Notes for ECE2810J

Tianzong Cheng

Contents

1. Asymptotic Algorithm Analysis

2. Comparison Sort

2.1. Bubble Sort

```
// Function to perform Bubble Sort
void bubbleSort(int arr[], int n) {
  for (int i = 0; i < n - 1; i++) {
    // Flag to optimize the algorithm
    bool swapped = false;

    // Last i elements are already in place, so we don't need to check them
    for (int j = 0; j < n - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        // Swap arr[j] and arr[j+1]
        int temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
        swapped = true;
      }
    }

    // If no two elements were swapped in inner loop, the array is already
    // sorted
    if (!swapped) {
      break;
    }
  }
}
```

Note that the last `i` elements don't need to be checked: `for (int j = 0; j < n - i - 1; j++)`.

2.2. Selection Sort

Note that selection sort is **not stable**.

```
// Function to perform Selection Sort
void selectionSort(int arr[], int n) {
  for (int i = 0; i < n - 1; i++) {
    int minIndex = i;
    for (int j = i + 1; j < n; j++) {
      if (arr[j] < arr[minIndex]) {
        minIndex = j;
      }
    }
    // Swap the found minimum element with the current element
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
  }
}
```

2.3. Insertion Sort

Review this before exam!

```
// Function to perform Insertion Sort
void insertionSort(int arr[], int n) {
  for (int i = 1; i < n; i++) {
    int key = arr[i];
    int j = i - 1;
```

```
    // Move elements of arr[0..i-1], that are greater than key,
    // to one position ahead of their current position
    while (j >= 0 && arr[j] > key) {
      arr[j + 1] = arr[j];
      j--;
    }
    arr[j + 1] = key;
  }
}
```

2.4. Merge Sort

```
// Merge two subarrays of arr[]
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(std::vector<int> &arr, int l, int m, int r) {
  int n1 = m - l + 1;
  int n2 = r - m;

  // Create temporary arrays
  std::vector<int> L(n1);
  std::vector<int> R(n2);

  // Copy data to temporary arrays L[] and R[]
  for (int i = 0; i < n1; i++) {
    L[i] = arr[l + i];
  }
  for (int i = 0; i < n2; i++) {
    R[i] = arr[m + 1 + i];
  }

  // Merge the temporary arrays back into arr[l..r]
  int i = 0; // Initial index of first subarray
  int j = 0; // Initial index of second subarray
  int k = l; // Initial index of merged subarray

  while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
      arr[k] = L[i];
      i++;
    } else {
      arr[k] = R[j];
      j++;
    }
    k++;
  }

  // Copy the remaining elements of L[], if there are any
  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }

  // Copy the remaining elements of R[], if there are any
  while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
```

```
  }
}

// Main function to perform Merge Sort
void mergeSort(std::vector<int> &arr, int l, int r) {
  if (l < r) {
    // Same as (l+r)/2, but avoids overflow for large l and r
    int m = l + (r - l) / 2;

    // Sort first and second halves
    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);

    // Merge the sorted halves
    merge(arr, l, m, r);
  }
}
```

2.5. Quick Sort

Note that the worst case of quick sort is $O(n^2)$, and it is weakly in place because of the stack space usage during recursion.

```
// Function to partition the array into two subarrays based on a pivot element
// Elements smaller than the pivot are on the left, and elements greater than
// the pivot are on the right.
int partition(std::vector<int> &arr, int low, int high) {
  int pivot = arr[high]; // Choose the rightmost element as the pivot
  int i = (low - 1);     // Index of the smaller element

  for (int j = low; j <= high - 1; j++) {
    // If the current element is smaller than or equal to the pivot
    if (arr[j] <= pivot) {
      i++;
      // Swap arr[i] and arr[j]
      std::swap(arr[i], arr[j]);
    }
  }

  // Swap arr[i+1] and arr[high] (or the pivot)
  std::swap(arr[i + 1], arr[high]);
  return (i + 1);
}

// Function to perform Quick Sort
void quickSort(std::vector<int> &arr, int low, int high) {
  if (low < high) {
    // Partition the array into two subarrays
    int pi = partition(arr, low, high);

    // Recursively sort the subarrays
    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
  }
}
```

3. Non-Comparison Sort

3.1. Counting Sort

Time complexity: $O(n + k)$

3.1.1. Simple Version

```
// Function to perform counting sort
void countingSort(vector<int> &arr) {
  // Find the maximum element in the array
  int max_element = arr[0];
  for (int i = 1; i < arr.size(); ++i) {
    if (arr[i] > max_element) {
      max_element = arr[i];
    }
  }

  // Create a count array to store the count of each element
  vector<int> count(max_element + 1, 0);

  // Count the occurrences of each element in the input array
  for (int i = 0; i < arr.size(); ++i) {
    count[arr[i]]++;
  }

  // Reconstruct the sorted array from the count array
  int index = 0;
  for (int i = 0; i <= max_element; ++i) {
    while (count[i] > 0) {
      arr[index] = i;
      index++;
      count[i]--;
    }
  }
}
```

3.1.2. General Version

Review before exam!

```
// Counting sort implementation that is stable and considers additional
// information
void countingSort(std::vector<Item> &items, int maxKey) {
  int n = items.size();
  std::vector<int> count(maxKey + 1, 0); // Array C[k+1]
  std::vector<Item> sortedItems(n);      // Output array to store sorted items

  // Step 1: Count occurrences of each key
  for (int i = 0; i < n; ++i) {
    count[items[i].key]++;
  }

  // Step 2: Accumulate counts to get positions
  for (int i = 1; i <= maxKey; ++i) {
    count[i] += count[i - 1];
  }

  // Step 3: Place items in sorted order based on their counts
  // Traverse from right to left to maintain stability
```

```
  for (int i = n - 1; i >= 0; --i) {
    int key = items[i].key;
    int pos = count[key] - 1; // Position in sorted array
    sortedItems[pos] = items[i];
    count[key]--; // Decrement count for stability
  }

  // Copy the sorted items back into the original array
  items = sortedItems;
}
```

3.2. Bucket Sort

Time complexity: $O\left(n \log\left(\frac{n}{c}\right)\right)$

If c is close to n, $O(n)$. If c is close to 1, $O(n \log n)$.

Note that in the following example, it is assumed that all the elements in the array are floating-point numbers between 0 and 1.

```
// Function to perform bucket sort
void bucketSort(vector<float> &arr) {
  int n = arr.size();

  // Create an array of empty buckets
  vector<vector<float>> buckets(n);

  // Place elements into buckets based on their values
  for (int i = 0; i < n; ++i) {
    int bucketIndex = n * arr[i]; // Calculate the index of the bucket
    buckets[bucketIndex].push_back(arr[i]);
  }

  // Sort each bucket using Quick Sort
  for (int i = 0; i < n; ++i) {
    quickSort(buckets[i], 0, buckets[i].size() - 1);
  }

  // Concatenate the sorted buckets to get the final sorted array
  int index = 0;
  for (int i = 0; i < n; ++i) {
    for (float num : buckets[i]) {
      arr[index] = num;
      index++;
    }
  }
}
```

Remakrs: Bucket sort is more often used on continuous values and requires knowing the range of the input data beforehand. When the data is uniformly distributed within a known range, bucket sort performs the best.

3.3. Radix Sort

Time complexity: $O(kn)$

```
// Function to perform LSD Radix Sort using Bucket Sort
void lsdRadixSort(vector<int> &arr) {
  int max = getMax(arr);
```

```
  int numDigits = static_cast<int>(log10(max)) + 1;

  // Perform Bucket Sort for each digit, from right to left
  for (int exp = 1; exp <= numDigits; exp++) {
    bucketSort(arr, exp);
  }
}
```

```
Original Array: 170 4532 754 9045 8021 240 222 6666
170 240 8021 222 4532 754 9045 6666
222 8021 4532 240 9045 754 6666 170
8021 9045 170 222 240 4532 6666 754
170 222 240 754 4532 6666 8021 9045
Sorted Array: 170 222 240 754 4532 6666 8021 9045
```

4. Linear Time Selection

4.1. Randomized Selection

```
int partition(vector<int> &arr, int low, int high) {
  int pivot = arr[low];
  int left = low + 1;
  int right = high;

  while (true) {
    while (left <= right && arr[left] < pivot)
      left++;
    while (left <= right && arr[right] > pivot)
      right--;

    if (left <= right) {
      swap(arr[left], arr[right]);
    } else {
      break;
    }
  }

  swap(arr[low], arr[right]);
  return right;
}

int randomizedSelection(vector<int> &arr, int low, int high, int k) {
  if (low == high) {
    return arr[low];
  }

  // int pivotIndex = rand() % (high - low + 1) + low;
  // swap(arr[low], arr[pivotIndex]);

  int pivotPosition = partition(arr, low, high);

  if (k == pivotPosition) {
    return arr[pivotPosition];
  } else if (k < pivotPosition) {
    return randomizedSelection(arr, low, pivotPosition - 1, k);
  } else {
    return randomizedSelection(arr, pivotPosition + 1, high, k);
  }
}
```

4.2. Deterministic Selection

Key idea: find the **median of medians** rather than randomly selecting a pivot.

Note that **insertion sort** is often used to sort the small group because it is efficient for small size. It is also stable and in-place at the same time.

```
// Function to find the median of a small vector
int findMedian(vector<int> &arr, int left, int right) {
  sort(arr.begin() + left, arr.begin() + right + 1);
  return arr[(left + right) / 2];
}

// Deterministic selection algorithm
```

```cpp
int deterministicSelection(vector<int> &arr, int low, int high, int k) {
  if (low == high) {
    return arr[low];
  }

  int n = high - low + 1;

  // Divide the array into groups of size 5
  vector<int> medians;
  for (int i = 0; i < n / 5; i++) {
    int left = low + i * 5;
    int right = left + 4;
    medians.push_back(findMedian(arr, left, right));
  }

  // Find the median of medians
  int medianOfMedians =
      (medians.size() == 1)
          ? medians[0]
          : deterministicSelection(medians, 0, medians.size() - 1,
                                   medians.size() / 2);

  // Partition the array based on the median of medians
  int pivotIndex = partition(arr, low, high, medianOfMedians);

  if (k == pivotIndex) {
    return arr[pivotIndex];
  } else if (k < pivotIndex) {
    return deterministicSelection(arr, low, pivotIndex - 1, k);
  } else {
    return deterministicSelection(arr, pivotIndex + 1, high, k);
  }
}
```

4.2.1. Time Complexity Anlysis

Find the median of medians: $T\left(\frac{n}{5}\right)$

$T(n) \leq cn + T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right)$

Time complexity: $O(n)$, prove with induction.

Remarks: Deterministic selection has a bigger constant for average time complexity but has a better worst case time complexity.

4.3. Comparing R-Select and D-Select

	R-Select	D-Select
Choose pivot	$O(1)$	$T\left(\frac{5}{n}\right)$
Partition	$O(n)$	$O(n)$
Max size of subproblem	$\frac{3}{4}n$	$\frac{7}{10}n$
Expression	$T(n) \leq T\left(\frac{3}{4}n\right) + O(n)$	$T(n) \leq cn + T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right)$
Time complexity	$O(n)$	$O(n)$