DSC40B: Theoretical Foundations of Data Science II

Lecture 4: *Expected time complexity*

Instructor: Yusu Wang

Previously

- Worst-case and best-case time complexity analysis
  - Worst-case time complexity analysis
    - Most commonly used. Guarantees performance even in the worst case
- However, both worst- and best-case time can be caused by just some specific input
- How about average time complexity
  - Intuitively measures how the algorithm works on a typical input?

Expected Analysis

- Probabilistic method:
  - Given a distribution for all possible inputs
  - Derive expected time based on distribution

- Randomized algorithm:
  - Add randomness in the algorithm
  - Analyze the expected behavior of the algorithm

A simple example

```
def linear_search(A, k):
    for i, x in enumerate(A):
        if x == k:
            return i
    return None
```

- What is worst case time complexity?
- What is expected / average time complexity?

Expected Running Time

- Expected / average running time
  - $ET(n) = \sum_I \Pr(I)\, time(I)$
  - $\Pr(I)$ = probability of input type *I*
  - $time(I)$ = running time given input type *I*

- To analyze, **need to assume** a probabilistic distribution for all inputs

Linear-search algorithm

- Expected running time =

$$\Pr(K \notin A)\text{time}(K \notin A) + \sum_{i=1}^{n} \Pr(A[i] = K)\, time(A[i] = K)$$

- If we assume
  - $\Pr(K \notin A)$ = 0
  - All permutations are equally likely
    - implies $\Pr(A[i] = K) = \frac{1}{n}$
- $time(A[i] = K) = ci$
- Then expected running time = $\sum_i \left(\frac{1}{n}\right) * ci = \Theta(n)$

Remark

- For probabilistic analysis
  - An input probabilistic distribution input model **has to be assumed**!
  - For a fixed input, the running time is fixed.
  - The average / expected time complexity is for if we consider running it for a range of inputs, what the average behavior is.

- Randomized algorithm
  - No assumption in input distribution!
  - Randomness is added in the algorithm
    - For a fixed input, the running time is **NOT** fixed.
    - The expected time is what we can expect when we run the algorithm on **any single** input.

Analyzing randomized algorithms

Randomized linear search example

```
def rand_linear_search(A, k):
        random.shuffle(A)
        for i, x in enumerate(A):
            if x == k:
                return i
        return None
```

- What is expected / average time complexity?
  - Assuming we only search for keys already in A
  - $\Pr(A[i] = k) = \frac{1}{n}$
  - $\mathrm{ET(n)} = \mathrm{Pr(k \notin A)time(K \notin A)} + \sum_{i=1}^{n} \Pr(A[i] = k)\, time(A[i] = k)$
  - $= \sum_i \left(\frac{1}{n}\right) * ci = \Theta(n)$

Review of Expectation

- X is a random variable
- The expectation of X is
  - $E(X) = \sum_I \Pr(X = I)\ I$
  - E.g, coin flip

- Linearity of expectation:
  - $E(X_1 + X_2) = E(X_1) + E(X_2)$
- Conditional expectation:
  - $E(X) = E(X\,|Y)\Pr(Y) + E(X\,|\,Not\ Y)(1\ - \Pr(Y))$

Use of linearity of expectation

```
Input    : Array A of n integers.
function func1(A[ ],n)
s ← 0;
for i ← 1 to n do
    A[i] ← A[n − i + 1];
    s ← s + func2(A, n);
end
return (s);
```

$$ET_1(n) = n\,ET_2(n) + \mathrm{cn}$$

- $ET_2(n)$ = expected running time for func2
- What is $ET_1(n)$?

Use of Conditional expectation

```
function func1(A[ ],n)
Flip a coin;
if heads then
    a ← func2(A, n);
else
    a ← func3(A, n);
end
return (a);
```

$$ET_1(n) = \Pr(head)\, ET_2(n) \\ +(1 - \Pr(head))\, ET_3(n) + c$$

- $ET_2(n)$ = expected running time of func2
- $ET_3(n)$ = expected running time of func3
- What is the expected running time of func1?

Randomized example 1

```
Func1(A, n)
/* A is an array of integers
s ← 0;
k ← Random(n);
for i ← 1 to k do
    for j ← 1 to k do
        s ← s + A[i] * A[j];
    end
end
return (s);
```

- $Random(n)$:
  - returns a number k s.t. the probability that $k = i$ for any $i \in [1, n]$ is $\Pr[k = i] = \frac{1}{n}$

Running time analysis

- **Worst Case:**
  - $T(n) = \Theta(n^2)$

- **Expected running time:**
  - Step 1: identify different possible cases
  - Step 2: find the probability of each case
  - Step 3: find the running time of each case

  - $ET(n) = \sum_{i=1}^{n} \Pr(k = i) \cdot (ci^2) = \sum_{i=1}^{n} \frac{1}{n} \cdot (ci^2)$
    $= \frac{c}{n} \sum_{i=1}^{n} i^2 = \Theta(n^2)$

Randomized example 2

```
Func1(A, n)
/* A is an array of integers
s ← 0;
k ← Random(n);
if k ≤ log n then
    for i ← 1 to n do
        s ← s + A[i] * A[n];
    end
end
return (s);
```

Running time analysis

- Worst case running time
  - $T(n) = \Theta(n)$
- Best case?
- Expected analysis
  - Step 1:
    - Identify there are two cases: $k \leq \log n$ and $k > \log n$
  - Step 2:
    - Find probability of the two cases:
      - $\Pr[k \leq \log n] = \frac{\log n}{n}$; $\Pr[k > \log n] = 1 - \frac{\log n}{n}$
  - Step 3:
    - Find time complexity for each case:
      - $time(k \leq \log n) = cn$; $time(k > \log n) = c'$

Expected analysis

- Expected running time:
  - $ET(n) = \Pr(k \leq \log n) \cdot time(k \leq \log n) + \Pr(k > \log n) \cdot time(k > \log n)$
    $$= \frac{\log n}{n} \cdot (cn) + \left(1 - \frac{\log n}{n}\right) \cdot (c')$$
  - $\Rightarrow ET(n) = \Theta(\log n)$

These are artificial examples. We will see later a randomized algorithm for the sorting problem.

Lower bound theory

Problems and algorithms

- There can be many algorithms for solving the same problem.
  - Some have better time complexity than others.
- An important question:
  - For a given problem, what is the best possible time complexity?
- Such questions can be hard to answer
  - as typically we cannot ``enumerate'' all possible algorithms
- Often we try to provide a lower-bound
  - that is as tight as we can
  - Sometimes we know we have the right (tight) bound when there is an algorithm whose worst-case running time matches this lower bounds

Lower Bound

- No algorithm can have a better (worst case) time complexity than a theoretical lower bound.

Definition:

$f(n)$ is a theoretical lower bound for a problem if every possible algorithm's worst-case time complexity is $\Omega(f(n))$.

A simple example

- The Search problem:
  - Input: given an arbitrary array A of numbers and a key k
  - Output: return whether $k \in A$ or not
- A trivial lower-bound
  - $\Omega(1)$
  - Not wrong, but useless
- A better lower-bound
  - $\Omega(n)$
  - as in the worst case, any algorithm will have to inspect every element in A

Can we get a better lower-bound?

Tight Lower bound

- A lower-bound $f(n)$ for problem-P is tight if there exists an algorithm for problem-P whose worst-case running time is $\Theta(f(n))$.
  - In some sense, this algorithm has optimal running time.

- Back to the Search problem
  - There is an algorithm
    - $T(n) = \Theta(n)$
  - Hence the lower bound
    - $\Omega(n)$ is tight

```
def linear_search(A, k):
        for i, x in enumerate(A):
            if x == k:
                return i
        return None
```

FIN