Algorithms & Data Structures II

Sebastian F. Taylor

November 24, 2025

Contents

Introduction to Dynamic Programming **1**
- Weighted Interval Scheduling . . . 1
- Knapsack . . . 1
- Sequence Alignment . . . 1
- Max Flow & Min Cut . . . 2
- Flow Algorithms . . . 2
- Running the algorithms . . . 2
- Flow Problem Recipe . . . 2
  - Build the Graph . . . 2
  - Describe the Algorithm . . . 2
  - Analyse Runtime . . . 3

Randomised Algorithms **3**
- Expected Value . . . 3
- Useful Sums . . . 3

Hashing **3**
- Chained Hashing . . . 3
- Linear Probing . . . 4
- Hash Functions . . . 4
- Performance Comparison . . . 4

Introduction to 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.

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`
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

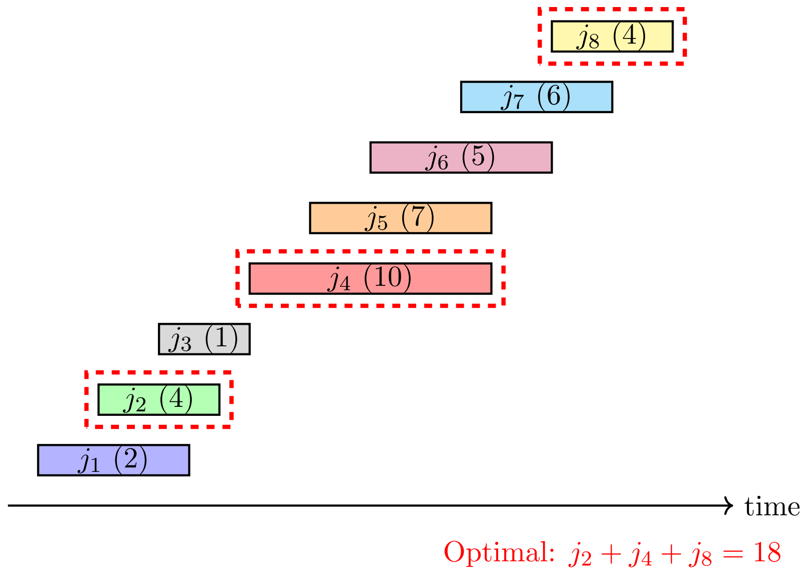

Figure 1: Weighted Interval Scheduling Example: Jobs are sorted by finish time. The optimal solution selects non-overlapping jobs with maximum total value.

Max Flow & Min Cut

Flow Algorithms

Name	Running Time	Explanation	Notes
Ford-Fulkerson	$O(\lvert f^{\cdot}\rvert m)$	$\lvert f^{\cdot}\rvert$ = 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 $x = (x_1, x_2, \ldots, x_r)$ in base m (prime), define

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

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

Performance Comparison

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

$^{\cdot}$ Expected time (when using universal hash function)