3 Advanced Sorting

Readings: CLRS 7,8 (material), CLRS 5.2,C.3 (math review)

3.1 Randomized QuickSort

Recall the QuickSort algorithm:

```
QuickSort(from,to):
  if to>from
  | i:=Partition(from,to);
  | QuickSort(from,i-1);
  | QuickSort(i+1,to);
```

The Partition function chooses a pivot element from $A[from..to]$ and partitions the array into elements that are smaller than or equal to the pivot, and elements that are larger than pivot. In such partition, the pivot element itself will end up at its proper position in the sorted array, and its index is returned by the Partition. The following implementation of the Partition chooses deterministically the right-most element as the pivot.

```
Partition(from,to):
  // post: pivot is at its correct position A[ret]
  //       from<=j<=ret => A[j]<=pivot
  //       ret<j<=to => A[j]>pivot
  pivot:=A[to];
  i:=from-1;
  for j:=from to to
  | // inv: from<=k<=i => A[k]<=pivot
  | //      i<k<j => A[k]>pivot
  | if A[j]<=pivot
  | | i:=i+1
  | | swap(A[j],A[i])
  return i;
```

Figure 7.1 in CLRS demonstrates the function of the Partition.

Running time:

- **Worst-case:** $\Theta(n^2)$
  Consider for example the case when the array is already sorted, or when all the elements of the array are the same
- **Best-case:** $\Theta(n \log n)$
  This happens if we always manage to split the array into two roughly equally-sized halves.
- **Average-case:** $\Theta(n \log n)$
  Average running time over all permutations of n elements. Formal proof is complex, and we will not return to it here (see some notes in CLRS 7.2).

Randomizing Partition: Consider picking the pivot *randomly* from all available array elements, instead of always choosing the last element:

```
RandomizedPartition(from,to):
  // post: pivot is at its correct position A[ret]
  //       from<=j<=ret => A[j]<=pivot
  //       ret<j<=to => A[j]>pivot
  *** change here ***
  swap(A[to],A[random(from,to)]); // pick a random index as a pivot
  *******************
  pivot:=A[to];
  i:=from-1;
  for j:=from to to
  | // inv: from<=k<=i => A[k]<=pivot
  | //      i<k<j => A[k]>pivot
  | if A[j]<=pivot
  | | i:=i+1
  | | swap(A[j],A[i])
  return i;
```

How is such a change going to help?

- In some cases, the algorithm can still run in quadratic time.
- HOWEVER: If all elements are distinct, there are no *consistently bad* input, because everything depends on the choice of pivot, and since we are using randomization, this changes every time we run the algorithm.
- Further modification to the algorithm is needed to improve running time if there are many elements with the same value.
- We will use notion of *expected running time* to capture the running time of randomized algorithms.

3.1.1 Detour: Expected Running Time

Review of expectations:

- A **random variable** X is a function that associates a numerical value with outcome of some random event.

  Example: A roll of a dice is a random variable with possible values $1, 2, \ldots, 6$. If the dice is fair, $\Pr(X = i) = 1/6$. If we want to look at multiple rolls of a dice, we use multiple random variables $X_1, X_2, X_3, \ldots$.

  If we are looking at a sum of numbers on three dice, we have a random variable for each dice: X_1, X_2, X_3. Each of these random variables has values between 1 and 6. The outcome of the whole experiment is again a random variable $X = X_1 + X_2 + X_3$ with values between 3 and 18.

- An **indicator random variable** Y for some condition C is either 1, if the condition C is satisfied, or 0, if the condition C is not satisfied.

  Example: If Y is an indicator random variable for condition "the dice rolls number 6", then $Y = 1$ if the roll of the dice results in number 6, and $Y = 0$ otherwise.

- **Expected value** of a random variable X is defined as follows:

$$E[X] = \sum_x x \cdot \Pr(X = x)$$

Example: Expected value of a roll of a fair dice is:

$$E[X] = \sum_{i=1}^{6} i \cdot \Pr(\text{dice rolls to side } i) = \sum_{i=1}^{6} i \cdot 1/6 = \frac{1+2+3+4+5+6}{6} = 3.5$$

Expected value of an indicator random variable Y for some condition C is:

$$E[Y] = 0. \Pr(\neg C) + 1. \Pr(C) = \Pr(C)$$

- **Linearity of expectation** is a rule that allows us to compute an expected value of sums of random variables:

  $$E[X+Y] = E[X] + E[Y]$$

  Example: Expected value of the sum of three dice rolls is $E[X] = E[X_1 + X_2 + X_3] = E[X_1] + E[X_2] + E[X_3] = 3 \cdot 3.5 = 10.5$

Running time of randomized algorithms:

- The **running time** $T_A(x, R)$ **of a randomized algorithm** A for a particular input x and a *sequence of random numbers* $r = (r_1, r_2, \ldots)$ is the time that the algorithm requires to solve the input x, if the random numbers generated during the program's run are $r_1, r_2, \ldots$ (in that order).

- The **expected running time** $T_A^{(exp)}(x)$ **of a randomized algorithm** A for a particular input x is the expected value of random variable $T_A(x, R)$, where R is a random variable corresponding to the sequence of random numbers generated during the run of the algorithm:

  $$T_A^{(\text{exp})}(x) = E[T_A(x, R)] = \sum_{r=(r_1,r_2,\ldots)} T_A(x, r) \cdot \Pr(R = r)$$

- The **worst-case expected running time** $T_A^{(exp)}(n)$ **of a randomized algorithm** is a function of the size n of the input, where $T_A^{(exp)}(n)$ is the largest expected time required to solve an input of size n:

  $$T_A^{(\text{exp})}(n) = \max\{T_A^{(exp)}(x) \mid |x| = n\}$$

3.1.2 Analyzing Randomized QuickSort

Assumption: In this analysis, we will assume that all input numbers are distinct. We can denote these numbers $z_1, z_2, \ldots, z_n$ so that $z_1 < z_2 < \ldots < z_n$. Note that at the beginning, these numbers are stored in array $A[1 \ldots n]$ in some order, while at the end of the sort we will have $A[i] = z_i$ for all i.

The running time of the randomized QuickSort is dominated by two factors:

- The number of recursive calls. In the worst case this is $O(n)$. (During each recursive call, one element needs to be chosen as a pivot, and each element is chosen as a pivot at most once.)

- The number of comparisons.

All other operations can be associated with one or the other of the two factors. Thus to compute the expected runtime of the randomized QuickSort, we need to answer the following question.

$\boxed{\textbf{What is the expected number of comparisons?}}$ Let X be the random variable representing the number of comparisons during the execution of QuickSort, and let $X_{i,j}$ be an indicator random variable for condition "numbers z_i and z_j are compared at some point during the execution of QuickSort". Note, that the numbers only get compared in the partitioning phase, always with currently selected pivot element. Therefore, any two numbers are compared at most once during the QuickSort. Thus we can write:

$$X = \sum_{i,j} X_{i,j}$$

Consider sequence of pivots in the order as they are chosen during the execution of the QuickSort. To compute expected value of $X_{i,j}$, we need to recognize that from the point of view of the two numbers z_i and z_j (without loss of generality assume $z_i < z_j$) there are three phases during the execution of the QuickSort:

1. While the chosen pivot is z_k such that $k < i$ or $k > j$, it may happen that both z_i and z_j get compared to z_k. However, they do not get compared to each other, and they both stay in the same partition. These steps in the computation do not have any bearing on the value of random variable $X_{i,j}$.
2. First time the pivot z_k is chosen so that $i \le k \le j$, the elements z_i and z_j will be separated into two different partitions. At that point, both z_i and z_j are compared to z_k.
3. After that, the elements z_i and z_j are in two different partitions and they are processed by separate recursive calls. So they will never get compared again.

So the only way z_i and z_j may get compared to each other is in phase 2. That only happens, if either z_i or z_j are chosen as a pivot. Therefore the probability that z_i and z_j get compared is equal to the following probability:

$$\Pr(z_i \text{ or } z_j \text{ chosen as a pivot} \mid \text{pivot chosen from } z_i, z_{i+1}, \ldots, z_j) = \frac{2}{j-i+1},$$

and therefore $\boxed{E[X_{i,j}] = 2/(j-i+1).}$

Now, to compute the expected number for comparisons:

$$\begin{aligned}
E[X] &= E\left[\sum_{i,j} X_{i,j}\right] = \sum_{i,j} E[X_{i,j}] \\
&= \sum_{i=1}^{n} \sum_{j=i+1}^{n} 2/(j-i+1) \\
&= 2\sum_{i=1}^{n} \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-i+1} \qquad [k := n-i+1] \\
&= 2\sum_{k=1}^{n} \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{k} \\
&= 2\sum_{k=1}^{n} (H_k - 1) = 2\sum_{k=1}^{n} H_k - 2n \\
&= 2((n+1)H_n - n) - 2n = 2(n+1)\underbrace{H_n}_{\Theta(\log n)} - 4n = \Theta(n \log n)
\end{aligned}$$

(You can find definitions and formulas for algebraic manipulation of harmonic numbers H_k in TCSCS.)

Therefore the expected number of comparisons of QuickSort is $\Theta(n \log n)$ and we have proved the following claim:

Theorem 1. *Expected running time of randomized QuickSort is* $\Theta(n \log n)$.

3.1.3 Random Number Generators

Computers are by their nature deterministic. Outcome of every instruction is well-defined and always the same for the same set of inputs. We have seen above that using random numbers can improve running time of algorithms. Where do we get the random numbers from?

$\boxed{\textbf{Pseudo-random number generators}}$ To overcome the deterministic nature of the computer and to create an illusion of randomness, we create a function RANDOM that returns a value between 0 and $max-1$. The function will have "hidden inputs" so that the output of the function will change with every call.

The sequence of numbers generated by a pseudo-random generator should "look" statistically random. This means that if we look at the sequence, we should not be able to observe any biases such as:

- Some numbers occur significantly more often than other numbers
- Odd numbers occur significantly more often than even numbers
- The sequence contains arithmetic progressions that are longer or shorted than would be expected by chance
- ...

This is often not the case for built-in random number generators. Biases introduced by such generators can negatively affect performance of randomized algorithms.

Example: Linear Congruential Generator

- We keep a hidden variable *seed*. Before we can generate pseudo-random numbers, we have to initialize *seed* to some value. The *seed* needs to be always initialized to a different value (such as value of computer clock), otherwise we will get the same sequence of numbers.

  (On the other hand, setting *seed* to a fixed value may be beneficial for program debugging.)

- A pseudo-random number is generated as follows:

```
seed = ( a*seed + c ) mod max
return seed
```

Disadvantages of the linear congruential generator:

- Values of a, c, and max must be set with care.

  For example, if we set both c and max to be even numbers, then after generating the first even number, all the rest of the numbers will also be even.

- Value of max must be large. This is because once we generate the same number for the second time, the sequence of the generated numbers will start repeating itself.

In practice, more complicated functions are used.

3.2 Lower Bound on Running Time of Comparison Based Sorting

Most of the sorts we have seen so far (Selection Sort, Insertion Sort, Merge Sort, Heap Sort, and QuickSort) are based on *comparisons*: the elements to be sorted can be accessed only using the following operations:

- comparing two elements (for simplicity assume that all comparisons are $\leq$; in practice we can of course use also $\geq$, $=$, $<$, and $>$, but the important part here is that all of them give us only an answer "true" or "false")
- moving elements around (such as: copying element, or exchanging two elements)

The best of these algorithms have worst-case running time $O(n \log n)$. In this section we show that *any correct comparison-based sorting algorithm requires at least $\Omega(n \log n)$ comparison operations*, and therefore the worst-case running time of such algorithm is at least $\Omega(n \log n)$.

This result is much different from other lower bounds we proved so far. Instead of proving lower bound on running time of a *particular algorithm A*, we claim that *any algorithm that solves the sorting problem* (as long as it is based on comparisons) must have the worst-case running time of at least $\Omega(n \log n)$. We do not make any assumptions on what the structure or the design of the algorithm is. Thus, based on our claim, it is inherently impossible to design a comparison-based sorting algorithm that would run in $\omega(n \log n)$ time.

First, we introduce so called *decision trees* that can serve as a visualization of sorting algorithm, showing all the comparisons that are made to solve every possible input of the sorting problem. For simplicity, we assume that the input contains distinct numbers $a_1, a_2, \ldots, a_n$ (however, everything can be easily extended to the scenario where some numbers can be the same).

Definition 1 (Decision trees for comparison-based sorting)**.** *The decision tree for an algorithm A is a binary tree, where each internal node of the tree corresponds to a comparison made by the algorithm. The left subtree of a node always corresponds to the continuing execution of the algorithm if the answer to the comparison is "true", and the right subtree corresponds to the continuing execution of the algorithm if the answer to the comparison is "false".*

Each leaf of the decision tree corresponds to a permutation of values $a_1, a_2, \ldots, a_n$ so that the values are sorted (i.e., it corresponds to the output of the sorting algorithm).

Figure 8.1 in CLRS shows a decision tree of Insertion Sort on a three element array. The notation $a_i : a_j$ corresponds to comparing elements a_i and a_j with comparison $a_i < a_j$ (thus left subtree always corresponds to the case when $a_i < a_j$, while right subtree corresponds to the case when $a_i \geq a_j$).

Note that for every possible output of the algorithm, the nodes on the path from the root to the leaf corresponding to that output represents the comparisons made by the algorithm to get to that output. Therefore the worst-case number of comparisons is equal to the length of the longest path from the root to a leaf, i.e. to the height of the tree.

Theorem 2. *Any comparison-based sort requires at least $\Omega(n \log n)$ comparisons in the worst case, and therefore its worst-case running time is $\Omega(n \log n)$.*

Proof. Consider a comparison-based algorithm A for sorting and its decision tree for n-element arrays. The comparison tree is a binary tree and the number of leaves is $n!$ (because there are $n!$ different possible outputs of the algorithm).

Every binary tree with heigh at most h has less than 2^h leaves. Therefore if a tree has $n!$ leaves, its height must be at least $\log(n!)$.

$$\begin{aligned}
\log(n!) &= \log(n.(n-1)\ldots 1) = \log n + \log(n-1) + \ldots + \log 1 \\
&\geq \log n + \log(n-1) + \ldots + \log(n/2) \geq \log(n/2) + \log(n/2) + \ldots + \log(n/2) \\
&= (n/2)\log(n/2) = (n/2)\log n - (n/2) \\
&= \Omega(n \log n)
\end{aligned}$$

Therefore the height of the decision tree is at least $\Omega(n \log n)$ and therefore the worst-case number of comparisons made by the algorithm A is at least $\Omega(n \log n)$. □

Note: In this section we have shown that the worst-case running time for a comparison-based sorting is at least $\Omega(n \log n)$. By similar method, the same lower bound can be proven for average-case running time and for expected running time of randomized algorithms for sorting.

This result also proves that algorithms (such as heap sort or merge sort) with running time $O(n \log n)$ are optimal—their asymptotic running time cannot be improved.

3.3 Linear-Time Sorting

In the previous section we have shown that every comparison-based sortin algorithm requires running time of at least $\Omega(n \log n)$. However, if we make some additional assumptions about the input, and introduce other operations beyond the comparisons, we can achieve a better running times.

Counting Sort Assumption: we want to sort n objects according to a key, which is an integer number from interval $[0, k]$.

```
clear array count[0..k];

for i:=1 to n
| count[A[i].key]++;

pos[0]:=1;
for i:=1 to k
| pos[i]:=pos[i-1]+count[i-1];
// now pos[i] is the first position where
// integer i will come in the sorted array B

for i:=1 to n
| B[pos[A[i].key]]:=A[i];
| pos[A[i].key]++;
```

Note: The above implementation of counting sort is a **stable sort**: if two elements x and y are equal, then their relative order is preserved after sorting. This feature is sometimes important if there is additional information associated with each number in the array.

Running time: $\Theta(n + k)$

Radix Sort Assumption: we want to sort n integer numbers which have at most d digits (can be straightforwardly adapted for strings that have at most d characters).

```
for i:=1 to d
| use counting sort to sort A[1..n] by
| the d-th least significant digit (i.e. k=10)
| // inv: array is sorted by last i digits
```

The algorithm is demonstrated in Figure 8.3 of CLRS.

Theorem 3. *After i-th iteration of radix sort the array A is sorted by i least significant digits.*

Proof. We will prove the claim by induction on value i.

- **Base case.** For $i = 0$ the claim holds trivially.
- **Induction step.** Lets assume that after $i-1$ iterations the array is sorted by $i-1$ least significant digits.

  Consider relative order of two elements a_k and a_ℓ. After the i-th iteration, we need to consider two cases:

  - If the i-th least significant digit of a_k and a_ℓ is different then the counting sort used in the i-th iteration will order a_k and a_ℓ by this digit, and their relative order will be consistent with sorted order by i least significant digits.
  - If the i-th least significant digit of a_k and a_ℓ is the same then the counting sort preserves their order (because it is a stable sort), and therefore from induction hypothesis they will be ordered by their $i-1$ least significant digits. This order is consistent with sorted order by i least significant digits (since the i-th digit is the same).

  Therefore, relative order of any two elements is consistent with the sorted order by i least significant digits, and the array after i iteration is thus sorted by i least significant digits.

□

Running time: $\Theta(nd)$

Bucket Sort Assumption: we want to sort n real numbers that are chosen independently randomly uniformly from interval $[0, 1)$.

```
bucket[0..n-1] is an array of empty lists
for i:=1 to n
| insert A[i] into bucket[floor(A[i]*n)]

for i:=0 to n-1
| sort list bucket[i] by insertion sort

concatenate bucket[0], bucket[1], ..., bucket[n-1]
```

The algorithm is demonstrated in Figure 8.4 in CLRS.

Theorem 4. *Expected running time for bucket sort is $O(n)$.*

Proof. Let n_i be a random variable representing the number of elements in the i-th bucket. Let $X_{i,j}$ be an indicator random variable of a condition that element j belongs to the i-th bucket. Clearly:

$$n_i = \sum_{j=1}^{n} X_{i,j}$$

Running time of the algorithm can be represented as a random variable X and can be measured as:

$$X = \sum_i n_i^2$$

To compute the expected running time, we need to compute:

$$
\begin{aligned}
E[X] &= E\left[\sum_i n_i^2\right] = \sum_i E[n_i^2] \\
&= \sum_i E\left[\left[\sum_j X_{i,j}\right]^2\right] \\
&= \sum_i E\left[\sum_{j,k} X_{i,j}X_{i,k}\right] \\
&= \sum_i \left(\sum_j E[X_{i,j}^2] + \sum_{j\neq k} E[X_{i,j}X_{i,k}]\right)
\end{aligned}
$$

Note, that $X_{i,j}X_{i,k} = 1$ if and only if both elements j and k belong to the bucket i. These are two independent random events. Therefore the expected value is:

$$E[X_{i,j}X_{i,k}] = \Pr(X_{i,j}X_{i,k} = 1) = \frac{1}{n}\cdot\frac{1}{n} = \frac{1}{n^2}$$

Also, $X_{i,j}^2 = 1$ if and onl if $X_{i,j} = 1$, and therefore:

$$E[X_{i,j}^2] = \Pr(X_{i,j}^2 = 1) = \frac{1}{n}$$

Therefore the expected running time of the bucket sort algorithm is:

$$E[X] = \sum_i \left(\sum_j \frac{1}{n} + \sum_{j\neq k} \frac{1}{n^2}\right) = \frac{n^2}{n} + \frac{n\cdot n\cdot(n-1)}{n^2} = n + n - 1 = O(n)$$

$\square$

3.4 Sorting Summary and Applications

Sort	Running time	Analysis
Merge Sort	$\Theta(n \log n)$	worst-case
Heap Sort	$\Theta(n \log n)$	worst-case
Randomized Quick Sort	$\Theta(n^2)$	worst-case
	$\Theta(n \log n)$	expected
Counting Sort	$\Theta(n+k)$	worst-case
numbers from $[0,k]$		
Radix Sort	$\Theta(dn)$	worst-case
d digit numbers		
Bucket Sort	$\Theta(n)$	expected
random real numbers from $[0,1)$		

Applications:

- **Finding convex hull:** Find convex hull of a given set of points. A *convex hull* is a subset of points defining a polygon that include all the other points inside it or on its boundary.

  Solution: Algorithm called Graham's scan sorts all the points counter-clockwise by their polar angle with respect to the point with the lowest y coordinate. This establishes a polygon which can be then transformed to the convex hull by a simple linear-time scanning procedure. Detailed description of the algorithm can be found in CLRS in Section 33.3.

- **Anagram problem:** We are given a dictionary of all words in English. Which word in English has the largest number of English anagrams? The two words are *anagrams* of each other, if one can be obtain from the other by rearranging its letters. For example, words *photomicrographic* and *microphotographic* are anagrams.

  Solution: We first sort all letters in each word. This creates a signature of each word, which is the same for all anagrams of the same word. After that, we find the number of duplicates of each signature by sorting all words by their signatures. The signature with the largest number of associated words corresponds to the largest group of anagrams in English. The whole solution can be written in less than 20 lines of code. More detailed description can be found in the book "Programming Pearls" by Jon Bentley.