Data structures and Algorithms
Sorting

Pham Quang Dung

Hanoi, 2012

Outline

Introduction to sorting

- Put elements of a list in a certain order
- Designing efficient sorting algorithms is very important for other algorithms (search, merge, etc.)
- Each object is associated with a key and sorting algorithms work on these keys.
- Two basic operations that used mostly by sorting algorithms
  - Swap(a, b): swap the values of variables a and b
  - Compare(a, b): return
    - true if a is before b in the considered order
    - false, otherwise.
- Without loss of generality, suppose we need to sort a list of numbers in nondecreasing order

- A sorting algorithm is called **in-place** if the size of additional memory required by the algorithm is $\mathcal{O}(1)$ (which does not depend on the size of the input array)
- A sorting algorithm is called **stable** if it maintains the relative order of elements with equal keys
- A sorting algorithm uses only comparison for deciding the order between two elements is called **Comparison-based sorting algorithm**

Outline

Insertion Sort

- At iteration k, put the k^{th} element of the original list in the right order of the sorted list of the first k elements ($\forall k = 1, \ldots, n$)
- Result: after k^{th} iteration, we have a sorted list of the first k^{th} elements of the original list

```
void insertion_sort(int a[], int n){
  int k;
  for(k = 2; k <= n; k++){
    int last = a[k];
    int j = k;
    while(j > 1 && a[j-1] > last){
      a[j] = a[j-1];
      j--;
    }
    a[j] = last;
  }
}
```

Listing 1: insertion sort

Outline

Selection Sort

- Put the smallest element of the original list in the first position
- Put the second smallest element of the original list in the second position
- Put the third smallest element of the original list in the third position
- ...

```
void selection_sort(int a[], int n){
  for(int k = 1; k <= n; k++){
    int min = k;
    for(int i = k+1; i <= n; i++)
      if(a[min] > a[i])
        min = i;
    swap(a[k],a[min]);
  }
}
```

Listing 2: selection sort

Outline

- Pass from the beginning of the list: compare and swap two adjacent elements if they are not in the right order
- Repeat the pass until no swaps are needed

```
void bubble_sort(int a[], int n){
  int swapped;
  do{
    swapped = 0;
    for(int i = 1; i < n; i++)
    if(a[i] > a[i+1]){
      swap(a[i],a[i+1]);
      swapped = 1;
    }
  }while(swapped == 1);
}
```

Listing 3: bubble sort

Outline

Merge sort

Divide-and-conquer

- Divide the original list of $n/2$ into two lists of $n/2$ elements
- Recursively merge sort these two lists
- Merge the two sorted lists

Merge sort

```
void merge(int a[], int L, int M, int R){
  // merge two sorted list a[L..M] and a[M+1..R]
  int i = L;// first position of the first list a[L..M]
  int j = M+1;// first position of the second list a[M+1..R]
  for(int k = L; k <= R;k++){
    if(i > M){// the first list is all scanned
      TA[k] = a[j]; j++;
    }else if(j > R){// the second list is all scanned
      TA[k] = a[i]; i++;
    }else{
      if(a[i] < a[j]){
        TA[k] = a[i]; i++;
      }else{
        TA[k] = a[j]; j++;
      }
    }
  }
  for(int k = L; k <= R; k++)
    a[k] = TA[k];
}
```

Merge sort

```
void merge_sort(int a[], int L, int R){
  if(L < R){
    int M = (L+R)/2;
    merge_sort(a,L,M);
    merge_sort(a,M+1,R);
    merge(a,L,M,R);
  }
}
```

Outline

Quick sort

- Pick an element, called a **pivot**, from the original list
- Rearrange the list so that:
  - All elements less than **pivot** come before the **pivot**
  - All elements greater or equal to to **pivot** come after **pivot**
- Here, **pivot** is in the right position in the final sorted list (it is fixed)
- Recursively sort the sub-list before **pivot** and the sub-list after **pivot**

Quick sort

```
void quick_sort(int a[], int L, int R){
  if(L < R){
   int index = (L+R)/2;
    index = partition(a,L,R,index);
    if(L < index)
      quick_sort(a,L,index-1);
    if(index < R)
      quick_sort(a,index+1,R);
  }
}
```

Listing 4: Quick sort algorithm

Quick sort

```
int partition(int a[], int L, int R, int indexPivot){
  int pivot = a[indexPivot];
  swap(a[indexPivot],a[R]);// put the pivot in the end of the list
  int storeIndex = L; // store the right position of pivot at the end of the partition procedure

  for(int i = L; i <= R-1; i++){
    if(a[i] < pivot){
      swap(a[storeIndex],a[i]);
      storeIndex++;
    }
  }
  swap(a[storeIndex],a[R]); // put the pivot in the right position and return this position

  return storeIndex;
}
```

Listing 5: partition

Outline

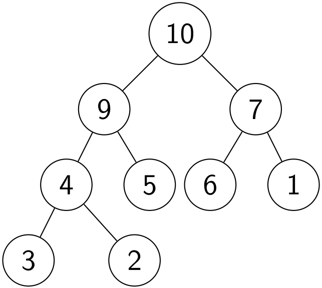
Heap sort

Sort a list $A[1..N]$ in nondecreasing order

1. Build a heap out of $A[1..N]$
2. Remove the largest element and put it in the N^{th} position of the list
3. Reconstruct the heap out of $A[1..N-1]$
4. Remove the largest element and put it in the $N-1^{th}$ position of the list
5. ...

Heap sort - Heap structure

- Shape property: Complete binary tree with level L
- Heap property: each node is greater than or equal to each of its children (max-heap)



1	2	3	4	5	6	7	8	9
10	9	7	4	5	6	1	3	2

Heap sort

- Heap corresponding to a list $A[1..N]$
  - Root of the tree is $A[1]$
  - Left child of node $A[i]$ is $A[2 * i]$
  - Right child of node $A[i]$ is $A[2 * i + 1]$
  - Height is $logN + 1$
- Operations
  - Build-Max-Heap: construct a heap from the original list
  - Max-Heapify: repair the following binary tree so that it becomes Max-Heap
    - A tree with root $A[i]$
    - $A[i] < max(A[2 * i], A[2 * i + 1])$: heap property is not hold
    - Subtrees rooted at $A[2 * i]$ and $A[2 * i + 1]$ are Max-Heap

Heap sort

```
void heapify(int a[], int i, int n){
  // array to be heapified is a[i..n]
  int L = 2*i;
  int R = 2*i+1;
  int max = i;
  if(L <= n && a[L] > a[i])
    max = L;
  if(R <= n && a[R] > a[max])
    max = R;
  if(max != i){
    swap(a[i],a[max]);
    heapify(a,max,n);
  }
}
```

Listing 6: heapify

Heap sort

```
void buildHeap(int a[], int n){
    // array is a[1..n]
  for(int i = n/2; i >= 1; i--){
    heapify(a,i,n);
  }
}

void heap_Sort(int a[], int n){
    // array  is a[1..n]
  buildHeap(a,n);
  for(int i = n; i > 1; i--){
    swap(a[1],a[i]);
    heapify(a,1,i-1);
  }
}
```

Listing 7: heapSort

Outline

Counting sort

- **Input**: array of n integers $a_1, \ldots, a_n$, where $0 \neq a_i \neq k$ with $k = \mathcal{O}(n)$
- **Main idea**:
  - For each element x, compute the rank $r[x]$ as number of elements of the array which is less than or equal to x
  - Place x at position $r[x]$

Counting sort

a

6	2	6	2	6	1	1	4	6	5	1	1

r

1	2	3	4	5	6
4	6	6	7	8	12

								6	6	6	6

							5	6	6	6	6

						4	5	6	6	6	6

				2	2	4	5	6	6	6	6

1	1	1	1	2	2	4	5	6	6	6	6

Counting sort

```
void countingSort(int a[], int r[], int c[], int n, int k){
  // a [1..n] is the array to be sorted
  // n is the number of elements of the array a
  // k is the maximum of a, and 0 is minimum of a

  // count r[i] - the number of elements of a having value i
  for(int i = 0; i <= k; i++) r[i] = 0;
  for(int i = 1; i <= n; i++) r[a[i]]++;
  // compute rank r[x] of x
  for(int x = 1; x <= k; x++) r[x] = r[x] + r[x-1];

  // sort
  for(int i = n; i >= 1; i--){
    c[r[a[i]]] = a[i];// place a[i] in its right position
    r[a[i]] = r[a[i]]-1;// reduce rank of a[i] by one
  }
}
```

Outline

Radix sort

- Not comparison-based sorting algorithm
- Each key (integer) is represented by a sequence of d numerical digits in a given radix (e.g., radix is 10: $\overline{a_d a_{d-1} \dots a_1}$)
- Principle
  - Take the least significant digit of each key
  - Group the keys based on that digit while keep the original order of keys (**stable** sort)
  - Repeat the grouping process with each more significant digit

Radix sort

Algorithm 1: RadixSort(A, d)

foreach $i \in 1..d$ **do**
    Sort A based on the i^{th} digits of keys using **stable** sort;

Radix sort

Example

$A[1..10] = 2980, 0020, 0242, 3002, 1145, 1045, 2626, 1005, 3180, 4146$

1. Step 1: 298**0**,002**0**,318**0**,024**2**,300**2**,114**5**,104**5**,100**5**,262**6**,414**6**
2. Step 2: 30**0**2,10**0**5,00**2**0,26**2**6,02**4**2,11**4**5,10**4**5,41**4**6,29**8**0,31**8**0
3. Step 3: 3**0**02,1**0**05,0**0**20,1**0**45,1**1**45,4**1**46,3**1**80,0**2**42,2**6**26,2**9**80
4. Step 4: **0**020,**0**242,**1**005,**1**045,**1**145,**2**626,**2**980,**3**002,**3**180,**4**146

Radix sort

```
void radix_sort_10(long a[], int n){
  int max = -10000;
  for(int i = 0; i < n; i++) if(max < a[i]) max = a[i];
  long tmp[MAX_SIZE];
  int exp = 1;
  while(max/exp > 0){
    int bin_sz[10] = {0};
    int idx[10];
    for(int i = 0; i < n; i++){
      int c = (a[i]/exp) % 10;
      bin_sz[c]++;
      tmp[i] = a[i];
    }
    idx[0] = 0;
    for(int i = 1; i < 10; i++)
    idx[i] = idx[i-1] + bin_sz[i-1];
    for(int i = 0; i < n; i++){
      int c = (tmp[i]/exp)%10;
      a[idx[c]] = tmp[i];
      idx[c]++;
    }
    exp = exp*10;
  }
}
```

Outline

Bucket sort

- Assume a uniform distribution on the input
- The input (array $A[1..n]$) is generated by a random process that distributes uniformly and independently over the interval [0,1)
- Main idea
  - Interval [0,1) is divided into n equal-size subintervals (or buckets)
  - Distribute $A[1..n]$ into the buckets
  - Sort elements in each bucket
  - Concatenate the sorted lists of the buckets in order to establish the sorted list of the original array

Bucket sort

Algorithm 2: BUCKET-SORT(A)

Let $B[0..n-1]$ be a new array;
foreach $i = 0, \ldots, n-1$ **do**
    Make B[i] an empty list;
foreach $i = 1..n$ **do**
    Insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$;
foreach $i = 0, \ldots, n-1$ **do**
    Sort list $B[i]$ with insertion sort;
Concatenate the list $B[0], B[1], \ldots, B[n-1]$ together in order;

- Analysis the **average-case running time**
- Let n_i be the random variable denoting the number of element placed in bucket $B[i]$
- $T(n) = \Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}(n_i^2)$
- Average-case running time is $E[T(n)] = E[\Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}(n_i^2)] = \Theta(n) + \sum_{i=1}^{n-1} \mathcal{O}(E[n_i^2])$
- We'll prove that $E[n_i^2] = 2 - \frac{1}{n}$ (next slide)

Bucket sort - Analysis

$$X_{ij} = \begin{cases} 1, & \text{if } A[j] \text{ falls in bucket } i \\ 0, & \text{otherwise} \end{cases}$$

- $n_i = \sum j = 1^n X_{ij}$
- $E[n_i^2] = E[(\sum_{j=1}^n X_{ij})^2] =$
  $E[\sum_{j=1}^n X_{ij}^2 + \sum_{1\le j\le n} \sum_{k\in\{1,\dots,n\}\setminus\{j\}} X_{ij}X_{ik}] =$
  $\sum_{j=1}^n E[X_{ij}^2] + \sum_{1\le j\le n} \sum_{k\in\{1,\dots,n\}\setminus\{j\}} E[X_{ij}X_{ik}]$
- X_{ij} is 1 with probability $\frac{1}{n}$ and 0 otherwise, therefor
  $E[X_{ij}^2] = 1^2 * \frac{1}{n} + 0^2 * (1 - \frac{1}{n}) = \frac{1}{n}$
- When $k \neq j$, X_{ij} and X_{ik} are independent, hence
  $E[X_{ij} * X_{ik}] = E[X_{ij}]E[X_{ik}] = \frac{1}{n} * \frac{1}{n} = \frac{1}{n^2}$
- Finally, we have
  $E[n_i^2] = 2 - \frac{1}{n} \Rightarrow E[T(n)] = \Theta(n) + n * \mathcal{O}(2 - \frac{1}{n}) = \Theta(n)$