Lecture 5: Program Analysis
Big-O Notation

Rui Wang
Coming Up

• Assignment 2 is posted and due on Thursday, Feb 11.
  – It has both a programming part and written part.
Coming Up

• Change to Office hours this week:
  – My office hour is canceled this afternoon
  – The Wednesday office hours this week end at 4.
  – Junghee’s office hours have permanently moved from Monday to Friday 1-3pm (starting next week).

• Discussion section tomorrow:
  – Review of Array and ArrayList.
Today

• Computation cost of the binary search
• Big-O Notation
  – Constant
  – Linear
  – Logarithm
  – Quadratic
• Comparison of costs
Advantage of the Ordered Array

• The cost of binary search is much lower than linear search, especially for large arrays.
• For searching in an array of 100 numbers
  – Linear search takes a maximum of 100 steps, and on average 50 steps.
  – Binary search takes a maximum of 7 steps.
Analyzing the Computation Cost

• We will take a look at the number of steps it takes to run an algorithm.
  – The # of steps in terms of the problem size $n$
  – For simplicity, we only count the number of times a loop statement is executed.
    – Each statement takes 1 unit of time.

• This will be an estimate of the running time or computation cost
Example 1

double sum = 0.0;
for (int i = 0; i < n; i ++) {
    sum += array[i];
}

How many steps?
Only count the loop statements (update to the loop variable i is ignored).
Example 1

double sum = 0.0;
for (int i = 0; i < n; i++) {
    sum += array[i];
}

How many steps?
Loop will be executed n times; and there is 1 loop statement. So overall:

\( n \)
Example 2

double sum = 0.0;
for (int i = 0; i < n; i += 2) {
    sum += array[i];
}

How many steps?
Example 2

double sum = 0.0;
for (int i = 0; i < n; i += 2) {
    sum += array[i];
}

How many steps?
Loop will be executed n/2 times. So overall:

\[ \frac{n}{2} \]
Example 3 – Multiple Statements

```java
for (int i = 0; i < n; i ++) {
    for (int j = 0; j < n; j ++) {
        int x = i*j;
        sum += x;
    }
}
```

How many steps?
Example 3 – Multiple Statements

```java
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        int x = i*j;
        sum += x;
    }
}
```

How many steps?
Loop will be executed $n^2$ times; and there are 2 loop statements, so overall:

$$2 \cdot n^2$$
Increase of Cost w.r.t. $n$

• Although the first example takes twice as many steps, both of them are **linear** to the problem size $n$
  – If $n$ is 3 times larger, both costs are 3 times larger
Increase of Cost w.r.t. $n$

- Although the first example takes twice as many steps, both of them are **linear** to the problem size $n$  
  - If $n$ is 3 times larger, both costs are 3 times larger

- The third example is different:  
  - If $n$ is 3 times larger, it becomes 9 times more expensive.  
  - Therefore the cost is quadratic w.r.t. to problem size.
Increase of Cost w.r.t. $n$

- In practice, we care a lot about how the cost increases w.r.t. the problem size, rather than the absolute cost.
Increase of Cost w.r.t. $n$

• In practice, we care a lot about how the cost increases w.r.t. the problem size, rather than the absolute cost.

• Therefore we can ignore the constant scale factor in the cost function, and concentrate on the part relevant to $n$.

• If the cost grows linearly, we say the cost is $O(n)$ regardless of whether the absolute cost is $n$, $2n$, $n/2$ or $1000n$. 
Big-O Notation

• Formal Definition:
Assume the cost of an algorithm is $T(n)$, then

$$T(n) = O(f(n))$$

if there exists a constant $c$ such that

$$T(n) \leq c \cdot f(n)$$

is always true for sufficiently large $n$
Big-O Notation

• Formal Definition:
Assume the cost of an algorithm is \( T(n) \), then
\[
T(n) = O(f(n))
\]
if there exists a constant \( c \) such that
\[
T(n) \leq c \cdot f(n)
\]
is always true for sufficiently large \( n \)
• In other words, \( c \cdot f(n) \) is the upper bound of the computation cost.
Big-O Notation

• We can also say that the computation cost $T(n)$ grows in the order of $f(n)$

• Example:

  $$T(n) = 10 \, n^2$$

  using Big-O notation, we say the computation cost is: $O(n^2)$

  or equivalently: the computation cost is on the order of $n^2$
Big-O Notation

• What if $T(n)$ has multiple terms?
• For example:

$$T(n) = n^3 + 100 \cdot n^2 + 1000$$
Big-O Notation

• What if $T(n)$ has multiple terms?
• For example:
  $$T(n) = n^3 + 100 n^2 + 1000$$
• From calculus, we know that the growth rate, as $n$ becomes large, is dominated by the highest-order term. So
  $$T(n) = O(n^3)$$
  (even though the other two terms have much larger constants)
Big-O Notation

• **Summary**
  1. Write down the cost $T(n)$
  2. Look for the highest order term in $T(n)$
  3. Ignore constant scaling factors
Example 4

```java
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        sum += i*j;
    }
}
```
Example 4

```c
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        sum += i * j;
    }
}
```

\[
n + (n - 1) + (n - 2) + ... + 1 + 0 = \frac{n(n+1)}{2}
\]
Example 5

- What about this:

```java
double product = 1.0;
for (int i = 1; i <= n; i *= 2) {
    product *= i;
}
```
Example 5

• What about this:

```java
double product = 1.0;
for (int i = 1; i <= n; i *= 2) {
    product *= i;
}
```

• This has a logarithmic cost:

\[ O(\log_2 n) \]

or \( O(\log n) \) as the change of base is merely a matter of a constant factor.
Binary Search

• Now let’s take a look at the binary search.
Binary Search

• Now let’s take a look at the binary search.
• As we shrink the search range by 2 each time, the number of steps required for search is a logarithm of nElems. Thus:

\[ T(n) = O(\log n) \]
Big-O Notation

• Why is this a big deal?
  – Is the logarithmic cost much better than linear?
• It is, particularly for large-scale problems (i.e. the problem size \( n \) is large)
  – It is so even though binary search seems to involve more steps per iteration.
• Comparing quadratic vs. linear vs. log vs. constant cost.
Big-O Notation

• Try to do the following experiment:
  – Insert 1 million elements to an ordered array.
  – Use linear search to find an element.
  – Repeat it 100 times to get an estimate of the average search time.
  – Now use binary search instead.
  – Compare the average search time of the two.
Disadvantage of Ordered Array

• On the other hand, insertion is slower.

• However, overall an ordered array still wins in cases where searching is more frequent than insertion.
Big-O Notation

• From calculus we know that: in terms of the order:
  
  *exponentials* >
  *polynomials* >
  *logarithms* >
  *constant*. 
Big-O Notation

• From calculus we know that: in terms of the order:
  - exponentials > polynomials > logarithms > constant.

Increasing order

- $O(3^n)$
- $O(2^n)$
- $O(n^3)$
- $O(n^2)$
- $O(n \log n)$
- $O(n)$
- $O(\log n)$
- $O(1)$

Exponential
Polynomial
Log-linear
Linear
Log
Constant
Big-O Notation

- What about the following?

\[
\begin{align*}
O(n) & \quad O(\log n^2) \\
O(n) & \quad O((\log n)^2) \\
O(\log n) & \quad O(\log \log n) \\
O(n) & \quad O(2^{\log_2 n}) \\
O(n!) & \quad O(2^n)
\end{align*}
\]
Big-O Notation

• What about the following?

\[ O(n) > O(\log n^2) \]
\[ O(n) > O((\log n)^2) \]
\[ O(\log n) > O(\log \log n) \]
\[ O(n) = O(2^{\log_2 n}) \]
\[ O(n!) > O(2^n) \]
Example 6

• What about this:

```java
double product = 1.0;
for (int i = 1; i <= n; i *= 2) {
    for (int j = 1; j <= i; j ++) {
        product *= j;
    }
}
```
Example 6

• What about this:

```java
double product = 1.0;
for (int i = 1; i <= n; i *= 2) {
    for (int j = 1; j <= i; j ++) {
        product *= j;
    }
}
```

$1 + 2 + 4 + 8 + ... + 2^{\log_2 n} = 2^{(\log_2 n)+1} = O(n)$
Big-O Notation

• Remember: Big-O notation does **not** provide the **absolute** running cost; it provides an evaluation of the **growth rate of the cost**: i.e. how the cost increases w.r.t. to problem size **n**.