CSCC73 A6

Saad Makrod

November 2022

Q1 DP Algo to Find Largest Nonconsecutive Sum

<u>SUBPROBLEMS TO SOLVE:</u> For each i, $1 \leq i \leq n$, let

$$S(i) = \text{the maximum nonconsecutive sum from } A[1..i] \tag{*}$$

<u>SOLVING THE ORIGINAL PROBLEM:</u> The answer to the original problem is $S(n)$

<u>RECURSIVE FORMULA TO COMPUTE THE SUBPROBLEMS:</u>

$$S(i) = \begin{cases} 0 & i = 0 \\ max(A[i], S[i-1]) & i = 1 \\ max(A[i] + S[i-2], S[i-1]) & 1 < i \leq n \end{cases} \tag{†}$$

<u>JUSTIFICATION WHY (†) CORRECTLY COMPUTES (*):</u> There are three cases,

CASE 1. If $i = 0$ then we have an empty subsequence of A so (†) returns the correct result in this case.

CASE 2. If $i = 1$ then either the max is 0 or the element itself since $S[i-2]$ is not well defined. We know that $S[0] = 0$ so (†) returns the correct result in this case.

CASE 3. In this case $S[i]$ either includes $A[i]$ or it does not include $A[i]$. If $S[i]$ includes $A[i]$ then by definition since the sum is nonconsecutive it cannot include $A[i-1]$. This means that $S[i] = A[i] + S'$ where S' is another non-consecutive sum from $A[1..i-2]$. Note that this is by a standard cut and paste argument. By definition $S' = S[i-2]$. Thus in this case $S[i] = A[i] + S[i-2]$ which is computed by (†). Furthermore we should note that $S[j] \geq 0$ for all $1 \leq j \leq n$ since we know that an empty subsequence has sum 0. Thus we do not need to handle any negative cases. If $S[i]$ does not include $A[i]$ then by definition since the sum is optimal for $A[1..i-1]$. Note that this is by a standard cut and paste argument. By definition the sum being optimal for $A[1..i-1]$ is equivalent to $S[i-1]$. Thus in this case $S[i] = S[i-1]$ which is computed by (†). Thus since (†) returns the max of the two possible sums (†) returns the correct result in this case.

<u>PSEUDOCODE:</u>

MAX_NONCONSECUTIVE_SUM(A)

$M = \emptyset$

$S[0] = 0; S[1] = max(A[1], S[0])$

for $i = 2$ to n

$S[i] = max(A[i] + S[i-2], S[i-1])$

$i = n$

while $i \geq 1$

if $S[i] \neq S[i-1]$

$M = M \cup A[i]$

$i = i - 1$

$i = i - 1$

return M

RUNNING TIME ANALYSIS: There are two for loops, one to calculate the $S[i]$s and the other to construct the subsequence which is returned. Both loops run in $\Theta(n)$ time. Additionally all other operations are $O(1)$ operations. Thus this is a $\Theta(n)$ algorithm.

Q2 DP Algo to Minimize Makespan of 2 Process Scheduling

SUBPROBLEMS TO SOLVE: Let $A = [t_1, t_2, ..., t_n]$. For each i, $1 \leq i \leq n$, let

$$M(i) = \text{the set of loads } \{m_1, m_2\} \text{ that can be generated from } A[1..i] \qquad (*)$$

Note that m_1 is the load on machine 1 and m_2 is the load on machine 2. The makespan can be determined by $\max(m_1, m_2)$. Note that this particular set defines equality between two elements $\{m_1, m_2\}$ and $\{m_1', m_2'\}$ as the following $m_1 = m_1'$ and $m_2 = m_2'$ or $m_1 = m_2'$ and $m_2 = m_1'$.

SOLVING THE ORIGINAL PROBLEM: The answer to the original problem is min makespan($M(n)$)

RECURSIVE FORMULA TO COMPUTE THE SUBPROBLEMS:

$$M(i) = \begin{cases} \{A[1], 0\} & i = 1 \\ \{\{m_1 + A[i], m_2\}, \{m_1, m_2 + A[i]\} : \{m_1, m_2\} \in M(i-1)\} & 1 < i \leq n \end{cases} \qquad (\dagger)$$

JUSTIFICATION WHY (†) CORRECTLY COMPUTES (*): There are two cases,

CASE 1. If we have $i = 1$ then there is only one possible makespan arrangement with the way equality is defined which is $\{A[1], 0\}$. Thus (†) returns the correct result.

CASE 2. In the case where $1 < i \leq n$ we need to generate all new loads which include $A[i]$. Note that by definition $M[i-1]$ is the set of loads $\{m_1, m_2\}$ that can be generated from $A[1..i-1]$. To get $M(i)$ we need to now consider $A[1..i]$. Note that since we have the set of loads generated by $A[1..i-1]$ we can calculate this by considering each load in $M(i-1)$ and adding $A[i]$ to either m_1 or m_2. This is because for each set of loads that consider $A[1..i-1]$ we can make it consider $A[1..i]$ by adding $A[i]$ to either m_1 or m_2.

For sake of contradiction assume that this is not the case and there is some load that considers tasks $t_1, t_2, ..., t_i$ that is not generated by considering each load in $M(i-1)$. This means that there is some arrangement of $t_1, t_2, ..., t_{i-1}$ (if we exclude t_i). However by definition this arrangement must have been considered by $M(i-1)$ which is a contradiction.

Thus (†) computes the correct result in this case.

PSEUDOCODE:

MIN_MAKESPAN(A)

    M(i) = $\{A[i], 0\}$

    $L_1 = \{\{A[i]\}, \emptyset\}$

    $L_2 = \emptyset$

    for i = 2 to n

        S = $\emptyset$; $L_2 = L_1$; $L_1 = \emptyset$; i = 1;

        for $\{m_1, m_2\} \in M(i-1)$

            S = S $\cup \{\{m_1 + A[i], m_2\}, \{m_1, m_2 + A[i]\}\}$

            $\{l_1, l_2\} = L_2[i]$

            $L_1 = L_1 \cup \{l_1 \cup \{A[i]\}, l_2\} \cup \{l_1, l_2 \cup \{A[i]\}\}$

            i++

        M(i) = S

    i = index of min makespan(M)

    return $L_1[i]$

RUNNING TIME ANALYSIS: Note that there are two for loops the first for loos will executes in $\Theta(n)$ since it loops from 2 to n. Note that to analyze the run time of the second for loop it is important to note that the number of possible assignments is bounded from above by T. This means the number of possible unique load pairs is also T. Since $M(i)$ is a set this means that the number of elements in $M(i)$ is unique. This implies that the number of elements in $M(i)$ is bounded from above by T. Thus the second for loop runs in $O(T)$. Lastly to find the index of the min makespan will take $O(T)$ since as justified earlier $M(i)$ has at most T elements. Note that all other operations are $O(1)$. Thus we have $\Theta(n)O(T) = O(T)$ algorithm which is $O(nT)$ runtime.

Q3 Modified Floyd-Warshall to Find Number of Min Weight $u \to v$ Paths

Consider the following algorithm,

```
MODIFIED_FLOYD_WARSHALL(G,wt)
    for i = 1 to n
        for j = 1 to n
            if i = j then C(i, j, 0) = 0
            elif (i, j) ∈ E then C(i, j, 0) = wt(i, j)
            else C(i, j, 0) = ∞
    for k = 1 to n
        for i = 1 to n
            for j = 1 to n
                if C(i, j, k-1) ≤ C(i, k, k-1) + C(k, j, k-1) then C(i, j, k) = C(i, j, k-1)
                    else C(i, j, k) = C(i, k, k-1) + C(k, j, k-1)
    for i = 1 to n
        for j = 1 to n
            if C(i, j, n) ≠ ∞ then N[i, j] = 1
            else N[i, j] = 0
    for k = 1 to n
        for i = 1 to n
            for j = 1 to n
                if C(i, j, n) = C(i, k, n) + C(k, j, n) and C(i, j, n) ≠ ∞ and k ≠ i and k ≠ j
                    N[i, j] = N[i, j] + N[i, k] * N[k, j]
    return N
```

<u>CORRECTNESS</u>: Note that I will refer to each top-level for loop as loop 1, 2, 3, 4 in the order they appear in the algorithm. We want a N returned such that for each pair of nodes $u, v \in V$ N[u, v] is the number of minimum-weight $u \to v$ paths. The algorithm will first proceed as normal and calculate the minimum-weight $u \to^k v$ paths. There are two cases to consider, one being that there is no $u \to v$ path and the other being that there are $u \to v$.

The first case is trivial. If there are no $u \to v$ paths then $C(u, v, n) = \infty$. As seen above in loop 3 N[u, v] is set to 0. Furthermore, if $C(u, v, n) = \infty$ then it is never modified in loop 4 (this can be seen straight from the algorithm). Thus N[u, v] will be 0 as wanted.

In the second case where $u \to v$ paths exists we want the number of minimum-weight $u \to v$ paths returned. We know that if $C(u, v, n) \neq \infty$ then a minimum weight path must exist and the weight of the path is exactly $C(u, v, n)$. Thus we can compare the weight of $C(u, v, n)$ with different paths by having an intermediate node which we will call k. If we ever find a path where $C(u, v, n) = C(u, k, n) + C(k, v, n)$ then we have found a shortest $u \to v$ path. However there can be multiple ways to get from $u \to k$ and $k \to v$ thus we have to take the product to get the total $u \to v$ paths. This is exactly what is done by loop 4.

Thus the N returned is such that for each pair of nodes $u, v \in V$ N[u, v] is the number of minimum-weight $u \to v$ paths.