Exercise Series N°01 : Algorithm Analysis

Exercise I.1 (Big-Oh Notation)

Q1) Assume that each of the expressions below gives the processing time $\boldsymbol{T(n)}$ spent by an algorithm for solving a problem of size $\boldsymbol{n}$. Select the dominant term(s) with the steepest increase in $\boldsymbol{n}$ and specify the lowest Big-Oh complexity of each algorithm.

Expression	Dominant term (s)	O(. . .)
$5 + 0.001n^3 + 0.025n$	…	…
$500n + 100n^{1.5} + 50n\ log_{10}\ n$	…	…
$0.3n + 5n^{1.5} + 2.5n^{1.75}$	…	…
$n^2 log_2\ n + n\ (log_2\ n)^2$	…	…
$n\ log_3\ n + n\ log_2\ n$	…	…
$3\ log_8\ n + log_2\ log_2\ log_2\ n$	…	…
$100n + 0.01n^2$	…	…
$0.01n + 100n^2$	…	…
$2n + n^{0.5} + 0.5n^{1.25}$	…	…
$0.01n\ log_2\ n + n\ (log_2\ n)^2$	…	…
$100n\ log_3\ n + n^3 + 100n$	…	…
$0.03\ log_4\ n +\ log_2\ log_2\ n$	…	…

Q2) The statements below show some features of "Big-Oh" notation for the functions $\boldsymbol{f \equiv f(n)}$ and $\boldsymbol{g \equiv g(n)}$. Determine whether each statement is **TRUE** or **FALSE** and correct the formula in the latter case.

Statement	Is it TRUE or FALSE?	If it is FALSE then write the correct formula
Rule of sums: $\mathcal{O}(f + g) = \mathcal{O}(f) + \mathcal{O}(g)$	…	…
Rule of products: $\mathcal{O}(f \times g) = \mathcal{O}(f) \times \mathcal{O}(g)$	…	…
Transitivity: $If\ g = \mathcal{O}(f)\ and\ h = \mathcal{O}(f), then\ g = \mathcal{O}(h)$	…	…
$5n + 8n^2 + 100n^3 = \mathcal{O}(n^4)$	…	…
$5n + 8n^2 + 100n^3 = \mathcal{O}(n^2 \log n)$	…	…

Exercise I.2 (Expression & Big-Oh Notation)

Q1) Simplify each summation to an expression in terms of $\boldsymbol{n}$, and provide the big-O growth of the expression.

a) $\sum_{i=1}^{n}(n-2i+3)$

b) $\sum_{i=0}^{n-1}(4i^2-2i+7)$

c) $\sum_{j=10}^{n} j$

d) $\sum_{i=1}^{n}\sum_{j=1}^{n} j$

e) $\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i)$

Exercise I.3 (Runtime & Big-Oh Notation)

Q1) Prove that $\boldsymbol{T(n)=2n^3+9n^2}$ is $\boldsymbol{\mathcal{O}(n^3)}$ using the formal definition of the Big-Oh notation.
Hint: Find a constant $\boldsymbol{c}$ and threshold $\boldsymbol{n_0}$ such that $\boldsymbol{cn^3 \geq T(n)}$ for $\boldsymbol{n \geq n_0}$.

Q2) Prove that $\boldsymbol{T(n)=a_0+a_1n++a_2n^2+a_3n^3}$ is $\boldsymbol{\mathcal{O}(n^3)}$ using the formal definition of the Big-Oh notation.

Q3) Algorithms **A** and **B** spend exactly $\boldsymbol{T_A(n)=5\,n\,log_{10}n}$ and $\boldsymbol{T_B(n)=25\,n}$ microseconds, respectively, for a problem of size $\boldsymbol{n}$. Which algorithm is better in the Big-Oh sense? For which problem sizes does it outperform the other?

Q4) Algorithms **A** and **B** spend exactly $\boldsymbol{T_A(n)=c_A\,n\,log_{10}n}$ and $\boldsymbol{T_B(n)=c_B\,n^2}$ microseconds, respectively, for a problem of size $\boldsymbol{n}$. Find the best algorithm for processing $\boldsymbol{n=2^{20}}$ data items if the algorithm **A** spends **10** microseconds to process **1024** items and the algorithm **B** spends only **1** microsecond to process **1024** items.

Exercise I.4 (Code Runtime & Big-Oh Notation)

For each of the following code fragments, provide an appropriate summation expression that models its running time $\boldsymbol{T(n)}$. Then simplify the summation expression into an expression in terms of $\boldsymbol{n}$, and determine a Big-O notation of the execution time $\boldsymbol{T(n)}$.

Q1)

```
sum = 0;
for(int i=0; i < n; i++)
   sum++;
```

Q2)

```
sum = 0;
for(int i=0; i < n; i++)
   for(int j=0; j < n; j++)
      sum++;
```

Q3)

```
sum = 0;
for(i=0; i < n; i++)
   for(j=0; j < n*n; j++)
      sum++;
```

Q4)

```
sum = 0;
for(i=0; i < n; i++)
   for(j=0; j < i; j++)
      sum++;
```

Exercise I.5 (Number of Operations & Time Complexity)

Consider the following function, computing the multiplication of two positive integers $\boldsymbol{m}$ and $\boldsymbol{n}$:

```
        int Mult(int m, int n)
1:      r = 0 ;
2:      while (n > 0){
3:         if (n%2 == 1)
4:            r += m ;
5:         m *= 2 ;
6:         n /= 2 ;}
7:      return r;
```

Q1) Count the number of primitive operations performed on each line.

Q2) By increasing this number, give a time complexity relative to $\boldsymbol{n}$ in "Big-Oh"" notation.

Exercise I.6 (Complexity of Algorithms)

Consider the following code, with two nested "**while**". We want to measure the complexity of this nesting as a function of n. Hence, we use the variable "**counter**", which is incremented each time it passes through the internal "**while**".

```
int counter = 0;
int i = 1, j;
while (i < n){
   j = i + 1 ;
   while (j <= n){
      counter = counter + 1 ;
      j = j + 1 ;
   }
   i = i * 2 ;
}
```

Q1) What is the final counter value when $n = 16$?

Q2) Consider the special case where n is a power of 2: assume that $n = 2^p$ with p known. What is the final counter value as a piece of code of p ? Justify your answer.

Q3) Express the previous result as a function of n.

Q4) Conclude the worst-case complexity , in "Big-Oh"" notation, of this piece of code.

Exercise I.7 (Time Complexity Using Recurrence Relation)

Write a recurrence relation for the following recursive functions:

Q1)

```
void fun1(int n) {
     if (int n > 0){
         printf("%d", n);
         fun1(n-1);
      }
}
```

Q2)

```
int fibnacci (int n) {
   if (n = = 0)
      return 0;
   if (n = = 1)
      return 1;
   return fibnacci(n-1)+fibnacci(n-2);
}
```

Q3)

```
Fast_Power(float x, int n){
    if (n==0)
       return 1;
    elseif (n==1)
       return x;
    elseif ((n%2)==0)            //if n is even
       return Fast_Power(x, n/2)*Fast_Power(x,n/2);
    else
       return x* Fast_Power(x, n/2)*Fast_Power(x,n/2);
}
```

Exercise I.8 (Counting Function Calls)

For each of the following code snippets, compute the number of calls to f as a function of n. Provide **both** the exact number of calls and a maximally simplified, tight asymptotic bound in big-O notation

Q1)

```
f();
int i = 0;
while (i <= n){
     f();
     i= i+1;
}
```

Q2)
```
int i = 0;
while (i*i <= n){
    f();
    f();
    for (int j=1, j<=n, j++)
        f();
    i= i+1;
}
```

Q3)
```
void g(int n){
  int i = 0;
  while (i < n){
    f();
    i= i+1;
  }
  g(n/2);
  g(n/2);
  g(n/2);
}
```

Exercise I.9 (Recurrence Relations: Iterative Substitution Method)

Solve the following recurrence relations using the iterative substitution method:

Q1) $T(n) = T(n-1) + 2, \quad T(1) = 1$

Q2) $T(n) = 2T(n/2) + n, \quad T(1) = 1$

Q3) $T(n) = 2T(n/2) + 1, \quad T(1) = 1$

Q4) $T(n) = 2T(n-1) + n, \quad T(1) = 1$

Q5) $T(n) = 2T(\sqrt{n}) + 1, \quad T(1) = 1$

Q6) $T(n) = 2T(\sqrt{n}) + n, \quad T(1) = 1$

Q7) $T(n) = 2T(\sqrt{n}) + log\ n, \quad T(1) = 1$

Exercise I.10 (Recurrence Relations: Master Theorem)

Solve the following recurrence relations using the master theorem :

Q1) $T(n) = 9T(n/3) + n, \quad T(1) = \mathcal{O}(1)$

Q2) $T(n) = T(2n/3) + 1, \quad T(1) = \mathcal{O}(1)$

Q3) $T(n) = 3T(n/4) + n\ log\ n, \quad T(1) = \mathcal{O}(1)$

Q4) $T(n) = 2T(n/2) + n, \quad T(1) = \mathcal{O}(1)$

Q5) $T(n) = 4T(n/2) + \ n, \quad T(1) = \mathcal{O}(1)$

Q6) $T(n) = 4T(n/4) + n^2, \quad T(1) = \mathcal{O}(1)$

Q7) $T(n) = \frac{1}{2}T(n/2) + 1, \quad T(1) = \mathcal{O}(1)$

Q8) $T(n) = 2^n T(n/2) + 1, \quad T(1) = \mathcal{O}(1)$

Q9) $T(n) = 2T(n/4) - n^2 log\ n, \quad T(1) = \mathcal{O}(1)$

Q10) $T(n) = 2T(n/2) - n/log\ n\ , \quad T(1) = \mathcal{O}(1)$

Exercise I.11 (Recurrence Relations: Recurrence Tree Method)

Solve the following recurrence relations using the master theorem recursion tree method :

Q1) $T(n) = 2T(n/2) + 1, \quad T(1) = \mathcal{O}(1)$

Q2) $T(n) = 2T(n/2) + n^2, \quad T(1) = \mathcal{O}(1)$

Q3) $T(n) = T(n/3) + T(2n/3) + n, \quad T(1) = \mathcal{O}(1)$

Exercise I.12 (Time Complexity of Iterative Algorithm)

Work out the computational complexity of the following pieces of code:

Q1)

```
int sum = 0;
for (int i= 1; i <= n + 2; i++){
    sum++; }
for (int j= 1; j <= n * 2; j++){
    sum += 5;
}
cout << sum << endl;
```

Q2)

```
int sum = 0;
for(int i = 1; i <= N-5; i++){
   for(int j= 1; j<= N-5; j+=2){
   sum++;
}
}
cout << sum << endl;
```

Q3)

```
int sum = n;
for (int i= 0; i< 1000000; i++){
    for(int j = 1; j <= i; j++)
{
         sum += n;
    }
    for(int j = 1; j <= i; j++){
        sum += n;
    }
    for(int j = 1; j <= i; j++){
        sum += n;
    }
}
cout << sum << endl;
```

Q4)

```
int sum = 0;
for (int i= 1; i <= n - 2; i++){
    for(int j= 1; j<= i+4; j++){
       sum++;
    }
sum++;
}
for (int i = 1; i <= 100; i++) {
    sum++;
}
cout << sum << endl;
```

Q5)

```
int sum = 0;
for (int i = 1; i <= n; i++){
  for (int j= 1; j <= n*n; j++){
       sum++;
  }
  for (int j= 1; j <= 100; j++){
       sum++;
  }
  for (int j = 1; j <= n; j++){
       sum++;
  }
  sum++;
}
cout << sum << endl;
```

Q6)

```
int sum = 0;
for (int i = 1; i <= n; i++) {
    for(int j= 1; j<= 100; j++){
       sum++;
    }
}
for (int k = 1; k <= 10000;
k++){
    sum++;
}
cout << sum << endl;
```

Q7)

```
int sum = 0;
for (int i = 0; i < n * 2; i++){
  for (int j = 0; j < 100; j++){
    for(int k=0; k< j*j*j; k++){
       sum++;
    }
  }
}
cout << sum << endl;
```

Q8)

```
int sum = 0;
for (int i = 0; i < n * 2; i++)
{
  for (int j = 0; j < i/2; j++)
{
   for (int k= 0; k < n*n; k++){
       sum++;
   }
  }
}
cout << sum << endl;
```

Q9)

```
int count = 0;
for(int i = n; i > 0; i /= 2)
   for(int j = 0; j < i; j++)
      count++;
```

Q10)

```
int sum = 0;
for (int i=n; i>0; i/=2) {
  for (int j=1; j<n; j*=2) {
    for (int k=0; k<n; k+=2) {
        sum++;
    }
  }
}
cout << sum << endl;
```

Q11)

```
int sum = 0;
for (int i=1; i < n; i *= 2 ) {
  for (int j = n; j > 0; j/= 2
){
    for (int k = j; k < n;
k+=2){
        sum += (i + j * k );
    }
  }
}
cout << sum << endl;
```

Q12)

```
int sum = 0;
for (int i=1; i<n; i++) {
  for (int j=i; j<n; j++) {
    for (int k=0; k<j; k++) {
        sum++;
    }
  }
}
cout << sum << endl;
```

Q13) assuming that $n = 2^m$:

```
int sum = 0;
for (int i=n; i>0; i--) {
  for (int j=1; j<n; j*=2) {
    for (int k=0; k<j; k++) {
       sum++;
    }
  }
}
cout << sum << endl;
```

Exercise I.13 (Time Complexity of Recursive Algorithm)

The same question from the previous exercise for the following recursive call.

Q1)

```
int funct1(int n) {
  if (n <= 0)
     return 1;
  else
     return 1 + funct1(n-1);
}
```

Q2)

```
int funct2(int n) {
  if (n <= 0)
     return 1;
  else
     return 1 + funct2(n-5);
}
```

Q3)

```
int funct3(int n) {
  if (n <= 0)
     return 1;
  else
     return 1 + funct3(n/5);
}
```

Q4)

```
int funct4(int n) {
  for(int i=0; i<n; i++) {
     funct4(n-1);
  }
}
```

Q5)

```
int funct5(int n) {
  for (int i= 0; i < n; i += 2){
    // operation
  }
  if (n <= 0)
     return 1;
  else
     return 1 + funct5(n-5);
}
```

Q6)

```
void funct6(int a, int b, int c){
  if (a <= 0) {
     cout<<"funct6:"<<b<<c<<endl;
  }
  else {
     funct6(a-1, b+1, c);
     funct6(a-1, b, c+1);
  }
}
```