1 Basic Algorithms

1.1 Sorting

1.1.1 Selection Sort

Method The algorithm finds the i^{th} smallest number and swap it with the i^{th} number in the array.
Implementation

Algorithm 1 selectionSort(A)

```
for $i = 1$ to $n - 1$ do
  $pt \leftarrow i$
  for $j = i + 1$ to $n$ do
    if $A[i] < A[pt]$ then
      $pt \leftarrow j$   \\keep finding the smallest number in the rest of the array
    end if
    swap($A[i], A[pt]$)
  end for
end for
```

Complexity

Average

$$E[T_i] = \Theta(\frac{1}{\frac{1}{n-i}}) = \Theta(n-i) \ \rightarrow \ E[T] = \sum_{i=1}^{n-1} T_i = \Theta(n^2) \quad (1)$$

Ideal(array is already sorted)

$$T = \sum_{i=1}^{n-1} \Theta(1) = \Theta(n) \quad (2)$$

1.1.2 Bubble Sort

Method The bubble sort algorithm selects the largest number in an array at one time. The largest number ‘floats’ onto the top like a bubble.
Implementation

Algorithm 2 bubbleSort(A)

```
for $i = 1$ to $n$ do
  for $j = 1$ to $n - i - 1$ do
    if  $A[j] > A[j + 1]$ then
      swap($A[j], A[j + 1]$)
    end if
  end for
end for
```

Complexity For every floating up process, if the magnitude of the current array is k, then the floating process has the time cost

$$\begin{aligned} E[T_k] &= E[\text{swapping times}] \cdot \Theta(1) \\ &= \left(\frac{1}{k}\sum_{j=1}^{k}(k-j)\right) \cdot \Theta(1) = \Theta(k) \end{aligned} \tag{3}$$

Then the total time complexity is

$$E[T] = \sum_{k=1}^{n} T_k = \Theta(n^2) \tag{4}$$

1.1.3 Insertion Sort

Method Insertion sort separates the array into the unsorted part and the sorted one. Each time it deals with the first element in the unsorted part and find the right place for it.
Implementation

Algorithm 3 insertionSort(A)

```
Require: $n = A.\text{size} \geq 2$
  for $i = 2$ to $n$ do
    $key \leftarrow A[i]$
    $j \leftarrow i - 1$
    while $j > 0$ and $A[j] > key$ do
      $A[j+1] \leftarrow A[j]$
      $j = j - 1$
    end while
  end for
```

Complexity Same as selection sort, the time complexity of insertion sort is

$$E[T] = \sum_{k=1}^{n} T_k = \Theta(n^2), \quad T_k = \Theta(k) \tag{5}$$

1.1.4 Counting Sort

Method The counting sort requires an extra auxiliary array to store the elements temporarily. For every value in the elements, the algorithm counts the number of times it appears, store it in the extra array, and thus calculating its prefix sum (index). Then the elements will be assigned back to the original array.
Implementation

Algorithm 4 countingSort(A)

```
Require: $low \leftarrow$ smallest element,   $up \leftarrow$ largest element,   $C.\text{size} = w = up - low + 1$
  for $i = 1$ to $w$ do
    $C[i] = 0$
  end for
  for $i = 1$ to $n$ do
    $C[A[j]] \leftarrow C[A[j]] + 1$
  end for
  $j \leftarrow 1$
  for $i = 1$ to $w$ do
    while $C[i] \neq 0$ do
      $A[j] \leftarrow i$
      $C[i] \leftarrow C[i] - 1$
      $j \leftarrow j + 1$
    end while
  end for
```

Complexity Scanning and filling the auxiliary array, as well as refilling the original array both take linear time. Thus the time complexity is

$$E[T] = \Theta(n + w) \tag{6}$$

1.1.5 Fast Sort

Method The fast sort algorithm is based on a recursive manipulation, namely PARTITION. It selects the last element in the array (pivot), and separates it into the left part (all elements in it are smaller than the pivot) and the right part (vice versa).

Algorithm 5 PARTITION(A, *left*, *right*, *pivot*)

```
  if $A.\text{size} = right - left + 1 = 1$ then
    $A \leftarrow A$
  else
    swap($A[pivot], A[right]$)
    $j \leftarrow left$
    for $i = left$ to $right - 1$ do
      if $A[i] \leq A[right]$ then
        swap($A[i], A[j]$)
        $j \leftarrow j + 1$
      end if
    end for
    swap($A[right], A[j]$)
    return $j$
  end if
```

Hence the fast sort algorithm can be aligned by recursively calling the PARTITION manipulation.

Algorithm 6 fastSort(A)

```
Require: $A$.size $= n$,   $middle \leftarrow$ PARTITION($A$, 1, $n$, random(1, $n$))
  $A$.left $\leftarrow A.[1, \ldots, middle - 1]$
  $A$.right $\leftarrow A.[middle + 1, \ldots, n]$
  fastSort($A$.left, 1, $middle - 1$)
  fastSort($A$.right, $middle + 1$, $n$)
```

Complexity The average time complexity can be given by the recurrence

$$E[T(n)] = E[T(j)] + E[T(n-j)] + \Theta(n) \tag{7}$$

where j is the index of the eventual position of the randomly selected pivot. The expected height of the recursion tree is obviously $\Theta(\log n)$. Hence the time complexity is

$$E[T(n)] = \Theta(n\log n) \tag{8}$$

1.1.6 Radix Sort

Method Radix sort requires the elements in the array to be of same digits. The algorithm sort the elements according to one definite digit at a time with a stable sorting method (usually the counting sort).
Implementation

Algorithm 7 radixSort(A)

```
Require: $A[i]_{1 \le i \le n}$ has same digits $d$
  for $i = 1$ to $d$ do
    sort the elements according to the $d^{th}$ digit
  end for
```

Complexity The inner sorting requires a stable sorting method. A usual selection is the counting sort. Denote the magnitude to the range of numbers in all the digits as w, and the quantity of the array elements as n. The upper bound of time is intuitive:

$$E[T] = O(nw) \tag{9}$$

1.1.7 Bucket Sort

Method Bucket sort deals with float numbers that falls into the interval $[0, 1]$. For integers, we can use a definite function to map them into this interval. Then we separate $[0, 1]$ into n sub-intervals (buckets), inside which we apply the elements using insertion sort (an ideal stable sorting method). Then we sort the buckets using radix sort depending on the first non-zero digits of the first elements in the buckets.

Implementation

Algorithm 8 bucketSort(A)

```
Require: $A[i], 1 \leq i \leq n$ hashed into $[0, 1]$
Require: $bucket[i] \leftarrow$ elements in $[(i-1)/n, i/n]$
  for $i = 1$ to $n$ do
    insertionSort($bucket[i]$)
  end for
  radixSort($A$) according to $bucket[i][1]$
```

Complexity Inside each bucket, the insertion sort takes square time complexity, and the external sorting takes linear time. Hence, we obtain the equation that

$$T(n) = \Theta(n) + \sum_{i=1}^{n} \Theta(n_i^2) \tag{10}$$

The distribution of the elements only affects the last term, so we are interested in $E[\sum_{i=1}^{n} \Theta(n_i^2)]$.

$$\begin{aligned}
E[\sum_{i=1}^{n} \Theta(n_i^2)] &= \Theta(\sum_{i=1}^{n} E(n_i^2)) \\
&= \Theta\left[\sum_{i=1}^{n}\sum_{j=0}^{n} j^2 p(n_i^2 = j)\right] = \Theta\left[\sum_{i=1}^{n}\sum_{j=0}^{n} j^2((\frac{1}{n})^j(1-\frac{1}{n})^{n-j})\right] \\
&= \Theta\left[\sum_{i=1}^{n} \frac{\mathrm{d}}{\mathrm{d}t}\sum_{j=0}^{n} e^{tj}(\frac{1}{n})^j(1-\frac{1}{n})^{n-j}\right] \\
&= \Theta(\sum_{i=1}^{n} 2 - \frac{1}{n}) = \Theta(n)
\end{aligned} \tag{11}$$

Hence, $T(n) = \Theta(n)$. The bucket sort is a linear-time sorting.

2 Mathematics

2.1 Euclid Algorithm

In order to find the greatest common divisor for integer a, b, where $a \geq b$, Euclid algorithm gives

$$\gcd(a, b) = \gcd(a \bmod b, b) \tag{12}$$

The proof is almost intuitive, as

$$\exists\ k,\ a = kb + n\big|_{n \leq b} \ \rightarrow\ \gcd(a, b) = \gcd(kb + n, b) = \gcd(n, b) = \gcd(a \bmod b, b) \tag{13}$$

The following gives the implementation

Algorithm 9 gcd(a,b)

```
Require: $a \geq b$
  if $b = 0$ then
    return $a$
  else
    return gcd($b, a \bmod b$)
  end if
```

2.2 Chinese Remainder Theorem

Given a linear residual equation

$$\begin{aligned} x &\equiv a_1 (\bmod\ n_1) \\ x &\equiv a_2 (\bmod\ n_2) \\ &\vdots \\ x &\equiv a_k (\bmod\ n_k) \end{aligned} \tag{14}$$

where

$$\gcd(n_1, n_2, \ldots, n_k) = 1 \tag{15}$$

Solution Chinese Remainder Theorem gives

$$x = \left(\sum_{i=1}^{k} c_i a_i \right) \bmod n \tag{16}$$

where

$$m_i = \frac{\prod_{j=1}^{k} n_j}{n_i}, \quad (m_i m_i^{-1}) \bmod n_i = 1, \quad c_i = m_i m_i^{-1} \tag{17}$$

Rationality Firstly we calculate

$$\begin{aligned} (\sum_{j=1}^{k} c_j a_j) \bmod n_i &= \left(\sum_{j=1}^{k} m_j m_j^{-1} a_j \right) \bmod n_i \\ &= \sum_{j=1}^{k} a_j m_j^{-1} \left(\frac{\prod_{p=1}^{k} n_k}{n_j} \right) \bmod n_i \\ &= a_i m_i m_i^{-1} \bmod n_i \end{aligned} \tag{18}$$

Thus

$$\begin{aligned} x &\equiv a_i m_i m_i^{-1} (\bmod\ n_i) \\ &\equiv a_i (\bmod\ n_i) \end{aligned} \tag{19}$$