Section Notes 0

1 Asymptotic Algorithm Analysis

When designing an algorithm, we should always take the time and space complexity into consideration. Time complexity represents how much time it takes to run your program, while space complexity stands for the memory usage for your code (such as using arrays, pointers, etc.) The former one matters when you need to run your program in required time, and the latter one determines the required computational resource for the running machine.

For this section, we will go deep into the calculation for time complexity, and you can compute the space complexity in the similar way.

1.1 Best, Worst, and Average cases

For most of the cases, running time depends on the size of inputs. This is why we have best, worst and average cases for running time.

According to the given input, the running time may differ a lot, which leads to distinct best, worst, and average running time.

Here are the two examples mentioned in the lecture.

```cpp
// REQUIRES: a (an array of size n)
// EFFECTS: return the index of the element that equals key. If no such element,
    return n.
int search(int a[], unsigned int n, int key) {
  for (unsigned int i = 0; i < n; i++) {
    if (a[i] == key) return i;
  }
  return n;
}
```

We can see that:

- **Best case**: If the first entry of array a equals to *key* ($a[0] = key$), the function will end after the first iteration, which only runs $\underline{1}$ iteration.
- **Worst case**: If the last entry of array a equals to *key* ($a[n-1] = key$) or if there is no such element that equals to *key* (only slight difference between these two cases, the latter one is worse), the function will run $\underline{n}$ iterations.
- **Average case**: Suppose *key* is uniformly located in the array. Then, on average, the function will run $\underline{n/2}$ iterations.

```cpp
// REQUIRES: a is an array of size n
// EFFECTS: return the sum
int sum(int a[], unsigned int n) {
  int result = 0;
  for(unsigned int i = 0; i < n; i++) {
    result += a[i];
  }
  return result;
}
```

For this case, input will not influence the number of iterations (always running n iterations) Hence the best/worst/average cases all run $\underline{n}$ iterations.

We often focus more on average case and worst case(especially). In terms of average case, we need to know the distribution of input (such as uniformly distributed). Worst case time complexity can give an upper bound to each algorithm, which is quite popular when we compare different algorithms for one specific problem. We will see its usage in the next lecture (sorting).

Remark 1.1. Before we move on to the next part, something needs to be clarified ahead:

- **Do not mix up the concept of "iteration", "step", and "operation"**: In the previous examples, we use "running X iterations" to represent the time complexity (in lecture, we use "comparing X times" which has similar meaning"). However, such description is not rigorous enough, and just for convenience and better understanding for current stage. The number of "operation" better measures the complexity of each algorithm which will be talked about later. In terms of "step", this is a macro concept, and we will try to avoid using it in any questions.
- If it is not specified, when we talk about time complexity, you need to give the answer under the **worst case**.

1.2 Big-Oh,Big-Omega, Big-Theta

1.2.1 Big-Oh O

Definition 1.2. A non-negatively valued function $T(n)$ is in the set $O(f(n))$ if there exist two positive constants c and n_0 such that $T(n) \leq c \cdot f(n)$ for all $n > n_0$.

Hope these examples can give you a straightforward overview on Big-oh notation.

(i) $T(n) = n^2$ is in the set $O(n^2)$

(ii) $T(n) = 2n^2$ is in the set $O(n^2)$

(iii) $T(n) = 3n^2 + \frac{1}{4}n$ is in the set $O(n^2)$

(iv) $T(n) = 2n^2$ is in the set $O(n^3)$

Here you can see that Big-oh notation is similar to concept of an upper bound. We can simply find the (c, n_0) pair for this example (such as $(c, n_0) = (2, 1)$).

(v) $T(n) = \frac{\sqrt{n}}{3} + 1$ is in the set $O(\sqrt{n})$

(vi) $T(n) = n \log n$ is in the set $O(n \log n)$

The $\log n$ complexity often exists in Divide and Conquer algorithms.

(vii) $T(n) = \log_2 n$ is in the set $O(\log n)$

Try to prove this! (Hint: Properties of logarithm) Then you can see that for all complexity in the form of $\log_m n$, we can always write it into the form of $O(\log n)$, and that's why we do not specify the number of m and write it as $O(\log n)$.

(viii) $T(n) = 2^{n+2}$ is in the set $O(2^n)$

A sufficient Condition of Big-Oh:
If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$, then $f(n)$ is $O(g(n))$.
We can regard this as the transform of original definition. Feel free to use either of them for better understanding.

1.2.2 Big-Omega Ω

Definition 1.3. A non-negatively valued function $T(n)$ is in the set $\Omega(f(n))$ if there exist two positive constants c and n_0 such that $T(n) \geq c \cdot f(n)$ for all $n > n_0$.

The definition of Big-Omega notation is quite similar to the one of Big-Oh notation. The former one represents the ”lower bound” of a function, while the latter one represents the ”upper bound”.

1.3 Big-Theta Θ

Definition 1.4. A non-negatively valued function $T(n)$ is in the set $\Theta(f(n))$ if it is in both $O(g(n))$ and $\Omega(g(n))$.

When Big-Oh and Big-Omega coincide, we indicate this by using Big-Theta notation. It gives the tightest definition of a function's complexity. When dealing with complexity of a function, we often consider Big-Oh and Big-Theta notations.

Remark 1.5. Look carefully on the mark of notations! (Especially O and Θ)

1.4 Time complexity of a program

Here I would like to give more clarifications and inspects on how to correctly calculate the number of operations and the related complexity.

1.4.1 Number of operations in a for loop

```
for (int i = 0; i < n; i++) {
  roihn++;
}
```

For the code above, we cannot simply say that there are n operations in total. Instead, the following statements should also be counted as operations:

- $i < n$;
- $i++$;
- $roihn++$;

Actually, according to the mechanism of for loop in c++, we will run $i++$ for n times, and run $i < n$ for $n+1$ times (The last time is to jump out of the loop).

Therefore, we have: $3n + 1$ operations in total.

1.4.2 Clearer way of calculation

What is the time complexity of the following code?

```
roihn = 0;
for (int i = 0; i < n; i++) {
  for (int j = 0; j < i; j++) {
    roihn++;
  }
}
```

Solution: The number of iterations N can be calculated as:

$$\begin{aligned} N &= \sum_{i=0}^{n-1}\sum_{j=0}^{i-1} 1 \\ &= \sum_{i=0}^{n-1} i \\ &= \frac{(0+n-1)n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \end{aligned}$$

Hence the time complexity of the code is $O(n^2)$.

What is the time complexity of the following code?

```cpp
roihn = 0;
for (int i = 0; i < n; i += 2) {
  for (int j = 0; j < i; j += 3) {
    roihn++;
  }
}
```

Solution: The number of iterations N can be calculated as:

$$\begin{aligned} N &= \sum_{i=0,\ i\%2=0}^{n-1}\ \sum_{j=0,\ j\%3=0}^{i-1} 1 \qquad \text{(* Here \% means mod.)} \\ &= \sum_{i=0,\ i\%2=0}^{n-1} \frac{i}{3} \\ &= \frac{(0+(n-1)/3)(n/2)}{2} \\ &= \frac{1}{12}n^2 - \frac{1}{12}n \end{aligned}$$

Suppose n is an odd number then the time complexity of the code is $O(n^2)$.

Here we only do rough calculation because it is already enough for us to get the complexity in the big-oh or big-theta notation.

What is the time complexity of the following code?

```cpp
roihn = 0;
for (int i = 1; i <= n; i *= 2) {
  roihn++;
}
```

Solution: Here it is simpler to directly write out some terms.

- For the 1st iteration, $i = 1$.

- For the 2nd iteration, $i = 2$.
- For the 3rd iteration, $i = 4$.
- For the 4th iteration, $i = 8$.
- For the kth iteration, $i = 2^{k-1}$.

Suppose the loop ends when $i = 2^{k-1}$ at kth iteration. Since the stopping statement for this loop is $i <= n$, we have:

$$\begin{aligned} 2^{k-1} &> n \\ n &< \log_2(k-1) \end{aligned}$$

Hence the time complexity of the code is $O(\log n)$.

What is the time complexity of the following code?

```
int fibo(int k) {
  if (k == 0) return 0;
  if (k == 1 || k == 2) return 1;
  return fibo(k-1) + fibo(k - 2);
}
```

Solution: We can roughly say that: for each k, we need to call another two functions until it meets the stopping stage.

For $fibo(n)$, it will call 2 functions: $fibo(n-1)$ and $fibo(n-2)$;

For $fibo(n-1)$ and $fibo(n-2)$, they will call 4 functions: $fibo(n-2)$, $fibo(n-3)$, $fibo(n-3)$ and $fibo(n-4)$

...

Hence, the complexity of the code is $O(2^k)$.

You may notice that the recursive function is quite slow, and this is because we repeat calling functions with same k. You can see that we call $fibo(n-3)$ twice above. To smartly boost the efficiency, you will learn dynamic programming, which stores the values for each k when it is first called, at the end of the semester.