Dynamic Programming

CSE 2320 – Algorithms and Data Structures
University of Texas at Arlington

Alexandra Stefan
(Includes images, formulas and examples from CLRS, Dr. Bob Weems, wikipedia)
Approaches for solving DP Problems

**Greedy**
- Typically not optimal solution (for DP-type problems)
- Build solution
- Use a criterion for picking
- Commit to a choice and do not look back

**Brute Force**
- Optimal solution
- Produce all possible combinations, [check if valid], and keep the best.
- Time: exponential
- Space: depends on implementation
- It may be hard to generate all possible combinations

**DP**
- Optimal solution
- Write math function, sol, that captures the dependency of solution to current pb on solutions to smaller problems
- Can be implemented in any of the following: iterative, memoized, recursive

**Iterative (bottom-up) - BEST**
- Optimal solution
- sol is an array (1D or 2D). Size: n+1
- Fill in sol from 0 to n
- Time: polynomial (or pseudo-polynomial for some problems)
- Space: polynomial (or pseudo-polynomial)
- To recover the choices that gave the optimal answer, must backtrace => must keep picked array (1D or 2D).

**Memoized**
- Optimal solution
- Combines recursion and usage of sol array.
- sol is an array (1D or 2D)
- Fill in sol from 0 to n
- Time: same as iterative version (typically)
- Space: same as iterative version (typically) + space for frame stack. (Frame stack depth is typically smaller than the size of the sol array)

**Recursive**
- Optimal solution
- Time: exponential (typically) => DO NOT USE
- Space: depends on implementation (code). E.g. store all combinations, or generate, evaluate on the fly and keep best seen so far.
- Easy to code given math function

**Sliding window**
- Improves the iterative solution
- Saves space
- If used, cannot recover the choices (gives the optimal value, but not the choices)

DP can solve:
- Some types of **counting problems** (e.g. stair climbing)
- Some type of **optimization problems** (e.g. Knapsack)
- Some type of **recursively defined** pbs (e.g. Fibonacci)

Some DP solutions have **pseudo polynomial** time
Dynamic Programming (DP) - CLRS

• Dynamic programming (DP) applies when a problem has both of these properties:

1. **Optimal substructure**: “optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently”.

2. **Overlapping subproblems**: “a recursive algorithm revisits the same problem repeatedly”.

• Dynamic programming is typically used to:
  – Solve optimization problems that have the above properties.
  – Solve counting problems – e.g. Stair Climbing or Matrix Traversal.
  – Speed up existing recursive implementations of problems that have overlapping subproblems (property 2) – e.g. Fibonacci.

• Compare **dynamic programming** with **divide and conquer**.
Bottom-Up vs. Top Down

• There are two versions of dynamic programming.
  – Bottom-up.
  – Top-down (or memoization).

• Bottom-up:
  – Iterative, solves problems in sequence, from smaller to bigger.

• Top-down:
  – Recursive, start from the larger problem, solve smaller problems as needed.
  – For any problem that we solve, store the solution, so we never have to compute the same solution twice.
  – This approach is also called memoization.
Top-Down Dynamic Programming (Memoization)

• Maintain an array/table where solutions to problems can be saved.

• To solve a problem P:
  – See if the solution has already been stored in the array.
  – If yes, return the solution.
  – Else:
    • Issue recursive calls to solve whatever smaller problems we need to solve.
    • Using those solutions obtain the solution to problem P.
    • Store the solution in the solutions array.
    • Return the solution.
Requirements for using dynamic programming:

- The answer to our problem, \( P \), can be easily obtained from answers to smaller problems.
- We can order problems in a sequence \( (P_0, P_1, P_2, \ldots, P_K) \) of reasonable size, so that:
  - \( P_k \) is our original problem \( P \).
  - The initial problems, \( P_0 \) and possibly \( P_1, P_2, \ldots, P_R \) up to some \( R \), are easy to solve (they are \textit{base cases}).
  - For \( i > R \), each \( P_i \) can be easily solved using solutions to \( P_0, \ldots, P_{i-1} \).
- If these requirements are met, we solve problem \( P \) as follows:
  - Create the sequence of problems \( P_0, P_1, P_2, \ldots, P_K \), such that \( P_k = P \).
  - For \( i = 0 \) to \( K \), solve \( P_k \).
  - Return solution for \( P_k \).
Steps  (Dr. Weems)

1. Identify problem input
2. Identify the cost/gain function (name it, describe it)
3. Give the math formula for the cost function for all cases: base cases and general case
4. Order the problems & solve them
5. Recover the choices that gave the optimal value

Other

1. Brute force solution
2. Recursive solution (most likely exponential and inefficient)
3. Memoized solution
Fibonacci Numbers
Fibonacci Numbers

• Generate Fibonacci numbers
  – 3 solutions: inefficient recursive, memoization (top-down dynamic programming (DP)), bottom-up DP.
  – Not an optimization problem but it has overlapping subproblems => DP eliminates recomputing the same problem over and over again.

• Weighted interval scheduling

• Matrix multiplication
Fibonacci Numbers

- $\text{Fibonacci}(0) = 0$
- $\text{Fibonacci}(1) = 1$
- If $N \geq 2$:
  \[
  \text{Fibonacci}(N) = \text{Fibonacci}(N-1) + \text{Fibonacci}(N-2)
  \]
- How can we write a function that computes Fibonacci numbers?
Fibonacci Numbers

• Fibonacci(0) = 0
• Fibonacci(1) = 1
• If N >= 2: Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
• Consider this function: what is its running time?

Notice the mapping/correspondence of the mathematical expression and code.

```c
int Fib(int i)
{
    if (i < 1) return 0;
    if (i == 1) return 1;
    return Fib(i-1) + Fib(i-2);
}
```
Fibonacci Numbers

- Fibonacci(0) = 0
- Fibonacci(1) = 1
- If N >= 2: Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
- Consider this function: what is its running time?
  - g(N) = g(N-1) + g(N-2) + constant
  - \( g(N) \geq \text{Fibonacci}(N) \Rightarrow g(N) = \Omega(\text{Fibonacci}(N)) \Rightarrow g(N) = \Omega(1.618^N) \)
  - Also \( g(N) \leq 2g(N-1)+\text{constant} \Rightarrow g(N) \leq c2^N \Rightarrow g(N) = O(2^N) \)
  - \( g(N) \) is exponential
  - We cannot compute Fibonacci(40) in a reasonable amount of time (with this implementation).
  - See how many times this function is executed.
  - Draw the tree

```c
int Fib(int i)
{
    if (i < 1) return 0;
    if (i == 1) return 1;
    return Fib(i-1) + Fib(i-2);
}
```
Fibonacci Numbers

- Fibonacci(0) = 0
- Fibonacci(1) = 1
- If N >= 2: Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
- Alternative to inefficient recursion: compute from small to large and store data in an array.

Notice the mapping/correspondence of the mathematical expression and code.

**linear version (Iterative, bottom-up):**

```c
int Fib_iter (int i) {
    int F[i+1];
    F[0] = 0;  F[1] = 1;
    int k;
    for (k = 2; k <= i; k++) F[k] = F[k-1] + F[k-2];
    return F[i];
}
```

**exponential version:**

```c
int Fib(int i) {
    if (i < 1) return 0;
    if (i == 1) return 1;
    return Fib(i-1) + Fib(i-2);
}
```
Applied scenario

• $F(N) = F(N-1) + F(N-2)$, $F(0) = 0$, $F(1) = 1$,

• Consider a webserver where clients can ask what the value of a certain Fibonacci number, $F(N)$ is, and the server answers it.

  How would you do that? (the back end, not the front end)
(Assume a uniform distribution of $F(N)$ requests over time most $F(N)$ will be asked.)

• Constraints:
  – Each loop iteration or function call costs you 1 cent.
  – Each loop iteration or function call costs the client 0.001 seconds wait time
  – Memory is cheap

• How would you charge for the service? (flat fee/function calls/loop iterations?)

• Think of some scenarios of requests that you could get. Think of it with focus on:
  – “good sequence of requests”
  – “bad sequence of requests”
  – Is it clear what good and bad refer to here?
Fibonacci Numbers

- Fibonacci(0) = 0, Fibonacci(1) = 1
- If N >= 2: Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
- Alternative: remember values we have already computed.
- Draw the new recursion tree and discuss time complexity.

**memoized version:**

```c
int Fib_mem_wrap(int i) {
    int sol[i+1];
    if (i<=1) return i;
    sol[0] = 0; sol[1] = 1;
    for(int k=2; k<=i; k++)  sol[k]=-1;
    Fib_mem(i,sol);
    return sol[i];
}
int Fib_mem (int i, int[] sol)  {
    if (sol[i]!=-1) return sol[i];
    int res = Fib_mem(i-1, sol) + Fib_mem(i-2, sol);
    sol[i] = res;
    return res;
}
```

**exponential version:**

```c
int Fib(int i)  {
    if (i < 1) return 0;
    if (i == 1) return 1;
    return Fib(i-1) + Fib(i-2);
}
```
Fibonacci and DP

• Computing the Fibonacci number is a DP problem.
• It is a counting problem (not an optimization one).
• We can make up an ‘applied’ problem for which the DP solution function is the Fibonacci function.
  Consider: A child can climb stairs one step at a time or two steps at a time (but he cannot do 3 or more steps at a time). How many different ways can they climb? E.g. to climb 4 stairs you have 5 ways: 
  \{1,1,1,1\}, \{2,1,1\}, \{1,2,1\}, \{1,1,2\}, \{2,2\}
The Knapsack Problem

Problem:
• A thief breaks into a store.
• The maximum total weight that he can carry is $W$.
• There are $N$ types of items at the store.
• Each type $t_i$ has a value $v_i$ and a weight $w_i$.
• What is the maximum total value that he can carry out?
• What items should he pick to obtain this maximum value?

Variations based on item availability:
• Unlimited amounts – Unbounded Knapsack
• Limited amounts – Bounded Knapsack
• Only one item – 0/1 Knapsack
• Items can be ‘cut’ – Continuous Knapsack
  (or Fractional Knapsack)

Image from Wikipedia: https://en.wikipedia.org/wiki/Knapsack_problem
Variations of the Knapsack Problem

**Unbounded:**
Have unlimited number of each object. Can pick any object, any number of times. (Same as the stair climbing with gain.)

**Bounded:**
Have a limited number of each object. Can pick object i, at most $x_i$ times.

**0-1** (special case of Bounded):
Have only one of each object. Can pick either pick object i, or not pick it. This is on the web.

**Fractional:**
For each item can take the whole quantity, or a fraction of the quantity.

All versions have:
- $N$: number of different types of objects
- $W$: the maximum capacity (kg)
- $v_1, v_2, ..., v_N$: Value for each object. ($$$)
- $w_1, w_1, ..., w_N$: Weight of each object. (kg)

The bounded version will have the amounts: $c_1, c_2, ..., c_N$ of each item.
Worksheet: Unbounded Knapsack

Max capacity: W=17

<table>
<thead>
<tr>
<th>Item type:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight (kg)</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Value ($$)</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>13</td>
<td>15</td>
</tr>
</tbody>
</table>

Math cost function:

\[ S(k) = 0, \quad \forall k < \min_{1 \leq i \leq n} w_i \quad \text{(base cases, no item fits)} \]

\[ S(k) = \max_{v_i \text{ s.t. } w_i \leq k} \{val_i + S(k - w_i)\} \quad (1 \leq i \leq n) \]

Where \( k = \text{current weight} = \text{current problem size} \)

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sol</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Picked</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A, 3, 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B, 4, 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C, 7,11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D,8,13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E, 9,15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rows A,B,C,D,E are used to compute the final solution, in Sol and Picked. They show your work.
### Answers: Unbounded Knapsack

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sol</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>13</td>
<td>15</td>
<td>15</td>
<td>17</td>
<td>19</td>
</tr>
<tr>
<td>Picked</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A, 3, 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B, 4, 6</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C, 7,11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>D,8,13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E,9,15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Red – optimal, underscore – value(money)
Unbounded Knapsack – recover the items

Find the items that give the optimal value. For example in the data below, what items will give me value 31 for a max weight of 22?

Note that the item values are different from those on the previous page. (They are from a different problem instance.)

<table>
<thead>
<tr>
<th>Item type:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight (kg)</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

ID of picked item
Unbounded Knapsack – recover the items

Find the items that give the optimal value. For example in the data below, what items will give me value 31 for a max weight of 22?

Note that the item values are different from those on the previous page. (They are from a different problem instance.)

<table>
<thead>
<tr>
<th>Item type:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight (kg)</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Answer: E, C, A, A
Warning!

Webpage: https://www.codeproject.com/Articles/706838/Bounded-Knapsack-Algorithm

The presented solution for the unbounded knapsack:

“The unbounded knapsack problem is fairly easy to solve:

• Determine the value-weight ratio of every item.
• Starting with the highest value-weight ratio item, place as many of this item as will fit into the sack.
• Move onto the next-highest value-weight item and repeat step 2 until the sack is full or there are no other items.”

What is described there is the Greedy version, which will NOT give the optimal answer for all problems.

• Try to verify what you read on the web:
  – Use trusted sites such as Wikipedia,
  – See if you get the same answer from several sources.
  – Check blog posts
Iterative Solution for Unbounded Knapsack

/* Assume arrays v and w store the item info starting at
index 1: first item has value v[1] and weight w[1] */
int knapsack(int W, int n, int * v, int * w){
    int sol[W+1]; int picked[W+1];
    sol[0] = 0;
    for(k=1; k<=W; k++) {
        mx = 0; choice = -1; // no item
        for(i=1; i<n; i++) {
            if (k>=w[i]) {
                with_i = v[i]+sol[k-w[i]];
                if (mx < with_i) {
                    mx = with_i;
                    choice = i;
                }
            }
        } // for i
        sol[k]=mx; picked[k] = choice;
    } // for k
    return sol[W];
} //Time: Θ(nW) [pseudo polynomial: store W in lgW bits] Space: Θ(W)

Math cost function:
\[ Sol(k) = 0, \quad \forall k < \min_{1 \leq i \leq n} w_i \] (base cases, no item fits)
\[ Sol(k) = \max_{\forall i, s.t. w_i \leq k} \{val_i + Sol(k - w_i)\} \] (1 ≤ i ≤ n)
Where \( k = \text{current weight} = \text{current problem size} \)
int knapsack(int W, int n, int * v, int * w){
    int sol[W+1];
    sol[0] = 0;  // value 0 (cannot carry anything)
    for(k=1; k<W; k++) sol[k] = -1; // negative, safe
    return = knap_helper(W, n, v, w, sol);
}

/* Assume arrays v and w store the item info starting at index 1: first item has value v[1] and weight w[1] */
int knap_helper(int k, int n, int * v, int * w, int* sol){
    if (sol[k] != -1) return sol[k]; // check if already computed
    mx = 0;
    for(i=1;i<n;i++) {
        if (k>=w[i]) {
            with_i = v[i]+ knap_helper(k-w[i], n, v, w, sol);
            if (mx < with_i) mx = with_i;
        } // for i
    } // for i
    sol[k]=mx; // remember to save the computation in sol !
    return mx;
} // Time: Θ(nW) [pseudo polynomial]   Space: Θ(W) (for sol and stack)
Recursive Solution

(OPTIONAL)

- We want to compute: knap(17).
- knap(17) will be the maximum of these five values:
  - val_A = 4 + knap(14)
  - val_B = 6 + knap(13)
  - val_C = 11 + knap(10)
  - val_D = 13 + knap(9)
  - val_E = 15 + knap(8)

Math cost function:
\[ Sol(k) = 0, \forall k < \min_{1\leq i\leq n} w_i \]  
(base cases, no item fits)
\[ Sol(k) = \max_{\text{vi.s.t.} w_i \leq k} \{val_i + Sol(k - w_i)\} \]

Where \( k \) = current weight = current problem size

<table>
<thead>
<tr>
<th>item type:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight:</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>value</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>13</td>
<td>15</td>
</tr>
</tbody>
</table>

```c
int knapsack(int W, int n, int * v, int * w){
    if (W == 0) return 0;
    int max_value = 0;
    int i;
    for (i = 1; i <= n; i++)  {
        if (W < w[i]) continue;
        int value = v[i] + knapsack(W-w[i],n,v,w);
        if (value > max_value) max_value = value;
    }
    return max_value;
}
```
Recursive Solution for Knapsack (OPTIONAL)

running time?
very slow
(exponential)

How can we make it faster?

```c
int knapsack(int W, int n, int * v, int * w){
    if (W == 0) return 0;
    int max_value = 0;
    int i;
    for (i = 1; i <= n; i++) {
        if (W < w[i]) continue;
        int value = v[i] + knapsack(W-w[i],n,v,w);
        if (value > max_value) max_value = value;
    }
    return max_value;
}
```
Draw the recursion tree (OPTIONAL)

- Inefficient recursion
- Memoized version
Time complexity

- Note that the recursive inefficient version is a **brute force solution** (it generates all sets with total weight less than the knapsack capacity)
  - B.c. it recalculates every smaller problem
  - Draw the recursion tree.

- Discuss the time complexity of all 3 methods.
  - Let \( W = \text{maxWeight} \), \( n = \text{number of different items} \)
  - **Bottom-up (iterative)**: \( O(W \times n) \)
    - Imagine case with an item of weight 1.
    - **Pseudo-polynomial** – polynomial in the **value** of one of the inputs: \( W \).
  - **Recursive with memoization**: \( O(W \times n) \)
    - Worst case tree height is \( W \), every problem size is at most solved once (as an internal node) every internal node will have at most \( n \) branches.
  - **Inefficient recursion**: \( O(n^W) \) (assume all items have weight 1)
Performance Comparison

- **Recursive version: (knapsack_recursive.c)**
  - Runs reasonably fast for max_weight <= 60.
  - Starts getting noticeably slower after that.
  - For max_weight = 75 I gave up waiting after 3 minutes.

- **Bottom-up version: (knapsack_bottom_up.c)**
  - Tried up to max_weight = 100 million.
  - No problems, very fast.
  - Took 4 seconds for max_weight = 100 million.

- **Top-down version: (knapsack_top_down.c)**
  - Very fast, but crashes around max_weight = 57,000.
  - The system cannot handle that many recursive function calls.
Worksheet: 0/1 Knapsack (not fractional)

Value using first i items:
\[ \text{Sol}[i][k] = \max\{\text{Sol}[i-1][k-w[i]] + v[i], \text{Sol}[i-1][k]\} \]

optimal solution (for a \textbf{smaller problem size}), excluding item i

optimal solution (for \textbf{this problem size}), excluding item i

Math cost function:
\[ \text{Sol}(0, k) = 0, \forall k \]
\[ \text{Sol}(i, k) = 0, \forall k \text{ s.t. } k < \min_{0 \leq t \leq i} w_t \]
\[ \text{Sol}(i, k) = \max\{\text{sol}(i - 1, k), v_i + \text{sol}(i - 1, k - w_i)\} \]

Where \( k \) = \textit{current weight} = \textit{current problem size}

| index | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|-------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| No item |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| A, 3, 4 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| B, 4, 6 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| C, 7,11 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| D,8,13  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| E, 9,15 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
**Answer: 0/1 Knapsack (not fractional)**

optimal solution (for a **smaller** problem size), excluding item i

optimal solution (for this problem size), excluding item i

Value using **first** \(i\) **items**:
\[
\text{Sol}[i] [k] = \max\{\text{Sol}[i-1] [k - w[i]] + v[i], \text{Sol}[i-1] [k]\}
\]

Math cost function:
\[
\text{Sol}(0, k) = 0, \forall k
\]
\[
\text{Sol}(i, k) = 0, \forall k \text{ s.t. } k < \min_{\forall 0 \leq t \leq i} w_t
\]
\[
\text{Sol}(i, k) = \max\{\text{sol}(i - 1, k), v_i + \text{sol}(i - 1, k - w_i)\}
\]

Where \(k = \text{current weight} = \text{current problem size}\)

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>No item</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>A, 3, 4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4*</td>
<td>4*</td>
<td>4*</td>
<td>4*</td>
<td>4*</td>
<td>4*</td>
<td>4*</td>
<td>4*</td>
<td>4*</td>
<td>4*</td>
<td>4*</td>
<td>4*</td>
<td>4*</td>
<td>4*</td>
</tr>
<tr>
<td>B, 4, 6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>6*</td>
<td>6*</td>
<td>6*</td>
<td>10*</td>
<td>10*</td>
<td>10*</td>
<td>10*</td>
<td>10*</td>
<td>10*</td>
<td>10*</td>
<td>10*</td>
<td>10*</td>
<td>10*</td>
</tr>
<tr>
<td>C, 7,11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>11*</td>
<td>11*</td>
<td>11*</td>
<td>15*</td>
<td>17*</td>
<td>17*</td>
<td>17*</td>
<td>17*</td>
<td>21*</td>
<td>21*</td>
</tr>
<tr>
<td>D,8,13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>11</td>
<td>13*</td>
<td>13*</td>
<td>15</td>
<td>17*</td>
<td>19*</td>
<td>19*</td>
<td>21</td>
<td>24*</td>
<td>24*</td>
</tr>
<tr>
<td>E, 9,15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>11</td>
<td>13</td>
<td>15*</td>
<td>15*</td>
<td>17</td>
<td>19*</td>
<td>21*</td>
<td>24</td>
<td>24</td>
<td>26*</td>
</tr>
</tbody>
</table>

**Iterative Solution for 0/1 Knapsack**

/* Assume arrays v and w store the item info starting at index 1: first item has value v[1] and weight w[1] */

```c
int knapsack01(int W, int n, int * v, int * w){
    int sol[n+1][W+1];
    for(k=0; k<=W; k++) { sol[0][k] = 0;}
    for(i=1; i<=n; i++) {
        for(k=0;k<=W;k++) {
            sol[i][k] = sol[i-1][k]; // solution without item i
            if (k>w[i]) {
                with_i = v[i]+sol[i-1][k-w[i]];
                if (sol[i][k] < with_i) {   // better choice
                    sol[i][k] = with_i;      // keep it
                }
            }
        } // for k
    } // for i
    return sol[n][W];
} // Time: $\Theta(nW)$  Space: $\Theta(nW)$ [pseudo polynomial]
```
Unbounded vs 0/1 Knapsack Solutions

• Unbounded (unlimited number of items)
  – Need only one (or two) 1D arrays: sol (and picked).
  – The other rows (one per item) are added to show the work that we do in order to figure out the answers that go in the table. There is NO NEED to store it.
  – Similar problem: Minimum Number of Coins for Change (solves a minimization, not a maximization problem): https://www.youtube.com/watch?v=Y0ZqKpToTic

• 0/1 (most web resources show this problem)
  – MUST HAVE one or two 2D tables, of size: (items+1) x (max_weight+1).
  – Each row (corresponding to an item) gives the solution to the problem using items from the beginning up to and including that row.
  – Whenever you look back to see the answer for a precomputed problem you look precisely on the row above because that gives a solution with the items in the rows above (excluding this item).
    • Unbounded knapsack can repeat the item => no need for sol excluding the current item => 1D
Sliding Window
(improves memory usage)

- Optimize the memory usage: store only smaller problems that are needed.
- NOTE: if a sliding window is used the choices cannot be recovered (i.e. cannot recover what items to pick to achieve the computed optimal value).
- Unbounded: the sliding window size is the max of the items weights => \( \Theta(\max_i(w_i)) \)
- 0/1: the sliding window is 2 rows from the table => \( \Theta(W) \)
- Draw the sliding window arrays for the above problems.
- How do you update the arrays?
Hint for DP problems

• For a DP problem you can typically write a MATH function that gives the solution for problem of size $N$ in terms of smaller problems.

• It is straightforward to go from this math function to code:
  – Iterative: The math function ‘maps’ to the sol array
  – Recursive: The math function ‘maps’ to recursive calls

• Typically the math function will be a
  – Min/max (over itself applied to smaller $N$)
  – Sum (over itself applied to smaller $N$)
Weighted Interval Scheduling

(Job Scheduling)
Weighted Interval Scheduling (a.k.a. Job Scheduling)

Problem:
Given n jobs where each job has a start time, finish time and value, \((s_i, f_i, v_i)\) select a subset of them that do not overlap and give the largest total value.

Preprocessing:
- Sort jobs in increasing order of their finish time.
- For each job, \(i\), compute the last job prior to \(i\), \(p(i)\), that does not overlap with \(i\).

E.g.:
\[
\begin{align*}
\text{(start, end, value)} \\
(6, 8, \$2) \\
(2, 5, \$6) \\
(3, 11, \$5) \\
(5, 6, \$3) \\
(1, 4, \$5) \\
(4, 7, \$2)
\end{align*}
\]

After preprocessing:
\[
\begin{align*}
\text{JobId (start, end, value, p(i))} \\
1 (1, 4, \$5, ) \\
2 (2, 5, \$6, ) \\
3 (5, 6, \$3, ) \\
4 (4, 7, \$2, ) \\
5 (6, 8, \$2, ) \\
6 (3, 11, \$5, )
\end{align*}
\]
Weighted Interval Scheduling
(a.k.a. Job Scheduling)

Problem:
– Given n jobs where each job has a start time, finish time and value, \((s_i, f_i, v_i)\)
 select a subset of them that do not overlap and give the largest total value.

Preprocessing:
• Sort jobs in increasing order of their finish time. –already done here
• For each job \(i\), compute the last job prior to \(i\), \(p(i)\), that does not overlap with \(i\).

Solve the problem:
Steps: one step for each job.
Option: pick it or not
Smaller problems: 2:
\(pb1 = \text{jobs } 1\) to \(i-1\), \(=> \text{sol}(i-1)\)
\(pb2 = \text{jobs } 1\) to \(p(i)\) (where \(p(i)\) is the last job before \(i\) that does not overlap with \(i\). \(=> \text{sol}(p(i))\)

Solution function:
\[
\text{sol}(0) = 0 \\
\text{sol}(i) = \max\{\text{sol}(i - 1), v(i) + \text{sol}(p(i))\}
\]

Time complexity: \(O(n)\) (Fill out \(\text{sol}(i)\) in constant time for each \(i\))
Solve the problem:
Steps: one step for each job.
Option: pick it or not  (pick job i or not pick it)
Smaller problems: 2:
  pb1 = jobs 1 to i-1,   =>   sol(i-1)
  pb2 = jobs 1 to p(i)   (where p(i) is the last job before i
  that does not overlap with i.   =>   sol(p(i))
Solution function:
  sol(0) = 0
  sol(i) = max{sol(i - 1), v(i) + sol(p(i))}

Time complexity: ______

<table>
<thead>
<tr>
<th>i</th>
<th>v_i</th>
<th>p_i</th>
<th>sol(i)</th>
<th>sol(i) used i</th>
<th>In optimal solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Optimal value: ____, jobs picked to get this value: __________
Solve the problem:
Steps: one step for each job.
Option: pick it or not  (pick job i or not pick it)
Smaller problems: 2:
    pb1 = jobs 1 to i-1,  =>  sol(i-1)
    pb2 = jobs 1 to p(i)   (where p(i) is the last job before i  
                     that does not overlap with i.  =>  sol(p(i))
Solution function:
    sol(0) = 0
    sol(i) = \max\{sol(i - 1), v(i) + sol(p(i))\}

Time complexity: O(n)
  ( Fill in sol(i) in constant time for each i )

<table>
<thead>
<tr>
<th>i</th>
<th>v_i</th>
<th>p_i</th>
<th>sol(i)</th>
<th>sol(i) used i</th>
<th>In optimal solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>0</td>
<td>5 = max{0, 5+0}</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>0</td>
<td>6 = max{5, 6+0}</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>9 = max{6, 3+6}</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>9 = max{9, 2+5}</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3</td>
<td>11 = max{9, 2+9}</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>0</td>
<td>11 = max{11, 5+0}</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

Optimal value: 11, jobs picked to get this value: 2,3,5
Another example

• Notations conventions:
  – Jobs are already sorted by end time
  – Horizontal alignment is based on time. **In this example, only consecutive jobs overlap**, (e.g. jobs 1 and 3 do not overlap).

<table>
<thead>
<tr>
<th>Job (ID)</th>
<th>Job value</th>
<th>E.g.:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>(Job, start, end, value)</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>(1, 3pm, 5pm, 2$)</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>(2, 4pm, 6pm, 3$)</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>(3, 5pm, 7pm, 2$)</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>(4, 6pm, 8pm, 4$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(5, 7pm, 9pm, 2$)</td>
</tr>
</tbody>
</table>

Time complexity: $O(n)$
Recovering the Solution

• Example showing that when computing the optimal gain, we cannot decide which jobs will be part of the solution and which will not. We can only recover the jobs picked AFTER we computed the optimum gain and by going from end to start.

<table>
<thead>
<tr>
<th>i</th>
<th>v_i</th>
<th>p_i</th>
<th>sol(i)</th>
<th>sol(i) used i</th>
<th>In optimal solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>No</td>
<td>-</td>
</tr>
</tbody>
</table>

Time complexity: $O(n)$
Job Scheduling – Brute Force Solution

- For each job we have the option to include it (0) or not (1). Gives:
  - The power set for a set of 5 elements, or
  - All possible permutations with repetitions over n positions with values 0 or 1 => $O(2^n)$
  - Note: exclude sets with overlapping jobs.
- Time complexity: $O(2^n)$

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Valid</th>
<th>Total value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>yes</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>yes</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>yes</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>yes</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>yes</td>
<td>4 (=2+2)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>
Bottom-up (BEST)

Math function:
\[\text{sol}(0) = 0\]
\[\text{sol}(i) = \max\{\text{sol}(i - 1), v(i) + \text{sol}(p(i))\}\]

// Bottom-up (the most efficient solution)
int js_iter(int* v, int*p, int n){
    int j, with_j, without_j;
    int sol[n+1];
    // optionally, may initialize it to -1 for safety
    sol[0] = 0;
    for(j = 1; j <= n; j++){
        with_j = v[j] + sol[p[j]];
        without_j = sol[j-1];
        if ( with_j >= without_j)
            sol[j] = with_j;
        else
            sol[j] = without_j;
    }
    return sol[n];
}
Recursive (inefficient)

Math function:
\[
\begin{align*}
sol(0) &= 0 \\
sol(i) &= \max\{sol(i-1), v(i) + sol(p(i))\}
\end{align*}
\]

// Inefficient recursive solution:
int jsr(int* v, int*p, int n){
    if (n == 0) return 0;
    int res;
    int with_n = v[n] + jsr(v,p,p[n]);
    int without_n = jsr(v,p,n-1);
    if ( with_n >= without_n)
        res = with_n;
    else
        res = without_n;
    return res;
}
Memoization (Recursion combined with saving)

// Memoization efficient recursive solution:
int jsm(int* v, int*p, int n, int* sol){
    if (sol[n] != -1) // already computed.
        return sol[n]; // Used when rec call for a smaller problem.
    int res;
    int with_n = v[n] + jsm(v,p,p[n],sol);
    int without_n = jsm(v,p,n-1,sol);
    if (with_n >= without_n) res = with_n;
    else res = without_n;
    sol[n] = res;
    return res;
}

int jsr_out(int* v, int*p, int n){
    int sol[n+1];
    int j;
    sol[0] = 0;
    for (j = 1; j<= n; j++) sol[j] = -1; //not computed
    jsm(v,p,n,sol);
    return sol[n];
}

Math function:
sol(0) = 0
sol(i) = max{sol(i − 1), v(i) + sol(p(i))}
Function call tree for the memoized version

<table>
<thead>
<tr>
<th>Job i</th>
<th>p(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
</tr>
</tbody>
</table>

Round nodes – internal nodes. Require recursive calls. Square nodes – leaves, show calls that return without any new recursive calls.

To estimate the number of method calls note that every problem size is an internal node only once and that every node has exactly 0 or 2 children. A property of such trees states that the number of leaves is one more than the number of internal nodes => there are at most \((1+2N)\) calls. Here: \(N = 10\) jobs to schedule.
2D Matrix Traversal

P1. All possible ways to traverse a 2D matrix.
   – Start from top left corner and reach bottom right corner.
   – You can only move: 1 step to the right or one step down at a time. (No diagonal moves).
   – Variation: Allow to move in the diagonal direction as well.
   – Variation: Add obstacles (cannot travel through certain cells).

P2. Add fish of various gains. Take path that gives the most gain.
   – Variation: Add obstacles.
Longest Common Subsequence (LCS)
Longest Common Subsequence (LCS)

• Given 2 sequences, find the longest common subsequence (LCS)

• Example:
  – A B C B D A B
  – B D C A B A

• Examples of subsequences of the above sequences:
  – BCBA (length 4)
  – BDAB
  – CBA (length 3)
  – CAB
  – BB (length 2)

Show the components of the solution. Can you show a solution similar that of an Edit distance problem?
LCS
Smaller Problems

• Original problem:
  A B C B D A B
  B D C A B A

• Smaller problems:

• Smaller problems that can be base cases:
<table>
<thead>
<tr>
<th>Original problem (LCS length)</th>
<th>A B C B D A B B D C A B A (4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smaller problems (LCS length)</td>
<td>&quot;ABCB&quot; &quot;BD&quot; (1) &quot;AB&quot; &quot;DC&quot; (0)</td>
</tr>
</tbody>
</table>
| Smaller problems that can be base cases (LCS length) | "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" "" "" "" (0) "" 

""ACBDAB"
Dependence on Subproblems (recursive case)

$c(i,j)$ – depends on $c(i-1,j-1), c(i-1,j), c(i,j-1)$ (grayed areas show solved subproblems)

The function below clearly shows the dependence on the smaller problems and that the optimal value of all possibilities is kept. I would use this one!

$$c(i, j) = \begin{cases} 0, & i = 0 \text{ or } j = 0 \\ \max\{c(i-1,j), c(i,j-1), c(i-1,j-1) + 1\}, & x_i = y_j, i, j > 0 \\ \max\{c(i-1,j), c(i,j-1)\}, & x_i \neq y_j, i, j > 0 \end{cases}$$

This case makes the solution grow (finds an element of the subsequence)

$c(i-1,j-1) + 1$, if $x_i = y_j$

$x_i$ is ignored

$c(i,j)$

$y_j$ is ignored

Here indexes start from 1
Longest Common Subsequence

$$c(i,j) = \begin{cases} 
0, & i = 0 \text{ or } j = 0 \\
c(i-1, j-1) + 1, & x_i = y_j, i,j > 0 \\
\max\{c(i-1, j), c(i, j-1)\}, & x_i \neq y_j, i,j > 0 
\end{cases}$$
Iterative solution

\[
c(i, j) = \begin{cases} 
0, & \text{if } i = 0 \text{ or } j = 0 \\
c(i - 1, j - 1) + 1, & \text{if } x_i = y_j, i, j > 0 \\
\max\{c(i - 1, j), c(i, j - 1)\}, & \text{if } x_i \neq y_j, i, j > 0 
\end{cases}
\]

LCS-LENGTH(X, Y)
1. \( m = X.length \)
2. \( n = Y.length \)
3. let \( b[1..m, 1..n] \) and \( c[0..m, 0..n] \) be new tables
4. for \( i = 1 \) to \( m \)
5. \hspace{1em} \( c[i, 0] = 0 \)
6. for \( j = 0 \) to \( n \)
7. \hspace{1em} \( c[0, j] = 0 \)
8. for \( i = 1 \) to \( m \)
9. \hspace{2em} for \( j = 1 \) to \( n \)
10. \hspace{3em} if \( x_i = y_j \)
11. \hspace{4em} \( c[i, j] = c[i - 1, j - 1] + 1 \)
12. \hspace{4em} \( b[i, j] = "\downarrow" \)
13. \hspace{3em} elseif \( c[i - 1, j] \geq c[i, j - 1] \)
14. \hspace{4em} \( c[i, j] = c[i - 1, j] \)
15. \hspace{4em} \( b[i, j] = "\uparrow" \)
16. \hspace{3em} else \( c[i, j] = c[i, j - 1] \)
17. \hspace{4em} \( b[i, j] = "\leftarrow" \)
18. return \( c \) and \( b \)
Recover the subsequence

CLRS pseudocode

PRINT-LCS(b, X, i, j)
1   if i == 0 or j == 0
2       return
3   if b[i, j] == “\n”
4       PRINT-LCS(b, X, i - 1, j - 1)
5       print x_i
6 elseif b[i, j] == “↑”
7       PRINT-LCS(b, X, i - 1, j)
8 else PRINT-LCS(b, X, i, j - 1)
Longest Increasing Subsequence (LIS)
Longest Increasing Subsequence

Given an array of values, find the longest increasing subsequence.
Example: A = { 3,5,3,9,3,4,3,1,2,1,4}

Variations:
- Repetitions NOT allowed: strictly increasing subsequence. E.g.: 3,5,9
- Repetitions allowed: increasing subsequence. E.g.: 3,3,3,4,4

Simple solution: reduce to LCS problem.
For a more efficient solution tailored for the LIS problem see Dr. Weems notes.

<table>
<thead>
<tr>
<th>No repetitions</th>
<th>With repetitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution for given instance:</td>
<td>1,2,4 (also: or 3,5,9)</td>
</tr>
<tr>
<td>Solution concept: reduce to LCS</td>
<td>X = {min(A), min(A)+1, ....,max(A)}</td>
</tr>
<tr>
<td></td>
<td>LIS(A) = LCS (A,X)</td>
</tr>
<tr>
<td></td>
<td>E.g.: X = {1,2,3,4,5,6,7,8,9}</td>
</tr>
<tr>
<td>Time complexity</td>
<td>(\Theta(n+v +nv) = \Theta(nv)) depends on min and max values from A: v = max(A)-min(A)+1</td>
</tr>
<tr>
<td>Space complexity</td>
<td>(\Theta(nv))</td>
</tr>
</tbody>
</table>
LIS to LCS reduction

• A = {3,5,3,9,3,4,3,1,2,1,4}
• LIS with NO repetitions:
  – Produce: X = {1,2,3,4,5,6,7,8,9}
  – LIS(A) = LCS(A,X)
  – If v>>n , where v = |X| = max(A)-min(A)+1, use
    X = {1,2,3,4,5,9} (unique elements of A sorted in increasing order)
    E.g. A = {50,1, 1, 800,50, 1,100000}, use X = {1,50,800,100000},
      (NOT: x = {1,2,3,4,5,6……………………………,100000})

• LIS WITH repetitions:
  – Produce X = {1,1,2,3,3,3,3,4,4,5,9}
  – LIS(A) = LCS(A,X)
The Edit Distance

Application: Spell-checker

Spell checker
- Computes the “edit distance” between the words: the smaller distance, the more similar the words.

Edit distance
- Minimum cost of all possible alignments between two words.

• Other: search by label/title (e.g. find documents/videos with cats)
• This is a specific case of a more general problem: time series alignment.
• Another related problem is: Subsequence Search.
Alignments

Examples of different alignments for the same words

- S E T S
  1 1 1 1 1
  R E S E T
  Cost/distance: 5

- S E T S
  1 1 0 0 0 1
  R E S E T
  Cost/distance: 3

- S - E T S
  1 1 1 0 1 1
  R E S E - T
  Cost/distance: 5

• No cross-overs: The letters must be in the order in which they appear in the string.

Incorrect alignment

- S E T S
  1 × 1 1
  R E S E T

Pair cost:
Same letters: 0
Different letters: 1
Letter-to-dash: 1
Alignment cost: sum of costs of all pairs in the alignment.
Edit distance: minimum alignment cost over all possible alignments.
The Edit Distance

- Edit distance – the cost of the best alignment
  - Minimum cost of all possible alignments between two words.
  - (The smaller distance, the more similar the words)

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>E</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Edit distance: minimum alignment cost over all possible alignments.
Alignment cost: sum of costs of all pairs in the alignment.

Pair cost:
- Same letters: 0
- Different letters: 1
- Letter to dash: 1

<table>
<thead>
<tr>
<th></th>
<th>-</th>
<th>-</th>
<th>S</th>
<th>E</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>-</th>
<th>-</th>
<th>S</th>
<th>E</th>
<th>T</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>-</th>
<th>-</th>
<th>P</th>
<th>E</th>
<th>T</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>E</th>
<th>S</th>
<th>E</th>
<th>T</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Notations, Subproblems

• Notation:
  – $X = x_1, x_2, \ldots, x_n$
  – $Y = y_1, y_2, \ldots, y_m$
  – $\text{Dist}(i,j) =$ the smallest cost of all possible alignments between substrings $x_1, x_2, \ldots, x_i$ and $y_1, y_2, \ldots, y_j$.
  – $\text{Dist}(i,j)$ will be recorded in a matrix at cell $[i,j]$.

• Subproblems of ("SETS", "RESET"):  
  – Problem size can change by changing either $X$ or $Y$ (from two places):

• What is Dist for all of the above problems?
Notations, Subproblems

• Notation:
  – \( X = x_1, x_2, \ldots, x_n \)
  – \( Y = y_1, y_2, \ldots, y_m \)
  – \( \text{Dist}(i,j) = \text{the smallest cost of all possible alignments between substrings } x_1, x_2, \ldots, x_i \text{ and } y_1, y_2, \ldots, y_j \).
  – \( \text{Dist}(i,j) \) will be recorded in a matrix at cell \([i,j]\).

• Subproblems of ("SETS", "RESET"):
  – Problem size can change by changing either \( X \) or \( Y \) (from two places):
    – ("S", "RES")
    – ("", "R"), ("", "RE"), ("", "RES"), ..., ("", "RESET")
    – ("S", ""), ("SE", ""), ("SET", ""), ("SETS", "")
    – ("", "" )

• What is \( \text{Dist} \) for all of the above problems?
Dependence on Subproblems

- $Dist(i, j)$ – depends on $Dist(i-1, j-1)$, $Dist(i-1, j)$, $Dist(i, j-1)$

(below, grayed areas show the solved subproblems)

\[
\begin{align*}
Dist(i-1, j-1) + 0, & \quad \text{if } x_i = y_j \text{ or } \\
Dist(i-1, j-1) + 1, & \quad \text{if } x_i \neq y_j \\
Dist(i-1, j) + 1, & \quad \text{(insert in Y)} \\
Dist(i, j-1) + 1, & \quad \text{(insert in X)}
\end{align*}
\]
Edit Distance

Filling out the distance matrix

• Each cell will have the answer for a specific subproblem.

• Special cases:
  – Dist(0,0) =
  – Dist(0,j) =
  – Dist(i,0) =
  – Dist(i,j) =

• Complexity (where: |X| = n, |Y| = m):
  Time:                        Space:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&quot;&quot;&quot;&quot;</td>
<td>R</td>
<td>E</td>
<td>S</td>
<td>E</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Represents some alignment between “RESE” and “SET”
Edit Distance – Cost function

• Each cell will have the answer for a specific subproblem.

• Special cases:
  – Dist(0,0) = 0
  – Dist(0,j) = 1 + Dist(0,j-1)
  – Dist(i,0) = 1 + Dist(i-1,0)
  – Dist(i,j) = \min \{ \text{Dist}(i-1,j)+1, \text{Dist}(i,j-1)+1, \text{Dist}(i-1,j-1) \} \text{ if } x_i = y_j \text{ or}
    \min \{ \text{Dist}(i-1,j)+1, \text{Dist}(i,j-1)+1, \text{Dist}(i-1,j-1)+1 \} \text{ if } x_i \neq y_j

• Complexity (where: |X| = n, |Y| = m): Time: O(n*m)         Space: O(n*m)

NOTE: Use this definition where for Dist(i,j) the min of the 3 possible smaller problems is used regardless of how letters x_i and y_j compare.
Edit Distance

Recover the alignment – Worksheet
(Using the arrow information)

X = SETS
Y = RESET

Time complexity: O(.........)
(where: |X| = n, |Y| = m)

```
<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>E</th>
<th>S</th>
<th>E</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Aligned Pair | Update
------------ | -------
\( \leftarrow \) | \( x_i \)
\( \rightarrow \) | \( y_j \)
\( \uparrow \) | \( x_i \)
\( \downarrow \) | \( y_j \)

Start at:
i = ........
j = ........

How big will the solution be (as num of pairs)?

\[ X = \text{SETs} \]
\[ Y = \text{RESET} \]
# Edit Distance

## Recover the alignment

Here the pairs are filled in from the LEFT end to the RIGHT end and printed from RIGHT to LEFT.

**X** = SETS

**Y** = RESET

(Where: |X| = n, |Y| = m)

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td></td>
<td>R</td>
<td>E</td>
<td>S</td>
<td>E</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>***</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>S</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>E</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>S</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**Print from right to left.**

**Sum of costs of pairs in the alignment string is the same as table[4][5]:**

1 + 0 + 0 + 0 + 1 + 1 = 3

### Time complexity: \( O(n+m) \)

<table>
<thead>
<tr>
<th></th>
<th>Aligned Pair</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \leftarrow )</td>
<td>( x_i )</td>
<td>( i = i-1 )</td>
</tr>
<tr>
<td>( \uparrow )</td>
<td>( y_j )</td>
<td>( j = j-1 )</td>
</tr>
<tr>
<td>( \uparrow )</td>
<td>( x_i )</td>
<td>( i = i-1 )</td>
</tr>
<tr>
<td>( \leftarrow )</td>
<td>( - )</td>
<td>( j = j-1 )</td>
</tr>
</tbody>
</table>

**Start at:**

\( i = 4 \)

\( j = 5 \)

**How big will the solution be (as num of pairs)?**

\( n+m \)
• What is the best alignment between 
  abcdefghijk
  cdXYZefgh
Edit Distance

Recover the alignment - Method 2: (based only on distances)

Even if the choice was not recorded, we can backtrace based on the distances: see from what direction (cell) you could have gotten here.

<table>
<thead>
<tr>
<th></th>
<th>w</th>
<th>w</th>
<th>a</th>
<th>b</th>
<th>u</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>e</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>f</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>y</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>y</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>y</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>y</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

first: abcdefyyy
second: wwabudef
edit distance:
Alignment:
# Edit Distance

## Recover the alignment - Method 2:
(based only on distances)

Even if the choice was not recorded, we can backtrace based on the distances: see from what direction (cell) you could have gotten here.

<table>
<thead>
<tr>
<th></th>
<th>w</th>
<th>w</th>
<th>a</th>
<th>b</th>
<th>u</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>e</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>f</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>y</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>y</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

**first:** abcdefyyy  
**second:** wwbabudef  
**edit distance:** 6  
**Alignment:**

- - a b c d e f y y y  
w w a b u d e f - - -  
1 1 0 0 1 0 0 0 1 1 1
Sample Exam Problem

On the right is part of an edit distance table. CART is the complete second string. AL is the end of the first string (the first letters of this string are not shown).

a. (6 points) Fill-out the empty rows (finish the table).

b. (4 points) How many letters are missing from the first string (before AL)? Justify your answer.

c. (8 points) Using the table and the information from part b), for each of the letters C and A in the second string, CART, say if it could be one of the missing letters of the first string: Yes (it is one of the missing letters – ‘proof’), No (it is not among the missing ones – ‘proof’), Maybe (it may or may not be among the missing ones – give example of both cases).

- C: Yes/No/Maybe. Justify:

- A: Yes/No/Maybe. Justify:

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>A</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Edit Distance
Sliding Window

• Space complexity improvement:
  – Keep only two rows
  – Keep only one row
Motivation for Edit Distance

• The Edit Distance essentially does *Time Series Alignment*

• Other examples of problems solved with Time Series Alignment:
  – Given observed temperatures, find location:
    • Collected in a database temperatures at different hours over one day in various places (labelled with the name). Given a query consisting of temperatures collected in an unknown place, find the place with the most similar temperatures. Issues:
      – Not same number of measurements for every place and query.
      – Not at the exact same time. (E.g. maybe Mary recorded more temperatures throughout the day and none in the night, and John collected uniformly throughout the day and night.)
  – Find videos showing a similar sign
  – Find shapes in images (after image processing extracted relevant features)

• Find a substring in a string
  – E.g. swear words in Pokemon Names
  – Uses two additional sink states (at the beginning and end of the small query)
DP in an exam

• Must be able to solve these problems in the shown format and with the methods covered in class.

• Components of every DP problem:
  – Problem description and representation,
  – Cost function (math formulation and/or implementation), recover the solution, brute force method
    • time complexity for all three items above
  – Solve on paper:
    • Find optimum.
    • Recover choices that give optimum solution.
DP or not DP?

- I consider DP problems to be optimization problems.
  - CLRS gives only optimization problems as DP
- Other sources list non-optimization problems as DP as well.
- Main point: **optimization problems are hard. Consider DP for them.**

- DP requires two properties:
  - Optimal substructure – applicable to optimization problems
    - An optimal solution to the big problem can be constructed from optimal solutions to smaller problems.
  - Overlapping subproblems – applicable to all problems (including Fibonacci, stair climbing, matrix traversal)
    - The pattern of the solution in both cases (e.g. matrix traversal with and without optimization) is the same except that in one case you add the answer from the subproblems and in the other you pick the max.
    - Results in an ordering of the subproblems, where each next depends on the previous ones.
Divide and Conquer vs DP vs Recursion

• Both DP and Divide and Conquer can be easily solved using recursion
  – For Divide and conquer you typically want to use recursion (ok if you do)
  – For DP you want to use memoization or the iterative solution (recursion is exponential)

• Recursive problem that does not allow neither DP nor Div & Conq:
  – Place N queens on an NxN check board (s.t. they do not attack each other)

• Divide and conquer problems:
  – Sum of elements in an array
    • Give the solution here
  – Factorial with splitting in half (or Sorting elements in an array (Mergesort))

  • Notes:
    • Problems of same size (e.g. 3 elements) are different (depending on the actual values and order of the elements). => cannot ‘reuse’ solutions to smaller problems.
    • Unlike DP, the solution does not depend on where we cut (ok to cut into subproblems at any place)

• Why Dynamic Programming (DP) and not Divide and Conquer
  – We cannot apply Div&Conq because we cannot commit to a ‘point’ in the solution (e.g., for Job Scheduling, choose if a certain job will be selected or not).
When DP does not work

• Queens placement problem:
  – Fails overlapping subproblems: Cannot reuse solutions to smaller problems (even if just counting).
    • The placement on any column depends on the placement on previous columns => cannot reuse the solution from previous columns (it will in fact certainly not work).

• Longest path (with no cycles) in a graph
  – Fails optimal substructure: if C is part of an optimal solution for A->D, then the A->C and C->D parts of that solution, are optimal solution for those smaller problems. Fails. There is a longer path from C to D: C,B,D
  – Note: Shortest path has this property:
    • A->D: A,B,D
    • C is not part of an optimal solution
Dynamic Programming

Space Requirements

• Quadratic or linear
  – Quadratic: edit distance, LCS, 0/1 Knapsack
  – Linear: Unbounded Knapsack

• If you only need the optimal cost, you may be able to optimize it,
  – Quadratic -> linear: keep only the previous row (e.g. LCS)
  – Linear -> constant: keep only the computations for the few immediately smaller problems that are needed (e.g. Fibonacci)

• If you need to also backtrace the solution (choices) that gave this optimal cost, you have to keep information (either cost or choice) for all the smaller subproblems.
List of DP Problems

- Stair Climbing
- LCS (Longest Common Subsequence) *
- LIS (Longest Increasing Subsequence)
- Edit distance *
- Job scheduling *
- Knapsack
  - Special case: Rod cutting
  - Equivalent: stair climbing with gain
- Matrix traversal
  - All ways to traverse a matrix – not an optimization problem
  - All ways to traverse it when there are obstacles
  - Most gain when there is fish that you can eat/catch.
    - Variant: add obstacles
    - Monotone decimal / Strictly monotone decimal (consecutive digits are ≤ or <)
- Shuttle to airport
- Subset sum
- Chain matrix multiplication
  - Order in which to multiply matrices to minimize the cost.
Motivation for DP

• Applications:
  – Edit distance and other similarity measures:
    • Word similarity
    • Time series similarity (ASL, compare trends, data that can be represented as sequences and require alignment) that requires
      – Can you guess how we used DP to find videos of similar signs in ASL?

• Note that for some problems it is even hard to find the Brute Force solution or the Greedy solution
  – e.g. for the Edit Distance
Analyzing recursive functions

- Look at the implementations for
  - Knapsack
  - Rod cutting
  - Memoization motivation example (the math definition)
Worksheet: Inefficient Recursive Solution (CLRS: Rod-Cutting)

Write the recurrence formula for this function: $T(N)$ =........

```
CUT-ROD(p, n)
1 if n == 0
2 return 0
3 q = -\infty
4 for i = 1 to n
5 q = max(q, p[i] + CUT-ROD(p, n - i))
6 return q
```

Source: CLRS
Answers:

Inefficient Recursive Solution
(CLRS: Rod-Cutting)

\[ T(n) = \sum_{i=1}^{n} T(n-i) + \Theta(n) \]

\begin{verbatim}
CUT-ROD(p, n)
1  if n == 0
2       return 0
3  q = -\infty
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
\end{verbatim}

Note the difference between
- what the function computes and returns (for this I need the information in the array p)
and
- the time complexity: T(N)
(for this I do not need to know p)

Source: CLRS
Example: recursion, time complexity, benefits of memoization

\[
F(n) = \begin{cases} 
5 & \text{if } 0 \leq n \leq 3 \\
F(n+3) - 2 & \text{if } n \text{ is odd} \\
F(n/2) + F(n-4) & \text{if } n \text{ is even}
\end{cases}
\]

Example used to show:
- Benefits of memoization
- Difference between evaluating a function and the time complexity of a program solving that function

• Evaluate the function in points: 2, 3, 4, 5, 6, 7.

• Give a C implementation (code) for this math function.
  – Draw the tree showing the recursive calls for both recursions: inefficient and memoized.
  – Use the tree drawn above to estimate the time complexity for the memoized version. (Hint: notice that each number (problem size) is an internal node at most once in a binary tree.)

• What is the time complexity of your function?
  – Write the recursive formula for your implementation. (For solving: bring to one recursive case.)

• Benefits of memoization:
  ** for F(10) not needed: F(7) and F(9)
  ** for F(20) not needed: F of: 7, 9, 11, 13, 14, 15, 17, 18, 19
Worksheet

\[ F(n) = \begin{cases} 
3 & \text{if } n \leq 1 \\
F(n-1) & \text{if } n \text{ is odd} \\
F(n/2) + 10 & \text{if } n \text{ is even}
\end{cases} \]

```c
int foo(int n){
    if (n <= 1) return 3;
    if (n%2 == 1)
        return foo(n-1);
    else
        return foo(n/2)+10;
}
```

- Math definition => code => recurrence formula => Θ
- Recursive inefficient => memoized
Answer

\[ F(n) = \begin{cases} 
3 & \text{if } n \leq 1 \\
F(n-1) & \text{if } n \text{ is odd} \\
F(n/2)+10 & \text{if } n \text{ is even}
\end{cases} \]

\begin{verbatim}
int foo(int n) {
    if (n <= 1) return 3;
    if (n%2 == 1)
        return foo(n-1);
    else
        return foo(n/2)+10;
}
\end{verbatim}

\[ T(n) = c \quad \text{if } n \leq 1 \]
\[ T(n) = T(n-1) + c \quad \text{if } n \text{ is odd} \]
\[ T(n) = T(n/2) + c \quad \text{if } n \text{ is even} \]

Note that in this case it is a coincidence that the recurrence formulas, \( T(n) \), for the time complexity are similar to what the function computes (and the math definition). Typically they are different. Even here you can see the difference in the case when \( n \) is odd: the \(+c\) in the \( T(n) \) formula.
Answer

int foo(int n) {
    if (n <= 1) return 3;
    if (n%2 == 1)
        return 5*foo(n-1);
    else
        return foo(n/2)+10;
}

\[
T(n) = c \quad \text{if } n \leq 1 \\
T(n) = T(n-1) + c \quad \text{if } n \text{ is odd} \\
T(n) = T(n/2) + c \quad \text{if } n \text{ is even}
\]

Note that when \( n \) is odd, we still use \( T(n-1) \) and not \( 5T(n-1) \), because still only one recursive call is made in the foo function when \( n \) is odd. (The result of that call is multiplied with 5, but that affects what the function computes, not its time complexity.)

Because there are 2 actual calls to \( \text{foo}(n-1) \) we will have \( 2*T(n-1) \) in the recurrence formula.
From Recursive Formula to Code Worksheet

• Give a piece of code/pseudocode for which the time complexity recursive formula is:
  – $T(N) = N \times T(N-1) + n$ or
  – $T(N) = N \times T(N-1) + \Theta(N)$

Assume $T(1) = c$
• Give a piece of code/pseudocode for which the time complexity recursive formula is:
  - \( T(N) = N \cdot T(N-1) + n \) or
  - \( T(N) = N \cdot T(N-1) + \Theta(N) \)

Assume \( T(1) = c \)

```c
int foo(int N){
    if (N <= 1) return 3;
    for(int i=1; i<=N; i++)
        foo(N-1);
}
```
Compare

```c
int foo1(int N) {
    if (N <= 1) return 3;
    for(int i=1; i<=N; i++){
        foo1(N-1);
    }
}
```

```c
t(N) = N*T(N-1) + cN
```

```c
int foo2(int N) {
    if (N <= 1) return 3;
    for(int i=1; i<=N; i++){
        printf("A");
    }
    foo2(N-1); // foo2(N-1) is not in the loop
}
```

```c
t(N) = T(N-1) + cN
```
DP - summary

• Most problems that can be solved with DP have a solution that can be expressed with a cost function.

• If the cost function is given the implementation is straight-forward. Implementation options:
  – Iterative
  – Recursive
  – Memoized

• Guidelines for finding the cost function:
  – Express the solution to the problem as a sequence of independent choices/steps
  – Try to write the cost of the current problem using answers to subproblems:
    • You must consider the choice and the answer to the subproblem.
Additional Problems
Stair Climbing
Stair Climbing

• Problem:
  A child has to climb N steps of stairs. He can climb 1, 2 or 3 steps at a time.
  How many different ways are there to climb the stairs?
  
  • We count permutations as different ways, e.g.:
    – (3, 1, 3, 1) is different than (3, 3, 1, 1)
      (2 jumps of 3 and 2 jumps of 1 in both cases, but in different order)
  • Let $C(n)$ – number of different ways to climb n stairs.

• Solution:
  
  – Hints:
    • What dictates the problem size (e.g. give 2 problems of different sizes).
    • What is the smallest problem size that you can solve?
Stair Climbing

• Problem:
  A child has to climb N steps of stairs. He can climb 1, 2 or 3 steps at a time. How many different ways are there to climb the stairs?
  • We count permutations as different ways, e.g.:
    – (3, 1, 3, 1) is different than (3, 3, 1, 1)
      (2 jumps of 3 and 2 jumps of 1 in both cases, but in different order)
  • Let $C(n)$ – number of different ways to climb n stairs.

• Solution:
  – What is the last jump he can do before he reaches the top?
    • 1, 2 or 3  => $C(n) = C(n-1) + C(n-2) + C(n-3)$
  – Set-up base cases
    • $C(0) = 1$ (stays in place, this allows us to cover N = 1, 2, 3 by recursion)
    • $C(x) = 0$, for all x < 0
  – Test that it works for the first few small n: 1, 2, 3, 4:
    • $C(1) = C(0) + C(-1) + C(-2) = 1 + 0 + 0 = 1$
    • $C(2) = C(1) + C(0) + C(-1) = 1 + 1 + 0 = 2$ ( (1, 1) or (2) )
    • $C(3) = C(2) + C(1) + C(0) = 2 + 1 + 1 = 4$ ( (1,1,1) , (1,2), (2,1), (3) )
    • $C(4) = C(3) + C(2) + C(1) = 4 + 2 + 1 = 7$
Stair Climbing

• Write code
## Stair Climbing Worksheet

Possible ways to climb this many stairs.

Count based on possible last steps.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sol</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Stair climbing with gain

Solve for 17 stairs with the Gain (e.g. health points, HP,) per jump size given below. Here the only jump sizes that give gain are the ones in the table. You can assume that other jump sizes are allowed, but they give 0 gain. That allows us to consider ALL problems size (for example a problem of size 1 is allowed: 1 step we have 0 gain.)

<table>
<thead>
<tr>
<th>Jump Size</th>
<th>2</th>
<th>3</th>
<th>9</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Gain</td>
<td>2</td>
<td>4</td>
<td>18</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Stair climbing with gain

<table>
<thead>
<tr>
<th>Jump</th>
<th>Gain (health points (HP))</th>
<th>Jump size</th>
<th>Remaining stairs [rem_stairs] (smaller problem size)</th>
<th>Sol[rem_stairs] ($$) (answer to smaller problem)</th>
<th>New total Gain (HP) (for each possible choice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>4 ( =6-2 )</td>
<td>4</td>
<td></td>
<td>6 ( = 4+2)</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3 ( =6-3 )</td>
<td>4</td>
<td></td>
<td>8 ( = 4+4)</td>
</tr>
<tr>
<td>18</td>
<td>9</td>
<td>-</td>
<td>-</td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

Solution (gain) for problems of this size.

Last choice that gives the optimal solution.

Gain based on possible last steps. These rows show the work needed to fill-in the answers in Sol and Picked.

The top table shows the details of the work from the last 3 rows of the bottom table.

Pick the maximum as the answer for this problem size, 6.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sol</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Picked</td>
<td>-1</td>
<td>-1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Gain with this jump

Remaining stairs

Index 0 1 2 3 4 5 6 7 8 9 10 11

Pick: 0,2 1,2 2,4 3,6 4,6

Pick: 0,4 1,4 2,6 3,8

Pick: - - - - - - - - - -
### Stair climbing with gain - Answers

<table>
<thead>
<tr>
<th>Jump</th>
<th>Gain (health points (HP))</th>
<th>Jump size</th>
<th>Remaining stairs [rem_stairs] (smaller problem size)</th>
<th>Sol[rem_stairs] ($$) (answer to smaller problem)</th>
<th>New total Gain (HP) (for each possible choice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4 (=6-2)</td>
<td>4</td>
<td>6 (=4+2)</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3 (=6-3)</td>
<td>4</td>
<td>8 (=4+4)</td>
</tr>
<tr>
<td>18</td>
<td>9</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The top table shows the details of the work from the last 3 rows of the bottom table.

Pick the maximum as the answer for this problem size, 6.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sol</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>10</td>
<td>18</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>Picked</td>
<td>-1</td>
<td>-1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>9</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

Gain based on possible last steps. These rows show the work needed to fill-in the answers in Sol and Picked.

**Solution (gain) for problems of this size.**

Last choice that gives the optimal solution.
Gain based on possible last steps. These rows show the work needed to fill-in the answers in Sol and Picked. In an implementation (code) you will NOT need a 2D array.

**Solution (gain) for problems of this size.**

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sol</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>10</td>
<td>18</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>Picked</td>
<td>-1-1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>9</td>
<td>9</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2 (2)</td>
<td>-</td>
<td>-</td>
<td>0,2</td>
<td>1,2</td>
<td>2,4</td>
<td>3,6</td>
<td>4,6</td>
<td>6,8</td>
<td>6,10</td>
<td>7,10</td>
<td>8,12</td>
<td>9,20</td>
</tr>
<tr>
<td>3 (4)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0,4</td>
<td>1,4</td>
<td>2,6</td>
<td>3,8</td>
<td>4,8</td>
<td>5,10</td>
<td>6,12</td>
<td>7,12</td>
<td>8,14</td>
</tr>
<tr>
<td>9 (18)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0,18</td>
<td>1,18</td>
<td>2,20</td>
</tr>
</tbody>
</table>
Rod Cutting Problem

• Given:
  – A rod of a certain length, \( n \).
  – An array of prices for rods of sizes between 1 and \( n \).
  – What is the best way to cut the rod so that you make the most money out of selling it?

• Optimization problem:
  – There are many ways to cut the rod, you want the best one.

<table>
<thead>
<tr>
<th>Length ( i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price ( p_i )</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
<td>24</td>
<td>30</td>
</tr>
</tbody>
</table>

\[
 r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) \]
Example of an Inefficient Recursive Solution (CLRS: Rod-Cutting)

Recursive solution idea:
- Consider all possible sizes for the first piece and make a recursive call for cutting the remaining piece
- Add the values (of the first piece and the result of the recursive call)
- Return the max of all these combinations

```plaintext
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

Source: CLRS
Rod Cutting Problem

• Recursive solution:
  – Try cutting a first piece of all possible size and make a recursive call for cutting the remaining piece
  – Add the values (of the first piece and the result of the recursive call)
  – Return the max of all these combinations

• Recursion tree
  – Example for $n = 4$
    • Notice how many times a recursive call is made for problems of the same size: 8 times for size 0, 4 times for size 1, 2 times for size 2
      – We will fix this inefficiency.
  – Properties of a tree for rod of size $n$:
    • Total nodes (including leaves): $2^n$. (proof by induction)
    • Total leaves: $2^{n-1}$. (proof by induction)
    • Number of leaves = number of different ways to cut the rod
      – Each path down to a leaf corresponds to a different way of cutting the rod.
Rod Cutting Problem

- 3 solutions:
  - Recursive – inefficient (NOT Dynamic Programming (DP)) – 
    - $\Theta(2^n)$
  - Top-down DP (Memoization) - recursive with saving the computed results
    - $\Theta(n^2)$
  - Bottom-up DP - non-recursive
    - $\Theta(n^2)$

- Recursion tree
  - For plain recursion
  - For memoization
    - Also see the solution array

- DP solution complexity - $\Theta(n^2)$
Shuttle-to-Airport Problem
Shuttle-to-Airport Problem

(Dr. Bob Weems, notes07)
(We may cover this at the end)

\[ U(i) - \text{optimal cost to reach top node } i \]
\[ L(i) - \text{optimal cost to reach bottom node } i \]
\[ U(0) = L(0) = 0 \]
\[ U(i) = \min\{U(i-1) + UD(i), \: L(i-1) + LD(i) + LU(i)\} \]
\[ L(i) = \min\{L(i-1) + LD(i), \: U(i-1) + UD(i) + UL(i)\} \]

Time complexity: \( O(n) \)
Fill out \( U \) and \( L \) arrays with constant local cost.
Shuttle-to-Airport Problem

Brute Force

• Evaluate all routes that end at the top n-th point.
  – The top n-th and bottom n-th points will have the same value, so we will use the top n-th point as the airport.
    • These are really the same point (the airport). We artificially split it into two points to make the problem representation more uniform (as opposed to having it as a special case).
    • Any route that ends at the bottom n-th point is a duplicate of the same route ending at the top point (with the use of one of the vertical n-th arrows with cost 0).

A route ending at the top.
Using the gray arrow, we get the duplicate route ending at the bottom.

A route ending at the bottom.
Using the gray arrow, we get the duplicate route ending at the top.
Shuttle-to-Airport Problem
Brute Force: Version 1

• VERSION 1: count all the ways to get to any point.

• Let:
  – top(i) be the total number of different ways to get to the top point, i.
  – bottom(i) be the total number of different ways to get to the bottom point, i.

• Then:
  – top(0) = bottom(0) = 1 (we are already at the starting point)
  – top(i) = bottom(i) = top(i-1) + bottom(i-1)
  – => top(i) = 2*top(i-1) => top(i) = 2^i.
    • Straight forward proof by induction that top(i) == bottom(i) and that top(i) = 2^i.
  – => top(n) = 2^n routes

• => O(2^n) complexity of brute force
Shuttle-to-Airport Problem
Brute Force: version 2

• Version 1: count representations of all different routes.

• Notice that:
  – A horizontal arrow will always be picked (it moves us closer to the airport)
  – A vertical arrow may or may not be picked.
  – There cannot be two consecutive vertical arrows. Between any two horizontal arrows in a solution, there may be nothing, or a vertical arrow.

• The difference between two routes is in the placement of vertical arrows.

  ⇒ Encode only the vertical movement: 1-used (either up or down), 0-not used
  ⇒ is encoded as: 1 0 1 1
  ⇒ $2^n$ routes $\Rightarrow$ $O(2^n)$
  ⇒ Note that the ‘reflection’ routes will lead to the $n$-th bottom point and that is a duplicate solution, thus ignored.
Shuttle-to-Airport Problem
Backtracing

- Version 1: record choices (h - horizontal or v-vertical arrow to reach that node)
- Version 2: use the cost function to see which way you came from:
  - If \( (U(i-1) + UD(i)) \leq (L(i-1) + LD(i) + LU(i)) \)
    - Horizontal (upper direct arrow used)
  - Else
    - Vertical (lower to upper arrow used)
  - Similar for \( (L(i-1) + LD(i)) \), and \( (U(i-1) + UD(i) + UL(i)) \)
- Hint, if you store U and L in a 2D array as rows, you can easily switch from top to bottom by using \( (row+1)\%2 \). The row value indicates: 0-top, 1-bottom
- Time complexity: O(n)
Shuttle-to-Airport Backtracing

Example showing that in the first path (when computing the optimal cost) we do not know what nodes will be part of the solution. Here the top 3 nodes have costs smaller than their bottom correspondents, but the final top cost is too large and the bottom route becomes optimal.

Route (in reversed order, starting from 6 top):
6 top, 5 top, 5 bottom, 4 bottom, 3 bottom, 3 top, 2 top, 2 bottom, 1 bottom, start point
# Worksheet Knapsack Example 2

<table>
<thead>
<tr>
<th>Item</th>
<th>Item value ($)</th>
<th>Item weight (Kg)</th>
<th>Remaining capacity (Kg)</th>
<th>Sol[rem_capacity] ($)</th>
<th>New total value ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>10</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>11</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>13</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sol</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Picked</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 (4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 (5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 (10)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 (11)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 (13)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Answers Knapsack Example 2

### Table: Knapsack Solution

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sol</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>10</td>
<td>11</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>17</td>
<td>18</td>
<td>20</td>
<td>21</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>Picked</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>A, 3, 4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0,</td>
<td>4</td>
<td>1,4</td>
<td>2,4</td>
<td>3,8</td>
<td>4,9</td>
<td>5,9</td>
<td>6,</td>
<td>12</td>
<td>7,</td>
<td>14</td>
<td>8,</td>
<td>15</td>
<td>9,</td>
<td>17</td>
</tr>
<tr>
<td>B, 4, 5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0,</td>
<td>5</td>
<td>1,5</td>
<td>2,0</td>
<td>3,9</td>
<td>4,</td>
<td>10</td>
<td>5,</td>
<td>10</td>
<td>6,</td>
<td>13</td>
<td>7,</td>
<td>15</td>
<td>8,</td>
</tr>
<tr>
<td>C, 7,10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>D,8,11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0,</td>
<td>11</td>
<td>1,</td>
<td>11</td>
<td>2,</td>
<td>11</td>
<td>3,</td>
<td>15</td>
</tr>
<tr>
<td>E,9,13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0,</td>
<td>13</td>
<td>1,</td>
<td>13</td>
<td>2,</td>
<td>13</td>
<td>3,</td>
<td>17</td>
</tr>
</tbody>
</table>