積を台形の足し合わせとして考え近似する求め方である。区分求積法での微小幅を台形の高さ、上底と下底をそれぞれ f(a),f(a+h) とし、面積を求める。

$$S = \sum_{k=0}^{n-1} \frac{h}{2}\{f(k) + f(k+h)\} \tag{2}$$

上記の考え方を利用するとこのように積分値 S をかける。このアルゴリズムを使用してプログラムすることもできると考えた。

参考文献

・ニュートン・コーツの公式 - Wikipedia
https://ja.wikipedia.org/wiki/

4　課題４

4.1　プログラムの作成

任意の正の整数 n が与えられたとき，n が偶数ならば、n を 2 で割り，n が奇数ならば、n に 3 をかけて 1 を足す，という操作を n が 1 になるまで繰り返すプログラムを作成しなさい.

4.1.1　プログラムの説明

```
1  let rec recursion n =
2  if n mod 2 = 0 then recursion (n / 2)
3  else if n = 1 then 1
4  else recursion ((n * 3) + 1)
5  ;;

6  recursion 6;;
7  recursion 7;;
```

プログラムの説明

2行目：偶数ならば2で割る作業である。if 文の条件式を mod 2 = 0 として再帰させている。

3行目：奇数ならば3をかけて1を足す作業である。if 文の条件式を mod 2 = 1 として再帰させている。

4.2　ステップ数についての考察

考察を行うためにステップ数を出力するプログラムも作成した。それが以下である。
（ソースコードにも添付してあります。）

```
1  let rec recursion n =
2  if n mod 2 = 0 then recursion (n / 2) + 1
3  else if n = 1 then 0
4  else recursion ((n * 3) + 1) + 1
5  ;;

6  recursion 6;;
7  recursion 7;;
```

4.2.1　ステップ数について

調べたい自然数	ステップ数
2	1
6	8
8	3
10	6
15	17
26	10
27	111
28	18

表 5　ステップ数

問題の条件で操作し続け、１に収束しない自然数は自分で試行する限り存在しなかった。表３はある自然数のステップ数についてまとめたものである。偶数ならば２で割ることから n が 2^n になったとき一気に１まで収束し、n が 2^n でないときはいずれ奇数が出現する。一方で奇数の時は３倍して１を足すので操作後必ず偶数になる。この繰り返しを行えば、感覚的にいつかは 2^n となり、1 に収束するはずである。１から順にステップ数を調べたところ２７のとき異常な外れ値 111 をとった。

4.2.2　コラッツ予想について

コラッツ予想とは数論の未解決問題のひとつである。n が偶数の場合、n を 2 で割る n が奇数の場合、n に 3 をかけて 1 を足す。どんな初期値から始めても、この操作を有限回行えば必ず 1 に到達するという予想である。いまだ解決されていないことから予想とされている。

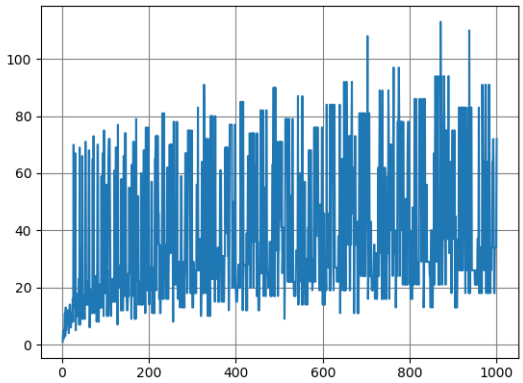

図 2　ステップ数を表したグラフ

図２は０から１０００までの自然数とステップ数との関係性を表したものである。このグラフから、基本的

には自然数が大きくなればステップ数も増える傾向にある。その中でたまに突出してステップ数の大きな自然数が現れ、外れ値をとっている。例えば表 3 の中で 20 代の自然数は 10 前後のステップ数であるが、27 のステップ数は 111 となっていることがわかる。これもグラフから読み取れる外れ値の一つであると考えられる。

自分のプログラムでステップ数を調べていっても規則性は見つからなかった。コラッツ予想について調べたところステップ数の規則性すら証明されていないことが分かった。

参考文献

・コラッツの問題 - Wikipedia
https://www.bing.com/ck/a?!p=4db98778d6f11a58469bdb82931576e
26d7bc4802a96ff54a100192649722cccJmltdHM9MTY1NDQ2MzIyNiZpZ3VpZD0yO
TkyMTYzNS1jZWYyLTQxZGQtOTc0YS0yY2M1ZjEyOTczZWImaW5zaWQ9NTE5Mwptn=3fc
lid=7420f3b1-e513-11ec-b11e-bba6c85a9129u=a1aHR0cHM6Ly9qYS53aWtpcGVk
aWEub3JnL3dpa2kvJUUzJTgyJUIzJUUzJTgzJUE5JUUzJTgzJTgzJUUzJTgzJT
g0JUUzJTgxJUFFJUU1JTk1JThGJUU5JUExJThDntb=1

画像引用

・コラッツ予想｜【コード付き】Python でプログラミング作成してみた!!
https://picolinateu.hatenablog.com/entry/2019/03/27/034941

5　課題５

5.1　プログラムの作成

2 次元座標における点（x，y）を任意個与えられたとき，これらをすべての点を結ぶ閉路のうちで最長になる点の順序を求めるプログラムを作成しなさい．

例：(x,y) = (5,5), (15,10), (10,33), (13,4), (1,12)

最長並び順：(5,5), (10,33), (13,4), (1,12), (15,10)

距離（参考）= 97.34236...

(*サブ関数１：２点間の距離を計算する関数*)

```
1 let calculate_length (x1 ,y1) (x2 , y2) =
2   sqrt ((x1 -. x2) *. (x1 -. x2) +. (y1 -. y2) *. (y1 -. y2))
3 ;;
```

(*サブ関数２：基準点から最も遠い点の出力*)

```
4 let rec max (x1 ,y1) lst =
5   match lst with
6     [] -> failwith "Error"
7    | (x2 , y2) :: (x3 , y3):: rest ->
8      if calculate_length(x1 , y1) (x3 , y3) > calculate_length (x1 , y1) (x2 , y2)
9         then max (x1, y1) ((x3, y3)::rest)
10     else max (x1 , y1) ((x2 , y2) :: rest)
11   | (x4, y4) :: rest ->  (x4, y4)
12;;
```

(*サブ関数３：リストから n 番目の要素の出力*)

```
13 let rec posl n lst =
14   match (n , lst) with
15     (1 ,a :: rest) -> a
16   | (n' , a :: rest) when n' > 0 ->( posl (n-1) rest ) ;;
```

(*サブ関数４：リストから指定した要素の出力*)

```
17 let rec pull_out n lst =
18   match (n , lst) with
19    | (_ , []) -> []
20    | (n , a :: rest) ->
21       if n = a then (pull_out n rest)
22       else a :: (pull_out n rest)
23 ;;
```

```
(*メイン関数１：最長並び順を出力*)
24 let shuffle lst =
25  let rec shuffle2 lst =
26   match lst with
27    [] -> []
28   |[(x1 , y1)] -> []
29   |(x1 , y1) :: rest ->
30    (max (x1 , y1) rest) :: shuffle2 ( max (x1 , y1) rest :: (pull_out (max (x1 , y1) rest) rest))
31   in posl 1 lst :: shuffle2 lst
32 ;;

33shuffle [(5.,5.); (15.,10.); (10.,33.); (13.,4.); (1.,12.)];;

(*メイン関数２：最長並び順の距離を出力*)
34 let  max_distance lst =
35   let newlst = (shuffle lst)
36   in let rec max_distance2 newlst  =
37       match newlst with
38         [] -> 0.
39       | [(x1 , y1)] -> calculate_length (posl 1 lst) (x1 , y1)
40       | [(x1 , y1); (x2 , y2)] ->
41           calculate_length (x1 , y1) (x2 , y2) +. max_distance2 [(x2 , y2)]
42       | (x1 , y1) :: (x2 , y2) :: rest ->
43           (calculate_length (x1 , y1) (x2 , y2)) +. max_distance2 ((x2 , y2) :: rest)
44   in max_distance2 newlst
45 ;;

46 max_distance [(5.,5.); (15.,10.); (10.,33.); (13.,4.); (1.,12.)];;
```

プログラムの説明

１～３行目：サブ関数として２点間の距離を計算する関数 calculate_length を用意する。授業でも扱ったことのある内容なので説明は省略する。

４～１２行目：サブ関数として引数を基準点 (x1,y1) とリストとし、基準点から一番距離が遠い点をリストの中から探し、その点を出力する関数 max を用意する。関数 calculate_length を使用して (x1,y1) とリストの中の全要素の２点間の距離を計算して大小比較を行っている。８や１０行目で２点間の距離の大小比較を行う際に複数回比較する必要があるので、再帰関数とした。１１行目では最後にリストの中に残った要素 (x4,y4) が最遠点であるのでこの組を出力している。

１３～１６行目：サブ関数として引数を自然数 n とリストにし、リストの n 番目の要素を出力する関数 posl を用意する。。これは以前の課題内容でもあったので説明は省略する。

１７行目～２３行目：サブ関数として指定した要素をリストから取り除き、取り除いたリストを出力する関数 pull_out を用意する。２１から２２行目で指定した要素に一致すればそれを出力、一致しなければ調べていない要素を調べるために再帰させている。

２４行目～３２行目：すべての点を結ぶ閉路のうちで最長になる点の順序を求めるプログラムのメイン関数である。今回、任意個の点の入力をリストとして引数にしている。２５から２６行目で新たな再帰関数を用意し、match 文を使用する。２８行目はリストを並べ終わったとき match 文を終了させるプログラムである。２９行目から３０行目では、リストの先頭要素を基準点 (x1,x2) とし、サブ関数 max を使用し最遠点 max (x1,x2 rest) を基準点の次の要素としてリストに加えつつ、基準点を max (x1,x2) に変更して再帰させる。再帰させるとき、引数の lst は rest から新たな基準点を除いたリストに左から新たな基準点を加えたものとしている。よってこの時サブ関数 pull_out を使用している。３１行目では一番初めの点を加える必要があったのでサブ関数 posl を使用して最長並び順にしたリストに加えている。

３４～４５行目：点の順序を求めた後に距離を出力するメイン関数である。３４行目の引数リストは未整列のリストでもいいように３５，３６行目で新たな newlst を整列済みのリストにしてこの関数の中で整列も行えるようにした。３７行目以降が距離の計算である。４２，４３行目が隣同士の要素の２点間の距離を出力し、再帰させている。４０，４１行目は再帰させていった結果、最後の要素の距離計算をするときの場合である。３９行目では、閉路の距離を出力するためにリストの最初と最後の要素の距離計算をしている。この時点でリストには最後の要素しか残っていないため、未整列の lst を使用し、lst の最初の要素を引き抜くためにサブ関数 posl を使用した。最後に４４行目で足し合わせた距離の出力を行っている。

5.2 アルゴリズムについて考察

今回採用したアルゴリズムは、基準点から最遠点へと結び、また更にその点から最遠点を結んでいっ手出来る閉路がすべての点を結ぶ閉路のうちで最長になる点の順序になるという考え方である。感覚的にはこの性質が成り立つ気もするが調べてみてもこれが成り立つかどうか確認できず、自分で証明することもできなかった自分が採用したアルゴリズムが正しかったとすると、基準点から最遠点を出力するプログラムをサブ関数として用意すればメイン関数を再帰させればうまくプログラムできそうだと考えた。

自分のプログラムでは入力するとき組を要素としたリストにし、出力する際もリストとした。本来ならば組を順に出力すべきであるが、ocaml のリストの性質として要素の方が同じであれば組も要素としてとることができる。また、リストで入力してしまえば組の順序を変えたり、大小を比較することが格段と容易になる。これらの利点からリストを使用することが再提起であると考えた。

次に思いついたアルゴリズムは、n 個の点で作り得る閉路の距離をすべて計算し最大値を出力するものである。プログラムする上ではこちらのアルゴリズムの方が簡単に書けるかもしれない。しかしプログラムの計算量がかなり多く、効率が悪い。例えば、n=5 の時たった 4 点の最長経路を調べるだけでも 4!=24 通りの経路の距離計算をしなければならない。また、5 点の閉路を作るとき 5 本の線が必要となり、一つの経路ごとに 5 回 2 点間の距離計算をしなければならないため合計 $24\times5 = 120$ 回もの距離計算を行っている。これはとても非効率である。よって今回は最初のプログラムを採用した。

5.2.1 オイラー路について

今回の問題のように任意個の点を一筆書きで表し、グラフの全ての辺を通る路のこと。また全ての辺をちょうど 1 度だけ通る閉路は、オイラー閉路という。ただし、今回は最長経路となる点の並びを考えるため、同じ

点に戻ることは考えておらず少し異なっている。この問題もオイラー路に似たプログラムの課題ではないかと考えた。

（参考文献：オイラーグラフの定理（一筆書きできる条件）とその証明）
https://www.bing.com/ck/a?!p=81074621a3c8ef628bb8ba404ab265466069
3bd8a6a5d3fe81ed1c1ee1b03537JmltdHM9MTY1NDU1MjUwMyZpZ3VpZD0yN2Uw
NjcwYS1jNWMyLTQ0MjktODNhYi0xNDZmZmU3ZjU5M2ImaW5zaWQ9NTE4Nwptn=3fcli
d=51b75407-e5e3-11ec-9bf2-2dbe47ff3a6du=a1aHR0cHM6Ly9tYW5hYml0aW1lcy
5qcC9tYXRoLzY0Mgntb=1

6　課題６

6.1　プログラムの作成

階段を 1 回で 1 段または 2 段上ることができる人が，0 段目から n 段目までの階段を上るとき，上る方法が何通りあるか求めるプログラムを作成しなさい．
例：4 段のときは 5 通り
例：7 段のときは 21 通り

6.2　プログラムの説明

```
1  let fib n =
2  if n = 0 then failwith "Error"
3  else
4    let rec fib2 n =
5      if n = 0 then 1
6      else if n = 1 then 1
7      else fib2 (n-1) + fib2 (n-2)
8    in fib2 n
9  ;;

10 fib 0 ;;
11 fib 1 ;;
12 fib 7 ;;
13 fib 8 ;;
```

プログラムの説明

２行目：n が０であるときエラーを出力する。（n が０であるときを定義しなくてよいならばプログラムは４から７行目で完成する。）

５行目：n が０のとき１通りを足す。

６行目：n が１のとき１通りを足す。

７行目：n が２以上のとき、n-1 段を上がる方法と n-2 段を上がる方法を足し合わせ、再帰させる。

6.3　アルゴリズムについての考察

n 段まで残り k 段にいるとき、１段か２段を上る選択肢がある。１段登ったとき、いる場所は残り k-1 段となり、２段登ったとき残り k-2 段となる。１段登るのか２段登るのかの選択は自由であるため、

$$(\text{残り } k \text{ 段の上り方}) = (\text{残り } k-1 \text{ 段の上り方。}) + (\text{残り } k-2 \text{ 段の上り方。})$$

が成立する。このことから再起関数を利用して簡単にプログラムすることができる。

再起を終わらせる部分、つまりプログラムの５，６行目では１通りしか残されていないため１とする。

6.3.1 結果の考察

n が１から１０までの時それぞれ階段の上る方法を出力した結果は、

$1-->2-->3-->5-->8-->13-->21-->34-->55-->89$ となった。

この順序の規則性に注目するとフィボナッチ数列になっていることがわかる。フィボナッチ数列とは、$a_{n+1} = a_n + a_{n-1}$ただし $a_1 = 1, a_2 = 2, n = 2, 3, 4, , ,$ を満たす数列のことであり、プログラムで再帰させたときの式と一致していることがわかる。