# Dynamic Programming

**Contents:**

# Rod Cutting

**Problem:** *Given a rod of length $N$ and a list $P$ containing the value of pieces of lengths 1 to $N$, return the maximum value that can be obtained by cutting the rod.*

**Solution:**

Like most DP problems, it is often good to start with the brute-force solution and then optimize it. At each "cut" point of the rod, we can choose to either cut it there or to not cut it there. For a rod of length $N$, we will have $N - 1$ cut points, and so the total number of possible cut combinations will be $2^{N-1}$. The brute-force solution would be to try all possible combinations, and then evaluate all of them in time $N$ to find the maximum. This would give an overall runtime proportional to $\Theta(2^n)$.

To get to the improved DP solution, we have to figure out what the subproblems are. The issue with the brute-force solution is that it does repeated work, i.e. the same subproblems are computed multiple times. To fix this, we can make use of a memo table (call it $R$) to store previously computed results. To start off, we note that:

$R[0] = 0$

$R[1] = P[1]$

These will be our base cases. Each higher index $i$ of $R$ will then store the maximum value that can be obtained by making cuts in a rod of length $i$, and eventually we will return $R[N]$ as our final result.

**Pseudocode:**

```
func cutRod(N, P) {
    R[0] = 0
    for i = 1 to N {
        maxVal = -INF
        for j = 1 to i {
            // Take the max over all possible cuts, considering previous values
            maxVal = max(maxVal, P[j] + P[i-j])
        }
        R[i] = maxVal
    }
    return R[N]
}
```

**Complexity:**

The above algorithm has two nested for-loops, and so the complexity will be proportional to $\Theta(n^2)$.

## Computing Binomial Coefficients

**Problem:** *Given a binomial coefficient $\binom{n}{k}$, compute its result.*

**Solution:**

A binomial coefficient can be computed by writing it out in the form $\frac{n!}{k!(n-k)!}$, but computing these factorials can quickly become inefficient for larger values of $n$. To get around this, we make use of the following equation:

$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$

This can be implemented in a straightforward recursive program. However, each subproblem makes two new recursive calls, and so the runtime will clearly be exponential. To improve the runtime, we can make use of a table/cache to memoize the algorithm.

**Pseudocode:**

Naive recursive algorithm:

```
func binom(n, k) {
    if (k = 0 or k = n) return 1
    else if (k > n) return 0
    else {
        return binom(n-1, k) + binom(n-1, k-1)
    }
}
```

DP algorithm:

```
func binom(n, k, cache) {
    if (k = 0 or k = n) return 1
    else if (k > n) return 0
    else if (cache[n, k] != 0) return cache[n, k]
    else {
        cache[n, k] = binom(n-1, k, cache) + binom(n-1, k-1, cache)
        return cache[n, k]
    }
}
```

**Complexity:**

In the naive implementation, the runtime is $\Theta(2^n)$. The DP implementation improves the runtime to $\Theta(nk)$.

## The K-Subsets Partitioning Problem

**Problem:** *Given a list $L$ of elements [1,2...,N], count the number of ways to partition it into $k$ non-empty subsets.*

**Solution:**

Suppose we pick an arbitrary element from $L$. With this element, we have two choices: either we put it into a new subset, or we put it into an existing subset. This gives us a natural recurrence:

$$P[n,k] = P[n-1,k] \cdot k + P[n-1,k-1]$$

Where $P[n,k]$ represents the number of possible subset partitions for a list of size $n$. The $P[n-1,k]$ term represents the option where we place an element into an existing subset, and the $P[n-1,k-1]$ term represents the option where the element is placed into a new subset (and there are now $k-1$ more subsets that we need to make).

It is important to note that the first term in the recurrence must be multiplied by $k$. This is to account for all of the possible subsets that an element can be placed in.

**Pseudocode:**

```
func subsetPartition(n, k) {
    P[n, 1] = 1
    P[n, n] = 1

    for i = 1 to n {
        for j = 1 to min(k, i) {
            if (i = j or j = 1) P[i, j] = 1
            else {
                P[i, j] = P[i-1, j]*j + P[i-1, j-1]
            }
        }
    }
    return P[n, k]
}
```

**Complexity:**

The brute-force solution considers all possibilities and gives a runtime proportional to $\Theta(2^n)$. The DP solution has the same recurrence as the Binomial Coefficients problem (with the only difference being the multiplication by $k$), and it also has the same runtime of $\Theta(nk)$. This can also be seen with the two nested for-loops above.

## The Travelling Salesman Problem

**Problem:** *A salesman has a collection of cities that he wants to pass through. Additionally, he would like to pass through each city exactly once and return to the city at which he started. His time is valuable, so he would like to do this tour of cities in the minimum time possible. Given the weighted graph of cities, find the closed path (tour) of minimum weight.*

**Solution:**

The brute-force solution to this problem is relatively straightforward, at least conceptually. If we have $N$ cities that are all connected, then there are $(N-1)!$ possible paths that the salesman can take, since the starting city is fixed (the starting city actually doesn't matter at all, any city can be picked). Once all permutations are generated, we can check all of them in time $N$ to find the permutation of minimum weight and return that as our result. This takes time $\Theta(n!)$ overall.

With dynamic programming, we can improve the runtime to be exponential instead of factorial. First, we can assume we are given a matrix $dist$, where $dist[i, j]$ gives us the length/weight of the path between cities $i$ and $j$. In our subproblems, we would like to keep track of the minimum weight path between two cities, but we also need to keep track of which cities are in that path so that we do not get any cycles. We can then define $L[1, S, j]$ as the shortest path from city $1$ (assumed to be the starting city) to city $j$, and $S$ will contain the cities present in that path (minus $j$).

*Note:* $L[1, \varnothing, j] = dist[1, j]$

The key to getting the subproblems is to consider all possible *last* vertices that can be visited before leaving $S$. Then:

$$L[1, S, j] = min_{l \in S}(L[1, S - \{l\}, l] + dist[l, j])$$

By removing $l$ from $S$, we reduce our problem size and create a subproblem. The DP algorithm will iteratively fill in $L[1, S, j]$ for all $S \subseteq \{2, 3..., N\}$ where $j \notin S$.

**Pseudocode:**

```
func tsp(dist) {
    // Fill base cases in table
    for j = 2 to n {
        L[1, {}, j] = dist[1, j]
    }

    for k = 2 to n-1 { // Consider subsets of all sizes
        for all subsets where |S| = k and S is a subset of {1,2...,n} {
            for all j not in S {
                // The min is over all l in S
                L[1, S, j] = min(L[1, S-{l}, l] + dist[l, j])
            }
        }
    }
}
```

**Complexity:**

The DP algorithm considers all possible subsets, of which there are $2^n$ possibilities. On top of this, for each subset the algorithm must check that the current city is not in $S$, which takes time $\leq n$, and it must also take the minimum over all $l \in S$, which also takes time $\leq n$. Therefore the overall runtime is $\Theta(n^2 2^n)$.

*Note: the extra space/memory required will be $\Theta(n2^n)$ for $n$ rows and $2^n$ columns.*

Once the table is filled, we still need to find the optimal path. To do this, we iterate over all possible last cities and take the minimum:

$$Shortest = min_{j \in \{2...,n\}}(L[1, \{2, 3..., n\} - \{j\}, j] + dist[j, 1])$$

This step takes $\Theta(n)$.

## The Knapsack Problem

**Problem:** Given a knapsack of size $k$ and a list $N$ of objects where each index represents the size of the object, determine if there exists a subset of $N$ such that the sum of the sizes of the objects is exactly $k$.

**Solution:**

The brute-force solution to this problem involves generating all $2^n$ subsets and then checking to see if any sum up to $k$. Since this gives an exponential runtime, we would like to do better.

To implement a DP solution, we will define two tables: $L[i, j]$ and $S[i, j]$. $L[i, j]$ will contain whether it is possible (i.e. true or false) to make a subset of size $j$ using only up to the $i^{th}$ element from $N$. $S[i, j]$ will be true if $N[i]$ is part of the solution to $L[i, j]$ and false otherwise. Then, we can note that:

$$P[0, 0] = true$$

$$P[0, j] = false, j > 0$$

To create our DP relationship, we can observe all of the possible cases for $P[i, j]$:

- There is an element $N[j]$, $j < i$ such that $N[j] = j$, i.e. a single element fills the entire capacity. Then clearly $P[i, j] = true$.
- If there is no single element in the subset of $N$ that fills the entire capacity, then we have 2 options:
    - Do not include the current element in the final solution. Then, we must see if $P[i-1, j]$ is true or false.
    - Include the current element in the final solution. Then we check $P[i-1, j-N[i]]$.

This can be done to fill up the entire table. It is worth noting that if the goal of the algorithm is only to see if a solution exists, then only $P[i, j]$ is required. However, if actual solution itself needs to be returned, then $S[i, j]$ is required as well.

**Pseudocode:**

Without $S[i, j]$:

```
func knapsack(N, k) {
    for i = 0 to size(N)-1 {
        for j = 0 to k {
            if (i = 0 and j = 0) {
                P[0,0] = true
            } else if (i = 0 and j > 0) {
                P[0,j] = false
            } else {
                if (N[i] = j) {
                    P[i,j] = true
                } else if (N[i] > j) { // Element is too big
                    P[i,j] = P[i-1,j]
                } else {
                    // Check if it possible to make a subset of size j
                    // by including the element or excluding it
                    P[i,j] = P[i-1, j] or P[i-1, j-N[i]]
                }
            }
        }
    }
    return P[size(N)-1, k]
}
```

With $S[i, j]$:

```
func knapsack(N, k) {
    for i = 0 to size(N)-1 {
        for j = 0 to k {
            if (i = 0 and j = 0) {
                P[0,0] = true
            } else if (i = 0 and j > 0) {
                P[0,j] = false
            } else {
                if (N[i] = j) {
                    P[i,j] = true
                    S[i,j] = true
                } else if (N[i] > j) { // Element is too big
                    P[i,j] = P[i-1,j]
                    S[i,j] = false
                } else if (P[i-1, j] = true) {
                    P[i,j] = true
                    S[i,j] = false
                } else if (P[i-1, j-N[i]] = true) {
                    P[i,j] = true
                    S[i,j] = true
                } else {
                    P[i,j] = false
                    S[i,j] = false
                }
            }
        }
    }

    // Return a solution if one exists
    j = k
    if (P[size(N)-1, k] = true) {
        for i = size(N)-1 to 0 {
            if (S[i,j] = true) {
                output N[i]
                j = j - N[i]
            }
        }
    } else {
        // No solution exists
    }
}
```

**Complexity:**

The first algorithm without $S[i, j]$ uses two nested for loops and runs in time $\Theta(nk)$. The second algorithm also has a second loop at the end which takes time $\leq N$, so the overall complexity is still $\Theta(nk)$.

## Worker Assignment

**Problem:** *Assume we are given $N$ workers and $N$ jobs. We want to assign each worker to exactly one job, and to do this we are given a table $N[i,j]$, where this would represent the value obtained from assigning worker $i$ to job $j$. Given this, maximize the total value of the job assignments.*

**Solution:**

The brute-force solution to this problem would be to generate all possible assignments, and check the sum of each to find the maximum. This would take time $\Theta(n! \cdot n)$.

We can improve this runtime with dynamic programming. The general idea is to pick a worker and assign them to a job. We don't know if this is the optimal selection, but we can assume that it is. Then, we are left with a smaller matrix that does not include the selected worker and the assigned job. This gives us a natural subproblem structure.

To implement the algorithm, we will use a table $best[A, B]$, where $A$ is a subset of all workers and $B$ is a subset of all jobs, that represents the highest value job assignment that can be made with the sets $A$ and $B$. The DP approach will be to compute $best[A, B]$ for all possible subsets $A, B$ of the same size. The recurrence will then be:

$best[A, B] = max_{i \in A, j \in B}(N[i, j] + best[A - \{i\}, B - \{j\}])$

**Pseudocode:**

```
func assign(N) {
    for k = 0 to size(N)-1 { // Iterate over all possible subset sizes
        for all subsets A, B of size k {
            if (k = 0) best[A,B] = 0
            else {
                // For all i in A, j in B
                best[A,B] = max(N[i,j] + best[A-{i},B-{j}])
            }
        }
    }
    return best[N,N]
}
```

**Complexity:**

Generating all subsets takes time $\Theta(2^n)$, and this is done for both $A$ and $B$. On top of this, finding the max at each iteration takes time $\Theta(n^2)$. Therefore the overall runtime is $\Theta(2^n \cdot 2^n \cdot n^2) = \Theta(4^n n^2)$.

## Longest Common Subsequence

## Optimal Binary Search Tree

## Matrix Chains