WS 2007/2008

Fundamental Algorithms

Dmytro Chibisov, Jens Ernst

Fakultät für Informatik
TU München

http://www14.in.tum.de/lehre/2007WS/fa-cse/

Fall Semester 2007

Remark: The value of f_n grows very fast as a function of n. It can be shown that

$$f_n \geq 2^{\lfloor \frac{n-1}{2} \rfloor}.$$

For example,

If one arithmetic operation costs $1\mu s$, computing f_{1000} by this algorithm takes $4.3 \cdot 10^{195}$ years.

Remark: The value of f_n grows very fast as a function of n. It can be shown that

$$f_n \geq 2^{\lfloor \frac{n-1}{2} \rfloor}.$$

For example,

- $f_{100} = 3.54 \cdot 10^{20}$
- $f_{1000} = 4.53 \cdot 10^{208}$
- $\cdots$

If one arithmetic operation costs $1\mu s$, computing f_{1000} by this algorithm takes $4.3 \cdot 10^{195}$ years.

Remark: The value of f_n grows very fast as a function of n. It can be shown that

$$f_n \geq 2^{\lfloor \frac{n-1}{2} \rfloor}.$$

For example,

- $f_{100} = 3.54 \cdot 10^{20}$
- $f_{1000} = 4.53 \cdot 10^{208}$
- $\cdots$

If one arithmetic operation costs $1 \mu s$, computing f_{1000} by this algorithm takes $4.3 \cdot 10^{195}$ years.

Remark: The value of f_n grows very fast as a function of n. It can be shown that

$$f_n \geq 2^{\lfloor \frac{n-1}{2} \rfloor}.$$

For example,

- $f_{100} = 3.54 \cdot 10^{20}$
- $f_{1000} = 4.53 \cdot 10^{208}$
- $\cdots$

If one arithmetic operation costs $1\mu s$, computing f_{1000} by this algorithm takes $4.3 \cdot 10^{195}$ years.

Remark: The value of f_n grows very fast as a function of n. It can be shown that

$$f_n \geq 2^{\lfloor \frac{n-1}{2} \rfloor}.$$

For example,

- $f_{100} = 3.54 \cdot 10^{20}$
- $f_{1000} = 4.53 \cdot 10^{208}$
- $\cdots$

If one arithmetic operation costs $1\mu s$, computing f_{1000} by this algorithm takes $4.3 \cdot 10^{195}$ years.

Remark: The value of f_n grows very fast as a function of n. It can be shown that

$$f_n \geq 2^{\lfloor \frac{n-1}{2} \rfloor}.$$

For example,

- $f_{100} = 3.54 \cdot 10^{20}$
- $f_{1000} = 4.53 \cdot 10^{208}$
- $\cdots$

If one arithmetic operation costs $1 \mu s$, computing f_{1000} by this algorithm takes $4.3 \cdot 10^{195}$ years.

Wy is this simple “top-down” algorithm so inefficient?

The reason lies in the repeated identical calls of the function f.

Example 1

$f(6)$ calls $f(5)$ and $f(4)$.

Ideally, identical calls should not be repeated. This suggests that a “bottom-up” approach which memorizes previously computed results might be more efficient.

Wy is this simple “top-down” algorithm so inefficient?

The reason lies in the repeated identical calls of the function f.

Example 1

$f(6)$ calls $f(5)$ and $f(4)$.

Ideally, identical calls should not be repeated. This suggests that a “bottom-up” approach which memorizes previously computed results might be more efficient.

Wy is this simple “top-down” algorithm so inefficient?

The reason lies in the repeated identical calls of the function f.

Example 1

$f(6)$ calls $f(5)$ and $f(4)$.

Ideally, identical calls should not be repeated. This suggests that a “bottom-up” approach which memorizes previously computed results might be more efficient.

Wy is this simple “top-down” algorithm so inefficient?

The reason lies in the repeated identical calls of the function f.

Example 1

$f(6)$ calls $f(5)$ and $f(4)$.

- In $f(5)$, the call $f(2)$ occurs 3 times
- In $f(4)$, the call $f(2)$ occurs twice

Ideally, identical calls should not be repeated. This suggests that a “bottom-up” approach which memorizes previously computed results might be more efficient.

Wy is this simple “top-down” algorithm so inefficient?

The reason lies in the repeated identical calls of the function f.

Example 1

$f(6)$ calls $f(5)$ and $f(4)$.

- In $f(5)$, the call $f(2)$ occurs 3 times
- In $f(4)$, the call $f(2)$ occurs twice

Ideally, identical calls should not be repeated. This suggests that a “bottom-up” approach which memorizes previously computed results might be more efficient.

Wy is this simple “top-down” algorithm so inefficient?

The reason lies in the repeated identical calls of the function f.

Example 1

$f(6)$ calls $f(5)$ and $f(4)$.

- In $f(5)$, the call $f(2)$ occurs 3 times
- In $f(4)$, the call $f(2)$ occurs twice

Ideally, identical calls should not be repeated. This suggests that a “bottom-up” approach which memorizes previously computed results might be more efficient.

Wy is this simple “top-down” algorithm so inefficient?

The reason lies in the repeated identical calls of the function f.

Example 1

$f(6)$ calls $f(5)$ and $f(4)$.

- In $f(5)$, the call $f(2)$ occurs 3 times
- In $f(4)$, the call $f(2)$ occurs twice

Ideally, identical calls should not be repeated. This suggests that a “bottom-up” approach which memorizes previously computed results might be more efficient.

0.1 2nd Algorithm for Computing Fibonacci Numbers

In this algorithm we compute the sequence $f_1, f_2, f_3, \ldots, f_{n-1}, f_n$ in this ascending order.

0.1 2nd Algorithm for Computing Fibonacci Numbers

In this algorithm we compute the sequence $f_1, f_2, f_3, \ldots, f_{n-1}, f_n$ in this ascending order.

- At each point in time, we store those previous results that are still needed to compute the next element of the sequence.
- For step k (where $f_1, f_2, \ldots, f_{k-1}$ are already known), we should have the values f_{k-2} and f_{k-1} in memory. Let us call these values x and y, respectively.
- To compute $z := f_k$, all we need to do is $z := x + y$.
- For the next step, we set $x := y$ and $y := z$ and start over.

0.1 2nd Algorithm for Computing Fibonacci Numbers

In this algorithm we compute the sequence $f_1, f_2, f_3, \ldots, f_{n-1}, f_n$ in this ascending order.

- At each point in time, we store those previous results that are still needed to compute the next element of the sequence.
- For step k (where $f_1, f_2, \ldots, f_{k-1}$ are already known), we should have the values f_{k-2} and f_{k-1} in memory. Let us call these values x and y, respectively.
- To compute $z := f_k$, all we need to do is $z := x + y$.
- For the next step, we set $x := y$ and $y := z$ and start over.

0.1 2nd Algorithm for Computing Fibonacci Numbers

In this algorithm we compute the sequence $f_1, f_2, f_3, \ldots, f_{n-1}, f_n$ in this ascending order.

- At each point in time, we store those previous results that are still needed to compute the next element of the sequence.
- For step k (where $f_1, f_2, \ldots, f_{k-1}$ are already known), we should have the values f_{k-2} and f_{k-1} in memory. Let us call these values x and y, respectively.
- To compute $z := f_k$, all we need to do is $z := x + y$.
- For the next step, we set $x := y$ and $y := z$ and start over.

0.1 2nd Algorithm for Computing Fibonacci Numbers

In this algorithm we compute the sequence $f_1, f_2, f_3, \ldots, f_{n-1}, f_n$ in this ascending order.

- At each point in time, we store those previous results that are still needed to compute the next element of the sequence.
- For step k (where $f_1, f_2, \ldots, f_{k-1}$ are already known), we should have the values f_{k-2} and f_{k-1} in memory. Let us call these values x and y, respectively.
- To compute $z := f_k$, all we need to do is $z := x + y$.
- For the next step, we set $x := y$ and $y := z$ and start over.

Algorithm:

```
unsigned f(unsigned n){
   if (n ≤ 2) then return 1
   else{
      x := 1
      y := 1
      for i := 3 to n do
         z := x + y
         x := y
         y := z
      od
      return z
      }
   fi
}
```

The complexity of this algorithm is easy to see. Let $t_{\text{iter}}(n)$ be the number of arithmetic operations performed as a function of the input value n. Then it holds that

Assuming again that one operation takes $1\mu s$, it now takes 0.001sec. to compute f_{1000}.

Remark: This iterative algorithm uses a very restricted form of dynamic programming. We will see more of this later in this course.

The complexity of this algorithm is easy to see. Let $t_{\text{iter}}(n)$ be the number of arithmetic operations performed as a function of the input value n. Then it holds that

- $t_{\text{iter}}(1) = 0$
- $t_{\text{iter}}(2) = 0$
- $t_{\text{iter}}(n) = n - 2$ for $n \geq 3$.

Assuming again that one operation takes $1\mu s$, it now takes 0.001sec. to compute f_{1000}.

Remark: This iterative algorithm uses a very restricted form of dynamic programming. We will see more of this later in this course.

The complexity of this algorithm is easy to see. Let $t_{\text{iter}}(n)$ be the number of arithmetic operations performed as a function of the input value n. Then it holds that

- $t_{\text{iter}}(1) = 0$
- $t_{\text{iter}}(2) = 0$
- $t_{\text{iter}}(n) = n - 2$ for $n \geq 3$.

Assuming again that one operation takes $1\mu s$, it now takes 0.001sec. to compute f_{1000}.

Remark: This iterative algorithm uses a very restricted form of dynamic programming. We will see more of this later in this course.

The complexity of this algorithm is easy to see. Let $t_{\text{iter}}(n)$ be the number of arithmetic operations performed as a function of the input value n. Then it holds that

- $t_{\text{iter}}(1) = 0$
- $t_{\text{iter}}(2) = 0$
- $t_{\text{iter}}(n) = n - 2$ for $n \geq 3$.

Assuming again that one operation takes $1\mu s$, it now takes 0.001sec. to compute f_{1000}.

Remark: This iterative algorithm uses a very restricted form of dynamic programming. We will see more of this later in this course.

The complexity of this algorithm is easy to see. Let $t_{\text{iter}}(n)$ be the number of arithmetic operations performed as a function of the input value n. Then it holds that

- $t_{\text{iter}}(1) = 0$
- $t_{\text{iter}}(2) = 0$
- $t_{\text{iter}}(n) = n - 2$ for $n \geq 3$.

Assuming again that one operation takes $1\mu s$, it now takes 0.001sec. to compute f_{1000}.

Remark: This iterative algorithm uses a very restricted form of dynamic programming. We will see more of this later in this course.

The complexity of this algorithm is easy to see. Let $t_{\mathsf{iter}}(n)$ be the number of arithmetic operations performed as a function of the input value n. Then it holds that

- $t_{\mathsf{iter}}(1) = 0$
- $t_{\mathsf{iter}}(2) = 0$
- $t_{\mathsf{iter}}(n) = n - 2$ for $n \geq 3$.

Assuming again that one operation takes $1\mu s$, it now takes 0.001sec. to compute f_{1000}.

Remark: This iterative algorithm uses a very restricted form of dynamic programming. We will see more of this later in this course.

The complexity of this algorithm is easy to see. Let $t_{\text{iter}}(n)$ be the number of arithmetic operations performed as a function of the input value n. Then it holds that

- $t_{\text{iter}}(1) = 0$
- $t_{\text{iter}}(2) = 0$
- $t_{\text{iter}}(n) = n - 2$ for $n \geq 3$.

Assuming again that one operation takes $1\mu s$, it now takes 0.001sec. to compute f_{1000}.

Remark: This iterative algorithm uses a very restricted form of dynamic programming. We will see more of this later in this course.

0.2 3rd Algorithm for Computing Fibonacci Numbers

For completeness, let us mention another way of computing f_n for $n \geq 1$. The recurrence relation defining f_n can be solved, and an explicit representation can be obtained. It holds that

$$f_n = \frac{1}{\sqrt{5}} \cdot \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]$$

However, this would require us to work with fractional values at a sufficient level of precision and analyzing the complexity accordingly.

0.2 3rd Algorithm for Computing Fibonacci Numbers

For completeness, let us mention another way of computing f_n for $n \geq 1$. The recurrence relation defining f_n can be solved, and an explicit representation can be obtained. It holds that

$$f_n = \frac{1}{\sqrt{5}} \cdot \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]$$

However, this would require us to work with fractional values at a sufficient level of precision and analyzing the complexity accordingly.

0.2 3rd Algorithm for Computing Fibonacci Numbers

For completeness, let us mention another way of computing f_n for $n \geq 1$. The recurrence relation defining f_n can be solved, and an explicit representation can be obtained. It holds that

$$f_n = \frac{1}{\sqrt{5}} \cdot \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]$$

However, this would require us to work with fractional values at a sufficient level of precision and analyzing the complexity accordingly.

1. Time and Space Complexity

The resource usage of an algorithm is measured as a function of its input size (or, as in the previous example, of one of its input values).

Definition 2

Let $x := (x_1, x_2, \ldots, x_m)$ be some input. The uniform input size $||x||_u$ is defined as

$$||x||_u := m.$$

Definition 3

The uniform time complexity $t^u(x)$ of an algorithm for input x is the number of operations performed by the algorithm upon input x.

More realistically, the actual size of input x in bits, rather than its length as a vector, should be taken into account.

1. Time and Space Complexity

The resource usage of an algorithm is measured as a function of its input size (or, as in the previous example, of one of its input values).

Definition 2

Let $x := (x_1, x_2, \ldots, x_m)$ be some input. The uniform input size $||x||_u$ is defined as

$$||x||_u := m.$$

Definition 3

The uniform time complexity $t^u(x)$ of an algorithm for input x is the number of operations performed by the algorithm upon input x.

More realistically, the actual size of input x in bits, rather than its length as a vector, should be taken into account.

1. Time and Space Complexity

The resource usage of an algorithm is measured as a function of its input size (or, as in the previous example, of one of its input values).

Definition 2

Let $x := (x_1, x_2, \ldots, x_m)$ be some input. The uniform input size $||x||_u$ is defined as

$$||x||_u := m.$$

Definition 3

The uniform time complexity $t^u(x)$ of an algorithm for input x is the number of operations performed by the algorithm upon input x.

More realistically, the actual size of input x in bits, rather than its length as a vector, should be taken into account.

Lemma 2

For some value $x \in \mathbf{N}_0$, the length $\ell(x)$ of the representation of x as a binary number is

$$\ell(x) = \lfloor \log x \rfloor + 1.$$

This immediately follows from the fact that $2^{\ell(x)-1} \le x < 2^{\ell(x)}$.

Definition 4

The logarithmic size $||x||_{\log}$ of input $x = (x_1, x_2, \dots, x_m)$ is

$$||x||_{\log} = \sum_{i=1}^{m} \ell(x_i) = m + \sum_{i=1}^{m} \lfloor \log x_i \rfloor.$$

Lemma 2

For some value $x \in \mathbf{N}_0$, the length $\ell(x)$ of the representation of x as a binary number is

$$\ell(x) = \lfloor \log x \rfloor + 1.$$

This immediately follows from the fact that $2^{\ell(x)-1} \leq x < 2^{\ell(x)}$.

Definition 4

The logarithmic size $||x||_{\log}$ of input $x = (x_1, x_2, \ldots, x_m)$ is

$$||x||_{\log} = \sum_{i=1}^{m} \ell(x_i) = m + \sum_{i=1}^{m} \lfloor \log x_i \rfloor.$$

Definition 5

The logarithmic time complexity $t^{\log}(x)$ of an algorithm for input x is the total of the logarithmic costs of all operations carried out by the algorithm, given input x. The logarithmic cost of an operation is the total size of all arguments of this operation (in binary representation). See example.

Example 6

The logarithmic cost of the operation "$a := a + c[d[i]]$" is

$$\ell(a) + \ell(i) + \ell(c[i]) + \ell(c[d[i]]).$$

Definition 5

The logarithmic time complexity $t^{\log}(x)$ of an algorithm for input x is the total of the logarithmic costs of all operations carried out by the algorithm, given input x. The logarithmic cost of an operation is the total size of all arguments of this operation (in binary representation). See example.

Example 6

The logarithmic cost of the operation "$a := a + c[d[i]]$" is

$$\ell(a) + \ell(i) + \ell(c[i]) + \ell(c[d[i]]).$$

Remark: Statements on the resource usage of a given algorithm upon specific inputs x are hardly useful. Rather, we need some way to argue over *all* inputs of a given length. This calls for a worst-case consideration.

Definition 7

Let $t : \mathbf{N} \longrightarrow \mathbf{N}$ be some function. An algorithm is said to have uniform (logarithmic) time complexity $t(n)$ if and only if

$$\begin{array}{lcll} t^u(n) & := & \max\{t^u(x) : ||x||_u = n\} & \leq \quad t(n) \\ (t^{\log}(n) & := & \max\{t^{\log}(x) : ||x||_{\log} = n\} & \leq \quad t(n)), \end{array}$$

respectively.

This means that worst-case inputs of size n are considered when bounding the running time from above by function $t(n)$.

Remark: Statements on the resource usage of a given algorithm upon specific inputs x are hardly useful. Rather, we need some way to argue over *all* inputs of a given length. This calls for a worst-case consideration.

Definition 7

Let $t : \mathbf{N} \longrightarrow \mathbf{N}$ be some function. An algorithm is said to have uniform (logarithmic) time complexity $t(n)$ if and only if

$$\begin{array}{lcll} t^u(n) & := & \max\{t^u(x) : ||x||_u = n\} & \leq & t(n) \\ (t^{\log}(n) & := & \max\{t^{\log}(x) : ||x||_{\log} = n\} & \leq & t(n)), \end{array}$$

respectively.

This means that worst-case inputs of size n are considered when bounding the running time from above by function $t(n)$.

Yet more realism can be achieved by considering average case time complexity. This requires that, for each value of n, a probability distribution over all possible inputs x of length n be known. Using this information, the uniform or logarithmic time complexities $t^u(x)$ (or $t^{\log}(x)$) can be considered.

In this lecture we will not cover average case analysis.

Definition 8

The uniform space complexity $s^u(x)$ of an algorithm for input x is the number of storage locations used by the algorithm, given x.

Definition 9

The logarithmic space complexity $s^{\log}(x)$ of an algorithm for inptu x is the sum of the maximum lengths (in binary form) of the values written to the storage registers by the algorithm upon input x.

Yet more realism can be achieved by considering average case time complexity. This requires that, for each value of n, a probability distribution over all possible inputs x of length n be known. Using this information, the uniform or logarithmic time complexities $t^u(x)$ (or $t^{\log}(x)$) can be considered.

In this lecture we will not cover average case analysis.

Definition 8

The uniform space complexity $s^u(x)$ of an algorithm for input x is the number of storage locations used by the algorithm, given x.

Definition 9

The logarithmic space complexity $s^{\log}(x)$ of an algorithm for inptu x is the sum of the maximum lengths (in binary form) of the values written to the storage registers by the algorithm upon input x.

Yet more realism can be achieved by considering average case time complexity. This requires that, for each value of n, a probability distribution over all possible inputs x of length n be known. Using this information, the uniform or logarithmic time complexities $t^u(x)$ (or $t^{\log}(x)$) can be considered.

In this lecture we will not cover average case analysis.

Definition 8

The uniform space complexity $s^u(x)$ of an algorithm for input x is the number of storage locations used by the algorithm, given x.

Definition 9

The logarithmic space complexity $s^{\log}(x)$ of an algorithm for inptu x is the sum of the maximum lengths (in binary form) of the values written to the storage registers by the algorithm upon input x.

Definition 10

Let $s : \mathbf{N} \longrightarrow \mathbf{N}$ be some function. An algorithm is said to have uniform (logarithmic) space complexity if and only if

$$\begin{array}{lcll} s^u(n) & := & \max\{s^u(x) : ||x||_u = n\} & \leq & s(n) \\ (s^{\log}(n) & := & \max\{s^{\log}(x) : ||x||_{\log} = n\} & \leq & s(n)), \end{array}$$

respectively.

Like in the case of time complexity, average case analysis can be applied to space complexity. This will not be covered in this course.

Definition 10

Let $s : \mathbf{N} \longrightarrow \mathbf{N}$ be some function. An algorithm is said to have uniform (logarithmic) space complexity if and only if

$$\begin{array}{lcll} s^u(n) & := & \max\{s^u(x) : ||x||_u = n\} & \leq & s(n) \\ (s^{\log}(n) & := & \max\{s^{\log}(x) : ||x||_{\log} = n\} & \leq & s(n)), \end{array}$$

respectively.

Like in the case of time complexity, average case analysis can be applied to space complexity. This will not be covered in this course.