1 Exercises and Solutions

Most of the exercises below have solutions but you should try first to solve them. Each subsection with solutions is after the corresponding subsection with exercises. T

1.1 Time complexity and Big-Oh notation: exercises

1. A sorting method with "Big-Oh" complexity $O(n \log n)$ spends exactly 1 millisecond to sort 1,000 data items. Assuming that time $T(n)$ of sorting n items is directly proportional to $n \log n$, that is, $T(n) = cn \log n$, derive a formula for $T(n)$, given the time $T(N)$ for sorting N items, and estimate how long this method will sort 1,000,000 items.

2. A quadratic algorithm with processing time $T(n) = cn^2$ spends $T(N)$ seconds for processing N data items. How much time will be spent for processing $n = 5000$ data items, assuming that $N = 100$ and $T(N) = 1$ms?

3. An algorithm with time complexity $O(f(n))$ and processing time $T(n) = cf(n)$, where $f(n)$ is a known function of n, spends 10 seconds to process 1000 data items. How much time will be spent to process 100,000 data items if $f(n) = n$ and $f(n) = n^3$?

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

Expression	Dominant term(s)	$O(\ldots)$
$5 + 0.001n^3 + 0.025n$		
$500n + 100n^{1.5} + 50n \log_{10} n$		
$0.3n + 5n^{1.5} + 2.5 \cdot n^{1.75}$		
$n^2 \log_2 n + n(\log_2 n)^2$		
$n \log_3 n + n \log_2 n$		
$3 \log_8 n + \log_2 \log_2 \log_2 n$		
$100n + 0.01n^2$		
$0.01n + 100n^2$		
$2n + n^{0.5} + 0.5n^{1.25}$		
$0.01n \log_2 n + n(\log_2 n)^2$		
$100n \log_3 n + n^3 + 100n$		
$0.003 \log_4 n + \log_2 \log_2 n$		

5. The statements below show some features of "Big-Oh" notation for the functions $f \equiv f(n)$ and $g \equiv g(n)$. 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: $O(f+g) = O(f) + O(g)$		
Rule of products: $O(f \cdot g) = O(f) \cdot O(g)$		
Transitivity: if $g = O(f)$ and $h = O(f)$ then $g = O(h)$		
$5n + 8n^2 + 100n^3 = O(n^4)$		
$5n+8n^2+100n^3 = O(n^2 \log n)$		

6. Prove that $T(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3$ is $O(n^3)$ using the formal definition of the Big-Oh notation. *Hint:* Find a constant c and threshold n_0 such that $cn^3 \geq T(n)$ for $n \geq n_0$.

7. Algorithms **A** and **B** spend exactly $T_{\mathrm{A}}(n) = 0.1n^2 \log_{10} n$ and $T_{\mathrm{B}}(n) = 2.5n^2$ microseconds, respectively, for a problem of size n. Choose the algorithm, which is better in the Big-Oh sense, and find out a problem size n_0 such that for any larger size $n > n_0$ the chosen algorithm outperforms the other. If your problems are of the size $n \leq 10^9$, which algorithm will you recommend to use?

8. Algorithms **A** and **B** spend exactly $T_{\mathrm{A}}(n) = c_{\mathrm{A}} n \log_2 n$ and $T_{\mathrm{B}}(n) = c_{\mathrm{B}} n^2$ microseconds, respectively, for a problem of size n. Find the best algorithm for processing $n = 2^{20}$ data items if the algoritm **A** spends 10 microseconds to process 1024 items and the algorithm **B** spends only 1 microsecond to process 1024 items.

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

10. One of the two software packages, **A** or **B**, should be chosen to process very big databases, containing each up to 10^{12} records. Average processing time of the package **A** is $T_{\mathrm{A}}(n) = 0.1 \cdot n \cdot \log_2 n$ microseconds, and the average processing time of the package **B** is $T_{\mathrm{B}}(n) = 5 \cdot n$ microseconds.

Which algorithm has better performance in a "Big-Oh" sense? Work out exact conditions when these packages outperform each other.

11. One of the two software packages, **A** or **B**, should be chosen to process data collections, containing each up to 10^9 records. Average processing time of the package **A** is $T_{\mathrm{A}}(n) = 0.001n$ milliseconds and the average processing time of the package **B** is $T_{\mathrm{B}}(n) = 500\sqrt{n}$ milliseconds. Which algorithm has better performance in a "Big-Oh" sense? Work out exact conditions when these packages outperform each other.

12. Software packages **A** and **B** of complexity $O(n \log n)$ and $O(n)$, respectively, spend exactly $T_{\mathrm{A}}(n) = c_{\mathrm{A}} n \log_{10} n$ and $T_{\mathrm{B}}(n) = c_{\mathrm{B}} n$ milliseconds to process n data items. During a test, the average time of processing $n = 10^4$ data items with the package **A** and **B** is 100 milliseconds and 500 milliseconds, respectively. Work out exact conditions when one package actually outperforms the other and recommend the best choice if up to $n = 10^9$ items should be processed.

13. Let processing time of an algorithm of Big-Oh complexity $O(f(n))$ be directly proportional to $f(n)$. Let three such algorithms **A**, **B**, and **C** have time complexity $O(n^2)$, $O(n^{1.5})$, and $O(n \log n)$, respectively. During a test, each algorithm spends 10 seconds to process 100 data items. Derive the time each algorithm should spend to process 10,000 items.

14. Software packages **A** and **B** have processing time exactly $T_{\mathrm{EP}} = 3n^{1.5}$ and $T_{\mathrm{WP}} = 0.03n^{1.75}$, respectively. If you are interested in faster processing of up to $n = 10^8$ data items, then which package should be choose?

1.2 Time complexity and Big-Oh notation: solutions

1. Because processing time is $T(n) = cn \log n$, the constant factor $c = \frac{T(N)}{N \log N}$, and $T(n) = T(N)\frac{n \log n}{N \log N}$. Ratio of logarithms of the same base is independent of the base (see Appendix in the textbook), hence, any appropriate base can be used in the above formula (say, base of 10). Therefore, for $n = 1000000$ the time is $T(1,000,000) = T(1,000) \cdot \frac{1000000 \log_{10} 1000000}{1000 \log_{10} 1000} = 1 \cdot \frac{1000000 \cdot 6}{1000 \cdot 3} = 2,000$ ms

2. The constant factor $c = \frac{T(N)}{N^2}$, therefore $T(n) = T(N)\frac{n^2}{N^2} = \frac{n^2}{10000}$ ms and $T(5000) = 2,500$ ms.

3. The constant factor $c = \frac{T(1000)}{f(1000)} = \frac{10}{f(1000)}$ milliseconds per item. Therefore, $T(n) = 10\frac{f(n)}{f(1000)}$ ms and $T(100,000) = 10\frac{f(100,000)}{f(1000)}$ ms. If $f(n) = n$ then $T(100,000) = 1000$ ms. If $f(n) = n^3$, then $T(100,000) = 10^7$ ms.

4.

Expression	Dominant term(s)	$O(\ldots)$
$5 + 0.001n^3 + 0.025n$	$0.001n^3$	$O(n^3)$
$500n + 100n^{1.5} + 50n \log_{10} n$	$100n^{1.5}$	$O(n^{1.5})$
$0.3n + 5n^{1.5} + 2.5 \cdot n^{1.75}$	$2.5n^{1.75}$	$O(n^{1.75})$
$n^2 \log_2 n + n(\log_2 n)^2$	$n^2 \log_2 n$	$O(n^2 \log n)$
$n \log_3 n + n \log_2 n$	$n \log_3 n$, $n \log_2 n$	$O(n \log n)$
$3 \log_8 n + \log_2 \log_2 \log_2 n$	$3 \log_8 n$	$O(\log n)$
$100n + 0.01n^2$	$0.01n^2$	$O(n^2)$
$0.01n + 100n^2$	$100n^2$	$O(n^2)$
$2n + n^{0.5} + 0.5n^{1.25}$	$0.5n^{1.25}$	$O(n^{1.25})$
$0.01n \log_2 n + n(\log_2 n)^2$	$n(\log_2 n)^2$	$O(n(\log n)^2)$
$100n \log_3 n + n^3 + 100n$	n^3	$O(n^3)$
$0.003 \log_4 n + \log_2 \log_2 n$	$0.003 \log_4 n$	$O(\log n)$

5.

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

6. It is obvious that $T(n) \leq |a_0| + |a_1|n + |a_2|n^2 + |a_3|n^3$. Thus if $n \geq 1$, then $T(n) \leq cn^3$ where $c = |a_0| + |a_1| + |a_2| + |a_3|$ so that $T(n)$ is $O(n^3)$.

7. In the Big-Oh sense, the algorithm **B** is better. It outperforms the algo-

rithm **A** when $T_{\mathrm{B}}(n) \leq T_{\mathrm{A}}(n)$, that is, when $2.5n^2 \leq 0.1n^2 \log_{10} n$. This inequality reduces to $\log_{10} n \geq 25$, or $n \geq n_0 = 10^{25}$. If $n \leq 10^9$, the algorithm of choice is **A**.

8. The constant factors for **A** and **B** are:

$$c_{\mathrm{A}} = \frac{10}{1024 \log_2 1024} = \frac{1}{1024}; \quad c_{\mathrm{B}} = \frac{1}{1024^2}$$

Thus, to process $2^{20} = 1024^2$ items the algorithms **A** and **B** will spend

$$T_{\mathrm{A}}(2^{20}) = \frac{1}{1024} 2^{20} \log_2(2^{20}) = 20280\mu\mathrm{s} \text{ and } T_{\mathrm{B}}(2^{20}) = \frac{1}{1024^2} 2^{40} = 2^{20}\mu\mathrm{s},$$

respectively. Because $T_{\mathrm{B}}(2^{20}) \gg T_{\mathrm{A}}(2^{20})$, the method of choice is **A**.

9. In the Big-Oh sense, the algorithm **B** is better. It outperforms the algorithm **A** if $T_{\mathrm{B}}(n) \leq T_{\mathrm{A}}(n)$, that is, if $25n \leq 5n \log_{10} n$, or $\log_{10} n \geq 5$, or $n \geq 100,000$.

10. In the "Big-Oh" sense, the algorithm **B** of complexity $O(n)$ is better than **A** of complexity $O(n \log n)$. he package **B** has better performance in a The package **B** begins to outperform **A** when $(T_{\mathrm{A}}(n) \geq T_{\mathrm{B}}(n)$, that is, when $0.1n \log_2 n \geq 5 \cdot n$. This inequality reduces to $0.1 \log_2 n \geq 5$, or $n \geq 2^{50} \approx 10^{15}$. Thus for processing up to 10^{12} data items, the package of choice is **A**.

11. In the "Big-Oh" sense, the package **B** of complexity $O(n^{0.5})$ is better than **A** of complexity $O(n)$. The package **B** begins to outperform **A** when $(T_{\mathrm{A}}(n) \geq T_{\mathrm{B}}(n)$, that is, when $0.001n \geq 500\sqrt{n}$. This inequality reduces to $\sqrt{n} \geq 5 \cdot 10^5$, or $n \geq 25 \cdot 10^{10}$. Thus for processing up to 10^9 data items, the package of choice is **A**.

12. In the "Big-Oh" sense, the package **B** of linear complexity $O(n)$ is better than the package **A** of $O(n \log n)$ complexity. The processing times of the packages are $T_{\mathrm{A}}(n) = c_{\mathrm{A}} n \log_{10} n$ and $T_{\mathrm{B}}(n) = c_{\mathrm{B}} n$, respectively. The tests allows us to derive the constant factors:

$$\begin{array}{rcccc} c_{\mathrm{A}} & = & \frac{100}{10^4 \log_{10} 10^4} & = & \frac{1}{400} \\ c_{\mathrm{B}} & = & \frac{500}{10^4} & = & \frac{1}{20} \end{array}$$

The package **B** begins to outperform **A** when we must estimate the data size n_0 that ensures $T_{\mathrm{A}}(n) \geq T_{\mathrm{B}}(n)$, that is, when $\frac{n \log_{10} n}{400} \geq \frac{n}{20}$. This inequality reduces to $\log_{10} n \geq \frac{400}{20}$, or $n \geq 10^{20}$. Thus for processing up to 10^9 data items, the package of choice is **A**.

13.

	Complexity	Time to process 10,000 items
A1	$O(n^2)$	$T(10,000) = T(100) \cdot \frac{10000^2}{100^2} = 10 \cdot 10000 = 100,000$ sec.
A2	$O(n^{1.5})$	$T(10,000) = T(100) \cdot \frac{10000^{1.5}}{100^{1.5}} = 10 \cdot 1000 = 10,000$ sec.
A3	$O(n \log n)$	$T(10,000) = T(100) \cdot \frac{10000 \log 10000}{100 \log 100} = 10 \cdot 200 = 2,000$ sec.

14. In the Big-Oh sense, the package **A** is better. But it outperforms the package **B** when $T_{\mathrm{A}}(n) \leq T_{\mathrm{B}}(n)$, that is, when $3n^{1.5} \leq 0.03n^{1.75}$. This inequality reduces to $n^{0.25} \geq 3/0.03 (= 100)$, or $n \geq 10^8$. Thus for processing up to 10^8 data items, the package of choice is **B**.

1.3 Recurrences and divide-and-conquer paradigm: exercises

1. Running time $T(n)$ of processing n data items with a given algorithm is described by the recurrence:

$$T(n) = k \cdot T\left(\frac{n}{k}\right) + c \cdot n; \;\; T(1) = 0.$$

Derive a closed form formula for $T(n)$ in terms of c, n, and k. What is the computational complexity of this algorithm in a "Big-Oh" sense? *Hint*: To have the well-defined recurrence, assume that $n = k^m$ with the integer $m = \log_k n$ and k.

2. Running time $T(n)$ of processing n data items with another, slightly different algorithm is described by the recurrence:

$$T(n) = k \cdot T\left(\frac{n}{k}\right) + c \cdot k \cdot n; \;\; T(1) = 0.$$

Derive a closed form formula for $T(n)$ in terms of c, n, and k and detemine computational complexity of this algorithm in a "Big-Oh" sense. *Hint*: To have the well-defined recurrence, assume that $n = k^m$ with the integer $m = \log_k n$ and k.

3. What value of $k = 2$, 3, or 4 results in the fastest processing with the above algorithm? *Hint*: You may need a relation $\log_k n = \frac{\ln n}{\ln k}$ where ln denotes the natural logarithm with the base $e = 2.71828\ldots$).

4. Derive the recurrence that describes processing time $T(n)$ of the recursive method:

```
public static int recurrentMethod(
              int[] a, int low, int high, int goal ) {
  int target = arrangeTarget( a, low, high );
  if ( goal < target )
    return recurrentMethod( a, low, target-1, goal );
  else if ( goal > target )
    return recurrentMethod( a, target+1, high, goal );
  else
    return a[ target ];
}
```

The range of input variables is $0 \leq \texttt{low} \leq \texttt{goal} \leq \texttt{high} \leq \texttt{a.length} - 1$. A non-recursive method `arrangeTarget()` has linear time complexity $T(n) = c \cdot n$ where $n = \texttt{high} - \texttt{low} + 1$ and returns integer `target` in the range $\texttt{low} \leq \texttt{target} \leq \texttt{high}$. Output values of `arrangeTarget()` are equiprobable in this range, e.g. if $\texttt{low} = 0$ and $\texttt{high} = n - 1$, then every $\texttt{target} = 0, 1, \ldots, n-1$ occurs with the same probability $\frac{1}{n}$. Time for performing elementary operations like if–else or return should not be taken into account in the recurrence.

Hint: consider a call of `recurrentMethod()` for $n = \texttt{a.length}$ data items, e.g. `recurrentMethod( a, 0, a.length - 1, goal )` and analyse all recurrences for $T(n)$ for different input arrays. Each recurrence involves the number of data items the method recursively calls to.

5. Derive an explicit (closed–form) formula for time $T(n)$ of processing an array of size n if $T(n)$ is specified implicitly by the recurrence:

$$T(n) = \frac{1}{n}\left(T(0) + T(1) + \cdots + T(n-1)\right) + c \cdot n; \quad T(0) = 0$$

Hint: You might need the n-th harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln n + 0.577$ for deriving the explicit formula for $T(n)$.

6. Determine an explicit formula for the time $T(n)$ of processing an array of size n if $T(n)$ relates to the average of $T(n-1), \ldots, T(0)$ as follows:

$$T(n) = \frac{2}{n}\left(T(0) + \cdots + T(n-1)\right) + c$$

where T(0) = 0.

Hint: You might need the equation $\frac{1}{1\cdot 2} + \frac{1}{2\cdot 3} + \cdots + \frac{1}{n(n+1)} = \frac{n}{n+1}$ for deriving the explicit formula for $T(n)$.

7. The obvious linear algorithm for exponentiation x^n uses $n-1$ multiplications. Propose a faster algorithm and find its Big-Oh complexity in the case when $n = 2^m$ by writing and solving a recurrence formula.

1.4 Recurrences and divide-and-conquer paradigm: solutions

1. The closed-form formula can be derived either by "telescoping"[1] or by guessing from a few values and using then math induction.

[1] If you know difference equations in math, you will easily notice that the recurrences are the difference equations and "telescoping" is their solution by substitution of the same equation for gradually decreasing arguments: $n-1$ or n/k.

(a) Derivation with "telescoping": the recurrence $T(k^m) = k \cdot T(k^{m-1}) + c \cdot k^m$ can be represented and "telescoped" as follows:

$$\begin{array}{rcl} \frac{T(k^m)}{k^m} & = & \frac{T(k^{m-1})}{k^{m-1}} + c \\ \frac{T(k^{m-1})}{k^{m-1}} & = & \frac{T(k^{m-2})}{k^{m-2}} + c \\ \cdots & \cdots & \cdots \\ \frac{T(k)}{k} & = & \frac{T(1)}{1} + c \end{array}$$

Therefore, $\frac{T(k^m)}{k^m} = c \cdot m$, or $T(k^m) = c \cdot k^m \cdot m$, or $T(n) = c \cdot n \cdot \log_k n$. The Big-Oh complexity is $O(n \log n)$.

(b) Another version of telescoping: the like substitution can be applied directly to the initial recurrence:

$$\begin{array}{rcl} T(k^m) & = & k \cdot T(k^{m-1}) + c \cdot k^m \\ k \cdot T(k^{m-1}) & = & k^2 \cdot T(k^{m-2}) + c \cdot k^m \\ \cdots & \cdots & \cdots \\ k^{m-1} \cdot T(k) & = & k^m \cdot T(1) + c \cdot k^m \quad \text{so that } T(k^m) = c \cdot m \cdot k^m \end{array}$$

(c) Guessing and math induction: let us construct a few initial values of $T(n)$, starting from the given $T(1)$ and successively applying the recurrence:

$$\begin{array}{rcl} T(1) & = & 0 \\ T(k) & = & k \cdot T(1) + c \cdot k = c \cdot k \\ T(k^2) & = & k \cdot T(k) + c \cdot k^2 = 2 \cdot c \cdot k^2 \\ T(k^3) & = & k \cdot T(k^2) + c \cdot k^3 = 3 \cdot c \cdot k^3 \end{array}$$

This sequence leads to an assumption that $T(k^m) = c \cdot m \cdot k^m$. Let us explore it with math induction. The inductive steps are as follows:

(*i*) The base case $T(1) \equiv T(k^0) = c \cdot 0 \cdot k^0 = 0$ holds under our assumption.

(*ii*) Let the assumption holds for $l = 0, \ldots, m-1$. Then

$$\begin{array}{rcl} T(k^m) & = & k \cdot T(k^{m-1}) + c \cdot k^m \\ & = & k \cdot c \cdot (m-1) \cdot k^{m-1} + c \cdot k^m \\ & = & c \cdot (m-1+1) \cdot k^m = c \cdot m \cdot k^m \end{array}$$

Therefore, the assumed relationship holds for all values $m = 0, 1, \ldots, \infty$, so that $T(n) = c \cdot n \cdot \log_k n$.

The Big-Oh complexity of this algorithm is $O(n \log n)$.

2. The recurrence $T(k^m) = k \cdot T(k^{m-1}) + c \cdot k^{m+1}$ telescopes as follows:

$$\begin{array}{lcl} \frac{T(k^m)}{k^{m+1}} & = & \frac{T(k^{m-1})}{k^m} + c \\ \frac{T(k^{m-1})}{k^m} & = & \frac{T(k^{m-2})}{k^{m-1}} + c \\ \cdots & \cdots & \cdots \\ \frac{T(k)}{k^2} & = & \frac{T(1)}{k} + c \end{array}$$

Therefore, $\frac{T(k^m)}{k^{m+1}} = c \cdot m$, or $T(k^m) = c \cdot k^{m+1} \cdot m$, or $T(n) = c \cdot k \cdot n \cdot \log_k n$. The complexity is $O(n \log n)$ because k is constant.

3. Processing time $T(n) = c \cdot k \cdot n \cdot \log_k n$ can be easily rewritten as $T(n) = c\frac{k}{\ln k} \cdot n \cdot \ln n$ to give an explicit dependence of k. Because $\frac{2}{\ln 2} = 2.8854$, $\frac{3}{\ln 3} = 2.7307$, and $\frac{4}{\ln 4} = 2.8854$, the fastest processing is obtained for $k = 3$.

4. Because all the variants:

$$\begin{array}{llll} T(n) = T(0) + cn & \text{or} \quad T(n) = T(n-1) + cn & \text{if} & \texttt{target} = 0 \\ T(n) = T(1) + cn & \text{or} \quad T(n) = T(n-2) + cn & \text{if} & \texttt{target} = 1 \\ T(n) = T(2) + cn & \text{or} \quad T(n) = T(n-3) + cn & \text{if} & \texttt{target} = 2 \\ \cdots & \cdots & & \cdots \\ T(n) = T(n-1) + cn & \text{or} \quad T(n) = T(0) + cn & \text{if} & \texttt{target} = n-1 \end{array}$$

are equiprobable, then the recurrence is

$$T(n) = \frac{1}{n}\left(T(0) + \ldots + T(n-1)\right) + c \cdot n$$

5. The recurrence suggests that $nT(n) = T(0)+T(1)+\cdots+T(n-2)+T(n-1)+cn^2$. It follows that $(n-1)T(n-1) = T(0)+\ldots+T(n-2)+c\cdot(n-1)^2$, and by subtracting the latter equation from the former one, we obtain the following basic recurrence: $nT(n) - (n-1)T(n-1) = T(n-1) + 2cn - c$. It reduces to $nT(n) = n \cdot T(n-1) + 2cn - c$, or $T(n) = T(n-1) + 2c - \frac{c}{n}$. Telescoping results in the following system of equalities:

$$\begin{array}{lcllll} T(n) & = & T(n-1) & +2c & -\frac{c}{n} \\ T(n-1) & = & T(n-2) & +2c & -\frac{c}{n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ T(2) & = & T(1) & +2c & -\frac{c}{2} \\ T(1) & = & T(0) & +2c & -c \end{array}$$

Because $T(0) = 0$, the explicit expression for $T(n)$ is:

$$T(n) = 2cn - c \cdot \left(1 + \frac{1}{2} + \cdots + \frac{1}{n}\right) = 2cn - H_n \equiv 2cn - \ln n - 0.577$$

6. The recurrence suggests that $nT(n) = 2(T(0)+T(1)+\ldots+T(n-2)+T(n-1))+c\cdot n$. Because $(n-1)T(n-1) = 2(T(0)+\ldots+T(n-2))+c\cdot(n-1)$, the subtraction of the latter equality from the former one results in the following basic recurrence $nT(n) - (n-1)T(n-1) = 2T(n-1) + c$. It reduces to $nT(n) = (n+1)T(n-1) + c$, or $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{c}{n(n+1)}$. Telescoping results in the following system of equalities:

$$\begin{array}{lllll} \frac{T(n)}{n+1} & = & \frac{T(n-1)}{n} & + & \frac{c}{n(n+1)} \\ \frac{T(n-1)}{n} & = & \frac{T(n-2)}{n-1} & + & \frac{c}{(n-1)n} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \frac{T(2)}{3} & = & \frac{T(1)}{2} & + & \frac{c}{2\cdot 3} \\ \frac{T(1)}{2} & = & \frac{T(0)}{1} & + & \frac{c}{1\cdot 2} \end{array}$$

Because $T(0) = 0$, the explicit expression for $T(n)$ is:

$$\frac{T(n)}{n+1} = \frac{1}{1\cdot 2} + \frac{c}{2\cdot 3} + \cdots \frac{1}{(n-1)n} + \frac{c}{n(n+1)} = c\cdot\frac{n}{n+1}$$

so that $T(n) = c\cdot n$.

An alternative approach (guessing and math induction):

$$\begin{array}{lllll} T(0) & = & 0 & & \\ T(1) & = & \frac{2}{1}\cdot 0 + c & = & c \\ T(2) & = & \frac{2}{2}(0+c)+c & = & 2c \\ T(3) & = & \frac{2}{3}(0+c+2c)+c & = & 3c \\ T(4) & = & \frac{2}{4}(0+c+2c+3c)+c & = & 4c \end{array}$$

It suggests an assumption $T(n) = cn$ to be explored with math induction:

(i) The assumption is valid for $T(0) = 0$.

(ii) Let it be valid for $k = 1,\ldots,n-1$, that is, $T(k) = kc$ for $k = 1,\ldots,n-1$. Then

$$\begin{array}{lll} T(n) & = & \frac{2}{n}(0+c+2c+\ldots+(n-1)c)+c \\ & = & \frac{2}{n}\frac{(n-1)n}{2}c + c \\ & = & (n-1)c + c = cn \end{array}$$

The assumption is proven, and $T(n) = cn$.

7. A straightforward linear exponentiation needs $\frac{n}{2} = 2^{m-1}$ multiplications to produce x^n after having already $x^{\frac{n}{2}}$. But because $x^n = x^{\frac{n}{2}}\cdot x^{\frac{n}{2}}$, it can be computed with only one multiplication more than to compute $x^{\frac{n}{2}}$. Therefore, more efficient exponentiation is performed as follows: $x^2 = x\cdot x$, $x^4 = x^2\cdot x^2$, $x^8 = x^4\cdot x^4$, etc. Processing time for such more efficient

algorithm corresponds to a recurrence: $T(2^m) = T(2^{m-1}) + 1$, and by telescoping one obtains:

$$\begin{array}{lcl} T(2^m) & = & T(2^{m-1}) + 1 \\ T(2^{m-1}) & = & T(2^{m-2}) + 1 \\ \ldots & & \ldots \\ T(2) & = & T(1) + 1 \\ T(1) & = & 0 \end{array}$$

that is, $T(2^m) = m$, or $T(n) = \log_2 n$. The "Big-Oh" complexity of such algorithm is $O(\log n)$.

1.5 Time complexity of code: exercises

1. Work out the computational complexity of the following piece of code:

```
for( int i = n;  i > 0;  i /= 2 ) {
  for( int j = 1;  j < n;  j *= 2 ) {
    for( int k = 0;  k < n;  k += 2 ) {
         ... // constant number of operations
    }
  }
}
```

2. Work out the computational complexity of the following piece of code.

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

3. Work out the computational complexity of the following piece of code assuming that $n = 2^m$:

```
for( int i = n;  i > 0;  i-- ) {
  for( int j = 1;  j < n;  j *= 2 ) {
    for( int k = 0;  k < j;  k++ ) {
         ... // constant number C of operations
    }
  }
}
```

4. Work out the computational complexity (in the "Big-Oh" sense) of the following piece of code and explain how you derived it using the basic features of the "Big-Oh" notation:

```
for( int bound = 1; bound <= n; bound *= 2 ) {
   for( int i = 0; i < bound; i++ ) {
      for( int j = 0; j < n; j += 2 ) {
         ... // constant number of operations
      }
      for( int j = 1; j < n; j *= 2 ) {
         ... // constant number of operations
      }
   }
}
```

5. Determine the average processing time $T(n)$ of the recursive algorithm:

```
1    int myTest( int n ) {
2      if ( n <= 0 ) return 0;
3      else {
4        int i = random( n - 1 );
5        return myTest( i ) + myTest( n - 1 - i );
6      }
7    }
```

   providing the algorithm `random( int n )` spends one time unit to return a random integer value uniformly distributed in the range $[0, n]$ whereas all other instructions spend a negligibly small time (e.g., $T(0) = 0$).

   Hints: derive and solve the basic recurrence relating $T(n)$ in average to $T(n-1), \ldots, T(0)$. You might need the equation $\frac{1}{1\cdot 2} + \frac{1}{2\cdot 3} + \cdots + \frac{1}{n(n+1)} = \frac{n}{n+1}$ for deriving the explicit formula for $T(n)$.

6. Assume that the array a contains n values, that the method `randomValue` takes constant number c of computational steps to produce each output value, and that the method `goodSort` takes $n \log n$ computational steps to sort the array. Determine the Big-Oh complexity for the following fragments of code taking into account only the above computational steps:

```
for( i = 0; i < n; i++ ) {
  for( j = 0; j < n; j++ )
    a[ j ] = randomValue( i );
  goodSort( a );
}
```

1.6 Time complexity of code: solutions

1. In the outer `for`-loop, the variable `i` keeps halving so it goes round $\log_2 n$ times. For each `i`, next loop goes round also $\log_2 n$ times, because of doubling the variable `j`. The innermost loop by `k` goes round $\frac{n}{2}$ times. Loops are nested, so the bounds may be multiplied to give that the algorithm is $O\left(n(\log n)^2\right)$.

2. Running time of the inner, middle, and outer loop is proportional to n, $\log n$, and $\log n$, respectively. Thus the overall Big-Oh complexity is $O(n(\log n)^2)$.

   More detailed optional analysis gives the same value. Let $n = 2^k$. Then the outer loop is executed k times, the middle loop is executed $k + 1$ times, and for each value $j = 2^k, 2^{k-1}, \ldots, 2, 1$, the inner loop has different execution times:

j	Inner iterations
2^k	1
2^{k-1}	$(2^k - 2^{k-1})\frac{1}{2}$
2^{k-2}	$(2^k - 2^{k-2})\frac{1}{2}$
$\ldots$	$\ldots$
2^1	$(2^k - 2^1)\frac{1}{2}$
2^0	$(2^k - 2^0)\frac{1}{2}$

   In total, the number of inner/middle steps is

   $$\begin{aligned} 1 + k \cdot 2^{k-1} - (1 + 2 + \ldots + 2^{k-1})\frac{1}{2} &= 1 + k \cdot 2^{k-1} - (2^k - 1)\frac{1}{2} \\ &= 1.5 + (k-1) \cdot 2^{k-1} \equiv (\log_2 n - 1)\frac{n}{2} \\ &= O(n \log n) \end{aligned}$$

   Thus, the total complexity is $O(n(\log n)^2)$.

3. The outer `for`-loop goes round n times. For each `i`, the next loop goes round $m = \log_2 n$ times, because of doubling the variable `j`. For each `j`, the innermost loop by `k` goes round `j` times, so that the two inner loops together go round $1 + 2 + 4 + \ldots + 2^{m-1} = 2^m - 1 \approx n$ times. Loops are nested, so the bounds may be multiplied to give that the algorithm is $O\left(n^2\right)$.

4. The first and second successive innermost loops have $O(n)$ and $O(\log n)$ complexity, respectively. Thus, the overall complexity of the innermost part is $O(n)$. The outermost and middle loops have complexity $O(\log n)$ and $O(n)$, so a straightforward (and valid) solution is that the overall complexity is $O(n^2 \log n)$.

More detailed analysis shows that the outermost and middle loops are interrelated, and the number of repeating the innermost part is as follows:

$$1 + 2 + \ldots + 2^m = 2^{m+1} - 1$$

where $m = \lfloor \log_2 n \rfloor$ is the smallest integer such that $2^{m+1} > n$. Thus actually this code has quadratic complexity $O(n^2)$. But selection of either answer ($O(n^2 \log n)$ and $O(n^2)$ is valid.

5. The algorithm suggests that $T(n) = T(i) + T(n-1-i) + 1$. By summing this relationship for all the possible random values $i = 0, 1, \ldots, n-1$, we obtain that in average $nT(n) = 2(T(0)+T(1)+\ldots+T(n-2)+T(n-1))+n$. Because $(n-1)T(n-1) = 2(T(0) + \ldots + T(n-2)) + (n-1)$, the basic recurrence is as follows: $nT(n) - (n-1)T(n-1) = 2T(n-1) + 1$, or $nT(n) = (n+1)T(n-1) + 1$, or $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{1}{n(n+1)}$. The telescoping results in the following system of expressions:

$$\begin{array}{lllll}
\frac{T(n)}{n+1} & = & \frac{T(n-1)}{n} & + & \frac{1}{n(n+1)} \\
\frac{T(n-1)}{n} & = & \frac{T(n-2)}{n-1} & + & \frac{1}{(n-1)n} \\
\ldots & \ldots & \ldots & \ldots & \ldots \\
\frac{T(2)}{3} & = & \frac{T(1)}{2} & + & \frac{1}{2\cdot 3} \\
\frac{T(1)}{2} & = & \frac{T(0)}{1} & + & \frac{1}{1\cdot 2}
\end{array}$$

Because $T(0) = 0$, the explicit expression for $T(n)$ is:

$$\frac{T(n)}{n+1} = \frac{1}{1\cdot 2} + \frac{1}{2\cdot 3} + \cdots \frac{1}{(n-1)n} + \frac{1}{n(n+1)} = \frac{n}{n+1}$$

so that $T(n) = n$.

6. The inner loop has linear complexity cn, but the next called method is of higher complexity $n \log n$. Because the outer loop is linear in n, the overall complexity of this piece of code is $n^2 \log n$.