

ОБ ОПЫТЕ ИСПОЛЬЗОВАНИЯ СРЕДЫ WOLFRAM MATHEMATICA В КУРСЕ ДИСКРЕТНОЙ МАТЕМАТИКИ

Иванов О. А.¹, доктор педагогических наук, профессор, o.a.ivanov@spbu.ru
Фридман Г. М.², доктор технических наук, профессор, grifri@finec.ru

¹Санкт-Петербургский государственный университет,
Университетский пр., д. 28, Петергоф, 198504, Санкт-Петербург, Россия

²Санкт-Петербургский государственный экономический университет,
Садовая ул., д. 21, 191023, Санкт-Петербург, Россия

Аннотация

В работе обосновывается необходимость включения компьютерного практикума в качестве составной части курса дискретной математики. Приводятся и обсуждаются примеры заданий такого практикума, выполняемых студентами с использованием компьютерной среды Wolfram Mathematica.

Ключевые слова: дискретная математика, компьютерный практикум, структуры данных и алгоритмы, Wolfram Mathematica.

Цитирование: Иванов О. А., Фридман Г. М. Об опыте использования среды Wolfram Mathematica в курсе дискретной математики // Компьютерные инструменты в образовании. 2019. № 2. С. 43–54; doi:10.32603/2071-2340-2019-2-43-54

1. ВВЕДЕНИЕ

Подзаголовком к названию дисциплины «Дискретная математика» для студентов прикладных направлений (таких как «Прикладная математика и информатика» и «Бизнес-информатика») служат слова *структуры данных и алгоритмы*. Авторы всегда считали и продолжают считать, что, пока не написан программный код, реализующий некоторый алгоритм, нельзя считать, что этот алгоритм до конца понят. При этом в процессе написания кода важно разумно организовать используемые в нем данные, то есть понять, какова должна быть их *структура*. Из этого логически следует, что составной частью курса дискретной математики должен быть компьютерный практикум. В качестве программного продукта для проведения этого практикума мы используем (и рекомендуем использовать другим) среду Wolfram Mathematica, которая отличается простотой синтаксиса, богатством языка (Wolfram Language) и дружелюбием интерфейса.

Компьютерный практикум играет и важную педагогическую роль, приучая студентов четко выражать свои мысли, поскольку программный код работает так, как он написан, а не так, как студентом «было задумано».

По материалам занятий со студентами была написана книга [1], содержащая:

- теоретический материал, излагаемый в первом семестре в курсе дискретной математики для студентов направления «Прикладная математика и информатика» СПбГЭУ;
- введение в программирование в компьютерной среде Mathematica;
- одиннадцать «уроков программирования» с заданиями по курсу дискретной математики, при выполнении которых используется среда Mathematica.

В этой работе мы приводим и обсуждаем примеры заданий компьютерного практикума. Часть вошедшего в нее материала была представлена на конференции в Российском университете дружбы народов (см. [2]).

2. С ЧЕГО НАЧАТЬ?

В этом разделе приведены примеры заданий, предлагаемых студентам на самых первых занятиях практикума. С чего следует начинать, если на лекциях еще не введены основные понятия дискретной математики? Имеет смысл начать с обсуждения эффективности алгоритмов и их программных реализаций. Как вы увидите, с этой точки зрения первое задание является вполне поучительным.

Задание 1. Рассмотрим три следующие функции:

```
In[1]:= polyValueSum[a_, x_] := Sum[a[[i]] xi-1, {i, Length@a}]
polyValueNest[a_, x_] := a.NestList[# x &, 1, Length@a - 1]
polyValueFold[a_, x_] := Fold[#2 + x #1 &, 0, Reverse@a]
```

а) Убедитесь в том, что каждая из них вычисляет значение в точке x многочлена, коэффициентами которого являются элементы списка a .

б) Сравните скорости работы этих функций и объясните полученные вами результаты.

Отметим, что в книге [1, с. 114–125] обсуждаются одиннадцать разных кодов для вычисления значений многочлена. Введем список a коэффициентов многочлена и точку 2 Kurdubovs, в которой будем искать его значение.

```
In[4]:= a = RandomReal[{0, 1}, 50001];
x0 = RandomReal[];
```

Посмотрим на время работы этих функций для многочленов, степень которых равна 50000.

```
In[6]:= ClearSystemCache[]
polyValueSum[a, x0] // AbsoluteTiming
polyValueNest[a, x0] // AbsoluteTiming
polyValueFold[a, x0] // AbsoluteTiming
```

```
Out[7]= {0.048942, 1.72742}
```

```
Out[8]= {0.00457364, 1.72742}
```

```
Out[9]= {0.00704453, 1.72742}
```

В полученных результатах следует обратить внимание на значительное отставание в быстродействии первой функции. Функция `polyValueNest` строит список

$\{1, x, \dots, x^{n-1}\}$ посредством $n - 1$ умножений, тогда как первая функция строит каждую из этих степеней по отдельности. С другой стороны, число умножений, осуществляемых при вычислении значения многочлена посредством функции `polyValueFold` всего лишь в два раза меньше, чем при вычислении посредством второй из этих функций. Отметим, что функция `polyValueNest` работает быстрее именно в силу того, что `NestList` работает быстрее, чем `Fold`, хотя сам алгоритм менее эффективен.

Задание 2. Числа Фибоначчи.

а) Объясните, почему вам не удастся вычислить сороковое число Фибоначчи при помощи следующего кода

```
In[10]:= Clear[fibR]
         fibR[n_] := fibR[n - 1] + fibR[n - 2];
         fibR[1] = fibR[2] = 1;
```

б) Напишите эффективный код, вычисляющий числа Фибоначчи.

а) Ответ на первый вопрос связан с рассматривавшейся на занятиях задачей о числе обращений к функции для вычисления n -го члена последовательности, заданной рекуррентным соотношением второго порядка: $x_n = f(x_{n-1}, x_{n-2})$. Как известно, для этого необходимо $F_n - 1$ раз вызвать функцию f . Поскольку сороковое число Фибоначчи равно 102334155, то для вычисления F_{40} потребуется лишь на одно сложение меньше, что слишком много.

Интересно, что подсчет числа вызовов можно визуализировать. Напишем функцию, вычисляющую число обращений к f :

```
In[13]:= Clear[p, q, f, t]
         t[n_] := f[t[n - 1], t[n - 2]];
         t[1] = p;
         t[2] = q;
In[16]:= s[n_] := Count[t[n], f, Infinity, Heads -> True]
```

Посмотрим на первые значения последовательности s_n :

```
In[17]:= s /@ Range@20
Out[17]= {0, 0, 1, 2, 4, 7, 12, 20, 33, 54, 88,
          143, 232, 376, 609, 986, 1596, 2583, 4180, 6764}
```

Те, кто сам не догадался, что за последовательность появилась на экране, могут посмотреть на ответ, который дает Mathematica:

```
In[18]:= FindSequenceFunction[%]
Out[18]= -1 + Fibonacci[#1] &
```

б) Вот пример функции (не самой быстрой), вычисляющей числа Фибоначчи:

```
In[19]:= fibN[n_] := Nest[{Total@#, #[[1]]} &, {1, 0}, n - 1][[1]]
In[20]:= fibN /@ Range@10
Out[20]= {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}
```

Отметим, что различные коды для вычислений чисел Фибоначчи приведены в [1, с. 139–144].

Для выполнения следующего задания компьютер по сути не нужен, достаточно калькулятора, но авторы любят предлагать его на одном из первых занятий по компьютерному практикуму.

Задание 3. Рассмотрим задачу «о ханойской башне». Выясните, сколько времени потребуется для того, чтобы переложить башню из а) 20 колец и б) 40 колец, если каждую секунду перекладывается одно кольцо.

Итак, как же изменится время, если число колец всего-то навсего увеличили в два раза?

```
In[21]:= 220 / (24. × 60 × 60)
          240 / (365 × 24. × 60 × 60)
```

```
Out[21]= 12.1363
```

```
Out[22]= 34 865.3
```

Ответ всегда впечатляет: в первом случае надо немногим более 12 дней, во втором — более 34 500 лет! При изучении любого предмета важна эмоциональная составляющая процесса обучения. Одно дело — *знать*, что алгоритмы экспоненциальной сложности не имеют практического значения, другое дело — *увидеть* это.

Кстати Mathematica и сама может перевести секунды в годы

```
In[23]:= N@UnitConvert[Quantity[240, "Second"], "Year"]
```

```
Out[23]= 34 865.3 yr
```

Задание 4. Рассмотрим следующий рекурсивный алгоритм «быстрого» возведения числа в заданную степень. Если число n четно, то полагаем $x^n = (x^{n/2})^2$, если же n нечетно, то $x^n = x \cdot x^{n-1}$. Напишите функцию, вычисляющую число умножений, требуемых для возведения числа в данную степень, изобразите графически зависимость числа умножений от показателя степени и объясните полученный результат.

Сначала полезно просчитать «руками», что для возведения в тысячную степень нужно не 999, а всего 14 умножений. Рекурсивная функция пишется без труда.

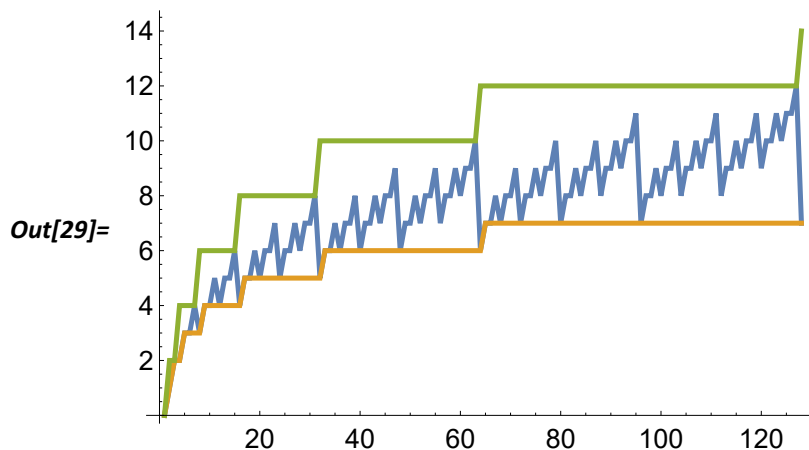
```
In[24]:= Clear[numOfMult]
          numOfMult[n_?EvenQ] := 1 + numOfMult[n / 2]
          numOfMult[n_?OddQ] := 1 + numOfMult[n - 1]
          numOfMult[1] := 0;
```

Для значений степени, меняющихся от 1 до 128, вычислим соответствующее число умножений:

```
In[28]:= tab = Table[
          {numOfMult[n], Ceiling@Log2@n, 2 Floor@Log2@n}, {n, 128}
        ] // Transpose;
```

и отобразим результат на графике. Мы получим следующую «пилу»:

```
In[29]:= ListLinePlot[tab, TicksStyle → Directive[14, Black],
          PlotStyle → Thickness[0.0075]
        ]
```



Интересно обсудить, каким значениям показателя степени соответствуют минимумы и максимумы на этом графике. Кроме того, можно добавить линии (изображенные на этом рисунке желтым и зеленым цветом), между которыми и располагается график, и попросить студентов получить соответствующие им оценки для числа умножений.

Уже на первом этапе обучения можно предлагать задачи, связанные с разработкой несложных алгоритмов. На примере следующего задания можно также привести первый пример кода, основанного на правилах замены.

Задание 5. Напишите несколько кодов, осуществляющих удаление дубликатов символов из данного списка (с сохранением порядка первого вхождения символа в этот список) и сравните их эффективность.

Wolfram Mathematica позволяет писать код, скажем так, «описательного характера». Что значит «убрать дубликаты с сохранением порядка первого вхождения»? А это значит, что в последовательности $\dots, a, \dots, a, \dots$ надо оставить только первый символ a . Так и сделаем, используя шаблоны:

```
In[30]:= delduplF[t_] := t //. {x___, a_, y___, a_, z___} => {x, a, y, z}
```

Следующие две функции написаны в процедурном стиле программирования и отличаются тем, что в первой из них производятся изменения в заданном списке, тогда как во второй искомый список строится последовательно, как только обнаруживается, что текущий элемент данного списка отсутствует в построенном ранее списке.

```
In[31]:= delduplP1[t_] := Module[{x = t, res = {}},
  While[Length@x > 0, AppendTo[res, x[[1]]];
  x = DeleteCases[x, x[[1]]];
  res
]
```

```
In[32]:= delduplP2[t_] := Module[{res = {}, n = Length@t},
  Do[If[FreeQ[res, t[[i]]], AppendTo[res, t[[i]]], {i, n}];
  res
]
```

Естественно, эта задача требует от студентов повторения работы встроенной функции `DeleteDuplicates` с предикатом `SameQ`. Сравним скорость работы функций для списка из 20 000 чисел:

```
In[33]:= t = RandomInteger[{0, 5000}, 20000];
```

```
In[34]:= ClearSystemCache[]
DeleteDuplicates@t; // AbsoluteTiming
deldup1P1@t; // AbsoluteTiming
deldup1P2@t; // AbsoluteTiming
```

```
Out[35]= {0.000138572, Null}
```

```
Out[36]= {2.16468, Null}
```

```
Out[37]= {1.18454, Null}
```

Как и следовало ожидать, вторая работает быстрее остальных, но всем ОЧЕНЬ далеко до встроенной DeleteDuplicates. Что же касается функции deletedup1F, то в данном случае правила замены работают на порядки медленнее. Вот пример для списка длины 1000:

```
In[38]:= deldup1F[RandomInteger[{0, 20}, 1000]]; // AbsoluteTiming
```

```
Out[38]= {7.58175, Null}
```

3. СВЯЗИ МЕЖДУ ТЕОРИЕЙ И ПРАКТИКОЙ

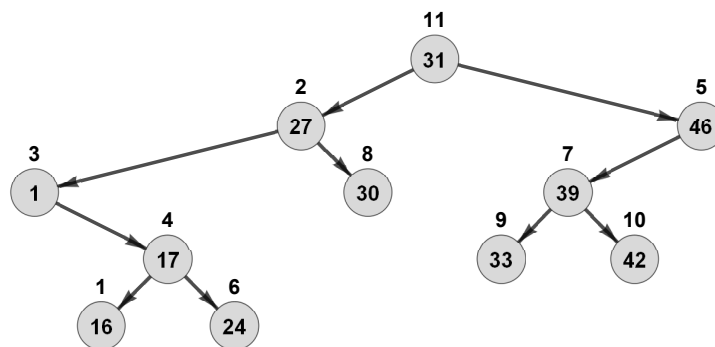
В работе [2] отмечалось, что есть двусторонняя связь между теорией и практикой, которая может быть выражена следующей диаграммой:

Теория \Rightarrow Программирование \Rightarrow Полученные данные \Rightarrow Теория

Первое из заданий проверяет понимание студентами структуры двоичного дерева поиска (и, конечно, его определения). К сожалению, слишком часто при выполнении этого задания они начинают использовать встроенную функцию SortBy.

Задание 6. Для данного двоичного дерева поиска напишите коды, строящие:

- список номеров вершин этого дерева в порядке возрастания хранящихся в них ключей и
- упорядоченный список ключей этого дерева.
- Напишите также функцию, проверяющую выполнение основного условия на ключи в двоичном дереве поиска.



Само дерево задается так:

```
In[39]:= bst = {{16, 0, 0, 4}, {27, 3, 8, 11}, {1, 0, 4, 2},
  {17, 1, 6, 3}, {46, 7, 0, 11}, {24, 0, 0, 4}, {39, 9, 10, 5},
  {30, 0, 0, 2}, {33, 0, 0, 7}, {42, 0, 0, 7}, {31, 2, 5, 0}};
```

Алгоритм формирования списка номеров вершин этого дерева в порядке возрастания их ключей чрезвычайно прост. Поскольку по определению двоичного дерева поиска для любой его вершины значения ключей во всех вершинах ее левого поддерева меньше значения ключа, хранящегося в этой вершине, а во всех вершинах ее правого поддерева ключи — больше, то для каждой вершины двоичного дерева поиска надо сначала выписать все вершины ее левого поддерева, затем саму эту вершину, а далее — все вершины ее правого поддерева. Так и следует написать.

```
In[40]:= vertList[bst_, 0] := {}
  vertList[bst_, r_] :=
  vertList[bst, bst[[r, 2]]] ~Join~ {r} ~Join~ vertList[bst, bst[[r, 3]]]
```

Заметьте, что код дает возможность не проверять непустоту поддеревьев рассматриваемой вершины. Для проверки этого кода применим его к изображенному выше дереву:

```
In[42]:= vertList[bst, 11]
Out[42]= {3, 1, 4, 6, 2, 8, 11, 9, 7, 10, 5}
```

По сути дела мы выполнили и пункт б). Действительно, поскольку у нас уже есть «правильная» последовательность вершин, то достаточно выбрать ключи в этом порядке.

```
In[43]:= keysList[bst_, r_] := bst[[vertList[bst, r], 1]]
```

```
In[44]:= keysList[bst, 11]
Out[44]= {1, 16, 17, 24, 27, 30, 31, 33, 39, 42, 46}
```

Наконец, укажем алгоритм для проверки основного свойства двоичного дерева поиска. Опять-таки, все достаточно просто. Надо выписать ключи в «правильном порядке» в соответствии с определением двоичного дерева поиска и проверить, что получившийся список значений ключей является строго возрастающей последовательностью.

Задание 7. Придумайте и реализуйте алгоритм, генерирующий случайное дерево.

Как ни странно, но в ядре Wolfram Mathematica пока нет встроенной функции, генерирующей случайное дерево. А поскольку функция RandomGraph генерирует и несвязные графы, то нет возможности использовать ее для получения деревьев, задав соотношение между числом его вершин и числом ребер.

Все, что следует сделать — это осознать, что помеченное дерево задается своим кодом Прюфера. Таким образом можно сгенерировать случайный список длины $n - 2$, содержащий числа от 1 до n , а затем построить дерево, для которого этот список является его кодом Прюфера.

Следующая функция строит дерево по его коду Прюфера.

```

In[45]:= fromPruferCode[l_List] :=
Module[{s = {}, d = Range[Length@l + 2], f = 1, vert},
Do[vert = First@Complement[d, f];
AppendTo[s, f[[1]] -> vert];
d = DeleteCases[d, vert];
f = Rest@f, {i, 1, Length@f}];
AppendTo[s, d[[1]] -> d[[2]]];
s]

```

Сгенерируем случайный список и воспользуемся этой функцией.

```

In[46]:= tree = fromPruferCode@RandomInteger[{1, 10}, 8]

```

```

Out[46]= {6 -> 5, 8 -> 6, 9 -> 8, 3 -> 9, 4 -> 3, 1 -> 4, 2 -> 1, 7 -> 2, 7 -> 10}

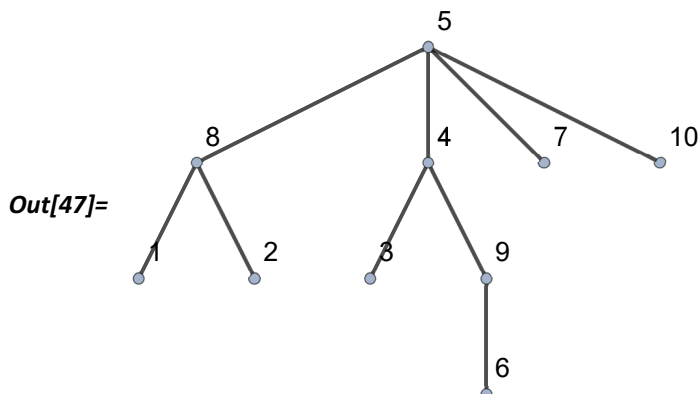
```

Теперь можно посмотреть на полученное дерево.

```

In[47]:= Graph[tree, VertexLabels -> "Name", EdgeStyle -> Directive[Black, Thick],
VertexLabelStyle -> Directive[Black, Plain, 14], ImageSize -> 300]

```



Задание 8. Придумайте и реализуйте алгоритм поиска базисных циклов в графе, заданном списком своих ребер. При этом ответом должны быть списки последовательно проходимых вершин каждого из этих циклов.

Наиболее естественный алгоритм связан с использованием теоремы Пуанкаре–Веблена–Александера. Если A — это матрица инцидентности ориентированного графа, то в пространстве решений системы $Ax = 0$ есть базис, состоящий из циклических векторов. Таким образом, план решения состоит в следующем. Построим матрицу инцидентности графа, далее посредством встроенной функции `NullSpace` найдем искомый базис из циклических векторов. Для каждого базисного вектора x строим набор соответствующих ему ребер графа. Если $x_i = -1$, то меняем ориентацию i -го ребра. В результате мы получим набор ребер цикла, но не в порядке их прохождения по этому циклу. Пусть, к примеру, получен набор ребер $\{5, 10\}, \{2, 7\}, \{3, 2\}, \{10, 3\}, \{7, 5\}$. Ясно, что они образуют цикл $5, 10, 3, 2, 7$. Надо только написать функцию, которая по списку ребер цикла строит последовательность вершин, получаемую при обходе по этому циклу.

Мы не будем приводить здесь полный набор кодов, а ограничимся только реализацией последнего шага алгоритма. В качестве примера рассмотрим набор t ребер цикла $1, 2, 3, 4, 5, 6, 7, 8$.


```
In[48]:= t = Partition[Range@8, 2, 1, 1] [[RandomSample[Range@8]]]
```

```
Out[48]= {{4, 5}, {1, 2}, {8, 1}, {6, 7}, {2, 3}, {3, 4}, {5, 6}, {7, 8}}
```

Приведем коды функций, написанных в разных стилях программирования. Начнем с функции, основанной на использовании правил замены:

```
In[49]:= cycleVert[t_] := Quiet@
  ReplaceRepeated[t, {a_, b___, c_, d___} := {c, a, b, d} /; a[[1]] == c[[2]],
  MaxIterations -> Length@t - 1] [[All, 1]]
```

Следующий код написан в функциональном стиле:

```
In[50]:= cycleVertF[t_] := Module[{v = t[[1]], rule = Rule@@@Rest@t},
  Nest[Append[#, Last@# /. rule] &, v, Length@t - 2]
```

Наконец, функция, написанная в стиле традиционного программирования.

```
In[51]:= cycleVertP[t_] := Module[{v = t, s},
  Do[s = i + 1;
  While[v[[s, 1]] != v[[i, 2]], s++];
  If[s != i + 1, {v[[i + 1]], v[[s]]} = {v[[s]], v[[i + 1]]}],
  {i, Length@t - 1}];
  v[[All, 1]]
]
```

Убедимся, что все они работают корректно.

```
In[52]:= cycleVert@t
  cycleVertF@t
  cycleVertP@t
```

```
Out[52]= {5, 6, 7, 8, 1, 2, 3, 4}
```

```
Out[53]= {4, 5, 6, 7, 8, 1, 2, 3}
```

```
Out[54]= {4, 5, 6, 7, 8, 1, 2, 3}
```

Конечно, первая из этих функций на больших списках будет работать значительно медленнее других. А вторую из них можно значительно ускорить.

4. ОБ ЭФФЕКТИВНОСТИ ПРЕДЛОЖЕННОЙ МЕТОДИКИ

Из изложенного выше следует, что компьютерный практикум в курсе дискретной математики играет две роли, способствуя, с одной стороны, лучшему пониманию студентами теоретического материала этого курса, с другой — развитию их алгоритмического мышления. Что касается его эффективности в первой из этих ролей, то здесь мы можем сослаться на материал работы [3]. В ней приведены результаты анонимных опросов, проводившихся со студентами направления «бизнес-информатика» СПбГУ и направления «прикладная математика и информатика» СПбГЭУ. Подавляющее большинство опрошенных студентов положительно оценили то влияние, которое оказал компьютерный практикум на их понимание дискретной математики.

Теперь, каким образом можно оценить влияние практикума на развитие алгоритмического мышления студентов? Во втором семестре 1-го курса одно из заданий состоит

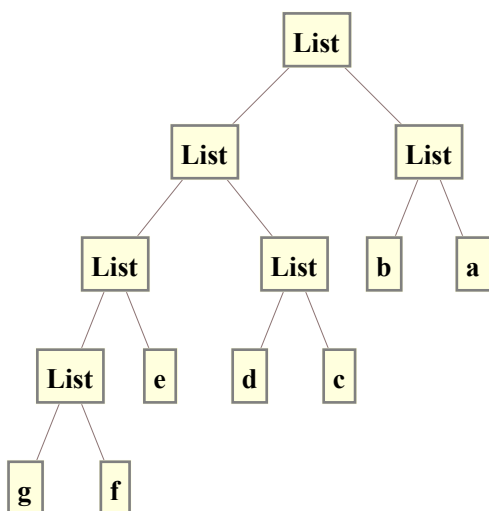
в выполнении так называемого «мини-проекта». Студентам предлагается написать код некоторого алгоритма (а порой и самостоятельно разработать алгоритм). Код, приведенный выше в строке ввода In[45], строящий дерево по его коду Прюфера, как раз был написан студентами первого (!) курса.

Приведем еще один пример выполненного студентами проекта (он приведен без изменений, поэтому не стоит судить слишком строго). Задание состояло в написании функции, осуществляющей декодировку сообщений, закодированных посредством алфавитного двоичного кодирования.

Прежде всего надо было понять, что схему кодирования естественно хранить в виде вложенного двоичного дерева.

```
In[55]:= treeCode = { { { { "g", "f" }, "e" }, { "d", "c" } }, { "b", "a" } };
          treeCode // TreeForm
```

```
Out[56]//TreeForm=
```



Рассмотрим сообщение

```
In[57]:= codeMessage1 = "00010010101110";
```

Оно переводится в набор правил, определяющих движение по дереву кодирования:

```
In[58]:= listOf = Characters[codeMessage1] /. {"1" → 2, "0" → 1};
```

Следующая функция ищет первый символ в сообщении:

```
In[59]:= step1[listOf_] := Module[{lst = treeCode, mess = listOf, i = 1},
  While[! StringQ[lst],
    lst = lst[[listOf[[i]]]]; i++
  ];
  {lst, listOf[[i ;;]]}
]
```

Применим ее данному сообщению:

```
In[60]:= step1@listOf
```

```
Out[60]= {f, {1, 1, 2, 1, 2, 1, 2, 2, 2, 1}}
```

Процесс декодирования состоит в последовательном применении этой функции.

```
In[61]:= decoding[{}] := Nothing;  
        decoding[listOf_] := {decoding[step1[listOf][[2]]],  
                             AppendTo[listMess, step1[listOf][[1]]][[1]]};
```

Получаем ответ:

```
In[63]:= listMess = {};  
        decoding[listOf];  
        Reverse@listMess // StringJoin
```

```
Out[65]= fedab
```

Безусловно, необходимо сказать, что навыки программирования в среде Mathematica студенты получают и в курсе «Системы компьютерной математики», изучаемой в семестре 2.

Список литературы

1. *Иванов О. А., Фридман Г. М.* Дискретная математика и программирование в Wolfram Mathematica (для бакалавров). Учебник для вузов. СПб.: «Питер», 2019.
2. *Иванов О. А., Фридман Г. М.* Компьютерный практикум с Mathematica в курсе дискретной математики. Тезисы докладов 5-й Международной конференции, посвященной 95-летию чл.-корр. РАН Л. Д. Кудрявцева. М.: РУДН, 2018. С. 451–452.
3. *Ivanov O. A., Ivanova V. V., Saltan A. A.* Discrete Mathematics course supported by CAS Mathematica // International Journal of Mathematical Education in Science and Technology. 2017. Vol. 48, № 6. P. 953–963. doi: 10.1080/0020739X.2017.1319979

Поступила в редакцию 05.02.2019, окончательный вариант — 28.03.2019.

Computer tools in education, 2019

№ 2: 43–54

<http://cte.eltech.ru>

doi:10.32603/2071-2340-2019-2-43-54

On Experience of Wolfram Mathematica Applications in Discrete Mathematics Course

Ivanov O. A.¹, PhD, professor, o.a.ivanov@spbu.ru

Fridman G. M.², PhD, professor, grifri@finec.ru

¹Saint Petersburg State University, 28, University pr., Petergof, 198504, Saint Petersburg, Russia

²Saint Petersburg State University of Economics, 21, Sadovaya st., 191023, Saint Petersburg, Russia

Abstract

A necessity is advocated in the paper of computer-aided lectures and seminars on Discrete Mathematics with use of Wolfram Mathematica. A variety of examples are presented and discussed of the exercises being given to students in the framework of the teaching course

Keywords: *discrete mathematics, computer practicum, algorithms and data structures, Wolfram Mathematica.*

Citation: O. A. Ivanov and G. M. Fridman., "On Experience of Wolfram Mathematica Applications in Discrete Mathematics Course," *Computer tools in education*, no. 2, pp. 43-54, 2019 (in Russian); doi:10.32603/2071-2340-2019-2-43-54

References

1. O. A. Ivanov and G. M. Fridman, *Discrete mathematics and programming in Wolfram Mathematica (for bachelors). Textbook for high schools*, Peter, Saint-Petersburg, 2019 (in Russian).
2. O. A. Ivanov and G. M. Fridman, "Computer workshop with Mathematica in the course of discrete mathematics," *Abstracts of the 5th International Conference dedicated to the 95th anniversary of the corresponding member*, L. D. Kudryavtseva, eds., RAS, RUDN, Moscow, pp. 451–452, 2018 (in Russian).
3. O. A. Ivanov, V. V. Ivanova, and A. A. Saltan, Discrete Mathematics course supported by CAS Mathematica, *International Journal of Mathematical Education in Science and Technology*, vol. 48, no. 6. pp. 953–963, 2017 (in Russian); doi: 10.1080/0020739X.2017.1319979

Received 05.02.2019, the final version — 28.03.2019.