Державний університет «Одеська Політехніка»

Інститут комп’ютерних систем

Кафедра інформаційних систем

Розрахунково-графічна робота

з дисципліни «Теорія алгоритмів»

Тема: «Алгоритм пошуку простих чисел»

Виконав:

студент групи АІ-222

Кулібаба Сергій

Прийняли:

Арсірій О.О.

Одеса 2023

**Завдання:** Розробіть та реалізуйте модифікацію сортування купою за алгоритмом Jason Morrison (J сортування), порівняйте його з класичним сортуванням купою. Покажіть переваги для майже відсортованих масивів **Мета:** Розглянути різні алгоритми, та виявити більш оптимальний з них.

**Опис вхідних та вихідних даних:**

Вхідні дані:

Масив : [3, 15, 11, 6, 9, 14, 10, 12, 1, 7, 8, 2, 13, 4, 5]

Вихідні дані:

Відсортований масив: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

**Розробка та опис загальної схеми алгоритму:**

Перед тим як починати опис алгоритму з мого завдання, зазначу, що задля якісного пояснення суті його роботи, потрібно також згадати його алгоритм-«прородитель». Так і зроблю:

1. **Сортування купою**

Це сортування також називають пірамідальним. Розглянемо принципи його роботи:

1. Створюється бінарна купа з масиву, де кожен батьківський елемент більший за його дітей.

2. Елементи масиву перебудовуються в сортуюче дерево, щоб у корені з'явився найбільший елемент.

3. Найбільший елемент міняється місцями з останнім елементом несортованої частини масиву.

4. Сортуюче дерево перебудовується для несортованої частини, знову "піднімаючи" найбільший елемент.

5. Процедура повторюється, поки всі елементи не будуть відсортовані

Псевдокод алгоритму:

function main:

for arr.length / 2 - 1 to 0:

heapify(arr, i, arr.length);

maxHeap(arr);

for arr.length - 1 to 1:

swap(arr, i, 0);

heapify(arr, 0, i);

function heapify(int[] arr, int i, int j):

int left = i \* 2 + 1;

int right = i \* 2 + 2;

int main = i;

if left < j && arr[left] > arr[main] -> main = left;

if right < j && arr[right] > arr[main] -> main = right;

if i != main:

swap(arr, i, main);

heapify(arr, main, j);

function swap(int[] arr, int first, int second):

int temp = arr[first];

arr[first] = arr[second];

arr[second] = temp;

Моделювання:

1. main:

* Спочатку в циклі for здійснюється виклик функції heapify для побудови максимальної купи (maxHeap). Цей цикл йде з половини масиву мінус один до нуля, що відповідає останнім батьківським вузлам у купі.
* Після створення максимальної купи виконується цикл з кінця масиву до його початку (не включаючи нульовий елемент). У цьому циклі відбувається обмін елементів (за допомогою swap), після чого викликається heapify, щоб відновити властивість максимальної купи для частини масиву, що залишилася.

2. heapify:

* Ця функція приймає масив arr, індекс поточного вузла i та розмір купи j. Вона порівнює поточний вузол з його дітьми (якщо вони існують) і, якщо якийсь із дітей більше, змінює їх місцями.
* Якщо відбувається обмін, рекурсивно викликається heapify для піддерева з коренем у тому вузлі, куди було переміщено вихідний вузол.

3. swap:

* Проста функція обміну, яка змінює місцями два елементи масиву.

Після виконання цього алгоритму, потрібно перейти вже до j-sort (так званий алгоритм Джейсона Моррисона). Розглянемо як він працює. Теоретично:

1. **J-sort**

Основні кроки J-сортування можна описати так:

1. Побудова купи

2. Створення нисходящої купи.

3. Створюємо «зеркальну» купу.

4. Робимо купу сортуючую та нисходящою.

5. Дороблюємо сортування вставками, якщо потрібно

Псевдокод алгоритму:

procedure jHeapSort(arr):

n := length(arr)

for i from 1 to n - 1:

current := arr[i]

j := i - 1

while j >= 0 and arr[j] < current:

arr[j + 1] := arr[j]

j := j - 1

arr[j + 1] := current

for i from n - 1 down to 1:

swap(arr[0], arr[i])

heapify(arr, i, 0)

procedure heapify(arr, n, i):

largest := i

left := 2 \* i + 1

right := 2 \* i + 2

if left < n and arr[left] > arr[largest]:

largest := left

if right < n and arr[right] > arr[largest]:

largest := right

if largest ≠ i:

swap(arr[i], arr[largest])

heapify(arr, n, largest)

arr := [3, 15, 11, 6, 9, 14, 10, 12, 1, 7, 8, 2, 13, 4, 5]

jHeapSort(arr)

print(arr)

Моделювання:

1. Для сортування масиву arr, спершу визначимо його довжину та збережемо її у змінну n.
2. Виконуємо перший цикл for, який проходить через елементи масиву від 1 до n-1. На кожній ітерації цього циклу вибираємо поточний елемент current і порівнюємо його з попередніми елементами масиву. Знаходимо правильне місце для current в відсортованій частині масиву, вставляючи його на відповідну позицію. Для цього використовуємо цикл while, який порівнює current з попередніми елементами і зсуває їх, якщо вони менше current. Коли знайдене правильне місце для current, вставляємо його в цей індекс.
3. Після виконання першого циклу масив міститиме відсортовану частину, але в навпаки відсортованому порядку. Тому виконуємо другий цикл for, який проходить через масив від n-1 до 1 і обмінює перший елемент (найбільший) з останнім елементом масиву. Після обміну викликаємо функцію heapify, щоб відновити властивість бінарної купи для залишкової частини масиву.
4. Функція heapify визначає найбільший елемент серед поточного вузла та його двох дітей (лівого та правого), якщо вони існують. Якщо найбільший елемент не є поточним вузлом, то обмінюємо їх і рекурсивно викликаємо heapify для дітей.

Цей алгоритм буде повторюватися, доки масив повністю не відсортується, і на кожній ітерації він витягне найбільший елемент та додасть його до відсортованої частини масиву, зменшуючи розмір купи і відновлюючи її властивість.

Після виконання алгоритму Джейсона Моррисона, може виникнути ситуація, коли потрібно досортувати масив, використовуючи сортування вставками. У випадку вище наведеного масиву, це не потрібно, але задля повного опису алгоритму j-sort, потрібно ще й розглянути сортування вставками. Теоретичний опис надано нижче:

1. **Сортування вставками**

Сортування вставками - це один із простих алгоритмів сортування, який використовується для упорядкування масиву елементів. Основна ідея цього алгоритму полягає в тому, щоб послідовно вставляти кожен елемент масиву на відповідне місце в уже відсортованій частині масиву. Нижче наведено основні кроки сортування вставками:

1. Починаємо з першого елемента масиву, вважаючи його вже відсортованим.

2. Вибираємо наступний елемент (перший невідсортований елемент) і вставляємо його на відповідне місце в вже відсортованій частині масиву.

3. Повторюємо цей процес для всіх елементів масиву, починаючи з другого і закінчуючи останнім.

4. Для вставки елемента він порівнюється з кожним елементом відсортованої частини масиву, поки не знайдете місце, де він має бути вставлений. При цьому всі більші за нього елементи зсуваються вправо, щоб зробити місце для вставки.

5. Повторюємо цей процес для всіх елементів, доки весь масив не буде відсортований.

Псевдокод алгоритму:

procedure InsertionSort(A)

n = length(A)

for i = 1 to n - 1 do

key = A[i]

j = i - 1

while j >= 0 and A[j] > key do

A[j + 1] = A[j]

j = j - 1

end while

A[j + 1] = key

end for

end procedure

Моделювання:

1. Беремо елемент A[i] з масиву A, починаючи з другого елементу (i = 1).

2. Порівнює його з усіма попередніми елементами A[j], де j = i - 1, i - 2, ..., 0.

3. Поки A[j] більше за key і j >= 0, ми зсуваємо A[j] вправо на одну позицію, щоб звільнити місце для key.

4. Потім вставляємо key на правильне місце в масиві.

5. Повторюємо цей процес для всіх елементів масиву.

**Розробка та опис програмної реалізації алгоритму**

1. **Код алгоритму на мові JavaScript**

function jHeapSort(arr) {

    const n = arr.length;

    // Побудова зеркальної купи з сортуванням вставками

    for (let i = 1; i < n; i++) {

      let current = arr[i];

      let j = i - 1;

      while (j >= 0 && arr[j] < current) {

        arr[j + 1] = arr[j];

        j--;

      }

      arr[j + 1] = current;

    }

    // Починаємо вилучати елементи один за одним з купи

    for (let i = n - 1; i > 0; i--) {

      // Переміщуємо поточний корінь (найбільший елемент) на кінець масиву

      [arr[0], arr[i]] = [arr[i], arr[0]];

      // Викликаємо heapify на зменшеній купі для забезпечення властивості купи

      heapify(arr, i, 0);

    }

  }

  function heapify(arr, n, i) {

    let largest = i;

    const left = 2 \* i + 1;

    const right = 2 \* i + 2;

    if (left < n && arr[left] > arr[largest]) {

      largest = left;

    }

    if (right < n && arr[right] > arr[largest]) {

      largest = right;

    }

    if (largest !== i) {

      // Обмін елементів та рекурсивне викликання heapify для підкучі

      [arr[i], arr[largest]] = [arr[largest], arr[i]];

      heapify(arr, n, largest);

    }

  }

  // Приклад використання:

  const arr = [3, 15, 11, 6, 9, 14, 10, 12, 1, 7, 8, 2, 13, 4, 5];

  jHeapSort(arr);

  console.log(arr);

**Покрокове виконання завдання:**

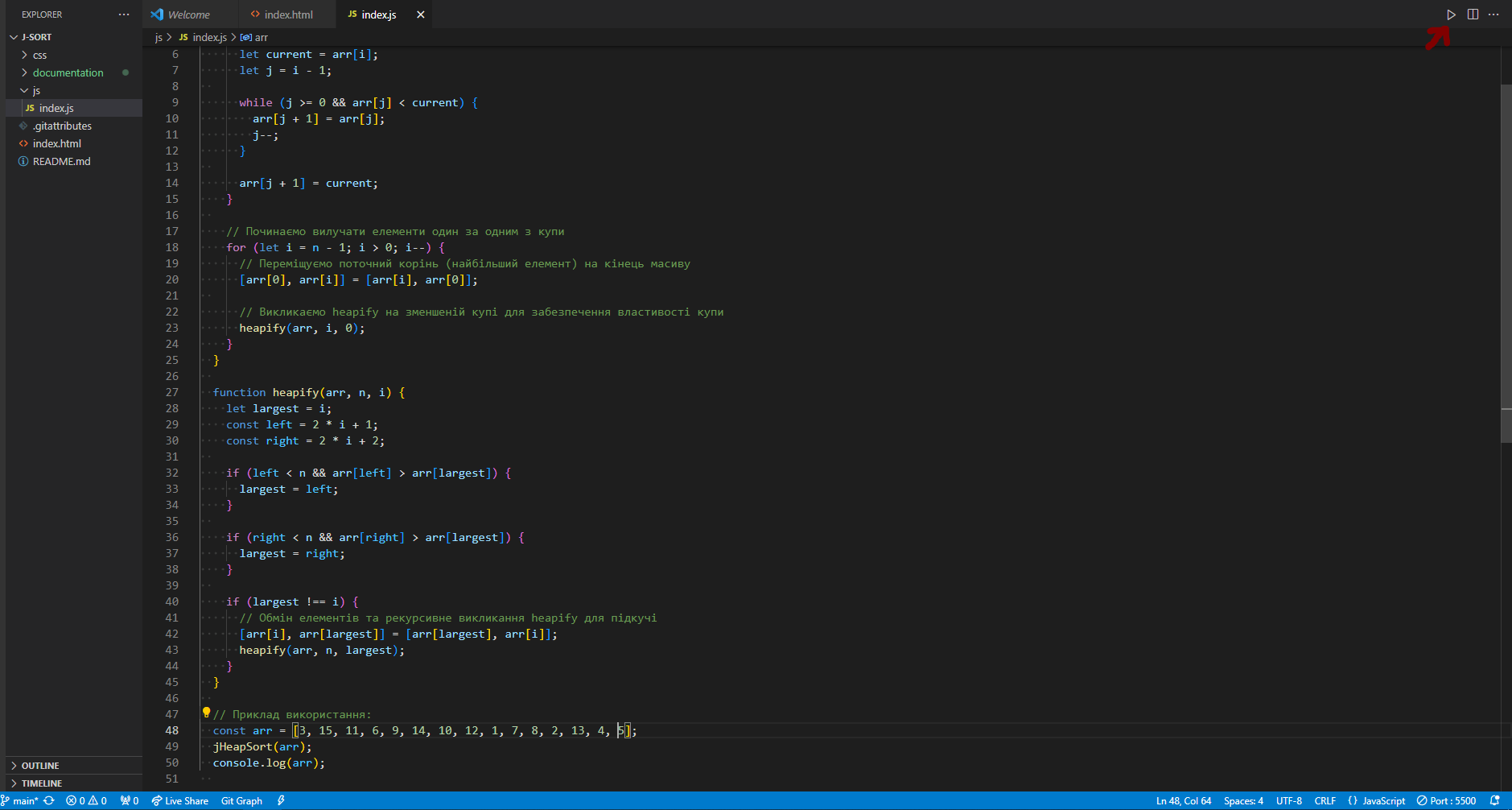
(Перед тим як перейти до покрокового виконання, зазначу, що я використовую IDE Visual Studio Code для праці з веб-застосунками та іншими програмами, які використовують JavaScript. Для візуалізації результату сортування, буде використано output саме цього редактору коду)

1. Вводимо бажаний масив (це можна зробити напряму у коді програми):



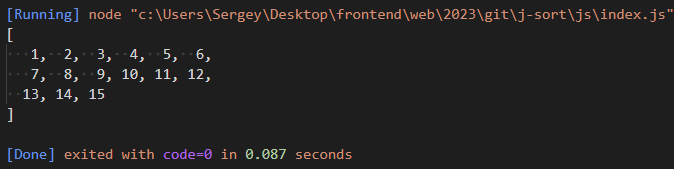
Мал. 1 приклад введенного масиву

1. Запускаємо алгоритм (робимо це всередині IDE):



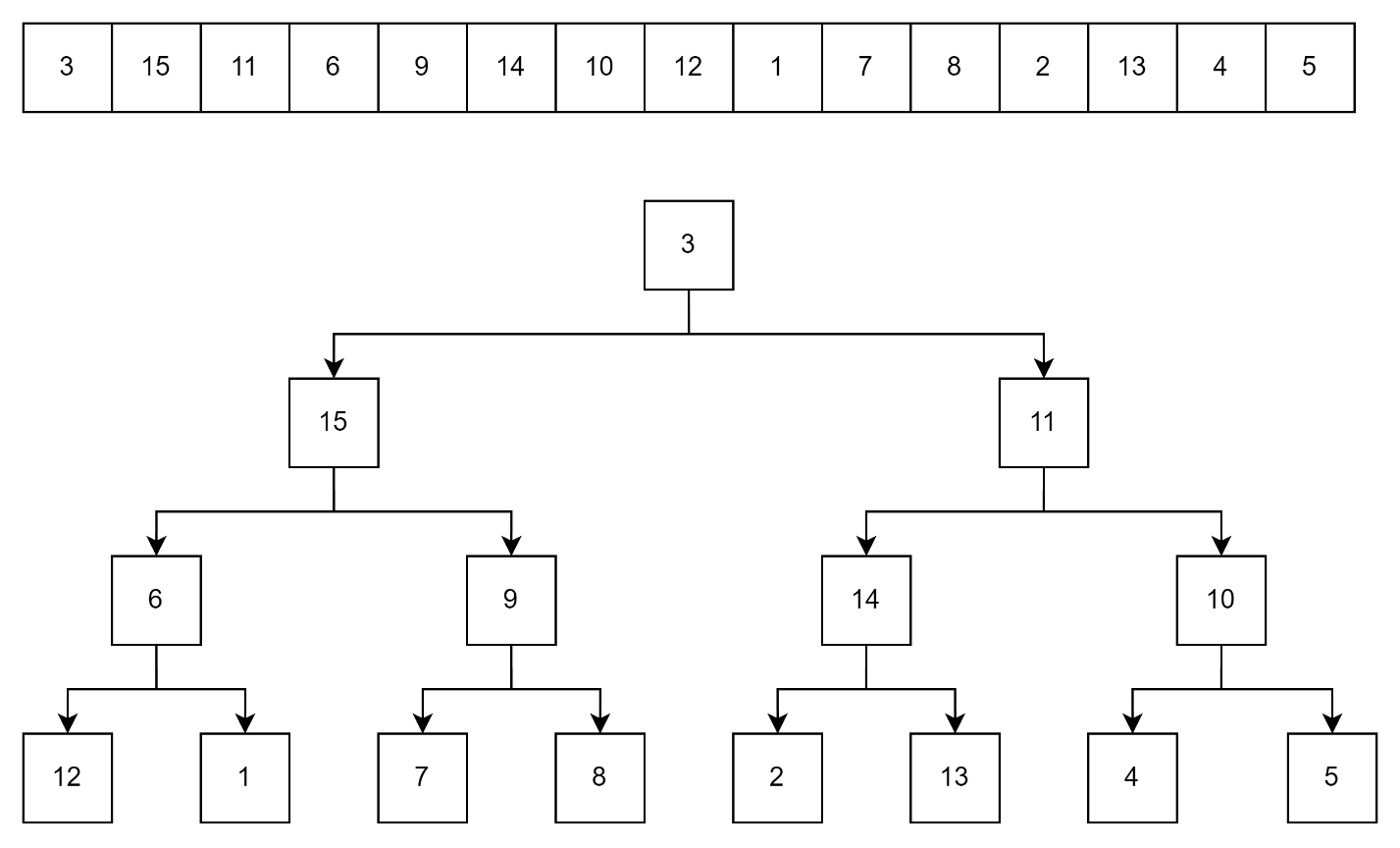
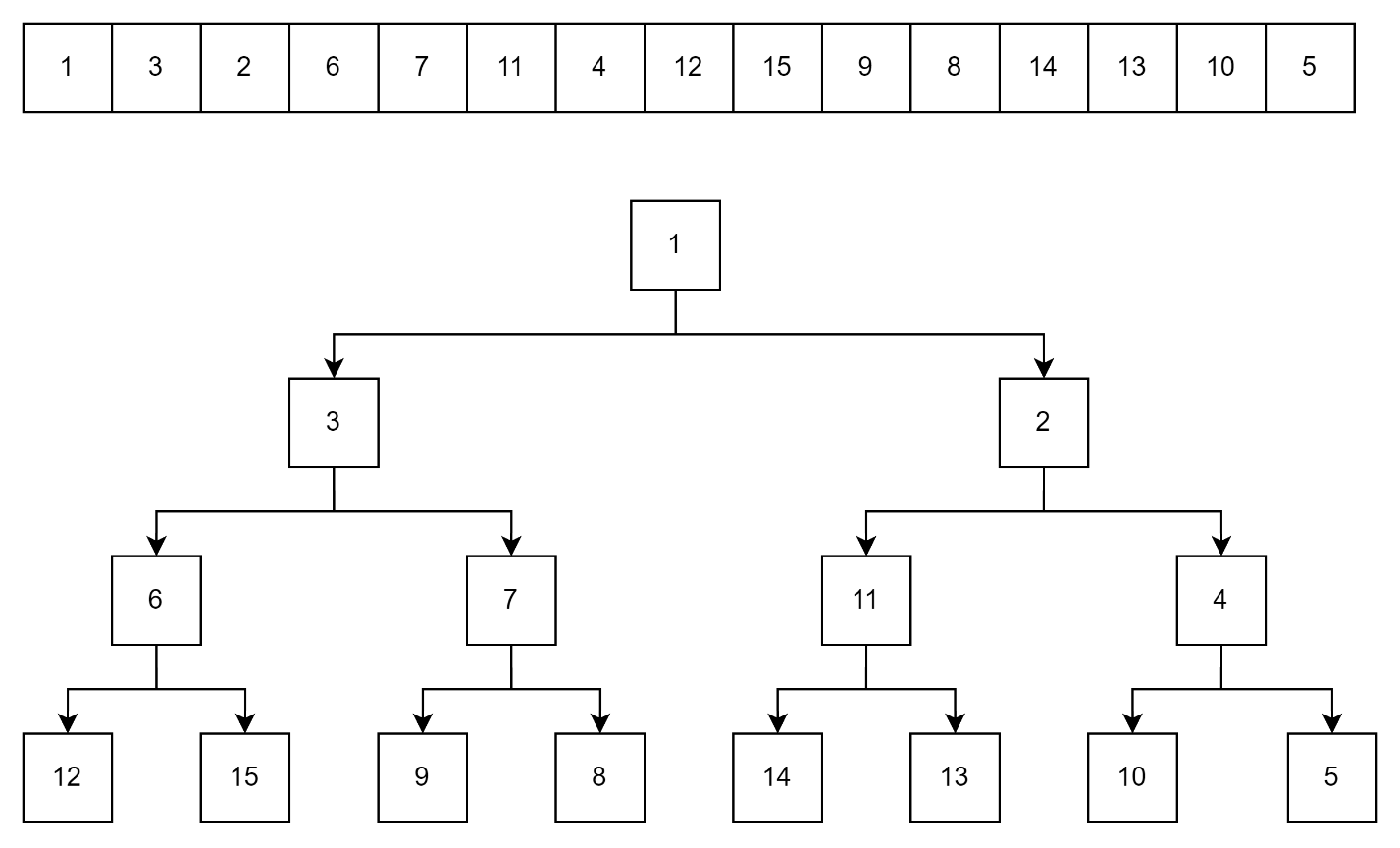
Мал. 2 компіляція коду

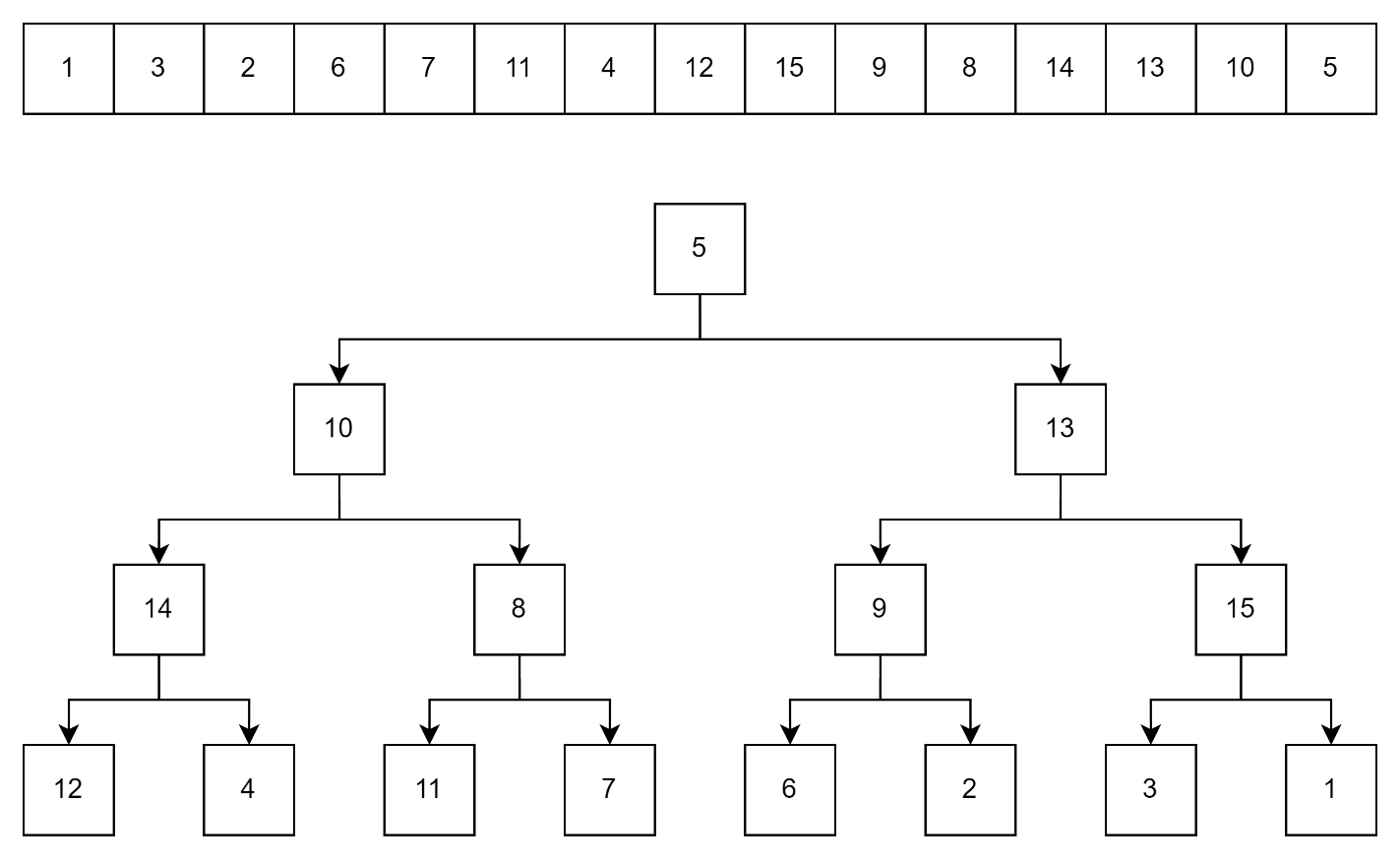
1. Отримаємо вивід у консоль:



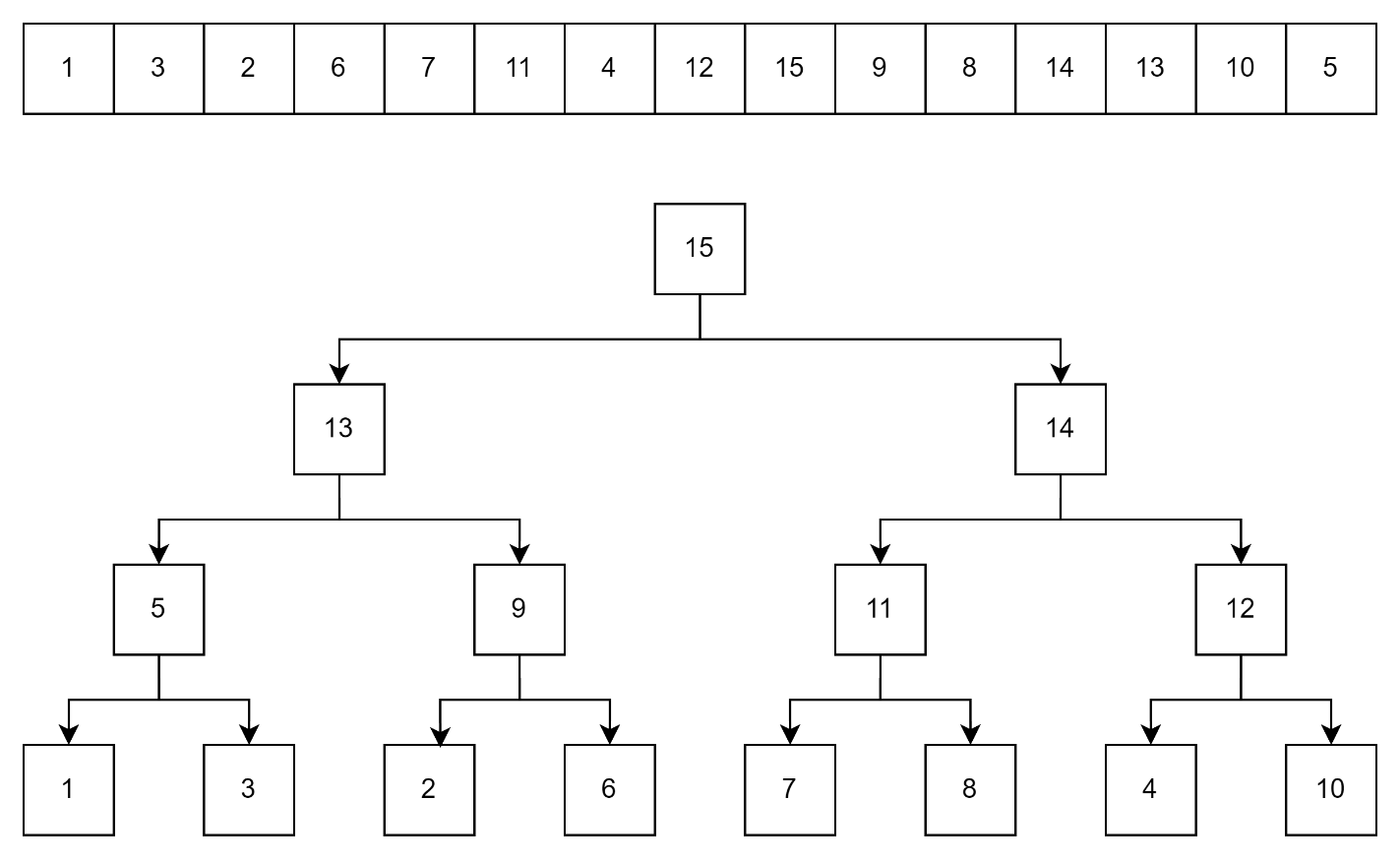
Мал. 3 Відсортований масив за результатами роботи програми

**Візуалізація:**

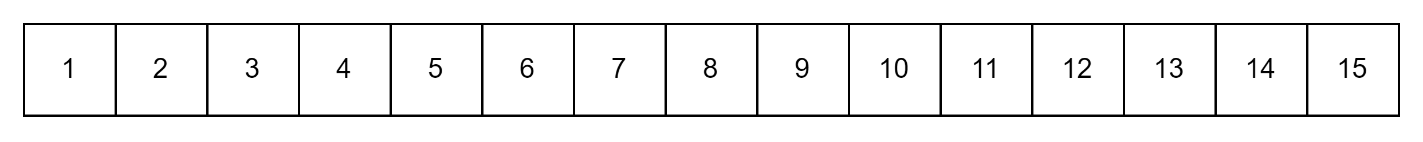
1. Робимо з масиву звичайне дерево: 
2. Робимо дерево сортируючим та незростаючим: 
3. Будуємо дзеркальне дерево:



1. Робимо дерево сортуючим та неспадаючим:



1. Досортовуємо масив використовуючи сортування вставками:



**Висновки, щодо досягнення мети Розрахунково-Графічної Роботи:**

На початку, маю зазначити, що алгоритм сортування j-sort, є куди менш популярним ніж алгоритм сортування купою, тому мої висновки можуть бути не зовсім коректними: я спираюся на інформацію, що знайшов в мережі інтернет. Для початку, розглянемо основні властивості алгоритмів — те, за що їх люблять:

1. Класичне сортування купою:

* Завжди працює за O(n log n) часу, незалежно від вихідного впорядкування даних.
* Це in-place алгоритм, тому він не потребує значної додаткової пам'яті.

Окремо розглянемо застосування для майже відсортованих масивів. Тут сортування купою не демонструє особливої ефективності, його продуктивність залишається «звичайною».

2. J-sort (Jason Morrison):

* Ефективність: J-сортування може бути оптимізоване для різних сценаріїв, зокрема (!) для майже відсортованих масивів.
* Також є in-place алгоритмом, подібно до класичного сортування купою.

Також зазначу, що однією з основних переваг J-sort є його здатність більш ефективно обробляти майже саме відсортовані масиви, зменшуючи кількість порівнянь та обмінів. Це є можливим завдяки тому, що цим алгоритмом можуть бути визначені частини масиву, які вже майже відсортовані — таким чином, зменшується необхідність у повному перебудовуванні купи. Ця властивість називається адаптуванням до "ступеня відсортованості" вхідного масиву.

Якщо занурюватися до абсолютних чисел, то тут звичайне сортування купою показує результат O(n log n) — при будь-якому підбору даних. Однак, у j-sort, ситуація вже буде унікальною: якої складності є масив, є він майже відсортованим, та чи потрібно потім буде досортування алгоритмом вставок? Я би виділив тут три сценарію:

1. За найсприятливішого розкладу. Одноразове пірамідобудування – це O(n).
2. Якщо потрібно будувати дві купи, це вже O(2n).
3. При праці з майже відсортованим масивом, сортування вставками може відсортувати дані за рекордні O(n).
4. У випадку, коли також необхідно довершити це все сортуванням вставками, буде O(n2).

Але, незважаючи на це все, J-сортування є більш підходящим для ситуацій, де масиви є майже відсортованими, оскільки воно може використовувати цю властивість для оптимізації продуктивності, зменшуючи загальний час виконання. В абсолютних цифрах — навіть при найгіршому сценарію — це все ще краще звичайного сортування купою.