Lecture Notes: CS 5381 Analysis of Algorithms

Based on the book Introduction to Algorithms (third edition) by Cormen, Leiserson, Rivest, and Stein

Table of Contents

1. Growth of Functions

1.1 Asymptotic notations

The order of growth of the **running time** of an algorithm gives a simple characterization of the algorithm's efficiency and also allows us to compare the relative performance of algorithms.

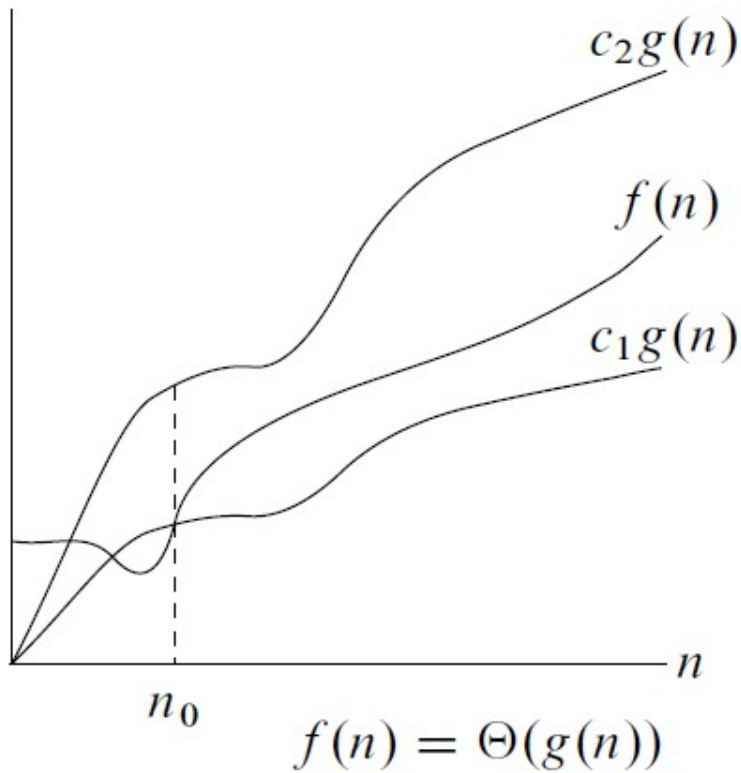
When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the **asymptotic** efficiency of algorithms. That is, we are concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases.

The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of **natural numbers**.

Θ-notation

For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions:

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



A function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be 'sandwiched' between $c_1 g(n)$ and $c_1 g(n)$, for sufficiently large n.

Example (Show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$)

We must determine positive constants c_1, c_2, and n_0 such that

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2, \quad \forall n \geq n_0.$$

Dividing by n^2 yields

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2.$$

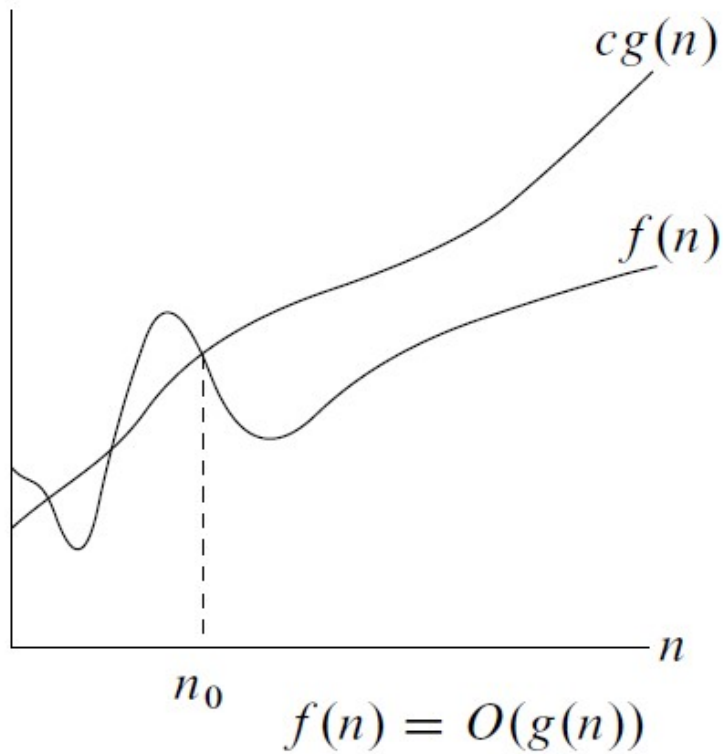
Right-hand inequality holds for any value of $n \geq 1$ by choosing any constant $c_2 \geq 1/2$. Left-hand inequality holds for $n \geq 7$ by choosing $c_1 = 1/14$. Therefore, by choosing $c_1 = 1/14$, $c_2 = 1/2$, and $n_0 = 7$, the claimed result is shown. Certainly, other choices for the constants exist, but the important thing is that some choice exists.

O-notation

The Θ-notation asymptotically bounds a function from above and below. When we have only an **asymptotic upper bound**, we use *O*-notation.

For a given function $g(n)$, we denote by $O(g(n))$ the set of functions:

$$O(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$



We use O-notation to give an upper bound on a function, to within a constant factor.

Note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$ since Θ-notation is a stronger notion than O-notation. Written set-theoretically, we have $\Theta(g(n)) \subseteq O(g(n))$.

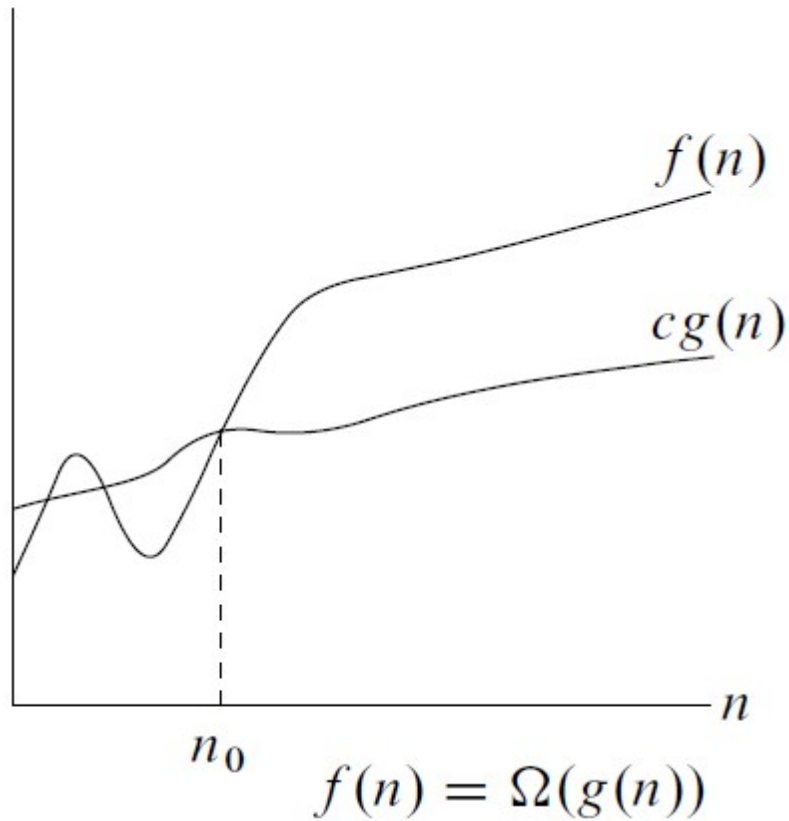
Since O-notation describes an upper bound, when we use it to bound the worst-case running time of an algorithm, we have a bound on the running time of the algorithm on every input.

Ω-notation

Just as O-notation provides an asymptotic upper bound on a function, Ω-notation provides an **asymptotic lower bound**.

For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions:

$$\Omega(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$



From the definitions of the asymptotic notations we have seen thus far, it is easy to show the following important theorem.

Theorem

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

o-notation

The asymptotic upper bound provided by O-notation may or may not be asymptotically tight.

We use o-notation to denote an upper bound that is not asymptotically tight:

$$o(g(n)) = \{f(n) : \text{ for any positive constant } c > 0, \text{ there exists a positive constant } n_0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$$

The definitions of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $0 \leq f(n) \leq cg(n)$ holds for **some** constant $c > 0$, but in $f(n) = o(g(n))$, the bound $0 \leq f(n) < cg(n)$ holds for **all** constants $c > 0$.

Intuitively, in o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity. That is,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

This may be considered as another definition of the o-notation.

ω-notation

By analogy, ω-notation is to Ω-notation as o-notation is to O-notation.

We use ω-notation to denote an lower bound that is not asymptotically tight:

$$\begin{aligned}\omega(g(n)) = \{f(n) :\ &\text{for any positive constant } c > 0, \text{ there exists a positive} \\ &\text{constant } n_0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}\end{aligned}$$

The relation $f(n) = \omega(g(n))$ implies that

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

That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity.

Some basic properties

Transitivity:

$$f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \quad \text{imply} \quad f(n) = \Theta(h(n)),$$
$$f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \quad \text{imply} \quad f(n) = O(h(n)),$$
$$f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \quad \text{imply} \quad f(n) = \Omega(h(n)),$$
$$f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) \quad \text{imply} \quad f(n) = o(h(n)),$$
$$f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \quad \text{imply} \quad f(n) = \omega(h(n)).$$

Reflexivity:

$$f(n) = \Theta(f(n)),$$
$$f(n) = O(f(n)),$$
$$f(n) = \Omega(f(n)).$$

Analogies between the asymptotic comparison of two functions *f* and *g* and the comparison of two real numbers *a* and *b*

$$
\begin{aligned}
f(n) &= O(g(n)) \quad \text{is like} \quad a \leq b\ , \\
f(n) &= \Omega(g(n)) \quad \text{is like} \quad a \geq b\ , \\
f(n) &= \Theta(g(n)) \quad \text{is like} \quad a = b\ , \\
f(n) &= o(g(n)) \quad \text{is like} \quad a < b\ , \\
f(n) &= \omega(g(n)) \quad \text{is like} \quad a > b\ .
\end{aligned}
$$

Monotonicity

A function $f(n)$ is **monotonically increasing** if $m \leq n$ implies $f(m) \leq f(n)$. Similarly, it is **monotonically decreasing** if $m \leq n$ implies $f(m) \geq f(n)$. A function $f(n)$ is **strictly increasing** if $m < n$ implies $f(m) < f(n)$ and **strictly decreasing** if $m < n$ implies $f(m) > f(n)$.

Floors and ceilings

For any real number x, we denote the greatest integer less than or equal to x by $\lfloor x \rfloor$ (floor of x) and the least integer greater than or equal to x by $\lceil x \rceil$ (ceiling of x). For all real x, one has

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

Polynomials

Given a nonnegative integer d, a **polynomial in n of degree d** is a function $p(n)$ of the form

$$p(n) = \sum_{i=0}^{d} a_i n^i,$$

where the constants $a_0, a_1, \ldots, a_d$ are the **coefficients** of the polynomial and $a_d \neq 0$.

A polynomial is asymptotically positive if and only if $a_d > 0$. For an asymptotically positive polynomial $p(n)$ of degree d, we have $p(n) = \Theta(n^d)$.

We say that a function $f(n)$ is **polynomially bounded** if $f(n) = O(n^k)$ for some constant k.

Exponentials

For all n and $a \geq 1$, the exponential function a^n an is monotonically increasing in n.

For all real constants a and b such that $a > 1$,

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

Thus, any exponential function with a base strictly greater than 1 grows faster than any polynomial function.

We have for all real x,

$$\mathrm{e}^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!}.$$

When $x \to 0$, the approximation of e^x by $1 + x$ is quite good:

$$\mathrm{e}^x = 1 + x + \Theta(x^2).$$

Logarithms

Notations

$$\begin{aligned}
\lg n &= \log_2 n && \text{(binary logarithm)} ,\\
\ln n &= \log_e n && \text{(natural logarithm)} ,\\
\lg^k n &= (\lg n)^k && \text{(exponentiation)} ,\\
\lg \lg n &= \lg(\lg n) && \text{(composition)} .
\end{aligned}$$

If we hold $b > 1$ constant, then for $n > 0$, the function $\log_b n$ is strictly increasing.

We say that a function $f(n)$ is **polylogarithmically bounded** if $f(n) = O(\lg^k n)$ for some constant k.

For $a > 0, b > 0$, we have the limit

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

so that

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

Thus, any positive polynomial function grows faster than any polylogarithmic function.

Factorials

The notation $n!$ is defined for integers $n \geq 0$ as

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n > 0. \end{cases}$$

Thus, $n! = 1 \cdot 2 \cdot 3 \cdots n$.

Stirling's approximation

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

is helpful in showing

$$\lg(n!) = \Theta(n \lg n).$$

2. Divide-and-Conquer

2.1 Preliminaries

In **divide-and-conquer**, we solve a problem recursively, applying three steps at each level of the recursion:

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

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

When the subproblems are large enough to solve recursively, we call that the **recursive case**. Once the subproblems become small enough that we no longer recurse, we say that we have the **base case**.

A **recurrence** is an equation or inequality that describes a function in terms of its value on smaller inputs.

Recurrences go hand in hand with the divide-and-conquer paradigm, because they give us a natural way to characterize the running times of divide-and-conquer algorithms.

Suppose the running time $T(n)$ of an algorithm is given by the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1. \end{cases}$$

How can we find the running time of the above algorithm?

We will eventually be able to show that its asymptotic running time is

$$T(n) = \Theta(n \lg n).$$

In practice, we neglect certain technical details when we solve recurrences.

Boundary conditions are typically ignored. The reason is that although changing the value of $T(1)$ changes the exact solution to the recurrence, the solution does not change by more than a constant factor, and so the order of growth is unchanged.

We also often omit floors, ceilings, and boundary conditions. We forge ahead without these details and later determine whether or not they matter.

As an example, the recurrence in the last page is usually stated as

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

We will study three methods for solving recurrences - that is, for obtaining asymptotic Θ or O bounds on the solution:

In the **substitution method**, we guess a bound and then use mathematical induction to prove our guess correct.

The **recursion-tree method** converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.

The **master method** provides bounds for recurrences of the form

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

where $a \geq 1$, $b \geq 1$, and $f(n)$ is a given function.

The recurrence characterizes a divide-and-conquer algorithm that creates a subproblems, each of which is $1/b$ the size of the original problem, and in which the divide and combine steps together take $f(n)$ time.

The recurrences of the form in the master method arise frequently for many practical algorithms. We will use the master method to prove the running time on page 25.

However, not every algorithm can be studied by the master method. For example, a recursive algorithm might divide subproblems into the **unequal sizes**, such as a $2/3$ and $1/3$ splits. If the divide and combine steps take linear time, such an algorithm would give rise to the recurrence

$$T(n) = T(2n/3) + T(n/3) + \Theta(n),$$

which is different than the master equation.

2.2 Solving recurrences: substitution method

The substitution method for solving recurrences comprises two steps:

1. Guess the form of the solution.
2. Use mathematical induction to show that the solution is correct.

We substitute the guessed solution for the function when applying the inductive hypothesis to smaller values; hence the name substitution method.

We can use the substitution method to establish either upper or lower bounds on a recurrence.

Example (Determine an upper bound on the recurrence $T(n) = 2T(\lfloor n/2 \rfloor) + n$)

As motivated by the result on page 25, we guess that the solution is

$$T(n) = O(n \lg n)$$

For an appropriate choice of the constant $c > 0$, the substitution method requires us to prove that

$$T(n) \leq cn \lg n.$$

We start by assuming that this bound holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$, yielding

$$T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor.$$

Substituting into the recurrence yields

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n \ , \end{aligned}$$

where the last step holds as long as $c \geq 1$.

There is no general way to conjecture the correct solutions to recurrences. The recursion-tree method, to be discussed later, may help come up with a conjecture in some cases.

Some heuristics may help you become a good guesser. If a recurrence is similar to one you have seen before, then guessing a similar solution is reasonable.

As an example, consider the recurrence

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n,$$

where after realizing that when n grows large, the difference between $\lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor + 17$ is diminishing. Consequently, we make the guess that

$$T(n) = O(n \lg n),$$

which in fact is the correct answer.

2.3 Solving recurrences: recursion-tree method

Drawing out a recursion tree may serve as a straightforward way to devise a good guess.

In a recursion tree, each node represents the cost of a single subproblem. We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion.

When using a recursion tree to generate a good guess, you can often tolerate a small amount of 'sloppiness', since you will be verifying your guess later.

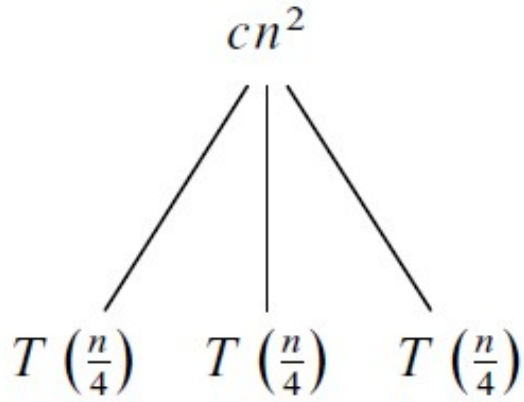
Example (Provide a conjecture on running time of the recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ using the recursion-tree method)

Since we know that floors and ceilings usually do not matter when solving recurrences. We start by focusing on finding an upper bound by considering the recursion-tree for the recurrence

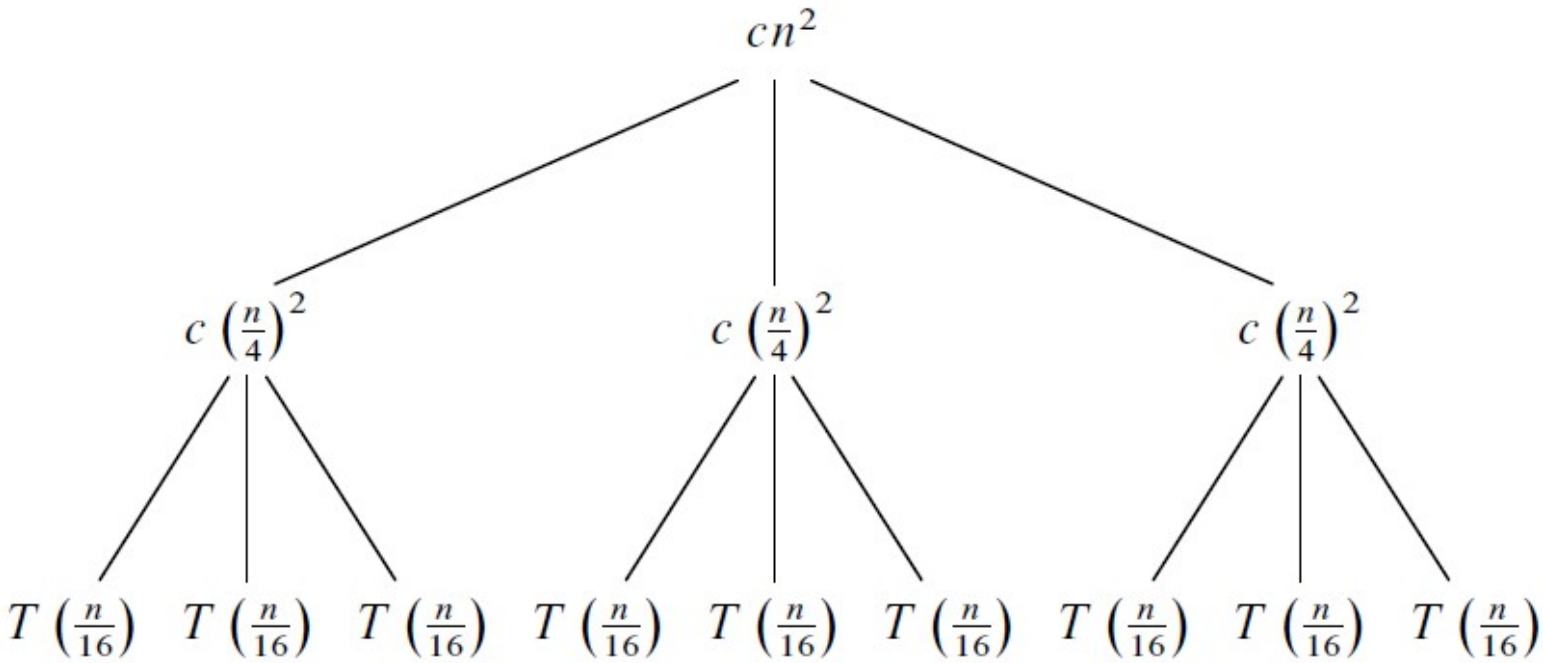
$$T(n) = 3T(n/4) + cn^2$$

with $c > 0$.

We also assume that n is a power of 4 so that all subproblem sizes are integers.



$T(n)$ is expanded into an equivalent tree representing the recurrence. The cn^2 term at the root represents the cost at the top level of recursion, and the three subtrees of the root represent the costs incurred by the subproblems of size $n/4$.



This process carried one step further by expanding each node with cost $T(n/4)$. The cost for each of the three children of the root is $c(n/4)^2$.

We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence.

Because subproblem sizes decrease by a factor of 4 each time we go down one level, we eventually must reach a boundary condition.

The subproblem size for a node at depth i is $n/4^i$. Thus, the subproblem size hits $n = 1$ when $n/4^i = 1$ or, equivalently, when $i = \log_4 n$.

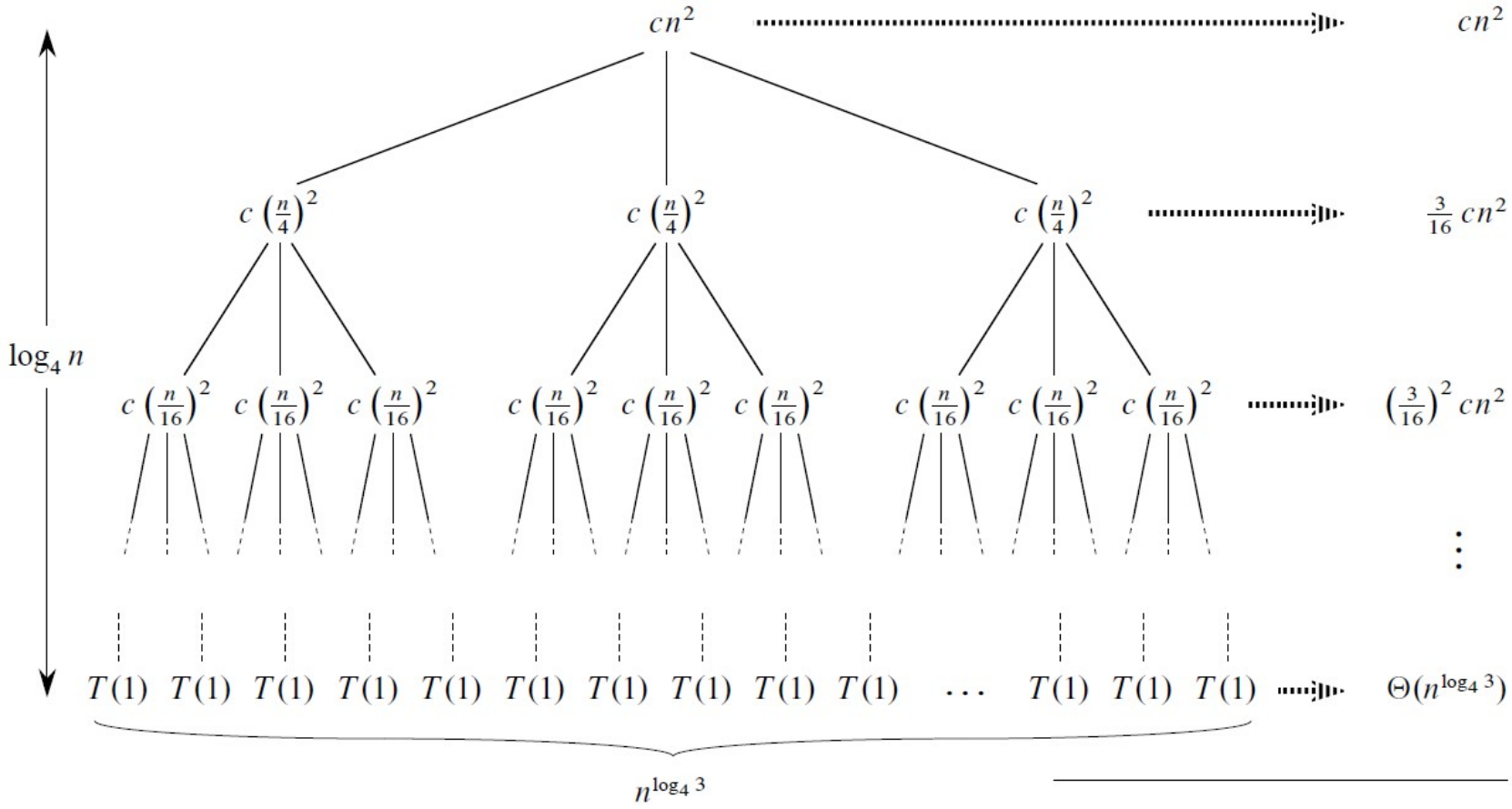
Next we determine the cost at each level of the tree.

Each level has three times more nodes than the level above, and so the number of nodes at depth i is 3^i.

Because subproblem sizes reduce by a factor of 4 for each level we go down from the root, each node at depth i, for $i = 0, 1, 2, \ldots, \log_4 n - 1$, has a cost of $c(n/4^i)^2$.

Multiplying, we see that the total cost over all nodes at depth i is $3^i c(n/4^i)^2 = (3/16)^i cn^2$.

The bottom level, at depth $log_4 n$, has $3^{log_4 n} = n^{log_4 3}$ nodes, each contributing cost $T(1)$, for a total cost of $n^{log_4 3} T(1)$, which is $\Theta(n^{log_4 3})$.


cn^2
cn^2
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$\frac{3}{16}cn^2$
$\log_4 n$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $\cdots$ $T(1)$ $T(1)$ $T(1)$
$\Theta(n^{\log_4 3})$
$n^{\log_4 3}$

Total: $O(n^2)$

Now we add up the costs over all levels to determine the cost for the entire tree:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\ &= O(n^2) \,. \end{aligned}$$

Thus, we have derived a guess of $T(n) = O(n^2)$ for our original recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. We see that the cost of the root cn^2 dominates the total cost of the tree.

Now we can use the substitution method to verify that our guess was correct. We want to show that $T(n) \leq dn^2$ for some constant $d > 0$.

We have

$$\begin{aligned} T(n) &\leq 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\leq 3d \lfloor n/4 \rfloor^2 + cn^2 \\ &\leq 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16} dn^2 + cn^2 \\ &\leq dn^2 , \end{aligned}$$

where the last step holds as long as $d \geq 16c/13$ (that is $c \leq 13d/16$).

2.4 Solving recurrences: master method

The master method provides a cookbook method for solving recurrences of the form

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

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

The above recurrence describes the running time of an algorithm that divides a problem of size n into a subproblems, each of size n/b. 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.

Note that replacing each of the a terms $T(n/b)$ with $T(\lfloor n/b \rfloor)$ or $T(\lceil n/b \rceil)$ will not affect the asymptotic behavior of the recurrence.

Theorem (Master theorem)

Let $a \geq 1$ 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 $T(\lfloor n/b \rfloor)$ or $T(\lceil n/b \rceil)$. Then $T(n)$ has following asymptotic bounds:

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

In each case, we compare the function $f(n)$ with the function $n^{\log_b a}$. Intuitively, the larger of the two functions determines the solution to the recurrence:

If, as in case 1, the function $n^{\log_b a}$ a is the larger, then the solution is $T(n) = \Theta(n^{\log_b a})$.

If, as in case 3, the function $f(n)$ is the larger, then the solution is $T(n) = \Theta(f(n))$.

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

Note that the three cases do not cover all the possibilities for $f(n)$.

Example (Find running time of the recurrence $T(n) = 9T(n/3) + n$)

We have $a = 9$, $b = 3$, $f(n) = n$ and thus $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$.

Since $f(n) = O(n^{\log_3 9 - \epsilon})$, where $\epsilon = 1$, case 1 applies and we obtain that $T(n) = \Theta(n^2)$.

Example (Find running time of the recurrence $T(n) = T(2n/3) + 1$)

We have $a = 1$, $b = 3/2$, $f(n) = 1$ and thus $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$.

Since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, case 2 applies and we obtain that $T(n) = \Theta(\lg n)$.

Example (Find running time of the recurrence $T(n) = 3T(n/4) + n\lg n$)

We have $a = 3$, $b = 4$, $f(n) = n\lg n$ and thus $n^{\log_b a} = n^{\log_4 3} = \Theta(n^{0.79})$.

Since $f(n) = \Omega(n^{\log_4 3+\epsilon})$, where $\epsilon \approx 0.2$, case 3 applies and we obtain that $T(n) = \Theta(n\lg n)$ after verifying the regularity condition.

Regularity condition: we have, for sufficiently large n, that

$$af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n\lg(n) = cf(n)$$

for $c = 3/4$.

Example (Find running time of recurrence $T(n) = 2T(n/2) + \Theta(n)$)

This is what we have promised on page 25.

We have $a = 2$, $b = 2$, $f(n) = \Theta(n)$ and thus $n^{\log_b a} = n^{\log_2 2} = n$.

Since $f(n) = \Theta(n)$, case 2 applies and we obtain that $T(n) = \Theta(n \lg n)$.

3. Probabilistic Analysis

3.1 Hiring problem

Suppose that you need to hire a new office assistant by using an employment agency.

The employment agency sends you candidates for interview. You must pay the employment agency a small fee to interview each applicant.

You are committed to having, at all times, the best possible person for the job. Therefore, you decide that, after interviewing each applicant, if that applicant is better qualified than the current office assistant, you will fire the current office assistant and hire the new applicant.

The pseudocode HIRE-ASSISTANT of *n* candidates is given below.

```
HIRE-ASSISTANT(n)
1  best = 0          // candidate 0 is a least-qualified dummy candidate
2  for i = 1 to n
3      interview candidate i
4      if candidate i is better than candidate best
5          best = i
6          hire candidate i
```

The procedure assumes that you are able to, after interviewing candidate *i* , determine whether candidate *i* is the best candidate you have seen so far.

To initialize, the procedure creates a dummy candidate, numbered 0, who is less qualified than each of the other candidates.

We are interested in the costs incurred by the HIRE-ASSISTANT algorithm. The analytical techniques used are the same as analyzing the running time.

Interviewing has a low cost, say c_i, whereas hiring is expensive, costing c_h. Letting m be the number of people hired, the total cost associated with this algorithm is $O(c_i n + c_h m)$.

No matter how many people we hire, we always interview n candidates and thus always incur the cost $c_i n$ associated with interviewing.

We therefore concentrate on analyzing $c_h m$, the hiring cost. This quantity varies with each run of the algorithm.

Worst-case analysis

In the worst case, we actually hire every candidate that we interview.

This situation occurs if the candidates come in strictly increasing order of quality, in which case we hire n times, for a total hiring cost of $O(c_h n)$.

Of course, the candidates do not always come in increasing order of quality. In fact, we have no idea about the order in which they arrive, nor do we have any control over this order.

Therefore, it is natural to ask what we expect to happen in a typical or average case.

Probabilistic analysis

Probabilistic analysis is the use of probability in the analysis of problems.

In order to perform a probabilistic analysis, we must use knowledge of, or make assumptions about, the distribution of the inputs.

Then we analyze our algorithm, computing an average-case running time, where we take the average over the distribution of the possible inputs.

We are, in effect, averaging the running time over all possible inputs. When reporting such a running time, we will refer to it as the **average-case running time**.

We rank each candidate with a unique number from 1 through n, to denote the rank of applicant, and adopt the convention that a higher rank corresponds to a better qualified applicant.

We can assume that the applicants come in a random order, which is equivalent of saying that the ranks form a **uniform random permutation**; that is, each of the possible $n!$ permutations appears with equal probability.

In the section 3.2, we will perform a probabilistic analysis of the hiring problem.

Randomized algorithms

In order to use probabilistic analysis, we need to know something about the distribution of the inputs. In many cases, we know very little about the input distribution.

Yet we often can use probability and randomness as a tool for algorithm design and analysis, by making the behavior of part of the algorithm random.

To develop a randomized algorithm for the hiring problem, we must have greater control over the order in which we interview the candidates.

The employment agency sends us a list of the n candidates in advance. Each time, we choose, randomly, which candidate to interview.

Instead of relying on a guess that the candidates come to us in a random order, we have instead gained control of the process and enforced a random order.

We call an algorithm **randomized** if its behavior is determined not only by its input but also by values produced by a **random-number generator**.

RANDOM(a, b) returns an integer between a and b, inclusive, with each such integer being equally likely. For example, RANDOM$(3, 7)$ returns either 3, 4, 5, 6, or 7, each with probability $1/5$.

When analyzing the running time of a randomized algorithm, we take the expectation of the running time over the distribution of values returned by the random number generator. The resulting running time of a randomized algorithm is referred to as an **expected running time**.

In general, we discuss the average-case running time when the probability distribution is over the inputs to the algorithm, and we discuss the expected running time when the algorithm itself makes random choices.

3.2 Indicator random variables

In order to analyze many algorithms, including the hiring problem, we use indicator random variables.

Indicator random variables provide a convenient method for converting between probabilities and expectations.

Then the indicator random variable $I\{A\}$ associated with event A is defined as

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

As a simple example, let us determine the expected number of heads that we obtain when flipping a fair coin with $\Pr\{H\} = \Pr\{T\} = 1/2$.

We define an indicator random variable X_H associated with the coin coming up heads

$$\begin{aligned} X_H &= I\{H\} \\ &= \begin{cases} 1 & \text{if H occurs} \\ 0 & \text{if T occurs.} \end{cases} \end{aligned}$$

The expected number of heads obtained in one flip of the coin is simply the expected value of our indicator variable X_H

$$\begin{aligned} E[X_H] &= E[I\{H\}] \\ &= 1 \cdot Pr\{H\} + 0 \cdot Pr\{T\} \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) = 1/2. \end{aligned}$$

In general, the expected value of an indicator random variable associated with an event A is equal to the probability that A occurs

$$E[X_A] = Pr\{A\}.$$

Returning to the hiring problem, we now compute the expected number of times that we hire a new office assistant. In order to use a probabilistic analysis, we assume that the candidates arrive in a random order.

Let $X_i, i = 0, \ldots, n$ be the indicator random variable associated with the event in which the i-th candidate is hired

$$\begin{aligned} X_i &= I\{\text{candidate } i \text{ is hired}\} \\ &= \begin{cases} 1 & \text{if candidate } i \text{ is hired} \\ 0 & \text{if candidate } i \text{ is not hired.} \end{cases} \end{aligned}$$

Thus, we need to compute $E[X]$, where X the number of times we hire a new office assistant

$$X = X_1 + X_2 + \cdots + X_n.$$

Candidate i is hired exactly when candidate i is better than each of candidates 1 through $i - 1$.

Because we have assumed that the candidates arrive in a random order, any one of these first i candidates is equally likely to be the best-qualified so far.

Candidate i has a probability of $1/i$ of being better qualified than candidates 1 through $i - 1$ and thus a probability of $1/i$ of being hired

$$E[X_i] = \frac{1}{i}$$

We can now compute $E[X]$

$$\begin{aligned} E[X] &= E\left[\sum_{i}^{n} X_i\right] \\ &= \sum_{i=1}^{n} E[X_i] \\ &= \sum_{i=1}^{n} \frac{1}{i} \\ &= \ln n + O(1), \end{aligned}$$

where the last equality follows from the asymptotic behavior of harmonic sum.

Assuming that the candidates are presented in a random order, algorithm HIRE-ASSISTANT has an average-case total hiring cost of $O(c_h \ln n)$.

The average-case hiring cost is a significant improvement over the worst-case hiring cost $O(c_h n)$.

4. Dynamic Programming

Dynamic programming

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. (Programming in this context refers to a tabular method, not to writing computer code.)

Divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.

In contrast, dynamic programming applies when the subproblems overlapthat is, when subproblems share subsubproblems.

A dynamic-programming algorithm solves each subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subproblem.

We typically apply dynamic programming to optimization problems.

Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

When developing a dynamic-programming algorithm, we follow a sequence of three essential steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.

In this section, we use the dynamic-programming method to solve some optimization problems.

We first examine the problem of cutting a rod into rods of smaller length in way that maximizes their total value.

We then study how we can multiply a chain of matrices while performing the fewest total scalar multiplications.

Given these examples of dynamic programming, we will discuss two key characteristics that a problem must have for dynamic programming to be a viable solution technique.

4.1 Rod cutting problem

Our first example uses dynamic programming to solve a simple problem in deciding where to cut steel rods.

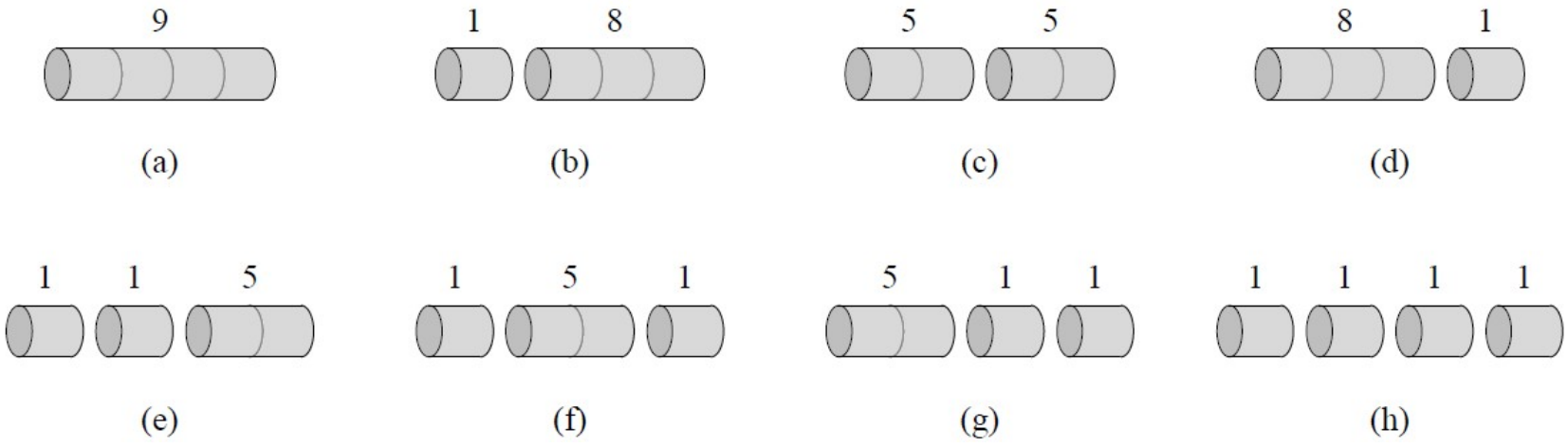
We assume that we know, for $i = 1, 2, \ldots,$ the price p_i in dollars for a rod of length i inches.

An example of price table

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Rod-cutting problem: Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

Consider the case when $n = 4$, we have 8 ways of cutting, where above each piece is its value:



We see that cutting a 4-inch rod into two 2-inch pieces produces revenue $p_2 + p_2 = 5 + 5 = 10$, which is optimal.

We can cut up a rod of length n in 2^n different ways, since we have an independent option of cutting, or not cutting, at distance i inches from the left end, for $i = 1, 2, \ldots, n-1$.

We denote a decomposition into pieces using ordinary additive notation, so that $7 = 2 + 2 + 3$ indicates that a rod of length 7 is cut into three pieces - two of length 2 and one of length 3.

If an optimal solution cuts the rod into k pieces, for some $1 \leq k \leq n$, then an optimal decomposition

$$n = i_1 + i_2 + \cdots + i_k$$

of the rod into pieces of lengths $i_1, i_2, \ldots, i_k$ provides maximum corresponding revenue

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}.$$

For our sample problem, we can determine the optimal revenue r_i, for $i = 1, 2, \ldots, 10$, by inspection, with the corresponding optimal decompositions

$$
\begin{aligned}
r_1 &= 1 \quad \text{from solution } 1 = 1 \quad \text{(no cuts)}, \\
r_2 &= 5 \quad \text{from solution } 2 = 2 \quad \text{(no cuts)}, \\
r_3 &= 8 \quad \text{from solution } 3 = 3 \quad \text{(no cuts)}, \\
r_4 &= 10 \quad \text{from solution } 4 = 2 + 2, \\
r_5 &= 13 \quad \text{from solution } 5 = 2 + 3, \\
r_6 &= 17 \quad \text{from solution } 6 = 6 \quad \text{(no cuts)}, \\
r_7 &= 18 \quad \text{from solution } 7 = 1 + 6 \text{ or } 7 = 2 + 2 + 3, \\
r_8 &= 22 \quad \text{from solution } 8 = 2 + 6, \\
r_9 &= 25 \quad \text{from solution } 9 = 3 + 6, \\
r_{10} &= 30 \quad \text{from solution } 10 = 10 \quad \text{(no cuts)}.
\end{aligned}
$$

More generally, we can frame the values r_n for $n \geq 1$ in terms of optimal revenues from shorter rods:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1).$$

The first argument, p_n, corresponds to making no cuts at all and selling the rod of length n.

The other $n - 1$ arguments to max correspond to the maximum revenue obtained by making an initial cut of the rod into two pieces of size i and $n - i$, for each $i = 1, 2, \ldots, n - 1$, and then optimally cutting up those pieces further, obtaining revenues r_i and r_{n-i} from those two pieces.

Since we don't know ahead of time which value of i optimizes revenue, we have to consider all possible values for i and pick the one that maximizes revenue.

Once we make the first cut, we may consider the two pieces as independent instances of the rod-cutting problem.

The overall optimal solution incorporates optimal solutions to the two related subproblems, maximizing revenue from each of those two pieces.

We say that the rod-cutting problem exhibits **optimal substructure**: optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.

Another way to arrange a recursive structure for the rod cutting problem is to view a decomposition as consisting of a first piece of length i cut off the left-hand end, and then a right-hand remainder of length $n - i$. Only the remainder, and not the first piece, may be further divided.

We thus obtain the following simpler version of the revenue optimization of rod cutting problem

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

In this formulation, an optimal solution embodies the solution to only one related subproblem - the remainder - rather than two.

The procedure in the next slides implements the above equation in a straightforward, top-down, recursive manner.

```
CUT-ROD(p, n)
1  if n == 0
2       return 0
3  q = -∞
4  for i = 1 to n
5       q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

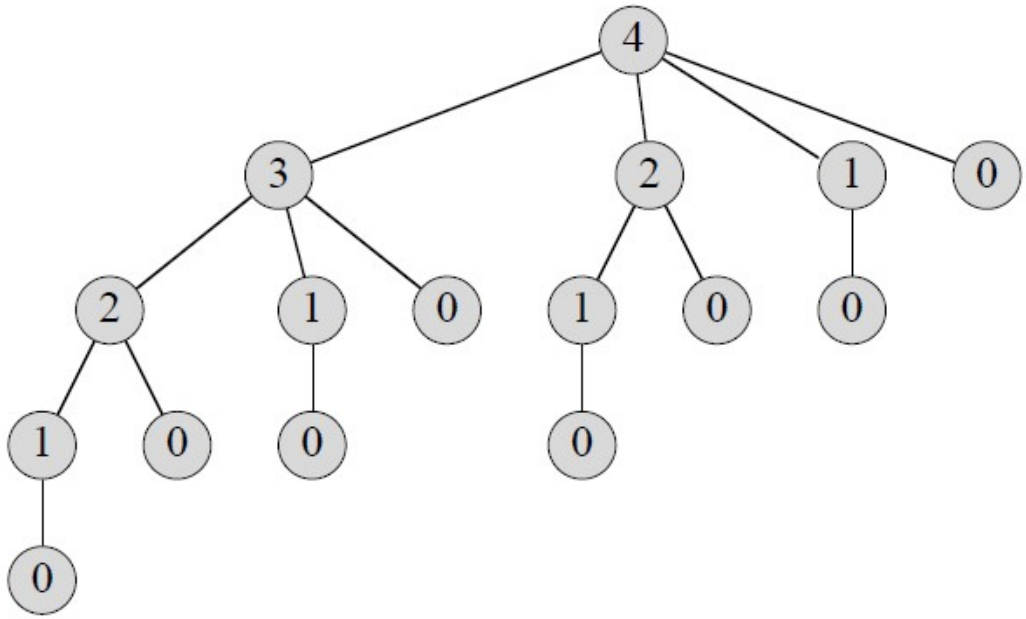
The algorithm CUT-ROD takes as input an array $p[1..n]$ of prices and an integer n, and it returns the maximum revenue possible for a rod of length n.

If $n = 0$, no revenue is possible, and so CUT-ROD returns 0 in line 2.

Line 3 initializes the maximum revenue q to ∞, so that the **for** loop in lines $4 - 5$ correctly computes the q value.

CUT-ROD algorithm is inefficient as it calls itself recursively over and over again with the same parameter values.

For $n = 4$, CUT-ROD(p, n) calls CUT-ROD$(p, n - j)$ for $i = 1, 2, \ldots, n$ as illustrated below



Each node label gives the size n of the corresponding subproblem, so that an edge from a parent with label s to a child with label t corresponds to cutting off an initial piece of size $s - t$ and leaving a remaining subproblem of size t.

A path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n.

Dynamic programming for optimal rod cutting

Having observed that a naive recursive solution is inefficient because it solves the same subproblems repeatedly.

We arrange for each subproblem to be solved only once, saving its solution. If we need to refer to this subproblem's solution again later, we can just look it up, rather than recompute it.

Dynamic programming thus uses additional memory to save computation time; it serves an example of a **time-memory trade-off**.

The savings may be dramatic: an exponential-time solution may be transformed into a polynomial-time solution.

A dynamic-programming approach runs in polynomial time when the number of distinct subproblems involved is polynomial in the input size and we can solve each such subproblem in polynomial time.

There are usually two equivalent ways to implement a dynamic-programming approach.

The first approach is **top-down with memoization**. In this approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem.

The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the value in the usual manner.

We say that the recursive procedure has been memoized; it remembers what results it has computed previously.

The second approach is the **bottom-up method**.

This approach typically depends on some natural notion of the size of a subproblem, such that solving any particular subproblem depends only on solving smaller subproblems.

We sort the subproblems by size and solve them in size order, smallest first. When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions.

We solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

These two approaches yield algorithms with the same asymptotic running time.

Here is the the pseudocode for the top-down CUT-ROD procedure, with memoization:

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = -∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
8  r[n] = q
9  return q
```

The main procedure MEMOIZED-CUT-ROD initializes a new auxiliary array $r[0..n]$ with the value $-\infty$, a convenient choice to denote unknown.

It then calls its helper routine, MEMOIZED-CUT-ROD-AUX, which is just the memoized version of our previous algorithm CUT-ROD on page 74.

It first checks in line 1 to see whether the desired value is already known and, if it is, then line 2 returns it.

Otherwise, lines $3-7$ compute the desired value q in the usual manner, line 8 saves it in $r[n]$.

The bottom-up version:

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4       q = -∞
5       for i = 1 to j
6            q = max(q, p[i] + r[j - i])
7       r[j] = q
8  return r[n]
```

BOTTOM-UP-CUT-ROD uses the natural ordering of the subproblems: a subproblem of size i is smaller than a subproblem of size j if $i < j$.

Thus, the procedure solves subproblems of sizes $j = 0, 1, \ldots, n$, in that order.

Line 1 of BOTTOM-UP-CUT-ROD creates a new array $r[0..n]$ in which to save the results of the subproblems.

Line 2 initializes $r[0]$ to 0, since a rod of length 0 earns no revenue.

Lines $3-6$ solve each subproblem of size j, for $j = 1, 2, \ldots, n$, in order of increasing size.

The approach used to solve a problem of a particular size j is the same as that used by CUT-ROD, except that line 6 now directly references array entry $r[j-i]$ instead of making a recursive call to solve the subproblem of size $j-i$.

Line 7 saves in $r[j]$ the solution to the subproblem of size j.

The running time of procedure BOTTOM-UP-CUT-ROD is $\Theta(n^2)$, due to its doubly-nested loop structure.

The number of iterations of its inner **for** loop, in lines $5-6$, forms an arithmetic series.

The running time of its top-down counterpart, MEMOIZED-CUT-ROD, is also $\Theta(n^2)$.

MEMOIZED-CUT-ROD solves subproblems for sizes $0, 1, \ldots, n$ just once.

To solve a subproblem of size i, the **for** loop of lines $6-7$ iterates i times.

Thus, the total number of iterations of this **for** loop, over all recursive calls, forms an arithmetic series, giving a total of $\Theta(n^2)$ iterations.

4.2 Matrix-chain multiplication

We are given a sequence (chain) $A_1, A_2, \ldots, A_n$ of n matrices to be multiplied, and we wish to compute the product

$$A_1 A_2 \cdots A_n.$$

We can evaluate the expression using the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together.

Matrix multiplication is associative, and so all parenthesizations yield the same product.

A product of matrices is **fully parenthesized** if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses.

For example, we can fully parenthesize the product $A_1 A_2 A_3 A_4$ in five ways:

$$
\begin{aligned}
&(A_1(A_2(A_3A_4))) ,\\
&(A_1((A_2A_3)A_4)) ,\\
&((A_1A_2)(A_3A_4)) ,\\
&((A_1(A_2A_3))A_4) ,\\
&(((A_1A_2)A_3)A_4) .
\end{aligned}
$$

How we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product.

Consider first the cost of multiplying two matrices. The standard algorithm is given by the pseudocode:

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2      error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4      for i = 1 to A.rows
5          for j = 1 to B.columns
6              c_ij = 0
7              for k = 1 to A.columns
8                  c_ij = c_ij + a_ik · b_kj
9      return C
```

We can multiply two matrices A and B only if they are compatible: the number of columns of A must equal the number of rows of B.

If A is a $p \times q$ matrix and B is a $q \times r$ matrix, the resulting matrix C is a $p \times r$ matrix.

The time to compute C is dominated by the number of scalar multiplications in line 8, which is pqr.

In what follows, we shall express costs in terms of the number of scalar multiplications.

Consider the product of three matrices of the dimensions 10×100, 100×5, and 5×50, respectively.

If we multiply according to the parenthesization $((A_1 A_2)A_3)$, we perform $10 \cdot 100 \cdot 5 = 5,000$ scalar multiplications to compute the product $A_1 A_2$, plus another $10 \cdot 5 \cdot 50 = 2,500$ scalar multiplications to multiply this matrix by A_3, for a total of $7,500$ scalar multiplications.

In instead we multiply according to the parenthesization $(A_1(A_2 A_3))$, we perform $100 \cdot 5 \cdot 50 = 25,000$ scalar multiplications to compute the product $A_2 A_3$, plus another $10 \cdot 100 \cdot 50 = 50,000$ scalar multiplications to multiply this matrix by A_1, for a total of $75,000$ scalar multiplications.

Thus, computing the product according to the first parenthesization is 10 times faster.

We state the **matrix-chain multiplication problem** as follows: given a chain $A_1, A_2, \ldots, A_n$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost.

Exhaustively checking all possible parenthesizations leads to grows as $\Omega(2^n)$, thus the number of parenthesizations is exponential in n.

Dynamic programming for optimal parenthesization

We shall use the dynamic-programming method to determine how to optimally parenthesize a matrix chain.

In doing so, recall that a sequence of three essential steps is:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.

Dynamic programming for optimal parenthesization

For our first step in the dynamic-programming paradigm, we find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems.

Denote $A_{i..j}$, $i \leq j$, as the product $A_i A_{i+1} \cdots A_j$. To parenthesize the product, we must split the product between A_k and A_{k+1} for some integer k in the range $i \leq k < j$.

That is, for some value of k, we first compute the matrices $A_{i..k}$ and $A_{k+1..j}$ and then multiply them together to produce the final product $A_{i..j}$.

The cost of parenthesizing this way is the cost of computing the matrix $A_{i..k}$, plus the cost of computing $A_{k+1..j}$, plus the cost of multiplying them together.

The optimal substructure of this problem is as follows.

Suppose that to optimally parenthesize $A_i A_{i+1} \cdots A_j$, we split the product between A_k and A_{k+1}. Then the way we parenthesize the prefix subchain $A_i A_{i+1} \cdots A_k$ within this optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \cdots A_k$.

The reason is that if there were a less costly way to parenthesize $A_i A_{i+1} \cdots A_k$, then we could substitute that parenthesization in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ to produce another way to parenthesize $A_i A_{i+1} \cdots A_j$ whose cost was lower than the optimum: a contradiction.

A similar observation holds for how we parenthesize the subchain $A_{k+1} A_{k+2} \cdots A_j$ in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$: it must be an optimal parenthesization of $A_{k+1} A_{k+2} \cdots A_j$.

Thus, we can build an optimal solution to an instance of the matrix-chain multiplication problem by splitting the problem into two subproblems: optimally parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$.

We then find optimal solutions to subproblem instances, and then combining these optimal subproblem solutions.

We must ensure that when we search for the correct place to split the product, we have considered all possible places, so that we are sure of having examined the optimal one.

A recursive solution

Next, we define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems.

We pick as our subproblems the problems of determining the minimum cost of parenthesizing $A_i A_{i+1} \cdots A_j$ for $1 \leq i \leq j \leq n$.

Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$. For the full problem, the lowest cost way to compute $A_{1..n}$ would thus be $m[1, n]$.

If $i = j$, the problem is trivial; the chain consists of just one matrix $A_{i..i} = A_i$, so that no scalar multiplications are necessary to compute the product. That is $m[i, i] = 0$ for $i = 1, 2, \ldots, n$.

To compute $m[i, j]$ when $i < j$, we take advantage of the structure of an optimal solution, and we can define $m[i, j]$ recursively as follows.

Let us assume that to optimally parenthesize, we split the product $A_i A_{i+1} \cdots A_j$ between A_k and A_{k+1}, where $i \leq k < j$. Then, $m[i, j]$ equals the minimum cost for computing the subproducts $A_{i..k}$ and $A_{k+1..j}$, plus the cost of multiplying these two matrices together.

Recalling that each matrix A_i is $p_{i-1} \times p_i$, we see that computing the matrix product $A_{i..k} A_{k+1..j}$ takes $p_{i-1} p_k p_j$ scalar multiplications. Thus, we obtain

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j.$$

There are only $j - i$ possible values for k, namely, $k = i, i + 1, \ldots, j - 1$. Since the optimal parenthesization must use one of these values for k, we need only check them all to find the best.

Thus, our recursive definition for the minimum cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i > j. \end{cases}$$

The $m[i,j]$ values give the costs of optimal solutions to subproblems, but they do not provide all the information we need to construct an optimal solution.

To help us do so, we define $s[i,j]$ to be a value of k at which we split the product $A_i A_{i+1} \cdots A_j$ in an optimal parenthesization.

That is, $s[i,j]$ equals the value of k such that

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j.$$

Computing the optimal costs

Observe that we have relatively few distinct subproblems: one subproblem for each choice of i and j satisfying $1 \leq i \leq j \leq n$ or $\Omega(n^2)$ in all.

A recursive algorithm may encounter each subproblem many times in different branches of its recursion tree.

This property of overlapping subproblems is the second hallmark of when dynamic programming applies (the first hallmark being optimal substructure).

Instead of computing the solution to recurrence $m[i, j]$ recursively, we compute the optimal cost by using a tabular, bottom-up approach.

We shall implement the tabular, bottom-up method in the procedure MATRIX-CHAIN-ORDER, which appears on the next page.

This algorithm assumes that matrix A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, \ldots, n$. Its input is a sequence $p = (p_0, p_1, \ldots, p_n)$, where $p.length = n + 1$.

The procedure uses an auxiliary table $m[1..n, 1..n]$ for storing the $m[i, j]$ costs and another auxiliary table $s[1..n-1, 2..n]$ that records which index k achieved the optimal cost in computing $m[i, j]$.

We shall use the table s to construct an optimal solution.

MATRIX-CHAIN-ORDER(p)

```
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

The algorithm first sets $m[i, j] = 0$ for $i = 1, 2, \ldots, n$ in lines $3 - 4$.

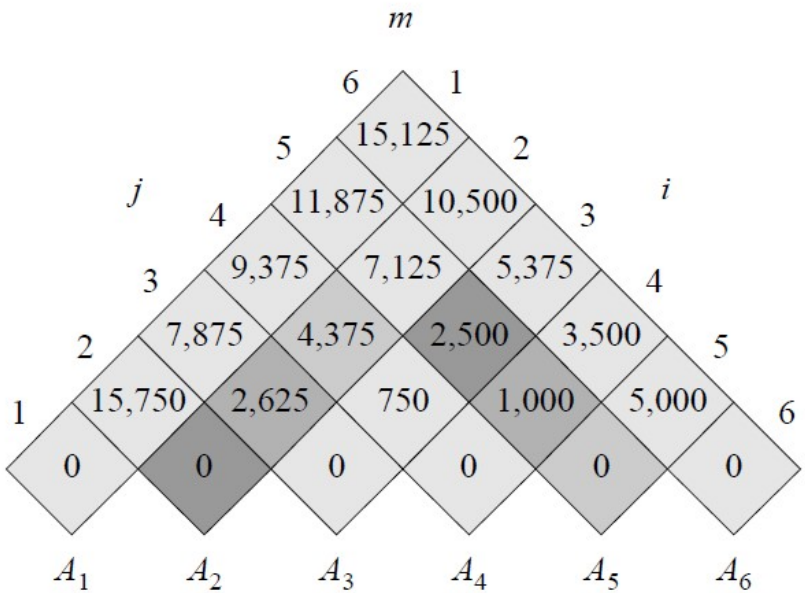
It then uses recurrence of $m[i, j]$ to compute $m[i, i + 1]$ for $i = 1, 2, \ldots, n - 1$ (the minimum costs for chains of length $l = 2$) during the first execution of the **for** loop in lines $5 - 13$.

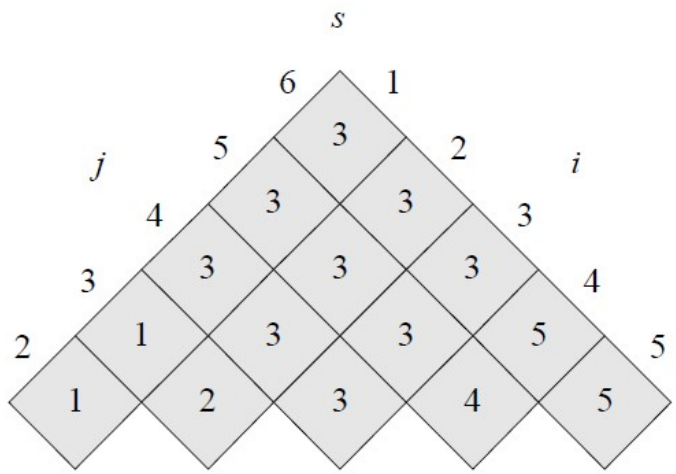
The second time through the loop, it computes $m[i, i + 2]$ for $i = 1, 2, \ldots, n - 2$ (the minimum costs for chains of length $l = 3$), and so forth.

At each step, the $m[i, j]$ cost computed in lines $10 - 13$ depends only on table entries $m[i, k]$ and $m[k + 1, j]$ already computed.

We illustrate this algorithm on a chain of $n = 6$ matrices.

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25





Since we have defined $m[i, j]$ only for $i \leq j$, only the portion of the table m strictly above the main diagonal is used.

The figure shows the table rotated to make the main diagonal run horizontally. The matrix chain is listed along the bottom.

Using this layout, we can find the minimum cost $m[i, j]$ for multiplying a subchain $A_i A_{i+1} \cdots A_j$ of matrices at the intersection of lines running from A_i and A_j in different directions.

Each horizontal row in the table contains the entries for matrix chains of the same length.

MATRIX-CHAIN-ORDER computes the rows from bottom to top and from left to right within each row. It computes each entry $m[i, j]$ using the products $p_{i-1} p_k p_j$ for $k = i, i + 1, \ldots, j - 1$.

The minimum number of scalar multiplications to multiply the 6 matrices is $m[1,6] = 15,125$.

Of the darker entries, the pairs that have the same shading are taken together in line 10 when computing $m[2,5]$, whose optimal value turns out to be 7125 (question 3 of assignment 2).

A simple inspection of the nested loop structure of MATRIX-CHAIN-ORDER yields a running time of $O(n^3)$ for the algorithm. The loops are nested three deep, and each loop index (l, i, and k) takes on at most $n - 1$ values.

Thus, MATRIX-CHAIN-ORDER is much more efficient than the exponential-time method of enumerating all possible parenthesizations and checking each one.

4.3 Theory of dynamic programming

Although we have just worked through two examples of the dynamic programming method, you might still be wondering just when the method applies.

From an engineering perspective, when should we look for a dynamic programming solution to a problem?

We examine the two key ingredients that an optimization problem must have in order for dynamic programming to apply: **optimal substructure** and **overlapping subproblems**.

Optimal substructure

The first step in solving an optimization problem by dynamic programming is to characterize the structure of an optimal solution.

Recall that a problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.

Whenever a problem exhibits optimal substructure, we have a good clue that dynamic programming might apply.

In dynamic programming, we build an optimal solution to the problem from optimal solutions to subproblems.

Consequently, we must take care to ensure that the range of subproblems we consider includes those used in an optimal solution.

We discovered optimal substructure in both of the problems we have examined in this chapter so far.

In rod cutting problem, we observed that the optimal way of cutting up a rod of length n involves optimally cutting up the two pieces resulting from the first cut.

In optimal parenthesization problem, we observed that an optimal parenthesization of $A_i A_{i+1} \cdots A_j$ that splits the product between A_k and A_{k+1} contains within it optimal solutions to the problems of parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$.

You will find yourself following a common pattern in discovering optimal substructure:

1. You show that a solution to the problem consists of making a choice, such as choosing an initial cut in a rod or choosing an index at which to split the matrix chain. Making this choice leaves one or more subproblems to be solved.

2. You suppose that for a given problem, you are given the choice that leads to an optimal solution. You do not concern yourself yet with how to determine this choice. You just assume that it has been given to you.

3. Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems.

4. You show that the solutions to the subproblems used within an optimal solution to the problem must themselves be optimal by using a cut-and-paste technique. You do so by supposing that each of the subproblem solutions is not optimal and then deriving a contradiction.

Dynamic programming often uses optimal substructure in a bottom-up fashion. That is, we first find optimal solutions to subproblems and, having solved the subproblems, we find an optimal solution to the problem.

The cost of the problem solution is usually the subproblem costs plus a cost that is directly attributable to the choice itself.

In rod cutting, first we solved the subproblems of determining optimal ways to cut up rods of length i for $i = 0, 1, \ldots, n-1$, and then we determined which such subproblem yielded an optimal solution for a rod of length n. The cost attributable to the choice itself is the term p_i.

In matrix-chain multiplication, we determined optimal parenthesizations of subchains of $A_i A_{i+1} \cdots A_j$, and then we chose the matrix A_k at which to split the product. The cost attributable to the choice itself is the term $p_{i+1} p_k p_j$.

Optimal substructure

The second ingredient that an optimization problem must have for dynamic programming to apply is that the space of subproblems must be small in the sense that a recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems.

Typically, the total number of distinct subproblems is a polynomial in the input size.

When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has overlapping subproblems.

In contrast, a problem for which a divide-and-conquer approach is suitable usually generates brand-new problems at each step of the recursion.

Dynamic-programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed, using constant time per lookup.

5. Greedy Algorithms

Greedy Algorithms

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step.

For many optimization problems, using dynamic programming to determine the best choices is overkill; simpler, more efficient algorithms will do.

A **greedy algorithm** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

This section explores optimization problems for which greedy algorithms provide optimal solutions.

Greedy algorithms do not always yield optimal solutions, but for many problems they do.

We shall first examine a simple but nontrivial problem, the activity-selection problem, for which a greedy algorithm efficiently computes an optimal solution.

We shall arrive at the greedy algorithm by first considering a dynamic programming approach and then showing that we can always make greedy choices to arrive at an optimal solution.

We then review the basic elements of the greedy approach: **Greedy-choice property** and **optimal substructure**, while comparing Greedy and dynamic programming.

5.1 Activity-selection problem

We study the problem of scheduling several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities.

Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed activities that wish to use a resource, such as a lecture hall, which can serve only one activity at a time.

Each activity a_i has a start time s_i and a finish time f_i, where $0 \leq s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$.

Activities a_i and a_j are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$.

In the **activity-selection problem**, we wish to select a maximum-size subset of mutually compatible activities.

We assume that the activities are sorted in monotonically increasing order of finish time

$$f_1 \leq f_2 \leq \cdots \leq f_{n-1} \leq f_n.$$

We shall see later the advantage that this assumption provides.

Consider the example of the following set S of activities

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

In this example, the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities. It is not a maximum subset since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger, which in fact is a largest subset of mutually compatible activities. Another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.

We shall solve this problem in several steps.

We start by thinking about a dynamic programming solution, in which we consider several choices when determining which subproblems to use in an optimal solution.

We shall then observe that we need to consider only one choice - the greedy choice - and that when we make the greedy choice, only one subproblem remains.

Based on these observations, we shall develop a recursive greedy algorithm to solve the activity-scheduling problem.

Although the steps we shall go through in the following are slightly more involved than is typical when developing a greedy algorithm, they illustrate the relationship between greedy algorithms and dynamic programming.

The optimal substructure of the activity-selection problem

We can easily verify that the activity-selection problem exhibits optimal substructure.

Let us denote by S_{ij} the set of activities that start after activity a_i finishes and that finish before activity a_j starts.

Suppose that we wish to find a maximum set of mutually compatible activities in S_{ij}, and suppose further that such a maximum set is A_{ij}, which includes some activity a_k.

By including a_k in an optimal solution, we are left with two subproblems: finding mutually compatible activities in the set S_{ik} (activities that start after activity a_i finishes and that finish before activity a_k starts) and finding mutually compatible activities in the set S_{kj} (activities that start after activity a_k finishes and that finish before activity a_j starts).

Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$, so that A_{ik} contains the activities in A_{ij} that finish before a_k starts and A_{kj} contains the activities in A_{ij} that starts after a_k finishes.

Thus, we have $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, and so the maximum-size set A_{ij} of mutually compatible activities in S_{ij} consists of $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ activities.

The usual cut-and-paste argument shows that the optimal solution A_{ij} must also include optimal solutions to the two subproblems for S_{ik} and S_{kj}.

If we could find a set A'_{kj} of mutually compatible activities in S_{kj} where $|A'_{kj}| \geq |A_{kj}|$, then we could use A'_{kj}, rather than A_{kj}, in a solution to the subproblem for S_{ij}. We would have constructed a set of $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$ mutually compatible activities, which contradicts the assumption that A_{ij} is an optimal solution.

This way of characterizing optimal substructure suggests that we might solve the activity-selection problem by dynamic programming.

If we denote the size of an optimal solution for the set S_{ij} by $c[i, j]$, then we would have the recurrence

$$c[i, j] = c[i, k] + c[k, j] + 1.$$

Of course, if we did not know that an optimal solution for the set S_{ij} includes activity a_k, we would have to examine all activities in S_{ij} to find which one to choose, so that

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset. \end{cases}$$

We could then develop a recursive algorithm and memoize it, or we could work bottom-up and fill in table entries as we go along. But we would be overlooking another important characteristic of the activity-selection problem that we can use to great advantage.

Making the greedy choice

What if we could choose an activity to add to our optimal solution without having to first solve all the subproblems?

That could save us from having to consider all the choices inherent in recurrence of $c[i, j]$.

Intuition suggests that we should choose an activity that leaves the resource available for as many other activities as possible.

Now, of the activities we end up choosing, one of them must be the first one to finish. Our intuition tells us, therefore, to choose the activity in S with the earliest finish time, since that would leave the resource available for as many of the activities that follow it as possible

In other words, since the activities are sorted in monotonically increasing order by finish time, the greedy choice is activity a_1.

If we make the greedy choice, we have only one remaining subproblem to solve: finding activities that start after a_1 finishes.

Let $S_k = \{a_i \in S : s_i \geq f_k\}$ be the set of activities that start after activity a_k finishes.

If we make the greedy choice of activity a_1, then S_1 remains as the only subproblem to solve.

Optimal substructure tells us that if a_1 is in the optimal solution, then an optimal solution to the original problem consists of activity a_1 and all the activities in an optimal solution to the subproblem S_1.

This greedy choice - in which we choose the first activity to finish - is always an optimal solution (proof omitted).

Thus, we see that although we might be able to solve the activity-selection problem with dynamic programming, we don't need to.

Instead, we can repeatedly choose the activity that finishes first, keep only the activities compatible with this activity, and repeat until no activities remain.

Moreover, because we always choose the activity with the earliest finish time, the finish times of the activities we choose must strictly increase.

We can consider each activity just once overall, in monotonically increasing order of finish times.

An algorithm to solve the activity-selection problem does not need to work bottom-up, like a table-based dynamic-programming algorithm.

Instead, it can work top-down, choosing an activity to put into the optimal solution and then solving the subproblem of choosing activities from those that are compatible with those already chosen.

Greedy algorithms typically have this top-down design: make a choice and then solve a subproblem, rather than the bottom-up technique of solving subproblems before making a choice.

A recursive greedy algorithm

Now that we have seen how to bypass the dynamic-programming approach and instead use a top-down, greedy algorithm, we can write a straightforward, recursive procedure to solve the activity-selection problem.

The procedure RECURSIVE-ACTIVITY-SELECTOR takes the start and finish times of the activities, represented as arrays s and f, the index k that defines the subproblem S_k it is to solve, and the size n of the original problem. It returns a maximum-size set of mutually compatible activities in S_k.

We assume that the n input activities are already ordered by monotonically increasing finish time.

In order to start, we add the fictitious activity a_0 with $f_0 = 0$, so that subproblem S_0 is the entire set of activities S. The initial call, which solves the entire problem, is RECURSIVE-ACTIVITY-SELECTOR(s,f,0,n).

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

In a given recursive call RECURSIVE-ACTIVITY-SELECTOR(s,f,k,n), the **while** loop of lines $2-3$ looks for the first activity in S_k to finish.

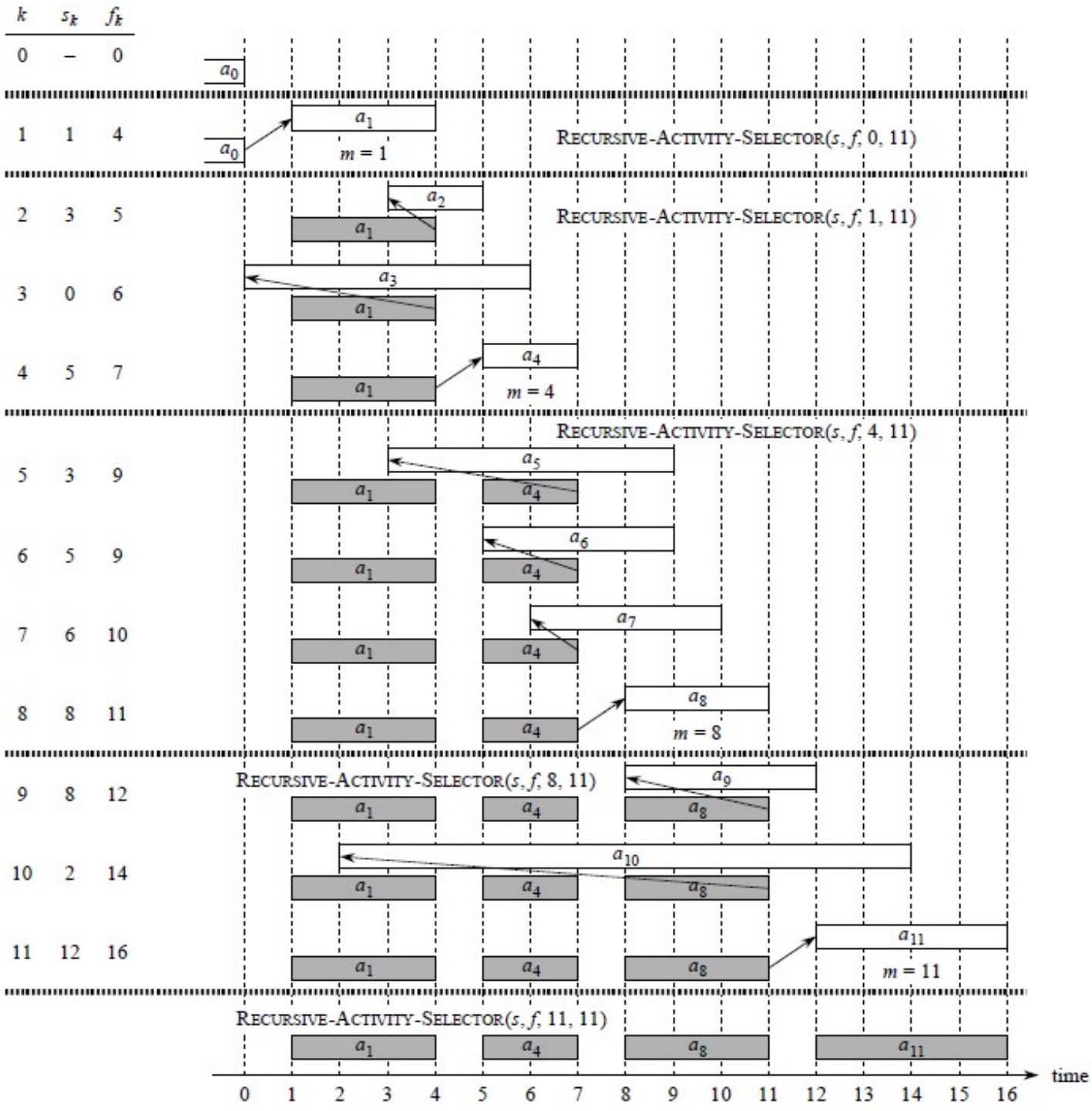
The loop examines $a_{k+1}, a_{k+2}, \ldots, a_n$, until it finds the first activity a_m that is compatible with a_k; such an activity has $s_m \geq f_k$.

If the loop terminates because it finds such an activity, line 5 returns the union of $\{a_m\}$ and the maximum-size subset of S_m returned by the recursive call RECURSIVE-ACTIVITY-SELECTOR(s,f,m,n).

Alternatively, the loop may terminate because $m > n$, in which case we have examined all activities in S_k without finding one that is compatible with a_k. In this case, $S_k = \emptyset$, and so the procedure returns $\emptyset$ in line 6.

Assuming that the activities have already been sorted by finish times, the running time of the call RECURSIVE-ACTIVITY-SELECTOR(s,f,0,n) is $\Theta(n)$.

This is because over all recursive calls, each activity is examined exactly once in the **while** loop test of line 2.


k
s_k
f_k
0 – 0
1 1 4
2 3 5
3 0 6
4 5 7
5 3 9
6 5 9
7 6 10
8 8 11
9 8 12
10 2 14
11 12 16
a_0
a_1
a_2
a_3
a_4
a_5
a_6
a_7
a_8
a_9
a_{10}
a_{11}
$m = 1$
$m = 4$
$m = 8$
$m = 11$
RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 11)
RECURSIVE-ACTIVITY-SELECTOR(s, f, 1, 11)
RECURSIVE-ACTIVITY-SELECTOR(s, f, 4, 11)
RECURSIVE-ACTIVITY-SELECTOR(s, f, 8, 11)
RECURSIVE-ACTIVITY-SELECTOR(s, f, 11, 11)
time
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

The above example shows the operation of RECURSIVE-ACTIVITY-SELECTOR on the 11 activities on page 113. Activities considered in each recursive call appear between horizontal lines.

The fictitious activity a_0 finishes at time 0, and the initial call RECURSIVE-ACTIVITY-SELECTOR(s,f,0,11) selects activity a_1.

In each recursive call, the activities that have already been selected are shaded, and the activity shown in white is being considered.

If the starting time of an activity occurs before the finish time of the most recently added activity (the arrow between them points left), it is rejected. Otherwise (the arrow points directly up or to the right), it is selected.

The last recursive call, RECURSIVE-ACTIVITY-SELECTOR(s,f,11,11) returns $\emptyset$. The resulting set of selected activities is $\{a_1, a_4, a_8, a_{11}\}$.

5.2 Elements of greedy strategy

A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices.

At each decision point, the algorithm makes the choice that seems best at the moment.

The process that we followed to develop a greedy algorithm for the activity-selection problem was a bit more involved than is typical. We went through the following steps:

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution.
3. Show that if we make the greedy choice, then only one subproblem remains.
4. Develop a recursive algorithm that implements the greedy strategy.

In going through these steps, we saw in great detail the dynamic programming underpinnings of a greedy algorithm.

In the activity-selection problem, we first defined the subproblems S_{ij}, where both i and j varied. We then found that if we always made the greedy choice, we could restrict the subproblems to be of the form S_k.

Alternatively, we could have fashioned our optimal substructure with a greedy choice in mind, so that the choice leaves just one subproblem to solve.

In the activity-selection problem, we could have started by dropping the second subscript and defining subproblems of the form S_k.

Then, we could have proven that a greedy choice (the first activity a_m to finish in S_k), combined with an optimal solution to the remaining set S_m of compatible activities, yields an optimal solution to S_k.

More generally, we design greedy algorithms according to the following sequence of steps:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

We shall use this more direct process of greedy algorithms. Nevertheless, beneath every greedy algorithm, there is almost always a more cumbersome dynamic-programming solution.

Greedy-choice property

How can we tell whether a greedy algorithm will solve a particular optimization problem?

No way works all the time, but the greedy-choice property and optimal substructure are the two key ingredients.

If we can demonstrate that the problem has these properties, then we are well on the way to developing a greedy algorithm for it.

The first key ingredient is the **greedy-choice property**: we can assemble a globally optimal solution by making locally optimal (greedy) choices. In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems.

Here is where greedy algorithms differ from dynamic programming.

In dynamic programming, we make a choice at each step, but the choice usually depends on the solutions to subproblems.

Consequently, we typically solve dynamic-programming problems in a bottom-up manner, progressing from smaller subproblems to larger subproblems.

In a greedy algorithm, we make whatever choice seems best at the moment and then solve the subproblem that remains.

The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems.

Therefore, unlike dynamic programming, which solves the subproblems before making the first choice, a greedy algorithm makes its first choice before solving any subproblems.

A dynamic-programming algorithm proceeds bottom up, whereas a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, reducing each given problem instance to a smaller one.

We can usually make the greedy choice more efficiently than when we have to consider a wider set of choices.

For example, in the activity-selection problem, assuming that we had already sorted the activities in monotonically increasing order of finish times, we needed to examine each activity just once.

By preprocessing the input or by using an appropriate data structure (often a priority queue), we often can make greedy choices quickly, thus yielding an efficient algorithm.

Optimal substructure

A problem exhibits **optimal substructure** if an optimal solution to the problem contains within it optimal solutions to subproblems.

This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms.

As an example of optimal substructure, recall how we demonstrated in the activity-selection problem that if an optimal solution to subproblem S_{ij} includes an activity a_k, then it must also contain optimal solutions to the subproblems S_{ik} and S_{kj}.

Given this optimal substructure, we argued that if we knew which activity to use as a_k, we could construct an optimal solution to S_{ij} by selecting a_k along with all activities in optimal solutions to the subproblems S_{ik} and S_{kj}.

Based on this observation of optimal substructure, we were able to devise the recurrence $c[i, j]$ that described the value of an optimal solution.

Greedy versus dynamic programming

Because both the greedy and dynamic-programming strategies exploit optimal substructure, you might be tempted to generate a dynamic programming solution to a problem when a greedy solution suffices.

Conversely, you might mistakenly think that a greedy solution works when in fact a dynamic-programming solution is required.

Understanding the subtleties between the two techniques is also an active research topic in the design and analysis of algorithms.

6. Graph Algorithms

Graph Algorithms

Graph problems pervade computer science, and algorithms for working with them are fundamental to the field.

Hundreds of interesting computational problems are studied in terms of graphs. In this and the next section, we touch on a few of the more significant ones.

We will show how one can represent a graph in a computer before discussing algorithms based on searching a graph using either breadth-first search or depth-first search.

We will then discuss how to compute a minimum-weight spanning tree of a graph: the least-weight way of connecting all of the vertices together when each edge has an associated weight.

When we characterize the running time of a graph algorithm on a given graph $G = (V, E)$, we usually measure the size of the input in terms of the number of vertices $|V|$ and the number of edges $|E|$ of the graph. That is, we describe the size of the input with two parameters, not just one.

Inside asymptotic notation (such as O-notation or Θ-notation), the symbol V denotes $|V|$ and the symbol E denotes $|E|$.

For example, we might say, the algorithm runs in time $O(VE)$ meaning that the algorithm runs in time $O(|V||E|)$. This convention makes the running-time formulas easier to read, without risk of ambiguity.

Another convention we adopt appears in pseudocode. We denote the vertex set of a graph G by $G.V$ and its edge set by $G.E$. That is, the pseudocode views vertex and edge sets as attributes of a graph.

6.1 Representations of graphs

We can choose between two standard ways to represent a graph $G = (V, E)$ as a collection of adjacency lists or as an adjacency matrix. Either way applies to both directed and undirected graphs.

Because the adjacency-list representation provides a compact way to represent **sparse** graphs - those for which $|E|$ is much less than $|V|^2$ it is usually the method of choice. Most of the graph algorithms presented in this course assume that an input graph is represented in adjacency list form.

We may prefer an adjacency-matrix representation, however, when the graph is **dense** $|E|$ is close to $|V|^2$ or when we need to be able to tell quickly if there is an edge connecting two given vertices.

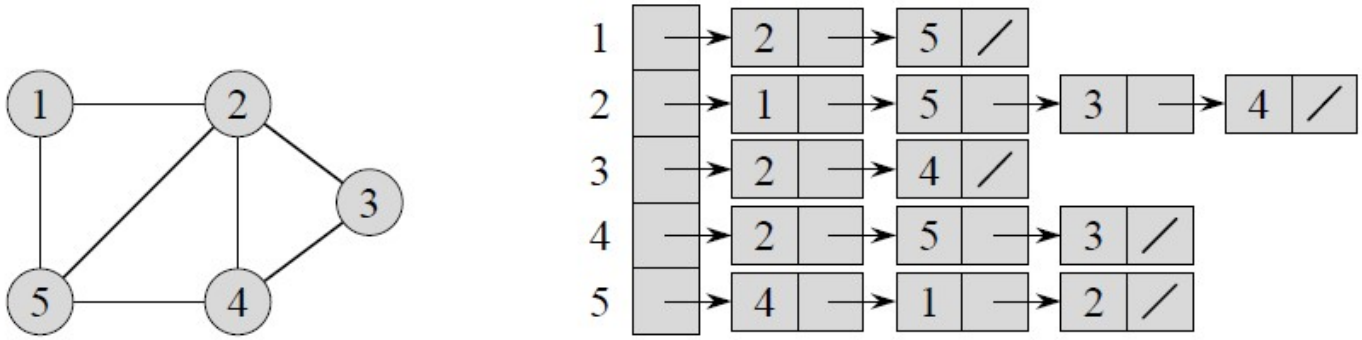
The **adjacency-list representation** of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V.

For each $u \in V$, the adjacency list $Adj[u]$ contains all the vertices such that there is an edge $(u, v) \in E$. That is, $Adj[u]$ consists of all the vertices adjacent to u in G.

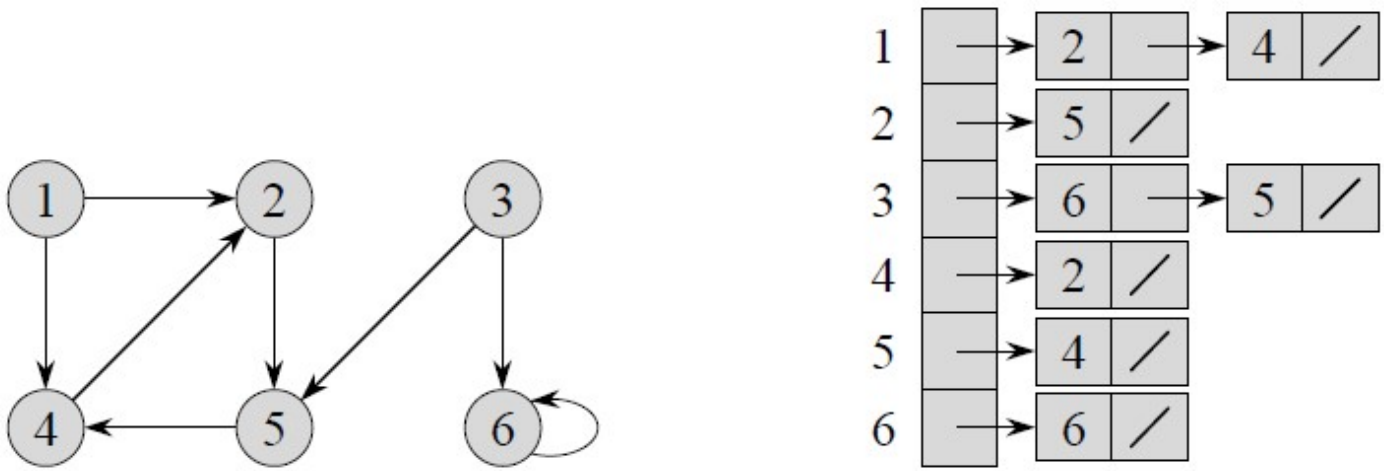
Since the adjacency lists represent the edges of a graph, in pseudocode we treat the array Adj as an attribute of the graph $G.Adj[u]$.

We show in the next page adjacency-list representations of an undirected graph and a directed graph.

Adjacency-list representation of an undirected graph G with 5 vertices and 7 edges:



Adjacency-list representation of a directed graph G with 6 vertices and 8 edges:



If G 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 $Adj[u]$.

If G is an undirected graph, the sum of the lengths of all the adjacency lists is $2|E|$, since if (u, v) is an undirected edge, then u appears in v's adjacency list and vice versa.

For both directed and undirected graphs, the adjacency-list representation has the desirable property that the amount of memory it requires is $\Theta(V + E)$.

We can readily adapt adjacency lists to represent **weighted graphs**, that is, graphs for which each edge has an associated weight, typically given by a weight function $w : E \to \mathbb{R}$.

Let $G = (V, E)$ be a weighted graph with weight function w. We simply store the weight $w(u, v)$ of the edge $(u, v) \in E$ with vertex in u's adjacency list.

The adjacency-list representation is quite robust in that we can modify it to support many other graph variants.

A potential disadvantage of the adjacency-list representation is that it provides no quicker way to determine whether a given edge (u, v) is present in the graph than to search for in the adjacency list $Adj[u]$.

An adjacency-matrix representation of the graph remedies this disadvantage.

For the **adjacency-matrix representation** of a graph $G = (V, E)$, we assume that the vertices are numbered $1, 2, \ldots, |V|$ in some arbitrary manner.

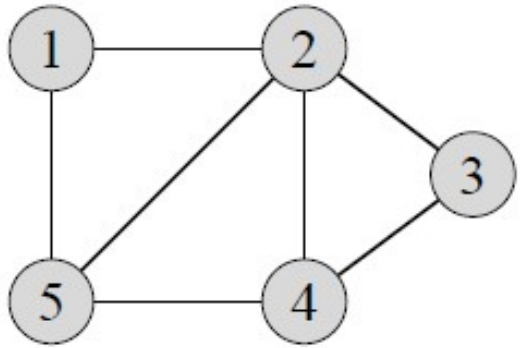
Then the adjacency-matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

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

The adjacency matrix of a graph requires $\Theta(V^2)$ memory, independent of the number of edges in the graph.

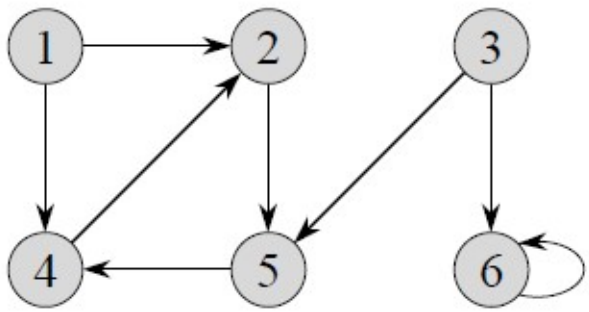
We show in the next page adjacency matrices of an undirected graph and a directed graph.

Adjacency matrix of an undirected graph G with 5 vertices and 7 edges:



	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

Adjacency matrix of a directed graph G with 6 vertices and 8 edges:



	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

Observe the symmetry along the main diagonal of the adjacency matrix of the undirected graph. This is because (u, v) and (v, u) represent the same edge so that the adjacency matrix A of an undirected graph is its own transpose: $A = A^T$.

Like the adjacency-list representation of a graph, an adjacency matrix can represent a weighted graph.

If $G = (V, E)$ is a weighted graph with edge-weight function w, we can simply store the weight $w(u, v)$ of the edge $(u, v) \in E$ as the entry in row u and column v of the adjacency matrix.

If an edge does not exist, we can store a NIL value as its corresponding matrix entry.

Representing attributes

Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges.

We indicate these attributes using our usual notation, such as $v.d$ for an attribute d of a vertex v.

When we indicate edges as pairs of vertices, we use the same style of notation. For example, if edges have an attribute f, then we denote this attribute for edge (u, v) by $(u, v).f$.

Implementing vertex and edge attributes in real programs can be another story entirely. For a given situation, your decision will likely depend on the programming language you are using, the algorithm you are implementing, and how the rest of your program uses the graph.

6.2 Breadth-first search

Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms.

Prim's minimum-spanning-tree algorithm and Dijkstra's single-source shortest-paths algorithm use ideas similar to those in breadth-first search.

Given a graph $G = (V, E)$ and a distinguished source vertex s, breadth-first search systematically explores the edges of G to discover every vertex that is reachable from s.

It computes the distance (smallest number of edges) from s to each reachable vertex.

It also produces a breadth-first tree with root s that contains all reachable vertices.

For any vertex reachable from s, the simple path in the breadth-first tree from s to v corresponds to a shortest path from s to v in G, that is, a path containing the smallest number of edges.

The algorithm works on both directed and undirected graphs.

Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$.

To keep track of progress, breadth-first search colors each vertex white, gray, or black. All vertices start out white and may later become gray and then black.

A vertex is **discovered** the first time it is encountered during the search, at which time it becomes nonwhite.

Gray and black vertices, therefore, have been discovered, but breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner

If $(u, v) \in E$ and vertex u is black, then vertex v is either gray or black; that is, all vertices adjacent to black vertices have been discovered.

Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.

Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex s.

Whenever the search discovers a white vertex v in the course of scanning the adjacency list of an already discovered vertex u, the vertex v and the edge (u, v) are added to the tree.

We say that u is the **predecessor** or **parent** of v in the breadth-first tree. Since a vertex is discovered at most once, it has at most one parent.

Ancestor and descendant relationships in the breadth-first tree are defined relative to the root s as usual: if u is on the simple path in the tree from the root s to vertex v, then u is an ancestor of v and v is a descendant of u.

The breadth-first-search algorithm BFS in the next page assumes that the input graph $G = (V, E)$ is represented using adjacency lists.

It attaches several additional attributes to each vertex in the graph. We store the color of each vertex $u \in V$ in the attribute $u.color$ and the predecessor of u in the attribute $u.\pi$.

If u has no predecessor (if $u = s$ or u has not been discovered), then $u.\pi =$NIL.

The attribute $u.d$ holds the distance from the source s to vertex u computed by the algorithm.

The algorithm also uses a first-in, first-out queue Q to manage the set of gray vertices.

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each v ∈ G.Adj[u]
13            if v.color == WHITE
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)
18       u.color = BLACK
```

The procedure BFS works as follows.

With the exception of the source vertex s, lines $1-4$ paint every vertex white, set $u.d$ to be infinity for each vertex u, and set the parent of every vertex to be NIL.

Line 5 paints s gray, since we consider it to be discovered as the procedure begins.

Line 6 initializes $s.d$ to 0, and line 7 sets the predecessor of the source to be NIL.

Lines $8-9$ initialize Q to the queue containing just the vertex s.

The **while** loop of lines $10 - 18$ iterates as long as there remain gray vertices, which are discovered vertices that have not yet had their adjacency lists fully examined.

This **while** loop maintains the following invariant.

At the test in line 10, the queue Q consists of the set of gray vertices. Prior to the first iteration, the only gray vertex, and the only vertex in Q, is the source vertex s.

Line 11 determines the gray vertex u at the head of the queue Q and removes it from Q.

The **for** loop of lines $12 - 17$ considers each vertex v in the adjacency list of u.

If v is white, then it has not yet been discovered, and the procedure discovers it by executing lines $14 - 17$.

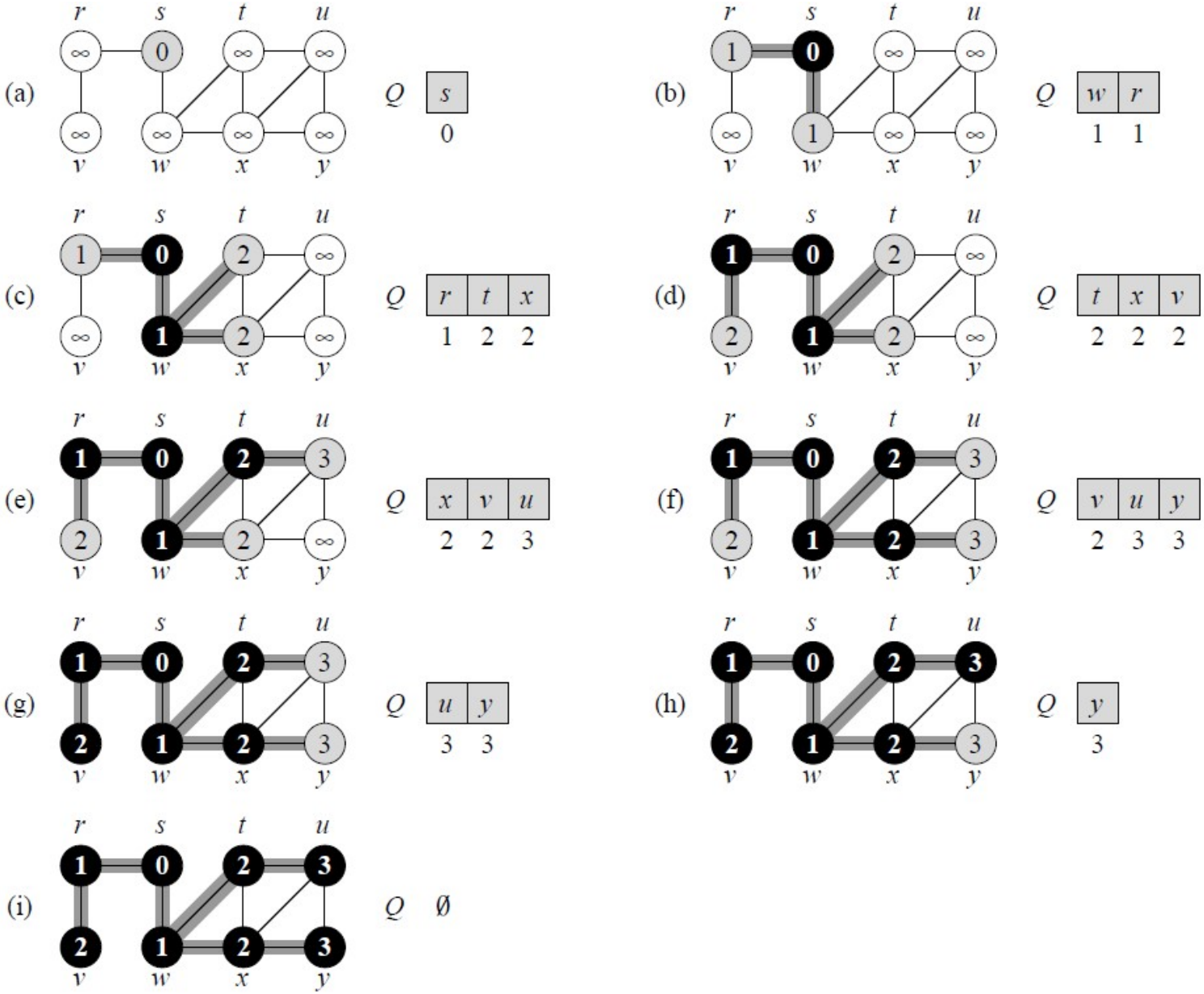
The procedure paints vertex v gray, sets its distance $v.d$ to $u.d + 1$, records u as its parent $v.\pi$, and places it at the tail of the queue Q.

Once the procedure has examined all the vertices on u's adjacency list, it blackens u in line 18.

The loop invariant is maintained because whenever a vertex is painted gray (in line 14) it is also enqueued (in line 17), and whenever a vertex is dequeued (in line 11) it is also painted black (in line 18).

The results of breadth-first search may depend upon the order in which the neighbors of a given vertex are visited in line 12: the breadth-first tree may vary, but the distances d computed by the algorithm will not.

The operation of BFS on an undirected graph. The queue Q is shown at the beginning of each iteration of the while loop. Vertex distances appear below vertices in the queue.



Analysis

After initialization, breadth-first search never whitens a vertex, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once. Thus, the total time devoted to queue operations is $O(V)$.

Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all the adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(E)$.

Therefore, the total running time of the BFS procedure is $O(V + E)$ - the breadth-first search runs in time linear in the size of the adjacency-list representation of G.

6.3 Depth-first search

The strategy followed by depth-first search is, as its name implies, to search deeper in the graph whenever possible.

Depth-first search explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it.

Once all of v's edges have been explored, the search backtracks to explore edges leaving the vertex from which was discovered.

This process continues until we have discovered all the vertices that are reachable from the original source vertex.

If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex.

As in breadth-first search, whenever depth-first search discovers a vertex v during a scan of the adjacency list of an already discovered vertex u, it records this event by setting v's predecessor attribute $v.\pi$ to u.

Unlike breadth-first search, whose predecessor subgraph forms a tree, the predecessor subgraph produced by a depth-first search may be composed of several trees, because the search may repeat from multiple sources.

Therefore, we define the **predecessor subgraph** of a depth-first search slightly differently from that of a breadth-first search: we let

$$\begin{aligned} G_\pi &= (V, E_\pi) \\ E_\pi &= \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq NIL\}. \end{aligned}$$

The predecessor subgraph of a depth-first search forms a **depth-first forest** comprising several **depth-first trees**. The edges in E_π are **tree edges**.

As in breadth-first search, depth-first search colors vertices during the search to indicate their state.

Each vertex is initially white, is grayed when it is **discovered** in the search, and is blackened when it is **finished**, that is, when its adjacency list has been examined completely.

This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

Besides creating a depth-first forest, depth-first search also **timestamps** each vertex.

Each vertex v has two timestamps: the first timestamp $v.d$ records when v is first discovered (and grayed), and the second timestamp $v.f$ records when the search finishes examining v's adjacency list (and blackens).

These timestamps provide important information about the structure of the graph and are generally helpful in reasoning about the behavior of depth-first search.

Timestamps are integers between 1 and $2|V|$, since there is one discovery event and one finishing event for each of the $|V|$ vertices.

Vertex u is WHITE before time $u.d$, GRAY between time $u.d$ and time $u.f$, and BLACK thereafter.

The following pseudocode is the basic depth-first-search algorithm. The input graph *G* may be undirected or directed.

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
 1  time = time + 1                  // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]            // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK                  // blacken u; it is finished
 9  time = time + 1
10  u.f = time
```

Procedure DFS works as follows.

Lines $1-3$ paint all vertices white and initialize their π attributes to NIL.

Line 4 resets the global time counter.

Lines $5-7$ check each vertex in V in turn and, when a white vertex is found, visit it using DFS-VISIT.

Every time DFS-VISIT(G, u) is called in line 7, vertex u becomes the root of a new tree in the depth-first forest.

When DFS returns, every vertex u has been assigned a discovery time $u.d$ and a finishing time $u.f$.

In each call DFS-VISIT(G, u), vertex u is initially white. Line 1 increments the global variable time, line 2 records the new value of time as the discovery time $u.d$, and line 3 paints u gray.

Lines $4 - 7$ examine each vertex v adjacent to u and recursively visit if it is white.

As each vertex $v \in Adj[u]$ is considered in line 4, we say that edge (u, v) is explored by the depth-first search.

Finally, after every edge leaving u has been explored, lines $8 - 10$ paint u black, increment time, and record the finishing time in $u.f$.

Note that the results of depth-first search may depend upon the order in which line 5 of DFS examines the vertices and upon the order in which line 4 of DFS-VISIT visits the neighbors of a vertex.

These different visitation orders tend not to cause problems in practice, as we can usually use any depth-first search result effectively, with essentially equivalent results.

Classification of edges

Another interesting property of depth-first search is that the search can be used to classify the edges of the input graph $G = (V, E)$. The type of each edge can provide important information about a graph.

We can define four edge types in terms of the depth-first forest G_π produced by a depth-first search on G:

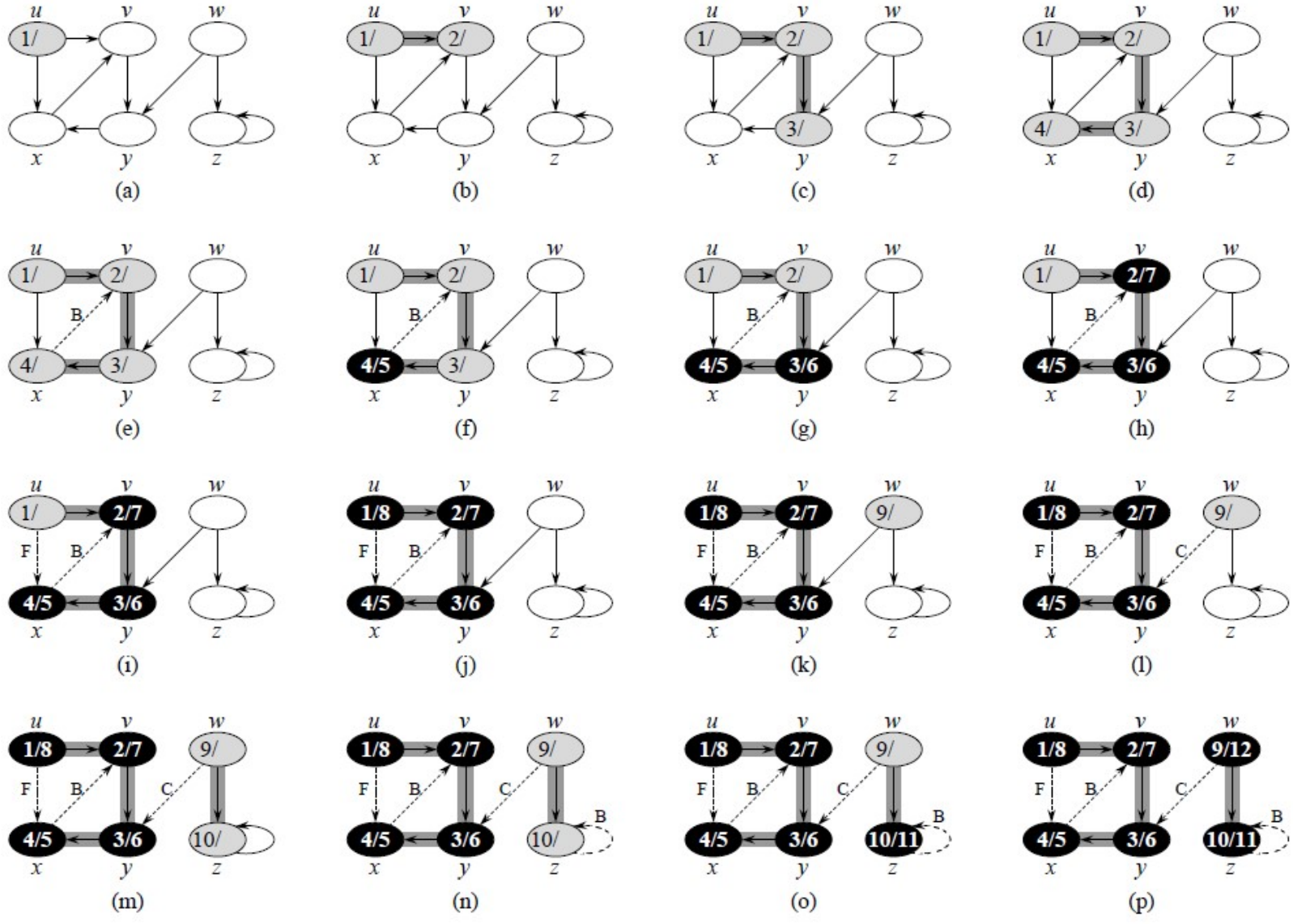
1. **Tree edges** are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).

2. **Back edges** are those edges (u, v) connecting a vertex u to an ancestor in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.

3. **Forward edges** are those nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.

4. **Cross edges** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled *B*, *C*, or *F* according to whether they are back, cross, or forward edges. Timestamps within vertices indicate discovery time/finishing times.



Analysis

The loops on lines $1-3$ and lines $5-7$ of DFS take time $\Theta(V)$, exclusive of the time to execute the calls to DFS-VISIT.

The procedure DFS-VISIT is called exactly once for each vertex $v \in V$, since the vertex u on which DFS-VISIT is invoked must be white and the first thing DFS-VISIT does is paint vertex u gray.

During an execution of DFS-VISIT(G, v), the loop on lines $4-7$ executes $|Adj[v]|$. Since $\sum_{v \in V} |Adj[v]| = \Theta(E)$, the total cost of executing lines $4-7$ of DFS-VISIT is $\Theta(E)$.

The running time of DFS is therefore $\Theta(V + E)$.

6.4 Minimum spanning tree

Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together. To interconnect a set of n pins, we can use an arrangement of $n - 1$ wires, each connecting two pins.

Of all such arrangements, the one that uses the least amount of wire is usually the most desirable.

We can model this wiring problem with a connected, undirected graph $G = (V, E)$, where V is the set of pins, E is the set of possible interconnections between pairs of pins, and for each edge $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost (amount of wire needed) to connect u and v.

We then wish to find an acyclic subset $T \subset E$ that connects all of the vertices and whose total weight

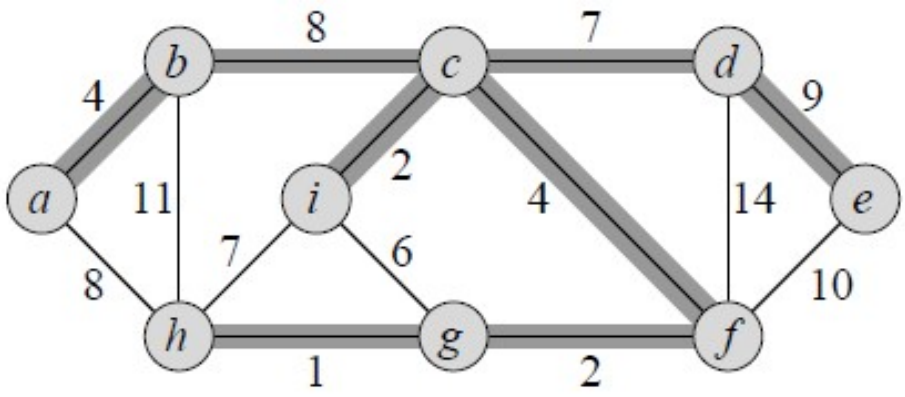
$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

is minimized.

Since T is acyclic and connects all of the vertices, it must form a tree, which we call a **spanning tree** since it spans the graph G.

We call the problem of determining the tree T the **minimum-spanning-tree problem**.

An example of a connected graph and a minimum spanning tree



The weights on edges are shown, and the edges in a minimum spanning tree are shaded. The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another spanning tree with weight 37.

We now introduces a generic minimum-spanning-tree method that grows a spanning tree by adding one edge at a time.

Assume that we have a connected, undirected graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$, and we wish to find a minimum spanning tree for G.

The generic method is based on a greedy strategy, which grows the minimum spanning tree one edge at a time.

The method manages a set of edges A, maintaining the loop invariant: Prior to each iteration, A is a subset of some minimum spanning tree.

At each step, we determine an edge (u, v) that we can add to A without violating this invariant, in the sense that $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree.

We call such an edge a **safe edge** for A, since we can add it safely to A while maintaining the invariant.

A pseudocode:

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

We use the loop invariant as follows:

Initialization: After line 1, the set A trivially satisfies the loop invariant.

Maintenance: The loop in lines $2-4$ maintains the invariant by adding only safe edges.

Termination: All edges added to A are in a minimum spanning tree, and so the set A returned in line 5 must be a minimum spanning tree.

The key part is, of course, finding a safe edge in line 3.

One must exist, since when line 3 is executed, the invariant dictates that there is a spanning tree T such that $A \subseteq T$.

Within the **while** loop body, A must be a proper subset of T, and therefore there must be an edge $(u, v) \in T$ such that $(u, v) \notin A$ and (u, v) is safe for A.

Before providing a rule for recognizing safe edges, we first need some definitions.

A **cut** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V.

We say that an edge $(u, v) \in E$ **crosses** the cut $(S, V - S)$ if one of its endpoints is in S and the other is in $V - S$.

We say that a cut **respects** a set A of edges if no edge in A crosses the cut.

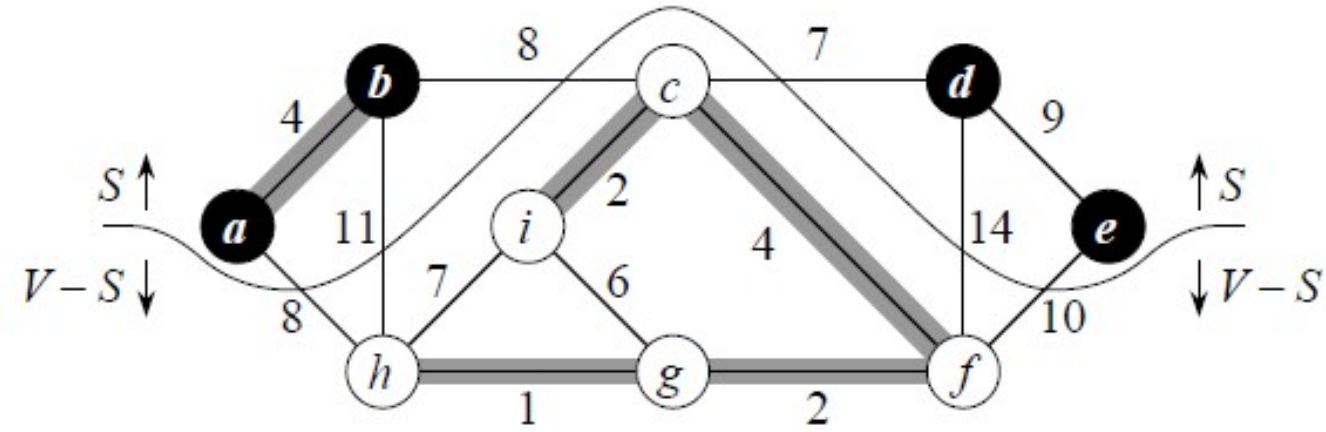
An edge is a **light edge** crossing a cut if its weight is the minimum of any edge crossing the cut.

Note that there can be more than one light edge crossing a cut in the case of ties.

More generally, we say that an edge is a light edge satisfying a given property if its weight is the minimum of any edge satisfying the property.

Safe edges rule:
Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any 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.

Example:



Black vertices are in the set S, and white vertices are in $V - S$. The edges crossing the cut are those connecting white vertices with black vertices. The edge (d, c) is the unique light edge crossing the cut. A subset A of the edges is shaded; note that the cut $(S, V - S)$ respects A, since no edge of A crosses the cut.

This rule gives us a better understanding of the workings of the GENERIC-MST algorithm on a connected graph $G = (V, E)$.

As the method proceeds, the set A is always acyclic; otherwise, a minimum spanning tree including A would contain a cycle, which is a contradiction.

At any point in the execution, the graph $G_A = (V, A)$ is a forest, and each of the connected components of G_A is a tree.

Moreover, any safe edge (u, v) for A connects distinct components of G_A, since $A \cup \{(u, v)\}$ must be acyclic.

The **while** loop in lines $2 - 4$ of GENERIC-MST executes $|V| - 1$ times because it finds one of the $|V| - 1$ edges of a minimum spanning tree in each iteration.