Содержание

Data structures and alghorithms

Data structures

Dynamic arrays

Их идея очень проста, вместо статического массива с фиксированной длиной мы используем массив переменной длины. Это удобно тк не всегда удобно работать со статическим массивом, тем более когда его надо обработать. Наиболее популярный динамический массив в C++ это vector:

```
vector<int> v;
v.push_back(3); // [3]
v.push_back(2); // [3, 2]
v.push_back(5); // [3, 2, 4]
```

Вывод чисел аналогичен привычному:

```
cout << v[0] << "\n"; // 3
cout << v[1] << "\n"; // 2
cout << v[2] << "\n"; // 5
```

Также у вектора есть удобный метод size():

```
for (int i=0; i<v.size(); ++i) {
  cout << v[i] << "\n";
}
```

Но не забываем про v.at()

Но последние стандарты C++ позволяют упростить вывод чисел:

```
for (auto x : v) {
  cout << x << "\n";
}
```

Есть и другие интересные методы у vector, быть может потом напишу про них.

Stack

Он предоставляет нам две O(1) операции:

- push(): добавление элемента “наверх”
- pop(): удаление элемента “сверху”

А также одну O(1) функцию доступа:

- top(): возвращение “верхнего” элемента

Таким образом в стеке у нас есть доступ только к элементу сверху(top).

Stack = FILO (first in last out)

```
stack<int> s;
s.push(3); // {3}
s.push(2); // {2, 3}
s.push(5); // {5, 2, 3}
cout << s.top(); // 5
s.pop();
cout << s.top(); // 2
```

Подробнее о стеке можно почитать здесь.

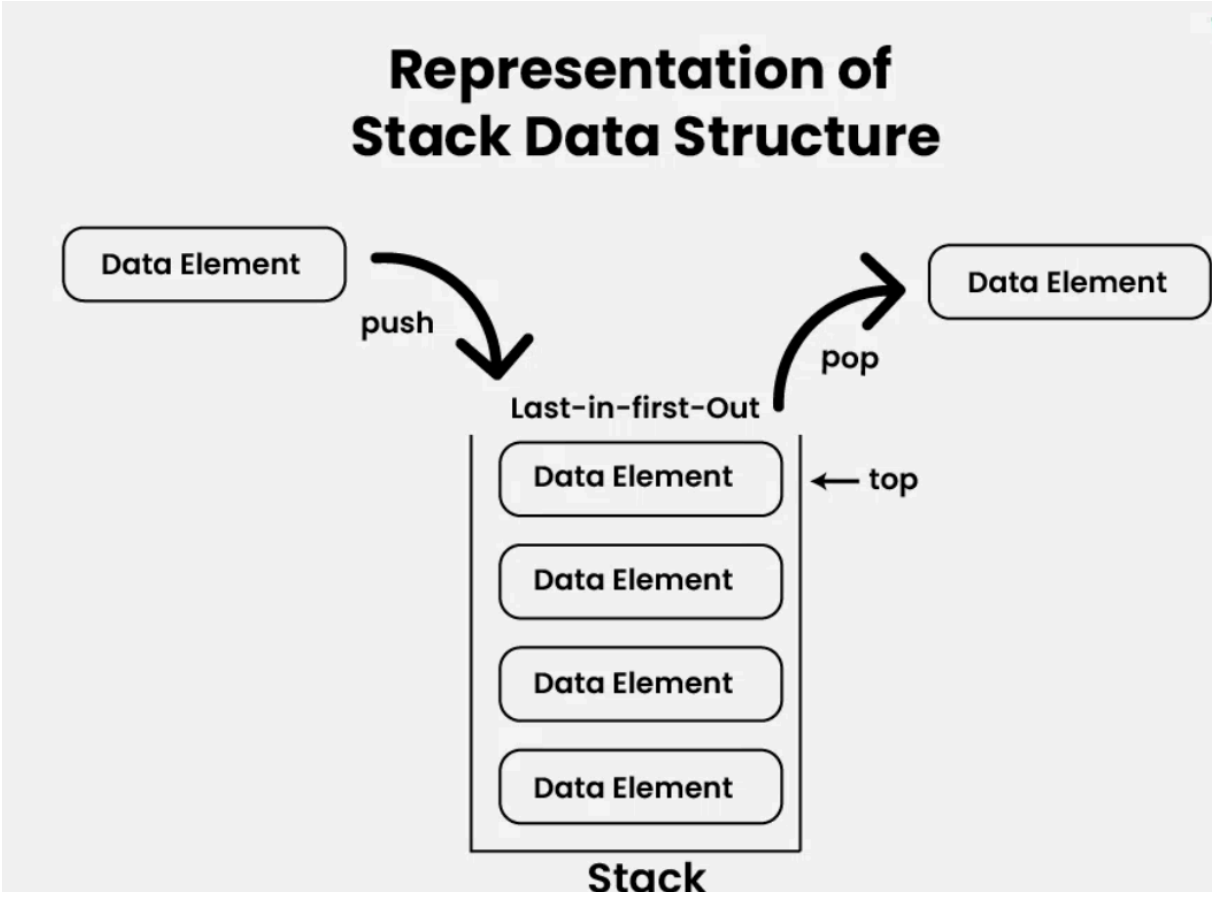


Figure 1: Схема работы стека

Queue

Очередь тоже предоставляет две O(1) операции:

- push(): добавление элемента в “конец” очереди
- pop(): удаление первого элемента из очереди

А также одну O(1) операцию доступа:

- front(): возвращает первый элемент очереди

Таким образом в очереди у нас есть доступ к первому и последнему элементам, а сама queue это FIFO(first in first out).

```
queue<int> q;
q.push(3); // {3}
q.push(2); // {3, 2}
q.push(5); // {3, 2, 5}
cout << q.front(); // 3
q.pop();
cout << q.front() // 2
```

Подробнее про очередь можно почитать здесь.

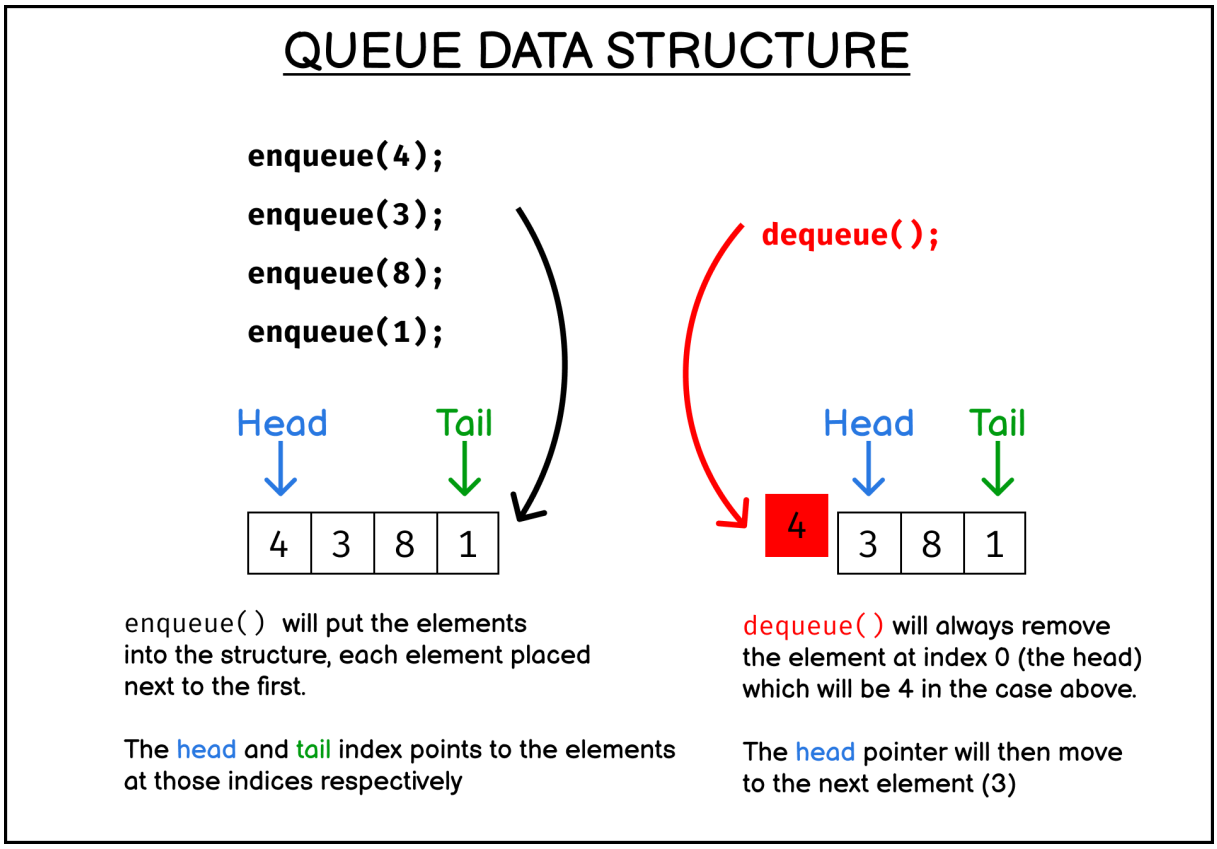


Figure 2: Схема работы очереди (да, тут другие названия, но суть та же)

Deque

это двусторонняя очередь. Она предоставляет нам следующие O(1) методы:

- push_back()
- pop_back()
- push_front()
- pop_front()

А также O(1) операции доступа:

- back()
- front()

```
deque<int> d;
d.push_back(5); // {5}
d.push_back(2); // {5, 2}
d.push_front(3); // {3, 5, 2}
d.pop_back(); // {3, 5}
d.pop_front(); // {5}
```

Внутреннее устройство сложнее чем у вектора, поэтому deque медленнее, но тем не менее O(1) есть. Подробнее о деке можно почитать здесь.

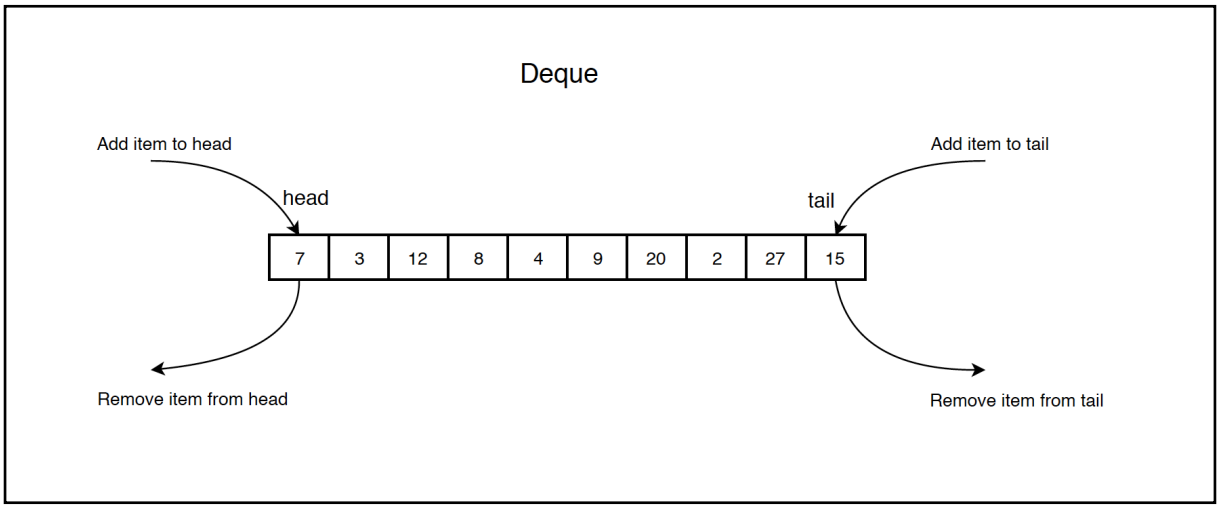


Figure 3: Схема работы дека

Sets

Множеством называется структура данных, в которой хранится набор элементов. Основные операции над множествами - вставка, поиск и удаление. Множества реализованы так, что все эти операции эффективны, что часто позволяет улучшить время работы алгоритмов.

Множества и мультимножества

В стандартной библиотеке C++ имеются две структуры, относящиеся к множествам:

- set основана на сбалансированном двоичном дереве поиска, его операции работают за время O(log n)
- unordered_set основана на хэш-таблице и работает в среднем за O(1). (да, есть крайне малая вероятность выполнения работы алгоритма за O(n))

(Позже мы обсудим, что за деревья и хэш-таблицы такие).

Пример работы из коробки:

```
set<int> s;
s.insert(3);
s.insert(2);
s.insert(5);
cout << s.count(3) << '\n'; // 1
cout << s.count(4) << '\n'; // 0
s.erase(3);
s.insert(4);
cout << s.count(3) << "\n"; // 0
cout << s.count(4) << "\n"; // 1
```

)

Идея множеств в том, что в ней нет дубликатов, только уникальные элементы. Если элемент уже есть во множестве, его невозможно добавить дважды.

```
set<int> s;
s.insert(3);
s.insert(3);
cout << s.count(3) << "\n"; // 1
```

Множества можно использовать как вектор, однако доступ к элементам с помощью оператора [] невозможен. В коде ниже выводится количество элементов, а затем эти элементы перебираются:

```
cout << s.size() << "\n";
for (auto x : s) {
  cout << x << "\n";
}
```

Функция find(x) возвращает итератор, указывающий на элемент со значением x. Если же множество не содержит x, то возвращается итератор end().

```
auto it = s.find(x);
if (it == s.end()) {
  // x не найден
}
```

Упорядоченные и неупорядоченные множества

Как можно догадаться из названия либо множество упорядочено, либо нет, но что же это значит? Если порядок неважен, берем set, иначе unordered_set:

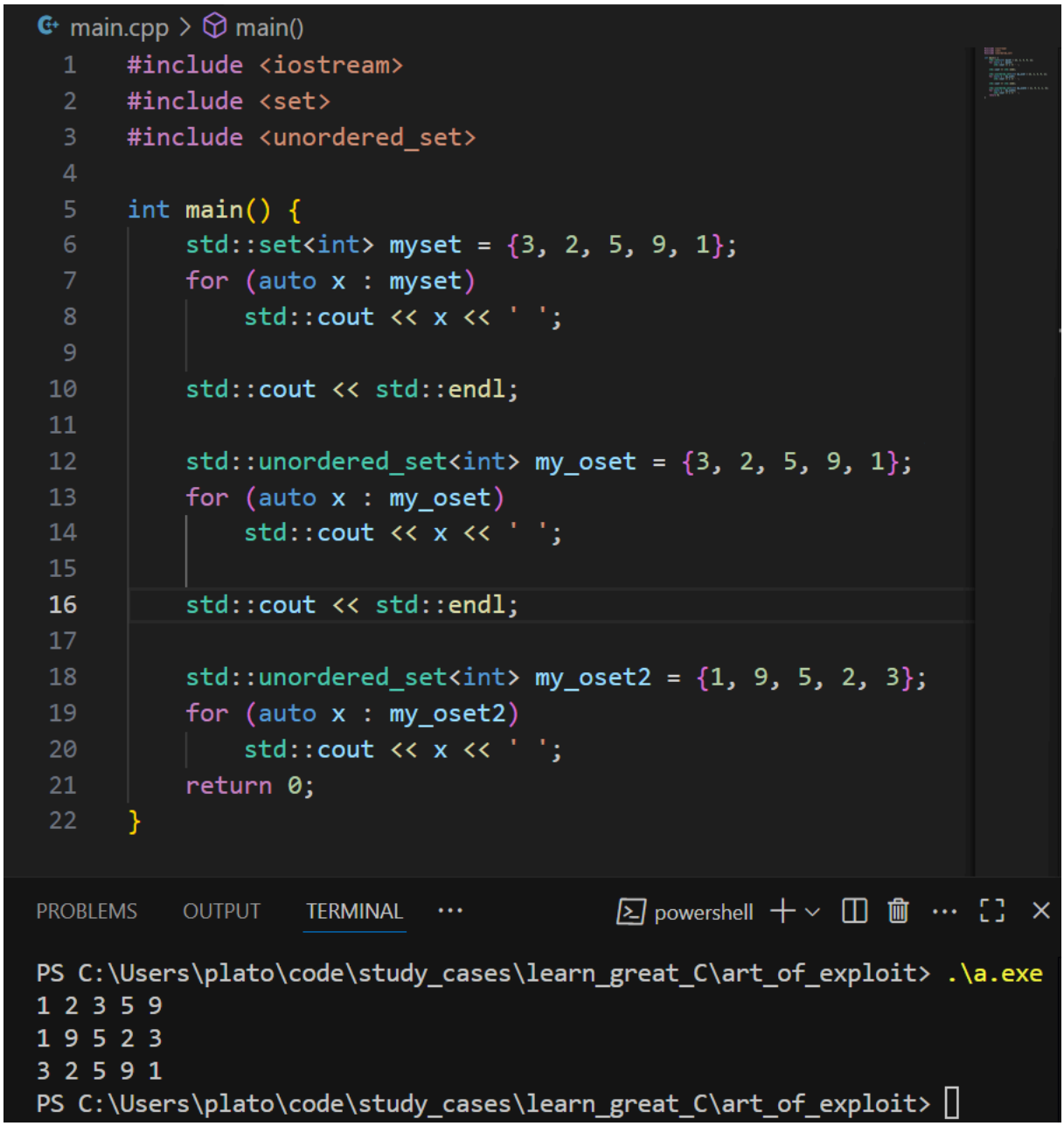


Figure 4: Пример работы set и unordered set

Multisets

Да, есть еще и мультимножества и они тоже либо упорядочены либо нет, отличие от множеств в том, что они допускают повторения.

```
multiset<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 3
```

Функция erase удаляет все копии значения из мультимножества.

```
s.erase(5);
cout << s.count(5) << "\n"; // 0
```

Если требуется удалить только одно значение, то можно поступить так:

s.erase(s.find(5)); cout << s.count(5) << “n”;

Но count и erase работают за O(k), где *k* количество подсчитываемых или удаляемых элементов

Maps

А вот и наш любимчик map(отображения), идея проста, есть ключ, тогда есть и значение(как логин и пароль, или как индекс и значение по данному индексу).

Но в map ключом может быть и строка, и число, и любой другой тип, при этом необязательно последовательный.

Устройство Отображений таково:
1. У нас есть Хэш-функция
2. Передав в Хэш-функцию ключ, мы получаем число, взяв остаток от числа(некоторого n, это либо capacity, либо size) получим номер бакета (ячейки памяти, куда в дальнейшем и будем складывать наши ключи со значениями, если бакет переполнится, то произойдет рехэширование)

В STL имеется две структуры как и в случае со множествами:
- map, основан на сбалансированном двоичном дереве со временем доступа O(log n).
- unordered_map, в основе которого лежит техника хэширования со средним временем доступа к элементам O(1).

Приступим к примерам:

```
map<string, int> m;
m["monkey"] = 4;
m["banana"] = 3;
m["harpsichord"] = 9;
cout << m["banana"] << "\n"; // 3
```

Если в map такого ключа нет, то он создастся и в него запишется заданное значение.

При простом обращении к элементу по несуществующему ключу, помимо создания ключа, будет присвоено значение по умолчанию. Например, в следующем коде в отображение добавляется ключ “aybabtu” со значением 0.

```
map<string, int> m;
cout << m["aybabtu"] << "\n"; // 0
```

)

Функция count проверяет, существует ли ключ в отображении:

```
if (m.count("aybabtu")) {
  // работаем с ключом
}
```

А так можно напечатать все ключи и значения отображения:

```
for (auto x : m) {
  cout << x.first << " " << x.second << "\n";
}
```

Priority Queue

Очередь с приоритетом это мультимножество(или множество), которое поддерживает вставку, а также извлечение и удаление минимального или максимального элемента(в зависимости от очереди). Вставка и удаление занимают O(log n), а извлечение O(1).

Очередь с приоритетом обычно основана на heap structure, а не сбалансированном двоичном дереве, что в разы проще.

По умолчанию, элементы в C++ PQ отсортированы по убыванию, и все же это реально найти и удалить наибольший элемент в очереди. Код ниже это демонстрирует:

```
priority_queue<int> q;
q.push(3);
q.push(5);
q.push(7);
q.push(2);
cout << q.top() << "\n"; // 7
q.pop();
cout << q.top() << "\n"; // 5
q.pop();;
q.push(6);
cout << q.top() << "\n"; // 6
q.pop();
```

Для того, чтобы PQ поддерживал поиск и удаление наименьшего элемента, можно сделать следующее:

```
priority_queue<int, vector<int>, greater<int>> q;
```

Algorithms

Sorting Algorithms

Нечего обсуждать, есть массив - надо отсортировать...

Bubble Sort

Пузырьковая сортировка это самый простой и популярный сортировочный алгоритм, который работает за $O(n^2)$. Логика проста: у нас есть n раундов и на каждом из них алгоритм сравнивает два последовательных элемента, если они в неправильном порядке, то их меняют местами.

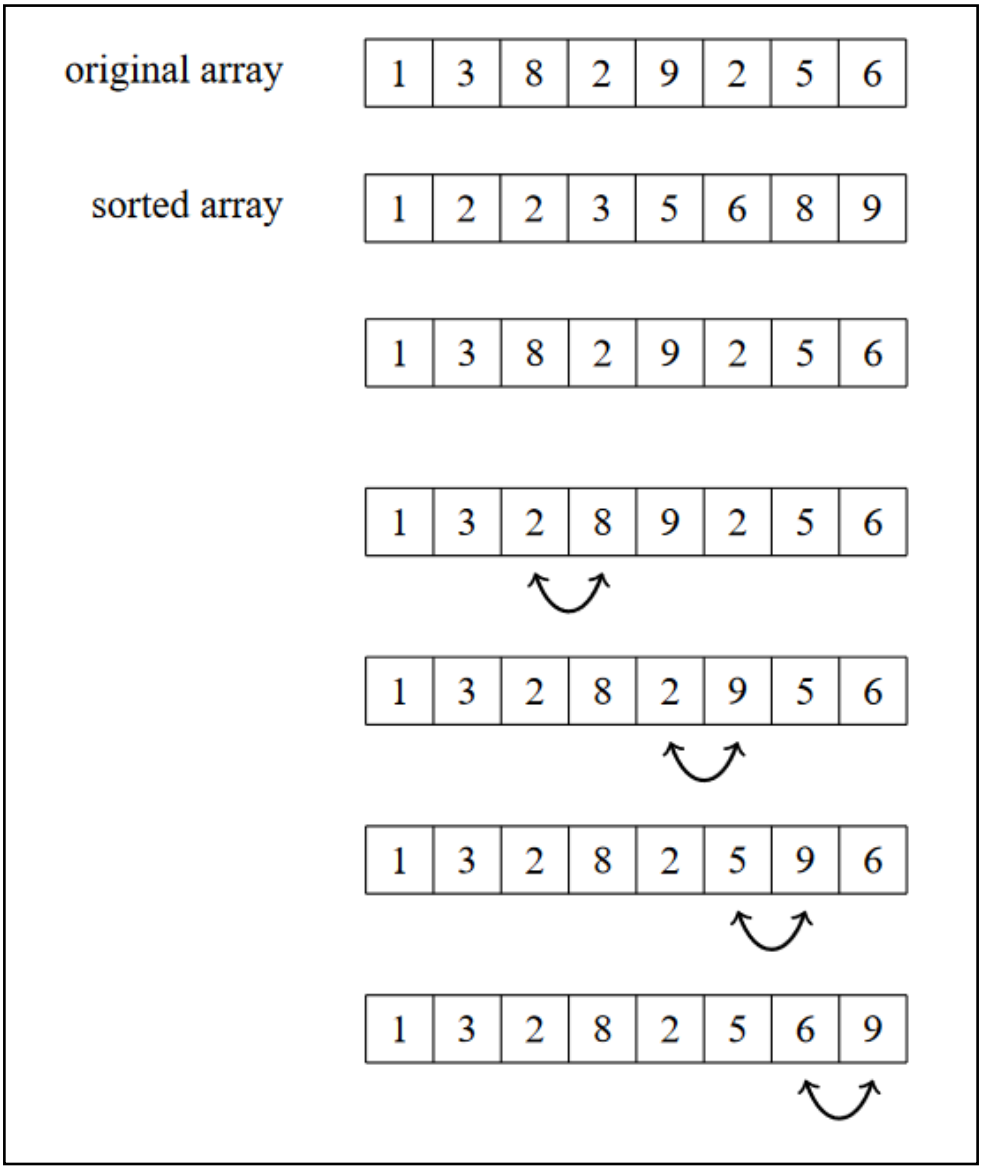


Figure 5: Схема работы Bubble Sort

Реализация

```
for (int i=0; i<n; ++i) {
  for (int j=0; j<n; ++j) {
    // по возрастанию
    if (array[j] > array[j+1]) {
      swap(array[j], array[j+1]);
    }
  }
}
```

Таким образом после первой операции самый большой элемент окажется на правильном месте, в общем говоря, после k раундов, k-наибольших элементов будут на правильных местах. Так что после n операций весь массив будет отсортирован.

Пара слов про анализ сложности

Также для анализа сложности алгоритма сортировки удобно использовать *инверсии*

$$(a, b) : a < b \text{ и } \text{array[a]} > \text{array[b]}$$

То есть такие пары чисел (вообще-то это транспозиции), у которых индексы идут по возрастанию, а сами числа не соответствуют возрастанию.

0	1	2	3	4	5	6	7
1	2	2	6	3	5	9	8

инверсии: (3, 4), (3, 5) и (6, 7)

Число инверсий показывает сколько работы нужно для сортировки массива, то есть массив отсортирован тогда, когда инверсий не осталось. С другой стороны массив чисел по убыванию содержит максимальное количество инверсий:

$$1 + 2 + ... + (n - 1) = \frac{n(n-1)}{2} = O(n^2),$$

Очевидно, что swap от последовательных элементов неправильного порядка удаляет только 1 инверсию, поэтому остальные инверсии остаются, а сложность алгоритма все также $O(n^2)$.

Insertion Sort

Разберем еще один $O(n^2)$ алгоритм - сортировка вставкой.

Алгоритм:

1. проходя по массиву начиная со 2 элемента, берем i-й элемент.
2. ищем позицию для вставки и сдвигаем элементы пока она не будет обнаружена.
3. вставляем элемент в подходящую позицию

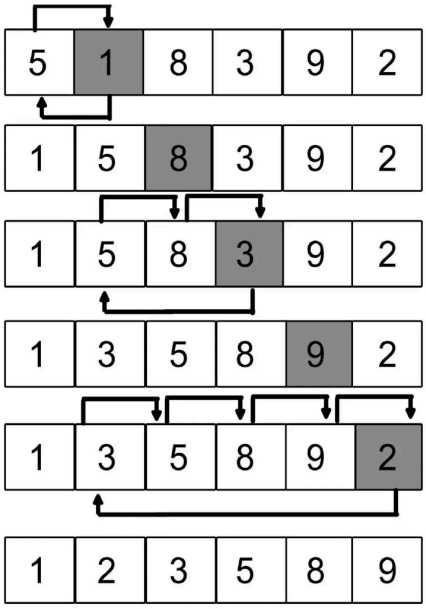


Figure 6: Схема работы Insertion Sort

Реализация:

```
void insertion_sort(int* array, int size) {
  int j, temp;
  for (int i=0; i<size; ++i) {
      temp = array[i];
      j = i-1;
      while(j >= 0 && array[j] > temp) {
          array[j+1] = array[j];
          j--;
      }
      array[j+1] = temp;
  }
}
```

Selection Sort

Сортировка выбором это грубо говоря оптимизированный пузырек. Вместо сравнения 2х соседних элементов по всему массиву, ищется минимальный от i до конечного.

Реализация алгоритма:

1. Запускаем проход по индексам массива
2. Ищем индекс минимального элемента от i до конечного
3. Меняем i-й элемент с минимальным

Таким образом каждый раз поиск минимального элемента будет в уменьшаемом диапазоне.

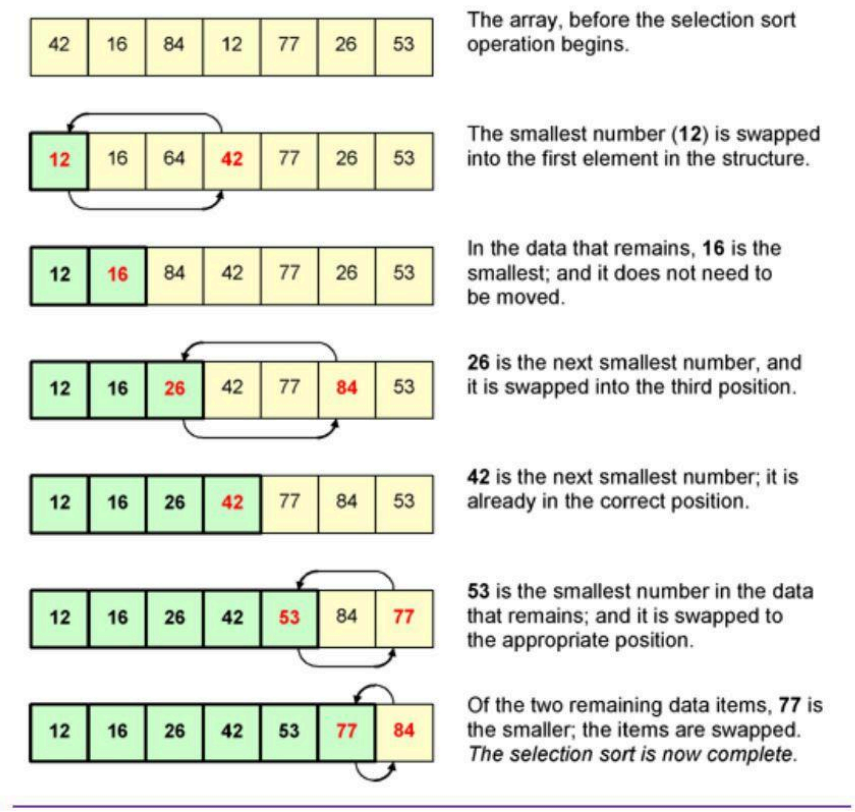


Figure 7: Схема работы Selection Sort

Реализация:

```
void selection_sort(int* array, int size) {
  for (int i=0; i<size; ++i) {
      int min = array[i];
      int ind_min = i;
      for (int j=i+1; j<size; ++j) {
          if (array[j] < min) {
              min = array[j];
              ind_min = j;
          }
      }
      array[ind_min] = array[i];
      array[i] = min;
  }
}
```

Временная сложность: $O(n^2)$

Merge Sort

Наконец-то поговорим об алгоритме из категории “Разделяй и Властвуй”. Для создания эффективного алгоритма сортировки нам недостаточно замены соседних элементов в случае bubble sort, здесь то и появляется merge sort с O(n log n) time.

Вкратце алгоритм разделяет массив на подмассивы сортирует их, а затем склеивает их сортируя вновь.

Merge sort sorts a subarray array[a…b] as follows:

1. if $a == b$, skip, cause a subarray with 1 element is already sorted.
2. Calculate the position of the middle element:

$$k = \left\lfloor \frac{a+b}{2} \right\rfloor$$

3. Recursively sort the subarray array[a…k].
4. Recursively sort the subarray array[k+1…b].
5. Merge the sorted subarrays array[a…k] and array[k+1…b] into sorted subarray [a…b].

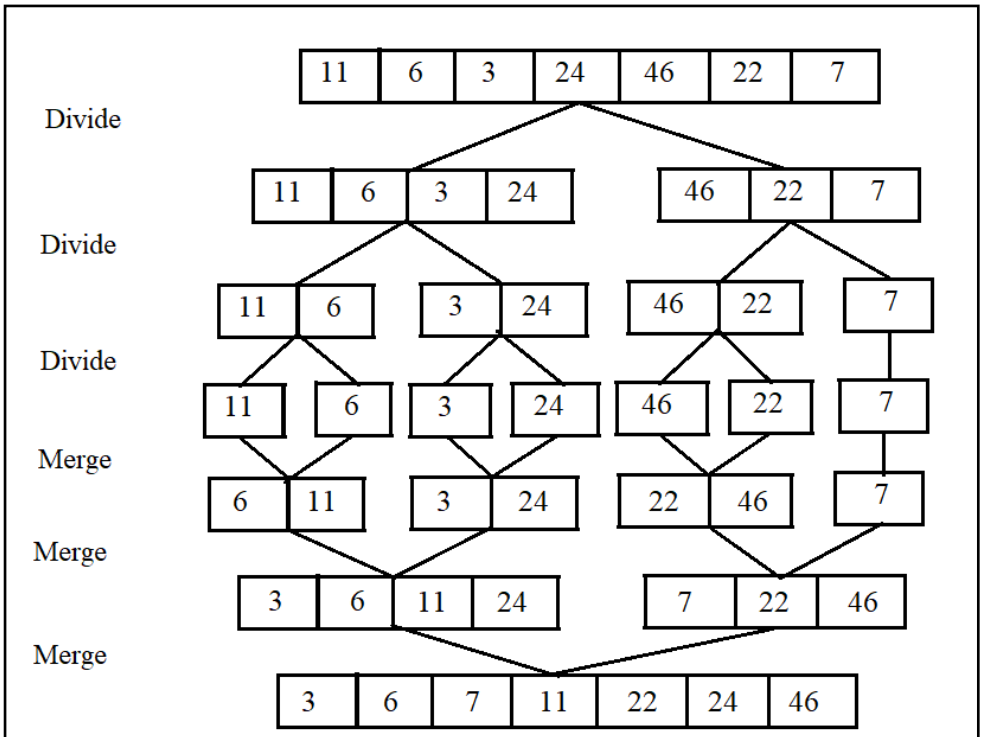


Figure 8: Схема работы Merge Sort

Реализация

```
    // thx too my old friend..
    void merge(int* ar, size_t size) {
    int* res_ar = new int[size];
    int ind_res = 0;
    int middle = size >> 1;
    int left = 0;
    int right = middle;
    // sort array while left and right are in right range
    while (left < middle && right < size) {
        // store the least of subarrays into res
        if (ar[left] <= ar[right])
            res_ar[ind_res++] = ar[left++];
        else
            res_ar[ind_res++] = ar[right++];
    }
    // if numbers remain, then sort and store them too
    while (left < middle)
        res_ar[ind_res++] = ar[left++];
    while (right < size)
        res_ar[ind_res++] = ar[right++];
    // rewrite sorted array into primary
    for (size_t i=0; i<size; ++i) {
        ar[i] = res_ar[i];
    }
    delete [] res_ar;
}
void merge_sort(int* ar, size_t size) {
    if (size <= 1)
        return;
    // calculate middle (w/ bit shift)
    int middle = size >> 1;
    // dividing array into 2 subarrays
    merge_sort(&ar[0], middle);
    merge_sort(&ar[middle], size-middle);
    // merge them
    merge(ar, size);
  }
```

Временная сложность: $O(n \log n) = O(n)O(\log n)$

Since there are O(log n) recursive levels, and processing each level takes a total of O(n) time, the algorithm works in O(n log n) time.

Sorting Lower Bound

Реально ли достичь сортировки быстрее чем $O(n \log n)$? Это невозможно, если сортировка основана на сравнении элементов.

Нижняя граница временной сложности может быть доказана рассмотрением сортировки как процесса, где каждое сравнение двух элементов дает больше информации о содержимом массива:

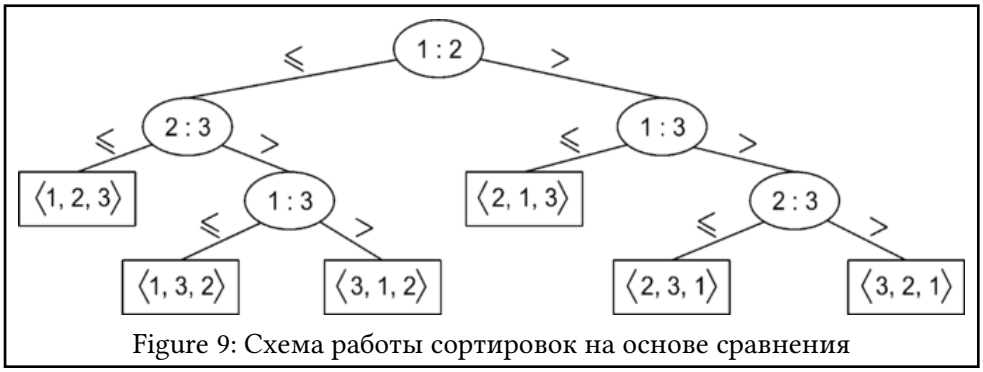


Figure 9: Схема работы сортировок на основе сравнения

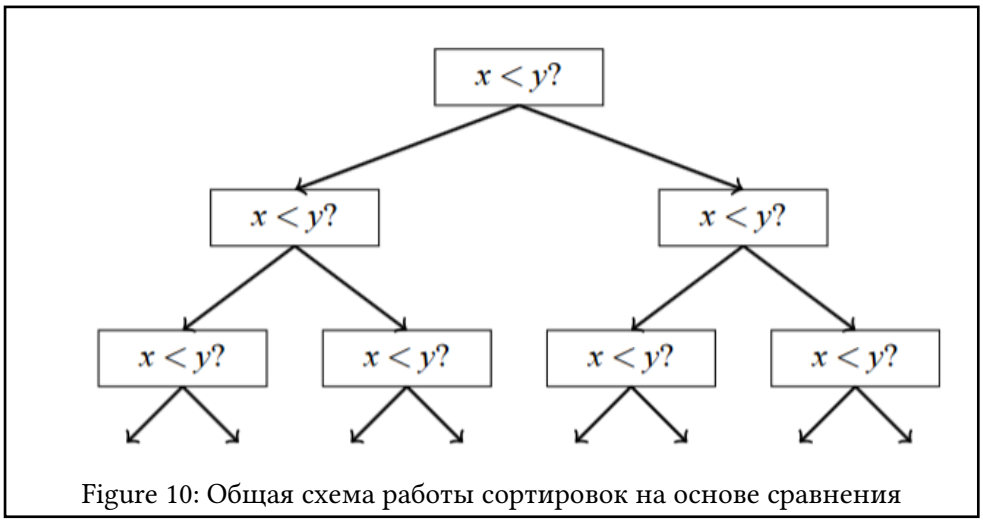


Figure 10: Общая схема работы сортировок на основе сравнения

"$x < y$?" означает сравнение некоторых элементов x и y, верное утверждение отправляет в левую ветку, иначе в правую.

Значит у каждого узла именно 2 потомка(тоже узла). Т.к. на вход мы получаем массив из n элементов, то количество возможных массивов равно кол-ву его различных перестановок - n!, а тогда количество листьев дерева не менее n!

(в противном случае некоторые перестановки были бы не достижимы из корня, а значит, алгоритм неправильно работал бы на некоторых исходных данных).

Т.к двоичное дерево высоты(глубины) h имеет не более 2^h листьев имеем:

$$n! \leq l \leq 2^h,$$

где l число листьев. Тогда получим, что высота дерева не менее:

$$\log_2(n!) = \log_2(1) + \log_2(2) + ... + \log_2(n).$$

Несложно доказать оценку:

$$\log_2(n!) \geq \left(\frac{n}{2}\right) *$$

И наконец получаем:

$$\frac{n}{2} * \log_2\left(\frac{n}{2}\right) = \frac{n}{2}(\log_2(n) - 1) = \Omega(n \log(n))$$

Доп. источник

Counting Sort

Но нижняя оценка временной сложности не подходит для сортировок, не основанных на сравнениях. Сортировка подсчетом основана на подсчете элементов и работает за $O(m)$, где m - диапазон чисел.

Рассмотрим первое приближение алгоритма, когда числа строго неотрицательные.

Логика алгоритма:

1. инициализируется массив с max+1 количеством нулей, где max максимальный элемент массива.
2. затем начинается проход по исходному массиву с подсчетом включений:

counting_array[array[i]] += 1;

3. Затем сортируем исходный массив:

array[index++] = i, повторяя counting_array[array[i]] раз, чтобы учесть повторения.

Реализация:

```
void counting_sort(int* array, size_t size) {
  int max = array[get_ind_max(array, size)];
  int counting_array[max+1] = {0};
  for (int i=0; i<size; ++i) {
      counting_array[array[i]] += 1;
  }
  int index = 0;
  for (int i=0; i < max; ++i) {
      for (int j=0; j<counting_array[i]; ++j) {
          array[index++] = i;
      }
  }
}
```

Но можно улучшить алгоритм, чтобы он учитывал и отрицательные числа:

```
void counting_sort2(int* array, size_t size) {
  int max = array[get_ind_max(array, size)];
  int min = array[get_ind_min(array, size)];
  int counting_array[max-min+1] = {0};
  for (int i=0; i<size; ++i) {
      counting_array[array[i]-min] += 1;
  }
  int index = 0;
  for (int i=min; i < max; ++i) {
      for (int j=0; j<counting_array[i-min]; ++j) {
          array[index++] = i;
      }
  }
}
```

Теоретически можно сделать алгоритм рабочим не только для целых чисел, но для этого нужно использовать hashmap.(но это кал какой-то)

Quick Sort

Логика алгоритма:

Реализация:

☐

Complete Search

Generating Subsets

Возьмем множество {0, 1, 2} как же создать все его подмножества?

Перейдем к алгоритмам.

Method 1

Весьма интересный способ - это использование рекурсии:

```
void search(int k) {
  if (k == n) {
    // process subset
  } else {
    search(k+1);
    subset.push_back(k);
    search(k+1);
    subset.pop_back();
  }
}
```

Алгоритм на словах: начинаем с 0, затем если n не конечный, запускаем два серча, один без добавления в подмножество, другой с добавлением в итоге получим все подмножества:

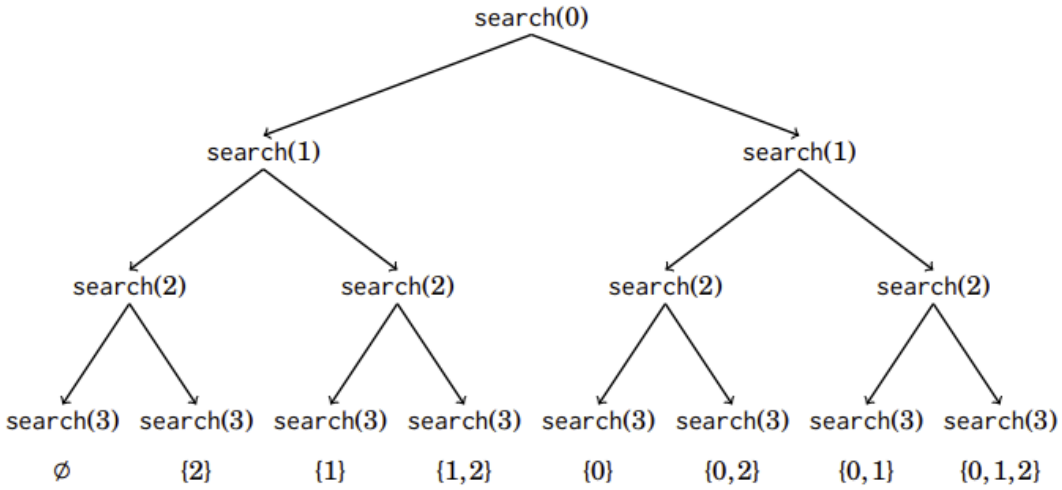


Figure 11: схема алгоритма

Method 2:

Следующий метод заключается, в битовом представлении целого числа. Каждое подмножество можно представить как множество n битов числа от 0 до 2^n-1. Например 11001 - {0, 3, 4}.

```
for (int b=0; b<(1<<n); b++) {
  vector<int> subset;
  for (int i=0; i<n; i++) {
    if (b&(1<<i))
      subset.push_back(i);
  }
}
```

Generating Permutations

Next step is making permutations. For example the permutations for {0, 1, 2} are {0, 1, 2}, {0, 2, 1}, {1, 0, 2}, {1, 2, 0}, {2, 0, 1} and {2, 1, 0}.

Method 1

Используя рекурсию, можно сгенерировать перестановки.

```
vector<int> permutation;

void search() {
  if (permutation.size() == n) {
    // process permutation
  } else {
    for (int i=0; i<n; ++i) {
      if (chosen[i]) continue;
      chosen[i]=true;
      permutation.push_back(i);
      search();
      chosen[i]=false;
      permutation.pop_back();
    }
  }
}
```

Каждый вызов функции добавляет новый элемент в перестановку, учитывая те, которые уже созданы.

Method 2

Также есть метод из STL:

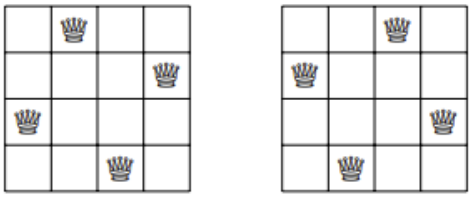
```
vector<int> permutation;
for (int i=0; i<n; ++i) {
  permutation.push_back(i);
}
do {
  // process permutation
} while (next_permutation(permutation.begin(), permutation.end()));
```

Backtracking

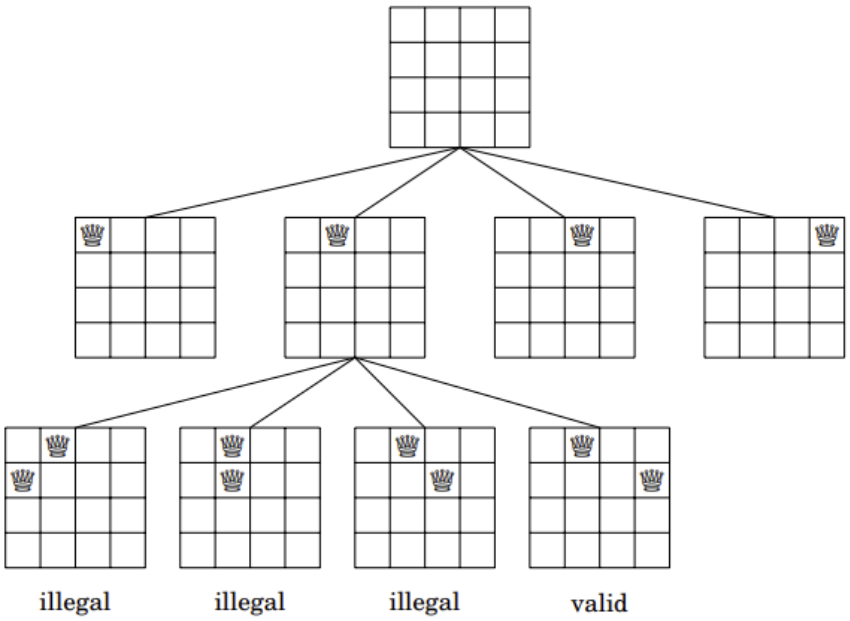
Данных подход начинает с 0, а затем прибавляет в решения шаг за шагом. Рекурсивный перебор(поиск) идет через всевозможные различные способы построения решения.

В качестве примера рассмотрим задачу вычисления способов расстановки n дамок на доске n*n так, чтобы они не могли атаковать друг друга.

Для 4 существует только 2 решения:



Задача может быть решена этим способом, добавляя по дамке в каждый из столбцов ряд за рядом, неподходящие способы будем убирать.

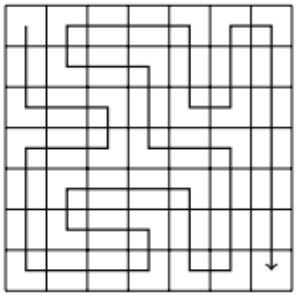


Реализация:

```
void search(int y) {
  if (y == n) {
    count++;
    return;
  }
  for (int x=0; x<n; x++) {
    if (column[x] || diag1[x+y] || diag2[x-ly+n-1]) continue;
    column[x] = diag1[x+y] = diag2[x-y+n-1];
    search(y+1);
    column[x] = diag1[x+y] = diag2[x-y+n-1] = 0;
  }
}
```

Pruning the search

Рассмотрим задачу, в которой нужно посчитать всевозможные пути в клетке n*n клеток с верхнего левого в правый нижний углы, посетив каждую клетку 1 раз. Например, для 7*7 есть 111712 путей. Оставим сетку 7*7.



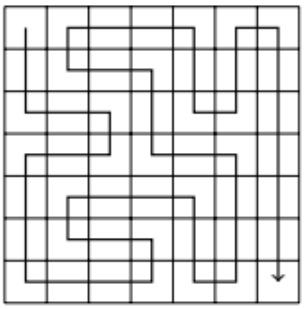
Basic algorithm

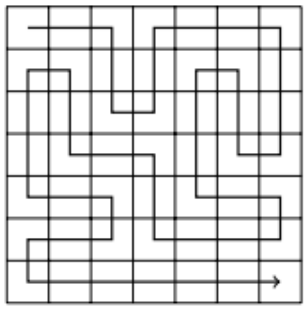
Ничего сложного, просто берем backtrack и все.

- время работы: 483 секунды
- количество рекурсивных вызовов: 76 миллиардов

Optimization 1

В каждом решении мы сначала идем либо вправо, либо вниз, а значит всегда есть два пути симметричных друг другу относительно диагонали. То есть, мы имеем право выбрать куда идти на первой итерации и умножить на 2 полученный результат.

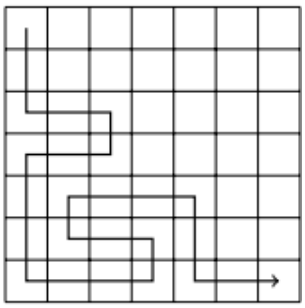




- running time: 244 секунды
- количество рекурсивных вызовов: 38 млрд

Optimization 2

Если путь достигает нижнего правого угла, не посетив все остальные клетки, то очевидно что дальше рассчитывать ничего не нужно.

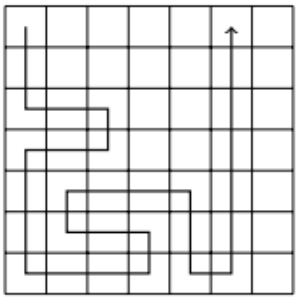


- running time: 119 секунд

- количество рекурсивных вызовов: 20 млрд

Optimization 3

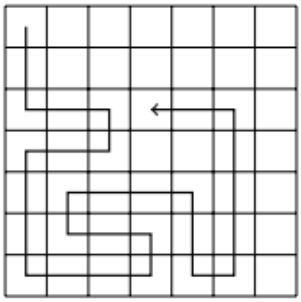
Модернизируем предыдущий способ, если “змея” достигла стены и может повернуть и вправо, и влево, то заполнить все клетки невозможно.



- running time: 1.8 секунд
- количество рекурсивных вызовов: 221 миллион

Optimization 4

А идея прошлой оптимизации в свою очередь может быть модернизирована еще больше, например, если “змея” может повернуть либо вправо, либо влево но не может идти прямо, то достичь всех клеток невозможно.



- running time: 0.6 секунд
- количество рекурсивных вызовов: 69 млн

Вдумайтесь было: 483 секунды, 76 млрд стало: 0.6 секунд, 69 млн выигрыш: 805, 1101,45

В общем суть вы уловили, надо срубать самые корневые ответвления и наш алгоритм будет бесподобен.

Meet in the middle

Берем задачу, делим на 2 меньшие равные подзадачи, решаем их, а затем объединяем результат. “Разделяй и властвуй” во всей красе. Подобное есть в merge sort и binary search.

Gready algorithms

Идея проста, на каждой итерации делаем наиболее выгодное действие, но тогда мы имеем шанс потерять большую выгоду, спрятанную за невыгодными сделками.

Coin Problem

Задача проста, у вас есть монетки, ваша задача собрать монетки так, чтобы получить заданное число, за минимальное количество монеток:

$\{1, 2, 5, 10, 20, 50, 100, 200\}, n = 520 : n = 200 * 2 + 100 + 20.$

So, basic algorithm is taking most largest possible coin on each step. Но он верен не всегда, например возьмем n=6, а монеты {1, 3, 4} жадный алгоритм возьмет сначала 4, а потом возьмет по 1, хотя правильнее дважды взять 3.

Так, что важно понимать что какие-то задачи Жадный алгоритм способен решить, а какие-то нет. Сам по себе он не оптимален.

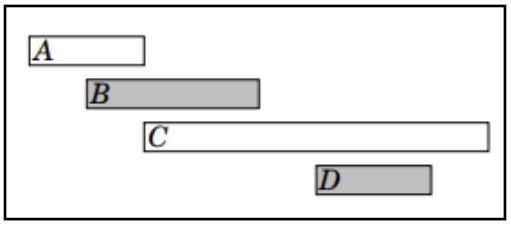
Scheduling

Большинство задач с расписанием, могут быть решены жадным алгоритмом. Классическая задача такова:

Дано n событий, с их временами их начала и конца, составьте расписание, чтобы можно было включить как можно больше событий.

event	starting time	ending time
A	1	3
B	2	5
C	3	9
D	6	8

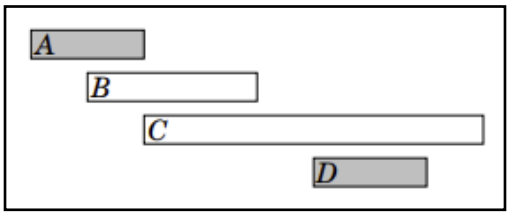
В данном случае ответ - 2, например B и D:



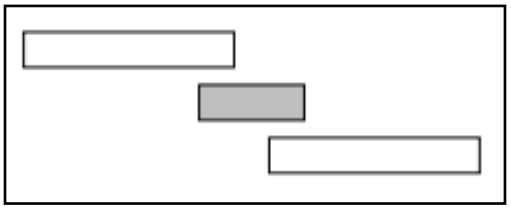
Реально ли составить жадный алгоритм, чтобы он решал эту задачу при любых исходных?

Algorithm 1

Первая идея это выбирать наиболее короткие события из возможных.



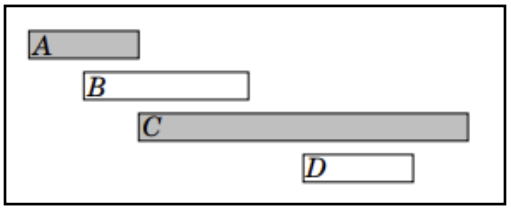
Однако, данная стратегия не может быть верной всегда, тк надо учитывать еще и время начала события:



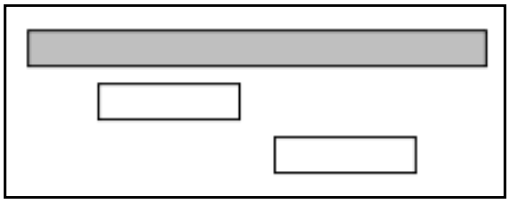
Тут же и вовсе можно уместить два события, а не одно))

Algorithm 2

Можно выбирать следующее событие ближайший после текущего события.

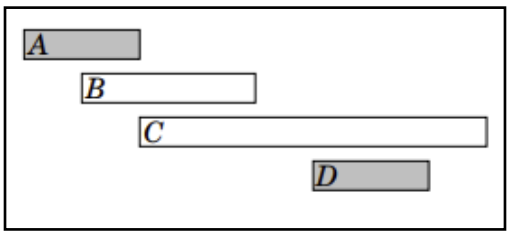


Даже выбирая ближайшие короткие, мы столкнемся с проблемой:



Algorithm 3

Теперь идея заключается в том, чтобы выбирать те ближайшие события, которые заканчиваются как можно быстрее:



Алгоритм является наиболее оптимальным потому что решены все проблемы задачи: • найдено ближайшее событие • при этом оно заканчивается раньше остальных

Tasks and deadlines

Рассмотрим задачу, в которой даны n задач с продолжительностью и дедлайном. За каждую задачу получаешь d-x очков где d - дедлайн, а x - время, когда таска закончилась, нужно определить расписание с наибольшим счетом.

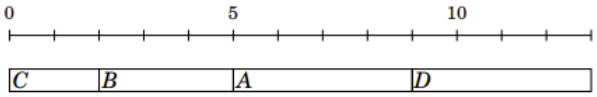
В данной задаче, правильное расписание выглядит так:

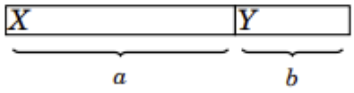
task	duration	deadline
A	4	2
B	3	5
C	2	7
D	4	5

$$\mathrm{C} : 5$$
$$D : 0$$
$$A : -7$$
$$D : -8$$

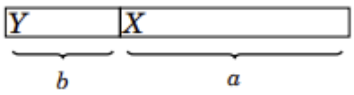
Удивительно, что оптимальное решение не зависит от дедлайна в целом, но верный жадный алгоритм заключается в сортировке таск по их длительности по возрастанию.

Дело в том, что даже если две задачи будут идти друг за другом и при этом первая таска длится дольше второй, можно добиться лучшего решения поменяв их местами.





Here $a > b$, so we should swap the tasks:



Minimizing sums

Даны n чисел, нужно найти такое x, которое минимизирует сумму:

$$|a_1 - x|^c + |a_2 - x|^c + ... + |a_n - x|^c$$

Мы фокусируемся на кейсах c=1, c=2.

Case c=1

Например, для чисел {1, 2, 9, 2, 6} лучшее решение это x=2:

$$|1 - 2| + |2 - 2| + |9 - 2| + |2 - 2| + |6 - 2| = 12$$

В общем, медиана чисел и решает задачу.

Case c=2

Теперь все сложнее:

$$|a_1 - x|^2 + |a_2 - x|^2 + ... + |a_n - x|^2$$

для прошлых чисел ответ 4, получен как среднее арифметическое всех чисел.

$$(1 - 4)^2 + (2 - 4)^2 + (9 - 4)^2 + (2 - 4)^2 + (6 - 4)^2 = 46$$

В общем случае для $c = 2$, лучшее решение это среднее арифметическое:

$$nx^2 - 2x(a_1 + a_2 + ... + a_n) + (a_1^2 + a_2^2 + ... + a_n^2)$$

Причем последнее слагаемое не зависит от x, значит можем отбросить ее. То что осталось формирует функцию: $nx^2 - 2xs$, где $s = a_1 + a_2 + ... + a_n$ Это парабола с ветвями вверх с корнями $x = 0$ и $x = \frac{2s}{n}$, тогда $x_{\min} = \frac{s}{n}$, что и является средним арифметическим.

Data compression

Каждому символу дан бинарный код, нам дана строка, задача сжать строку до минимального количества бит. (если Фано знаете то можете не читать)

character	codeword
A	00
B	01
C	10
D	11

A строка: AABACDACA результат: 000001001011001000 (18 bit)

Для сжатия строки наиболее частым буквам присвоим коды, наименьшей длины.

character	codeword
A	0
B	110
C	10
D	111

результат: 001100101110100 (15 bit)

С экономили 3 бита, но для нашей задачи надо что-бы ни один из кодов не был префиксом другого.

character	codeword
A	10
B	11
C	1011
D	111

Figure 31: пример неподходящих кодов

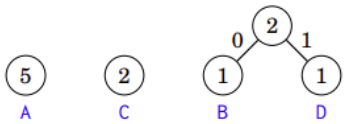
Huffman coding

Кодировка Хаффмана - это жадный алгоритм, который возвращает оптимальный код для сжатия данной строки.

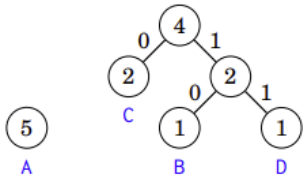
Алгоритм строит бинарное дерево, основанное на частоте чисел в строке, и код каждого символа может быть “прочтен” по пути.

Пример для AABACDACA:

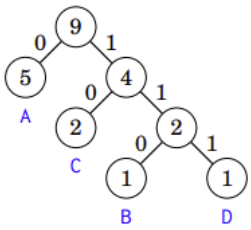
Взяли частоту появления букв и каждой ноде присвоили букву + частоту.



Берем и суммируем ноды по возрастанию: 1 с 1, 2 с 2 и тд



Суммируем дальше



Просуммировали, а теперь смотрим какая нода где, та что слева(или та что как бы больше по частоте), а справа ноды меньше.

character	codeword
A	0
B	110
C	10
D	111

Составляем дерево Хаффмана.

Overall, строим бинарное дерево начиная с листочков, каждые два листочка или два корня или корень и листочек образуют корень, частота которого равна сумме частот.

Dynamic Programming

Динамическое программирование это техника, которая комбинирует в себе корректность полного перебора и эффективность жадных алгоритмов. Динамическое программирование может быть использовано если задача может быть разделена “наложением” подзадач, которые могут быть решены независимо от остальных.

Coin Problem

Дано множество монет: coins = {c1, c2, c3, … ck} и цель собрать заданный номинал за минимальное количество монет.

Жадный алгоритм как мы помним справлялся отлично, но только в редких случаях, например в монетах евро стандарта. Поэтому жадный алгоритм не создавал оптимальное решение.

Теперь же самое время решить задачу эффективно, используя динамическое программирование. Алгоритм динамического программирования основан на рекурсивной функции, которая идет через всевозможные варианты как собрать эту сумму(как с брутфорсом). Но эффективность этого алгоритма в мемоизации и просчитывании ответа на каждую подзадачу только ОДИН раз.

Recursive formulation

Идея динамического программирования это сформулировать задачу рекурсивно, что позволит решить задачу, решив маленькие подзадачи(разделяй и властвуй). В случае нашей задачи, рекурсивная формулировка такова: какое наименьшее количество монет необходимо для данной суммы.

Возьмем монеты coins = {1, 3, 4}, первые значения функции таковы:

```
solve(0) = 0
solve(1) = 1
solve(2) = 2
solve(3) = 1
solve(4) = 1
solve(5) = 2
solve(6) = 2
solve(7) = 2
solve(8) = 2
solve(9) = 3
solve(10) = 3
```

Как вы могли догадаться функция в общем имеет вид:

$$\text{solve(x)} = \begin{cases} \infty & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ \min_{c \in \text{ coins}} \text{solve(x-c)} + 1 & \text{if } x > 0 \end{cases}$$

Если x < 0, то решить задачу невозможно, если же x=0, 0 монет будет достаточно, а для x>0 применяем рекурсию и находим минимальное количество монет.

```
int solve(int x) {
  if (x < 0) return INF;
  if (x == 0) return 0;
  int best = INF;
  for (auto c : coins) {
    best = min(best, solve(x-c)+1);
  }
  return best;
}
```

Но эта функция все еще не эффективна.

Using memoization

Идея динамического программирования это использование *мемоизации* для эффективного вычисления значений рекурсивной функции. Это и значит, что значения функции хранятся в массиве после их вычисления.

```
bool ready[N];
int value[N];

int solve(int x) return INF:
  if (x == 0) return 0;
  if (ready[x]) return value[x];
  int best = INF;
  for (auto c : coins) {
    best = min(best, solve(x-c)+1);
  }
  value[x] = best;
  ready[x] = true;
  return best;
```

Асимптотическая сложность: O(nk), n-target sum, k-coins. Но при всем этом мы могли написать код так(не уч отриц значения):

```
value[0] = 0;
for (int x=1; x<=n; x++) {
  value[x] = INF;
  for (auto c : coins) {
    if (x-c >= 0) {
      value[x] = min(value[x], value[x-c]+1);
    }
  }
}
```

На деле, большинство соревновательных программистов предпочитают именно этот код, тк он меньше и имеет ниже константный фактор(че). Но думать о алгоритмах динамического программирования проще в рекурсии.

Constructing a solution

В предыдущих главах мы уже описывали процесс создания оптимального решения. В случае с coin problem, мы можем объявить другой массив, который для каждой суммы монет хранит первую монету в оптимальном решении.

```
int first[N];
value[0] = 0;
for (int x = 1; x <= n; x++) {
  value[x] = INF;
  for (auto c : coins) {
    if (x-c >= 0 && value[x-c]+1 < value[x]) {
      value[x] = value[x-c]+1;
      first[x] = c;
    }
  }
}
```

Теперь код может быть использован для вывода оптимального решения для заданного n.

Counting the number of solutions

Let us now consider another version of the coin problem where our task is to calculate the total number of ways to produce a sum x using the coins. For example, if coins = {1,3,4} and x=5, there are a total of 6 ways:

1+1+1+1+1	3+1+1
1+1+3	1+4
1+3+1	4+1

Снова мы можем решить задачу рекурсивно. Пусть solve(x) обозначает число способов всех получения заданной суммы x. Прошлый пример уже показывал все варианты, а само решение ничуть не ново:

$$\text{solve(x)} = \text{solve(x-1)} + \text{solve(x-3)} + \text{solve(x-4)}$$

Ну, а теперь доведем до ума решение, чтобы оно работало всегда.

$$\text{solve(x)} = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x = 0 \\ \sum_{c \in \text{ coins}} \text{solve(x-c)} & \text{if } x > 0 \end{cases}$$

Объяснять нечего, сами все видите, теперь код:

Забегая вперед, что-то похожее мы увидим в задаче о рюкзаке. Часто число решений столь велико, что не требуется высчитывать конкретное число, но при этом достаточно дать ответ по модулю m, где, например, $m = 10^9 + 7$. Этого можно достичь следующим образом:

Сейчас все идеи динамического программирования были обсуждены, приступим к демонстрации на задачах, чтобы увидеть возможности ДП.