Foundations

Textbook Reading

Chapters 2 & 3

Overview

Review of things you should know

- Proof by contradiction
- Arrays, linked lists, stacks, and queues

Analysis of algorithms

- Worst-case and average-case running time
- Asymptotic notation

Stable Matching: The Gale-Shapley Algorithm

StableMatching(M, W)

```
while there exists an unmarried man m
    do m proposes to the most preferable woman w he has not proposed to yet
        if w is unmarried or likes m better than her current partner m′
            then if w is married
                    then w divorces m′
                 w marries m
```

Questions we can and should ask about the algorithm:

- Is there always a stable matching?
- Does the algorithm always terminate?
- Does the algorithm always produce a stable matching?
- How efficient is the algorithm? Can we bound its running time?

Stable Matching: The Gale-Shapley Algorithm

StableMatching(M, W)

```
while there exists an unmarried man m
    do m proposes to the most preferable woman w he has not proposed to yet
        if w is unmarried or likes m better than her current partner m′
            then if w is married
                     then w divorces m′
                 w marries m
```

Questions we can and should ask about the algorithm:

- Is there always a stable matching?
- Does the algorithm always terminate?
- Does the algorithm always produce a stable matching?
- How efficient is the algorithm? Can we bound its running time?

Stable Matching: The Gale-Shapley Algorithm

StableMatching(M, W)

```
while there exists an unmarried man m who has not proposed to all women yet
    do m proposes to the most preferable woman w he has not proposed to yet
        if w is unmarried or likes m better than her current partner m′
            then if w is married
                    then w divorces m′
                 w marries m
```

Questions we can and should ask about the algorithm:

- Is there always a stable matching?
- Does the algorithm always terminate?
- Does the algorithm always produce a stable matching?
- How efficient is the algorithm? Can we bound its running time?

Termination of the Gale-Shapley Algorithm

StableMatching(M, W)

```
while there exists an unmarried man m who has not proposed to all women yet
   do m proposes to the most preferable woman w he has not proposed to yet
      if w is unmarried or likes m better than her current partner m′
         then if w is married
                 then w divorces m′
              w marries m
```

Lemma: The Gale-Shapley Algorithm terminates after at most n^2 iterations.

Termination of the Gale-Shapley Algorithm

StableMatching(M, W)

```
while there exists an unmarried man m who has not proposed to all women yet
   do m proposes to the most preferable woman w he has not proposed to yet
        if w is unmarried or likes m better than her current partner m′
           then if w is married
                   then w divorces m′
                w marries m
```

Lemma: The Gale-Shapley Algorithm terminates after at most n^2 iterations.

There are n men.

Termination of the Gale-Shapley Algorithm

StableMatching(M, W)

```
while there exists an unmarried man m who has not proposed to all women yet
   do m proposes to the most preferable woman w he has not proposed to yet
        if w is unmarried or likes m better than her current partner m′
           then if w is married
                    then w divorces m′
                w marries m
```

Lemma: The Gale-Shapley Algorithm terminates after at most n^2 iterations.

There are n men.
Every man can propose to n women.

Termination of the Gale-Shapley Algorithm

StableMatching(M, W)

```
while there exists an unmarried man m who has not proposed to all women yet
    do m proposes to the most preferable woman w he has not proposed to yet
        if w is unmarried or likes m better than her current partner m′
            then if w is married
                    then w divorces m′
                 w marries m
```

Lemma: The Gale-Shapley Algorithm terminates after at most n^2 iterations.

There are n men.
Every man can propose to n women.
No man proposes to the same woman twice.

Termination of the Gale-Shapley Algorithm

StableMatching(M, W)

```
while there exists an unmarried man m who has not proposed to all women yet
   do m proposes to the most preferable woman w he has not proposed to yet
        if w is unmarried or likes m better than her current partner m′
           then if w is married
                   then w divorces m′
                w marries m
```

Lemma: The Gale-Shapley Algorithm terminates after at most n^2 iterations.

There are n men.
Every man can propose to n women.
No man proposes to the same woman twice.
$\Rightarrow$ There are n^2 proposals to be made.

Termination of the Gale-Shapley Algorithm

StableMatching(M, W)

```
while there exists an unmarried man m who has not proposed to all women yet
   do m proposes to the most preferable woman w he has not proposed to yet
        if w is unmarried or likes m better than her current partner m′
           then if w is married
                   then w divorces m′
                w marries m
```

Lemma: The Gale-Shapley Algorithm terminates after at most n^2 iterations.

There are n men.
Every man can propose to n women.
No man proposes to the same woman twice.
$\Rightarrow$ There are n^2 proposals to be made.

In every iteration, one proposal is made.

Termination of the Gale-Shapley Algorithm

StableMatching(M, W)

```
while there exists an unmarried man m who has not proposed to all women yet
   do m proposes to the most preferable woman w he has not proposed to yet
        if w is unmarried or likes m better than her current partner m′
           then if w is married
                      then w divorces m′
                   w marries m
```

Lemma: The Gale-Shapley Algorithm terminates after at most n^2 iterations.

There are n men.
Every man can propose to n women.
No man proposes to the same woman twice.
$\Rightarrow$ There are n^2 proposals to be made.

In every iteration, one proposal is made.
$\Rightarrow$ There are at most n^2 iterations.

Everybody Gets Married: Proof by Contradiction

Lemma: At the end of the Gale-Shapley Algorithm every woman (and hence every man) is married.

Everybody Gets Married: Proof by Contradiction

Lemma: At the end of the Gale-Shapley Algorithm every woman (and hence every man) is married.

Proof by contradiction:

- Assume that what we want to prove is incorrect.
- Prove that this leads to a contradiction.

Everybody Gets Married: Proof by Contradiction

Lemma: At the end of the Gale-Shapley Algorithm every woman (and hence every man) is married.

Proof by contradiction:

- Assume that what we want to prove is incorrect.
- Prove that this leads to a contradiction.

Assumption: There is an unmarried woman at the end of the algorithm.

Everybody Gets Married: Proof by Contradiction

Lemma: At the end of the Gale-Shapley Algorithm every woman (and hence every man) is married.

Proof by contradiction:

- Assume that what we want to prove is incorrect.
- Prove that this leads to a contradiction.

Assumption: There is an unmarried woman at the end of the algorithm.

A woman, once married, stays married (not necessarily to the same man).

Everybody Gets Married: Proof by Contradiction

Lemma: At the end of the Gale-Shapley Algorithm every woman (and hence every man) is married.

Proof by contradiction:

- Assume that what we want to prove is incorrect.
- Prove that this leads to a contradiction.

Assumption: There is an unmarried woman at the end of the algorithm.

A woman, once married, stays married (not necessarily to the same man).

If there is an unmarried woman w, there must be an unmarried man m.

Everybody Gets Married: Proof by Contradiction

Lemma: At the end of the Gale-Shapley Algorithm every woman (and hence every man) is married.

Proof by contradiction:

- Assume that what we want to prove is incorrect.
- Prove that this leads to a contradiction.

Assumption: There is an unmarried woman at the end of the algorithm.

A woman, once married, stays married (not necessarily to the same man).
If there is an unmarried woman w, there must be an unmarried man m.
When the algorithm terminates, m must have proposed to all women, including w.

Everybody Gets Married: Proof by Contradiction

Lemma: At the end of the Gale-Shapley Algorithm every woman (and hence every man) is married.

Proof by contradiction:

- Assume that what we want to prove is incorrect.
- Prove that this leads to a contradiction.

Assumption: There is an unmarried woman at the end of the algorithm.

A woman, once married, stays married (not necessarily to the same man).
If there is an unmarried woman w, there must be an unmarried man m.
When the algorithm terminates, m must have proposed to all women, including w.
w would have married m then.

Contradiction.

Stable Matching: The Gale-Shapley Algorithm

StableMatching(M, W)

```
while there exists an unmarried man m who has not proposed to all women yet
    do m proposes to the most preferable woman w he has not proposed to yet
        if w is unmarried or likes m better than her current partner m′
            then if w is married
                    then w divorces m′
                 w marries m
```

Questions we can and should ask about the algorithm:

- Is there always a stable matching?
- Does the algorithm always terminate?
- Does the algorithm always produce a stable matching?
- How efficient is the algorithm? Can we bound its running time?

Stable Matching: The Gale-Shapley Algorithm

StableMatching(M, W)

```
while there exists an unmarried man m
   do m proposes to the most preferable woman w he has not proposed to yet
       if w is unmarried or likes m better than her current partner m′
          then if w is married
                     then w divorces m′
                  w marries m
```

Questions we can and should ask about the algorithm:

- Is there always a stable matching?
- Does the algorithm always terminate?
- Does the algorithm always produce a stable matching?
- How efficient is the algorithm? Can we bound its running time?

Stable Matching: The Gale-Shapley Algorithm

StableMatching(M, W)

```
while there exists an unmarried man m
    do m proposes to the most preferable woman w he has not proposed to yet
        if w is unmarried or likes m better than her current partner m′
            then if w is married
                    then w divorces m′
                 w marries m
```

Questions we can and should ask about the algorithm:

- Is there always a stable matching?
- Does the algorithm always terminate?
- Does the algorithm always produce a stable matching?
- How efficient is the algorithm? Can we bound its running time?

Correctness of the Gale-Shapley Algorithm

Lemma: The matching produced by the Gale-Shapley Algorithm is stable.

Correctness of the Gale-Shapley Algorithm

Lemma: The matching produced by the Gale-Shapley Algorithm is stable.

Proof by contradiction:

Assume there exist two marriages (m, w) and (m', w') such that $m' \prec_w m$ and $w \prec_{m'} w'$.

Correctness of the Gale-Shapley Algorithm

Lemma: The matching produced by the Gale-Shapley Algorithm is stable.

Proof by contradiction:

Assume there exist two marriages (m, w) and (m', w') such that $m' \prec_w m$ and $w \prec_{m'} w'$.

Since $w \prec_{m'} w'$, m' must have proposed to w before getting married to w'.

Correctness of the Gale-Shapley Algorithm

Lemma: The matching produced by the Gale-Shapley Algorithm is stable.

Proof by contradiction:

Assume there exist two marriages (m, w) and (m', w') such that $m' \prec_w m$ and $w \prec_{m'} w'$.

Since $w \prec_{m'} w'$, m' must have proposed to w before getting married to w'.
Let m'' be w's partner immediately after m' proposed to her.

- If w accepts m', then $m'' = m'$.
- If w rejects m', then $m'' \prec_w m'$.

Correctness of the Gale-Shapley Algorithm

Lemma: The matching produced by the Gale-Shapley Algorithm is stable.

Proof by contradiction:

Assume there exist two marriages (m, w) and (m', w') such that $m' \prec_w m$ and $w \prec_{m'} w'$.

Since $w \prec_{m'} w'$, m' must have proposed to w before getting married to w'. Let m'' be w's partner immediately after m' proposed to her.

- If w accepts m', then $m'' = m'$.
- If w rejects m', then $m'' \prec_w m'$.

$\Rightarrow$ $m'' \preceq_w m'$

Correctness of the Gale-Shapley Algorithm

Lemma: The matching produced by the Gale-Shapley Algorithm is stable.

Proof by contradiction:

Assume there exist two marriages (m, w) and (m', w') such that $m' \prec_w m$ and $w \prec_{m'} w'$.

Since $w \prec_{m'} w'$, m' must have proposed to w before getting married to w'.
Let m'' be w's partner immediately after m' proposed to her.

- If w accepts m', then $m'' = m'$.
- If w rejects m', then $m'' \prec_w m'$.

$\Rightarrow$ $m'' \preceq_w m'$

Let $m'' = m_1, m_2, \ldots, m_k = m$ be the sequence of partners w has from this time on.

Correctness of the Gale-Shapley Algorithm

Lemma: The matching produced by the Gale-Shapley Algorithm is stable.

Proof by contradiction:

Assume there exist two marriages (m, w) and (m', w') such that $m' \prec_w m$ and $w \prec_{m'} w'$.

Since $w \prec_{m'} w'$, m' must have proposed to w before getting married to w'.
Let m'' be w's partner immediately after m' proposed to her.

- If w accepts m', then $m'' = m'$.
- If w rejects m', then $m'' \prec_w m'$.

$\Rightarrow$ $m'' \preceq_w m'$

Let $m'' = m_1, m_2, \ldots, m_k = m$ be the sequence of partners w has from this time on.
If $k = 1$, then $m = m'' \preceq_w m'$.

Correctness of the Gale-Shapley Algorithm

Lemma: The matching produced by the Gale-Shapley Algorithm is stable.

Proof by contradiction:

Assume there exist two marriages (m, w) and (m', w') such that $m' \prec_w m$ and $w \prec_{m'} w'$.

Since $w \prec_{m'} w'$, m' must have proposed to w before getting married to w'.
Let m'' be w's partner immediately after m' proposed to her.

- If w accepts m', then $m'' = m'$.
- If w rejects m', then $m'' \prec_w m'$.

$\Rightarrow$ $m'' \preceq_w m'$

Let $m'' = m_1, m_2, \ldots, m_k = m$ be the sequence of partners w has from this time on.
If $k = 1$, then $m = m'' \preceq_w m'$.
If $k > 1$, then $m = m_k \prec_w m_{k-1} \prec_w \cdots \prec_w m_2 \prec_w m_1 = m'' \preceq_w m'$.

Correctness of the Gale-Shapley Algorithm

Lemma: The matching produced by the Gale-Shapley Algorithm is stable.

Proof by contradiction:

Assume there exist two marriages (m, w) and (m', w') such that $m' \prec_w m$ and $w \prec_{m'} w'$.

Since $w \prec_{m'} w'$, m' must have proposed to w before getting married to w'.
Let m'' be w's partner immediately after m' proposed to her.

- If w accepts m', then $m'' = m'$.
- If w rejects m', then $m'' \prec_w m'$.

$\Rightarrow \; m'' \preceq_w m'$

Let $m'' = m_1, m_2, \ldots, m_k = m$ be the sequence of partners w has from this time on.
If $k = 1$, then $m = m'' \preceq_w m'$.
If $k > 1$, then $m = m_k \prec_w m_{k-1} \prec_w \cdots \prec_w m_2 \prec_w m_1 = m'' \preceq_w m'$.
$\Rightarrow \; m \preceq_w m'$

Contradiction.

More Questions

Does the final matching depend on the order in which the men propose?

Is the process fair?

Can the algorithm be implemented efficiently?

Can we implement a faster algorithm?

More Questions

Does the final matching depend on the order in which the men propose?
No!

Is the process fair?

Can the algorithm be implemented efficiently?

Can we implement a faster algorithm?

More Questions

Does the final matching depend on the order in which the men propose?
No!

Is the process fair?
No! The men fare much better than the women.

Can the algorithm be implemented efficiently?

Can we implement a faster algorithm?

More Questions

Does the final matching depend on the order in which the men propose?
No!

Is the process fair?
No! The men fare much better than the women.

Can the algorithm be implemented efficiently?

Can we implement a faster algorithm?
Yes, using randomization.

Computational Tractability

Informally, we consider a problem computationally tractable if it can be solved using reasonable resources.

Resources:

- Running time
- Memory usage
- Disk usage
- Number of messages sent across the network
- Energy
- ...

Model of Computation: The RAM Model

We would like to be able to predict the running time of algorithms before implementing them.

We would like our analysis to be applicable to a wide range of machines.

$\Rightarrow$ We need to base our analysis on a model of computation that captures the characteristics of a wide range of machines.

Model of Computation: The RAM Model

We would like to be able to predict the running time of algorithms before implementing them.

We would like our analysis to be applicable to a wide range of machines.

$\Rightarrow$ We need to base our analysis on a model of computation that captures the characteristics of a wide range of machines.

The Random Access Machine (RAM) model:

Elementary operations take constant time:

- Arithmetic operations: addition, subtraction, multiplication, division
- Boolean operations: and, or, not
- If-statements
- Checking of loop conditions
- Memory access

Model of Computation: The RAM Model

We would like to be able to predict the running time of algorithms before implementing them.

We would like our analysis to be applicable to a wide range of machines.

$\Rightarrow$ We need to base our analysis on a model of computation that captures the characteristics of a wide range of machines.

The Random Access Machine (RAM) model:

Elementary operations take constant time:

- Arithmetic operations: addition, subtraction, multiplication, division
- Boolean operations: and, or, not
- If-statements
- Checking of loop conditions
- Memory access

$\Rightarrow$ By counting elementary operations, we can compare the actual running times of two algorithms up to constant factors.

Efficient Algorithm = Polynomial Running Time

Most algorithms are fast for small inputs. We care about their behaviour for non-trivial (i.e., large) inputs.

$\Rightarrow$ We would like to express the running time as a function of the input size n.

Efficient Algorithm = Polynomial Running Time

Most algorithms are fast for small inputs. We care about their behaviour for non-trivial (i.e., large) inputs.

$\Rightarrow$ We would like to express the running time as a function of the input size n.

Definition: We consider an algorithm efficient if its running time is polynomial in the input size.

Efficient Algorithm = Polynomial Running Time

Most algorithms are fast for small inputs. We care about their behaviour for non-trivial (i.e., large) inputs.

$\Rightarrow$ We would like to express the running time as a function of the input size n.

Definition: We consider an algorithm efficient if its running time is polynomial in the input size.

Motivation:

If the input size doubles, the running time should increase by only a constant factor.

Efficient Algorithm = Polynomial Running Time

Most algorithms are fast for small inputs. We care about their behaviour for non-trivial (i.e., large) inputs.

$\Rightarrow$ We would like to express the running time as a function of the input size n.

Definition: We consider an algorithm efficient if its running time is polynomial in the input size.

Motivation:

If the input size doubles, the running time should increase by only a constant factor.

Questions:

Is n^{100} efficient?

Efficient Algorithm = Polynomial Running Time

Most algorithms are fast for small inputs. We care about their behaviour for non-trivial (i.e., large) inputs.

$\Rightarrow$ We would like to express the running time as a function of the input size n.

Definition: We consider an algorithm efficient if its running time is polynomial in the input size.

Motivation:

If the input size doubles, the running time should increase by only a constant factor.

Questions:

Is n^{100} efficient? No.

Efficient Algorithm = Polynomial Running Time

Most algorithms are fast for small inputs. We care about their behaviour for non-trivial (i.e., large) inputs.

$\Rightarrow$ We would like to express the running time as a function of the input size n.

Definition: We consider an algorithm efficient if its running time is polynomial in the input size.

Motivation:

If the input size doubles, the running time should increase by only a constant factor.

Questions:

Is n^{100} efficient? No.

Is $n^{1+0.02 \lg n}$ inefficient?

Efficient Algorithm = Polynomial Running Time

Most algorithms are fast for small inputs. We care about their behaviour for non-trivial (i.e., large) inputs.

$\Rightarrow$ We would like to express the running time as a function of the input size n.

Definition: We consider an algorithm efficient if its running time is polynomial in the input size.

Motivation:

If the input size doubles, the running time should increase by only a constant factor.

Questions:

Is n^{100} efficient? No.

Is $n^{1+0.02 \lg n}$ inefficient? No.

Efficient Algorithm = Polynomial Running Time

Most algorithms are fast for small inputs. We care about their behaviour for non-trivial (i.e., large) inputs.

$\Rightarrow$ We would like to express the running time as a function of the input size n.

Definition: We consider an algorithm efficient if its running time is polynomial in the input size.

Motivation:

If the input size doubles, the running time should increase by only a constant factor.

Questions:

Is n^{100} efficient? No.

Is $n^{1+0.02 \lg n}$ inefficient? No.

Justification: Overwhelmingly, polynomial-time algorithms are fast in practice and exponential-time algorithms are not.

Running Time May Depend on Specific Input

InsertionSort(A, n)

```
for i = 2 to n
    do x = A[i]
       j = i − 1
       while j > 0 and A[j] > x
           do A[j + 1] = A[j]
       A[j + 1] = x
```

Running Time May Depend on Specific Input

InsertionSort(A, n)

```
for i = 2 to n
    do x = A[i]
       j = i − 1
       while j > 0 and A[j] > x
           do A[j + 1] = A[j]
       A[j + 1] = x
```

Running time: Linear for sorted inputs, quadratic for inputs sorted in reverse order (and in fact for most inputs).

Running Time May Depend on Specific Input

InsertionSort(A, n)

```
for i = 2 to n
    do x = A[i]
       j = i − 1
       while j > 0 and A[j] > x
           do A[j + 1] = A[j]
       A[j + 1] = x
```

Running time: Linear for sorted inputs, quadratic for inputs sorted in reverse order (and in fact for most inputs).

How do we unify this into one function $T(n)$?

Worst-Case and Average-Case Running Time

The worst-case running time of an algorithm A is a function T(n) defined as the maximum running time of A over all possible inputs of size n.

The average-case running time of an algorithm A is a function T(n) defined as the average running time of A over all possible inputs of size n.

Asymptotic Running Time

Scenario: Given two algorithms A and B which we want to compare.

Do we care which one is faster for small inputs?

Asymptotic Running Time

Scenario: Given two algorithms A and B which we want to compare.

Do we care which one is faster for small inputs?

Not really. We care most about which one is faster for large inputs, where efficiency really matters.

Asymptotic Running Time

Scenario: Given two algorithms A and B which we want to compare.

Do we care which one is faster for small inputs?

Not really. We care most about which one is faster for large inputs, where efficiency really matters.

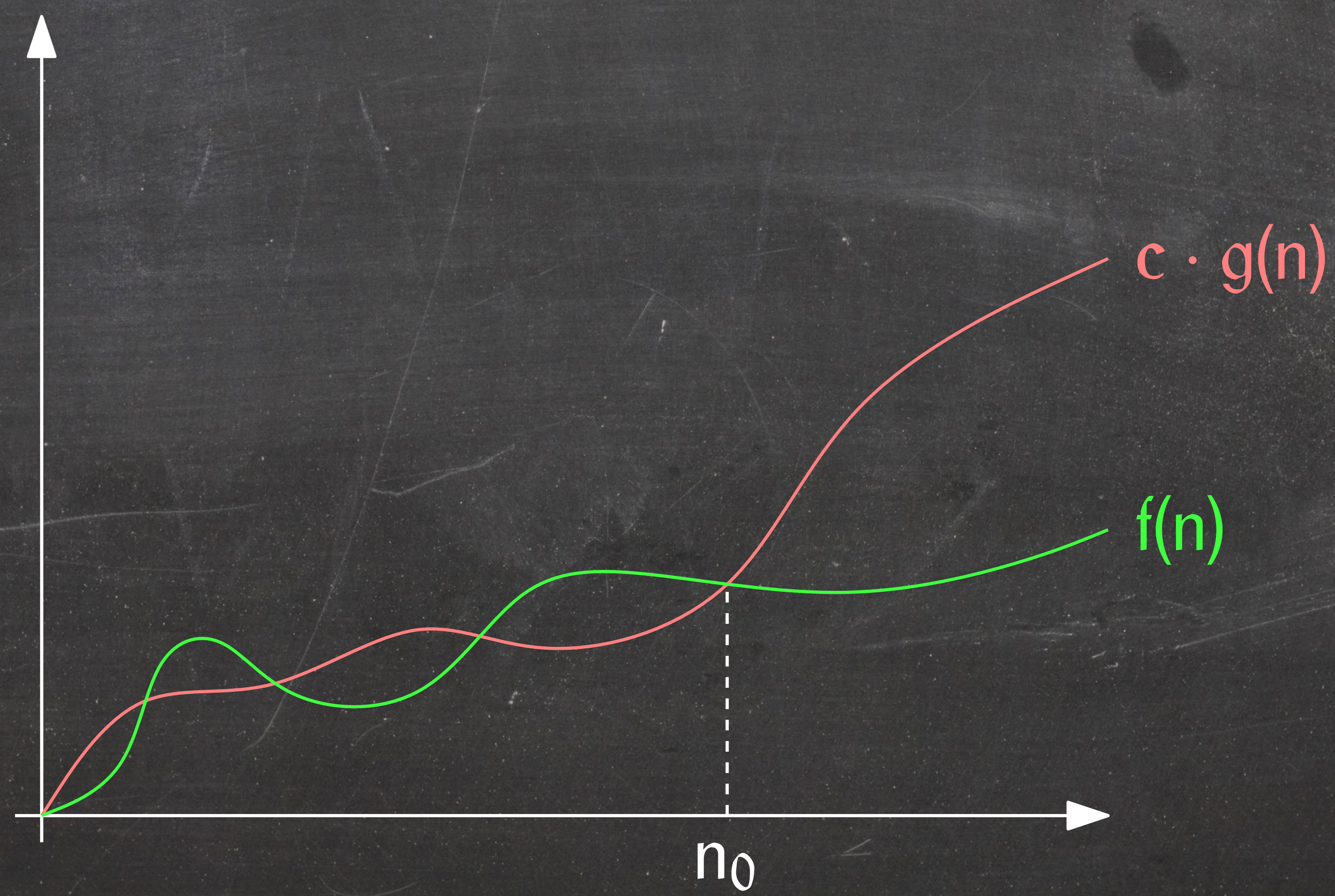
Formally: We want $T_A(n) < T_B(n)$ for all $n \geq n_0$, where n_0 is the smallest input size we consider to be "large".

O-Notation

$f(n) \in O(g(n))$ means that $f(n)$ is at most a constant factor larger than $g(n)$ for large enough n.

Formally:

$$f(n) \in O(g(n))$$
$$\Updownarrow$$
$$\exists c > 0, n_0 \geq 0 \ \forall n \geq n_0 : f(n) \leq c \cdot g(n)$$

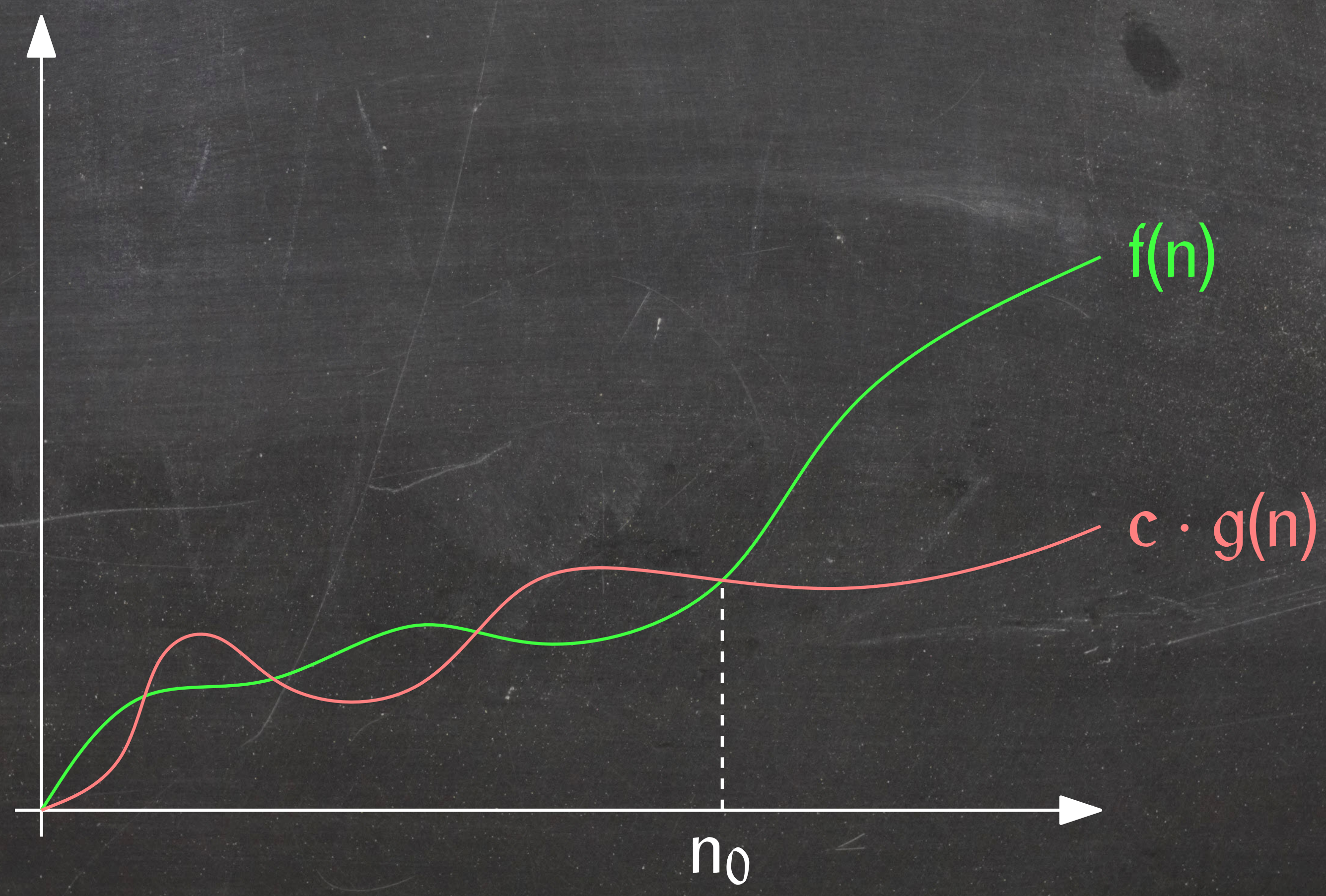


Ω-Notation

$f(n) \in \Omega(g(n))$ means that $f(n)$ is at most a constant factor smaller than $g(n)$ for large enough n.

Formally:

$$f(n) \in \Omega(g(n))$$
$$\Updownarrow$$
$$\exists c > 0, n_0 \geq 0 \; \forall n \geq n_0 : f(n) \geq c \cdot g(n)$$

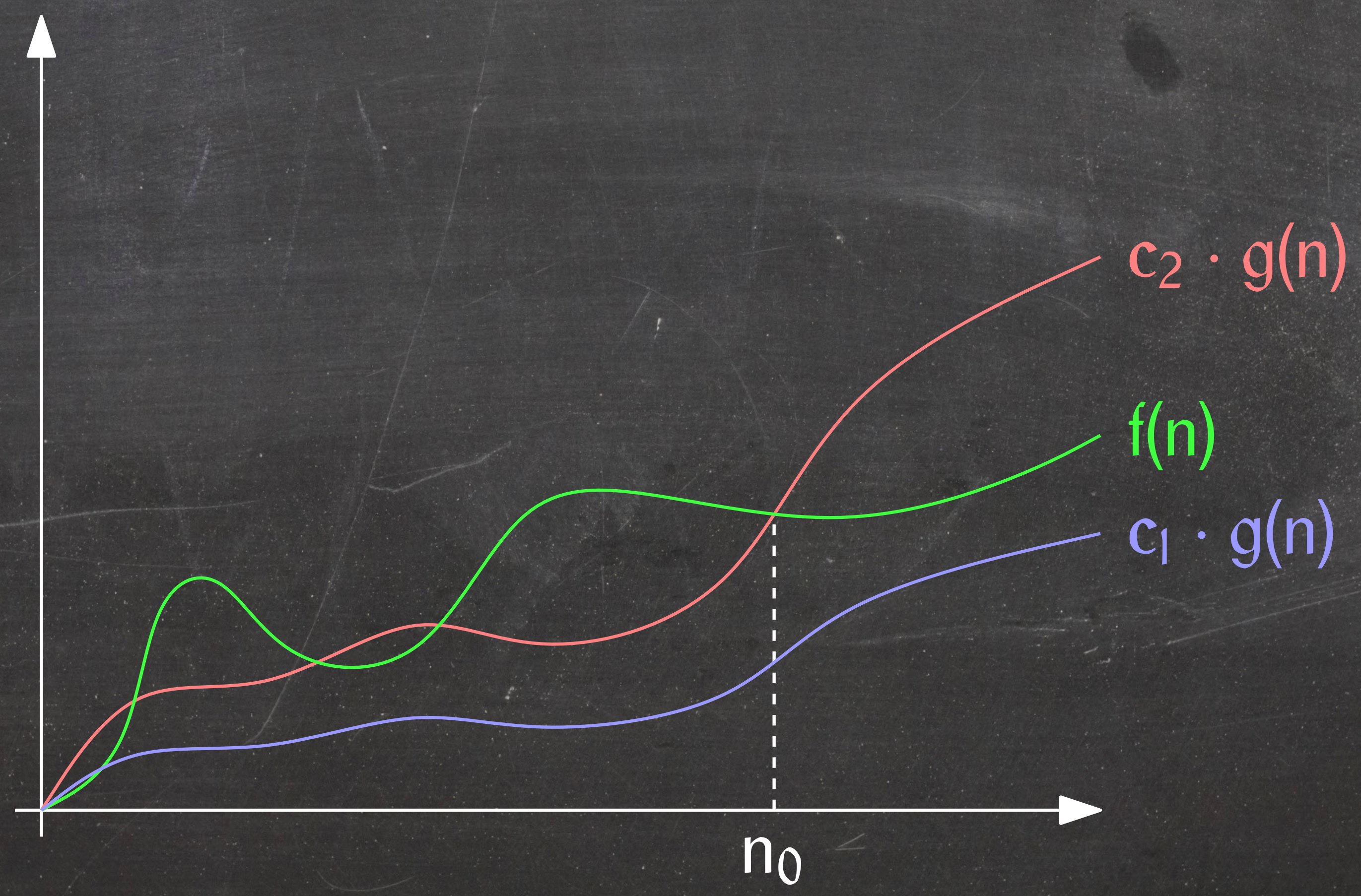


Θ-Notation

$f(n) \in \Theta(g(n))$ means that the difference between $f(n)$ and $g(n)$ is at most a constant factor for large enough n.

Formally:

$$f(n) \in \Theta(g(n))$$
$$\Updownarrow$$
$$\exists c_1 > 0, c_2 > 0, n_0 \geq 0 \; \forall n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

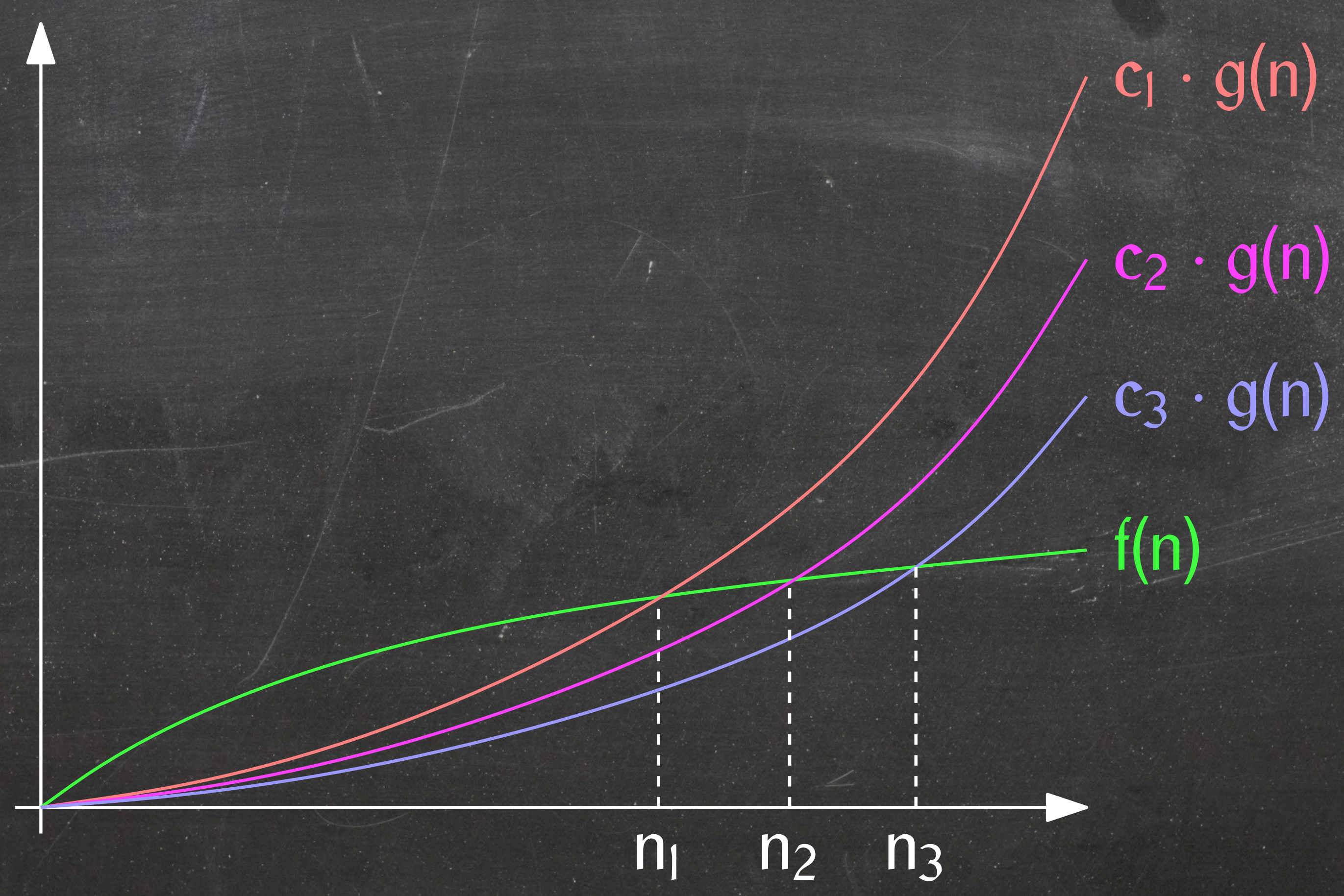


o-Notation

$f(n) \in o(g(n))$ means that the ratio between $g(n)$ and $f(n)$ grows without bounds as n grows. An algorithm with running time $f(n)$ is much faster than one with running time $g(n)$ for large enough inputs, even if run on a slower computer!

Formally:

$$f(n) \in o(g(n))$$
$$\Updownarrow$$
$$\forall c > 0 \; \exists n_0 \geq 0 \; \forall n \geq n_0 : f(n) \leq c \cdot g(n)$$

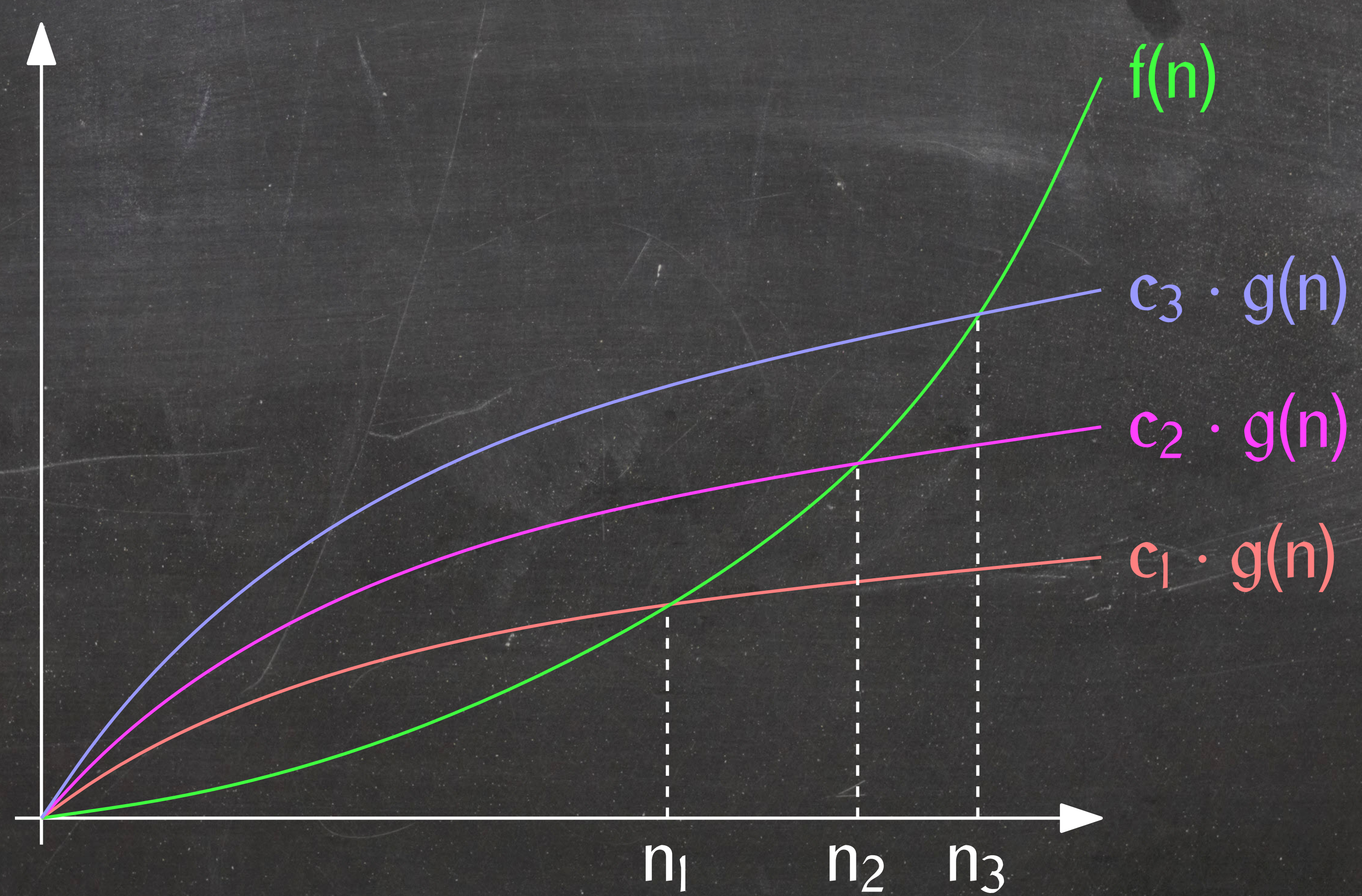


ω-Notation

$f(n) \in \omega(g(n))$ means that the ratio between $f(n)$ and $g(n)$ grows without bounds as n grows. An algorithm with running time $g(n)$ is much faster than one with running time $f(n)$ for large enough inputs, even if run on a slower computer!

Formally:

$$f(n) \in \omega(g(n))$$
$$\Updownarrow$$
$$\forall c > 0\ \exists n_0 \geq 0\ \forall n \geq n_0 : f(n) \geq c \cdot g(n)$$



A Few Simple Facts

$$f(n) \in O(f(n)) \qquad f(n) \in \Omega(f(n)) \qquad f(n) \in \Theta(f(n))$$

$$f(n) \in O(g(n)) \text{ and } g(n) \in O(h(n)) \Longrightarrow f(n) \in O(h(n))$$
$$f(n) \in \Omega(g(n)) \text{ and } g(n) \in \Omega(h(n)) \Longrightarrow f(n) \in \Omega(h(n))$$
$$f(n) \in \Theta(g(n)) \text{ and } g(n) \in \Theta(h(n)) \Longrightarrow f(n) \in \Theta(h(n))$$

$$f(n) \in O(g(n)) \Longleftrightarrow g(n) \in \Omega(f(n))$$
$$f(n) \in o(g(n)) \Longleftrightarrow g(n) \in \omega(f(n))$$

$$f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n)) \Longleftrightarrow f(n) \in \Theta(g(n))$$

$$f_1(n) \in O(g_1(n)) \text{ and } f_2(n) \in O(g_2(n)) \Longrightarrow f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$$

$$f(n) \in O(g(n)) \Longrightarrow f(n) + g(n) \in O(g(n))$$

Asymptotic Analysis and Limits

The following relationships hold for positive increasing functions f(n) and g(n). Since the running times of algorithms are positive and increasing, we can use these rules when analyzing algorithms.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \iff f(n) \in o(g(n))$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0 \implies f(n) \in \Theta(g(n))$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \implies a^{f(n)} \in o(a^{g(n)}) \text{ for any } a > 1$$

$$f(n) \in o(g(n)) \implies a^{f(n)} \in o(a^{g(n)}) \text{ for any } a > 1$$

$$f(n) \in \Theta(g(n)) \not\Longrightarrow a^{f(n)} \in \Theta(a^{g(n)})$$

Asymptotic Analysis and Algorithm Performance

What does it mean if $T_A(n) \in O(T_B(n))$?

Asymptotic Analysis and Algorithm Performance

What does it mean if $T_A(n) \in O(T_B(n))$?

Algorithm A is at most a constant factor slower than algorithm B.
The constant factor may be large!!!

Asymptotic Analysis and Algorithm Performance

What does it mean if $T_A(n) \in O(T_B(n))$?

Algorithm A is at most a constant factor slower than algorithm B.
The constant factor may be large!!!

What does it mean if $T_A(n) \in o(T_B(n))$?

Asymptotic Analysis and Algorithm Performance

What does it mean if $T_A(n) \in O(T_B(n))$?

Algorithm A is at most a constant factor slower than algorithm B.
The constant factor may be large!!!

What does it mean if $T_A(n) \in o(T_B(n))$?

For sufficiently large n, algorithm A will outperform algorithm B.

Asymptotic Analysis and Algorithm Performance

What does it mean if $T_A(n) \in O(T_B(n))$?

Algorithm A is at most a constant factor slower than algorithm B.
The constant factor may be large!!!

What does it mean if $T_A(n) \in o(T_B(n))$?

For sufficiently large n, algorithm A will outperform algorithm B.

Can we ignore constants?

Asymptotic Analysis and Algorithm Performance

What does it mean if $T_A(n) \in O(T_B(n))$?

Algorithm A is at most a constant factor slower than algorithm B.
The constant factor may be large!!!

What does it mean if $T_A(n) \in o(T_B(n))$?

For sufficiently large n, algorithm A will outperform algorithm B.

Can we ignore constants?

In a first filter step to select possible candidate algorithms and during algorithm design, this is helpful.

Subsequent choices have to be based on our experience, analyses that do take constants into account, or experimental evaluation.

Asymptotic Analysis and Algorithm Performance

What does it mean if $T_A(n) \in O(T_B(n))$?

Algorithm A is at most a constant factor slower than algorithm B.
The constant factor may be large!!!

What does it mean if $T_A(n) \in o(T_B(n))$?

For sufficiently large n, algorithm A will outperform algorithm B.

Can we ignore constants?

In a first filter step to select possible candidate algorithms and during algorithm design, this is helpful.

Subsequent choices have to be based on our experience, analyses that do take constants into account, or experimental evaluation.

What do we gain?

Asymptotic Analysis and Algorithm Performance

What does it mean if $T_A(n) \in O(T_B(n))$?

Algorithm A is at most a constant factor slower than algorithm B.
The constant factor may be large!!!

What does it mean if $T_A(n) \in o(T_B(n))$?

For sufficiently large n, algorithm A will outperform algorithm B.

Can we ignore constants?

In a first filter step to select possible candidate algorithms and during algorithm design, this is helpful.

Subsequent choices have to be based on our experience, analyses that do take constants into account, or experimental evaluation.

What do we gain?

A simple, succinct expression of the performance of an algorithm.

Implementation of the Gale-Shapley Algorithm

StableMatching(M, W)

```
while there exists an unmarried man m
    do m proposes to the most preferable woman w he has not proposed to yet
        if w is unmarried or likes m better than her current partner m′
            then if w is married
                    then w divorces m′
                 w marries m
```

Questions we can and should ask about the algorithm:

- Is there always a stable matching?
- Does the algorithm always terminate?
- Does the algorithm always produce a stable matching?
- How efficient is the algorithm? Can we bound its running time?

Implementation of the Gale-Shapley Algorithm

StableMatching(M : Array[Man], W : Array[Woman])

```
Q = an empty queue
for every man m ∈ M
    do Q.enqueue(m)
while not Q.isEmpty()
    do m = Q.dequeue()
       w = W[m.nextOnList()]
       if not w.isMarried()
          then w.marry(m)
          else m′ = w.partner()
               if w.prefers(m, m′)
                  then w.marry(m)
                       Q.enqueue(m′)
                  else Q.enqueue(m)
```

Implementation of the Gale-Shapley Algorithm

StableMatching(M : Array[Man], W : Array[Woman])

```
Q = an empty queue
for every man m ∈ M
    do Q.enqueue(m)
while not Q.isEmpty()
    do m = Q.dequeue()
        w = W[m.nextOnList()]
        if not w.isMarried()
            then w.marry(m)
            else  m′ = w.partner()
                if w.prefers(m, m′)
                    then w.marry(m)
                        Q.enqueue(m′)
                    else  Q.enqueue(m)
```

Queue:

- $O(1)$ time per operation

Implementation of the Gale-Shapley Algorithm

StableMatching(M : Array[Man], W : Array[Woman])

```
Q = an empty queue
for every man m ∈ M
    do Q.enqueue(m)
while not Q.isEmpty()
    do m = Q.dequeue()
        w = W[m.nextOnList()]
        if not w.isMarried()
            then w.marry(m)
            else m′ = w.partner()
                if w.prefers(m, m′)
                    then w.marry(m)
                         Q.enqueue(m′)
                    else Q.enqueue(m)
```

Queue:

- $O(1)$ time per operation

Man:

- Preference list = array + current index/list
- nextOnList = access + increase index or pointer jump on list

$\Rightarrow$ $O(1)$ time

Implementation of the Gale-Shapley Algorithm

StableMatching(M : Array[Man], W : Array[Woman])

```
Q = an empty queue
for every man m ∈ M
    do Q.enqueue(m)
while not Q.isEmpty()
    do m = Q.dequeue()
        w = W[m.nextOnList()]
        if not w.isMarried()
            then w.marry(m)
            else  m′ = w.partner()
                if w.prefers(m, m′)
                    then w.marry(m)
                         Q.enqueue(m′)
                    else  Q.enqueue(m)
```

Woman:

- Stores pointer to her partner

$\Rightarrow$ isMarried/marry/partner take O(1) time

Implementation of the Gale-Shapley Algorithm

StableMatching(M : Array[Man], W : Array[Woman])

```
Q = an empty queue
for every man m ∈ M
    do Q.enqueue(m)
while not Q.isEmpty()
    do m = Q.dequeue()
       w = W[m.nextOnList()]
       if not w.isMarried()
          then w.marry(m)
          else m′ = w.partner()
               if w.prefers(m, m′)
                  then w.marry(m)
                       Q.enqueue(m′)
                  else Q.enqueue(m)
```

Woman:

- Stores pointer to her partner

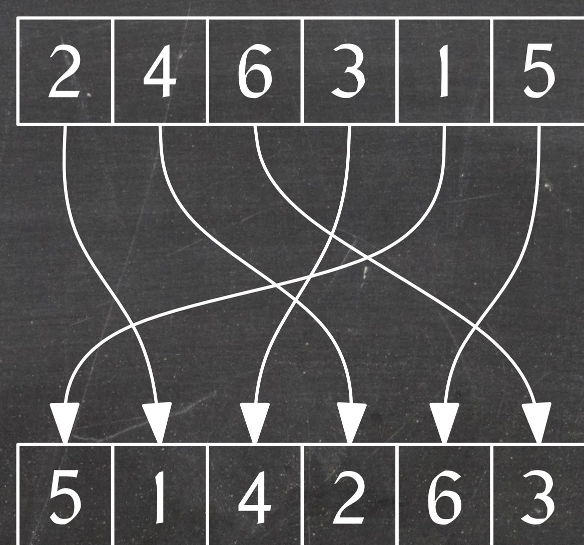
$\Rightarrow$ isMarried/marry/partner take $O(1)$ time

- prefers takes $O(1)$ time if we have an inverted preference list:
  - Map every man to his rank in the preference list.

Inverting a Preference List

InvertPreflist(w : Woman)

```
L = new array of size |w.preflist|
for i = 1 to |w.preflist|
    do L[w.preflist[i]] = i
w.preflist = L
```



This takes linear time.

Implementation of the Gale-Shapley Algorithm

StableMatching(M : Array[Man], W : Array[Woman])

```
Q = an empty queue
for every man m ∈ M
    do Q.enqueue(m)
for every woman w ∈ W
    do InvertPreflist(w)
while not Q.isEmpty()
    do m = Q.dequeue()
       w = W[m.nextOnList()]
       if not w.isMarried()
           then w.marry(m)
           else m′ = w.partner()
                if w.prefers(m, m′)
                    then w.marry(m)
                         Q.enqueue(m′)
                    else Q.enqueue(m)
```

The Gale-Shapley algorithm can be implemented to run in $O(n^2)$ time.

Summary

Review of things you should know

- Proof by contradiction
- Arrays, linked lists, stacks, and queues

Analysis of algorithms

- Worst-case and average-case running time
- Asymptotic notation