Top-Down Approach to Algorithms

Reduction: Solve a problem by using a solution to a "simpler" problem.

Top-Down Approach to Algorithms

Reduction: Solve a problem by using a solution to a "simpler" problem.

The selection problem:

- **Input:** An array $A[1..n]$ of NUMBERs and a NAT k.
- **Output:** The kth smallest element of A.

Top-Down Approach to Algorithms

Reduction: Solve a problem by using a solution to a "simpler" problem.

The selection problem:

- **Input:** An array $A[1..n]$ of NUMBERs and a NAT k.
- **Output:** The kth smallest element of A.

One solution:

1. Sort A.
2. Return $A[k]$.

Top-Down Approach to Algorithms

Reduction: Solve a problem by using a solution to a "simpler" problem.

The selection problem:

- **Input:** An array $A[1..n]$ of NUMBERs and a NAT k.
- **Output:** The kth smallest element of A.

One solution:

1. Sort A.
2. Return $A[k]$.

We have reduced selection to sorting.

Sorting

We may sort an array $A[1..n]$ for $n > 1$ by

Sorting

We may sort an array $A[1..n]$ for $n > 1$ by

1. sorting $A[1..n-1]$; then

Sorting

We may sort an array $A[1..n]$ for $n > 1$ by

1. sorting $A[1..n-1]$; then
2. inserting $A[n]$ into $A[1..n-1]$ at the proper location.

Sorting

We may sort an array $A[1..n]$ for $n > 1$ by

1. sorting $A[1..n-1]$; then
2. inserting $A[n]$ into $A[1..n-1]$ at the proper location.

If $n \leq 1$, then $A[1..n]$ is already sorted.

Sorting

We may sort an array $A[1..n]$ for $n > 1$ by

1. sorting $A[1..n-1]$; then
2. inserting $A[n]$ into $A[1..n-1]$ at the proper location.

If $n \leq 1$, then $A[1..n]$ is already sorted.
We have reduced larger instances of sorting to smaller instances.

Recursive Insertion Sort

Precondition: $A[1..n]$ is an array of $\textsc{Number}$s, n is a $\textsc{Nat}$.
Postcondition: $A[1..n]$ is a permutation of its initial values such that for $1 \leq i < j \leq n$, $A[i] \leq A[j]$.

$\textsc{InsertSort}(A[1..n])$
    if $n > 1$
        $\textsc{InsertSort}(A[1..n-1])$
        $\textsc{Insert}(A[1..n])$

Recursive Insertion Sort

Precondition: $A[1..n]$ is an array of $\text{N}\textsc{umber}$s, n is a $\text{N}\textsc{at}$.
Postcondition: $A[1..n]$ is a permutation of its initial values such that for $1 \leq i < j \leq n$, $A[i] \leq A[j]$.

$\textsc{InsertSort}(A[1..n])$
    if $n > 1$
        $\textsc{InsertSort}(A[1..n-1])$
        $\textsc{Insert}(A[1..n])$

Precondition: $A[1..n]$ is an array of $\text{N}\textsc{umber}$s such that n is a $\text{N}\textsc{at}$, and for $1 \leq i < j \leq n-1$, $A[i] \leq A[j]$.
Postcondition: $A[1..n]$ is a permutation of its initial values such that for $1 \leq i < j \leq n$, $A[i] \leq A[j]$.

$\textsc{Insert}(A[1..n])$

Maximum Subsequence Sum

Input: An array $A[0..n-1]$ of (possibly negative) $\text{N\scriptsize UMBER}$s.
Output: The maximum sum of any contiguous subsequence of A; i.e.,

$$\max\left\{\sum_{k=i}^{j-1} A[k] \mid 0 \leq i \leq j \leq n\right\}.$$

A Naive Algorithm

Precondition: $A[0..n-1]$ is an array of NUMBERs, n is a NAT.

Postcondition: Returns the maximum subsequence sum of A.

$\text{MAXSUMITER}(A[0..n-1])$
    $m \leftarrow 0$
    for $i \leftarrow 0$ **to** n
        for $j \leftarrow i$ **to** n
            $sum \leftarrow 0$
            for $k \leftarrow i$ **to** $j-1$
                $sum \leftarrow sum + A[k]$
            $m \leftarrow \text{MAX}(m, sum)$
    return m

Improving the Naive Algorithm

Precondition: $A[0..n-1]$ is an array of $\mathrm{N{\scriptsize UMBER}}$s, n is a $\mathrm{N{\scriptsize AT}}$.
Postcondition: Returns the maximum subsequence sum of A.

$\mathrm{M{\scriptsize AX}S{\scriptsize UM}O{\scriptsize PT}}(A[0..n-1])$
    $m \leftarrow 0$
    for $i \leftarrow 0$ **to** $n-1$
        $sum \leftarrow 0$
        for $k \leftarrow i$ **to** $n-1$
            $sum \leftarrow sum + A[k]$
            $m \leftarrow \mathrm{M{\scriptsize AX}}(m, sum)$
    return m

Reducing to a Smaller Problem

We can reduce an instance of size $n > 0$ to an instance of size $n - 1$:

Reducing to a Smaller Problem

We can reduce an instance of size $n > 0$ to an instance of size $n - 1$:

1. Find the maximum subsequence sum of the first $n - 1$ elements.

Reducing to a Smaller Problem

We can reduce an instance of size $n > 0$ to an instance of size $n - 1$:

1. Find the maximum subsequence sum of the first $n - 1$ elements.
2. Find the maximum *suffix* sum; i.e.,

$$\max\left\{\sum_{k=i}^{n-1} A[k] \mid 0 \leq i \leq n\right\}.$$

Reducing to a Smaller Problem

We can reduce an instance of size $n > 0$ to an instance of size $n - 1$:

1. Find the maximum subsequence sum of the first $n - 1$ elements.
2. Find the maximum *suffix* sum; i.e.,

$$\max\left\{\sum_{k=i}^{n-1} A[k] \mid 0 \leq i \leq n\right\}.$$

3. Return the maximum of these two values.

Reducing to a Smaller Problem

We can reduce an instance of size $n > 0$ to an instance of size $n - 1$:

1. Find the maximum subsequence sum of the first $n - 1$ elements.
2. Find the maximum *suffix* sum; i.e.,

$$\max \left\{ \sum_{k=i}^{n-1} A[k] \mid 0 \leq i \leq n \right\}.$$

3. Return the maximum of these two values.

If $n = 0$, the maximum subsequence sum is 0.

Finding the Maximum Suffix Sum

We can find the maximum suffix sum in a similar way; i.e., if $n > 0$:

Finding the Maximum Suffix Sum

We can find the maximum suffix sum in a similar way; i.e., if $n > 0$:

1. Find the maximum suffix sum of the first $n - 1$ elements.

Finding the Maximum Suffix Sum

We can find the maximum suffix sum in a similar way; i.e., if $n > 0$:

1. Find the maximum suffix sum of the first $n - 1$ elements.
2. Add the last element.

Finding the Maximum Suffix Sum

We can find the maximum suffix sum in a similar way; i.e., if $n > 0$:

1. Find the maximum suffix sum of the first $n - 1$ elements.
2. Add the last element.
3. Return the maximum of this sum and 0.

Finding the Maximum Suffix Sum

We can find the maximum suffix sum in a similar way; i.e., if $n > 0$:

1. Find the maximum suffix sum of the first $n - 1$ elements.
2. Add the last element.
3. Return the maximum of this sum and 0.

If $n = 0$, the maximum subsequence sum is 0.

Maximal Subsequence Sum, Top-Down

Precondition: $A[0..n-1]$ is an array of Numbers, n is a Nat.

Postcondition: Returns the maximum subsequence sum of A.

$\text{MaxSumTD}(A[0..n-1])$
    if $n = 0$
        return 0
    else
        return $\text{Max}(\text{MaxSumTD}(A[0..n-2]),$
$\text{MaxSuffixTD}(A[0..n-1]))$

Maximal Suffix Sum, Computed Top-Down

Precondition: $A[0..n-1]$ is an array of NUMBERs, n is a NAT.
Postcondition: Returns the maximum suffix sum of A.

$\text{MAXSUFFIXTD}(A[0..n-1])$
  if $n = 0$
    return 0
  else
    return
      $\text{MAX}(0, A[n-1] + \text{MAXSUFFIXTD}(A[0..n-2]))$

Divide and Conquer

We can reduce an instance of size $n > 1$ to instances of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$.

Divide and Conquer

We can reduce an instance of size $n > 1$ to instances of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$.
The maximum of the solutions to the smaller instances does not include any segments that start in the first instance and end in the last instance.

Divide and Conquer

We can reduce an instance of size $n > 1$ to instances of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$.
The maximum of the solutions to the smaller instances does not include any segments that start in the first instance and end in the last instance.
We therefore need to find the maximum suffix sum of the first instance and the maximum prefix sum of the second.

An Algorithm based on Divide and Conquer

Precondition: $A[lo..hi]$ is an array of $\textsc{Number}$s, $lo \leq hi$, and both lo and hi are $\textsc{Nat}$s.
Postcondition: Returns the maximum subsequence sum of $A[lo..hi]$.

$\textsc{MaxSumDC}(A[lo..hi])$
  if $lo = hi$
    return $\textsc{Max}(0, A[lo])$
  else
    $mid \leftarrow \lfloor (lo + hi)/2 \rfloor$; $mid1 \leftarrow mid + 1$
    $sum1 \leftarrow \textsc{MaxSumDC}(A[lo..mid])$
    $sum2 \leftarrow \textsc{MaxSumDC}(A[mid1..hi])$
    $sum3 \leftarrow \textsc{MaxSuffix}(A[lo..mid]) +$
          $\textsc{MaxPrefix}(A[mid1..hi])$
    return $\textsc{Max}(sum1, sum2, sum3)$

Bottom-up Computation

We can often save stack space by implementing a top-down design in a bottom-up fashion:

Bottom-up Computation

We can often save stack space by implementing a top-down design in a bottom-up fashion:

1. Compute solutions to the smallest instances.

Bottom-up Computation

We can often save stack space by implementing a top-down design in a bottom-up fashion:

1. Compute solutions to the smallest instances.
2. Using the top-down solution as a guide, combine the solutions of smaller instances to obtain solutions to larger instances.

Maximum Suffix Sum, Computed Bottom-Up

Precondition: $A[lo..hi]$ is an array of NUMBERs, $lo \le hi$, and both lo and hi are NATs.
Postcondition: Returns the maximum suffix sum of $A[lo..hi]$.

$\text{MAXSUFFIXBU}(A[lo..hi])$
    $m \leftarrow 0$
    // **Invariant:** m is the maximum suffix sum of
    // $A[lo..i-1]$
    for $i \leftarrow lo$ **to** hi
        $m \leftarrow \text{MAX}(0, m + A[i])$
    return m

Maximum Subsequence Sum, Bottom-Up

Precondition: $A[0..n-1]$ is an array of NUMBERs, n is a NAT.

Postcondition: Returns the maximum subsequence sum of A.

$\text{MAXSUMBU}(A[0..n-1])$
    $m \leftarrow 0$; $msuf \leftarrow 0$
    // **Invariant:** m is the maximum subsequence sum
    // of $A[0..i-1]$, $msuf$ is the maximum suffix sum
    // for $A[0..i-1]$
    for $i \leftarrow 0$ **to** $n-1$
        $msuf \leftarrow \text{MAX}(0, msuf + A[i])$
        $m \leftarrow \text{MAX}(m, msuf)$
    return m