Data Structures and Algorithms

Bachelor's Degree in Informatics Engineering (UPC)

Fall 2021/2022

Instructors

Gabriel Valiente	Amalia Duch	Salvador Roura	Albert Oliveras
T10 (A5E02)			
P11 (A5002)	P12 (A5103)		
L11 (A5002)	L12 (A5103)	L13 (A6203)	L14 (A5106)

Outline

Lecture 1
Analysis of Algorithms I

Analysis of Algorithms I

Contents

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009

- A way to describe behavior of functions in the limit. We are studying asymptotic efficiency.
- Describe growth of functions.
- Focus on what's important by abstracting away low-order terms and constant factors.
- How we indicate running times of algorithms.
- A way to compare sizes of functions:

$$f \in O(g) \qquad f \leqslant g \qquad\qquad f \in o(g) \qquad f < g$$
$$f \in \Omega(g) \qquad f \geqslant g \qquad\qquad f \in \omega(g) \qquad f > g$$
$$f \in \Theta(g) \qquad f = g$$

Analysis of Algorithms I
3.1 Asymptotic notation
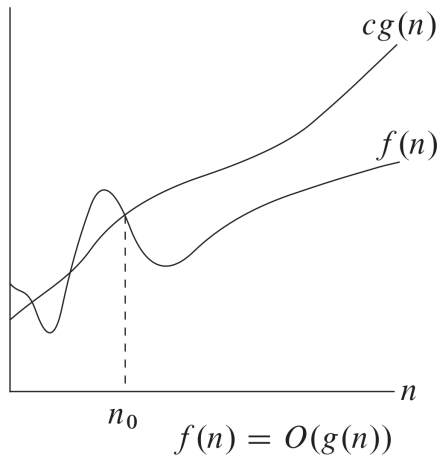
O-notation

$$f(n) = O(g(n))$$

Analysis of Algorithms I
3.1 Asymptotic notation

O-notation

$O(g(n)) = \{f(n) :$ there exist positive constants c and n_0
$\qquad\qquad$ such that $0 \leqslant f(n) \leqslant c\,g(n)$ for all $n \geqslant n_0\}$
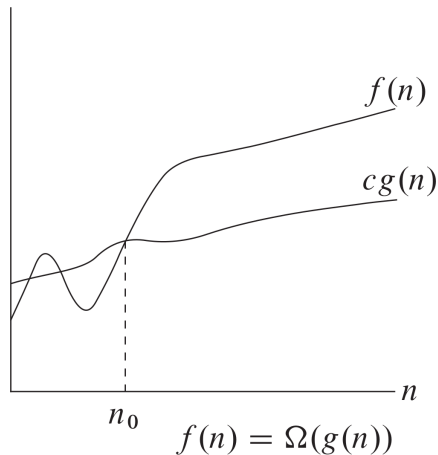
Another view, probably easier to use:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

- $g(n)$ is an asymptotic upper bound for $f(n)$.
- If $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$.
- $2n^2 = O(n^3)$, with $c = 1$ and $n_0 = 2$.

Ω-notation

$$f(n) = \Omega(g(n))$$

Analysis of Algorithms I
3.1 Asymptotic notation

Ω-notation

$\Omega(g(n)) = \{f(n) :$ there exist positive constants c and n_0
such that $0 \leqslant c\,g(n) \leqslant f(n)$ for all $n \geqslant n_0\}$

Another view, probably easier to use:
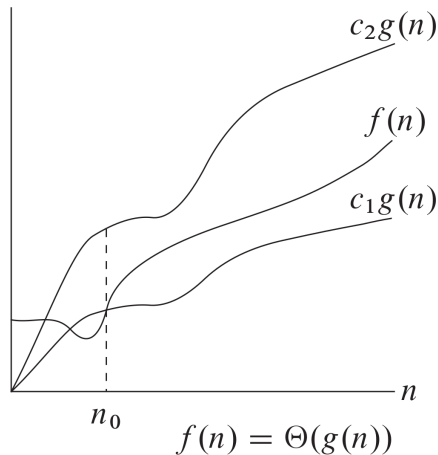
$$\lim_{n \to \infty} \frac{f(n)}{g(n)} > 0$$

- $g(n)$ is an asymptotic lower bound for $f(n)$.
- If $f(n) \in \Omega(g(n))$, we write $f(n) = \Omega(g(n))$.
- $\sqrt{n} = \Omega(\lg n)$, with $c = 1$ and $n_0 = 16$.

Analysis of Algorithms I
3.1 Asymptotic notation

Θ-notation

$$f(n) = \Theta(g(n))$$

Analysis of Algorithms I
3.1 Asymptotic notation

Θ-notation

$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0$
$\text{such that } 0 \leqslant c_1 \, g(n) \leqslant f(n) \leqslant c_2 \, g(n) \text{ for all } n \geqslant n_0\}$

Another view, probably easier to use:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+$$

- $g(n)$ is an asymptotically tight bound for $f(n)$.
- If $f(n) \in \Theta(g(n))$, we write $f(n) = \Theta(g(n))$.
- $n^2/2 - 2n = \Theta(n^2)$, with $c_1 = 1/4$, $c_2 = 1/2$, and $n_0 = 8$.

o-notation

$$o(g(n)) = \{f(n) : \text{for all constants } c > 0, \text{ there exists a constant}$$
$$n_0 > 0 \text{ such that } 0 \leqslant f(n) < c\,g(n) \text{ for all } n \geqslant n_0\}$$

Another view, probably easier to use: $\lim\limits_{n \to \infty} \frac{f(n)}{g(n)} = 0$.

- $n^{1.9999} = o(n^2)$.
- $n^2/\lg n = o(n^2)$.
- $n^2 \neq o(n^2)$ (just like $2 \not< 2$).
- $n^2/1000 \neq o(n^2)$.

Analysis of Algorithms I
3.1 Asymptotic notation

ω-notation

$\omega(g(n)) = \{f(n) :$ for all constants $c > 0$, there exists a constant
$n_0 > 0$ such that $0 \leqslant c\,g(n) < f(n)$ for all $n \geqslant n_0\}$

Another view, probably easier to use: $\lim\limits_{n \to \infty} \frac{f(n)}{g(n)} = \infty$.

- $n^{2.0001} = \omega(n^2)$.
- $n^2 \lg n = \omega(n^2)$.
- $n^2 \neq \omega(n^2)$.

Analysis of Algorithms I
3.1 Asymptotic notation

Comparisons of functions

Transitivity:

$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ implies $f(n) = \Theta(h(n))$.

Same for O, Ω, o, and ω.

Reflexivity:

$f(n) = \Theta(f(n))$.

Same for O and Ω.

Symmetry:

$f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.

Transpose symmetry:

$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

$f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

Comparisons of functions

- $f(n)$ is asymptotically smaller than $g(n)$ if $f(n) = o(g(n))$.
- $f(n)$ is asymptotically larger than $g(n)$ if $f(n) = \omega(g(n))$.

No trichotomy

Although intuitively, we can liken O to $\leqslant$, Ω to $\geqslant$, and so on, unlike real numbers, where $a < b$, $a = b$, or $a > b$, we might not be able to compare functions.

Example

$n^{1+\sin n}$ and n, since $1 + \sin n$ oscillates between 0 and 2.

Monotonicity

- $f(n)$ is monotonically increasing if $m \leqslant n \implies f(m) \leqslant f(n)$.
- $f(n)$ is monotonically decreasing if $m \geqslant n \implies f(m) \geqslant f(n)$.
- $f(n)$ is strictly increasing if $m < n \implies f(m) < f(n)$.
- $f(n)$ is strictly decreasing if $m > n \implies f(m) > f(n)$.

Analysis of Algorithms I
3.2 Standard notations and common functions

Exponentials

Useful identities:

- $a^{-1} = 1/a$,
- $(a^m)^n = a^{mn}$,
- $a^m a^n = a^{m+n}$.

Can relate rates of growth of polynomials and exponentials: for all real constants a and b such that $a > 1$,

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0,$$

which implies that $n^b = o(a^n)$.

Analysis of Algorithms I
3.2 Standard notations and common functions

Logarithms

Notations:

- $\lg n = \log_2 n$ (binary logarithm),
- $\ln n = \log_e n$ (natural logarithm),
- $\lg^k n = (\lg n)^k$ (exponentiation),
- $\lg \lg n = \lg(\lg n)$ (composition).

Useful identities for all real $a > 0$, $b > 0$, $c > 0$, and n, and where logarithm bases are not 1:

- $a = b^{\log_b a}$,
- $\log_c(ab) = \log_c a + \log_c b$,
- $\log_b a^n = n \log_b a$,
- $\log_b a = (\log_c a)/(\log_c b)$,

- $\log_b(1/a) = -\log_b a$,
- $\log_b a = 1/\log_a b$,
- $a^{\log_b c} = c^{\log_b a}$.

Analysis of Algorithms I
3.2 Standard notations and common functions

Logarithms

Just as polynomials grow more slowly than exponentials, logarithms grow more slowly than polynomials. In

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0,$$

substitute $\lg n$ for n and 2^a for a:

$$\lim_{n \to \infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n \to \infty} \frac{\lg^b n}{n^a} = 0,$$

implying that $\lg^b n = o(n^a)$.

Factorials

$n! = 1 \cdot 2 \cdot 3 \cdots n$. Special case: $0! = 1$. Can use Stirling's approximation,

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right),$$

to derive that $\lg(n!) = \Theta(n \lg n)$.

Analysis of Algorithms I
Order of growth

Order of growth	Name	
$O(1) = O(n^0)$	Constant	Polynomial
$O(\log n)$	Logarithmic	Polynomial
$O(\sqrt{n}) = O(n^{1/2})$		Polynomial
$O(n)$	Linear	Polynomial
$O(n \log n)$	Quasi-linear	Polynomial
$O(n^2)$	Quadratic	Polynomial
$O(n^3)$	Cubic	Polynomial
$O(2^n)$		Exponential
$O(3^n)$		Exponential
$O(n!)$		Exponential

Analysis of Algorithms I
Order of growth

Linear
$f = O(n)$ $f(2n) \approx 2f(n)$ $f(10n) \approx 10f(n)$

Quadratic
$f = O(n^2)$ $f(2n) \approx 4f(n)$ $f(10n) \approx 100f(n)$
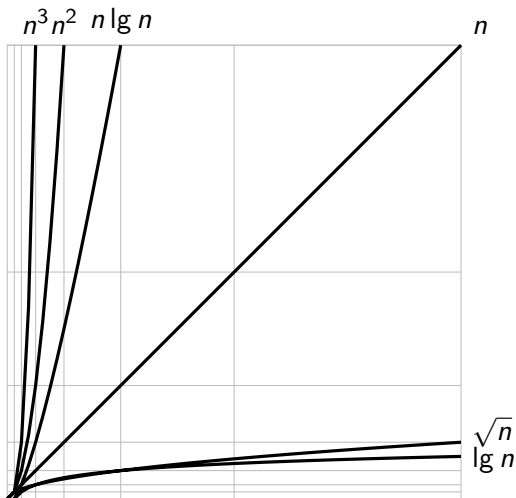
Exponential
$f = O(2^n)$ $f(n+1) \approx 2f(n)$ $f(n+10) \approx 1000f(n)$

Analysis of Algorithms I
Order of growth

Analysis of Algorithms I
Epilogue

You are facing a high wall that stretches infinitely in both directions. There is a door in the wall, but you don't know how far away or in which direction. It is pitch dark, but you have a very dim lighted candle that will enable you to see the door when you are right next to it. Show that there is an algorithm that enables you to find the door by walking at most $O(n)$ steps, where n is the number of steps that you would have taken if you knew where the door is and walked directly to it. What is the constant multiple in the big-O bound for your algorithm?

Lecture 2
Analysis of Algorithms II

Analysis of Algorithms II

Contents

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009

Analysis of Algorithms II
Overview

- Recurrences give us a natural way to characterize the running times of recursive algorithms.
- A recurrence is a function defined in terms of
  - one or more base cases, and
  - itself, with smaller arguments.

Example (Solution: $T(n) = n$.)

- $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + 1 & \text{if } n > 1 \end{cases}$

Example (Solution: $T(n) = n \lg n + n$.)

- $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$

Analysis of Algorithms II
Overview

Example (Solution: $T(n) = \lg \lg n$.)

- $T(n) = \begin{cases} 0 & \text{if } n = 2 \\ T(\sqrt{n}) + 1 & \text{if } n > 2 \end{cases}$

Example (Solution: $\Theta(n \lg n)$.)

- $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/3) + T(2n/3) + n & \text{if } n > 1 \end{cases}$

Analysis of Algorithms II
Overview

Analyzing recursive algorithms

In algorithm analysis, we usually express both the recurrence and its solution using asymptotic notation.

- Example: $T(n) = 2T(n/2) + \Theta(n)$, with solution $T(n) = \Theta(n \lg n)$.

- The boundary conditions are usually expressed as $T(n) = O(1)$ for sufficiently small n.

- When we desire an exact, rather than asymptotic, solution, we need to deal with boundary conditions.

- In practice, we just use asymptotics most of the time, and we ignore boundary conditions.

Analysis of Algorithms II
4.3 The substitution method for solving recurrences

Substitution method

1. Guess the solution.
2. Use induction to find the constants and show that the solution works.

Example

- $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$

1. Guess: $T(n) = n \lg n + n$.

2. Induction:

   Basis: $n = 1$ implies $n \lg n + n = 1 = T(n)$.

   Inductive step: Inductive hypothesis is that $T(k) = k \lg k + k$
   for all $k < n$.

We will use this inductive hypothesis for $T(n/2)$.

Example (Continued)

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n \\
&= 2\left(\frac{n}{2}\lg\frac{n}{2} + \frac{n}{2}\right) + n \qquad \text{(by inductive hypothesis)} \\
&= n\lg\frac{n}{2} + n + n \\
&= n(\lg n - \lg 2) + n + n \\
&= n\lg n - n + n + n \\
&= n\lg n + n
\end{aligned}
$$

Master method

- Used for many recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

where $a \geqslant 1$, $b > 1$, and $f(n) > 0$.

- Describe the running time of an algorithm that divides a problem of size n into a subproblems, each of size n/b, where a and b are positive constants.

- The a subproblems are solved recursively, each in time $T(n/b)$.

- The function $f(n)$ encompasses the cost of dividing the problem and combining the results of the subproblems.

Theorem (Master theorem)

Let $a \geqslant 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n)$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. *If $f(n) = O(n^{\log_b a}/n^{\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.*
2. *If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.*
3. *If $f(n) = \Omega(n^{\log_b a} n^{\epsilon})$ for some constant $\epsilon > 0$, and if $a f(n/b) \leqslant c f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

Master theorem

- In each of the three cases, we compare $f(n)$ with $n^{\log_b a}$, and the larger of the two functions determines the solution to the recurrence.

- Case 1: If $f(n)$ is polynomially smaller than $n^{\log_b a}$, the solution is $T(n) = \Theta(n^{\log_b a})$.

- Case 2: If the two functions are the same size, we multiply by a logarithmic factor, and the solution is $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$.

- Case 3: If $f(n)$ is polynomially larger than $n^{\log_b a}$, the solution is $T(n) = \Theta(f(n))$.

Master theorem

- The three cases do not cover all the possibilities for $f(n)$.
- There is a gap between cases 1 and 2 when $f(n)$ is smaller than $n^{\log_b a}$ but not polynomially smaller.
- There is a gap between cases 2 and 3 when $f(n)$ is larger than $n^{\log_b a}$ but not polynomially larger.
- If the function $f(n)$ falls into one of these gaps, or if the regularity condition in case 3 fails to hold, you cannot use the master method to solve the recurrence.
- However, the regularity condition is satisfied by most of the polynomially bounded functions that we shall encounter.

4.5 The master method for solving recurrences

Example ($T(n) = 8T(n/2) + \Theta(n^2)$)

For this recurrence, we have $a = 8$, $b = 2$, $f(n) = \Theta(n^2)$, and $n^{\log_b a} = n^{\lg 8} = n^3$. Since $f(n)$ is polynomially smaller than n^3, because $n^2 = O(n^{3-\epsilon})$ for $\epsilon = 1$, case 1 applies, and $T(n) = \Theta(n^3)$.

Example ($T(n) = 2T(n/2) + \Theta(n)$)

For this recurrence, we have $a = 2$, $b = 2$, $f(n) = \Theta(n)$, and $n^{\log_b a} = n^{\lg 2} = n$. Since $f(n)$ and n are the same size, case 2 applies, and $T(n) = \Theta(n \lg n)$.

4.5 The master method for solving recurrences

Example ($T(n) = 5T(n/2) + \Theta(n^3)$)

For this recurrence, we have $a = 5$, $b = 2$, $f(n) = \Theta(n^3)$, and $n^{\log_b a} = n^{\lg 5}$. Since $f(n)$ is polynomially larger than $n^{\lg 5}$, because $n^3 = \Omega(n^{\lg 5 + \epsilon})$ for $\epsilon = 3 - \lg 5 > 0$, and the regularity condition holds, because $a f(n/b) = 5(n/2)^3 = 5n^3/8 \leqslant c\, n^3$ for $c = 5/8 < 1$, case 3 applies, and $T(n) = \Theta(n^3)$.

Example ($T(n) = 3T(n/4) + n \lg n$)

For this recurrence, we have $a = 3$, $b = 4$, $f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $n^{\log_4 3}$ is polynomially smaller than $f(n)$, because $n \lg n = \Omega(n^{\log_4 3 + \epsilon})$ for $\epsilon = 1 - \log_4 3 > 0$, and the regularity condition holds, because $a f(n/b) = 3(n/4) \lg(n/4) \leqslant (3/4) n \lg n = c f(n)$ for $c = 3/4 < 1$ and sufficiently large n, case 3 applies, and $T(n) = \Theta(n \lg n)$.

Example ($T(n) = 2T(n/2) + n \lg n$)

For this recurrence, we have $a = 2$, $b = 2$, $f(n) = n \lg n$, and $n^{\log_b a} = n^{\lg 2} = n$.

- $f(n) = n \lg n$ is asymptotically larger than $n^{\log_b a} = n$.
- $f(n) = n \lg n$ is not polynomially larger than $n^{\log_b a} = n$.
- The ratio $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ is asymptotically less than n^ϵ for any constant $\epsilon > 0$.
- The recurrence falls into the gap between case 2 and case 3.

Theorem (Master theorem, simplified)

Let $a \geqslant 1$ and $b > 1$ be constants, let $k \geqslant 0$ be a constant, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + n^k$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. *If $k < \log_b a$, then $T(n) = \Theta(n^{\log_b a})$.*
2. *If $k = \log_b a$, then $T(n) = \Theta(n^k \lg n)$.*
3. *If $k > \log_b a$, then $T(n) = \Theta(n^k)$.*

Theorem (Master theorem, simplified)

Let $a \geqslant 1$ and $b > 1$ be constants, let $k \geqslant 0$ be a constant, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = a\,T(n/b) + n^k$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. *If $b^k < a$, then $T(n) = \Theta(n^{\log_b a})$.*
2. *If $b^k = a$, then $T(n) = \Theta(n^k \lg n)$.*
3. *If $b^k > a$, then $T(n) = \Theta(n^k)$.*

Analysis of Algorithms II
4.5 The master method for solving recurrences

Example ($T(n) = 8T(n/2) + \Theta(n^2)$)

For this recurrence, we have $a = 8$, $b = 2$, $f(n) = \Theta(n^2)$, and $n^{\log_b a} = n^{\lg 8} = n^3$. Since $f(n)$ is polynomially smaller than n^3, because $n^2 = O(n^{3-\epsilon})$ for $\epsilon = 1$, case 1 applies, and $T(n) = \Theta(n^3)$.

Since $k = 2 < 3 = \lg 8 = \log_b a$

Since $b^k = 2^2 < 8 = a$

Analysis of Algorithms II
4.5 The master method for solving recurrences

Example ($T(n) = 2T(n/2) + \Theta(n)$)

For this recurrence, we have $a = 2$, $b = 2$, $f(n) = \Theta(n)$, and $n^{\log_b a} = n^{\lg 2} = n$. Since $f(n)$ and n are the same size, case 2 applies, and $T(n) = \Theta(n \lg n)$.

Since $k = 1 = \lg 2 = \log_b a$

Since $b^k = 2^1 = 2 = a$

Example ($T(n) = 5T(n/2) + \Theta(n^3)$)

For this recurrence, we have $a = 5$, $b = 2$, $f(n) = \Theta(n^3)$, and $n^{\log_b a} = n^{\lg 5}$. Since $f(n)$ is polynomially larger than $n^{\lg 5}$, because $n^3 = \Omega(n^{\lg 5 + \epsilon})$ for $\epsilon = 3 - \lg 5 > 0$, and the regularity condition holds, because $a\, f(n/b) = 5(n/2)^3 = 5n^3/8 \leqslant c\, n^3$ for $c = 5/8 < 1$, case 3 applies, and $T(n) = \Theta(n^3)$.

Since $k = 3 > \lg 5 = \log_b a$

Since $b^k = 2^3 > 5 = a$

Lecture 3
Divide and Conquer I

Divide and Conquer I

Contents

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009

Divide and Conquer I
2.3.1 The divide-and-conquer approach

Divide and conquer

A common approach to design algorithms.

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively.

Base case: If the problems are small enough, just solve them by brute force.

Combine the subproblem solutions to give a solution to the original problem.

Divide and Conquer I
2.3.1 The divide-and-conquer approach

Merge sort

- A sorting algorithm based on divide and conquer.
- We state each subproblem as sorting a subarray $A[p..r]$.
- Initially, $p = 1$ and $r = n$, but these values change as we recurse through subproblems.

To sort $A[p..r]$:

Divide by splitting into two subarrays $A[p..q]$ and $A[q + 1..r]$, where q is the halfway point of $A[p..r]$.

Conquer by recursively sorting the two subarrays $A[p..q]$ and $A[q + 1..r]$.

Combine by merging the two sorted subarrays $A[p..q]$ and $A[q + 1..r]$ to produce a single sorted subarray $A[p..r]$.

Divide and Conquer I
2.3.1 The divide-and-conquer approach

Merge sort

```
MERGE-SORT(A, p, r)
    if p < r then                          check for base case
        q = ⌊(p + r)/2⌋                            divide
        MERGE-SORT(A, p, q)                        conquer
        MERGE-SORT(A, q + 1, r)                    conquer
        MERGE(A, p, q, r)                          combine
```

The initial call is MERGE-SORT($A, 1, n$).

Divide and Conquer I
2.3.1 The divide-and-conquer approach

Merge sort

Example (Bottom-up view for $n = 8$)
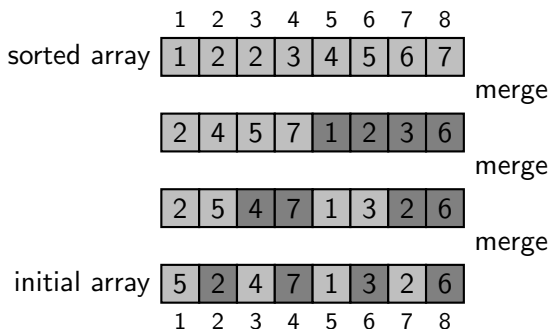
Merge sort

```
MERGE(A, p, q, r)
    n₁ = q − p + 1                    i = 1
    n₂ = r − q                        j = 1
    L[1..n₁ + 1] = new array         for k = p to r do
    R[1..n₂ + 1] = new array             if L[i] ⩽ R[j] then
    for i = 1 to n₁ do                       A[k] = L[i]
        L[i] = A[p + i − 1]                  i = i + 1
    for j = 1 to n₂ do               else
        R[j] = A[q + j]                      A[k] = R[j]
    L[n₁ + 1] = ∞                            j = j + 1
    R[n₂ + 1] = ∞
```

Merge sort

- Merging takes $\Theta(n)$ time, where $n = r - p + 1$ is the number of elements being merged.
    - The first two **for** loops take $\Theta(n_1 + n_2) = \Theta(n)$ time.
    - The last **for** loop makes n iterations, each taking constant time, for $\Theta(n)$ time.
- We use a special sentinel value, ∞, to simplify the code.
- We know in advance that there are exactly $r - p + 1$ nonsentinel elements. We can stop once we have performed $r - p + 1$ basic steps. Never a need to check for sentinels, since they are always greater than any element.
- Rather than even counting basic steps, just fill up the output array from index p up through and including index r.

Overview

We use a recurrence to describe the running time of a divide-and-conquer algorithm. Let $T(n)$ be the running time on a problem of size n.

- If the problem size is small enough (say, $n \leqslant c$ for some constant c), we have a base case. The brute-force solution takes constant time.

- Otherwise, suppose that we divide into a subproblems, each $1/b$ the size of the original. (In merge sort, $a = b = 2$.)

- Let the time to divide a problem of size n be $D(n)$.

- We have a subproblems to solve, each of size n/b. Each subproblem takes $T(n/b)$ time to solve. We spend $aT(n/b)$ time solving subproblems.

- Let the time to combine solutions be $C(n)$.

Divide and Conquer I
2.3.2 Analyzing divide-and-conquer algorithms

Overview

- We get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leqslant c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Analyzing merge sort

- For simplicity, assume that n is a power of 2. Each divide step yields two subproblems, both of size exactly $n/2$.

- The base case occurs when $n = 1$. When $n \geqslant 2$, we have

  Divide Just compute q as the average of p and r.
  Conquer Recursively solve 2 subproblems, each of size $n/2$.
  Combine Merging n elements takes $\Theta(n)$ time.

- The recurrence for merge sort running time is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2\,T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- By the master theorem, we can show that this recurrence has the solution $T(n) = \Theta(n \lg n)$.

Divide and Conquer I
2.3.1 The divide-and-conquer approach

Exercise 2.3-5

- Consider the searching problem: Given a sequence A of n numbers and a value v, output an index i such that $v = A[i]$ or the special value NIL if v does not appear in A.
- Observe that if the sequence A is sorted, we can check the midpoint of the sequence against v and eliminate half of the sequence from further consideration.
- The binary search algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time.
- Write pseudocode, either iterative or recursive, for binary search.
- Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

Solution to Exercise 2.3-5

- BINARY-SEARCH takes a sorted array A, a value v, and a range [$low..high$] of the array, in which we search for the value v.

- The procedure compares v to the array entry at the midpoint of the range and decides to eliminate half the range from further consideration.

- We give both iterative and recursive versions, each of which returns either an index i such that $A[i] = v$, or NIL, if no entry of $A[low..high]$ contains the value v.

- An initial call to either version should have the parameters A, v, 1, n.

Solution to Exercise 2.3-5

```
BINARY-SEARCH(A, v, low, high)
    while low ⩽ high do
        mid = ⌊(low + high)/2⌋
        if v = A[mid] then
            return mid
        else if v > A[mid] then
            low = mid + 1
        else
            high = mid − 1
    return NIL
```

Divide and Conquer I
2.3.1 The divide-and-conquer approach

Solution to Exercise 2.3-5

```
BINARY-SEARCH(A, v, low, high)
    if low > high then
        return NIL
    mid = ⌊(low + high)/2⌋
    if v = A[mid] then
        return mid
    else if v > A[mid] then
        return BINARY-SEARCH(A, v, mid + 1, high)
    else
        return BINARY-SEARCH(A, v, low, mid − 1)
```

Divide and Conquer I
2.3.1 The divide-and-conquer approach

Solution to Exercise 2.3-5

- An initial call to either version should have the parameters A, v, 1, n.
- Both procedures terminate the search unsuccessfully when the range is empty (when $low > high$) and terminate it successfully if the value v has been found.
- Based on the comparison of v to the middle element in the searched range, the search continues with the range halved.
- The recurrence for these procedures is therefore $T(n) = T(n/2) + \Theta(1)$, whose solution is $T(n) = \Theta(\lg n)$.

Divide and Conquer I
7.1 Description of quicksort

- Quicksort is based on the three-step process of divide-and-conquer. To sort the subarray $A[p..r]$:

Divide: Partition $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$, such that each element in the first subarray $A[p..q-1]$ is $\leqslant A[q]$ and $A[q]$ is $\leqslant$ each element in the second subarray $A[q+1..r]$.

Conquer: Sort the two subarrays by recursive calls to QUICKSORT.

Combine: No work is needed to combine the subarrays, because they are sorted in place.

- Perform the divide step by a procedure PARTITION, which returns the index q that marks the position separating the subarrays.

```
QUICKSORT(A, p, r)
    if p < r then
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
```

- The initial call is QUICKSORT($A, 1, n$).

Divide and Conquer I
7.1 Description of quicksort

Hoare partitioning (Problem 7-1)

```
PARTITION(A, p, r)
    x = A[p]                          if i < j then
    i = p − 1                             exchange A[i] with A[j]
    j = r + 1                         else
    while TRUE do                         return j
        repeat
            j = j − 1
        until A[j] ⩽ x
        repeat
            i = i + 1
        until A[i] ⩾ x
```
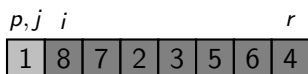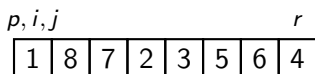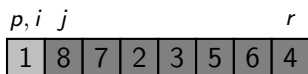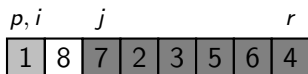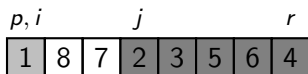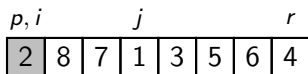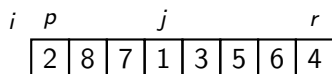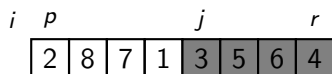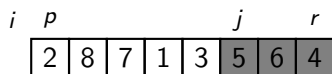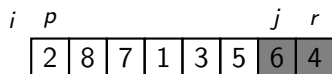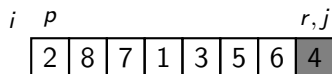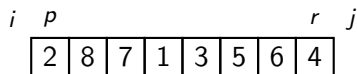
Divide and Conquer I
7.1 Description of quicksort

- Hoare PARTITION always selects the first element $A[p]$ in the subarray $A[p..r]$ as the pivot, the element around which to partition.

- As the procedure executes, the array is partitioned into three regions, some of which may be empty:
  - All entries in $A[p..i]$ are $\leqslant$ pivot.
  - All entries in $A[j..r]$ are $\geqslant$ pivot.
  - No entries in $A[i+1..j-1]$ have yet been examined, and so we do not know how they compare to the pivot.
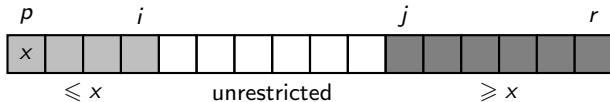
Correctness of Hoare partitioning

Use the loop invariant to prove correctness of PARTITION:

- Before the loop starts, all the conditions of the loop invariant are satisfied, because $A[p]$ is the pivot and the subarrays $A[p..i]$ and $A[j..r]$ are empty.

- While the loop is running, index i is incremented and index j is decremented until $A[i] \geqslant$ pivot $\geqslant A[j]$. If these inequalities are strict, $A[i]$ is too large to belong to the first region and $A[j]$ is too small to belong to the second region. Thus, by swapping $A[i]$ and $A[j]$, we can extend the two regions.

- When the **while** loop terminates, all elements in A are partitioned into two subarrays $A[p..q]$ and $A[q+1..r]$, where $p \leqslant q < r$, such that $A[p..q] \leqslant A[q+1..r]$, and the value $q = j$ is returned.

Divide and Conquer I
7.1 Description of quicksort

Lomuto partitioning

```
PARTITION(A, p, r)
    x = A[r]
    i = p − 1
    for j = p to r − 1 do
        if A[j] ⩽ x then
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    return i + 1
```

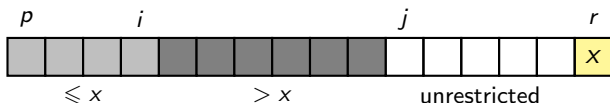# Divide and Conquer I
7.1 Description of quicksort
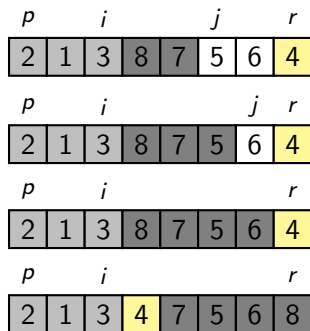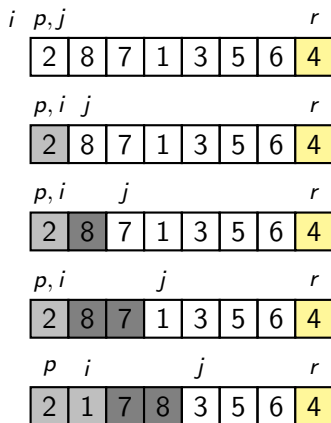
- Lomuto PARTITION always selects the last element $A[r]$ in the subarray $A[p..r]$ as the pivot, the element around which to partition.
- As the procedure executes, the array is partitioned into four regions, some of which may be empty:
  - All entries in $A[p..i]$ are $\leqslant$ pivot.
  - All entries in $A[i+1..j-1]$ are $>$ pivot.
  - No entries in $A[j..r-1]$ have yet been examined, and so we do not know how they compare to the pivot.
  - $A[r] =$ pivot.

Correctness of Lomuto partitioning

Use the loop invariant to prove correctness of PARTITION:

- Before the loop starts, all the conditions of the loop invariant are satisfied, because $A[r]$ is the pivot and the subarrays $A[p..i]$ and $A[i+1..j-1]$ are empty.

- While the loop is running, if $A[j] \leqslant$ pivot, then $A[j]$ and $A[i+1]$ are swapped and then i and j are incremented. If $A[j] >$ pivot, then only j is incremented.

- When the loop terminates, $j = r$, so all elements in A are partitioned into one of the three cases: $A[p..i] \leqslant$ pivot, $A[i+1..r-1] >$ pivot, and $A[r] =$ pivot.

The last two lines of PARTITION move the pivot element from the end of the array to between the two subarrays.

Worst case

- Occurs when the subarrays are completely unbalanced every time.
- Have 0 elements in one subarray and $n - 1$ elements in the other subarray.
- Since the recursive call on an array of size 0 just returns, $T(0) = \Theta(1)$, and the recurrence for the running time is

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

- Using the substitution method, this recurrence has the solution $T(n) = \Theta(n^2)$.

Best case

- Occurs when the subarrays are completely balanced every time.

- Each subarray has at most $n/2$ elements, since one is of size $\lfloor n/2 \rfloor$ and the other is of size $\lceil n/2 \rceil - 1$.

- The recurrence for the running time is then

$$T(n) = 2T(n/2) + \Theta(n)$$

- By case 2 of the master theorem, this recurrence has the solution $T(n) = \Theta(n \lg n)$.

Divide and Conquer I
8.1 Lower bounds for sorting

Lower bounds

- $\Omega(n)$ to examine all the input.
- All sorting algorithms seen so far (bubble sort, selection sort, insertion sort, merge sort, quicksort) are comparison sort algorithms and are $\Omega(n \lg n)$.
- We will show that $\Omega(n \lg n)$ is a lower bound for comparison sort algorithms.

Decision tree

- Abstraction of any comparison sort.
- Represents comparisons made by a specific sorting algorithm on inputs of a given size. We are counting only comparisons.
- Abstracts away everything else: control and data movement.

Divide and Conquer I
8.1 Lower bounds for sorting

Example (Decision tree for insertion sort on 3 elements)
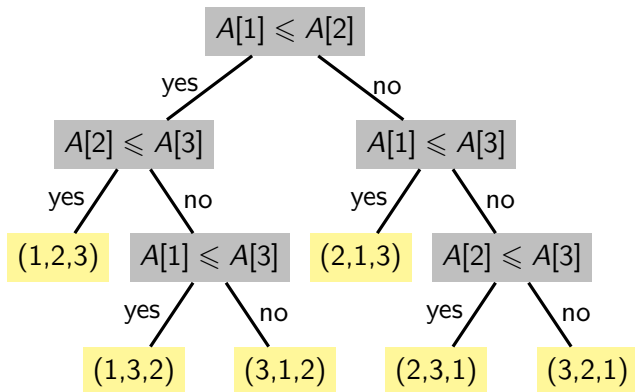
Divide and Conquer I
8.1 Lower bounds for sorting

Decision tree

- How many leaves on the decision tree? There are at least $n!$ leaves, because every permutation must appear at least once.

- For any comparison sort, there is one tree for each n. View the tree as if the algorithm splits in two at each node, based on the information it has determined up to that point.

- The tree models all possible execution traces.

- What is the length of the longest path from root to leaf? Depends on the algorithm. $\Theta(n^2)$ for insertion sort. $\Theta(n \lg n)$ for merge sort.

Divide and Conquer I
8.1 Lower bounds for sorting

Lemma
Any binary tree of height h has at most 2^h leaves.

Theorem
Any decision tree that sorts n elements has height $\Omega(n \lg n)$.

Corollary
Merge sort is an asymptotically optimal comparison sort.

Divide and Conquer I
8.1 Lower bounds for sorting

Lemma
Any binary tree of height h has at most 2^h leaves.

Proof.
By induction on h.

Basis: $h = 0$. The tree is just a node, which is a leaf. $2^h = 1$.

Inductive step: Assume true for height $h - 1$. Extend tree of height $h - 1$ by making as many new leaves as possible. Each leaf becomes parent to two new leaves. Let $L(h)$ be the maximum number of leaves in a binary tree of height h. Then:

$$L(h) = 2 \cdot L(h - 1) = 2 \cdot 2^{h-1} = 2^h$$

Divide and Conquer I
8.1 Lower bounds for sorting

Theorem
Any decision tree that sorts n elements has height $\Omega(n \lg n)$.

Proof.
Consider a decision tree of height h with ℓ leaves corresponding to a comparison sort on n elements.

- Because each of the $n!$ permutations of the input appears as some leaf, we have $n! \leqslant \ell$.
- By the lemma, we have $n! \leqslant \ell \leqslant 2^h$.
- By taking logarithms, $h \geqslant \lg(n!)$
- Using Stirling's approximation, $n! > (n/e)^n$. Then:

$$h > \lg(n/e)^n = n \lg(n/e) = n \lg n - n \lg e = \Omega(n \lg n)$$

Corollary

Merge sort is an asymptotically optimal comparison sort.

Proof.

- The length of the longest simple path from the root of a decision tree to any of its leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs.

- The worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree.

- The $O(n \lg n)$ upper bound on the running time for merge sort matches the $\Omega(n \lg n)$ worst-case lower bound from the theorem.

Lecture 4
Divide and Conquer II

Divide and Conquer II

Contents

- Karatsuba's algorithm

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009

Divide and Conquer II
Karatsuba's algorithm

Integer multiplication

- If $x = (x_{n-1}, \ldots, x_1, x_0)$ and $y = (y_{n-1}, \ldots, y_1, y_0)$ are long n-bit integers, then

$$x = \sum_{i=0}^{n-1} x_i 2^i, \qquad y = \sum_{j=0}^{n-1} y_j 2^j, \qquad x \cdot y = \sum_{i=0}^{n-1} x_i 2^i \sum_{j=0}^{n-1} y_j 2^j$$

Analysis

- Two nested loops, each iterate n times, and innermost loop body takes constant time: $\Theta(n^2)$ time.

Divide and Conquer II
Karatsuba's algorithm

Simple divide-and-conquer method

```
MULT(x, y, n)
    if n = 1 then
        return x y
    else
        partition x into a, b such that x = a 2^{n/2} + b
        partition y into c, d such that y = c 2^{n/2} + d
        ac = MULT(a, c, n/2)
        ad = MULT(a, d, n/2)
        bc = MULT(b, c, n/2)
        bd = MULT(b, d, n/2)
        return ac 2^n + (ad + bc) 2^{n/2} + bd
```

Divide and Conquer II
Karatsuba's algorithm

Analysis

- Because multiplying by 2^n and $2^{n/2}$ is equivalent to shifting,

$$x\,y = (a\,2^{n/2} + b)\,(c\,2^{n/2} + d) = a\,c\,2^n + (a\,d + b\,c)\,2^{n/2} + b\,d$$

- Recurrence is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leqslant 1 \\ 4\,T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- By the master method, solution is $T(n) = \Theta(n^2)$.
- Asymptotically, no better than the obvious method.

Divide and Conquer II
Karatsuba's algorithm

Karatsuba's method

```
MULT(x, y, n)
    if n = 1 then
        return x y
    else
        partition x into a, b such that x = a 2^{n/2} + b
        partition y into c, d such that y = c 2^{n/2} + d
        ac = MULT(a, c, n/2)
        bd = MULT(b, d, n/2)
        abcd = MULT(a + b, c + d, n/2)
        return ac 2^n + (abcd - ac - bd) 2^{n/2} + bd
```

Divide and Conquer II
Karatsuba's algorithm

Analysis

- Because multiplying by 2^n and $2^{n/2}$ is equivalent to shifting,

$$x\,y = a\,c\,2^n + ((a+b)\,(c+d) - a\,c - b\,d)\,2^{n/2} + b\,d$$

- Recurrence is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leqslant 1 \\ 3\,T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- By the master method, solution is $T(n) = \Theta(n^{\lg 3})$.

Divide and Conquer II
Karatsuba's algorithm
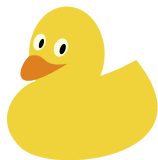
Anatoly
Karatsuba
(1937–2008)

Yuri Ofman
(1939–)
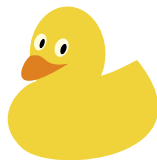
Andrey
Kolmogorov
(1903–1987)

Divide and Conquer II
4.2 Strassen's algorithm for matrix multiplication

Matrix multiplication

- If $A = (a_{ij})$ and $B = (b_{ij})$ are square $n \times n$ matrices, then in the product $C = A \cdot B$, we define the entry c_{ij} by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

for $i, j = 1, \ldots, n$.

- Need to compute n^2 entries of C.
- Each entry is the sum of n values.

Divide and Conquer II
4.2 Strassen's algorithm for matrix multiplication

Obvious method

```
MULT(A, B, n)
    let C be a new n × n matrix
    for i = 1 to n do
        for j = 1 to n do
            c_ij = 0
            for k = 1 to n do
                c_ij = c_ij + a_ik · b_kj
    return C
```

Analysis

- Three nested loops, each iterates n times, and innermost loop body takes constant time: $\Theta(n^3)$ time.

Divide and Conquer II
4.2 Strassen's algorithm for matrix multiplication

Is that the best we can do?

- Seems like any algorithm to multiply matrices must take $\Omega(n^3)$ time:
  - Must compute n^2 entries.
  - Each entry is the sum of n terms.
- But with Strassen's method, we can multiply matrices in $o(n^3)$ time.
  - Strassen's algorithm runs in $\Theta(n^{\lg 7})$ time.
  - $2.80 \leqslant \lg 7 \leqslant 2.81$
  - Hence, runs in $O(n^{2.81})$ time.

Simple divide-and-conquer method

- Assume that n is a power of 2.
- Partition each of A, B, and C into four $n/2 \times n/2$ matrices:

$$A = \left(\begin{array}{cc} A_{11} & A_{12} \\ A_{21} & A_{22} \end{array} \right) B = \left(\begin{array}{cc} B_{11} & B_{12} \\ B_{21} & B_{22} \end{array} \right) C = \left(\begin{array}{cc} C_{11} & C_{12} \\ C_{21} & C_{22} \end{array} \right)$$

- Rewrite $C = A \cdot B$ as

$$\left(\begin{array}{cc} C_{11} & C_{12} \\ C_{21} & C_{22} \end{array} \right) = \left(\begin{array}{cc} A_{11} & A_{12} \\ A_{21} & A_{22} \end{array} \right) \cdot \left(\begin{array}{cc} B_{11} & B_{12} \\ B_{21} & B_{22} \end{array} \right)$$

Divide and Conquer II
4.2 Strassen's algorithm for matrix multiplication

Simple divide-and-conquer method

- This corresponds to the four equations
  - $C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$
  - $C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$
  - $C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$
  - $C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$
- Each of these four equations multiplies two $n/2 \times n/2$ matrices and then adds their $n/2 \times n/2$ products.
- Use these equations to get a divide-and-conquer algorithm.

Divide and Conquer II
4.2 Strassen's algorithm for matrix multiplication

Simple divide-and-conquer method

```
MULT(A, B, n)
    let C be a new n × n matrix
    if n = 1 then
        c₁₁ = a₁₁ · b₁₁
    else
        Partition A, B, and C into four n/2 × n/2 matrices
        C₁₁ = MULT(A₁₁, B₁₁, n/2) + MULT(A₁₂, B₂₁, n/2)
        C₁₂ = MULT(A₁₁, B₁₂, n/2) + MULT(A₁₂, B₂₂, n/2)
        C₂₁ = MULT(A₂₁, B₁₁, n/2) + MULT(A₂₂, B₂₁, n/2)
        C₂₂ = MULT(A₂₁, B₁₂, n/2) + MULT(A₂₂, B₂₂, n/2)
    return C
```

Divide and Conquer II
4.2 Strassen's algorithm for matrix multiplication

Simple divide-and-conquer method

```
MULT(A, B, n)
    let C be a new n × n matrix
    if n = 1 then
        c_{11} = a_{11} · b_{11}
    else
        Partition A, B, and C into four n/2 × n/2 matrices
        C_{11} = MULT(A_{11}, B_{11}, n/2) + MULT(A_{12}, B_{21}, n/2)
        C_{12} = MULT(A_{11}, B_{12}, n/2) + MULT(A_{12}, B_{22}, n/2)
        C_{21} = MULT(A_{21}, B_{11}, n/2) + MULT(A_{22}, B_{21}, n/2)
        C_{22} = MULT(A_{21}, B_{12}, n/2) + MULT(A_{22}, B_{22}, n/2)
    return C
```

Analysis

- Let $T(n)$ be the time to multiply two $n \times n$ matrices.

  Base case: When $n = 1$, perform one scalar multiplication, $T(1) = \Theta(1)$.

  Recursive case: When $n > 1$,

    - Dividing takes $\Theta(1)$ time, using index calculations.
    - Conquering makes 8 recursive calls, each multiplying $n/2 \times n/2$ matrices, contributing $8T(n/2)$ time.
    - Combining takes $\Theta(n^2)$ time to add $n/2 \times n/2$ matrices four times.

Analysis

- Recurrence is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

- By the master method, solution is $T(n) = \Theta(n^3)$.
- Asymptotically, no better than the obvious method.

Divide and Conquer II
4.2 Strassen's algorithm for matrix multiplication

Strassen's method

- Perform only 7 recursive multiplications of $n/2 \times n/2$ matrices, rather than 8.
- Will cost several additions of $n/2 \times n/2$ matrices, but just a constant number more.

Analysis

- Recurrence is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

- By the master method, solution is $T(n) = \Theta(n^{\lg 7})$.

Divide and Conquer II
4.2 Strassen's algorithm for matrix multiplication

Strassen's method

- Assume that n is a power of 2.
- Partition each of A, B, and C into four $n/2 \times n/2$ matrices:

$$A = \left(\begin{array}{cc} A_{11} & A_{12} \\ A_{21} & A_{22} \end{array} \right) B = \left(\begin{array}{cc} B_{11} & B_{12} \\ B_{21} & B_{22} \end{array} \right) C = \left(\begin{array}{cc} C_{11} & C_{12} \\ C_{21} & C_{22} \end{array} \right)$$

- Rewrite $C = A \cdot B$ as

$$\left(\begin{array}{cc} C_{11} & C_{12} \\ C_{21} & C_{22} \end{array} \right) = \left(\begin{array}{cc} A_{11} & A_{12} \\ A_{21} & A_{22} \end{array} \right) \cdot \left(\begin{array}{cc} B_{11} & B_{12} \\ B_{21} & B_{22} \end{array} \right) =$$

$$= \left(\begin{array}{cc} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_5 + P_1 - P_3 - P_7 \end{array} \right)$$

Divide and Conquer II
4.2 Strassen's algorithm for matrix multiplication

Strassen's method

- Compute 10 sums or differences of $n/2 \times n/2$ matrices.

$B_{12} - B_{22}$	$A_{11} + A_{12}$	$A_{21} + A_{22}$	$B_{21} - B_{11}$	$A_{11} + A_{22}$
$B_{11} + B_{22}$	$A_{12} - A_{22}$	$B_{21} + B_{22}$	$A_{11} - A_{21}$	$B_{11} + B_{12}$

- Recursively compute 7 products of $n/2 \times n/2$ matrices.

$$P_1 = A_{11}\,(B_{12} - B_{22})$$
$$P_2 = (A_{11} + A_{12})\,B_{22}$$
$$P_3 = (A_{21} + A_{22})\,B_{11}$$
$$P_4 = A_{22}\,(B_{21} - B_{11})$$
$$P_5 = (A_{11} + A_{22})\,(B_{11} + B_{22})$$
$$P_6 = (A_{12} - A_{22})\,(B_{21} + B_{22})$$
$$P_7 = (A_{11} - A_{21})\,(B_{11} + B_{12})$$

Strassen's method

- $P_5 + P_4 - P_2 + P_6 = \cdots = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$
- $P_1 + P_2 = \cdots = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$
- $P_3 + P_4 = \cdots = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$
- $P_5 + P_1 - P_3 - P_7 = \cdots = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$

$$
\begin{aligned}
P_1 + P_2 &= A_{11}\left(B_{12} - B_{22}\right) + \left(A_{11} + A_{12}\right) B_{22} \\
&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} + A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\
&= A_{11} \cdot B_{12} + A_{12} \cdot B_{22}
\end{aligned}
$$

Divide and Conquer II
4.2 Strassen's algorithm for matrix multiplication

Strassen's method

```
MULT(A, B, n)
    let C be a new n × n matrix
    if n = 1 then
        c₁₁ = a₁₁ · b₁₁
    else
        Partition A, B, and C into four n/2 × n/2 matrices
        S₁ = B₁₂ − B₂₂
        . . .
        P₁ = MULT(A₁₁, S₁, n/2)
        . . .
        C₁₁ = P₅ + P₄ − P₂ + P₆
        . . .
    return C
```
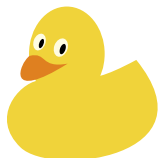
Volker Strassen
(1936–)
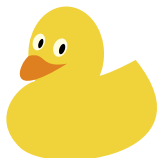
Don
Coppersmith
(1950–)
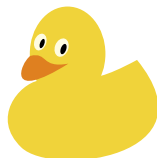
Shmuel
Winograd
(1936–2019)

Divide and Conquer II
9.1 Minimum and maximum

Selection problem

- Given a set A of n (distinct) numbers and an integer i, with $1 \leqslant i \leqslant n$, find the element $x \in A$ that is larger than exactly $i-1$ other elements in A (the *i*th smallest element of A).
- We can solve the selection problem in $O(n \lg n)$ time
  - Sort the numbers using an $O(n \lg n)$ time algorithm, such as merge sort.
  - Return the *i*th element in the sorted array.
- The minimum is the *i*th smallest element for $i = 1$.
- The maximum is the *i*th smallest element for $i = n$.
- The (lower) median is the *i*th smallest element for $i = n/2$.

Divide and Conquer II
9.1 Minimum and maximum

- We can easily obtain an upper bound of $n - 1$ comparisons for finding the minimum of a set of n elements.
  - Examine each element in turn and keep track of the smallest one.
  - This is the best we can do, because each element, except the minimum, must be compared to a smaller element at least once.

```
MINIMUM(A, n)
    min = A[1]
    for i = 2 to n do
        if min > A[i] then
            min = A[i]
    return min
```

- The maximum can be found in exactly the same way.

Divide and Conquer II
9.1 Minimum and maximum

Simultaneous minimum and maximum

- A simple algorithm to find both the minimum and the maximum is to find each one independently.
  - There will be $n - 1$ comparisons for the minimum and $n - 2$ comparisons for the maximum, for a total of $2n - 2$ comparisons.
  - This will result in $\Theta(n)$ time, which is asymptotically optimal.

Divide and Conquer II
9.1 Minimum and maximum

Simultaneous minimum and maximum

- In fact, at most $3\lfloor n/2 \rfloor$ comparisons suffice to find both the minimum and the maximum.
  - Maintain both the minimum and maximum elements seen thus far.
  - Do not compare each element to the minimum and maximum separately, but process elements in pairs.
  - Compare the elements of a pair first to each other, then compare the smaller to the minimum so far and the larger to the maximum so far.
- This leads to only 3 comparisons for every 2 elements.

Simultaneous minimum and maximum

- Setting up the initial values for the minimum and maximum depends on whether n is odd or even.
  - If n is odd, set both the minimum and maximum to the value of the first element, then process the rest of the elements in pairs.
  - If n is even, compare the first two elements to determine the initial values of the minimum and maximum, then process the rest of the elements in pairs.

Simultaneous minimum and maximum

- Analysis of the total number of comparisons:
  - If n is odd, we perform $3(n-1)/2 = 3\lfloor n/2 \rfloor$ comparisons.
  - If n is even, we perform 1 initial comparison followed by $3(n-2)/2$ comparisons, for a total of $3n/2 - 2$.

$$\frac{3(n-2)}{2} + 1 = \frac{3n-6}{2} + 1 = \frac{3n}{2} - 3 + 1 = \frac{3n}{2} - 2$$

- In either case, the total number of comparisons is at most $3\lfloor n/2 \rfloor$.

Divide and Conquer II
9.3 Selection in worst-case linear time

- We can find the ith smallest element in $O(n)$ time in the worst case.
- The algorithm SELECT finds the desired element by recursively partitioning the input array.
- We guarantee a good split upon partitioning the array.
- We use the deterministic partitioning algorithm from quicksort, but modified to take the element to partition around as an input parameter.

Lomuto partitioning

```
PARTITION(A, p, r, x)
    i = 1
    while A[i] ≠ x do
        i = i + 1
    exchange A[i] with A[r]
    i = p - 1
    for j = p to r - 1 do
        if A[j] ⩽ x then
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    return i + 1
```

SELECT works on an array of $n > 1$ elements. It executes the following steps:

1. Divide the n elements into groups of 5. Get $\lceil n/5 \rceil$ groups: $\lfloor n/5 \rfloor$ groups with exactly 5 elements and, if 5 does not divide n, one group with the remaining $n \bmod 5$ elements.

2. Find the median of each of the $\lceil n/5 \rceil$ groups:
   - Run insertion sort on each group. Takes $O(1)$ time per group since each group has at most 5 elements.
   - Then just pick the median from each group, in $O(1)$ time.

3. Find the median x of the $\lceil n/5 \rceil$ medians by a recursive call to SELECT. (If $\lceil n/5 \rceil$ is even, then by our convention, x is the lower median.)

4. Using the modified version of PARTITION that takes the pivot element as input, partition the input array around x. Let x be the kth smallest element of the array after partitioning, so that there are $k - 1$ elements on the low side of the partition and $n - k$ elements on the high side.

5. Now there are three possibilities:

   - If $i = k$, just return x.
   - Uf $i < k$, return the ith smallest element on the low side of the partition by making a recursive call to SELECT.
   - Uf $i > k$, return the $(i - k)$th smallest element on the high side of the partition by making a recursive call to SELECT.

```
SELECT(A, p, r, i)
    divide A into n/5 groups of 5 elements each
    M = median of each of the n/5 groups
    x = SELECT(M, 1, n/5, n/10)
    k = PARTITION(A, p, r, x)
    if i = k then
        return x
    else
        if i < k then
            return SELECT(A, p, k − 1, i)
        else
            return SELECT(A, k + 1, r, i − k)
```

Analysis

- Start by getting a lower bound on the number of elements that are greater than the partitioning element x.
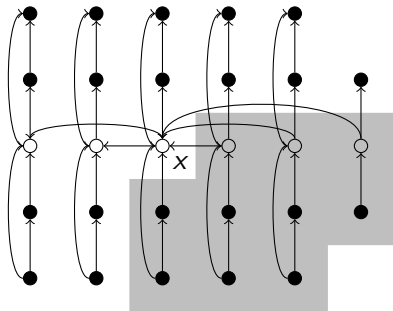
- Each group is a column.
- Each white circle is the median of a group.
- Arrows go from larger elements to smaller elements.
- Elements in the region on the lower right are known to be greater than x.

Analysis

- At least half of the medians found in step 2 are $\geqslant x$.
- Look at the groups containing these medians that are $\geqslant x$.
  All of them contribute 3 elements that are $> x$ (the median of
  the group and the 2 elements in the group greater than the
  group's median), except for 2 of the groups:
    - The group containing x, which has only 2 elements $> x$.
    - The group with < 5 elements.
- Forget about these 2 groups. That leaves at least

$$\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2$$

groups with 3 elements known to be $> x$.

Analysis

- Thus, we know that at least

$$3 \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right) \geqslant \frac{3n}{10} - 6$$

  elements are $> x$.

- Symmetrically, the number of elements that are $< x$ is at least $3n/10 - 6$.

- Therefore, when we call SELECT recursively in step 5, it is on at most $7n/10 + 6$ elements.

Divide and Conquer II
9.3 Selection in worst-case linear time

Analysis

- Develop a recurrence for the worst-case running time $T(n)$ of SELECT:
    - Steps 1, 2, and 4 each take $O(n)$ time.
        - Making groups of 5 elements.
        - Sorting $\lceil n/5 \rceil$ groups in $O(1)$ time each.
        - Partitioning the n-element array around x.
    - Step 3 takes time $T(\lceil n/5 \rceil)$.
    - Step 5 takes time at most $T(7n/10 + 6)$, assuming that T is monotonically increasing.
- Assume that $T(n) = O(1)$ for small enough n. We will use $n < 140$ as small enough. Why 140? We will see why later.

Analysis

- We get the recurrence

$$
T(n) \leqslant
\begin{cases}
O(1) & \text{if } n < 140 \\
T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{if } n \geqslant 140
\end{cases}
$$

- We solve this recurrence by substitution. Inductive hypothesis is that $T(n) \leqslant cn$ for some constant c and all $n > 0$.

- Assume that c is large enough that $T(n) \leqslant cn$ for all $n < 140$. So we are concerned only with the case $n \geqslant 140$.

- Pick a constant a such that the function described by the $O(n)$ term in the recurrence is $\leqslant an$ for all $n > 0$.

Divide and Conquer II
9.3 Selection in worst-case linear time

Analysis

- Substitute the inductive hypothesis in the right-hand side of the recurrence:

$$
\begin{aligned}
T(n) &\leqslant c\lceil n/5\rceil + c(7n/10 + 6) + an \\
&\leqslant cn/5 + c + 7cn/10 + 6c + an \\
&= 9cn/10 + 7c + an \\
&= cn + (-cn/10 + 7c + an)
\end{aligned}
$$

- This last quantity is $\leqslant cn$ if

$$
\begin{aligned}
-cn/10 + 7c + an &\leqslant 0 \\
cn/10 - 7c &\geqslant an \\
c(n - 70) &\geqslant 10an \\
c &\geqslant 10a(n/(n - 70))
\end{aligned}
$$

Analysis

- Because we assumed $n \geqslant 140$, we have $n/(n - 70) \leqslant 2$.

- Thus, $20a \geqslant 10a(n/(n - 70))$, so choosing $c \geqslant 20a$ gives $c \geqslant 10a(n/(n - 70))$, which in turn gives us the condition that we need to show that $T(n) \leqslant cn$.

- We conclude that $T(n) = O(n)$, so that SELECT runs in linear time in all cases.

- Why 140? We could have used any integer > 70.

  - For $n > 70$, the fraction $n/(n - 70)$ decreases as n increases.

  - We picked $n \geqslant 140$ so that the fraction would be $\leqslant 2$, which is an easy constant to work with.

Exercise 9.3-3

Show how quicksort can be made to run in $O(n \lg n)$ time in the worst case, assuming that all elements are distinct.

Solution to Exercise 9.3-3

- A modification to quicksort that allows it to run in $O(n \lg n)$ time in the worst case uses the deterministic PARTITION algorithm that was modified to take an element to partition around as an input parameter.

- SELECT takes an array A, the bounds p and r of the subarray in A, and the rank i, and in time linear in the size of the subarray $A[p..r]$ it returns the ith smallest element in $A[p..r]$.

Divide and Conquer II
9.3 Selection in worst-case linear time

```
BEST-CASE-QUICKSORT(A, p, r)
    if p < r then
        i = ⌊(r − p + 1)/2⌋
        x = SELECT(A, p, r, i)
        q = PARTITION(A, p, r, x)
        BEST-CASE-QUICKSORT(A, p, q − 1)
        BEST-CASE-QUICKSORT(A, q + 1, r)
```

Solution to Exercise 9.3-3 (Cont'd)

- For an n-element array, the largest subarray that
  BEST-CASE-QUICKSORT recourses on has $n/2$ elements.
  This situation occurs when $n = r - p + 1$ is even: then the
  subarray $A[q + 1..r]$ has $n/2$ elements, and the subarray
  $A[p..q - 1]$ has $n/2 - 1$ elements.

- Because BEST-CASE-QUICKSORT always recourses on
  subarrays that are at most half the size of the original array,
  the recurrence for the worst-case running time is
  $T(n) \leqslant 2T(n/2) + \Theta(n) = O(n \lg n)$.

Divide and Conquer II
9.3 Selection in worst-case linear time

Manuel Blum
(1938–)

Robert W. Floyd
(1936–2001)
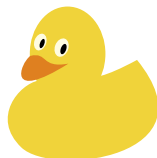
Vaughan Pratt
(1944–)
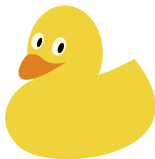
Ronald L. Rivest
(1947–)
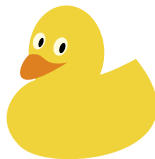
Robert E. Tarjan
(1948–)

Lecture 5
Dictionaries I

Dictionaries I

Contents

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009

Dictionaries I
Overview

- Many applications require a dynamic set that support only the dictionary operations search, insert, and delete.
- A hash table is effective for implementing a dictionary.
  - The average time to search for an element in a hash table is $\Theta(1)$, under reasonable assumptions.
  - Searching for an element in a hash table takes $\Theta(n)$ time in the worst case, however.
- A hash table is a generalization of an ordinary array.
  - With an ordinary array, we store the element whose key is k in position k of the array.
  - Given a key k, we find the element whose key is k by just looking in the k-th position in the array. This is called direct addressing.

Dictionaries I
Overview

- A hash table is a generalization of an ordinary array.
  - Direct addressing is applicable when we can afford to allocate an array with one position for every possible key.
- We use a hash table when we do not want to (or cannot) allocate an array with one position per possible key.
  - We use a hash table when the number of keys actually stored is small relative to the number of possible keys.
  - A hash table is an array, but it typically uses a size proportional to the number of keys to be stored (rather than the number of possible keys).
  - Given a key k, do not use k as the index into the array.
  - Instead, compute a function of k, and use that value to index the array. We call this function a hash function.

Dictionaries I
5.2 Indicator random variables

- A simple yet powerful technique for computing the expected value of a random variable.

- Given a sample space S and an event A, the indicator random variable

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs} \\ 0 & \text{if } A \text{ does not occur} \end{cases}$$

- The expected value of an indicator random variable associated with an event A is equal to the probability that A occurs.

Lemma
For an event A in a sample space S, let $X_A = I\{A\}$. Then
$E[X_A] = Pr\{A\}$.

Proof.
$E[X_A] = E[I\{A\}] = 1 \cdot Pr\{A\} + 0 \cdot Pr\{\bar{A}\} = Pr\{A\}$.

Dictionaries I
5.2 Indicator random variables

Example (Expected number of heads in one flip of a fair coin.)

- Sample space is $S = \{H, T\}$.
- $Pr\{H\} = Pr\{T\} = 1/2$.
- $X_H = I\{H\}$ (number of heads in one flip).
- $E[X_H] = E[I\{H\}] = 1 \cdot Pr\{H\} + 0 \cdot Pr\{T\} = 1/2$.

Dictionaries I
5.2 Indicator random variables

Example (Expected number of heads in n flips of a fair coin.)

- Sample space is $S = \{H, T\}$.
- $Pr\{H\} = Pr\{T\} = 1/2$.
- $X_i = I\{\text{the } i\text{th flip results in event } H\}$, for $i = 1, \ldots, n$.
- $X_H = \sum_{i=1}^{n} X_i$ (number of heads in n flips).
- $E[X_i] = Pr\{H\} = 1/2$, for $i = 1, \ldots, n$.
- $E[X_H] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} 1/2 = n/2$.

Dictionaries I
5.2 Indicator random variables

Exercise 5.2-5
Let $A[1..n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an inversion. Suppose that the elements of A form a uniform random permutation of $\{1, 2, \ldots, n\}$. Use indicator random variables to compute the expected number of inversions.

Solution to Exercise 5.2-5
- $X_{ij} = I\{A[i] > A[j]\}$ for $1 \leqslant i < j \leqslant n$ (indicator random variable for one inversion in the array).
- $Pr\{X_{ij} = 1\} = 1/2$ because given two distinct random numbers, the probability that the first is greater than the second is $1/2$.
- $E[X_{ij}] = 1/2$.

Solution to Exercise 5.2-5 (Cont'd)

- $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$ (indicator random variable for the number of inversions in the array).

- $E[X] = E[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}]$.

$$
\begin{aligned}
E[X] &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1/2 \\
&= \binom{n}{2} \frac{1}{2} = \frac{n(n-1)}{2} \cdot \frac{1}{2} = \frac{n(n-1)}{4}
\end{aligned}
$$

- The expected number of inversions is $n(n-1)/4$.

Dictionaries I
11.1 Direct-address tables

- How to compute hash functions? We will look at the multiplication and division methods.
- What to do when the hash function maps multiple keys to the same table entry? We will look at chaining and open addressing.
- Maintain a dynamic set in which each element has a key drawn from the universe $U = \{0, 1, \ldots, m - 1\}$, where m is not too large.
- Assume that no two elements have the same key.

Dictionaries I
11.1 Direct-address tables

- Represent the dynamic set by an array, or direct-address table, denoted by $T[0..m-1]$.
  - Each position, or slot, corresponds to a key in U.
  - If there is an element x with key k, then $T[k]$ contains a pointer to x.
  - Otherwise, $T[k]$ is empty, represented by NIL.
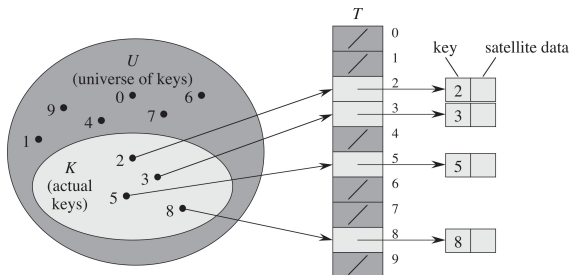
- Dictionary operations are trivial and take $O(1)$ time each.

SEARCH(T, k)
   return $T[k]$

INSERT(T, x)
   $T[x.key] = x$

DELETE(T, x)
   $T[x.key] = NIL$

Dictionaries I
11.2 Hash tables

- The problem with direct addressing is, if the universe U is large, storing a table T of size $|U|$ may be impractical, or even impossible.

- Often, the set K of keys actually stored is small, compared to U, and most of the space allocated for T is wasted.

- When K is much smaller than U, a hash table requires much less storage than a direct-address table.

- Can reduce the storage requirement to $\Theta(|K|)$.

- Can still get $O(1)$ search time, but in the average case, not the worst case.

- Instead of storing an element with key k in slot k, use a hash function h and store the element in slot $h(k)$.

Dictionaries I
11.2 Hash tables

- $h : U \to \{0, 1, \ldots, m-1\}$ maps the universe U of keys into the slots of a hash table $T[0..m-1]$.

- An element with key k hashes to slot $h(k)$.
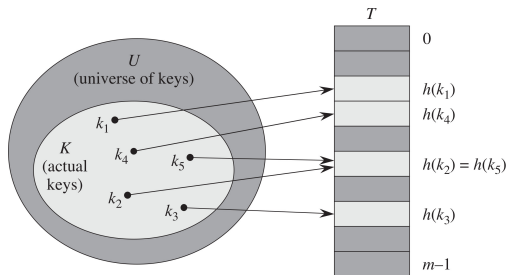
- When two or more keys hash to the same slot, there is a collision.

Dictionaries I
11.2 Hash tables

- In the simplest collision resolution, chaining, we put all elements that hash to the same slot into a linked list.
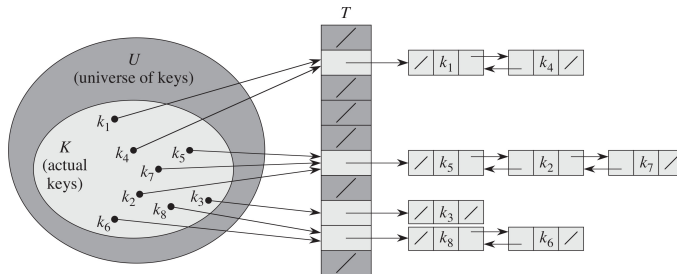
- Slot j contains a pointer to the head of the list of all stored elements that hash to j.
- If there are no such elements, slot j contains NIL.

- Dictionary operations are easy to implement when collisions are resolved by chaining.

- The worst-case running time for search is proportional to the length of the list of elements in slot $h(k)$.

    SEARCH(T, k)
      search for an element with key k in list $T[h(k)]$

- The worst-case running time for insertion is $O(1)$, assuming that the element x being inserted is not already in the list.

    INSERT(T, x)
      insert x at the head of list $T[h(x.key)]$

- The worst-case running time for deletion is $O(1)$ if the lists are doubly linked.

    DELETE(T, x)
      delete x from the list $T[h(x.key)]$

Dictionaries I
11.2 Hash tables

- How long does it take to search for an element with a given key in a hast table with chaining?

- Given a hash table T with m slots that stores n elements, define the load factor α for T as n/m, the average number of elements stored in a chain.

- The analysis will be in terms of α, which can be less than, equal to, or greater than 1.

- The worst case is when all n keys hash to the same slot, creating a list of length n.

- The worst-case time for searching is $\Theta(n)$ plus the time to compute the hash function.

- The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots.

Dictionaries I
11.2 Hash tables

- Assume simple uniform hashing, where any given element is equally likely to hash into any of the m slots.
- For $j = 0, 1, \ldots, m-1$, denote the length of list $T[j]$ by n_j, so that $n = n_0 + n_1 + \cdots + n_{m-1}$.
- Average value of n_j is $E[n_j] = \alpha = n/m$.
- Assume that we can compute the hash function in $O(1)$ time, so that the time required to search for an element with key k depends on the length $n_{h(k)}$ of the list $T[h(k)]$.
- If the hash table contains no element with key k, then the search is unsuccessful.
- If the hash table does contain an element with key k, then the search is successful.

Dictionaries I
11.2 Hash tables

Theorem

In a hash table in which collisions are resolved by chaining, under the assumption of simple uniform hashing, an unsuccessful search takes average-case time $\Theta(1 + \alpha)$.

Proof.

- Any key k not already in the table is equally likely to hash to any of the m slots.

- To search unsuccessfully for a key k we need to search to the end of list $T[h(k)]$, which has expected length $E[n_{h(k)}] = \alpha$.

- The expected number of elements examined in an unsuccessful search is α.

- Adding the time for computing $h(k)$, the total time required is $\Theta(1 + \alpha)$.

Dictionaries I
11.2 Hash tables

In a successful search, the probability that a list is searched is proportional to the number of elements it contains.

Theorem
In a hash table in which collisions are resolved by chaining, under the assumption of simple uniform hashing, a successful search takes average-case time $\Theta(1 + \alpha)$.

Proof.
- Assume that the element being searched for is equally likely to be any of the n elements in the table.
- The number of elements examined during a successful search for x is 1 plus the number of elements that appear before x in the list.
- These elements were all inserted after x was inserted, because we insert at the head of the list.

Dictionaries I
11.2 Hash tables

Proof.

- To find the expected number of elements examined, we take the average (over the n elements x in the table) of 1 plus the expected number of elements added to the list after x was added to the list.

- Let x_i denote the ith element inserted into the table, for $i = 1, \ldots, n$, and let $k_i = \text{key}[x_i]$.

- $X_{ij} = I\{h(k_i) = h(k_j)\}$, for all i, j.

- $Pr\{h(k_i) = h(k_j)\} = 1/m$ (simple uniform hashing).

- $E[X_{ij}] = 1/m$.

Proof.

- The expected number of elements examined in a successful search is

$$
E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]
$$

$$
= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right) = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)
$$

$$
= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) = 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right)
$$

$$
= 1+\frac{n-1}{2m} = 1+\frac{\alpha}{2}-\frac{\alpha}{2n}
$$

Dictionaries I
11.2 Hash tables

Proof.

- Adding the time for computing $h(k)$, the total time required for a successful search is $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$.

Interpretation

- If the number of slots is proportional to the number of elements in the table, $n = O(m)$, and then $\alpha = n/m = O(m)/m = O(1)$, and searching takes constant time on average.

- Since insertion and deletion take $O(1)$ worst-case time, we can support all dictionary operations in $O(1)$ time on average.

What makes a good hash function?

- Ideally, the hash function satisfies the assumption of simple uniform hashing.
- In practice, it is not possible to satisfy this assumption, since we do not know in advance the probability distribution that keys are drawn from, and the keys may not be drawn independently.
- Often use heuristics, based on the domain of the keys, to create a hash function that performs well.

Keys as natural numbers

- Hash functions assume that the keys are natural numbers.
- When they are not, have to interpret them as natural numbers.

Example (Interpret a character string as an integer expressed in some radix notation.)

Suppose the string is CLRS.

- ASCII values: C = 67, L = 76, R = 82, S = 83.
- There are 128 basic ASCII values.
- Interpret CLRS as
  $(67 \cdot 128^3) + (76 \cdot 128^2) + (82 \cdot 128^1) + (83 \cdot 128^0) = 141,764,947$.

Dictionaries I
11.3 Hash functions

Division method

- $h(k) = k \bmod m$.
- Example: $m = 20$ and $k = 91$. $h(k) = 11$.
- Advantage: Fast, since it requires just one division operation.
- Disadvantage: Have to avoid certain values of m.
  - Powers of 2 are bad. If $m = 2^p$ for integer p, then $h(k)$ is just the least significant p bits of k.
  - If k is a character string interpreted in radix 2^p, then $m = 2^p - 1$ is bad. Permuting characters in a string does not change its hash value (Exercise 11.3-3).
- Good choice for m: A prime not too close to an exact power of 2. Example: $n = 2000$, average of 3 elements in an unsuccessful search, $m = 701$, $h(k) = k \bmod 701$.

Division method

- Good choice for m: A prime not too close to an exact power of 2. Example: $n = 2000$, average of 3 elements in an unsuccessful search, $m = 701$, $h(k) = k \bmod 701$.
    - $2000/3 = 666.\overline{6}$
    - $2^9 = 512 < 666.\overline{6} < 1,024 = 2^{10}$
    - $(2^9 + 2^{10})/2 = 768$
    - $761 \leqslant 768 \leqslant 769$, 761 prime, 769 prime
    - Both $m = 761$ and $m = 769$ are good choices.

Search trees

- Data structures that support many dynamic-set operations.

- Can be used as both a dictionary and as a priority queue.

- Basic operations take time proportional to the height of the tree.

  - For complete binary tree with n nodes: worst case $\Theta(\log n)$.
  - For linear chain of n nodes: worst case $\Theta(n)$.

- Different types of search trees include binary search trees, red-black trees, AVL trees, and B-trees.

- We will cover binary search trees, tree walks, operations on binary search trees, and AVL trees.

Dictionaries I
12.1 What is a binary search tree

An important data structure for dynamic sets.

- Accomplish many dynamic-set operations in $O(h)$ time, where h is the height of the tree.

- We represent a binary tree by a linked data structure in which each node is an object.

- $T.root$ points to the root of tree T.

- Each node contains the attributes
    - *key* (and possibly other satellite data)
    - *left* (points to left child)
    - *right* (points to right child)
    - *p* (points to parent), with $T.root.parent = $ NIL

- Stored keys must satisfy the binary-search-tree property.
    - If y is in left subtree of x, then $y.key \leqslant x.key$.
    - If y is in right subtree of x, then $y.key \geqslant x.key$.

- For any node x, the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$.
- The worst-case running time for most search-tree operations is proportional to the height of the tree.

Example (Binary search trees)
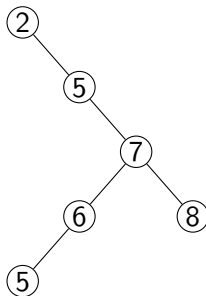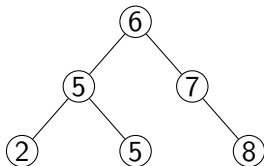
Dictionaries I
12.1 What is a binary search tree

- The binary-search-tree property allows us to print keys in a binary search tree in order, recursively, using an algorithm called an inorder tree walk.

- Elements are printed in monotonically increasing order.

```
INORDER-TREE-WALK(x)
    if x ≠ NIL then
        INORDER-TREE-WALK(x.left)
        print x.key
        INORDER-TREE-WALK(x.right)
```

- Correctness of inorder tree walk follows by induction directly from the binary-search-tree property.

- Intuitively, the walk takes $\Theta(n)$ time for a tree with n nodes, because we visit and print each node once.

Searching

- Initial call is TREE-SEARCH($T.root$, k).

```
TREE-SEARCH(x, k)
    if x = NIL or k = x.key then
        return x
    else if k < x.key then
        return TREE-SEARCH(x.left, k)
    else
        return TREE-SEARCH(x.right, k)
```

- The algorithm recourses, visiting nodes on a downward path from the root.

- Running time is $O(h)$, where h is the height of the tree.

Dictionaries I
12.2 Querying a binary search tree

Minimum and maximum

- The binary-search-tree property guarantees that
  - the minimum key of a binary search tree is located at the leftmost node
  - the maximum key of a binary search tree is located at the rightmost node
- Traverse the appropriate pointers (*left* or *right*) until NIL is reached.

```
TREE-MINIMUM(x)
    while x.left ≠ NIL do
        x = x.left
    return x
```

Minimum and maximum

- Traverse the appropriate pointers (*left* or *right*) until NIL is reached.

  TREE-MAXIMUM(x)
      while $x.right \neq$ NIL **do**
          $x = x.right$
      return x

- Both procedures visit nodes that form a downward path from the root to a leaf.

- Running time is $O(h)$, where h is the height of the tree.

Dictionaries I
12.2 Querying a binary search tree

Successor and predecessor

- Assuming that all keys are distinct, the successor of a node x is the node y with the smallest key greater than $x.key$.

- If node x has the largest key in the binary search tree, then we say that the successor of x is NIL.

- There are two cases:
    - If the right subtree of node x is non-empty, then the successor of x is just the leftmost node in the right subtree of x.
    - If the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.

Exercise 12.2-6

Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. (Recall that every node is its own ancestor.)

Solution to Exercise 12.2-6

- First, we establish that y must be an ancestor of x. If y were not an ancestor of x, then let z denote the lowest common ancestor of x and y. By the binary-search-tree property, $x < z < y$ and then, y cannot be the successor of x.

- Next, observe that $y.left$ must be an ancestor of x because if it were not, then $y.right$ would be an ancestor of x, which implies either $y < x \leqslant y.right$ or $y < y.right \leqslant x$ and then, y cannot be the successor of x.

- Finally, suppose that y is not the lowest ancestor of x whose left child is also an ancestor of x. Let z denote this lowest ancestor. Then z must be in the left subtree of y, which implies $x < z < y$ and then, y cannot be the successor of x.

Successor and predecessor

```
TREE-SUCCESSOR(x)
    if x.right ≠ NIL then
        return TREE-MINIMUM(x.right)
    y = x.p
    while y ≠ NIL and x = y.right do
        x = y
        y = y.p
    return y
```

- To find y, we simply go up the tree from x until we encounter a node that is the left child of its parent.
- TREE-PREDECESSOR is symmetric to TREE-SUCCESSOR.

Successor and predecessor

- Both procedures visit nodes either on a path down the tree or on a path up the tree.
- Running time is $O(h)$, where h is the height of the tree.
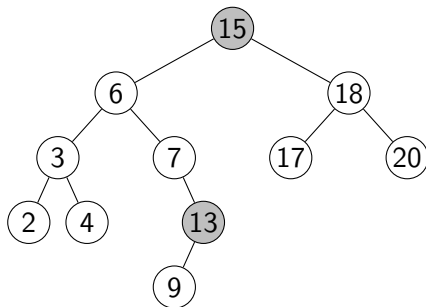
Dictionaries I
12.2 Querying a binary search tree

Exercise 12.2-3

Write the TREE-PREDECESSOR procedure.

Solution to Exercise 12.2-3

```
TREE-PREDECESSOR(x)
    if x.left ≠ NIL then
        return TREE-MAXIMUM(x.left)
    y = x.p
    while y ≠ NIL and x = y.left do
        x = y
        y = y.p
    return y
```

- To find y, we simply go up the tree from x until we encounter a node that is the right child of its parent.

Dictionaries I
12.3 Insertion and deletion

Insertion

- To insert a value v into a binary search tree T, the procedure takes a node z with $z.key = v$, $z.left = \text{NIL}$, $z.right = \text{NIL}$.

Example (Inserting an item with key 13 into a binary search tree)
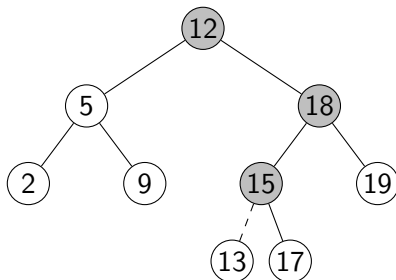
Dictionaries I
12.3 Insertion and deletion

Exercise 12.3-1
Give a recursive version of the TREE-INSERT procedure.

Solution to Exercise 12.3-1

- Beginning at the root of the tree, trace a downward path, maintaining two pointers.
  - Pointer x to trace the downward path, looking for a NIL to replace with item z.
  - Trailing pointer y to keep track of the parent of x.
- Running time is $O(h)$, where h is the height of the tree.

TREE-INSERT(T, z)
   TREE-INSERT(NIL, $T.root, z$)

Dictionaries I
12.3 Insertion and deletion

Solution to Exercise 12.3-1

```
TREE-INSERT(y, x, z)
    if x ≠ NIL then
        if z.key < x.key then
            TREE-INSERT(x, x.left, z)
        else
            TREE-INSERT(x, x.right, z)
    z.p = y
    if y = NIL then
        T.root = z
    else if z.key < y.key then
        y.left = z
    else
        y.right = z
```

Solution to Exercise 12.3-1

- To insert a value v into a binary search tree T, take a node z with $z.key = v$, $z.left = \text{NIL}$, $z.right = \text{NIL}$ and $x = T.root$.

```
TREE-INSERT(x, z)
   if x = NIL then
      return z
   else
      if z.key ⩽ x.key then
         x.left = TREE-INSERT(x.left, z)
         x.left.p = x
      else
         x.right = TREE-INSERT(x.right, z)
         x.right.p = x
      return x
```

Deletion

- Deleting node z from binary search tree T has three cases.
    - If z has no children, just remove it (modifying its parent to replace z with NIL as a child).
    - If z has just one child, then elevate that child to take the position of z in the tree (by modifying the parent of z to replace z by its child).
    - If z has two children, then find the successor y of z and replace z by y, splicing y out of its current location.
- Running time is $O(h)$, where h is the height of the tree.

Deletion

- If z has no children, just remove it (modifying its parent to replace z with NIL as a child).

  (Node z is the left child of its parent, if any.)
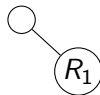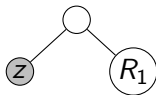
  (Node z is the right child of its parent, if any.)

Deletion

- If z has just one child, then elevate that child to take the position of z in the tree (by modifying the parent of z to replace z by its child).

  (Node z has no left child.)
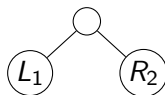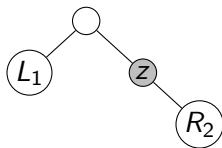
  (Node z has no right child.)
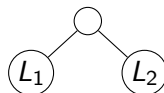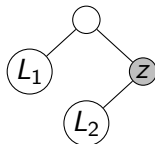
Dictionaries I
12.3 Insertion and deletion

Deletion

- If z has two children, then find the successor y of z and replace z by y, splicing y out of its current location.
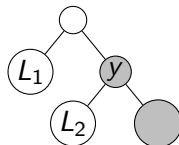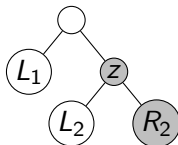
Dictionaries I
12.3 Insertion and deletion

Deletion

- Deleting node z from binary search tree T

```
TREE-DELETE(T, z)
    if z ≠ NIL then
        if z.left = NIL then
            z = z.right
        else if z.right = NIL then
            z = z.left
        else
            y = TREE-DELETE-MINIMUM(T, z.right)
            y.left = z.left
            y.right = z.right
            z = y
```

- Running time is $O(h)$, where h is the height of the tree.

Dictionaries I
12.3 Insertion and deletion

Deletion

```
TREE-DELETE-MINIMUM(z)
    if z.left ≠ NIL then
        return TREE-DELETE-MINIMUM(z.left)
    else
        y = z
        z = z.right
        return y
```

- Running time is $O(h)$, where h is the height of the tree.

Exercise 12.3-3
We can sort a given set of n numbers by first building a binary
search tree containing these numbers (using TREE-INSERT
repeatedly to insert the numbers one by one) and then printing the
numbers by an inorder tree walk. What are the worst-case and
best-case running times for this sorting algorithm?

Solution to Exercise 12.3-3

```
TREE-SORT(A)
    let T be an empty binary search tree
    for i = 1 to n do
        TREE-INSERT(T.root, A[i])
    INORDER-TREE-WALK(T.root)
```

- The worst case occurs when a linear chain of nodes results from the repeated TREE-INSERT operations, with $\Theta(n^2)$ running time.

- The best case occurs when a binary tree of height $\Theta(\log n)$ results from the repeated TREE-INSERT operations, with $\Theta(n \log n)$ running time.

Lecture 6
Dictionaries II

Dictionaries II

Contents

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009

Dictionaries II
13-3 AVL trees

Problem 13-3

- An AVL tree is a binary search tree that is height balanced: for each node x, the heights of the left and right subtrees of x differ by at most 1.

- To implement an AVL tree, we maintain an extra attribute in each node: $x.h$ is the height of node x.

- As for any other binary search tree T, we assume that $T.root$ points to the root node.

Problem 13-3-a

- Prove that an AVL tree with n nodes has height $O(\log n)$.

- Hint: Prove that an AVL tree of height h has at least F_h nodes, where F_h is the hth Fibonacci number.

Solution to Problem 13-3-a

- Let $T(h)$ be the minimum number of nodes in an AVL tree of height h. We want to show that $T(h) \geqslant F_h$.

- We proceed by induction. For the base case, note that $T(1) \geqslant T(0) \geqslant 1$. Thus, $T(1) \geqslant F_1$ and $T(0) \geqslant F_0$.

- Now, assume that $T(h') \geqslant F_{h'}$ for all $h' < h$.

- The root of an AVL tree of height h has a child of height at least $h - 1$ and another child of height at least $h - 2$.

- Thus, $T(h) \geqslant T(h-1) + T(h-2)$ and, by induction hypothesis, $T(h) \geqslant F_{h-1} + F_{h-2} = F_h$.

- Since $F_h \geqslant \phi^h / \sqrt{5}$, an AVL tree with n nodes and height h must satisfy $n \geqslant T(h) \geqslant F_h \geqslant \phi^h / \sqrt{5}$. Thus, $h = O(\log n)$.

Exercise 3.2-6
Show that the golden ratio $\phi = (1 + \sqrt{5})/2$ and its conjugate
$\hat{\phi} = (1 - \sqrt{5})/2$ both satisfy the equation $x^2 = x + 1$.

Solution to Exercise 3.2-6

$$\phi^2 = \left(\frac{1+\sqrt{5}}{2}\right)^2 = \frac{6+2\sqrt{5}}{4} = \frac{3+\sqrt{5}}{2} = \frac{1+\sqrt{5}}{2} + 1 = \phi + 1$$

$$\hat{\phi}^2 = \left(\frac{1-\sqrt{5}}{2}\right)^2 = \frac{6-2\sqrt{5}}{4} = \frac{3-\sqrt{5}}{2} = \frac{1-\sqrt{5}}{2} + 1 = \hat{\phi} + 1$$

Dictionaries II
13-3 AVL trees

Exercise 3.2-7
Prove by induction that the ith Fibonacci number satisfies the
equality $F_i = (\phi^i - \hat{\phi}^i)/\sqrt{5}$, where ϕ is the golden ratio and $\hat{\phi}$ is its
conjugate.

Solution to Exercise 3.2-6
- $F_0 = (\phi^0 - \hat{\phi}^0)/\sqrt{5} = (1-1)/\sqrt{5} = 0$
- $F_1 = (\phi^1 - \hat{\phi}^1)/\sqrt{5} = \sqrt{5}/\sqrt{5} = 1$

$$
\begin{aligned}
F_{i+1} &= F_i + F_{i-1} = (\phi^i - \hat{\phi}^i)/\sqrt{5} + (\phi^{i-1} - \hat{\phi}^{i-1})/\sqrt{5} \\
&= ((\phi^i - \hat{\phi}^i) + (\phi^{i-1} - \hat{\phi}^{i-1}))/\sqrt{5} \\
&= ((\phi^i + \phi^{i-1}) - (\hat{\phi}^i + \hat{\phi}^{i-1}))/\sqrt{5} \\
&= (\phi^{i-1}(\phi+1) - \hat{\phi}^{i-1}(\hat{\phi}+1))/\sqrt{5} \\
&= (\phi^{i-1}\phi^2 - \hat{\phi}^{i-1}\hat{\phi}^2)/\sqrt{5} = (\phi^{i+1} - \hat{\phi}^{i+1})/\sqrt{5}
\end{aligned}
$$

Solution to Problem 13-3-a

- Now, $|\hat{\phi}| < 1$ implies $|\hat{\phi}^i|/\sqrt{5} < 1/\sqrt{5} < 1/2$.
- Thus, $F_i = (\phi^i - \hat{\phi}^i)/\sqrt{5} = \lfloor \phi^i/\sqrt{5} + 1/2 \rfloor \geqslant \phi^i/\sqrt{5}$.
- Therefore, $h \leqslant \log_\phi(\sqrt{5}\,n) = \log_\phi \sqrt{5} + \log_\phi n$, and then $h = O(\log n)$.

Dictionaries II
13-3 AVL trees

Problem 13-3-b

- To insert into an AVL tree, we first place a node into the appropriate place in binary search tree order.
- Afterward, the tree might no longer be height balanced.
- Specifically, the heights of the left and right children of some node might differ by 2. Describe a procedure BALANCE(x), which takes a subtree rooted at x whose left and right children are height balanced and have heights that differ by at most 2, that is, $|x.right.h - x.left.h| \leqslant 2$, and alters the subtree rooted at x to be height balanced.
- Hint: Use rotations.

Dictionaries II
13-3 AVL trees

Solution to Problem 13-3-b

- After an insertion into an AVL tree, four generic situations may arise that lead to an unbalanced tree, which can be balanced using one or two rotations.
- Left rotation.
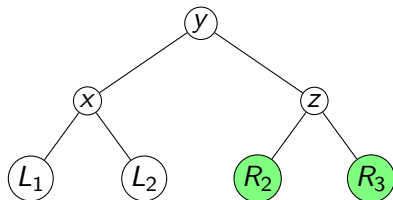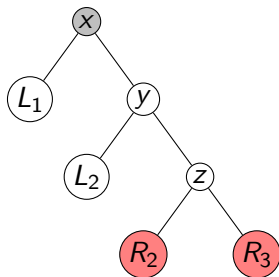
Solution to Problem 13-3-b
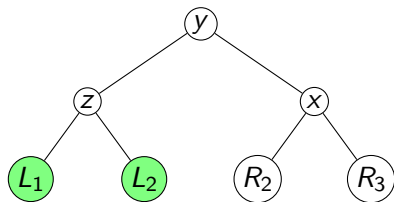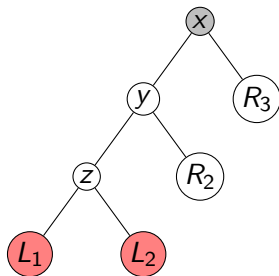
- Right rotation.

Dictionaries II
13-3 AVL trees

Solution to Problem 13-3-b

- Left-right rotation.
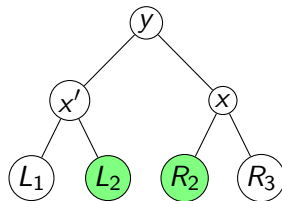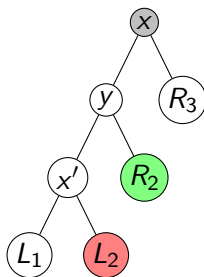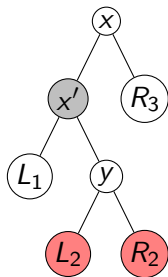
Solution to Problem 13-3-b

- Right-left rotation.
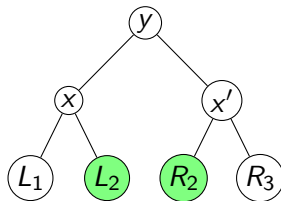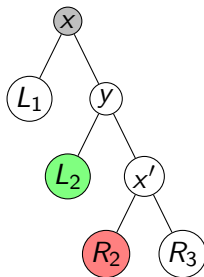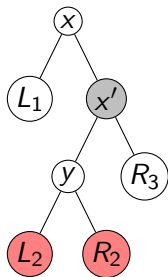
Solution to Problem 13-3-b

```
BALANCE(x)
    if |x.right.h − x.left.h| ⩽ 1 then
        return x
    else if x.left.h > x.right.h then
        y = x.left
        if y.left.h < y.right.h then
            LEFT-ROTATE(y)
        return RIGHT-ROTATE(x)
    else
        y = x.right
        if y.left.h > y.right.h then
            RIGHT-ROTATE(y)
        return LEFT-ROTATE(x)
```

Solution to Problem 13-3-b

- LEFT-ROTATE performs the corresponding tree rotation and returns the root of the subtree after the rotation.

```
LEFT-ROTATE(x)
    y = x.right
    x.right = y.left
    y.left = x
    x.h = 1 + max(x.left.h, x.right.h)
    y.h = 1 + max(x.h, y.right.h)
    x = y
    return x
```

Solution to Problem 13-3-b

- RIGHT-ROTATE performs the corresponding tree rotation and returns the root of the subtree after the rotation.

```
RIGHT-ROTATE(x)
    y = x.left
    x.left = y.right
    y.right = x
    x.h = 1 + max(x.left.h, x.right.h)
    y.h ← 1 + max(y.left.h, x.h)
    x = y
    return x
```

Problem 13-3-c

- Using part (b), describe a recursive procedure
  AVL-INSERT(x, z) that takes a node x within an AVL tree
  and a newly created node z (whose key has already been filled
  in), and adds z to the subtree rooted at x, maintaining the
  property that x is the root of an AVL tree.

- As in TREE-INSERT, assume that $z.key$ has already been
  filled in and that $z.left =$ NIL and $z.right =$ NIL; also assume
  that $z.h = 0$.

- Thus, to insert the node z into the AVL tree T, we call
  AVL-INSERT($T.root, z$).

Solution to Problem 13-3-c

```
AVL-INSERT(x, z)
    if x = NIL then
        return z
    if z.key ⩽ x.key then
        x.left = AVL-INSERT(x.left, z)
        x.left.p = x
        x.h = x.left.h + 1
    else
        x.right = AVL-INSERT(x.right, z)
        x.right.p = x
        x.h = x.right.h + 1
    x = BALANCE(x)
    return x
```

Dictionaries II
13-3 AVL trees

Problem 13-3-d

- Show that AVL-INSERT, run on an *n*-node AVL tree, takes $O(\lg n)$ time and performs $O(1)$ rotations.

Solution to Problem 13-3-d

- The height of the AVL tree is $O(\log n)$.
- Therefore, the insertion and update of the height attribute take $O(\log n)$ time.
- In addition, the BALANCE operation decreases the height of the originally unbalanced tree by 1 after the rotation and thus, it will not cause any propagation of rotations to the rest of the tree.
- Therefore, AVL-INSERT takes $O(\log n)$ time to insert the node, and performs $O(1)$ rotations.

- Like merge sort, but unlike insertion sort, heapsort runs in $O(n \log n)$ time.

- Like insertion sort, but unlike merge sort, heapsort sorts in place (only a constant number of array elements are stored outside the input array at any time).

- Heapsort combines the best of the two sorting algorithms.

- The heap data structure is useful for heapsort and it also makes an efficient priority queue.

Dictionaries II
6.1 Heaps

Heap data structure

- A heap data structure is an array object that we can view as a nearly complete binary tree.
    - Each node of the tree corresponds to an element of the array.
    - The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.
- An array A that represents a heap is an object with two attributes:
    - $A.length$ (the number of elements in the array)
    - $A.heap\text{-}size$ (how many elements in the heap are stored within the array)
- Only the elements in $A[1..A.heap\text{-}size]$, where $0 \leqslant A.heap\text{-}size \leqslant A.length$, are valid elements of the heap.

Dictionaries II
6.1 Heaps

Example (A max-heap viewed as a binary tree and an array.)

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1
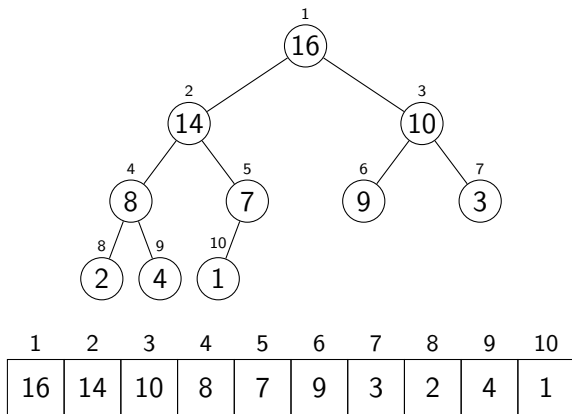
Dictionaries II
6.1 Heaps

Heap data structure

- The root of the tree is $A[1]$.
- Given the index i of a node, we can easily compute the indices of its parent, left child, and right child.

```
PARENT(i)
    return ⌊i/2⌋
```

```
LEFT(i)
    return 2i
```

```
RIGHT(i)
    return 2i + 1
```

- Computing indices is fast with the binary representation.
  - Shift right by one bit position to divide by 2.
  - Shift left by one bit position to multiply by 2.

Heap property

- There are two kinds of heaps: max-heaps and min-heaps.
- In a max-heap, the max-heap property is that for every node i other than the root, $A[\text{PARENT}(i)] \geqslant A[i]$.
  - The largest element is at the root.
  - The subtree rooted at a node contains values no larger than that contained at the node itself.
- In a min-heap, the min-heap property is that for every node i other than the root, $A[\text{PARENT}(i)] \leqslant A[i]$.
  - The smallest element is at the root.
  - The subtree rooted at a node contains values no smaller than that contained at the node itself.
- The heapsort algorithm uses max-heaps.

- MAX-HEAPIFY is used to maintain the max-heap property.

```
MAX-HEAPIFY(A, i)
    ℓ = LEFT(i)
    r = RIGHT(i)
    largest = i
    if ℓ ⩽ A.heap-size and A[ℓ] > A[largest] then
        largest = ℓ
    if r ⩽ A.heap-size and A[r] > A[largest] then
        largest = r
    if largest ≠ i then
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```

- MAX-HEAPIFY is used to maintain the max-heap property.
  - Assume that the trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but that $A[i]$ might be smaller than its children (thus violating the max-heap property).
  - After MAX-HEAPIFY, the tree rooted at i is a max-heap.
- At each step,
  - Compare $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$.
  - If necessary, swap $A[i]$ with the larger of the two children to preserve the max-heap property.
  - Continue this process of comparing ans swapping down the heap, until the tree rooted at i is a max-heap.
- Since a heap is a nearly complete binary tree, the running time of MAX-HEAPIFY on a node of height h is $O(h) = O(\log n)$.

Example (MAX-HEAPIFY)

6.2 Maintaining the heap property

Example (MAX-HEAPIFY)

Dictionaries II
6.2 Maintaining the heap property

Example (MAX-HEAPIFY)
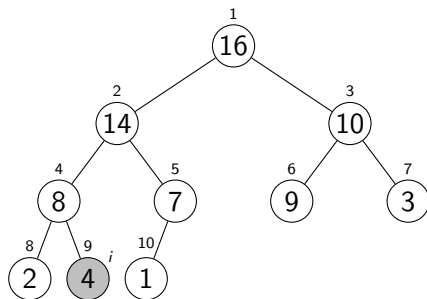
Dictionaries II
6.3 Building a heap

- We can use MAX-HEAPIFY bottom-up to convert an array $A[1..n]$, where $n = A.length$, into a max-heap.

- The elements in the subarray $A[\lfloor n/2 \rfloor + 1..n]$ are all leaves of the tree, that is, 1-element heaps to begin with.

> BUILD-MAX-HEAP(A)
> $A.heap\text{-}size = A.length$
>   **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1 **do**
>   MAX-HEAPIFY(A, i)

Exercise 6.1-7
Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by
$\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

Dictionaries II
6.3 Building a heap
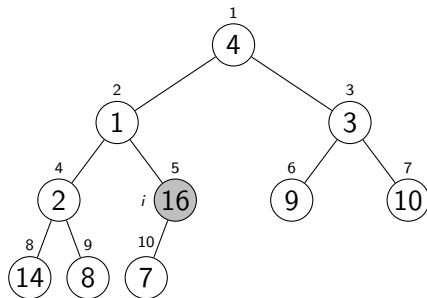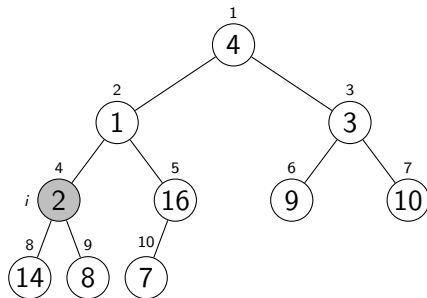
Example (BUILD-MAX-HEAP)
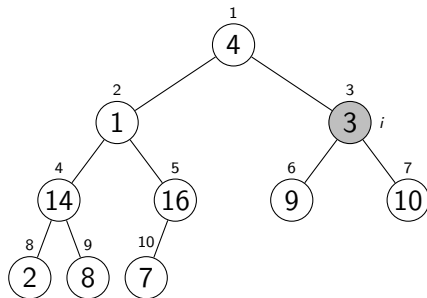
Dictionaries II
6.3 Building a heap

Example (BUILD-MAX-HEAP)
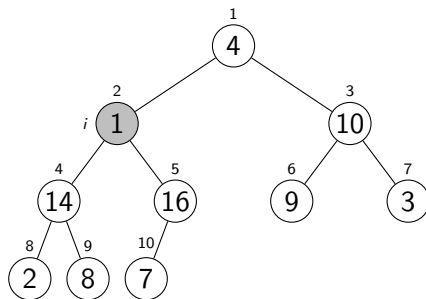
Dictionaries II
6.3 Building a heap

Example (BUILD-MAX-HEAP)
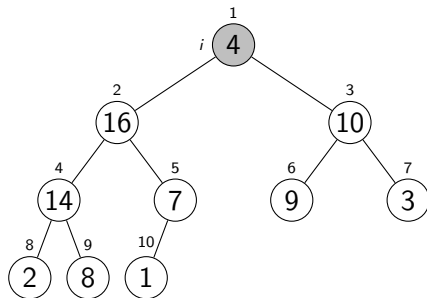
Example (BUILD-MAX-HEAP)

Example (BUILD-MAX-HEAP)

Example (BUILD-MAX-HEAP)
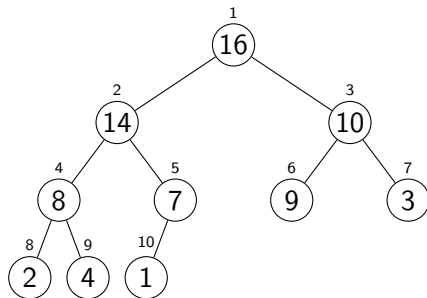
Dictionaries II
6.3 Building a heap

- The running time of BUILD-MAX-HEAP is $O(n \log n)$, because it makes $O(n)$ calls to MAX-HEAPIFY, each of which takes $O(\log n)$ time.

- However, MAX-HEAPIFY takes time linear in the height of the node in the tree, and most nodes have a small height.

Exercise 6.1-2
Show that an n-element heap has height $\lfloor \lg n \rfloor$.

Exercise 6.3-3
Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

Exercise 6.1-1
What are the minimum and maximum numbers of elements in a heap of height h?

Solution to Exercise 6.1-1
Since a heap is an almost-complete binary tree (complete at all levels except possibly the lowest), it has at most $2^{h+1} - 1$ elements (if it is complete) and at least $2^h - 1 + 1 = 2^h$ elements (if the lowest level has just 1 element and the other levels are complete).

Solution to Exercise 6.1-2
- Given an n-element heap of height h, we know that $2^h \leqslant n \leqslant 2^{h+1} - 1 < 2^{h+1}$.
- Thus, $h \leqslant \lg n < h + 1$. Since h is an integer, $h = \lfloor \lg n \rfloor$.

Solution to Exercise 6.3-3

- For a complete binary tree, it is easy to show that there are $\lceil n/2^{h+1} \rceil$ nodes of height h. For an incomplete tree, it can be proved by induction on h that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h.

- Base case: The nodes of height 0 are the leaves and there are $\lceil n/2 \rceil$ leaves in an n-element heap, by Exercise 6.1-7.

- Inductive step: Assume there are at most $\lceil n/2^{k+1} \rceil$ nodes of height k in any n-element heap for any $k < h$. Removing all the leaves from a tree of height h yields a new tree of height $h - 1$ with $n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$ nodes. The number of nodes of height $h - 1$ in the new tree is the same as the number of nodes of height h in the original tree, and it is given by $\lceil \lfloor n/2 \rfloor / 2^{h-1+1} \rceil \leqslant \lceil (n/2)/2^h \rceil = \lceil n/2^{h+1} \rceil$.

- MAX-HEAPIFY takes $O(h)$ time on a node of height h.
- The total cost of BUILD-MAX-HEAP is

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

- Evaluate the last summation by substituting $x = 1/2$ in the formula $\sum_{k=0}^{\infty} k \, x^k = x/(1-x)^2$ for $|x| < 1$, yielding

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

- Thus, the running time of BUILD-MAX-HEAP is $O(n)$.

- Build a max-heap from the input array.

- Starting with the root (the largest element), place the largest element into the correct place in the array by swapping it with the element in the last position of the array.

- Discard this last node from the heap by decreasing the heap size, and restore the max-heap property.

- Repeat this process until only one node (the smallest element) remains, and therefore it is in the correct place in the array.

```
HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    for i = A.length downto 2 do
        exchange A[1] with A[i]
        A.heap-size = A.heap-size − 1
        MAX-HEAPIFY(A, 1)
```

Dictionaries II
6.4 The heapsort algorithm
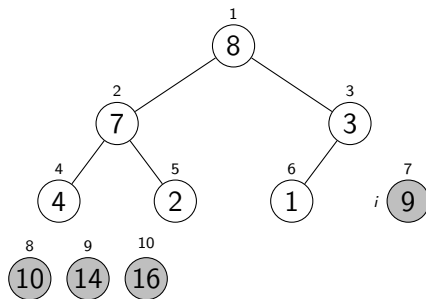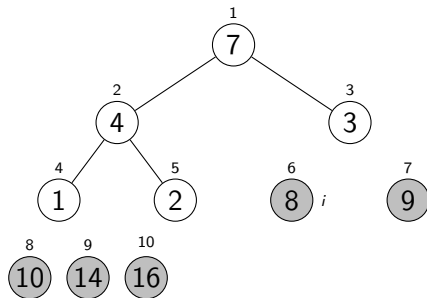
Example (HEAPSORT)

Dictionaries II
6.4 The heapsort algorithm

Example (HEAPSORT)

Example (HEAPSORT)

Dictionaries II
6.4 The heapsort algorithm

Example (HEAPSORT)

Example (HEAPSORT)

Example (HEAPSORT)

Example (HEAPSORT)

Example (HEAPSORT)

Example (HEAPSORT)

Example (HEAPSORT)

- HEAPSORT takes $O(n \log n)$ time, since BUILD-MAX-HEAP takes $O(n)$ time and each of the $n-1$ calls to MAX-HEAPIFY takes $O(\log n)$ time.

Exercise 6.4-3
What is the running time of HEAPSORT on an array A of length n that is already sorted in increasing order? What about decreasing order?

Exercise 6.4-4
Show that the worst-case running time of HEAPSORT is $\Omega(n \log n)$.

Heap implementation of priority queue

- A heap gives a good compromise between fast insertion but slow extraction and vice versa. Both take $O(\log n)$ time.

- Focus on max-priority queues, which are based on max-heaps.

- A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key, supporting the dynamic set operations:

    - INSERT(S, x) to insert the element x into the set S.
    - MAXIMUM(S) to return the element of S with the largest key.
    - EXTRACT-MAX(S) to remove and return the element of S with the largest key.
    - INCREASE-KEY(S, x, k) to increase the (at least as large as k) value of the key of element x to the new value k.

Dictionaries II
6.5 Priority queues

Finding the largest element

> HEAP-MAXIMUM(A)
>   **return** $A[1]$

- Running time is $O(1)$.

Dictionaries II
6.5 Priority queues

Extracting the largest element

```
HEAP-EXTRACT-MAX(A)
    if A.heap-size < 1 then
        error "heap underflow"
    max = A[1]
    A[1] = A[A.heap-size − 1]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
    return max
```

- Running time on an n-element heap is $O(\log n)$.

Increasing key value

```
HEAP-INCREASE-KEY(A, i, k)
    if k < A[i] then
        error "new key is smaller than current key"
    A[i] = k
    while i > 1 and A[PARENT(i)] < A[i] do
        exchange A[i] with A[PARENT(i)]
        i = PARENT(i)
```

- Running time on an n-element heap is $O(\log n)$.

Dictionaries II
6.5 Priority queues
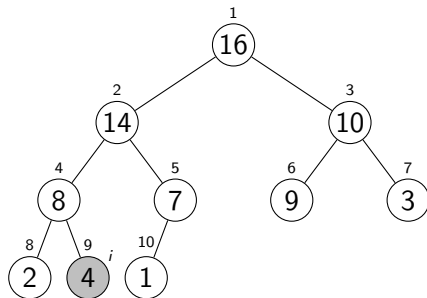
Example (HEAP-INCREASE-KEY)

Dictionaries II
6.5 Priority queues
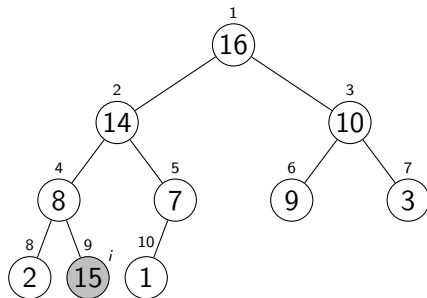
Example (HEAP-INCREASE-KEY)

Example (HEAP-INCREASE-KEY)
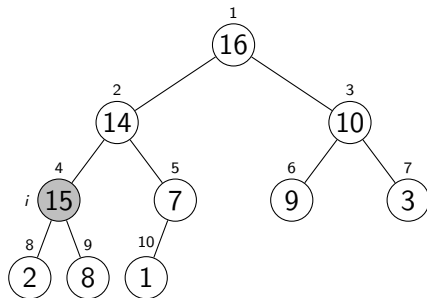
Dictionaries II
6.5 Priority queues

Example (HEAP-INCREASE-KEY)
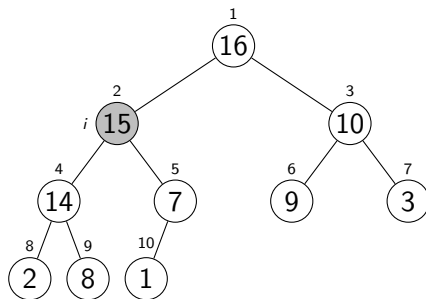
Dictionaries II
6.5 Priority queues

Inserting into the heap

```
HEAP-MAX-INSERT(A, k)
    A.heap-size = A.heap-size + 1
    A[A.heap-size] = −∞
    HEAP-INCREASE-KEY(A, A.heap-size, k)
```

- Running time on an n-element heap is $O(\log n)$.

Summary

- Priority-queue operations take $O(\log n)$ time.
    - MAX-HEAP-INSERT(A, k)
    - HEAP-MAXIMUM(A)
    - HEAP-EXTRACT-MAX(A)
    - HEAP-INCREASE-KEY(A, i, k)

Lecture 7
Graphs I

Graphs I

Contents

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009

Graphs I
22.1 Representation of graphs

Overview

- A graph $G = (V, E)$ can be directed or undirected.
- Two standard ways to represent a graph for algorithms:
  - A collection of adjacency lists.
  - An adjacency matrix.
- A graph $G = (V, E)$ is sparse if $|E|$ is much less than $|V|^2$.
- A graph $G = (V, E)$ is dense if $|E|$ is close to $|V|^2$.
- When expressing the running time of an algorithm, it is often in terms of both $|V|$ and $|E|$.

Adjacency lists

- The adjacency-list representation of a graph $G = (V, E)$ is an array Adj of $|V|$ lists, one for each vertex in V.

- For each $u \in V$, the adjacency list $Adj[u]$ has all the vertices v such that there is an edge $(u, v) \in E$.

- In pseudocode, we treat the array as an attribute of the graph, leading to notation such as $G.Adj[u]$ for all the vertices adjacent to u in G.

- If $G = (V, E)$ is a directed graph, the sum of the lengths of all the adjacency lists is $|E|$, since an edge of the form (u, v) is represented by having v appear in $G.Adj[u]$.

Graphs I
22.1 Representation of graphs

Adjacency lists

- If $G = (V, E)$ is an undirected graph, the sum of the lengths of all the adjacency lists is $2|E|$, since an undirected edge of the form (u, v) is represented by having v appear in $G.Adj[u]$ and vice versa.

- For both directed and undirected graphs, the adjacency-list representation uses $\Theta(|V| + |E|)$ space.

- Listing all vertices adjacent to vertex u takes $\Theta(\deg(u))$ time.

- Determining whether $(u, v) \in E$ takes $O(\deg(u))$ time.

- Adjacency lists can be adapted to represent a weighted graph $G = (V, E)$, with edge-weight function $w : E \to \mathbb{R}$.

Graphs I
22.1 Representation of graphs

Example (Undirected graph)
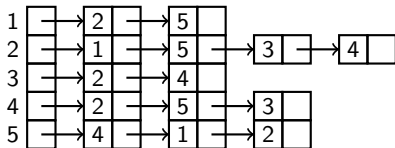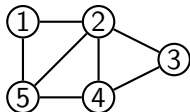
Example (Directed graph)
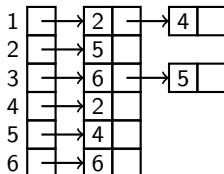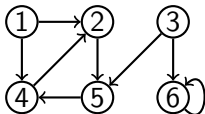
Graphs I
22.1 Representation of graphs

Adjacency matrix

- The adjacency matrix representation of a graph $G = (V, E)$ with the vertices numbered $1, 2, \ldots, |V|$ (in some arbitrary manner) is a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

- For both directed and undirected graphs, the adjacency-matrix representation uses $\Theta(|V|^2)$ space, independent of $|E|$.

- Listing all vertices adjacent to vertex u takes $\Theta(|V|)$ time.

- Determining whether $(u, v) \in E$ takes $\Theta(1)$ time.

- Adjacency matrices can be adapted to represent a weighted graph $G = (V, E)$, with edge-weight function $w : E \to \mathbb{R}$.

Graphs I
22.1 Representation of graphs

Example (Undirected graph)
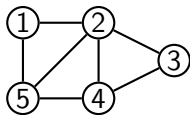
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Example (Directed graph)
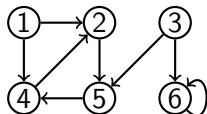
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Representing attributes

- Most graph algorithms need to maintain attributes for vertices and/or edges.
- We indicate these attributes using dot notation.
  - $v.d$ for attribute d of vertex v.
  - $(u, v).f$ for attribute f of edge (u, v).
- When representing a graph $G = (V, E)$ using adjacency lists, vertex attributes can be represented in additional arrays, such as an array $d[1, \ldots, |V|]$ to store $v.d$ in array entry $d[v]$.

Overview

- The strategy used by depth-first search is to search deeper in the graph whenever possible.
- Depth-first search colors vertices during the search to indicate their state.
  white undiscovered.
  gray discovered, but not finished (not done exploring from it).
  black finished (have found everything reachable from it).
- Depth-first search timestamps each vertex.
  $v.d$ when v is first discovered (and grayed).
  $v.f$ when v is finished (and blackened).
- Depth-first search can also compute the predecessor $v.\pi$ of each vertex v.

Graphs I
22.3 Depth-first search

- The discovery time and finishing time timestamps are integers such that $1 \leqslant u.d < u.f \leqslant 2|V|$ for every vertex u.
- The input graph G may be directed or undirected.
- The variable time is a global variable used for timestamping.

```
DFS(G)
    for each vertex u ∈ G.V do
        u.color = WHITE
    time = 0
    for each vertex u ∈ G.V do
        if u.color = WHITE then
            DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
    time = time + 1
    u.d = time
    u.color = GRAY                              discover u
    for each vertex v ∈ G.Adj[u] do            explore (u, v)
        if v.color = WHITE then
            DFS-VISIT(G, v)
    time = time + 1
    u.f = time
    u.color = BLACK                             finish u
```

- Depth-first search takes $\Theta(|V| + |E|)$ time, because the loops over all the vertices take $\Theta(|V|)$ time, exclusive of DFS-VISIT, which is called once for each (white) vertex, the loop in DFS-VISIT executes once for each adjacent vertex, and the sum of the lengths of all the adjacency lists is $\Theta(|E|)$.
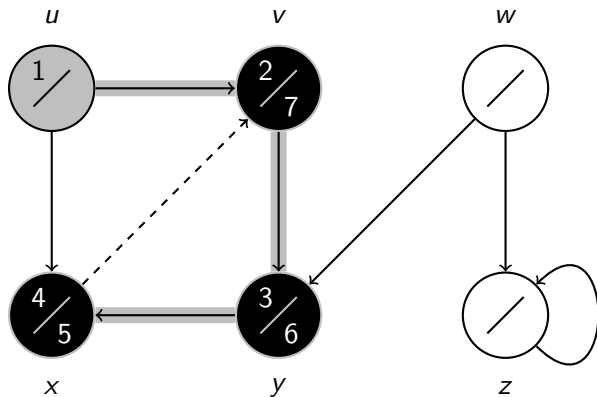
Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Graphs I
22.3 Depth-first search

Theorem (Parenthesis theorem)

For any two vertices u and v, exactly one of the following holds:

1. *The intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint and neither u nor v is a descendant of the other.*
2. *$[u.d, u.f]$ is contained in $[v.d, v.f]$ and u is a descendant of v.*
3. *$[v.d, v.f]$ is contained in $[u.d, u.f]$ and v is a descendant of u.*
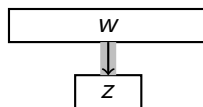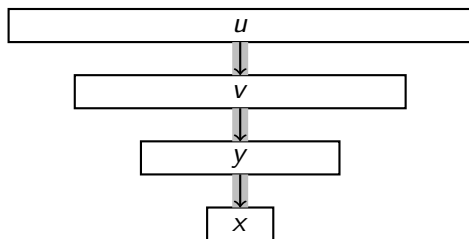
Corollary (Nesting of descendant intervals)

Vertex v is a proper descendant of vertex u if and only if $u.d < v.d < v.f < u.f$.

Theorem (White-path theorem)

Vertex v is a descendant of vertex u if and only if at time u.d, there is a path from u to v consisting entirely of white vertices.

```
1    2    3    4    5    6    7    8    9   10   11   12

(    (    (    (    )    )    )    )    (    (    )    )
```

Classification of edges

- Edge (u, v) is a tree edge if vertex v was first discovered by exploring edge (u, v).
- Edge (u, v) is a back edge if vertex v is an ancestor of vertex u.
- Edge (u, v) is a forward edge if it is not a tree edge and vertex v is a descendant of vertex u.
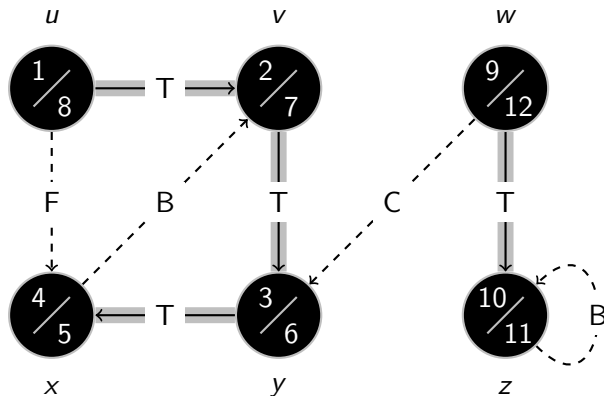- Edge (u, v) is a cross edge if it is not a tree edge, a back edge, or a forward edge.

Theorem
In a depth-first search of an undirected graph, every edge is either a tree edge or a back edge.

Example

Graphs I
22.3 Depth-first search

Exercise 22.3-7
Rewrite the procedure DFS, using a stack to eliminate recursion.

Solution to Exercise 22.3-7

```
DFS(G)
    for each vertex u ∈ G.V do
        u.color = WHITE
    time = 0
    for each vertex u ∈ G.V do
        if u.color = WHITE then
            DFS-VISIT(G, u)
```

Solution to Exercise 22.3-7 (Cont'd)

```
DFS-VISIT(G, u)
    time = time + 1; u.d = time; u.color = GRAY
    S = ∅
    PUSH(S, u)
    while S ≠ ∅ do
        u = POP(S)
        for each v ∈ G.Adj[u] do
            if v.color = WHITE then
                time = time + 1; v.d = time; v.color = GRAY
                PUSH(S, v)
        time = time + 1; u.f = time; u.color = BLACK
```
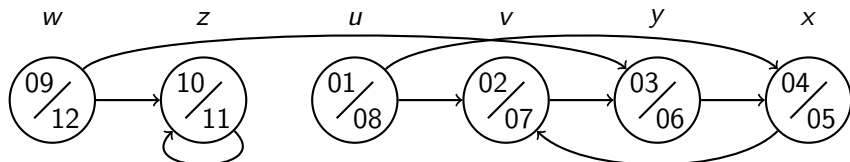
Graphs I
22.4 Topological sort

- A directed acyclic graph is a directed graph with no cycles.
- A topological sort of a directed acyclic graph $G = (V, E)$ is a linear ordering of V such that if $(u, v) \in E$, then u appears before v in the ordering.

Example

TOPOLOGICAL-SORT(G)
    call DFS(G) to compute finishing times $v.f$ for all $v \in G.V$
    output vertices in order of decreasing finishing times

Correctness

- It suffices to show that, if $(u, v) \in E$, then $v.f < u.f$.
- Consider any edge (u, v) being explored by DFS(G).
    - v cannot be gray, since then v would be an ancestor of u and (u, v) would be a back edge.
    - If v is white, v becomes a descendant of u and, by the parenthesis theorem, $v.f < u.f$.
    - If v is black, v has already been finished, and $v.f$ has already been set. We are still exploring from u and, when we assign a finishing time to u, we will have $v.f < u.f$.

Lemma
A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.

Proof.
Suppose there is a back edge (u, v). Then vertex v is an ancestor of vertex u. Thus, G contains a path from vertex v to vertex u, and the back edge (u, v) completes a cycle.

Conversely, suppose that G contains a cycle c. Let v be the first vertex discovered in c, and let (u, v) be the preceding edge in c. At time $v.d$, the vertices of c form a path of white vertices from v to u. By the white-path theorem, vertex u is a descendant of vertex v. Therefore, (u, v) is a back edge.

Exercise 22.4-5

Another way to perform topological sorting on a directed acyclic graph $G = (V, E)$ is to repeatedly find a vertex of in-degree 0, output it, and remove it and all of its outgoing edges from the graph. Explain how to implement this idea so that it runs in time $O(|V| + |E|)$. What happens to this algorithm if G has cycles?

Graphs I
22.4 Topological sort

Solution to Exercise 22.4-5

```
TOPOLOGICAL-SORT(G)
    for each vertex u ∈ G.V do
        u.indegree = 0
    time = 0
    for each vertex u ∈ G.V do
        for each v ∈ G.Adj[u] do
            v.indegree = v.indegree + 1
    Q = ∅
    for each vertex u ∈ G.V do
        if u.indegree = 0 then
            ENQUEUE(Q, u)
```

Solution to Exercise 22.4-5 (Cont'd)

```
while Q ≠ ∅ do
    u = DEQUEUE(Q)
    output u
    for each v ∈ G.Adj[u] do
        v.indegree = v.indegree − 1
        if v.indegree = 0 then
            ENQUEUE(Q, v)
for each vertex u ∈ G.V do
    if u.indegree ≠ 0 then
        report that there is a cycle
```

Lecture 8
Graphs II

Graphs II

Contents

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein.
  Introduction to Algorithms. The MIT Press, 3rd edition, 2009

Overview

- The strategy used by breadth-first search is to expand the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.

- Breadth-first search computes the distance (smallest number of edges) $v.d$ from a distinguished source vertex s to every vertex v that is reachable from s.

- Breadth-first search can color vertices during the search to indicate their state.

- Breadth-first search can also compute the predecessor $v.\pi$ of each vertex v.

- The input graph G may be directed or undirected.

```
BFS(G, s)
    for each vertex u ∈ G.V − {s} do
        u.color = WHITE
        u.d = ∞
    s.color = GRAY
    s.d = 0
    Q = ∅
    ENQUEUE(Q, s)
    while Q ≠ ∅ do
        u = DEQUEUE(Q)
        for each v ∈ G.Adj[u] do
            if v.color = WHITE then
                v.color = GRAY
                v.d = v.d + 1
                ENQUEUE(Q, v)
        u.color = BLACK
```

- All vertices start out white and may later become gray and then black.
- Gray and black vertices have been discovered.
- Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.
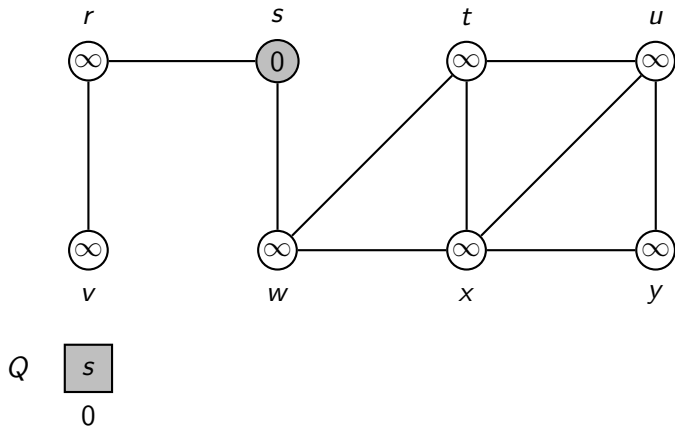
Analysis

- Breadth-first search takes $\Theta(|V| + |E|)$ time.
    - $O(|V|)$ because each vertex is enqueued at most once.
    - $O(|E|)$ because each vertex is dequeued at most once and each edge (u, v) is examined only once when u is dequeued.
- Breadth-first search takes time linear in the size of the adjacency-list representation of G.

Example

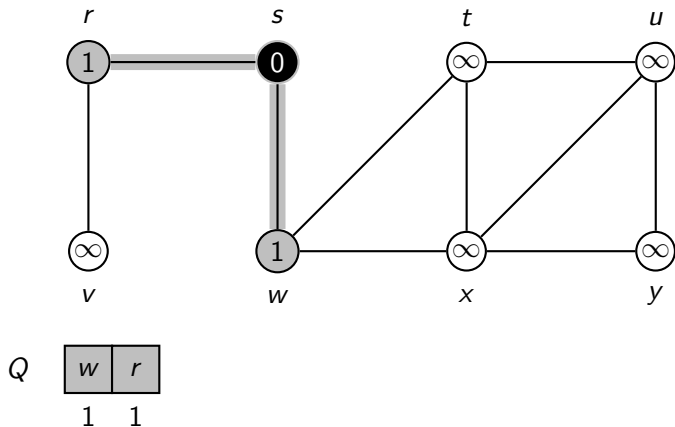Example

Graphs II
22.2 Breadth-first search

Example

Example

Example

Example

Example

Example

Graphs II
22.2 Breadth-first search

Example

$Q \quad \emptyset$
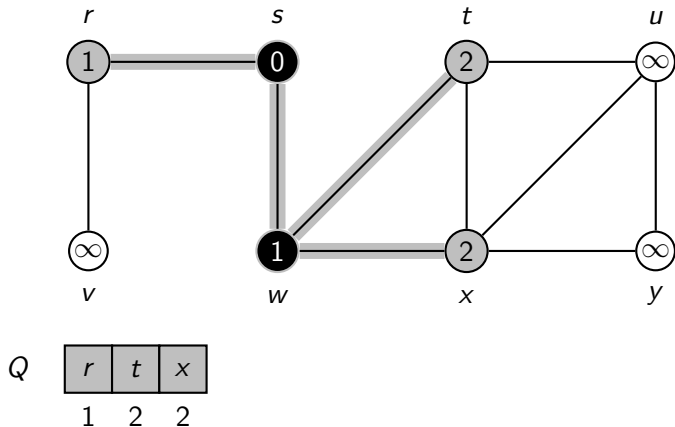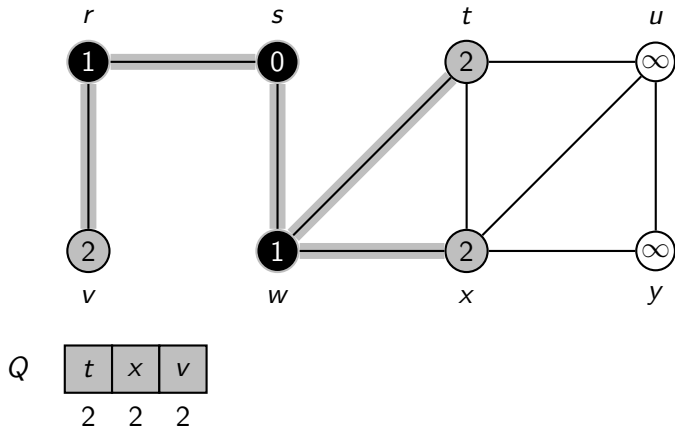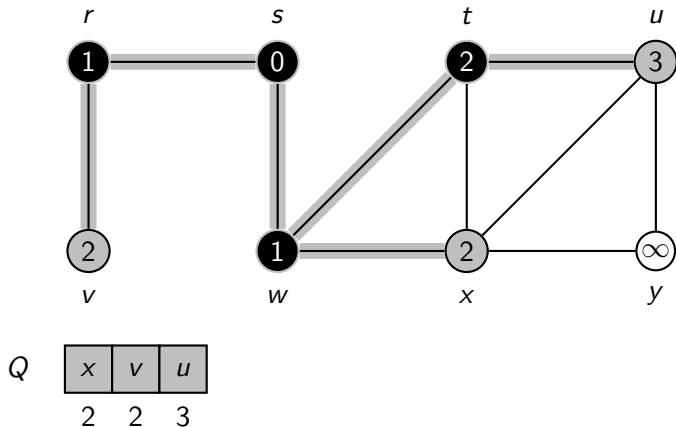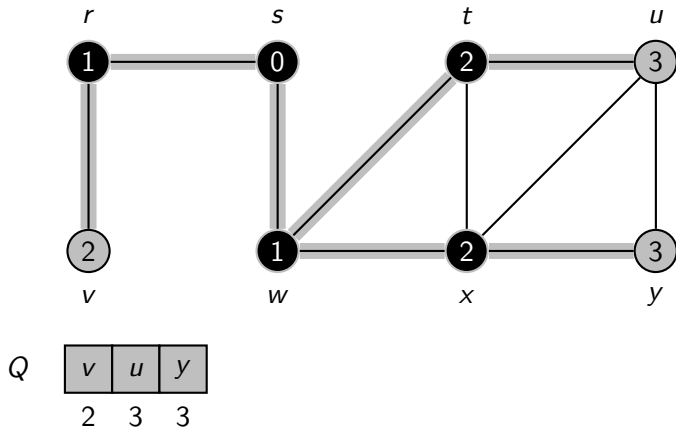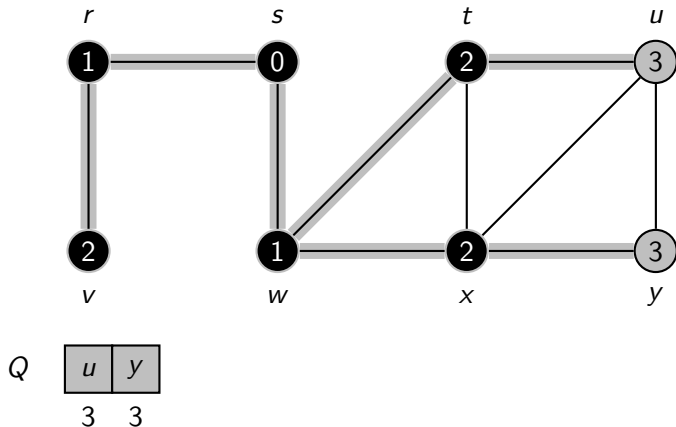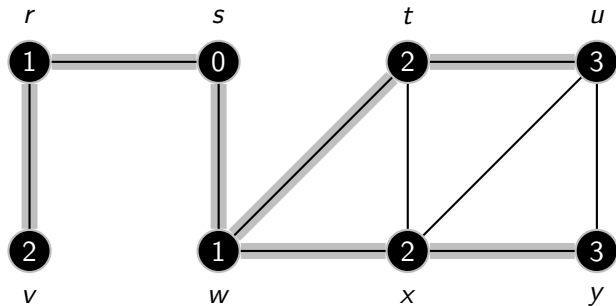
Exercise 22.2-4

What is the running time of BFS if we represent its input graph by an adjacency matrix and modify the algorithm to handle this form of input?

Solution to Exercise 22.2-4

Since the procedure scans the adjacencies of each vertex at most once, the total time spent in scanning adjacencies is $O(|V|^2)$ and the total running time of the BFS procedure is $O(|V| + |V|^2) = O(|V|^2)$. Thus, BFS runs in time linear in the size of the adjacency-matrix representation of G.

Overview

- Given a directed graph $G = (E, V)$ with weight function $w : E \to \mathbb{R}$, find a shortest path (a path of smallest total weight) from a given source vertex $s \in V$ to each vertex $v \in V$.

- BFS is a single-source shortest-paths algorithm that works on unweighted graphs (where each edge has unit weight).

Graphs II
24.3 Dijkstra's algorithm

Example
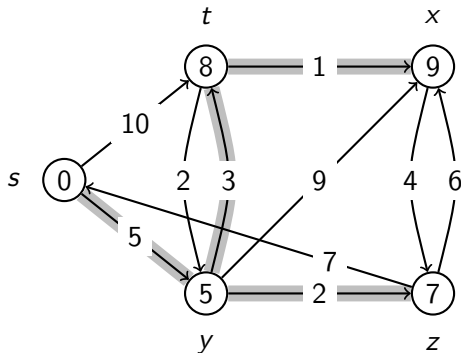
Graphs II
24.3 Dijkstra's algorithm

Lemma
Any subpath of a shortest path is a shortest path.

Proof.

- Suppose $p = u \rightsquigarrow x \rightsquigarrow y \rightsquigarrow v$ is a shortest path from u to v.
- Then $\delta(u, v) = w(p) = w(u \rightsquigarrow x) + w(x \rightsquigarrow y) + w(y \rightsquigarrow v)$.
- Now suppose there exists a shorter path $x \rightsquigarrow y$.
- Then $w(x \rightsquigarrow y) < w(x \rightsquigarrow y)$.
- Construct $p' = u \rightsquigarrow x \rightsquigarrow y \rightsquigarrow v$.
- Then $\begin{aligned} w(p') &= w(u \rightsquigarrow x) + w(x \rightsquigarrow y) + w(y \rightsquigarrow v) \\ &< w(u \rightsquigarrow x) + w(x \rightsquigarrow y) + w(y \rightsquigarrow v) \\ &= w(p) \end{aligned}$
- Contradicts the assumption that p is a shortest path.

Negative-weight edges

- If the graph $G = (V, E)$ contains no negative-weight cycles reachable from the source $s \in V$, then for all $v \in V$, the shortest-path weight $\delta(s, v)$ remains well defined, even if it has a negative value.
- If the graph contains a negative-weight cycle reachable from the source $s \in V$, shortest-path weights are not well-defined.
  - No path from s to a vertex on the cycle can be a shortest path.
  - We can always find a path with lower weight by following the proposed "shortest" path and then traversing the negative-weight cycle.
  - If there is a negative-weight cycle on some path from s to v, we define $\delta(s, v) = -\infty$.

Graphs II
24.3 Dijkstra's algorithm

Cycles

- Shortest paths cannot contain cycles.
    - Cannot contain negative-weight cycles.
      Shortest-path weights are not well-defined.
    - Cannot contain positive-weight cycles.
      We can get a shorter path by omitting the cycle.
    - Can contain zero-weight cycles.
      We can still get a shortest path by omitting the cycle.
- Assume that shortest paths are simple (they have no cycles).
- Any acyclic path in a graph $G = (V, E)$ contains at most $|V|$ distinct vertices and at most $|V| - 1$ edges.

Graphs II
24.3 Dijkstra's algorithm

Representing shortest paths

- Maintain for each vertex $v \in V$ a predecessor $v.\pi$ that is either another vertex or NIL.
- The chain of predecessors originating at a vertex v runs backwards along a shortest path from s to v.

```
PRINT-PATH(G, s, v)
    if v = s then
        print s
    else if v.π = NIL then
        print "no path from" s "to" v "exists"
    else
        PRINT-PATH(G, s, v.π)
        print v
```

Graphs II
24.3 Dijkstra's algorithm

Relaxation

- For each vertex $v \in V$, the shortest-path estimate $v.d$ is an upper bound on the weight of a shortest path from source s to v.

```
INITIALIZE-SINGLE-SOURCE(G, s)
    for each vertex v ∈ G.V do
        v.d = ∞
        v.π = NIL
    s.d = 0
```

- After initialization, we have $v.\pi = \text{NIL}$ for all $v \in V$, $s.d = 0$, and $v.d = \infty$ for all $v \in V - \{s\}$.

Relaxation

- The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path estimate $v.d$ by going through u and taking (u, v).

```
RELAX(u, v, w)
    if v.d > u.d + w(u, v) then
        v.d = u.d + w(u, v)
        v.π = u
```

- After initialization, shortest-path algorithms repeatedly relax edges.

Properties of shortest paths and relaxation

Lemma (Triangle inequality)

For any edge $(u, v) \in E$, we have $\delta(s, v) \leqslant \delta(s, u) + w(u, v)$.

Lemma (Upper-bound property)

We always have $v.d \geqslant \delta(s, v)$ for all vertices $v \in V$, and once $v.d = \delta(s, v)$, it never changes.

Lemma (No-path property)

If $\delta(s, v) = \infty$, then we always have $v.d = \infty$.

Properties of shortest paths and relaxation

Lemma (Convergence property)

If $s \rightsquigarrow u \rightarrow v$ is a shortest path, and if $u.d = \delta(s, u)$ before relaxing (u, v), then $v.d = \delta(s, v)$ at all times afterward.

Lemma (Path-relaxation property)

If we relax the edges of a shortest path $s = v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_k$ in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, even intermixed with other relaxations, then $v_k.d = \delta(s, v_k)$.

- Assume all edge weight are non-negative.
- Maintain a set S of vertices whose final shortest-path weights are determined.
- Maintain a set $V - S$ of vertices in a min-priority queue Q.

```
DIJKSTRA(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    S = ∅
    Q = G.V
    while Q ≠ ∅ do
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}
        for each v ∈ G.Adj[u] do
            RELAX(u, v, w)
```

Graphs II
24.3 Dijkstra's algorithm
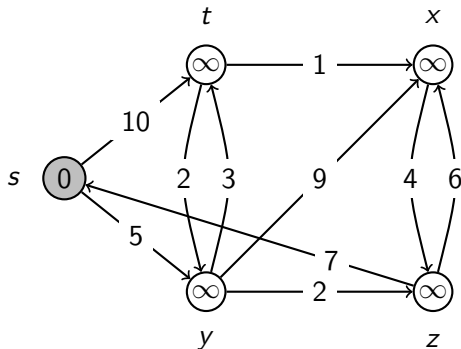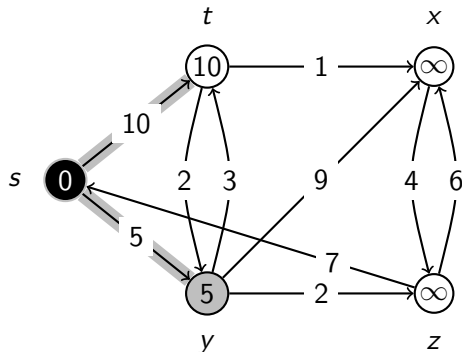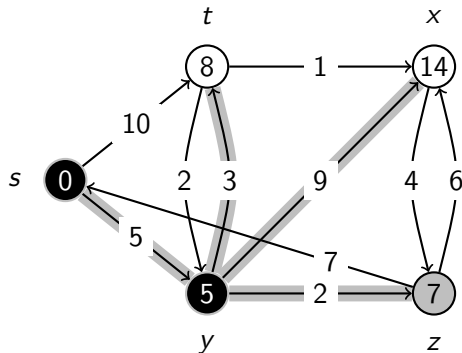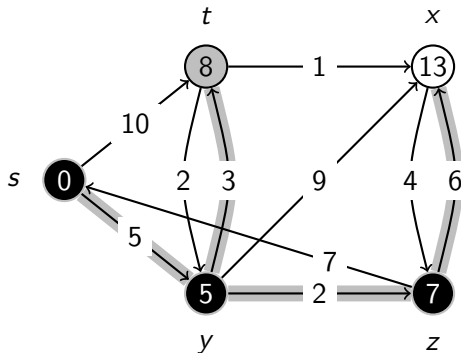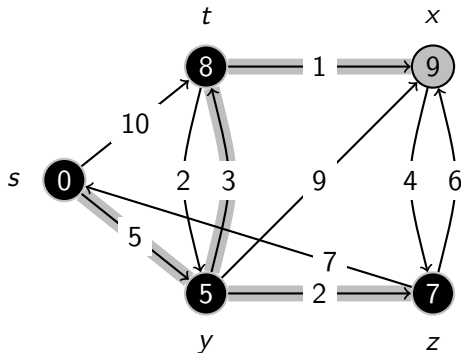
Example

Example

Example

Graphs II
24.3 Dijkstra's algorithm

Example

Example

Example
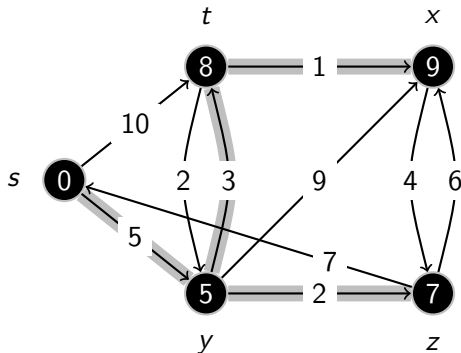
Correctness

- Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V - S$ to add to set S.

- Each time it adds a vertex u to set S, we have $u.d = \delta(s, u)$.

Theorem (CLRS Theorem 24.6)

Dijkstra's algorithm, run on a directed graph $G = (V, E)$ with non-negative weight function $w : E \to \mathbb{R}$ and source vertex $s \in V$, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

Analysis

- Dijkstra's algorithm takes $O((|V| + |E|) \log |V|)$ time, by implementing the min-priority queue with a min-heap.
  - $O(|V|)$ time to build the min-heap.
  - $|V|$ min-heap EXTRACT-MIN operations, each taking $O(\log |V|)$ time.
  - At most $|E|$ min-heap DECREASE-KEY operations, each taking $O(\log |V|)$ time.

Graphs II
23.1 Growing a minimum spanning tree

- Given a connected, undirected graph $G = (E, V)$ with weight function $w : E \to \mathbb{R}$, find a minimum spanning tree (a spanning tree of smallest total weight) for G.
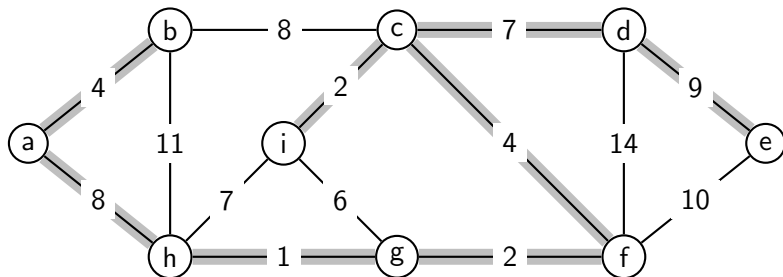
Example

Graphs II
23.1 Growing a minimum spanning tree

- Given a connected, undirected graph $G = (E, V)$ with weight function $w : E \to \mathbb{R}$, find a minimum spanning tree (a spanning tree of smallest total weight) for G.

Example
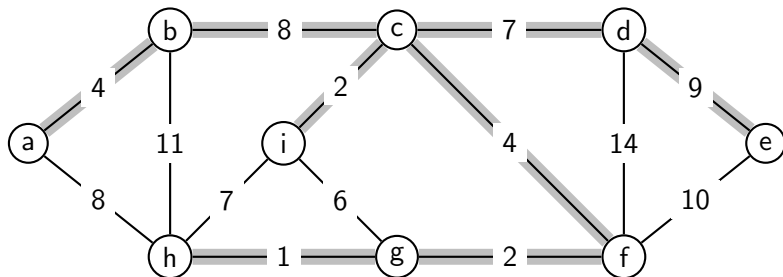
Graphs II
23.1 Growing a minimum spanning tree

- Maintain the loop invariant that, prior to each iteration, A is a subset of some minimum spanning tree.

- An edge (u, v) is safe for A if $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree.

```
GENERIC-MST(G, w)
    A = ∅
    while A does not form a spanning tree do
        find an edge (u, v) that is safe for A
        A = A ∪ {(u, v)}
    return A
```

Graphs II
23.1 Growing a minimum spanning tree

Correctness

- A cut $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V.
- An edge $(u, v) \in E$ crosses the cut $(S, V - S)$ if one endpoint is in S and the other is in $V - S$.
- A cut respects a set A of edges if no edge in A crosses the cut.
- An edge is a light edge crossing a cut if its weight is the minimum of any edge crossing the cut.

Theorem (CLRS Theorem 23.1)

Let A be a subset of some minimum spanning tree for $G = (V, E)$, let $(S, V - S)$ be a cut of G that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Then, edge (u, v) is safe for A.

Prim's algorithm

- Special case of the generic minimum-spanning-tree method.
- Much like Dijkstra's algorithm.
- The edges in the set A always form a tree, that starts from an arbitrary root vertex r and grows until the tree A spans all the vertices in V.
- Each step adds to the tree A a light edge that connects A to an isolated vertex, and the edge is safe for A.
- When the algorithm terminates, the edges in A form a minimum spanning tree.
- Need a fast way to find a light edge to add to the tree formed by the edges in A.

Prim's algorithm

- Use a min-priority queue Q to keep all vertices that are not in the tree.
- For each vertex $v \in V$, the attribute $v.key$ is the minimum weight of any edge connecting v to a vertex not in the tree.
- Set $v.key = \infty$ if there is no such edge.

Prim's algorithm

```
MST-PRIM(G, w, r)
    for each vertex u ∈ G.V do
        u.key = ∞
        u.π = NIL
    r.key = 0
    Q = G.V
    while Q ≠ ∅ do
        u = EXTRACT-MIN(Q)
        for each v ∈ G.Adj[u] do
            if v ∈ Q and w(u, v) < v.key then
                v.π = u
                v.key = w(u, v)
```

Graphs II
23.2 The algorithms of (Kruskal and) Prim

Prim's algorithm

```
MST-PRIM(G, w, r)
    for each vertex u ∈ G.V do
        u.key = ∞
        u.π = NIL
        INSERT(Q, u)
    DECREASE-KEY(Q, r, 0)
    while Q ≠ ∅ do
        u = EXTRACT-MIN(Q)
        for each v ∈ G.Adj[u] do
            if v ∈ Q and w(u, v) < v.key then
                v.π = u
                DECREASE-KEY(Q, v, w(u, v))
```
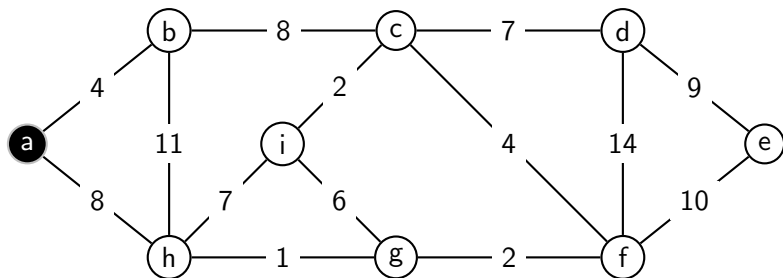
Example

Example

Example

Example

Example

Example

Example

Example

Example
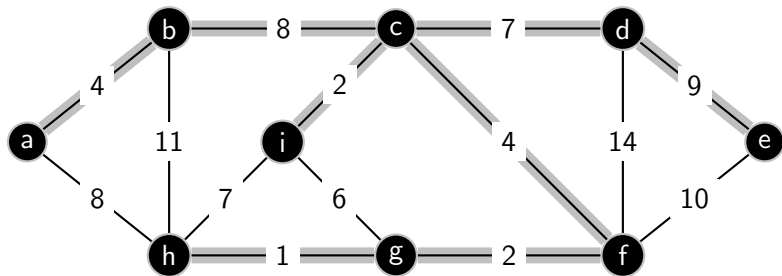
Graphs II
23.2 The algorithms of (Kruskal and) Prim

Prim's algorithm

- Implement the test for membership in Q by keeping a bit for each vertex $v \in V$ that tells whether or not $v \in Q$.

Analysis

- Prim's algorithm takes $O(|E| \log |V|)$ time, by implementing the min-priority queue with a min-heap.
    - $O(|V|)$ time to build the min-heap.
    - $|V|$ min-heap EXTRACT-MIN operations, each taking $O(\log |V|)$ time.
    - $|E|$ min-heap DECREASE-KEY operations, each taking $O(\log |V|)$ time.
    - $|V| + |E| = O(|E|)$ because the graph is connected.

Exercise 23.2-2

Suppose that we represet the graph $G = (V, E)$ as an adjacency matrix. Give a simple implementation of Prim's algorithm for this case that runs in $O(|V|^2)$ time.

Graphs II
23.2 The algorithms of (Kruskal and) Prim

Solution to Exercise 23.2-2

```
MST-PRIM(G, r)
    for each vertex u ∈ G.V do
        u.key = ∞
        u.pred = NIL
    r.key = 0
    Q = G.V
    while Q ≠ ∅ do
        u = EXTRACT-MIN(Q)
        for each vertex v ∈ G.V do
            if A[u, v] ≠ 0 and v ∈ Q and A[u, v] < v.key then
                v.pred = u
                v.key = A[u, v]
```

Lecture 9
Exhaustive Search and Generation I

Exhaustive Search and Generation I

Contents

- Exercise: All subsets
- Exercise: All permutations

Reading

- R. E. Neapolitan. *Foundations of Algorithms*. Jones & Bartlett Learning, 5th edition, 2015

Exhaustive Search and Generation I
The backtracking technique

- Backtracking is a technique is used to solve problems in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion.

- The backtracking technique is a modified depth-first search of a state space tree, checking whether each node is promising, and, if it is non-promising, going back to the parent of the node and proceeding with the search on the next child.

  - A node is non-promising if when visiting the node we determine that it cannot possibly lead to a solution. Otherwise, it is promising.
  - Going back to the parent of the node is called pruning the state space tree.
  - The subtree consisting of the visited nodes is called the pruned state space tree.

Exhaustive Search and Generation I
The backtracking technique

- A backtracking algorithm is identical to a depth-first search, except that the children of a node are visited only when the node is promising and there is not a solution at the node.

- In some backtracking algorithms a solution can be found before reaching a leaf in the state space tree.

- A general algorithm for backtracking is as follows.

```
BACKTRACK(u)
   if PROMISING(u) then
      if SOLUTION(u) then
         write the solution
      else
         for each child v of u do
            BACKTRACK(v)
```

Exhaustive Search and Generation I
The backtracking technique

- You may have observed that there is some inefficiency in the general algorithm for backtracking.
- We check whether a partial solution is promising after passing it to the procedure. (This means that non-promising partial solutions are unnecessarily placed on the recursion stack.)
- We could avoid this by checking whether a partial solution is promising before passing it.

```
BACKTRACK(u)
   for each child v of u do
      if PROMISING(v) then
         if SOLUTION(v) then
            write the solution
         else
            BACKTRACK(v)
```

Exhaustive Search and Generation I
The n-queens problem

> Backtracking is a technique is used to solve problems in which a **sequence** of objects is chosen from a specified **set** so that the sequence satisfies some **criterion**.

- The goal in the n-queens problem is to position n queens on an $n \times n$ chessboard so that no two queens threaten each other; that is, no two queens may be in the same row, column, or diagonal.

- The sequence in this problem is the n positions in which the queens are placed.

- The set for each choice is the n^2 possible positions on the chessboard.

- The criterion is that no two queens can threaten each other.

Exhaustive Search and Generation I
The *n*-queens problem

Exhaustive Search and Generation I
The *n*-queens problem

Exhaustive Search and Generation I
The *n*-queens problem

Exhaustive Search and Generation I
The *n*-queens problem

Exhaustive Search and Generation I
The *n*-queens problem

Exhaustive Search and Generation I
The *n*-queens problem

Exhaustive Search and Generation I
The *n*-queens problem

Exhaustive Search and Generation I
The *n*-queens problem

Exhaustive Search and Generation I
The *n*-queens problem

Exhaustive Search and Generation I
The *n*-queens problem

Exhaustive Search and Generation I
The *n*-queens problem

Exhaustive Search and Generation I
The *n*-queens problem

Exhaustive Search and Generation I
The *n*-queens problem

Exhaustive Search and Generation I
The *n*-queens problem

Exhaustive Search and Generation I
The *n*-queens problem

Exhaustive Search and Generation I
The *n*-queens problem

Exhaustive Search and Generation I
The *n*-queens problem

- A portion of the pruned state space tree for $n = 4$.
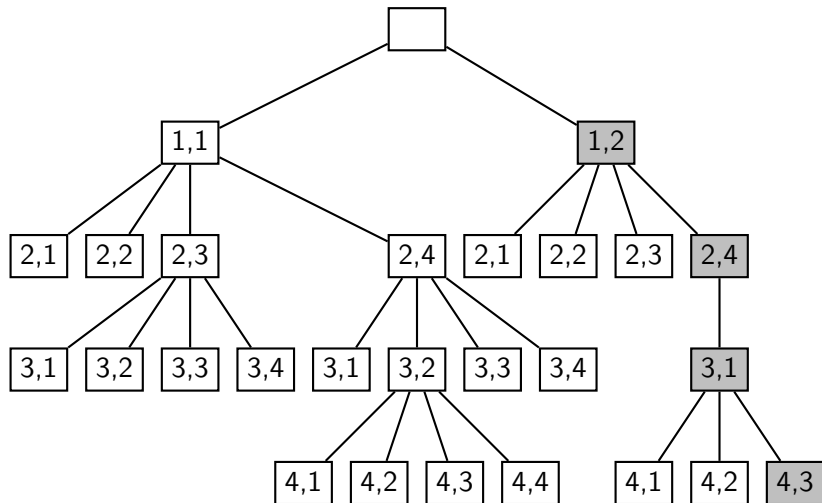
Exhaustive Search and Generation I
The *n*-queens problem

- Let *col*[*i*] be the column where the queen in the *i*th row is placed.

```
N-QUEENS(0)
```

```
N-QUEENS(i)
    if PROMISING(i) then
        if i = n then
            write col[1], . . . , col[n]
        else
            for j = 1 to n do
                col[i + 1] = j
                N-QUEENS(i + 1)
```

Exhaustive Search and Generation I
The *n*-queens problem

```
PROMISING(i)
    for j = 1 to i − 1 do
        if col[i] = col[j] or abs(col[i] − col[j]) = i − j then
            return FALSE
    return TRUE
```

Exhaustive Search and Generation I
The sum-of-subsets problem

> Backtracking is a technique is used to solve problems in which a **sequence** of objects is chosen from a specified **set** so that the sequence satisfies some **criterion**.

- Given n positive integers (weights) w_i and a positive integer W, the goal in the sum-of-subsets problem is to find a subset of the integers that sum to W.

- The sequence in this problem is the n integers.

- The set for each choice is to include or to exclude the integer.

- The criterion is that the chosen integers sum to W.

Exhaustive Search and Generation I
The sum-of-subsets problem

- The pruned state space tree for $n = 4$, $W = 13$, and $w_1 = 3$, $w_2 = 4$, $w_3 = 5$, $w_4 = 6$.

$w_1 = 3$

$w_2 = 4$

$w_3 = 5$

$w_4 = 6$

Exhaustive Search and Generation I
The sum-of-subsets problem

- Let *include*[i] be true if and only if $w[i]$ is to be included.

```
SUM-OF-SUBSETS(0, 0)
```

```
SUM-OF-SUBSETS(i, w)
    if PROMISING(i, w) then
        if w = W then
            write include[1], ..., include[i]
        else
            include[i + 1] = TRUE
            SUM-OF-SUBSETS(i + 1, w + w[i + 1])
            include[i + 1] = FALSE
            SUM-OF-SUBSETS(i + 1, w)
```

```
PROMISING(i, w)
    return w ⩽ W
```

Exhaustive Search and Generation I
Graph coloring

Backtracking is a technique is used to solve problems in which a **sequence** of objects is chosen from a specified **set** so that the sequence satisfies some **criterion**.

- The goal in the graph coloring problem is to color an undirected graph using at most m different colors, so that no two adjacent vertices are the same color.
- The sequence in this problem is the vertices of the graph.
- The set for each choice is the m possible colors.
- The criterion is that no two adjacent vertices are the same color.

Exhaustive Search and Generation I
Graph coloring

- A portion of the pruned state space tree for $m = 3$,
  $V = \{v_1, v_2, v_3, v_4\}$, and
  $E = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_3), (v_3, v_4)\}$.
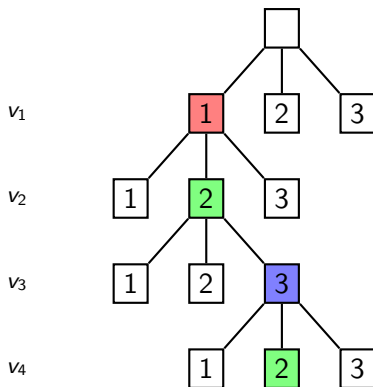
Exhaustive Search and Generation I
Graph coloring

- A node is nonpromising if a vertex that is adjacent to the vertex being colored at the node has already been colored the color that is being used at the node.
- Let $color[i]$ be the color assigned to the ith vertex.

```
M-COLORING(0)
```

```
M-COLORING(i)
    if PROMISING(i) then
        if i = n then
            write color[1], ..., color[n]
        else
            for c = 1 to m do
                color[i + 1] = c
                M-COLORING(i + 1)
```

Exhaustive Search and Generation I
Graph coloring

- The undirected graph is represented by an adjacency matrix $G.Adj[1..n, 1..n]$.

```
PROMISING(i)
   for j = 1 to i − 1 do
      if G.Adj[i, j] and color[i] = color[j] then
         return FALSE
   return TRUE
```

Exhaustive Search and Generation I
Graph coloring

Exercise CLRS 34-3.a

Give an efficient algorithm to determine a 2-coloring of a graph, if one exists.

Solution

```
2-COLORING(G)
    for each vertex u ∈ G.V do
        u.color = WHITE
    for each vertex u ∈ G.V do
        if u.color = WHITE then
            u.color = BLUE
            if not 2-COLORING(G, u) then
                return FALSE
    return TRUE
```

Exhaustive Search and Generation I
Graph coloring

```
2-COLORING(G, u)
    Q = ∅
    ENQUEUE(Q, u)
    while Q ≠ ∅ do
        u = DEQUEUE(Q)
        for each vertex v ∈ G.Adj[u] do
            if v.color = WHITE then
                if u.color = BLUE then
                    v.color = RED
                else
                    v.color = BLUE
                ENQUEUE(Q, v)
            else
                if u.color = v.color then
                    return FALSE
    return TRUE
```

Exhaustive Search and Generation I
The hamiltonian circuits problem

Backtracking is a technique is used to solve problems in which a **sequence** of objects is chosen from a specified **set** so that the sequence satisfies some **criterion**.

- The goal in the hamiltonian circuits problem is to find a hamiltonian circuit (a path that starts at a given vertex, visits each vertex in the graph exactly once, and ends at the starting vertex) in a connected, undirected graph.

- The sequence in this problem is the vertices of the graph.

- The set for each choice is the vertices of the graph.

- The criterion is that the chosen vertices form a simple path or cycle in the graph.

Exhaustive Search and Generation I
The hamiltonian circuits problem

- A portion of the pruned state space tree for $V = \{v_1, \ldots, v_6\}$ and $E = \{(v_1, v_2), (v_1, v_4), (v_1, v_5), (v_2, v_3), (v_2, v_4), (v_2, v_5), (v_2, v_6), (v_3, v_5), (v_3, v_6), (v_4, v_5), (v_5, v_6)\}$.

Exhaustive Search and Generation I
The hamiltonian circuits problem

- The undirected graph is represented by an adjacency matrix $G.Adj[1..n, 1..n]$.
- A hamiltonian circuit is represented by a vector $I[1..n]$ of vertex indices.

```
HAMILTONIAN-CIRCUITS(0)
```

```
HAMILTONIAN-CIRCUITS(i)
    if PROMISING(i) then
        if i = n then
            write I[1], . . . , I[n]
        else
            for j = 1 to n do
                I[i + 1] = j
                HAMILTONIAN-CIRCUITS(i + 1)
```

```
PROMISING(i)
    if i = n and not G.Adj[l[n − 1], l[0]] then
        return FALSE
    else if i > 1 and not G.Adj[l[i − 1], l[i]] then
        return FALSE
    else
        for j = 1 to i − 1 do
            if l[i] = l[j] then
                return FALSE
    return TRUE
```

Exhaustive Search and Generation I
The 0-1 knapsack problem

> Backtracking is a technique is used to solve problems in which a **sequence** of objects is chosen from a specified **set** so that the sequence satisfies some **criterion**.

- Given a set of n items, each of which has a positive integer weight w_i and a positive integer profit p_i, and given a positive integer W, the goal in the 0-1 knapsack problem is to find a subset of the items that maximizes the total profit under the constraint that the total weight cannot exceed W.

- The sequence in this problem is the n items.

- The set for each choice is to include or to exclude the item.

- The criterion is that the total weight of the chosen items does not exceed W.

Exhaustive Search and Generation I
The 0-1 knapsack problem

- A portion of the pruned state space tree for $n = 4$, $W = 16$, $w_1 = 2$, $w_2 = 5$, $w_3 = 10$, $w_4 = 5$, and $p_1 = 40$, $p_2 = 30$, $p_3 = 50$, $p_4 = 10$.

$w_1 = 2 \qquad p_1 = 40$

$w_2 = 5 \qquad p_2 = 30$

$w_3 = 10 \qquad p_3 = 50$

$w_4 = 5 \qquad p_4 = 10$

Exhaustive Search and Generation I
The 0-1 knapsack problem

- Let *include*[i] be true if and only if $w[i]$ is to be included.

```
0-1-KNAPSACK(i, w, p)
    if w ⩽ W and p > max_p then
        max_p = p
        best_i = i
        best_include = include
    if PROMISING(i, w, p) then
        include[i + 1] = TRUE
        0-1-KNAPSACK(i + 1, w + w[i + 1], p + p[i + 1])
        include[i + 1] = FALSE
        0-1-KNAPSACK(i + 1, w, p)
```

Exhaustive Search and Generation I
The 0-1 knapsack problem

```
best_i = 0
max_p = 0
0-1-KNAPSACK(0, 0, 0)
write best_include[1], ..., best_include[best_i]
```

- A node is non-promising if when visiting the node we determine that it cannot possibly lead to a solution.
- In optimization problems, a node is non-promising if when visiting the node we determine that no expansion to the children can possibly lead to a solution.

```
PROMISING(i, w, p)
    return w < W
```

Exhaustive Search and Generation I
Exercise: All subsets

Exercise
Show that the number of subsets, including the empty set, of a set containing n items is 2^n.

Solution
For $0 \leqslant k \leqslant n$, the number of subsets of size k is the number of combinations of n objects taken k at a time, which is $\binom{n}{k}$. This means that the total number of subsets is

$$\sum_{k=0}^{n} \binom{n}{k} = \sum_{k=0}^{n} \binom{n}{k} 1^k 1^{n-k} = (1+1)^n = 2^n$$

The second-to-last equality is by the Binomial theorem:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$$

Exhaustive Search and Generation I
## Exercise: All subsets

> Backtracking is a technique is used to solve problems in which a **sequence** of objects is chosen from a specified **set** so that the sequence satisfies some **criterion**.

- The goal in the all subsets problem is to write all the subsets of the set of n integers $1, \ldots, n$.
- The **sequence** in this problem is the n integers $1, \ldots, n$.
- The **set** for each choice is to include or to exclude the integer.
- The **criterion** is none.

Exhaustive Search and Generation I
Exercise: All subsets

- Let $include[i]$ be true if and only if i is to be included.

```
ALL-SUBSETS(0)
```

```
ALL-SUBSETS(i)
    if PROMISING(i) then
        if i = n then
            write include[1], ..., include[n]
        else
            include[i + 1] = TRUE
            ALL-SUBSETS(i + 1)
            include[i + 1] = FALSE
            ALL-SUBSETS(i + 1)
```

```
PROMISING(i)
    return TRUE
```

Exhaustive Search and Generation I
Exercise: All permutations

> Backtracking is a technique is used to solve problems in which a **sequence** of objects is chosen from a specified **set** so that the sequence satisfies some **criterion**.

- The goal in the all permutations problem is to write all the permutations of n integers $1, \ldots, n$.

- The **sequence** in this problem is the n integers $1, \ldots, n$.

- The **set** for each choice is the n integers $1, \ldots, n$.

- The **criterion** is that the chosen integers are not repeated.

Exhaustive Search and Generation I
Exercise: All permutations

- Let $P[1], \ldots, P[n]$ be a permutation of $1, \ldots, n$.

```
P = ∅
ALL-PERMUTATIONS(0)
```

```
ALL-PERMUTATIONS(i)
    if PROMISING(i) then
        if i = n then
            write P[1], . . . , P[n]
        else
            for j = 1 to n do
                PUSH(P, j)
                ALL-PERMUTATIONS(i + 1)
                POP(P)
```

```
PROMISING(i)
    for j = 1 to i − 1 do
        if P[i] = P[j] then
            return FALSE
    return TRUE
```

Lecture 10
Exhaustive Search and Generation II

Exhaustive Search and Generation II

Contents

Reading

- R. E. Neapolitan. *Foundations of Algorithms*. Jones & Bartlett Learning, 5th edition, 2015

Exhaustive Search and Generation II
Branch-and-bound

- The branch-and-bound technique is an extension of the backtracking technique for optimization problems.

- A branch-and-bound algorithm computes a number (bound) at a node to determine whether the node is promising.

- The number is a bound on the value of the solution that could be obtained by expanding beyond the node.

- If that bound is no better than the value of the best solution found so far, the node is non-promising. Otherwise, it is promising.

- Better is smaller or larger, depending on the optimal value for the problem being a minimum or a maximum.

Exhaustive Search and Generation II
Branch-and-bound

- A general algorithm for branch-and-bound is as follows.

```
BRANCH_AND_BOUND(u, best)
   if PROMISING(u) then
       for each child v of u do
           if VALUE(v) is better than best then
               best = VALUE(v)
           if BOUND(v) is better than best then
               BRANCH_AND_BOUND(v, best)
```

Exhaustive Search and Generation II
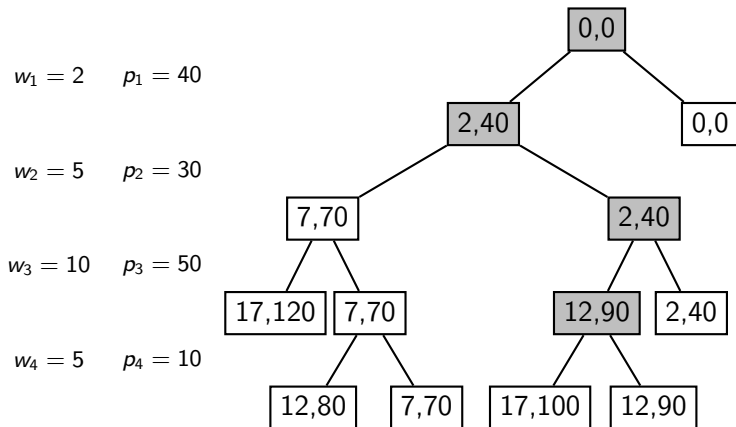## The sum-of-subsets problem

> Backtracking is a technique is used to solve problems in which a **sequence** of objects is chosen from a specified **set** so that the sequence satisfies some **criterion**.

- Given n positive integers (weights) w_i and a positive integer W, the goal in the sum-of-subsets problem is to find a subset of the integers that sum to W.

- The sequence in this problem is the n integers.

- The set for each choice is to include or to exclude the integer.

- The criterion is that the chosen integers sum to W.

Exhaustive Search and Generation II
The sum-of-subsets problem

- The pruned state space tree for $n = 4$, $W = 13$, and $w_1 = 3$, $w_2 = 4$, $w_3 = 5$, $w_4 = 6$.

Exhaustive Search and Generation II
The sum-of-subsets problem

- Let *include*[i] be true if and only if $w[i]$ is to be included.

```
SUM-OF-SUBSETS(0, 0)
```

```
SUM-OF-SUBSETS(i, w)
    if PROMISING(i, w) then
        if w = W then
            write include[1], . . . , include[i]
        else
            include[i + 1] = TRUE
            SUM-OF-SUBSETS(i + 1, w + w[i + 1])
            include[i + 1] = FALSE
            SUM-OF-SUBSETS(i + 1, w)
```

```
PROMISING(i, w)
    return w ⩽ W
```

Exhaustive Search and Generation II
The sum-of-subsets problem

- If we sort the weights in non-decreasing order before doing the search, there is an obvious sign telling us that a node is non-promising.

- Let w be the sum of the weights that have been included up to a node at the ith level.

- If w_{i+1} would bring the value of w above W, then so would any other weight following it.

- Therefore, unless $w = W$ (which means there is a solution at the node), a node at the ith level is non-promising if
$w + w_{i+1} > W$.

Exhaustive Search and Generation II
The sum-of-subsets problem

- Let *include*[i] be true if and only if $w[i]$ is to be included, let t be the total weight of the remaining weights, and let, initially, $t = \sum_{j=1}^{n} w[j]$.

SUM-OF-SUBSETS($0, 0, t$)

```
SUM-OF-SUBSETS(i, w, t)
    if PROMISING(i, w, t) then
        if w = W then
            write include[1], ..., include[i]
        else
            include[i + 1] = TRUE
            SUM-OF-SUBSETS(i + 1, w + w[i + 1], t − w[i + 1])
            include[i + 1] = FALSE
            SUM-OF-SUBSETS(i + 1, w, t − w[i + 1])
```

Exhaustive Search and Generation II
The sum-of-subsets problem

```
PROMISING(i, w, t)
    return w + t ⩾ W and (w = W or w + w[i + 1] ⩽ W)
```

Exhaustive Search and Generation II
The 0-1 knapsack problem

> Backtracking is a technique is used to solve problems in which a **sequence** of objects is chosen from a specified **set** so that the sequence satisfies some **criterion**.

- Given a set of n items, each of which has a positive integer weight w_i and a positive integer profit p_i, and given a positive integer W, the goal in the 0-1 knapsack problem is to find a subset of the items that maximizes the total profit under the constraint that the total weight cannot exceed W.

- The sequence in this problem is the n items.

- The set for each choice is to include or to exclude the item.

- The criterion is that the total weight of the chosen items does not exceed W.

Exhaustive Search and Generation II
The 0-1 knapsack problem

- A portion of the pruned state space tree for $n = 4$, $W = 16$, $w_1 = 2$, $w_2 = 5$, $w_3 = 10$, $w_4 = 5$, and $p_1 = 40$, $p_2 = 30$, $p_3 = 50$, $p_4 = 10$.

Exhaustive Search and Generation II
The 0-1 knapsack problem

- Let *include*[i] be true if and only if $w[i]$ is to be included.

```
0-1-KNAPSACK(i, w, p)
   if w ⩽ W and p > max_p then
       max_p = p
       best_i = i
       best_include = include
   if PROMISING(i, w, p) then
       include[i + 1] = TRUE
       0-1-KNAPSACK(i + 1, w + w[i + 1], p + p[i + 1])
       include[i + 1] = FALSE
       0-1-KNAPSACK(i + 1, w, p)
```

Exhaustive Search and Generation II
The 0-1 knapsack problem

```
best_i = 0
max_p = 0
0-1-KNAPSACK(0, 0, 0)
write best_include[1], . . . , best_include[best_i]
```

- A node is non-promising if when visiting the node we determine that it cannot possibly lead to a solution.
- In optimization problems, a node is non-promising if when visiting the node we determine that no expansion to the children can possibly lead to a solution.

```
PROMISING(i, w, p)
  return w < W
```

Exhaustive Search and Generation II
The 0-1 knapsack problem

- An obvious sign that a node is non-promising is that the total weight of the items included up to the node is $w \geqslant W$.

- If we sort the items in non-increasing order of profit-to-weight ratio before doing the search, there is a less obvious sign telling us that a node is non-promising.

- Let w be the sum of the weights that have been included up to a node at the ith level, and let *bound* be the total profit of all the items up to, but not including, the item that would bring the total weight above W, plus the fraction of that item allowed by the remaining weight.

- If we are able to get only a fraction of this last weight, *bound* is still an upper bound on the profit we could achieve by expanding beyond the node.

- Therefore, a node at the ith level is non-promising if $bound \leqslant max\_p$.

Exhaustive Search and Generation II
The 0-1 knapsack problem

- A portion of the pruned state space tree for $n = 4$, $W = 16$, $w_1 = 2$, $w_2 = 5$, $w_3 = 10$, $w_4 = 5$, and $p_1 = 40$, $p_2 = 30$, $p_3 = 50$, $p_4 = 10$.

$w_1 = 2 \quad p_1 = 40$

$w_2 = 5 \quad p_2 = 30$
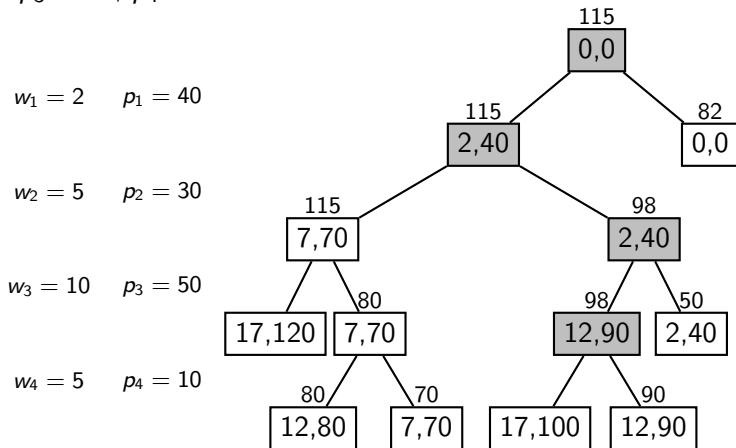
$w_3 = 10 \quad p_3 = 50$

$w_4 = 5 \quad p_4 = 10$

Exhaustive Search and Generation II
The 0-1 knapsack problem

- $115 = 0 + 40 + 30 + (16 - 7) \, 50/10$

$w_1 = 2 \quad p_1 = 40$

$w_2 = 5 \quad p_2 = 30$
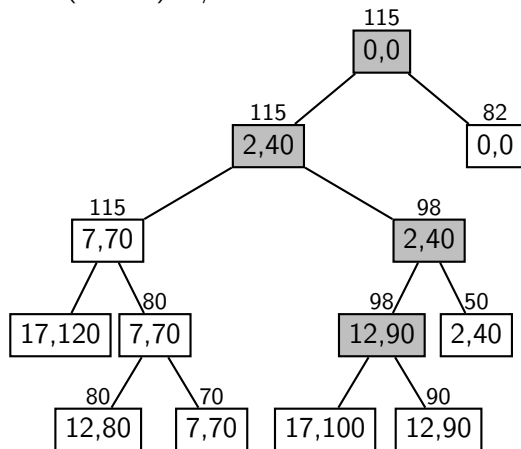
$w_3 = 10 \quad p_3 = 50$

$w_4 = 5 \quad p_4 = 10$

Exhaustive Search and Generation II
The 0-1 knapsack problem

- $115 = 40 + 30 + (16 - 7) \, 50/10$

$$w_1 = 2 \qquad p_1 = 40$$

$$w_2 = 5 \qquad p_2 = 30$$
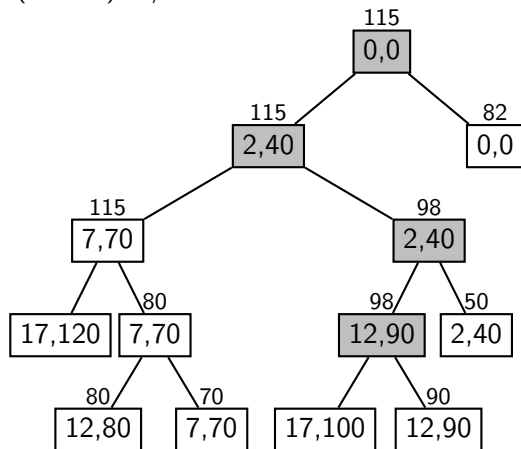
$$w_3 = 10 \qquad p_3 = 50$$

$$w_4 = 5 \qquad p_4 = 10$$

Exhaustive Search and Generation II
The 0-1 knapsack problem

- $82 = 0 + 30 + 50 + (16 - 15)\,10/5$
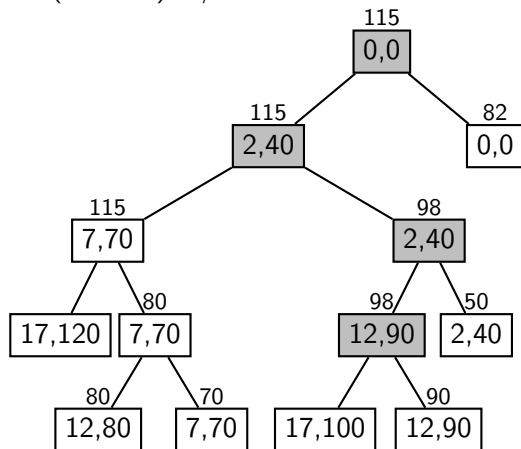
Exhaustive Search and Generation II
The 0-1 knapsack problem

- $115 = 70 + (16 - 7)\,50/10$

$w_1 = 2 \quad p_1 = 40$

$w_2 = 5 \quad p_2 = 30$
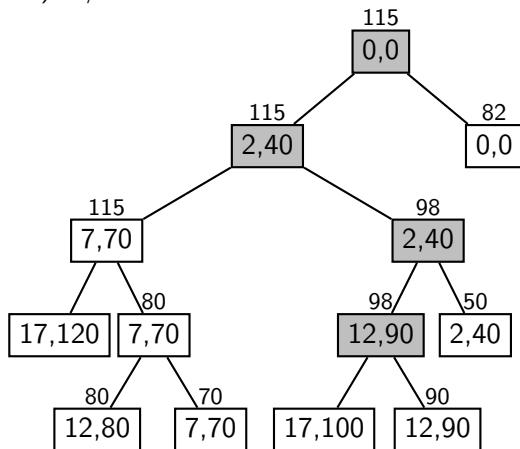
$w_3 = 10 \quad p_3 = 50$

$w_4 = 5 \quad p_4 = 10$

Exhaustive Search and Generation II
The 0-1 knapsack problem

- $98 = 40 + 50 + (16 - 12)\,10/5$
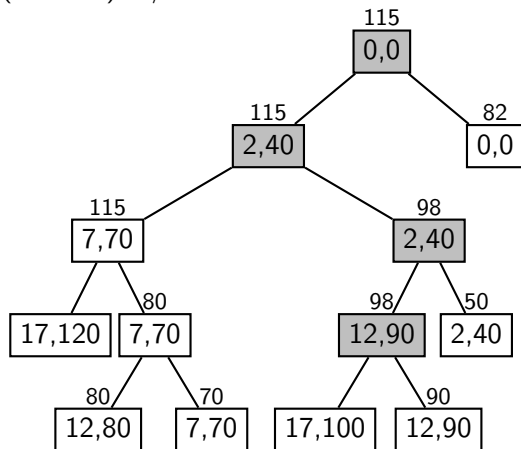
Exhaustive Search and Generation II
The 0-1 knapsack problem

- $80 = 70 + 10$

$w_1 = 2 \quad p_1 = 40$

$w_2 = 5 \quad p_2 = 30$
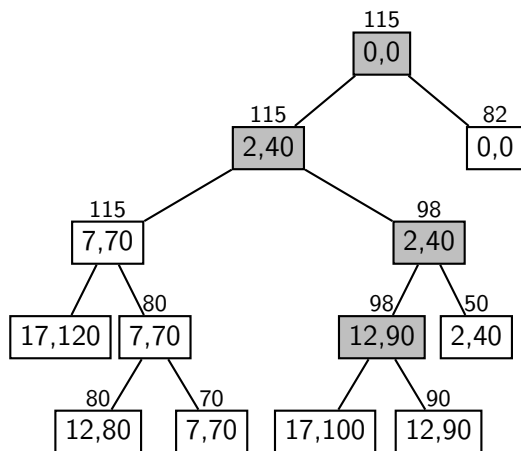
$w_3 = 10 \quad p_3 = 50$

$w_4 = 5 \quad p_4 = 10$

Exhaustive Search and Generation II
The 0-1 knapsack problem

- $98 = 90 + (16 - 12)\,10/5$
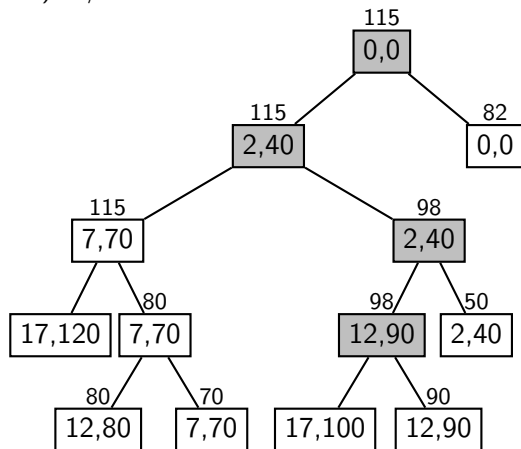
Exhaustive Search and Generation II
The 0-1 knapsack problem

- $50 = 40 + 10$

$w_1 = 2 \quad p_1 = 40$

$w_2 = 5 \quad p_2 = 30$

$w_3 = 10 \quad p_3 = 50$

$w_4 = 5 \quad p_4 = 10$
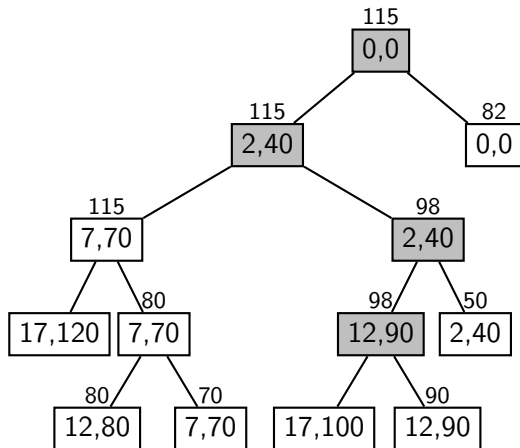
Exhaustive Search and Generation II
The 0-1 knapsack problem

```
PROMISING(i, w, p)
    if w ⩾ W then
        return FALSE
    else
        j = i + 1
        bound = p
        t = w
        while j ⩽ n and t + w[j] ⩽ W do
            t = t + w[j]
            bound = bound + p[j]
            j = j + 1
        if j ⩽ n then
            bound = bound + (W − t) p[j]/w[j]
        return bound > max_p
```

Exhaustive Search and Generation II
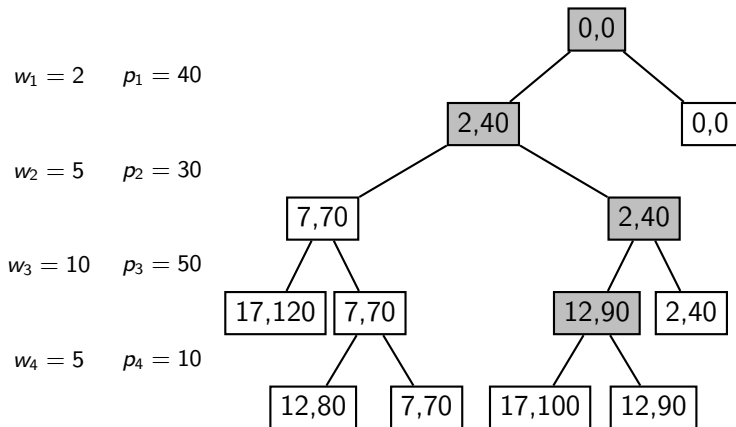# The hamiltonian circuits problem

Backtracking is a technique is used to solve problems in which a **sequence** of objects is chosen from a specified **set** so that the sequence satisfies some **criterion**.

- The goal in the hamiltonian circuits problem is to find a hamiltonian circuit (a path that starts at a given vertex, visits each vertex in the graph exactly once, and ends at the starting vertex) in a connected, undirected graph.

- The sequence in this problem is the vertices of the graph.

- The set for each choice is the vertices of the graph.

- The criterion is that the chosen vertices form a simple path or cycle in the graph.

Exhaustive Search and Generation II
The hamiltonian circuits problem

- A portion of the pruned state space tree for $V = \{v_1, \ldots, v_6\}$ and $E = \{(v_1, v_2), (v_1, v_4), (v_1, v_5), (v_2, v_3), (v_2, v_4), (v_2, v_5), (v_2, v_6), (v_3, v_5), (v_3, v_6), (v_4, v_5), (v_5, v_6)\}$.

Exhaustive Search and Generation II
The hamiltonian circuits problem

- The undirected graph is represented by an adjacency matrix $G.Adj[1..n, 1..n]$.
- A hamiltonian circuit is represented by a vector $I[1..n]$ of vertex indices.

```
HAMILTONIAN-CIRCUITS(0)
```

```
HAMILTONIAN-CIRCUITS(i)
   if PROMISING(i) then
      if i = n then
         write I[1], . . . , I[n]
      else
         for j = 1 to n do
            I[i + 1] = j
            HAMILTONIAN-CIRCUITS(i + 1)
```

Exhaustive Search and Generation II
The hamiltonian circuits problem

```
PROMISING(i)
    if i = n and not G.Adj[l[n − 1], l[0]] then
        return FALSE
    else if i > 1 and not G.Adj[l[i − 1], l[i]] then
        return FALSE
    else
        for j = 1 to i − 1 do
            if l[i] = l[j] then
                return FALSE
    return TRUE
```

Exhaustive Search and Generation II
The traveling salesman problem

> Backtracking is a technique is used to solve problems in which a **sequence** of objects is chosen from a specified **set** so that the sequence satisfies some **criterion**.

- The goal in the traveling salesman problem is to find a hamiltonian circuit (a path that starts at a given vertex, visits each vertex in the graph exactly once, and ends at the starting vertex) of smallest weight in a connected, directed graph.

- The sequence in this problem is the vertices of the graph.

- The set for each choice is the vertices of the graph.

- The criterion is that the chosen vertices form a simple path or cycle in the graph.

Exhaustive Search and Generation II
The traveling salesman problem

Example (Adjacency matrix representation of a graph that has an edge from every vertex to every other vertex, and an optimal tour for the graph.)

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$
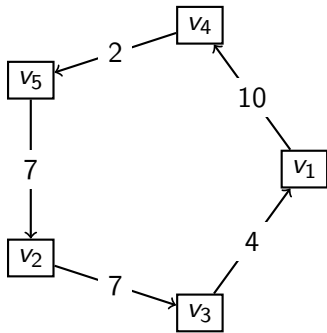
Exhaustive Search and Generation II
The traveling salesman problem
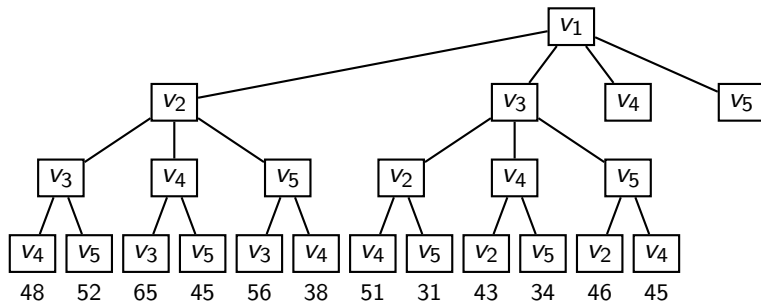
- A portion of the pruned state space tree for the previous graph.

Exhaustive Search and Generation II
The traveling salesman problem

- A portion of the pruned state space tree for the previous graph.
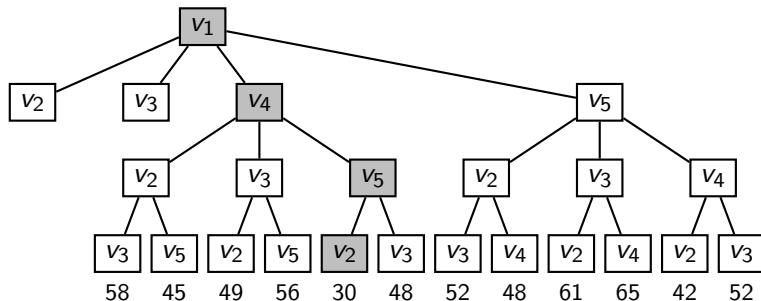
- In optimization problems, we need to be able to determine a bound for each node.

- Because of the objective in the 0-1 knapsack problem (to maximize profit while keeping the total weight from exceeding W), we computed an upper bound on the amount of profit that could be obtained by expanding beyond a given node, and we called a node promising if its bound was greater than the current maximum profit.

- In the traveling salesman problem, we need to determine a lower bound on the length of any tour that can be obtained by expanding beyond a given node, and we call the node promising if its bound is less than the current minimum tour length.

Exhaustive Search and Generation II
The traveling salesman problem

- The undirected graph is represented by a weighted adjacency matrix $G.Adj[1..n, 1..n]$.
- A tour is represented by a vector $I[1..n]$ of vertex indices.

TRAVELING-SALESMAN$(0, 0)$

TRAVELING-SALESMAN(i, w)
    if $i = n$ and $w < min\_w$ **then**
        $w = min\_w$
        $best\_I = I$
    if PROMISING(i, w) **then**
        for $j = 1$ **to** n **do**
            $I[i + 1] = j$
            TRAVELING-SALESMAN$(i + 1, w + G.Adj[I[i], j])$

Exhaustive Search and Generation II
The traveling salesman problem

$min\_w = \infty$
$best\_I = 0, 0, \ldots, 0$
TRAVELING-SALESMAN(0, 0)
write $best\_I[1], \ldots, best\_I[n]$

- A node is non-promising if when visiting the node we determine that it cannot possibly lead to a solution.
- In optimization problems, a node is non-promising if when visiting the node we determine that no expansion to the children can possibly lead to a solution.

PROMISING(i, w)
   return $w + bound < min\_w$

- A bound can be obtained by adding the length of the shortest edge going out of each of the remaining vertices.

Exhaustive Search and Generation II
The traveling salesman problem

```
PROMISING(i, w)
    if i = n and G.Adj[I[n − 1], I[0]] = 0 then
        return FALSE
    else if i > 1 and G.Adj[I[i − 1], I[i]] = 0 then
        return FALSE
    else
        for j = 1 to i − 1 do
            if I[i] = I[j] then
                return FALSE
    bound = 0
    for j in {1, . . . , n} − {I[1], . . . , I[i]} do
        bound = bound + min_{k=1}^{n}{G.Adj[j, k] : G.Adj[j, k] ≠ 0}
    return w + bound < min_w
```

Lecture 11
Notions of Intractability I

Notions of Intractability I

Contents

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009

Notions of Intractability I
Overview

- There are problems that can be solved by a polynomial-time algorithm in time $O(n^k)$ for some constant k, where n is the size of the input to the problem.
- There are other problems that can be solved, but not in time $O(n^k)$ for any constant k.
- There are also problems that cannot be solved by any computer, no matter how much time we allow.
- We think of problems that are solvable by polynomial-time algorithms as being tractable, or easy, and problems that require more than polynomial time as being intractable, or hard.
- Several hard problems seem on the surface to be similar to easy problems: shortest versus longest paths, eulerian versus hamiltonian cycle, and 2-CNF versus 3-CNF-satisfiability.

Notions of Intractability I
Overview

- The class P consists of those problems that are solvable in polynomial time.
- The class NP consists of those problems that are verifiable in polynomial time (given a certificate of a solution, we can verify that the certificate is correct in time polynomial in the size of the input to the problem).
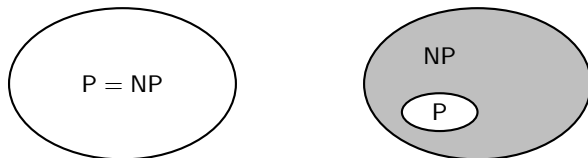
- The class NPC (NP-complete) consists of those problems that are in NP and are as hard as any problem in NP (if any problem in NPC can be solved in polynomial time, then every problem in NP can be solved in polynomial time).

Notions of Intractability I
Overview

- In an optimization problem, each feasible solution has an associated value, and we wish to find a feasible solution with the best value.
- In a decision problem, the answer is simply "yes" or "no".
- We can cast an optimization problem as a related decision problem by imposing a bound on the value to be optimized.

Example

- The SHORTEST-PATH optimization problem is, given an undirected graph G and vertices u and v, find a path from u to v that uses the fewest edges.
- The PATH decision problem is, given a directed graph G, vertices u and v, and an integer k, does a path exist from u to v consisting of at most k edges?

Notions of Intractability I
Overview

- The decision problem is no harder than the related optimization problem.
- If an optimization problem is easy, the related decision problem is easy as well.
- If we can provide evidence that a decision problem is hard, we also provide evidence that the related optimization problem is hard.

Example

- We can solve PATH by solving SHORTEST-PATH and then comparing the number of edges in the shortest path found to k.

- The notion of showing that one problem is no harder or no easier than another applies even when both problems are decision problems.

- Suppose we want to solve a decision problem A in polynomial time, we already know how to solve a different decision problem B in polynomial time, and we have a polynomial-time algorithm that transforms any instance of A into some instance of B such that the answers are the same.
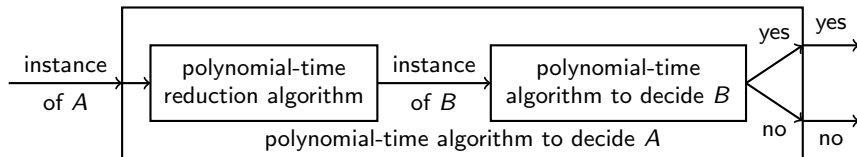
Notions of Intractability I
Overview

- The reduction algorithm provides us a way to solve problem A in polynomial time.
  - Given an instance α of problem A, use a polynomial-time reduction algorithm to transform it to an instance β of problem B.
  - Run the polynomial-time decision algorithm for B on the instance β.
  - Use the answer for β as the answer for α.
- As long as each of these steps takes polynomial time, all three together do also, and so we have a way to decide on α in polynomial time.
- By "reducing" solving problem A to solving problem B, we use the "easiness" of B to prove the "easiness" of A.

Notions of Intractability I
Overview

- Now, suppose we have a decision problem A for which we already know that no polynomial-time algorithm can exist, and we have a polynomial-time algorithm that transforms any instance of A into some instance of B such that the answers are the same.

- We can use a simple proof by contradiction to show that no polynomial-time algorithm can exist for B. In fact, if B had a polynomial-time algorithm, then we would have a way to solve problem A in polynomial time.

- We cannot assume that there is absolutely no polynomial-time algorithm for problem A, but we can prove that problem B is hard on the assumption that problem A is also hard.

- We need a "first" hard problem. The problem we shall use is the circuit-satisfiability problem.

Notions of Intractability I
34.1 Polynomial time

- We generally regard polynomial-time solvable problems as tractable, but for philosophical, not mathematical, reasons.

- Although we may reasonably regard a problem that requires time $\Theta(n^{100})$ to be intractable, very few practical problems require time on the order of such a high-degree polynomial.

- For many reasonable models of computation, a problem that can be solved in polynomial time in one model can be solved in polynomial time in another.

- The class of polynomial-time solvable problems has nice closure properties, since polynomials are closed under addition, multiplication, and composition.

Abstract problems

- We define an abstract problem Q to be a binary relation on a set I of problem instances and a set S of problem solutions.

- For example, an instance for SHORTEST-PATH is a triple consisting of a graph and two vertices. A solution is a sequence of vertices in the graph, with perhaps the empty sequence denoting that no path exists.

- We can view an abstract decision problem as a function that maps the instance set I to the solution set $\{0, 1\}$.

- For example, a decision problem related to SHORTEST-PATH is the problem PATH. If $i = \langle G, u, v, k \rangle$ is an instance of the decision problem PATH, then $\text{PATH}(i) = 1$ if a shortest path from u to v has at most k edges, and $\text{PATH}(i) = 0$ otherwise.

Encodings

- An encoding of a set S of abstract objects is a mapping e from S to the set of binary strings (or to a set of strings over a finite alphabet having at least 2 symbols).

- A concrete problem is a problem whose instance set is the set of binary strings.

- An algorithm solves a concrete problem in time $O(T(n))$ if, when it is provided a problem instance i of length $n = |i|$, the algorithm can produce the solution in $O(T(n))$ time.

- A concrete problem is polynomial-time solvable if there exists an algorithm to solve it in time $O(n^k)$ for some constant k.

- The complexity class P is the set of concrete decision problems that are polynomial-time solvable.

Encodings

- We can use encodings to map abstract problems to concrete problems.

- Given an abstract decision problem Q mapping an instance set I to $\{0, 1\}$, an encoding $e : I \rightarrow \{0, 1\}^{\star}$ can induce a related concrete decision problem $e(Q)$.

- If the solution to an abstract-problem instance $i \in I$ is $Q(i) \in \{0, 1\}$, then the solution to the concrete-problem instance $e(I) \in \{0, 1\}^{\star}$ is also $Q(i)$.

- Some binary strings might represent no meaningful abstract-problem instance. We assume they map to 0.

- The concrete problem produces the same solutions as the abstract problem on binary-string instances that represent the encodings of abstract-problem instances.

Encodings

- We want to extend the definition of polynomial-time solvability from concrete problems to abstract problems by using encodings as the bridge, but the efficiency of solving a problem depends quite heavily on the encoding.

- Suppose that an integer k is to be provided as the sole input to an algorithm, and suppose that the running time of the algorithm is $\Theta(k)$.

- If k is provided in unary representation, then the input length is $n = k$, and the running time of the algorithm is $\Theta(k) = \Theta(n)$, which is polynomial in the size of the input.

- If k is provided in binary representation, then the input length is $n = \lfloor \lg k \rfloor + 1$, and the running time of the algorithm is $\Theta(k) = \Theta(2^n)$, which is exponential in the size of the input.

Encodings

- If we rule out "expensive" encodings such as unary ones, the actual encoding of a problem makes little difference to whether the problem can be solved in polynomial time.

- A function $f : \{0,1\}^\star \to \{0,1\}^\star$ is polynomial-time computable if there exists a polynomial-time algorithm A that, given any input $x \in \{0,1\}^\star$, produces an output $f(x)$.

- For some set I of problem instances, two encodings e_1 and e_2 are polynomially related if there exist two polynomial-time computable functions f_{12} and f_{21} such that for any $i \in I$, we have $f_{12}(e_1(i)) = e_2(i)$ and $f_{21}(e_2(i)) = e_1(i)$.

- If two encodings e_1 and e_2 of an abstract problem are polynomially related, whether the problem is polynomial-time solvable or not is independent of which encoding we use.

Lemma (CLRS Lemma 34.1)

Let Q be an abstract decision problem on an instance set I, and let e_1 and e_2 be polynomially related encodings on I. Then, $e_1(Q) \in P$ if and only if $e_2(Q) \in P$.

Proof.

- Suppose $e_1(Q)$ can be solved in time $O(n^k)$ for some constant k and, for any problem instance i, the encoding $e_1(i)$ can be computed from the encoding $e_2(i)$ in time $O(n^c)$ for some constant c, where $n = |e_2(i)|$.

- To solve problem $e_2(Q)$, on input $e_2(i)$, we first compute $e_1(i)$ and then run the algorithm for $e_1(Q)$ on $e_1(i)$.

- Converting encodings takes time $O(n^c)$, thus $|e_1(i)| = O(n^c)$.

- Solving the problem on $e_1(i)$ takes time $O(|e_1(i)|^k) = O(n^{ck})$, which is polynomial since both c and k are constants.

Encodings

- We assume that the encoding of an integer is polynomially related to its binary representation, and that the encoding of a finite set is polynomially related to its encoding as a list of its elements, enclosed in braces and separated by commas.

- With such a standard encoding in hand, we can derive reasonable encodings of other mathematical objects, such as tuples, graphs, and formulas.

- To denote the standard encoding of an object, we shall enclose the object in angle braces. Thus, $\langle G \rangle$ denotes the standard encoding of a graph G.

- Since we implicitly use an encoding that is polynomially related to this standard encoding, the choice of encoding has no effect on whether a problem is polynomial-time solvable.

A formal-language framework

- The set of instances for any decision problem Q is the set $\Sigma^\star$, where $\Sigma = \{0, 1\}$.

- Since Q is entirely characterized by those problem instances that produce a 1 (yes) answer, we can view Q as a language L over $\Sigma = \{0, 1\}$, where $L = \{x \in \Sigma^\star : Q(x) = 1\}$.

- For example, the decision problem PATH has the corresponding language

$$\text{PATH} = \{\langle G, u, v, k \rangle : G = (V, E) \text{ is an undirected graph,}$$
$$u, v \in V,$$
$$k \geqslant 0 \text{ is an integer, and}$$
$$\text{there exists a path from } u \text{ to } v \text{ in } G$$
$$\text{consisting of at most } k \text{ edges}\}$$

A formal-language framework

- An algorithm A accepts a string $x \in \{0, 1\}^\star$ if, given input x, the output $A(x)$ of the algorithm is 1.

- The language accepted by an algorithm A is the set of strings $L = \{x \in \{0, 1\}^\star : A(x) = 1\}$.

- An algorithm A rejects a string x if $A(x) = 0$.

- Even if language L is accepted by an algorithm A, the algorithm will not necessarily reject a string $x \notin L$ provided as input to it. For example, the algorithm may loop forever.

- A language L is decided by an algorithm A if every binary string in L is accepted by A and every binary string not in L is rejected by A.

A formal-language framework

- A language L is accepted in polynomial time by an algorithm A if it is accepted by A and if there exists a constant k such that for any string $x \in L$ of length n, algorithm A accepts x in time $O(n^k)$.

- A language L is decided in polynomial time by an algorithm A if there exists a constant k such that for any string $x \in \{0,1\}^\star$ of length n, algorithm A correctly decides whether $x \in L$ in time $O(n^k)$.

- To accept a language, an algorithm need only produce an answer when provided a string in L, but to decide a language, it must correctly accept or reject every string $x \in \{0,1\}^\star$.

A formal-language framework

- The language PATH can be accepted in polynomial time. One algorithm verifies that G encodes an undirected graph, verifies that u and v are vertices in G, uses breadth-first search to compute a shortest path from u to v in G, and then compares the number of edges on the shortest path obtained with k.

- If G encodes an undirected graph and the path found from u to v has at most k edges, the algorithm outputs 1 and halts. Otherwise, the algorithm runs forever.

- This algorithm does not decide PATH, since it does not output 0 for instances in which a shortest path has more than k edges. A decision algorithm for PATH must reject binary strings that do not belong to PATH.

A formal-language framework

- P is the class of languages that can be decided in polynomial time.

$$P = \{L \subseteq \{0,1\}^\star : \text{there exists an algorithm } A$$
$$\text{that decides } L \text{ in polynomial time}\}$$

- P is also the class of languages that can be accepted in polynomial time.

Theorem (CLRS Theorem 34.2)

$P = \{L : L \text{ is accepted by a polynomial-time algorithm}\}$.

- Suppose that for a given instance $\langle G, u, v, k \rangle$ of the decision problem PATH, we are also given a path p from u to v.

- We can easily check whether p is a path in G and whether the length of p is at most k, and if so, we can view p as a "certificate" that the instance indeed belongs to PATH.

- For the decision problem PATH, this certificate does not seem to buy us much.

- After all, PATH belongs to P and so verifying membership from a given certificate takes as long as solving the problem from scratch.

- However, there are problems for which we know of no polynomial-time decision algorithm and yet, given a certificate, verification is easy.

Hamiltonian cycles

- A hamiltonian cycle of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex in V.

- The size of a (simple) cycle is the number of edges in it.

- The longest cycle problem is the optimization problem of finding a simple cycle of maximum size in a graph.

- As a decision problem, we ask whether a graph has a simple cycle of a given size k.

- In the hamiltonian cycle problem, we ask whether a graph has a hamiltonian cycle. As a formal language,

$$\text{HAM-CYCLE} = \{\langle G \rangle : G \text{ is a hamiltonian graph}\}$$

Notions of Intractability I
34.2 Polynomial-time verification

Hamiltonian cycles

- Given a problem instance $\langle G \rangle$, one possible decision algorithm lists all permutations of the vertices of G and then checks each permutation to see if it is a hamiltonian path. What is the running time of this algorithm?

- If we use the "reasonable" encoding of a graph as its adjacency matrix, the number m of vertices in the graph is $\Omega(\sqrt{n})$, where $n = |\langle G \rangle|$ is the length of the encoding of G.

- There are $m!$ possible permutations of the vertices, and therefore the running time is $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$, which is not $O(n^k)$ for any constant k.

- Thus, this naive algorithm does not run in polynomial time.

- In fact, the hamiltonian-cycle problem is NP-complete.

Verification algorithms

- Suppose that a friend tells you that a given graph G is hamiltonian, and then offers to prove it by giving you the vertices in order along the hamiltonian cycle.

- It would certainly be easy enough to verify the proof: simply verify that the provided cycle is hamiltonian by checking whether it is a permutation of the vertices of V and whether each of the consecutive edges along the cycle actually exists in the graph.

- You could certainly implement this verification algorithm to run in $O(n^2)$ time, where n is the length of the encoding of G.

- Thus, a proof that a hamiltonian cycle exists in a graph can be verified in polynomial time.

Verification algorithms

- A verification algorithm is a two-argument algorithm A, where one argument is an ordinary input string x and the other is a binary string y called a certificate.

- A two-argument algorithm A verifies an input string x if there exists a certificate y such that $A(x, y) = 1$.

- The language verified by a verification algorithm A is

  $$L = \{x \in \{0, 1\}^\star : \text{there exists } y \in \{0, 1\}^\star \text{ such that } A(x, y) = 1\}$$

- Intuitively, an algorithm A verifies a language L if for any string $x \in L$, there exists a certificate y that A can use to prove that $x \in L$. Moreover, for any string $x \notin L$, there must be no certificate proving that $x \in L$.

Verification algorithms

- In the hamiltonian-cycle problem, the certificate is the list of vertices in some hamiltonian cycle.

- If a graph is hamiltonian, the hamiltonian cycle itself offers enough information to verify this fact.

- Conversely, if a graph is not hamiltonian, there can be no list of vertices that fools the verification algorithm into believing that the graph is hamiltonian, since the verification algorithm carefully checks the proposed "cycle" to be sure.

The complexity class NP

- The complexity class NP is the class of languages that can be verified by a polynomial-time algorithm.

- A language L belongs to NP if and only if there exist a two-input polynomial-time algorithm A and a constant c such that

$$L = \{x \in \{0,1\}^\star : \text{there exists a certificate } y \in \{0,1\}^\star$$
$$\text{with } |y| = O(|x|^c)$$
$$\text{such that } A(x,y) = 1\}$$

- Such an algorithm A verifies language L in polynomial time.

- For example, HAM-CYCLE $\in$ NP.

The complexity class NP

- If $L \in$ P, then $L \in$ NP, since if there is a polynomial-time algorithm to decide L, the algorithm can be easily converted to a two-argument verification algorithm that simply ignores any certificate and accepts exactly those input strings it determines to be in L. Thus, P $\subseteq$ NP.

- It is unknown whether P $=$ NP, but most rearchers believe that P and NP are not the same class.
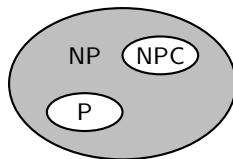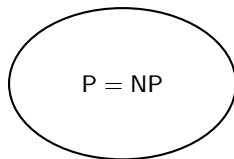
- Perhaps the most compelling reason why theoretical computer scientists believe that $P \neq NP$ comes from the existence of the class of "NP-complete" problems.

- This class has the intriguing property that if any NP-complete problem can be solved in polynomial time, then every problem in NP has a polynomial-time solution, that is, $P = NP$.

- Despite years of study, though, no polynomial-time algorithm has ever been discovered for any NP-complete problem.

- The language HAM-CYCLE is one NP-complete problem. If we could decide HAM-CYCLE in polynomial time, then we could solve every problem in NP in polynomial time.

- The NP-complete languages are, in a sense, the "hardest" languages in NP.

Reducibility

- A problem Q can be reduced to another problem Q' if any instance of Q can be "easily rephrased" as an instance of Q', the solution to which provides a solution to the instance of Q.

- For example, the problem of solving linear equations reduces to the problem of solving quadratic equations.

- Thus, if a problem Q reduces to another problem Q', then Q is "no harder to solve" than Q'.

- A language L_1 is polynomial-time reducible to a language L_2, written $L_1 \leqslant_P L_2$, if there exists a polynomial-time computable function $f : \{0,1\}^\star \to \{0,1\}^\star$ such that for all $x \in \{0,1\}^\star$, $x \in L_1$ if and only if $f(x) \in L_2$.

- f is the reduction function, and a polynomial-time algorithm F that computes f is a reduction algorithm.

Reducibility

- The reduction function f provides a polynomial-time mapping such that if $x \in L_1$, then $f(x) \in L_2$. Moreover, if $x \notin L_1$, then $f(x) \notin L_2$.
- Thus, the reduction function maps any instance x of the decision problem represented by the language L_1 to an instance $f(x)$ of the problem represented by L_2.
- Providing an answer to whether $f(x) \in L_2$ directly provides the answer to whether $x \in L_1$.
- Polynomial-time reductions give us a powerful tool for proving that various languages belong to P.

Lemma (CLRS Lemma 34.3)

If $L_1, L_2 \subseteq \{0,1\}^\star$ are languages such that $L_1 \leqslant_P L_2$, then $L_2 \in P$ implies $L_1 \in P$.

Proof.
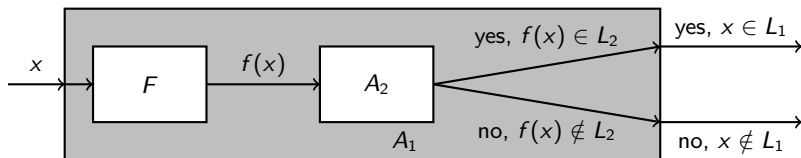
NP-completeness

- Polynomial-time reductions provide a formal means for showing that one problem is at least as hard as another, to within a polynomial-time factor.

- That is, if $L_1 \leqslant_P L_2$, then L_1 is not more than a polynomial factor harder than L_2.

- NP-complete languages are the hardest problems in NP.

- A language L is NP-complete if $L \in$ NP and $L' \leqslant_P L$ for every $L' \in$ NP.

- A language L is NP-hard if $L' \leqslant_P L$ for every $L' \in$ NP.

- NPC is the class of NP-complete languages.

Theorem (CLRS Theorem 34.4)

If any NP-complete problem is polynomial-time solvable, then
$P = NP$. *Equivalently, if any problem in NP is not polynomial-time solvable, then no NP-complete problem is polynomial-time solvable.*

Proof.

- Suppose that $L \in P$ and also that $L \in NPC$.
- For any $L' \in NP$, we have $L' \leqslant_P L$ by the definition of NP-completeness and, by the previous lemma, we have $L' \in P$.
- The second statement is the contrapositive of the first statement.

NP-completeness

- Most theoretical computer scientists believe that $P \neq NP$.

- Someone may yet come up with a polynomial-time algorithm for an NP-complete problem, thus proving that $P = NP$.

- Since no polynomial-time algorithm for any NP-complete problem has yet been discovered, a proof that a problem is NP-complete provides excellent evidence that it is intractable.

Circuit satisfiability

- A boolean combinatorial circuit consists of one or more boolean combinatorial elements interconnected by wires.
- A truth assignment for a boolean combinatorial circuit is a set of boolean input values, and a satisfying assignment is a truth assignment that causes the output of the circuit to be 1.
- A one-output boolean combinational circuit with a satisfying assignment is a satisfiable boolean combinatorial circuit.
- As a decision problem, in the circuit satisfiability problem we ask whether a boolean combinatorial circuit composed of AND, OR, and NOT gates is satisfiable. As a formal language,

  CIRCUIT-SAT = {⟨C⟩ : C is a satisfiable boolean circuit}

Notions of Intractability I
34.3 NP-completeness and reducibility

Circuit satisfiability

- A circuit C with k inputs has up to 2^k possible assignments.
- When the size of C is polynomial in k, checking all possible assignments to the inputs of C takes $\Omega(2^k)$ time, which is more than polynomial in the size of C.

Lemma (CLRS Lemma 34.5)

The circuit-satisfiability problem belongs to the class NP.

Lemma (CLRS Lemma 34.6)

The circuit-satisfiability problem is NP-hard.

Theorem (CLRS Theorem 34.7)

The circuit-satisfiability problem is NP-complete.

Lecture 12
Notions of Intractability II

Notions of Intractability II

Contents

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009

Notions of Intractability II
34.4 NP-completeness proofs

- It can be shown that the circuit-satisfiability problem is NP-complete by a direct proof that $L \leqslant_P$ CIRCUIT-SAT for every language $L \in$ NP.
- We can prove that a language is NP-complete without directly reducing every language in NP to the given language.

Lemma (CLRS Lemma 34.8)

If L is a language such that $L' \leqslant_P L$ for some $L' \in$ NPC, then L is NP-hard. If, in addition, $L \in$ NP, then $L \in$ NPC.

Proof.

- Since L' is NP-complete, for all $L'' \in$ NP, we have $L'' \leqslant_P L'$.
- By assumption, $L' \leqslant_P L$, and thus by transitivity, we have $L'' \leqslant_P L$, which shows that L is NP-hard.
- If $L \in$ NP, we also have $L \in$ NPC.

Notions of Intractability II
34.4 NP-completeness proofs

- By reducing a known NP-complete language L' to L, we implicitly reduce every language in NP to L. This gives a method for proving that a language L is NP-complete.

  – Prove $L \in$ NP.

  – Select a known NP-complete language L'.

  – Describe an algorithm that computes a function f mapping every instance $x \in \{0,1\}^\star$ of L' to an instance $f(x)$ of L.

  – Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0,1\}^\star$.

  – Prove that the algorithm computing f runs in polynomial time.

- As we develop a catalog of known NP-complete problems, we will have more choices for languages from which to reduce.

Formula satisfiability

- A boolean formula consists of n boolean variables, m boolean connectives (any boolean function with one or two inputs, and one output), and (non-redundant) parentheses.

- We can encode a boolean formula ϕ in a length that is polynomial in $n + m$.

- A truth assignment for a boolean formula ϕ is a set of values for the variables in ϕ, and a satisfying assignment is a truth assignment that causes it to evaluate to 1.

- A formula with a satisfying assignment is a satisfiable formula.

- As a decision problem, in the formula satisfiability problem we ask whether a given boolean formula is satisfiable. As a formal language, SAT $= \{\langle\phi\rangle : \phi$ is a satisfiable boolean formula$\}$

Formula satisfiability

- A naive algorithm for determining whether a boolean formula ϕ with n variables is satisfiable is to list all assignments, and check each one to see whether it is a satisfying assignment.

- If the length of $\langle \phi \rangle$ is polynomial in n, the running time of this algorithm is $\Omega(2^n)$, which is not polynomial in the length of $\langle \phi \rangle$.

- A polynomial-time algorithm for the formula satisfiability problem is unlikely to exist.

Theorem (CLRS Theorem 34.9)

Satisfiability of boolean formulas is NP-complete.

Proof.

- To show that SAT belongs to NP, we show that a certificate consisting of a satisfying assignment for an input formula ϕ can be verified in polynomial time.

- The verifying algorithm simply replaces each variable in the formula with its corresponding value and then evaluates the expression. This task is easy to do in polynomial time.

- If the expression evaluates to 1, then the algorithm has verified that the formula is satisfiable.

- We next prove that CIRCUIT-SAT $\leqslant_P$ SAT, which shows that the formula satisfiability problem is NP-hard.

Proof (Cont'd).

- We need to show how to reduce any instance of circuit satisfiability to an instance of formula satisfiability in polynomial time.

- The formula ϕ has a variable x_i for each wire x_i in the circuit.

- We can express how each gate operates as a clause involving the variables of its incident wires.

- The formula ϕ produced by the reduction algorithm is the AND of the circuit-output variable with the conjunction of clauses describing the operation of each gate.

- Given a circuit C, it is straightforward to produce such a formula ϕ in polynomial time.

Example
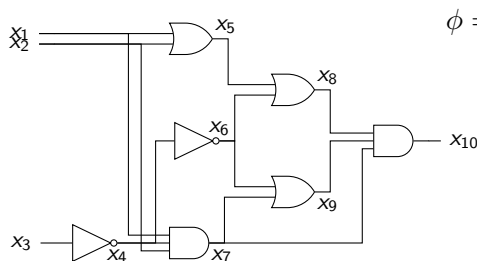
$$\phi = x_{10} \wedge (x_4 \leftrightarrow \neg x_3)$$
$$\wedge (x_5 \leftrightarrow (x_1 \vee x_2))$$
$$\wedge (x_6 \leftrightarrow \neg x_4)$$
$$\wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4))$$
$$\wedge (x_8 \leftrightarrow (x_5 \vee x_6))$$
$$\wedge (x_9 \leftrightarrow (x_6 \vee x_7))$$
$$\wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))$$

Notions of Intractability II
34.4 NP-completeness proofs

Formula satisfiability

- We still have to show that the circuit C is satisfiable exactly when the formula ϕ is satisfiable.
- If C has a satisfying assignment, then each wire of the circuit has a well-defined value, and the output of the circuit is 1.
- Therefore, when we assign wire values to variables in ϕ, each clause of ϕ evaluates to 1, and thus the conjunction of all clauses evaluates to 1.
- Conversely, if some assignment causes ϕ to evaluate to 1, the circuit C is satisfiable by an analogous argument.

3-CNF satisfiability

- A literal in a boolean formula is an occurrence of a variable or its negation.

- A boolean formula is in conjunctive normal form, or CNF, if it is expressed as an AND of clauses, each of which is the OR of one or more literals.

- A boolean formula is in 3-conjunctive normal form, or 3-CNF, if each clause has exactly three distinct literals.

- As a decision problem, in the 3-CNF satisfiability problem we ask whether a given boolean formula in 3-CNF is satisfiable. As a formal language,

  3-CNF-SAT $= \{\langle \phi \rangle : \phi$ is a satisfiable boolean formula in 3-CNF$\}$

- A polynomial-time algorithm that can determine the satisfiability of boolean formulas is unlikely to exist, even when they are expressed in this simple normal form.

Theorem (CLRS Theorem 34.10)

Satisfiability of boolean formulas in 3-conjunctive normal form is NP-complete.

Proof.

- We can show that 3-CNF-SAT belongs to NP with the same argument we used in the proof that SAT is NP-complete.
- We next prove that SAT $\leqslant_P$ 3-CNF-SAT, which shows that the 3-CNF satisfiability problem is NP-hard.

Proof.

- First, we construct a binary tree for the input formula ϕ, with literals as leaves and connectives as internal nodes.

- We can think of the binary tree as a circuit for computing the function.

- Then, we introduce a variable y_i for the output of each internal node, and we rewrite the original formula ϕ as the AND of the root variable and a conjunction of clauses describing the operation of each node.

- The formula ϕ' thus obtained is a conjunction of clauses ϕ'_i, each of which has at most 3 literals. The only requirement that we might fail to meet is that each clause has to be an OR of 3 literals.

Proof (Cont'd).

- Second, we convert each clause ϕ_i' of the formula ϕ' into conjunctive normal form.

- We construct a truth table for ϕ_i' by evaluating all possible assignments to its variables.

- Using the truth-table entries that evaluate to 0, we build a formula in disjunctive normal form, or DNF, that is equivalent to $\neg\phi_i'$.

- We then negate this formula and convert it into a CNF formula ϕ_i'' by using DeMorgan's laws for propositional logic.

- Now, ϕ' is equivalent to the CNF formula ϕ_i'' consisting of the conjunction of the ϕ_i''. Moreover, each clause of ϕ_i'' has at most 3 literals.

Proof (Cont'd).

- Third, we convert each clause ϕ_i'' of the formula ϕ'' into one or more clauses in conjunctive normal form, using two auxiliary variables p and q.
- If ϕ_i'' has 3 distinct literals, then include ϕ_i'' as a clause of ϕ'''.
- If ϕ_i'' has 2 distinct literals ℓ_1 and ℓ_2, then include

$$(\ell_1 \vee \ell_2 \vee p) \wedge (\ell_1 \vee \ell_2 \vee \neg p)$$

as clauses of ϕ'''.

- Regardless of the value of p, one of the clauses is equivalent to $\ell_1 \vee \ell_2$, and the other evaluates to 1, which is the identity for AND.

Proof (Cont'd).

- If ϕ_i'' has 1 distinct literal ℓ, then include

$$(\ell \vee p \vee q) \wedge (\ell \vee p \vee \neg q) \wedge (\ell \vee \neg p \vee q) \wedge (\ell \vee \neg p \vee \neg q)$$

   as clauses of ϕ'''.

- Regardless of the values of p and q, one of the four clauses is equivalent to ℓ, and the other 3 evaluate to 1, which is the identity for AND.

Proof (Cont'd).

- We can see that the 3-CNF formula ϕ''' is satisfiable if and only if ϕ is satisfiable by inspecting each of the three steps.

- Like the reduction from CIRCUIT-SAT to SAT, the construction of ϕ' from ϕ in the first step preserves satisfiability.

- The second step produces a CNF formula ϕ'' that is algebraically equivalent to ϕ'.

- The third step produces a 3-CNF formula ϕ''' that is effectively equivalent to ϕ'', since any assignment to the variables p and q produces a formula that is algebraically equivalent to ϕ''.

Proof (Cont'd).

- Each of the constructions can be done in polynomial time and thus, the reduction can be computed in polynomial time.
- Constructing ϕ' from ϕ introduces at most 1 variable and 1 clause into ϕ' for each connective in ϕ.
- Constructing ϕ'' from ϕ' introduces at most 8 clauses into ϕ'' for each clause of ϕ', since each clause of ϕ' has at most 3 variables, and the truth table for each clause has at most $2^3 = 8$ rows.
- Constructing ϕ''' from ϕ'' introduces at most 4 clauses into ϕ''' for each clause of ϕ''.
- Thus, the size of the resulting formula ϕ''' is polynomial in the length of the original formula.

CIRCUIT-SAT
↓
SAT
↓
3-CNF-SAT

CLIQUE   SUBSET-SUM

VERTEX-COVER
↓
HAM-CYCLE
↓
TSP
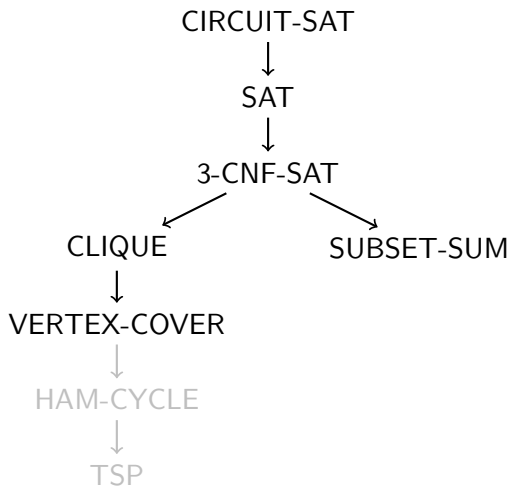
The clique problem

- A clique in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E.

- The size of a clique is the number of vertices in it.

- The clique problem is the optimization problem of finding a clique of maximum size in a graph.

- As a decision problem, we ask whether a graph has a clique of a given size k.

The clique problem

- A naive algorithm for determining whether a graph $G = (V, E)$ with $|V|$ vertices has a clique of size k is to list all k-subsets of V, and check each one to see whether it forms a clique.

- The running time of this algorithm is $\Omega(k^2 \binom{|V|}{k})$, which is polynomial if k is a constant.

- In general, however, k could be near $|V|/2$, in which case the algorithm does not run in polynomial time. For $1 \leqslant k \leqslant n$,

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1} = \frac{n}{k}\frac{n-1}{k-1}\cdots\frac{n-k+1}{1} \geqslant \left(\frac{n}{k}\right)^k$$

and thus, for $k = n/2 = |V|/2$, we have $2^{|V|/2} \leqslant \binom{|V|}{|V|/2}$.

Theorem (CLRS 34.11)

The clique problem is NP-complete.

Proof.

- To show that CLIQUE belongs to NP, for a given graph $G = (V, E)$, we use the set $V' \subseteq V$ of vertices in the clique as a certificate for G.

- We can check whether V' is a clique in polynomial time by checking whether, for each pair $u, v \in V'$, the edge (u, v) belongs to E.

- We next prove that 3-CNF-SAT $\leqslant_P$ CLIQUE, which shows that the clique problem is NP-hard.

- The reduction algorithm begins with an instance of 3-CNF-SAT.

Proof (Cont'd).

- Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ be a boolean formula in 3-CNF with k clauses.

- For $r = 1, 2, \ldots, k$, each clause C_r has exactly three distinct literals ℓ_1^r, ℓ_2^r, and ℓ_3^r.

- We shall construct a graph G such that ϕ is satisfactible if and only if G has a clique of size k.

- We construct the graph $G = (V, E)$ as follows.

- For each clause $C_r = (\ell_1^r \vee \ell_2^r \vee \ell_3^r)$ in ϕ, we place a triple of vertices v_1^r, v_2^r, and v_3^r into V.

Proof (Cont'd).

- We put an edge between two vertices v_i^r and v_j^s if both of the following hold:
    - v_i^r and v_j^s are in different triples, that is, $r \neq s$, and
    - their corresponding literals are consistent, that is, ℓ_i^r is not the negation of ℓ_j^s.
- We can easily build this graph from ϕ in polynomial time.

Notions of Intractability II
34.5 NP-complete problems

Example

Graph G for $\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$.
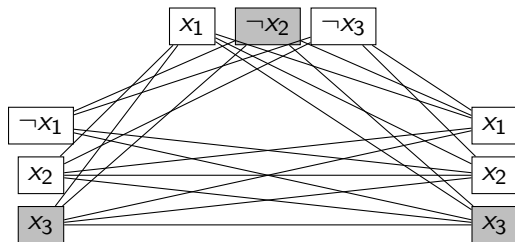
- A satisfying assignment of ϕ has $x_2 = 0$, $x_3 = 1$, and x_1 either 0 or 1.
- A corresponding clique of size $k = 3$ consists of the vertices corresponding to $\neg x_2$ from the first clause, x_3 from the second clause, and x_3 from the third clause.

Proof (Cont'd).

- We must show that this transformation of ϕ into G is a reduction.

- First, suppose that ϕ has a satisfying assignment. Then each clause C_r contains at least one literal ℓ_i^r that is assigned 1, and each such literal corresponds to a vertex v_i^r.

- Picking one such "true" literal from each clause yields a set V' of k vertices. We claim that V' is a clique.

- For any two vertices $v_i^r, v_j^s \in V'$, where $r \neq s$, both corresponding literals ℓ_i^r and ℓ_j^s map to 1 by the given satisfying assignment, and thus the literals cannot be complements. Thus, by the construction of G, the edge (v_i^r, v_j^s) belongs to E.

Proof (Cont'd).

- Conversely, suppose that G has a clique V' of size k.
- No edges in G connect vertices in the same triple, and so V' contains exactly one vertex per triple.
- We can assign 1 to each literal ℓ_i^r such that $v_i^r \in V'$ without fear of assigning 1 to both a literal and its complement, since G contains no edges between inconsistent literals.
- Each clause is satisfied, and so ϕ is satisfied.
- Any variables that do not correspond to a vertex in the clique may be set arbitrarily.

Notions of Intractability II
34.5 NP-complete problems

The vertex-cover problem

- A vertex cover of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V$ or $v \in V$ (or both).
- The size of a vertex cover is the number of vertices in it.
- The vertex-cover problem is the optimization problem of finding a vertex cover of minimum size in a graph.
- As a decision problem, we ask whether a graph has a vertex cover of a given size k.

Theorem (CLRS 34.12)

The vertex-cover problem is NP-complete.

Proof.

- To show that VERTEX-COVER belongs to NP.
- Suppose we are given graph $G = (V, E)$ and an integer k. The certificate we choose is the vertex cover $V' \subseteq V$ itself.
- The verification algorithm affirms that $|V'| = k$, and then it checks, for each edge $(u, v) \in E$, that $u \in V'$ or $v \in V'$. We can easily verify the certificate in polynomial time.
- We next prove that CLIQUE $\leqslant_P$ VERTEX-COVER, which shows that the vertex-cover problem is NP-hard.
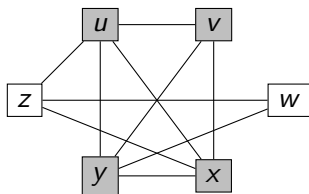
Proof (Cont'd).

- This reduction relies on the notion of the "complement" of a graph.
- Given an undirected graph $G = (V, E)$, we define the complement of G as $\bar{G} = (V, \bar{E})$, where $\bar{E} = \{(u, v) : u, v \in V, u \neq v, (u, v) \notin E\}$.
- In other words, $\bar{G}$ is the graph containing exactly those edges that are not in G.
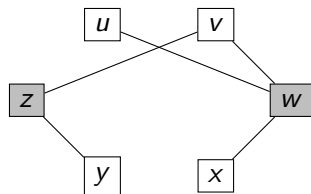
Notions of Intractability II
34.5 NP-complete problems

Example

An undirected graph $G = (V, E)$ with clique $V' = \{u, v, x, y\}$, and the graph $\bar{G} = (V, \bar{E})$ produced by the reduction algorithm that has vertex cover $V - V' = \{w, z\}$.

$$G = (V, E) \qquad\qquad \bar{G} = (V, \bar{E})$$

Proof (Cont'd).

- The reduction algorithm takes as input an instance $\langle G, k \rangle$ of the clique problem. It computes the complement $\bar{G}$, which we can easily do in polynomial time.

- The output of the reduction algorithm is the instance $\langle \bar{G}, |V| - k \rangle$ of the vertex-cover problem.

- To complete the proof, we show that this transformation is indeed a reduction: the graph G has a clique of size k if and only if the graph $\bar{G}$ has a vertex cover of size $|V| - k$.

Proof (Cont'd).

- Suppose that G has a clique $V' \subseteq V$ with $|V'| = k$. We claim that $V - V'$ is a vertex cover in $\bar{G}$.

- Let (u, v) be any edge in $\bar{E}$. Then, $(u, v) \notin E$, which implies that at least one of u or v does not belong to V', since every pair of vertices in V' is connected by an edge of E.

- Equivalently, at least one of u or v is in $V - V'$, which means that edge (u, v) is covered by $V - V'$.

- Since (u, v) was chosen arbitrarily from $\bar{E}$, every edge of $\bar{E}$ is covered by a vertex in $V - V'$.

- Hence, the set $V - V'$, which has size $|V| - k$, forms a vertex cover for $\bar{G}$.

Proof (Cont'd).

- Conversely, suppose that $\bar{G}$ has a vertex cover $V' \subseteq V$, where $|V'| = |V| - k$.

- Then, for all $u, v \in V$, if $(u, v) \in \bar{E}$, then $u \in V'$ or $v \in V'$ or both.

- The contrapositive of this implication is that for all $u, v \in V$, if $u \notin V'$ and $v \notin V'$, then $(u, v) \in E$.

- In other words, $V - V'$ is a clique, and it has size $|V| - |V'| = k$.

The subset-sum problem

- We are given a finite set S of positive integers and an integer target $t > 0$.

- We ask whether there exists a subset $S' \subseteq S$ whose elements sum to t.

- For example, if $S = \{1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16808, 17206, 117705, 117993\}$ and $t = 138457$, then the subset $S' = \{1, 2, 7, 98, 343, 686, 2409, 17206, 117705\}$ is a solution.

Theorem (CLRS 34.15)

The subset-sum problem is NP-complete.

Proof.

- To show that SUBSET-SUM is in NP, for an instance $\langle S, t \rangle$ of the problem, we let the subset S' be the certificate.

- A verification algorithm can check whether $t = \sum_{s \in S'} s$ in polynomial time.

- We now prove that 3-CNF-SAT $\leqslant_{\mathrm{P}}$ SUBSET-SUM, which shows that the subset-sum problem is NP-hard.

Proof (Cont'd).

- Given a 3-CNF formula ϕ over variables $x_1, x_2, \ldots, x_n$ with clauses $C_1, C_2, \ldots, C_k$, each containing exactly three distinct literals, the reduction algorithm constructs an instance $\langle S, t \rangle$ of the subset-sum problem such that ϕ is satisfiable if and only if there exists a subset of S whose sum is exactly t.

- Without loss of generality, we make two simplifying assumptions about the formula ϕ.
  - First, no clause contains both a variable and its negation, for such a clause is automatically satisfied by any assignment of values to the variables.
  - Second, each variable appears in at least one clause, because it does not matter what value is assigned to a variable that appears in no clauses.

Proof (Cont'd).

- The reduction creates two numbers in set S for each variable x_i and two numbers in S for each clause C_j.

- We shall create numbers in base 10, where each number contains $n + k$ digits and each digit corresponds to either one variable or one clause.

- Base 10 (and other bases, as we shall see) has the property we need of preventing carries from lower digits to higher digits.

Notions of Intractability II
34.5 NP-complete problems

Example

		x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	=	1	0	0	1	0	0	1
v_1'	=	1	0	0	0	1	1	0
v_2	=	0	1	0	0	0	0	1
v_2'	=	0	1	0	1	1	1	0
v_3	=	0	0	1	0	0	1	1
v_3'	=	0	0	1	1	1	0	0
s_1	=	0	0	0	1	0	0	0
s_1'	=	0	0	0	2	0	0	0
s_2	=	0	0	0	0	1	0	0
s_2'	=	0	0	0	0	2	0	0
s_3	=	0	0	0	0	0	1	0
s_3'	=	0	0	0	0	0	2	0
s_4	=	0	0	0	0	0	0	1
s_4'	=	0	0	0	0	0	0	2
t	=	1	1	1	4	4	4	4

Example (Cont'd)

- The formula in 3-CNF is $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, where
  $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$,
  $C_3 = (\neg x_1 \vee \neg x_2 \vee x_3)$, and $C_4 = (x_1 \vee x_2 \vee x_3)$.

- A satisfying assignment of ϕ is $\langle x_1 = 0, x_2 = 0, x_3 = 1 \rangle$.

- The set S produced by the reduction consists of the base-10 numbers $\{1001001, 1000110, 100001, 101110, 10011, 11100,$
  $1000, 2000, 100, 200, 10, 20, 1, 2\}$.

- The target t is 1114444.

- The subset S' contains v_1', v_2', and v_3, corresponding to the satisfying assignment, and slack variables s_1, s_1', s_2', s_3, s_4, and s_4' to achieve the target value of 4 in the digits labeled by C_1 through C_4.

Proof (Cont'd).

- We label each digit position by either a variable or a clause. The least significant k digits are labeled by the clauses, and the most significant n digits are labeled by variables.

- The target t has a 1 in each digit labeled by a variable and a 4 in each digit labeled by a clause.

- For each variable x_i, set S contains two integers v_i and v_i'. Each of v_i and v_i' has a 1 in the digit labeled by x_i and a 0 in the other variable digits.

- If literal x_i appears in clause C_j, then the digit labeled by C_j in v_i contains a 1. If literal $\neg x_i$ appears in clause C_j, then the digit labeled by C_j in v_i' contains a 1. All other digits labeled by clauses in v_i and v_i' are 0.

- Note that all v_i and v_i' values in set S are unique.

Proof (Cont'd).

- For each clause C_j, set S contains two integers s_j and s_j'. Each of s_j and s_j' has a 0 in all digits other than the one labeled by C_j.

- For s_j, there is a 1 in the C_j digit, and s_j' has a 2 in this digit. These integers are "slack variables," which we use to get each clause-labeled digit position to add to the target value of 4.

- Note that the greatest sum of digits in any one digit position is 6, which occurs in the digits labeled by clauses (three 1 from the v_i and v_i' values, plus 1 and 2 from the s_j and s_j' values).

- Interpreting these numbers in base 10, therefore, no carries can occur from lower digits to higher digits.

- In fact, any base b, where $b \geqslant 7$, would work. The first instance is the set S and target t interpreted in base 7.

Proof (Cont'd).

- We can perform the reduction in polynomial time. The set S contains $2n + 2k$ values, each of which has $n + k$ digits, and the time to produce each digit is polynomial in $n + k$. The target t has $n + k$ digits, and the reduction produces each in constant time.

- We now show that the 3-CNF formula ϕ is satisfiable if and only if there exists a subset $S' \subseteq S$ whose sum is t.

Notions of Intractability II
34.5 NP-complete problems

Proof (Cont'd).

- First, suppose that ϕ has a satisfying assignment. For $i = 1, 2, \ldots, n$, if $x_i = 1$ in this assignment, then include v_i in S'. Otherwise, include v'_1.

- In other words, we include in S' exactly the v_i and v'_i values that correspond to literals with the value 1 in the satisfying assignment.

- Having included either v_i or v'_i, but not both, for all i, and having put 0 in the digits labeled by variables in all s_j and s'_j, we see that for each variable-labeled digit, the sum of the values of S' must be 1, which matches those digits in the target t.

Proof (Cont'd).

- Because each clause is satisfied, the clause contains some literal with the value 1. Therefore, each digit labeled by a clause has at least one 1 contributed to its sum by a v_i or v_i' value in S'.

- In fact, 1, 2, or 3 literals may be 1 in each clause, and so each clause-labeled digit has a sum of 1, 2, or 3 from the v_i and v_i' values in S'.

- We achieve the target of 4 in each digit labeled by clause C_j by including in S' the appropriate nonempty subset of slack variables $\{s_j, s_j'\}$.

- Since we have matched the target in all digits of the sum, and no carries can occur, the values of S' sum to t.

Proof (Cont'd).

- Now, suppose that there is a subset $S' \subseteq S$ that sums to t. The subset S' must include exactly one of v_i and v_i' for each $i = 1, 2, \ldots, n$, for otherwise the digits labeled by variables would not sum to 1.

- If $v_i \in S'$, we set $x_i = 1$. Otherwise, $v_i' \in S'$, and we set $x_i = 0$. We claim that every clause C_j, for $j = 1, 2, \ldots, k$, is satisfied by this assignment.

- To prove this claim, note that to achieve a sum of 4 in the digit labeled by C_j, the subset S' must include at least one v_i or v_i' value that has a 1 in the digit labeled by C_j, since the contribution of the slack variables s_j and s_j' together sum to at most 3.

Proof (Cont'd).

- If S' includes a v_i that has a 1 in the digit labeled by C_j, then the literal x_i appears in clause C_j. Since we have set $x_i = 1$ when $v_i \in S'$, clause C_j is satisfied.

- If S' includes a v_i' that has a 1 in the digit labeled by C_j, then the literal $\neg x_i$ appears in clause C_j. Since we have set $x_i = 0$ when $v_i' \in S'$, clause C_j is again satisfied.

- Thus, all clauses of ϕ are satisfied, which completes the proof.

Lecture 13
Review

Review

Contents

- Analysis of Algorithms I
- Analysis of Algorithms II
- Divide and Conquer I
- Divide and Conquer II
- Dictionaries I
- Dictionaries II

- Graphs I
- Graphs II
- Exhaustive Search and Generation I
- Exhaustive Search and Generation II
- Notions of Intractability I
- Notions of Intractability II

Analysis of Algorithms I

Contents

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009

Analysis of Algorithms II

Contents

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009

Divide and Conquer I

Contents

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009

Divide and Conquer II

Contents

- Karatsuba's algorithm
- Strassen's algorithm for matrix multiplication . . . . CLRS 4.2
- Minimum and maximum . . . . CLRS 9.1
- Selection in worst-case linear time . . . . CLRS 9.3

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009

Dictionaries I

Contents

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009

Dictionaries II

Contents

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009

Graphs I

Contents

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009

Graphs II

Contents

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009

Exhaustive Search and Generation I

Contents

- Exercise: All subsets
- Exercise: All permutations

Reading

- R. E. Neapolitan. *Foundations of Algorithms*. Jones & Bartlett Learning, 5th edition, 2015

Exhaustive Search and Generation II

Contents

Reading

- R. E. Neapolitan. *Foundations of Algorithms*. Jones & Bartlett Learning, 5th edition, 2015

Notions of Intractability I

Contents

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009

Notions of Intractability II

Contents

Reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009