Recursion and Dynamic Programming

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

• Recursion is a fundamental concept in computer science.

• **Recursive algorithms**: algorithms that solve a problem by solving one or more smaller instances of the same problem.
  
  – Example:
    • Base case:

• **Recursive functions**: functions that call themselves.
  
  – Example:
    • Base case:

• **Recursive data types**: data types that are defined using references to themselves.
  
  – Example:
    • Base case:

• In all recursive concepts, there are one or more **base cases**. No recursive concept can be understood without understanding its base cases.
Recursion

• Recursion is a fundamental concept in computer science.

• **Recursive algorithms**: algorithms that solve a problem by solving one or more smaller instances of the same problem.
  – Example: binary search
    • Base case: one or no element in collection.

• **Recursive functions**: functions that call themselves.
  – Example: \( N! = N \times (N-1)! \)
    • Base case: \( N = 0 \)

• **Recursive data types**: data types that are defined using references to themselves.
  – Example: Nodes in the implementation of linked lists.
    • Base case: NULL

• In all recursive concepts, there are one or more **base cases**. No recursive concept can be understood without understanding its base cases.
Recursive Algorithms

• **Recursive algorithms**: algorithms that solve a problem by solving one or more smaller instances of the same problem.

• A recursive algorithm can always be implemented both:
  – using recursive functions, and
  – without recursive functions.

• Example of a recursive function:
Recursive Algorithms

• **Recursive algorithms**: algorithms that solve a problem by solving one or more smaller instances of the same problem.

• A recursive algorithm can always be implemented both:
  – using recursive functions, and
  – without recursive functions.

• Example of a recursive function: the factorial.
  – Recursive definition?
  – Non-recursive definition?
Recursive Algorithms

Recursive Definition:

```c
int factorial(int N) {
    if (N == 0) return 1;
    return N * factorial(N - 1);
}
```

Non-Recursive Definition:

```c
int factorial(int N) {
    int result = 1;
    int i;
    for (i = 2; i <= N; i++) result *= i;
    return result;
}
```
Recursive Algorithms

Recursive Definition:

```c
int factorial(int N) {
    if (N == 0) return 1;
    return N * factorial(N - 1);
}
```

Non-Recursive Definition:

```c
int factorial(int N) {
    int result = 1;
    int i;
    for (i = 2; i <= N; i++) result *= i;
    return result;
}
```

How is factorial(3) evaluated?
Analyzing a Recursive Program

• Analyzing a recursive program involves answering two questions:
  – Does it always terminate?
  – Does it always compute the right result?
• Both questions are answered by induction.
• Example: does the factorial function on the right always compute the right result?
• Proof: by induction.

Recursive Definition:

```c
int factorial(int N)
{
    if (N == 0) return 1;
    return N*factorial(N-1);
}
```
Analyzing a Recursive Program

• Proof: by induction.
• Step 1: (the base case)
  – For \( N = 0 \), \( \text{factorial}(0) \) returns 1, which is correct.
• Step 2: (using the inductive hypothesis)
  – Suppose that \( \text{factorial}(N) \) returns the right result for \( N = K \), where \( K \) is an integer \( \geq 0 \). ( \( \text{factorial}(K) = K! \) )
  – Then, for \( N = K+1 \), \( \text{factorial}(N) \) returns:
    \[
    N \times \text{factorial}(N-1) = (K+1) \times \text{factorial}(K) = (K+1) \times K! = (K+1)! = N!
    \]
  – Thus, for \( N = K+1 \), \( \text{factorial}(N) \) also returns the correct result.
• Thus, by induction, \( \text{factorial}(N) \) computes the correct result for all \( N \).

Recursive Definition:

```c
int factorial(int N) {
    if (N == 0) return 1;
    return N*factorial(N-1);
}
```

Where precisely was the inductive hypothesis used?
Analyzing a Recursive Program

• Proof: by induction.

• Step 1: (the base case)
  – For \( N = 0 \), \( \text{factorial}(0) \) returns 1, which is correct.

• Step 2: (using the inductive hypothesis)
  – Suppose that \( \text{factorial}(N) \) returns the right result for \( N = K \), where \( K \) is an integer \( \geq 0 \).
    \( (\text{factorial}(K) = K! \) )
  – Then, for \( N = K+1 \), \( \text{factorial}(N) \) returns:
    \[ N \times \text{factorial}(N-1) = (K+1) \times \text{factorial}(K) = (K+1) \times K! = (K+1)! = N! \]
  – Thus, for \( N = K+1 \), \( \text{factorial}(N) \) also returns the correct result.

• Thus, by induction, \( \text{factorial}(N) \) computes the correct result for all \( N \).

Recursive Definition:

```c
int factorial(int N) {
    if (N == 0) return 1;
    return N*factorial(N-1);
}
```

Where precisely was the inductive hypothesis used?

In substituting \( K! \) for \( \text{factorial}(K) \).
Guidelines for Designing Recursive Functions

• Guidelines for writing recursive functions:
  – They must explicitly solve one or more base cases.
  – Each recursive call must involve smaller values of the arguments, or smaller sizes of the problem.
  – (Strictly speaking, the only way to verify the correctness is a mathematical proof.)
Example Violation of the Guidelines

int puzzle(int N)
{
    if (N == 1) return 1;
    if (N % 2 == 0)
        return puzzle(N/2);
    else return puzzle(3*N+1);
}

• How does this function violate the guidelines we just stated?
Example Violation of the Guidelines

```c
int puzzle(int N) {
    if (N == 1) return 1;
    if (N % 2 == 0)
        return puzzle(N/2);
    else return puzzle(3*N+1);
}
```

• How does this function violate the guidelines we just stated?
• The function does NOT always call itself with smaller values.
• Consequence: it is hard to prove if this function always terminates.
• No one has actually been able to prove or disprove that!!!
Euclid's Algorithm

```c
int gcd(int m, int n)
{
    if (n == 0) return m;
    return gcd(n, m % n);
}
```

- One of the most ancient algorithms.
- Computes the greatest common divisor of two numbers.
- It is based on the property that if T divides X and Y, then T also divides X mod Y.
- How is gcd(96, 36) evaluated?
Euclid's Algorithm

```c
int gcd(int m, int n)
{
    if (n == 0) return m;
    return gcd(n, m % n);
}
```

• How is gcd(96, 36) evaluated?
• $gcd(96, 36) = gcd(36, 24) = gcd(24, 12) = gcd(12, 0) = 12$. 

Recursive Vs. Non-Recursive Implementations

• In some cases, recursive functions are much easier to read.
  – They make crystal clear the mathematical structure of the algorithm.
  – To process recursive data types, such as nodes, oftentimes it is easy to write recursive functions.

• Example: \texttt{int count(link x)}
  – count how many links there are between \(x\) and the end of the list (\(x\) should be included in the count).
  – Recursive solution?
  – Base case? Recursive function?
Recursive Vs. Non-Recursive Implementations

• In some cases, recursive functions are much easier to read.
  – They make crystal clear the mathematical structure of the algorithm.
  – To process recursive data types, such as nodes, oftentimes it is easy to write recursive functions.

• Example: int count(link x)
  – count how many links there are between x and the end of the list (x should be included in the count).
  – Recursive solution? count(x) = 1 + count(x->next)
  – Base case: x = NULL. Recursive function:
    int count(link x)
    {
      if (x == NULL) return 0;
      return 1 + count(x->next);
    }
Recursive Vs. Non-Recursive Implementations

• However, any recursive function can also be written in a non-recursive way.
• Oftentimes recursive functions run slower. Why?
Recursive Vs. Non-Recursive Implementations

• However, any recursive function can also be written in a non-recursive way.

• Oftentimes recursive functions run slower. Why?
  – Recursive functions generate many function calls.
  – The CPU has to pay a price (perform a certain number of operations) for each function call.
  – Possible solution: use tail-recursion when possible (some compilers have optimizations it)

• Non-recursive implementations are oftentimes somewhat uglier (and more buggy, harder to debug) but more efficient.
  – Compromise: make first version recursive, second non-recursive.
Address the inefficiency of recursive functions
Fibonacci Numbers

• Fibonacci(0) = 0
• Fibonacci(1) = 1
• If N >= 2:
  – Fibonacci(N) = Fibonacci(N-1) + 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?

```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?
  - \( g(N) = g(N-1) + g(N-2) + \text{constant} \)
  - \( g(N) = O(\text{Fibonacci}(N)) = O(1.618^N) \)
  - We cannot even compute Fibonacci(40) in a reasonable amount of time.
  - See how many times this function is executed.

```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: remember values we have already computed.

**linear version:**

```c
int Fibonacci(int i)
{
    int * F = malloc(sizeof(int) * (i+1));
    F[0] = 0;    F[1] = 1;
    int j;
    for (j = 2; j <= i; j++) F[j] = F[j-1] + F[j-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);
}
```
Bottom-up Dynamic Programming

• The technique we have just used is called **bottom-up dynamic programming**.
• It is widely applicable, in a large variety of problems.
Bottom-up Dynamic Programming

• 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 \((P_0, P_1, P_2, ..., P_K)\) of reasonable size, so that:
    • \(P_k\) is our original problem P.
    • The initial problems, \(P_0\) and possibly \(P_1, P_2, ..., P_R\) up to some \(R\), are easy to solve (they are **base cases**).
    • For \(i > R\), each \(P_i\) can be easily solved using solutions to \(P_0, ..., P_{i-1}\).
  
• If these requirements are met, we solve problem P as follows:
  – Create the sequence of problems \(P_0, P_1, P_2, ..., P_K\), such that \(P_k = P\).
  – For \(i = 0\) to \(K\), solve \(P_K\).
  – Return solution for \(P_K\).
Bottom-up Dynamic Programming

• 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 \((P_0, P_1, P_2, ..., P_K)\) of reasonable size, so that:
    • \(P_k\) is our original problem P.
    • The initial problems, \(P_0\) and possibly \(P_1, P_2, ..., P_R\) up to some \(R\), are easy to solve (they are base cases).
    • For \(i > R\), each \(P_i\) can be easily solved using solutions to \(P_0, ..., P_{i-1}\).

• If these requirements are met, we solve problem P as follows:
  – Create the sequence of problems \(P_0, P_1, P_2, ..., P_K\), such that \(P_k = P\).
  – For \(i = 0\) to \(K\), solve \(P_k\).
  – Return solution for \(P_k\).

How can we relate all this terminology to the problem of computing Fibonacci numbers?
Dynamic Programming for Fibonacci

• Requirements for using dynamic programming:
  – The answer to our problem P can be easily obtained from answers to smaller problems. Yes! \( \text{Fib}(N) = \text{Fib}(N-1) + \text{Fib}(N-2) \)
  – We can order problems in a sequence \((P_0, P_1, P_2, ..., P_K)\) of reasonable size, so that:
    • \( P_k \) is our original problem \( P \).
    • The initial problems, \( P_0 \) and possibly \( P_1, P_2, ..., P_R \) up to some \( R \), are easy to solve (they are base cases).
    • For \( i > R \), each \( P_i \) can be easily solved using solutions to \( P_0, ..., P_{i-1} \).
  – Yes!
    • \( P_i \) is the problem of computing Fibonacci(i).
    • \( P_N \) is our problem, since we want to compute Fibonacci(N).
    • \( P_0, P_1 \) are base cases.
    • For \( i >= 2 \), Fib(i) is easy to solve given Fib(0), Fib(1), ..., Fib(i-1).
Dynamic Programming for Fibonacci

• 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 \( P_k \).

• That is exactly what this function does.

linear version:

```c
int Fibonacci(int i) {
    int * F = malloc(sizeof(int) * (i+1));
    F[0] = 0;
    F[1] = 1;
    int j;
    for (j = 2; j <= i; j++) F[j] = F[j-1] + F[j-2];
    return F[i];
}
```
Bottom-Up vs. Top Down

• When the conditions that we stated previously are satisfied, we can use dynamic programming.

• There are two versions of dynamic programming.
  – Bottom-up.
  – Top-down.

• We have already seen how bottom-up works.
  – It solves problems in sequence, from smaller to bigger.

• Top-down dynamic programming takes the opposite approach:
  – 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

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

• To solve a problem P:
  – See if the solution has already been stored in the array.

• If so, just return the solution.

• Otherwise:
  – 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.
Top-Down Solution for Fibonacci

• Textbook solution:

```c
int F(int i)
{
    int t;
    if (knownF[i] != unknown) return knownF[i];
    if (i == 0) t = 0;
    if (i == 1) t = 1;
    if (i > 1) t = F(i-1) + F(i-2);
    knownF[i] = t;
    return t;
}
```

• This is a partial solution. Initialization of `known` is not shown.
Top-Down Solution for Fibonacci

• General strategy:

• Create a top-level function that:
  – Creates memory for the array of solutions.
  – Initializes the array by marking that all solutions are currently "unknown".
  – Calls a helper function, that takes the same arguments, plus the solutions array.

• The helper function:
  – If the solution it wants is already computed, returns the solution.
  – If we have a base case, computes the result directly.
  – Otherwise: computes the result using recursive calls.
  – Stores the result in the solutions array.
  – Returns the result.

• How do we write these two functions for Fibonacci?
int Fibonacci(int number)
{
    // Creating memory for the array of solutions.
    int * solutions = malloc(sizeof(int) * (number +1));
    int index;

    // Marking the solutions to all cases as "unknown".
    // We use the convention that -1 stands for "unknown".
    for (index = 0; index <= number; index++)  solutions[index] = -1;

    int result = FibHelper(number, solutions);
    free(solutions);
    return result;
}
Helper Function

```c
int FibHelper(int N, int * solutions)
{
    // if problem already solved, return stored solution.
    if (solutions[N] != -1) return solutions[N];
    int result;

    if (N == 0) result = 0;  // base case
    else if (N == 1) result = 1;  // base case

    // recursive case
    else result = FibHelper(N-1, solutions) + FibHelper(N-2, solutions);

    solutions[N] = result;  // memoization
    return result;
}
```
The Fibonacci numbers are just a toy example for dynamic programming, as they can be computed with a simple for loop.

The classic problem for introducing dynamic programming is the knapsack problem.

- A thief breaks in at the store.
- The thief can only carry out of the store items with a total weight of $W$.
- 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 items that the thief can carry out?
- What items should the thief carry out to obtain this maximum value?

We will make two important assumptions:

- That the store has unlimited quantities of each item type.
- That the weight of each item is an integer $\geq 1$. 
Example

<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>5</td>
<td>10</td>
<td>11</td>
<td>13</td>
</tr>
</tbody>
</table>

- For example, suppose that the table above describes the types of items available at the store.
- Suppose that the thief can carry out a maximum weight of 17.
- What are possible combinations of items that the thief can carry out?
  - Two A's, a B, and a C: weight = 17, value = 23.
  - A D and an E: weight = 17, value = 24.
- The question is, what is the best combination?
Solving the Knapsack Problem

• Can you propose any algorithm (even horribly slow) for finding the best combination?

• One approach: consider all possible sets of items.
  – Would that work?
Solving the Knapsack Problem

• One approach: consider all possible sets of items.

• Would that work? NO!!!
  – We have unlimited quantities of each item.
  – The number of all possible sets is infinite, so it takes infinite time to consider them (if we do not use the weight limit constraint).

• An algorithm that takes infinite time IS NOT THE SAME THING as an algorithm that is horribly slow.
  – Horribly slow algorithms eventually terminate, so mathematically they are valid solutions.
  – Algorithms that take infinite time never terminate, so they are mathematically not valid solutions.
Solving the Knapsack Problem

• To use dynamic programming, we need to identify whether solving our problem can be done easily if we have already solved smaller problems.

• What would be a smaller problem?
  – Our original problem is: find the set of items with weight $\leq W$ that has the most value.
Solving the Knapsack Problem

• To use dynamic programming, we need to identify whether solving our problem can be done easily if we have already solved smaller problems.

• What would be a smaller problem?
  – Our original problem is: find the set of items with weight $\leq W$ that has the most value.

• A smaller problem is: find the set of items with weight $\leq W'$ that has the most value, where $W' < W$.

• If we have solved the problem for all $W' < W$, how can we use those solutions to solve the problem for $W$?
Solving the Knapsack Problem

• Our original problem is: find the set of items with weight <= W that has the most value.
• A smaller problem is: find the set of items with weight <= W' that has the most value, where W' < W.
• If we have solved the problem for all W' < W, how can we use those solutions to solve the problem for W?

```c
int knap(int W, int * weights, int * values):
{
    max_value = 0;
    For each type of item i:
        value = values[i] + knap(W - weights[i]);
        if (value > max_value) max_value = value.
    return max_value;
}
```
How Does This Work?

- We want to compute: knap(17).
- knap(17) can be computed from which values?
  - val_A = ???
  - val_B = ???
  - val_C = ???
  - val_D = ???
  - val_E = ???

```c
int knap(int W, int * weights, int * values):
{
    max_value = 0;
    For each type of item i:
        value = values[i] + knap(W - weights[i]);
        if (value > max_value)
            max_value = value;
    return max_value;
}
```

<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>5</td>
<td>10</td>
<td>11</td>
<td>13</td>
</tr>
</tbody>
</table>
How Does This Work?

- We want to compute: knap(17).
- knap(17) will be the maximum of these five values:
  - val_A = 4 + knap(14)
  - val_B = 5 + knap(13)
  - val_C = 10 + knap(10)
  - val_D = 11 + knap(9)
  - val_E = 13 + knap(8)

```c
int knap(int W, int * weights, int * values):
{
    max_value = 0;
    For each type of item i:
        value = values[i] + knap(W - weights[i]);
        if (value > max_value)
            max_value = value;
    return max_value;
}
```
Recursive Solution for Knapsack

pseudocode:

```c
int knap(int W, int *weights, int *values):
{
    max_value = 0;
    For each type of item i:
        value = values[i] + knap(W - weights[i], weights, values);
        if (value > max_value)
            max_value = value;
    return max_value;
}
```

What is missing from this pseudocode if we want a complete solution?
Recursive Solution for Knapsack

pseudocode:

```c
int knap(int W, int * weights, int * values):
{
    max_value = 0;
    For each type of item i:
        value = values[i] + knap(W - weights[i], weights, values);
        if (value > max_value)
            max_value = value;
    return max_value;
}
```

What is missing from this pseudocode if we want a complete solution? The base case: `knap(0) = 0`
Recursive Solution for Knapsack

```c
struct Items {
    int number;
    char ** types;
    int * weights;
    int * values;
};

int knapsack(int max_weight, struct Items items) {
    if (max_weight == 0) return 0;
    int max_value = 0;
    int i;
    for (i = 0; i < items.number; i++) {
        int rem = max_weight - items.weights[i];
        if (rem < 0) continue;
        int value = items.values[i] + knapsack(rem, items);
        if (value > max_value) max_value = value;
    }
    return max_value;
}
```
Recursive Solution for Knapsack

```c
int knapsack(int max_weight, struct Items items) {
    if (max_weight == 0) return 0;
    int max_value = 0;
    int i;
    for (i = 0; i < items.number; i++) {
        int rem = max_weight - items.weights[i];
        if (rem < 0) continue;
        int value = items.values[i] + knapsack(rem, items);
        if (value > max_value) max_value = value;
    }
    return max_value;
}
```

running time?
Recursive Solution for Knapsack

int knapsack(int max_weight, struct Items items)
{
    if (max_weight == 0) return 0;
    int max_value = 0;
    int i;
    for (i = 0; i < items.number; i++)
    {
        int rem = max_weight - items.weights[i];
        if (rem < 0) continue;
        int value = items.values[i] + knapsack(rem, items);
        if (value > max_value) max_value = value;
    }
    return max_value;
}
Bottom-Up Dynamic Programming for the Knapsack Problem

• 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 **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 \( P_K \).

How can we relate all this terminology to the Knapsack Problem?
Bottom-Up Dynamic Programming for the Knapsack Problem

• Requirements for using dynamic programming:
  – The answer to our problem P can be easily obtained from answers to smaller problems. Yes! Knapsack(W) uses answers for W-1, W-2, ..., W-max_weight.
  – We can order problems in a sequence (P₀, P₁, P₂, ..., Pₖ) of reasonable size, so that:
    • Pₖ is our original problem P.
    • The initial problems, P₀ and possibly P₁, P₂, ..., Pᵣ up to some R, are easy to solve (they are base cases).
    • For i > R, each Pᵢ can be easily solved using solutions to P₀, ..., Pᵢ₋₁.
  – Yes!
    • Pᵢ is the problem of computing Knapsack(i).
    • Pₘ is our original problem, since we want to compute Knapsack (W).
    • P₀ is the base case.
    • For i >= 1, Knapsack(i) is easy to solve given Knapsack (0), Knapsack(1), ..., Knapsack(i-1).
Bottom-Up Solution

int knapsack(int max_weight, Items items)

• Create array of solutions.
• Base case: solutions[0] = 0.
• For each weight in \{1, 2, ..., max_weight\}
  – max_value = 0.
  – For each item in items:
    • remainder = weight - item.weight.
    • if (remainder < 0) continue;
    • value = item.value + solutions[remainder].
    • If (value > max_value) max_value = value.
• Return solutions[max_weight].
Top-Down Solution

Top-level function (almost identical to the top-level function for Fibonacci top-down solution):

```c
int knapsack(int max_weight, Items items)
```

- Create array of solutions.
- Initialize all values in solutions to "unknown".
- `result = helper_function(max_weight, items, solutions)`
- Free up the array of solutions.
- Return result.
int helper_function(int weight, Items items, int * solutions)

• // Check if this problem has already been solved.
  • if (solutions[weight] !="unknown") return solutions[weight].
• If (weight == 0) result = 0. // Base case
• Else:
  – result = 0.
  – For each item in items:
    • remainder = weight - item.weight.
    • if (remainder < 0) continue;
    • value = item.value + helper_function(remainder, items, solutions).
    • If (value > result) result = value.
• solutions[weight] = result. // Memoization
• Return result.
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.
Limitation of All Three Solutions

- Each of the solutions returns a number.
- Is a single number all we want to answer our original problem?
Limitation of All Three Solutions

• Each of the solutions returns a number.
• Is a single number all we want to answer our original problem?
  — No. Our original problem was to find the best set of items.
  — It is nice to know the best possible value we can achieve.
  — But, we also want to know the actual set of items that achieves that value.
• This will be left as a homework for you.
Weighted Interval Scheduling (WIS)

• Suppose you are a plumber.
• You are offered N jobs.
• Each job has the following attributes:
  – **start**: the start time of the job.
  – **finish**: the finish time of the job.
  – **value**: the amount of money you get paid for that job.
• What is the best set of jobs you can take up?
  – You want to make the most money possible.
• Why can't you just take up all the jobs?
Weighted Interval Scheduling (WIS)

• Suppose you are a plumber.
• You are offered N jobs.
• Each job has the following attributes:
  – **start**: the start time of the job.
  – **finish**: the finish time of the job.
  – **value**: the amount of money you get paid for that job.
• What is the best set of jobs you can take up?
  – You want to make the most money possible.
• Why can't you just take up all the jobs?
• Because you cannot take up two jobs that are overlapping.
Example WIS Input

- We assume, for simplicity, that jobs have been sorted in ascending order of the finish time.
  - We have not learned yet good methods for sorting that we can use.
- If we take job A, we cannot take any other job that starts BEFORE job A finishes.
- Can we do both job 0 and job 1?
- Can we do both job 0 and job 2?

<table>
<thead>
<tr>
<th>job ID</th>
<th>start</th>
<th>finish</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4.5</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5.3</td>
<td>6.1</td>
<td>5.5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7.2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>0.5</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>12.5</td>
<td>4.5</td>
</tr>
<tr>
<td>6</td>
<td>8.2</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>15.3</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>10.5</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>17.5</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>19</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>20.5</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>17</td>
<td>23</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>20.2</td>
<td>24.1</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>19</td>
<td>25</td>
<td>10</td>
</tr>
</tbody>
</table>
### Example WIS Input

- We assume, for simplicity, that jobs have been sorted in ascending order of the finish time.
  - We have not learned yet good methods for sorting that we can use.
- If we take job A, we cannot take any other job that starts BEFORE job A finishes.
- Can we do both job 0 and job 1?
  - Yes.
- Can we do both job 0 and job 2?
  - No (they overlap).

<table>
<thead>
<tr>
<th>job ID</th>
<th>start</th>
<th>finish</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4.5</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5.3</td>
<td>6.1</td>
<td>5.5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7.2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>0.5</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>12.5</td>
<td>4.5</td>
</tr>
<tr>
<td>6</td>
<td>8.2</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>15.3</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>10.5</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>17.5</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>19</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>20.5</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>17</td>
<td>23</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>20.2</td>
<td>24.1</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>19</td>
<td>25</td>
<td>10</td>
</tr>
</tbody>
</table>
Example WIS Input

- A possible set of jobs we could take: 0, 1, 5, 10, 13.
- What is the value?
  - $3 + 5.5 + 4.5 + 6 + 6 = 25$.
- Can you propose any algorithm (even horribly slow) for finding the best set of jobs?

<table>
<thead>
<tr>
<th>job ID</th>
<th>start</th>
<th>finish</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4.5</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5.3</td>
<td>6.1</td>
<td>5.5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7.2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>0.5</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>12.5</td>
<td>4.5</td>
</tr>
<tr>
<td>6</td>
<td>8.2</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>15.3</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>10.5</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>17.5</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>19</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>20.5</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>17</td>
<td>23</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>20.2</td>
<td>24.1</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>19</td>
<td>25</td>
<td>10</td>
</tr>
</tbody>
</table>
Example WIS Input

• Simplest algorithm for finding the best subset of jobs:
  – Consider all possible subsets of jobs.
  – Ignore subsets with overlapping jobs.
  – Find the subset with the best total value.

• Time complexity? If we have N jobs, what is the total number of subsets of jobs?

<table>
<thead>
<tr>
<th>job ID</th>
<th>start</th>
<th>finish</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4.5</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5.3</td>
<td>6.1</td>
<td>5.5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7.2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>0.5</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>12.5</td>
<td>4.5</td>
</tr>
<tr>
<td>6</td>
<td>8.2</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>15.3</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>10.5</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>17.5</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>19</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>20.5</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>17</td>
<td>23</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>20.2</td>
<td>24.1</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>19</td>
<td>25</td>
<td>10</td>
</tr>
</tbody>
</table>
Example WIS Input

• Simplest algorithm for finding the best subset of jobs:
  – Consider all possible subsets of jobs.
  – Ignore subsets with overlapping jobs.
  – Find the subset with the best total value.

• Time complexity? If we have N jobs, what is the total number of subsets of jobs?
  – Total number of subsets: \(2^N\).
  – Exponential time complexity.

<table>
<thead>
<tr>
<th>job ID</th>
<th>start</th>
<th>finish</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4.5</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5.3</td>
<td>6.1</td>
<td>5.5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7.2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>0.5</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>12.5</td>
<td>4.5</td>
</tr>
<tr>
<td>6</td>
<td>8.2</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>15.3</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>10.5</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>17.5</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>19</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>20.5</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>17</td>
<td>23</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>20.2</td>
<td>24.1</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>19</td>
<td>25</td>
<td>10</td>
</tr>
</tbody>
</table>
Solving WIS With Dynamic Programming

- To use dynamic programming, we must relate the solution to our problem to solutions to smaller problems.
- For example, consider job 14.
- What kind of problems that exclude job 14 would be relevant in solving the original problem, that includes job 14?

<table>
<thead>
<tr>
<th>job ID</th>
<th>start</th>
<th>finish</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4.5</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5.3</td>
<td>6.1</td>
<td>5.5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7.2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>0.5</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>12.5</td>
<td>4.5</td>
</tr>
<tr>
<td>6</td>
<td>8.2</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>15.3</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>10.5</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>17.5</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>19</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>20.5</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>17</td>
<td>23</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>20.2</td>
<td>24.1</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>19</td>
<td>25</td>
<td>10</td>
</tr>
</tbody>
</table>
Solving WIS With Dynamic Programming

• We can easily solve the problem for jobs 0-14, given solutions to these two smaller problems:
    – When job 14 is available, the best set using jobs 0-13 is still an option to us, although not necessarily the best one.
  • Problem 2: best set using jobs 0-10.
    – Why is this problem relevant?

<table>
<thead>
<tr>
<th>job ID</th>
<th>start</th>
<th>finish</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4.5</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5.3</td>
<td>6.1</td>
<td>5.5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7.2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>0.5</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>12.5</td>
<td>4.5</td>
</tr>
<tr>
<td>6</td>
<td>8.2</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>15.3</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>10.5</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>17.5</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>19</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>20.5</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>17</td>
<td>23</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>20.2</td>
<td>24.1</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>19</td>
<td>25</td>
<td>10</td>
</tr>
</tbody>
</table>
Solving WIS With Dynamic Programming

• We can easily solve the problem for jobs 0-14, given solutions to these two smaller problems:
    – When job 14 is available, the best set using jobs 0-13 is still an option to us, although not necessarily the best one.
  • Problem 2: best set using jobs 0-10.
    – Why is this problem relevant?
    – Because job 10 is the last job before job 14 that does NOT overlap with job 14.
    – Thus, job 14 can be ADDED to the solution for jobs 0-10.

<table>
<thead>
<tr>
<th>job ID</th>
<th>start</th>
<th>finish</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4.5</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5.3</td>
<td>6.1</td>
<td>5.5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7.2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>0.5</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>12.5</td>
<td>4.5</td>
</tr>
<tr>
<td>6</td>
<td>8.2</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>15.3</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>10.5</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>17.5</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>19</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>20.5</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>17</td>
<td>23</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>20.2</td>
<td>24.1</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>19</td>
<td>25</td>
<td>10</td>
</tr>
</tbody>
</table>
Solving WIS With Dynamic Programming

- We can easily solve the problem for jobs 0-14, given solutions to these two smaller problems:
  - Problem 2: best set using jobs 0-10.
- The solution for jobs 0-14 is simply the best of these two options:
  - Best set using jobs 0-10, plus job 14.
- How can we write this solution in pseudocode?
Solving WIS With Dynamic Programming

- **Step 1:** to make our life easier, we will insert a zero job at the beginning. The zero job:
  - Starts at time zero
  - Finishes at time zero.
  - Has zero value.

- **Step 2:** we need to preprocess jobs, so that for each job i we compute:
  - **last [i]** = the index of the last job preceding job i that does NOT overlap with job i.

<table>
<thead>
<tr>
<th>job ID</th>
<th>start</th>
<th>finish</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4.5</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5.3</td>
<td>6.1</td>
<td>5.5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>7.2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>0.5</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>12.5</td>
<td>4.5</td>
</tr>
<tr>
<td>7</td>
<td>8.2</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>15.3</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>10.5</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>17.5</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>19</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>16</td>
<td>20.5</td>
<td>8</td>
</tr>
<tr>
<td>13</td>
<td>17</td>
<td>23</td>
<td>12</td>
</tr>
<tr>
<td>14</td>
<td>20.2</td>
<td>24.1</td>
<td>6</td>
</tr>
<tr>
<td>15</td>
<td>19</td>
<td>25</td>
<td>10</td>
</tr>
</tbody>
</table>
Solving WIS With Dynamic Programming

- **Step 1**: to make our life easier, we will insert a zero job at the beginning. The zero job:
  - Starts at time zero
  - Finishes at time zero.
  - Has zero value.

- **Step 2**: we need to preprocess jobs, so that for each job i we compute:
  - \textbf{last [i]} = the index of the last job preceding job i that does NOT overlap with job i.
Solving WIS With Dynamic Programming

float wis(jobs, last)

- N = number of jobs.
- Initialize solutions array.
- solutions[0] = 0.
- For (i = 1 to N)
  - S1 = solutions[i-1].
  - L = last[i].
  - SL = solutions[L].
  - S2 = SL + jobs[i].value.
  - solutions[i] = max(S1, S2).
- Return solutions[N];
Backtracking

• As in our solution to the knapsack problem, the pseudocode we just saw returns a number:
  – The best total value we can achieve.
• In addition to the best value, we also want to know the set of jobs that achieves that value.
• This is a general issue in dynamic programming.
• How can we address it?
Backtracking

• As in our solution to the knapsack problem, the pseudocode we just saw returns a number:
  – The best total value we can achieve.
• In addition to the best value, we also want to know the set of jobs that achieves that value.
• This is a general issue in dynamic programming.
• There is a general solution, called backtracking.
• The key idea is:
  – In DP the final solution is always built from smaller solutions.
  – At each smaller problem, we have to choose which (even smaller) solutions to use for solving that problem.
  – We must record, for each smaller problem, the choice we made.
  – At the end, we backtrack and recover the individual decisions that led to the best solution.
Backtracking for the WIS Solution

• First of all, what should the function return?
Backtracking for the WIS Solution

• First of all, what should the function return?
  – The best value we can achieve.
  – The set of intervals that achieves that value.
• How can we make the function return both these things?
• The solution that will be preferred throughout the course:
  – Define a Result structure containing as many member variables as we need to store in the result.
  – Make the function return an object of that structure.
Backtracking for the WIS Solution

• First of all, what should the function return?
  – The best value we can achieve.
  – The set of intervals that achieves that value.

```c
struct WIS_result
{
    float value;
    list set;
};

struct WIS_result wis(struct Intervals intervals)
```
Backtracking Solution

Result wis(jobs, last)

• $N = \text{number of jobs}$.
• $\text{solutions}[0] = 0$.
• For ($i = 1$ to $N$)
  – $L = \text{last}[i]$.
  – $SL = \text{solutions}[L]$.
  – $S1 = \text{solutions}[i-1]$.
  – $S2 = SL + \text{jobs}[i].value$.
  – $\text{solutions}[i] = \max(S1, S2)$.

• How can we keep track of the decisions we make?

<table>
<thead>
<tr>
<th>last</th>
<th>job ID</th>
<th>start</th>
<th>finish</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4.5</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>5.3</td>
<td>6.1</td>
<td>5.5</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>3</td>
<td>7.2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>0.5</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>7</td>
<td>12.5</td>
<td>4.5</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>8.2</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>9</td>
<td>15.3</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>10.5</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>9</td>
<td>17.5</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>13</td>
<td>19</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>12</td>
<td>16</td>
<td>20.5</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>13</td>
<td>17</td>
<td>23</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
<td>20.2</td>
<td>24.1</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>15</td>
<td>19</td>
<td>25</td>
<td>10</td>
</tr>
</tbody>
</table>
Backtracking Solution

Result wis(jobs, last)

- \( N \) = number of jobs.
- \( \text{solutions}[0] = 0. \)
- For (\( i = 1 \) to \( N \))
  - \( L = \text{last}[i]. \)
  - \( SL = \text{solutions}[L]. \)
  - \( S1 = \text{solutions}[i-1]. \)
  - \( S2 = SL + \text{jobs}[i].\text{value}. \)
  - \( \text{solutions}[i] = \max(S1, S2). \)

- How can we keep track of the decisions we make?
- Remember the last job of each solution.
Backtracking Solution

Result \text{wis}(\text{jobs, last})

- N = number of jobs.
- \text{solutions}[0] = 0.
- \text{used}[0] = 0.
- For (i = 1 to N)
  - L = last[i].
  - SL = \text{solutions}[L].
  - S1 = \text{solutions}[i-1].
  - S2 = SL + \text{jobs}[i].value.
  - \text{solutions}[i] = \max(S1, S2).
  - If S2 > S1 then used[i] = i.
  - Else used[i] = used[i-1].
Backtracking Solution

• // backtracking part
• list set = new List.
• counter = used[N].
• while(counter != 0)
  – job = jobs[counter].
  – insertAtBeginning(set, job).
  – counter = ???
• WIS_result result.
• result.value = solutions[N].
• result.set = set.
• return result.
Backtracking Solution

- // backtracking part
- list set = new List.
- counter = used[N].
- while(counter != 0)
  - job = jobs[counter].
  - insertAtBeginning(set, job).
  - counter = used[last[counter]].
- WIS_result result.
- result.value = solutions[N].
- result.set = set.
- return result.

<table>
<thead>
<tr>
<th>last</th>
<th>job ID</th>
<th>start</th>
<th>finish</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4.5</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>5.3</td>
<td>6.1</td>
<td>5.5</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>3</td>
<td>7.2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>0.5</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>7</td>
<td>12.5</td>
<td>4.5</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>8.2</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>9</td>
<td>15.3</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>10.5</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>9</td>
<td>17.5</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>13</td>
<td>19</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>12</td>
<td>16</td>
<td>20.5</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>13</td>
<td>17</td>
<td>23</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
<td>20.2</td>
<td>24.1</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>15</td>
<td>19</td>
<td>25</td>
<td>10</td>
</tr>
</tbody>
</table>
Matrix Multiplication: Review

- Suppose that $A_1$ is of size $S_1 \times S_2$, and $A_2$ is of size $S_2 \times S_3$.
- What is the time complexity of computing $A_1 \times A_2$?
- What is the size of the result?
Matrix Multiplication: Review

• Suppose that $A_1$ is of size $S_1 \times S_2$, and $A_2$ is of size $S_2 \times S_3$.
• What is the time complexity of computing $A_1 \times A_2$?
• What is the size of the result? $S_1 \times S_3$.
• Each number in the result is computed in $O(S_2)$ time by:
  – multiplying $S_2$ pairs of numbers.
  – adding $S_2$ numbers.
• Overall time complexity: $O(S_1 \times S_2 \times S_3)$. 
Optimal Ordering for Matrix Multiplication

• Suppose that we need to do a sequence of matrix multiplications:
  – result = \(A_1 \times A_2 \times A_3 \times ... \times A_K\)

• The number of columns for \(A_i\) must equal the number of rows for \(A_{i+1}\).

• What is the time complexity for performing this sequence of multiplications?
Optimal Ordering for Matrix Multiplication

• Suppose that we need to do a sequence of matrix multiplications:
  – result = $A_1 \times A_2 \times A_3 \times \ldots \times A_K$

• The number of columns for $A_i$ must equal the number of rows for $A_{i+1}$.

• What is the time complexity for performing this sequence of multiplications?

• The answer is: it depends on the order in which we perform the multiplications.
An Example

• Suppose:
  – \( A_1 \) is 17x2.
  – \( A_2 \) is 2x35.
  – \( A_3 \) is 35x4.
• \((A_1 \times A_2) \times A_3\):

• \( A_1 \times (A_2 \times A_3)\):
An Example

• Suppose:
  – $A_1$ is 17x2.
  – $A_2$ is 2x35.
  – $A_3$ is 35x4.

• $(A_1 * A_2) * A_3$:
  – $17*2*35 = 1190$ multiplications and additions to compute $A_1 * A_2$.
  – $17*35*4 = 2380$ multiplications and additions to compute multiplying the result of $(A_1 * A_2)$ with $A_3$.
  – Total: 3570 multiplications and additions.

• $A_1 * (A_2 * A_3)$:
  – $2*35*4 = 280$ multiplications and additions to compute $A_2 * A_3$.
  – $17*2*4 = 136$ multiplications and additions to compute multiplying $A_1$ with the result of $(A_2 * A_3)$.
  – Total: 416 multiplications and additions.
Adaptation to Dynamic Programming

- Suppose that we need to do a sequence of matrix multiplications:
  - result = $A_1 \times A_2 \times A_3 \times ... \times A_k$

- To figure out if and how we can use dynamic programming, we must address the standard two questions we always need to address for dynamic programming:

1. Can we define a set of smaller problems, such that the solutions to those problems make it easy to solve the original problem?

2. Can we arrange those smaller problems in a sequence of reasonable size, so that each problem in that sequence only depends on problems that come earlier in the sequence?
Defining Smaller Problems

1. Can we define a set of smaller problems, whose solutions make it easy to solve the original problem?
   - Original problem: optimal ordering for $A_1 * A_2 * A_3 * ... * A_K$

• Yes! Suppose that, for every $i$ between 1 and $K-1$ we know:
  - The best order (and best cost) for multiplying matrices $A_1, ..., A_i$.
  - The best order (and best cost) for multiplying matrices $A_{i+1}, ..., A_K$.

• Then, for every such $i$, we obtain a possible solution for our original problem:
  - Multiply matrices $A_1, ..., A_i$ in the best order. Let $C_1$ be the cost of that.
  - Multiply matrices $A_{i+1}, ..., A_K$ in the best order. Let $C_2$ be the cost of that.
  - Compute $(A_1 * ... * A_i) * (A_{i+1} * ... * A_K)$. Let $C_3$ be the cost of that.
    - $C_3 = \text{rows of } (A_1 * ... * A_i) \times \text{cols of } (A_1 * ... * A_i) \times \text{cols of } (A_{i+1} * ... * A_K)$.
      - $= \text{rows of } A_1 \times \text{cols of } A_i \times \text{cols of } A_K$
  - Total cost of this solution = $C_1 + C_2 + C_3$. 


Defining Smaller Problems

1. Can we define a set of smaller problems, whose solutions make it easy to solve the original problem?
   – Original problem: optimal ordering for $A_1 \times A_2 \times A_3 \times \ldots \times A_K$
   • Yes! Suppose that, for every $i$ between 1 and $K-1$ we know:
     – The best order (and best cost) for multiplying matrices $A_1, \ldots, A_i$.
     – The best order (and best cost) for multiplying matrices $A_{i+1}, \ldots, A_K$.
   • Then, for every such $i$, we obtain a possible solution.
   • We just need to compute the cost of each of those solutions, and choose the smallest cost.
   • Next question:

2. Can we arrange those smaller problems in a sequence of reasonable size, so that each problem in that sequence only depends on problems that come earlier in the sequence?
Defining Smaller Problems

2. Can we arrange those smaller problems in a sequence of reasonable size, so that each problem in that sequence only depends on problems that come earlier in the sequence?

- To compute answer for $A_1 \times A_2 \times A_3 \times \ldots \times A_K$:
  For $i = 1, \ldots, K-1$, we had to consider solutions for:
  - $A_1, \ldots, A_i$.
  - $A_{i+1}, \ldots, A_K$.

- So, what is the set of all problems we must solve?


Defining Smaller Problems

2. Can we arrange those smaller problems in a sequence of reasonable size, so that each problem in that sequence only depends on problems that come earlier in the sequence?

• To compute answer for $A_1 \times A_2 \times A_3 \times \ldots \times A_k$:
  For $i = 1, \ldots, K-1$, we had to consider solutions for:
  – $A_1, \ldots, A_i$.
  – $A_{i+1}, \ldots, A_K$.

• So, what is the set of all problems we must solve?

• For $M = 1, \ldots, K$.
  – For $N = 1, \ldots, M$.
    • Compute the best ordering for $A_N \times \ldots \times A_M$.

• What is the number of problems we need to solve? Is the size reasonable?
  – We must solve $\Theta(K^2)$ problems. We consider this a reasonable number.
Defining Smaller Problems

• The set of all problems we must solve:
  • For $M = 1, ..., K$.
    – For $N = 1, ..., M$.
      • Compute the best ordering for $A_N \cdot ... \cdot A_M$.
  • What is the order in which we must solve these problems?
Defining Smaller Problems

• The set of all problems we must solve, in the correct order:
  • For $M = 1, ..., K$.
    – For $N = M, ..., 1$.
      • Compute the best ordering for $A_N \times ... \times A_M$.
  • $N$ must go from $M$ to 1, NOT the other way around.
• Why? Because, given $M$, the larger the $N$ is, the smaller the problem is of computing the best ordering for $A_N \times ... \times A_M$. 

Solving These Problems

• For $M = 1, \ldots, K$.
  – For $N = M, \ldots, 1$.
    • Compute the best ordering for $A_N \times \cdots \times A_M$.

• What are the base cases?
  • $N = M$.
    – $\text{costs}[N][M] = 0$.
  • $N = M - 1$.
    – $\text{costs}[N][M] = \text{rows}(A_N) \times \text{cols}(A_N) \times \text{cols}(A_M)$.

• Solution for the recursive case:
Solving These Problems

• For $M = 1, \ldots, K$.
  – For $N = M, \ldots, 1$.
    • Compute the best ordering for $A_N \ast \ldots \ast A_M$.

• Solution for the recursive case:

  • minimum_cost = 0
  • For $R = N, \ldots, M-1$:
    – cost1 = costs[N][R]
    – cost2 = costs[R+1][M]
    – cost3 = rows($A_N$) * cols($A_R$) * cols($A_M$)
    – cost = cost1 + cost2 + cost3
    – if (cost < minimum_cost) minimum_cost = cost

• costs[N][M] = minimum_cost
The Edit Distance

• Suppose A and B are two strings.
• By applying insertions, deletions, and substitutions, we can always convert A to B.
• Insertion example: we insert an 'r' at position 2, to convert "cat" to "cart".
• Deletion example: we delete the 'r' at position 2, to convert "cart" to "cat".
• Substitution example: we replace the 'o' at position 1 with an 'i', to convert "dog" to "dig".
• Note: each insertion/deletion/substitution inserts, deletes, or changes only one character, NOT multiple characters.
The Edit Distance

• For example, to convert "chicken" to "ticket":
• One solution:
  – Substitute 'c' with 't'.
  – Delete 'h'.
  – Replace 'n' with 't'.
  – Total: three operations.
• Another solution:
  – Delete 'c'.
  – Substitute 'h' with 't'.
  – Replace 'n' with 't'.
  – Total: three operations.
The Edit Distance

• Question: given two strings A and B, what is the smallest number of operations we need in order to convert A to B?
• The answer is called the *edit distance* between A and B.
• This distance, and variations, have significant applications in various fields, including bioinformatics and pattern recognition.
Visualizing the Edit Distance

• Edit distance between "chicken" and "ticket" = ?
Visualizing the Edit Distance

• Edit distance between "chicken" and "ticket" = 3

chicken

ticket

 xxx .... x

• Three operations:
  – Substitution: 'c' with 't'.
  – Insertion: 'h'.
  – Substitution: 'n' with 't'.

Visualizing the Edit Distance

• Edit distance between "lazy" and "crazy" = ?
Visualizing the Edit Distance

• Edit distance between "lazy" and "crazy" = 2

\[ l - a z y \]
\[ c r a z y \]
\[ x x . . . \]

• Two operations:
  – Substitution: 'l' with 'c'.
  – Insertion: 'r'.

Visualizing the Edit Distance

• Edit distance between "intimidation" and "immigration" = ?
Visualizing the Edit Distance

• Edit distance between "intimidation" and "immigration" = 5

intimidation
---immigration
.xxx.x...xx........

• Five operations:
  – Deletion: 'n'.
  – Deletion: 't'.
  – Substitution: 'i' with 'm'.
  – Substitution: 'd' with 'g'.
  – Insertion: 'r'.
Computing the Edit Distance

• What is the edit distance between:
  – GATTACACCGTCTCGGGCATCCATAATGG
  – CATTATAGGTGAACTTGCGCGTTATGC

• Unlike previous examples, here the answer is not obvious.
• The two strings above are (very small) examples of DNA sequences, using the four DNA letters: ACGT.
• In practice, the sequences may have thousands or millions of letters.
• We need an algorithm for computing the edit distance between two strings.
Computing the Edit Distance

• To find a dynamic programming solution, we must find a sequence of problems such that:
  – Each problem in the sequence can be easily solved given solutions to the previous problems.
  – The number of problems in the sequence is not too large (e.g., not exponential).

• Any ideas?

• Given strings A and B, can you identify smaller problems that are related to computing the edit distance between A and B?
Computing the Edit Distance

• Notation:
  – $S[i, ..., j]$ is the substring of $S$ that includes all letters from position $i$ to position $j$.
  – $|S|$ indicates the length of string $S$.

• Using this notation:
  – $A = A[0, ..., |A|-1]$
  – $B = B[0, ..., |B|-1]$

• The solution for $edit\_distance(A, B)$ depends on the solutions to three smaller problems:
  – $edit\_distance(A[0, ..., |A|-1], B[0, ..., |B|-2])$
  – $edit\_distance(A[0, ..., |A|-2], B[0, ..., |B|-1])$
  – $edit\_distance(A[0, ..., |A|-2], B[0, ..., |B|-2])$
Computing the Edit Distance

• The solution for edit_distance(A, B) depends on the solutions to three smaller problems:
  • Problem 1: edit_distance(A[0, ..., |A|-1], B[0, ..., |B|-2])
    – Edit distance from A to B, excluding the last letter of B.
    – We can insert the last letter of B to that solution.
• Example:
  – A = "intimidation". \(|A| = 12.\)
  – B = "immigration". \(|B| = 11.\)
• edit_distance(A[0, ..., 11], B[0, ..., 9]) = 6
  intimid\_ation
  i---m\_migration
• From this, we obtain a solution with cost 7.
Computing the Edit Distance

• Problem 2: edit_distance(A[0, ..., |A|-2], B[0, ..., |B|-1])
  – Edit distance from A to B, excluding the last letter of A.
  – We can insert the last letter of A to that solution.

• Example:
  – A = "intimidation". |A| = 12.
  – B = "immigration". |B| = 11.

• edit_distance(A[0, ..., 10], B[0, ..., 10]) = 6

  intimidation

  immigration

• This solution converts "intimidation" to "immigration".

• Using one more deletion (of the final 'n' of "intimidation"), we convert "intimidation" to "immigration" with cost 7.
Computing the Edit Distance

• Problem 3: edit_distance(A[0, ..., |A|-2], B[0, ..., |B|-2])
  – Edit distance from A to B, excluding the last letter of both A and B.
• Example:
  – A = "intimidation". |A| = 12.
  – B = "immigration". |B| = 11.
• edit_distance(A[0, ..., 10], B[0, ..., 9]) = 5
  inte\textcolor{red}{m}id\textcolor{red}{i}atio\textcolor{red}{n} to \textcolor{red}{im}\textcolor{red}{m}igratio\textcolor{red}{n}
• This solution converts "intimidatio" to "immigratio".
• The same solution converts "intimidation" to "immigration", because both words have the same last letter.
Computing the Edit Distance

• Problem 3: edit_distance(A[0, ..., |A|-2], B[0, ..., |B|-2])
  – Edit distance from A to B, excluding the last letter of both A and B.

• Example:
  – A = "nation". |A| = 6.
  – B = "patios". |B| = 6.

• edit_distance(A[0, ..., 10], B[0, ..., 9]) = 1

n a t i o

p a t i o

• This solution converts "natio" to "patio".

• The same solution, plus one substitution ('n' with 's') converts "nation" to "patios", with cost 2.
Computing the Edit Distance

• Summary: \( \text{edit\_distance}(A, B) \) is the smallest of the following three:
  - 1: \( \text{edit\_distance}(A[0, ..., |A|-1], B[0, ..., |B|-2]) + ? \)
  - 2: \( \text{edit\_distance}(A[0, ..., |A|-2], B[0, ..., |B|-1]) + ? \)
  - 3: \( \text{edit\_distance}(A[0, ..., |A|-2], B[0, ..., |B|-2]) + ? \)
Computing the Edit Distance

• Summary: edit_distance(A, B) is the smallest of the following three:
  – 1: edit_distance(A[0, ..., |A|-1], B[0, ..., |B|-2]) + 1
  – 2: edit_distance(A[0, ..., |A|-2], B[0, ..., |B|-1]) + 1
  – 3: either edit_distance(A[0, ..., |A|-2], B[0, ..., |B|-2]).
    • If the last letter of A is the same as the last letter of B.
    – or edit_distance(A[0, ..., |A|-2], B[0, ..., |B|-2]) + 1.
      • If the last letter of A is not the same as the last letter of B.
Computing the Edit Distance

• What sequence of problems do we need to solve in order to compute edit_distance(A, B)?
Computing the Edit Distance

• What sequence of problems do we need to solve in order to compute edit_distance(A, B)?
• For each $i$ in $0, \ldots, |A|-1$
  – For each $j$ in $0, \ldots, |B|-1$
    • Compute edit_distance(A[0, ..., i], B[0, ..., j]).
• The total number of problems we need to solve is $|A| \times |B|$, which is manageable.
• What are the base cases?
Computing the Edit Distance

• Base case 1: \( \text{edit\_distance}('', '') = 0. \)
  – The edit distance between two empty strings.
• Base case 2: \( \text{edit\_distance}('', B[0, ..., j]) = j+1. \)
• Base case 3: \( \text{edit\_distance}(A[0, ..., i], '') = i+1. \)
Computing the Edit Distance

• For convenience, we define $A[0, -1] = \text{""}$, $B[0, -1] = \text{""}$. 
• Then, we can rewrite the previous base cases like this:
  • Base case 1: $\text{edit\_distance}(A[0, -1], B[0, -1]) = 0$. 
    – The edit distance between two empty strings.
  • Base case 2: $\text{edit\_distance}(A[0, -1], B[0, ..., j]) = j+1$. 
  • Base case 3: $\text{edit\_distance}(A[0, ..., i], B[0, -1]) = i+1$. 

Computing the Edit Distance

• Recursive case: if $i \geq 0$, $j \geq 0$:

• $\text{edit\_distance}(A[0, ..., i], B[0, ..., j]) = \text{smallest of these three values:}$
  
  – 1: $\text{edit\_distance}(A[0, ..., i-1], B[0, ..., j]) + 1$
  
  – 2: $\text{edit\_distance}(A[0, ..., i], B[0, ..., j-1]) + 1$

  – 3: either $\text{edit\_distance}(A[0, ..., i-1], B[0, ..., j-1])$
    
    
    – or $\text{edit\_distance}(A[0, ..., i-1], B[0, ..., j-1]) + 1$.
      
      • If $A[i] \neq B[j]$. 

•
Extra materials
Evaluating Prefix Expressions

- Prefix expressions: they place each operand BEFORE its two arguments.
- Example: * + 7 * * 4 6 + 8 9 5
Evaluating Prefix Expressions

• Code for evaluating prefix expressions:

```c
char *a; int i;
int eval()
{
    int x = 0;
    while (a[i] == ' ') i++;
    if (a[i] == '+')
        { i++; return eval() + eval(); }
    if (a[i] == '*')
        { i++; return eval() * eval(); }
    while ((a[i] >= '0') && (a[i] <= '9'))
        x = 10*x + (a[i++]-'0');
    return x;
}
```

Example: * + 7 * * 4 6 + 8 9 5:
Evaluating Prefix Expressions

• Code for evaluating prefix expressions:

```c
char *a; int i;
int eval()
{
    int x = 0;
    while (a[i] == ' ') i++;
    if (a[i] == '+')
        { i++; return eval() + eval(); } 
    if (a[i] == '*')
        { i++; return eval() * eval(); } 
    while ((a[i] >= '0') && (a[i] <= '9'))
        x = 10*x + (a[i++]-'0');
    return x;
}
```

Example: * + 7 * * 4 6 + 8 9 5:
- * wait wait
- + wait wait
- 7
Evaluating Prefix Expressions

- Code for evaluating prefix expressions:

```c
char *a; int i;
int eval()
{
    int x = 0;
    while (a[i] == ' ') i++;
    if (a[i] == '+')
        { i++; return eval() + eval(); }
    if (a[i] == '*')
        { i++; return eval() * eval(); }
    while ((a[i] >= '0') && (a[i] <= '9'))
        x = 10*x + (a[i++]-'0');
    return x;
}
```

Example: * + 7 * * 4 6 + 8 9 5:
- * wait wait
- + 7 wait
- * wait wait
- * wait wait
- 4
Evaluating Prefix Expressions

• Code for evaluating prefix expressions:

```c
char *a; int i;
int eval()
{
    int x = 0;
    while (a[i] == ' ') i++;
    if (a[i] == '+')
    {
        i++; return eval() + eval(); }
    if (a[i] == '*')
    {
        i++; return eval() * eval(); }
    while ((a[i] >= '0') && (a[i] <= '9'))
    { x = 10*x + (a[i++]-'0');
      return x;
    }
}
```

Example: * + 7 * * 4 6 + 8 9 5:
• * wait wait
• + 7 wait
• * wait wait
• * 4 wait
• 6
Evaluating Prefix Expressions

- Code for evaluating prefix expressions:

```c
char *a; int i;
int eval()
{
    int x = 0;
    while (a[i] == ' ') i++;
    if (a[i] == '+')
        { i++; return eval() + eval(); }
    if (a[i] == '*')
        { i++; return eval() * eval(); }
    while ((a[i] >= '0') && (a[i] <= '9'))
        x = 10*x + (a[i++]-'0');
    return x;
}
```

Example: * + 7 * * 4 6 + 8 9 5:
- * wait wait
- + 7 wait
- * wait wait
- * 4 6 = 24
Evaluating Prefix Expressions

• Code for evaluating prefix expressions:

```
char *a; int i;
int eval()
{
    int x = 0;
    while (a[i] == ' ') i++;
    if (a[i] == '+')
        { i++; return eval() + eval(); }
    if (a[i] == '*')
        { i++; return eval() * eval(); }
    while ((a[i] >= '0') && (a[i] <= '9'))
        x = 10*x + (a[i++]-'0');
    return x;
}
```

Example: * + 7 * * 4 6 + 8 9 5:
• * wait wait
• + 7 wait
• * 24 wait
• + wait wait
Evaluating Prefix Expressions

• Code for evaluating prefix expressions:

```c
char *a; int i;
int eval()
{
    int x = 0;
    while (a[i] == ' ') i++;
    if (a[i] == '+')
        { i++; return eval() + eval(); }
    if (a[i] == '*')
        { i++; return eval() * eval(); }
    while ((a[i] >= '0') && (a[i] <= '9'))
        x = 10*x + (a[i++]-'0');
    return x;
}
```

Example: * + 7 * * 4 6 + 8 9 5:
• * wait wait
• + 7 wait
• * 24 wait
• + wait wait
• 8
Evaluating Prefix Expressions

• Code for evaluating prefix expressions:

```c
char *a; int i;
int eval()
{
    int x = 0;
    while (a[i] == ' ') i++;
    if (a[i] == '+')
        { i++; return eval() + eval(); }
    if (a[i] == '*')
        { i++; return eval() * eval(); }
    while ((a[i] >= '0') && (a[i] <= '9'))
        x = 10*x + (a[i++]-'0');
    return x;
}
```

Example: * + 7 * * 4 6 + 8 9 5:
•  * wait wait
•  + 7 wait
•  * 24 wait
•  + 8 wait
•  9
Evaluating Prefix Expressions

• Code for evaluating prefix expressions:

```c
char *a; int i;
int eval()
{
    int x = 0;
    while (a[i] == ' ') i++;
    if (a[i] == '+')
        { i++; return eval() + eval(); }
    if (a[i] == '*')
        { i++; return eval() * eval(); }
    while ((a[i] >= '0') && (a[i] <= '9'))
        x = 10*x + (a[i++]-'0');
    return x;
}
```

Example: * + 7 * * 4 6 + 8 9 5:
• * wait wait
• + 7 wait
• * 24 wait
• + 8 9 = 17
Evaluating Prefix Expressions

• Code for evaluating prefix expressions:

```c
char *a; int i;
int eval()
{
    int x = 0;
    while (a[i] == ' ') i++;
    if (a[i] == '+')
    {
        i++; return eval() + eval();
    }
    if (a[i] == '*')
    {
        i++; return eval() * eval();
    }
    while ((a[i] >= '0') && (a[i] <= '9'))
    {
        x = 10*x + (a[i++]-'0');
    }
    return x;
}
```

Example: * + 7 * * 4 6 + 8 9 5:
• * wait wait
• + 7 wait
• * 24 17 = 408
Evaluating Prefix Expressions

- Code for evaluating prefix expressions:

```c
char *a; int i;
int eval()
{
    int x = 0;
    while (a[i] == ' ') i++;
    if (a[i] == '+')
        { i++; return eval() + eval(); }
    if (a[i] == '*')
        { i++; return eval() * eval(); }
    while ((a[i] >= '0') && (a[i] <= '9'))
        x = 10*x + (a[i++]-'0');
    return x;
}
```

Example: * + 7 * * 4 6 + 8 9 5:
- * wait wait
- + 7 408 = 415
Evaluating Prefix Expressions

• Code for evaluating prefix expressions:

```c
char *a; int i;
int eval()
{
    int x = 0;
    while (a[i] == ' ') i++;  
    if (a[i] == '+') 
    { i++; return eval() + eval(); }
    if (a[i] == '*') 
    { i++; return eval() * eval(); }
    while ((a[i] >= '0') && (a[i] <= '9'))
    { x = 10*x + (a[i++]-'0'); }
    return x;
}
```

Example: \( * + 7 * * 4 6 + 8 9 5: \)
- \( * 415 \) wait
- \( 5 \)
Evaluating Prefix Expressions

• Code for evaluating prefix expressions:

```c
char *a; int i;
int eval()
{
    int x = 0;
    while (a[i] == ' ') i++;
    if (a[i] == '+')
        { i++; return eval() + eval(); }
    if (a[i] == '*')
        { i++; return eval() * eval(); }
    while ((a[i] >= '0') && (a[i] <= '9'))
        x = 10*x + (a[i++]-'0');
    return x;
}
```

Example: * + 7 * * 4 6 + 8 9 5:
• * 415 5 = 2075