

UNIT 11

Algorithms

(5 Hours) — Data Structures and Algorithms (CACS201)

Earlier units studied individual data structures and the specific operations that act on them. This unit steps back and looks at **algorithms as a subject in their own right** — how they are classified, how they are designed, and what can be done when a problem is too hard to solve exactly. We examine the distinction between deterministic and non-deterministic computation, the divide-and-conquer design strategy, the difference between series and parallel execution, and the heuristic and approximation methods used on intractable problems.

11.1 Introduction

An **algorithm** is a finite, well-defined sequence of unambiguous instructions that transforms a given input into the required output in a finite amount of time. The word derives from the name of the 9th-century Persian mathematician *Al-Khwarizmi*.

11.1.1 Characteristics of an Algorithm

Property	Requirement
Input	Zero or more quantities are supplied externally.
Output	At least one quantity is produced.
Definiteness	Every instruction is clear and unambiguous — it has exactly one meaning.
Finiteness	The algorithm terminates after a finite number of steps for every valid input.
Effectiveness	Every operation is basic enough to be carried out exactly, in principle by a person with paper and pencil.
Correctness	For every valid input the algorithm produces the correct output.
Generality	It solves the whole class of problems, not just one particular instance.

11.1.2 Why Algorithms Are Classified

The same problem can usually be solved by many different algorithms, and those algorithms may differ enormously in speed, memory use, ease of implementation and even in whether they guarantee the right answer. Classifying algorithms gives us a vocabulary for discussing these trade-offs and for choosing sensibly.

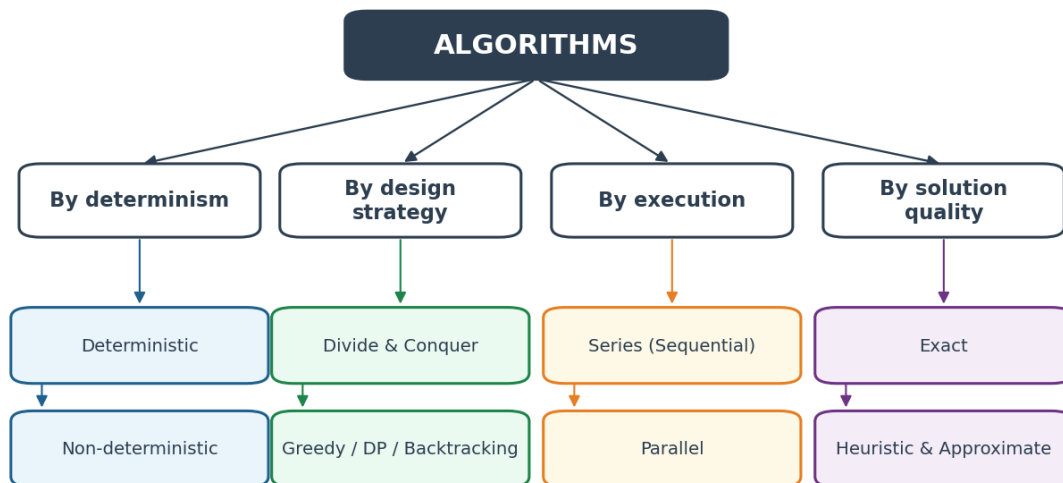


Figure 11.1 — Common ways of classifying algorithms

Note: These classifications are independent of one another. A single algorithm can be simultaneously deterministic, divide-and-conquer, parallel and exact — merge sort running on a multi-core processor is exactly that.

11.2 Deterministic and Non-deterministic Algorithms

11.2.1 Deterministic Algorithms

An algorithm is **deterministic** if the result of every operation is uniquely defined, so that at each step there is exactly one possible next state. Given a particular input, the algorithm always follows the same sequence of steps and always produces the same output, no matter how many times it is run or on which machine.

Almost every algorithm actually implemented on a real computer is deterministic. Binary search, merge sort, Dijkstra's algorithm and matrix multiplication are all deterministic: their behaviour is completely predictable from the input.

11.2.2 Non-deterministic Algorithms

An algorithm is **non-deterministic** if the outcome of at least one operation is *not* uniquely defined — at some step the algorithm may choose from several permitted alternatives. It is a theoretical device: we imagine a machine that, whenever a choice arises, always makes the correct one, or equivalently that explores all the alternatives simultaneously.

Non-deterministic algorithms are written using three conceptual operations introduced by Horowitz and Sahni:

Operation	Meaning
choice(S)	Arbitrarily select one element from the set S. The selection is assumed to be the one that leads to success, if any such element exists.
failure()	Signals an unsuccessful completion of this computation path.
success()	Signals a successful completion; the algorithm terminates.

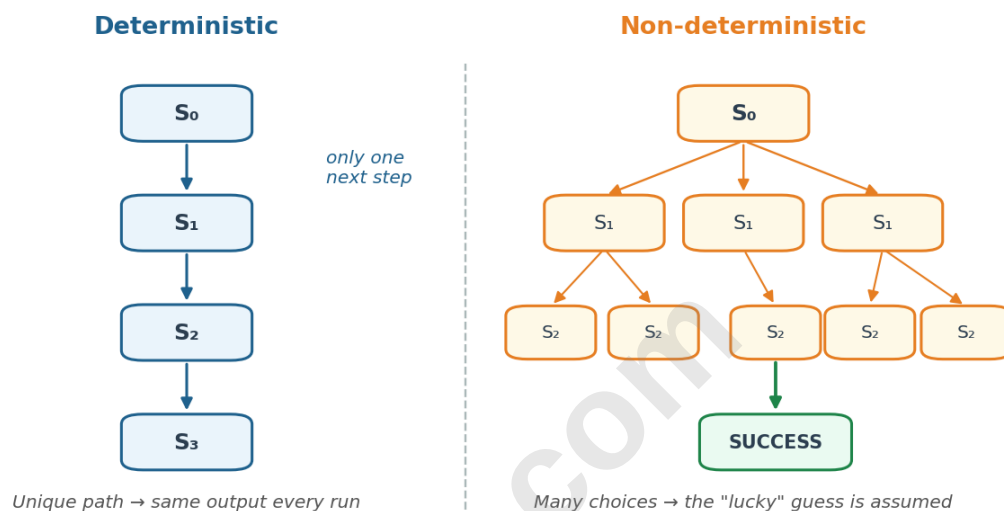


Figure 11.2 — A deterministic computation follows one path; a non-deterministic one branches

Example 11.1 — Non-deterministic search

Write a non-deterministic algorithm that finds the position of a key x in an unsorted array $A[0 \dots n-1]$.

Solution

NDSEARCH(A, n, x)

$j = \text{choice}(0, 1, 2, \dots, n-1)$ // guess the index

if $A[j] = x$ then print j ; success()

print 0; failure()

This runs in $O(1)$ non-deterministic time, because the single choice operation is counted as one step. The equivalent deterministic algorithm — a linear scan — needs $O(n)$ time.

11.2.3 Comparison

Basis	Deterministic	Non-deterministic
Next step	Uniquely determined at every stage	May be chosen from several alternatives
Output	Same output for the same input, every run	Output may differ between runs

Basis	Deterministic	Non-deterministic
Execution path	A single path	A tree of possible paths
Implementable?	Yes — this is how real computers work	No — it is a theoretical model only
Time taken	Usually higher	Lower, because the correct guess is assumed free
Model of computation	Deterministic Turing Machine (DTM)	Non-deterministic Turing Machine (NDTM)
Example	Binary search, quick sort, Dijkstra	Non-deterministic sorting, SAT, knapsack

11.2.4 The Classes P and NP

Non-determinism matters because it defines the most important complexity classes in computer science:

- **P** (Polynomial time) — the set of decision problems that can be solved by a *deterministic* algorithm in polynomial time, $O(n^k)$ for some constant k . These are regarded as the tractable, efficiently solvable problems.
- **NP** (Non-deterministic Polynomial time) — the set of decision problems that can be solved by a *non-deterministic* algorithm in polynomial time. Equivalently, a proposed solution can be *verified* in polynomial time by a deterministic algorithm.
- **NP-hard** — problems at least as hard as every problem in NP. Every NP problem reduces to them in polynomial time. They need not themselves lie in NP.
- **NP-complete** — problems that are in NP *and* NP-hard; the hardest problems inside NP. Examples: the satisfiability problem (SAT), the travelling salesperson decision problem, 0/1 knapsack, graph colouring and the clique problem.

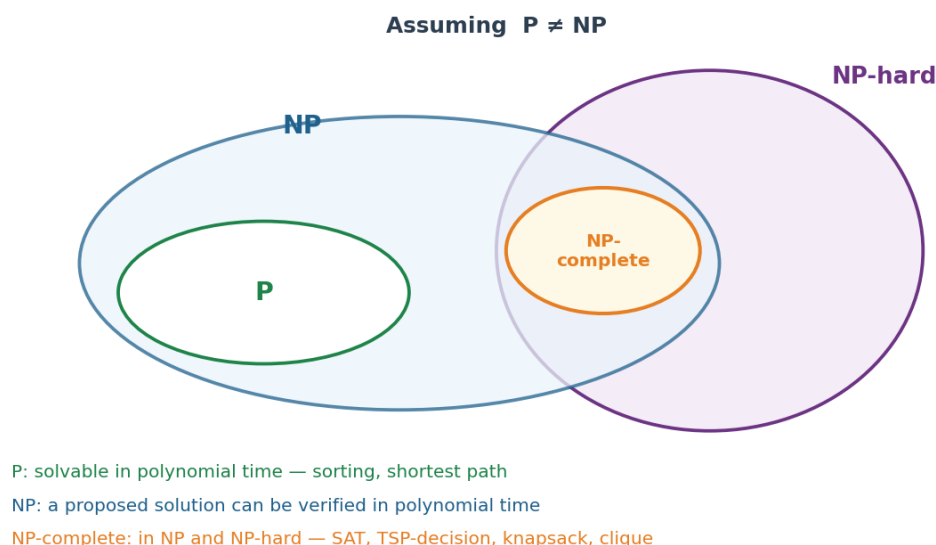


Figure 11.3 — Relationship between P, NP, NP-complete and NP-hard

Clearly $P \subseteq NP$, since any deterministic algorithm is a special case of a non-deterministic one. Whether $P = NP$ — that is, whether every problem whose solution can be checked quickly can also be solved quickly — remains the most famous unsolved problem in computer science. It is widely believed that $P \neq NP$, which is precisely why heuristic and approximation algorithms (Section 11.5) are so important in practice.

Note: If any one NP-complete problem were shown to be solvable in deterministic polynomial time, then $P = NP$ would follow and every problem in NP would become tractable at once.

11.3 Divide and Conquer Algorithms

Divide and conquer is a general algorithm-design strategy that solves a problem by breaking it into smaller instances *of the same problem*, solving those recursively, and assembling their answers into a solution for the original. It is the natural strategy whenever a problem is self-similar at smaller scales.

11.3.1 The Three Steps

Step	What happens
1. Divide	Split the problem of size n into a (usually two) smaller subproblems, each of size roughly n/b .
2. Conquer	Solve each subproblem recursively. When a subproblem becomes small enough — the base case — solve it directly without further recursion.
3. Combine	Merge the solutions of the subproblems into a solution for the original problem.

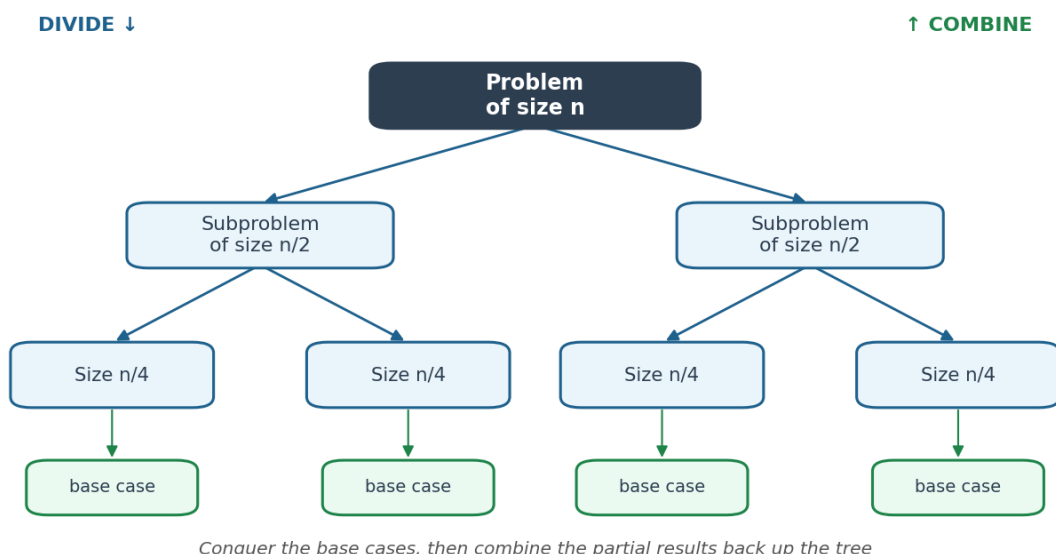


Figure 11.4 — The divide-and-conquer strategy applied recursively

11.3.2 General Form and Recurrence

Algorithm: DANDC(P)

```

if SMALL(P) then
    return S(P)                                // base case - solve directly
else
    divide P into subproblems P1, P2, ..., Pk
    return COMBINE( DANDC(P1), DANDC(P2), ..., DANDC(Pk) )

```

The running time of a divide-and-conquer algorithm satisfies a **recurrence relation**. If the problem of size n is split into a subproblems of size n/b , and the dividing and combining work costs $f(n)$, then:

$$T(n) = a \cdot T(n/b) + f(n)$$

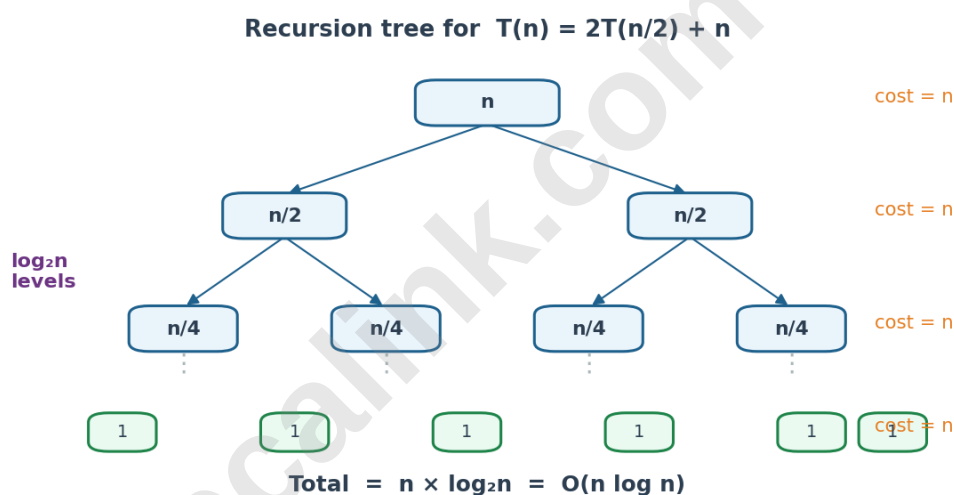


Figure 11.5 — Recursion tree for $T(n) = 2T(n/2) + n$, giving $O(n \log n)$

11.3.3 The Master Theorem

The **Master Theorem** gives the solution of $T(n) = a \cdot T(n/b) + O(n^d)$ directly, where $a \geq 1$, $b > 1$ and $d \geq 0$:

Case	Condition	Solution
1	$a < b^d$ (combining dominates)	$T(n) = \Theta(n^d)$
2	$a = b^d$ (work is even at every level)	$T(n) = \Theta(n^d \log n)$
3	$a > b^d$ (the leaves dominate)	$T(n) = \Theta(n^{\log_b a})$

Example 11.2

Solve the recurrence of merge sort, $T(n) = 2T(n/2) + n$, using the Master Theorem.

Solution

Here $a = 2$, $b = 2$ and $f(n) = n = n^1$, so $d = 1$.

Compute $b^d = 2^1 = 2$, and $a = 2$. Therefore $a = b^d$, which is Case 2.

Hence $T(n) = \Theta(n^1 \log n) = \Theta(n \log n)$, confirming the result obtained from the recursion tree in Figure 11.5.

Example 11.3

Solve $T(n) = 4T(n/2) + n$ using the Master Theorem.

Solution

$a = 4$, $b = 2$, $d = 1$, so $b^d = 2^1 = 2$ and $a = 4 > 2 \rightarrow$ Case 3.

$T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$.

This is the recurrence of the naive divide-and-conquer matrix multiplication; Strassen's method reduces a from 8 to 7 for the $n \times n$ case and thereby lowers the exponent.

11.3.4 Applications of Divide and Conquer

Algorithm	Divide	Combine	Complexity
Binary Search	Halve the sorted array at the midpoint	Nothing — only one half is searched	$O(\log n)$
Merge Sort	Split the array into two equal halves	Merge the two sorted halves	$O(n \log n)$
Quick Sort	Partition about a pivot	Nothing — sublists are already in place	$O(n \log n)$ avg
Strassen's Matrix Mult.	Split each matrix into four $n/2 \times n/2$ blocks	Add and subtract the 7 products	$O(n^{2.81})$
Max-Min Problem	Split the list into two halves	Compare the two maxima and the two minima	$O(n)$
Closest Pair of Points	Split the plane by a vertical median line	Check the strip near the dividing line	$O(n \log n)$
Karatsuba Multiplication	Split each integer into high and low halves	Combine 3 products with shifts and adds	$O(n^{1.585})$

Example 11.4 — Binary search as divide and conquer

Search for the key 23 in the sorted array 4, 8, 15, 16, 23, 42, 55.

Solution

low = 0, high = 6 → mid = 3, A[3] = 16. Since $23 > 16$, **discard the whole left half** and set low = 4.

low = 4, high = 6 → mid = 5, A[5] = 42. Since $23 < 42$, discard the right half and set high = 4.

low = 4, high = 4 → mid = 4, A[4] = 23 → **found at index 4**.

Only 3 comparisons were needed instead of 5 for a linear scan. The recurrence is $T(n) = T(n/2) + 1$, giving $O(\log n)$.

11.3.5 Advantages and Limitations

- **Efficiency.** It frequently reduces a $\Theta(n^2)$ brute-force method to $\Theta(n \log n)$, which is the single biggest practical win in algorithm design.
- **Natural parallelism.** The subproblems are independent, so they can be solved simultaneously on different processors — see Section 11.4.
- **Cache efficiency.** Subproblems eventually become small enough to fit entirely in cache, which speeds up real execution considerably.
- **Limitation — recursion overhead.** Each recursive call costs a stack frame. For small n this overhead can exceed the savings, which is why practical implementations switch to insertion sort for small subarrays.
- **Limitation — extra memory.** Some combine steps, such as merge sort's, need $O(n)$ auxiliary space.
- **Limitation — repeated subproblems.** If the subproblems overlap, plain divide and conquer recomputes them many times. The naive recursive Fibonacci algorithm is $O(2^n)$ for exactly this reason; **dynamic programming**, which stores subproblem results, is the correct tool there.

11.4 Series and Parallel Algorithms

11.4.1 Series (Sequential) Algorithms

A **series** or **sequential** algorithm executes its instructions one after another, in a strict order, on a single processor. At any instant exactly one operation is in progress, and each step may depend on the results of all the steps before it. This is the classical model of computation — the **RAM (Random Access Machine)** model — and every algorithm studied so far in this course has been of this kind.

11.4.2 Parallel Algorithms

A **parallel** algorithm divides a problem into parts that can be executed *simultaneously* on multiple processors, with the partial results combined afterwards. Parallelism is possible only where the subtasks are **independent** — that is, where no subtask needs a result that another is still computing. Such a dependency is called a **data dependency**, and it is the fundamental limit on how much of an algorithm can be parallelised.

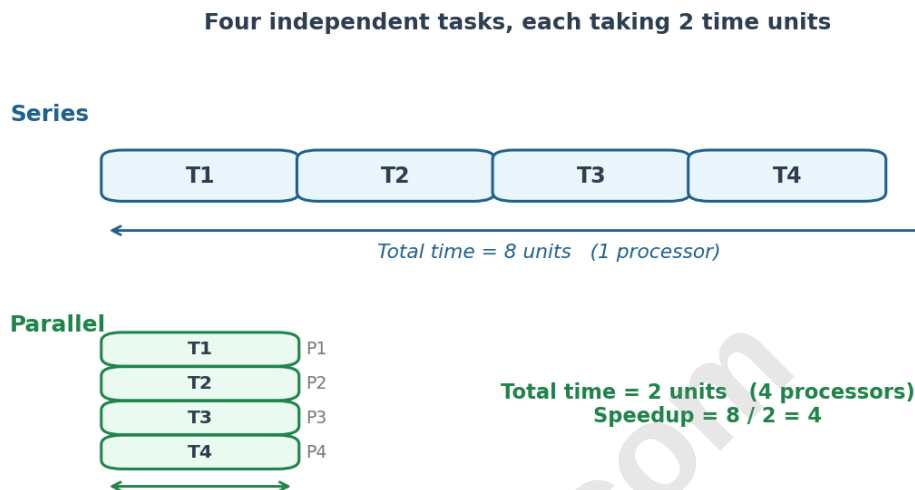


Figure 11.6 — The same four tasks executed in series and in parallel

11.4.3 Performance Measures

Measure	Definition	Formula
Speedup	How many times faster the parallel version is than the best sequential one	$S(N) = T_1 / T_n$
Efficiency	How well the available processors are being used	$E(N) = S(N) / N$
Cost	Total processor-time consumed	$C(N) = N \times T_n$
Scalability	Whether efficiency is maintained as N grows	—

Here T_1 is the running time on one processor and T_n the running time on N processors. In the ideal case $S(N) = N$ and $E(N) = 1$, but this is almost never achieved because of communication cost, synchronisation, load imbalance and the part of the algorithm that simply cannot be parallelised.

11.4.4 Amdahl's Law

Amdahl's Law quantifies the ceiling imposed by the sequential part of a program. If a fraction p of the work can be parallelised and the remaining $(1 - p)$ must run serially, then the speedup on N processors is:

$$S(N) = 1 / [(1 - p) + p/N]$$

As $N \rightarrow \infty$ this approaches $1 / (1 - p)$. The serial fraction therefore sets a hard upper bound on the speedup, regardless of how many processors are added.

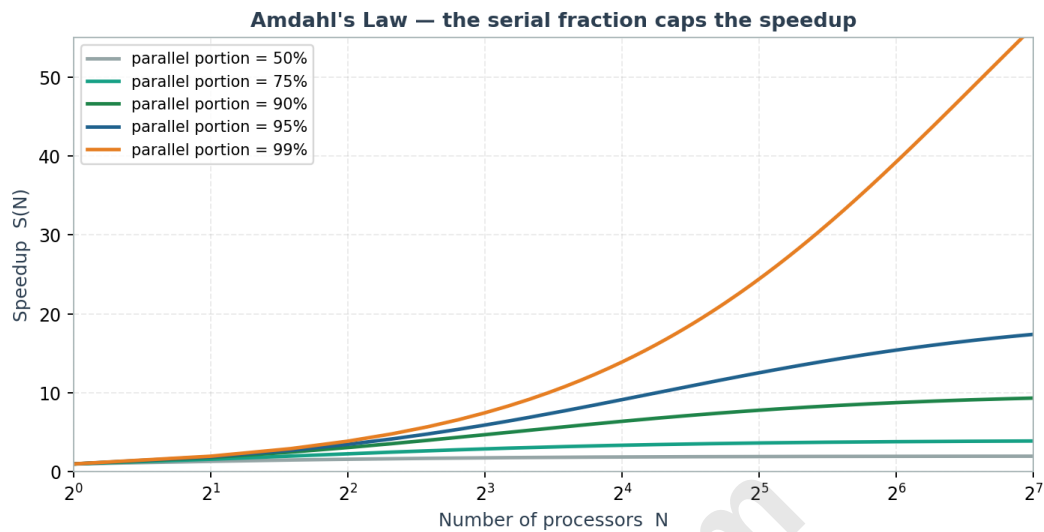


Figure 11.7 — Speedup saturates as the number of processors grows

Example 11.5

A program spends 80% of its time in a parallelisable loop and 20% in unavoidably serial code. Find the speedup on 4 processors, on 16 processors, and the theoretical maximum.

Solution

Here $p = 0.8$ and $1 - p = 0.2$.

N = 4: $S = 1 / (0.2 + 0.8/4) = 1 / 0.4 = 2.5\times$ (efficiency = $2.5/4 = 62.5\%$)

N = 16: $S = 1 / (0.2 + 0.8/16) = 1 / 0.25 = 4.0\times$ (efficiency = $4/16 = 25\%$)

N $\rightarrow \infty$: $S = 1 / 0.2 = 5.0\times$

Quadrupling the processors from 4 to 16 gains only 60% more speed, and no number of processors can ever exceed 5 $\times$.

11.4.5 Comparison

Basis	Series (Sequential)	Parallel
Execution	One instruction at a time	Many instructions at the same time
Processors	Single processing unit	Multiple processors or cores
Design	Simpler — straightforward control flow	Harder — needs decomposition and synchronisation
Debugging	Deterministic and repeatable	Difficult — race conditions, deadlock, non-repeatable bugs

Basis	Series (Sequential)	Parallel
Speed	Limited by the single processor's clock	Potentially much faster, bounded by Amdahl's Law
Cost	Low hardware cost	Higher hardware and communication cost
Examples	Linear search, insertion sort, sequential file read	Parallel merge sort, parallel matrix multiplication, MapReduce

Note: Divide-and-conquer algorithms parallelise particularly well: the two recursive calls of merge sort are completely independent of one another and can be assigned to two different processors, with only the final merge step needing to be sequential.

11.5 Heuristic and Approximation Algorithms

Many problems of real importance — routing vehicles, scheduling machines, packing containers, colouring maps — are NP-hard. For these, no known algorithm finds the exact optimum in polynomial time, and an exhaustive search of all possibilities is hopeless: the travelling salesperson problem on just 20 cities has more than 10^{17} possible tours. When an exact answer is out of reach we trade **optimality** for **time**.

11.5.1 Heuristic Algorithms

A **heuristic** is a practical rule of thumb that finds a good solution quickly, without any guarantee that the solution is optimal or even that it is close to optimal. Heuristics are typically derived from experience or from insight into the structure of the problem. The word comes from the Greek *heuriskein*, "to find".

- **Fast** — usually polynomial, often linear.
- **No guarantee** — the quality of the answer is not bounded in advance, and may be poor on adversarial inputs.
- **Can be trapped** — a local search heuristic may settle at a **local optimum** that is much worse than the global optimum.

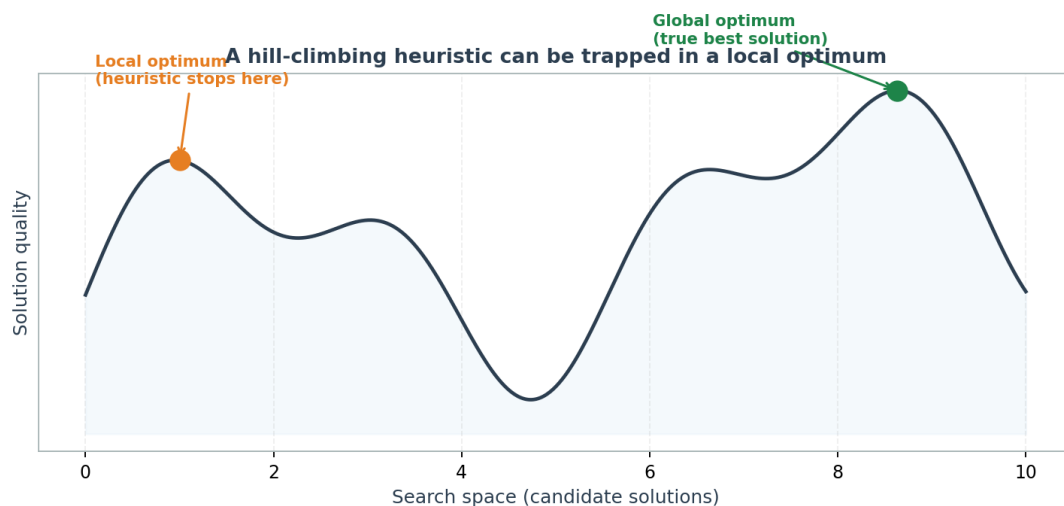


Figure 11.8 — A local search heuristic may stop at a local optimum

Well-known heuristic techniques include:

Technique	Idea
Greedy heuristic	At each step take the choice that looks best right now, and never reconsider it.
Hill climbing	Repeatedly move to a neighbouring solution that is better; stop when no neighbour improves.
Simulated annealing	Like hill climbing, but occasionally accept a worse move — with a probability that falls over time — so as to escape local optima.
Genetic algorithms	Maintain a population of solutions and evolve it by selection, crossover and mutation.
Tabu search	Hill climbing that keeps a short-term memory of recently visited solutions and forbids returning to them.
Ant colony optimisation	Simulated ants deposit pheromone on good paths, biasing later ants towards them.

11.5.2 Approximation Algorithms

An **approximation algorithm** is a heuristic with a *proof attached*. It runs in polynomial time and comes with a mathematical guarantee on how far its answer can be from the true optimum. This guarantee is expressed as the **approximation ratio** ρ :

$$\rho = C / C^* \text{ for a minimisation problem, } \rho = C^* / C \text{ for a maximisation problem}$$

where C is the cost of the solution produced and C^* is the cost of the optimal solution. An algorithm with $\rho \leq 2$ is called a **2-approximation algorithm**: its answer is never worse than twice the optimum. The closer ρ is to 1, the better the algorithm.

Example 11.6 — Nearest-neighbour heuristic for TSP

Apply the nearest-neighbour heuristic to a 6-city travelling salesperson problem and compare the tour it produces with the optimal tour.

Solution

Heuristic rule: start at any city; repeatedly travel to the nearest city not yet visited; finally return to the start.

The rule is greedy, runs in $O(n^2)$ time, and is trivially easy to implement.

Starting at A the rule visits F, E, D, C, B and returns to A, giving a tour of cost ≈ 2.91 against the optimum of ≈ 2.75 — about **6% worse**.

The tour is acceptable, but the heuristic gives no promise: on a worst-case instance the nearest-neighbour tour can be $\Theta(\log n)$ times the optimum.

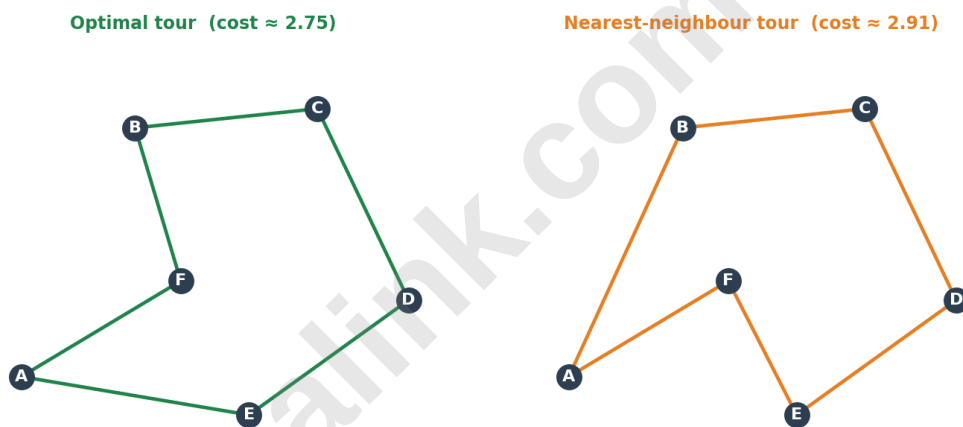


Figure 11.9 — Optimal TSP tour compared with the nearest-neighbour tour

For the **metric** TSP — where the distances satisfy the triangle inequality — much better is possible. The **minimum spanning tree (MST) heuristic** builds an MST, walks it twice, and short-cuts repeated vertices; this is a proved **2-approximation**. **Christofides' algorithm** improves the guarantee to **1.5**.

Problem	Approximation algorithm	Ratio ρ
Metric TSP	MST doubling with short-cutting	2
Metric TSP	Christofides' algorithm	1.5
Vertex Cover	Repeatedly take both endpoints of an uncovered edge	2
Bin Packing	First-Fit Decreasing	$11/9$ asymptotically
Set Cover	Greedy — repeatedly take the set covering the most new elements	$\ln n$

Problem	Approximation algorithm	Ratio ρ
0/1 Knapsack	Fully polynomial-time approximation scheme (FPTAS)	$1 + \epsilon$

*Note: A **PTAS** (polynomial-time approximation scheme) lets the user choose any $\epsilon > 0$ and returns a solution within a factor $(1 + \epsilon)$ of the optimum, at the cost of a running time that grows as ϵ shrinks. An **FPTAS** additionally guarantees that the running time is polynomial in $1/\epsilon$.*

11.5.3 Heuristic versus Approximation

Basis	Heuristic Algorithm	Approximation Algorithm
Guarantee on quality	None — quality is not bounded	A proved bound, the ratio ρ
Guarantee on time	Usually fast, but not always proved	Provably polynomial time
Basis	Experience, intuition, rule of thumb	Mathematical analysis and proof
Worst case	May be arbitrarily bad	Never worse than $\rho \times$ optimum
Typical use	Very large or poorly understood problems	NP-hard problems with known structure
Examples	Hill climbing, genetic algorithms, simulated annealing, tabu search	Christofides' TSP, 2-approx vertex cover, FFD bin packing

11.5.4 When to Use Them

- The problem is **NP-hard** and the input is large enough that exact methods would take impractically long.
- A **good** answer delivered now is worth more than a **perfect** answer delivered too late — as in real-time route planning or live scheduling.
- The input data is itself **approximate or noisy**, so an exact optimum of the model would not be an exact optimum of reality anyway.
- The solution will be **refined later** by a human expert or by a local improvement step.

11.6 Summary of the Four Classifications

Classification	Type A	Type B	Distinguishing question
Determinism	Deterministic	Non-deterministic	Is the next step uniquely defined?
Design strategy	Divide and conquer	Greedy, DP, backtracking, ...	Is the problem split into smaller copies of itself?

Classification	Type A	Type B	Distinguishing question
Execution model	Series (sequential)	Parallel	Can independent parts run at the same time?
Solution quality	Exact	Heuristic / approximate	Is the optimum guaranteed?

Key Takeaways

- ✓ An **algorithm** is a finite, definite, effective sequence of steps that turns input into output. Its essential properties are input, output, definiteness, finiteness, effectiveness, correctness and generality.
- ✓ A **deterministic** algorithm has exactly one possible next step at every stage and gives the same output on every run. A **non-deterministic** algorithm may choose among several alternatives and is assumed always to guess correctly — it is a theoretical model, not something a real machine executes.
- ✓ **P** is the class solvable in polynomial time deterministically; **NP** is the class solvable in polynomial time non-deterministically, equivalently verifiable in polynomial time. **NP-complete** problems lie in NP and are NP-hard. Whether $P = NP$ is still unresolved.
- ✓ **Divide and conquer** works in three steps — divide, conquer, combine — and yields the recurrence $T(n) = a \cdot T(n/b) + f(n)$, solvable by the recursion-tree method or the **Master Theorem**.
- ✓ Binary search, merge sort, quick sort, Strassen's and Karatsuba multiplication and the closest-pair problem are all divide-and-conquer; where subproblems overlap, use dynamic programming instead.
- ✓ A **series** algorithm runs one instruction at a time on one processor; a **parallel** algorithm runs independent subtasks simultaneously on several processors. Parallelism requires the absence of data dependencies.
- ✓ **Speedup** $S(N) = T_1/T_n$ and **efficiency** $E(N) = S(N)/N$ measure parallel performance. **Amdahl's Law**, $S(N) = 1/[(1-p) + p/N]$, shows that the serial fraction caps the speedup at $1/(1-p)$ however many processors are used.
- ✓ A **heuristic** finds a good solution fast with no guarantee of quality; it can be trapped in a local optimum. Examples: hill climbing, simulated annealing, genetic algorithms, tabu search.
- ✓ An **approximation algorithm** runs in polynomial time and carries a proved **approximation ratio** $\rho = C/C^*$. Christofides' algorithm is a 1.5-approximation for metric TSP; the greedy vertex-cover algorithm is a 2-approximation.
- ✓ Heuristic and approximation methods are the practical response to NP-hardness: when the exact optimum is unreachable, a provably good answer obtained quickly is the right engineering choice.

— End of Unit 11 —