Fundamentals

Textbook reading

Chapter 0
Chapter 1
Overview

**Design principle:**
- Steady progress towards a solution

**Proof techniques:**
- Proof by contradiction

**Basics of algorithm analysis:**
- Asymptotic notation ($\Theta$, $\Omega$, and $\Theta$)
- Linear time, $n \lg n$ time, quadratic time, exponential time

**Basic data structures:**
- Arrays
- Linked lists
Stable Marriages: The Gale-Shapley Algorithm

**Proposal Algorithm** ($M$, $W$)

1. **while** there is an unmarried man $m$

2. **do** $m$ chooses his favourite woman $w$ he has not proposed to yet

3. $m$ proposes to $w$

4. **if** $w$ is not married or likes $m$ better than her current partner $m'$

5. **then** $w$ divorces $m'$

6. $w$ marries $m$

- Is there always a set of $n$ stable marriages?
- Does the algorithm ever terminate?
- Does the algorithm always produce a correct answer?
- How efficient is the algorithm? Can we give an upper bound on its running time?
Stable Marriages: The Gale-Shapley Algorithm

**PROPOSAL-ALGORITHM**\((M, W)\)

1. while there is an unmarried man \(m\)
2. do \(m\) chooses his favourite woman \(w\) he has not proposed to yet
3. \(m\) proposes to \(w\)
4. if \(w\) is not married or likes \(m\) better than her current partner \(m'\)
5. then \(w\) divorces \(m'\)
6. \(w\) marries \(m\)

- Is there always a set of \(n\) stable marriages?
- Does the algorithm ever terminate?
- Does the algorithm always produce a correct answer?
- How efficient is the algorithm? Can we give an upper bound on its running time?
Stable Marriages: The Gale-Shapley Algorithm

**PROPOSAL-ALGORITHM**\((M, W)\)

1. while there is an unmarried man \(m\)
2. do \(m\) chooses his favourite woman \(w\) he has not proposed to yet
3. \(m\) proposes to \(w\)
4. if \(w\) is not married or likes \(m\) better than her current partner \(m'\)
5. then \(w\) divorces \(m'\)
6. \(w\) marries \(m\)

- Is there always a set of \(n\) stable marriages?
- Does the algorithm ever terminate?
- Does the algorithm always produce a correct answer?
- How efficient is the algorithm? Can we give an upper bound on its running time?
Stable Marriages: The Gale-Shapley Algorithm

**Proposal Algorithm** \((M, W)\)

1. while there is an unmarried man \(m\)

2. \hspace{1cm} do \(m\) chooses his favourite woman \(w\) he has not proposed to yet

3. \hspace{2cm} \(m\) proposes to \(w\)

4. \hspace{2cm} if \(w\) is not married or likes \(m\) better than her current partner \(m'\)

5. \hspace{3cm} then \(w\) divorces \(m'\)

6. \hspace{3cm} \(w\) marries \(m\)

- Is there always a set of \(n\) stable marriages?
- Does the algorithm ever terminate?
- Does the algorithm always produce a correct answer?
- How efficient is the algorithm? Can we give an upper bound on its running time?
Stable Marriages: The Gale-Shapley Algorithm

**PROPOSAL-ALGORITHM**(\(M, W\))

1. while there is an unmarried man \(m\) who has not proposed to all women
2. do \(m\) chooses his favourite woman \(w\) he has not proposed to yet
3. \(m\) proposes to \(w\)
4. if \(w\) is not married or likes \(m\) better than her current partner \(m'\)
5. then \(w\) divorces \(m'\)
6. \(w\) marries \(m\)

- Is there always a set of \(n\) stable marriages?
- Does the algorithm ever terminate?
- Does the algorithm always produce a correct answer?
- How efficient is the algorithm? Can we give an upper bound on its running time?
Lemma: The proposal algorithm terminates after at most \( n^2 \) iterations.
Lemma: The proposal algorithm terminates after at most $n^2$ iterations.

- There are $n$ men
Termination of the Proposal Algorithm

Lemma: The proposal algorithm terminates after at most $n^2$ iterations.

- There are $n$ men
- Every man can propose to $n$ women
Lemma: The proposal 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 Proposal Algorithm

Lemma: The proposal 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

∴ There are $n^2$ proposals to be made
Lemma: The proposal 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
  \[\therefore \text{There are } n^2 \text{ proposals to be made}\]

- In every iteration, one proposal is made.
Lemma: The proposal 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
  \[ \therefore \text{There are } n^2 \text{ proposals to be made} \]

- In every iteration, one proposal is made.

  \[ \therefore \text{The total number of iterations is at most } n^2. \]
Lemma: The proposal 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
  \[ \therefore \text{ There are } n^2 \text{ proposals to be made} \]

- In every iteration, one proposal is made.

\[ \therefore \text{ The total number of iterations is at most } n^2. \]

To terminate, an algorithm has to make steady progress towards a solution.
Lemma: When the proposal algorithm terminates every woman is married (and, therefore, every man is married).
Lemma: When the proposal algorithm terminates every woman is married (and, therefore, every man is married).

Proof by contradiction:
- Assume that what we want to prove is not correct.
- Prove that this leads to a contradiction.
  - Either to the assumption we made
  - Or to some known fact
Lemma: When the proposal algorithm terminates every woman is married (and, therefore, every man is married).

Assumption: There is an unmarried woman $w$ at the end of the algorithm.
Lemma: When the proposal algorithm terminates every woman is married (and, therefore, every man is married).

Assumption: There is an unmarried woman \( w \) at the end of the algorithm.

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


**Lemma:** When the proposal algorithm terminates every woman is married (and, therefore, every man is married).

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

- A woman, once married, stays married (not necessarily to the same man)
- If there’s an unmarried woman $w$, there must be an unmarried man $m$
Lemma: When the proposal algorithm terminates every woman is married (and, therefore, every man is married).

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

- A woman, once married, stays married (not necessarily to the same man)
- If there’s 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:** When the proposal algorithm terminates every woman is married (and, therefore, every man is married).

**Assumption:** There is an unmarried woman \( w \) at the end of the algorithm.

- A woman, once married, stays married (not necessarily to the same man)
- If there’s 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
**Lemma:** When the proposal algorithm terminates every woman is married (and, therefore, every man is married).

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

- A woman, once married, stays married (not necessarily to the same man)
- If there’s 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 Marriages: The Gale-Shapley Algorithm

**Proposal Algorithm** \((M, W)\)

1. while there is an unmarried man \(m\) that has not proposed to all women
2. do \(m\) chooses his favourite woman \(w\) he has not proposed to yet
3. \(m\) proposes to \(w\)
4. if \(w\) is not married or likes \(m\) better than her current partner \(m'\)
5. then \(w\) divorces \(m'\)
6. \(w\) marries \(m\)

- Is there always a set of \(n\) stable marriages?
- Does the algorithm ever terminate?
- Does the algorithm always produce a correct answer?
- How efficient is the algorithm? Can we give an upper bound on its running time?
Stable Marriages: The Gale-Shapley Algorithm

**PROPOSAL-ALGORITHM** \((M, W)\)

1. while there is an unmarried man \(m\)
2. do \(m\) chooses his favourite woman \(w\) he has not proposed to yet
3. \(m\) proposes to \(w\)
4. if \(w\) is not married or likes \(m\) better than her current partner \(m'\)
5. then \(w\) divorces \(m'\)
6. \(w\) marries \(m\)

- Is there always a set of \(n\) stable marriages?
- Does the algorithm ever terminate?
- Does the algorithm always produce a correct answer?
- How efficient is the algorithm? Can we give an upper bound on its running time?
Stable Marriages: The Gale-Shapley Algorithm

**PROPOSAL-ALGORITHM** \((M, W)\)

1. while there is an unmarried man \(m\)

2. do \(m\) chooses his favourite woman \(w\) he has not proposed to yet  

3. \(m\) proposes to \(w\)

4. if \(w\) is not married or likes \(m\) better than her current partner \(m'\)  

5. then \(w\) divorces \(m'\)

6. \(w\) marries \(m\)

- Is there always a set of \(n\) stable marriages?
- Does the algorithm ever terminate?
- Does the algorithm always produce a correct answer?
- How efficient is the algorithm? Can we give an upper bound on its running time?
Correctness of the Proposal Algorithm

**Lemma:** When the proposal algorithm terminates, all marriages are stable.

- Let \((w, m)\) and \((w', m')\) be two marriages so that
  - \(w\) prefers \(m'\) over \(m\) and
  - \(m'\) prefers \(w\) over \(w'\).

- Since \(m'\) prefers \(w\) over \(w'\), he must have proposed to \(w\) before getting married with \(w'\).
Let $m''$ be the man that $w$ is married to immediately after $m'$ has proposed.

- If $w$ accepts $m'$, then $m'' = m'$
- If $w$ rejects $m'$, then $m'' \succeq_w m'$ (read: $w$ prefers $m''$ over $m'$)

∴ $m'' \preceq_w m'$
Let $m''$ be the man that $w$ is married to immediately after $m'$ has proposed.

- If $w$ accepts $m'$, then $m'' = m'$
- If $w$ rejects $m'$, then $m'' \succ_w m'$ (read: $w$ prefers $m''$ over $m'$)

∴ $m'' \succeq_w m'$

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

$m' \preceq_w m_1 <_w m_2 <_w \cdots <_w m_k = m$. 
Let $m''$ be the man that $w$ is married to immediately after $m'$ has proposed.

- If $w$ accepts $m'$, then $m'' = m'$
- If $w$ rejects $m'$, then $m'' \succeq_w m'$ (read: $w$ prefers $m''$ over $m'$)

$\therefore m'' \succeq_w m'$

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

$m' \preceq_w m_1 \prec_w m_2 \prec_w \cdots \prec_w m_k = m$.

$\therefore w$ prefers $m$ over $m'$
Let $m''$ be the man that $w$ is married to immediately after $m'$ has proposed.
- If $w$ accepts $m'$, then $m'' = m'$
- If $w$ rejects $m'$, then $m'' \succ_w m'$ (read: $w$ prefers $m''$ over $m'$)

\[ \therefore m'' \succeq_w m' \]

Let $m'' = m_1, m_2, \ldots, m_k = m$ be the sequence of partners $w$ has from this time on till the end of the algorithm.
- $m' \succeq_w m_1 \prec_w m_2 \prec_w \cdots \prec_w m_k = m$.

\[ \therefore w \text{ prefers } m \text{ over } m' \]

**Contradiction.**
Let $m''$ be the man that $w$ is married to immediately after $m'$ has proposed.
- If $w$ accepts $m'$, then $m'' = m'$
- If $w$ rejects $m'$, then $m'' \succ_w m'$ (read: $w$ prefers $m''$ over $m'$)
\[ \therefore m'' \succeq_w m' \]

Let $m'' = m_1, m_2, \ldots, m_k = m$ be the sequence of partners $w$ has from this time on till the end of the algorithm.
- $m' \preceq_w m_1 \prec_w m_2 \prec_w \cdots \prec_w m_k = m$.
\[ \therefore w \text{ prefers } m \text{ over } m' \]

**Contradiction.**

**Corollary:** There always exists a set of stable marriages.
Stable Marriages: The Gale-Shapley Algorithm

**PROPOSAL-ALGORITHM** \( (M, W) \)

1. **while** there is an unmarried man \( m \)
2. **do** \( m \) chooses his favourite woman \( w \) he has not proposed to before
3. \( m \) proposes to \( w \)
4. **if** \( w \) is not married or likes \( m \) better than her current partner \( m' \)
5. **then** \( w \) divorces \( m' \)
6. \( w \) marries \( m \)

- Is there always a set of \( n \) stable marriages?
- Does the algorithm ever terminate?
- Does the algorithm always produce a correct answer?
- How efficient is the algorithm? Can we give an upper bound on its running time?
Stable Marriages: The Gale-Shapley Algorithm

PROPOSAL-ALGORITHM \((M, W)\)

1. while there is an unmarried man \(m\)
2. do \(m\) chooses his favourite woman \(w\) he has not proposed to before
3. \(m\) proposes to \(w\)
4. if \(w\) is not married or likes \(m\) better than her current partner \(m'\)
5. then \(w\) divorces \(m'\)
6. \(w\) marries \(m\)

- Is there always a set of \(n\) stable marriages?
- Does the algorithm ever terminate?
- Does the algorithm always produce a correct answer?
- How efficient is the algorithm? Can we give an upper bound on its running time?
Does the final set of marriages depend on the order in which the men propose?
- It does not.

Is the process fair?
- The men may fare much better than the women.

Can the algorithm be implemented efficiently?

Can we design a faster algorithm?
- Yes, using randomization.
Computational Tractability

Criterion of efficiency = resource consumption:

- Running time
- Memory usage
- Disk usage
- Amount of communication in distributed computations
- ...
An algorithm is efficient if, when implemented, it runs quickly on real input instances.
Definition of Efficiency — First Attempt

An algorithm is efficient if, when implemented, it runs quickly on real input instances.

Pros:
Definition of Efficiency — First Attempt

An algorithm is efficient if, when implemented, it runs quickly on real input instances.

Pros:

- Captures what we really care about.
An algorithm is efficient if, when implemented, it runs quickly on real input instances.

Pros:

- Captures what we really care about.

Cons:
Definition of Efficiency — First Attempt

An algorithm is efficient if, when implemented, it runs quickly on real input instances.

Pros:
- Captures what we really care about.

Cons:
- Efficiency of algorithm depends on quality of implementation and speed of machine.
- Does not capture the dependence of running time on input size.
- What are “real input instances”?

Therefore: Need a definition that is **platform-independent**, **instance-independent**, and **captures the dependence on the input size**.
Model of Computation

**RAM Model:**

Elementary operations take constant time:
- Arithmetic operations: addition, subtraction, multiplication, division
- Boolean operations: and, or, not
- If-statements
- Checking of loop conditions
- Assignment

*By counting elementary operations, we can compare the running times of two algorithms when run on the same computer.*
Running Time Depends on Structure of Input

An algorithm’s running time may differ greatly for different input instances of the same size.

Example: Insertion Sort

```
INSERTIONSORT(A, n)
1    for i ← 2 to n
2       do key ← A[i]
3          j ← i - 1
4          while j > 0 and A[j] > key
6           A[j + 1] ← key
```
Running Time Depends on Structure of Input

An algorithm’s running time may differ greatly for different input instances of the same size.

Example: Insertion Sort

```plaintext
INSERTIONSORT(A, n)
  1  for i ← 2 to n
  2    do key ← A[i]
  3      j ← i - 1
  4      while j > 0 and A[j] > key
  6      A[j + 1] ← key
```

How do we unify this into one figure?
Worst-Case and Average-Case Analysis

The **worst-case running time** of an algorithm $\mathcal{A}$ is a function $T_{\mathcal{A}}(n)$ such that $T_{\mathcal{A}}(n)$ is the maximal running time of $\mathcal{A}$ over all inputs of length $n$.

The **average-case running time** of an algorithm $\mathcal{A}$ is a function $T_{\mathcal{A}}(n)$ such that $T_{\mathcal{A}}(n)$ is the average of the running times of $\mathcal{A}$ on all inputs of length $n$. 
Definition of Efficiency — Second Attempt

An algorithm is efficient if, at an analytical level, it achieves a better worst-case performance than the brute-force method.
Definition of Efficiency — Second Attempt

An algorithm is efficient if, at an analytical level, it achieves a better worst-case performance than the brute-force method.

Pros:
Definition of Efficiency — Second Attempt

An algorithm is efficient if, at an analytical level, it achieves a better worst-case performance than the brute-force method.

Pros:

- Platform-independent
- Instance-independent
- Captures dependence on input size
- Captures the idea that we should try to be intelligent in solving the problem
Definition of Efficiency — Second Attempt

An algorithm is efficient if, at an analytical level, it achieves a better worst-case performance than the brute-force method.

Pros:

■ Platform-independent
■ Instance-independent
■ Captures dependence on input size
■ Captures the idea that we should try to be intelligent in solving the problem

Cons:
Definition of Efficiency — Second Attempt

An algorithm is efficient if, at an analytical level, it achieves a better worst-case performance than the brute-force method.

Pros:

- Platform-independent
- Instance-independent
- Captures dependence on input size
- Captures the idea that we should try to be intelligent in solving the problem

Cons:

- Vague
Efficiency = Polynomial Running Time

An algorithm is efficient if its worst-case running time is polynomial in the input size.

Motivation:

- If input size doubles, running time should increase by only a constant factor.
Efficiency = Polynomial Running Time

An algorithm is efficient if its worst-case running time is polynomial in the input size.

Motivation:

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

Questions:
Efficiency = Polynomial Running Time

An algorithm is efficient if its worst-case running time is polynomial in the input size.

Motivation:

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

Questions:

- Is $n^{100}$ efficient?
Efficiency = Polynomial Running Time

An algorithm is efficient if its worst-case running time is polynomial in the input size.

**Motivation:**

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

**Questions:**

- Is $n^{100}$ efficient?
- Is $n^{1+0.02 \log n}$ inefficient?
Efficiency = Polynomial Running Time

An algorithm is efficient if its worst-case running time is polynomial in the input size.

Motivation:

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

Questions:

- Is $n^{100}$ efficient? No.
- Is $n^{1+0.02\log n}$ inefficient?
Efficiency = Polynomial Running Time

An algorithm is efficient if its worst-case running time is polynomial in the input size.

Motivation:

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

Questions:

- Is $n^{100}$ efficient? No.
- Is $n^{1+0.02 \log n}$ inefficient? No.
Efficiency = Polynomial Running Time

An algorithm is efficient if its worst-case running time is polynomial in the input size.

Motivation:

- If input size doubles, 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.
**Scenario:** Given two algorithms $A_1$ and $A_2$ which we want to compare.

Do we care which algorithm is faster for small inputs?
Asymptotic Running Time

**Scenario:** Given two algorithms $A_1$ and $A_2$ which we want to compare.

Do we care which algorithm is faster for small inputs?

What matters is which algorithm is faster for large inputs.
**Asymptotic Running Time**

**Scenario:** Given two algorithms $A_1$ and $A_2$ which we want to compare.

Do we care which algorithm is faster for small inputs?

What matters is which algorithm is faster for large inputs.

**Formally:**

We want that $T_1(n) < T_2(n)$, for all $n \geq n_0$, where $n_0$ is a constant.
\[ f(n) = \Theta(g(n)) \]

\[ \iff \]

\[ \exists c > 0, n_0 \geq 0 \forall n \geq n_0 : f(n) \leq c \cdot g(n) \]
**Ω-Notation**

\[ f(n) = \Omega(g(n)) \]

\[ \exists c > 0, n_0 \geq 0 \forall n \geq n_0 : f(n) \geq c \cdot g(n) \]
\( f(n) = \Theta(g(n)) \)

\[ \iff \]

\[ \exists c_1 > 0, c_2 > 0, n_0 \geq 0 \forall n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \]
\textbf{\textit{o}-Notation}

\[ f(n) = o(g(n)) \]

\[ \forall c > 0 \exists n_0 \geq 0 \forall n \geq n_0 : f(n) \leq c \cdot g(n) \]
**ω-Notation**

\[ f(n) = \omega(g(n)) \]

\[ \forall c > 0 \exists n_0 \geq 0 \forall n \geq n_0 : f(n) \geq c \cdot g(n) \]
A Few Simple Facts

\[ f(n) = \mathcal{O}(f(n)) \quad f(n) = \Omega(f(n)) \quad f(n) = \Theta(f(n)) \]

\[ f(n) = \mathcal{O}(g(n)) \text{ and } g(n) = \mathcal{O}(h(n)) \implies f(n) = \mathcal{O}(h(n)) \]

\[ f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \implies f(n) = \Omega(h(n)) \]

\[ f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \implies f(n) = \Theta(h(n)) \]

\[ f(n) = \mathcal{O}(g(n)) \iff g(n) = \Omega(f(n)) \]

\[ f(n) = o(g(n)) \iff g(n) = \omega(f(n)) \]

\[ f(n) = \mathcal{O}(g(n)) \text{ and } f(n) = \Omega(g(n)) \implies f(n) = \Theta(g(n)) \]

\[ f_1(n) = \mathcal{O}(g_1(n)) \text{ and } f_2(n) = \mathcal{O}(g_2(n)) \implies f_1(n) + f_2(n) = \mathcal{O}(g_1(n) + g_2(n)) \]

\[ f(n) = \mathcal{O}(g(n)) \implies f(n) + g(n) = \mathcal{O}(g(n)) \]
Asymptotic Analysis and Limits

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

\[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = c \iff f(n) = \Theta(g(n)) \]

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

\[ f(n) = o(g(n)) \implies a^{f(n)} = o(a^{g(n)}), \text{ for any } a > 1 \]

\[ f(n) = \Theta(g(n)) \nleftrightarrow a^{f(n)} = \Theta(a^{g(n)}) \]
Asymptotic Notation and Algorithm Performance

What does it mean if $T_1(n) = \mathcal{O}(T_2(n))$?
Asymptotic Notation and Algorithm Performance

What does it mean if $T_1(n) = O(T_2(n))$?

- Algorithm $A_1$ is by at most a constant factor slower than $A_2$.
- The factor may be large!!!
What does it mean if $T_1(n) = \mathcal{O}(T_2(n))$?
- Algorithm $A_1$ is by at most a constant factor slower than $A_2$.
- The factor may be large!!!

What does it mean if $T_1(n) = o(T_2(n))$?
Asymptotic Notation and Algorithm Performance

What does it mean if $T_1(n) = \mathcal{O}(T_2(n))$?
- Algorithm $A_1$ is by at most a constant factor slower than $A_2$.
- The factor may be large!!!

What does it mean if $T_1(n) = o(T_2(n))$?
- For sufficiently large $n$, algorithm $A_1$ will outperform algorithm $A_2$. 
Asymptotic Notation and Algorithm Performance

What does it mean if $T_1(n) = \mathcal{O}(T_2(n))$?
- Algorithm $A_1$ is by at most a constant factor slower than $A_2$.
- The factor may be large!!!

What does it mean if $T_1(n) = o(T_2(n))$?
- For sufficiently large $n$, algorithm $A_1$ will outperform algorithm $A_2$.

Can we ignore constants?
Asymptotic Notation and Algorithm Performance

What does it mean if $T_1(n) = \mathcal{O}(T_2(n))$?
- Algorithm $A_1$ is by at most a constant factor slower than $A_2$.
- The factor may be large!!!

What does it mean if $T_1(n) = o(T_2(n))$?
- For sufficiently large $n$, algorithm $A_1$ will outperform algorithm $A_2$.

Can we ignore constants?
- In a first filter step of possible candidates, this is desirable.
- Subsequent choices have to be based on our experience or analyses that take constants into account.
Asymptotic Notation and Algorithm Performance

What does it mean if $T_1(n) = \mathcal{O}(T_2(n))$?
- Algorithm $A_1$ is by at most a constant factor slower than $A_2$.
- The factor may be large!!

What does it mean if $T_1(n) = o(T_2(n))$?
- For sufficiently large $n$, algorithm $A_1$ will outperform algorithm $A_2$.

Can we ignore constants?
- In a first filter step of possible candidates, this is desirable.
- Subsequent choices have to be based on our experience or analyses that take constants into account.

What do we gain?
Asymptotic Notation and Algorithm Performance

What does it mean if $T_1(n) = \mathcal{O}(T_2(n))$?
- Algorithm $A_1$ is by at most a constant factor slower than $A_2$.
- The factor may be large!!!

What does it mean if $T_1(n) = o(T_2(n))$?
- For sufficiently large $n$, algorithm $A_1$ will outperform algorithm $A_2$.

Can we ignore constants?
- In a first filter step of possible candidates, this is desirable.
- Subsequent choices have to be based on our experience or analyses that take constants into account.

What do we gain?
- A simple, succinct expression of the performance of an algorithm.
**Example:** Merging two sorted sequences

```
1 8 11 30 37 54
3 12 23 61 77
```

```
1 3 8 11 12 23 30 37 54 61 77
```
**Linear Time (2)**

**`Merge(A, B)`**

1. \( C \leftarrow \emptyset \)
2. while \( A \) and \( B \) are non-empty
3. \hspace{1em} do Let \( a \) be the first element in \( A \)
4. \hspace{2em} Let \( b \) be the first element in \( B \)
5. \hspace{3em} if \( a < b \)
6. \hspace{4em} then Remove \( a \) from \( A \)
7. \hspace{5em} Append \( a \) to \( C \)
8. \hspace{3em} else Remove \( b \) from \( B \)
9. \hspace{4em} Append \( b \) to \( C \)
10. \hspace{2em} if \( A \) is non-empty
11. \hspace{3em} then Append \( A \) to \( C \)
12. \hspace{2em} else Append \( B \) to \( C \)
13. return \( C \)
**Linear Time (3)**

**MERGE**($A, p, q, r$)

1. $n_1 \leftarrow q - p + 1$
2. $n_2 \leftarrow r - q$
3. **for** $i \leftarrow 1$ **to** $n_1$
4. **do** $L[i] \leftarrow A[p + i - 1]$
5. **for** $i \leftarrow 1$ **to** $n_2$
6. **do** $R[i] \leftarrow A[q + i]$
7. $L[n_1 + 1] \leftarrow \infty$
8. $R[n_2 + 1] \leftarrow \infty$
9. $i \leftarrow 1$
10. $j \leftarrow 1$
11. **for** $k \leftarrow 1$ **to** $r - p + 1$
12. **do if** $L[i] < R[j]$
13. **then** $A[k] \leftarrow L[i]$
14. $i \leftarrow i + 1$
15. **else** $A[k] \leftarrow R[j]$
16. $j \leftarrow j + 1$
**Example:** Sorting a sequence

```
MERGE-SORT(A, p, r)
1    if r > p
2      then q ← \left\lfloor \frac{p+r}{2} \right\rfloor
3      Merge-Sort(A, p, q)
4      Merge-Sort(A, q + 1, r)
5      Merge(A, p, q, r)
```
\( \mathcal{O}(n \lg n) \) Time

**Example:** Sorting a sequence

```plaintext
MERGE-SORT(A, p, r)
1   if \( r > p \)
2        then \( q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor \)
3         Merge-Sort(A, p, q)
4         Merge-Sort(A, q + 1, r)
5         Merge(A, p, q, r)
```

**Comments:**
- \( \mathcal{O}(n \lg n) \) is the best we can do for sorting.
- \( \mathcal{O}(n \lg n) \) is a typical time bound for divide-and-conquer algorithms.
**Example:** Closest pair

Given a set $P$ of $n$ points, find the two points in $P$ whose distance is smallest.
**Closest Pair**

1. $d \leftarrow +\infty$
2. for every point $p \in P$
3. do for every point $q \in P$ such that $q \neq p$
4. do if $\text{dist}(p, q) < d$
5. then $d \leftarrow \text{dist}(p, q)$
6. pair $\leftarrow (p, q)$
7. return pair
Quadratic Time (2)

CLOSESTPAIR \( P \)

1. \( d \leftarrow +\infty \)
2. \textbf{for} every point \( p \in P \)
3. \ \textbf{do} for every point \( q \in P \) such that \( q \neq p \)
4. \ \textbf{do} if \( \text{dist}(p, q) < d \)
5. \ \textbf{then} \( d \leftarrow \text{dist}(p, q) \)
6. \ \text{pair} \leftarrow (p, q)
7. \textbf{return} pair

Comments:
- Quadratic time is easy to recognize here because there are two nested loops iterating over the whole input.
- Clever use of divide-and-conquer and geometric insight leads to an \( O(n \lg n) \) solution.
**Example:** Matrix multiplication

\[
\text{MATRIX-MULTIPLY}(A, B, n)
\]

1. \textbf{for} \( i \leftarrow 1 \) \textbf{to} \( n \)
2. \hspace{1em} \textbf{do for} \( j \leftarrow 1 \) \textbf{to} \( n \)
3. \hspace{2em} \textbf{do} \( x \leftarrow 0 \)
4. \hspace{3em} \textbf{for} \( k \leftarrow 1 \) \textbf{to} \( n \)
5. \hspace{4em} \textbf{do} \( x \leftarrow x + A[i][k] \times B[k][j] \)
6. \hspace{3em} \( C[i][j] \leftarrow x \)
7. \textbf{return} \( C \)
Example: Matrix multiplication

**Matrix-Multiply** \((A, B, n)\)

1. for \(i \leftarrow 1\) to \(n\)
2. do for \(j \leftarrow 1\) to \(n\)
3. do \(x \leftarrow 0\)
4. for \(k \leftarrow 1\) to \(n\)
5. do \(x \leftarrow x + A[i][k] \times B[k][j]\)
6. \(C[i][j] \leftarrow x\)
7. return \(C\)

Comments:
- Again, cubic time is easy to recognize here because there are three nested loops iterating over a complete row or column.
- Clever use of divide-and-conquer leads to a sub-cubic algorithm (Strassen’s algorithm).
**Example:** Finding an independent set of size $k$

An independent set is a subset, $S$, of vertices such that no two vertices in $S$ are adjacent.

**Goal:** Find an independent set of size $k$, if one exists.
INDEPENDENT-SET \((G, k)\)

1. for every set \(S\) of \(k\) vertices
2. do if \(S\) is independent
3. then return \(S\)
4. return \(\emptyset\)
Polynomial Time (2)

**INDEPENDENT-SET**($G, k$)

1. for every set $S$ of $k$ vertices
2. do if $S$ is independent
3. then return $S$
4. return $\emptyset$

**Analysis:**

- There are $\binom{n}{k} = \frac{n(n-1)(n-2)\ldots(n-k+1)}{1\cdot2\ldots\cdot k} = \mathcal{O}(n^k)$ sets of $k$ vertices
Polynomial Time (2)

**INDEPENDENT-SET**\((G, k)\)

1. For every set \(S\) of \(k\) vertices
2. Do if \(S\) is independent
3. Then return \(S\)
4. Return \(\emptyset\)

**Analysis:**

- There are \(\binom{n}{k} = \frac{n(n-1)(n-2)\ldots(n-k+1)}{1\cdot2\ldots k} = \mathcal{O}(n^k)\) sets of \(k\) vertices
- Testing each for independence takes \(\mathcal{O}(n^2)\) time
**INDEPENDENT-SET**($G, k$)

1. for every set $S$ of $k$ vertices
2. do if $S$ is independent
3. then return $S$
4. return $\emptyset$

**Analysis:**

- There are $\binom{n}{k} = \frac{n(n-1)(n-2)\ldots(n-k+1)}{1\cdot2\ldots\cdot k} = O(n^k)$ sets of $k$ vertices
- Testing each for independence takes $O(n^2)$ time

Total: $O(n^{k+2})$
Example: Finding a largest independent set

**MAXIMUM-INDEPENDENT-SET**$(G)$

1. for every set $S$ of vertices, by decreasing cardinality
2. do if $S$ is independent
3. then return $S$
Example: Finding a largest independent set

**MAXIMUM-INDEPENDENT-SET**\((G)\)
1. for every set \(S\) of vertices, by decreasing cardinality
2. do if \(S\) is independent
3. then return \(S\)

**Analysis:**
- There are \(2^n\) sets of vertices

\[ \text{Running time} = \mathcal{O}(n^2 \cdot 2^n) \]
Example: Finding a largest independent set

\[
\text{MAXIMUM-INDEPENDENT-SET}(G)
\]

1. for every set \( S \) of vertices, by decreasing cardinality
2. do if \( S \) is independent
3. then return \( S \)

Analysis:
- There are \( 2^n \) sets of vertices
- \( \therefore \) Running time = \( O(n^2 \cdot 2^n) \)

Comments:
- Independent set is an NP-complete problem, that is, it is unlikely to have a polynomial-time solution.
- \( n! = \) brute-force solution to sequencing problems
- \( 2^n = \) brute-force solution to subset selection problems
Sublinear Time

Example: Binary search

\[
\text{Binary-Search}(A, p, r, x)
\]

1. \textbf{if } p > r \\
2. \textbf{then return } false \\
3. \textbf{else} \quad q \leftarrow \left\lfloor \frac{p + r}{2} \right\rfloor \\
4. \quad \textbf{if } A[q] = x \\
5. \quad \textbf{then return } q \\
6. \quad \textbf{else if } x < A[q] \\
7. \quad \textbf{then } \text{Binary-Search}(A, p, q - 1, x) \\
8. \quad \textbf{else } \text{Binary-Search}(A, q + 1, r, x)
**Sublinear Time**

**Example:** Binary search

```
BINARY-SEARCH(A, p, r, x)
1  if p > r
2      then return false
3  else q ← ⌊(p+r)/2⌋
4      if A[q] = x
5          then return q
6          else if x < A[q]
7              then Binary-Search(A, p, q − 1, x)
8              else Binary-Search(A, q + 1, r, x)
```

**Analysis:**
- Every iteration halves the set of candidate positions
- Running time is $O(\lg n)$
Implementation of Gale-Shapley Algorithm

**PROPOSAL-ALGORITHM**(\(M, W\))
1. while there is an unmarried man \(m\)
2. do \(m\) chooses his favourite woman \(w\) he has not proposed to yet
3. \(m\) proposes to \(w\)
4. if \(w\) is not married or likes \(m\) better than her current partner \(m'\)
5. then \(w\) divorces \(m'\)
6. \(w\) marries \(m\)

**Lemma:** The Gale-Shapley algorithm can be implemented using elementary data structures so that it runs in \(O(n^2)\) time.

**Tools:**
- Arrays
- A linked list or another array
Summary

Design principle:
- Steady progress towards a solution

Proof techniques:
- Proof by contradiction

Basics of algorithm analysis:
- Asymptotic notation ($\mathcal{O}$, $\Omega$, and $\Theta$)
- Linear time, $n \lg n$ time, quadratic time, exponential time

Basic data structures:
- Arrays
- Linked lists