COMP 3711 Fall 2021

Design & Analysis of Algorithms

Written By: Ljm

Topic 3 **Randomized Algorithms**

1 Recap: Probability

Expectation: $\mathbb{E}(X) = \sum i \cdot \Pr(X = i)$

Linearity of expectation: $\mathbb{E}(X + Y) = \mathbb{E}(X) + \mathbb{E}(Y)$, no matter X and Y are independent or not.

Indicator random variables: X only takes 0 or 1, which means $\mathbb{E}(X) = \Pr(X = 1)$.

Example 1: coin comes up heads with probability p and tails with $1 - p$. Find the expectation of flips X until first head is seen.

$$\begin{aligned}
\mathbb{E}(X) &= \sum_{j=1}^{\infty} j \cdot \Pr(X = j) \\
&= \sum_{j=1}^{\infty} j \cdot (1-p)^{j-1} \cdot p \\
&= \frac{p}{1-p} \sum_{j=1}^{\infty} j \cdot (1-p)^{j} \\
&= \frac{p}{1-p} \cdot \frac{1-p}{p} = \frac{1}{p}
\end{aligned}$$

The last step is somehow mysterious, the brief idea is:

$$\left(\sum_{n=1}^{\infty} x^i\right)' = \sum_{n=1}^{\infty} i \cdot x^{i-1}$$

multiply each side by x, and notice the left hand side is the derivative of geometric series, then:

$$x \cdot \left(\frac{x}{1-x}\right)' = \sum_{n=1}^{\infty} i \cdot x^i$$

This gives the last step above.

Example 2: Roll two dice. What is the expected total value X?

It is trivial that $\mathbb{E}(X_1) = \mathbb{E}(X_2) = 3.5$, then:

$$\mathbb{E}(X_1 + X_2) = \mathbb{E}(X_1) + \mathbb{E}(X_2) = 7$$

2 The Hiring Problem

Consider we're looking for an assistant and there are n candidates. We would like to hire the best one, so we interview one by one, and if the current one is better than the best one we've seen before, we just fire the previous one and hire the current one.

Algorithm 1: Hire-Assistant(n)

```
best ← 0
for $i$ ← 1 to $n$ do
    interview candidate $i$
    if candidate $i$ is better than best then
        fire best
        hire candidate $i$
        best ← $i$
    end
end
```

This algorithm runs fine, but there is one problem that we may hire too many people(worst case n) before we finally find the best one, so it may cost lots of money to fire old ones. In order to make things better, we consider interview the candidates *in a random order*.

Algorithm 2: Hire-Assistant(n)

```
randomly permute all $n$ candidates
best ← 0
for $i$ ← 1 to $n$ do
    interview candidate $i$
    if candidate $i$ is better than best then
        fire best
        hire candidate $i$
        best ← $i$
    end
end
```

So what is the expected number of hires in the algorithm above? Let an indicator variable X_i where:

$$X_i = \begin{cases} 1 & \text{, if we hire candidate } i \\ 0 & \text{, if we don't} \end{cases}$$

Then apparently the number of hires $X = X_1 + \cdots + X_n$, thus the expected number of hires, $\mathbb{E}(X) = \mathbb{E}(X_1) + \cdots + \mathbb{E}(X_n)$ according to the linearity of expectation.

Then what is $\mathbb{E}(X_i)$? Since X_i is an indicator variable, $\mathbb{E}(X_i) = \Pr(X_i = 1)$. We hire the candidate i if and only if he/she is the best among all first i candidates, and since they are arranged randomly, the probability that the best one among first i candidates is at the last position is $\frac{1}{i}$. Thus, $\mathbb{E}(X_i) = \frac{1}{i}$, and

$$\mathbb{E}(X) = \mathbb{E}(X_1) + \cdots + \mathbb{E}(X_n) = 1 + \frac{1}{2} + \cdots + \frac{1}{n-1} + \frac{1}{n} = \Theta(\log n)$$

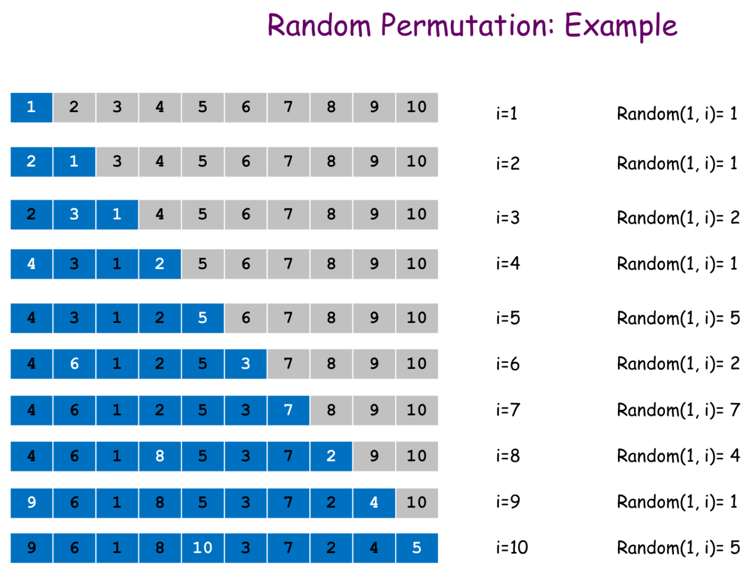
3 Generating a random permutation

Notice that in the hiring problem above, we first need to randomly permute those candidates. But how can we do that? (Here randomly means all permutations, in total $n!$, should appear with equal probability $\frac{1}{n!}$)

Let's first look at the implementation of this algorithm, and then explain why it can perform the job well. Assume out computer has a procedure $Random(i, j)$ that can generates a **random uniform** integer between i and j. (**uniform** means each int occurs with same probability)

Algorithm 3: RandomPermute(A)

```
1 $n \leftarrow A.length$
2 for $i \leftarrow 1$ to $n$ do
3     swap $A[i]$ with $A[Random(1, i)]$
4 end
```


Random Permutation: Example
1 2 3 4 5 6 7 8 9 10 i=1 Random(1, i)= 1
2 1 3 4 5 6 7 8 9 10 i=2 Random(1, i)= 1
2 3 1 4 5 6 7 8 9 10 i=3 Random(1, i)= 2
4 3 1 2 5 6 7 8 9 10 i=4 Random(1, i)= 1
4 3 1 2 5 6 7 8 9 10 i=5 Random(1, i)= 5
4 6 1 2 5 3 7 8 9 10 i=6 Random(1, i)= 2
4 6 1 2 5 3 7 8 9 10 i=7 Random(1, i)= 7
4 6 1 8 5 3 7 2 9 10 i=8 Random(1, i)= 4
9 6 1 8 5 3 7 2 4 10 i=9 Random(1, i)= 1
9 6 1 8 10 3 7 2 4 5 i=10 Random(1, i)= 5

In the example above, take the last step as an example, the random number we generated is 5, so we will swap the item at index 5, which is 5, with the last item 10. Then 10 ends up with index 5 and 5 ends up with index 10.

Notice this process is actually *revertible*, from the end to beginning. For example, we notice at the end, $A[5] = 10$, so the last step we must have swapped $A[5]$ with $A[10]$, since before last step, $A[10] = 10$. Then if we "revert" this step, i.e., swap $A[5]$ and $A[10]$, then the array will back to the status before last step.

Then, if we denote the random number generated at step i to be r_i, the n-tuple $(r_1, \cdots, r_n)$ will generate a permutation of the array. Moreover, given the result permutation, we can actually "revert" the process, like we discussed above, to get the $(r_1, \cdots, r_n)$ generated. This means, a tuple $(r_1, \cdots, r_n)$ is **uniquely corresponding** to a permutation at last. So if two tuples are different, the permutations they generate are different as well!

Now we can easily prove the correctness of this algorithm, since each tuple is generated with probability $\frac{1}{n!}$, (each r_i is generated with probability $\frac{1}{n}$), then each permutation is also generated with probability $\frac{1}{n!}$.

Here we will give another proof by induction, to show that "after the i-th iteration, $A[1 \cdots i]$ has been randomly permuted."

Base case: $i = 1$, trivial.

Inductive step: Assume $A[1 \cdots i-1]$ has been randomly permuted after $i-1$ iterations of the algorithm, then we will calculated the probability that $A[1 \cdots i] = (a_1, \cdots, a_i)$ appears after the i-th iteration.

For example, if $i = 10$, then after $(i-1)$ steps, the array must be something like this:

$$[2, 8, 7, 3, 4, 1, 5, 6, 9, 10]$$

then what is the probability that we generate $[2, 8, 10, 3, 4, 1, 5, 6, 9, 7]$ after i-th step? We must swap $A[3] = 7$ with $A[10] = 10$, which means $\mathit{Random}(1, i)$ must return 3 to achieve that.

To summarize the above paragraph, we can get $A[1 \cdots i] = (a_1, \cdots, a_i)$ after i-th iteration *if and only if*

- $\mathit{Random}(1, i)$ returns a specific number and
- $A[1 \cdots i-1]$ is a specific $(a_1, \cdots, a_{i-1})$

The second point above has a probability $\frac{1}{(i-1)!}$ according to assumption of induction, the first point above has a probability $\frac{1}{i}$. Hence, the probability that $A[1 \cdots i] = (a_1, \cdots, a_i)$ is $\frac{1}{(n-1)!} \cdot \frac{1}{i} = \frac{1}{i!}$, which means that it's a random permutation.

4 Quick Sort

Quick Sort is somehow similar to **Merge Sort**. Instead, it chooses a **pivot** each time, and then **partitions** the array so that all items less than or equal to the pivot are on the left and all items greater than pivot are on the right.

Then it recursively calls QuickSort to left and right sides.

Algorithm 4: QuickSort(A, l, r)

```
if $l \geq r$ then
    return
end
$pivot \leftarrow$ Partition($A$, $l$, $r$)
QuickSort($A$, $l$, $pivot - 1$)    // quick sort left part
QuickSort($A$, $pivot + 1$, $r$)    // quick sort right part
```

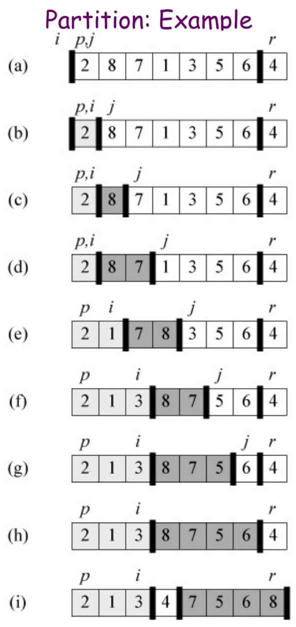
For the **Partition** process, we need to choose a pivot first, here we simply choose the last element as pivot. Then we divide the array $A[l \cdots r]$ into two parts: $A[l \cdots i]$ are all smaller or equal than pivot $A[r]$, $A[i+1 \cdots j-1]$ are all larger than pivot $A[r]$, while $A[j \cdots r-1]$ are not decided yet.

To divide, we perform "swaps": iterate j from l to $r-1$, if $A[j]$ is smaller or equal than pivot $A[r]$, we expand the area of left part, i.e., $i \leftarrow i+1$, and then swap $A[j]$ with $A[i]$. This will enlarge smaller subpart by size 1, and put the newly found item into that part. On the contrary, if $A[j]$ is larger than pivot, then we just expand the larger subpart by size 1, say $j \leftarrow j+1$, and no need to do other stuff since $A[j]$ will be already contained after we expand the part.

Here is the implementation of this process:

Algorithm 5: Partition(A, l, r)

```
$x \leftarrow A[r]$    // choose last item to be pivot
$i \leftarrow l - 1$    // No item found yet, so there should be nothing in $A[l \cdots i]$
for $j \leftarrow l$ to $r - 1$ do
    if $A[j] \leq x$ then
        $i \leftarrow i + 1$    // expand left part
        swap $A[i]$ and $A[j]$    // include the new item into left part
    end
end
swap $A[i+1]$ and $A[r]$    // make pivot to be at middle
return $i + 1$    // return pivot
```



Now we would like to analyze the running time of Quick Sort. Firstly, for the best case, where we *can always select median element as pivot*, we can divide the array into two parts every time, which gives $\Theta(n \log n)$. However, for the worst case, for example, if we always select the smallest(or largest) element as pivot, then it will be $\Theta(n^2)$.

Thus, in reality, we **randomly** choose a pivot in the array, and this is **randomized algorithm**: making a random choice each time it chooses a pivot.

For quick sort, or a randomized algorithm, we often care about its **expected running time**, denote as $\mathbb{E}[T(I,R)]$, where I is the input(the array in quick sort), while R is a random string or numbers that decide what we choose as random each time.

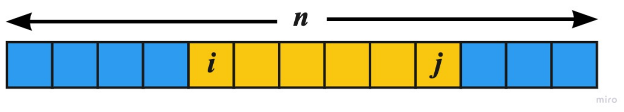
Without loss of generality, we can assume all elements are distinct, since if this is not the case, we can regard original input $A[1\cdots n]$ as $B[1\cdots n]$, where $B[i] = \{A[i], i\}$. Then all elements in B are distinct and we can just manipulate on $B[]$ instead. Also, remember the running time is *proportional to* number of comparisons. Notice that for any two items, they will either not be compared, or be compared only once, that is when one of them is the pivot. Since any two items can *at most compare once*, we can define indicator random variable:

$$X_{ij} = \begin{cases} 1 & \text{, if } i\text{–th smallest item is ever compared with } j\text{–th smallest item} \\ 0 & , otherwise \end{cases}$$

Then,

$$\begin{aligned} \mathbb{E}(runtime) &\leq \mathbb{E}\left(c \cdot \left(\sum_{i<j} X_{ij}\right)\right) \\ &= c \cdot \sum_{i<j} \mathbb{E}(X_{ij}) \\ &= c \cdot \sum_{i<j} \Pr(X_{ij} = 1) \end{aligned}$$

So now we consider how to find $\Pr(X_{ij} = 1)$, that is, the probability of i-th smallest and j-th smallest elements are ever compared. (In the image below, assume items are arranged in ascending order)



In the image above, we divide the array into two color regions, where between i and j(including i and j) are colored with yellow.

- if pivot is in **blue region**, then i and j will be then allocated to the same subpart of array, during this process, they cannot be compared to each other.

- if pivot is in **yellow region**, then this level is the last chance for i and j to be compared. And moreover, i and j will be compared *if and only if* i or j is chosen to be the pivot this level.

Thus, either they will be compared or not is *decided at the level which pivot is chosen in yellow region.* And at that level, they will be compared if and only if one of them is chosen as the pivot. Therefore,

$$\Pr(X_{ij} = 1) = \frac{2}{j - i + 1}$$

Now we can calculate:

$$\mathbb{E}(runtime) = c \cdot \sum_{i<j} \mathbb{E}(X_{ij}) = c \cdot \sum_{i=1}^{n-1} \sum_{j=2}^{n} \frac{2}{j - i + 1}$$

The summation is hard to calculate directly, but let's make a table:

When $i = 1$, $\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n-2} + \frac{1}{n-1} + \frac{1}{n}$

When $i = 2$, $\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n-2} + \frac{1}{n-1}$

When $i = 3$, $\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n-2}$

...

When $i = n - 1$, $\frac{1}{2}$

To sum up *by column*, we get:

$$(n-1) \cdot \frac{1}{2} + (n-2) \cdot \frac{1}{3} + (n-3) \cdot \frac{1}{4} + \cdots + 1 \cdot \frac{1}{n}$$

And with some tricks:

$$\begin{aligned}
&(n-1) \cdot \frac{1}{2} + (n-2) \cdot \frac{1}{3} + (n-3) \cdot \frac{1}{4} + \cdots + 1 \cdot \frac{1}{n} \\
\leq\ & n \cdot \frac{1}{2} + n \cdot \frac{1}{3} + n \cdot \frac{1}{4} + \cdots + n \cdot \frac{1}{n} \\
=\ & n \cdot \left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots \frac{1}{n} \right) \\
\leq\ & n \cdot \int_1^n \frac{1}{x}\, dx = n \ln n
\end{aligned}$$

Despite the constant c, the expected running time for quick sort is still $\Theta(n \log n)$.

5 Randomized Selection

Problem: Given an array A of n distinct elements, found the i-th smallest element in A.

This is the end of lecture note. Last modified: Sep 20.