Counting Combinations

To choose \( r \) things out of \( n \), either
- Choose the first item. Then we must choose the remaining \( r - 1 \) items from the other \( n - 1 \) items. Or
- Don’t choose the first item. Then we must choose the \( r \) items from the other \( n - 1 \) items.

Therefore,

\[
\binom{n}{r} = \binom{n - 1}{r - 1} + \binom{n - 1}{r}
\]

Then,

\[
T(n) = \begin{cases} 
  c & \text{if } n = 1 \\
  2T(n - 1) + d & \text{otherwise}
\end{cases}
\]

for some constants \( c, d \).

Hence,

\[
T(n) = 2T(n - 1) + d = 2(2T(n - 2) + d) + d = 4T(n - 2) + 2d + d = 4(2T(n - 3) + d) + 2 + d = 8T(n - 3) + 4d + 2d + d = 2^{i}T(n - i) + d \sum_{j=0}^{i-1} 2j = 2^{n-1}T(1) + d \sum_{j=0}^{n-2} 2j = (c + d)2^{n-1} - d
\]

Hence, \( T(n) = \Theta(2^n) \).
Example

The problem is, the algorithm solves the same subproblems over and over again!

A Better Algorithm

Pascal’s Triangle. Use a table T[0..n, 0..r].

\[ T[i, j] \text{ holds } \binom{i}{j}. \]

function choose(n, r)
  for i := 0 to n - r do T[i, 0] := 1;
  for i := 0 to r do T[i, i] := 1;
  for j := 1 to r do
    for i := j + 1 to n - r + j do
      T[i, j] := T[i - 1, j - 1] + T[i - 1, j]
  return(T[n, r])

Initialization

Required answer
General Rule

To fill in $T[i, j]$, we need $T[i - 1, j - 1]$ and $T[i - 1, j]$ to be already filled in.

Example

Analysis

How many table entries are filled in?

$$(n-r+1)(r+1) = nr + n - r^2 + 1 \leq n(r+1) + 1$$

Each entry takes time $O(1)$, so total time required is $O(n^2)$.

This is much better than $O(2^n)$.

Space: naive, $O(nr)$. Smart, $O(r)$. 

Filling in the Table

Fill in the columns from left to right. Fill in each of the columns from top to bottom.

Numbers show the order in which the entries are filled in.
Dynamic Programming

When divide and conquer generates a large number of identical subproblems, recursion is too expensive.

Instead, store solutions to subproblems in a table.

This technique is called dynamic programming.

Divide and Conquer

function choose(n,r)
if r=0 or r=n then return(1) else
    return(choose(n-1,r-1)+choose(n-1,r))

Dynamic Programming

function choose(n,r)
for i:=0 to n-r do T[i,0]:=1
for i:=0 to r do T[i,i]:=1
for j:=1 to r do
    for i:=j+1 to n-r+j do
        T[i,j]:=T[i-1,j-1]+T[i-1,j]
return(T[n,r])

Dynamic Programming Technique

To design a dynamic programming algorithm:

Identification:
- devise divide-and-conquer algorithm
- analyze — running time is exponential
- same subproblems solved many times

Construction:
- take part of divide-and-conquer algorithm
  that does the “conquer” part and replace recursive calls with table lookups
- instead of returning a value, record it in a table entry
- use base of divide-and-conquer to fill in start of table
- devise “look-up template”
- devise for-loops that fill the table using “look-up template”

The Knapsack Problem

Given n items of length s_1, s_2, ..., s_n, is there a subset of these items with total length exactly S?

∑_{i=1}^{n} s_i = S
Divide and Conquer

Want knapsack\((i, j)\) to return \textbf{true} if there is a subset of the first \(i\) items that has total length exactly \(j\).

\[
s_1 \quad s_2 \quad s_3 \quad s_4 \quad \cdots \quad s_{i-1} \quad s_i
\]

\[
\text{\textcolor{black}{\rule{6cm}{.5mm}}}
\]

\[
\text{\textcolor{black}{\rule{6cm}{.5mm}}}
\]

When can \textbf{knapsack}\((i, j)\) return \textbf{true}? Either the \(i\)th item is used, or it is not.

\[
\text{\textcolor{black}{\rule{6cm}{.5mm}}}
\]

The Code

Call \textbf{knapsack}\((n, S)\).

\[
\textbf{function knapsack}(i, j) \\
\textbf{comment} \text{ returns true if } s_1, \ldots, s_i \text{ can fill } j \\
\text{ if } i = 0 \text{ then return}(\text{false}) \\
\text{ else if } \text{knapsack}(i - 1, j) \text{ then return}(\text{true}) \\
\text{ else if } s_i \leq j \text{ then} \\
\text{ return}(\text{knapsack}(i - 1, j - s_i))
\]

Let \(T(n)\) be the running time of \textbf{knapsack}\((n, S)\).

\[
T(n) = \begin{cases} 
  c & \text{if } n = 1 \\
  2T(n - 1) + d & \text{otherwise}
\end{cases}
\]

Hence, by standard techniques, \(T(n) = \Theta(2^n)\).

Dynamic Programming

Store \textbf{knapsack}\((i, j)\) in table \(t[i, j]\).

\(t[i, j]\) is set to \textbf{true} iff either:
- \(t[i - 1, j]\) is \textbf{true}, or
- \(t[i - 1, j - s_i]\) makes sense and is \textbf{true}.

This is done with the following code:

\[
t[i, j] := t[i - 1, j] \\
\text{if } j - s_i \geq 0 \text{ then} \\
\begin{align*}
  t[i, j] &:= t[i, j] \text{ or } t[i - 1, j - s_i] \\
\end{align*}
\]

\[
\text{\textcolor{black}{\rule{6cm}{.5mm}}}
\]

\[
\text{\textcolor{black}{\rule{6cm}{.5mm}}}
\]
The Algorithm

function knapsack(s₁, s₂, ..., sₙ, S)
1. t[0, 0] := true
2. for j := 1 to S do t[0, j] := false
3. for i := 1 to n do
4.     for j := 0 to S do
5.         t[i, j] := t[i - 1, j]
6.         if j - sᵢ ≥ 0 then
7.             t[i, j] := t[i, j] or t[i - 1, j - sᵢ]
8.     return(t[n, S])

Analysis:
- Lines 1 and 8 cost \( O(1) \).
- The for-loop on line 2 costs \( O(S) \).
- Lines 5–7 cost \( O(1) \).
- The for-loop on lines 4–7 costs \( O(S) \).
- The for-loop on lines 3–7 costs \( O(nS) \).

Therefore, the algorithm runs in time \( O(nS) \).
This is usable if \( S \) is small.

Example

\( s₁ = 1, s₂ = 2, s₃ = 2, s₄ = 4, s₅ = 5, s₆ = 2, \\
\quad s₇ = 4, S = 15. \)

\[
\begin{align*}
  t[i, j] &:= t[i - 1, j] \text{ or } t[i - 1, j - sᵢ] \\
  t[3, 3] &:= t[2, 3] \text{ or } t[2, 3 - s₃] \\
  t[3, 3] &:= t[2, 3] \text{ or } t[2, 1] \\
\end{align*}
\]
### Assigned Reading

CLR Section 16.2.

POA Section 8.2.

---

**Question:** Can we get by with 2 rows?

**Question:** Can we get by with 1 row?