CS310 – Advanced Data Structures and Algorithms

Algorithms: Divide and Conquer, Greedy, and Dynamic Programming
Spring, 2021
Algorithm Techniques

There are patterns in algorithms worth studying
We’ll cover:

• Divide and conquer: we already saw examples
• Greedy algorithm: follow what appears to be best at each step, example coming up
• Dynamic programming: save partial results as you go, then reuse them
What to read?

• Unfortunately, our main text, although named “Algorithms”, does not organize material into these categories. It does cover some of the individual algorithms: I’ll reference them as we go along.
  • See page 272 for categorizing mergesort as “divide and conquer”
• The recommended text by Jon Kleinberg and Eva Tardos is excellent on this, having full chapters on each category, but it is a little advanced for us.
• Weiss organizes algorithms. His Sec. 7.5 is on divide and conquer, and Sec. 7.6 is on Dynamic Programming. Greedy algorithms are discussed briefly at the start of Sec. 7.6.
Divide and Conquer

• Mergesort: mergesort halves of array, merge them
• Quick sort: partition into two parts using pivot, Quicksort each part and put together result of lo-part, pivot, high-part.
• Binary search of sorted array: look at middle item, search left half or right half depending on middle item vs. target item
• Now another example not sorting or searching...

3/4/2021
Maximum Contiguous Subsequence Sum

Problem: Given a sequence of integers \( (A_1, A_2, \ldots, A_N) \), possibly negative.

Identify the subsequence \( (A_i, \ldots, A_j) \) that corresponds to the maximum value of

\[
\sum_{k=i}^{j} A_k
\]

Reference: Weiss Sec. 7.5.1,

- Naïve approach is cubic (examine all \( O(N^2) \) sequences and sum each one).
- Use a divide-and-conquer algorithm.
Divide-and-Conquer Algorithm for max. contig. subsequences

Sample input is \{4, -3, 5, -2, -1, 2, 6, -2\}

Some Possibilities for contiguous subsequences: first 3, 2^{nd} through 5^{th}, last two.
\{4, -3, 5, -2, -1, 2, 6, -2\}

- We want to divide the problem in half, but some sequences cross over.
- See 3 possible cases for a subsequence:
  1. in the first half
  2. in the second half
  3. begins in the first half and ends in the second half
Divide-and-Conquer Algorithm for max. contig. subsequences

{4, -3, 5, -2, -1, 2, 6, -2}

- We want to divide the problem in half, but some sequences cross over.
- Divide and conquer: believe you can do the smaller problem and build on it.
- The max. contig subsequence = max one from left half or max one from right half or one that crosses over (case 3).
- The two recursive calls take care of the two halves, but we need to handle the cross-over case in additional code.

3/4/2021
Divide-and-Conquer Algorithm

Sample input is \{4, -3, 5, -2, -1, 2, 6, -2\}

Trick to analyze cross-over case (case 3): we know it goes through the mid-point and some distance in each direction. Compute the subsequence sums both ways out and find the max partial sums

For case 3: \textit{sum} = \textit{sum 1}^{\text{st}} + \textit{sum 2}^{\text{nd}}

<table>
<thead>
<tr>
<th>First Half</th>
<th>Second Half</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>-3</td>
</tr>
<tr>
<td>4*</td>
<td>0</td>
</tr>
</tbody>
</table>

Running sum from the center (*denotes maximum for each half).
Divide-and-Conquer Algorithm

Sample input is \{4, -3, 5, -2, -1, 2, 6, -2\}

For case 3: \( \text{sum} = \text{sum } 1^{\text{st}} + \text{sum } 2^{\text{nd}} \)

<table>
<thead>
<tr>
<th>First Half</th>
<th>Second Half</th>
<th>Values</th>
<th>Running sums</th>
</tr>
</thead>
<tbody>
<tr>
<td>4  -3  5  -2</td>
<td>-1  2  6  -2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4*  0  3  -2</td>
<td>-1  1  7*  5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Running sum from the center (*denotes maximum for each half).

So the best cross-over subsequence is \{4, -3, 5, -2, -1, 2, 6, -2\}. We found it in \(O(n)\) processing.
So Case 3 is solved in linear time.

Summary:
Recursively compute the max subsequence sum in the first half
Recursively compute the max subsequence sum in the second half
Compute, via 2 loops, the max subsequence sum that begins in the first half but ends in the second half
Choose the largest of the 3 sums

See Weiss 7.5 if interested in performance analysis
Result: O(N \log N)
Divide-and-Conquer Algorithm

```java
private static int maxSumRec( int[] a, int left, int right )
{
    int maxLeftBorderSum = 0, maxRightBorderSum = 0;
    int leftBorderSum = 0, rightBorderSum = 0;
    int center = ( left + right ) / 2;

    if( left == right ) // Base case
        return a[ left ] > 0 ? a[ left ] : 0;

    int maxLeftSum = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );

    for( int i = center; i >= left; i-- )
    {
        leftBorderSum += a[ i ];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }

    for( int i = center + 1; i <= right; i++ )
    {
        rightBorderSum += a[ i ];
        if( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }

    return max3( maxLeftSum, maxRightSum, maxLeftBorderSum + maxRightBorderSum );
}
```
Greedy algorithms

- Not discussed as such in S&W (has index entry). See pg. 287 of Weiss. Whole chapter (Chap. 4) in Kleinberg and Tardos.
- A greedy person grabs everything they can as soon as possible.
- Similarly a greedy algorithm makes decisions that appear to be the best thing to do at each step.
- Example: Change-making greedy algorithm for “change” amount, given many US coins of each size:

  Loop until change == 0:
  
  Find largest-valued coin less than change, use it.
  
  change = change – coin-value;
Problem – making change

• Task – buy a cup of coffee (say it costs 63 cents).
• You are given an unlimited number of coins of all types (neglect 50 cents and 1 dollar).
• Pay exact change.
• What is the combination of coins you'd use?

![1 cent](image1.png) 5 cent 10 cent 25 cent
Greedy algorithms - change making

- Logically, we'd minimize the number of coins.
- Change-making with the fewest number of US coins—have 1, 5, 10, 25 unit coins to work with.
- Clearly we want to mainly use large-value coins to minimize the total number. So for 27 cents, clearly we can’t do better than $25 + 2(1)$.
- What about 63? Use as many 25s as fit, $63 = 2\times(25) + 13$, then as many 10s as fit in the remainder: $63 = 2\times(25) + 1\times(10) + 3$, no 5's fit, so we have $63 = 2\times(25) + 1\times(10) + 3\times(1)$, 6 coins.
Change making: when greedy doesn’t work…

- The greedy method gives the optimal solution for US coinage.
- With different coinage, the greedy algorithm doesn’t always find the optimal solution.
- Example of a coinage with an additional 21 cent piece. Then $63 = 3(21)$, but greedy says use 2 25s, 1 10, and 3 1’s, a total of 6 coins.
- The coin values need to be spread out enough to make greedy work. But even some spread-out cases don’t work. Consider having pennies, dimes and quarters, but no nickels.
- Then 30 by greedy uses 1 quarter and 5 pennies, ignoring the best solution of 3 dimes.
(Very bad) recursive solution

coins = {25, 10, 5, 1, 21}
makeChange(amt)
If amt in coins return 1
minCoins = amt
loop over j from 1 to amt/2
  thisCoins = makeChange(j) + makeChange(amt-j)
  if thisCoins < minCoins
    minCoins = thisCoins
Lots and lots of redundant calls!

More than double recursion: multiway recursion!
(Very bad) recursive solution

\[ T(n) = T(n-1) + T(n-2) + T(n-3) + \ldots + T(n/2) + \ldots \]

worse than \( T(n) = T(n-1) + T(n-2) \) the famous Fibonacci sequence discussed previously. Fibonacci is exponential, so this certainly is.
Better idea
Looking for change for 63 cents with 21-cent piece

- We know we have 1, 5, 10, 21 and 25 cent coins.
- Therefore, the optimal solution must be the minimum of the following:
  1 (A 1 cent) + optimal solution for 62.
  1 (A 5 cent) + optimal solution for 58.
  1 (A 10 cent) + optimal solution for 53.
  1 (A 21 cent) + optimal solution for 42.
  1 (A 25 cent) + optimal solution for 38.
- This reduces the number of recursive calls drastically.
- Naïve implementation still makes lots of redundant calls.
Dynamic programming (DP) implementation

- DP Idea – instead of performing the same calculation over and over again, save previously calculated results to an array (or map).
- The answer to a large change depends only on results of smaller calculations, so we can calculate the optimal answer for all the smaller change values and save it to an array.
- Then go over the array and minimize on:
  - change(K) = \( \min \{ \text{change}(K-n) + 1 \} \)
  - For all N types of coins of value n
  - For ex., change(33) = \( \min \{ \text{change}(32) + 1, \text{change}(28) + 1, \text{change}(23) + 1, \text{change}(8) + 1 \} \)
- Runtime - \( O(N \times K) \).
DP for Change Making

- \#coins = change(N) = \min_k \left( \text{change}(N-k) + 1 \right)
  - Where k = coin size (1, 5, 10, 21, 25) for example
- We can turn this into recursive search with DP helping
- Or build up a table of coinsUsed by looking at change-for-1-cent, then change-for-2-cents, ...
- Get coinsUsed = \{0, 1, 2, 3, 4, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, ...\} up to 14 cents, the last being coinsUsed[14] = 5 (one dime, 4 pennies)
- Now work on 15 cents: first value 15 coins, try to do better...
Idea of DP change making: look up previous results

- So far coinsUsed = {0,1,2,3,4,1,2,3,4,5,1,2,3,4,5, ...} up to 14 cents
- Now work on 15 cents: first value is 15 coins, try to do better...
  - Try a penny as part, leaves 14 cents, look up coinsUsed[14] = 5, so new value is 5+1 = 6, better than 15, so the new min value.
  - Try a nickel as part, leaves 10 cents, look up coinsUsed[10] = 1 coin, so 2 in all, much better, a new min value
  - Try a dime as part, leaves 5, look up and find 1 coin, 2 in all, not better
  - Try a 21-cent piece, too big, try 25, too big, done: answer is 2 coins, fill in coinsUsed[15] = 2

3/4/2021
public static void makeChange( int [] coins, int maxChange, int [] coinsUsed, int [] lastCoin )
{
    coinsUsed[ 0 ] = 0; lastCoin[ 0 ] = 1;
    for( int cents = 1; cents <= maxChange; cents++ ) {
        int minCoins = cents;
        int newCoin = 1;
        for( int j = 0; j < coins.length; j++ ) {
            if( coins[ j ] > cents )   continue; // Cannot use coin j
            if( coinsUsed[ cents - coins[ j ] ] + 1 < minCoins ) {
                minCoins = coinsUsed[ cents - coins[ j ] ] + 1;
                newCoin = coins[ j ];
            }
        }
        coinsUsed[ cents ] = minCoins;
        lastCoin[ cents ] = newCoin;
    }
}

• This fills coinsUsed[] as shown on last slide, and also lastCoin[], used to trace back and find the actual coins used
Another famous example is the sequence of binomial coefficients can be generated by Pascal’s triangle:

```
1
1 2 1
1 3 3 1
1 4 6 4 1
\ / \ / \ / \\
5
```

Each number is the sum of the two closest above it.
Binomial coefficients

- Start a new row with 1’s on the edges.
- The row number is N, and the entries are k=0, k=1, ..., k=N across a row, so for example

\[ C(4,0) = 1, \ C(4,1) = 4, \ C(4,2) = 6, \ C(4,3) = 4, \ C(4,4) = 1. \]

- These are the coefficients of the binomial expansion
- \((x + y)^4 = x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4\)
Binomial coefficients and n choose k

- Also, $C(N, k) =$ number of ways to choose a set of $k$ objects from $N$
- Ex. $C(4, 2) = 6$ The 2-sets of 4 numbers are the 6 sets:
  \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}$
- Recursion: $C(N, k) = C(N-1, k) + C(N-1, k-1)$ This is just the sum rule of Pascal’s triangle.
n choose k - relationships

- Base cases: \( C(N, 0) = 1, C(N, N) = 1 \)
- To choose \( k \) objects from \( N \), set one object \( x \) aside and find all the ways of choosing \( k \) objects from the remaining \( N-1 \).
- These are all the sets we want that don’t include \( x \), \( C(N-1, k) \) in number.
- The sets that do include \( x \) also need \( k-1 \) other objects from the other \( N-1 \), \( C(N-1, k-1) \) in number.
- So \( C(N, k) = C(N-1, k) + C(N-1, k-1) \)
If we write a recursive function:

```python
combo(N, k):
    if (k == 0) return 1
    if (k == N) return 1
    return combo(N-1, k) + combo(N-1, k-1)
```

Note the double recursion, without halving the “N” value, so dangerous recursion.
Binomial coefficient - recursion

We get exponential runtime $T(N)$

$$T(N, k) = T(N-1, k) + T(N-1, k-1) \quad \text{-- 2 terms in N-1}$$

$$= T(N-2, k) + \ldots \quad \text{4 terms in N-2}$$

$$= \ldots \text{some of these hit base cases and stop}$$
Efficient calculation of binomial coefficients

- If we save and reuse values, it’s much faster. In other words, use Pascal’s triangle to generate all the coefficients.
- One way: set up a table and use it for each N in turn.

\[
\begin{align*}
C[1][0] &= 1 \\
C[1][1] &= 1 \\
\text{for } n \text{ up to } N \\
&\quad \text{for } k \text{ up to } n \\
&\quad \quad C[n][k] = C[n-1][k] + C[n-1][k-1]
\end{align*}
\]

- O(1) to fill each spot in NxN array, so O(N^2)
Map approach to dynamic programming

• Another approach: set up Map from (N, k) to value.
• Case of classic dynamic programming, saving partial results along the way.
• if N and k both ints, long key = N + (long)k>>32 (Pack two ints in a long)
• Or OO way: create a Pair class, with N, k fields, getters and setters. Also need equals and hashCode for HashMap…
• Either way, have key(N,k) holding the pair.
Map approach to dynamic programming

```python
combo(N, k):
    val = M.get(key(N,k))
    if (val != null) return val
    if (k == 0|| k == N)
        val = 1
    else val = combo(N-1, k) + combo(N-1, k-1)
    M.put(key(N, k), val)
    return val
```

Once this recursion reaches a cell, it fills it in, so work bounded by number of cells “below” (N, k), which is \( \leq O(N^2) \).

“Below” means entries \((i,j)\) where \(i\leq N\) and \(j\leq k\)
Visualizing the DP process: 10 slides to show steps

Recall:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
\ /  
10 = C(5,2)
```

Start with (base cases):

```
1
1 1
1 _ _ 1  N=2
1 _ _ _ 1  N=3
1 _ _ _ _ 1  N=4
1 _ _ _ _ _ 1  N=5
```

place for $C(5,2)$
Visualizing the DP process

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>N=2</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N=3</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N=4</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N=5</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

combo(N=5, k=2):
val = M.get(key(N,k))
if (val != null) return val
if (k == 0 || k == N)
    val = 1
else val = combo(N-1, k) + combo(N-1, k-1)
M.put(key(N, k), val)
return val

At top level, calling combo(4,2) and eventually combo(4,1), but not yet
Visualizing the DP process

1
1 1
1 _ 1 N=2
1 _ _ 1 N=3
1 _ _ _ 1 N=4
1 _ _ _ _ 1 N=5

C(5,2)

 combo(N=5, k=2):
  val = M.get(key(N,k))
  if (val != null) return val
  if (k == 0 || k == N)
    val = 1
  else
    val = combo(N-1, k) + combo(N-1, k-1)
  M.put(key(N, k), val)
  return val

At top level, calling
Combo(5,2) and then
combo(4,2)
(simpler rep.)
Visualizing the DP process

\[
\begin{align*}
&1 \\
&1 \ 1 \\
&1 \ _ \ 1 \quad N=2 \\
&1 \ _ \ _ \ 1 \quad N=3 \\
&1 \ _ \ _ \ _ \ 1 \quad N=4 \\
&1 \ _ \ _ \ _ \ _ \ 1 \quad N=5 \\
&C(5,2)
\end{align*}
\]

\[
\text{combo}(N=4, k=2): \quad \text{val} = \text{M.get(key}(N,k)) \\
\text{if (val != null) return val} \\
\text{if (k == 0 || k == N)} \\
\quad \text{val} = 1 \\
\text{else val} = \text{combo}(N-1, k) + \text{combo}(N-1, k-1) \\
\text{M.put(key}(N, k), \text{val}) \\
\text{return val}
\]

\[
\text{combo}(4,2) \text{ calls combo}(3,1)=1 \quad \text{and then combo}(3,2)
\]
Visualizing the DP process

```
1
1 1
1 _ _ 1 N=3
1 _ _ _ 1 N=4
1 _ _ _ _ 1 N=5
```

```java
combo(N=3, k=2):
    val = M.get(key(N,k))
    if (val != null) return val
    if (k == 0 || k == N)
        val = 1
    else val = combo(N-1, k) + combo(N-1, k-1)
    M.put(key(N, k), val)
    return val

combo(3,2) calls combo(2,1)=1 and then combo(2,2), and similarly at the next level
Fills in C(2,1) = 2 in map
```
Visualizing the DP process

1
1 1
1 2 1 N=2
1 3 _ 1 N=3
1 4 _ _ 1 N=4
1 _ _ _ _ 1 N=5

C(5,2)

combo(3=5, k=2):
  val = M.get(key(N,k))
  if (val != null) return val
  if (k == 0 || k == N)
    val = 1
  else val = combo(N-1, k) +
            combo(N-1, k-1)
  M.put(key(N, k), val)
return val

combo(3,2) finishes, putting 3 in Map for key =(3,2)
Then combo(4,2) finishes...
Visualizing the DP process

1
1 1
1 2 1 N=2
1 3 _ 1 N=3
1 4 _ _ 1 N=4
1 _ _ _ _ 1 N=5

combo(N=5, k=2):
val = M.get(key(N,k))
if (val != null) return val
if (k == 0 || k == N)
    val = 1
else val = combo(N-1, k) +
    combo(N-1, k-1)
M.put(key(N, k), val)
return val

combo(5,2) regains control
And calls combo(4,3), the second
Recursive call.

3/4/2021
Visualizing the DP process

\[
\begin{array}{cccc}
1 \\
1 & 1 \\
1 & 2 & 1 & N=2 \\
1 & 3 & _ & 1 & N=3 \\
1 & 4 & _ & _ & 1 & N=4 \\
1 & _ & _ & _ & _ & 1 & N=5 \\
\end{array}
\]

\[C(5,2)\]

combo(5,2) regains control
And calls combo(4,3), which
Calls two ways, first of which can look up the answer in the map
Second way: calls combo 2 ways, both of which are immediately available.
Visualizing the DP process

 combo(5,2) is almost done: on The way back from the recursive Calls, it fills in two more Map entries.

```
1 1 2 1 N=2
1 3 3 1 N=3
1 4 6 _ 1 N=4
1 _ _ _ _ 1 N=5
```

C(5,2)
Visualizing the DP process

That sector is contained within \((i,j)\) where \(i \leq 2\) and \(j \leq 5\)

i.e. entries “below” \((5,2)\)

 combo\((5,2)\) is done, and returns The value 10.

It has visited 10 spots, and Filled in 5 entries in the map.

The region covered is \(k+1\) across And \(N\) down, so this is 
\(T(N) = O(Nk) \leq O(N^2)\)
Another Greedy Example: Interval scheduling

- Input: set of requests for periods of time (intervals) to use a special resource, like a supercomputer or a piano
- Task: Find maximum number of compatible time intervals out of this set. Two intervals are compatible if they don’t overlap.

Example (activities in each line are compatible):

```
[-----------------]  [-----------------]  [-----------------]  [-----------------]
[-----------------]  [-----------------]  [-----------------]  [-----------------]
[-----------------]  [-----------------]  [-----------------]  [-----------------]
 time
```
Greedy, but how? (discussion from Wikipedia)

- Several algorithms, that may look promising at first sight, actually do not find the optimal solution:
- Selecting the intervals that start earliest is not an optimal solution, because if the earliest interval happens to be very long, accepting it would make us reject many other shorter requests.
- Selecting the shortest intervals or selecting intervals with the fewest conflicts is also not optimal.
The right Greediness...

- The following greedy algorithm does find the optimal solution:
  - Select the interval, \( x \), with the earliest finishing time.
  - Remove \( x \), and all intervals intersecting \( x \), from the set of candidate intervals.
  - Repeat until the set of candidate intervals is empty.
Why this greedy algorithm works (proof)

• Whenever we select an interval at step 1 (green one in figure), we may have to remove many intervals in step 2 (red).
• However, all these intervals necessarily cross the finishing time of $x$, and thus they all cross each other (see figure).

• Hence, at most 1 of these intervals (green or red) can be in the optimal solution.
• Hence, for every interval in the optimal solution, there is an interval in the greedy solution.
• Hence, for every interval in the optimal solution (blue), there is an interval in the greedy solution (green, like money).

• That’s a subset relationship.

• But if the green set is actually bigger than the optimal, it would be better than the optimal solution, not possible.

• Thus greedy provides the optimal solution
Implementation for Interval Scheduling

The straightforward algorithm is

• Put all the intervals in a set \( S \) \( O(N) \)
• Loop until done: \( O(N) \) passes
  • Find the interval \( I \) with earliest endtime \( O(N) \)
  • Drop \( I \) and the intervals that overlap with \( I \) from the set \( O(N) \)
• So \( T(N) = O(N^2) \)
• Can we do better?
Faster Implementation for Interval Scheduling (from K&T)

• Since the finishing time is crucial here, we sort the original set of intervals by end-time, at cost $T(N) = N\log N$, well below the $O(N^2)$ we’re trying to beat. We can renumber them too, so the first-finishing appears first, etc.

• Then we iterate through the sequence:
  • Find the first interval, of end-time $E$, and select it into result set
  • Iterate through following intervals, all of which have end-time $\geq E$, skipping those with start-time $\leq E$, until see one with start-time $> E$, which we leave for the next pass.

• $T(N) = O(N)$ for this second part, so $T(N) = N\log N$ overall
K&T (Kleinberg & Tardos) Discussion

- Practical scheduling: we don’t always know all the requests when we have to make a choice among the ones we have. This harder problem is “online” scheduling.
- The requests can have weights (profits, say), and we want to maximize the total weight for the selected requests.
  - This problem is known as Weighted Interval Scheduling.
  - No greedy algorithm is known for this problem.
  - This is revisited in K&T Sec.6.1, where recursive search and dynamic programming are used, and the resulting algorithm is $T(N) = O(N)$ after the input is sorted by end-time.