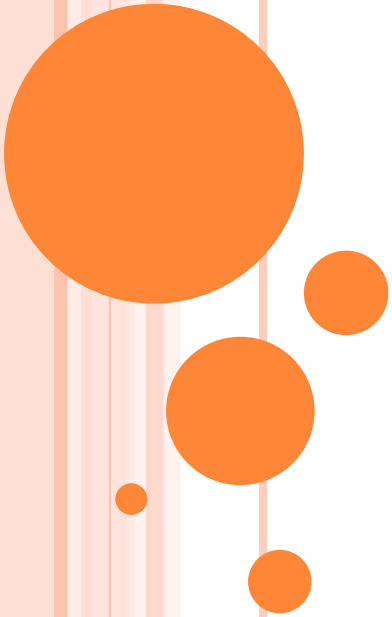


# Algorithm Review Class

# Question Answering

- **QA class (ONLY ONCE)**:
  - **19 June, 2023, Monday, 18th week**
  - **15:00 - 17:00**
  - **Tencent meeting (腾讯会议)**

- **Exam time:**
  - **27 June, 2023, Tuesday, N320**
  - **10:20-12:10**

| Chapter | Topic |
|---|---|
| Introduction | • Asymptotic notation: $O$, $\Omega$, $\Theta$<br>• Common rules for asymptotic analysis |
| Divide-and-conquer | • Hallmarks: optimal substructure and independent sub-problem<br>• Master theorem and its proof<br>• Mergesort ($O(n\log n)$)<br>• Selection ($O(n)$) |
| Graph algorithms | • *explore* ($O(\|E\|)$) and DFS ($O(\|V\|+\|E\|)$)<br>• DAG: topological ordering, Shortest paths ($O(\|V\|+\|E\|)$)<br>• BFS ($O(\|V\|+\|E\|)$)<br>• Dijkstra's algorithm ($O((\|V\|+\|E\|)\log\|V\|)$)<br>• Priority queue implementations: array, binary heap and $d$-ary heap<br>• Bellman-Ford algorithm ($O(\|V\|\|E\|)$) |
| Greedy algorithms | • MST: Kruskal's algorithm ($O((\|V\|+\|E\|)\log\|V\|)$), Prim's algorithm ($O((\|V\|+\|E\|)\log\|V\|)$)<br>• Huffman encoding ($O(n\log n)$)<br>• Greedy algorithm for set cover (approximation ratio $\ln n$) |
| Dynamic programming | • Hallmarks: optimal substructure and overlapping sub-problem<br>• Longest Increasing Subsequence (LIS) ($O(n)$)<br>• Edit distance ($O(mn)$)<br>• Knapsack $O(nW)$<br>• Chain matrix multiplication ($O(n^3)$) |
| NP-complete | • P, NP, Reduction, NP-completeness (NPC)<br>• Examples for reduction<br>• Approximated algorithm for vertex cover (approximation ratio 2) |

# Examples of question type

- Filling
- Choice
- True/False
- Short Questions
- Design and Analysis (50 points)
  - D&C, DP, Greedy, …

- How to review?
  - PPTs + Exercises

# HW2-1

A *contiguous subsequence* of a list $S$ is a subsequence made up of consecutive elements of $S$. For instance, if $S$ is

$$5, 15, -30, 10, -5, 40, 10,$$

then $15, -30, 10$ is a contiguous subsequence but $5, 15, 40$ is not. Give a linear-time algorithm for the following task:

> *Input:* A list of numbers, $a_1, a_2, \ldots, a_n$.
>
> *Output:* The contiguous subsequence of maximum sum (a subsequence of length zero has sum zero).

For the preceding example, the answer would be $10, -5, 40, 10$, with a sum of $55$.

(*Hint:* For each $j \in \{1, 2, \ldots, n\}$, consider contiguous subsequences ending exactly at position $j$.)

*Subproblems:* Define an array of subproblems $D(i)$ for $0 \leq i \leq n$. $D(i)$ will be the largest sum of a (possibly empty) contiguous subsequence ending exactly at position $i$.

*Algorithm and Recursion:* The algorithm will initialize $D(0) = 0$ and update the $D(i)$'s in ascending order according to the rule:

$$D(i) = \max\{0, D(i-1) + a_i\}$$

The largest sum is then given by the maximum element $D(i)^*$ in the array $D$. The contiguous subsequence of maximum sum will terminate at $i^*$. Its beginning will be at the first index $j \leq i^*$ such that $D(j-1) = 0$, as this implies that extending the sequence before $j$ will only decrease its sum.

*Correctness:* The contiguous subsequence of largest sum ending at $i$ will either be empty or contain $a_i$. In the first case, the value of the sum will be 0. In the second case, it will be the sum of $a_i$ and the best sum we can get ending at $i-1$, i.e. $D(i-1) + a_i$. Because we are looking for the largest sum, $D(i)$ will be the maximum of these two possibilities.

*Running Time:* The running time for this algorithm is $O(n)$, as we have $n$ subproblems and the solution of each can be computed in constant time. Moreover, the identification of the optimal subsequence only requires a single $O(n)$ time pass through the array $D$.

# HW2-2

Assume $aa=ab=bb=b$，$ac=bc=ca=a$，$ba=cb=cc=c$ on the set $A=\{a, b, c\}$. Given a string $x = x_1 x_2 \ldots x_n$ , design a dynamic programming algorithm to check whether there is a computational order such that the final result is $a$.

For example,

$x=bbbba \Rightarrow$ Yes. $(b(bb))(ba) = (bb)(ba) = b(ba) = bc = a$

$x=bca \Rightarrow$ No. $(bc)a = aa = b, b(ca) = ba = c$

Answer: given a string $x_i \cdots x_j$, let $X[i, j]$ be a 3×2 matrix which stores all possible results under the given computational rules, companied with the corresponding splitting position $k$.

$$X[i, j] = X[i, k] \cdot X[k+1, j], 1 \le i < j \le n, i \le k < j$$
$$X[i, i] = x_i, i = 1, 2, \ldots, n$$

1. *for* $i \leftarrow 1$ *to* $n$
2. $\quad X[i, i] \leftarrow x_i$
3. *for* $r \leftarrow 2$ *to* $n-1$
4. $\quad$ *for* $i \leftarrow 1$ *to* $n-r+1$
5. $\quad\quad j \leftarrow i+r-1$
6. $\quad\quad$ *for* $k \leftarrow i$ *to* $j-1$
7. $\quad\quad\quad X[i, j] \leftarrow X[i, k] \cdot X[k+1, j]$
8. *for* $k \leftarrow 1$ *to* $n-1$
9. $\quad$ *if* $X[1, k] \cdot X[k+1, n] = "a"$ *then return* "1"
10. *return* "0"

# Exercise 0.1

In each of the following situations, indicate whether $f = O(g)$, or $f = \Omega(g)$, or both (in which case $f = \Theta(g)$).

| | $f(n)$ | $g(n)$ |
|---|---|---|
| (a) | $n - 100$ | $n - 200$ |
| (b) | $n^{1/2}$ | $n^{2/3}$ |
| (c) | $100n + \log n$ | $n + (\log n)^2$ |
| (d) | $n \log n$ | $10n \log 10n$ |
| (e) | $\log 2n$ | $\log 3n$ |
| (f) | $10 \log n$ | $\log(n^2)$ |
| (g) | $n^{1.01}$ | $n \log^2 n$ |
| (h) | $n^2 / \log n$ | $n(\log n)^2$ |
| (i) | $n^{0.1}$ | $(\log n)^{10}$ |
| (j) | $(\log n)^{\log n}$ | $n / \log n$ |
| (k) | $\sqrt{n}$ | $(\log n)^3$ |
| (l) | $n^{1/2}$ | $5^{\log_2 n}$ |
| (m) | $n2^n$ | $3^n$ |
| (n) | $2^n$ | $2^{n+1}$ |
| (o) | $n!$ | $2^n$ |
| (p) | $(\log n)^{\log n}$ | $2^{(\log_2 n)^2}$ |
| (q) | $\sum_{i=1}^{n} i^k$ | $n^{k+1}$ |



a) $n - 100 = \Theta(n - 200)$

b) $n^{1/2} = O(n^{2/3})$

c) $100n + \log n = \Theta(n + (\log n)^2)$

d) $n \log n = \Theta(10n \log 10n)$

e) $\log 2n = \Theta(\log 3n)$

f) $10 \log n = \Theta(\log(n^2))$

g) $n^{1.01} = \Omega\left(n\left(\log^2 n\right)\right)$

h) $n^2 / \log n = \Omega(n(\log n)^2)$

i) $n^{0.1} = \Omega((\log n)^{10})$

j) $(\log n)^{\log n} = \Omega(n / \log n)$

k) $\sqrt{n} = \Omega((\log n)^3)$

l) $n^{1/2} = O(5^{\log_2 n})$

m) $n2^n = O(3^n)$

n) $2^n = \Theta(2^{n+1})$

o) $n! = \Omega(2^n)$

p) $(\log n)^{\log n} = O(2^{(\log_2 n)^2})$

q) $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$

$$a)\frac{n-100}{n-200}=1+\frac{100}{n-200}\le 2\quad (n\to\infty)$$

$$c)\left(\log n\right)^2=O(n)$$

$$g)n^{1.01}=n\cdot n^{0.01}=\Omega(n\left(\log^2 n\right))$$

$$h)\frac{n^2/\log n}{n\left(\log n\right)^2}=\frac{n}{\left(\log n\right)^3}\to\infty\quad (n\to\infty)$$

$$j)\frac{\left(\log n\right)^{\log n}}{n/\log n}=\frac{\left(\log n\right)^{\log n+1}}{n},\left(\log n\right)^{\log n}=n^{\log\log n}$$

$$or\ let\ k=\log n, so\ \frac{k^{k+1}}{2^k}=\left(\frac{k}{2}\right)^k\cdot k\to\infty\left(k\to\infty\right)$$

$$k)\frac{\sqrt{n}}{\left(\log n\right)^3}=\sqrt{\frac{n}{\left(\log n\right)^6}}$$

a) $n-100=\Theta(n-200)$
b) $n^{1/2}=O(n^{2/3})$
c) $100n+\log n=\Theta(n+(\log n)^2)$
d) $n\log n=\Theta(10n\log 10n)$
e) $\log 2n=\Theta(\log 3n)$
f) $10\log n=\Theta(\log(n^2))$
g) $n^{1.01}=\Omega(n(\log^2 n))$
h) $n^2/\log n=\Omega(n(\log n)^2)$
i) $n^{0.1}=\Omega((\log n)^{10})$
j) $(\log n)^{\log n}=\Omega(n/\log n)$
k) $\sqrt{n}=\Omega((\log n)^3)$

$l) 5^{\log_2 n} = n^{\log_2 5}$

$o) n! = n \times (n-1) \times (n-2) \times \ldots \times 1 > 2 \times 2 \times 2 \times \ldots \times 2$ (when $n \geq 4$)

so $n! = \Omega(2^n)$

$p) 2^{(\log_2 n)^2} = 2^{(\log_2 n) \cdot (\log_2 n)} = n^{\log_2 n}$

l) $n^{1/2} = O(5^{\log_2 n})$
m) $n2^n = O(3^n)$
n) $2^n = \Theta(2^{n+1})$
o) $n! = \Omega(2^n)$
p) $(\log n)^{\log n} = O(2^{(\log_2 n)^2})$
q) $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$

prove： $n! = o(n^n)$

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left[1+\theta\left(\frac{1}{n}\right)\right] \text{ (Stirling's formula)}$$

$$\lim_{n\to\infty}\frac{n!}{n^n} = \frac{\sqrt{2\pi n}\left[1+\theta\left(\frac{1}{n}\right)\right]}{e^n} = 0 \Rightarrow n! = o(n^n)$$

$n! = n\times(n-1)\times(n-2)\times\ldots\times 1 < n\times n\times n\times\ldots\times n$

so $n! = o(n^n)$

prove: $\log(n!)=\theta(n\log n)$

$$\log(n!) = \sum_{i=1}^{n}\log i \le \sum_{i=1}^{n}\log n = n\log n \Rightarrow \log(n!)=O(n\log n)$$

if $n$ is even (when $n$ is odd, use $\lfloor n/2 \rfloor$ to replace $n/2$):

$$\log(n!) \ge \sum_{i=n/2}^{n}\log i \ge \sum_{i=n/2}^{n}\log(n/2) = n/2\log(n/2) = (n\log n)/2 - n/2$$

when $n \ge 4$, $(n\log n)/2 - n/2 \ge (n\log n)/4$ (since $(n\log n) \ge 2n$)

so $\log(n!) \ge (n\log n)/4 \Rightarrow \log(n!)=\Omega(n\log n)$

therefore, $\log(n!)=\theta(n\log n)$

$q$)there are $n$ difference formulae:

$$\{i=1\}: 2^{k+1}-1^{k+1}=\binom{k+1}{1}\times 1^k+\binom{k+1}{2}\times 1^{k-1}+\ldots+\binom{k+1}{k}\times 1+1$$

$$\{i=2\}: 3^{k+1}-2^{k+1}=\binom{k+1}{1}\times 2^k+\binom{k+1}{2}\times 2^{k-1}+\ldots+\binom{k+1}{k}\times 2+1$$

$$\{i=3\}: 4^{k+1}-3^{k+1}=\binom{k+1}{1}\times 3^k+\binom{k+1}{2}\times 3^{k-1}+\ldots+\binom{k+1}{k}\times 3+1$$

...

$$\{i=j\}: (j+1)^{k+1}-j^{k+1}=\binom{k+1}{1}j^k+\binom{k+1}{2}j^{k-1}+\ldots+\binom{k+1}{k}j+1$$

$$\{i=j+1\}: (j+2)^{k+1}-(j+1)^{k+1}=\binom{k+1}{1}(j+1)^k+\binom{k+1}{2}(j+1)^{k-1}+\ldots+\binom{k+1}{k}(j+1)+1$$

...

$$\{i=n\}: (n+1)^{k+1}-n^{k+1}=\binom{k+1}{1}n^k+\binom{k+1}{2}n^{k-1}+\ldots+\binom{k+1}{k}n+1$$

then add these $n$ formulae together (only the first term and the last term left):

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

$$n^{k+1}\le(n+1)^{k+1}\le\left[\binom{k+1}{1}+\binom{k+1}{2}+\ldots+\binom{k+1}{k}\right]\sum_{i=1}^{n}i^k\Rightarrow n^{k+1}=O\left(\sum_{i=1}^{n}i^k\right)$$

$$\sum_{i=1}^{n}i^k\le n\cdot n^k=n^{k+1}\Rightarrow n^{k+1}=\Omega\left(\sum_{i=1}^{n}i^k\right)$$

so $\sum_{i=1}^{n}i^k=\theta(n^{k+1})$

# EXERCISE 0.2

Show that, if $c$ is a positive real number, then $g(n) = 1 + c + c^2 + \cdots + c^n$ is:

(a) $\Theta(1)$ if $c < 1$.

(b) $\Theta(n)$ if $c = 1$.

(c) $\Theta(c^n)$ if $c > 1$.

The moral: in big-$\Theta$ terms, the sum of a geometric series is simply the first term if the series is strictly decreasing, the last term if the series is strictly increasing, or the number of terms if the series is unchanging.

0.2. By the formula for the sum of a partial geometric series, for $c \neq 1$: $g(n) = \frac{1-c^{n+1}}{1-c} = \frac{c^{n+1}-1}{c-1}$.

a) $1 > 1 - c^{n+1} > 1 - c$. So: $\frac{1}{1-c} > g(n) > 1$.

b) For $c = 1$, $g(n) = 1 + 1 + \cdots + 1 = n + 1$.

c) $c^{n+1} > c^{n+1} - 1 > c^n$. So: $\frac{c^{n+1}}{c-1} > g(n) > \frac{c^n}{c-1}$.

# EXERCISE 0.4

Is there a faster way to compute the $n$th Fibonacci number than by `fib2` (page 13)? One idea involves *matrices*.

We start by writing the equations $F_1 = F_1$ and $F_2 = F_0 + F_1$ in matrix notation:

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}.$$

Similarly,

$$\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

and in general

$$\boxed{\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}.}$$

So, in order to compute $F_n$, it suffices to raise this $2 \times 2$ matrix, call it $X$, to the $n$th power.

(a) Show that two $2 \times 2$ matrices can be multiplied using 4 additions and 8 multiplications.

But how many matrix multiplications does it take to compute $X^n$?

(b) Show that $O(\log n)$ matrix multiplications suffice for computing $X^n$. (*Hint:* Think about computing $X^8$.)

# EXERCISE 0.4

a) For any $2 \times 2$ matrices $X$ and $Y$:

$$XY = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} \begin{pmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{pmatrix} = \begin{pmatrix} x_{11}y_{11} + x_{12}y_{21} & x_{11}y_{12} + x_{12}y_{22} \\ x_{21}y_{11} + x_{22}y_{21} & x_{21}y_{12} + x_{22}y_{22} \end{pmatrix}$$

This shows that every entry of $XY$ is the addition of two products of the entries of the original matrices. Hence every entry can be computed in 2 multiplications and one addition. The whole matrix can be calculated in 8 multiplications and 4 additions.

b) First, consider the case where $n = 2^k$ for some positive integer $k$. To compute, $X^{2^k}$, we can recursively compute $Y = X^{2^{k-1}}$ and then square $Y$ to have $Y^2 = X^{2^k}$ Unfolding the recursion, this can be seen as repeatedly squaring $X$ to obtain $X^2, X^4, \cdots, X^{2^k} = X^n$. At every squaring, we are doubling the exponent of $X$, so that it must take $k = \log n$ matrix multiplications to produce $X^n$. This method can be easily generalized to numbers that are not powers of 2, using the following recursion:

$$X^n = \begin{cases} (X^{\lfloor n/2 \rfloor})^2 & \text{if } n \text{ is even} \\ X \cdot (X^{\lfloor n/2 \rfloor})^2 & \text{if } n \text{ is odd} \end{cases}$$

The algorithm still requires $O(\log n)$ matrix multiplications, of which $\log n$ are squares and at most $\log n$ are multiplications by $X$.

# Exercise 2.3

Section 2.2 describes a method for solving recurrence relations which is based on analyzing the recursion tree and deriving a formula for the work done at each level. Another (closely related) method is to expand out the recurrence a few times, until a pattern emerges. For instance, let's start with the familiar $T(n) = 2T(n/2) + O(n)$. Think of $O(n)$ as being $\leq cn$ for some constant $c$, so: $T(n) \leq 2T(n/2) + cn$. By repeatedly applying this rule, we can bound $T(n)$ in terms of $T(n/2)$, then $T(n/4)$, then $T(n/8)$, and so on, at each step getting closer to the value of $T(\cdot)$ we do know, namely $T(1) = O(1)$.

$$
\begin{aligned}
T(n) &\leq 2T(n/2) + cn \\
&\leq 2[2T(n/4) + cn/2] + cn \;=\; 4T(n/4) + 2cn \\
&\leq 4[2T(n/8) + cn/4] + 2cn \;=\; 8T(n/8) + 3cn \\
&\leq 8[2T(n/16) + cn/8] + 3cn \;=\; 16T(n/16) + 4cn \\
&\vdots
\end{aligned}
$$

# Exercise 2.3

A pattern is emerging... the general term is

$$T(n) \leq 2^k T(n/2^k) + kcn.$$

Plugging in $k = \log_2 n$, we get $T(n) \leq nT(1) + cn \log_2 n = O(n \log n)$.

(a) Do the same thing for the recurrence $T(n) = 3T(n/2) + O(n)$. What is the general $k$th term in this case? And what value of $k$ should be plugged in to get the answer?

(b) Now try the recurrence $T(n) = T(n-1) + O(1)$, a case which is not covered by the master theorem. Can you solve this too?

# Exercise 2.3

a)

$$T(n) \leq 3T\left(\frac{n}{2}\right) + cn \leq \ldots \leq 3^k T\left(\frac{n}{2^k}\right) + cn \sum_{i=0}^{k-1} \left(\frac{3}{2}\right)^i =$$

$$= 3^k T\left(\frac{n}{2^k}\right) + 2cn\left(\left(\frac{3}{2}\right)^k - 1\right)$$

For $k = \log_2 n$, $T(\frac{n}{2^k}) = T(1) = d = O(1)$. Then:

$$T(n) = dn^{log_2 3} + 2cn\left(\frac{n^{log_2 3}}{n} - 1\right) = \Theta(n^{log_2 3})$$

as predicted by the Master theorem.

b) $T(n) \leq T(n-1) + c \leq \ldots \leq T(n-k) + kc$. For $k = n$, $T(n) = T(0) + nc = \Theta(n)$.

$$T(n) \leq 3[3T(n/4) + cn/2] + cn \leq 3^2 T(n/4) + \frac{3}{2}cn + cn$$

$$\leq 3^2[3T(n/8) + cn/4] + \frac{3}{2}cn + cn$$

$$\leq 3^3 T(n/8) + [\left(\frac{3}{2}\right)^2 + \frac{3}{2} + 1]cn$$

# Exercise 2.4

Suppose you are choosing between the following three algorithms:

- Algorithm $A$ solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.
- Algorithm $B$ solves problems of size $n$ by recursively solving two subproblems of size $n-1$ and then combining the solutions in constant time.
- Algorithm $C$ solves problems of size $n$ by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

What are the running times of each of these algorithms (in big-$O$ notation), and which would you choose?

# Exercise 2.4

a) This is a case of the Master theorem with $a = 5, b = 2, d = 1$. As $a > b^d$, the running time is $O(n^{\log_b a}) = O(n^{\log_2 5}) = O(n^{2.33})$.

b) $T(n) = 2T(n-1)+C$, for some constant $C$. $T(n)$ can then be expanded to $C\sum_{i=0}^{n-1} 2^i + 2^n T(0) = O(2^n)$.

c) This is a case of the Master theorem with $a = 9, b = 3, d = 2$. As $a = b^d$, the running time is

**$O(n^d \log_3 n) = O(n^2 \log_3 n)$**

$$T(n) = 2T(n-1)+C = 2[2T(n-2)+C]+C$$
$$= 2^2 T(n-2)+(2+1)C = 2^2[2T(n-3)+C]+(2+1)C$$
$$= 2^3 T(n-3)+(2^2+2+1)C$$
$$= 2^k T(n-k)+C\sum_{i=0}^{k-1} 2^i$$

# Exercise 2.5

Solve the following recurrence relations and give a $\Theta$ bound for each of them.

(a) $T(n) = 2T(n/3) + 1$

(b) $T(n) = 5T(n/4) + n$

(c) $T(n) = 7T(n/7) + n$

(d) $T(n) = 9T(n/3) + n^2$

(e) $T(n) = 8T(n/2) + n^3$

(f) $T(n) = 49T(n/25) + n^{3/2} \log n$

(g) $T(n) = T(n-1) + 2$

(h) $T(n) = T(n-1) + n^c$, where $c \geq 1$ is a constant

(i) $T(n) = T(n-1) + c^n$, where $c > 1$ is some constant

(j) $T(n) = 2T(n-1) + 1$

(k) $T(n) = T(\sqrt{n}) + 1$

$$49^i \times \left(\frac{n}{25^i}\right)^{\frac{3}{2}} \log \frac{n}{25^i} = n^{\frac{3}{2}} \times \left(\frac{49}{125}\right)^i \times (\log n - i \log 25) \le n^{\frac{3}{2}} \times \left(\frac{49}{125}\right)^i \times \log n$$

$$i \le \log_{25} n \Rightarrow i \log 25 \le \log n$$

# Exercise 2.5

a) $T(n) = 2T(n/3) + 1 = \Theta(n^{\log_3 2})$ by the Master theorem.

b) $T(n) = 5T(n/4) + n = \Theta(n^{\log_4 5})$ by the Master theorem.

c) $T(n) = 7T(n/7) + n = \Theta(n \log_7 n)$ by the Master theorem.

d) $T(n) = 9T(n/3) + n^2 = \Theta(n^2 \log_3 n)$ by the Master theorem.

e) $T(n) = 8T(n/2) + n^3 = \Theta(n^3 \log_2 n)$ by the Master theorem.

f) $T(n) = 49T(n/25) + n^{3/2} \log n = \Theta(n^{3/2} \log n)$. Apply the same reasoning of the proof of the Master Theorem. The contribution of level $i$ of the recursion is

$$\left(\frac{49}{25^{3/2}}\right)^i n^{3/2} \log\left(\frac{n}{25^{3/2}}\right) = \left(\frac{49}{125}\right)^i O(n^{3/2} \log n)$$

Because the corresponding geometric series is dominated by the contribution of the first level, we obtain $T(n) = O(n^{3/2} \log n)$. But, $T(n)$ is clearly $\Omega(n^{3/2} \log n)$. Hence, $T(n) = \Theta(n^{3/2} \log n)$.

g) $T(n) = T(n-1) + 2 = \Theta(n)$.

$$h) T(n) = T(n-1) + n^c = T(n-2) + (n-1)^c + n^c$$

$$= T(n-3) + (n-2)^c + (n-1)^c + n^c = \sum_{i=1}^{n} i^c + T(0) = \Theta(n^{c+1})$$

# HOMEWORK

- Ex.2.5

(a) $T(n) = 2T(n/3) + 1$

(b) $T(n) = 5T(n/4) + n$

(c) $T(n) = 7T(n/7) + n$

(i) $T(n) = T(n-1) + c^n$, where $c > 1$ is some constant

(j) $T(n) = 2T(n-1) + 1$

(k) $T(n) = T(\sqrt{n}) + 1$

a) $T(n) = 2T(n/3) + 1 = \Theta(n^{\log_3 2})$ by the Master theorem.

b) $T(n) = 5T(n/4) + n = \Theta(n^{\log_4 5})$ by the Master theorem.

c) $T(n) = 7T(n/7) + n = \Theta(n \log_7 n)$ by the Master theorem.



$$i)T(n) = T(n-1) + c^n = T(n-2) + c^{n-1} + c^n$$

$$= T(n-3) + c^{n-2} + c^{n-1} + c^n = \sum_{i=1}^{n} c^i + T(0) = \Theta(c^n)$$

$$j)T(n) = 2T(n-1) + 1 = 2[2T(n-2) + 1] + 1$$

$$= 2^2 T(n-2) + (2+1) = 2^2[2T(n-3) + 1] + (2+1)$$

$$= 2^3 T(n-3) + (2^2 + 2 + 1) = 2^k T(n-k) + \sum_{i=0}^{k-1} 2^i = 2^n T(0) + \sum_{i=0}^{n-1} 2^i = \Theta(2^n)$$

$$k)T(n) = [T(\sqrt{\sqrt{n}}) + 1] + 1 = [T(\sqrt{\sqrt{\sqrt{n}}}) + 1] + 2$$

$$= k + T(b) \quad s.t. \quad b^{\overbrace{2*2...*2}^{k}} = n$$

$$\Rightarrow b^{2^k} = n \Rightarrow k = \log\log_b n \Rightarrow T(n) = O(\log\log n)$$

# Exercise 2.12

How many lines, as a function of $n$ (in $\Theta(\cdot)$ form), does the following program print? Write a recurrence and solve it. You may assume $n$ is a power of 2.

```
function f(n)
  if n > 1:
    print_line(``still going'')
    f(n/2)
    f(n/2)
```

Each function call prints one line and calls the same function on input of half the size, so the number of printed lines is $P(n) = 2P(\frac{n}{2}) + 1$. By the Master theorem $P(n) = \Theta(n)$.

# Exercise 2.14

You are given an array of $n$ elements, and you notice that some of the elements are duplicates; that is, they appear more than once in the array. Show how to remove all duplicates from the array in time $O(n \log n)$.

Sort the array in time $O(n \log n)$. Then, in one linear time pass copy the elements to a new array, eliminating the duplicates.

# Exercise 2.15

In our median-finding algorithm (Section 2.4), a basic primitive is the `split` operation, which takes as input an array $S$ and a value $v$ and then divides $S$ into three sets: the elements less than $v$, the elements equal to $v$, and the elements greater than $v$. Show how to implement this `split` operation *in place*, that is, without allocating new memory.

The simplest way to implement **split** in place is the following:

```
function split(a[1, ··· , n], v)
store= 1
for i = 1 to n:
    if a[i] < v:
        swap a[i] and a[store]
        store = store +1
for i =store to n:
    if a[i] = v:
        swap a[i] and a[store]
        store = store +1
```

The first for loop passes through the array bringing the elements smaller than v to the front, so splitting the array into a subarray of elements smaller than $v$ and one of elements larger or equal to $v$. The second for loop uses the same strategy on the latter subarray to split into a subarray of elements equal to $v$ and one of elements larger than $v$. The body of both for loops takes constant time, so the running time is $O(n)$.

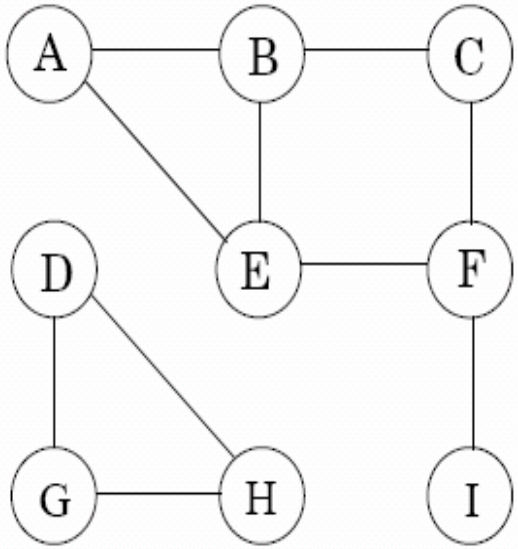
# Exercise 2.17

Given a sorted array of distinct integers $A[1, \ldots, n]$, you want to find out whether there is an index $i$ for which $A[i] = i$. Give a divide-and-conquer algorithm that runs in time $O(\log n)$.

First examine the middle element $A[\frac{n}{2}]$. If $A[\frac{n}{2}] = \frac{n}{2}$, we are done. If $A[\frac{n}{2}] > \frac{n}{2}$, then every subsequent element will also be bigger than its index since the array values grow at least as fast as the indices. Similarly, if $A[\frac{n}{2}] < \frac{n}{2}$, then every previous element in the array will be less than its index by the same reasoning. So after the comparison, we only need to examine half of the array. We can recurse on the appropriate half of the array. If we continue this division until we get down to a single element and this element does not have the desired property, then such an element does not exist in $A$. We do a constant amount of work with each function call. So our recurrence relation is $T(n) = T(n/2) + O(1)$, which solves to $T(n) = O(\log n)$.

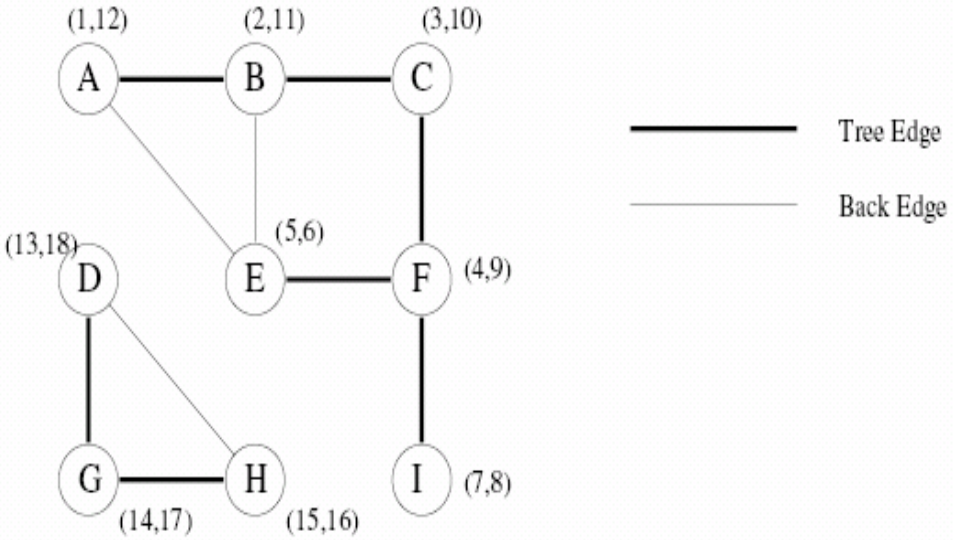
# EXERCISE 3.1

Perform a depth-first search on the following graph; whenever there's a choice of vertices, pick the one that is alphabetically first. Classify each edge as a tree edge or back edge, and give the `pre` and `post` number of each vertex.



# Exercise 3.1

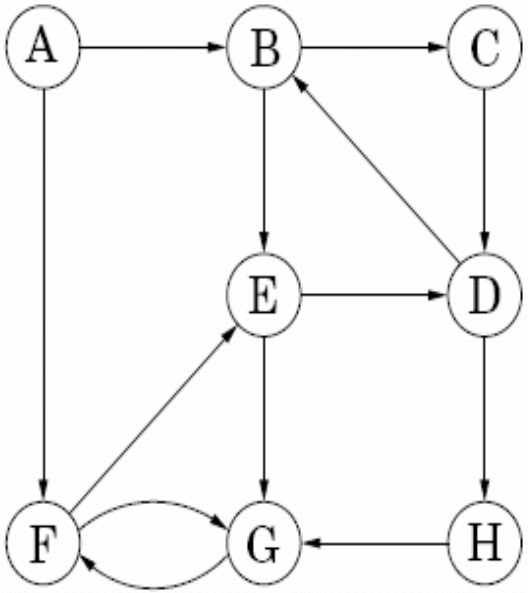
The figure below gives the `pre` and `post` numbers of the vertices in parentheses. The tree and back edges are marked as indicated.



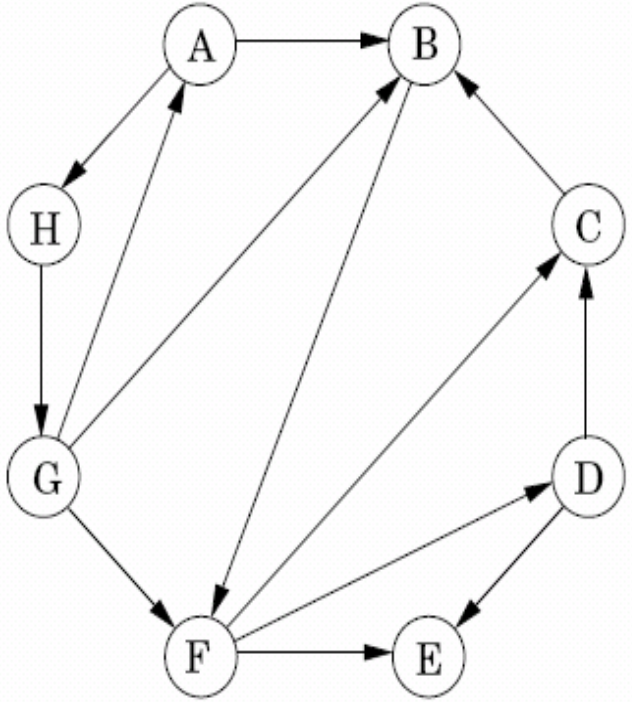
# EXERCISE 3.2

Perform depth-first search on each of the following graphs; whenever there's a choice of vertices, pick the one that is alphabetically first. Classify each edge as a tree edge, forward edge, back edge, or cross edge, and give the `pre` and `post` number of each vertex.

(a)

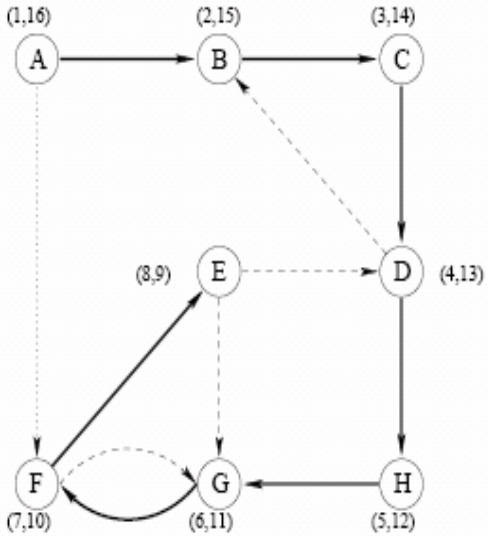



(b)

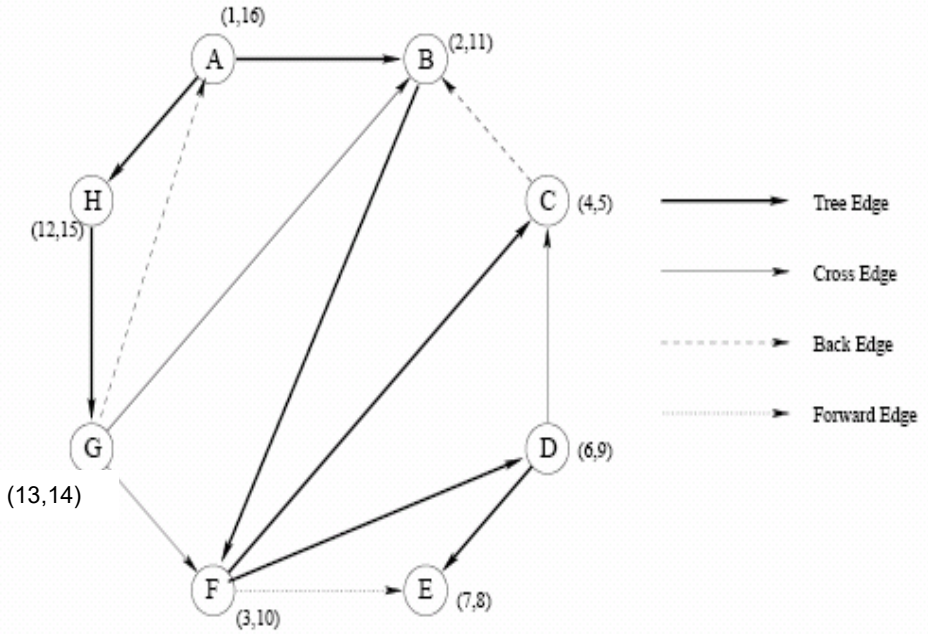


# EXERCISE 3.2

The figure below shows `pre` and `post` numbers for the vertices in parentheses. Different edges are marked as indicated.




(a)

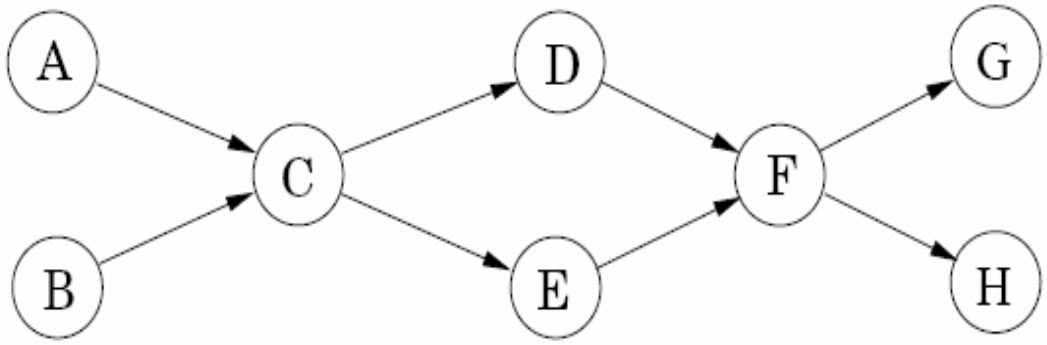



(b)

| `pre/post` *ordering for* $(u, v)$ | *Edge type* |
|---|---|
| $[_u \; [_v \; ]_v \; ]_u$ | Tree/forward |
| $[_v \; [_u \; ]_u \; ]_v$ | Back |
| $[_v \; ]_v \; [_u \; ]_u$ | Cross |

# Exercise 3.3

Run the DFS-based topological ordering algorithm on the following graph. Whenever you have a choice of vertices to explore, always pick the one that is alphabetically first.




(a) Indicate the `pre` and `post` numbers of the nodes.

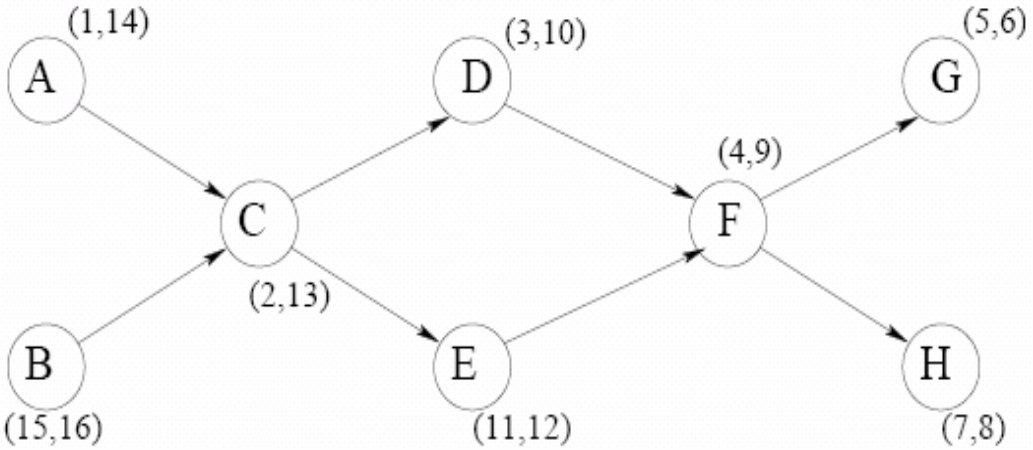
(b) What are the sources and sinks of the graph?

(c) What topological ordering is found by the algorithm?

(d) How many topological orderings does this graph have?

# Exercise 3.3

(a) The figure below shows the `pre` and `post` times in parentheses.




(b) The vertices $A, B$ are sources and $G, H$ are sinks.

(c) Since the algorithm outputs vertices in decreasing order of `post` numbers, the ordering given is $B, A, C, E, D, F, H, G$.

(d) Any ordering of the graph must be of the form $\{A, B\}, C, \{D, E\}, F, \{G, H\}$, where $\{A, B\}$ indicates $A$ and $B$ may be in any order within these two places. Hence the total number of orderings is $2^3 = 8$.

# Exercise 3.8

*Pouring water.* We have three containers whose sizes are 10 pints, 7 pints, and 4 pints, respectively. The 7-pint and 4-pint containers start out full of water, but the 10-pint container is initially empty. We are allowed one type of operation: pouring the contents of one container into another, stopping only when the source container is empty or the destination container is full. We want to know if there is a sequence of pourings that leaves exactly 2 pints in the 7- or 4-pint container.

(a) Model this as a graph problem: give a precise definition of the graph involved and state the specific question about this graph that needs to be answered.

(b) What algorithm should be applied to solve the problem?

(c) Find the answer by applying the algorithm.

# Exercise 3.8

(a) Let $G = (V, E)$ be our (directed) graph. We will model the set of nodes as triples of numbers $(a_0, a_1, a_2)$ where the following relationships hold: Let $S_0 = 10$, $S_1 = 7$,$S_2 = 4$ be the sizes of the corresponding containers. $a_i$ will correspond at the actual contents of the $i^{th}$ container. It must hold $0 \leq a_i \leq S_i$ for $i = 0, 1, 2$ and at any given node $a_0 + a_1 + a_2 = 11$ (the total amount of water we started from). An edge between two nodes $(a_0, a_1, a_2)$ and $(b_0, b_1, b_2)$ exists if both the following are satisfied :

- the two nodes differ in exactly two coordinates (and the third one is the same in both).
- if $i, j$ are the coordinates they differ in, then either $a_i = 0$ or $a_j = 0$ or $a_i = S_i$ or $a_j = S_j$.

The question that needs to be answered is wether there exists a path between the nodes $(0, 7, 4)$ and $(*, 2, *)$ or $(*, *, 2)$ where $*$ stands for any (allowed) value of the corresponding coordinate.

(b) Given the above description, it is easy to see that a DFS algorithm on that graph should be applied, starting from node $(0, 7, 4)$ with an extra line of code that halts and answers 'YES' if one of the desired nodes is reached and 'NO' if all the connected component of the starting node is exhausted an no desired vertex is reached.

(c) It is easy to see that after a few steps of the algorithm (depth 6 on the dfs tree) the node (2,7,2) is reached, so we answer 'YES'.

# Exercise 3.9

For each node $u$ in an undirected graph, let `twodegree`$[u]$ be the sum of the degrees of $u$'s neighbors. Show how to compute the entire array of `twodegree`$[\cdot]$ values in linear time, given a graph in adjacency list format.

First, the degree of each node can be determined by counting the number of elements in its adjacency list. The array `twodegree` can then be computed by initializing `twodegree` to 0 for every vertex, and then modifying `explore` as follows to add the degree of each neighbor of a vertex $u$ to `twodegree`$[u]$.

```
explore(G, u) {
    visited(u) = true
    previsit(u)
    for each edge (u, v) ∈ E:
        twodegree[u] = twodegree[u] + degree[v]
        if not visited(v):  explore(v)
    postvisit(u)
}
```

# Exercise 3.19

As in the previous problem, you are given a binary tree $T = (V, E)$ with designated root node. In addition, there is an array $x[\cdot]$ with a value for each node in $V$. Define a new array $z[\cdot]$ as follows: for each $u \in V$,

$$z[u] = \text{the maximum of the } x\text{-values associated with } u\text{'s descendants.}$$

Give a linear-time algorithm which calculates the entire $z$-array.

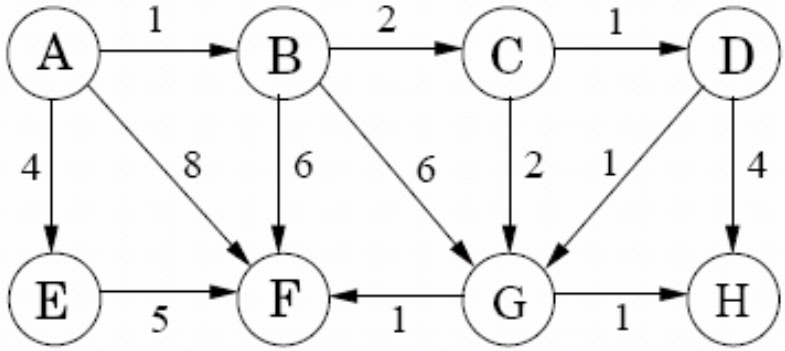
# Exercise 3.19

We modify the explore procedure so that explore called on a node returns the maximum x value in the corresponding subtree. The parent stores this as its z value, and returns the maximum of this and its own x value.

```
explore(G, u) {
      visited(u) = true
      z(u) = −∞
      temp = 0
      for each edge (u, v) ∈ E:
          If not visited(v): {
              temp = explore(G, v)
              if temp > z(u): z(u) = temp
          }
      postvisit(u)
      return max{z(u), x(u)}
}
```

# Exercise 4.1

Suppose Dijkstra's algorithm is run on the following graph, starting at node *A*.

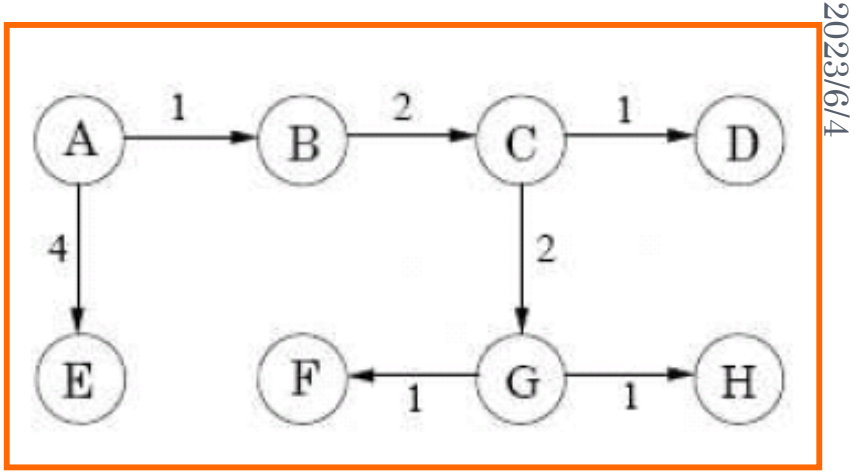


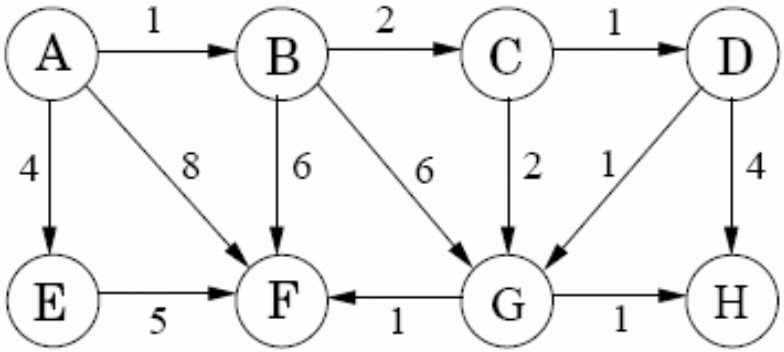

(a) Draw a table showing the intermediate distance values of all the nodes at each iteration of the algorithm.

(b) Show the final shortest-path tree.

# Exercise 4.1

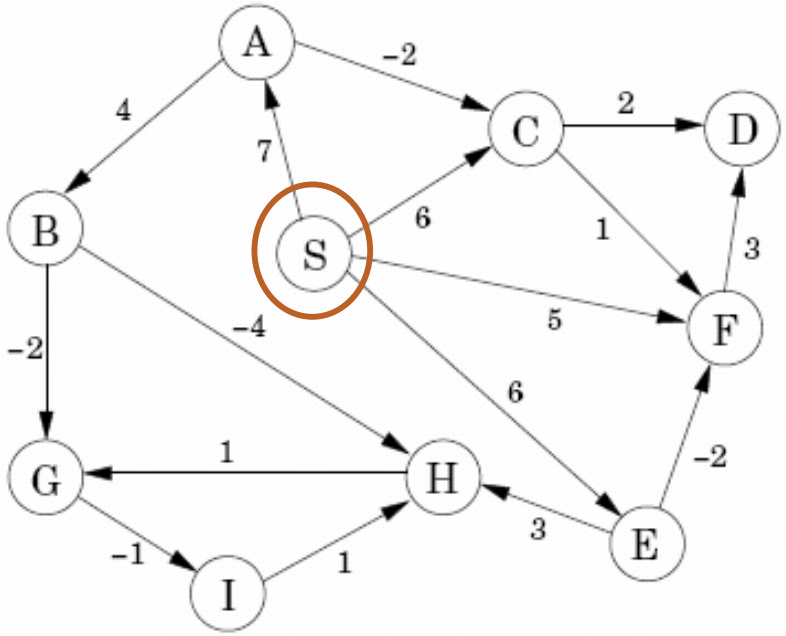
| Node | Iteration 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B | ∞ | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| C | ∞ | ∞ | 3 | 3 | 3 | 3 | 3 | 3 |
| D | ∞ | ∞ | ∞ | 4 | 4 | 4 | 4 | 4 |
| E | ∞ | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| F | ∞ | 8 | 7 | 7 | 7 | 7 | 6 | 6 |
| G | ∞ | ∞ | 7 | 5 | 5 | 5 | 5 | 5 |
| H | ∞ | ∞ | ∞ | ∞ | 8 | 8 | 6 | 6 |



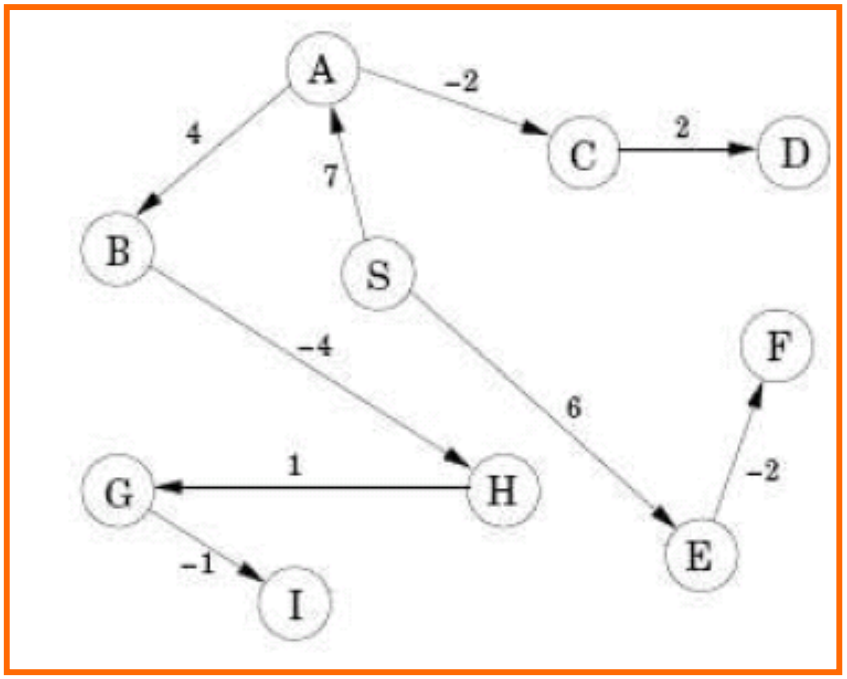
# Exercise 4.2

Just like the previous problem, but this time with the Bellman-Ford algorithm.



# Exercise 4.2

| Node | Iteration 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| S | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | ∞ | 7 | 7 | 7 | 7 | 7 | 7 |
| B | ∞ | ∞ | 11 | 11 | 11 | 11 | 11 |
| C | ∞ | 6 | 5 | 5 | 5 | 5 | 5 |
| D | ∞ | ∞ | 8 | 7 | 7 | 7 | 7 |
| E | ∞ | 6 | 6 | 6 | 6 | 6 | 6 |
| F | ∞ | 5 | 4 | 4 | 4 | 4 | 4 |
| G | ∞ | ∞ | ∞ | 9 | 8 | 8 | 8 |
| H | ∞ | ∞ | 9 | 7 | 7 | 7 | 7 |
| I | ∞ | ∞ | ∞ | ∞ | 8 | 7 | 7 |

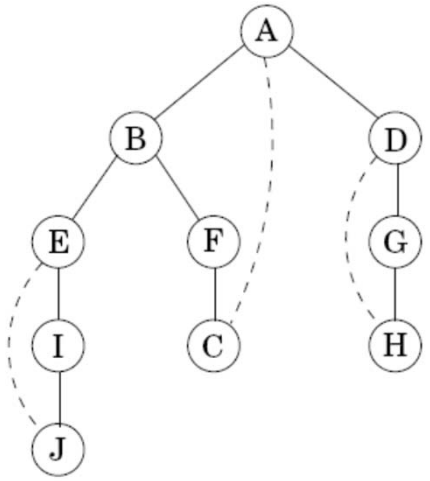


# Exercise 4.4

Here's a proposal for how to find the length of the shortest cycle in an undirected graph with unit edge lengths.

> When a back edge, say $(v, w)$, is encountered during a depth-first search, it forms a cycle with the tree edges from $w$ to $v$. The length of the cycle is level$[v]$ − level$[w]$ + 1, where the level of a vertex is its distance in the DFS tree from the root vertex. This suggests the following algorithm:
>
> - Do a depth-first search, keeping track of the level of each vertex.
> - Each time a back edge is encountered, compute the cycle length and save it if it is smaller than the shortest one previously seen.

Show that this strategy does not always work by providing a counterexample as well as a brief (one or two sentence) explanation.



# Exercise 4.4

The graph in Figure 3 is a counterexample: vertices are labelled with their level in the DFS tree, back edges are dashed. The shortest cycle consists of vertices $1-4-5$, but the cycle found by the algorithm is $1-2-3-4$. In general, the strategy will fail if the shortest cycle contains more than one back edge.

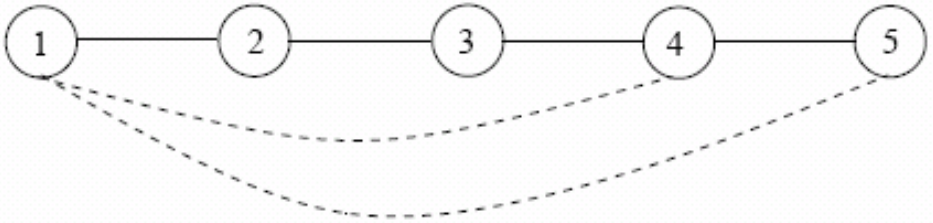



Figure 3: Counterexample for 4.4.

# Exercise 4.5

Often there are multiple shortest paths between two nodes of a graph. Give a linear-time algorithm for the following task.

*Input:* Undirected graph $G = (V, E)$ with unit edge lengths; nodes $u, v \in V$.

*Output:* The number of distinct shortest paths from $u$ to $v$.

# Exercise 4.5

We perform a BFS on the graph starting from $u$, and create a variable num_paths$(x)$ for the number of paths from $u$ to $x$, for all vertices $x$. If $x_1, x_2, \ldots x_k$ are vertices at depth $l$ in the BFS tree and $x$ is a vertex at depth $l+1$ such that $(x_1, x), \ldots, (x_k, x) \in E$ then we want to set num_paths$(x)$ = num_paths$(x_1)$+...+num_paths$(x_k)$. The easiest way to do this is to start with num_paths$(x) = 0$ for all vertices $x \neq u$ and num_paths$(u) = 1$. We then update num_paths$(y)$ = num_paths$(y)$ + num_paths$(x)$, for each edge $(x, y)$ that goes down one level in the tree. Since, we only modify BFS to do one extra operation per edge, this takes linear time. The pseudocode is as follows

```
function count_paths(G, u, v)
    for all x ∈ V:
        dist(x) = ∞
        num_paths(x) = 0

    dist(u) = 0
    num_paths(u) = 1
    Q = [u]
    while Q is not empty:
        x = eject(Q)
        for all edges (x, y) ∈ E
            if dist(y) = dist(x) + 1:
                num_paths(y) = num_paths(y) + num_paths(x)
            if dist(y) = ∞:
                inject(Q, y)
                dist(y) = dist(x) + 1
                num_paths(y) = num_paths(x)
```

# Exercise 4.15

Shortest paths are not always unique: sometimes there are two or more different paths with the minimum possible length. Show how to solve the following problem in $O((|V|+|E|)\log|V|)$ time.

*Input:* An undirected graph $G = (V, E)$; edge lengths $l_e > 0$; starting vertex $s \in V$.

*Output:* A Boolean array `usp`$[\cdot]$: for each node $u$, the entry `usp`$[u]$ should be `true` if and only if there is a *unique* shortest path from $s$ to $u$. (Note: `usp`$[s]$ = `true`.)

# Exercise 4.15

This can be done by slightly modifying Dijkstra's algorithm in Figure 4.8. The array `usp[·]` is initialized to `true` in the initialization loop. The main loop is modified as follows:

```
while H is not empty:
    u = deletemin(H)
    for all edges (u, v) ∈ E:

        if dist(v) > dist(u) + l(u, v):
            dist(v) = dist(u) + l(u, v)
            usp(v) = usp(u)
            decreasekey(H, v)
        if dist(v) = dist(u) + l(u, v):
            usp(v) = false
```

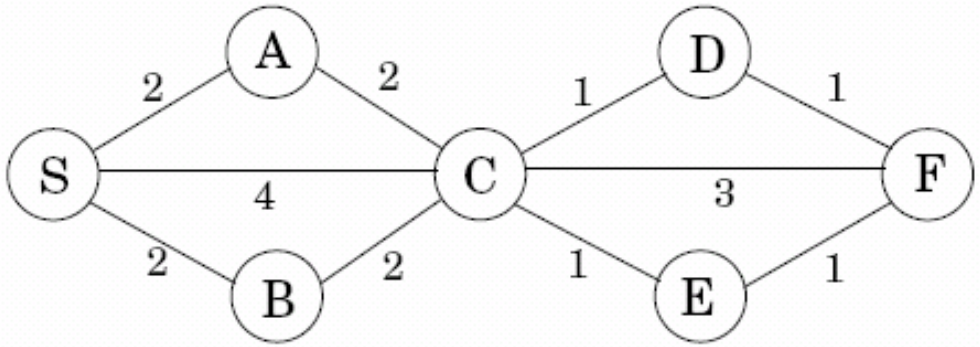
This will run in the required time when the heap is implemented as a binary heap.

# Exercise 4.18

In cases where there are several different shortest paths between two nodes (and edges have varying lengths), the most convenient of these paths is often *the one with fewest edges*. For instance, if nodes represent cities and edge lengths represent costs of flying between cities, there might be many ways to get from city $s$ to city $t$ which all have the same cost. The most convenient of these alternatives is the one which involves the fewest stopovers. Accordingly, for a specific starting node $s$, define

$$\texttt{best}[u] = \text{minimum number of edges in a shortest path from } s \text{ to } u.$$

In the example below, the `best` values for nodes $S, A, B, C, D, E, F$ are $0, 1, 1, 1, 2, 2, 3$, respectively.




Give an efficient algorithm for the following problem.

*Input:* Graph $G = (V, E)$; positive edge lengths $l_e$; starting node $s \in V$.
*Output:* The values of `best`$[u]$ should be set for *all* nodes $u \in V$.

# Exercise 4.18

This is another simple variation of Dijkstra's algorithm of Figure 4.8. In the initialization loop, `best`$(s)$ is set to 0 and all other entries of `best` are set to $\infty$. The main loop is modified as follows:

```
while H is not empty:
    u = deletemin(H)
    for all edges (u, v) ∈ E:
        if dist(v) > dist(u) + l(u, v):
            dist(v) = dist(u) + l(u, v)
            best(v) = best(u) + 1
            decreasekey(H, v)
        if dist(v) = dist(u) + l(u, v):
            if best(v) > best(u) + 1:
                best(v) = best(u) + 1
```

This has the same asymptotic running time as the original Dijkstra's algorithm, as the additional operations in the loop take constant time.

# Exercise 5.3

Design a linear-time algorithm for the following task.

*Input:* A connected, undirected graph $G$.
*Question:* Is there an edge you can remove from $G$ while still leaving $G$ connected?

Can you reduce the running time of your algorithm to $O(|V|)$?

Since the graph is given to be connected, it will have an edge whose removal still leaves it connected, if and only if it is not a tree i.e. has more than $|V| - 1$ edges. We perform a DFS on the graph until we see $|V|$ edges. If we can find $|V|$ edges then the answer is "yes" else it is "no". In either case, the time taken is $O(|V|)$.

# Exercise 5.5

Consider an undirected graph $G = (V, E)$ with nonnegative edge weights $w_e \geq 0$. Suppose that you have computed a minimum spanning tree of $G$, and that you have also computed shortest paths to all nodes from a particular node $s \in V$.

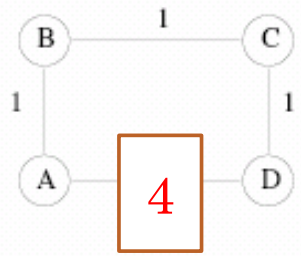
Now suppose each edge weight is increased by 1: the new weights are $w'_e = w_e + 1$.

(a) Does the minimum spanning tree change? Give an example where it changes or prove it cannot change.

(b) Do the shortest paths change? Give an example where they change or prove they cannot change.

(a) The minimum spanning tree does not change. Since, each spanning tree contains exactly $n - 1$ edges, the cost of each tree is increased $n - 1$ and hence the minimum is unchanged.

(b) The shortest paths may change. In the following graph, the shortest path from $A$ to $D$ changes from $AB - BC - CD$ to $AD$ if each edge weight is increased by 1.



# Exercise 5.7

Show how to find the *maximum* spanning tree of a graph, that is, the spanning tree of largest total weight.

Multiply the weights of all the edges by -1. Since both Kruskal's and Prim's algorithms work for positive as well as negative weights, we can find the minimum spanning tree of the new graph. This is the same as the maximum spanning tree of the original graph.

# Exercise 5.14

Suppose the symbols $a, b, c, d, e$ occur with frequencies $1/2, 1/4, 1/8, 1/16, 1/16$, respectively.

(a) What is the Huffman encoding of the alphabet?

(b) If this encoding is applied to a file consisting of 1,000,000 characters with the given frequencies, what is the length of the encoded file in bits?

(a) $a \to 0$, $b \to 10$ ,$c \to 110$, $d \to 1110$, $e \to 1111$.

(b) length $= \frac{1000000}{2} \cdot 1 + \frac{1000000}{4} \cdot 2 + \frac{1000000}{8} \cdot 3 + 2 \cdot \frac{1000000}{16} \cdot 4 = 1875000$

# Exercise 5.15

We use Huffman's algorithm to obtain an encoding of alphabet $\{a, b, c\}$ with frequencies $f_a, f_b, f_c$. In each of the following cases, either give an example of frequencies $(f_a, f_b, f_c)$ that would yield the specified code, or explain why the code cannot possibly be obtained (no matter what the frequencies are).

(a) Code: $\{0, 10, 11\}$

(b) Code: $\{0, 1, 00\}$

(c) Code: $\{10, 01, 00\}$

(a) $(f_a, f_b, f_c) = (2/3, 1/6, 1/6)$ gives the code $\{0, 10, 11\}$.

(b) This encoding is not possible, since the code for $a$ (0), is a prefix of the code for $c$ (00).

(c) This code is not optimal since $\{1, 01, 00\}$ gives a shorter encoding. Also, it does not correspond to a *full* binary tree and hence cannot be given by the Huffman algorithm.

# Exercise 5.17

Under a Huffman encoding of $n$ symbols with frequencies $f_1, f_2, \ldots, f_n$, what is the longest a codeword could possibly be? Give an example set of frequencies that would produce this case.

The longest codeword can be of length $n-1$. An encoding of $n$ symbols with $n-2$ of them having probabilities $1/2, 1/4, \ldots, 1/2^{n-2}$ and two of them having probability $1/2^{n-1}$ achieves this value.

# EXERCISE 5.32

A server has $n$ customers waiting to be served. The service time required by each customer is known in advance: it is $t_i$ minutes for customer $i$. So if, for example, the customers are served in order of increasing $i$, then the $i$th customer has to wait $\sum_{j=1}^{i-1} t_j$ minutes.

We wish to minimize the total waiting time

$$T = \sum_{i=1}^{n} (\text{time spent waiting by customer } i).$$

Give an efficient algorithm for computing the optimal order in which to process the customers.

# Exercise 5.32

We simply proceed by a greedy strategy, by sorting the customers in the increasing order of service times and servicing them in this order. The running time is $O(n \log n)$. To prove the correctness, for any ordering of the customers, let $s(j)$ denote the $j$th customer in the ordering. Then

$$T = \sum_{i=1}^{n} \sum_{j=1}^{i-1} t_{s(j)} = \sum_{i=1}^{n} (n-i) t_{s(i)}$$

For any ordering, if $t_{s(i)} > t_{s(j)}$ for $i < j$, then swapping the positions of the two customers gives a better ordering. Since, we can generate all possible orderings by swaps, an ordering which has the property that $t_{s(1)} \leq \ldots \leq t_{s(n)}$ must be the global optimum. However, this is exactly the ordering we output.

# Exercise 6.4

You are given a string of $n$ characters $s[1 \ldots n]$, which you believe to be a corrupted text document in which all punctuation has vanished (so that it looks something like "itwasthebestoftimes..."). You wish to reconstruct the document using a dictionary, which is available in the form of a Boolean function `dict`$(\cdot)$: for any string $w$,

$$\texttt{dict}(w) = \begin{cases} \texttt{true} & \text{if } w \text{ is a valid word} \\ \texttt{false} & \text{otherwise .} \end{cases}$$

(a) Give a dynamic programming algorithm that determines whether the string $s[\cdot]$ can be reconstituted as a sequence of valid words. The running time should be at most $O(n^2)$, assuming calls to `dict` take unit time.

(b) In the event that the string is valid, make your algorithm output the corresponding sequence of words.

# Exercise 6.4

a) *Subproblems:* Define an array of subproblems $S(i)$ for $0 \leq i \leq n$ where $S(i)$ is 1 if $s[1 \cdots i]$ is a sequence of valid words and is 0 otherwise.

*Algorithm and Recursion:* It is sufficient to initialize $S(0) = 1$ and update the values $S(i)$ in ascending order according to the recursion

$$S(i) = \max_{0 \leq j < i} \{S(j) : \texttt{dict}(s[j+1 \cdots i]) = \texttt{true}\}$$

Then, the string $s$ can be reconstructed as a sequence of valid words if and only if $S(n) = 1$.

*Correctness and Running Time:* Consider $s[1 \cdots i]$. If it is a sequence of valid words, there is a last word $s[j \cdots i]$, which is valid, and such that $S(j) = 1$ and the update will cause $S(i)$ to be set to 1. Otherwise, for any valid word $S[j \cdots i]$, $S(j)$ must be 0 and $S(i)$ will also be set to 0. This runs in time $O(n^2)$ as there are $n$ subproblems, each of which takes time $O(n)$ to be updated with the solution obtained from smaller subproblems.

b) Every time a $S(i)$ is updated to 1 keep track of the previous item $S(j)$ which caused the update of $S(i)$ because $s[j+1 \cdots i]$ was a valid word. At termination, if $S(n) = 1$, trace back the series of updates to recover the partition in words. This only adds a constant amount of work at each subproblem and a $O(n)$ time pass over the array at the end. Hence, the running time remains $O(n^2)$.

# Exercise 6.7

A subsequence is *palindromic* if it is the same whether read left to right or right to left. For instance, the sequence

$$A, C, G, T, G, T, C, A, A, A, A, T, C, G$$

has many palindromic subsequences, including $A, C, G, C, A$ and $A, A, A, A$ (on the other hand, the subsequence $A, C, T$ is *not* palindromic). Devise an algorithm that takes a sequence $x[1 \ldots n]$ and returns the (length of the) longest palindromic subsequence. Its running time should be $O(n^2)$.

*Subproblems:* Define variables $L(i,j)$ for all $1 \leq i \leq j \leq n$ so that, in the course of the algorithm, each $L(i,j)$ is assigned the length of the longest palindromic subsequence of string $x[i, \cdots, j]$.

*Algorithm and Recursion:* The recursion will then be:

$$L(i,j) = \max\{L(i+1,j), L(i,j-1), L(i+1,j-1) + \text{equal}(x_i, x_j)\}$$

where $\text{equal}(a,b)$ is 1 if $a$ and $b$ are the same character and is 0 otherwise, The initialization is the following:

$$\begin{aligned} \forall i, 1 \leq i \leq n \ , \quad & L(i,i) = 0 \\ \forall i, 1 \leq i \leq n-1 \ , \quad & L(i,i+1) = \text{equal}(x_i, x_i+1) \end{aligned}$$

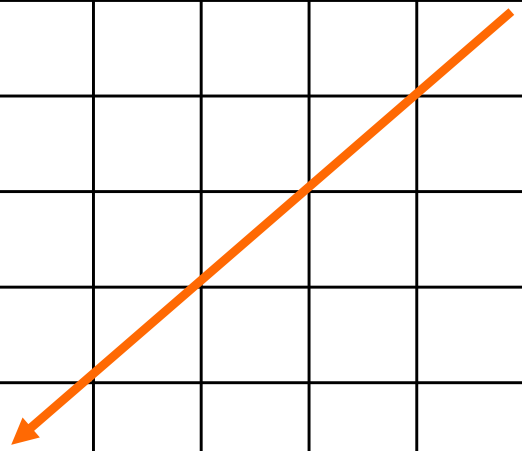
For s=2 to n-1
  for i=1 to n-s
    j=i+s

*Correctness and Running Time:* Consider the longest palindromic subsequence $s$ of $x[i, \cdots, j]$ and focus on the elements $x_i$ and $x_j$. There are then three possible cases:

- If both $x_i$ and $x_j$ are in $s$ then they must be equal and $L(i,j) = L(i+1,j-1) + \text{equal}(x_i, x_j)$
- If $x_i$ is not a part of $s$, then $L(i,j) = L(i+1,j)$.
- If $x_j$ is not a part of $s$, then $L(i,j) = L(i,j-1)$.

Hence, the recursion handles all possible cases correctly. The running time of this algorithm is $O(n^2)$, as there are $O(n^2)$ subproblems and each takes $O(1)$ time to evaluate according to our recursion.

| I/J | A | C | G | T | G | T | C | A | A | A | A | T | C | G |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 0 | | | | | | | | | | | | | |
| C | 0 | 0 | | | | | | | | | | | | |
| G | 0 | 0 | 0 | | | | | | | | | | | |
| T | 0 | 0 | 0 | 0 | | | | | | | | | | |
| G | 1 | 1 | 1 | 0 | 0 | | | | | | | | | |
| T | | 1 | 1 | 1 | 0 | 0 | | | | | | | | |
| C | | | 1 | 1 | 0 | 0 | 0 | | | | | | | |
| A | | | | 1 | 0 | 0 | 0 | 0 | | | | | | |
| A | | | | | 1 | 1 | 1 | 1 | 0 | | | | | |
| A | | | | | | 1 | 1 | 1 | 1 | 0 | | | | |
| A | | | | | | | 2 | 2 | 1 | 1 | 0 | | | |
| T | | | | | | | | 2 | 1 | 1 | 0 | 0 | | |
| C | | | | | | | | | 1 | 1 | 0 | 0 | 0 | |
| G | | | | | | | | | | 1 | 0 | 0 | 0 | 0 |



| I/J | A | C | G | T | G | T | C | A | A | A | A | T | C | G |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 0 | | | | | | | | | | | | | |
| C | 0 | 0 | | | | | | | | | | | | |
| G | 0 | 0 | 0 | | | | | | | | | | | |
| T | 0 | 0 | 0 | 0 | | | | | | | | | | |
| G | 1 | 1 | 1 | 0 | 0 | | | | | | | | | |
| T | 1 | 1 | 1 | 1 | 0 | 0 | | | | | | | | |
| C | 2 | 2 | 1 | 1 | 0 | 0 | 0 | | | | | | | |
| A | 3 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | | | | | | |
| A | 3 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | | | | | |
| A | 3 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | | | | |
| A | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 0 | | | |
| T | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 1 | 1 | 0 | 0 | | |
| C | 4 | 4 | 3 | 3 | 3 | 3 | 3 | 2 | 1 | 1 | 0 | 0 | 0 | |
| G | 4 | 4 | 4 | 4 | 4 | 3 | 3 | 2 | 1 | 1 | 0 | 0 | 0 | 0 |

# Exercise 6.21

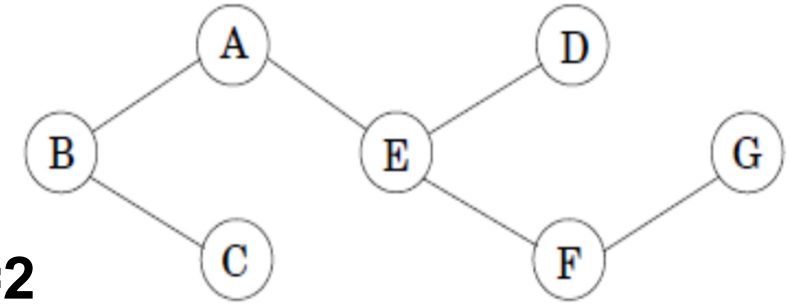


A *vertex cover* of a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ that includes at least one endpoint of every edge in $E$. Give a linear-time algorithm for the following task.

*Input:* An undirected tree $T = (V, E)$.
*Output:* The size of the smallest vertex cover of $T$.

For instance, in the following tree, possible vertex covers include $\{A, B, C, D, E, F, G\}$ and $\{A, C, D, F\}$ but not $\{C, E, F\}$. The smallest vertex cover has size 3: $\{B, E, G\}$.

**V(G)=0**
**V(D)=0**
**V(F)=min{1,1+0}=1**
**V(E)=min{2+0,1+1}=2**
**V(C)=0**
**V(A)=min{1+1,1+2}=2**
**V(B)=min{2+2,1+2}=3**




Best cover: {B, E, G} or {B, E, F}



$$V(i) = \min\{\#child + \sum V(grandchild), 1 + \sum V(child)\}$$

# Exercise 6.21



The subproblem $V(u)$ will be defined to be the size of the minimum vertex cover for the subtree rooted at node $u$. We have $V(u) = 0$ if $u$ is a leaf, as the subtree rooted at $u$ has no edges to cover. The crucial observation is that if a vertex cover does not use a node it has to use all its neighboring nodes. Hence, for any internal node $i$

$$V(i) = \min \left\{ \sum_{j:(i,j)\in E} \left( 1 + \sum_{k:(j,k)\in E} V(k) \right), 1 + \sum_{j:(i,j)\in E} V(j) \right\}$$

The algorithm can then solve all the subproblems in order of decreasing depth in the tree and output $V(n)$. The running time is linear in $n$ because while calculating $V(i)$ for all $i$ we look at most at $2 * |E| = O(n)$ edges in total.

$$V(i) = \min\{\#child + \sum V(grandchild), 1 + \sum V(child)\}$$

# Exercise 6.8

Given two strings $x = x_1x_2 \cdots x_n$ and $y = y_1y_2 \cdots y_m$, we wish to find the length of their *longest common substring*, that is, the largest $k$ for which there are indices $i$ and $j$ with $x_ix_{i+1} \cdots x_{i+k-1} = y_jy_{j+1} \cdots y_{j+k-1}$. Show how to do this in time $O(mn)$.

# Exercise 6.8

*Subproblems:* For $1 \leq i \leq n$ and $1 \leq j \leq m$, define subproblem $L(i,j)$ to be the length of the longest common substring of $x$ and $y$ *terminating* at $x_i$ and $y_j$. The recursion is:

$$L(i,j) = \begin{cases} L(i-1,j-1)+1 & \text{if equal}(x_i, y_j) = 1 \\ 0 & \text{otherwise} \end{cases}$$

The initialization is, for all $1 \leq i \leq n$ and $1 \leq j \leq m$:

$$L(0,0) = 0$$
$$L(i,0) = 0$$
$$L(0,j) = 0$$

The output of the algorithm is the maximum of $L(i,j)$ over all $1 \leq i \leq n$ and $1 \leq j \leq m$.

*Correctness and Running Time:* The initialization is clearly correct. Hence, it suffices to prove the correctness of the recursion. The longest common substring terminating at $x_i$ and $y_j$ must include $x_i$ and $y_j$: hence, it will be 0 if these characters are different and $L(i-1,j-1)+1$ if they are equal. The running time is $O(mn)$ as we have $mn$ subproblems and each takes constant time to evaluate through the recursion.

| X/Y | | A | B | C | B | D | A | B |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| D | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 |
| C | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| B | 0 | 0 | 2 | 0 | 1 | 0 | 0 | 2 |
| A | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |

# EXERCISE 8.5

Give a simple reduction from 3D MATCHING to SAT, and another from RUDRATA CYCLE to SAT. (*Hint:* In the latter case you may use variables $x_{ij}$ whose intuitive meaning is "vertex $i$ is the $j$th vertex of the Hamilton cycle"; you then need to write clauses that express the constraints of the problem.)

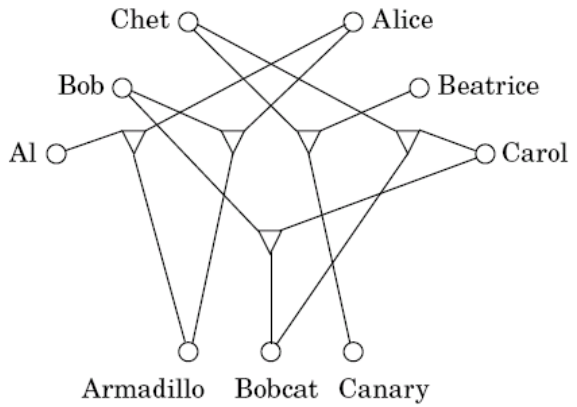
# Exercise 8.5

**3D-MATCHING to SAT**

We have a variable $x_{bgp}$ for each given triple $(b, g, p)$. We interpret $x_{bgp} = \texttt{true}$ as choosing the triple $(b, g, p)$. Suppose for boy $b$, $(b, g_1, p_1), \ldots, (b, g_k, p_k)$ are triples involving him. Then we add the clause $(x_{bg_1p_1} \vee \ldots \vee x_{bg_kp_k})$ so that at least one of the triples is chosen. Similarly for the triples involving each girl or pet. Also, for each pair of triples involving a common boy, girl or pet, say $(b_1, g, p_1)$ and $(b_2, g, p_2)$, we add a clause of the form $(\bar{x}_{b_1gp_1} \vee \bar{x}_{b_1gp_1})$ so that at most one of the triples is chosen.

The total number of triples, and hence the number of variables, is at most $n^3$. The first type of clauses involve at most $n^2$ variables and we add $3n$ such clauses, one for each boy, girl or pet. The second type of clauses involve triples sharing one common element. There are $3n$ ways to choose the common element and at most $n^4$ to choose the rest, giving at most $3n^5$ clauses. Hence, the size of the new formula is polynomial in the size of the input.

If there is a matching, then it must involve $n$ triples $(b_1, g_1, p_1), \ldots, (b_n, g_n, p_n)$. Setting the variables $x_{b_1g_1p_1}, \ldots, x_{b_ng_np_n}$ to `true` and the rest to `false` gives a satisfying assignment to the above formula. Similarly, choosing only the triples corresponding to the `true` variables in any satisfying assignment must correspond to a matching for the reasons mentioned above. Hence, the formula is satisfiable if and only if the given instance has a 3D matching.



# Exercise 8.5

**RUDRATA CYCLE to SAT**

We introduce variables $x_{ij}$ for $1 \leq i, j \leq n$ meaning that the $i$th vertex is at the $j$th position in the Rudrata cycle. Each vertex must appear at some position in the cycle. Thus, for every vertex $i$, we add the clause $x_{i1} \vee x_{i2} \vee \ldots \vee x_{in}$. This adds $n$ clauses with $n$ variables each.

Also, if the $i$th vertex appears at the $j$th position, then the vertex at $(j+1)$th position must be a neighbor of $i$. In other words, if $u, v$ are *not* neighbors, then either $u$ appears at the $j$th position, or $v$ appears at the $(j+1)$th position, but not both. Thus for every $(u, v) \notin E$ and for all $1 \leq j \leq n$, add the clause $(\bar{x}_{uj} \vee (\bar{x}_{v(j+1)})$. This adds at most $O(n^2) \times n = O(n^3)$ clauses with 2 variables each.

Using the "meanings" of the clauses given above, it is easy to see that every satisfying assignment gives a Rudrata cycle and vice-versa.