Algorithmic Techniques

Peter Lammich

Greedy Algorithms: Example

- Change-Making Problem: How many coins are needed to make a change of a?

Greedy Algorithms: Example

- Change-Making Problem: How many coins are needed to make a change of a?
- Algorithm: Iteratively take highest fitting coin.
  - E.g. $£11 = 5 + 5 + 1$

Greedy Algorithms: Example

- Change-Making Problem: How many coins are needed to make a change of a?
- Algorithm: Iteratively take highest fitting coin.
  - E.g. $£11 = 5 + 5 + 1$
- Only works for *canonical* coin systems (real ones are!)

Greedy Algorithms: Example

- Change-Making Problem: How many coins are needed to make a change of a?
- Algorithm: Iteratively take highest fitting coin.
  - E.g. $£11 = 5 + 5 + 1$
- Only works for *canonical* coin systems (real ones are!)
  - Consider coin system: 1-3-4, make change of 6

Greedy Algorithms: Example

- Change-Making Problem: How many coins are needed to make a change of a?
- Algorithm: Iteratively take highest fitting coin.
  - E.g. $£11 = 5 + 5 + 1$
- Only works for *canonical* coin systems (real ones are!)
  - Consider coin system: 1-3-4, make change of 6
  - Greedy: $6 = 4 + 1 + 1$

Greedy Algorithms: Example

- Change-Making Problem: How many coins are needed to make a change of a?
- Algorithm: Iteratively take highest fitting coin.
  - E.g. $£11 = 5 + 5 + 1$
- Only works for *canonical* coin systems (real ones are!)
  - Consider coin system: 1-3-4, make change of 6
  - Greedy: $6 = 4 + 1 + 1$ Optimal: $6 = 3 + 3$

Properties required for Greedy Algorithms

- Optimal Substructure
    - problem can be solved by solving sub-problems

Properties required for Greedy Algorithms

- Optimal Substructure
  - problem can be solved by solving sub-problems
  - Change-Making: take highest fitting coin, then solve problem for remaining amount

Properties required for Greedy Algorithms

- Optimal Substructure
  - problem can be solved by solving sub-problems
  - Change-Making: take highest fitting coin, then solve problem for remaining amount
  - Dijkstra: shortest paths for increasing set of nodes

Properties required for Greedy Algorithms

- Optimal Substructure
    - problem can be solved by solving sub-problems
    - Change-Making: take highest fitting coin, then solve problem for remaining amount
    - Dijkstra: shortest paths for increasing set of nodes
- Greedy Choice: decision needs not be reconsidered

Properties required for Greedy Algorithms

- Optimal Substructure
  - problem can be solved by solving sub-problems
  - Change-Making: take highest fitting coin, then solve problem for remaining amount
  - Dijkstra: shortest paths for increasing set of nodes
- Greedy Choice: decision needs not be reconsidered
  - Change-Making: Never take back coins

Properties required for Greedy Algorithms

- Optimal Substructure
    - problem can be solved by solving sub-problems
    - Change-Making: take highest fitting coin, then solve problem for remaining amount
    - Dijkstra: shortest paths for increasing set of nodes
- Greedy Choice: decision needs not be reconsidered
    - Change-Making: Never take back coins
    - Dijkstra: relaxed node has precise estimate

Dynamic Programming: Example

- Consider (general) coin system. $w_1 < \ldots < w_n$, $w_1 = 1$

Dynamic Programming: Example

- Consider (general) coin system. $w_1 < \ldots < w_n$, $w_1 = 1$
- Generalize: $\#(i, a)$ number of coins from $w_1, \ldots, w_i$ required for amount a

Dynamic Programming: Example

- Consider (general) coin system. $w_1 < \ldots < w_n$, $w_1 = 1$
- Generalize: $\#(i, a)$ number of coins from $w_1, \ldots, w_i$ required for amount a
    - number of coins for amount a: $\#(n, a)$

Dynamic Programming: Example

- Consider (general) coin system. $w_1 < \ldots < w_n$, $w_1 = 1$
- Generalize: $\#(i, a)$ number of coins from $w_1, \ldots, w_i$ required for amount a
  - number of coins for amount a: $\#(n, a)$
- Optimal substructure: to compute $\#(i, a)$:

Dynamic Programming: Example

- Consider (general) coin system. $w_1 < \ldots < w_n$, $w_1 = 1$
- Generalize: $\#(i, a)$ number of coins from $w_1, \ldots, w_i$ required for amount a
    - number of coins for amount a: $\#(n, a)$
- Optimal substructure: to compute $\#(i, a)$:
    - either take another w_i coin: $1 + \#(i, a - w_i)$

Dynamic Programming: Example

- Consider (general) coin system. $w_1 < \ldots < w_n$, $w_1 = 1$
- Generalize: $\#(i, a)$ number of coins from $w_1, \ldots, w_i$ required for amount a
  - number of coins for amount a: $\#(n, a)$
- Optimal substructure: to compute $\#(i, a)$:
  - either take another w_i coin: $1 + \#(i, a - w_i)$
  - or take no more w_i coins: $\#(i - 1, a)$

Dynamic Programming: Example

- Consider (general) coin system. $w_1 < \ldots < w_n$, $w_1 = 1$
- Generalize: $\#(i, a)$ number of coins from $w_1, \ldots, w_i$ required for amount a
  - number of coins for amount a: $\#(n, a)$
- Optimal substructure: to compute $\#(i, a)$:
  - either take another w_i coin: $1 + \#(i, a - w_i)$
  - or take no more w_i coins: $\#(i - 1, a)$
  - solution is the smaller of the two cases

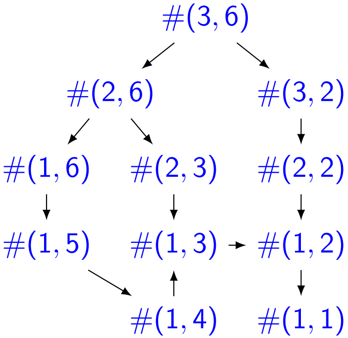
Dynamic Programming: Example

- Consider (general) coin system. $w_1 < \ldots < w_n$, $w_1 = 1$
- Generalize: $\#(i, a)$ number of coins from $w_1, \ldots, w_i$ required for amount a
  - number of coins for amount a: $\#(n, a)$
- Optimal substructure: to compute $\#(i, a)$:
  - either take another w_i coin: $1 + \#(i, a - w_i)$
  - or take no more w_i coins: $\#(i - 1, a)$
  - solution is the smaller of the two cases
  - edge cases: $i = 1$, $a = w_i$, $a < w_i$, $a = 0$

Overlapping Subproblems

$$\#(i, a) = \min(\#(i-1, a), 1 + \#(i, a - w_i)) \qquad \text{if } i > 1 \land a > w_i$$

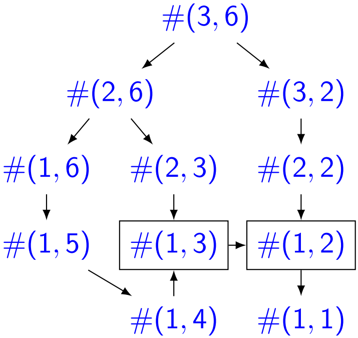
- Overlapping subproblems: Eg make 6 in coin system 1-3-4



Overlapping Subproblems

$$\#(i, a) = \min(\#(i-1, a), 1 + \#(i, a - w_i)) \qquad \text{if } i > 1 \wedge a > w_i$$

- Overlapping subproblems: Eg make 6 in coin system 1-3-4



Algorithm (Variant 1)

compute all subproblems in suitable order

```
procedure NUM($a$)
    if $a = 0$ then return $0$
    $r \leftarrow$ new array $n \times a$
    for $i = 1 \dots n$ do
        for $k = 1 \dots a$ do
            if $k = w_i$ then
                $r[i, k] \leftarrow 1$
            else if $k < w_i$ then
                $r[i, k] \leftarrow r[i - 1, k]$
            else if $i = 1$ then
                $r[i, k] \leftarrow 1 + r[i, k - w_1]$
            else
                $r[i, k] \leftarrow \min(r[i - 1, k], 1 + r[i, k - w_1])$
    return $r[n, a]$
```

Algorithm (Variant 2)

memorize already computed subproblems

```
global map $r \leftarrow$ empty map
procedure NUM_AUX($i, k$)
    if not defined $r[i, k]$ then
        if $k = w_i$ then
            $r[i, k] \leftarrow 1$
        else if $k < w_i$ then
            $r[i, k] \leftarrow \text{NUM\_AUX}(i - 1, k)$
        else if $i = 1$ then
            $r[i, k] \leftarrow 1 + \text{NUM\_AUX}(i, k - w_1)$
        else
            $r[i, k] \leftarrow \min(\text{NUM\_AUX}(i - 1, k), 1 + \text{NUM\_AUX}(i, k - w_1))$
    return $r[i, k]$
procedure NUM($a$)
    if $a = 0$ then return $0$
    return NUM_AUX(n,a)
```

Properties Required for Dynamic Programming

- Optimal substructure
- Overlapping subproblems
    - otherwise, simple recursion would be sufficient!

Floyd-Warshall Algorithm

- Find shortest path between *all* pairs of nodes (APSP)

Floyd-Warshall Algorithm

- Find shortest path between *all* pairs of nodes (APSP)
- Let $d(u, v, X)$ be distance between u and v, but only over intermediate nodes from set X!

Floyd-Warshall Algorithm

- Find shortest path between *all* pairs of nodes (APSP)
- Let $d(u, v, X)$ be distance between u and v,
  but only over intermediate nodes from set X!
- If we add another node to X, a new shortest path either uses this node, or not:

$$d(u, v, \{w\} \cup X) = \min(d(u, w, X) + d(w, v, X), d(u, v, X))$$

Floyd-Warshall Algorithm

- Find shortest path between *all* pairs of nodes (APSP)
- Let $d(u, v, X)$ be distance between u and v, but only over intermediate nodes from set X!
- If we add another node to X, a new shortest path either uses this node, or not:

$$d(u, v, \{w\} \cup X) = \min(d(u, w, X) + d(w, v, X), d(u, v, X))$$

```
procedure FLOYD__WARSHALL
    $dist[u, v] \leftarrow w(u, v)$ for all nodes $u$,$v$
    $dist[v, v] \leftarrow 0$ for all nodes $v$
    for $w \in V$ do            // Compute $d(u, v, X)$ for increasing set $X$
        for $u \in V$ do
            for $v \in V$ do
                $dist[u, v] \leftarrow \min(dist[u, v], dist[u, w] + dist[w, v])$
```

Longest Common Subsequence

- Subsequence of word w: erase letters from w
    - E.g.: "Hllo wrld" is subsequence of "Hello world"

Longest Common Subsequence

- Subsequence of word *w*: erase letters from *w*
  - E.g.: "Hllo wrld" is subsequence of "Hello world"
- Given two strings, find (a) longest common subsequence (LCS)

Longest Common Subsequence

- Subsequence of word w: erase letters from w
    - E.g.: "Hllo wrld" is subsequence of "Hello world"
- Given two strings, find (a) longest common subsequence (LCS)
    - is LCS unique?

Longest Common Subsequence

- Subsequence of word *w*: erase letters from *w*
  - E.g.: "Hllo wrld" is subsequence of "Hello world"
- Given two strings, find (a) longest common subsequence (LCS)
  - is LCS unique? No! "abc", "cba",

Longest Common Subsequence

- Subsequence of word *w*: erase letters from *w*
  - E.g.: "Hllo wrld" is subsequence of "Hello world"
- Given two strings, find (a) longest common subsequence (LCS)
  - is LCS unique? No! "abc", "cba", LCS: "a", also "b" and "c"

Longest Common Subsequence

- Subsequence of word w: erase letters from w
  - E.g.: "Hllo wrld" is subsequence of "Hello world"
- Given two strings, find (a) longest common subsequence (LCS)
  - is LCS unique? No! "abc", "cba", LCS: "a", also "b" and "c"
- Application: diff - tool

w_1 `dist[i,j] = dist[k,i] + dist[j,k] + 1`

w_2 `dist[i,j] = dist[i,k] + dist[k,j]`

$lcs(w_1, w_2)$ `dist[i,j] = dist[,] + dist[,]`

Longest Common Subsequence

- Subsequence of word w: erase letters from w
    - E.g.: "Hllo wrld" is subsequence of "Hello world"
- Given two strings, find (a) longest common subsequence (LCS)
    - is LCS unique? No! "abc", "cba", LCS: "a", also "b" and "c"
- Application: diff - tool

w_1	`dist[i,j] = dist[k,i] + dist[j,k] + 1`
w_2	`dist[i,j] = dist[i,k] + dist[k,j]`
$lcs(w_1, w_2)$	`dist[i,j] = dist[,] + dist[,]`

- letters in w_1 but not LCS: removed

Longest Common Subsequence

- Subsequence of word w: erase letters from w
  - E.g.: "Hllo wrld" is subsequence of "Hello world"
- Given two strings, find (a) longest common subsequence (LCS)
  - is LCS unique? No! "abc", "cba", LCS: "a", also "b" and "c"
- Application: diff - tool

  w_1 `dist[i,j] = dist[k,i] + dist[j,k] + 1`

  w_2 `dist[i,j] = dist[i,k] + dist[k,j]`

  $lcs(w_1, w_2)$ `dist[i,j] = dist[,] + dist[,]`

  - letters in w_1 but not LCS: removed
  - letters in w_2 but not LCS: added

LCS with Dynamic Programming

Optimal substructure

$$\begin{aligned} lcs(w_1x, w_2y) =& lcs(w_1, w_2)x && \text{if } x = y \\ =& \max(lcs(w_1x, w_2), lcs(w_1, w_2y)) && \text{otherwise} \end{aligned}$$

- max longer of the two sequences. Any if equal length.
- Idea: cases if last letter of each word is part of LCS

Cocke-Younger-Kasami Algorithm

- Recall: CF-Grammar in Chomsky Normal Form

Cocke-Younger-Kasami Algorithm

- Recall: CF-Grammar in Chomsky Normal Form
  - productions of form: $A \to \alpha$, $A \to BC$

Cocke-Younger-Kasami Algorithm

- Recall: CF-Grammar in Chomsky Normal Form
  - productions of form: $A \to \alpha$, $A \to BC$
  - every CF-grammar that does not accept ε has Chomsky-NF

Cocke-Younger-Kasami Algorithm

- Recall: CF-Grammar in Chomsky Normal Form
  - productions of form: $A \to \alpha$, $A \to BC$
  - every CF-grammar that does not accept ε has Chomsky-NF
- Problem: can word w be produced from nonterminal N ($N \to^* w$)

Cocke-Younger-Kasami Algorithm

- Recall: CF-Grammar in Chomsky Normal Form
  - productions of form: $A \to \alpha$, $A \to BC$
  - every CF-grammar that does not accept ε has Chomsky-NF
- Problem: can word w be produced from nonterminal N ($N \to^* w$)
  - $P(N, i, l)$ true iff $N \to w[i \ldots < i + l]$

Cocke-Younger-Kasami Algorithm

- Recall: CF-Grammar in Chomsky Normal Form
  - productions of form: $A \to \alpha$, $A \to BC$
  - every CF-grammar that does not accept ε has Chomsky-NF
- Problem: can word w be produced from nonterminal N ($N \to^* w$)
  - $P(N, i, l)$ true iff $N \to w[i \ldots < i + l]$
  - $P(N, i, 1)$ iff exists production $N \to w[i]$

Cocke-Younger-Kasami Algorithm

- Recall: CF-Grammar in Chomsky Normal Form
  - productions of form: $A \to \alpha$, $A \to BC$
  - every CF-grammar that does not accept ε has Chomsky-NF
- Problem: can word w be produced from nonterminal N ($N \to^* w$)
  - $P(N, i, l)$ true iff $N \to w[i \ldots < i + l]$
  - $P(N, i, 1)$ iff exists production $N \to w[i]$
  - for $l > 1$: $P(N, i, l)$ iff
    exists $l_1, l_2 \geq 1$ with $l = l_1 + l_2$ and production $N \to AB$
    such that: $P(A, i, l_1)$ and $P(B, i + l_1, l_2)$

Cocke-Younger-Kasami Algorithm

- Recall: CF-Grammar in Chomsky Normal Form
  - productions of form: $A \rightarrow \alpha$, $A \rightarrow BC$
  - every CF-grammar that does not accept ε has Chomsky-NF
- Problem: can word w be produced from nonterminal N ($N \rightarrow^* w$)
  - $P(N, i, l)$ true iff $N \rightarrow w[i \ldots < i + l]$
  - $P(N, i, 1)$ iff exists production $N \rightarrow w[i]$
  - for $l > 1$: $P(N, i, l)$ iff
    exists $l_1, l_2 \geq 1$ with $l = l_1 + l_2$ and production $N \rightarrow AB$
    such that: $P(A, i, l_1)$ and $P(B, i + l_1, l_2)$
- Compute P by iterating over lengths, start indices, splits, productions

Cocke-Younger-Kasami Algorithm

- Recall: CF-Grammar in Chomsky Normal Form
  - productions of form: $A \rightarrow \alpha$, $A \rightarrow BC$
  - every CF-grammar that does not accept ε has Chomsky-NF
- Problem: can word w be produced from nonterminal N ($N \rightarrow^* w$)
  - $P(N, i, l)$ true iff $N \rightarrow w[i \ldots < i + l]$
  - $P(N, i, 1)$ iff exists production $N \rightarrow w[i]$
  - for $l > 1$: $P(N, i, l)$ iff
    exists $l_1, l_2 \geq 1$ with $l = l_1 + l_2$ and production $N \rightarrow AB$
    such that: $P(A, i, l_1)$ and $P(B, i + l_1, l_2)$
- Compute P by iterating over lengths, start indices, splits, productions
  - yields $O(|w|^3 * n)$ algorithm, for grammar with n productions

Cocke-Younger-Kasami Algorithm

- Recall: CF-Grammar in Chomsky Normal Form
  - productions of form: $A \to \alpha$, $A \to BC$
  - every CF-grammar that does not accept ε has Chomsky-NF
- Problem: can word w be produced from nonterminal N ($N \to^* w$)
  - $P(N, i, l)$ true iff $N \to w[i \ldots < i + l]$
  - $P(N, i, 1)$ iff exists production $N \to w[i]$
  - for $l > 1$: $P(N, i, l)$ iff
    exists $l_1, l_2 \geq 1$ with $l = l_1 + l_2$ and production $N \to AB$
    such that: $P(A, i, l_1)$ and $P(B, i + l_1, l_2)$
- Compute P by iterating over lengths, start indices, splits, productions
  - yields $O(|w|^3 * n)$ algorithm, for grammar with n productions
  - good for general CF-grammars.

Cocke-Younger-Kasami Algorithm

- Recall: CF-Grammar in Chomsky Normal Form
    - productions of form: $A \to \alpha$, $A \to BC$
    - every CF-grammar that does not accept ε has Chomsky-NF
- Problem: can word w be produced from nonterminal N ($N \to^* w$)
    - $P(N, i, l)$ true iff $N \to w[i \ldots < i + l]$
    - $P(N, i, 1)$ iff exists production $N \to w[i]$
    - for $l > 1$: $P(N, i, l)$ iff
      exists $l_1, l_2 \geq 1$ with $l = l_1 + l_2$ and production $N \to AB$
      such that: $P(A, i, l_1)$ and $P(B, i + l_1, l_2)$
- Compute P by iterating over lengths, start indices, splits, productions
    - yields $O(|w|^3 * n)$ algorithm, for grammar with n productions
    - good for general CF-grammars.
    - much better algorithms for special grammars, like computer languages!

Conclusion

- Optimal Substructure
  - solve larger instance by smaller instances
  - often requires generalization, e.g.
    - only use coins $1 \dots i$
    - only use paths over certain intermediate nodes
- Greedy Choice: distinct subproblems, no need to backtrack
- Overlapping Subproblems: memorize already computed solutions