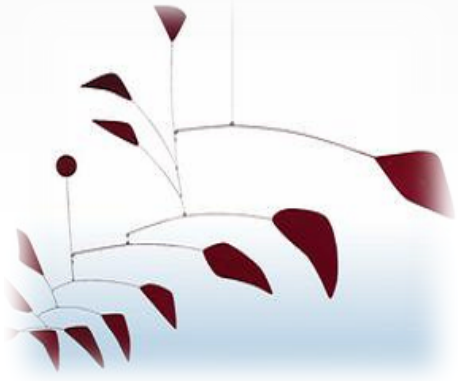


EECS 3101 - Design and Analysis of Algorithms

Shahin Kamali

Tutorial

York University

Picture is from the cover of the textbook CLRS.

Bucket Sort

Consider the following input for Bucket-Sort which is NOT uniformly sorted:

- one-third of the items are uniformly distributed in $[0, 0.3]$.
- one-third of items are uniformly distributed in $[0.3, 0.4]$.
- the remaining one-third are uniformly distributed in $(0.4, 1]$.

Bucket Sort

Consider the following input for Bucket-Sort which is NOT uniformly sorted:

- one-third of the items are uniformly distributed in $[0, 0.3]$.
- one-third of items are uniformly distributed in $[0.3, 0.4]$.
- the remaining one-third are uniformly distributed in $(0.4, 1]$.

Repeat the analysis from the class to describe the expected running time of Bucket-Sort with parameter $k = \sqrt{n}$ for the above input.

- You may assume that we a comparison-based sorting algorithm with running time $\Theta(n \log n)$ to sort items within each bucket.

Bucket Sort

- As before, distributing items between buckets takes $\Theta(n)$.

Bucket Sort

- As before, distributing items between buckets takes $\Theta(n)$.
  - The first group of $0.3k = 0.3\sqrt{n}$ buckets receive one-third of the numbers, that is, $n/3$ items. The expected number of items in each bucket is thus $(n/3)/(0.3\sqrt{n}) = \sqrt{n}/0.9 = \Theta(\sqrt{n})$. Therefore, sorting each of these buckets takes $\Theta(\sqrt{n}\log\sqrt{n}) = \Theta(\sqrt{n}\log n)$.

Bucket Sort

- As before, distributing items between buckets takes $\Theta(n)$.
  - The first group of $0.3k = 0.3\sqrt{n}$ buckets receive one-third of the numbers, that is, $n/3$ items. The expected number of items in each bucket is thus $(n/3)/(0.3\sqrt{n}) = \sqrt{n}/0.9 = \Theta(\sqrt{n})$. Therefore, sorting each of these buckets takes $\Theta(\sqrt{n}\log\sqrt{n}) = \Theta(\sqrt{n}\log n)$.
  - The second group of $0.1k = 0.1\sqrt{n}$ buckets receive one-third of the numbers. The expected number of items in each bucket is thus $(n/3)/(0.1\sqrt{n}) = \sqrt{n}/0.3 = \Theta(\sqrt{n})$. As before, sorting each of these buckets takes $\Theta(\sqrt{n}\log n)$.

Bucket Sort

- As before, distributing items between buckets takes $\Theta(n)$.
  - The first group of $0.3k = 0.3\sqrt{n}$ buckets receive one-third of the numbers, that is, $n/3$ items. The expected number of items in each bucket is thus $(n/3)/(0.3\sqrt{n}) = \sqrt{n}/0.9 = \Theta(\sqrt{n})$. Therefore, sorting each of these buckets takes $\Theta(\sqrt{n}\log\sqrt{n}) = \Theta(\sqrt{n}\log n)$.
  - The second group of $0.1k = 0.1\sqrt{n}$ buckets receive one-third of the numbers. The expected number of items in each bucket is thus $(n/3)/(0.1\sqrt{n}) = \sqrt{n}/0.3 = \Theta(\sqrt{n})$. As before, sorting each of these buckets takes $\Theta(\sqrt{n}\log n)$.
  - The third group of $0.6k = 0.6\sqrt{n}$ buckets receive one-third of the numbers, that is, $n/3$ items. The expected number of items in each bucket is thus $(n/3)/(0.6\sqrt{n}) = \sqrt{n}/1.8 = \Theta(\sqrt{n})$. As before, sorting each of these buckets takes $\Theta(\sqrt{n}\log n)$.

Bucket Sort

- As before, distributing items between buckets takes $\Theta(n)$.
  - The first group of $0.3k = 0.3\sqrt{n}$ buckets receive one-third of the numbers, that is, $n/3$ items. The expected number of items in each bucket is thus $(n/3)/(0.3\sqrt{n}) = \sqrt{n}/0.9 = \Theta(\sqrt{n})$. Therefore, sorting each of these buckets takes $\Theta(\sqrt{n}\log\sqrt{n}) = \Theta(\sqrt{n}\log n)$.
  - The second group of $0.1k = 0.1\sqrt{n}$ buckets receive one-third of the numbers. The expected number of items in each bucket is thus $(n/3)/(0.1\sqrt{n}) = \sqrt{n}/0.3 = \Theta(\sqrt{n})$. As before, sorting each of these buckets takes $\Theta(\sqrt{n}\log n)$.
  - The third group of $0.6k = 0.6\sqrt{n}$ buckets receive one-third of the numbers, that is, $n/3$ items. The expected number of items in each bucket is thus $(n/3)/(0.6\sqrt{n}) = \sqrt{n}/1.8 = \Theta(\sqrt{n})$. As before, sorting each of these buckets takes $\Theta(\sqrt{n}\log n)$.

Therefore, sorting all buckets take $\sqrt{n} \times \Theta(\sqrt{n}\log n) = \Theta(n\log n)$.
The overall complexity of the algorithm is thus $\Theta(n\log n)$.

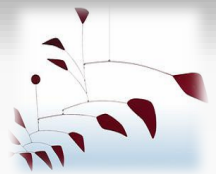
Egg Dropping Puzzle

- Suppose that we have n eggs, and wish to know which stories in a k-story building are safe to drop eggs from, and which will cause the eggs to break on landing. We make a few assumptions:
  - An egg that survives a fall can be used again.But a broken egg must be discarded.
  - The effect of a fall is the same for all eggs.
  - If an egg breaks when dropped, then it would break if dropped from a higher floor. Similarly, if an egg survives a fall then it would survive a shorter fall.
  - It is not ruled out that the first-floor windows break eggs, nor is it ruled out that the kth-floor do not cause an egg to break.

Egg Dropping Puzzle

- Suppose that we have n eggs, and wish to know which stories in a k-story building are safe to drop eggs from, and which will cause the eggs to break on landing. We make a few assumptions:
  - An egg that survives a fall can be used again.But a broken egg must be discarded.
  - The effect of a fall is the same for all eggs.
  - If an egg breaks when dropped, then it would break if dropped from a higher floor. Similarly, if an egg survives a fall then it would survive a shorter fall.
  - It is not ruled out that the first-floor windows break eggs, nor is it ruled out that the kth-floor do not cause an egg to break.

  Use a DP approach to find the floors from which eggs should be dropped so that the total number of trials are minimized. In other words, we need to find a **critical floor**, which is the lowest floor such that the egg breaks dropping from it.



Egg Dropping Puzzle

- **Step 1:**
  - Let $eggDrop(m, j)$ denote the minimum number of trials required to detect the critical floor among j floors ($1 \leq j \leq k$) using m eggs ($1 \leq m \leq n$); we want to find $eggDrop(n, k)$.
  - Setting $eggDrop(m, j)$: suppose we drop from floor x (where $1 \leq x \leq j$) in the first trial.

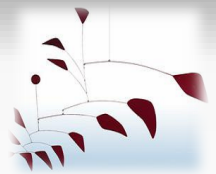
Egg Dropping Puzzle

- **Step 1:**
  - Let $eggDrop(m, j)$ denote the minimum number of trials required to detect the critical floor among j floors ($1 \leq j \leq k$) using m eggs ($1 \leq m \leq n$); we want to find $eggDrop(n, k)$.
  - Setting $eggDrop(m, j)$: suppose we drop from floor x (where $1 \leq x \leq j$) in the first trial.
    - If the egg breaks, the critical floor is in $[1, x]$. We need to search for the critical floor among $x - 1$ floors using $m - 1$ eggs.
    - if it survives, the critical floor is in $[x + 1, j]$. We need to search between $j - x$ floors using m eggs.

Egg Dropping Puzzle

- **Step 2:** Write a recursive formula for the minimum number of trials using m eggs and for j stories. We can write:

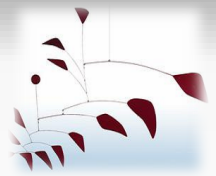
$$eggDrop(m, j) = \begin{cases} j & \text{if } j \leq 1 \\ j & \text{if } m = 1 \\ 1 + \min\limits_{x \in [1,j]} \max\{eggDrop(m-1, x-1), eggDrop(m, j-x)\} \end{cases}$$



Egg Dropping Puzzle

- **Step 3:** Fill the DP table bottom up. In this case, for setting $eggDrop(m, j)$ we look at the previous rows and columns. We can use the following simple code:

```
EggDropTable (n, k)
1.   for j = 1 to k
2.       for m = 1 to n
3.           if j ≤ 1 or m = 1
4.               eggDrop(m, j) ← j
5.           else
6.               eggDrop[m, j] ← ∞
7.               for x = 1 to j
8.                   candidate[x] ← 1 + max{eggDrop[m − 1, x − 1], eggDrop[m, j − x]}
9.                   if candidate[x] < eggDrop[m, j]
10.                      eggDrop[m, j] ← candidate[x]
11.  return eggDrop
```



Egg Dropping Puzzle

- **Step 4:** In Step 4, we usually move backward in the table to find the actual solution. Here, the question only asks for the number of trials. If the actual stories were intended, we had to store a flag in Line 11: $flag[m, j] \leftarrow x$

```
EggDropTable (n, k)
1.   for j = 1 to k
2.       for m = 1 to n
3.           if j ≤ 1 or m = 1
4.               eggDrop(m, j) ← j
5.           else
6.               eggDrop[m, j] ← ∞
7.               for x = 1 to j
8.                   candidate[x] ← 1 + max{eggDrop[m − 1, x − 1], eggDrop[m, j − x]}
9.                   if candidate[x] < eggDrop[m, j]
10.                      eggDrop[m, j] ← candidate[x]
11.                      flag[m, j] ← x
12.  return eggDrop
```

Maximum Product Cutting

- Given a rope of length n meters, cut the rope in different parts of integer lengths in a way that maximizes product of lengths of all parts. You must make at least one cut. Assume that the length of rope is more than 2 meters.

Maximum Product Cutting

- **Step 1:** Let $maxProd(m)$ be the maximum product for a rope of length m for $m \in [1, n]$; we want to find $maxProd(n)$.

Maximum Product Cutting

- **Step 1:** Let $maxProd(m)$ be the maximum product for a rope of length m for $m \in [1, n]$; we want to find $maxProd(n)$.

  Step 2: To set $maxProd[m]$, suppose the first cut has length i. If the remainder of $n - i$ is not cut any more, the profit is $i \times m - i$; otherwise, the profit is $i \times maxProd[m - i]$. So, for a cut of length i, the profit will be $\max\{i \times (m - i), i \times maxProd[m - i]\}$.
  Therefore $maxProd(m)$ can be written as following.

$$maxProd(m) = \begin{cases} 0 \quad \text{if } m \leq 1 \\ \max\limits_{i \in [1,m]} (\max\{i \times (m - i), i \times maxProd[m - i]\}) \end{cases}$$

Maximum Product Cutting

- **Step 3**: Fill the DP table:

```
RopeCutTable (n)
1.    for m = 1 to n
2.          if m ≤ 1
3.              maxProd(m) ← 0
4.          else
5.              maxProd[m] ← −∞
6.              for i = 1 to m
7.                  candidate[i] ← max{i × (m − i), i × eggDrop[m − i]}
8.                  if candidate[i] > maxProd[m]
9.                      maxProd[m] ← candidate[i]
10.                     flag[m] ← i
11.   return maxProd
```

Maximum Product Cutting

- **Step 4:** Retrieve the actual solution, i.e., the length of the cuts:

```
RetrieveCuts (n, maxProd)
1.    m ← n
2.    while m ≥ 2
3.          print flag[m]
4.          m ← m − flag[m]
```