情報科学実験１　レポート課題２

情報科学科 2 年 6319031 上瀬　友紀

2020 年 6 月 24 日

1　課題 1

1.1　ソーティングアルゴリズムの種類について調査を行い、適切に引用してまとめなさい。

バブルソート

バブルソートは昇順に並べる場合、左から右へと順に隣り合った要素の大小を比較し、右の方が大きくなるようにそれぞれを入れ替える、という作業を昇順に並ぶまで繰り返すというアルゴリズムである。要素数が N のとき、一周するのに必要な回数は N,N-1,N-2,...,1 となり、合計で $N(N-1)/2$ となってオーダーは $O(N^2)$ であるため、ソーティングアルゴリズムの中では比較的遅い方である。

選択ソート

選択ソートは、昇順に並べる場合要素内の最大値を探し、それを配列の先頭要素と交換することで整列を行うアルゴリズムである。要素数が N のとき、一周するのに必要な回数は N,N-1,N-2,...,1 となり、合計で $N(N-1)/2$ となってオーダーは $O(N^2)$ であるため、ソーティングアルゴリズムの中では比較的遅い方である。

クイックソート

クイックソートは、リストにおいてピボットと呼ぶ要素を軸に分割を繰り返して整列を行うアルゴリズムである。「分割を繰り返して整列を行う」ような手法を分割統治法と呼ぶ。
1, 要素数が 1 つかそれ以下なら整列済みとみなしてソート処理を行わない
2, ピボットとなる要素をピックアップする
3, ピボットを中心とした 2 つの部分に分割する - ピボットの値より大きい値を持つ要素と小さい値を持つ要素
4, 分割された部分 (サブリスト) に再帰的にこのアルゴリズムを適用する
分割統治法は手順 4. にあるように再帰処理で実現される。
平均計算量は $O(n \log n)$ で一般的に最速のアルゴリズムであるとされるが、最悪計算量は $O(n^2)$ であるため、場合によっては他のアルゴリズムのほうが速い場合がある。

マージソート

マージソートは整列されていないリストを 2 つのリストに分割して、それぞれを整列させた後、それらをマージして整列済みのひとつのリストを作るアルゴリズム。マージを利用してリストを整列するのでマージソートという名がついている。
1, 整列されていないリストを 2 つのサブリストに分割する
2, サブリストを整列する
3, サブリストをマージしてひとつの整列済みリストにする
平均計算量、最悪計算量共に O(n log n) であるため、一般的に安定した処理が実行できる。

バケットソート

バケットソート (ビンソート [Bin Sort] とも呼ばれる) はあらかじめデータがとりうる値すべての容器 (バケット) を順番どおりに並べて用意しておき、値を対応する容器に移すことでソートを行う整列アルゴリズム

である。データがとりうる値がわかっていなければソートのためのバケット [Bucket] を準備することができない為このアルゴリズムは使えない。比較を用いない整列アルゴリズムである。bucket を k 個使う場合、計算量は O(n + k) となり線形時間ソートとなるが、入力できる要素が k 種類というような制限がある。

参考文献:

https://www.codereading.com/algo_and_ds/algo/
https://qiita.com/r-ngtm/items/f4fa55c77459f63a5228

1.2 調査したアルゴリズムのうち、最低二つのアルゴリズムを再帰関数を利用して実装しなさい。それぞれについてアルゴリズムおよびプログラムの説明もすること。

今回は選択ソート、クイックソート、マージソート、バブルソートの 4 つを実装した。プログラムの説明中に行数が出てきた場合、ソースファイルの行数を指すものとする。また、各アルゴリズムで説明の様式がかなり異なるが、それぞれ同じ様式で説明しようとするとかなり難解になってしまう為ご容赦いただきたい。アルゴリズムの説明は 1.1 の通りのため、以下にはプログラムの説明を記す。

1.2.1 選択ソート

コード:1-1.ml

実行結果:

sentaku [4;4;3;2;1;1;6;0;-1];;
sentaku [0];;
sentaku [];;
- : int list = [-1; 0; 1; 1; 2; 3; 4; 4; 6]
- : int list = [0]
- : 'a list = []

関数 min でリストから最小のものを見つける (line10-16)-> それをリストの最初に持っていく (line21) -> リストの残りでまた最小を見つけて…の繰り返し。関数 check(line1-9) でリストから最小値を消して (line21) で最初に持ってきている。

1.2.2 クイックソート

コード:1-2.ml

実行結果:

quick [4;4;3;2;1;1;6;0;-1];;
quick [0];;
quick [];;
- : int list = [-1; 0; 1; 1; 2; 3; 4; 4; 6]
- : int list = [0]
- : 'a list = []

例えば lst = [2;4;1;3] とする。
まず [ ] でないので 2 :: [4;1;3] と分けられる (line10)。これを rip first lest に代入 (line11)、rip 2 [4;1;3] を計算し ([1],[4;3]) が出力される (line1-6)。
(quick [1])@(2 :: quick [4;3]) を計算すると (line12)、quick [1] = [1],quick [4;3] = [3;4](*1) より、[1;2;3;4] が出力される。
(*1)quick [4;3] は rip 4 [3] = ([3],[ ]) より (quick [3])@(4 :: quick[ ]) = [3;4] となる。

1.2.3 マージソート

コード:1-3.ml
実行結果:

```
merge [4;4;3;2;1;1;6;0;-1];;
merge [0];;
merge [];;
- : int list = [-1; 0; 1; 1; 2; 3; 4; 4; 6]
- : int list = [0]
- : 'a list = []
```

(1) 関数 bunkatsu... リストを分割。例:bunkatsu [4;3;2;1] -> [[4];[3];[2];[1]]
(2) 関数 hikaku... リスト内のリスト 2 つの最初ずつの大小を比較していく。例:hikaku [4] [3] -> [3;4], hikaku [4;6] [5;7] -> [4;5;6;7]
(3) 関数 in_merge...hikaku を 2 つずつ全てに実行する。例:in_merge [[1;3];[2;4];[5;7];[6;8]] -> [[1;2;3;4];[5;6;7;8]]
(4) 関数 merge... リストが一つになるまで in_merge を繰り返す。
(4) により、まずはじめに (1) の処理を行ったあとにソートが完了するまで (2) を含む (3) の処理を繰り返し行っている。

1.2.4 バブルソート

コード:1-4.ml
実行結果:

```
merge [4;4;3;2;1;1;6;0;-1];;
merge [0];;
merge [];;
- : int list = [-1; 0; 1; 1; 2; 3; 4; 4; 6]
- : int list = [0]
- : 'a list = []
```

bubble [3;2;1] のとき、bubble2 [3;2;1] -> 3 :: [2;1] より (line8)、bubble2 [1] = [1] であることから、m=1,rest2=[](line9)。n=2(line8) より bubble2 [2;1] = [1;2](line10-11)。同様にして、1::3::[2]=[1;3;2]=bubble2[3;2;1]。ここで、リストの左端にリストの最小値が来ているので、その最小値を除いたリストをもういちど bubble2 に通す。(line13-14) これを繰り返すことでソートが完了する。

1.3　実装したアルゴリズムについて、プログラムの実行時間を定量的に考察しなさい。

以下のリストを各アルゴリズムでマージしたときの実行時間を測定する。

[4;4;3;2;1;1;6;0;-1;3;4;5;6;7;7;7;56;5;4;43;3;3;4;4;54;4;65;6;7;7;6;5;4;4;3;
3;5;5;6;6;67;6;2;1;1;2;3;2;342;4;45;6;6;7;3;242321;23414154654;7856463523;
123131563467;6543534647456;7674564523;12412434654;76474567456356345;
23542354235235;5454234221;4323341234;67675667;45645345436543;676654675654;
4345345345;453535234;4353463645645;75674563;3465435563563;456345634563;
4663456435634;4564434343;453664343655346;3465434353465;346543643565343546;
45436655436;346543654536;64545634534356;6436434653453;346543245254;
6354654363464534;7658576;3453434;453456;324;132131;5345352;12313123;453453;46545647;
667456345;354241234;4;142314321432;3543535634;764745645;34532235;6545667567;
7856785674;543643563;54363466;4534523322222;56778795;244444465;76545635;
56756757547;5634563;245643;74565645645;567567568;4764564564564;76567574564;
45746758;57847654;45674674;4564;3453532;75768678;57567546;987654;2131;
325425757567;8676;4345325232;421;535636745746;7657564;65767;6;4;3;4;65;5;
76;567;8;56436;6547;76;68768;6785;746;3453;45;678;5678;87;56645;345;34;
654536;4576;56756;78586;6797;87;56765;465;46;46345;56;45745;45546;674566;
57641;22;11;22;33;44;55;656;767;7676;66;655;554;445;4545;445;4334;4334;
5;5556;6565;66;767;77;77;78;8778;8787;8;66;66;655;554;45;5445;444;445;55;5666;66;767;
77;66;677676;77;87;88;88;8787;76;65545;45;44;4343;6556;5465;6;656;7;7;7;
78787;7787878;8898;9989;989;9099;8887;78767;6565;54443;3434;44554;556;656;
6776;77;887;887;878;9898;98;88;787878;77;6766;554;4545;454;43434;4343;4343;
34543345;565;6565655;544;443;33;2223;3323;23233;3443;345454;454556;656;66;
6776;778;8878;889;9889;8776;655;54544;44;434;43;33;22;22332;2332;4334;43;434535;
354654665654;7655764574687578;665565;454544;343443;43343;232332;233443;
434554;5445654;566565;656565;44343;344354543;43345534;3432243;2332;343434;
4554454;5645665;5665;67677;76786776;7665;545445;454545;33434;34343;3324324;
343443;4534545;5456;566565;766776;7878;7;76;5;4;3;2323;322334;343443;4545;45454556;5656;];;

選択ソート

real 0m0.007s

user 0m0.005s

sys 0m0.000s

クイックソート

real 0m0.001s

user 0m0.001s

sys 0m0.000s

マージソート

real 0m0.001s

user 0m0.001s

sys 0m0.000s

バブルソート

real 0m0.053s

user 0m0.004s

sys 0m0.000s

以上のように、$O(n^2)$ である選択ソートとバブルソートよりも、$O(n \log n)$ であるクイックソートとマージソートの方が圧倒的に実行時間が短い。

2 課題 2

2.1 二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。

まず先に、様々な場所で使う関数を先に定義する。

```
(*リストの要素数を返す*)
let rec length lst =
  match lst with [] -> 0
  | _ :: rest -> 1 + (length rest)
let kyori lst = length lst
(*2つのリストを結合する*)
let rec append l1 l2 =
  match l1 with [] -> l2
  | first :: rest -> first :: (append rest l2)
(*リストを反転させる*)
let rec reverse lst =
  match lst with [] -> []
  | first :: rest -> append (reverse rest) [first]
(*n個のリストを結合する*)
let rec concat lst =
  match lst with [] -> []
  | n :: rest -> append n (concat rest)
(*リストの全要素にそれぞれ関数fで定義された処理を行う*)
let rec map f lst =
  match lst with
  [] -> []
  | v :: rest -> f v :: map f rest
(*リストyの先頭に要素xを追加する*)
let cons x y =(::) (x,y);;
```

以下、最長経路長とそのルートを出力するという目標に向かって関数を組んでいく。方針としては、与えられた座標のリストを list とすると、list 内の座標のすべての経路を求める -> その中の経路長が同じ経路を削除する-> 残った経路のすべての経路長を計算する -> その中から最長の値と、その時の経路を出力する

```
(* lstの各要素xと，xより左にある要素のリスト，xより右にある要素のリストを列挙する．
例えばselect [1;2;3] ;; では [([],1,[2;3]) ; ([1],2,[3]) ; ([1;2],3,[])] が出力される.*)
let rec select ls_empty lst = match lst with
  | [] -> []
  | x :: rest -> (reverse ls_empty, x, rest) :: select (x :: ls_empty) rest
let select lst = select [] lst
let rec retsu lst =
  let n = length lst in
(* リストから要素n個を選えらんでその全順列を出力する *)
let rec permutation n lst =
  match n, lst with
  | 0, _ -> [[]]
  | _, [] -> []
  | n, lst ->
    concat (
    map
     (fun (a, x, b) ->
      map
       (cons x)
       (permutation (n - 1) (concat [a;b]))) (
     select lst )) in
    permutation n lst
;;
(*順周りでルートが同じ順番を省く
例えば(0,0)->(2,2)->(3,3)->(0,0)と(2,2)->(3,3)->(0,0)->(2,2)は同じ経路長なので片方消す*)
let rec fact n =
  if n = 1 then 1 else fact (n-1) * n
let shoukyo lst =
  let lst2 = (retsu lst) in
  let k = 1 in
  let n = length lst in
    let rec in_shoukyo k lst2 =
    match lst2 with [] ->[]
    | x :: rest ->
    if k > (fact (n-1)) then []
```

```
    else x :: (in_shoukyo (k+1) rest) in
in_shoukyo k lst2
    ;;
(*リスト内の最初の要素を取り出し、それを最後に追加する*)
let first lst =
  match lst with [] -> failwith "error"
  |x :: rest -> x
;;

let tsuika lst =
  let lst_hanten = reverse lst in
  (first lst) :: lst_hanten;;

  let rec alltsuika lst =
    map tsuika (shoukyo lst)
    ;;
(*一周の経路長を求める*)
let kyori ((a,b),(c,d)) =
  let zyou n = n *. n in
  sqrt(zyou(float_of_int(a - c)) +. zyou(float_of_int(b - d)))
let rec keirocho lst =
  match lst with [] -> 0.
  | [x] -> 0.
  | x :: y :: rest -> kyori (x,y) +. (keirocho (y :: rest))
  ;;
  (*各ルートの経路長をリストにする*)
  let all_keirocho lst =
    let lst = alltsuika lst in
    let rec in_all_keirocho lst =
    match lst with [] -> []
    | x :: rest -> keirocho x :: (in_all_keirocho rest)
    in in_all_keirocho lst
  ;;
  (*経路長のリストから最長のものを取り出す*)
  let max_keirocho lst =
    let lst = all_keirocho lst in
  let rec max lst =
  match lst with [] -> failwith "error1"
  |[x] -> x
  |x :: y :: rest ->
```

```
  if x > y then max (x :: rest)
  else max (y :: rest) in
  max lst
  ;;
  (*最長経路長となるルートを出力*)
let max_route lst =
  let last = alltsuika lst in
  let rec in_max_route last =
  match last with [] ->[]
  | x :: rest ->
  if (keirocho x) = (max_keirocho lst) then x
  else in_max_route rest in
  in_max_route last
  ;;
```

実行結果:
max_route [(5,5);(15,10);(10,33);(13,4);(1,12)];;
- : (int * int) list = [(5, 5); (10, 33); (13, 4); (1, 12); (15, 10); (5, 5)]
max_keirocho [(5,5);(15,10);(10,33);(13,4);(1,12)];;
- : float = 97.3423653939681373

2.2 アルゴリズムについて考察しなさい。

まずこの課題は、巡回セールスマン問題 (tsp) の類題である。tsp は NP 困難であることが知られており、tsp は最短経路を求めるのに対し本課題は最長経路を求める問題であるという差異はあるが、それが問題解法自体に大きな影響を与えるとは考えづらいため、本課題も NP 困難であると考えるのが妥当であろう。NP 困難は、最適解が存在するが、計算量が膨大である為最適解を求めることが非常に困難な問題のことである。よって、この問題を解くアルゴリズムは計算量をいかに減らせるかが重要であると言える。効率化したアルゴリズムとしてはダイクストラ法などがあるが、このアルゴリズムを私の知識で ocaml で記述するのは不可能であると判断したため、効率は悪いが順列全探索にて全経路を調べたあと、できるだけ候補を絞ってから経路長を計算し、最長経路を求めるという方針を立てた。コードの順列を求めたあと同一の経路を省いている部分 (line55-69) について、本来この部分だけでは同じ順回りの場合しか省くことができず、逆回りで同じ経路を辿るものは残ってしまう。しかし今回逆回りも省こうとすると、かえって計算量が大幅に増加し、計算結果の出力までの時間が大幅に増加してしまう問題が発生したため、あえて全ては絞らずにある程度同じ経路が残った状態で計算するようにした。以下に逆回りを省くプログラムを記す。

```
(*逆回りで同じルートを辿る経路を省く
例えば(0,0)->(2,2)->(3,3)->(0,0)と(0,0)->(3,3)->(2,2)->(0,0)は同じ経路長なので片方消す*)
    let rev_shoukyo lst =
      let lst = alltsuika lst in
```

```
let rec in_rev_shoukyo lst =
  match lst with [] -> []
  |[x] -> [x]
  |x :: y :: [] -> if reverse x = y then [x] else [x;y]
  |x :: y :: rest -> if reverse x = y then x :: (in_rev_shoukyo rest)
                    else (in_rev_shoukyo (x :: (append rest [y])))
  in in_rev_shoukyo lst
  ;;
```

これを関数 alltsuika と kyori の間に挿入すると、処理自体はうまく行く。ただし、座標数が 7 を超えたあたりから計算結果の出力速度が大幅に低下し、実用的ではなくなってしまう。

3　課題 3

3.1　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。また、アルゴリズムを考察しなさい。

コード:3-1.ml

実行結果:

kaidan 3;;

- : int = 3

kaidan 5;;

- : int = 8

kaidan 50;;

- : int = 20365011074

一回目に上る段数で場合分けした漸化式をつくる。kaidan (n) を n 段ある階段の上り方の場合の数とすると、kaidan (n) = kaidan (n-1) + kaidan (n-2) である。3-1.ml で、これを再帰関数として定義した。

また、$kaidan(n) = a_n$ とすると、$x^2 - x - 1 = 0$ の特性方程式を解くと、$x = \frac{1 \pm \sqrt{5}}{2}$. ここで、$\frac{1 \pm \sqrt{5}}{2} = a, b$ とすると、$a_n - aa_{n-1} = b^{n-2}(2-a), a_n - ba_{n-1} = a^{n-2}(2-b)$ であり、2 式から a_{n-1} を消去し、a,b を代入すると、a_n の一般項は、

$$a_n = \frac{(\frac{1+\sqrt{5}}{2})^{n-1}(2 - \frac{1-\sqrt{5}}{2}) - (\frac{1-\sqrt{5}}{2})^{n-1}(2 - \frac{1+\sqrt{5}}{2})}{\sqrt{5}} \tag{1}$$

となる。実際、$(a_1 = 1, a_2 = 2, a_3 = 3, a_4 = 5, ...)$ となり、3-2.ml のように

```
let kaidan2 n =
  let rec power x n =
    if n = 0 then 1.
    else if n = 1 then x
    else x *. (power x (n-1))
    in int_of_float (((power(((1. +. sqrt(5.)) /. 2. )) (n-1))*.(2. -. ((1. -. sqrt(5.)) /. 2. ))
    -. (power(((1. -. sqrt(5.)) /. 2. )) (n-1))*.(2. -. ((1. +. sqrt(5.)) /. 2. )) ) /. sqrt(5.))
    ;;
```

としても上り方の場合の数を求めることができる。また、関数 kaidan 50、関数 kaidan2 50 において、

```
# kaidan 50;;
- : int = 20365011074
real    5m36.935s
user    4m56.353s
sys     0m0.055s
```

```
# kaidan2 50;;
- : int = 20365011074
real    0m0.001s
user    0m0.001s
sys     0m0.000s
```

見ての通り処理速度は kaidan2 が遥かに速いのは一目瞭然である。しかし、kaidan2 では float 型の計算が絡んで段数が更に大きくなった場合出力される値に正確なものと比べて誤差が出る可能性がある。よって、kaidan2 を用いるのは段数がある程度小さい場合か少量の誤差を無視できる場合に限られる。

4　課題 4

4.1　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

コード:4.ml
実行結果:
differ (fun x -> x *. x +. x +. 1.) (-0.5)
;;
- : float = 1.000000082740371e-05
x における f の微分係数は以下のように定義される。

$$f'(x) = \lim h \to 0 \frac{f(x+h) - f(x)}{h}$$

よって、関数 differ では h を 0 に限りなく近づけるのを、代わりに h に微小量を代入することによって f の微分を再現した。

4.2　関数 f の極値 1 つ求める関数 ext を定義しなさい。

コード:4.ml
実行結果:;;
ext (fun x -> x *. x +. x +. 1.) (-5.,5.)
;;
- : float = 0.750000002500002
x を指定された定義域内で動かしていき、微分係数が 0 になるもしくは 0 に非常に近づく点を探す（離散的な値しか代入することができないため、極値をとる x の値を飛ばしてしまう可能性があるため)。ある点の前後で符号が反転するとき、その点は極値である。
4.ml では、f_2 で関数を定義し、(b,c) で定義域を定義している。例えば (b,c) のとき、$b \leq x \leq c$ を表している。また、(line8) で、前後の微分係数の積を調べることにより、符号が一致しているかどうか調べている。

4.3　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。

$f'(x)$ の値を調べるために x の値を n 刻みで動かしていくとする。また、$f(x) = x^2 + x + 1$ とする。
$n = 10.e - 5$ のとき

```
- : float = 0.750000009999999717
real    0m0.012s
user    0m0.010s
sys     0m0.000s
```

$n = 10.e-6$ のとき

```
- : float = 0.750000000100000452
real    0m0.120s
user    0m0.078s
sys     0m0.000s
```

$n = 10.e-7$ のとき

```
- : float = 0.750000000000999312
real    0m0.835s
user    0m0.780s
sys     0m0.004s
```

$n = 10.e-8$ のとき

```
- : float = 0.75000000000001
real    0m7.546s
user    0m7.448s
sys     0m0.004s
```

$n = 10.e-9$ のとき

```
- : float = 0.750000000000000222
real    1m16.708s
user    1m14.760s
sys     0m0.004s
```

以上より、微小量の細かさが n 倍になるにつれて実行結果の精度は高くなるが、多少の誤差はあるが実行時間も n 倍になることが読み取れる。

5　課題 5

5.1　台形の面積を求める関数を定義しなさい。

　コード:5.ml
実行結果:
daikei 3. 4. 5.;;
- : float = 17.5
- : float = 24.1605004999997028
台形の面積は (上底 + 下底)× 高さ/2 で定義されるから、上底 =a、下底 =b、高さ =h とすると、

```
let daikei a b h =
  (a +. b) *. h /. 2.
;;
```

と書ける。

5.2　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
出力:$\int_a^b f(x)dx$

　コード:5.ml
実行結果:
integrate (fun x -> x *. x +. 2. *. x +. 1.) (-3.) 3.
;;
- : float = 24.1605004999997028
台形の区分求積法を用いる。要件の関数の場合、h を微少量とすると、上底 f(x)、下底 f(x+h)、高さ h の台形を考え、x=a,a+h,a+2h,... としながら、$x + h \geqq b$ となるまで台形の面積の和を取っていく。
5.ml において、f を任意の関数、a,b を定義域とする。例えば、integrate f a b のとき、$a \leq x \leq b$ である。

5.3　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。

　台形の高さを n とする。
$n = 10.e - 3$ のとき

```
- : float = 24.1605004999997028\\
real    0m0.001s\\
user    0m0.001s\\
sys     0m0.000s\\
```

$n = 10.e - 4$ のとき

```
- : float = 24.0160050004952268\\
real    0m0.002s\\
user    0m0.002s\\
sys     0m0.000s\\
```

$n = 10.e - 5$ のとき

```
- : float = 24.0000000100309663\\
real    0m0.045s\\
user    0m0.015s\\
sys     0m0.007s\\
```

以上より、台形の高さを小さくするにつれて、精度は高くなり、実行時間が増加することが読み取れる。
なお、$n = 10.e - 6$ とすると、以下のエラーが発生する。

```
Stack overflow during evaluation (looping recursion?).
```

このエラーは再帰関数においてループから抜け出せない場合に発生することが多いが、今回はそうではなくエラー文にも有るとおり、逐次入出力が繰り返されるデータを一時的に貯えるためのデータ構造である「スタック」が一杯になってしまいそれ以上蓄えることができなくなってしまっている。再帰的なループから抜け出せないときにこのエラーが出力されるのは、再帰的な処理毎にスタックが蓄えられ、無限に増えていってしまってる為である。今回は再帰関数で定義される処理 (微小な高さの台形の面積を足していく) の行われる回数が余りにも多いせいでエラーが出力されてしまった。課題 4 の際にこのエラーが出力されなかったのは、4 においては今回の台形の面積のような最終的な処理が終わるまで蓄えていなければならないデータが無いことが要因だと考えられる。例えば 4 の (line8) での符号の確認は一度確認が終われればその後そのデータを保存しておく必要がない。

6　課題 6

6.1　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。

　コード:6.ml

実行結果:

collatz 99;;

- : int = 1

n が偶数なら 2 で割り、奇数なら 3 かけて 1 を足すという処理を 1 になるまで繰り返すというのを関数 collatz で再帰的に定義している。

6.2　ステップ数 (プログラムが停止するまでの再帰の回数) について考察しなさい。

　課題 6 は、

「任意の正の整数 n をとり、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 を足すという操作を繰り返すと、最終的に 1 に到達するのではないか」

というコラッツ予想と呼ばれる問題に関連する課題である。コラッツ予想自体は未解決であるが、本課題のように 1 になるまで処理を繰り返すプログラムは書くことはできる。

　また、以下は縦軸にステップ数、横軸に n の値を取ったグラフである。これを見ると分かるように、n の値とステップ数には何らかの関係性があることが分かる。例えば十分に大きな偶数 n とその前の値 n-1 を考えると、多くの場合 n の方がステップ数が少なくなる傾向があると考えられる。(141(ステップ数 15) と 142(ステップ数 103) のような例外もある)。さらに、$n = 2^m \times l$ とすると、m の値が大きければ大きいほど、n の周辺の値よりもステップ数が小さくなる傾向があるとも考えられる。

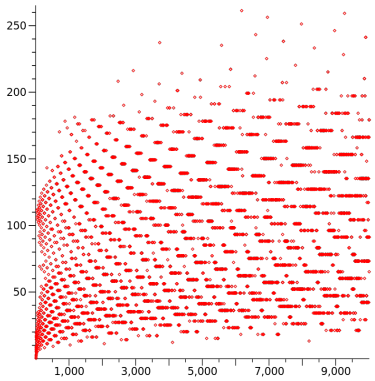

図1　n とステップ数の相関図