KAUNO TECHNOLOGIJOS UNIVERSITETAS
Informatikos fakultetas

Algoritmų sudarymas ir analizė

Laboratorinis darbas nr. 1 (13 var.)

Atliko:	IFF-6 gr. studentas Artūras Šlajus
Data:	2009-03-29
Dėstytojas:	Valdas Astrovas

Kaunas, 2009

1 Užduotis

Rikiavimo uždavinys.

Palyginkite tris rūšiavimo algoritmus, kai rūšiavimas atliekamas masyve ir dinaminiame sąraše.

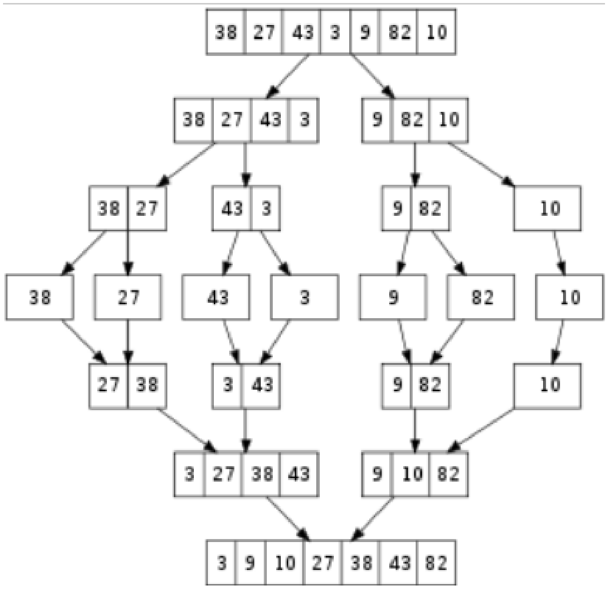
- Rūšiavimas „suliejimu“.
- Rūšiavimas „piramide“.
- Rūšiavimo algoritmas „Counting sort“

2 Algoritmų analizė

2.1 Rūšiavimas „suliejimu“

Masyvas rekursyviai skaidomas per puse, kol belieka po vieną elementą. Tada gauti masyvai rūšiuojant sujungiami atgal. Pagrindinė įdėją – mažesni sąrašai surūšiuojami greičiau negu dideli.

Algoritmo veikimo pavyzdys:



Teorinis algoritmo sudėtingumas: $\Theta(n \log n)$

2.2 Rūšiavimas „piramide“

Algoritmo esmė – sudaryti piramidę (heap) iš nesurūšiuotos masyvo dalies. Algoritmas vykdomas iteracijomis, kiekvienos iteracijos metu nerūšiuota masyvo dalis sumažinama vienetu, o iš nesurūšiuotos sudaroma nauja piramidė.

Teorinis algoritmo sudėtingumas: $\Theta(n \log n)$

2.3 Rūšiavimas „Counting sort“

Esmė – žinant min ir max reikšmes masyve sudaryti dar vieną masyvą (dydžio max - min), kurio nariai reikštų kiek kartų šis duomuo pasikartojo duomenų masyve. Tai vienas iš greičiausių rūšiavimo algoritmų, tačiau jis netinka, kai duomenų ruožas yra labai didelis.

Teorinis algoritmo sudėtingumas: $\Theta\ (n + k)$

2.4 Rezultatai

Algortmai realizuoti dvejomis kalbomis: JAVA ir C++.

Laikmena	Algoritmas	Elementų	Sekundžių	Operacijų
Su C++				
list	counting	3000	0	52985
list	counting	6000	0	55985
list	counting	9000	0	58991
list	counting	12000	0	61996
list	counting	15000	0	64996
list	counting	18000	0	67996
list	counting	21000	0.016	70996
list	counting	24000	0.016	73996
list	counting	27000	0.016	76998
list	counting	30000	0.016	79999
list	merge	3000	0.14	30957
list	merge	6000	0.547	67851
list	merge	9000	1.047	106930
list	merge	12000	1.813	147663
list	merge	15000	2.859	189334
list	merge	18000	4.079	231938
list	merge	21000	5.469	275704
list	merge	24000	7.078	319413
list	merge	27000	8.829	363757
list	merge	30000	10.828	408742
list	heap	3000	0.204	60227
list	heap	6000	0.782	132549
list	heap	9000	1.672	209129
list	heap	12000	3.063	289006
list	heap	15000	5.016	370172
list	heap	18000	7.391	453728
list	heap	21000	9.828	539544
list	heap	24000	12.563	626114
list	heap	27000	15.953	713014
list	heap	30000	19.875	800691
Su JAVA				
list	counting	3000	0.015	52985
list	counting	6000	0.062	55985
list	counting	9000	0.219	58991
list	counting	12000	0.468	61996
list	counting	15000	0.781	64996
list	counting	18000	1.172	67996
list	counting	21000	1.625	70996
list	counting	24000	2.156	73996
list	counting	27000	2.75	76998
list	counting	30000	3.39	79999

list	merge	3000	0.015	30957
list	merge	6000	0.016	67851
list	merge	9000	0.031	106930
list	merge	12000	0.031	147663
list	merge	15000	0.047	189334
list	merge	18000	0.047	231938
list	merge	21000	0.062	275704
list	merge	24000	0.094	319413
list	merge	27000	0.094	363757
list	merge	30000	0.109	408742
list	heap	3000	0.125	60238
list	heap	6000	0.516	132564
list	heap	9000	1.328	209140
list	heap	12000	2.469	289016
list	heap	15000	4	370188
list	heap	18000	6.016	453744
list	heap	21000	8.484	539562
list	heap	24000	11.218	626130
list	heap	27000	14.735	713028
list	heap	30000	18.328	800710
Laikmena	Algoritmas	Elementų	Sekundžių	Operacijų

Su C++

Laikmena	Algoritmas	Elementų	Sekundžių	Operacijų
vector	counting	250000	0.015	300000
vector	counting	500000	0.031	550000
vector	counting	750000	0.062	800000
vector	counting	1000000	0.063	1050000
vector	counting	1250000	0.078	1300000
vector	counting	1500000	0.094	1550000
vector	counting	1750000	0.078	1800000
vector	counting	2000000	0.125	2050000
vector	counting	2250000	0.11	2300000
vector	counting	2500000	0.156	2550000
vector	counting	2750000	0.172	2800000
vector	counting	3000000	0.172	3050000
vector	counting	3250000	0.188	3300000
vector	counting	3500000	0.218	3550000
vector	counting	3750000	0.234	3800000
vector	counting	4000000	0.249	4050000
vector	counting	4250000	0.249	4300000
vector	counting	4500000	0.266	4550000
vector	counting	4750000	0.312	4800000
vector	counting	5000000	0.297	5050000
vector	merge	250000	1.375	4168529
vector	merge	500000	2.734	8837131
vector	merge	750000	4.312	13705076
vector	merge	1000000	5.579	18673776
vector	merge	1250000	7.454	23762661
vector	merge	1500000	8.906	28910636
vector	merge	1750000	10.375	34111530
vector	merge	2000000	11.469	39349154
vector	merge	2250000	13.328	44660733

vector	merge	2500000	15.141	50025359
vector	merge	2750000	17.015	55413964
vector	merge	3000000	18.235	60820625
vector	merge	3250000	19.719	66256428
vector	merge	3500000	20.968	71723761
vector	merge	3750000	22.141	77203397
vector	merge	4000000	23.438	82696740
vector	merge	4250000	24.797	88218269
vector	merge	4500000	27.046	93820736
vector	merge	4750000	29.125	99430134
vector	merge	5000000	30.735	105049416

vector	heap	250000	0.438	8198400
vector	heap	500000	0.906	17396404
vector	heap	750000	1.421	27007186
vector	heap	1000000	1.984	36793658
vector	heap	1250000	2.5	46829144
vector	heap	1500000	3.063	57016444
vector	heap	1750000	3.672	67273750
vector	heap	2000000	4.188	77589062
vector	heap	2250000	4.891	88054494
vector	heap	2500000	5.391	98658662
vector	heap	2750000	6.125	109321256
vector	heap	3000000	6.703	120032302
vector	heap	3250000	7.5	130780154
vector	heap	3500000	8.016	141552202
vector	heap	3750000	8.609	152353586
vector	heap	4000000	9.329	163175876
vector	heap	4250000	9.953	174055302
vector	heap	4500000	10.453	185108728
vector	heap	4750000	11.187	196197020
vector	heap	5000000	11.922	207318322

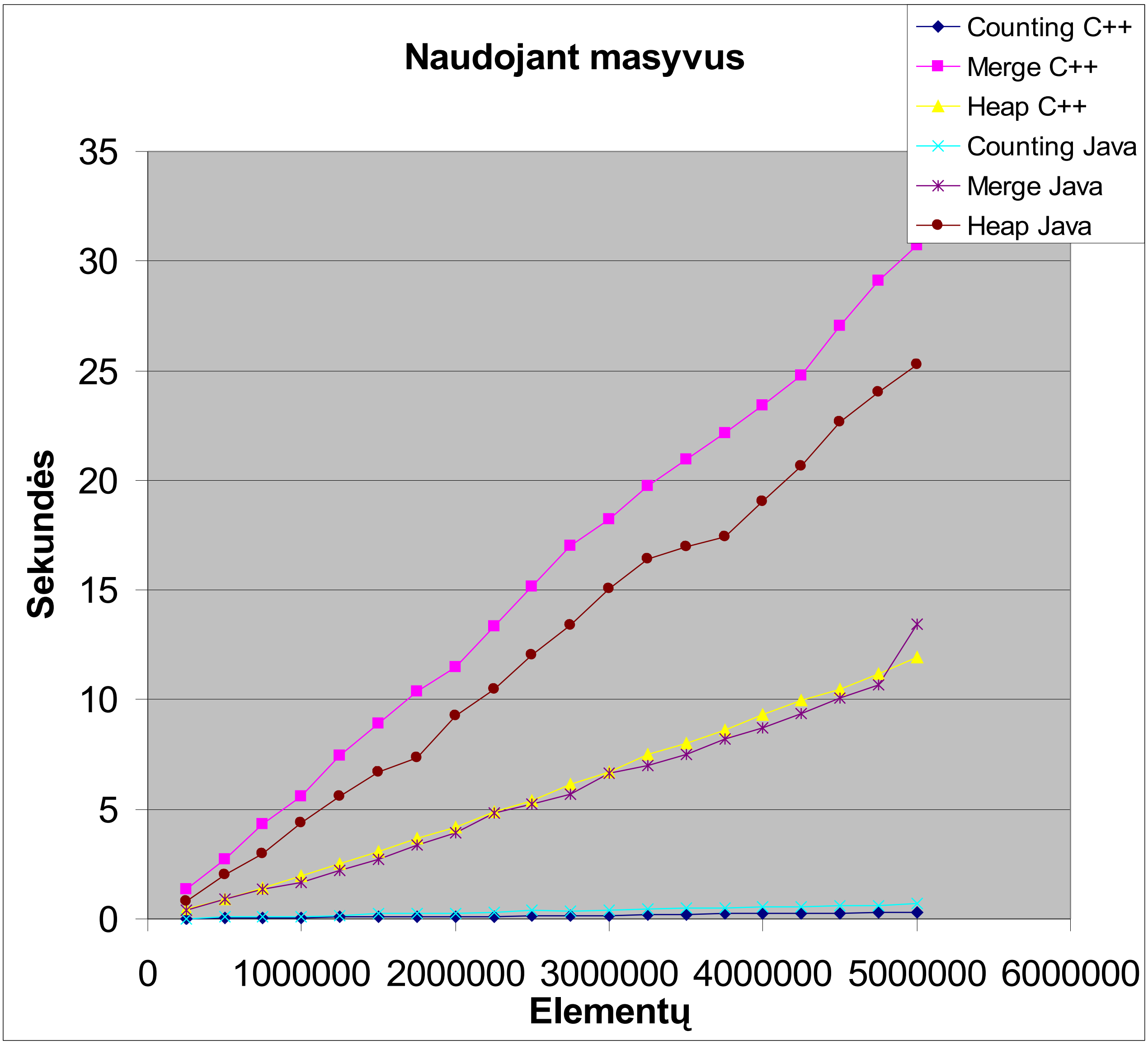
Su JAVA

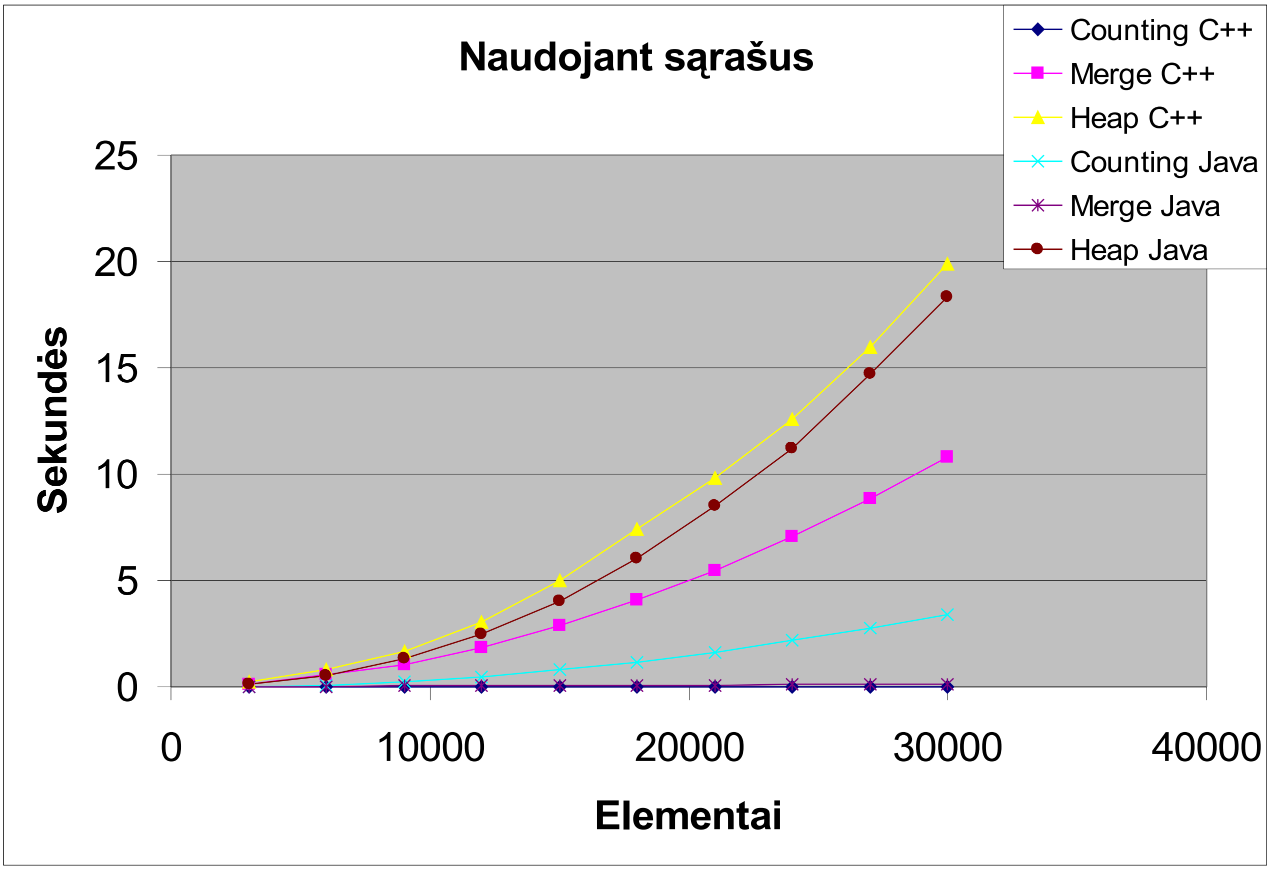
vector	counting	250000	0.015	300000
vector	counting	500000	0.078	550000
vector	counting	750000	0.109	800000
vector	counting	1000000	0.125	1050000
vector	counting	1250000	0.156	1300000
vector	counting	1500000	0.235	1550000
vector	counting	1750000	0.265	1800000
vector	counting	2000000	0.266	2050000
vector	counting	2250000	0.281	2300000
vector	counting	2500000	0.391	2550000
vector	counting	2750000	0.359	2800000
vector	counting	3000000	0.406	3050000
vector	counting	3250000	0.469	3300000
vector	counting	3500000	0.5	3550000
vector	counting	3750000	0.515	3800000
vector	counting	4000000	0.531	4050000
vector	counting	4250000	0.547	4300000
vector	counting	4500000	0.625	4550000
vector	counting	4750000	0.594	4800000
vector	counting	5000000	0.688	5050000

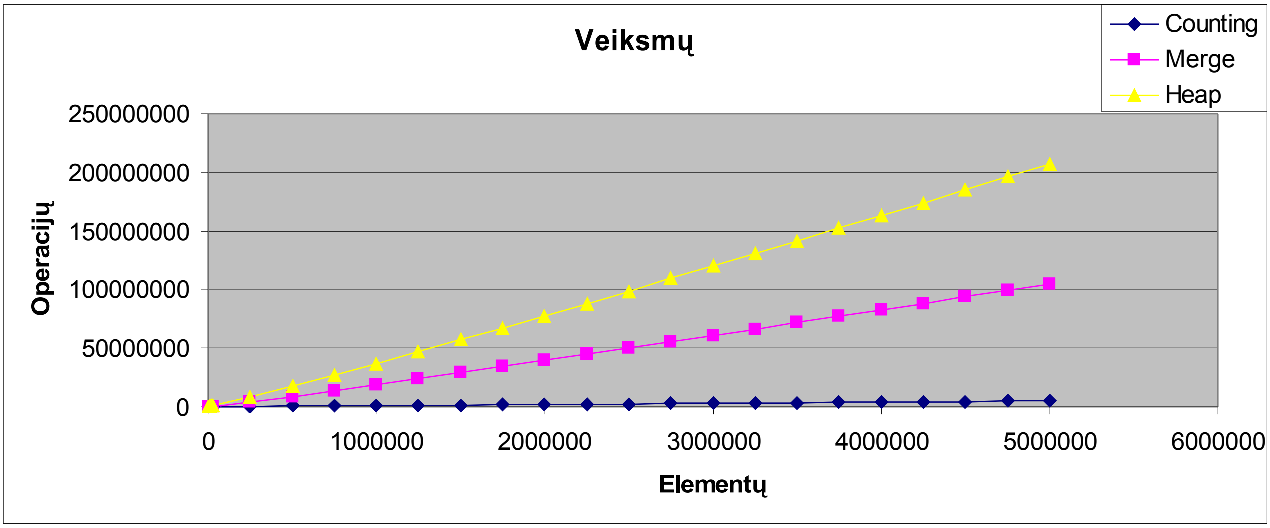
vector	merge	250000	0.391	4168529
vector	merge	500000	0.922	8837131
vector	merge	750000	1.359	13705076
vector	merge	1000000	1.657	18673776
vector	merge	1250000	2.234	23762661
vector	merge	1500000	2.719	28910636
vector	merge	1750000	3.36	34111530
vector	merge	2000000	3.953	39349154
vector	merge	2250000	4.859	44660733
vector	merge	2500000	5.219	50025359
vector	merge	2750000	5.672	55413964
vector	merge	3000000	6.625	60820625
vector	merge	3250000	6.984	66256428
vector	merge	3500000	7.5	71723761
vector	merge	3750000	8.188	77203397
vector	merge	4000000	8.734	82696740
vector	merge	4250000	9.36	88218269
vector	merge	4500000	10.078	93820736
vector	merge	4750000	10.672	99430134
vector	merge	5000000	13.453	105049416

vector	heap	250000	0.813	8198400
vector	heap	500000	2	17396404
vector	heap	750000	2.968	27007186
vector	heap	1000000	4.39	36793658
vector	heap	1250000	5.593	46829144
vector	heap	1500000	6.704	57016444
vector	heap	1750000	7.359	67273750
vector	heap	2000000	9.25	77589062
vector	heap	2250000	10.469	88054494
vector	heap	2500000	12.046	98658662
vector	heap	2750000	13.391	109321256
vector	heap	3000000	15.047	120032302
vector	heap	3250000	16.422	130780154
vector	heap	3500000	16.953	141552202
vector	heap	3750000	17.422	152353586
vector	heap	4000000	19.016	163175876
vector	heap	4250000	20.672	174055302
vector	heap	4500000	22.656	185108728
vector	heap	4750000	24	196197020
vector	heap	5000000	25.297	207318322

Žemiau pateikiami grafikai.


Naudojant masyvus
Counting C++
Merge C++
Heap C++
Counting Java
Merge Java
Heap Java
Sekundės
35
30
25
20
15
10
5
0
0
1000000
2000000
3000000
4000000
5000000
6000000
Elementų





3 Išvados

Kaip matome naudojant sąrašus telpant į tą patį laiką elementų galima apdoroti 100 kartų mažiau. Tai ko gero paaiškinama tuom, jog kreiptis į bet kurią masyvo vietą užtrunka vienodai (const).

Be to nors „heap“ ir „merge“ algoritmų teorinis sudėtingumas yra O(n*log(n)), bet iš grafiko matome, jog su masyvais tai panašiau į tiesę, tad sudėtingumo klasę O(n). Su sąrašais algoritmo laikas kinta pagal eksponentinį dėsnį.

„Counting sort“ nors ir nesimato grafike, tačiau jo suskaičiuotas sudėtingumas yra O(n+k), t.y.

prilygsta teoriniam sudėtingumui.

Be to – nors ir manoma, jog Java yra letesnė, tačiau keliuose algoritmuose ji veikė greičiau. Ko gero tai priklauso ir nuo kodo ir kompiliatoriaus optimizacijų.

4 Programų išeities tekstai

4.1 JAVA

4.1.1 Counting sort

```
/// countingSort - sort an array of values.
///
/// For best results the range of values to be sorted
/// should not be significantly larger than the number of
/// elements in the array.
  static LinkedList<Integer> list_counting_sort(LinkedList<Integer> source) {
    LinkedList<Integer> nums = new LinkedList<Integer>(source);
    int size = nums.size();
    // search for the minimum and maximum values in the input
    int i, min = nums.get(0), max = min;
    for (i = 1; i < size; ++i) {
      if (nums.get(i) < min) {
        min = nums.get(i);
      } else if (nums.get(i) > max) {
        max = nums.get(i);
      }
    }

    // create a counting array, counts, with a member for
    // each possible discrete value in the input.
    // request compiler to value-initialize all counts to 0.
    int distinct_element_count = max - min + 1;
    int[] counts = new int[distinct_element_count];

    // accumulate the counts -
    // each index in the counts array represents the value
    // of an element in the input nums array, so the result
    // of incrementing the sum at the index in the
    // counts array reflects the number of times the element
    // appears in the input array and therefore
    // must be copied to the sorted output.
    for (i = 0; i < size; ++i) {
      counts[nums.get(i) - min] += 1;
    }

    // store back into the input array the sorted value as
    // represented by the respective index in the counts array.
    // repeat for each additional occurrence of the value
    // found in the original array, as recorded by the
    // counts array.
    int j = 0;
    for (i = min; i <= max; i++) {
      number_of_operations++;
      for (int z = 0; z < counts[i - min]; z++) {
        number_of_operations++;
        nums.set(j++, i);
      }
    }

    return nums;
  }
/// countingSort - sort an array of values.
///
/// For best results the range of values to be sorted
/// should not be significantly larger than the number of
/// elements in the array.
  static ArrayList<Integer> vector_counting_sort(ArrayList<Integer> source) {
    ArrayList<Integer> nums = new ArrayList<Integer>(source);
    int size = nums.size();
    // search for the minimum and maximum values in the input
```

```
  int i, min = nums.get(0), max = min;
  for (i = 1; i < size; ++i) {
    if (nums.get(i) < min) {
      min = nums.get(i);
    } else if (nums.get(i) > max) {
      max = nums.get(i);
    }
  }

  // create a counting array, counts, with a member for
  // each possible discrete value in the input.
  // request compiler to value-initialize all counts to 0.
  int distinct_element_count = max - min + 1;
  int[] counts = new int[distinct_element_count];

  // accumulate the counts -
  // each index in the counts array represents the value
  // of an element in the input nums array, so the result
  // of incrementing the sum at the index in the
  // counts array reflects the number of times the element
  // appears in the input array and therefore
  // must be copied to the sorted output.
  for (i = 0; i < size; ++i) {
    counts[nums.get(i) - min] += 1;
  }

  // store back into the input array the sorted value as
  // represented by the respective index in the counts array.
  // repeat for each additional occurrence of the value
  // found in the original array, as recorded by the
  // counts array.
  int j = 0;
  for (i = min; i <= max; i++) {
    number_of_operations++;
    for (int z = 0; z < counts[i - min]; z++) {
      nums.set(j++, i);
      number_of_operations++;
    }
  }

  return nums;
}
```

4.1.2 Heap sort

```
static void list_sift_down(LinkedList<Integer> source, int start, int end) {
  // end represents the limit of how far down the heap to sift.

  int root = start;
  // While the root has at least one child
  while (root * 2 + 1 <= end) {
    // root*2+1 points to the left child
    int child = root * 2 + 1;

    // If the child has a sibling and the child's value is less than its sibling's...
    number_of_operations++;
    if (child + 1 <= end && source.get(child) < source.get(child + 1)) {
      // ... then point to the right child instead
      child++;
    }

    // out of max-heap order
    number_of_operations++;
    if (source.get(root) < source.get(child)) {
      swap(source, root, child);
      // repeat to continue sifting down the child now
      root = child;
    } else {
      return;
    }
  }
}

static void list_heapify(LinkedList<Integer> source) {
  // start is assigned the index in a of the last parent node
  int start = (source.size() - 2) / 2;
```

```
  while (start >= 0) {
    // sift down the node at index start to the proper place such that all nodes below
    // the start index are in heap order)
    list_sift_down(source, start, source.size() - 1);
    start--;
  // after sifting down the root all nodes/elements are in heap order
  }
}

static LinkedList<Integer> list_heap_sort(LinkedList<Integer> source) {
  LinkedList<Integer> result = new LinkedList<Integer>(source);
  list_heapify(result);

  int end = result.size() - 1;
  while (end > 0) {
    // swap the root(maximum value) of the heap with the last element of the heap
    swap(result, 0, end);
    // decrease the size of the heap by one so that the previous max value will
    // stay in its proper placement
    end--;
    // put the heap back in max-heap order
    list_sift_down(result, 0, end);
  }
  return result;
}

static void vector_sift_down(ArrayList<Integer> source, int start, int end) {
  // end represents the limit of how far down the heap to sift.

  int root = start;
  // While the root has at least one child
  while (root * 2 + 1 <= end) {
    // root*2+1 points to the left child
    int child = root * 2 + 1;
    // If the child has a sibling and the child's value is less than its sibling's...
    number_of_operations++;
    if (child + 1 <= end && source.get(child) < source.get(child + 1)) {
      // ... then point to the right child instead
      child++;
    }
    // out of max-heap order
    number_of_operations++;
    if (source.get(root) < source.get(child)) {
      swap(source, root, child);
      // repeat to continue sifting down the child now
      root = child;
    } else {
      return;
    }
  }
}

static void vector_heapify(ArrayList<Integer> source) {
  // start is assigned the index in a of the last parent node
  int start = (source.size() - 2) / 2;

  while (start >= 0) {
    // sift down the node at index start to the proper place such that all nodes below
    // the start index are in heap order)
    vector_sift_down(source, start, source.size() - 1);
    start--;
  // after sifting down the root all nodes/elements are in heap order
  }
}

static ArrayList<Integer> vector_heap_sort(ArrayList<Integer> source) {
  ArrayList<Integer> result = new ArrayList<Integer>(source);
  vector_heapify(result);

  int end = result.size() - 1;
  while (end > 0) {
    // swap the root(maximum value) of the heap with the last element of the heap
    swap(result, 0, end);
    // decrease the size of the heap by one so that the previous max value will
    // stay in its proper placement
    end--;
```

```
    // put the heap back in max-heap order
    vector_sift_down(result, 0, end);
  }
  return result;
}

static void swap(List<Integer> source, int indexA, int indexB) {
  int temp = source.get(indexA);
  source.set(indexA, source.get(indexB));
  source.set(indexB, temp);
}
```

4.1.3 Merge sort

```
static LinkedList<Integer> list_merge(LinkedList<Integer> left, LinkedList<Integer> right) {
  LinkedList<Integer> result = new LinkedList<Integer>();
  while (left.size() > 0 && right.size() > 0) {
    if (left.getFirst() <= right.getFirst()) {
      result.add(left.pop());
    } else {
      result.add(right.pop());
    }
    number_of_operations++;
  }

  result.addAll(left);
  result.addAll(right);

  return result;
}

static LinkedList<Integer> list_merge_sort(LinkedList<Integer> source) {
  LinkedList<Integer> work = new LinkedList<Integer>(source);
  int length = work.size();
  if (length <= 1) {
    return work;
  }

  int middle = length / 2 - 1;
  LinkedList<Integer> left = new LinkedList<Integer>();
  LinkedList<Integer> right = new LinkedList<Integer>();

  for (int i = 0; i <= middle; i++) {
    left.add(work.pop());
  }
  for (int i = middle + 1; i < length; i++) {
    right.add(work.pop());
  }

  left = list_merge_sort(left);
  right = list_merge_sort(right);
  return list_merge(left, right);
}

static ArrayList<Integer> vector_merge(ArrayList<Integer> left, ArrayList<Integer> right) {
  ArrayList<Integer> result = new ArrayList<Integer>(left.size() + right.size());
  int leftIndex = 0, rightIndex = 0;
  while (leftIndex < left.size() && rightIndex < right.size()) {
    if (left.get(leftIndex) <= right.get(rightIndex)) {
      result.add(left.get(leftIndex));
      leftIndex++;
    } else {
      result.add(right.get(rightIndex));
      rightIndex++;
    }
    number_of_operations++;
  }

  for (int i = leftIndex; i < left.size(); i++) {
    result.add(left.get(i));
  }
  for (int i = rightIndex; i < right.size(); i++) {
    result.add(right.get(i));
  }
```

```
    return result;
  }

  static ArrayList<Integer> vector_merge_sort(ArrayList<Integer> source) {
    int length = source.size();
    if (length <= 1) {
      return source;
    }

    int middle = length / 2 - 1;
    ArrayList<Integer> left = new ArrayList<Integer>(middle + 1);
    ArrayList<Integer> right = new ArrayList<Integer>(middle + 1);

    for (int i = 0; i <= middle; i++) {
      left.add(source.get(i));
    }
    for (int i = middle + 1; i < length; i++) {
      right.add(source.get(i));
    }

    left = vector_merge_sort(left);
    right = vector_merge_sort(right);
    ArrayList<Integer> result = vector_merge(left, right);
    return result;
  }
```

4.2 C++

4.2.1 Counting sort

```
/// countingSort - sort an array of values.
///
/// For best results the range of values to be sorted
/// should not be significantly larger than the number of
/// elements in the array.
list<int> list_counting_sort(list<int> &nums, long &number_of_operations) {
    int size = nums.size();
    // search for the minimum and maximum values in the input
    int i, min = nums.front(), max = min;
    for (list<int>::iterator it = nums.begin(); it != nums.end(); it++) {
        if (*it < min)
            min = *it;
        else if (*it > max)
            max = *it;
    }

    // create a counting array, counts, with a member for
    // each possible discrete value in the input.
    // request compiler to value-initialize all counts to 0.
    int distinct_element_count = max - min + 1;
    int *counts = new int[distinct_element_count]();

    // accumulate the counts -
    // each index in the counts array represents the value
    // of an element in the input nums array, so the result
    // of incrementing the sum at the index in the
    // counts array reflects the number of times the element
    // appears in the input array and therefore
    // must be copied to the sorted output.
    for (list<int>::iterator it = nums.begin(); it != nums.end(); it++)
        ++counts[ *it - min ];

    // store back into the input array the sorted value as
    // represented by the respective index in the counts array.
    // repeat for each additional occurrence of the value
    // found in the original array, as recorded by the
    // counts array.
    list<int> result;
    for (i = min; i <= max; i++) {
        number_of_operations++;
        for (int z = 0; z < counts[i - min]; z++) {
            result.push_back(i);
            number_of_operations++;
        }
```

```
    }

    delete[] counts;

    return result;
}

/// countingSort - sort an array of values.
///
/// For best results the range of values to be sorted
/// should not be significantly larger than the number of
/// elements in the array.
vector<int> vector_counting_sort(vector<int> &source, long &number_of_operations) {
    vector<int> nums = vector<int>(source);
    int size = nums.size();
    // search for the minimum and maximum values in the input
    int i, min = nums.at(0), max = min;
    for (i = 1; i < size; ++i) {
        if (nums.at(i) < min)
            min = nums.at(i);
        else if (nums[i] > max)
            max = nums.at(i);
    }

    // create a counting array, counts, with a member for
    // each possible discrete value in the input.
    // request compiler to value-initialize all counts to 0.
    int distinct_element_count = max - min + 1;
    int *counts = new int[distinct_element_count]();

    // accumulate the counts -
    // each index in the counts array represents the value
    // of an element in the input nums array, so the result
    // of incrementing the sum at the index in the
    // counts array reflects the number of times the element
    // appears in the input array and therefore
    // must be copied to the sorted output.
    for (i = 0; i < size; ++i) {
        ++counts[ nums[i] - min ];
    }

    // store back into the input array the sorted value as
    // represented by the respective index in the counts array.
    // repeat for each additional occurrence of the value
    // found in the original array, as recorded by the
    // counts array.
    int j = 0;
    for (i = min; i <= max; i++) {
        number_of_operations++;
        for (int z = 0; z < counts[i - min]; z++) {
            nums.at(j++) = i;
            number_of_operations++;
        }
    }

    delete[] counts;

    return nums;
}
```

4.2.2 Heap sort

```
void list_sift_down(list<int> &source, int start, int end, long &number_of_operations) {
    // end represents the limit of how far down the heap to sift.

    int root = start;
    // While the root has at least one child
    while (root * 2 + 1 <= end) {
        // root*2+1 points to the left child
        int child = root * 2 + 1;

        // If the child has a sibling and the child's value is less than its sibling's...
        list<int>::iterator it = source.begin();
        advance(it, child);
        if (child + 1 <= end) {
            list<int>::iterator it1 = it;
```

```
            it1++;

            number_of_operations++;
            if (*it < *it1) {
                // ... then point to the right child instead
                child++;
                it = it1;
            }
        }

        // out of max-heap order
        list<int>::iterator root_it = source.begin();
        advance(root_it, root);

        number_of_operations++;
        if (*root_it < *it) {
            list_swap(source, root_it, source, it);
            // repeat to continue sifting down the child now
            root = child;
        } else {
            return;
        }
    }
}

void list_heapify(list<int> &source, long &number_of_operations) {
    // start is assigned the index in a of the last parent node
    int start = (source.size() - 2) / 2;

    while (start >= 0) {
        // sift down the node at index start to the proper place such that all nodes below
        // the start index are in heap order)
        list_sift_down(source, start, source.size() - 1, number_of_operations);
        start--;
        // after sifting down the root all nodes/elements are in heap order
    }
}

list<int> list_heap_sort(list<int> &source, long &number_of_operations) {
    list<int> result = list<int>(source);
    list_heapify(result, number_of_operations);

    int end = result.size() - 1;
    while (end > 0) {
        // swap the root(maximum value) of the heap with the last element of the heap
        list<int>::iterator end_it = result.begin();
        advance(end_it, end);
        list_swap(result, result.begin(), result, end_it);
        // decrease the size of the heap by one so that the previous max value will
        // stay in its proper placement
        end--;
        // put the heap back in max-heap order
        list_sift_down(result, 0, end, number_of_operations);
    }
    return result;
}

void list_swap(list<int> &l1, list<int>::iterator it1,
        list<int> &l2, list<int>::iterator it2) {
    l1.insert(it1, *it2);
    l2.insert(it2, *it1);
    l1.erase(it1);
    l2.erase(it2);
}

void vector_sift_down(vector<int> &source, int start, int end, long &number_of_operations) {
    // end represents the limit of how far down the heap to sift.

    int root = start;
    // While the root has at least one child
    while (root * 2 + 1 <= end) {
        // root*2+1 points to the left child
        int child = root * 2 + 1;
        // If the child has a sibling and the child's value is less than its sibling's...
        number_of_operations++;
        if (child + 1 <= end && source.at(child) < source.at(child + 1)) {
            // ... then point to the right child instead
```

```
            child++;
        }
        // out of max-heap order
        number_of_operations++;
        if (source.at(root) < source.at(child)) {
            swap(source.at(root), source.at(child));
            // repeat to continue sifting down the child now
            root = child;
        } else {
            return;
        }
    }
}

void vector_heapify(vector<int> &source, long &number_of_operations) {
    // start is assigned the index in a of the last parent node
    int start = (source.size() - 2) / 2;

    while (start >= 0) {
        // sift down the node at index start to the proper place such that all nodes below
        // the start index are in heap order)
        vector_sift_down(source, start, source.size() - 1, number_of_operations);
        start--;
        // after sifting down the root all nodes/elements are in heap order
    }
}

vector<int> vector_heap_sort(vector<int> &source, long &number_of_operations) {
    vector<int> result = vector<int>(source);
    vector_heapify(result, number_of_operations);

    int end = result.size() - 1;
    while (end > 0) {
        // swap the root(maximum value) of the heap with the last element of the heap
        swap(result.at(0), result.at(end));
        // decrease the size of the heap by one so that the previous max value will
        // stay in its proper placement
        end--;
        // put the heap back in max-heap order
        vector_sift_down(result, 0, end, number_of_operations);
    }
    return result;
}

void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
```

4.2.3 Merge sort

```
list<int> list_merge(list<int> &left, list<int> &right, long &number_of_operations) {
    list<int> result;
    while (left.size() > 0 && right.size() > 0) {
        if (left.front() <= right.front()) {
            result.push_back(left.front());
            left.pop_front();
        } else {
            result.push_back(right.front());
            right.pop_front();
        }

        number_of_operations++;
    }

    result.splice(result.end(), left);
    result.splice(result.end(), right);

    return result;
}

list<int> list_merge_sort(list<int> source, long &number_of_operations) {
    int length = source.size();
    if (length <= 1) {
        return source;
```

```
    }

    int middle = length / 2 - 1;
    list<int> left, right;

    list<int>::iterator it = source.begin();
    for (int i = 0; i <= middle; i++) {
        left.push_back(*it);
        it++;
    }
    for (int i = middle + 1; i < length; i++) {
        right.push_back(*it);
        it++;
    }

    left = list_merge_sort(left, number_of_operations);
    right = list_merge_sort(right, number_of_operations);
    list<int> result = list_merge(left, right, number_of_operations);
    return result;
}

vector<int> vector_merge(vector<int> &left, vector<int> &right, long &number_of_operations) {
    vector<int> result;
    int leftIndex = 0, rightIndex = 0;
    while (leftIndex < left.size() && rightIndex < right.size()) {
        if (left.at(leftIndex) <= right.at(rightIndex)) {
            result.push_back(left.at(leftIndex));
            leftIndex++;
        } else {
            result.push_back(right.at(rightIndex));
            rightIndex++;
        }
        number_of_operations++;
    }

    for (int i = leftIndex; i < left.size(); i++) {
        result.push_back(left.at(i));
    }
    for (int i = rightIndex; i < right.size(); i++) {
        result.push_back(right.at(i));
    }

    return result;
}

vector<int> vector_merge_sort(vector<int> &source, long &number_of_operations) {
    int length = source.size();
    if (length <= 1) {
        return source;
    }

    int middle = length / 2 - 1;
    vector<int> left, right;
    left.reserve(middle + 1);
    right.reserve(middle + 1);

    for (int i = 0; i <= middle; i++) {
        left.push_back(source.at(i));
    }
    for (int i = middle + 1; i < length; i++) {
        right.push_back(source.at(i));
    }

    left = vector_merge_sort(left, number_of_operations);
    right = vector_merge_sort(right, number_of_operations);
    vector<int> result = vector_merge(left, right, number_of_operations);
    return result;
}
```