Algorithms & Data Structures II

Sebastian F. Taylor

November 27, 2025

Dynamic Programming

Weighted Interval Scheduling

The idea: Sort by finish time and select
`OPT(j) = max(v[j]) + OPT(p[j]), OPT(j-1))`. Either you take job j, and skip conflicts, or don't take it. The algorithm runs in $O(n \log n)$ time, where n is the number of jobs.

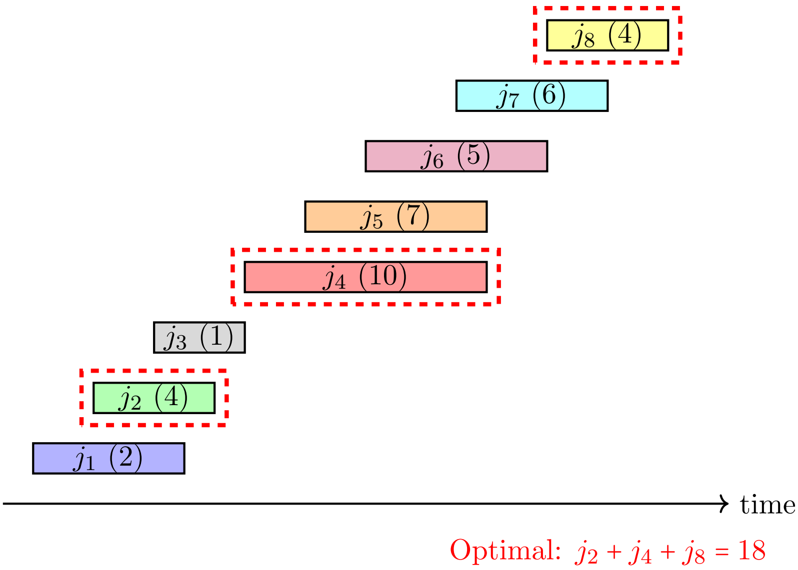


Figure 1: Weighted Interval Scheduling Example: Jobs are sorted by finish time. The optimal solution selects non-overlapping jobs with maximum total value.

Knapsack

The problem: Fill a backpack with as many books as possible.
`OPT(i,w) = max(OPT(i-1,w), v[i] + OPT(i-1, w-w[i]))` The option is to take an item i or not and the algorithm runs in $O(nW)$ time, where n is the number of items and W is knapsack capacity. The algorithm creates a 2D table consisting of $n \times W$

Sequence Alignment

The problem: Match characters, gap in X, or gap in Y. The running time of this algorithm is

Max Flow & Min Cut

Flow Algorithms

Name	Running Time	Explanation	Notes
Ford-Fulkerson	$O(\|f^*\|m)$	$\|f^*\| =$ max flow value $m = \#$ edges	Pseudopolynomial; can fail to terminate
Scaling	$O(m^2 \log C)$	$C =$ max edge capacity	Weakly polynomial; assumes integer capacities
Edmonds-Karp	$O(m^2 n)$	$n = \#$ vertices $m = \#$ edges	Uses BFS; strongly polynomial

Running the algorithms

When running the algorithms by hand, just use a basic approach—it's manageable when the graph is reasonably simple. For min-cut, find the saturated edges (at capacity) along the residual graph boundary between reachable and unreachable vertices from the source.

Flow Problem Recipe

Build the Graph

- Vertices (source + sink)
- Edges (capacities + direction)

Describe the Algorithm

- Find max flow (m) and INSERT ALGORITHM
- Return an answer

Analyse Runtime

- Build Parameters from question
- Remember the time it takes to build the graph

Randomised Algorithms

Expected Value

Expected value is the average outcome over many trials. For a discrete random variable X:

$$\mathbb{E}[X] = \sum_i x_i \cdot \mathbb{P}[X = x_i]$$

Common trick: Use indicator random variables. If $X = X_1 + X_2 + \cdots + X_n$ where each X_i is an indicator (0 or 1), then:

$$\mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} \mathbb{P}[X_i = 1]$$

Geometric distribution: If success probability is p, then expected number of trials until first success is $\frac{1}{p}$.

Example - QuickSort: Let X_{ij} be an indicator that equals 1 if elements i and j are compared, 0 otherwise. Total comparisons:

$$X = \sum_{i<j} X_{ij} \implies \mathbb{E}[X] = \sum_{i<j} \mathbb{E}[X_{ij}] = \sum_{i<j} \mathbb{P}[i \text{ and } j \text{ compared}]$$

Elements i and j are compared if one is chosen as pivot before any element between them, so $\mathbb{P}[i \text{ and } j \text{ compared}] = \frac{2}{j-i+1}$. Summing gives $\mathbb{E}[X] = \mathrm{O}(n \log n)$.

Useful Sums

$$\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}, \quad |x| < 1$$
$$\sum_{n=0}^{\infty} n \cdot x^n = \frac{x}{(1-x)^2}, \quad |x| < 1$$
$$\sum_{k=1}^{n} \frac{1}{k} = H(n), \quad \ln(n) < H(n) < \ln(n) + 1$$
$$\mathbb{P}[\text{first success in round } j] = (1-p)^{j-1} p$$
$$\mathbb{E}[X] = \sum x_i \cdot \mathbb{P}[X = x_i] \quad \text{where } x_i \in \text{possible outcomes}$$
$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$$
$$\text{if } \mathbb{P}[\text{success}] = p \text{ then } \mathbb{E}[\# \text{ tries to success}] = \frac{1}{p}$$

Hashing

Chained Hashing

Store elements in array of linked lists. Each element x goes to list at $A[h(x)]$. Operations take $\mathrm{O}(|A[h(x)]| + 1)$ time. With universal hashing, expected $\mathrm{O}(1)$ time per operation.

Linear Probing

Store elements directly in array. If $A[h(x)]$ is occupied, place x in next empty slot (wrapping around). Forms clusters of consecutive elements. Operations take $\mathrm{O}(C(h(x)) + 1)$ time where $C(i)$ is cluster size. Cache-efficient but DELETE is $\mathrm{O}(n^2)$.

Hash Functions

Universal hashing: Family H is universal if for any $x \neq y$, $\Pr[h(x) = h(y)] \leq \frac{1}{m}$ for random $h \in H$.

Dot product hashing: For $x = (x_1, x_2, \dots, x_r)$ in base m (prime), define

$$h_a(x) = (a_1 x_1 + a_2 x_2 + \dots + a_r x_r) \bmod m$$

where $a = (a_1, \dots, a_r)$ chosen randomly. This family is universal.

Performance Comparison

Data Structure	SEARCH	INSERT	DELETE	Space
Linked List	$O(n)$	$O(n)$	$O(n)$	$O(n)$
BBST	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Direct Addressing	$O(1)$	$O(1)$	$O(1)$	$O(\lvert U \rvert)$
Chained Hashing (universal)	$O(1)^*$	$O(1)^*$	$O(1)^*$	$O(n)$
Linear Probing	$O(C(h(x)) + 1)$	$O(C(h(x)) + 1)$	$O(n^2)$	$O(n)$

* Expected time (when using universal hash function)

Amortised Analysis

Aggregate Method

The idea: Analyze the total cost $T(m)$ of a worst-case sequence of m operations, then compute the amortized cost as $T(m)/m$ per operation.

Accounting Method

Assign a *cost* $\hat{c}_i$ to each operation. When $\hat{c}_i > c_i$, store the difference as credits. When $\hat{c}_i < c_i$, use stored credits to pay. Ensure $\sum_{i=1}^{m} \hat{c}_i \geq \sum_{i=1}^{m} c_i$ always holds.

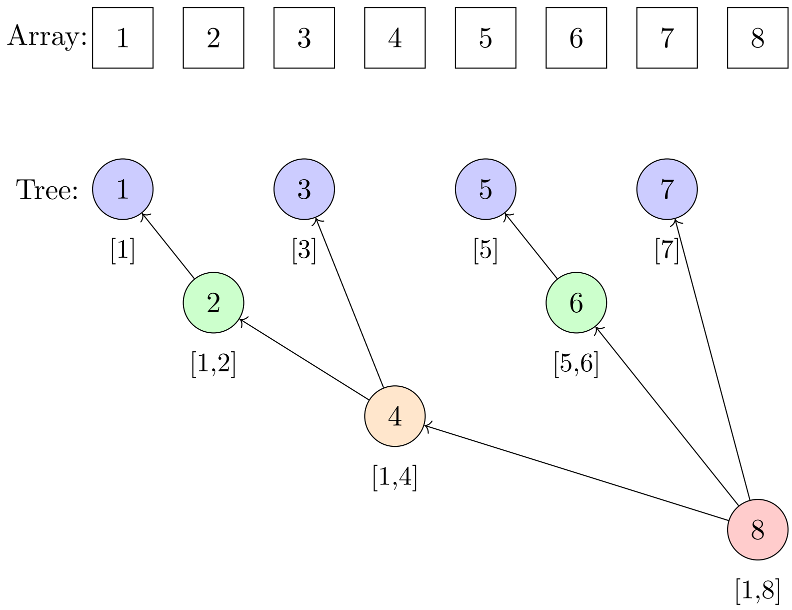
Potential Method

Define a potential function $\Phi(D)$ that maps the data structure state to a real value. Set amortized cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$. Require $\Phi(D_i) \geq 0$ for all i and $\Phi(D_0) = 0$.

Partial Sums & Dynamic Arrays

Partial Sums (Fenwick Trees)

A fenwick tree is effectively binary tree consisting of only the right child. This allows for an in-place data structure to compute the partial sum.



Example

here is an example on how to calculate a fenwick tree.

```
Index:  1   2   3   4   5   6   7   8
Value:  3   2   5   1   4   6   2   3

Index:  1   2   3   4   5   6   7   8
BIT:    3   2   5   1   4   6   2   3

Index:  1   2   3   4   5   6   7   8
BIT:    3   2   5   1   4   6   2   3

Index:  1   2   3   4   5   6   7   8
BIT:    3   5   5   1   4   6   2   3

Index:  1   2   3   4   5   6   7   8
BIT:    3   5   5   1   4   6   2   3

Index:  1   2   3   4   5   6   7   8
BIT:    3   5   5  11   4   6   2   3

Index:  1   2   3   4   5   6   7   8
BIT:    3   5   5  11   4   6   2   3

Index:  1   2   3   4   5   6   7   8
BIT:    3   5   5  11   4  10   2   3

Index:  1   2   3   4   5   6   7   8
BIT:    3   5   5  11   4  10   2   3

Index:  1   2   3   4   5   6   7   8
BIT:    3   5   5  11   4  10   2  26
```

```
Final Fenwick Tree:
Index:  1   2   3   4   5   6   7   8
BIT:    3   5   5  11   4  10   2  26
Range: [1][1,2][3][1,4][5][5,6][7][1,8]
```

Dynamic Arrays

Rotated Array

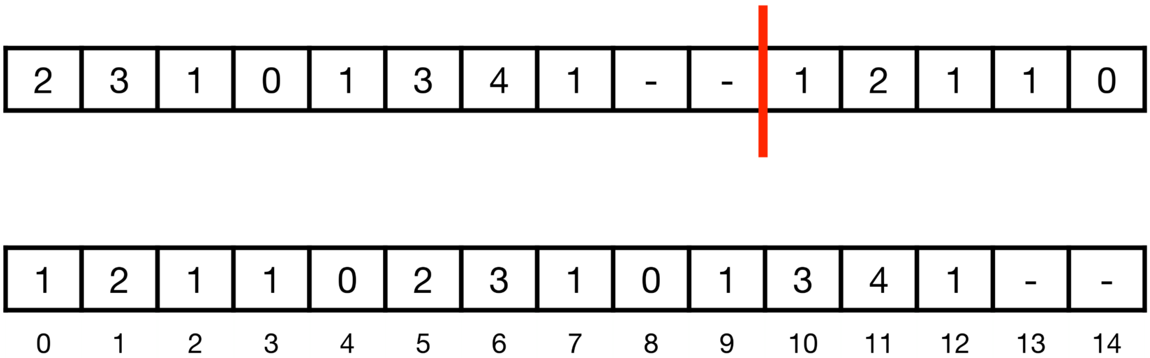


Figure 2: 1 level rotated array

Key idea: Store offset to mark the start of the array.

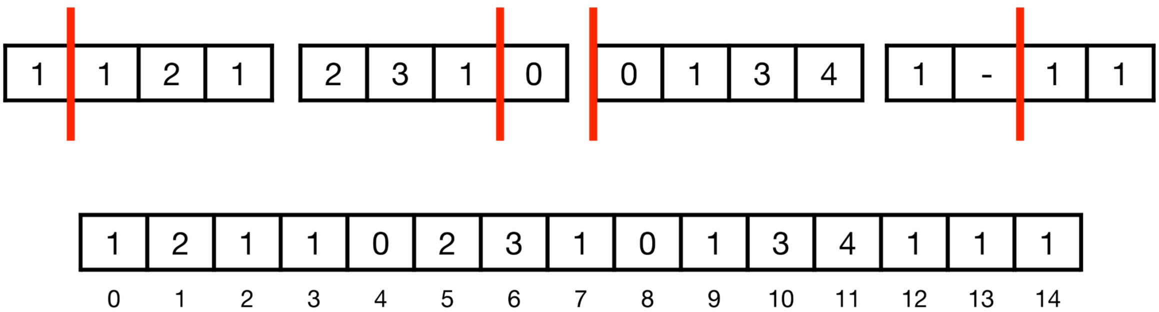


Figure 3: 2 level rotated array

Key idea: Split into $\sqrt{n}$ rotated buckets of size $\sqrt{n}$. When inserting/deleting, rebuild one bucket in $O(\sqrt{n})$ time, then propagate overflow/underflow.

Data Structure	Access	Insert	Delete	space
1 level rotated array	O(1)	O($\sqrt{n}$)	O($\sqrt{n}$)	O(n)
2 level rotated array	O(1)	O(n^{ε})	O(n^{ε}	O(n)

In a k-level rotated array, $\varepsilon = \frac{1}{k}$ determines that you partition n elements into $\sqrt[k]{n}$ buckets of size $\sqrt[k]{n}$ each, giving $O(n^{1/k}) = O(n^{\varepsilon})$ time for Insert/Delete operations.

NP Completeness

Boils down to if a problem is solvable in polynomial time or non-deterministic polynomial time. The mindset I have for this is best describes via exercise 2 from week *NP Completeness*.

Example - Customer Uniqueness (KT 8.2)

A store trying to analyse the behavior of its customers will often main a two-dimensional array A, where the rows correspond to its customers and the columns correspond to the products it sells. The entry $A[i, j]$ specifies the quantity of project j that has been purchased by customer i.

Here's a tiny example of such an array A

	liquid detergent	beer	diapers	cat litter
Raj	0	6	0	3
Alanis	2	0	0	7
Chelsea	0	0	0	7

One thing that a store might want to do with this data is the following. Let us say that a subset S off the customers is *diverse* if no two of the customers in S have ever bought the same product (i.e., for each product, at most one of the customers in s has ever bought it). A diverse set of customers can be useful, for example, as a target pool for market research.

We can now define the Diverse Subset Problem as follows: Given an $m \times n$ array A as defined above, and a number $k \leq m$, is there a subset of at least k of customers that is *diverse*?

Show that Diverse subset in NP-complete.

Solution

We show that $IS \leq_p DS$, where IS is Independent Set and DS is Diverse Subset.

Given a graph $G = (V, E)$ with $|V| = m$ vertices and $|E| = n$ edges, construct an $m \times n$ matrix A where vertices are customers (rows) and edges are products (columns). For each edge $(u, v) \in E$, set $A[u, (u, v)] = 1$ and $A[v, (u, v)] = 1$ (all other entries are 0).

Now G has an independent set of size k if and only if A has a diverse subset of size k. Since the construction takes polynomial time, DS is NP-complete.

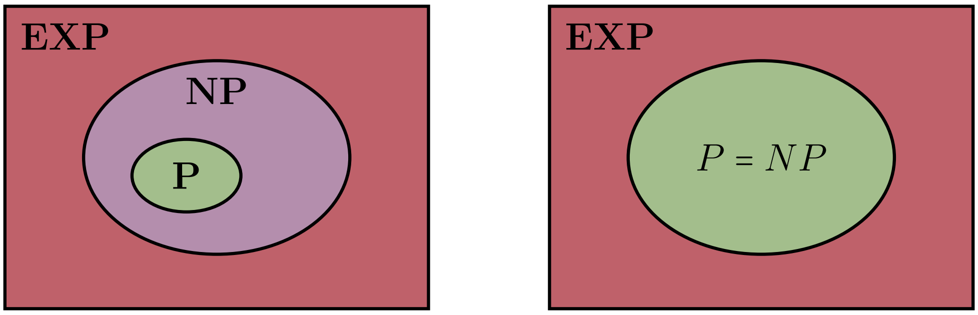
Describe the argument a little better

List of NP-complete Problems

- Subset Sum
- Independent Set (Graph Algorithm)
- Vertex Cover (Graph Algorithm)
- Set Cover
- Longest Path (Graph Algorithm)
- Max Cut (Graph Algorithm)

- 3-Coloring (Graph Algorithm)
- Hamiltonian Cycle (Graph Algorithm)
- Travelling Salesperson (Graph Algorithm)

Is P = NP?



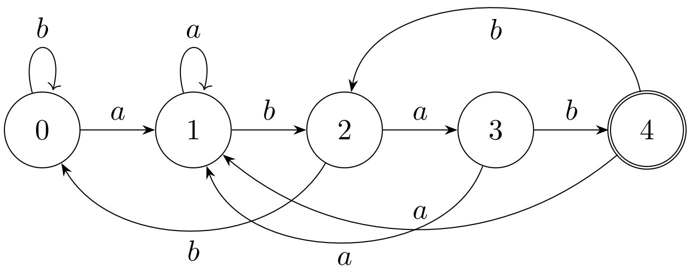
If $P \neq NP$ If $P = NP$

String Matching

Finite Automata

This is a way to match strings (similar to FSM). When building the graph, just remember that each nodes needs a edge pointing out for each letter in the alphabet.

Example: Finite Automaton for pattern "abab". *Note:* if a character path you want to take isn't shown, go back to the beginning.



Knuth–Morris–Pratt (KMP)

Given a prefix π for a string with length i, then build a graph from that pattern.

Example: KMP for pattern "abab"

The solid arrows show character transitions, while dashed red arrows show failure links (π values).

i	1	2	3	4
$\pi[i]$	0	0	1	2
`P[i]`	a	b	a	b

Figure 4: Prefix function table

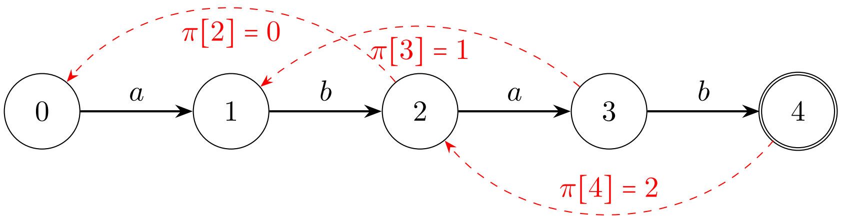


Figure 5: Automaton for KMP

String Matching Overview

	Construction Time	Matching Time	Total Time	Explanation
Finite Automaton	$O(m \cdot \lvert\Sigma\rvert)$	$O(n)$	$O(m \cdot \lvert\Sigma\rvert + n)$	Σ is the size of the alphabet, m is the size of the pattern
KMP	$O(m)$	$O(n)$	$O(n+m)$	n is the size of the string, m is the size of the pattern

Tries

	Search	Prefix Search	Construction	Space	Explanation
Trie	$O(m)$	$O(m + occ)$	$O(n)$	$O(n)$	m is query length, occ is number of matches, n is total length of all strings
Compact Trie	$O(m)$	$O(m + occ)$	$O(n)$	$O(s)$	Compressed version with edge labels, s is number of strings