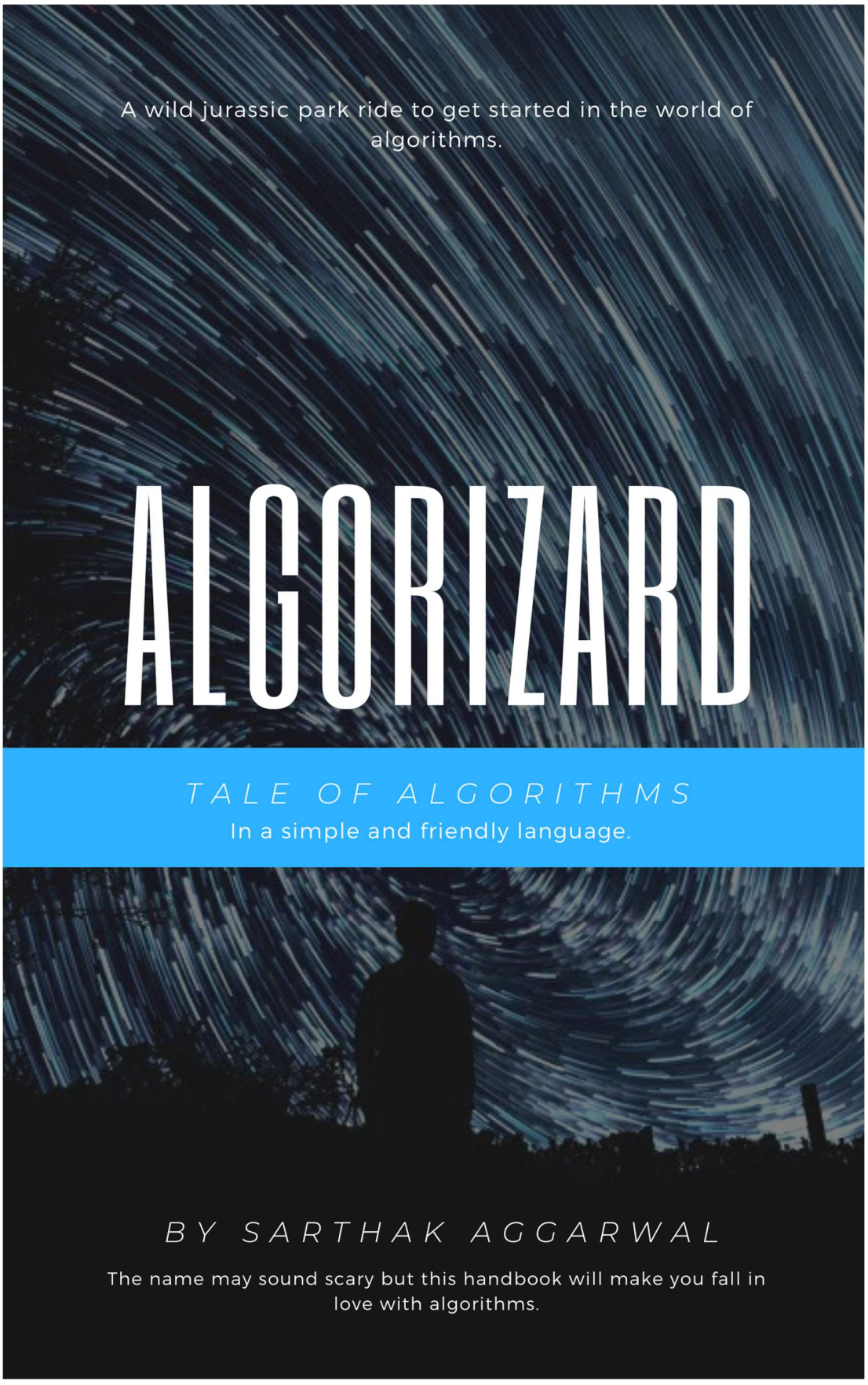

A wild jurassic park ride to get started in the world of algorithms.
ALGORIZARD
TALE OF ALGORITHMS
In a simple and friendly language.
BY SARTHAK AGGARWAL
The name may sound scary but this handbook will make you fall in love with algorithms.

Table of Contents

I) Asymptotic Notation 4
- Big-O notation 5
- Big-Omega notation 5
- Big-Theta notation 6
- Little-O notation 7

II) Recurrences and Its Accomplices 8
- Recursion Trees 8
- Master Method 9

III) Design Techniques 13
- Divide and Conquer 13
- Dynamic Programming 18
- Greedy Algorithms 27
- Huffman Coding 29

IV) Searching 34
- Sequential Search 34
- Binary Search 35
- Search Trees 38
- Balanced Search Trees 52

V) Prioritizing 57
- Heaps 59
- Heapsort 62
- Heap as Trees 64
- Heap as Arrays 66

VI) Sorting 75
- Bubble Sort 75
- Quick Sort 79
- Merge Sort 85
- Insertion Sort 94
- Selection Sort 98

VII) Graph Algorithms 102
- Graph Search 102
- Shortest Paths 109
- Minimum Spanning Trees 117
- Union-Find 122

VIII) Geometric Algorithms 129
- Plane-Sweep 129
- Delaunay Triangulations 134
- Alpha Shapes 140

IX) NP-Completeness 145
- Easy and Hard Problems 145
- NP-Complete problems 150
- Approximation Algorithms 155

Chapter 1

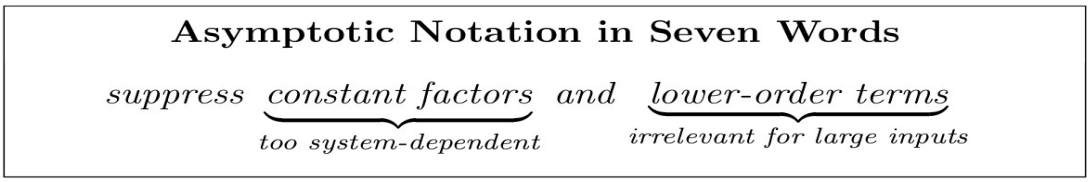
Asymptotic Notation

Motivation:

Asymptotic notation provides the basic vocabulary for discussing the design and analysis of algorithms. It's important that you know what programmers mean when they say that one piece of code runs in "big-O of n time," while other runs in "big-O of n-squared time."

Asymptotic notation is coarse enough to suppress all the details you want to ignore – details that depend on the choice of architecture, the choice of programming language, the choice of compiler and so on.

Asymptotic analysis helps us differentiate between better and worse approaches to sorting, better and worse approaches to multiplying two integers and so on.



Low-order terms, by definition, becomes increasingly irrelevant as you focus on large inputs, which are the inputs that require algorithmic ingenuity.

The constant factors are generally highly dependent on the details of the environment.

For example, analysis of Merge Sort gave an upper bound on its running time of *6nlog2n + 6n*

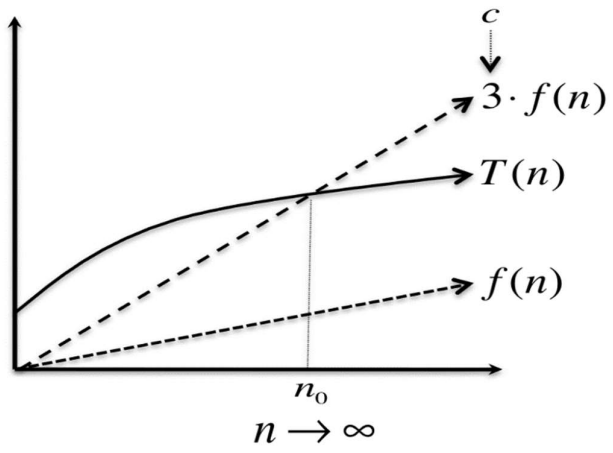
primitive operations, where *n* is the length of the input array. The lower-order term here is the *6n*, as *n* grows more slowly than $nlog_2n$, so it will be suppressed in asymptotic notation. The leading constant factor of *6* also gets suppressed, leaving us with the much simpler expression of *nlogn*. We would then say that the running time of Merge-Sort is *"big-O of nlogn,"* written *O(nlogn)*.

Big-O notation (English definition):

T(n) = O(f(n)) if and only if T (n) is eventually bounded above by a constant multiple of *f(n).*

Mathematical definition: -

T(n) = O(f(n)) if and only if there exist positive constants *c* and n_0 such that $T(n) \leq c.f(n)$ for all $n \geq n_0$.



A picture illustrating when *T (n) = O(f(n))*. The constant *c* quantifies the "constant multiple" of *f(n)*, and the constant n_0 quantifies "eventually."

Example of functions having $O(n^2)$:-

n^2

$1000n^2 + 1000n$

$n^{1.99999}$

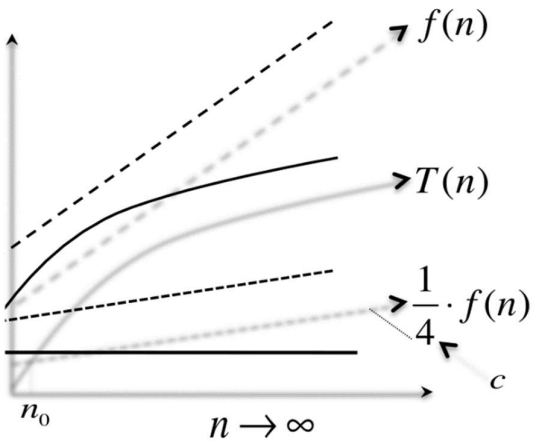
Big- Omega notation(english definition):

one function *T(n)* is big-omega of another function *f(n)* if and only if *T(n)* is eventually bounded below by a constant multiple of *f(n)*.

In this case, we write $T(n) = \Omega(f(n))$. As before, we use two constants *c* and n_0 to quantify "constant multiple" and "eventually."

Mathematical definition:

T(n) = Ω(f(n)) if and only if there exist positive constants *c* and n_0 such that *T(n) ≥ c.f(n)* for all $n \geq n_0$.



Example of functions in $\Omega(n^2)$:

n^3

$n^2 + n$

$n^{2.00001}$

Big-Theta notation

Big-theta notation, or simply theta notation, is analogous to "equal to." Saying that T (n) =Θ(f (n)) just means that both T (n) = Ω (f (n)) and T (n) = O(f (n)).T-

Mathematical version:

T (n) =Θ(f (n)) if and only if there exist positive constants c_1,c_2, and n_0 such that

$c_1.f(n) \leq T(n) \leq c_2.f(n)$ for all $n \geq n_0$.

$N^2/2 - 2n = \Theta(n^2)$ with $c_1 = 1/4$ $c_2 = 1/2$ and $n_0 = 8$

Little –O notation:-

If big-O notation is analogous to “less than or equal to,” little-o notation is analogous to “strictly less

Than.”

Mathematical version:

T (n) = o(f (n)) if and only if for every positive constant c > 0, there exists a choice of n_0 such that

$T(n) < c.f(n)$ for all $n \geq n_0$.

Examples: -

$n^{1.9999} = o(n^2)$

$n^2 \neq o(n^2)$

Question:

Let $T(n) = \frac{1}{2}n^2 + 3n$. Which of the following statements are true? (More than one correct answer.)

1. $T(n) = O(n)$
2. $T(n) = \Omega(n)$
3. $T(n) = \Theta(n^2)$
4. $T(n) = O(n^3)$

The correct answers are (2), (3), and (4).

Explanation:

T(n) is a quadratic function. The linear term 3n doesn’t matter for large n, so we should expect that $T(n) = \Theta(n^2)$ (answer (3)).

This automatically implies that $T(n) = \Omega(n^2)$ and hence $T(n) = \Omega(n)$ also (answer (2)).

Similarly, $T(n) = \Theta(n^2)$ implies that $T(n) = O(n^2)$ and hence also $T(n) = O(n^3)$ (answer (4)).

CHAPTER 2

Recurrences and Its Accomplices

Recursion is a particularly powerful kind of reduction, which can be described loosely as follows:

- If the given instance of the problem is small or simple enough, just solve it.
- Otherwise, reduce the problem to one or more simpler instances of the same problem.

Recursion is generally expressed in terms of recurrences. In other words, when an algorithm calls to itself, we can often describe its running time by a recurrence equation which describes the overall running time of a problem of size n in terms of the running time on smaller inputs.

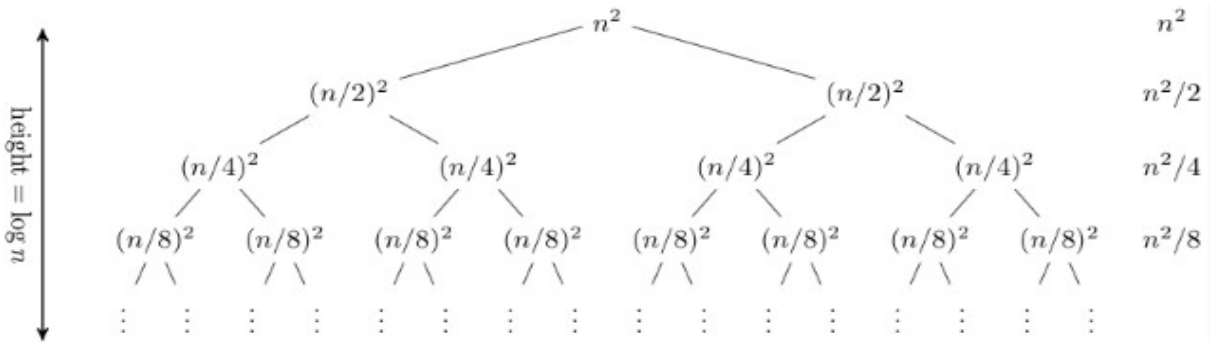
Recursion trees:

A *recursion tree* is useful for visualizing what happens when a recurrence is iterated. It diagrams the tree of recursive calls and the amount of work done at each call.

Recursion trees can be useful for gaining intuition about the closed form of a recurrence.

Consider the following recurrence $T(n) = 2T(n/2) + n^2$.

The following is the recursion tree for the above recurrence;-



This a geometric series, thus in the limit the sum is $O(n^2)$. The depth of the tree in this case does not really matter; the amount of work at each level is decreasing so quickly that the total is only a constant factor more than the root.

Master method:

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

n = size of input a = number of subproblems in the recursionn/b = size of each subproblem. All subproblems are assumed to have the same size.

f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

Here, a ≥ 1 and b > 1 are constants, and f(n) is an asymptotically positive function.

Formula:

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

where, T(n) has the following asymptotic bounds:(ϵ > 0 is a constant)

1. If $f(n) = O(n^{\log_b a-\epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.

If the cost of solving the sub-problems at each level increases by a certain factor, the value of $f(n)$ will become polynomially smaller than $n^{\log_b a}$. Thus, the time complexity is oppressed by the cost of the last level i.e. $n^{\log_b a}$

2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} * \log n)$.

If the cost of solving the sub-problem at each level is nearly equal, then the value of $f(n)$ will be $n^{\log_b a}$. Thus, the time complexity will be $f(n)$ times the total number of levels ie. $n^{\log_b a} * \log n$

3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$, then $T(n) = \Theta(f(n))$.

If the cost of solving the subproblems at each level decreases by a certain factor, the value of $f(n)$ will become polynomially larger than $n^{\log_b a}$. Thus, the time complexity is oppressed by the cost of $f(n)$.

Example 1:

Solve the following recurrence relation using Master's theorem-

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

Solution:

We compare the given recurrence relation with $T(n) = aT(n/b) + \theta (n^k \log^p n)$.

Then, we have-

$$a = 3$$

$$b = 2$$

$$k = 2$$

$$p = 0$$

Now, $a = 3$ and $b^k = 2^2 = 4$.

Clearly, $a < b^k$.

So, we follow case-03.

Since $p = 0$, so we have-

$$T(n) = \theta (n^k \log^p n)$$

$$T(n) = \theta (n^2 \log^0 n)$$

Thus,

$$\boxed{\mathbf{T(n) = \theta (n^2)}}$$

Example 2:

Solve the following recurrence relation using Master's theorem-

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

Solution:

We compare the given recurrence relation with $T(n) = aT(n/b) + \theta (n^k \log^p n)$.

Then, we have-

$$a = 2$$

$$b = 2$$

$$k = 1$$

$$p = 1$$

Now, $a = 2$ and $b^k = 2^1 = 2$.

Clearly, $a = b^k$.

So, we follow case-02.

Since $p = 1$, so we have-

$$T(n) = \theta\ (n^{\log_b a}.\log^{p+1} n)$$

$$T(n) = \theta\ (n^{\log_2 2}.\log^{1+1} n)$$

Thus,

$$\mathbf{T(n) = \theta\ (n\log^2 n)}$$

Example 3:

Solve the following recurrence relation using Master's theorem-

$$T(n) = \sqrt{2}T(n/2) + \log n$$

Solution:

We compare the given recurrence relation with $T(n) = aT(n/b) + \theta\ (n^k \log^p n)$.

Then, we have-

$$a = \sqrt{2}$$

$$b = 2$$

$$k = 0$$

$$p = 1$$

Now, $a = \sqrt{2} = 1.414$ and $b^k = 2^0 = 1$.

Clearly, $a > b^k$.

So, we follow case-01.

So, we have-

$$T(n) = \theta\ (n^{\log_b a})$$

$$T(n) = \theta\ (n^{\log_2 \sqrt{2}})$$

$$T(n) = \theta\ (n^{1/2})$$

Thus,

$$\mathbf{T(n) = \theta\ (\sqrt{n})}$$

“Seems attractive and limitless isnt it?

but every master have limitations.”

Limitations:

The master theorem cannot be used if:

- T(n) is not monotone. eg. T(n) = sin n
- f(n) is not a polynomial. eg. $f(n) = 2^n$
- a is not a constant. eg. a = 2n
- a < 1

CHAPTER 3

DESIGN TECHNIQUES

Divide and conquer

The paradigm:-

1. Divide the input into smaller subproblems.
2. Conquer the subproblems recursively.
3. Combine the solutions for the subproblems into a solution for the original problem.

We use quicksort as an example for an algorithm that follows the divide-and-conquer paradigm. It has the reputation of being the fasted comparison-based sorting algorithm.

The algorithm:

```
void QUICKSORT(int ℓ, r)

if ℓ < r then m = SPLIT(ℓ, r);

        QUICKSORT(ℓ, m − 1);

        QUICKSORT(m + 1, r)

Endif.
```

We assume the items are stored in A[0..n − 1]. The array is sorted by calling QUICKSORT(0, n − 1).

Splitting:

The performance of quicksort depends heavily on the performance of the split operation. The effect of splitting from ℓ to r is:

• x = A[ℓ] is moved to its correct location at A[m];

• no item in A[ℓ..m − 1] is larger than x;

• no item in A[m + 1..r] is smaller than x.

Figure: First, i and j stop at items 9 and 1, which are then swapped. Second, i and j cross and the pivot, 7, is swapped with item 2.

The above figure illustrates the process with a example. The nine items are split by moving a pointer i from left to right and another pointer j from right to left. The process stops when i and j cross. To get splitting right is a bit delicate, in particular in special cases.

Algorithm for split:

```
int SPLIT(int ℓ, r)
 x = A[ℓ]; i = ℓ; j = r + 1;
 repeat repeat i++ until x ≤ A[i];
        repeat j-- until x ≥ A[j];
        if i < j then SWAP(i, j) endif
 until i ≥ j;
 SWAP(ℓ, j); return j.
```

Running time:

The actions taken by quicksort can be expressed using a binary tree: each (internal) node represents a call and displays the length of the subarray; see Figure below. The worst case occurs when the pivot picked is either the greatest or the smallest element.

The total amount of time is proportional to the sum of lengths, which are the numbers of nodes in the corresponding subtrees. In the case, the sum is 29.

In this case the tree degenerates to a list without branching. The sum of lengths can be described by the following recurrence relation:

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

The running time in the worst case is therefore in $O(n^2)$.

In the best case the tree is completely balanced and the sum of lengths is described by the recurrence relation

$T(n) = n + 2.T\left(\frac{n-1}{2}\right)$.

If we assume $n = 2^k - 1$, we can write the relation as

$U(k) = (2^k - 1) + 2.U(k - 1)$

$= (2^k - 1) + 2(2^{k-1} - 1)+\ldots+ 2^k - 1(2 - 1)$

$= k.2^k - \sum_{i=0}^{k-1} 2^i$

$= 2^k . k - (2^k - 1)$

$= (n+1).\log_2(n+1)-n$

The running time in best case is therefore in O(nlogn).

Randomization:

One of the drawbacks of quicksort, as described until now, is that it is slow on rather common almost sorted sequences. The reason are pivots that tend to create unbalanced splittings. Such pivots tend to occur in practice more often than one might expect. Human and often also machine generated data is frequently biased towards certain distributions (in this case, permutations), and it has been said that 80% of the time or more, sorting is done on either already sorted or almost sorted files. Such situations can often be helped by transferring the algorithm's dependence on the input data to internally made random choices. In this particular case, we use randomization to make the choice of the pivot independent of the input data. Assume RANDOM(ℓ, r) returns an integer p ∈ [ℓ, r] with uniform probability:

$$\text{Prob}[\text{RANDOM}(\ell, r) = p] = \frac{1}{r-l+1}$$

for each ℓ ≤ p ≤ r. In other words, each p ∈ [ℓ, r] is equally likely. The following algorithm splits the array with a random pivot:

```
int RSPLIT(int ℓ, r)
 p = RANDOM(ℓ, r); SWAP(ℓ, p);
  return SPLIT(ℓ, r).
```

We get a randomized implementation by substituting RSPLIT for SPLIT. The behavior of this version of quicksort depends on p, which is produced by a random number generator.

Average analysis:

We assume that the items in A[0..n− 1] are pairwise different. The pivot splits A into

$$A[0..m-1],\ A[m],\ A[m+1..n-1].$$

By assumption on function RSPLIT, the probability for each m ∈ [0, n − 1] is 1 n . Therefore the average sum of array lengths split by QUICKSORT is

$T(n) = n + \frac{1}{n}.\sum_{m=0}^{n-1}\big(T(m) + T(n-m-1)\big)$.

To simplify, we multiply with n and obtain a second relation by substituting n − 1 for n:

$n.T(n) = n^2 + 2.\sum_{i=0}^{n-1} T(i)$, (1)

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

Next we subtract (2) from (1), we divide by n(n + 1), we use repeated substitution to express T (n) as a sum, and finally split the sum in two:

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

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

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

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

Bounding by sums:

The second sum is solved directly by transformation to a telescoping series:

$\sum_{i=1}^{n} \frac{1}{i(i+1)} = \sum_{i=1}^{n} (\frac{1}{i} - \frac{1}{i+1})$

$= 1 - \frac{1}{n+1}$

The first sum is bounded from above by the integral of 1 x for x ranging from 1 to n + 1; see Figure 3. The sum of $\frac{1}{i+1}$ is the sum of areas of the shaded rectangles, and because all rectangles lie below the graph of $\frac{1}{x}$ we get a bound for the total rectangle area:

$$\sum_{i=1}^{n} \frac{1}{i+1} < \int_{1}^{n+1} \frac{dx}{x} = \ln(n+1)$$

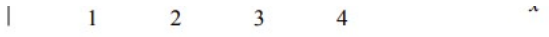


Figure: the areas of the rectangles are the terms in the sum, and the total rectangle area is bounded by the integral from 1 through n + 1.

We plug this bound back into the expression for the average running time:

$T(n) < (n+1).\sum_{i=1}^{n} \frac{2}{i+1}$

$<2.(n+1).\ln(n+1)$

$= \frac{2}{\log_2 e}, (n+1).\log_2(n+1).$

In words, the running time of quicksort in the average case is only a factor of about $2/\log_2 e = 1.386 \ldots$ slower than in the best case. This also implies that the worst case cannot happen very often, for else the average performance would be slower.

Stack size:

Another drawback of quicksort is the recursion stack, which can reach a size of Ω(n) entries. This can be improved by always first sorting the smaller side and simultaneously removing the tail-recursion:

```
void QUICKSORT(int ℓ, r)
  i = ℓ; j = r;
while i < j do
 m = RSPLIT(i, j);
if m − i < j − m
  then QUICKSORT(i, m − 1);i = m + 1
  else QUICKSORT(m + 1, j); j = m − 1
 endif
endwhile
```

In each recursive call to QUICKSORT, the length of the array is at most half the length of the array in the preceding call. This implies that at any moment of time the stack contains no more than 1 + log2 n entries. Note that without removal of the tail-recursion, the stack can reach Ω(n) entries even if the smaller side is sorted first.

Summary:

Quicksort incorporates two design techniques to efficiently sort n numbers: divide-and-conquer for reducing large to small problems and randomization for avoiding the sensitivity to worst-case inputs. The average running time of quicksort is in O(n log n) and the extra amount of memory it requires is in O(log n). For the deterministic version, the average is over all n! permutations of the input items. For the randomized version the average is the expected running time for every input sequence.

Questions:

Let α be some constant, independent of the input array length n, strictly between 0 and 12 . What is the probability that, with a randomly chosen pivot element, the Partition subroutine produces a split in which the size of both the resulting subproblems is at least α times the size of the original array?

a. α
b. 1 - α
c. 1 - 2α
d. 2 - 2α

Define the recursion depth of QuickSort as the maximum number of successive recursive calls it makes before hitting the base case—equivalently, the largest level of its recursion tree. In randomized QuickSort, the recursion depth is a random variable, depending on the pivots chosen. What is the minimum- and maximum-possible recursion depth of randomized QuickSort?

a. Minimum: $\theta(1)$; maximum: $\theta(n)$
b. Minimum: $\theta(\log n)$; maximum: $\theta(n)$
c. Minimum: $\theta(\log n)$; maximum: $\theta(n \log n)$
d. Minimum: $\theta(\text{root } n)$; maximum: $\theta(n)$

DYNAMIC PROGRAMMING

Sometimes, divide-and-conquer leads to overlapping subproblems and thus to redundant computations. It is not uncommon that the redundancies accumulate and cause an exponential amount of wasted time. We can avoid the waste using a simple idea: solve each subproblem only once. To be able to do that, we have to add a certain amount of book-keeping to remember subproblems we have already solved. The technical name for this design paradigm is dynamic programming

Let us understand this using a problem

Edit distance problem:

We illustrate dynamic programming using the edit distance problem, which is motivated by questions in genetics. We assume a finite set of characters or letters, Σ, which we refer to as the alphabet, and we consider strings or words formed by concatenating finitely many characters from the alphabet. The edit distance between two words is the minimum number of letter insertions, letter deletions, and letter substitutions required to transform one word to the other. For example, the edit distance between FOOD and MONEY is at most four:

FOOD → MOOD → MOND → MONED → MONEY

A better way to display the editing process is the gap representation that places the words one above the other, with a gap in the first word for every insertion and a gap in the second word for every deletion:

F O O D

M O N E Y

Columns with two different characters correspond to substitutions. The number of editing steps is therefore the number of columns that do not contain the same character twice.

Prefix property. It is not difficult to see that you cannot get from FOOD to MONEY in less than four steps. However, for longer examples it seems considerably more difficult to find the minimum number of steps or to recognize an optimal edit sequence. Consider for example

A L G O R I T H M

A L T R U I S T I C

Is this optimal or, equivalently, is the edit distance between ALGORITHM and ALTRUISTIC six? Instead of answering this specific question, we develop a dynamic programming algorithm that computes the edit distance between

an m-character string A[1..m] and an n-character string B[1..n]. Let E(i, j) be the edit distance between the prefixes of length i and j, that is, between A[1..i] and B[1..j]. The edit distance between the complete strings is therefore E(m, n). A crucial step towards the development of this algorithm is the following observation about the gap representation of an optimal edit sequence.

PREFIX PROPERTY. If we remove the last column of an optimal edit sequence then the remaining columns represent an optimal edit sequence for the remaining substrings.

We can easily prove this claim by contradiction: if the substrings had a shorter edit sequence, we could just glue the last column back on and get a shorter edit sequence for the original strings.

We can easily prove this claim by contradiction: if the substrings had a shorter edit sequence, we could just glue the last column back on and get a shorter edit sequence for the original strings.

Recursive formulation. We use the Prefix Property to develop a recurrence relation for E. The dynamic programming algorithm will be a straightforward implementation of that relation. There are a couple of obvious base cases:

• Erasing: we need i deletions to erase an i-character string, E(i, 0) = i.

• Creating: we need j insertions to create a jcharacter string, E(0, j) = j.

In general, there are four possibilities for the last column in an optimal edit sequence.

• Insertion: the last entry in the top row is empty, E(i, j) = E(i, j − 1) + 1.

• Deletion: the last entry in the bottom row is empty, E(i, j) = E(i − 1, j) + 1.

• Substitution: both rows have characters in the last column that are different, E(i, j) = E(i − 1, j − 1) + 1.

• No action: both rows end in the same character, E(i, j) = E(i − 1, j − 1).

Let P be the logical proposition A[i] ≠ B[j] and denote by |P| its indicator variable: |P| = 1 if P is true and |P| = 0 if P is false. We can now summarize and for i, j > 0 get the edit distance as the smallest of the possibilities:

E(i, j) = min E(i, j − 1) + 1 E(i − 1, j) + 1 E(i − 1, j − 1) + |P| .

The algorithm.

If we turned this recurrence relation directly into a divide-and-conquer algorithm, we would have the following recurrence for the running time:

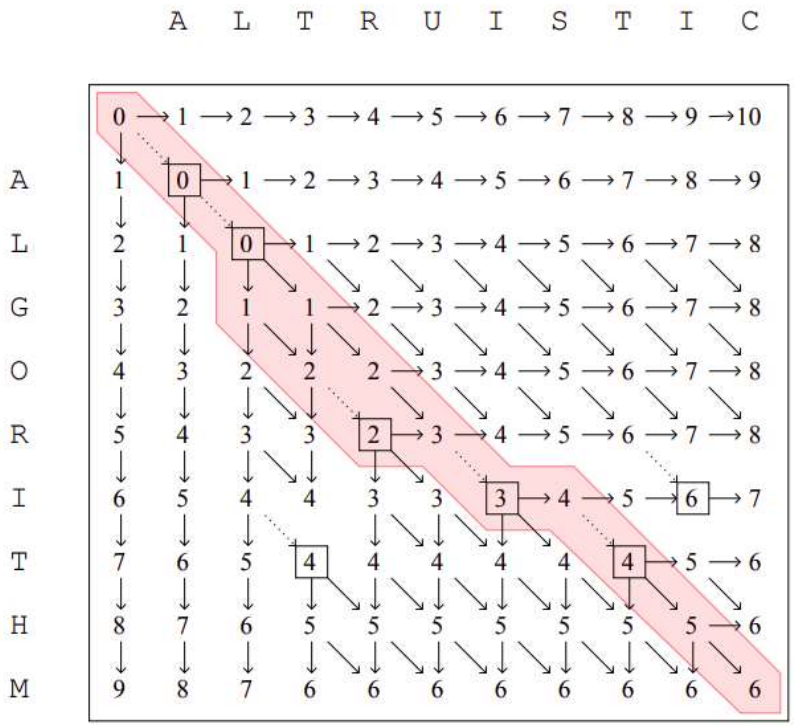
T (m, n) = T (m, n − 1) + T (m − 1, n) + T (m − 1, n − 1) + 1.

The solution to this recurrence is exponential in m and n, which is clearly not the way to go. Instead, let us build an m + 1 times n + 1 table of possible values of E(i, j). We can start by filling in the base cases, the entries in the 0-th row and column. To fill in any other entry, we need to know the values directly to the left, directly above, and both to the left and above. If we fill the table from top to bottom and from left to right then whenever we reach an entry, the entries it depends on are already available.

```
int EDITDISTANCE(int m, n)
 for i = 0 to m do E[i, 0] = i endfor;
 for j = 1 to n do E[0, j] = j endfor;
 for i = 1 to m do
  for j = 1 to n do
   E[i, j] = min{E[i, j − 1] + 1, E[i − 1, j] + 1, E[i − 1, j − 1] + |A[i] != B[j]|}
   endfor
endfor;
 return E[m, n].
```

Since there are (m+1)(n+1) entries in the table and each takes a constant time to compute, the total running time is in O(mn).

An example. The table constructed for the conversion of ALGORITHM to ALTRUISTIC is shown in Figure 5. Boxed numbers indicate places where the two strings have equal characters. The arrows indicate the predecessors that define the entries. Each direction of arrow corresponds to a different edit operation: horizontal for insertion, vertical for deletion, and diagonal for substitution. Dotted diagonal arrows indicate free substitutions of a letter for itself.



The table of edit distances between all prefixes of ALGORITHM and of ALTRUISTIC. The shaded area highlights the optimal edit sequences, which are paths from the upper left to the lower right corner

Recovering the edit sequence. By construction, there is at least one path from the upper left to the lower right corner, but often there will be several. Each such path describes an optimal edit sequence. For the example at hand, we have three optimal edit sequences:

A L G O R I T H M

A L T R U I S T I C

A L G O R I T H M

A L T R U I S T I C

They are easily recovered by tracing the paths backward, from the end to the beginning. The following algorithm recovers an optimal solution that also minimizes the number of insertions and deletions. We call it with the lengths of the strings as arguments, R(m, n).

void R(int i, j)

if i > 0 or j > 0

then switch incoming arrow:

case : R(i − 1, j − 1); print(A[i], B[j])

case ↓: R(i − 1, j); print(A[i], -)

case →: R(i, j − 1); print(, B[j]).

Endswitch

Endif.

Summary:

The structure of dynamic programming is again similar to divide-and-conquer, except that the subproblems to be solved overlap. As a consequence, we get different recursive paths to the same subproblems. To develop a dynamic programming algorithm that avoids redundant solutions, we generally proceed in two steps:

1. We formulate the problem recursively. In other words, we write down the answer to the whole problem as a combination of the answers to smaller subproblems.

2. We build solutions from bottom up. Starting with the base cases, we work our way up to the final solution and (usually) store intermediate solutions in a table.

For dynamic programming to be effective, we need a structure that leads to at most some polynomial number of different subproblems. Most commonly, we deal with sequences,

which have linearly many prefixes and suffixes and quadratically many contiguous substrings.

0/1 KNAPSACK PROBLEM:

Knapsack is basically means bag. A bag of given capacity.

We want to pack n items in your luggage.

- The ith item is worth v_i dollars and weight w_i pounds.
- Take as valuable a load as possible, but cannot exceed W pounds.
- v_i w_i W are integers.

2. W ≤ capacity
3. Value ← Max

Input:

- Knapsack of capacity
- List (Array) of weight and their corresponding value.

Output: To maximize profit and minimize weight in capacity.

The knapsack problem where we have to pack the knapsack with maximum value in such a manner that the total weight of the items should not be greater than the capacity of the knapsack.

In this 0/1 knapsack problem, item cannot be broken which means thief should take the item as a whole or should leave it. That's why it is called **0/1 knapsack Problem**.

- Each item is taken or not taken.
- Cannot take a fractional amount of an item taken or take an item more than once.

Step 1:

First, we create a 2-dimensional array (i.e. a table) of $n + 1$ rows and $w + 1$ columns.

A row number *i* represents the set of all the items from rows 1— *i*. For instance, the values in row 3 assumes that we only have items 1, 2, and 3.

A column number *j* represents the weight capacity of our knapsack. Therefore, the values in column 5, for example, assumes that our knapsack can hold 5 weight units.

Putting everything together, an entry in row *i*, column *j* represents the maximum value that can be obtained with items 1, 2, 3 … *i*, in a knapsack that can hold *j* weight units.

Let's call our table mat for matrix. Therefore, int[][] mat = new int[n + 1][w + 1].

Step 2:

We can immediately begin filling *some* entries in our table: the base cases, for which the solution is trivial. For instance, at row 0, when we have *no items* to pick from, the maximum value that can be stored in any knapsack must be 0. Similarly, at column 0, for a knapsack which can hold 0 weight units, the maximum value that can be stored in it is 0. (We're assuming that there are no massless, valuable items.)

We can do this with 2 for loops.

```
Int[][] mat = new int[n + 1][w + 1];

For (int r = 0 ; r < w + 1 ; r++) {

Mat[0][r] = 0;

}

For (int c = 0 ; r < n + 1 ; c++) {

Mat[c][0] = 0;

}
```

Step 3:

Now, we want to begin populating our table. As with all dynamic programming solutions, at each step, we will make use of our solutions to previous sub-problems.

Logic description:

The relationship between the value at row *i*, column *j* and the values to the previous sub-problems is as follows:

Recall that at row *i* and column *j*, we are tackling a sub-problem consisting of items 1, 2, 3 … *i* with a knapsack of *j* capacity. There are 2 options at this point: we can either include item *i* or not. Therefore, we need to compare the maximum value that we can obtain with and without item *i*.

The maximum value that we can obtain *without* item *i* can be found at row *i-1*, column *j*. This part's easy. The reasoning is straightforward: whatever maximum value we can obtain with items 1, 2, 3 … *i* must obviously be the same maximum value we can obtain with items 1, 2, 3 … *i - 1*, *if we choose not to include item i*.

To calculate the maximum value that we can obtain *with* item *i*, we first need to compare the weight of item *i* with the knapsack's weight capacity. Obviously, if item *i* weighs more than what the knapsack can hold, we can't include it, so it does not make sense to perform the calculation. In that case, the solution to this problem is simply the maximum value that we can obtain without item *i* (i.e. the value in the row above, at the same column).

However, suppose that item *i* weighs less than the knapsack's capacity. We thus have the option to include it, if it potentially increases the maximum obtainable value. The maximum obtainable value by including item *i* is thus = the value of item *i* itself + the maximum value that can be obtained with the remaining capacity of the knapsack. We obviously want to make full use of the capacity of our knapsack, and not let any remaining capacity go to waste.

Therefore, at row *i* and column *j* (which represents the maximum value we can obtain there), **we would pick either the maximum value that we can obtain *without* item *i*, or the maximum value that we can obtain *with* item *i*, *whichever is larger*.**

Example:

At row 3 (item 2), and column 5 (knapsack capacity of 4), we can choose to either include item 2 (which weighs 4 units) or not. If we choose not to include it, the maximum value we can obtain is the same as if we only have item 1 to choose from (which is found in the row above, i.e. 0). If we want to include item 2, the maximum value we can obtain with item 2 is the value of item 2 (40) + the maximum value we can obtain with the remaining capacity (which is 0, because the knapsack is completely full already).

At row 3 (item 2), and column 10 (knapsack capacity of 9), again, we can choose to either include item 2 or not. If we choose not to, the maximum value we can obtain is the same as that in the row above at the same column, i.e. 10 (by including only item 1, which has a value of 10). If we include item 2, we have a remaining knapsack capacity of 9 - 4 = 5. We can find out the maximum value that can be obtained with a capacity of 5 by looking at the row above, at column 5. Thus, the maximum value we can obtain by including item 2 is 40 (the value of item 2) + 10 = 50.

We pick the larger of 50 vs 10, and so the maximum value we can obtain with items 1 and 2, with a knapsack capacity of 9, is 50.

Step 4 (final solution):

Once the table has been populated, the final solution can be found at the last row in the last column, which represents the maximum value obtainable with *all the items* and the *full capacity* of the knapsack.

return mat[n][w];

GREEDY ALGORITHMS:

The philosophy of being greedy is shortsightedness. Always go for the seemingly best next thing, always optimize the presence, without any regard for the future, and never change your mind about the past. The greedy paradigm is typically applied to optimization problems. In this section, we first consider a scheduling problem and second the construction of optimal codes.

Let we are given a problem to sort the array a = {5, 3, 2, 9}. Someone says the array after sorting is {1, 3, 5, 7}. Can we consider the answer is correct? The answer is definitely “no” because the elements of the output set are not taken from the input set. Let someone says the array after sorting is {2, 5, 3, 9}. Can we admit the answer? The answer is again “no” because the output is not satisfying the objective function that is the first element must be less than the second, the second element must be less than the third and so on. Therefore, the solution is said to be a feasible solution if it satisfies the following constraints.

(i) Explicit constraints: - The elements of the output set must be taken from the input set.

(ii) Implicit constraints:-The objective function defined in the problem.

The best of all possible solutions is called the optimal solution. In other words we need to find the

solution which has the optimal (maximum or minimum) value satisfying the given constraints.

The choice of each step is a greedy approach is done based in the following:

1. It must be feasible
2. It must be locally optimal
3. It must be unalterable

The Greedy approach constructs the solution through a sequence of steps. Each step is chosen such

that it is the best alternative among all feasible choices that are available. The choice of a step once

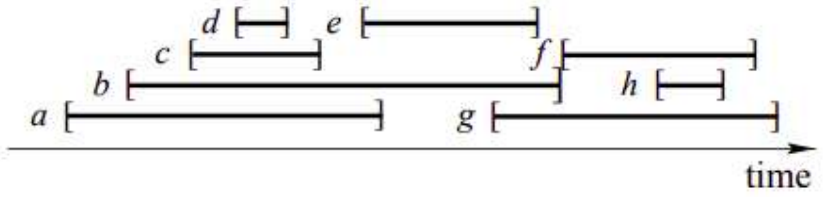
made cannot be changed in subsequent steps.

Problem 1:

Let us consider the problem of coin change. Suppose a greedy person has some 25p, 20p, 10p, 5paise coins. When someone asks him for some change then be wants to given the change with minimum number of coins. Now, let someone requests for a change of top then he first selects 25p. Then the remaining amount is 45p. Next, he selects the largest coin that is less than or equal to 45p i.e. 25p. The remaining 20p is paid by selecting a 20p coin. So the demand for top is paid by giving total 3 numbers of coins. This solution is an optimal solution. Now, let someone requests for a change of 40p then the Greedy approach first selects 25p coin, then a 10p coin and finally a 5p coin. However, the some could be paid with two 20p coins. So it is clear from this example that Greedy approach tries to find the optimal solution by selecting the elements one by one that are locally optimal. But Greedy method never gives the guarantee to find the optimal solution.

Problem 2: (A scheduling problem):

Consider a set of activities 1, 2, . . ., n. Activity i starts at time si and finishes at time fi > si . Two activities i and j overlap if [si , fi] ∩ [sj , fj] ≠ ∅. The objective is to select a maximum number of pairwise non-overlapping activities. An example is shown in Figure 6. The largest number of ac tivities can be scheduled by choosing activities with early finish times first. We first sort and reindex such that i < j implies fi ≤ fj .



A best schedule is c, e, f, but there are also others of the same size.

S = {1}; last = 1;

for i = 2 to n do

if flast < si then

S = S ∪ {i}; last = i

Endif

Endfor.

The running time is O(n log n) for sorting plus O(n) for the greedy collection of activities.

It is often difficult to determine how close to the optimum the solutions found by a greedy algorithm really are. However, for the above scheduling problem the greedy algorithm always finds an optimum. For the proof let 1 = i1 < i2 < . . . < ik be the greedy schedule constructed by the algorithm. Let j1 < j2 < . . . < jℓ be any other feasible schedule. Since i1 = 1 has the earliest finish time of any activity, we have fi1 ≤ fj1 . We can therefore add i1 to the feasible schedule and remove at most one activity, namely j1. Among the activities that do not overlap i1, i2 has the earliest finish time, hence fi2 ≤ fj2 . We can again add i2 to the feasible schedule and remove at most one activity, namely j2 (or possibly j1 if it was not removed before). Eventually, we replace the entire feasible schedule by the greedy schedule without decreasing the number of activities. Since we could have started with a maximum feasible schedule, we conclude that the greedy schedule is also maximum.

Problem 3: (huffman coding)

Suppose the string below is to be sent over a network.

Initial string = BCAADDDCCACACAC

Each character occupies 8 bits. There are a total of 15 characters in the above string. Thus, a total of 8 * 15 = 120 bits are required to send this string.

Using the Huffman Coding technique, we can compress the string to a smaller size.

Huffman coding first creates a tree using the frequencies of the character and then generates code for each character.

Once the data is encoded, it has to be decoded. Decoding is done using the same tree.

Huffman Coding prevents any ambiguity in the decoding process using the concept of **prefix code** ie. a code associated with a character should not be present in the prefix of any other code. The tree created above helps in maintaining the property.

Huffman coding is done with the help of the following steps.

1. Calculate the frequency of each character in the string.

B -> 1

C-> 6

A-> 5

D-> 3

2. Sort the characters in increasing order of the frequency. These are stored in a priority queue Q.

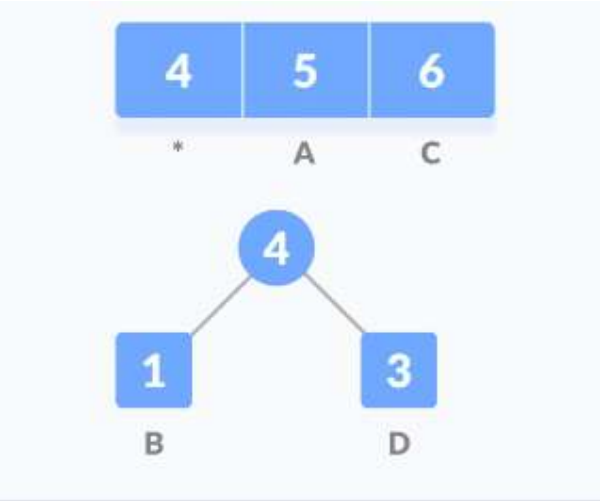
B -> 1

D -> 3

A -> 5

C -> 6

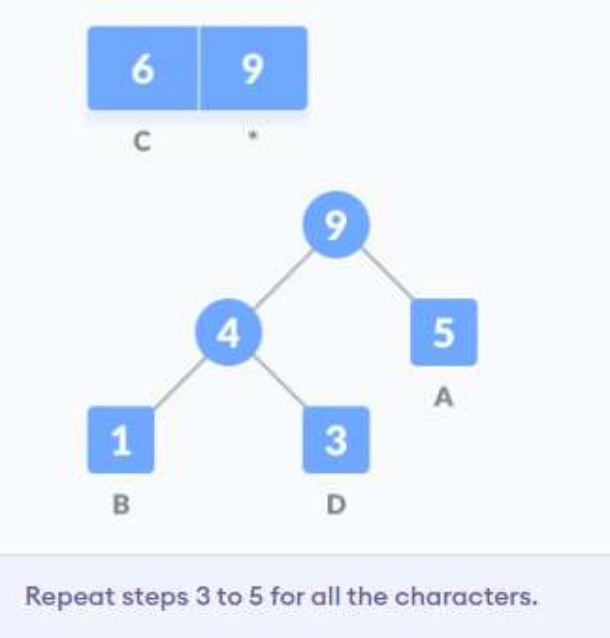
3. Make each unique character as a leaf node.
4. Create an empty node z. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z. Set the value of the z as the sum of the above two minimum frequencies.

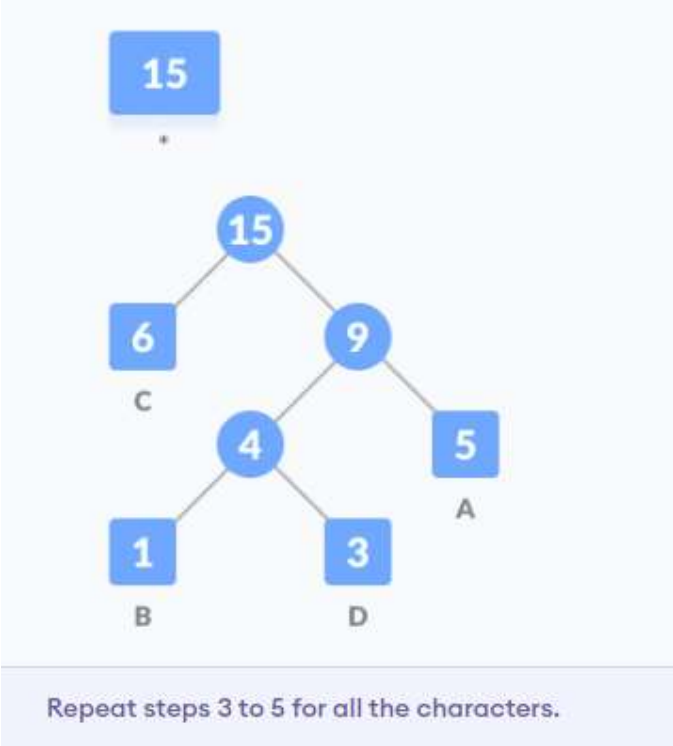


Getting the sum of the least numbers

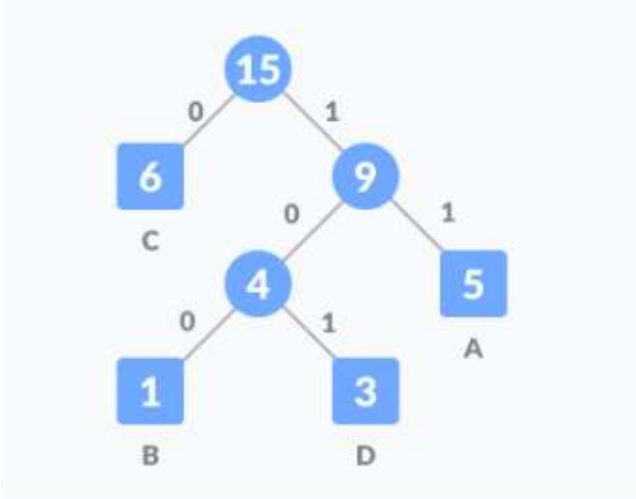
5. Remove these two minimum frequencies from Q and add the sum into the list of frequencies (* denote the internal nodes in the figure above).

6. Insert node z into the tree.
7. Repeat steps 3 to 5 for all the characters.





8. For each non-leaf node, assign 0 to the left edge and 1 to the right edge.



Assign 0 to the left edge and 1 to the right edge

For sending the above string over a network, we have to send the tree as well as the above compressed-code. The total size is given by the table below.

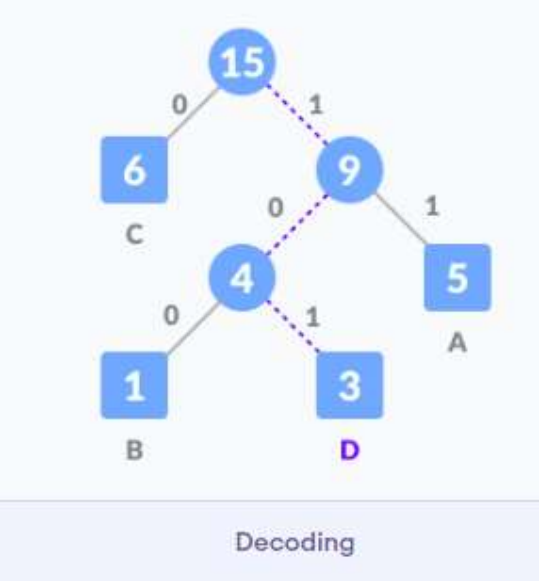
Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4 * 8 = 32 bits	15 bits		28 bits

Without encoding, the total size of the string was 120 bits. After encoding the size is reduced to 32 + 15 + 28 = 75.

Decoding the code:

For decoding the code, we can take the code and traverse through the tree to find the character.

Let 101 is to be decoded, we can traverse from the root as in the figure below.



Huffman Coding Algorithm:

create a priority queue Q consisting of each unique character.

sort then in ascending order of their frequencies.

for all the unique characters:

create a newNode

extract minimum value from Q and assign it to leftChild of newNode

extract minimum value from Q and assign it to rightChild of newNode

calculate the sum of these two minimum values and assign it to the value of newNode

insert this newNode into the tree

return rootNode

Huffman Coding Complexity:

The time complexity for encoding each unique character based on its frequency is O(nlog n).

Extracting minimum frequency from the priority queue takes place 2*(n-1) times and its complexity is O(log n). Thus the overall complexity is O(nlog n).

Huffman Coding Applications:

- Huffman coding is used in conventional compression formats like GZIP, BZIP2, PKZIP, etc.
- For text and fax transmissions

CHAPTER 4

SEARCHING

Sequential search

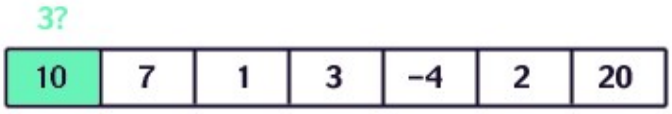
One of the most straightforward and elementary searches is the sequential search, also known as a linear search.

As a real world example, pickup the nearest phonebook and open it to the first page of names. We're looking to find the first "Smith". Look at the first name. Is it "Smith"? Probably not (it's probably a name that begins with 'A'). Now look at the next name. Is it "Smith"? Probably not. Keep looking at the next name until you find "Smith".

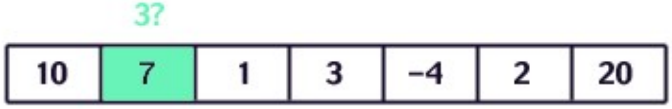
The above is an example of a sequential search. You started at the beginning of a sequence and went through each item one by one, in the order they existed in the list, until you found the item you were looking for. Of course, this probably isn't how you normally look up a name in the phonebook.

Now we'll look at this as related to computer science. Instead of a phonebook, we have an array. Although the array can hold data elements of any type, for the simplicity of an example we'll just use an array of integers, like the following:

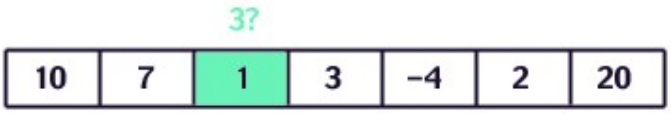
Lets search for the number 3. We start at the beginning and check the first element in the array. Is it 3?



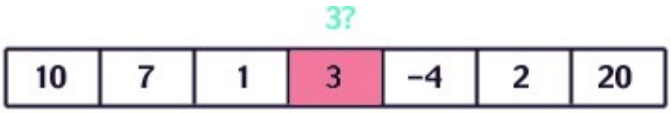
No, not it. Is it the next element?



Not there either. The next element?



Not there either. Next?



We found it!!! Now you understand the idea of linear searching; we go through each element, in order, until we find the correct value.

Binary search

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Idea:

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

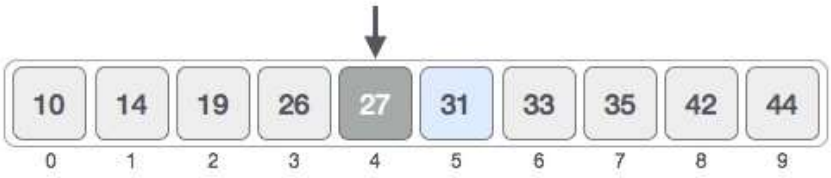
Working:

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

We change our low to mid + 1 and find the new mid value again.

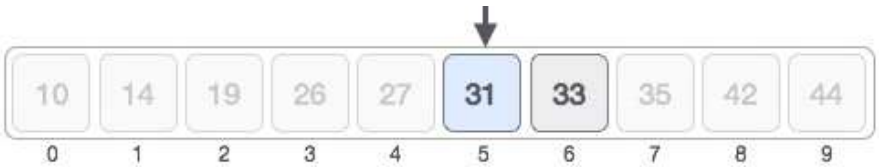
low = mid + 1

mid = low + (high - low) / 2

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Pseudocode:

```
Procedure binary_search
 A ← sorted array
 n ← size of array
 x ← value to be searched
 Set lowerBound = 1
 Set upperBound = n
  while x not found
 if upperBound < lowerBound
   EXIT: x does not exists.
 set midPoint = lowerBound + ( upperBound - lowerBound ) / 2
 ifA[midPoint] < x
   setlowerBound = midPoint + 1
 if A[midPoint] > x
   set upperBound = midPoint – 1
 if A[midPoint] = x
   EXIT: x found at location midPoint
end while
 end procedure
```

SEARCH TREES:

A search tree is a data structure for storing an evolving set of objects associated with keys (and possibly lots of other data). It maintains a total ordering over the stored objects, and can support a richer set of operations than a heap, at the expense of increased space and, for some operations, somewhat slower running times. We'll start with the "what" (that is, supported operations) before proceeding to the "why" (applications) and the "how" (optional implementation details)

SORTED ARRAYS

A good way to think about a search tree is as a dynamic version of a sorted array—it can do everything a sorted array can do, while also accommodating fast insertions and deletions.

You can do a lot of things with a sorted array.

Sorted Arrays: Supported Operations

Search: for a key k, return a pointer to an object in the data structure with key k (or report that no such object exists).

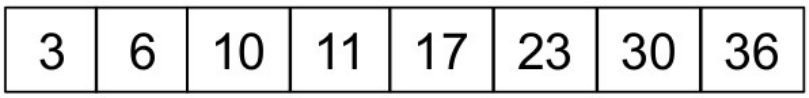
Min (Max): return a pointer to the object in the data structure with the smallest (respectively, largest) key. Predecessor (Successor): given a pointer to an object in the data structure, return a pointer to the object with the next-smallest (respectively, next-largest) key. If the given object has the minimum (respectively, maximum) key, report "none."

OutputSorted: output the objects in the data structure one by one in order of their keys.

Select: given a number i, between 1 and the number of objects, return a pointer to the object in the data structure with the ith-smallest key.

Rank: given a key k, return the number of objects in the data structure with key at most k.

Let's review how to implement each of these operations, with the following running example:



The Search operation uses binary search: First check if the object in the middle position of the array has the desired key. If so, return it. If not, recurse either on the left half (if the

middle object's key is too large) or on the right half (if it's too small). 1 For example, to search the array above for the key 8, binary search will: examine the fourth object (with key 11); recurse on the left half (the objects with keys 3, 6, and 10); check the second object (with key 6); recurse on the right half of the remaining array (the object with key 10); conclude that the rightful position for an object with key 8 would be between the second and third objects; and report "none." As each recursive call cuts the array size by a factor of 2, there are at most log 2 n recursive calls, where n is the length of the array. Because each recursive call does a constant amount of work, the operation runs in O(log n) time.

Min and Max are easy to implement in O(1) time: Return a pointer to the first or last object in the array, respectively

To implement Predecessor or Successor, use the Search operation to recover the position of the given object in the sorted array, and return the object in the previous or next position, respectively. These operations are as fast as Search—running in O(log n) time, where n is the length of the array.

The OutputSorted operation is trivial to implement in linear time with a sorted array: Perform a single front-to-back pass over the array, outputting each object in turn.

Select is easy to implement in constant time: Given an index i, return the object in the ith position of the array.

The Rank operation, which is like an inverse of Select, can be implemented along the same lines as Search: If binary search finds an object with key k in the ith position of the array, or if it discovers that k is in between the keys of the objects in the ith and (i + 1)th positions, the correct answer is i^2.

Summarizing, here's the final scorecard for sorted arrays:

Operation	Running time
SEARCH	$O(\log n)$
MIN	$O(1)$
MAX	$O(1)$
PREDECESSOR	$O(\log n)$
SUCCESSOR	$O(\log n)$
OUTPUTSORTED	$O(n)$
SELECT	$O(1)$
RANK	$O(\log n)$

Sorted arrays: supported operations and their running times, where n denotes the current number of objects stored in the array.

Unsupported Operations:

Could you really ask for anything more? With a static data set that does not change over time, this is an impressive list of supported operations. Many real-world applications are dynamic, however, with the set of relevant objects evolving over time. For example, employees come and go, and the data structure that stores their records should stay up to date. For this reason, we also care about insertions and deletions.

Sorted Arrays: Unsupported Operations

Insert: given a new object x, add x to the data structure.

Delete: for a key k, delete an object with key k from the data structure, if one exists.

These two operations aren't impossible to implement with a sorted array, but they're painfully slow—inserting or deleting an element while maintaining the sorted array property requires linear time in the worst case. Is there an alternative data structure that replicates all the functionality of a sorted array, while matching the logarithmic-time performance of a heap for the Insert and Delete operations?

Search Trees: Supported Operations

The raison d'être of a search tree is to support all the operations that a sorted array supports, plus insertions and deletions. All the perations except OutputSorted run in O(log n) time, where n is the number of objects in the search tree. The OutputSorted operation runs in O(n) time, and this is as good as it gets (since it must output n objects).

Here's the scorecard for search trees, with a comparison to sorted arrays:

Operation	Sorted Array	Balanced Search Tree
SEARCH	$O(\log n)$	$O(\log n)$
MIN	$O(1)$	$O(\log n)$
MAX	$O(1)$	$O(\log n)$
PREDECESSOR	$O(\log n)$	$O(\log n)$
SUCCESSOR	$O(\log n)$	$O(\log n)$
OUTPUTSORTED	$O(n)$	$O(n)$
SELECT	$O(1)$	$O(\log n)$
RANK	$O(\log n)$	$O(\log n)$
INSERT	$O(n)$	$O(\log n)$
DELETE	$O(n)$	$O(\log n)$

Balanced search trees vs. sorted arrays: supported operations and their running times, where n denotes the current number of objects stored in the data structure.

When to Use a Balanced Search Tree

If your application requires maintaining an ordered representation of a dynamically changing set of objects, the balanced search tree (or a data structure based on one) is usually the data structure of choice.

remember the principle of parsimony: Choose the simplest data structure that supports all the operations required by your application. If you need to maintain only an ordered representation of a static data set (with no insertions or deletions), use a sorted array instead of a balanced search tree; the latter would be overkill. If your data set is dynamic but you care only about fast minimum (or maximum) operations, use a heap instead of a balanced search tree. These simpler data structures do less than a balanced search tree, but what they do, they do better—faster (by a constant or logarithmic factor) and with less space (by a constant factor).

The Search Tree Property

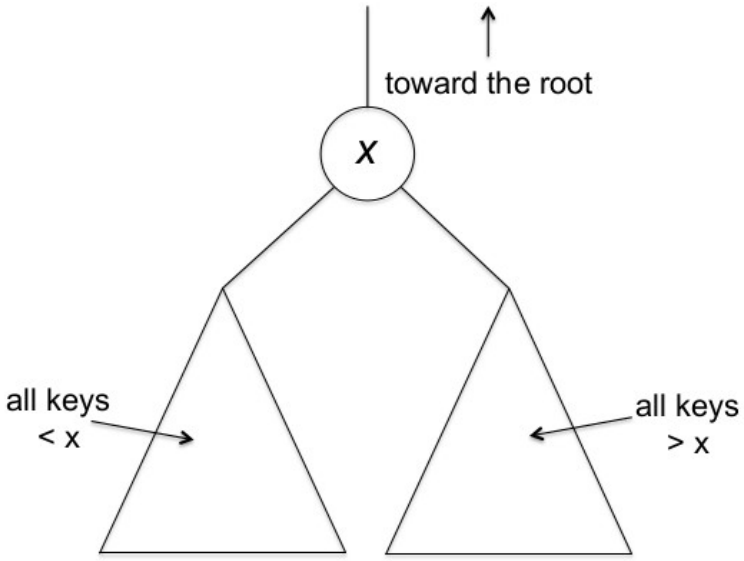
In a binary search tree, every node corresponds to an object (with a key) and has three pointers associated with it: a parent pointer, a left child pointer, and a right child pointer. Any of these pointers can be null, indicating the absence of a parent or child. The left subtree of a node x comprises the nodes reachable from x via its left child pointer, and similarly for the right subtree. The defining search tree property is:

The search tree property

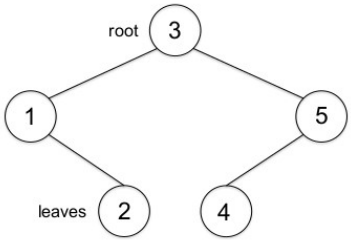
1. For every object x, objects in x's left subtree have keys smaller than that of x.

2. For every object x, objects in x's right subtree have keys larger than that of x.

The search tree property imposes a requirement for every node of a search tree, not just for the root:



For example, here's a search tree containing objects with the keys {1, 2, 3, 4, 5}, and a table listing the destinations of the three pointers at each node:



(a) Search tree

Node	Parent	Left	Right
1	3	null	2
2	1	null	null
3	null	1	5
4	5	null	null
5	3	4	null

(b) Pointers

A search tree and its corresponding parent and child pointers.

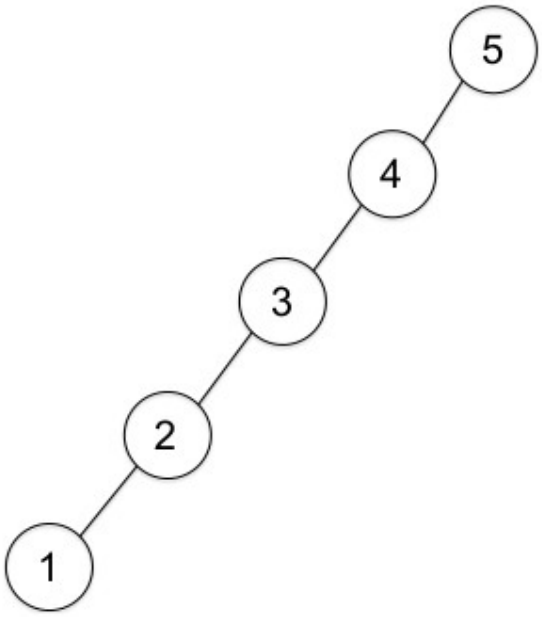
Binary search trees and heaps differ in several ways. Heaps can be thought of as trees, but they are implemented as arrays, with no explicit pointers between objects. A search tree explicitly stores three pointers per object, and hence uses more space (by a constant factor). Heaps don't need explicit pointers because they always correspond to full binary trees, while binary search trees can have an arbitrary structure.

Search trees have a different purpose than heaps. For this reason, the search tree property is incomparable to the heap property. Heaps are optimized for fast minimum computations, and the heap property— that a child's key is only bigger than its parent's key—makes the minimum-key object easy to find (it's the root). Search trees are optimized for—wait for it—search, and the search tree property is defined accordingly. For example, if you are searching for an object with the key 23 in a search tree and the root's key is 17, you know that the object can reside only in the root's right subtree, and can discard the objects in the

left subtree from further consideration. This should remind you of binary search, as befits a data structure whose raison d'être is to simulate a dynamically changing sorted array.

The Height of a Search Tree

Many different search trees exist for a given set of keys. Here's a second search tree containing objects with the keys {1, 2, 3, 4, 5}:



Both conditions in the search tree property hold, the second one vacuously (as there are no non-empty right subtrees). The height of a tree is defined as the length of a longest path from its root to a leaf. 10 Different search trees containing identical sets of objects can have different heights, as in our first two examples (which have heights 2 and 4, respectively). In general, a binary search tree containing n objects can have a height anywhere from

$$\underbrace{\approx \log_2 n}_{\substack{\text{perfectly balanced binary tree} \\ \text{(best-case scenario)}}} \quad \text{to} \quad \underbrace{n-1.}_{\substack{\text{chain, as above} \\ \text{(worst-case scenario)}}}$$

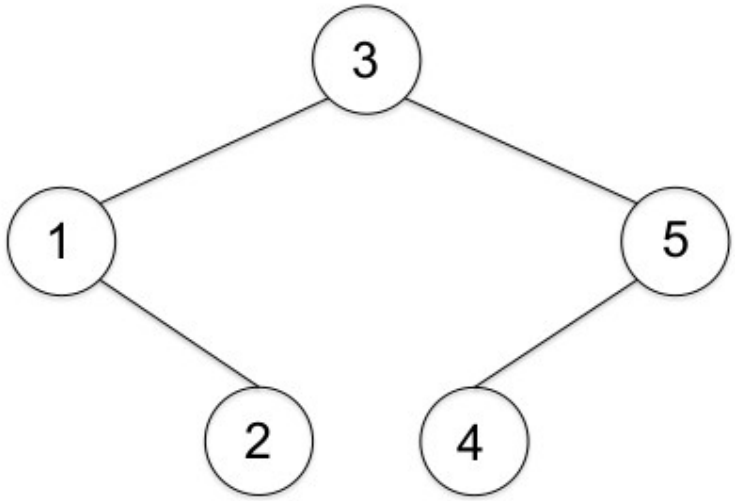
The rest of this section outlines how to implement all the operations of a binary search tree in time proportional to the tree's height (save OutputSorted, which runs in time linear in n). For the refinements of binary search trees that are guaranteed to have height O(log n), this leads to the logarithmic running times reported in the scorecard.

Implementing Search in O(height) Time

Let's begin with the Search operation:

for a key k, return a pointer to an object in the data structure with key k (or report that no such object exists).

The search tree property tells you exactly where to look for an object with key k. If k is less than (respectively, greater than) the root's key, such an object must reside in the root's left subtree (respectively, right tree). To search, follow your nose: Start at the root and repeatedly go left or right (as appropriate) until you find the desired object (a successful search) or encounter a null pointer (an unsuccessful search). For example, suppose we search for an object with key 2 in our first binary search tree:



Because the root's key (3) is too big, the first step traverses the left child pointer. Because the next node's key is too small (1), the second step traverses the right child pointer, arriving at the desired object. If we search for an object with key 6, the search traverses the root's right child pointer (as the root's key is too small). Because the next node's key (5) is also too small, the search tries to follow another right child pointer, encounters a null pointer, and halts the search (unsuccessfully).

Search

1. Start at the root node.

2. Repeatedly traverse left and right child pointers, as appropriate (left if k is less than the current node's key, right if k is bigger).

3. Return a pointer to an object with key k (if found) or "none" (upon reaching a null pointer).

The running time is proportional to the number of pointers followed, which is at most the height of the search tree (plus 1, if you count the final null pointer of an unsuccessful search).

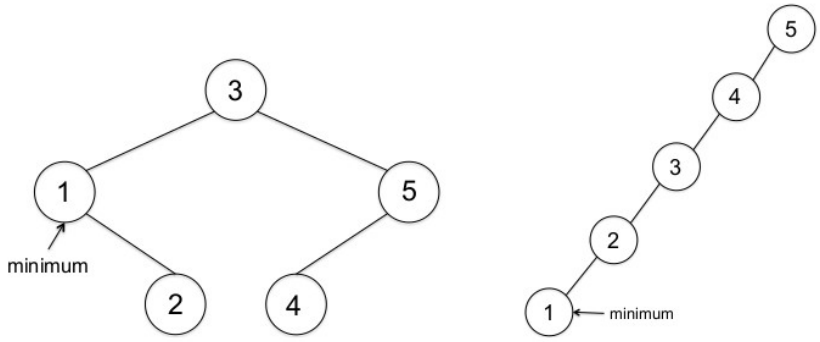
Implementing Min and Max in O(height) Time

The search tree property makes it easy to implement the Min and Max operations.

Min (Max): return a pointer to the object in the data structure with the smallest (respectively, largest) key.

Keys in the left subtree of the root can only be smaller than the root's key, and keys in the right subtree can only be larger. If the left subtree is empty, the root must be the minimum. Otherwise, the minimum of the left subtree is also the minimum of the entire tree. This suggests following the root's left child pointer and repeating the process.

For example, in the search trees we considered earlier:



repeatedly following left child pointers leads to the object with the minimum key.

Min (Max)

1. Start at the root node.

2. Traverse left child pointers (right child pointers) as long as possible, until encountering a null pointer.

3. Return a pointer to the last object visited.

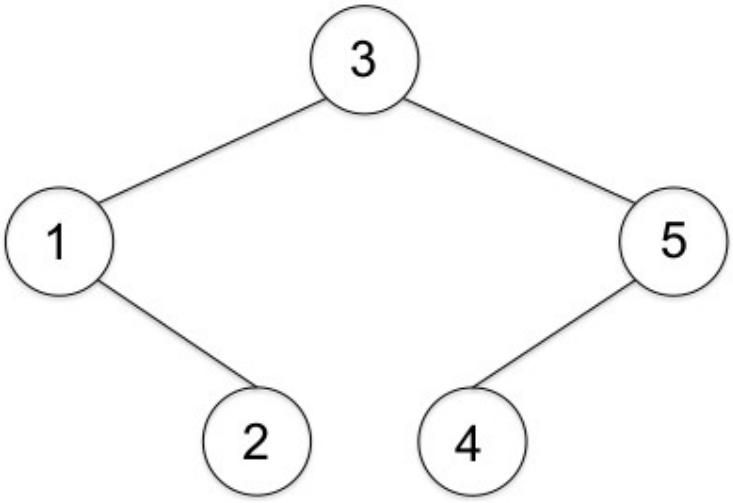
The running time is proportional to the number of pointers followed, which is O(height).

Implementing Predecessor in O(height) Time

Next is the Predecessor operation; the implementation of the Successor operation is analogous.

Predecessor: given a pointer to an object in the data structure, return a pointer to the object with the next- smallest key. (If the object has the minimum key, report “none.”)

Given an object x, where could x’s predecessor reside? Not in x’s right subtree, where all the keys are larger than x’s key (by the search tree property). Our running example illustrates two cases.



The predecessor might appear in the left subtree (as for the nodes with keys 3 and 5), or it could be an ancestor farther up in the tree (as for the nodes with keys 2 and 4).

The general pattern is: If an object x’s left subtree is non-empty, this subtree’s maximum element is x’s predecessor; otherwise, x’s predecessor is the closest ancestor of x that has a smaller key than x. Equivalently, tracing parent pointers upward from x, it is the destination of the first left turn. 12 For example, in the search tree above, tracing parent pointers upward from the node with key 4 first takes a right turn (leading to a node with the bigger key 5) and then takes a left turn, arriving at the correct predecessor (3). If x has an empty left subtree and no left turns above it, then it is the minimum in the search tree and has no predecessor (like the node with key 1 in the search tree above).

Predecessor

1. If x’s left subtree is non-empty, return the result of Max applied to this subtree.

2. Otherwise, traverse parent pointers upward toward the root. If the traversal visits consecutive nodes y and z with y a right child of z, return a pointer to z.

3. Otherwise, report “none.”

The running time is proportional to the number of pointers followed, which in all cases is O(height).

Implementing OutputSorted in O(n) Time

Recall the OutputSorted operation:

OutputSorted: output the objects in the data structure one by one in order of their keys.

A lazy way to implement this operation is to first use the Min operation to output the object with the minimum key, and then repeatedly invoke the Successor operation to output the rest of the objects in order. A better method is to use what's called an in-order traversal of the search tree, which recursively processes the root's left subtree, then the root, and then the root's right subtree. This idea meshes perfectly with the search tree property, which implies that Output-Sorted should first output the objects in the root's left subtree in order, followed by the object at the root, followed by the objects in the root's right subtree in order.

OutputSorted

1. Recursively call OutputSorted on the root's left subtree.

2. Output the object at the root.

3. Recursively call OutputSorted on the root's right subtree.

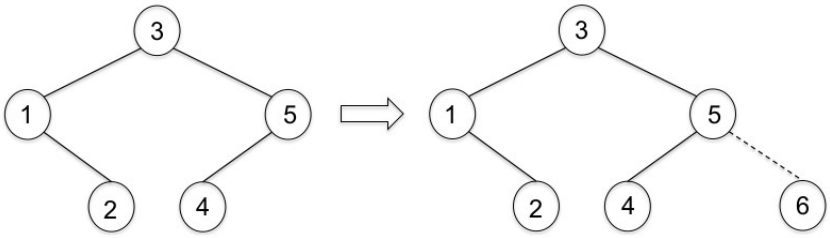
For a tree containing n objects, the operation performs n recursive calls (one initiated at each node) and does a constant amount of work in each, for a total running time of O(n).

Implementing Insert in O(height) Time

None of the operations discussed so far modify the given search tree, so they run no risk of screwing up the crucial search tree property. Our next two operations—Insert and Delete—make changes to the tree, and must take care to preserve the search tree property.

Insert: given a new object x, add x to the data structure.

The Insert operation piggybacks on Search. An unsuccessful search for an object with key k locates where such an object would have appeared. This is the appropriate place to stick a new object with key k (rewiring the old null pointer). In our running example, the correct location for a new object with key 6 is the spot where our unsuccessful search concluded:



What if there is already an object with key k in the tree? If you want to avoid duplicate keys, the insertion can be ignored. Otherwise, the search follows the left child of the existing object with key k, pushing onward until a null pointer is encountered.

Insert

1. Start at the root node.

2. Repeatedly traverse left and right child pointers, as appropriate (left if k is at most the current node's key, right if it's bigger), until a null pointer is encountered.

3. Replace the null pointer with one to the new object. Set the new node's parent pointer to its parent, and its child pointers to null.

The operation preserves the search tree property because it places the new object where it should have been. The running time is the same as for Search, which is O(height).

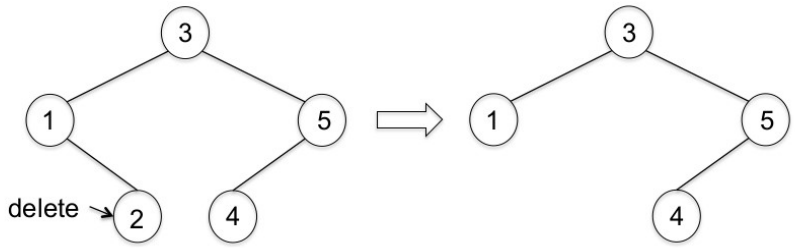
Implementing Delete in O(height) Time

In most data structures, the Delete operation is the toughest one to get right. Search trees are no exception.

Delete: for a key k, delete an object with key k from the search tree, if one exists.

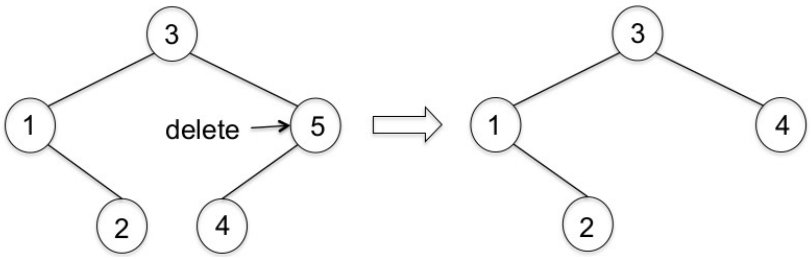
The main challenge is to repair a tree after a node removal so that the search tree property is restored.

The first step is to invoke Search to locate an object x with key k. (If there is no such object, Delete has nothing to do.) There are three cases, depending on whether x has 0, 1, or 2 children. If x is a leaf, it can be deleted without harm. For example, if we delete the node with key 2 from our favourite search tree:



For every remaining node y, the nodes in y's subtrees are the same as before, except possibly with x removed; the search tree property continues to hold.

When x has one child y, we can splice it out. Deleting x leaves y without a parent and x's old parent z without one of its children. The obvious fix is to let y assume x's previous position (as z's child). For example, if we delete the node with key 5 from our favourite search tree:

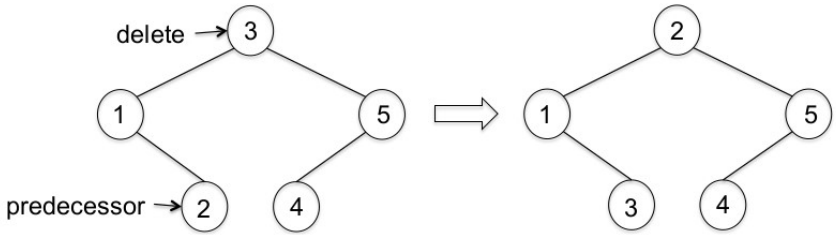


By the same reasoning as in the first case, the search property is preserved.

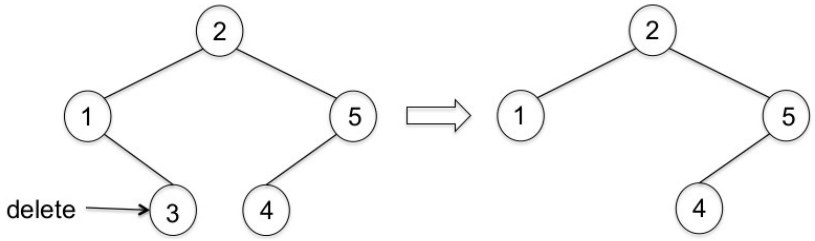
The hard case is when x has two children. Deleting x leaves two nodes without a parent, and it's not clear where to put them. In our running example, it's not obvious how to repair the tree after deleting its root.

The key trick is to reduce the hard case to one of the easy ones. First, use the Predecessor operation to compute the predecessor y of x. Because x has two children, its predecessor is the object in its

(non-empty!) left subtree with the maximum key. Since the maximum is computed by following right child pointers as long as possible, y cannot have a right child; it might or might not have a left child. Here's a crazy idea: Swap x and y! In our running example, with the root node acting as x:



This crazy idea looks like a bad one, as we've now violated the search tree property (with the node with key 3 in the left subtree of the node with key 2). But every violation of the search tree property involves the node x, which we're going to delete anyway. Because x now occupies y's previous position, it no longer has a right child. Deleting x from its new position falls into one of the two easy cases: We delete it if it also has no left child, and splice it out if it does have a left child. Either way, with x out of the picture, the search tree property is restored. Back to our running example:



Delete

1. Use Search to locate an object x with key k. (If no such object exists, halt.)

2. If x has no children, delete x by setting the appropriate child pointer of x's parent to null. (If x was the root, the new tree is empty.)

3. If x has one child, splice x out by rewiring the appropriate child pointer of x's parent to x's child, and the parent pointer of x's child to x's parent. (If x was the root, its child becomes the new root.)

4. Otherwise, swap x with the object in its left subtree that has the biggest key, and delete x from its new position (where it has at most one child).

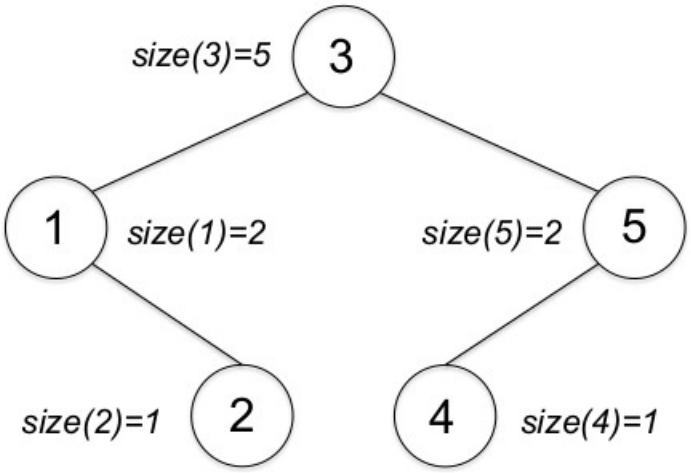
The operation performs a constant amount of work in addition to one Search and one Predecessor operation, so it runs in O(height) time.

Augmented Search Trees for Select

Finally, the **Select operation:**

Select: given a number i, between 1 and the number of objects, return a pointer to the object in the data structure with the ith-smallest key.

To get Select to run quickly, we'll augment the search tree by having each node keep track of information about the structure of the tree itself, and not just about an object. 17 Search trees can be augmented in many ways; here, we'll store at each node x an integer size(x) indicating the number of nodes in the subtree rooted at x (including x itself).



In our running example we have size(1) = 2, size(2) = 1, size(3) = 5, size(4) = 1, and size(5) = 2.

How is this additional information helpful? Imagine you're looking for the object with the 17th-smallest key (i.e., i = 17) in a search tree with 100 objects. Starting at the root, you can compute in constant time the sizes of its left and right subtrees. By the search tree property, every key in the left subtree is less than those at the root and in the right subtree. If the population of the left subtree is 25, these are the 25 smallest keys in the tree, including the 17th-smallest key. If its population is only 12, the right subtree contains all but the 13 smallest keys, and the 17th-smallest key is the 4th-smallest among its 87 keys. Either way, we can call Select recursively to locate the desired object.

Select

1. Start at the root and let j be the size of its left subtree. (If it has no left child pointer, then j = 0.)

2. If i = j + 1, return a pointer to the root.

3. If i < j + 1, recursively compute the ith-smallest key in the left subtree.

4. If i > j + 1, recursively compute the (i - j – 1)th smallest key in the right subtree.

Because each node of the search tree stores the size of its subtree, each recursive call performs only a constant amount of work. Each recursive call proceeds further downward in the tree, so the total amount of work is O(height).

Paying the piper. We still have to pay the piper. We've added and exploited metadata to the search tree, and every operation that modifies the tree must take care to keep this information up to date,

in addition to preserving the search tree property. You should think through how to re-implement the Insert and Delete operations, still running in O(height) time, so that all the subtree sizes remain accurate.

Question:

Suppose the node x in a search tree has children y and z. What is the relationship between size(x), size(y), and size(z)?

a) size(x) = max{size(y), size(z)} + 1

b) size(x) = size(y) + size(z)

c) size(x) = size(y) + size(z) + 1

d) There is no general relationship

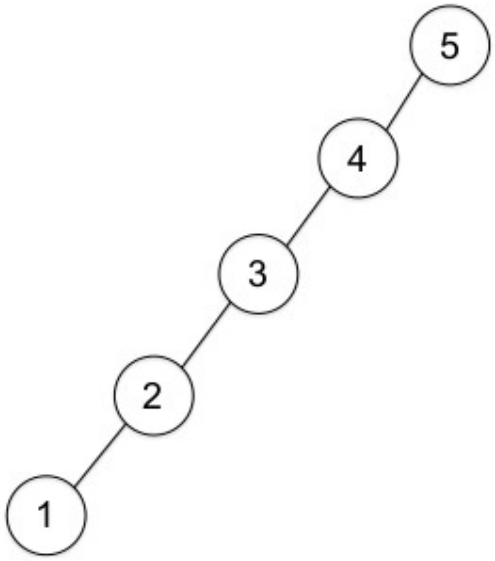
Correct answer: (c). Every node in the subtree rooted at x is either x itself, or a node in x's left subtree, or a node in x's right subtree. We therefore have

$$size(x) = \underbrace{size(y)}_{\text{nodes in left subtree}} + \underbrace{size(z)}_{\text{nodes in right subtree}} + \underbrace{1}_{x}.$$

BALANCED SEARCH TREES

Working Harder for Better Balance

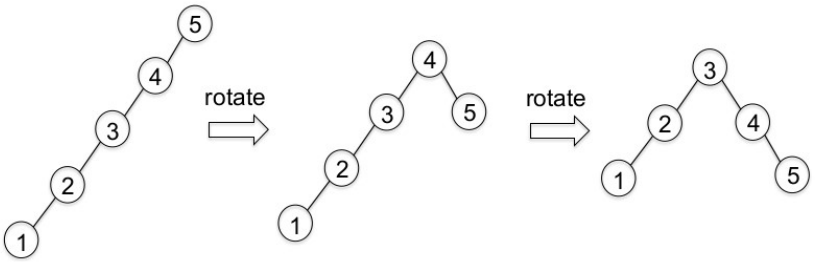
The running time of every binary search tree operation (save OutputSorted) is proportional to the tree's height, which can range anywhere from the best-case scenario of ⇡ log 2 n (for a perfectly balanced tree) to the worst-case scenario of n 1 (for a chain), where n is the number of objects in the tree. Badly unbalanced search trees really can occur, for example when objects are inserted in sorted or reverse sorted order:



The difference between a logarithmic and a linear running time is huge, so it's a win to work a little harder in Insert and Delete—still O(height) time, but with a larger constant factor—to guarantee that the tree's height is always O(log n). Several different types of balanced search trees guarantee O(log n) height and, hence, achieve the operation running times stated in the scorecard in Table The devil is in the implementation details, and they can get pretty tricky for balanced search trees. Happily, implementations are readily available and it's unlikely that you'll ever need to code up your own version from scratch. I encourage readers interested in what's under the hood of a balanced search tree to check out a textbook treatment or explore the open-source implementations and visualization demos that are freely available online. To whet your appetite for further study, let's conclude the chapter with one of the most ubiquitous ideas in balanced search tree implementations.

Rotations

All the most common implementations of balanced search trees use rotations, a constant-time operation that performs a modest amount of local rebalancing while preserving the search tree property. For example, we could imagine transforming the chain of five objects above into a more civilized search tree by composing two local rebalancing operations:



A rotation takes a parent-child pair and reverses their relationship. A right rotation applies when the child y is the left child of its parent x (and so y has a smaller key than x); after the rotation, x is the right child of y. When y is the right child of x, a left rotation makes x the left child of y.

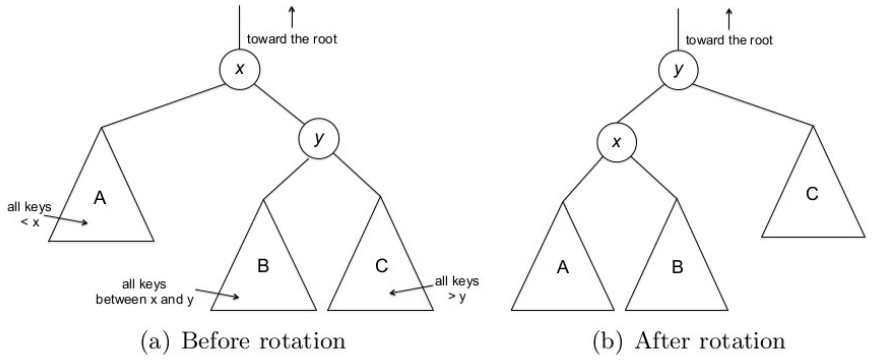


Fig: A left rotation in action.

The search tree property dictates the remaining details. For example, consider a left rotation, with y the right child of x. The search tree property implies that x's key is less than y's; that all the keys in x's left subtree ("A") are less than that of x (and y); that all the keys in y's right subtree ("C") are greater than that of y (and x); and that all the keys in y's left subtree ("B") are between those of x and y. After the rotation, y inherits x's old parent and has x as its new left child. There's a unique way to put all the pieces back together while preserving the search tree property, so let's just follow our nose.

There are three free slots for the subtrees A, B, and C: y's right child pointer and both child pointers of x. The search tree property forces us to stick the smallest subtree (A) as x's left child, and the largest subtree (C) as y's right child. This leaves one slot for subtree B (x's right child pointer), and fortunately the search tree property works out: All the subtree's keys are wedged between those of x and y, and the subtree winds up in y's left subtree (where it needs to be) and x's right subtree (ditto).

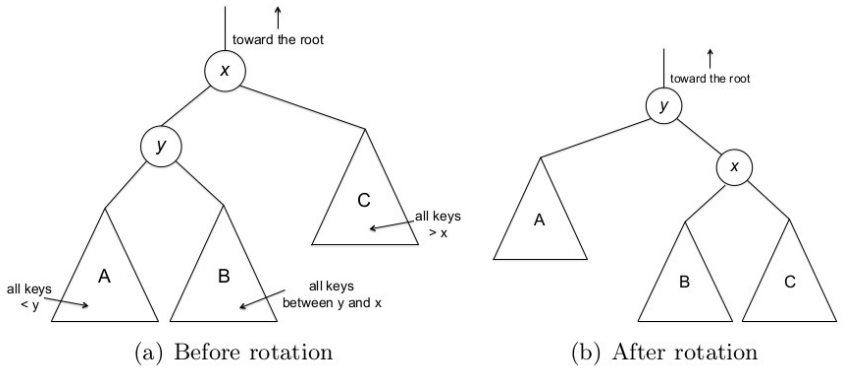


Fig: A right rotation in action.

A right rotation is then a left rotation in reverse because a rotation merely rewires a few pointers, it can be implemented with a constant number of operations. By construction, it preserves the search tree property.

The operations that modify the search tree—Insert and Delete— are the ones that must employ rotations. Without rotations, such an operation might render the tree a little more unbalanced. Since a single insertion or deletion can wreak only so much havoc, it should be plausible that a small—constant or perhaps logarithmic—number of rotations can correct any newly created imbalance. This is exactly what the aforementioned balanced search tree implementations do. The extra work from rotations adds *O(log n)* overhead to the Insert and Delete operations, leaving their overall running times at *O(log n)*.

The upshot:

1. If your application requires maintaining a totally ordered representation of an evolving set of objects, the balanced search tree is usually the data structure of choice.
2. Balanced search trees support the operations Search, Min, Max, Predecessor, Successor, Select, Rank, Insert, and Delete in O(log n) time, where n is the number of objects.
3. A binary search tree has one node per object, each with a parent pointer, a left child pointer, and a right child pointer.
4. The search tree property states that, at every node x of the tree, the keys in x's left subtree are smaller than x's key, and the keys in x's right subtree are larger than x's key.
5. The height of a search tree is the length of a longest path from its root to a leaf. A binary search tree with n objects can have height any-where from ⇡ log 2 n to n 1.
6. In a basic binary search tree, all the supported operations above can be implemented in O(height) time. (For Select and Rank, after augmenting the tree to maintain subtree sizes at each node.)

7. Balanced binary search trees do extra work in the Insert and Delete operations—still O(height) time, but with a larger constant factor—to guarantee that the tree's height is always O(log n).

CHAPTER 5

PRIORITIZING

Data structures are used in almost every major piece of software, so knowing when and how to use them is an essential skill for the serious programmer. The raison d'être of a data structure is to organize data so you can access it quickly and usefully. You've already seen a few examples.

There are many more data structures out there—in this book series, we'll see heaps, binary search trees, hash tables etc.

Why such a bewildering laundry list? Because different data structures support different sets of operations, making them well-suited for different types of programming tasks. For example, breadth- and depth-first search have different needs, necessitating two different data structures.

What are the pros and cons of different data structures, and how should you choose which one to use in a program? In general, the more operations a data structure supports, the slower the operations and the greater the space overhead. The following quote, widely attributed to Albert Einstein, is germane:

"Make things as simple as possible, but not simpler."

When implementing a program, it's important that you think carefully about exactly which operations you'll use over and over again. For example, do you care only about tracking which objects are stored in a data structure, or do you also want them ordered in a specific way? Once you understand your program's needs, you can follow the principle of parsimony and choose a data structure that supports all the desired operations and no superfluous ones.

Principle of Parsimony

Choose the simplest data structure that supports all the operations required by your application.

Taking It to the Next Level

What are your current and desired levels of expertise in data structures?

Level 0: *"What's a data structure?"*

Level 0 is total ignorance—someone who has never heard of a data structure and is unaware that cleverly organizing your data can dramatically improve a program's running time.

Level 1: *"I hear good things about hash tables."*

Level 1 is cocktail party-level awareness—at this level, you could at least have a conversation about basic data structures. 2 You have heard of several basic structures like search trees and hash tables, and are perhaps aware of some of their supported operations, but would be shaky trying to use them in a program or a technical interview.

Level 2: *"This problem calls out for a heap."*

With level 2, we're starting to get somewhere. This is someone who has solid literacy about basic data structures, is comfortable using them as a client in their own programs, and has a good sense of which data structures are appropriate for which types of programming tasks.

Level 3: *"I use only data structures that I wrote myself."*

Level 3, the most advanced level, is for hardcore programmers and computer scientists who are not content to merely use existing data structure implementations as a client. At this level, you have a detailed understanding of the guts of basic data structures, and exactly how they are implemented.

The biggest marginal empowerment comes from reaching level 2. Most programmers will, at some point, need to be educated clients of basic data structures like heaps, search trees, and hash tables.

Supported Operations

A heap is a data structure that keeps track of an evolving set of objects with keys and can quickly identify the object with the smallest key. For example, objects might correspond to employee records, with keys equal to their identification numbers. They might be the edges of a graph, with keys corresponding to edge lengths. Or they could correspond to events scheduled for the future, with each key indicating the time at which the event will occur.

Insert and Extract-Min

The most important things to remember about any data structure are the operations it supports and the time required for each. The two most important operations supported by heaps are the Insert and ExtractMin operations.

Heaps: Basic Operations

Insert: given a heap H and a new object x, add x to H.

ExtractMin: given a heap H, remove and return from H an object with the smallest key (or a pointer to it).

For example, if you invoke Insert four times to add objects with keys 12, 7, 29, and 15 to an empty heap, the *ExtractMin* operation will return the object with key 7. Keys need not be distinct; if there is more than one object in a heap with the smallest key, the *ExtractMin* operation returns an arbitrary such object.

It would be easy to support only the Insert operation, by repeatedly tacking on new objects to the end of an array or linked list (in constant time). The catch is that *ExtractMin* would require a linear-time exhaustive search through all the objects. It's also clear how to support only *ExtractMin*—sort the initial set of n objects by key once and for all up front (using *O(n log n)* preprocessing time), and then successive calls to *ExtractMin* peel off objects from the beginning of the sorted list one by one (each in constant time). Here the catch is that any straightforward implementation of Insert requires linear time (as you should check). The trick is to design a data structure that enables both operations to run super-quickly. This is exactly the raison d'être of heaps.

Theorem 10.1 (Running Time of Basic Heap Operations)

In a heap with n objects, the Insert and *ExtractMin* operations run in *O(log n)* time.

As a bonus, in typical implementations, the constant hidden by the big-O notation is very small, and there is almost no extra space overhead.

There's also a heap variant that supports the Insert and *ExtractMax* operations in *O(log n)* time, where n is the number of objects. One way to implement this variant is to switch the direction of all the inequalities in the implementation. A second way is to use a standard heap but negate the keys of objects before inserting them (which effectively transforms *ExtractMin* into *ExtractMax*). Neither variant of a heap supports both *ExtractMin* and *ExtractMax* simultaneously in *O(log n)* time—you have to pick which one you want.

Additional Operations

Heaps can also support a number of less essential operations.

Heaps: Extra Operations

FindMin: given a heap H, return an object with the smallest key (or a pointer to it).

Heapify: given objects x 1 , . . . , x n , create a heap containing them.

Delete: given a heap H and a pointer to an object x in H, delete x from H.

You could simulate a *FindMin* operation by invoking *Extract-Min* and then applying Insert to the result (in *O(log n)* time), but a typical heap implementation can avoid this circuitous solution and support *FindMin* directly in *O(1)* time. You could implement *Heapify* by inserting the n objects one by one into an empty heap (in *O(n log n)* total time) but there's a slick way to add n objects to an empty heap in a batch in total time *O(n)*. Finally, heaps can also support deletions of arbitrary objects—not just an object with the smallest key—in *O(log n)* time.

Theorem 10.2 (Running Time of Extra Heap Operations)

In a heap with n objects, the *FindMin, Heapify*, and *Delete* operations run in *O(1), O(n), and O(log n)* time, respectively.

Summarizing, here's the final scorecard for heaps:

Operation	Running time
INSERT	$O(\log n)$
EXTRACTMIN	$O(\log n)$
FINDMIN	$O(1)$
HEAPIFY	$O(n)$
DELETE	$O(\log n)$

Heaps: supported operations and their running times, where n denotes the current number of objects stored in the heap.

When to Use a Heap

If your application requires fast minimum (or maximum) computations on a dynamically changing set of objects, the heap is usually the data structure of choice.

Applications

The next order of business is to walk through several example applications and develop a feel for what heaps are good for. The common theme of these applications is the replacement of minimum computations, naively implemented using (linear-time) exhaustive search, with a sequence of (logarithmic-time) *ExtractMin* operations from a heap. Whenever you see an algorithm or program with lots of brute-force minimum or maximum computations, a light bulb should go off in your head: This calls out for a heap!

Application: Sorting

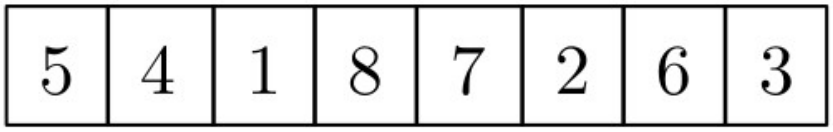
For our first application, let's return to the mother of all computational problems, sorting.

Problem: Sorting

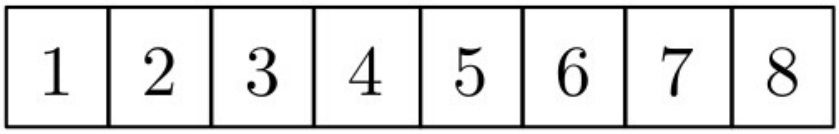
Input: An array of n numbers, in arbitrary order.

Output: An array of the same numbers, sorted from smallest to largest.

For example, given the input array



the desired output array is



Perhaps the simplest sorting algorithm is SelectionSort. This algorithm performs a linear scan through the input array to identify the minimum element, swaps it with the first element in the array, does a second scan over the remaining n 1 elements to identify and swap into the second position the second-smallest element, and so on. Each scan takes time proportional to the number of remaining elements, so the overall running time is →(ni=1 i) = →(n 2). 7 Because each iteration of SelectionSort computes a minimum element using exhaustive search, it calls out for a heap! The idea is simple: Insert all the elements in the input array into a heap, and populate the output array from left to right with successively

extracted minimum elements. The first extraction produces the smallest element; the second the smallest remaining element (the second-smallest overall); and so on.

HeapSort

Input: array A of n distinct integers.

Output: array B with the same integers, sorted from smallest to largest.

```
H := empty heap
for i = 1 to n do
        Insert A[i] into H
for i = 1 to n do
        B[i] := ExtractMin from H
```

Correct answer: (b). The work done by HeapSort boils down to 2n operations on a heap containing at most n objects. Because Theorem guarantees that every heap operation requires O(log n) time, the overall running time is O(n log n).

Theorem 10.3 (Running Time of HeapSort) For every input array of length n 1, the running time of HeapSort is O(n log n).

Let's take a step back and appreciate what just happened. We started with the least imaginative sorting algorithm possible, the quadratic-time SelectionSort algorithm. We recognized the pattern of repeated minimum computations, swapped in a heap data structure, and—boom!—out popped an O(n log n)-time sorting algorithm. This is a great running time for a sorting algorithm—it' seven optimal, up to constant factors, among comparison-based sorting algorithms. A neat byproduct of this observation is a proof that there's no comparison-based way to implement both the Insert and ExtractMin operations in better-than-logarithmic time: such a solution would yield a better-than-*O(n Log n)*-time comparison-based sorting algorithm, and we know this is impossible.

Application: Event Manager

Our second application, while a bit obvious, is both canonical and practical. Imagine you've been tasked with writing software that performs a simulation of the physical world. For example, perhaps you're contributing to a basketball video game. For the simulation, you must keep track of different events and when they should occur— the event that a player shoots the ball at a particular angle and velocity, that the ball consequently hits the back of the rim, that two players vie for the rebound at the same time, that one of these players commits an over-the-back foul on the other, and so on.

A simulation must repeatedly identify what happens next. This boils down to repeated minimum computations on the set of scheduled event times, so a light bulb should go off in your head: The problem calls out for a heap! If events are stored in a heap, with keys equal to their scheduled times, the ExtractMin operation hands you the next event on a silver platter, in logarithmic time. New events can be inserted into the heap as they arise (again, in logarithmic time).

Application: Median Maintenance

For a less obvious application of heaps, let's consider the median maintenance problem. You are presented with a sequence of numbers, one by one; assume for simplicity that they are distinct. Each time you receive a new number, your responsibility is to reply with the median element of all the numbers you've seen thus far. Thus, after seeing the first 11 numbers, you should reply with the sixth-smallest one you've seen; after 12, the sixth- or seventh-smallest; after 13, the seventh-smallest; and so on.

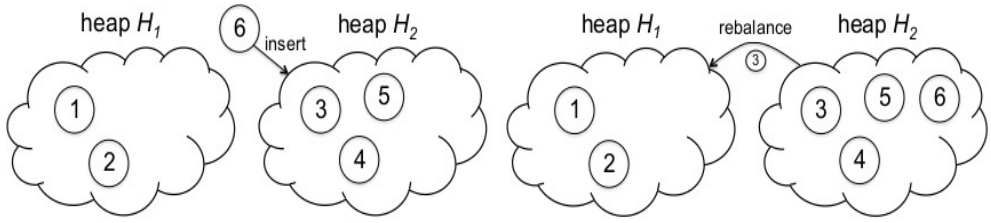
One approach to the problem, which should seem like overkill, is to recompute the median from scratch in every iteration. We could keep the elements seen so far in a sorted array, so that it's easy to compute the median element in constant time. The drawback is that updating the sorted array when a new number arrives can require linear time. Can we do better?

Using heaps, we can solve the median maintenance problem in just logarithmic time per round. I suggest putting the book down at this point and spending several minutes thinking about how this might be done.

The key idea is to maintain two heaps H 1 and H 2 while satisfying two invariants. The first invariant is that H 1 and H 2 are balanced, meaning they each contain the same number of elements (after an even round) or that one contains exactly one more element than the other (after an odd round). The second invariant is that H 1 and H 2 are ordered, meaning every element in H 1 is smaller than every element in H 2 . For example, if the numbers so far have been 1, 2, 3, 4, 5, then H 1 stores 1 and 2 and H 2 stores 4 and 5; the median element 3 is allowed to go in either one, as either the maximum element of H 1 or the minimum element of H 2 . If we've seen 1, 2, 3, 4, 5, 6, then the first three numbers are in H 1 and the second three are in H 2 ; both the maximum element of H 1 and the minimum element of H 2 are median elements. One twist: H 2 will be a standard heap, supporting Insert and ExtractMin, while H 1 will be the "max" variant described, supporting Insert and ExtractMax. This way, we can extract the median element with one heap operation, whether it's in H 1 or H 2 .

We still must explain how to update H 1 and H 2 each time a new element arrives so that they remain balanced and ordered. To figure out where to insert a new element x so that the heaps remain ordered, it's enough to compute the maximum element y in H 1 and the

minimum element z in H 2 . If x is less than y, it has to go in H 1 ; if it's more than z, it has to go in H 2 ; if it's in between, it can go in either one. Do H 1 and H 2 stay balanced even after x is inserted? Yes, except for one case: In an even round 2k, if x is inserted into the bigger heap (with k elements), this heap will contain k + 1 elements while the other contains only k 1 elements (Figure 10.1(a)). But this imbalance is easy to fix: Extract the maximum or minimum element from H 1 or H 2 , respectively (whichever contains more elements), and re-insert this element into the other heap (Figure 10.1(b)). The two heaps stay ordered (as you should check) and are now balanced as well. This solution uses a constant number of heap operations each round, for a running time of O(log i) in round i.



(a) Insertion can cause imbalance (b) Rebalancing

When inserting a new element causes the heap H 2 to have two more elements than H 1 , the smallest element in H 2 is extracted and re-inserted into H 1 to restore balance.

Heaps as Trees

There are two ways to visualize objects in a heap, as a tree (better for pictures and exposition) or as an array (better for an implementation).

Let's start with trees.

A heap can be viewed as a rooted binary tree—where each node has 0, 1, or 2 children—in which every level is as full as possible. When the number of objects stored is one less than a power of 2, every level is full . When the number of objects is between two such numbers, the only non-full layer is the last one, which is populated from left to right. A heap manages objects associated with keys so that the following heap property holds.

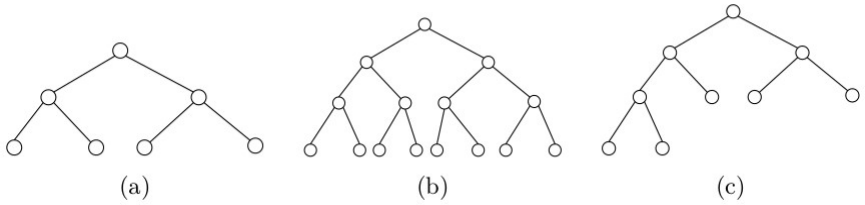
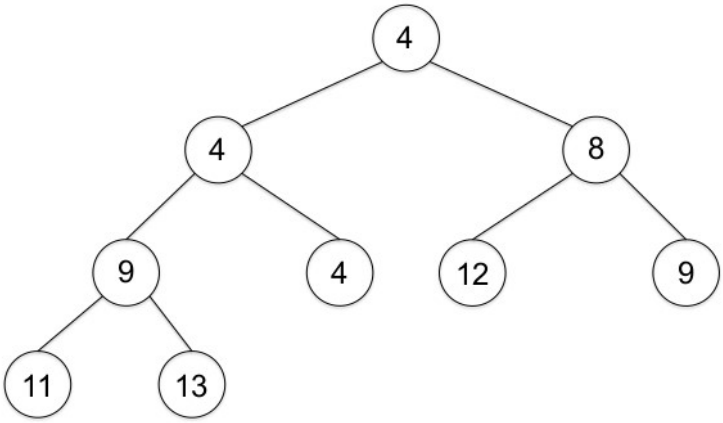


Fig: Full binary trees with 7, 15, and 9 nodes.

The Heap Property

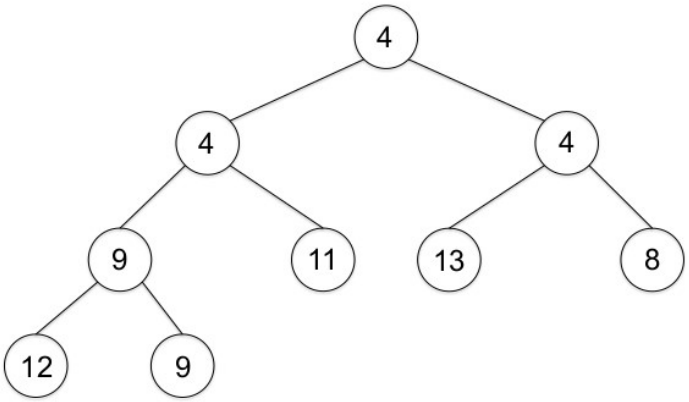
For every object x, the key of x is less than or equal to the keys of its children.

Duplicate keys are allowed. For example, here's a valid heap containing nine objects:



For every parent-child pair, the parent's key is at most that of the child.

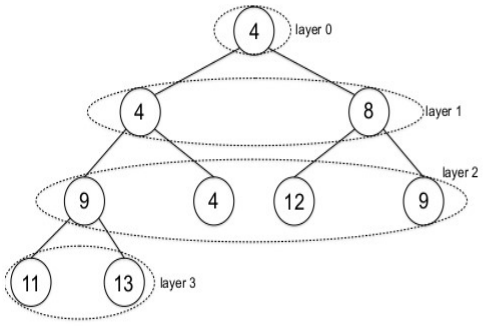
There's more than one way to arrange objects so that the heap property holds. Here's another heap, with the same set of keys:



Both heaps have a "4" at the root, which is also (tied for) the smallest of all the keys. This is not an accident: because keys only decrease as you traverse a heap upward, the root's key is as small as it gets. This should sound encouraging, given that the raison d'être of a heap is fast minimum computations.

Heaps as Arrays

In our minds we visualize a heap as a tree, but in an implementation we use an array with length equal to the maximum number of objects we expect to store. The first element of the array corresponds to the tree's root, the next two elements to the next level of the tree (in the same order), and so on



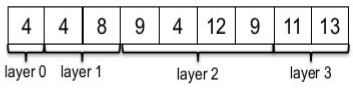


Fig: Tree representation array representation

Parent-child relationships in the tree translate nicely to the array. Assuming the array positions are labeled 1, 2, . . . , n, where n is the number of objects, the children of the object in position i correspond to the objects in positions 2i and 2i + 1 (if any). For example, in Figure 10.5, the children of the root (in position 1) are the next two objects (in positions 2 and 3), the children of the 8 (in position 3) are the objects in positions 6 and 7, and so on. Going in reverse, for a non-root object (in position i 2), i's parent is the object in position bi/2c. 20 For example, in Figure 10.5, the parent of the last object (in position 9) is the object in position b9/2c = 4.

Position of parent	$\lfloor i/2 \rfloor$ (provided $i \geq 2$)
Position of left child	$2i$ (provided $2i \leq n$)
Position of right child	$2i + 1$ (provided $2i + 1 \leq n$)

Relationships between the position i 2 {1, 2, 3, . . . , n} of an object in a heap and the positions of its parent, left child, and right child, where n denotes the number of objects in the heap.

There are such simple formulas to go from a child to its parent and back because we use only full binary trees. There is no need to explicitly store the tree; consequently, the heap data structure has minimal space overhead.

Implementing Insert in O(log n) Time

We'll illustrate the implementation of both the Insert and Extract- Min operations by example rather than by pseudocode. The challenge is to both keep the tree full and maintain the heap property after an object is added or removed. We'll follow the same blueprint for both operations:

1. Keep the tree full in the most obvious way possible.

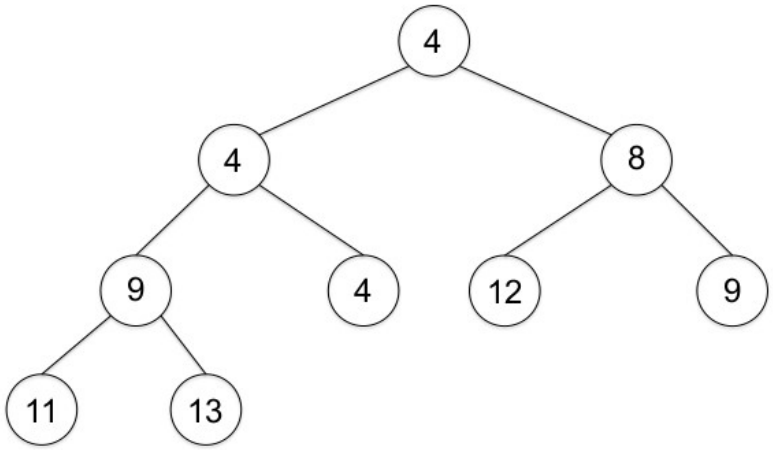
2. Play whack-a-mole to systematically squash any violations of the heap property.

Specifically, recall the Insert operation:

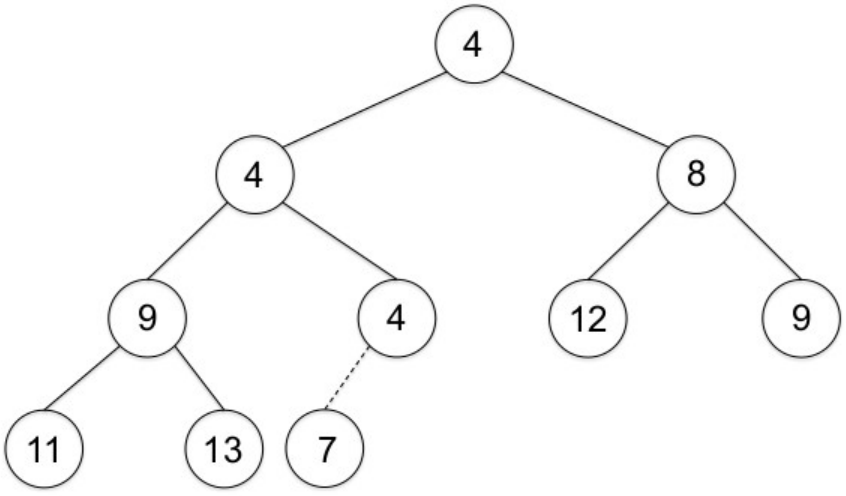
given a heap H and a new object x, add x to H.

After x's addition to H, H should still correspond to a full binary tree (with one more node than before) that satisfies the heap property. The operation should take *O(log n)* time, where n is the number of objects in the heap.

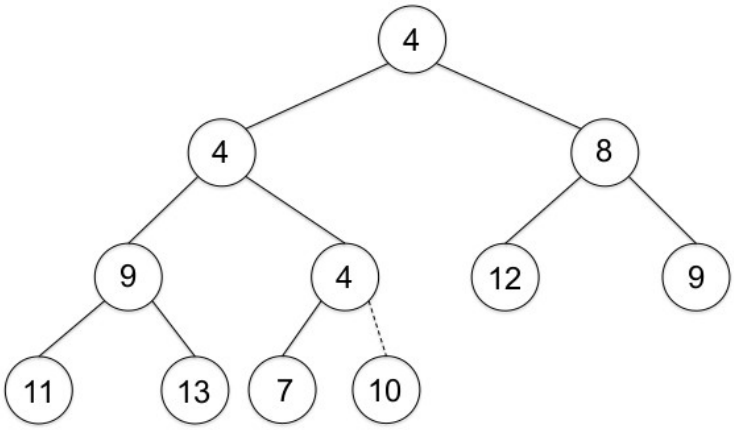
Let's start with our running example:



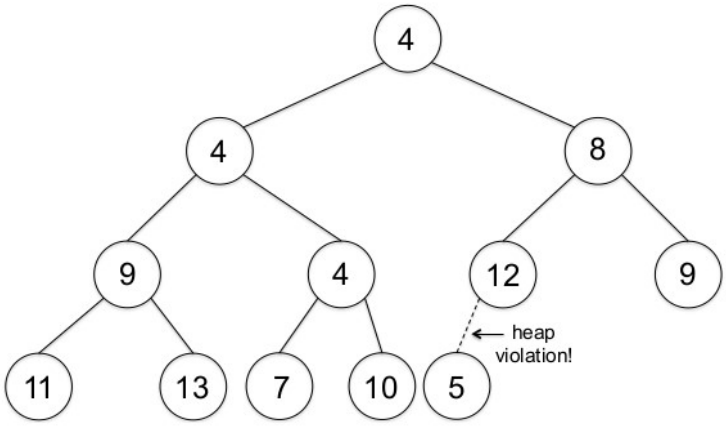
When a new object is inserted, the most obvious way to keep the tree full is to tack the new object onto the end of the array, or equivalently to the last level of the tree. (If the last level is already full, the object becomes the first at a new level.) As long as the implementation keeps track of the number n of objects (which is easy to do), this step takes constant time. For example, if we insert an object with key 7 into our running example, we obtain:



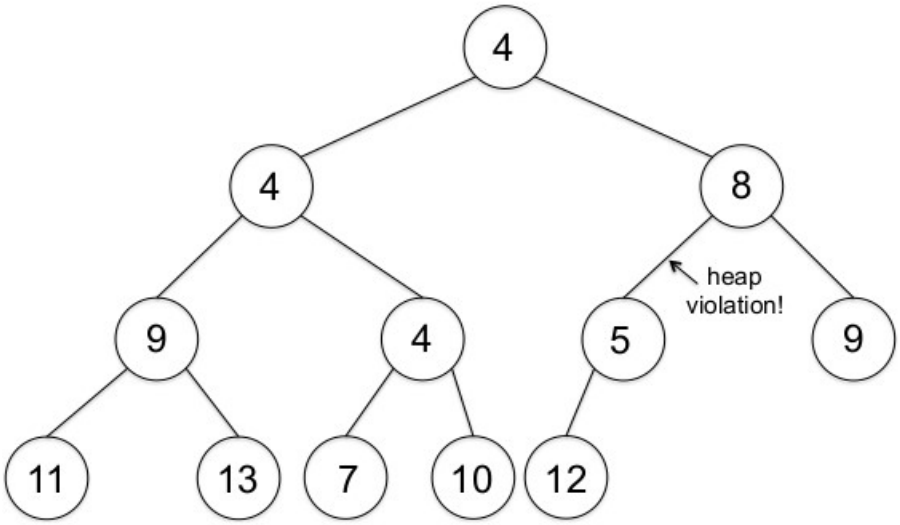
We have a full binary tree, but does the heap property hold? There's only one place it might fail—the one new parent-child pair (the 4 and the 7). In this case we got lucky, and the new pair doesn't violate the heap property. If our next insertion is an object with key 10, then again we get lucky and immediately obtain a valid heap:



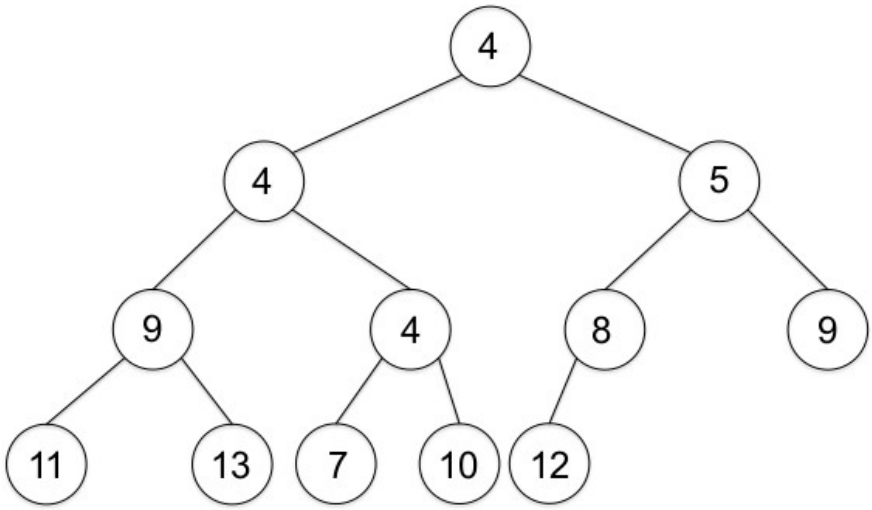
But suppose we now insert an object with key 5. After tacking it on at the end, our tree is:



Now we have a problem: The new parent-child pair (the 12 and the 5) violates the heap property. What can we do about it? We can at least fix the problem locally by swapping the two nodes in the violating pair:



This fixes the violating parent-child pair. We're not out of the woods yet, however, as the heap violation has migrated upward to the 8 and the 5. So we do it again, and swap the nodes in the violating pair to obtain:



This explicitly fixes the violating pair. We've seen that such a swap has the potential to push the violation of the heap property upward, but here it doesn't happen—the 4 and 5 are already in the correct order. You might worry that a swap could also push the violation downward. But this also doesn't happen—the 8 and 12 are already in the correct order. With the heap property restored, the insertion is complete.

In general, the Insert operation tacks the new object on to the end of the heap, and repeatedly swaps the nodes of a violating pair. At all times, there is at most one violating parent-child pair—the pair in which the new object is the child. Each swap pushes the

violating parent-child pair up one level in the tree. This process cannot go on forever—if the new object makes it to the root, it has no parent and there can be no violating parent-child pair.

Insert

1. Stick the new object at the end of the heap and increment the heap size.

2. Repeatedly swap the new object with its parent until the heap property is restored.

Because a heap is a full binary tree, it has ⇑ log 2 n levels, where n is the number of objects in the heap. The number of swaps is at most the number of levels, and only a constant amount of work is required per swap. We conclude that the worst-case running time of the Insert operation is *O(log n)*, as desired.

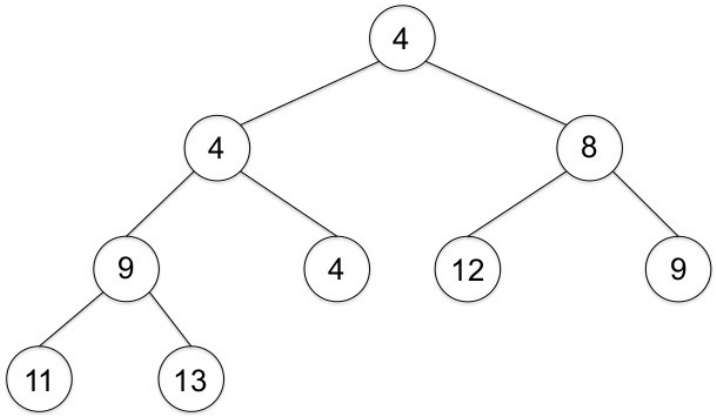
Implementing ExtractMin in *O(log n)* Time

Recall the ExtractMin operation:

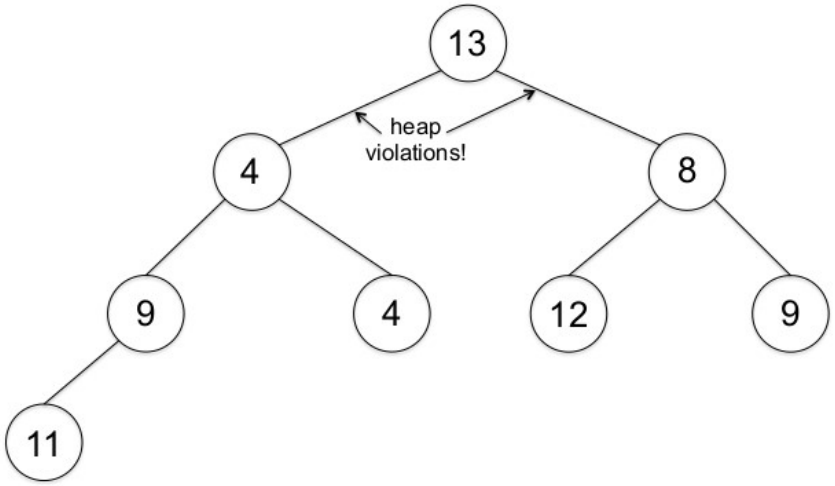
given a heap H, remove and return from H an object with the smallest key.

The root of the heap is guaranteed to be such an object. The challenge is to restore the full binary tree and heap properties after ripping out a heap's root.

We again keep the tree full in the most obvious way possible. Like Insert in reverse, we know that the last node of the tree must go elsewhere. But where should it go? Because we're extracting the root anyway, let's overwrite the old root node with what used to be the last node. For example, starting from the heap

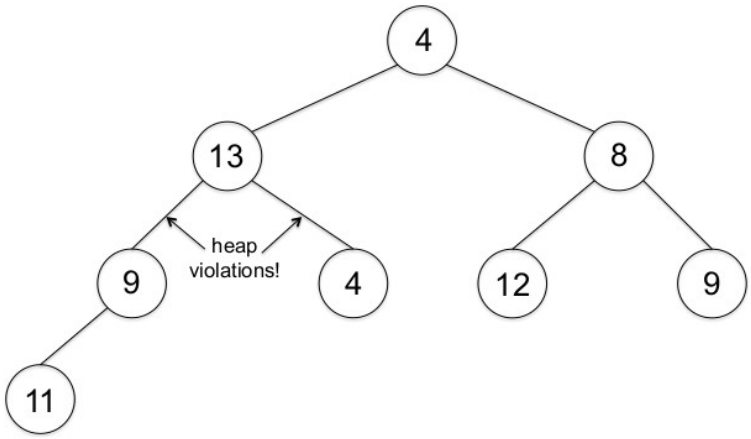


the resulting tree looks like

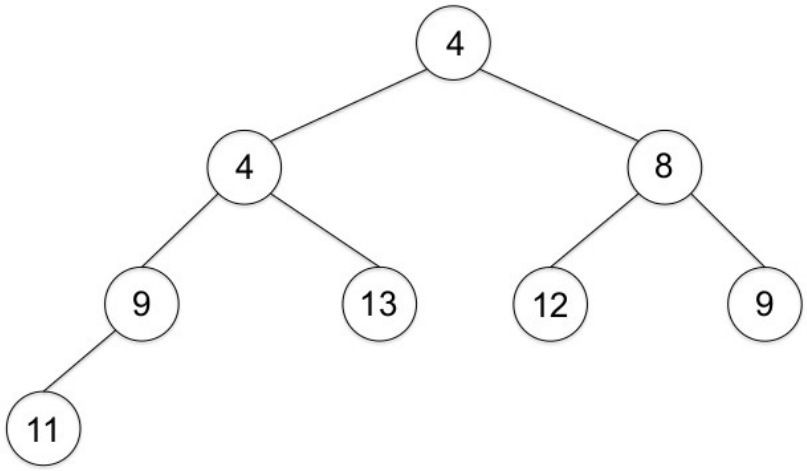


The good news is that we've restored the full binary tree property. The bad news is that the massive promotion granted to the object with key 13 has created two violating parent-child pairs (the 13 and 4 and the 13 and 8). Do we need two swaps to correct them?

The key idea is to swap the root node with the smaller of its two children:



There are no longer any heap violations involving the root—the new root node is smaller than both the node it replaced (that's why we swapped) and its other child (as we swapped the smaller child). 26 The heap violations migrate downward, again involving the object with key 13 and its two (new) children. So we do it again, and swap the 13 with its smaller child:



The heap property is restored at last, and now the extraction is complete.

In general, the ExtractMin operation moves the last object of a heap to the root node (by overwriting the previous root), and repeatedly swaps this object with its smaller child. At all times, there are at most two violating parent-child pairs—the two pairs in which the formerly-last object is the parent. Because each swap pushes this object down one level in the tree, this process cannot go on forever—it stops once the new object belongs to the last level, if not earlier.

ExtractMin

1. Overwrite the root with the last object x in the heap, and decrement the heap size.

2. Repeatedly swap x with its smaller child until the heap property is restored.

The number of swaps is at most the number of levels, and only a constant amount of work is required per swap. Because there are ⇑ log 2 n levels, we conclude that the worst-case running time of the ExtractMin operation is *O(log n)*, where n is the number of objects in the heap.

The upshot:

1. There are many different data structures, each optimized for a different set of operations.
2. The principle of parsimony recommends choosing the simplest data structure that supports all the operations required by your application.
3. If your application requires fast minimum (or maximum) computations on an evolving set of objects, the heap is usually the data structure of choice.

4. The two most important heap operations, Insert and ExtractMin, run in *O(log n)* time, where n is the number of objects.
5. Heaps also support FindMin in *O(1)* time, Delete in *O(log n)* time, and Heapify in *O(n)* time.
6. The HeapSort algorithm uses a heap to sort a length-n array in *O(n log n)* time.
7. Heaps can be visualized as full binary trees but are implemented as arrays.
8. The heap property states that the key of every object is less than or equal to the keys of its children.
9. The Insert and ExtractMin operations are implemented by keeping the tree full in the most obvious way possible and systematically squashing any violations of the heap property.

QUESTION:

What's the running time of HeapSort, as a function of the length n of the input array?

a) O(n)

b) O(n log n)

c) $O(n^2)$

d) $O(n^2 \log n)$

Solution: option (b) as heapsort have the worst and average case time complexity of

O(n log n) while the best is O(1).

CHAPTER 6

SORTING

Sorting Algorithms are methods of reorganizing a large number of items into some specific order such as highest to lowest, or vice-versa, or even in some alphabetical order.

These algorithms take an input list, processes it (i.e, performs some operations on it) and produce the sorted list.

The most common example we experience every day is sorting clothes or other items on an e-commerce website either by lowest-price to highest, or list by popularity, or some other order.

What is bubble sort?

Definition of bubble sort

Bubble sort is one of the fundamental forms of sorting in programming. Bubble sort algorithms move through a sequence of data (typically integers) and rearrange them into ascending or descending order one number at a time. To do this, the algorithm compares number X to the adjacent number Y. If X is higher than Y, the two are swapped and the algorithm starts over.

This process repeats until the entire range of numbers has been sorted in the desired order. For instance, if you were trying to arrange [1, 3, 2, 4] into ascending order, the bubble sort algorithm would run once, swapping the 3 with the 2.

In another matrix, however, your numbers might look like this: [3, 1, 4, 2]. In this case, the algorithm would run three times, swapping the 3 and the 1 the first time, then the 4 and the 2 the second time, and finally the 3 and the 2.

The name bubble sort comes from the fact that smaller or larger elements "bubble" to the top of a dataset. In the previous example of [3, 1, 4, 2], the 3 and 4 are bubbling up the dataset to find their proper positions.

Algorithm:

begin BubbleSort(arr)

```
  for all array elements
    if arr[i] > arr[i+1]
      swap(arr[i], arr[i+1])
    end if
  end for
  return arr
end BubbleSort
```

Another example:

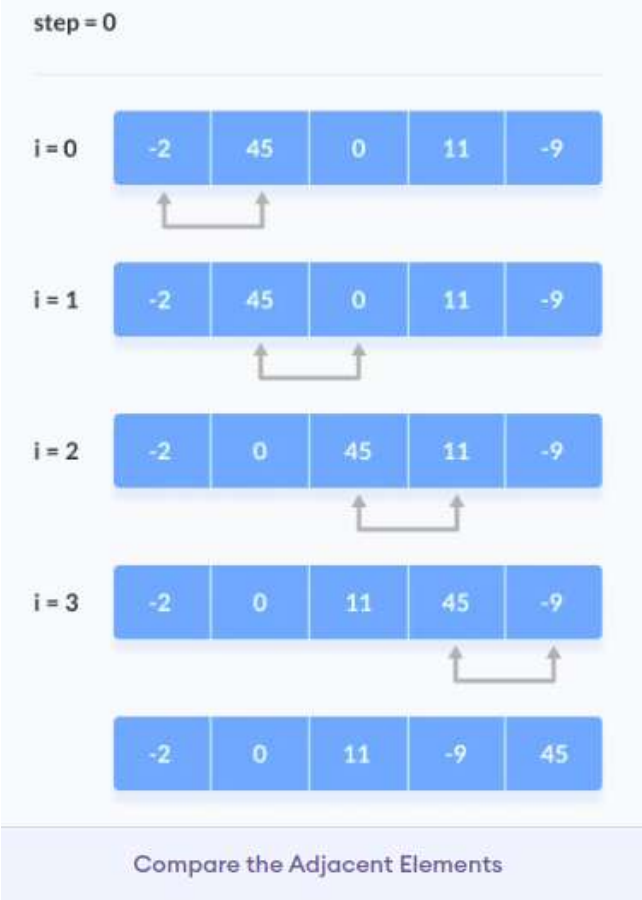
This algorithm is alternatively called the sinking sort for the opposite reason; some of the elements are sinking to the bottom of the dataset. In our example, the 1 and the 2 are sinking elements.

Working of Bubble Sort

Suppose we are trying to sort the elements in **ascending order**.

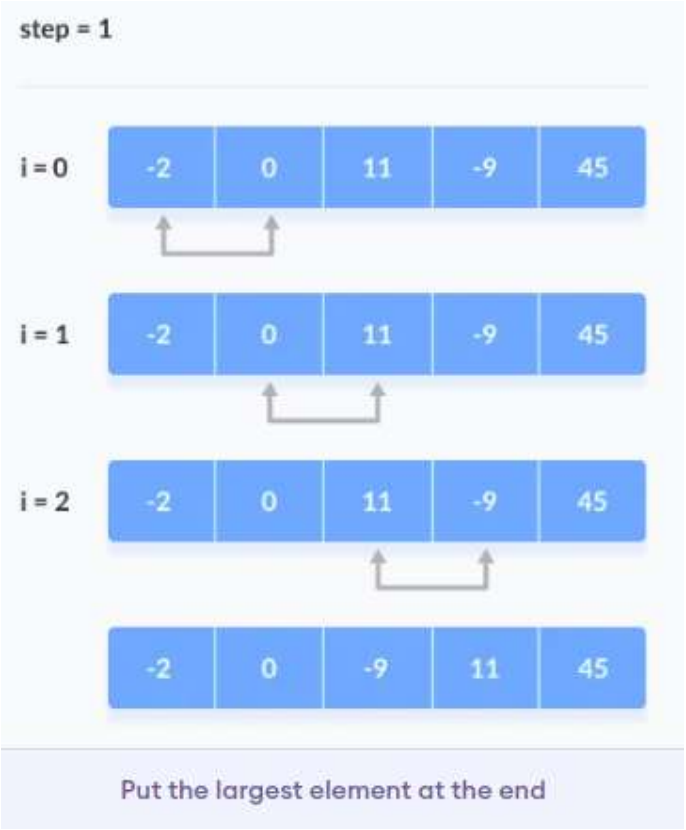
1. First Iteration (Compare and Swap)

1. Starting from the first index, compare the first and the second elements.
2. If the first element is greater than the second element, they are swapped.
3. Now, compare the second and the third elements. Swap them if they are not in order.
4. The above process goes on until the last element.

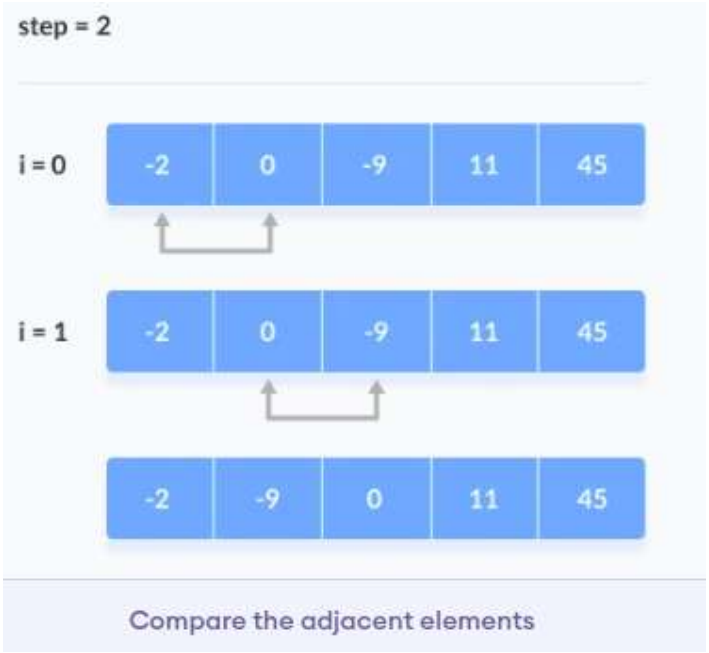


2. Remaining Iteration

The same process goes on for the remaining iterations.
After each iteration, the largest element among the unsorted elements is placed at the end.



In each iteration, the comparison takes place up to the last unsorted element.



The array is sorted when all the unsorted elements are placed at their correct positions.

Advantages of using bubble sort
Bubble sort's strong point is its simplicity. It takes just a few lines of code, is easy to read, and can be plugged in anywhere in your program. However, it's extremely inefficient for larger sets of numbers and should be used accordingly.

Quicksort

Quicksort is a sorting algorithm based on the **divide and conquer approach** where

1. An array is divided into subarrays by selecting a **pivot element** (element selected from the array).

   While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.

2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Working of Quicksort Algorithm

1. Select the Pivot Element

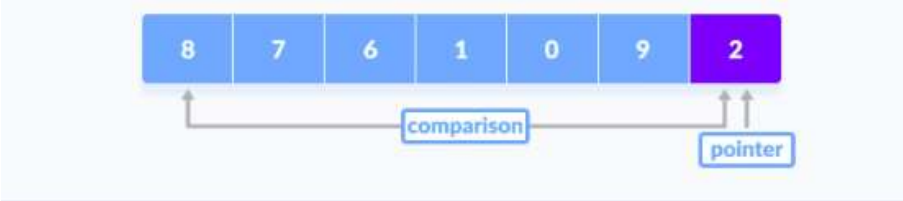
There are different variations of quicksort where the pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element.

2. Rearrange the Array

Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.

Here's how we rearrange the array:

1. A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index.

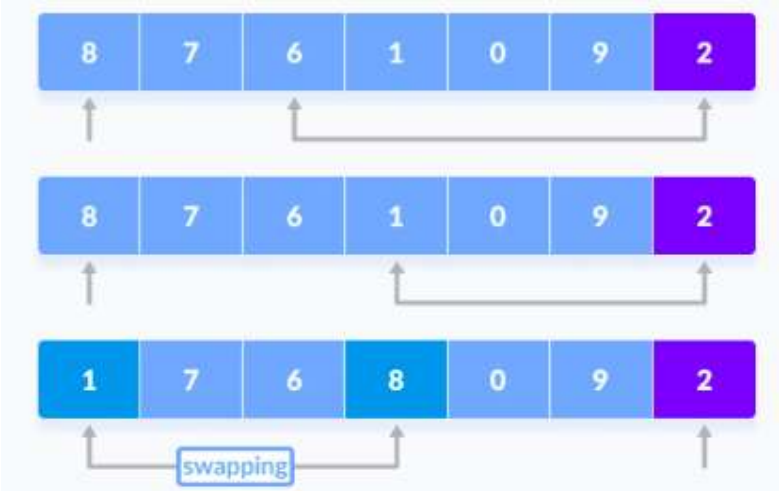


Comparison of pivot element with element beginning from the first index

2. If the element is greater than the pivot element, a second pointer is set for that element.

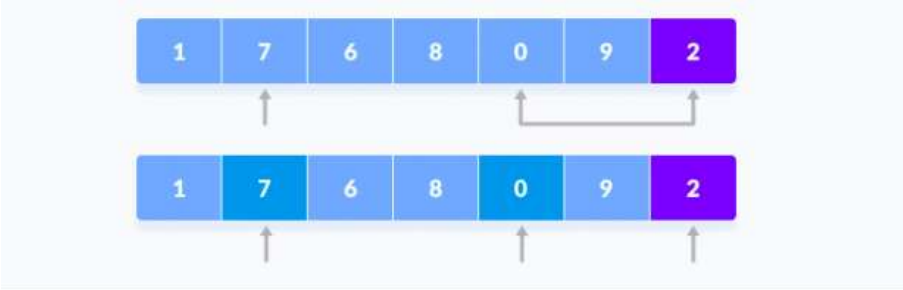
If the element is greater than the pivot element, a second pointer is set for that element.

3. Now, pivot is compared with other elements. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.



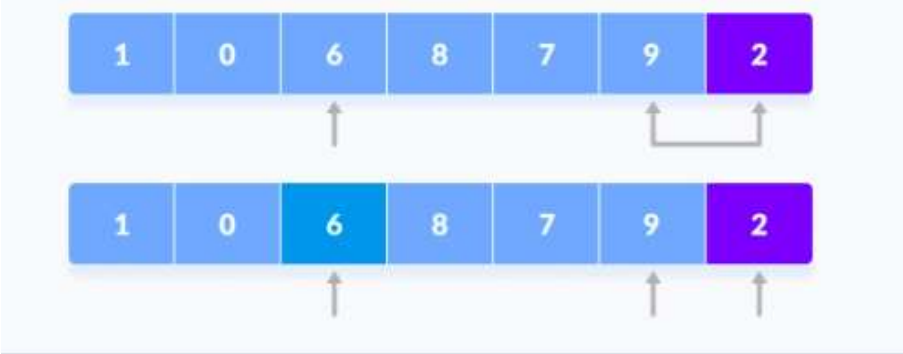
Pivot is compared with other elements.

4. Again, the process is repeated to set the next greater element as the second pointer. And, swap it with another smaller element.



The process is repeated to set the next greater element as the second pointer.

5. The process goes on until the second last element is reached.



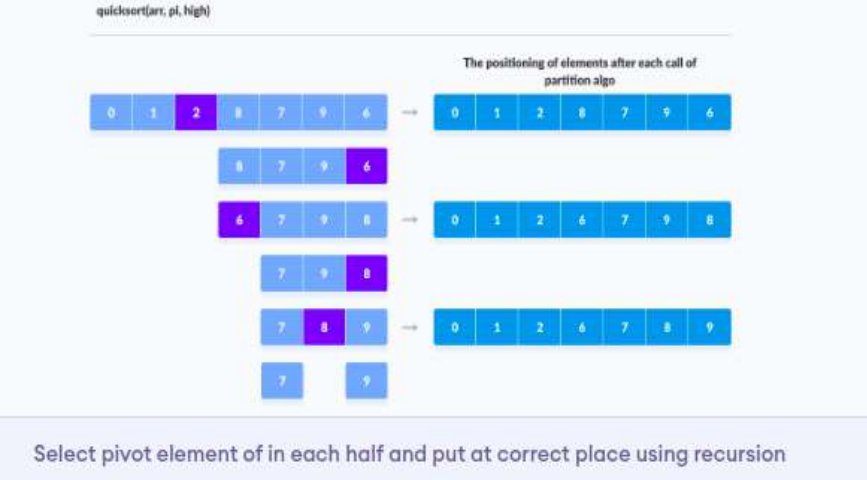
The process goes on until the second last element is reached.

6. Finally, the pivot element is swapped with the second pointer.

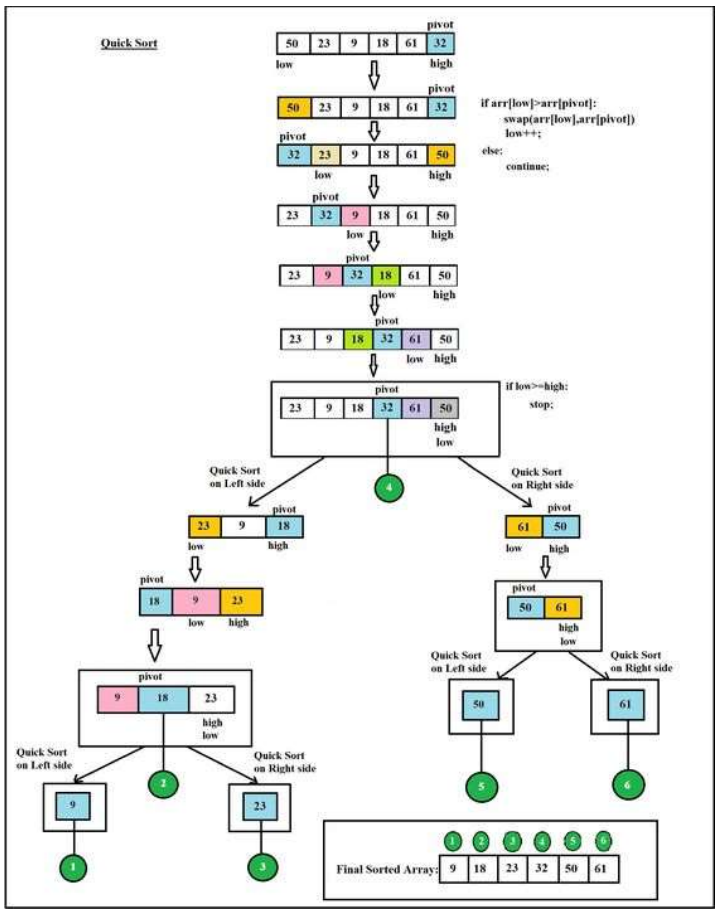
Finally, the pivot element is swapped with the second pointer.

3. Divide Subarrays

Pivot elements are again chosen for the left and the right sub-parts separately. And, **step 2** is repeated.



The subarrays are divided until each subarray is formed of a single element. At this point, the array is already sorted.



Quick Sort Algorithm

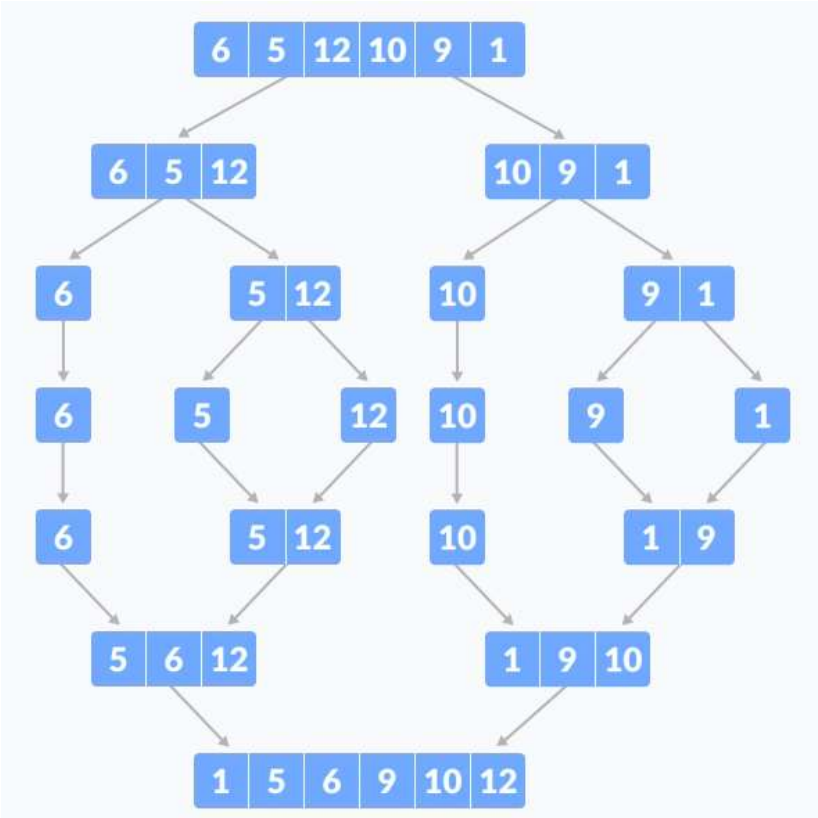
```
//Quick Sort Logic
void quickSort(int list[10],int first,int last){
 int pivot,i,j,temp;
  if(first < last){
          pivot = first;
                i = first;
                   j = last;
        while(i < j){
                    while(list[i] <= list[pivot] && i < last)
                                i++;
              while(list[j] && list[pivot])
                j--;
          if(i < j){
                              temp = list[i];
                    list[i] = list[j];
                              list[j] = temp;
                             }
          }
temp = list[pivot];
      list[pivot] = list[j];
        list[j] = temp;        quickSort(list,first,j-1);
quickSort(list,j+1,last);
  }
```

}

MERGE SORT

Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.

Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.



Merge Sort example

Divide and Conquer Strategy

Using the **Divide and Conquer** technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].

Divide

If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

Conquer

In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

Combine

When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r].

MergeSort Algorithm

The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1 i.e. p == r.

After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.

```
MergeSort(A, p, r):
  if p > r
```

```
        return
q = (p+r)/2
mergeSort(A, p, q)
mergeSort(A, q+1, r)
merge(A, p, q, r)
```

To sort an entire array, we need to call MergeSort(A, 0, length(A)-1).

As shown in the image below, the merge sort algorithm recursively divides the array into halves until we reach the base case of array with 1 element. After that, the merge function picks up the sorted sub-arrays and merges them to gradually sort the entire array.

Merge sort in action

The merge Step of Merge Sort

Every recursive algorithm is dependent on a base case and the ability to combine the results from base cases. Merge sort is no different. The most important part of the merge sort algorithm is, you guessed it, merge step.

The merge step is the solution to the simple problem of merging two sorted lists(arrays) to build one large sorted list(array).

The algorithm maintains three pointers, one for each of the two arrays and one for maintaining the current index of the final sorted array.

Have we reached the end of any of the arrays?

No:

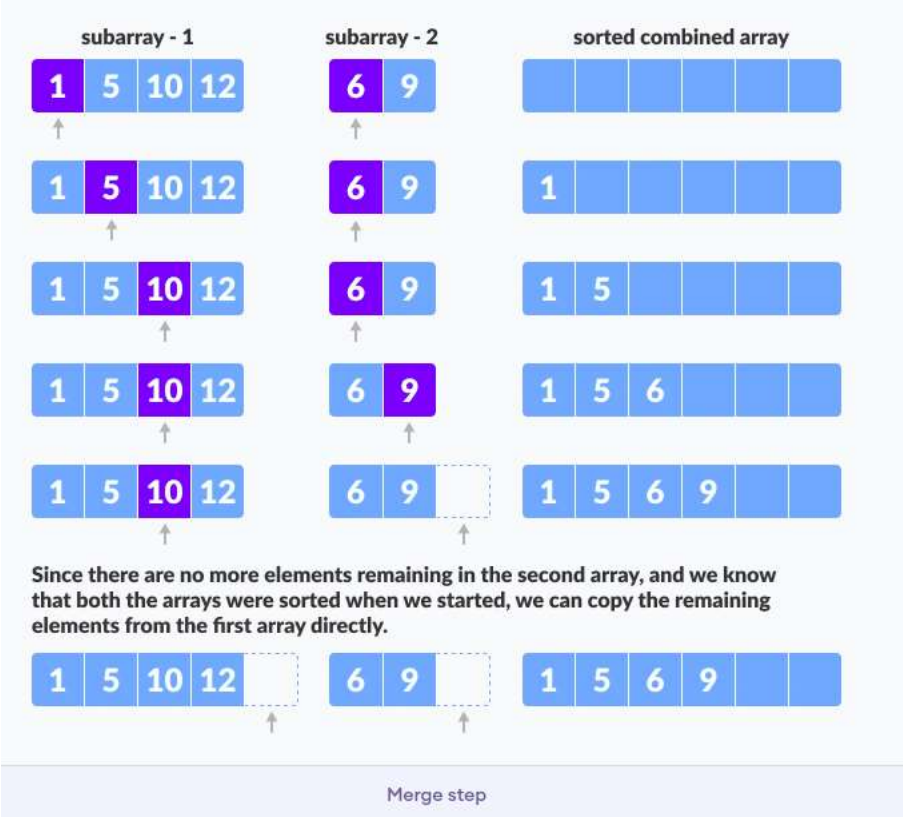
Compare current elements of both arrays

Copy smaller element into sorted array

Move pointer of element containing smaller element

Yes:

Copy all remaining elements of non-empty array



Writing the Code for Merge Algorithm

A noticeable difference between the merging step we described above and the one we use for merge sort is that we only perform the merge function on consecutive sub-arrays.

This is why we only need the array, the first position, the last index of the first subarray(we can calculate the first index of the second subarray) and the last index of the second subarray.

Our task is to merge two subarrays A[p..q] and A[q+1..r] to create a sorted array A[p..r]. So the inputs to the function are A, p, q and r

The merge function works as follows:

4. Create copies of the subarrays L ← A[p..q] and M ← A[q+1..r].
5. Create three pointers i, j and k
    a. i maintains current index of L, starting at 1
    b. j maintains current index of M, starting at 1
    c. k maintains the current index of A[p..q], starting at p.
6. Until we reach the end of either L or M, pick the larger among the elements from L and M and place them in the correct position at A[p..q]
7. When we run out of elements in either L or M, pick up the remaining elements and put in A[p..q]

```
// Merge two subarrays L and M into arr
void merge(int arr[], int p, int q, int r) {

    // Create L ← A[p..q] and M ← A[q+1..r]
    int n1 = q - p + 1;
    int n2 = r - q;

    int L[n1], M[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[p + i];
    for (int j = 0; j < n2; j++)
        M[j] = arr[q + 1 + j];

    // Maintain current index of sub-arrays and main array
    int i, j, k;
    i = 0;
    j = 0;
    k = p;

    // Until we reach either end of either L or M, pick larger among
    // elements L and M and place them in the correct position at A[p..r]
    while (i < n1 && j < n2) {
        if (L[i] <= M[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = M[j];
            j++;
        }
        k++;
    }

    // When we run out of elements in either L or M,
    // pick up the remaining elements and put in A[p..r]
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = M[j];
        j++;
        k++;
    }
}
```

Merge( ) Function Explained Step-By-Step

A lot is happening in this function, so let's take an example to see how this would work.

As usual, a picture speaks a thousand words.

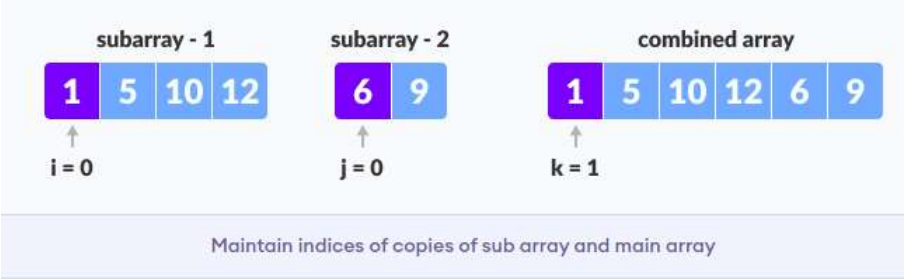
The array A[0..5] contains two sorted subarrays A[0..3] and A[4..5]. Let us see how the merge function will merge the two arrays.

Step 1: Create duplicate copies of sub-arrays to be sorted

```
// M[0,1] = A[4,5] = [6,9]
```

Step 2: Maintain current index of sub-arrays and main array

```
int i, j, k;
i = 0;
j = 0;
k = p;
```



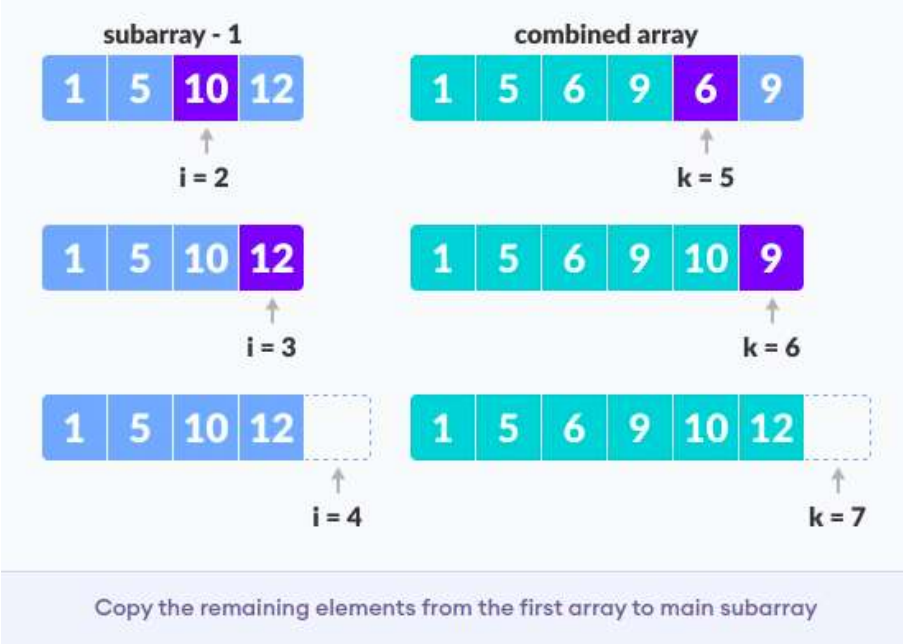
Maintain indices of copies of sub array and main array

Step 3: Until we reach the end of either L or M, pick larger among elements L and M and place them in the correct position at A[p..r]

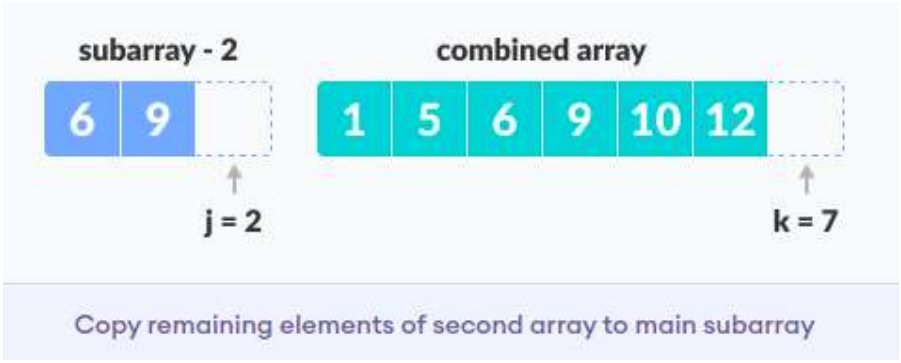
```
while (i < n1 && j < n2) {
    if (L[i] <= M[j]) {
        arr[k] = L[i]; i++;
    }
    else {
        arr[k] = M[j];
        j++;
    }
    k++;
}
```

Step 4: When we run out of elements in either L or M, pick up the remaining elements and put in A[p..r]

```
// We exited the earlier loop because j < n2 doesn't hold
  while (i < n1)
  {
      arr[k] = L[i];
      i++;
      k++;
  }
```




```
// We exited the earlier loop because i < n1 doesn't hold
    while (j < n2)
    {
        arr[k] = M[j];
        j++;
        k++;
    }
}
```



This step would have been needed if the size of M was greater than L.

At the end of the merge function, the subarray A[p..r] is sorted.

Insertion Sort Algorithm

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

Insertion sort works similarly as we sort cards in our hand in a card game.

We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

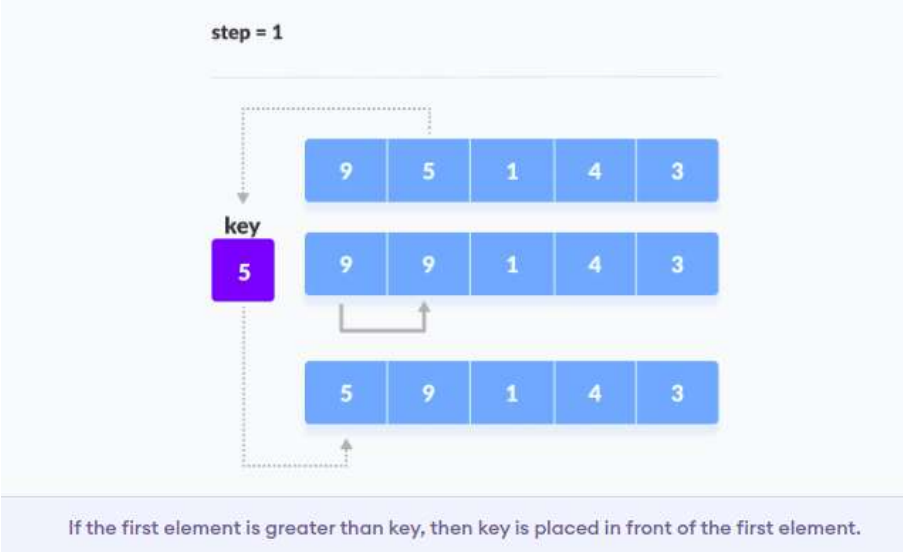
A similar approach is used by insertion sort.

Working of Insertion Sort

Suppose we need to sort the following array.

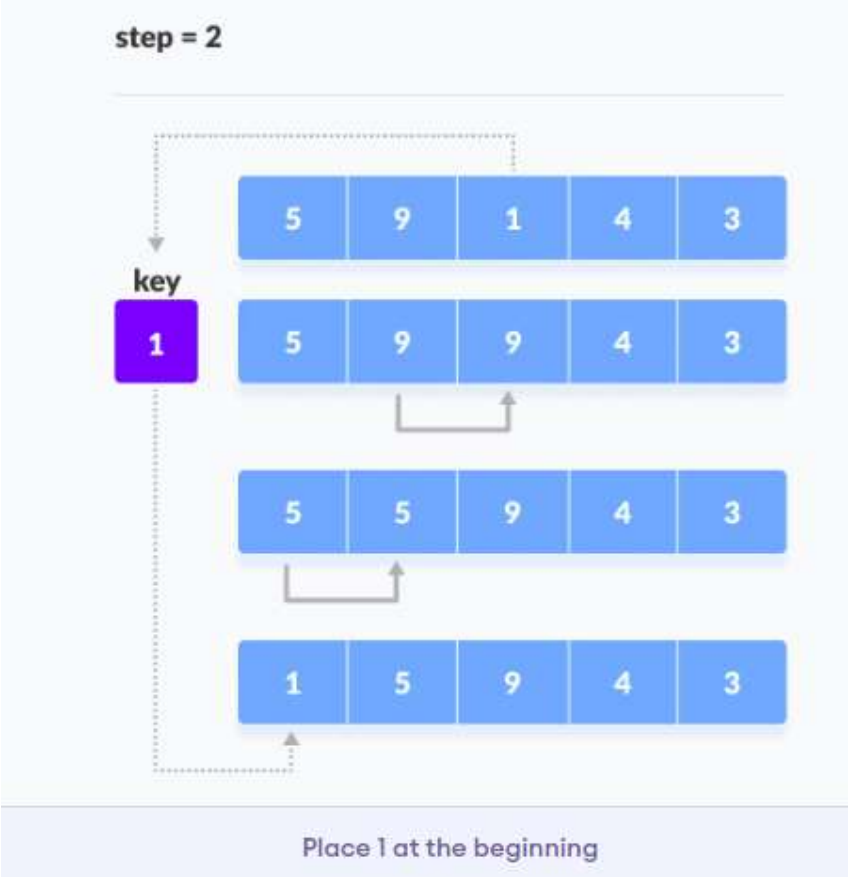
1. The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

   Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.

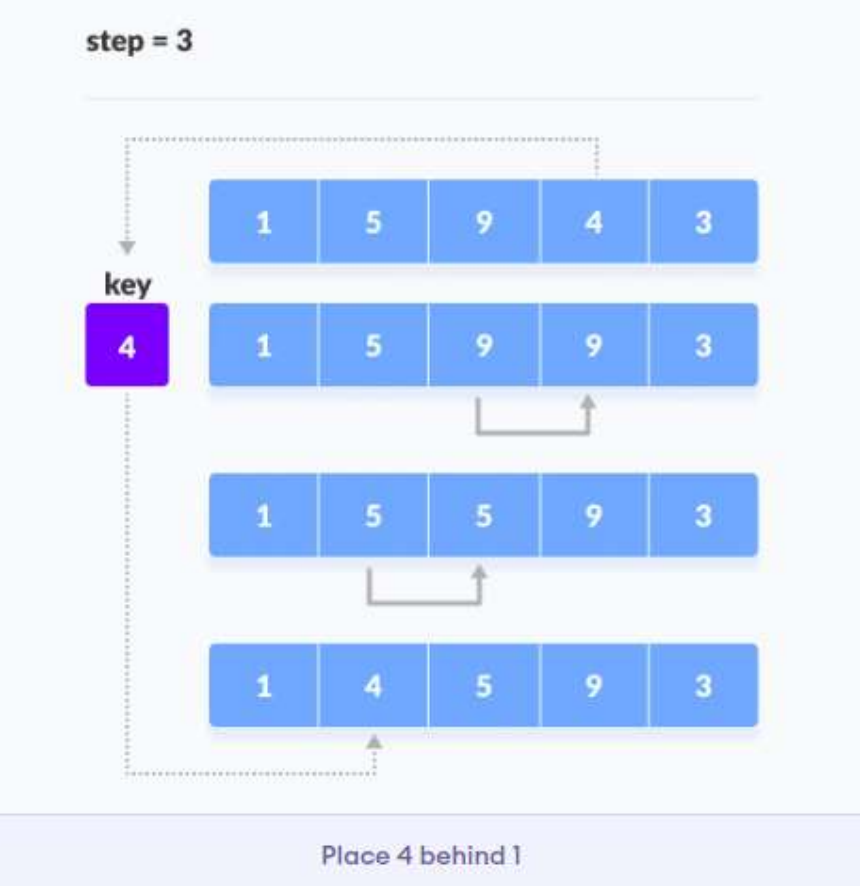


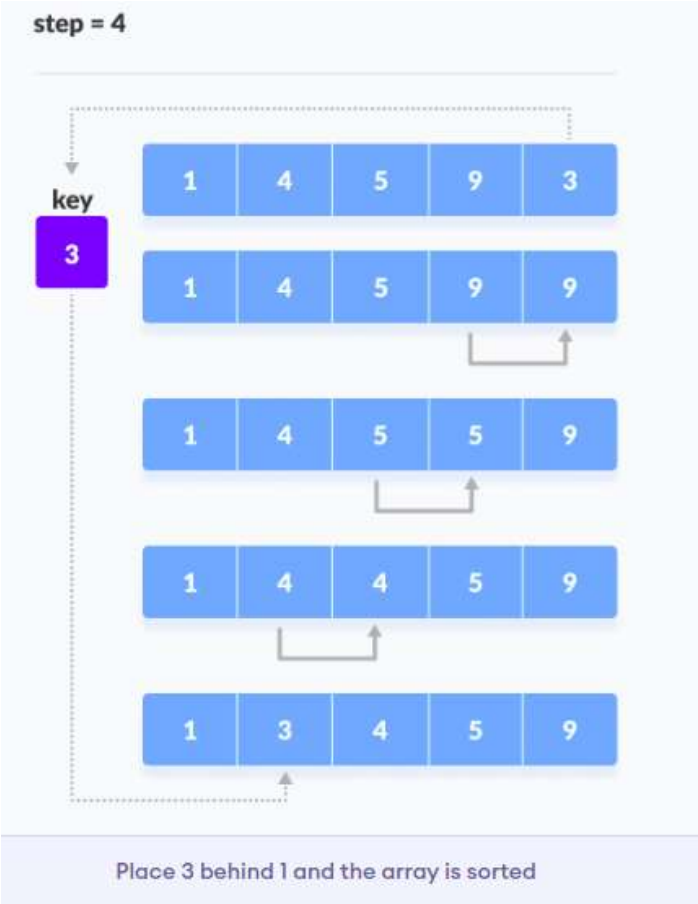
2. Now, the first two elements are sorted.

   Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.



3. Similarly, place every unsorted element at its correct position.






```
insertionSort(array)
  mark first element as sorted
  for each unsorted element X
    'extract' the element X
    for j <- lastSortedIndex down to 0
      if current element j > X
        move sorted element to the right by 1
    break loop and insert X here
end insertionSort
```

Selection Sort Algorithm

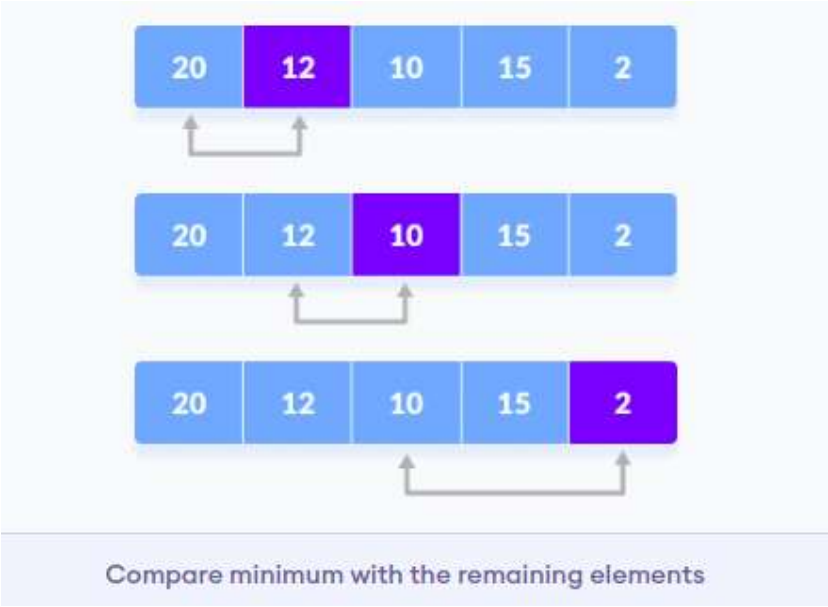
Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

Working of Selection Sort

1. Set the first element as minimum.

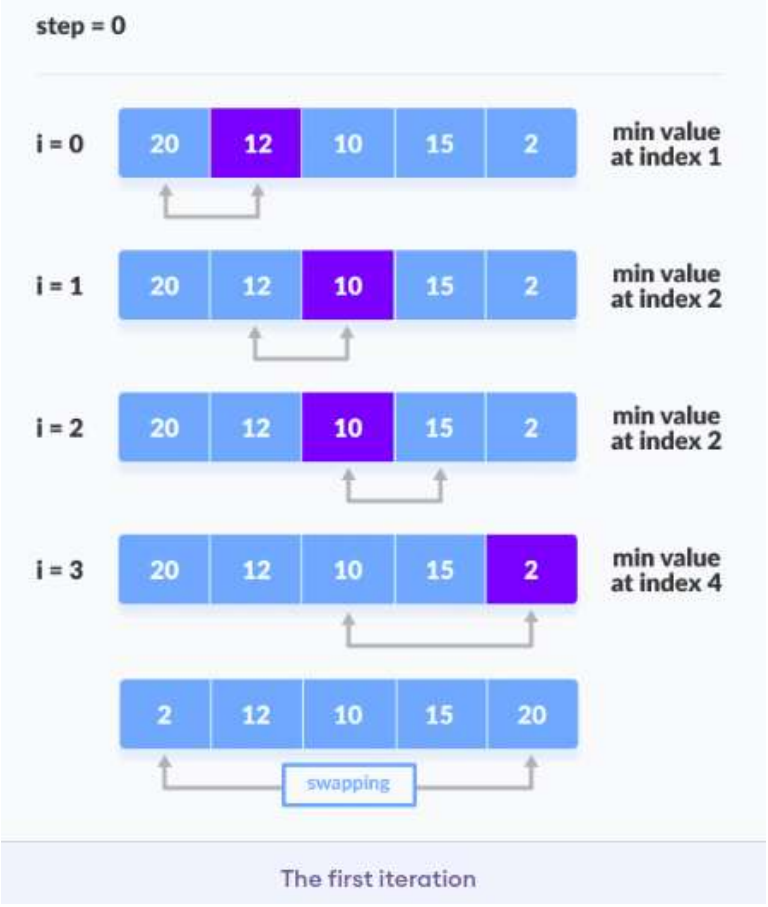
2. Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum.

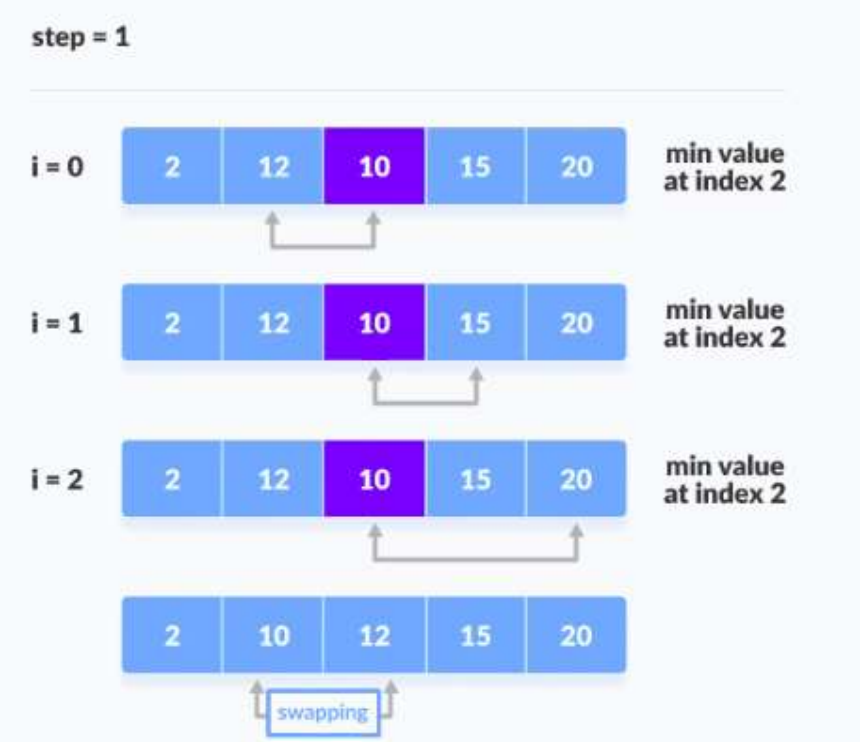
   Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.



3. After each iteration, minimum is placed in the front of the unsorted list.

4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.





The second iteration

The third iteration

Selection Sort Algorithm

```
selectionSort(array, size)
  repeat (size - 1) times
  set the first unsorted element as the minimum
  for each of the unsorted elements
    if element < currentMinimum
      set element as new minimum
  swap minimum with first unsorted position
end selectionSort
```

The crux of all sorting algos

Algorithm	Best	Average	Worst
Quick Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)
Merge Sort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)
Heap Sort	Ω(n log(n))	Θ(n log(n))	O(n log(n))

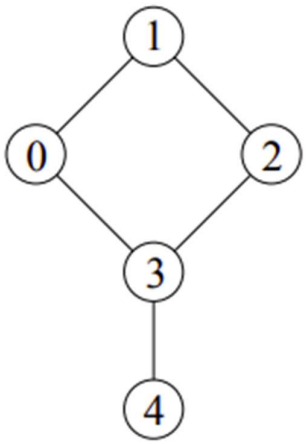
CHAPTER 7

GRAPH ALGORITHMS

GRAPH SEARCH

We can think of graphs as generalizations of trees: they consist of nodes and edges connecting nodes. The main difference is that graphs do not in general represent hierarchical organizations.

Types of graphs. Different applications require different types of graphs. The most basic type is the simple undirected graph that consists of a set V of vertices and a set E of edges. Each edge is an unordered pair (a set) of two vertices. We always assume V is finite, and we write $V\ 2$ for the collection of all unordered pairs.



A simple undirected graph with vertices 0, 1, 2, 3, 4 and edges {0, 1}, {1, 2}, {2, 3}, {3, 0}, {3, 4}.

Hence E is a subset of $V\ 2$. Note that because E is a set, each edge can occur only once. Similarly, because each edge is a set (of two vertices), it cannot connect to the same vertex twice. Vertices u and v are adjacent if $\{u, v\} \in E$. In this case u and v are called neighbors. Other types of graphs are

directed: $E \subseteq V \times V$.

weighted: has a weighting function $w : E \to R$.

labeled: has a labeling function $\ell : V \to Z$.

non-simple: there are loops and multi-edges.

A loop is like an edge, except that it connects to the same vertex twice. A multi-edge consists of two or more edges connecting the same two vertices.

Representation. The two most popular data structures for graphs are direct representations of adjacency. Let $V = \{0, 1, \ldots, n - 1\}$ be the set of vertices. The adjacency matrix is the n-by-n matrix $A = (a_{ij})$ with

$a_{ij} =$ 1 if $\{i, j\} \in E$, 0 if $\{i, j\} \notin$

For undirected graphs, we have $a_{ij} = a_{ji}$, so A is symmetric. For weighted graphs, we encode more information than just the existence of an edge and define a_{ij} as the weight of the edge connecting i and j. The adjacency matrix of the graph in Figure 50 is

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix},$$

which is symmetric. Irrespective of the number of edges, the adjacency matrix has n^2 elements and thus requires a quadratic amount of space.

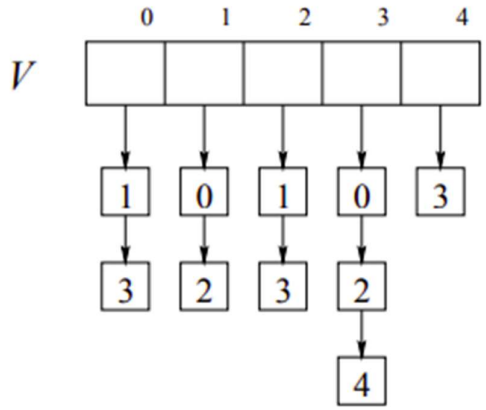


Figure: The adjacency list representation of the graph. Each edge is represented twice, once for each endpoint.

Often, the number of edges is quite small, maybe not much larger than the number of vertices. In these cases, the adjacency matrix wastes memory, and a better choice is a sparse matrix representation referred to as adjacency lists, which is illustrated in Figure 51. It consists of a linear array V for the vertices and a list of neighbors for each vertex. For most algorithms, we assume that vertices and edges are stored in structures containing a small number of fields:

struct Vertex {int d, f, π; Edge ∗adj};

struct Edge {int v; Edge ∗next

The d, f, π fields will be used to store auxiliary information used or created by the algorithms.

Depth-first search. Since graphs are generally not ordered, there are many sequences in which the vertices can be visited. In fact, it is not entirely straightforward to make sure that each vertex is visited once and only once. A useful method is depth-first search. It uses a global variable, time, which is incremented and used to leave time-stamps behind to avoid repeated visits.

```
void VISIT(int i)
1       time++; V [i].d = time;
        forall outgoing edges ij do
 2      if V [j].d = 0 then
3               V [j].π = i; VISIT(j)
        endif
```

endfor;

4 time++; V [i].f = time

The test in line 2 checks whether the neighbor j of i has already been visited. The assignment in line 3 records that the vertex is visited from vertex i. A vertex is first stamped in line 1 with the time at which it is encountered. A vertex is second stamped in line 4 with the time at which its visit has been completed. To prepare the search, we initialize the global time variable to 0, label all vertices as not yet visited, and call VISIT for all yet unvisited vertices.

time = 0;

forall vertices i do V [i].d = 0 endfor;

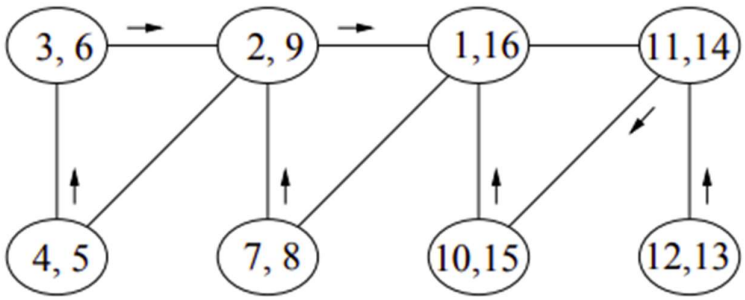
forall vertices i do

if V [i].d = 0 then V [i].π = 0; VISIT(i) endif

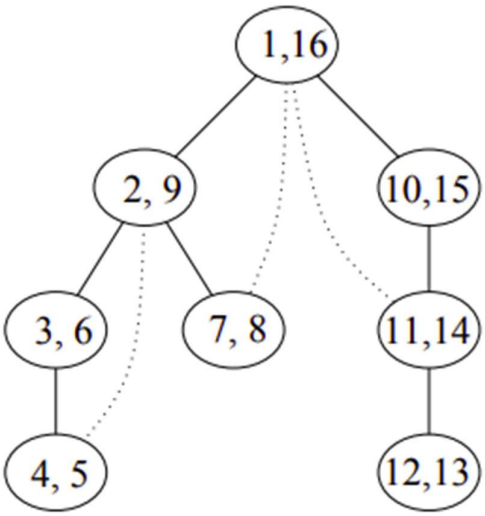
Endfor.

Let n be the number of vertices and m the number of edges in the graph. Depth-first search visits every vertex once and examines every edge twice, once for each endpoint. The running time is therefore O(n + m), which is proportional to the size of the graph and therefore optimal.

DFS forest. Figure illustrates depth-first search by showing the time-stamps d and f and the pointers π indicating the predecessors in the traversal. We call an edge {i, j} ∈ E a tree edge if i = V [j].π or j = V [i].π and a back edge, otherwise. The tree edges form the DFS forest of the graph.



The traversal starts at the vertex with time-stamp 1. Each node is stamped twice, once when it is first encountered and another time when its visit is complete. The forest is a tree if the graph is connected and a collection of two or more trees if it is not connected. Figure shows the DFS forest of the graph in Figure which, in this case, consists of a single tree. The time stamps d are consistent with the preorder traversal of the DFS forest.



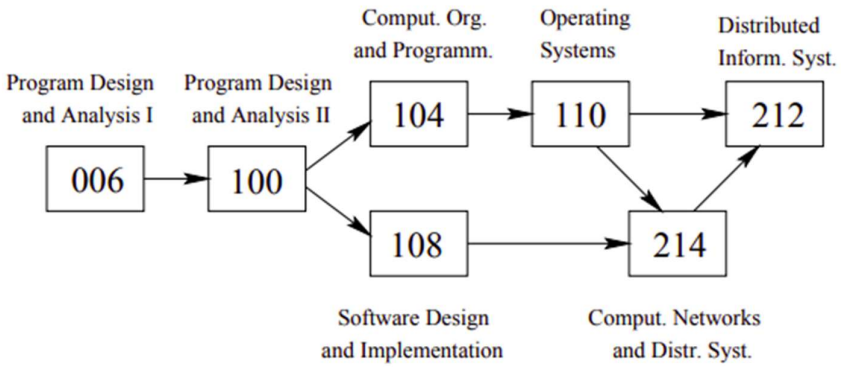
Tree edges are solid and back edges are dotted.

The time-stamps f are consistent with the postorder traversal. The two stamps can be used to decide, in constant time, whether two nodes in the forest live in different subtrees or one is a descendent of the other.

NESTING LEMMA. Vertex j is a proper descendent of vertex i in the DFS forest iff V [i].d < V [j].d as well as V [j].f < V [i].f.

Similarly, if you have a tree and the preorder and postorder numbers of the nodes, you can determine the relation between any two nodes in constant time.

Directed graphs and relations. As mentioned earlier, we have a directed graph if all edges are directed. A directed graph is a way to think and talk about a mathematical relation. A typical problem where relations arise is scheduling. Some tasks are in a definite order while others are unrelated. An example is the scheduling of undergraduate computer science courses, as illustrated in Figure 54. Abstractly, a relation is a pair (V, E), where V = {0, 1, . . ., n − 1} is a finite set of elements and $E \subseteq V \times V$ is a finite set of ordered pairs. Instead of (i, j) $\in$ E we write $i \prec j$ and instead of (V, E) we write (V, $\prec$). If $i \prec j$ then i is a predecessor of j and j is a successor of i. The terms relation, directed graph, digraph, and network are all synonymous.



A subgraph of the CPS course offering. The courses CPS104 and CPS108 are incomparable, CPS104 is a predecessor of CPS110, and so on.

Directed acyclic graphs. A cycle in a relation is a sequence $i0 \prec i1 \prec \ldots \prec ik \prec i0$. Even $i0 \prec i0$ is a cycle. A linear extension of $(V, \prec)$ is an ordering $j0, j1, \ldots, jn-1$ of the elements that is consistent with the relation. Formally this means that $jk \prec j\ell$ implies $k < \ell$. A directed graph without cycle is a directed acyclic graph.

EXTENSION LEMMA. $(V, \prec)$ has a linear extension iff it contains no cycle.

PROOF. "$\Longrightarrow$" is obvious. We prove "$\Longleftarrow$" by induction. A vertex $s \in V$ is called a source if it has no predecessor. Assuming $(V, \prec)$ has no cycle, we can prove that V has a source by following edges against their direction. If we return to a vertex that has already been visited, we have a cycle and thus a contradiction. Otherwise we get stuck at a vertex s, which can only happen because s has no predecessor, which means s is a source.

Let $U = V - \{s\}$ and note that$(U, \prec)$is a relation that is smaller than $(V, \prec)$. Hence $(U, \prec)$ has a linear extension by induction hypothesis. Call this extension X and note that s, X is a linear extension of $(V, \prec)$.

Topological sorting with queue. The problem of constructing a linear extension is called topological sorting. A natural and fast algorithm follows the idea of the proof: find a source s, print s, remove s, and repeat. To expedite the first step of finding a source, each vertex maintains its number of predecessors and a queue stores all sources. First, we initialize this information.

```
forall vertices j do V [j].d = 0 endfor;

forall vertices i do

forall successors j of i do V [j].d++ endfor

 endfor;
```

forall vertices j do

if V [j].d = 0 then ENQUEUE(j) endif

Endfor.

Next, we compute the linear extension by repeated deletion of a source.

while queue is non-empty do

s = DEQUEUE;

forall successors j of s do

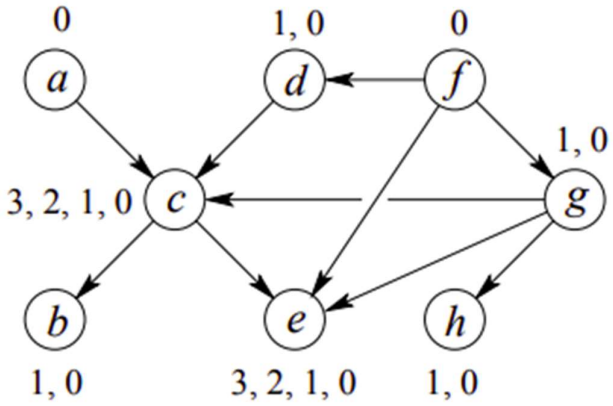
V [j].d--;

if V [j].d = 0 then ENQUEUE(j) endif

Endfor

Endwhile.

The running time is linear in the number of vertices and edges, namely O(n+m). What happens if there is a cycle in the digraph? We illustrate the above algorithm for the directed acyclic graph in Figure.



The numbers next to each vertex count the predecessors, which decreases during the algorithm.

The sequence of vertices added to the queue is also the linear extension computed by the algorithm. If the process starts at vertex a and if the successors of a vertex are ordered by name then we get a, f, d, g, c, h, b, e, which we can check is indeed a linear extension of the relation.

Topological sorting with DFS. Another algorithm that can be used for topological sorting is depth-first search. We output a vertex when its visit has been completed, that is, when all its successors and their successors and so on have already been printed. The linear extension is therefore generated from back to front.

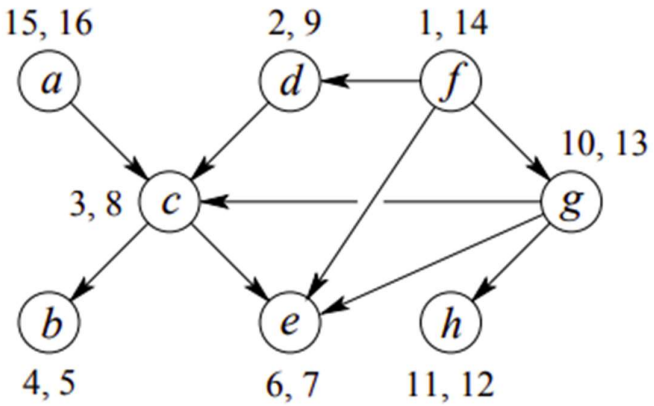


Figure: The numbers next to each vertex are the two time stamps applied by the depth-first search algorithm. The first number gives the time the vertex is encountered, and the second when the visit has been completed.

Figure shows the same digraph as Figure 55 and labels vertices with time stamps. Consider the sequence of vertices in the order of decreasing second time stamp:

a(16), f(14), g(13), h(12), d(9), c(8), e(7), b(5).

Although this sequence is different from the one computed by the earlier algorithm, it is also a linear extension of the relation.

Shortest Paths

One of the most common operations in graphs is finding shortest paths between vertices. This section discusses three algorithms for this problem: breadth-first search for unweighted graphs, Dijkstra's algorithm for weighted graphs, and the Floyd-Warshall algorithm for computing distances between all pairs of vertices.

Breadth-first search. We call a graph connected if there is a path between every pair of vertices. A (connected) component is a maximal connected subgraph. Breadthfirst search, or BFS, is a way to search a graph. It is similar to depth-first search, but while DFS goes as deep as quickly as possible, BFS is more cautious and explores a broad neighborhood before venturing deeper. The starting point is a vertex s. An example is shown in Figure 57.

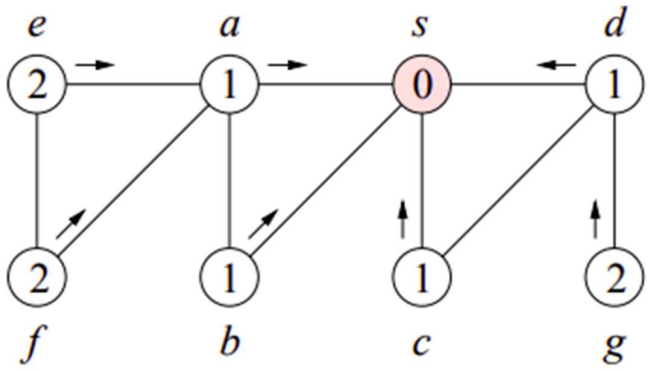


Figure 57: A sample graph with eight vertices and ten edges labeled by breath-first search. The label increases from a vertex to its successors in the search.

As before, we call and edge a tree edge if it is traversed by the algorithm. The tree edges define the BFS tree, which we can use to redraw the graph in a hierarchical manner, as in Figure 58. In the case of an undirected graph, no non-tree edge can connect a vertex to an ancestor in the BFS tree. Why? We use a queue to turn the idea into an algorithm.

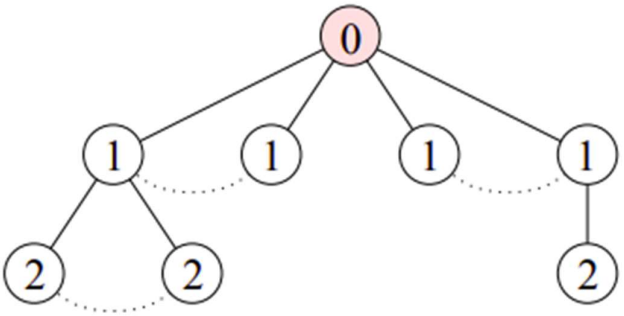


Figure 58: The tree edges in the redrawing of the graph in Figure 57 are solid, and the non-tree edges are dotted.

First, the graph and the queue are initialized.

forall vertices i do V [i].d = −1 endfor;

V [s].d = 0;

MAKEQUEUE; ENQUEUE(s); SEARCH.

A vertex is processed by adding its unvisited neighbors to the queue. They will be processed in turn.

void SEARCH

while queue is non-empty do

i = DEQUEUE;

forall neighbors j of i do

if V [j].d = −1 then

V [j].d = V [i].d + 1; V [j].π = i;

ENQUEUE(j)

endif

endfor

endwhile.

The label V [i].d assigned to vertex i during the traversal is the minimum number of edges of any path from s to i. In other words, V [i].d is the length of the shortest path from s to i. The running time of BFS for a graph with n vertices and m edges is O(n + m).

Single-source shortest path. BFS can be used to find shortest paths in unweighted graphs. We now extend the algorithm to weighted graphs. Assume V and E are the sets of vertices and edges of a simple, undirected graph with a positive weighting function w : E → R+. The length or weight of a path is the sum of the weights of its edges. The distance between two vertices is the length of the shortest path connecting them. For a given source s ∈ V , we study the problem of finding the distances and shortest paths to all other vertices. Figure 59 illustrates the problem by showing the shortest paths to the source s.

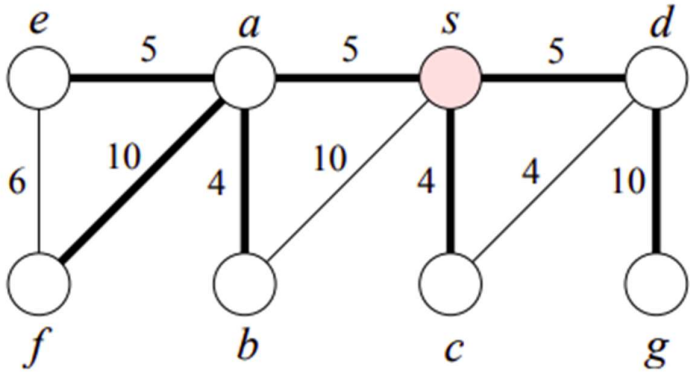


Figure 59: The bold edges form shortest paths and together the shortest path tree with root s. It differs by one edge from the breadth-first tree shown in Figure 57.

In the non-degenerate case, in which no two paths have the same length, the union of all shortest paths to s is a tree, referred to as the shortest path tree. In the degenerate case, we can break ties such that the union of paths is a tree.

As before, we grow a tree starting from s. Instead of a queue, we use a priority queue to determine the next vertex to be added to the tree. It stores all vertices not yet in the tree

and uses V [i].d for the priority of vertex i. First, we initialize the graph and the priority queue.

V [s].d = 0; V [s].π = −1; INSERT(s);

forall vertices i ≠ s do

V [i].d = ∞; INSERT(i)

Endfor.

After initialization the priority queue stores s with priority 0 and all other vertices with priority ∞.

Dijkstra's algorithm. We mark vertices in the tree to distinguish them from vertices that are not yet in the tree. The priority queue stores all unmarked vertices i with priority equal to the length of the shortest path that goes from i in one edge to a marked vertex and then to s using only marked vertices.

while priority queue is non-empty do

i = EXTRACTMIN; mark i;

forall neighbors j of i do

if j is unmarked then

V [j].d = min{w(ij) + V [i].d, V [j].d}

endif

endfor

Endwhile.

Table 3 illustrates the algorithm by showing the information in the priority queue after each iteration of the whileloop operating on the graph in Figure.

s	**0**							
a	∞	5	**5**					
b	∞	10	10	9	**9**			
c	∞	**4**						
d	∞	5	5	**5**				
e	∞	∞	∞	10	10	**10**		
f	∞	∞	∞	15	15	15	**15**	
g	∞	∞	∞	∞	15	15	15	**15**

Table 3: Each column shows the contents of the priority queue. Time progresses from left to right.

The marking mechanism is not necessary but clarifies the process. The algorithm performs n EXTRACTMIN operations and at most m DECREASEKEY operations. We compare the running time under three different data structures used to represent the priority queue. The first is a linear array, as originally proposed by Dijkstra, the second is a heap, and the third is a Fibonacci heap. The results are shown in Table 4. We get the best result with Fibonacci heaps for which the total running time is O(n log n + m).

	array	heap	F-heap
EXTRACTMINS	n^2	$n \log n$	$n \log n$
DECREASEKEYS	m	$m \log m$	m

Table 4: Running time of Dijkstra's algorithm for three different implementations of the priority queue holding the yet unmarked vertices.

Correctness. It is not entirely obvious that Dijkstra's algorithm indeed finds the shortest paths to s. To show that it does, we inductively prove that it maintains the following two invariants.

(A) For every unmarked vertex j, V [j].d is the length of the shortest path from j to s that uses only marked vertices other than j.

(B) For every marked vertex i, V [i].d is the length of the shortest path from i to s.

PROOF. Invariant (A) is true at the beginning of Dijkstra's algorithm. To show that it is maintained throughout the process, we need to make sure that shortest paths are computed correctly. Specifically, if we assume Invariant (B) for vertex i then the algorithm correctly updates the priorities V [j].d of all neighbors j of i, and no other priorities change.

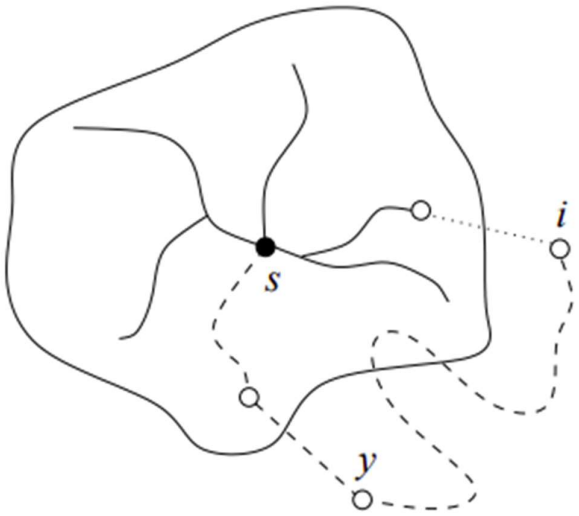


Figure 60: The vertex y is the last unmarked vertex on the hypothetically shortest, dashed path that connects i to s.

At the moment vertex i is marked, it minimizes V [j].d over all unmarked vertices j. Suppose that, at this moment, V [i].d is not the length of the shortest path from i to s. Because of Invariant (A), there is at least one other unmarked vertex on the shortest path. Let the last such vertex be y, as shown in Figure 60. But then V [y].d < V [i].d, which is a contradiction to the choice of i.

We used (B) to prove (A) and (A) to prove (B). To make sure we did not create a circular argument, we parametrize the two invariants with the number k of vertices that are marked and thus belong to the currently constructed portion of the shortest path tree. To prove (Ak) we need (Bk) and to prove (Bk) we need (Ak−1). Think of the two invariants as two recursive functions, and for each pair of calls, the parameter decreases by one and thus eventually becomes zero, which is when the argument arrives at the base case.

All-pairs shortest paths. We can run Dijkstra's algorithm n times, once for each vertex as the source, and thus get the distance between every pair of vertices. The running time is $O(n^2 \log n + nm)$ which, for dense graphs, is the same as $O(n^3)$. Cubic running time can be achieved with a much simpler algorithm using the adjacency matrix to store distances. The idea is to iterate n times, and after the k-th iteration, the computed distance between vertices i and j is the length of the shortest path from i to j that, other than i and j, contains only vertices of index k or less.

for k = 1 to n do

for i = 1 to n do

```
for j = 1 to n do
A[i, j] = min{A[i, j], A[i, k] + A[k, j]}
Endfor
Endfor
Endfor.
```

The only information needed to update A[i, j] during the k-th iteration of the outer for-loop are its old value and values in the k-th row and the k-th column of the prior adjacency matrix. This row remains unchanged in this iteration and so does this column. We therefore do not have to use two arrays, writing the new values right into the old matrix. We illustrate the algorithm by showing the adjacency, or distance matrix before the algorithm in Figure 61 and after one iteration in Figure 62.

	s	*a*	*b*	*c*	*d*	*e*	*f*	*g*
s	0	5	10	4	5			
a	5	0	4			5	10	
b	10	4	0					
c	4			0	4			
d	5			4	0			10
e		5				0	6	
f		10				6	0	
g					10			0

Figure 61: Adjacency, or distance matrix of the graph in Figure. All blank entries store ∞.

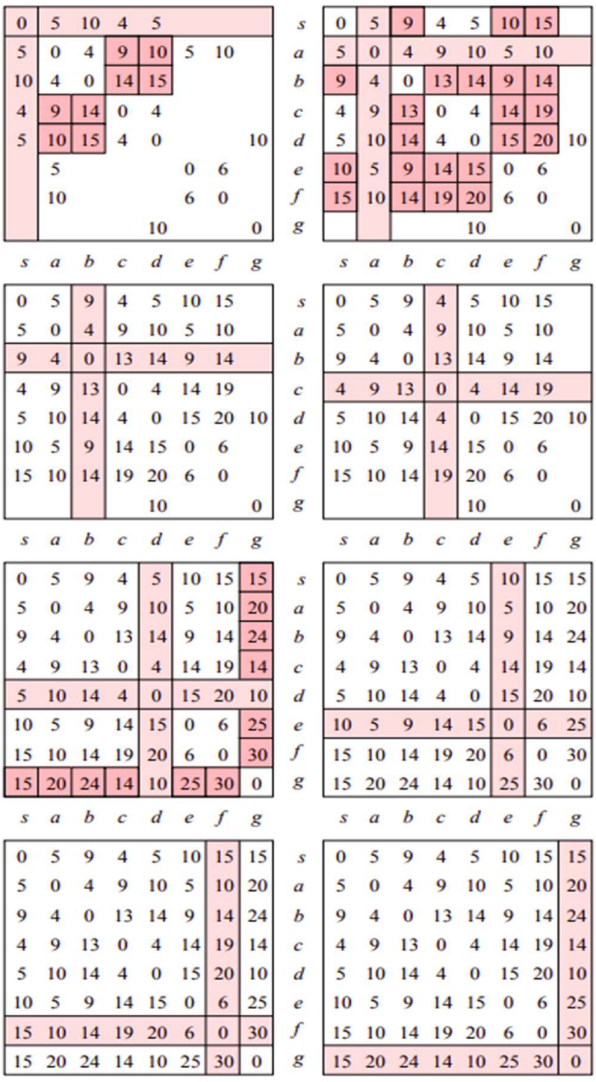


Figure 62: Matrix after each iteration. The k-th row and column are shaded and the new, improved distances are high-lighted.

The algorithm works for weighted undirected as well as for weighted directed graphs. Its correctness is easily verified inductively. The running time is $O(n^3)$.

Minimum Spanning Trees

When a graph is connected, we may ask how many edges we can delete before it stops being connected. Depending on the edges we remove, this may happen sooner or later. The slowest strategy is to remove edges until the graph becomes a tree. Here we study the somewhat more difficult problem of removing edges with a maximum total weight. The remaining graph is then a tree with minimum total weight. Applications that motivate this question can be found in life support systems modeled as graphs or networks, such as telephone, power supply, and sewer systems.

Free trees. An undirected graph (U, T) is a free tree if it is connected and contains no cycle. We could impose a hierarchy by declaring any one vertex as the root and thus obtain a rooted tree. Here, we have no use for a hierarchical organization and exclusively deal with free trees.

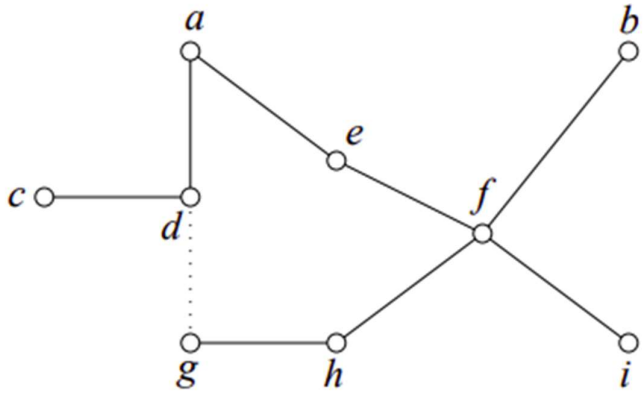


Figure 63: Adding the edge dg to the tree creates a single cycle with vertices d, g, h, f, e, a.

The number of edges of a free tree is always one less than the number of vertices. Whenever we add a new edge (connecting two old vertices) we create exactly one cycle. This cycle can be destroyed by deleting any one of its edges, and we get a new free tree, as in Figure 63. Let (V, E) be a connected and undirected graph. A subgraph is another graph (U, T) with $U \subseteq V$ and $T \subseteq E$. It is a spanning tree if it is a free tree with $U = V$.

Minimum spanning trees. For the remainder of this section, we assume that we also have a weighting function, $w : E \to R$. The weight of subgraph is then the total weight of its edges, $w(T) = P\ e \in T\ w(e)$. A minimum spanning tree, or MST of G is a spanning tree that minimizes the weight. The definitions are illustrated in Figure 64 which shows a graph of solid edges with a minimum spanning tree of bold edges. A generic algorithm for constructing an MST grows a tree by adding more and more edges. Let $A \subseteq E$ be a subset of some MST of a connected graph (V, E). An edge $uv \in E - A$ is safe for A if $A \cup \{uv\}$ is also subset of some MST. The generic algorithm adds safe edges until it arrives at an MST.

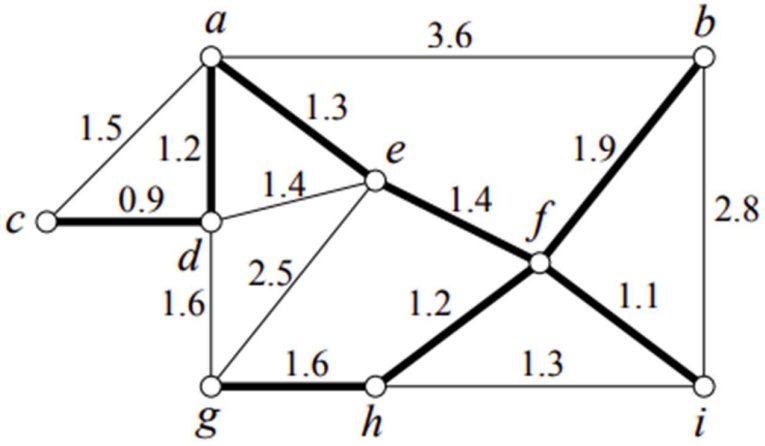


Figure 64: The bold edges form a spanning tree of weight 0.9 + 1.2 + 1.3 + 1.4 + 1.1 + 1.2 + 1.6 + 1.9 = 10.6.

A = ∅;

while (V, A) is not a spanning tree do

find a safe edge uv; A = A ∪ {uv}

Endwhile.

As long as A is a proper subset of an MST there are safe edges. Specifically, if (V, T) is an MST and A ⊆ T then all edges in T − A are safe for A. The algorithm will therefore succeed in constructing an MST. The only thing that is not yet clear is how to find safe edges quickly.

Cuts. To develop a mechanism for identifying safe edges, we define a cut, which is a partition of the vertex set into two complementary sets, V = W ∪˙ (V −W). It is crossed by an edge uv ∈ E if u ∈ W and v ∈ V −W, and it respects an edge set A if A contains no crossing edge. The definitions are illustrated in Figure.

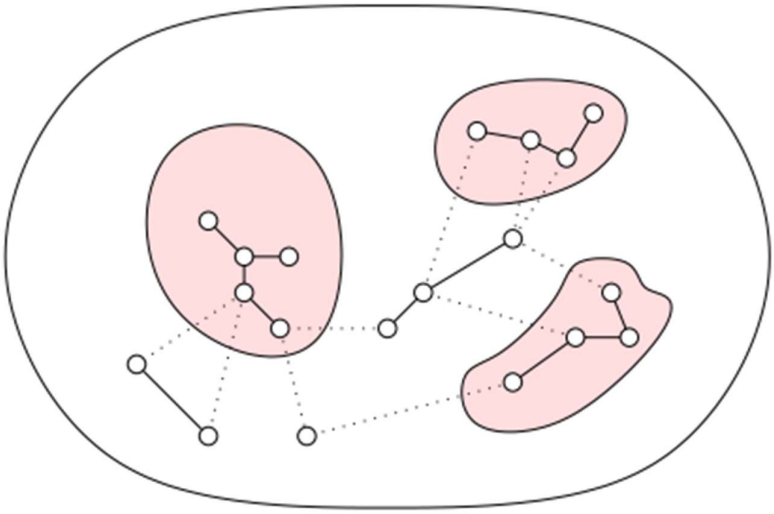


Figure: The vertices inside and outside the shaded regions form a cut that respects the collection of solid edges. The dotted edges cross the cut.

CUT LEMMA. Let A be subset of an MST and consider a cut $W \dot{\cup} (V - W)$ that respects A. If uv is a crossing edge with minimum weight then uv is safe for A.

PROOF. Consider a minimum spanning tree (V, T) with $A \subseteq T$. If $uv \in T$ then we are done. Otherwise, let $T' = T \cup \{uv\}$. Because T is a tree, there is a unique path from u to v in T . We have $u \in W$ and $v \in V - W$, so the path switches at least once between the two sets. Suppose it switches along xy, as in Figure 66. Edge xy crosses the cut, and since A contains no crossing edges we have $xy \notin A$. Because uv has minimum weight among crossing edges we have $w(uv) \leq w(xy)$. Define $T'' = T' - \{xy\}$. Then (V, T ″) is a spanning tree and because $w(T'') = w(T) - w(xy) + w(uv) \leq w(T)$ it is a minimum spanning tree. The claim follows because $A \cup \{uv\} \subseteq T''$.

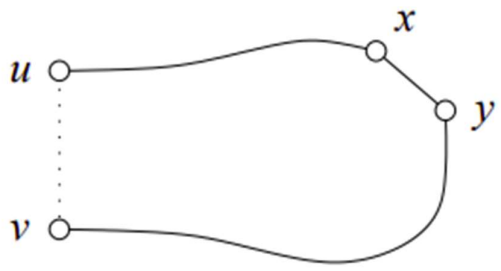


Figure: Adding uv creates a cycle and deleting xy destroys the cycle.

A typical application of the Cut Lemma takes a component of (V, A) and defines W as the set of vertices of that component. The complementary set $V - W$ contains all other vertices, and crossing edges connect the component with its complement.

Prim's algorithm. Prim's algorithm chooses safe edges to grow the tree as a single component from an arbitrary first vertex s. Similar to Dijkstra's algorithm, the vertices that do not yet belong to the tree are stored in a priority queue. For each vertex i outside the tree, we define its priority V [i].d equal to the minimum weight of any edge that connects i to a vertex in the tree. If there is no such edge then V [i].d = ∞. In addition to the priority, we store the index of the other endpoint of the minimum weight edge. We first initialize this information.

V [s].d = 0; V [s].π = −1; INSERT(s);

forall vertices i ≠ s do

V [i].d = ∞; INSERT(i)

Endfor.

The main algorithm expands the tree by one edge at a time. It uses marks to distinguish vertices in the tree from vertices outside the tree.

while priority queue is non-empty do

i = EXTRACTMIN; mark i;

forall neighbors j of i do

if j is unmarked and w(ij) < V [j].d then

V [j].d = w(ij); V [j].π = i

endif

endfor

Endwhile.

After running the algorithm, the MST can be recovered from the π-fields of the vertices. The algorithm together with its initialization phase performs n = |V | insertions into the priority queue, n extractmin operations, and at most m = |E| decrease key operations. Using the Fibonacci heap implementation, we get a running time of O(n log n + m), which is the same as for constructing the shortest-path tree with Dijkstra's algorithm.

Kruskal's algorithm. Kruskal's algorithm is another implementation of the generic algorithm. It adds edges in a sequence of non-decreasing weight. At any moment, the chosen edges form a collection of trees. These trees merge to form larger and fewer trees, until they eventually combine into a single tree. The algorithm uses a priority queue for the edges and a set system for the vertices. In this context, the term 'system' is just another word for 'set',

but we will use it exclusively for sets whose elements are themselves sets. Implementations of the set system will be discussed in the next lecture. Initially, $A = \emptyset$, the priority queue contains all edges, and the system contains a singleton set for each vertex, $C = \{\{u\} \mid u \in V\}$. The algorithm finds an edge with minimum weight that connects two components defined by A. We set W equal to the vertex set of one component and use the Cut Lemma to show that this edge is safe for A. The edge is added to A and the process is repeated. The algorithm halts when only one tree is left, which is the case when A contains $n - 1 = |V| - 1$ edges.

$A = \emptyset$;

while $|A| < n - 1$ do

uv = EXTRACTMIN;

find $P, Q \in C$ with $u \in P$ and $v \in Q$;

if $P \neq Q$ then

$A = A \cup \{uv\}$; merge P and Q

endif

Endwhile.

The running time is O(m log m) for the priority queue operations plus some time for maintaining C. There are two operations for the set system, namely finding the set that contains a given element, and merging two sets into one.

An example. We illustrate Kruskal's algorithm by applying it to the weighted graph in Figure 64. The sequence of edges sorted by weight is cd, fi, fh, ad, ae, hi, de, ef, ac, gh, dg, bf, eg, bi, ab. The evolution of the set system is illustrated in Figure 67, and the MST computed with Kruskal's algorithm and indicated with dotted edges is the same as in Figure 64.

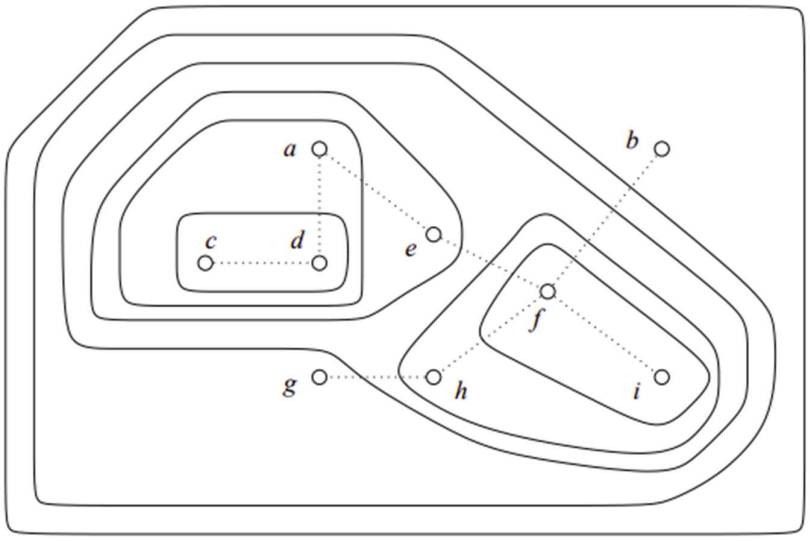


Figure: Eight union operations merge the nine singleton sets into one set.

The edges cd, fi, fh, ad, ae are all added to the tree. The next two edge, hi and de, are not added because they each have both endpoints in the same component, and adding either edge would create a cycle. Edge ef is added to the tree giving rise to a set in the system that contains all vertices other than g and b. Edge ac is not added, gh is added, dg is not, and finally bf is added to the tree. At this moment the system consists of a single set that contains all vertices of the graph.

As suggested by Figure, the evolution of the construction can be interpreted as a hierarchical clustering of the vertices. The specific method that corresponds to the evolution created by Kruskal's algorithm is referred to as single-linkage clustering.

UNION-FIND

In this lecture, we present two data structures for the disjoint set system problem we encountered in the implementation of Kruskal's algorithm for minimum spanning trees. An interesting feature of the problem is that m operations can be executed in a time that is only ever so slightly more than linear in m.

Abstract data type. A disjoint set system is an abstract data type that represents a partition C of a set [n] = {1, 2, . . ., n}. In other words, C is a set of pairwise disjoint subsets of [n] such that the union of all sets in C is [n]. The data type supports

set FIND(i): return P ∈ C with i ∈ P;

void UNION(P, Q) : C = C − {P, Q} ∪ {P ∪ Q}.

In most applications, the sets themselves are irrelevant, and it is only important to know when two elements belong to the same set and when they belong to different sets in the

system. For example, Kruskal's algorithm executes the operations only in the following sequence:

P = FIND(i); Q = FIND(j);

if P ≠ Q then UNION(P, Q) endif.

This is similar to many everyday situations where it is usually not important to know what it is as long as we recognize when two are the same and when they are different.

Linked lists. We construct a fairly simple and reasonably efficient first solution using linked lists for the sets. We use a table of length n, and for each i ∈ [n], we store the name of the set that contains i. Furthermore, we link the elements of the same set and use the name of the first element as the name of the set. Figure 68 shows a sample set system and its representation. It is convenient to also store the size of the set with the first element.

To perform a UNION operation, we need to change the name for all elements in one of the two sets. To save time, we do this only for the smaller set. To merge the two lists without traversing the longer one, we insert the shorter list between the first two elements of the longer list.

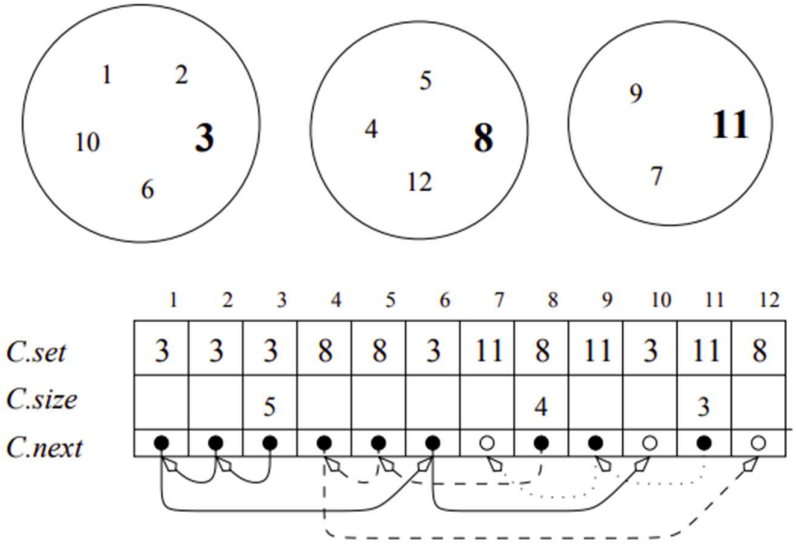


Figure: The system consists of three sets, each named by the bold element. Each element stores the name of its set, possibly the size of its set, and possibly a pointer to the next element in the same set.

void UNION(int P, Q)

if C[P].size < C[Q].size then P ↔ Q endif;

C[P].size = C[P].size + C[Q].size;

second = C[P].next; C[P].next = Q; t = Q;

while t ≠ 0 do

C[t].set = P; u = t; t = C[t].next

endwhile; C[u].next = second.

In the worst case, a single UNION operation takes time *Θ(n)*. The amortized performance is much better because we spend time only on the elements of the smaller set.

WEIGHTED UNION LEMMA. n − 1 UNION operations applied to a system of n singleton sets take time *O(n log n)*.

PROOF. For an element, i, we consider the cardinality of the set that contains it, σ(i) = C[FIND(i)].size. Each time the name of the set that contains i changes, σ(i) at least doubles. After changing the name k times, we have σ(i) ≥ 2 k and therefore k ≤ log2 n. In other words, i can be in the smaller set of a UNION operation at most log2 n times. The claim follows because a UNION operation takes time proportional to the cardinality of the smaller set.

Up-trees. Thinking of names as pointers, the above data structure stores each set in a tree of height one. We can use more general trees and get more efficient UNION operations at the expense of slower FIND operations. We consider a class of algorithms with the following commonalities:

• each set is a tree and the name of the set is the index of the root;

• FIND traverses a path from a node to the root;

• UNION links two trees.

It suffices to store only one pointer per node, namely the pointer to the parent. This is why these trees are called up-trees. It is convenient to let the root point to itself.

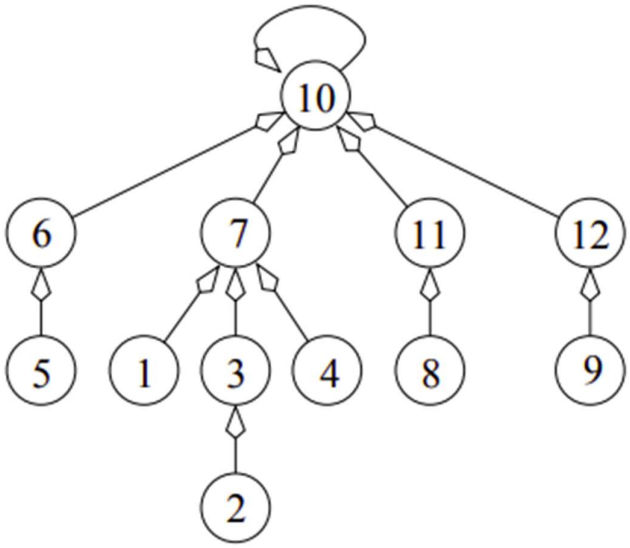


Figure a: The UNION operations create a tree by linking the root of the first set to the root of the second set.

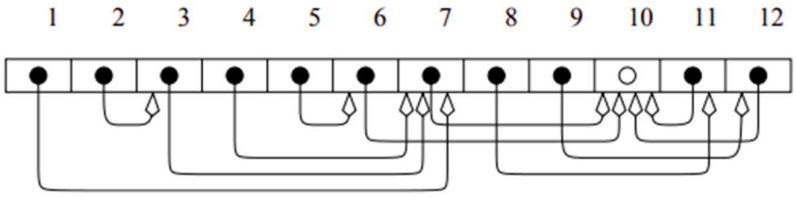


Figure b: The table stores indices which function as pointers as well as names of elements and of sets. The white dot represents a pointer to itself.

Figure a shows the up-tree generated by executing the following eleven UNION operations on a system of twelve singleton sets: 2 ∪ 3, 4 ∪ 7, 2 ∪ 4, 1 ∪ 2, 4 ∪ 10, 9 ∪ 12, 12 ∪ 2, 8 ∪ 11, 8 ∪ 2, 5 ∪ 6, 6 ∪ 1. Figure b shows the embedding of the tree in a table. UNION takes constant time and FIND takes time proportional to the length of the path, which can be as large as n − 1.

Weighted union. The running time of FIND can be improved by linking smaller to larger trees. This is the ide of weighted union again. Assume a field C[i].p for the index of the parent (C[i].p = i if i is a root), and a field C[i].size for the number of elements in the tree rooted at i. We need the size field only for the roots and we need the index to the parent field everywhere except for the roots. The FIND and UNION operations can now be implemented as follows:

```
int FIND(int i)

 if C[i].p ≠ i then return FIND(C[i].p) endif;

 return i.
```

void UNION(int i, j)

if C[i].size < C[j].size then i ↔ j endif;

C[i].size = C[i].size + C[j].size; C[j].p = i.

The size of a subtree increases by at least a factor of 2 from a node to its parent. The depth of a node can therefore not exceed $\log_2 n$. It follows that FIND takes at most time O(log n). We formulate the result on the height for later reference.

HEIGHT LEMMA. An up-tree created from n singleton nodes by n − 1 weighted union operations has height at most log2 n.

Path compression. We can further improve the time for FIND operations by linking traversed nodes directly to the root. This is the idea of path compression. The UNION operation is implemented as before and there is only one modification in the implementation of the FIND operation:

int FIND(int i)

if C[i].p ≠ i then C[i].p = FIND(C[i].p) endif;

return C[i].p.

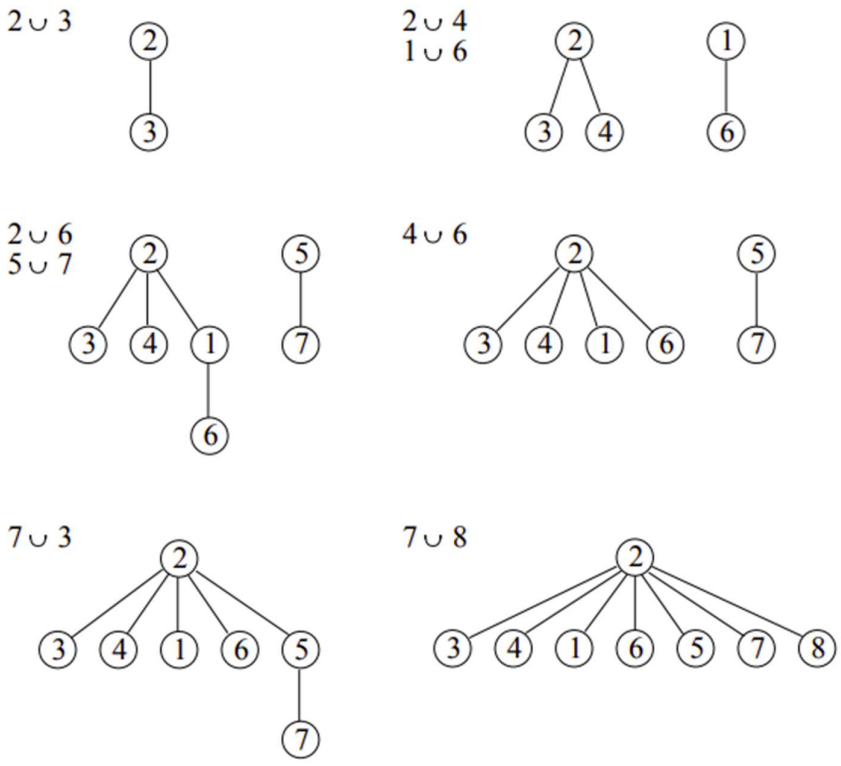


Figure: The operations and up-trees develop from top to bottom and within each row from left to right.

If i is not root then the recursion makes it the child of a root, which is then returned. If i is a root, it returns itself because in this case C[i].p = i, by convention. Figure illustrates the algorithm by executing a sequence of eight operations i U j, which is short for finding the sets that contain i and j, and performing a UNION operation if the sets are different. At the beginning, every element forms its own one-node tree. With path compression, it is difficult to imagine that long paths can develop at all.

Iterated logarithm. We will prove shortly that the iterated logarithm is an upper bound on the amortized time for a FIND operation. We begin by defining the function from its inverse. Let F(0) = 1 and F(i + 1) = 2F (i) . We have F(1) = 2, F(2) = 22 , and F(3) = 22 2 . In general, F(i) is the tower of i 2s. Table 5 shows the values of F for the first six arguments. For i ≤ 3, F is very small, but

i	0	1	2	3	4	5
F	1	2	4	16	$65,536$	$2^{65,536}$

Table 5: Values of F.

for i = 5 it already exceeds the number of atoms in our universe. Note that the binary logarithm of a tower of i 2s is a tower of i−1 2s. The iterated logarithm is the number of times we can take the binary logarithm before we drop down to one or less. In other words, the iterated logarithm is the inverse of F,

$\log^* n = \min\{i \mid F(i) \geq n\} = \min\{i \mid \log_2 \log_2 \ldots \log_2 n \leq 1\}$

where the binary logarithm is taken i times. As n goes to infinity, $\log^* n$ goes to infinity, but very slowly.

Levels and groups. The analysis of the path compression algorithm uses two Census Lemmas discussed shortly. Let A1, A2, . . . , Am be a sequence of UNION and FIND operations, and let T be the collection of up-trees we get by executing the sequence, but without path compression. In other words, the FIND operations have no influence on the trees. The level $\lambda(\mu)$ of a node μ is its height of its subtree in T plus one.

LEVEL CENSUS LEMMA. There are at most $n/2^{\ell-1}$ nodes at level ℓ.

PROOF. We use induction to show that a node at level ℓ has a subtree of at least $2^{\ell-1}$ nodes. The claim follows because subtrees of nodes on the same level are disjoint.

CHAPTER 8

GEOMETRIC ALGORITHMS

Plane-sweep

Plane-sweep is an algorithmic paradigm that emerges in the study of two-dimensional geometric problems. The idea is to sweep the plane with a line and perform the computations in the sequence the data is encountered. In this section, we solve three problems with this paradigm: we construct the convex hull of a set of points, we triangulate the convex hull using the points as vertices, and we test a set of line segments for crossings.

Convex hull. Let S be a finite set of points in the plane, each given by its two coordinates. The convex hull of S, denoted by conv S, is the smallest convex set that contains S. Figure 91 illustrates the definition for a set of nine points. Imagine the points as solid nails in a planar board. An intuitive construction stretches a rubber band around the nails. After letting go, the nails prevent the complete relaxation of the rubber band which will then trace the boundary of the convex hull.

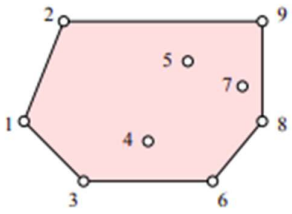


Figure : The convex hull of nine points, which we represent by the counterclockwise sequence of boundary vertices: 1, 3, 6, 8, 9, 2.

To construct the counterclockwise cyclic sequence of boundary vertices representing the convex hull, we sweep a vertical line from left to right over the data. At any moment in time, the points in front (to the right) of the line are untouched and the points behind (to the left) of the line have already been processed.

Step 1. Sort the points from left to right and relabel them in this sequence as x1, x2, . . . , xn.

Step 2. Construct a counterclockwise triangle from the first three points: x1x2x3 or x1x3x2.

Step 3. For i from 4 to n, add the next point xi to the convex hull of the preceding points by finding the two lines that pass through xi and support the convex hull.

The algorithm is illustrated in Figure 92, which shows the addition of the sixth point in the data set.

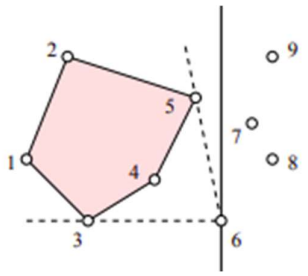


Figure: The vertical sweep-line passes through point 6. To add 6, we substitute 6 for the sequence of vertices on the boundary between 3 and 5.

Orientation test. A critical test needed to construct the convex hull is to determine the orientation of a sequence of three points. In other words, we need to be able to distinguish whether we make a left-turn or a right-turn as we go from the first to the middle and then the last point in the sequence. A convenient way to determine the orientation evaluates the determinant of a three-by-three matrix. More precisely, the points a = (a1, a2), b = (b1, b2), c = (c1, c2) form a left-turn iff

$$\det \begin{bmatrix} 1 & a_1 & a_2 \\ 1 & b_1 & b_2 \\ 1 & c_1 & c_2 \end{bmatrix} > 0.$$

The three points form a right-turn iff the determinant is negative and they lie on a common line iff the determinant is zero.

boolean LEFT(Points a, b, c)

return [a1(b2 − c2) + b1(c2 − a2) + c1(a2 − b2) > 0].

To see that this formula is correct, we may convince ourselves that it is correct for three non-collinear points, e.g. a = (0, 0), b = (1, 0), and c = (0, 1). Remember also that the determinant measures the area of the triangle and is therefore a continuous function that passes through zero only when the three points are collinear. Since we can continuously move every left-turn to every other left-turn without leaving the class of left-turns, it follows that the sign of the determinant is the same for all of them.

Finding support lines. We use a doubly-linked cyclic list of vertices to represent the convex hull boundary. Each node in the list contains pointers to the next and the previous nodes. In addition, we have a pointer last to the last vertex added to the list. This vertex is also the rightmost in the list. We add the i-th point by connecting it to the vertices μ → pt and λ → pt identified in a counterclockwise and a clockwise traversal of the cycle starting at last, as

illustrated in Figure. We simplify notation by using nodes in the parameter list of the orientation test instead of the points they store.

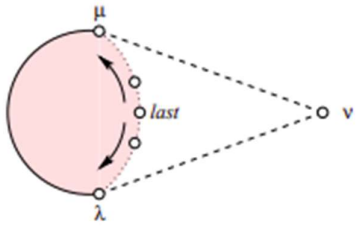


Figure 93: The upper support line passes through the first point μ → pt that forms a left-turn from v → pt to μ → next → pt.

μ = λ = last; create new node with v → pt = i;

while RIGHT(v, μ, μ → next) do

μ = μ → next

endwhile; while LEFT(v, λ, λ → prev) do

λ = λ → prev

endwhile;

v → next = μ; v → prev = λ;

μ → prev = λ → next = v; last = v.

The effort to add the i-th point can be large, but if it is then we remove many previously added vertices from the list. Indeed, each iteration of the for-loop adds only one vertex to the cyclic list. We charge $2 for the addition, one dollar for the cost of adding and the other to pay for the future deletion, if any. The extra dollars pay for all iterations of the while-loops, except for the first and the last. This implies that we spend only constant amortized time per point. After sorting the points from left to right, we can therefore construct the convex hull of n points in time *O(n).*

Triangulation. The same plane-sweep algorithm can be used to decompose the convex hull into triangles. All we need to change is that points and edges are never removed and a new point is connected to every point examined during the two while-loops. We define a (geometric) triangulation of a finite set of points S in the plane as a maximally connected straight-line embedding of a planar graph whose vertices are mapped to points in S. Figure 94 shows the triangulation of the nine points in Figure 91 constructed by the plane-sweep algorithm. A triangulation is not necessarily a maximally connected planar graph since the prescribed placement of the points fixes the boundary of the outer face to be the boundary of the convex hull. Letting k be the number of edges of that boundary, we would have to

add k − 3 more edges to get a maximally connected planar graph. It follows that the triangulation has m = 3n − (k + 3) edges and ℓ = 2n − (k + 2) triangles.

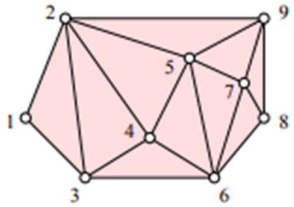


Figure: Triangulation constructed with the plane-sweep algorithm.

Line segment intersection. As a third application of the plane-sweep paradigm, we consider the problem of deciding whether or not n given line segments have pairwise disjoint interiors. We allow line segments to share endpoints but we do not allow them to cross or to overlap. We may interpret this problem as deciding whether or not a straight-line drawing of a graph is an embedding. To simplify the description of the algorithm, we assume no three endpoints are collinear, so we only have to worry about crossings and not about other overlaps.

How can we decide whether or not a line segment with endpoint u = (u1, u2) and v = (v1, v2) crosses another line segment with endpoints p = (p1, p2) and q = (q1, q2)? Figure 95 illustrates the question by showing the four different cases of how two line segments and the lines they span can intersect. The line segments cross iff uv intersects the line of pq and pq intersects the line of uv. This condition can be checked using the orientation test.

boolean CROSS(Points u, v, p, q)

return [(LEFT(u, v, p) xor LEFT(u, v, q)) and (LEFT(p, q, u) xor LEFT(p, q, v))].

We can use the above function to test all n^2 pairs of line segments, which takes time $O(n^2)$.

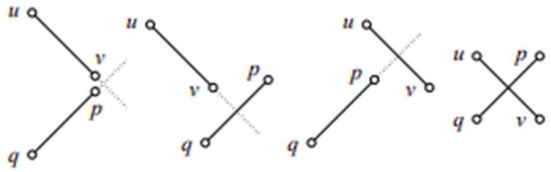


Figure: Three pairs of non-crossing and one pair of crossing line segments.

Plane-sweep algorithm. We obtain a faster algorithm by sweeping the plane with a vertical line from left to right, as before. To avoid special cases, we assume that no two endpoints are the same or lie on a common vertical line. During the sweep, we maintain the subset of

line segments that intersect the sweep-line in the order they meet the line, as shown in Figure. We store this subset in a dictionary, which is updated at every endpoint.

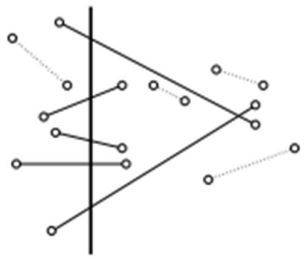


Figure: Five of the line segments intersect the sweep-line at its current position and two of them cross.

Only line segments that are adjacent in the ordering along the sweep-line are tested for crossings. Indeed, two line segments that cross are adjacent right before the sweep-line passes through the crossing, if not earlier.

Step 1. Sort the 2n endpoints from left to right and relabel them in this sequence as x1, x2, . . . , x2n. Each point still remembers the index of the other endpoint of its line segment.

Step 2. For i from 1 to 2n, process the i-th endpoint as follows:

Case 2.1 xi is left endpoint of the line segment xixj . Therefore, i < j. Insert xixj into the dictionary and let uv and pq be its predecessor and successor. If CROSS(u, v, xi , xj) or CROSS(p, q, xi , xj) then report the crossing and stop.

Case 2.2 xi is right endpoint of the line segment xixj . Therefore, i > j. Let uv and pq be the predecessor and the successor of xixj . If CROSS(u, v, p, q) then report the crossing and stop. Delete xixj from the dictionary.

We do an insertion into the dictionary for each left endpoint and a deletion from the dictionary for each right endpoint, both in time *O(log n)*. In addition, we do at most two crossing tests per endpoint, which takes constant time. In total, the algorithm takes time *O(n log n)* to test whether a set of n line segments contains two that cross.

Delaunay Triangulations:

The triangulations constructing by plane-sweep are typically of inferior quality, that is, there are many long and skinny triangles and therefore many small and many large angles. We study Delaunay triangulations which distinguish themselves from all other triangulations by a number of nice properties, including they have fast algorithms and they avoid small angles to the extent possible.

Plane-sweep versus Delaunay triangulation. Figures a and b show two triangulations of the same set of points, one constructed by plane-sweep and the other the Delaunay triangulation. The angles in the Delaunay triangulation seem consistently larger than those in the planesweep triangulation.

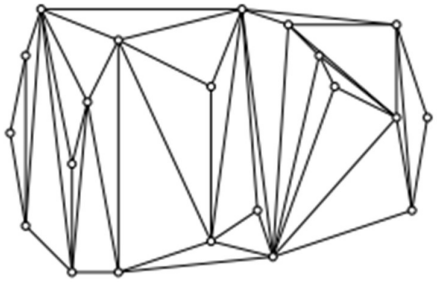


Figure a: Triangulation constructed by plane-sweep. Points on the same vertical line are processed from bottom to top.

This is not a coincidence and it can be proved that the Delaunay triangulation maximizes the minimum angle for every input set. Both triangulations contain the edges that bound the convex hull of the input set.

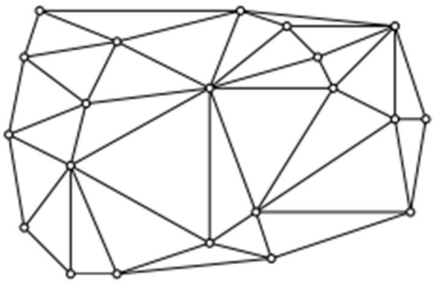


Figure b: Delaunay triangulation of the same twenty-one points triangulated in Figure a.

Voronoi diagram. We introduce the Delaunay triangulation indirectly, by first defining a particular decomposition of the plane into regions, one per point in the finite data set S. The

region of the point u in S contains all points x in the plane that are at least as close to u as to any other point in S, that is,

Vu = {x ∈ R 2 | kx − uk ≤ kx − vk, v ∈ S},

where kx − uk = [(x1 − u1) 2 + (x2 − u2) 2] 1/2 is the Euclidean distance between the points x and u. We refer to Vu as the Voronoi region of u. It is closed and its boundary consists of Voronoi edges which Vu shares with neighboring Voronoi regions. A Voronoi edge ends in Voronoi vertices which it shares with other Voronoi edges. The Voronoi diagram of S is the collection of Voronoi regions, edges and vertices. Figure c illustrates the definitions. Let n be the number of points in S. We list some of the properties that will be important later.

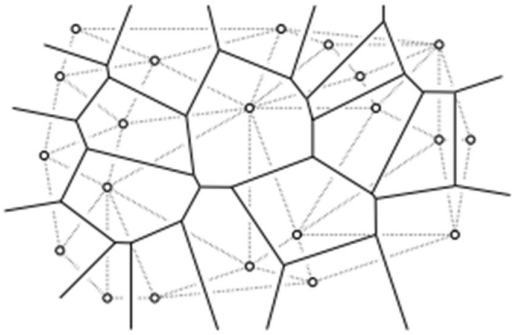


Figure c: The (solid) Voronoi diagram drawn above the (dotted) Delaunay triangulation of the same twenty-one points triangulated in Figures a and c. Some of the Voronoi edges are too far out to fit into the picture.

• Each Voronoi region is a convex polygon constructed as the intersection of n − 1 closed half-planes. • The Voronoi region Vu is bounded (finite) iff u lies in the interior of the convex hull of S.

• The Voronoi regions have pairwise disjoint interiors and together cover the entire plane.

Delaunay triangulation. We define the Delaunay triangulation as the straight-line dual of the Voronoi diagram. Specifically, for every pair of Voronoi regions Vu and Vv that share an edge, we draw the line segment from u to v. By construction, every Voronoi vertex, x, has j ≥ 3 closest input points. Usually there are exactly three closest points, u, v, w, in which case the triangle they span belongs to the Delaunay triangulation. Note that x is equally far from u, v, and w and further from all other points in S. This implies the empty circle property of Delaunay triangles: all points of S − {u, v, w} lie outside the circumscribed circle of uvw. Similarly, for each Delaunay edge uv, there is a circle that passes through u and v such that all points of S − {u, v} lie outside the circle. For example, the circle centered at the midpoint of the Voronoi edge shared by Vu and Vv is empty in this sense. This property can be used to

prove that the edge skeleton of the Delaunay triangulation is a straight-line embedding of a planar graph.

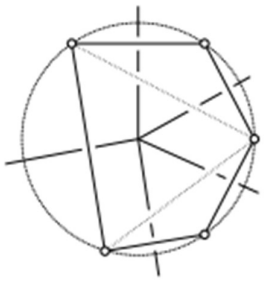


Figure d: A Voronoi vertex of degree 5 and the corresponding pentagon in the Delaunay triangulation. The dotted edges complete the triangulation by decomposing the pentagon into three triangles.

Now suppose there is a vertex with degree j > 3. It corresponds to a polygon with j > 3 edges in the Delaunay triangulation, as illustrated in Figure d. Strictly speaking, the Delaunay triangulation is no longer a triangulation but we can complete it to a triangulation by decomposing each j-gon into j − 2 triangles. This corresponds to perturbing the data points every so slightly such that the degree-j Voronoi vertices are resolved into trees in which j − 2 degree-3 vertices are connected by j − 3 tiny edges.

Local Delaunayhood. Given a triangulation of a finite point set S, we can test whether or not it is the Delaunay triangulation by testing each edge against the two triangles that share the edge. Suppose the edge uv in the triangulation T is shared by the triangles uvp and uvq. We call uv locally Delaunay, or lD for short, if q lies on or outside the circle that passes through u, v, p. The condition is symmetric in p and q because the circle that passes through u, v, p intersects the first circle in points u and v. It follows that p lies on or outside the circle of u, v, q iff q lies on or outside the circle of u, v, p. We also call uv locally Delaunay if it bounds the convex hull of S and thus belongs to only one triangle. The local condition on the edges implies a global property.

DELAUNAY LEMMA. If every edge in a triangulation K of S is locally Delaunay then K is the Delaunay triangulation of S.

Although every edge of the Delaunay triangulation is locally Delaunay, the Delaunay Lemma is not trivial. Indeed, K may contain edges that are locally Delaunay but do not belong to the Delaunay triangulation, as shown in Figure e. We omit the proof of the lemma.

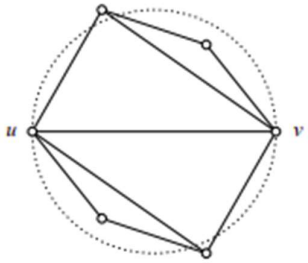


Figure e: The edge uv is locally Delaunay but does not belong to the Delaunay triangulation.

Edge-flipping. The Delaunay Lemma suggests we construct the Delaunay triangulation by first constructing an arbitrary triangulation of the point set S and then modifying it locally to make all edges lD. The idea is to look for non-lD edges and to flip them, as illustrated in Figure f. Indeed, if uv is a non-lD edge shared by triangles uvp and uvq then upvq is a convex quadrilateral and flipping uv means substituting one diagonal for the other, namely pq for uv.

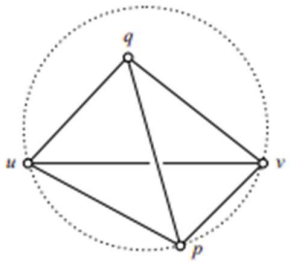


Figure f: The edge uv is non-lD and can be flipped to the edge pq, which is lD.

Note that if uv is non-lD then pq is lD. It is important that the algorithm finds non-lD edges quickly. For this purpose, we use a stack of edges. Initially, we push all edges on the stack and mark them.

```
while stack is non-empty do
        pop edge uv from stack and unmark it;
        if uv is non-lD then
                substitute pq for uv;
                for ab ∈ {up, pv, vq, qu} do
                        if ab is unmarked then push ab on the stack and mark it
                        endif
                endfor
        endif
Endwhile.
```

The marks avoid multiple copies of the same edge on the stack. This implies that at any one moment the size of the stack is less than 3n. Note also that initially the stack contains all non-lD edges and that this property is maintained as an invariant of the algorithm. The Delaunay Lemma implies that when the algorithm halts, which is when the stack is empty,

then the triangulation is the Delaunay triangulation. However, it is not yet clear that the algorithm terminates. Indeed, the stack can grow and shrink during the course of the algorithm, which makes it difficult to prove that it ever runs empty.

In-circle test. Before studying the termination of the algorithm, we look into the question of distinguishing lD from non-lD edges. As before we assume that the edge uv is shared by the triangles uvp and uvq in the current triangulation. Recall that uv is lD iff q lies outside the circle that passes through u, v, p. Let f : R 2 → R be defined by f(x) = x 2 1 + x 2 2 . As illustrated in Figure g, the graph of this function is a paraboloid in three-dimensional space and we write x + = (x1, x2, f(x)) for the vertical projection of the point x onto the paraboloid. Assuming the three points u, v, p do not lie on a common line then the points u +, v+, p+ lie on a non-vertical plane that is the graph of a function h(x) = αx1 + βx2 + γ. The projection of the intersection of the paraboloid and the plane back into R 2 is given by

0 = f(x) − h(x)

= x 2 1 + x 2 2 − αx1 − βx2 − γ,

which is the equation of a circle. This circle passes through u, v, p so it is the circle we have to compare q against.

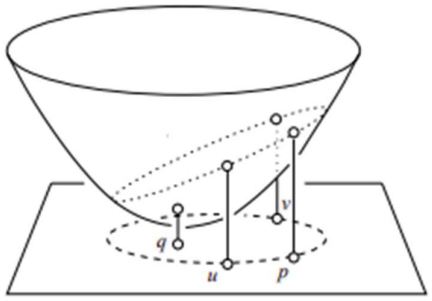


Figure g: The plane passing through u +, v+, p+ intersects the paraboloid in an ellipse whose projection into R 2 passes through the points u, v, p. The point q + lies below the plane iff q lies inside the circle.

We note that q lies inside the circle iff q + lies below the plane. The latter test can be based on the sign of the determinant of the 4-by-4 matrix

$$\Delta = \begin{bmatrix} 1 & u_1 & u_2 & u_1^2+u_2^2 \\ 1 & v_1 & v_2 & v_1^2+v_2^2 \\ 1 & p_1 & p_2 & p_1^2+p_2^2 \\ 1 & q_1 & q_2 & q_1^2+q_2^2 \end{bmatrix}.$$

Exchanging two rows in the matrix changes the sign. While the in-circle test should be insensitive to the order of the first three points, the sign of the determinant is not. We correct the change using the sign of the determinant of the 3-by-3 matrix that keeps track of the ordering of u, v, p along the circle,

$$\Gamma = \begin{bmatrix} 1 & u_1 & u_2 \\ 1 & v_1 & v_2 \\ 1 & p_1 & p_2 \end{bmatrix}.$$

Now we claim that s is inside the circle of u, v, p iff the two determinants have opposite signs:

boolean INCIRCLE(Points u, v, p, q)

return det Γ · det Δ < 0.

We first show that the boolean function is correct for u = (0, 0), v = (1, 0), p = (0, 1), and q = (0, 0.5). The sign of the product of determinants remains unchanged if we continuously move the points and avoid the configurations that make either determinant zero, which are when u, v, p are collinear and when u, v, p, q are cocircular. We can change any configuration where q is inside the circle of u, v, p continuously into the special configuration without going through zero, which implies the correctness of the function for general input points.

Termination and running time. To prove the edge-flip algorithm terminates, we imagine the triangulation lifted to R 3 . We do this by projecting the vertices vertically onto the paraboloid, as before, and connecting them with straight edges and triangles in space. Let uv be an edge shared by triangles uvp and uvq that is flipped to pq by the algorithm. It follows the line segments uv and pq cross and their endpoints form a convex quadrilateral, as shown in Figure h. After lifting the two line segments, we get u +v + passing above p +q +. We may thus think of the flip as gluing the tetrahedron u +v +p +q + underneath the surface obtained by lifting the triangulation.

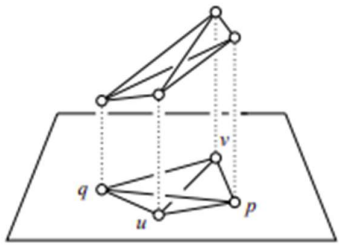


Figure h: A flip in the plane lifts to a tetrahedron in space in which the lD edge passes below the non-lD edge.

The surface is pushed down by each flip and never pushed back up. The removed edge is now above the new surface and can therefore not be reintroduced by a later flip. It follows that the algorithm performs at most n^2 flips and thus takes at most time $O(n^2)$ to construct the Delaunay triangulation of S. There are faster algorithms that work in time O(n log n) but we prefer the suboptimal method because it is simpler and it reveals more about Delaunay triangulations than the other algorithms.

The lifting of the input points to R 3 leads to an interesting interpretation of the edge-flip algorithm. Starting with a monotone triangulated surface passing through the lifted points, we glue tetrahedra below the surface until we reach the unique convex surface that passes through the points. The projection of this convex surface is the Delaunay triangulation of the points in the plane. This also gives a reinterpretation of the Delaunay Lemma in terms of convex and concave edges of the surface

Alpha shapes

Many practical applications of geometry have to do with the intuitive but vague concept of the shape of a finite point set. To make this idea concrete, we use the distances between the points to identify subcomplexes of the Delaunay triangulation that represent that shape at different levels of resolution.

Union of disks. Let S be a set of n points in R 2 . For each r ≥ 0, we write Bu(r) = {x ∈ R 2 | kx − uk ≤ r} for the closed disk with center u and radius r. Let U(r) = S u∈S Bu(r) be the union of the n disks. We decompose this union into convex sets of the form Ru(r) = Bu(r) ∩ Vu. Then

(i) Ru(r) is closed and convex for every point u ∈ S and every radius r ≥ 0;

(ii) Ru(r) and Rv(r) have disjoint interiors whenever the two points, u and v, are different;

(iii) U(r) = S u∈S Ru(r).

We illustrate this decomposition in Figure a. Each region Ru(r) is the intersection of n − 1 closed half-planes and a closed disk. All these sets are closed and convex, which implies (i). The Voronoi regions have disjoint interiors, which implies (ii). Finally, take a point x ∈ U(r) and let u be a point in S with x ∈ Vu. Then x ∈ Bu(r) and therefore x ∈ Ru(x). This implies (iii).

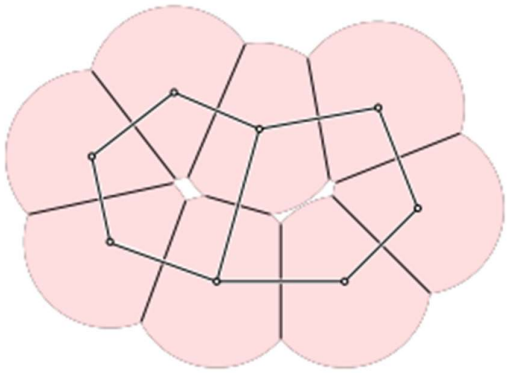


Figure a: The Voronoi decomposition of a union of eight disks in the plane and superimposed dual alpha complex.

Nerve. Similar to defining the Delaunay triangulation as the dual of the Voronoi diagram, we define the alpha complex as the dual of the Voronoi decomposition of the union of disks. This time around, we do this more formally. Letting C be a finite collection of sets, the nerve of C is the system of subcollections that have a non-empty common intersection,

$\text{Nrv } C = \{X \subseteq C \mid \setminus X \neq \emptyset\}$.

This is an abstract simplicial complex since $\text{T } X \neq \emptyset$ and $Y \subseteq X$ implies $\text{T } Y \neq \emptyset$. For example, if C is the collection of Voronoi regions then Nrv C is an abstract version of the Delaunay triangulation. More specifically, this is true provide the points are in general position and in particular no four points lie on a common circle. We will assume this for the remainder of this section. We say the Delaunay triangulation is a geometric realization of Nrv C, namely the one obtained by mapping each Voronoi region (a vertex in the abstract simplicial complex) to the generating point. All edges and triangles are just convex hulls of their incident vertices. To go from the Delaunay triangulation to the alpha complex, we substitute the regions Ru(r) for the Vu. Specifically,

$\text{Alpha}(r) = \text{Nrv } \{Ru(r) \mid u \in S\}$.

Clearly, this is isomorphic to a subcomplex of the nerve of Voronoi regions. We can therefore draw Alpha(r) as a subcomplex of the Delaunay triangulation; see Figure a. We call this geometric realization of Alpha(r) the alpha complex for radius r, denoted as A(r). The alpha shape for the same radius is the underlying space of the alpha complex, $|A(r)|$.

The nerve preserves the way the union is connected. In particular, their Betti numbers are the same, that is, $\beta p(U(r)) = \beta p(A(r))$ for all dimensions p and all radii r. This implies that the union and the alpha shape have the same number of components and the same number of holes. For example, in Figure 105 both have one component and two holes. We omit the proof of this property

Filtration. We are interested in the sequence of alpha shapes as the radius grows from zero to infinity. Since growing r grows the regions Ru(r), the nerve can only get bigger. In other words, $A(r) \subseteq A(s)$ whenever $r \leq s$. There are only finitely many subcomplexes of the Delaunay triangulation. Hence, we get a finite sequence of alpha complexes. Writing Ai for the i-th alpha complex, we get the following nested sequence,

$S = A1 \subset A2 \subset \ldots \subset Ak = D$

where D denotes the Delaunay triangulation of S. We call such a sequence of complexes a filtration. We illustrate this construction in Figure b. The sequence of alpha complexes begins with a set of n isolated vertices, the points in S.

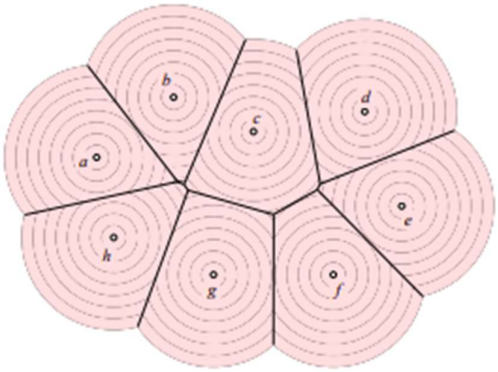


Figure b: A finite sequence of unions of disks, all decomposed by the same Voronoi diagram.

To go from one complex to the next, we either add an edge, we add a triangle, or we add a pair consisting of a triangle with one of its edges. In Figure b, we begin with eight vertices and get the following sequence of alpha complexes.

A1 = {a, b, c, d, e, f, g, h};

A2 = A1 ∪ {ah};

A3 = A2 ∪ {bc};

A4 = A3 ∪ {ab, ef };

A5 = A4 ∪ {de};

A6 = A5 ∪ {gh};

A7 = A6 ∪ {cd};

A8 = A7 ∪ {fg};

A9 = A8 ∪ {cg}.

Going from A7 to A8, we get for the first time a 1-cycle, which bounds a hole in the embedding. In A9, this hole is cut into two. This is the alpha complex depicted in Figure a. We continue.

A10 = A9 ∪ {cf };

A11 = A10 ∪ {abh, bh};

A12 = A11 ∪ {cde, ce};

A13 = A12 ∪ {cfg};

A14 = A13 ∪ {cef };

A15 = A14 ∪ {bch, ch};

A16 = A15 ∪ {cgh}.

At this moment, we have a triangulated disk but not yet the entire Delaunay triangulation since the triangle bcd and the edge bd are still missing. Each step is generic except when we add two equally long edges to A3.

Compatible ordering of simplices. We can represent the entire filtration of alpha complexes compactly by sorting the simplices in the order they join the growing complex. An ordering σ1, σ2, . . . , σm of the Delaunay simplices is compatible with the filtration if

1. the simplices in Ai precede the ones not in Ai for each i;

2. the faces of a simplex precede the simplex.

For example, the sequence

a, b, c, d, e, f, g, h; ah; bc; ab, ef ;

de; gh; cd; fg; cg; cf ; bh, abh; ce,

cde; cfg; cef ; ch, bch; cgh; bd; bcd

is compatible with the filtration in Figure b. Every alpha complex is a prefix of the compatible sequence but not necessarily the other way round. Condition 2 guarantees that every prefix is a complex, whether an alpha complex or not. We thus get a finer filtration of complexes

Ø = K0 ⊂ K1 ⊂ . . . ⊂ Km = D,

where Ki is the set of simplices from σ1 to σi . To construct the compatible ordering, we just need to compute for each Delaunay simplex the radius ri = r(σi) such that σi ∈ A(r) iff r ≥ ri . For a vertex, this radius is zero. For a triangle, this is the radius of the circumcircle. For an edge, we have two cases.

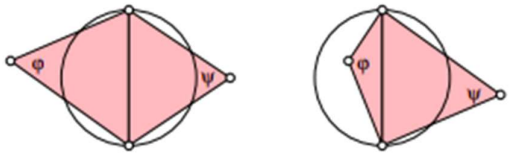


Figure c: Left: the middle edge belongs to two acute triangles. Right: it belongs to an obtuse and an acute triangle.

Let φ and ψ be the angles opposite the edge σi inside the two incident triangles. We have φ + ψ > 180◦ because of the empty circle property.

CASE 1. φ < 90◦ and ψ < 90◦ . Then ri = r(σi) is half the length of the edge.

CASE 2. φ ≥ 90◦ . Then ri = rj , where σj is the incident triangle with angle φ.

Both cases are illustrated in Figure 107. In Case 2, the edge σi enters the growing alpha complex together with the triangle σj . The total number of simplices in the Delaunay triangulation is m < 6n. The threshold radii can be computed in time O(n). Sorting the simplices into the compatible ordering can therefore be done in time O(n log n).

Betti numbers. In two dimensions, Betti numbers can be computed directly, without resorting to boundary matrices. The only two possibly non-zero Betti numbers are β0, the number of components, and β1, the number of holes. We compute the Betti numbers of Kj by adding the simplices in order.

β0 = β1 = 0;

for i = 1 to j do

switch dim σi : case 0: β0 = β0 + 1;

case 1: let u, v be the endpoints of σi ;

if FIND(u) = FIND(v) then β1 = β1 + 1

else β0 = β0 − 1;

UNION(u, v)

endif

case 2: β1 = β1 − 1

endswitch

Endfor.

All we need is tell apart the two cases when σi is an edge. This is done using a union-find data structure maintaining the components of the alpha complex in amortized time α(n) per simplex. The total running time of the algorithm for computing Betti numbers is therefore O(nα(n)).

CHAPTER 9

NP-COMPLETENESS

Easy and Hard Problems

The theory of NP-completeness is an attempt to draw a line between tractable and intractable problems. The most important question is whether there is indeed a difference between the two, and this question is still unanswered. Typical results are therefore relative statements such as "if problem B has a polynomial-time algorithm then so does problem C" and its equivalent contra-positive "if problem C has no polynomial-time algorithm then neither has problem B". The second formulation suggests we remember hard problems C and for a new problem B we first see whether we can prove the implication. If we can then we may not want to even try to solve problem B efficiently. A good deal of formalism is necessary for a proper description of results of this kind, of which we will introduce only a modest amount.

What is a problem? An abstract decision problem is a function $I \rightarrow \{0, 1\}$, where I is the set of problem instances and 0 and 1 are interpreted to mean FALSE and TRUE, as usual. To completely formalize the notion, we encode the problem instances in strings of zeros and ones: $I \rightarrow \{0,1\}^*$. A concrete decision problem is then a function $Q : \{0,1\}^* \rightarrow \{0, 1\}$. Following the usual convention, we map bit-strings that do not correspond to meaningful problem instances to 0.

As an example consider the shortest-path problem. A problem instance is a graph and a pair of vertices, u and v, in the graph. A solution is a shortest path from u and v, or the length of such a path. The decision problem version specifies an integer k and asks whether or not there exists a path from u to v whose length is at most k. The theory of NP-completeness really only deals with decision problems. Although this is a loss of generality, the loss is not dramatic. For example, given an algorithm for the decision version of the shortest-path problem, we can determine the length of the shortest path by repeated decisions for different values of k. Decision problems are always easier (or at least not harder) than the corresponding optimization problems. So in order to prove that an optimization problem is hard it suffices to prove that the corresponding decision problem is hard.

Polynomial time. An algorithm solves a concrete decision problem Q in time T (n) if for every instance $x \in \{0,1\}^*$ of length n the algorithm produces Q(x) in time at most T (n). Note

that this is the worst-case notion of time-complexity. The problem Q is polynomial-time solvable if $T(n) = O(n^k)$ for some constant k independent of n. The first important complexity class of problems is

P = set of concrete decision problems that are polynomial-time solvable.

The problems $Q \in P$ are called tractable or easy and the problems $Q \notin P$ are called intractable or hard. Algorithms that take only polynomial time are called efficient and algorithms that require more than polynomial time are inefficient. In other words, until now in this course we only talked about efficient algorithms and about easy problems. This terminology is adapted because the rather fine grained classification of algorithms by complexity we practiced until now is not very useful in gaining insights into the rather coarse distinction between polynomial and non-polynomial.

It is convenient to recast the scenario in a formal language framework. A language is a set $L \subseteq \{0,1\}^*$. We can think of it as the set of problem instances, x, that have an affirmative answer, Q(x) = 1. An algorithm $A : \{0,1\}^* \rightarrow \{0, 1\}$ accepts $x \in \{0,1\}^*$ if A(x) = 1 and it rejects x if A(x) = 0. The language accepted by A is the set of strings $x \in \{0,1\}^*$ with A(x) = 1. There is a subtle difference between accepting and deciding a language L. The latter means that A accepts every $x \in L$ and rejects every $x \notin L$. For example, there is an algorithm that accepts every program that halts, but there is no algorithm that decides the language of such programs. Within the formal language framework we redefine the class of polynomial-time solvable problems as

$P = \{L \subseteq \{0,1\}^* \mid L \text{ is accepted by a polynomial-time algorithm}\}$

$= \{L \subseteq \{0,1\}^* \mid L \text{ is decided by a polynomial-time algorithm}\}$.

Indeed, a language that can be accepted in polynomial time can also be decided in polynomial time: we keep track of the time and if too much goes by without x being accepted, we turn around and reject x. This is a nonconstructive argument since we may not know the constants in the polynomial. However, we know such constants exist which suffices to show that a simulation as sketched exists.

Hamiltonian cycles. We use a specific graph problem to introduce the notion of verifying a solution to a problem, as opposed to solving it. Let G = (V, E) be an undirected graph. A hamiltonian cycle contains every vertex $v \in V$ exactly once. The graph G is hamiltonian if it has a hamiltonian cycle. Figure a shows a hamiltonian cycle of the edge graph of a Platonic solid. How about the edge graphs of the other four Platonic solids? Define $L = \{G \mid G \text{ is hamiltonian}\}$.

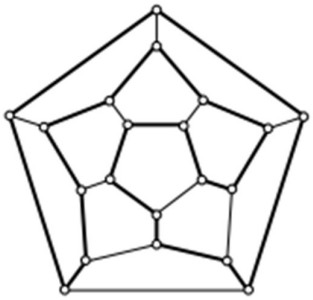


Figure b: The edge graph of the dodecahedron and one of its hamiltonian cycles.

We can thus ask whether or not L ∈ P, that is, whether or not there is a polynomial-time algorithm that decides whether or not a graph is hamiltonian. The answer to this question is currently not known, but there is evidence that the answer might be negative. On the other hand, suppose y is a hamiltonian cycle of G. The language L ′ = {(G, y) | y is a hamiltonian cycle of G} is certainly in P because we just need to make sure that y and G have the same number of vertices and every edge of y is also an edge of G.

Non-deterministic polynomial time. More generally, it seems easier to verify a given solution than to come up with one. In a nutshell, this is what NP-completeness is about, namely finding out whether this is indeed the case and whether the difference between accepting and verifying can be used to separate hard from easy problems. Call y ∈ {0,1}* a certificate. An algorithm A verifies a problem instance x ∈ {0,1}* if there exists a certificate y with A(x, y) = 1. The language verified by A is the set of strings x ∈ {0,1}* verified by A. We now define a new class of problems,

NP = {L ⊆ {0,1}* | L is verified by a polynomial-time algorithm}.

{0,1}*

More formally, L is in NP if for every problem instance x ∈ L there is a certificate y whose length is bounded from above by a polynomial in the length of x such that A(x, y) = 1 and A runs in polynomial time. For example, deciding whether or not G is hamiltonian is in NP.

The name NP is an abbreviation for non-deterministic polynomial time, because a non-deterministic computer can guess a certificate and then verify that certificate. In a parallel emulation, the computer would generate all possible certificates and then verify them in parallel. Generating one certificate is easy, because it only has polynomial length, but generating all of them is hard, because there are exponentially many strings of polynomial length.

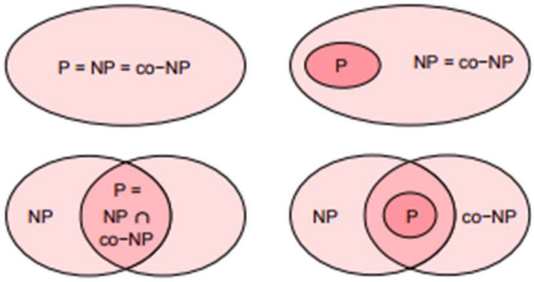


Figure b: Four possible relations between the complexity classes P, NP, and co-NP.

Non-deterministic machine are at least as powerful as deterministic machines. It follows that every problem in P is also in NP, P ⊆ NP. Define

co-NP = {L | $\overline{L}$ = {x ∉ L} ∈ NP},

which is the class of languages whose complement can be verified in non-deterministic polynomial time. It is not known whether or not NP = co-NP. For example, it seems easy to verify that a graph is hamiltonian but it seems hard to verify that a graph is not hamiltonian. We said earlier that if L ∈ P then $\overline{L}$ ∈ P. Therefore, P ⊆ co-NP. Hence, only the four relationships between the three complexity classes shown in Figure b are possible, but at this time we do not know which one is correct.

Problem reduction. We now develop the concept of reducing one problem to another, which is key in the construction of the class of NP-complete problems. The idea is to map or transform an instance of a first problem to an instance of a second problem and to map the solution to the second problem back to a solution to the first problem. For decision problems, the solutions are the same and need no transformation.

Language L_1 is polynomial-time reducible to language L_2, denoted $L_1 \leq_P L_2$, if there is a polynomial-time computable function f : {0,1}* → {0,1}* such that x ∈ L_1 iff f(x) ∈ L_2, for all x ∈ {0,1}*. Now suppose that L_1 is polynomial-time reducible to L_2 and that L_2 has a polynomial-time algorithm A_2 that decides L_2,

$x \xrightarrow{\mathbf{f}} f(x) \xrightarrow{A_2} \{0, 1\}$

We can compose the two algorithms and obtain a polynomial-time algorithm $A_1 = A_2 \circ f$ that decides L_1. In other words, we gained an efficient algorithm for L_1 just by reducing it to L_2.

REDUCTION LEMMA. If $L_1 \leq_P L2$ and $L_2 \in P$ then $L_1 \in P$.

In words, if L_1 is polynomial-time reducible to L_2 and L_2 is easy then L_1 is also easy. Conversely, if we know that L_1 is hard then we can conclude that L_2 is also hard. This motivates the following definition. A language $L \subseteq \{0,1\}^*$ is NP-complete if

(1) $L \in NP$;

(2) $L' \leq_P L$, for every $L' \in NP$.

Since every $L' \in NP$ is polynomial-time reducible to L, all L' have to be easy for L to have a chance to be easy. The L' thus only provide evidence that L might indeed be hard. We say L is NP-hard if it satisfies (2) but not necessarily (1). The problems that satisfy (1) and (2) form the complexity class

NPC = {L | L is NP-complete}.

All these definitions would not mean much if we could not find any problems in NPC. The first step is the most difficult one. Once we have one problem in NPC we can get others using reductions.

Satisfying boolean formulas. Perhaps surprisingly, a first NP-complete problem has been found, namely the problem of satisfiability for logical expressions. A boolean formula, φ, consists of variables, x1, x2, . . ., operators, ¬, ∧, ∨, ⟹, . . ., and parentheses. A truth assignment maps each variable to a boolean value, 0 or 1. The truth assignment satisfies if the formula evaluates to 1. The formula is satisfiable if there exists a satisfying truth assignment. Define SAT = {φ | φ is satisfiable}. As an example consider the formula

$\psi = (x1 \Longrightarrow x2) \Longleftrightarrow (x2 \vee \neg x1)$.

If we set x1 = x2 = 1 we get $(x1 \Longrightarrow x2) = 1$, $(x2 \vee \neg x1) = 1$ and therefore $\psi = 1$. It follows that $\psi \in SAT$.

In fact, all truth assignments evaluate to 1, which means that ψ is really a tautology. More generally, a boolean formula, φ, is satisfyable iff ¬φ is not a tautology.

SATISFIABILITY THEOREM. We have $SAT \in NP$ and $L' \leq_P SAT$ for every $L' \in NP$.

That SAT is in the class NP is easy to prove: just guess an assignment and verify that it satisfies. However, to prove that every $L' \in NP$ can be reduced to SAT in polynomial time is quite technical and we omit the proof. The main idea is to use the polynomial-time algorithm that verifies L' and to construct a boolean formula from this algorithm. To formalize this idea, we would need a formal model of a computer, a Touring machine, which is beyond the scope of this course.

NP-Complete Problems

In this section, we discuss a number of NP-complete problems, with the goal to develop a feeling for what hard problems look like. Recognizing hard problems is an important aspect of a reliable judgement for the difficulty of a problem and the most promising approach to a solution. Of course, for NP-complete problems, it seems futile to work toward polynomial-time algorithms and instead we would focus on finding approximations or circumventing the problems altogether. We begin with a result on different ways to write boolean formulas.

Reduction to 3-satisfiability. We call a boolean variable or its negation a literal. The conjunctive normal form is a sequence of clauses connected by ∧s, and each clause is a sequence of literals connected by ∨s. A formula is in 3-CNF if it is in conjunctive normal form and each clause consists of three literals. It turns out that deciding the satisfiability of a boolean formula in 3-CNF is no easier than for a general boolean formula. Define 3-SAT = {φ ∈ SAT | φ is in 3-CNF}. We prove the above claim by reducing SAT to 3-SAT.

SATISFIABILITY LEMMA. SAT ≤P 3-SAT.

PROOF. We take a boolean formula φ and transform it into 3-CNF in three steps.

Step 1. Think of φ as an expression and represent it as a binary tree. Each node is an operation that gets the input from its two children and forwards the output to its parent. Introduce a new variable for the output and define a new formula φ ′ for each node, relating the two input edges with the one output edge. Figure c shows the tree representation of the formula φ = (x1 ⟹ x2) ⟺ (x2 ∨ ¬x1). The new formula is

φ ′ = (y2 ⟺ (x1 ⟹ x2)) ∧(y3 ⟺ (x2 ∨ ¬x1)) ∧(y1 ⟺ (y2 ⟺ y3)) ∧ y1.

It should be clear that there is a satisfying assignment for φ iff there is one for φ ′ .

Step 2. Convert each clause into disjunctive normal form. The most mechanical way uses the truth table for each clause, as illustrated in Table 6. Each clause has at most three literals. For example, the negation of y2 ⟺ (x1 ⟹ x2) is equivalent to the disjunction of the conjunctions in the rightmost column. It follows that y2 ⟺ (x1 ⟹ x2) is equivalent to the negation of that disjunction, which by de Morgan's law is

(y2 ∨x1 ∨x2)∧(y2 ∨x1 ∨¬x2)∧(y2 ∨¬x1 ∨ ¬x2) ∧ (¬y2 ∨ ¬x1 ∨ x2).

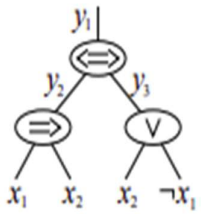


Figure c: The tree representation of the formula ϕ. Incidentally, ϕ is a tautology, which means it is satisfied by every truth assignment. Equivalently, ¬ϕ is not satisfiable.

y_2	x_1	x_2	$y_2 \Leftrightarrow (x_1 \Rightarrow x_2)$	prohibited
0	0	0	0	$\neg y_2 \wedge \neg x_1 \wedge \neg x_2$
0	0	1	0	$\neg y_2 \wedge \neg x_1 \wedge x_2$
0	1	0	1	
0	1	1	0	$\neg y_2 \wedge x_1 \wedge x_2$
1	0	0	1	
1	0	1	1	
1	1	0	0	$y_2 \wedge x_1 \wedge \neg x_2$
1	1	1	1	

Table: Conversion of a clause into a disjunction of conjunctions of at most three literals each.

Step 3. The clauses with fewer than three literals can be expanded by adding new variables. For example a ∨ b is expanded to (a ∨ b ∨ p) ∧ (a ∨ b ∨ ¬p) and (a) is expanded to

(a ∨ p ∨ q) ∧ (a ∨ p ∨ ¬q) ∧ (a ∨ ¬p ∨ q) ∧ (a ∨ ¬p ∨ ¬q).

Each step takes only polynomial time. At the end, we get an equivalent formula in 3-conjunctive normal form.

We note that clauses of length three are necessary to make the satisfiability problem hard. Indeed, there is a polynomial-time algorithm that decides the satisfiability of a formula in 2-CNF.

NP-completeness proofs. Using polynomial-time reductions, we can show fairly mechanically that problems are NP-complete, if they are. A key property is the transitivity of ≤P , that is, if L′ ≤P L_1 and L_1 ≤P L_2 then L ′ ≤P L_2, as can be seen by composing the two polynomial-time computable functions to get a third one.

REDUCTION LEMMA. Let L_1, L_2 ⊆ {0,1}* and assume L_1 ≤P L_2. If L_1 is NP-hard and L_2 ∈ NP then L_2 ∈ NPC.

A generic NP-completeness proof thus follows the steps outline below

Step 1. Prove that L_2 ∈ NP.

Step 2. Select a known NP-hard problem, L_1, and find a polynomial-time computable function, f, with x ∈ L_1 iff f(x) ∈ L_2.

This is what we did for L_2 = 3-SAT and L_1 = SAT. Therefore 3-SAT ∈ NPC. Currently, there are thousands of problems known to be NP-complete. This is often considered evidence that P ≠ NP, which can be the case only if P ∩ NPC = ∅, as drawn in Figure d.

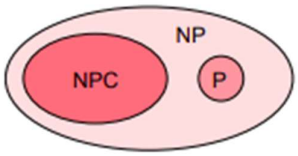


Figure d: Possible relation between P, NPC, and NP.

Cliques and independent sets. There are many NP-complete problems on graphs. A typical such problem asks for the largest complete subgraph. Define a clique in an undirected graph G = (V, E) as a subgraph (W, F) with F = $\binom{w}{2}$. Given G and an integer k, the CLIQUE problem asks whether or not there is a clique of k or more vertices.

CLAIM. CLIQUE ∈ NPC.

PROOF. Given k vertices in G, we can verify in polynomial time whether or not they form a complete graph. Thus CLIQUE ∈ NP. To prove property (2), we show that 3-SAT ≤P CLIQUE. Let φ be a boolean formula in 3-CNF consisting of k clauses. We construct a graph as follows:

(i) each clause is replaced by three vertices;

(ii) two vertices are connected by an edge if they do not belong to the same clause and they are not negations of each other.

In a satisfying truth assignment, there is at least one true literal in each clause. The true literals form a clique. Conversely, a clique of k or more vertices covers all clauses and thus implies a satisfying truth assignment.

It is easy to decide in time $O(k^2n^{k+2})$ whether or not a graph of n vertices has a clique of size k. If k is a constant, the running time of this algorithm is polynomial in n. For the CLIQUE problem to be NP-complete it is therefore essential that k be a variable that can be arbitrarily large. We use the NP-completeness of finding large cliques to prove the NP-completeness of large sets of pairwise nonadjacent vertices. Let G = (V, E) be an undirected graph. A subset W ⊆ V is independent if none of the vertices in W are adjacent or, equivalently, if E ∩ $\binom{w}{2}$ = ∅. Given G and an integer k, the INDEPENDENT SET problem asks whether or not there is an independent set of k or more vertices.

CLAIM. INDEPENDENT SET ∈ NPC.

PROOF. It is easy to verify that there is an independent set of size k: just guess a subset of k vertices and verify that no two are adjacent

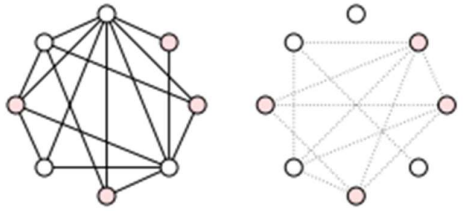


Figure e: The four shaded vertices form an independent set in the graph on the left and a clique in the complement graph on the right.

We complete the proof by reducing the CLIQUE to the INDEPENDENT SET problem. As illustrated in Figure e, W ⊆ V is independent iff W defines a clique in the complement graph, G = (V, $\binom{V}{2}$ − E). To prove CLIQUE ≤ P INDEPENDENT SET, we transform an instance H, k of the CLIQUE problem to the instance G = $\overline{H}$, k of the INDEPENDENT SET problem. G has an independent set of size k or larger iff H has a clique of size k or larger.

. Various NP-complete graph problems. We now describe a few NP-complete problems for graphs without proving that they are indeed NP-complete. Let G = (V, E) be an undirected graph with n vertices and k a positive integer, as before. The following problems defined for G and k are NP-complete.

An ℓ-coloring of G is a function χ : V → [ℓ] with χ(u) ≠ χ(v) whenever *u* and *v* are adjacent. The CHROMATIC NUMBER problem asks whether or not G has an ℓcoloring with ℓ ≤ k. The problem remains NP-complete for fixed k ≥ 3. For k = 2, the CHROMATIC NUMBER problem asks whether or not G is bipartite, for which there is a polynomial-time algorithm.

The *bandwidth* of G is the minimum ℓ such that there is a bijection β : V → [n] with |β(u) − β(v)| ≤ ℓ for all adjacent vertices *u* and *v*. The BANDWIDTH problem asks whether or not the bandwidth of G is k or less. The problem arises in linear algebra, where we permute rows and columns of a matrix to move all non-zero elements of a square matrix as close to the diagonal as possible. For example, if the graph is a simple path then the bandwidth is 1, as can be seen in Figure f. We can transform the adjacency matrix of G such that all non-zero diagonals are at most the bandwidth of G away from the main diagonal.

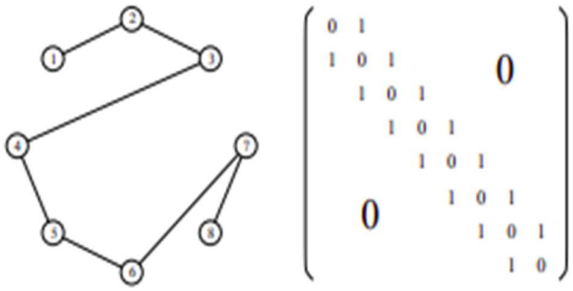


Figure f: Simple path and adjacency matrix with rows and columns ordered along the path.

Assume now that the graph G is complete, E = $\binom{v}{2}$, and that each edge, *uv*, has a positive integer weight, *w(uv)*. The TRAVELING SALESMAN problem asks whether there is a permutation $u_0, u_1, \ldots, u_{n-1}$ of the vertices such that the sum of edges connecting contiguous vertices (and the last vertex to the first) is k or less

$$\sum_{i=0}^{n-1} w(u_i u_{i+1}) \leq k,$$

where indices are taken modulo n. The problem remains NP-complete if w : E → {1, 2} (reduction to HAMILTONIAN CYCLE problem), and also if the vertices are points in the plane and the weight of an edge is the Euclidean distance between the two endpoints.

Set systems. Simple graphs are set systems in which the sets contain only two elements. We now list a few NP-complete problems for more general set systems. Letting V be a finite set, C ⊆ 2^V a set system, and k a positive integer, the following problems are NP-complete.

The PACKING problem asks whether or not C has k or more mutually disjoint sets. The problem remains NP complete if no set in C contains more than three elements, and there is a polynomial-time algorithm if every set contains two elements. In the latter case, the set system is a graph and a maximum packing is a maximum matching.

The COVERING problem asks whether or not C has k or fewer subsets whose union is V . The problem remains NP-complete if no set in C contains more than three elements, and there is a polynomial-time algorithm if every sets contains two elements. In the latter case, the set system is a graph and the minimum cover can be constructed in polynomial time from a maximum matching.

Suppose every element v ∈ V has a positive integer weight, w(v). The PARTITION problem asks whether there is a subset U ⊆ V with

$$\sum_{u \in U} w(u) \quad = \quad \sum_{v \in V-U} w(v).$$

The problem remains NP-complete if we require that U and V − U have the same number of elements.

Approximation Algorithms

Many important problems are NP-hard and just ignoring them is not an option. There are indeed many things one can do. For problems of small size, even exponential time algorithms can be effective and special subclasses of hard problems sometimes have polynomial-time algorithms. We consider a third coping strategy appropriate for optimization problems, which is computing almost optimal solutions in polynomial time. In case the aim is to maximize a positive cost, a σ(n)-approximation algorithm is one that guarantees to find a solution with cost $C \geq C^*/\sigma(n)$, where C^* is the maximum cost. For minimization problems, we would require $C \leq C^* \sigma(n)$. Note that $\sigma(n) \geq 1$ and if $\sigma(n) = 1$ then the algorithm produces optimal solutions. Ideally, σ is a constant but sometime even this is not achievable in polynomial time.

Vertex cover. The first problem we consider is finding the minimum set of vertices in a graph G = (V, E) that covers all edges. Formally, a subset $V' \subseteq V$ is a vertex cover if every edge has at least one endpoint in V ’. Observe that V ’ is a vertex cover iff $V - V'$ is an independent set. Finding a minimum vertex cover is therefore equivalent to finding a maximum independent set. Since the latter problem is NP-complete, we conclude that finding a minimum vertex cover is also NP-complete. Here is a straightforward algorithm that achieves approximation ratio $\sigma(n) = 2$, for all $n = |V|$.

V ’ = ∅; E’ = E;

while E’ ≠ ∅ do

select an arbitrary edge uv in E ’;

add u and v to V ’;

remove all edges incident to u or v from E’

endwhile.

Clearly, V ’ is a vertex cover. Using adjacency lists with links between the two copies of an edge, the running time is O(n + m), where *m* is the number of edges. Furthermore, we have σ = 2 because every cover must pick at least one vertex of each edge *uv* selected by the algorithm, hence $C \leq 2C^*$. Observe that this result does not imply a constant approximation ratio for the maximum independent set problem. We have $|V - V'| = n - C \geq n - 2C^*$, which we have to compare with $n - C^*$, the size of the maximum independent set. For $C^* = \frac{n}{2}$, the approximation ratio is unbounded.

Let us contemplate the argument we used to relate C and C^*. The set of edges *uv* selected by the algorithm is a *matching*, that is, a subset of the edges so that no two share a vertex. The size of the minimum vertex cover is at least the size of the largest possible matching.

The algorithm finds a matching and since it picks two vertices per edge, we are guaranteed at most twice as many vertices as needed. This pattern of bounding C^* by the size of another quantity (in this case the size of the largest matching) is common in the analysis of approximation algorithms. Incidentally, for bipartite graphs, the size of the largest matching is equal to the size of the smallest vertex cover. Furthermore, there is a polynomial-time algorithm for computing them.

Traveling salesman. Second, we consider the traveling salesman problem, which is formulated for a complete graph G = (V, E) with a positive integer cost function c : E → Z_+. A tour in this graph is a Hamiltonian cycle and the problem is finding the tour, A, with minimum total cost, c(A) = $\sum_{uv \in A} c(uv)$. Let us first assume that the cost function satisfies the triangle inequality, *c(uw) ≤ c(uv) + c(vw)* for all u, v, w ∈ V . It can be shown that the problem of finding the shortest tour remains NP-complete even if we restrict it to weighted graphs that satisfy this inequality. We formulate an algorithm based on the observation that the cost of every tour is at least the cost of the minimum spanning tree, $C^* \geq c(T)$.

1 Construct the minimum spanning tree T of G.

2 Return the preorder sequence of vertices in T .

Using Prim's algorithm for the minimum spanning tree, the running time is $O(n^2)$. Figure g illustrates the algorithm. The preorder sequence is only defined if we have a root and the neighbors of each vertex are ordered, but we may choose both arbitrarily.

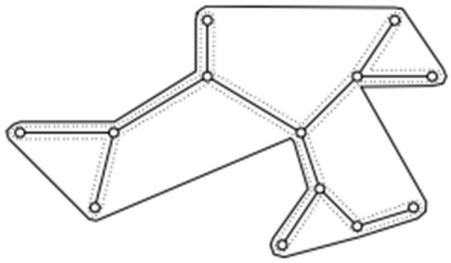


Figure g: The solid minimum spanning tree, the dotted traversal using each edge of the tree twice, and the solid tour obtained by taking short-cuts.

The cost of the returned tour is at most twice the cost of the minimum spanning tree. To see this, consider traversing each edge of the minimum spanning tree twice, once in each direction. Whenever a vertex is visited more than once, we take the direct edge connecting the two neighbors of the second copy as a short-cut. By the triangle inequality, this substitution can only decrease the overall cost of the traversal. It follows that $C \leq 2c(T) \leq 2C*$.

The triangle inequality is essential in finding a constant approximation. Indeed, without it we can construct instances of the problem for which finding a constant approximation is NP-

hard. To see this, transform an unweighted graph G′ = (V ′, E′) to the complete weighted graph G = (V, E) with

c(uv) = $\begin{cases} 1 & if\ uv\ \in E' \\ \sigma n + 1 & \text{otherwise} \end{cases}$

Any σ-approximation algorithm must return the Hamiltonian cycle of G′ , if there is one.

Set cover. Third, we consider the problem of covering a set X with sets chosen from a set system F. We assume the set is the union of sets in the system, X = ∪ F. More precisely, we are looking for a smallest subsystem F ′ ⊆ F with X = ∪ F ′ . The cost of this subsystem is the number of sets it contains, |F′ |. See Figure h for an illustration of the problem. The vertex cover problem is a special case: X = E and F contains all subsets of edges incident to a common vertex. It is special because each element (edge) belongs to exactly two sets. Since we no longer have a bound on the number of sets containing a single element, it is not surprising that the algorithm for vertex covers does not extend to a constant-approximation algorithm for set covers.

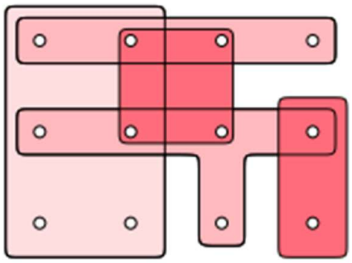


Figure h: The set X of twelve dots can be covered with four of the five sets in the system.

Instead, we consider the following greedy approach that selects, at each step, the set containing the maximum number of yet uncovered elements.

F ′ = ∅; X′ = X;

while X′≠ ∅ do

select S ∈ F maximizing |S ∩ X′ |;

F ′ = F ′ ∪ {S}; X′ = X′ − S

endwhile.

Using a sparse matrix representation of the set system (similar to an adjacency list representation of a graph), we can run the algorithm in time proportional to the total size of the sets in the system, $n = \sum_{S \in F} |S|$. We omit the details.

Analysis. More interesting than the running time is the analysis of the approximation ratio the greedy algorithm achieves. It is convenient to have short notation for the d-th harmonic number, $H_d = \sum_{i=1}^{d} \frac{1}{i}$ for d ≥ 0. Recall that $H_d \leq 1 + lnd$ for d ≥ 1. Let the size of the largest set in the system be m = max{|S| | S ∈ F }.

CLAIM. The greedy method is an H_m-approximation algorithm for the set cover problem.

PROOF. For each set S selected by the algorithm, we distribute $1 over the |S ∩ X′ | elements covered for the first time. Let cx be the cost allocated this way to x ∈ X. We have $|F'| = \sum_{x \in X} c_x$. If x is covered the first time by the *i*-th selected set, S_i , then

$$c_x = \frac{1}{|S_i - (u_i \cup \ldots \cup S_{i-1}|}$$

We have |F′ | ≤ P S∈F∗ P x∈S cx because the optimal cover, F ∗ , contains each element x at least once. We will prove shortly that P x∈S cx ≤ H|S| for every set S ∈ F. It follows that

$$|\mathcal{F}'| \leq \sum_{S \in \mathcal{F}^*} H_{|S|} \leq H_m |\mathcal{F}^*|,$$

as claimed.

For m = 3, we get σ = $H_3 = \frac{11}{6}$. This implies that for graphs with vertex-degrees at most 3, the greedy algorithm guarantees a vertex cover of size at most $\frac{11}{6}$ times the optimum, which is better than the ratio 2 guaranteed by our first algorithm.

We still need to prove that the sum of costs c_x over the elements of a set S in the system is bounded from above by $H_{|S|}$. Let u_i be the number of elements in S that are not covered by the first *i* selected sets, u_i = |S − (S_1 ∪ . . . ∪ S_i)|, and observe that the numbers do not increase. Let u_k−1 be the last non-zero number in the sequence, so |S| = u_0 ≥ . . . ≥ u_{k-1} > u_k = 0. Since $u_i - 1 - u_i$ is the number of elements in S covered the first time by S_i, we have

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} \frac{u_{i-1} - u_i}{|S_i - (S_1 \cup \ldots \cup S_{i-1})|}.$$

We also have u_{i-1} ≤ | S_i − (S_1 ∪ . . . ∪ S_{i-1})|, for all i ≤ k, because of the greedy choice of Si . If this were not the case, the algorithm would have chosen S instead of S_i in the construction of F′. The problem thus reduces to bounding the sum of ratios $\frac{u_i - 1 - u_i}{u_i - 1}$. It is

not difficult to see that this sum can be at least logarithmic in the size of S. Indeed, if we choose u_i about half the size of u_i−1, for all i ≥ 1, then we have logarithmically many terms, each roughly $\frac{1}{2}$. We use a sequence of simple arithmetic manipulations to prove that this lower bound is asymptotically tight:

$$\begin{aligned}\sum_{x\in S} c_x &\leq \sum_{i=1}^{k} \frac{u_{i-1} - u_i}{u_{i-1}} \\ &= \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}}.\end{aligned}$$

We now replace the denominator by j ≤ u_i−1 to form a telescoping series of harmonic numbers and get

$$\begin{aligned}\sum_{x\in S} c_x &\leq \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} \\ &= \sum_{i=1}^{k} \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right) \\ &= \sum_{i=1}^{k} (H_{u_{i-1}} - H_{u_i}).\end{aligned}$$

This is equal to H_{u_o} - H_{u_k} = $H_{|S|}$, which fills the gap left in the analysis of the greedy algorithm.