Notes for ECE2810J

Tianzong Cheng

Contents

1. Asymptotic Algorithm Analysis

- Big O Notation: upper bound
  - $f(n) = O(g(n)) \Leftrightarrow 0 \leq f(n) \leq cg(n), \forall n \geq n_0$
- Big Omega Notation: lower bound
- Big Theta Notation: tight bound
  - $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) = \Omega(g(n))$

1.1. Master Theorem

For $T(n) \leq aT\left(\frac{n}{b}\right) + O\left(n^d\right)$,

- If $a < b^d$, $T(n) = O\left(n^d\right)$;
- If $a = b^d$, $T(n) = O\left(n^d \log n\right)$
- If $a > b^d$, $T(n) = O\left(n^{\log_b a}\right)$

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
```

```
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	$O(n)$	$O(n)$

5. Hashing

5.1. Basics

Load factor: $L = \frac{n}{m}$, where n is the number of keys and m is the number of buckets.

$h(t) = c(t(\text{key}))$

- $t(\text{key})$ converts the key into an integer
- $c(\text{code})$ maps the integer to a bucket

5.2. Collision Resolution

5.2.1. Separate Chaining

Idea: Each table entry is a linked list.

5.2.2. Open Addressing

Idea: Collisions are resolved by probing.

Remove: Mark the entry as deleted rather than removing.

- Linear probing: $h_i(k) = (h(k) + i) \bmod m$
- Quadratic probing: $h_i(k) = (h(k) + i^2) \bmod m$
- Double hashing: $h_i(k) = (h(k) + i \cdot g(k)) \bmod m$

	Linear Probing	Quadratic Probing	Double Hashing
Successful	$\frac{1}{2}\left(1 + \frac{1}{1-L}\right)$	$\frac{1}{L} \ln\left(\frac{1}{1-L}\right)$	$\frac{1}{L} \ln\left(\frac{1}{1-L}\right)$
Unsuccessful	$\frac{1}{2}\left(1 + \left(\frac{1}{1-L}\right)^2\right)$	$\frac{1}{1-L}$	$\frac{1}{1-L}$

Summary: quadratic probing and double hashing reduces clustering.

5.2.3. Comparison

- If resizing is frequent, open addressing is better.
- If removing items is needed, separate chaining is better.

5.3. Hash Table Size

Calculate the minimum table size from the load factor, and pick **a prime number** which is larger than the minimum size.

5.4. Rehashing

Goal: resize the table when the load factor exceeds a threshold.

Usually $t(\text{key})$ is the same, and $c(\text{code})$ is different.

Amortized analysis: A method of analyzing algorithms that considers the entire sequence of operations of the program. The idea is that while certain operations may be costly, they **don't occur frequently**; the less costly operations are much more than the costly ones in the long run. Therefore, the cost of those expensive operations is averaged over a sequence of operations.

5.5. Universal Hashing

TODO

5.6. Bloom Filter

Goal: check whether a key is in a set without storing the whole set

5.6.1. Comparison to Hash Tables

- Pros
  - More space efficient
- Cons
  - Can't store an associated object, i.e., no key-value pair
  - No deletion
  - Small false positive probability, but no false negative

5.6.2. Algorithm

Components:

- An array of n bits
  - $n = b|S|$, where b is small real number.
- k hash functions

Operations:

- Insert: set $A[h_i(x)]$ to 1 for all i (hash functions)
- Find: return true if $A[h_i(x)]$ is 1 for all i.

5.6.3. Analysis of Error Probability

$P[A[j] = 0] = \left(1 - \frac{1}{n}\right)^{k|S|} \Rightarrow P[A[j] = 1] = 1 - \left(1 - \frac{1}{n}\right)^{k|S|} \approx 1 - e^{-\frac{k}{b}}$

False positive rate: $\varepsilon \approx \left(1 - e^{-\frac{k}{b}}\right)^k$

For a fixed b, ε is minimized when $k = (\ln 2) \cdot b$.

6. Trees

6.1. Concepts

- Sibling: nodes that share the same parent
- Height of node: length of the longest path from the node to a leaf
- Height of tree / Depth of tree: height of the root
- Number of levels of a tree: height of tree + 1
- Degree of node: number of children
- Degree of tree: the maximum degree of a node in a tree

Binary tree:
- Proper: every node has 0 / 2 children
- Complete:
  - every level except the lowest level is fully populated
  - the lowest level is populated from left to right
- Perfect: fully populated

6.2. Binary Tree Traversal

- Pre-order: **node**, left, right
- In-order: left, **node**, right
- Post order: left, right, **node**

We can determine one tree from in-order traversal and one of pre-order traversal and post-order travsersal. Several trees can have the same pre-order travsersal and post-order traversal.

Rebuild method: determine root from pre-order or post-order traversal, and then divide left subtree and right subtree by in-order traversal.

7. Priority Queue and Heap

7.1. Time Complexity of Priority Queue Implemented with Heap

- `isEmpty`, `size`, `getMin`: $O(1)$
- `enqueue`, `dequeueMin`: $O(\log n)$ in the worst case

7.2. Binary Heap

A binary heap is a complete binary tree.

For any node `v`, the key of `v` is smaller than or equal to the keys of any descendants of `v`. Thus, the root is always the smallest element in the tree.

The height of a binary heap (a complete binary tree) is $\lfloor \log_2(n+1) \rfloor - 1$.

Elements are stored in a level-order traversal of the tree.

7.2.1. Operations

Insert:

1. Insert as the rightmost leaf (last in the array);
2. **Percolate up** the new item to an appropriate spot.

```
void minHeap::percolateUp(int id) {
  while (id > 1 && heap[id / 2] > heap[id]) {
    swap(heap[id], heap[id / 2]);
    id = id / 2;
  }
}
```

Dequeue minimum:

1. Move the item in the rightmost leaf of the tree to the root `swap(heap[1], heap[size--])`
2. Percolate down the recently moved item at the root to its proper place to restore heap property. For each subtree, if the root has a larger search key than either of its children, swap the item in the root with that of the smaller child.

```
void minHeap::percolateDown(int id) {
  for (j = 2 * id; j <= size; j = 2 * id) {
    if (j < size && heap[j] > heap[j + 1]) {
      j++;
    }
    if (heap[id] <= heap[j]) {
      break;
    }
    swap(heap[id], heap[j]);
    id = j;
  }
}
```

7.2.2. Initializing a Min Heap

Idea: put the entries into a complete binary tree and run percolate down intelligently, which is also called **heapify**.

Worst time complexity: $O(n)$

Starting at the rightmost array position that has a child, percolate down all nodes in reverse level-order.

```
MinHeap(const vector<int> &arr) : heap(arr) {
  for (int i = (heap.size() / 2) - 1; i >= 0; i--) {
    percolateDown(i);
  }
}
```

7.3. Fibonacci Heap

Time complexity comparison between: binary heap (worst case) and Fibonacci heap (amortized analysis).

Operation	Binary Heap	Fibonacci Heap
insert	$\Theta(\log n)$	$\Theta(1)$
extractMin	$\Theta(\log n)$	$O(\log n)$
getMin	$\Theta(1)$	$\Theta(1)$
makeHeap	$\Theta(1)$	$\Theta(1)$
union	$\Theta(n)$	$\Theta(1)$
decreaseKey	$\Theta(\log n)$	$\Theta(1)$

7.3.1. Idea

A Fibonacci heap is a collection of rooted trees, each as a min heap. However, the min heap here can have degree larger than 2.

- Each node has
  - a pointer to its parent
  - a pointer to **one of its children**
  - degree
- Children are linked by circular, doubly linked list
- Roots are also linked by circular, doubly linked list, which is called **root list**

When we perform an `extractMin` operation, it will go through the entire root list and consolidate nodes to reduce the size of the root list. Overall idea: the operations on Fibonacci heaps **delay work as long as possible**.

7.3.2. Operations

7.3.2.1. Extract Minimum

1. Remove min and concatenate its children into root list
2. Consolidate the root list. Merge trees until every root in the root list has a distinct degree
3. Link all the roots in array `A` together; update `H.min`

Size of `A` is `D(n) + 1`, where `D(n)` is the maximum degree of any node in an `n`-node Fibonacci heap. $D(n) = \lfloor \log_{\varphi} n \rfloor$, where $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$

```cpp
void consolidate() {
  int maxDegree = static_cast<int>(log2(numNodes)) + 1;
  std::vector<FibonacciNode *> degreeTable(maxDegree, nullptr);

  FibonacciNode *current = minNode;
  std::vector<FibonacciNode *> toVisit;

  do {
    toVisit.push_back(current);
    current = current->right;
  } while (current != minNode);

  for (FibonacciNode *node : toVisit) {
    int degree = node->degree;
    while (degreeTable[degree] != nullptr) {
      FibonacciNode *sameDegreeNode = degreeTable[degree];
      if (node->key > sameDegreeNode->key) {
        std::swap(node, sameDegreeNode);
      }
      unionNodes(sameDegreeNode, node);
      degreeTable[degree] = nullptr;
      degree++;
    }
    degreeTable[degree] = node;
  }

  // Find minNode
}
```

7.3.2.1.1. Amortized Analysis

TODO

7.3.2.2. Decrease Key

If the min heap property is violated:

1. Cut between the node and its parent
2. Move the subtree to the root list
3. Change `H.min` pointer if necessary
4. If a node n not in the root list has lost a child for the second time, the subtree rooted at `n` should also be cut from `n` 's parent and move to the root list

Purpose: This balancing through cascading cuts is essential for preserving the amortized time complexity of key Fibonacci heap operations, ensuring they stay efficient.