МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Построение и анализ алгоритмов»

ТЕМА: ПОИСК С ВОЗВРАТОМ

Студент гр. 9382 _____ Кодуков А.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

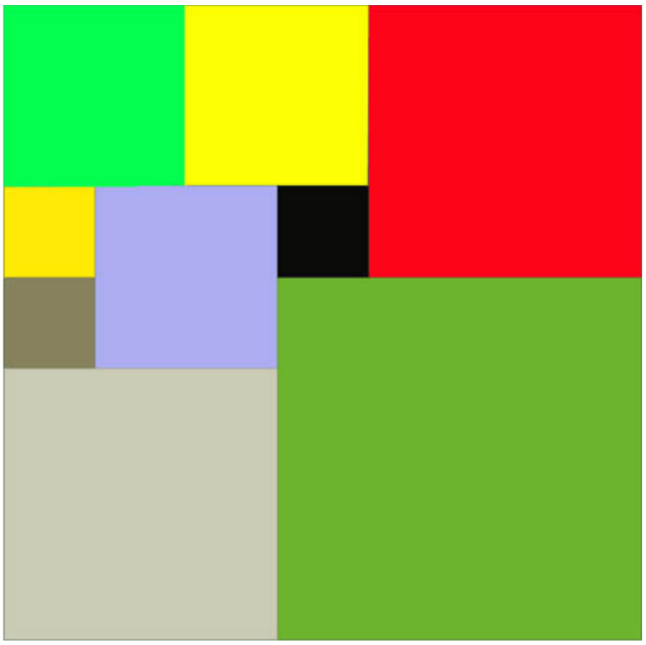
Цель работы:

Изучить и использовать на практике алгоритм поиска с возвратом

Задание:

У Вовы много квадратных обрезков доски. Их стороны (размер) изменяются от *1* до $N-1$, и у него есть неограниченное число обрезков любого размера. Но ему очень хочется получить большую столешницу - квадрат размера N . Он может получить ее, собрав из уже имеющихся обрезков(квадратов).

Например, столешница размера 7×7 может быть построена из 9 обрезков.



Внутри столешницы не должно быть пустот, обрезки не должны выходить за пределы столешницы и не должны перекрываться. Кроме того, Вова хочет использовать минимально возможное число обрезков.

Входные данные

Размер столешницы - одно целое число N $(2 \le N \le 20)$.

Выходные данные

Одно число K, задающее минимальное количество обрезков(квадратов), из которых можно построить столешницу(квадрат) заданного размера N. Далее должны идти K строк, каждая из которых должна содержать три целых числа x, y и w, задающие координаты левого верхнего угла $(1 \le x, y \le N)$ и длину стороны соответствующего обрезка(квадрата).

Вар. 1и:

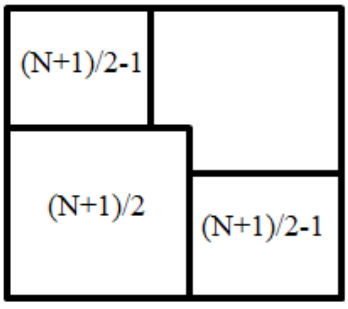
Итеративный бэктрекинг. Выполнение на Stepik двух заданий в разделе

Описание алгоритма:

Алгоритм поиска наилучшей упаковки основан на поиске с возвратом: квадраты ставятся подряд от угла, изначально выбирая наибольший возможный размер, пока поле не заполнится. Если понадобилось меньше квадратов, чем на прошлых итерациях, результат сохраняется. Затем алгоритм возвращается назад до тех пор, пока не встретит квадрат размера больше 1, стирает его и ставит вместо него квадрат меньшего на 1 размера. Алгоритм работает до тех пор, пока все необходимые расстановки не будут проверены.

Оптимизации:

- Карты с четной стороной всегда можно разделить на 4 квадрата без дополнительных вычислений (следствие 3 пункта)
- Если N составное, то разбиение будет аналогично разбиению карты с размером наименьшего простого делителя N
- Если N простое, то в оптимальном разбиении всегда будут присутствовать следующие квадраты:



- Таким образом, любой нечетный случай можно свести к простому N и рассматривать только разбиения, включающие эти 3 квадрата
- При достижении количества квадратов текущего лучшего разбиения можно вернуться назад на минимум 3 шага
- После заполнения карты помимо квадратов со стороной 1 удаляются квадраты со стороной 2, так как их разбиение и их перестановки в уже заполненной карте не изменят результат

- Проверка коллизий (выхода за границу и пересечение с другими квадратами) и поиск наибольшего возможного размера определяются за один цикл
- При определении наибольшего квадрата, который можно поместить, начиная с данной клетки, проверяются коллизии только по клеткам 2 смежных сторон, так как квадраты ставятся подряд, и остальные клетки потенциального квадрата автоматически не могут ни с чем пересечься
- После удаления квадрата запоминаются координаты его угла для использования следующим квадратом

Функции и структуры данных:

Структуры данных:

square – структура хранения данных о квадрате

step_stack – массив квадратов, описывающий заполнение карты

Хранение частичных решений:

step_stack curstack – текущее заполнение

step_stack best – заполнение с наименьшим количеством квадратов

Реализованные функции:

Поиск с возвратом

Сигнатура: `void Backtracking(int size)`

Аргументы:

- size – размер карты

Алгоритм:

- Сокращение размера карты, если N составное
- Расстановка первых 3 квадратов
- Пока не проверены все случаи расстановок квадратов в оставшейся области:
    - Пока карта не заполнена
        - Поставить квадрат максимально возможного размера (меньше удаленного с этого места квадрата, если такой есть) в первую свободную клетку

        - Проверить, не достигло ли текущее заполнения лучшего результата с прошлых итераций, если да, то сделать 2 шага назад и выйти из цикла
        - Найти следующую свободную клетку
    - Сравнить заполнение лучшим, обновить лучшее при необходимости
    - Сделать возврат по рекурсии до первого квадрата, который можно уменьшить
- Масштабировать лучший результат к изначальному размеру карты

Поставить один квадрат на карту

Сигнатура: `square PlaceSquare(int mapsize, square prediction)`

Аргументы:

- mapsize – размер карты
- prediction – первая пустая клетка/последний удаленный квадрат

Возвращаемое значение:

- (square) – поставленный квадрат

Алгоритм:

- Уменьшить размер prediction на 1
- Подобрать наибольший размер квадрата, если нет информации в prediction
- Заполнить соответствующие клетки карты

Поиск первой свободной клетки

Сигнатура: `square FindFreeCell(int mapsize)`

Аргументы:

- mapsize – размер карты

Возвращаемое значение:

- (square) – свободная клетка

Возврат

Сигнатура: `square StepBack(const int &size)`

Аргументы:

- size – размер карты

Возвращаемое значение:

- (square) – удаленный квадрат

Тестирование:

N	Квадраты	Количество	Всего квадратов установлено	Время работы
2	1 1 1 2 1 1 1 2 1 2 2 1	4	-	0.001
20	1 1 10 11 1 10 1 11 10 11 11 10	4	-	0.001
3	1 1 2 1 3 1 3 1 1 3 2 1 2 3 1 3 3 1	6	6	0.001
5	1 1 3 1 4 2 4 1 2 4 3 2 3 4 1 3 5 1 4 5 1 5 5 1	8	8	0.001
25	1 1 15 1 16 10 16 1 10 16 11 10 11 16 5 11 21 5 16 21 5 21 21 5	8	8	0.003
7	1 1 4 1 5 3 5 1 3 5 4 2 7 4 1 4 5 1 7 5 1 4 6 2 6 6 2	9	16	0.002
11	1 1 6 1 7 5 7 1 5 7 6 3 10 6 2 6 7 1	11	373	0.003

	6 8 1 10 8 1 11 8 1 6 9 3 9 9 3			
13	1 1 7 1 8 6 8 1 6 8 7 2 10 7 4 7 8 1 7 9 3 10 11 1 11 11 3 7 12 2 9 12 2	11	826	0.003
19	1 1 10 1 11 9 11 1 9 11 10 3 14 10 6 10 11 1 10 12 1 10 13 4 14 16 1 15 16 1 16 16 4 10 17 3 13 17 3	13	11097	0.005
23	1 1 12 1 13 11 13 1 11 13 12 2 15 12 5 20 12 4 12 13 1 12 14 3 20 16 1 21 16 3 12 17 7 19 17 2 19 19 5	13	41196	0.011
29	1 1 15 1 16 14 16 1 14 16 15 2 18 15 5 23 15 7 15 16 1 15 17 3 15 20 3 18 20 3	14	278660	14

	21 20 2 21 22 1 22 22 8 15 23 7			
37	1 1 19 1 20 18 20 1 18 20 19 2 22 19 5 27 19 11 19 20 1 19 21 3 19 24 8 27 30 3 30 30 8 19 32 6 25 32 1 26 32 1 25 33 5	15	3148369	1.041

Вывод:

В результате выполнения работы был изучен и реализован алгоритм поиска с возвратом.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <ctime>
#include <iostream>
#include <vector>
#include <cmath>

#define DEBUG

// map to test packaging
bool map[40][40];

// square to place on a map
struct square {
  int size, x_corner, y_corner;
};

// stack of square placing iterations
typedef std::vector<square> step_stack;

// current square placing stack
step_stack curstack;
// best packing
step_stack best;

// do step back on stack and delete last square from map
square StepBack(const int &size) {
  if (curstack.size() == 0) return square{-1, -1, -1};

  // get last placed square and do step back
  square lastsq = curstack.back();
  curstack.pop_back();
#ifdef DEBUG
  std::cout << "Deleted: " << lastsq.x_corner + 1 << " " << lastsq.y_corner + 1 << "
"
            << lastsq.size<< "\n";
#endif
  // clear last placed square
  for (int i = 0; i < lastsq.size; ++i)
    for (int j = 0; j < lastsq.size; ++j)
      map[i + lastsq.y_corner][j + lastsq.x_corner] = 0;

  return lastsq;
}

// find free cell in square map
square FindFreeCell(int mapsize) {
  for (int i = 0; i < mapsize; ++i)
    for (int j = 0; j < mapsize; ++j)
      if (map[i][j] == 0) return {0, j, i};
  // map is full
  return {-1, -1, -1};
}

// place one square on map using last deleted square position
square PlaceSquare(int mapsize, square prediction) {
  square res;
  // use last deleted square information
  if (prediction.size > 0) {
    res = prediction;
    res.size--;
  }
  else  // first step
```

```
    res = {mapsize - 1, 0, 0};

  // find size (less than size of previous and fits in map)
  int ressize = 1;
  while (ressize < res.size &&
          (res.x_corner + ressize) < mapsize &&
          (res.y_corner + ressize) < mapsize &&
         map[res.y_corner][res.x_corner + ressize] == 0)
    ressize++;
  res.size = ressize;
  // place square
  for (int i = 0; i < res.size; ++i)
    for (int j = 0; j < res.size; ++j)
      map[res.y_corner + i][res.x_corner + j] = 1;

  return res;
}
// find best packing with iterative backtracking
void Backtracking(int size) {
  bool step_back = false, full = false;
  int steps = 3, dividers = 1;
  square prediction{-1, 0, 0}, place_res{0, 0, 0};
  // redude map size to the smallest prime divider
  for (int i = 2; i <= size; i++) {
    if (size % i == 0) {
      dividers = size / i;
      size = i;
    }
  }
#ifdef DEBUG
  std::cout << "N reduced to the smallest prime divider, N = " << size << "\n";
#endif
  // place 3 first squares
#ifdef DEBUG
  std::cout << "Placing 3 first squares:\n";
  std::cout << "Placed square: 1 1 " << (size + 1) / 2 << "\n";
  std::cout << "Placed square: 1 " << (size + 1) / 2 + 1 << " " << size - (size + 1)
/ 2 << "\n";
  std::cout << "Placed square: " << (size + 1) / 2 + 1 << " 1 " << size - (size + 1)
/ 2 << "\n";
#endif
  place_res = PlaceSquare(size, {(size + 1) / 2 + 1, 0, 0});
  curstack.push_back(place_res);
  place_res = PlaceSquare(size, {size - (size + 1) / 2 + 1, 0, (size + 1) / 2});
  curstack.push_back(place_res);
  place_res = PlaceSquare(size, {size - (size + 1) / 2 + 1, (size + 1) / 2, 0});
  curstack.push_back(place_res);
  prediction = FindFreeCell(size); prediction.size = size - 1;

#ifdef DEBUG
  std::cout << "Start cycle:\n";
#endif
  // check all necessary variants
  do {
    int last_size = 0;

    full = false;
    step_back = false;

    // while map isn't full
    while (!full) {
      steps++;
      // place one square
      place_res = PlaceSquare(size, prediction);
#ifdef DEBUG
      std::cout << "Placed square: " << place_res.x_corner + 1 << " " <<
place_res.y_corner + 1 << " " << place_res.size << "\n";
```

```
#endif
      curstack.push_back(place_res);
      // go back if current result is worse then best
      if (best.size() > 0 && curstack.size() > 0 &&
          curstack.size() >= best.size()) {
        step_back = true;
#ifdef DEBUG
        std::cout << "Best packaging squares number was reached, go back:\n";
#endif
        StepBack(size);
        prediction = StepBack(size);
        break;
      }
      // prepare to next placing
      prediction = FindFreeCell(size);
      full = (prediction.size == -1);
      prediction.size = size;
    }
    if (!step_back) {
#ifdef DEBUG
    std::cout << "Map is full, current best packaging : " << best.size() <<
      " squares\nComparing with current packaging...\n";
#endif
      // compare with best result
      if (!step_back && curstack.size() < best.size() || best.size() == 0) {
        best = curstack;
#ifdef DEBUG
        std::cout << "Best packaging set to: " << best.size() << " squares\n";
#endif
      }
      else {
#ifdef DEBUG
      std::cout << "Best packaging wasn't changed\n";
#endif
      }
    }
    // go back once and while squares size = 1 (or 2 if map was full)
    do {
      prediction = StepBack(size);
    } while (prediction.size == 1 || (full && prediction.size == 2) &&
curstack.size() >= 3);
  } while (curstack.size() >= 3);
  // scale results to start map size
  for (int i = 0; i < best.size(); i++) {
    best[i].x_corner *= dividers;
    best[i].y_corner *= dividers;
    best[i].size *= dividers;
  }
  std::cout << "\nTotal placed squares: " << steps << "\n";
}

int main() {
  int size;

  std::cin >> size;

  double t0 = clock();
  if (size % 2 == 0) {
#ifdef DEBUG
    std::cout << "N is even, backtracking isn't required\n";
#endif
    std::cout << 4 << "\n";
    std::cout << 1 << " " << 1 << " " << size / 2 << "\n";
    std::cout << size / 2 + 1 << " " << 1 << " " << size / 2 << "\n";
    std::cout << 1 << " " << size / 2 + 1 << " " << size / 2 << "\n";
    std::cout << size / 2 + 1 << " " << size / 2 + 1 << " " << size / 2 << "\n";
  } else {
```

```
#ifdef DEBUG
    std::cout << "N is odd, starting backtracking...\n";
#endif
    Backtracking(size);
    std::cout << best.size() << "\n";
    for (auto &sq : best)
      std::cout << sq.x_corner + 1 << " " << sq.y_corner + 1 << " " << sq.size
                << "\n";
  }

  std::cout << "runtime = " << (clock() - t0) / 1000.0 << std::endl;
  system("pause");
  return 0;
}
```