



UNIVERSITY
OF TRIESTE

Algorithmic Design

University of Trieste – a.a. 2026/2027

Prof. Antonio Emanuele Cinà

`antonio.cina@units.it`

Department of Mathematics, Informatics and Geosciences

University of Trieste

a.a. 2026/2027

Chapter 6:

The limits of comparison sorting

Can stronger assumptions make sorting faster?

The methods studied so far discover order by comparing pairs of keys.

What the $n \log n$ lower bound says

Every deterministic comparison-sorting algorithm has inputs requiring $\Omega(n \log n)$ comparisons.

What changes if we already know the key range or can inspect individual digits?

- **Counting sort** uses a bounded integer key as an array index.
- **Radix sort** uses an order-preserving digit representation.

We can do better under these conditions, but must include the range and digit parameters in the cost.

State how the algorithm may learn key order

Definition (Comparison-sorting model)

A sorting algorithm works in the *comparison model* if it obtains all information about relative order by comparing two keys. It may move records and use auxiliary storage, but it does not inspect individual digits or use key values as array positions.

For n distinct keys, there are $n!$ possible relative input orders. A deterministic algorithm must distinguish all of them.

An execution follows a path of comparison results

- An internal node is one key comparison.
- An outgoing edge records a possible result.
- A root-to-leaf path is the comparison history of an input.
- A leaf identifies the permutation needed to sort that input.

With distinct keys, each comparison has two relevant outcomes. The longest path measures the worst-case number of comparisons.

Read questions, branches and leaves

At an internal node

The question $a_i \leq a_j$ compares two original input elements. The shorthand $i : j$ means the same comparison.

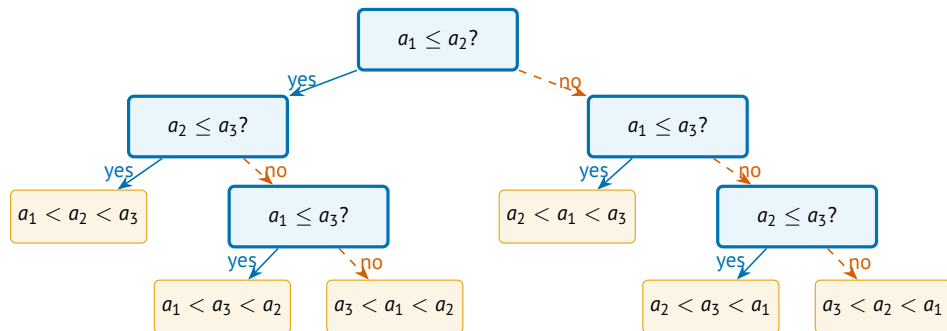
- Follow the solid blue **yes** branch when $a_i \leq a_j$.
- Follow the dashed orange **no** branch when $a_i > a_j$.

The elements are distinct, so equality needs no separate branch.

At a leaf

$\langle 1, 3, 2 \rangle$ means $a_1 < a_3 < a_2$. The tuple entries are *original indices*, not numerical values. The figure writes the inequalities explicitly.

One tree represents all executions of this algorithm



$3! = 6$ reachable leaves, so some path has length at least $\lceil \log_2 6 \rceil = 3$

Each input follows one root-to-leaf path. The labels a_1, a_2, a_3 always refer to the original input elements.

Follow an input through three comparisons

Let $a_1 = 6$, $a_2 = 8$, and $a_3 = 5$.

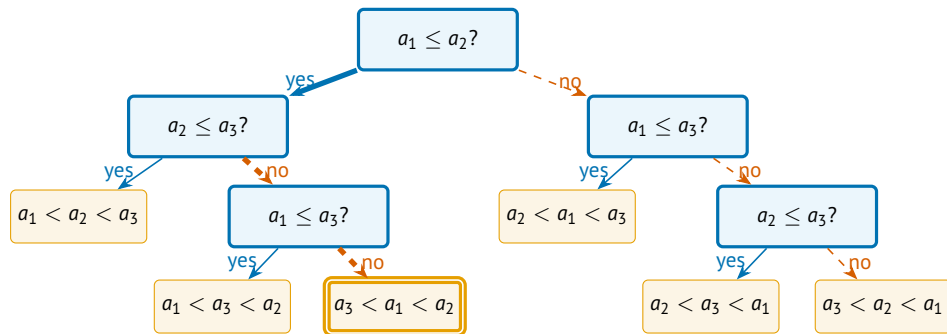
Step	Question	Evaluation	Branch
1	$a_1 \leq a_2?$	$6 \leq 8$: true	yes, solid blue
2	$a_2 \leq a_3?$	$8 \leq 5$: false	no, dashed orange
3	$a_1 \leq a_3?$	$6 \leq 5$: false	no, dashed orange

We reach the leaf $a_3 < a_1 < a_2$, or $\langle 3, 1, 2 \rangle$.

$$a_3 < a_1 < a_2 \quad \Longleftrightarrow \quad 5 < 6 < 8.$$

The path has three comparisons; it identifies the sorted order of the original records.

Locate the execution in the full tree



$3! = 6$ reachable leaves, so some path has length at least $\lceil \log_2 6 \rceil = 3$

For $(a_1, a_2, a_3) = (6, 8, 5)$, follow the thicker edges: yes, no, no. The double-outlined leaf gives $a_3 < a_1 < a_2$.

Six possible orders require six reachable leaves

Three distinct elements have $3! = 6$ possible relative orders.

Original indices in sorted order	Corresponding inequalities
$\langle 1, 2, 3 \rangle$	$a_1 < a_2 < a_3$
$\langle 1, 3, 2 \rangle$	$a_1 < a_3 < a_2$
$\langle 2, 1, 3 \rangle$	$a_2 < a_1 < a_3$
$\langle 2, 3, 1 \rangle$	$a_2 < a_3 < a_1$
$\langle 3, 1, 2 \rangle$	$a_3 < a_1 < a_2$
$\langle 3, 2, 1 \rangle$	$a_3 < a_2 < a_1$

A correct comparison sort must distinguish all six possibilities. Each requires a reachable leaf identifying its order.

Two comparisons cannot distinguish six orders

A binary decision tree of height h has at most 2^h leaves.

Maximum comparisons along a path	Maximum leaves
1	2
2	$2^2 = 4$
3	$2^3 = 8$

Six possible orders require

$$h \geq \lceil \log_2 6 \rceil = 3.$$

This is a worst-case claim

In the displayed tree, an already ordered input follows yes, yes and needs only two comparisons. The longest path needs three.

Count the leaves to bound the worst execution

Theorem (Comparison-sorting lower bound)

Every deterministic comparison-sorting algorithm performs $\Omega(n \log n)$ comparisons in the worst case on n distinct keys.

A binary tree of height h has at most 2^h leaves. Sorting needs at least $n!$ leaves, so

$$2^h \geq n! \implies h \geq \log_2(n!).$$

It remains to show that this logarithm grows at least as fast as $n \log n$.

The largest factors already force the lower bound

For $n \geq 2$, at least $n/2$ of the factors of $n!$ are at least $n/2$. Hence

$$n! \geq \left(\frac{n}{2}\right)^{n/2}, \quad \log_2(n!) \geq \frac{n}{2} \log_2 \frac{n}{2} = \Omega(n \log n).$$

Some input must follow a path at least this long.

Keep the scope of the conclusion

This is a worst-case comparison bound. It does not require every input to take $n \log n$ comparisons, and it does not cover digit inspection or direct indexing.

The logarithm of the factorial grows as $n \log n$

The preceding estimate gives $\log_2(n!) = \Omega(n \log n)$. For the matching upper bound, every factor of $n!$ is at most n :

$$n! = 1 \cdot 2 \cdots n \leq n^n.$$

Taking logarithms,

$$\log_2(n!) \leq n \log_2 n = O(n \log n).$$

Together,

$$\log_2(n!) = \Theta(n \log n).$$

Thus $h \geq \log_2(n!)$ forces a worst-case lower bound of $\Omega(n \log n)$ comparisons. It does not assert that every comparison algorithm achieves this running time.

Use a known integer range as an index set

Definition (Bounded integer-key instance)

Counting sort accepts n records whose keys belong to

$$\{0, 1, \dots, k\}.$$

For keys in an interval $[L, U]$, subtracting L gives the equivalent range $\{0, \dots, U - L\}$. The size of this range is part of the cost of the algorithm.

There are $k + 1$ counters, one per possible key. Direct indexing replaces pairwise comparisons.

For $[L, U]$, use index $\text{key}(x) - L$ while preserving the original record. The counter array has $U - L + 1$ positions.

Counts become positions for complete records

Start with $A = [2, 5, 3, 0, 2, 3, 0, 3]$ and keys from 0 to 5.

1. Count occurrences: $C = [2, 0, 2, 3, 0, 1]$.
2. Accumulate counts: $C = [2, 2, 4, 7, 7, 8]$.
3. Place complete records from right to left in the output B .

Frequencies tell us which values occur. Cumulative counts also locate the output block reserved for each key.

For example, $C[2] = 4$: keys at most 2 occupy positions 0, 1, 2, 3. The last available position for key 2 is $C[2] - 1 = 3$.

Keep positions and key values distinct

Zero-based arrays

$A[0 \dots n - 1]$ stores input records; $B[0 \dots n - 1]$ stores the output. The counter array $C[0 \dots k]$ is indexed by key value.

After cumulative counting,

$$C[v] = |\{x \in A \mid \text{key}(x) \leq v\}|.$$

The rightmost available position for key v is $C[v] - 1$.

After writing a record there, decrease $C[v]$ by one. The next equal key uses the preceding position.

These conventions agree with the Python implementations used in the course.

Stable Counting sort places complete records

```
Counting-sort(A, k)
  C[0 .. k] <- all zeroes
  for j <- 0 to length(A)-1
    C[key(A[j])] <- C[key(A[j])] + 1
  for value <- 1 to k
    C[value] <- C[value] + C[value-1]
  B[0 .. length(A)-1] <- empty
  for j <- length(A)-1 downto 0
    value <- key(A[j])
    position <- C[value]-1
    B[position] <- A[j]
    C[value] <- C[value]-1
  return B
```

Turn frequencies into output boundaries

(a) Count occurrences

	0	1	2	3	4	5	6	7
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

(b) Cumulative counts

	0	1	2	3	4	5
C	2	2	4	7	7	8

$C[v]$ = number of records
with key at most v

The indices of C are key values, not input positions.

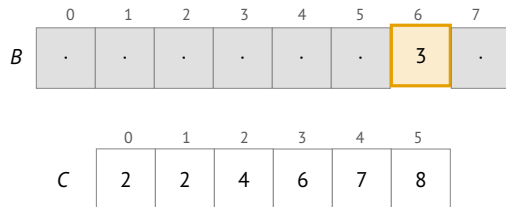
Initially, $C[3] = 3$ counts the three occurrences of key 3. After accumulation, $C[3] = 7$ counts all records with keys at most 3.

The rightmost free position for key 3 is therefore $C[3] - 1 = 6$.

Place the last input record

Input: $A = [2, 5, 3, 0, 2, 3, 0, 3]$. Scan from right to left.

(c) Place $A[7] = 3$



1. Read $A[7] = 3$.
2. Use $C[3] = 7$: position = $7 - 1 = 6$.
3. Write $B[6] \leftarrow 3$.
4. Decrease $C[3]$ to 6.

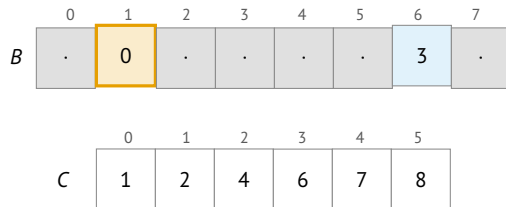
The diagram shows B and C *after* this placement.

Grey cells with dots are empty; blue cells were filled earlier. The orange outline marks the new entry at $B[6]$.

Place key 0 in its reserved block

Input: $A = [2, 5, 3, 0, 2, 3, 0, 3]$. Scan from right to left.

(d) Place $A[6] = 0$



1. Read $A[6] = 0$.
2. Use $C[0] = 2$: position = $2 - 1 = 1$.
3. Write $B[1] \leftarrow 0$.
4. Decrease $C[0]$ to 1.

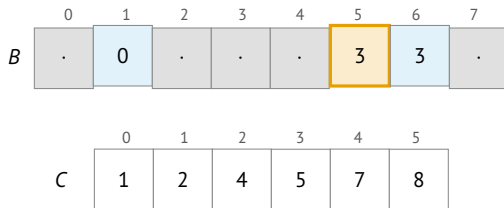
The diagram shows B and C *after* this placement.

Grey cells with dots are empty; blue cells were filled earlier. The orange outline marks the new entry at $B[1]$.

The next key 3 uses the preceding position

Input: $A = [2, 5, 3, 0, 2, 3, 0, 3]$. Scan from right to left.

(e) Place $A[5] = 3$



1. Read $A[5] = 3$.
2. Use $C[3] = 6$: position = $6 - 1 = 5$.
3. Write $B[5] \leftarrow 3$.
4. Decrease $C[3]$ to 5.

The diagram shows B and C *after* this placement.

Grey cells with dots are empty; blue cells were filled earlier. The orange outline marks the new entry at $B[5]$.

Read the completed output from left to right

(f) Final output

	0	1	2	3	4	5	6	7
<i>B</i>	0	0	2	2	3	3	3	5

All eight records have been placed. The output is sorted:

$$B = [0, 0, 2, 2, 3, 3, 3, 5].$$

The blocks for successive keys appear in increasing order. Each placement fills one free position and decreases the corresponding counter.

The next slide checks every placement in the reverse scan.

Complete the reverse scan

Input position j	Key $A[j]$	Position $C[A[j]] - 1$	Updated $C[A[j]]$
7	3	6	6
6	0	1	1
5	3	5	5
4	2	3	3
3	0	0	0
2	3	4	4
1	5	7	7
0	2	2	2

$$B = [0, 0, 2, 2, 3, 3, 3, 5].$$

Every record is placed once in its key block. Blocks occur in increasing key order.

Reverse placement preserves equal-key order

A sort is **stable** if records with equal keys retain their input order. Students with equal marks may still have different names and identifiers.

Tag the three occurrences of key 3

Their input order is $3_a, 3_b, 3_c$. The reverse scan encounters $3_c, 3_b, 3_a$, but fills their output block from right to left.

$$B[6] = 3_c, \quad B[5] = 3_b, \quad B[4] = 3_a.$$

Their output order remains $3_a, 3_b, 3_c$.

Scanning input left to right while filling right to left would reverse equal records. Values would remain sorted, but stability would be lost.

The key range contributes to time and space

Let n be the number of records and k the largest normalized key.

- Initialize $k + 1$ counters and accumulate them: $\Theta(k + 1)$ work.
- Count and place n records: $\Theta(n)$ work.
- Keep counters and an output array: $\Theta(n + k)$ auxiliary space for $n \geq 1$.

$$T(n, k) = \Theta(n + k), \quad S(n, k) = \Theta(n + k) \quad (n \geq 1).$$

Assume constant-time key access, array indexing and record movement.

Including empty input and $k = 0$, write $\Theta(n + k + 1)$ for time and allocated space.

Linear in records requires a bound on the range

Ten records with keys 0 and 10^9 still require $10^9 + 1$ counters in this implementation.

When can we call Counting sort linear in n ?

If $k = O(n)$, then $T(n, k) = \Theta(n)$ and auxiliary space is $\Theta(n)$.

The input description includes both the record count and the normalized range.

A small number of distinct values alone does not make the direct counter array small.

Inspect small digits instead of the whole key universe

Large keys may have a small number of digits drawn from a smaller alphabet.

Definition (LSD Radix sort requirements)

Least-significant-digit Radix sort, abbreviated LSD Radix sort, assumes that every key has d digit positions and that each digit belongs to $\{0, \dots, b - 1\}$. It applies a stable sorting method to positions $0, 1, \dots, d - 1$, beginning with the least significant position 0.

Digit values are $0, \dots, b - 1$. Each pass can use stable Counting sort with b counters.

Digit order must agree with the intended key order

Fixed-width nonnegative integers satisfy the representation requirement directly. Pad shorter numbers with leading zeroes.

Other representations need an explicit transformation

Other data types require a representation whose lexicographic digit order agrees with the desired key order.

The lexicographic order of transformed digits must agree with the intended order of keys. Stability alone cannot repair an incompatible representation.

Process digits from least to most significant

```
LSD-radix-sort(A, d, b)
  for position <- 0 to d-1
    A <- Stable-digit-sort(A, position, b)
  return A
```

Position 0 is the least significant digit. Each pass reorders complete records using just the current digit as its key.

Equal current digits must retain their order from the preceding pass.

Each stable pass orders a longer suffix

After p completed passes

Records are ordered by their p least significant digits.

The first pass orders the least significant digit.

In the next pass:

- Different current digits are put in increasing order.
- Equal current digits retain their already ordered suffixes, because the pass is stable.

After d passes, the suffix is the complete key.

The first pass orders the units digits

Input	329	457	657	839	436	720	355
After units	720	355	436	457	657	329	839
Units digit	0	5	6	7	7	9	9

The units digits are sorted; the complete keys are not yet sorted.

Within digit 7, 457 remains before 657. Within digit 9, 329 remains before 839.

These preserved orders become input to the tens pass.

The tens pass preserves the units order within ties

Before tens	720	355	436	457	657	329	839
After tens	720	329	436	839	355	457	657
Two-digit suffix	20	29	36	39	55	57	57

For tens digit 2, the previous units order keeps 720 before 329.

For tens digit 5, it keeps 355, 457, 657 in that order.

The two-digit suffixes are now ordered, even though the complete keys are not.

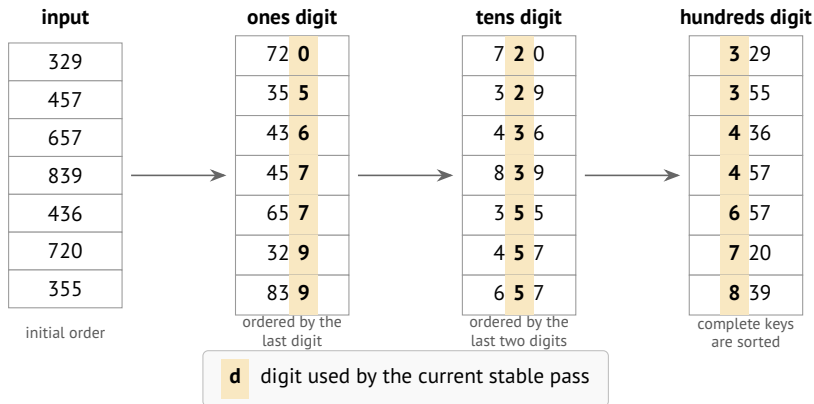
The final pass orders the complete keys

Before hundreds	720	329	436	839	355	457	657
After hundreds	329	355	436	457	657	720	839
Hundreds digit	3	3	4	4	6	7	8

Within hundreds digit 3, the suffix order 29 before 55 is retained. Within hundreds digit 4, 36 remains before 57.

All three digits have now been processed. The complete keys are sorted.

Follow the complete sequence of stable passes



An unstable pass can undo earlier work

After the units pass, the keys are ordered as

21, 22.

Their tens digits are both 2. An unstable tens pass may reverse this tie:

21, 22 $\longrightarrow$ 22, 21.

Stability is a requirement of LSD Radix sort

The current pass must preserve the suffix order established by previous passes whenever the current digits are equal.

Replacing the digit routine with an arbitrary in-place comparison sort does not, in general, give a valid LSD Radix sort.

Process the most important field last

The same stable-pass idea applies to records with several fields.

Sort dates into chronological order

1. Stably sort by day.
2. Stably sort by month, preserving day order within equal months.
3. Stably sort by year, preserving month-and-day order within equal years.

The final priority is year, then month, then day. The most important field is processed last.

Count both the passes and the digit alphabet

With Counting sort as the stable digit routine, one pass takes $\Theta(n + b)$ time and $\Theta(n + b)$ auxiliary space.

$$T(n, d, b) = \Theta(d(n + b)), \quad S(n, b) = \Theta(n + b).$$

Reuse the working arrays between passes; space is not multiplied by d .

Radix trade-off

Increasing b reduces the required number of digits d , but increases the counter array and its initialization cost.

These bounds assume constant-time digit extraction, direct indexing and record movement. Take $b \geq 2$ and $d \geq 1$.

The usual stable implementation is not in place. Better asymptotic time does not guarantee faster execution on every dataset.

A polynomial key universe can give linear sorting

Suppose $n \geq 2$ integers lie in $[0, n^3 - 1]$. Write each key in base n :

$$x = a_2 n^2 + a_1 n + a_0, \quad 0 \leq a_j < n.$$

There are three digits, and each digit uses only n possible values.

$$d = 3, \quad b = n, \quad T(n) = \Theta(3(n + n)) = \Theta(n).$$

Why this does not contradict the lower bound

Digit extraction and direct indexing are not pairwise comparisons. The linear result uses the word-RAM assumptions: keys and required operations fit the chosen word model.

A comparison rule does not automatically define digits

Radix sort needs an order-preserving positional representation and efficient, stable digit passes.

Example: ordering university applications

The rule may use eligibility, a score with exceptional cases, and a locale-dependent comparison of names. Unless this rule is decomposed into a finite sequence of ordered fields, no digit position has been defined.

A comparison sort can call the rule directly. Radix sort first needs a justified encoding.

Distinguish two situations: a representation needs to be transformed, or Radix sort is valid but offers no computational or practical advantage.

Character order may differ from numerical order

Consider the strings 2, 11, and 100.

Desired numerical order	$2 < 11 < 100$
Lexicographic string order	$100 < 11 < 2$

Sorting their characters directly would solve a different problem.

Repair for nonnegative integers

Pad on the left with leading zeroes to a common width:

$$002 < 011 < 100.$$

The lexicographic order now agrees with numerical order.

Variable-length strings need an end-marker convention

The simple LSD algorithm assumes exactly d positions for every key. For ordinary lexicographic order, pad shorter strings on the *right* with an end marker $\perp$ smaller than every alphabet symbol.

$$a \mapsto a\perp, \quad aa \mapsto aa, \quad b \mapsto b\perp.$$

Stable passes process positions from right to left. The number of passes is the maximum string length.

An alternative for strings

An MSD sort starts with the most significant character, then recursively sorts groups with equal prefixes. This may be a more natural approach.

Signed keys need an order-preserving transformation

An unsigned digit order applied to written signs and magnitudes does not automatically give numerical order.

Magnitude order is wrong for negative values

Ordering magnitudes would put -2 before -10 , although

$$-10 < -2.$$

Possible remedies include:

- translating a known bounded interval to nonnegative keys;
- separating negative and nonnegative values, with a carefully specified recombination rule.

The transformation must preserve the intended numerical order.

Floating-point representations need special care

Printed decimal strings may contain signs, decimal points and exponents, with different lengths.

Their lexicographic order cannot simply be assumed to match numerical order.

Raw bits are not a general solution

Machine bit patterns do not behave as ordinary unsigned integers across negative and positive values.

Specialized order-preserving transformations exist for fixed floating-point formats. The basic integer version of Radix sort requires such a transformation before it can be used.

A larger radix can make the counter array infeasible

Consider $n = 1,000$ unsigned 64-bit keys.

Digit choice	Radix b	Passes d
The complete key	2^{64}	1
One byte	256	8

One complete-key pass would require 2^{64} counters, an infeasible array.

Byte-sized digits give the much more reasonable bound

$$\Theta(8(n + 256)).$$

Choose the radix to balance the number of passes against counter-array size.

Many positions can make valid LSD sorting unattractive

Suppose strings have maximum length $d = 10^6$.

A cost hidden by calling the method linear

The simple LSD implementation may make one million full-array passes, even when most strings are short.

$$T(n, d, b) = \Theta(d(n + b)).$$

The algorithm can be correct while doing too much work.

A comparison sort, an MSD variant, or a trie-based method may examine much less irrelevant data.

Auxiliary storage may determine the method

A stable Counting-sort digit pass normally uses:

- an output array with n positions;
- a counter array with b positions.

$$S(n, b) = \Theta(n + b) \quad \text{auxiliary space.}$$

If additional storage is tightly constrained, in-place Heapsort or another comparison method may be preferable despite its $\Theta(n \log n)$ running time.

Constant-space Heapsort uses iterative heap repair.

Check the representation and the resource requirements

Before choosing Radix sort, check:

1. Does the positional representation preserve the intended order?
2. Is the number of positions d manageable?
3. Is the range b at each position manageable?
4. Is every digit-sorting pass stable and efficient?
5. Is the required auxiliary storage available?

Make any conversion explicit

For strings, signs, floating-point values or compound fields, state the conversion and prove that it preserves the intended order.

Choose a comparison method for its guarantees

Method	Worst-case time	Auxiliary data	Relevant condition
Insertion	$\Theta(n^2)$	$\Theta(1)$	small or nearly sorted; stable
Merge	$\Theta(n \log n)$	$\Theta(n)$ buffer	stability, predictable time
Quick	$\Theta(n^2)$	expected $\Theta(\log n)$ stack	array sorting
Heap	$\Theta(n \log n)$	constant*	memory, worst-case guarantee

*Heapsort uses constant auxiliary space with iterative repair. Recursive repair can use a logarithmic stack.

Quicksort takes expected $\Theta(n \log n)$ time under suitable pivot assumptions, but worst-case $\Theta(n^2)$ time. An in-place Merge sort would require a nonstandard implementation.

Choose Counting or Radix sort only with the required structure

Counting sort

Worst-case time $\Theta(n + k)$; auxiliary space $\Theta(n + k)$ for counters and output. Requires a bounded integer-key range. Linear in n when $k = O(n)$.

Radix sort

Worst-case time $\Theta(d(n + b))$; auxiliary space $\Theta(n + b)$ with stable Counting-sort passes. Requires order-preserving digits and a stable pass. Constant d and $b = O(n)$ give linear time.

There is no universally fastest method. Match the key representation, allowed operations, required stability, memory, and type of time guarantee.

Exercises on the scope of the lower bound

Exercise 1

Draw a comparison decision tree for three distinct keys. Label all six leaves with input orders and identify the longest root-to-leaf path.

Exercise 2

Prove $\log_2(n!) = \Theta(n \log n)$ without Stirling's approximation. State which half of the factorial product gives the lower bound.

Exercise 3

Explain precisely why the comparison lower bound does not forbid an $O(n)$ best case for Insertion sort or an $O(n + k)$ bound for Counting sort.

Exercises on Counting sort and the key range

Exercise 4

Trace stable Counting sort on $[2_a, 5, 3_a, 0_a, 2_b, 3_b, 0_b, 3_c]$. Show the counter array after frequency counting, after prefix sums, and after every output placement.

Exercise 5

The input contains one million records whose keys lie in $[10^9, 10^9 + 100]$. Describe the translation needed for Counting sort and compute the actual counter-array size.

Exercises on digit passes and radix choice

Exercise 6

Trace LSD Radix sort on $[329, 457, 657, 839, 436, 720, 355]$. Show the complete array after each decimal-digit pass and state the invariant represented by that array.

Exercise 7

For n integers in $[0, n^4 - 1]$, choose radix n and derive the running time. Repeat the analysis with radix $\sqrt{n}$ and compare the number of passes and counter-array sizes.

For Exercise 7, assume $n \geq 4$ is a perfect square so both radices are integers.

Exercise 8: test the Radix-sort assumptions

For each of the following datasets, decide whether the standard LSD Radix-sort algorithm applies directly, requires an order-preserving transformation, or should be replaced by another method. Justify every answer in terms of d , b , stability, and the intended order.

- (a) fixed-width eight-digit student identifiers containing only decimal digits;
- (b) the signed integers $[-12, 7, -3, 105, 0]$ written with an optional minus sign;
- (c) surnames of lengths between 1 and 40 under locale-dependent collation;
- (d) one thousand unsigned 64-bit integers using one complete integer as a digit;
- (e) dates stored as records with fixed-width year, month, and day fields.

For every case that requires a transformation, describe the transformation and explain why it preserves the required order.

Choose a method under an explicit contract

Exercise 9

Choose a sorting algorithm for each contract: stable arbitrary records, a memory-constrained array requiring a worst-case guarantee, ten million examination marks in $[0, 100]$, and fixed-width identifiers. For every choice, reject one alternative by naming the assumption or resource it violates.

Reading pointers

- CLRS, *Introduction to Algorithms*, 3rd edition, Chapter 8.
- Dasgupta, Papadimitriou, and Vazirani, Section 2.3 and the sorting lower-bound box.
- Horowitz, Sahni, and Rajasekaran, lower-bound and linear-time sorting material.

Chapter 7:

Graphs as computational models

Modelling relationships

Roads connect locations; hyperlinks connect web pages; friendships connect people; prerequisites connect courses.

Choose the model before its representation

State what the vertices and edges mean. A graph is the abstract object; an adjacency list or matrix is a way to store it.

Breadth-first search explores distance layers. **Depth-first search** explores a branch and then backtracks.

Both provide foundations for weighted graph algorithms in Module B.

Directed and undirected graphs

A graph is a pair $G = (V, E)$: vertices V and edges E . Write $n = |V|$ and $m = |E|$.

Directed graph

$E \subseteq V \times V$ consists of *ordered* pairs. An edge (u, v) goes from u to v ; it does not imply (v, u) .

Simple undirected graph

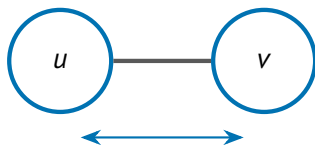
$E \subseteq \{\{u, v\} : u, v \in V, u \neq v\}$ consists of *unordered* pairs of distinct vertices. Thus $\{u, v\} = \{v, u\}$; loops and parallel edges are excluded.

The directed definition permits (v, v) unless loops are explicitly excluded. A set of pairs cannot represent parallel copies of the same edge.

Multigraphs, labels and weights need an explicitly declared model.

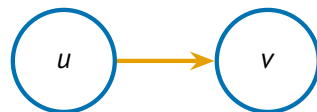
Direction is part of the model

Undirected edge $\{u, v\}$



the relation is symmetric

Directed edge (u, v)



only the direction $u \rightarrow v$ is present

In a road model, a one-way street gives a directed edge. A two-way street may give an undirected edge. (u, v) says nothing about whether (v, u) is present.

Counting edge incidences

The degree $\deg(v)$ counts incident edges in a simple undirected graph.

Handshake lemma

For every finite undirected graph $G = (V, E)$,

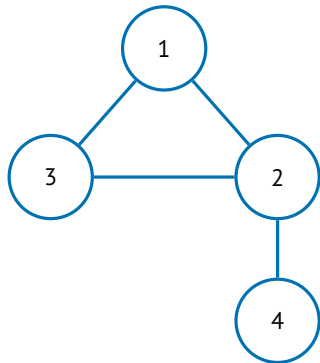
$$\sum_{v \in V} \deg(v) = 2m.$$

Consequently, the number of odd-degree vertices is even.

If loops are admitted, each loop contributes *two* to the degree of its endpoint.

Handshake lemma in a graph

The numbers inside the circles are vertex labels. The graph has $n = 4$ vertices and $m = 4$ edges.

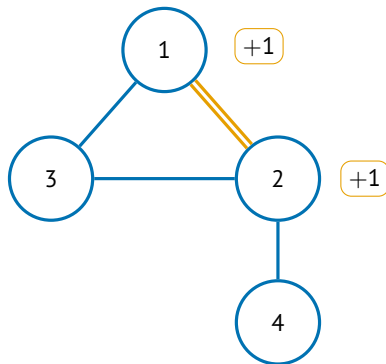


Vertex	Neighbours	Degree
1	2, 3	2
2	1, 3, 4	3
3	1, 2	2
4	2	1

Each degree counts the edges touching that vertex.

$$\sum_{v \in V} \deg(v) = 2 + 3 + 2 + 1 = 8 = 2 \cdot 4 = 2m.$$

Every edge contributes twice



Follow the double-lined edge $\{1, 2\}$

It contributes 1 to $\deg(1)$ and 1 to $\deg(2)$.

It is *one edge*, but it contributes *two* to the degree sum.

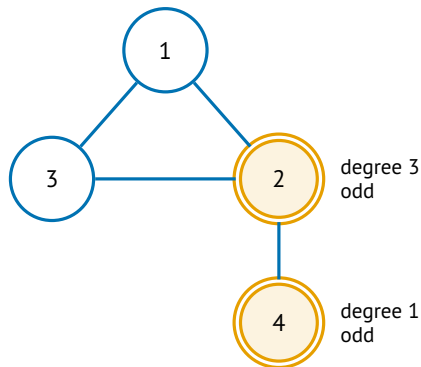
The same reasoning applies to each of the four edges, not just the highlighted one.

General argument

For any finite undirected graph, count edge ends by vertices to obtain $\sum_v \deg(v)$, or by edges to obtain $2m$. Both count the same incidences.

A loop, if admitted, has both ends at the same vertex and contributes 2 there.

Why odd degrees occur in pairs



Double circles mark odd-degree vertices.

In this graph

Even-degree vertices 1, 3: $2 + 2 = 4$.

Odd-degree vertices 2, 4: $3 + 1 = 4$.

There are **two** odd-degree vertices.

In any undirected graph

The handshake lemma makes the total degree even. Removing the even degrees leaves an even sum:

$$\sum_{\deg(v) \text{ odd}} \deg(v) = 2m - \sum_{\deg(v) \text{ even}} \deg(v).$$

A sum of odd integers is even only when it has an even number of terms.

Degree and graph density

For directed graphs, $\deg^-(v)$ counts incoming edges and $\deg^+(v)$ outgoing edges:

$$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = m.$$

Each edge contributes one head and one tail.

How many edges are present?

A simple undirected graph has at most $\binom{n}{2}$ edges.

- A sparse graph has relatively few edges, often with m of order n .
- A dense graph has a substantial fraction of the possible edges, with m of order n^2 .

Density helps us choose a storage representation.

Selecting vertices and edges

Definition (Subgraph and induced subgraph)

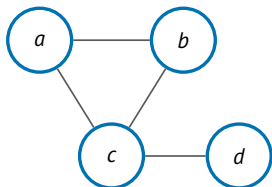
A graph $H = (V_H, E_H)$ is a subgraph of G when $V_H \subseteq V$ and its edges are edges of G with endpoints in V_H .

The subgraph induced by $S \subseteq V$ contains every edge of G whose endpoints both belong to S .

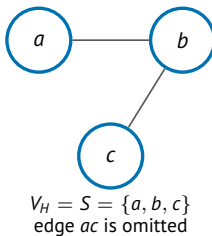
A general subgraph may omit an available edge. Once S is chosen, the induced subgraph $G[S]$ may not omit any internal edge.

Keeping every internal edge

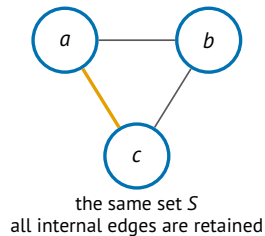
Original graph G



A subgraph H



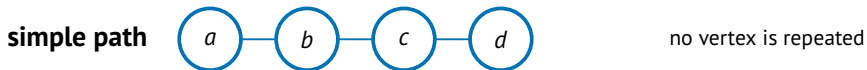
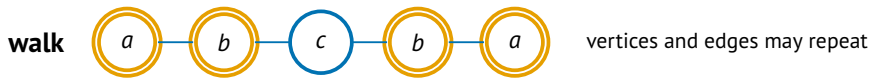
The induced subgraph $G[S]$



Both selections use $S = \{a, b, c\}$. The induced subgraph must keep ac as well as ab and bc .

What may be repeated?

A **walk** follows an edge at every step. A **trail** repeats no edge. A **simple path** repeats no vertex.

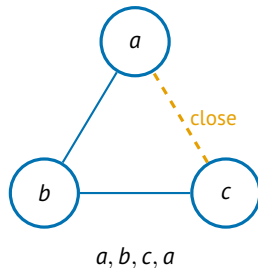


Every simple path is a trail; every trail is a walk. The converses are false.

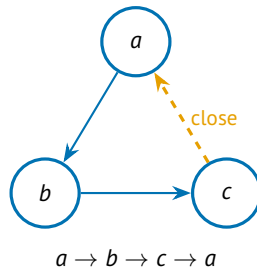
Closing a path into a cycle

A cycle starts and ends at the same vertex and repeats no other vertex.

Undirected cycle



Directed cycle



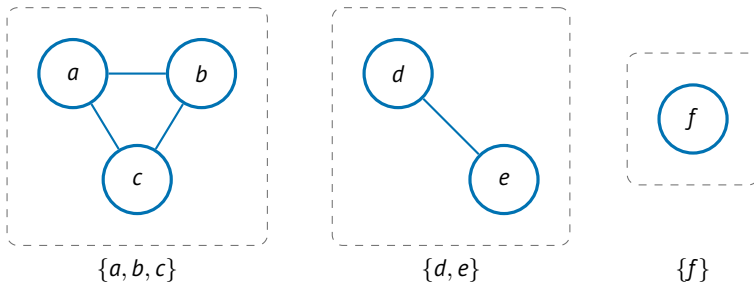
The dashed closing edge completes the path a, b, c . In the directed graph, every arrow must follow the traversal.

In a simple undirected graph, a cycle has at least three edges and reuses no edge: u, v, u is not a cycle. An admitted directed self-loop is a cycle of length one.

Matrix powers count *walks*, whereas many graph problems ask for *paths*.

Connected components

An undirected graph is connected if every pair of vertices is joined by a path. A connected component is a *maximal* connected subgraph.



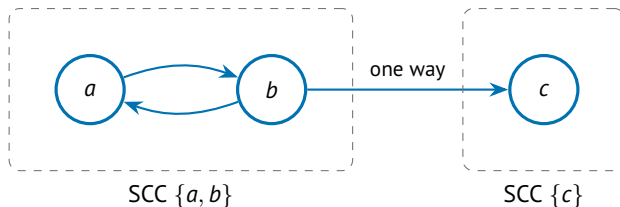
The dashed boundaries show three components. There is no path between different components; an isolated vertex forms a component by itself.

Why maximal?

$\{a, b\}$ is connected, but it is not a whole component: c can be included while preserving connectedness.

Directed reachability

Reachability follows the arrows. Being able to go from u to v does not imply that we can return.



From a we can reach b and c . From c we cannot reach a or b .

Strongly connected components (SCCs)

An SCC is a maximal set of vertices that can all reach one another. Here the SCCs are $\{a, b\}$ and $\{c\}$. A vertex reaches itself by a path of length zero.

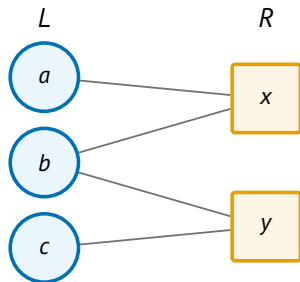
The algorithm for finding SCCs is deferred to later study.

Separating vertices into two parts

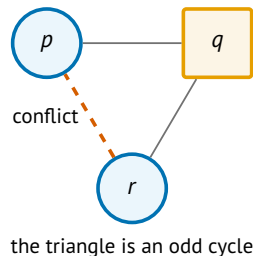
A graph is **bipartite** if $V = L \dot{\cup} R$ and every edge joins one vertex in L to one in R . No edge stays inside one part.

Examples: students and their courses; jobs and machines that can execute them.

Bipartite



Not bipartite



Why a bipartite graph has no odd cycle

Proposition (Bipartite characterization)

An undirected graph is bipartite if and only if it has no odd cycle.

Forward implication. Along a cycle, successive vertices alternate between L and R .

$$L \longrightarrow R \longrightarrow L \longrightarrow R \longrightarrow \dots$$

Returning to the starting part requires an even number of edges.

We will prove the converse after learning BFS: distance parity will construct the two parts.

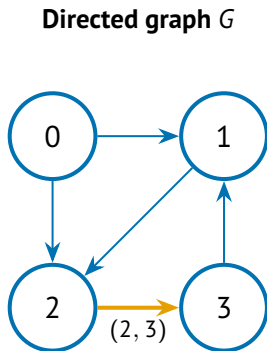
One graph, several representations

An algorithm needs to test edges and enumerate neighbours. A drawing alone does not specify how those operations are implemented.

- An **adjacency matrix** stores a position for every ordered pair.
- **Adjacency lists** store the outgoing neighbours of each vertex.
- An **edge list** stores the edge records directly.
- An **incidence matrix** has one row per vertex and one column per edge.

The graph stays the same; the operation costs change.

Testing edges in a matrix



Adjacency matrix

	target v			
	0	1	2	3
0	0	1	1	0
1	0	0	1	0
2	0	0	0	1
3	0	1	0	0

$M[2, 3] = 1$

The thick orange arrow $(2, 3)$ corresponds to the outlined entry $M[2, 3] = 1$.

The matrix uses $\Theta(n^2)$ space and tests an edge in $\Theta(1)$ time.

Enumerating neighbours from lists

Adjacency lists

0 :	1	2
1 :	2	
2 :	3	
3 :	1	

$3 \in \text{Adj}[2]$

Edge list

(0, 1)
(0, 2)
(1, 2)
(2, 3)
(3, 1)

The same edge appears as $3 \in \text{Adj}[2]$ and as the pair (2, 3).

Adjacency lists use $\Theta(n + m)$ space. In an undirected graph, each edge normally appears in two lists.

Edge lists and incidence matrices

An edge list stores m edge records. Including the vertex records gives $\Theta(n + m)$ space.

Incidence matrix

There is a row for each vertex and a column for each edge: an $n \times m$ array. It is useful algebraically, but usually larger than adjacency lists for sparse graphs.

The comparison table that follows concerns the adjacency matrix, adjacency lists and edge list. The exercises also ask you to examine incidence matrices.

Space and edge queries

Let $\deg^+(u)$ be the number of outgoing neighbours of u .

Operation	Matrix	Adjacency lists	Edge list
Space	$\Theta(n^2)$	$\Theta(n + m)$	$\Theta(n + m)$
Test (u, v)	$\Theta(1)$	$O(\deg^+(u))$	$O(m)$

The list bounds count inspected records. Including constant setup, write $O(1 + \deg^+(u))$ and $O(1 + m)$ for empty lists as well.

A matrix provides direct access; an unsorted list may need a scan.

The cost of scanning a graph

Operation	Matrix	Adjacency lists	Edge list
Enumerate neighbours of u	$\Theta(n)$	$\Theta(\deg^+(u))$	$\Theta(m)$
Scan all edges	$\Theta(n^2)$	$\Theta(n + m)$	$\Theta(m)$

These scan bounds count inspected entries; include an additive 1 for an empty scan.

Representation affects traversal time

Scanning every vertex and edge takes $\Theta(n + m)$ with adjacency lists. Scanning every matrix row takes $\Theta(n^2)$, even for a sparse graph.

Shortest paths measured in edges

Definition (Unweighted graph distance)

The distance $\delta(s, v)$ is the minimum number of edges on any path from s to v , and is ∞ when no such path exists.

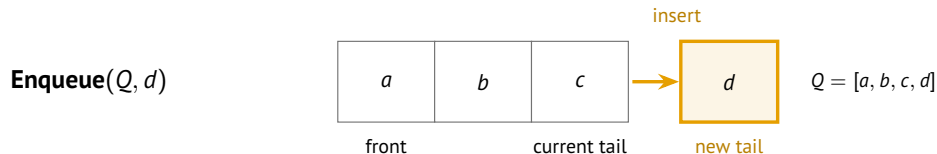
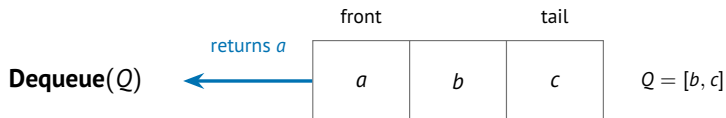
BFS answers both reachability and shortest-path questions in this unweighted sense.

$s \longrightarrow \text{distance 1} \longrightarrow \text{distance 2} \longrightarrow \dots$

Here distance counts edges; it does not sum edge weights.

First in, first out

Enqueue(Q, x) inserts at the tail. Dequeue(Q) removes and returns the front.



The rows above are *independent*: each starts from $[a, b, c]$. If we instead enqueue d and then dequeue, we return a and leave $[b, c, d]$.

Exploring one distance layer at a time

BFS keeps discovered but not yet processed vertices in a FIFO queue.

1. Remove the vertex u at the front.
2. Scan all neighbours of u .
3. For each newly discovered neighbour v , set its distance to $\text{distance}[u] + 1$ and insert it at the tail.

Why a queue?

Vertices discovered earlier are processed earlier. A later distance layer cannot overtake an earlier one.

Initializing a breadth-first search

```
BFS(G, s)
  for each vertex v in V
    colour[v] <- White
    distance[v] <- infinity
    parent[v] <- Nil
  colour[s] <- Grey
  distance[s] <- 0
  Q <- empty queue
  Enqueue(Q, s)
```

All vertices start undiscovered, with no parent and infinite distance. The source s has distance zero and enters the queue first.

Discovering each vertex once

```
while Q is not empty
  u <- Dequeue(Q)
  for each v in Adj[u]
    if colour[v] = White
      colour[v] <- Grey
      distance[v] <- distance[u] + 1
      parent[v] <- u
      Enqueue(Q, v)
  colour[u] <- Black
```

Continuation of BFS(G, s). Assign the distance and parent only at first discovery.

Mark a vertex when it enters the queue

White Undiscovered.

Grey Discovered: waiting in the queue or currently being processed.

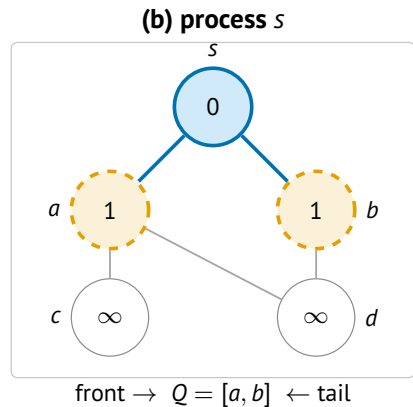
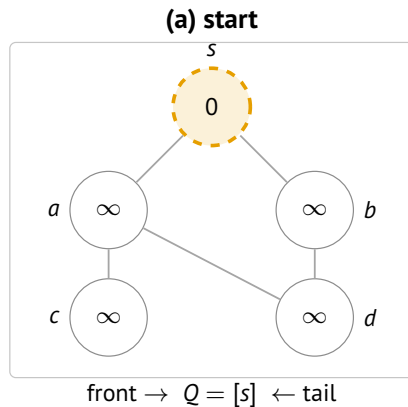
Black Complete: its adjacency list has been scanned.

Mark on enqueue

If marking waited until dequeue, two neighbours could insert the same apparently undiscovered vertex.

In the figures: dashed orange means queued, solid blue means processed, and a thin grey outline means undiscovered. Each node contains its distance.

Discovering the first layer

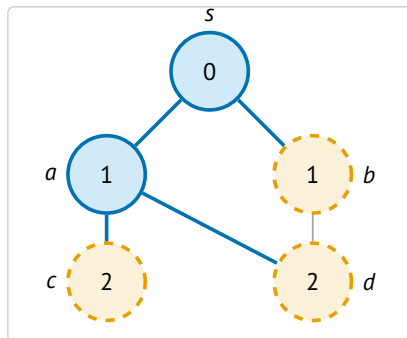


Processing s discovers a and b , both at distance 1, with parent s .

Thick blue edges are parent edges; dashed grey edges have been scanned without discovery.

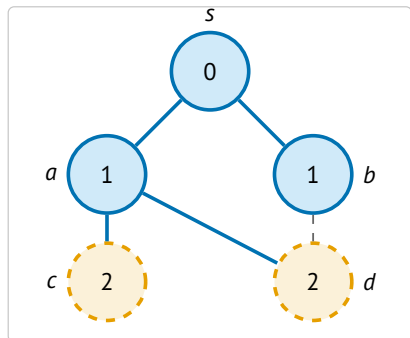
Discovering c and d

(c) process a



front $\rightarrow Q = [b, c, d] \leftarrow$ tail

(d) process b



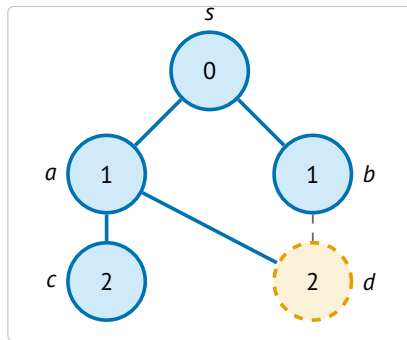
front $\rightarrow Q = [c, d] \leftarrow$ tail

Processing a discovers c, d at distance 2. When b scans d , it is already queued: no second insertion and no parent change.

Thick blue edges are parent edges; dashed grey edges have been scanned without discovery.

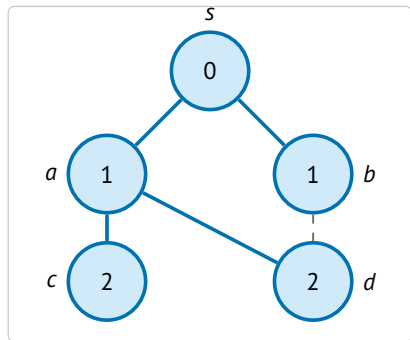
Finishing without duplicate insertions

(e) process c



front $\rightarrow Q = [d] \leftarrow$ tail

(f) process d



$Q = []$

Processing c and then d empties the queue. Parent edges are $s-a$, $s-b$, $a-c$, $a-d$.

Thick blue edges are parent edges; dashed grey edges have been scanned without discovery.

BFS finds shortest paths

Theorem (BFS shortest-path property)

For every vertex v reachable from s , BFS assigns

$$\text{distance}[v] = \delta(s, v),$$

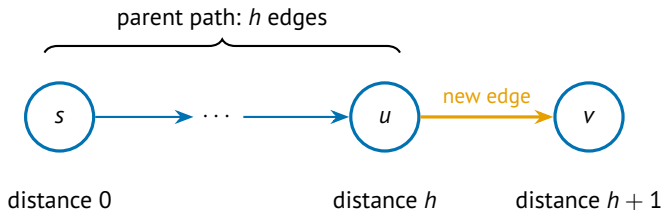
and its parent links form a shortest path.

Unreachable vertices retain distance ∞ and parent Nil.

To prove equality, show that the assigned distance is both the length of a real path and no larger than a shortest path.

Parent paths give the first inequality

Proof. At first discovery from u , BFS sets $\text{parent}[v] = u$ and $\text{distance}[v] = \text{distance}[u] + 1$.

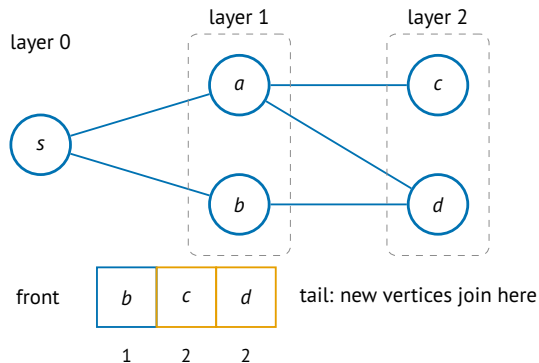


The source has distance 0. Inductively, if the parent path to u has h edges, appending (u, v) gives a path of $h + 1$ edges to v .

$$\delta(s, v) \leq \text{distance}[v].$$

The shortest path cannot be longer than this particular parent path.

FIFO keeps distance layers in order



In the running example, after processing a , the queue is $[b, c, d]$ with distances $[1, 2, 2]$.

Vertex b is processed before the vertices in layer 2. New discoveries are appended at the tail.

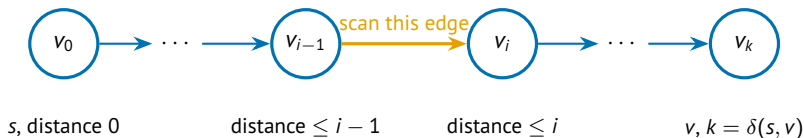
Queue invariant

Assigned distances in the queue are nondecreasing. While processing layer i , new vertices receive distance $i + 1$ and join after all remaining layer- i vertices.

Initially the queue contains only s at distance 0; each enqueue preserves this order.

Shortest paths give the reverse inequality

Proof. Fix any shortest path $s = v_0, v_1, \dots, v_k = v$. Its length is $k = \delta(s, v)$; it need not be the BFS parent path.



- Base: $v_0 = s$ is discovered at distance 0.
- Inductive step: if $\text{distance}[v_{i-1}] \leq i-1$, scanning the next edge discovers a white v_i at distance at most i .
- If v_i is already discovered, FIFO ensures its distance is no larger than this candidate.

$$\text{distance}[v] \leq k = \delta(s, v).$$

Combined with the parent-path inequality, this proves $\text{distance}[v] = \delta(s, v)$.

Accounting for initialization and scanned edges

Assume adjacency lists and constant-time queue operations. Let m_R count adjacency entries in lists of vertices reachable from s .

- Initializing all arrays costs $\Theta(n)$.
- Each reachable vertex is enqueued and dequeued once.
- Each reachable adjacency entry is scanned once.

$$T = \Theta(n + m_R) = O(n + m), \quad \text{auxiliary space } \Theta(n).$$

For a full traversal, or when all vertices are reachable, the time is $\Theta(n + m)$.

In an undirected graph, each reachable edge contributes two entries.

Qualification of the afternotes bound: a single-source run need not scan unreachable components.

Searching every component

For an undirected graph, restart the search from each still-white vertex.

1. Initialize all vertices once.
2. Choose a white vertex as a new root with distance 0.
3. Run the queue loop until that component is complete.
4. Continue until no white vertex remains.

Keep existing colours when starting another root: do not reset earlier components.

Parent edges form one rooted tree per component, collectively a **forest**. Distances are measured from the root of each component.

The full traversal costs $\Theta(n + m)$ with adjacency lists.

Assigning sides by distance parity

Run BFS in every undirected connected component and assign

$$L = \{v : \text{distance}[v] \text{ is even}\}, \quad R = \{v : \text{distance}[v] \text{ is odd}\}.$$

Each component uses its own BFS root.

- If every edge joins opposite sides, L, R form a bipartition.
- If an edge joins equal sides, it witnesses an odd cycle with the parent paths.

This gives the algorithm behind the converse of the bipartite characterization.

A same-side edge gives an odd cycle

Converse proof. Suppose an edge $\{u, v\}$ joins vertices of equal distance parity in a BFS tree. Let w be their lowest common ancestor: the last shared vertex on their root paths.

The tree paths from w to u and v share no further vertex. Add the edge $\{u, v\}$ to close a cycle of length

$$(\text{distance}[u] - \text{distance}[w]) + (\text{distance}[v] - \text{distance}[w]) + 1.$$

The two endpoint distances have the same parity, so this length is odd.

If the graph has no odd cycle, such an edge cannot exist. Every edge joins opposite parity classes, which therefore form a bipartition.

Following a branch and backtracking

DFS follows an unexplored branch as far as possible. It then returns to the most recent unfinished vertex and tries another edge.

The recursive calls act as a stack

The most recently entered unfinished call resumes first: **last in, first out** (LIFO), rather than the FIFO discipline of BFS.

A call for u waits while its child explores a branch. Only after all outgoing edges have been examined does u finish.

An outer loop is needed when one starting vertex cannot reach the entire graph.

Starting a new DFS tree

```
DFS(G)
  for each vertex u in V
    colour[u] <- White
    parent[u] <- Nil
  time <- 0
  for each vertex u in V
    if colour[u] = White
      DFS-visit(G, u)
```

Each still-white vertex starts a new tree. The clock time is shared by all calls, including calls in different trees.

Discovery, recursion and finishing

```
DFS-visit(G, u)
  time <- time + 1
  discover[u] <- time
  colour[u] <- Grey
  for each v in Adj[u]
    if colour[v] = White
      parent[v] <- u
      DFS-visit(G, v)
  colour[u] <- Black
  time <- time + 1
  finish[u] <- time
```

The same shared clock records both events. Assign a parent only when discovering a white vertex.

Every adjacency list is scanned once

Every vertex changes from white to grey to black exactly once.

- The outer loop includes vertices in every component or unreachable region.
- Each call scans the outgoing adjacency list of its vertex once.

$$T(n, m) = \Theta(n + m) \quad \text{with adjacency lists.}$$

The recursive stack can contain $\Theta(n)$ vertices. Colours, parents and timestamps also require $\Theta(n)$ auxiliary space.

This bound excludes storage of the input graph itself.

Nested and disjoint activity intervals

The interval

$$[\text{discover}[u], \text{finish}[u]]$$

is the time during which the call for u is active.

Theorem (DFS parenthesis property)

For any two vertices u and v , their DFS activity intervals are disjoint or one contains the other. The interval of u strictly contains that of v exactly when v is a proper descendant of u in the DFS forest.

$$\text{discover}[u] < \text{discover}[v] < \text{finish}[v] < \text{finish}[u].$$

What the recursive call stack guarantees

Proof. Suppose u is discovered before v .

- If the call for u has finished before v is discovered, the intervals are disjoint.
- Otherwise u is still active. Recursion discovers v within its unfinished branch. The call for v must finish before the call for u can finish.

If v was discovered first, exchange the roles of u and v .

These cases cover completed, active and not-yet-discovered vertices. Recursive discovery creates exactly the parent-child edges, so strict containment is equivalent to proper descent.

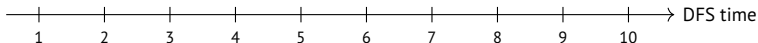
Reading discovery and finishing times

circle = discovery square = finish



b contains *c*
first recursive branch

d contains *e*
second recursive branch



The interval of *a* contains both branches. The intervals of *b* and *d* are disjoint: one branch finishes before the other begins.

Classifying a directed edge

When DFS scans (u, v) , inspect the state of v .

White: tree edge. It discovers a new vertex and sets its parent to u .

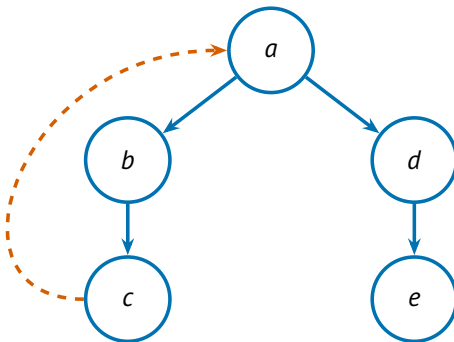
Grey: back edge. It reaches an active ancestor (or u itself for a self-loop).

Black: forward or cross edge. It reaches a completed vertex.

A forward edge reaches a proper descendant discovered through other tree edges. A cross edge joins disjoint activity intervals, possibly in different DFS trees.

Use the timestamps to distinguish these last two cases. An edge to a black vertex is not a back edge.

A back edge closes a cycle



Solid blue edges discover new vertices.

The dashed edge $c \rightarrow a$ reaches an active ancestor, so it is a back edge.

Together with the tree path $a \rightsquigarrow c$, it forms a directed cycle.

The tree path $a \rightarrow b \rightarrow c$ and the back edge $c \rightarrow a$ form a directed cycle. This figure illustrates tree and back edges.

Detecting directed cycles

Proposition (Back edges and directed cycles)

A directed graph contains a cycle exactly when DFS finds an edge to a grey ancestor.

Here the current vertex counts as an ancestor of itself if self-loops are admitted.

One direction. A back edge (u, v) closes a cycle with the DFS-tree path from v to u . If $u = v$, the edge itself is a cycle.

The test refers to *directed* graphs. In an undirected graph, the reverse occurrence of the parent edge needs separate treatment.

Why a cycle produces a back edge

Converse proof. Suppose a directed cycle exists. Let v be its first vertex discovered by DFS.

1. Every other cycle vertex is still white when v is discovered.
2. Before v can finish, DFS explores all still-white vertices reachable along the cycle, so they become descendants of v .
3. In particular, the predecessor of v on the cycle scans its edge to v while v is still active.

That edge reaches a grey ancestor and is a back edge. A self-loop gives the same conclusion immediately.

Representations and traversal traces

1. Compare representations

Compare adjacency matrices, adjacency lists, incidence matrices and edge lists for edge testing, neighbour enumeration, complete traversal and memory use.

2. Trace both searches

Trace BFS and DFS on the same disconnected graph. Give parent arrays, BFS distances, discovery and finishing times, and edge classifications.

State the vertex order, adjacency-list order and directed/undirected convention used in your trace.

Bipartiteness and a later extension

3. Find an odd-cycle witness

Extend BFS to test bipartiteness and reconstruct an odd cycle when an edge joins two vertices on the same side.

4. Deferred until topological sorting is introduced

Implement topological sorting by DFS finishing times and by Kahn's algorithm. Test a DAG with multiple valid orders and a directed graph containing a cycle.

Exercise 4 is retained from the afternotes but is not assigned here: its prerequisite section is currently inactive.

Thank you!



UNIVERSITY
OF TRIESTE