제 4 장 Algorithms

4.1 Introduction

- An algorithm is a step-by-step method of solving some problem.
- Algorithms typically have characteristics:
  - **Input** The algorithm receives input.
  - **Output** The algorithm produces output.
  - **Precision** The steps are precisely stated.
  - **Determinism** The intermediate results of each step of execution are unique and are determined only by the inputs and the results of the preceding steps.
  - **Finiteness** The algorithm terminates; that is, it stops after finitely many instructions have been executed.
  - **Correctness** The output produced by the algorithm is correct; that is, the algorithm correctly solves the problem.
  - **Generality** The algorithm applies to a set of inputs.

4.1 Introduction(Finding the Maximum of Three Numbers)

- 의사코드pseudocode는 Python 처럼 실제 코드와 비슷하다.
- Algorithm 4.1.1 This algorithm finds the largest of the numbers a, b, and c.

Input: a, b, c

Output: *large* (the largest of a, b, and c)

```
1. max3(a, b, c) {
2.     large = a
3.     if (b > large) // if b is larger than large, update large
4.         large = b
5.     if (c > large) // if c is larger than large, update large
6.         large = c
7.     return large
8. }
```

4.1 Introduction(Finding the Maximum Value in a Sequence)

- Algorithm 4.1.2 This algorithm finds the largest of the numbers $s_1, \ldots, s_n$.
  Input: s, n
  Output: *large*(the largest value in the sequence s)

```
max(s, n) {
    $large = s_1$
    for $i$ = 2 to $n$
        if ($s_i > large$)
            $large = s_i$
    return $large$
}
```

- *large* is the largest value in the subsequence $s_1, \ldots, s_i$.
  Large is a loop invariant using induction on i

4.2 Examples of Algorithms(Search)

- The Text_Search algorithm searches for an occurrence of the pattern p in text t. It returns the smallest index i such that p occurs in t starting at index i.
  If p does not occur in t, it returns 0.
- Check whether p occurs at index 1 in t. If so, the first occurrence of p in t was found.
  If not, check whether p occurs at index 2 in t, and so on.

4.2 Examples of Algorithms(Text Search)

Algorithm 4.2.1 Input: p (indexed from 1 to m), m, t (indexed from 1 to n), n
Output: i

```
text_search($p$, $m$, $t$, $n$) {
  for $i = 1$ to $n - m + 1$ {
    $j = 1$
    // $i$ is the index in $t$ of the first character of the substring
    // to compare with $p$, and $j$ is the index in $p$

    // the while loop compares $t_i \cdots t_{i+m-1}$ and $p_1 \cdots p_m$
    while ($t_{i+j-1} == p_j$) {
      $j = j + 1$
      if ($j > m$)
        return $i$
    }
  }
  return 0
}
```

$$\boxed{|t_{n-m+1} t_{n-m+2} \cdots t_{n-m+m}| = m}$$

…	t_{i+0}	t_{i+1}	t_{i+2}	…	t_{i+m-1}	…
	p_1	p_2	p_3	…	p_m	

4.2 Examples of Algorithms(Sorting)

- Insertion sort is one of the fastest algorithms for sorting small sequences.
- At the ith iteration of insertion sort, the first part of the sequence $s_1, \ldots, s_i$ will have been rearranged so that it is sorted.
- Insertion sort inserts s_{i+1} in $s_1, \ldots, s_i$ so that $s_1, \ldots, s_i, s_{i+1}$ is sorted. e.g., Inserts $s_5 = 16$, in nondecreasing order.

$s_1, \ldots, s_4$

8	13	20	27

Since $27 > 16$, 27 moves the right.

8	13	20		27

Since $20 > 16$, 20 moves the right.

8	13		20	27

Since $13 < 16$, insert 16.

8	13	16	20	27

4.2 Examples of Algorithms(Sorting)

- Algorithm 4.2.3 This algorithm sorts the sequence $s_1, \ldots, s_n$ in nondecreasing order. Input: $s, n,$ Output: s (sorted)

```
insertion_sort($s, n$) {
  for $i$ = 2 to $n$ {
    $val = s_i$  // save $s_i$
    $j = i - 1$
    // if $val < s_j$, move $s_j$ right to make room for $s_i$
    while ($j \geq 1 \wedge val < s_j$) {
      $s_{j+1} = s_j$
      $j = j - 1$
    }
    $s_{j+1} = val$  // insert val
  }
}
```

4.2 Examples of Algorithms(Time Space of Algorithms)

- Knowing the time (e.g., the number of steps) and space (e.g., the number of variables, length of the sequences involved) required by algorithms allows us to compare algorithms that solve the same problem. e.g., n steps or n^2 steps.

- Even for a fixed size n, the time required by Algorithm 4.2.3 depends on the input.
  - If the input sequence is already sorted in nondecreasing order, the body of the while loop will never be executed.
    We call this time the **best-case time**.
  - If the input sequence is sorted in decreasing order, the while loop will execute the maximum number of times.
    We call this time the **worst-case time**.

4.2 Examples of Algorithms(Shuffle)

- Algorithm 4.2.4 This algorithm shuffles the values in the sequence $a_1, \ldots, a_n$.
  // $rand(i, j)$ returns a random integer between i and j.
  Input: a, n Output: a (shuffled)

```
shuffle($a, n$) {
    for $i = 1$ to $n - 1$
        $swap\left(a_i, a_{rand(i,n)}\right)$
}
```

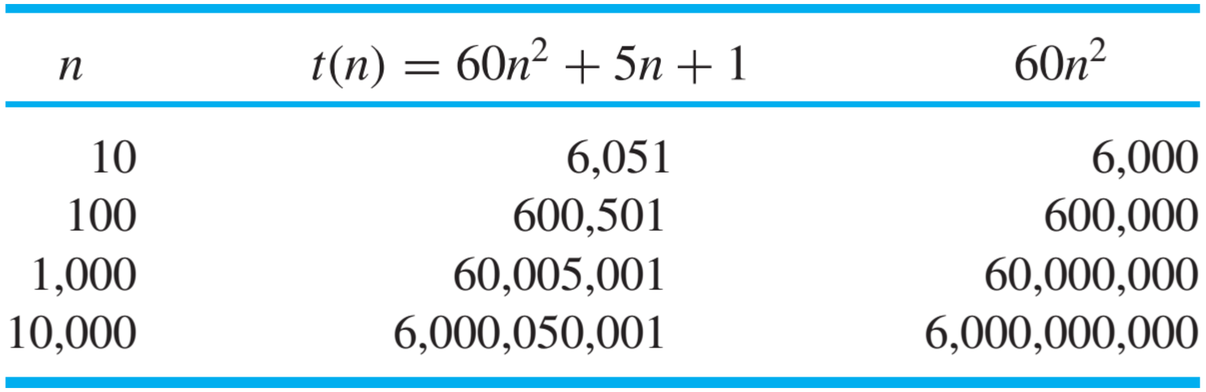
17	9	5	23	21	$i = 1, rand(1, 5) = 3$
5	9	17	23	21	$i = 2, rand(2, 5) = 5$
5	21	17	23	9	$i = 3, rand(3, 5) = 3$
5	21	17	23	9	$i = 4, rand(4, 5) = 5$
5	21	17	9	23	

4.3 Analysis of Algorithms

- Analysis of an algorithm refers to the process of deriving estimates for the time and space needed to execute the algorithm.
- The time needed to execute an algorithm is a function of the input. We use parameters that characterize the size of the input n. e.g., $2^{100} \approx 1.267 \times 10^{30}$, $100! \approx 9.33 \times 10^{157}$
- **Best-case time** for inputs of size n: the minimum time needed to execute the algorithm among all inputs of size n
- **Worst-case time** for inputs of size n: the maximum time needed to execute the algorithm among all inputs of size n
- **Average-case time:** the average time needed to execute the algorithm over some finite set of inputs all of size n.

4.3 Analysis of Algorithms(실행 시간)

- We are in interested how the best-case or worst-case time grows as the size of the input increases.
- Suppose that the worst-case time of an algorithm is
$$t(n) = 60n^2 + 5n + 1$$
for input of size n. For large n, $t(n)$ is approximately equal to the term $60n^2$.
- $t(n)$ grows like n^2 as n increases.



n	$t(n) = 60n^2 + 5n + 1$	$60n^2$
10	6,051	6,000
100	600,501	600,000
1,000	60,005,001	60,000,000
10,000	6,000,050,001	6,000,000,000

4.3 Analysis of Algorithms

- 정의 4.3.2 Let f and g be functions with domain $\{1, 2, 3, \ldots\}$.
- $f(n) = O(g(n))$
  if there exists a positive constant C_1 such that
  $|f(n)| \leq C_1|g(n)|$ for all but finitely many positive integers n.

  $f(n)$ is of order at most $g(n)$ or $f(n)$ is big oh of $g(n)$
- $f(n) = \Omega(g(n))$
  if there exists a positive constant C_2 such that
  $|f(n)| \geq C_2\,|g(n)|$ for all but finitely many positive integers n.

  $f(n)$ is of order at least $g(n)$ or $f(n)$ is omega of $g(n)$
- $f(n) = \Theta(g(n))$
  if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

  $f(n)$ is of order $g(n)$ or $f(n)$ is theta of $g(n)$

4.3 Analysis of Algorithms

- 예제 4.3.3 $60n^2 + 5n + 1 \le 60n^2 + 5n^2 + n^2 = 66n^2$ for all $n \ge 1$,

  we may take $C_1 = 66$ to obtain
  $$60n^2 + 5n + 1 = O(n^2).$$

- Since
  $$60n^2 + 5n + 1 \ge 60n^2 \text{ for all } n \ge 1,$$

- we may take $C_2 = 60$ to obtain
  $$60n^2 + 5n + 1 = \Omega(n^2).$$

- Since $60n^2 + 5n + 1 = O(n^2)$ and $60n^2 + 5n + 1 = \Omega(n^2)$,
  $$60n^2 + 5n + 1 = \Theta(n^2).$$

4.3 Analysis of Algorithms

- 정리 4.3.4 Let $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$ be a polynomial in n of degree k, where each a_i is nonnegative. Then $p(n) = \Theta(n^k)$.

- Proof) We show that $p(n) = O(n^k)$ and $p(n) = \Omega(n^k)$.

- Let $C_1 = a_k + a_{k-1} + \cdots + a_1 + a_0$. Then, for all n,

$$\begin{aligned} p(n) &= a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \\ &\le a_k n^k + a_{k-1} n^k + \cdots + a_1 n^k + a_0 n^k \\ &= (a_k + a_{k-1} + \cdots + a_1 + a_0) n^k = C_1 n^k \end{aligned}$$

  Therefore, $p(n) = O(n^k)$.

- For all n,

  $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \ge a_k n^k = C_2 n^k$, where $C_2 = a_k$. Therefore, $p(n) = \Omega(n^k)$.

4.3 Analysis of Algorithms

- 예제 4.3.7 We show that

$$1 + 2 + \cdots + n = O(n^2) \text{ and } 1 + 2 + \cdots + n = \Omega(n^2)$$

- $1 + 2 + \cdots + n \leq n + n + \cdots + n = n \cdot n = n^2$ for all $n \geq 1$. (3.1)
  It follows that $1 + 2 + \cdots + n = O(n^2)$.

- Throw away approximately the first half of the terms, to obtain

$$\begin{aligned} 1 + 2 + \cdots + n &= 1 + 2 + \cdots + \lceil (n+1)/2 \rceil + \cdots + (n-1) + n \\ &\geq \lceil (n+1)/2 \rceil + \cdots + (n-1) + n \\ &\geq \lceil (n+1)/2 \rceil + \cdots + \lceil (n+1)/2 \rceil + \lceil (n+1)/2 \rceil \\ &= \lceil n/2 \rceil \cdot \lceil (n+1)/2 \rceil \geq (n/2)(n/2) = n^2/4 \quad (3.2) \end{aligned}$$

  for all $n \geq 1$. We can now conclude that

$$1 + 2 + \cdots + n = \Omega(n^2).$$

4.3 Analysis of Algorithms

- 예제 4.3.8 Let k be a positive integer. We show that $1^k + 2^k + \cdots + n^k = O(n^{k+1})$ and $1^k + 2^k + \cdots + n^k = \Omega(n^{k+1})$

- $1^k + 2^k + \cdots + n^k \leq n^k + n^k + \cdots + n^k = n \cdot n^k = n^{k+1}$ for all $n \geq 1$; hence $1^k + 2^k + \cdots + n^k = O(n^{k+1})$.

- $$\begin{aligned} 1^k + 2^k + \cdots + n^k &= 1^k + 2^k + \cdots + \lceil (n+1)/2 \rceil^k + \cdots + n^k \\ &\geq \lceil (n+1)/2 \rceil^k + \cdots + n^k \\ &\geq \lceil (n+1)/2 \rceil^k + \cdots + \lceil (n+1)/2 \rceil^k \\ &= \lceil n/2 \rceil \lceil (n+1)/2 \rceil^k \\ &\geq (n/2)(n/2)^k = (1/2^{k+1}) n^{k+1} \end{aligned}$$

  for all $n \geq 1$. We conclude that $1^k + 2^k + \cdots + n^k = \Omega(n^{k+1})$

4.3 Analysis of Algorithms

- 예제 4.3.9 $\lg n! = \lg n + \lg(n-1) + \cdots + \lg 1$ for all $n \geq 1$.

$\boxed{\lg n = \log_2 n}$

- Since lg is an increasing function,
$\lg n + \lg(n-1) + \cdots + \lg 1 \leq \lg n + \lg n + \cdots + \lg n = n \lg n$
for all $n \geq 1$. We conclude that $\lg n! = O(n \lg n)$.

- For all $n \geq 4$, we have

$$
\begin{aligned}
&\lg n + \lg(n-1) + \cdots + \lg 1 \\
&\quad = \lg n + \lg(n-1) + \cdots + \lg\lceil (n+1)/2 \rceil + \lg(n/2) \ldots + \lg 1 \\
&\quad \geq \lg\lceil (n+1)/2 \rceil + \cdots + \lg\lceil (n+1)/2 \rceil = \lceil n/2 \rceil \lg\lceil (n+1)/2 \rceil \\
&\quad \geq (n/2)\lg(n/2) = (n/2)[\lg n - \lg 2] \\
&\quad = (n/2)[(\lg n)/2 + ((\lg n)/2 - 1)] \\
&\quad \geq (n/2)(\lg n)/2 = n \lg n/4
\end{aligned}
$$

$\boxed{\begin{array}{l} \lg 2 = \log_2 2 = 1 \\ \lg n/2 \geq 1 \text{ for all } n > 4 \end{array}}$

Therefore, $\lg n! = \Omega(n \lg n)$.

4.3 Analysis of Algorithms

- 예제 4.3.10 Show that if $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$.
- Proof Because $f(n) = \Theta(g(n))$, there are constants C_1 and C_2 such that

$$C_1|g(n)| \le |f(n)| \le C_2|g(n)|$$

for all but finitely many positive integers n.
Because $g(n) = \Theta(h(n))$, there are constants C_3 and C_4 such that

$$C_3|h(n)| \le |g(n)| \le C_4|h(n)|$$

for all but finitely many positive integers n.
Therefore,

$$C_1C_3|h(n)| \le C_1|g(n)| \le |f(n)| \le C_2|g(n)| \le C_2C_4|h(n)|$$

for all but finitely many positive integers n.
It follows that $f(n) = \Theta(h(n))$.

4.3 Analysis of Algorithms

- 예제 4.3.12 Suppose that an algorithm is known to take

$$60n^2 + 5n + 1$$

units of time to terminate in the worst case for inputs of size n. Since $60n^2 + 5n + 1 = \Theta(n^2)$, the worst-case time required by this algorithm is $\Theta(n^2)$.

- 예제 4.3.13 Find a theta notation in terms of n for the number of times the statement $x = x + 1$ is executed.

```
1. for $i = 1$ to $n$
2.     for $j = 1$ to $i$
3.         $x = x + 1$
```

Sol) First, i is set to 1 and, as j runs from 1 to 1, line 3 is executed 1 time. Next, i is set to 2 and, as j runs from 1 to 2, line 3 is executed 2 times, and so on. Thus the total number of times line 3 is executed is $1 + \cdots + n = \Theta(n^2)$.

4.3 Analysis of Algorithms(theta notation)

- 예제 4.3.14 Find a theta notation in terms of n for the number of times the statement $x = x + 1$ is executed:

```
1. $i = n$
2. while ($i \geq 1$) {
3.    $x = x + 1$
4.    $i = \lfloor i / 2 \rfloor$
5. }
```

- If n is an arbitrary positive integer, for some $k \geq 1$,

$$2^{k-1} \leq n < 2^k.$$

We use induction on k to show that in this case the statement $x = x + 1$ is executed k times.

If $k = 1$, we have $1 = 2^{1-1} \leq n < 2^1 = 2$. Therefore, n is 1.

In this case, the statement $x = x + 1$ is executed once.

Thus the Basis Step is proved.

4.3 Analysis of Algorithms(theta notation)

- Now suppose that if n satisfies
$$2^{k-1} \leq n < 2^k, \qquad (3.3)$$
the statement $x = x + 1$ is executed k times.
We must show that if n satisfies
$$2^k \leq n < 2^{k+1}, \qquad (3.4)$$
the statement $x = x + 1$ is executed $k + 1$ times.

```
1. $i = n$
2. while ($i \geq 1$) {
3.     $x = x + 1$
4.     $i = \lfloor i / 2 \rfloor$
5. }
```

- Suppose that n satisfies (3.4). At line 4, i is assigned to $\lfloor n/2 \rfloor$. Notice that $2^{k-1} \leq n/2 < 2^k$. Because 2^{k-1} is an integer, we must also have $2^{k-1} \leq \lfloor n/2 \rfloor < 2^k$.
By the inductive assumption (3.3), the statement $x = x + 1$ is executed k more times, for a total of $k + 1$ times.
The Inductive Step is complete.
- Therefore, if n satisfies (3.3), the statement $x = x + 1$ is executed k times.

4.3 Analysis of Algorithms(theta notation)

- If n satisfies

$$2^{k-1} \leq n < 2^k, \qquad (3.3)$$

  the statement $x = x + 1$ is executed k times.

- Suppose that n satisfies (3.3). Taking logarithms to the base 2, we have

$$k - 1 \leq \lg n < k.$$

- Therefore, k, the number of times the statement $x = x + 1$ is executed, satisfies $\lg n < k \leq 1 + \lg n$. Because k is an integer, $\lfloor \lg n \rfloor < k \leq 1 + \lfloor \lg n \rfloor$. It follows that $k = 1 + \lfloor \lg n \rfloor$. Since $1 + \lfloor \lg n \rfloor = \Theta(\lg n)$, a theta notation for the number of times the statement $x = x + 1$ is executed is $\Theta(\lg n)$.

4.3 Analysis of Algorithms(theta notation)

- 예제 4.3.15 Find a theta notation in terms of n for the number of times the statement $x = x + 1$ is executed.

```
1. $j = n$
2. while ($j \geq 1$) {
3.    for $i = 1$ to $j$
4.       $x = x + 1$
5.    $j = \lfloor j/2 \rfloor$
6. }
```

- Let $t(n)$ denote the number of times we execute the statement $x = x + 1$.
- The first time we arrive at the body of the while loop, the statement $x = x + 1$ is executed n times. Therefore, $t(n) \geq n$ for all $n \geq 1$ and $t(n) = \Omega(n)$.

4.3 Analysis of Algorithms(theta notation)

- After j is set to n, we arrive at the while loop for the first time. The statement $x = x + 1$ is executed n times. At line 5, j is replaced by $\lfloor n/2 \rfloor$; hence $j \le n/2$.
  If $j \ge 1$, we will execute $x = x + 1$ at most $n/2$ additional times in the next iteration of the while loop, and so on.

- If we let k denote the number of times we execute the body of the while loop, the number of times we execute $x = x + 1$ is at most

$$n + \frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{2^{k-1}}$$

```
1. $j = n$
2. while ($j \ge 1$) {
3.     for $i = 1$ to $j$
4.         $x = x + 1$
5.     $j = \lfloor j/2 \rfloor$
6. }
```

- Now

$$t(n) \le \frac{n(1 - 1/2^k)}{1 - 1/2} = 2n\left(1 - \frac{1}{2^k}\right) \le 2n$$

  for all $n \ge 1$, so $t(n) = O(n)$.

- Thus a theta notation for the number of times we execute $x = x + 1$ is $\Theta(n)$.

4.3 Analysis of Algorithms(Searching an Unordered Sequence)

- Algorithm 4.3.17 Searching an Unordered Sequence
  Given the sequence $s_1, \dots, s_n$ and a value key,
  this algorithm returns the index of key.
  If key is not found, the algorithm returns 0.

- Input: $s_1, \dots, s_n$, n, and key (the value to search for)
  Output: The index of key, or if key is not found, 0

-
```
1. linear_search($s, n, key$) {
2.    for $i = 1$ to $n$
3.       if ($key == s_i$)
4.          return $i$         // successful search
5.    return 0              // unsuccessful search
6. }
```

4.3 Analysis of Algorithms(Searching an Unordered Sequence)

- The best-case time
  If $s_1 = key$, line 3 is executed once. Thus the best-case time the algorithm is $\Theta(1)$.

- The worst-case time
  If key is not in the sequence, line 3 will be executed n times, so the worst-case time of the algorithm is $\Theta(n)$.

- The average-case time
  If key is found at the ith position, line 3 is executed i times; if key is not in the sequence, line 3 is executed n times. Thus the average number of times line 3 is executed is

$$\frac{(1 + 2 + \cdots + n) + n}{n + 1}$$

4.3 Analysis of Algorithms(Searching an Unordered Sequence)

- Now

$$\frac{(1 + 2 + \cdots + n) + n}{n + 1} \leq \frac{n^2 + n}{n + 1} \qquad \text{by (3.1)}$$
$$= \frac{n(n+1)}{n+1} = n$$

- So the average-case time of the algorithm is $O(n)$. Also,

$$\frac{(1 + 2 + \cdots + n) + n}{n + 1} \geq \frac{n^2/4 + n}{n + 1} \qquad \text{by (3.2)}$$
$$\geq \frac{n^2/4 + n/4}{n+1} = \frac{n}{4}$$

- So the average-case time of the algorithm is $\Omega(n)$.
- Thus the average-case time of the algorithm is $\Theta(n)$.

4.4 Recursive Algorithms

- A **recursive function** is a function that invokes itself. A **recursive algorithm** is an algorithm that contains a recursive function.
- Recursion is a powerful, elegant, and natural way to solve a large class of problems. A problem in this class can be solved using a **divide-and-conquer** technique in which the problem is decomposed into problems of the same type as the original problem.
- Each subproblem, in turn, is decomposed until the process yields subproblems that can be solved in a straightforward way.
- Finally, solutions to the subproblems are combined to obtain a solution to the original problem.

4.4 Recursive Algorithms($n!$)

- Algorithm 4.4.2 This recursive algorithm computes $n!$.
  Input: n, an integer greater than or equal to 0
  Output: $n!$

-
  ```
  1. $factorial(n)$ {
  2.    if ($n == 0$)        // the base cases
  3.       return 1
  4.    return $n * factorial(n - 1)$
  5. }
  ```

- We call the values for which a recursive function does *not* invoke itself the base cases.

4.4 Recursive Algorithms

- Theorem 4.4.3 Algorithm 4.4.2 returns $n!$, $n \geq 0$.
- Proof) Basis Step $(n = 0)$

  If $n = 0$, at line 3 the function correctly returns 1.

  Inductive Step Assume that Algorithm 4.4.2 correctly returns the value of $(n-1)!$, $n > 0$.
  Now suppose that n is input to Algorithm 4.4.2.
  Since $n \neq 0$, when we execute the function in Algorithm 4.4.2 we proceed to line 4. By the inductive assumption, the function correctly computes the value of $(n-1)!$. At line 4, the function correctly computes the value $(n-1)! \cdot n = n!$.
  Therefore, Algorithm 4.4.2 correctly returns the value of $n!$ for every integer $n \geq 0$.

```
1. $factorial(n)$ {
2.     if ($n$ == 0)
3.         return 1
4.     return $n * factorial(n - 1)$
5. }
```

4.4 Recursive Algorithms(피보나치 수열)

- 예제 4.4.7 $f_1 = 1,\ f_2 = 1,\ f_n = f_{n-1} + f_{n-2}$ for $n \geq 3$
- Show that $\sum_{k=1}^{n} f_k = f_{n+2} - 1$ for all $n \geq 1.$
- Basis step $(n = 1)$

  We must show that $\sum_{k=1}^{1} f_k = f_3 - 1.$

  Since $\sum_{k=1}^{1} f_k = f_1 = 1$ and $f_3 - 1 = f_2 + f_1 - 1 = 1,$
  the equation is verified.
- Inductive step

  We assume case $n,\ \sum_{k=1}^{n} f_k = f_{n+2} - 1$ and

  prove case $n + 1, \sum_{k=1}^{n+1} f_k = f_{n+3} - 1.$

$$\sum_{k=1}^{n+1} f_k = \sum_{k=1}^{n} f_k + f_{n+1} = (f_{n+2} - 1) + f_{n+1}$$

$$= (f_{n+2} + f_{n+1}) - 1 = f_{n+3} - 1$$