Design and Analysis of Algorithms
Dynamic Programming (I)

Outline

Algorithmic Paradigms

We have seen two elegant design paradigms.

- Divide-and-conquer. Break up a problem into independent subproblems, solve each subproblem, combine solutions to subproblems to form solution to original problem.
- Greedy. Build up a solution piece-by-piece, always choosing the next piece that offers the most obvious and immedeiate benefit.
  - The problems where choosing locally optimal aslo leads to globally optimal soulution are best fit for Greedy.

Algorithmic Paradigms

We have seen two elegant design paradigms.

- Divide-and-conquer. Break up a problem into independent subproblems, solve each subproblem, combine solutions to subproblems to form solution to original problem.
- Greedy. Build up a solution piece-by-piece, always choosing the next piece that offers the most obvious and immedeiate benefit.
    - The problems where choosing locally optimal aslo leads to globally optimal soulution are best fit for Greedy.

The two paradigms yield lots of efficient algorithms for a variety of important tasks.

Algorithmic Paradigms

We have seen two elegant design paradigms.

- Divide-and-conquer. Break up a problem into independent subproblems, solve each subproblem, combine solutions to subproblems to form solution to original problem.
- Greedy. Build up a solution piece-by-piece, always choosing the next piece that offers the most obvious and immedeiate benefit.
  - The problems where choosing locally optimal aslo leads to globally optimal soulution are best fit for Greedy.

The two paradigms yield lots of efficient algorithms for a variety of important tasks.

We now turn to another sledgehammer of the algorithms craft: dynamic programming, techniques of very broad applicability.

- Predictably, the generality often comes with a cost of efficiency.

Dynamic Programming History

Dynamic programming. Break up a problem into a series of overlapping subproblems of the same type, and build up solutions to larger and larger subproblems.

- fancy name for caching away intermediate results in a table for later reuse

Dynamic Programming History

Dynamic programming. Break up a problem into a series of overlapping subproblems of the same type, and build up solutions to larger and larger subproblems.

- fancy name for caching away intermediate results in a table for later reuse

Richard Bellman. Pioneered the systematic study of DP in 1950s.

- dynamic programming = planning over time ⇒ optimal plan multistage processes
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.

THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

1. Introduction. Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time t is determined by a set of quantities which we call state parameters, or state variables. At certain times, which may be prescribed in advance, or which may be determined by the process itself, we are called upon to make decisions which will affect the state of the system. These decisions are equivalent to transformations of the state variables, the choice of a decision being identical with the choice of a transformation. The outcome of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process that of maximizing some function of the parameters describing the final state.

Examples of processes fitting this loose description are furnished by virtually every phase of modern life, from the planning of industrial production lines to the scheduling of patients at a medical clinic; from the determination of long-term investment programs for universities to the determination of a replacement policy for machinery in factories; from the programming of training policies for skilled and unskilled labor to the choice of optimal purchasing and inventory policies for department stores and military establishments.

Dynamic Programming Applications

Areas

- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, compilers, systems, ...

Some famous dynamic programming algorithms

- Unix `diff` for comparing two files.
- Viterbi for hidden Markov models.
- De Boor for evaluating spline curves.
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context-free grammars.

Outline

Shortest Path in DAG

Finding shortest path (from a speical source node) is especially easy in directed acyclic graphs (DAGs). We recapitulate this case, because it lies at the heart of dynamic programming.

- Nodes of DAG can be linearized, i.e., arranged on a line so that all edges go from left to right
- Looking ahead, in this way we create an order

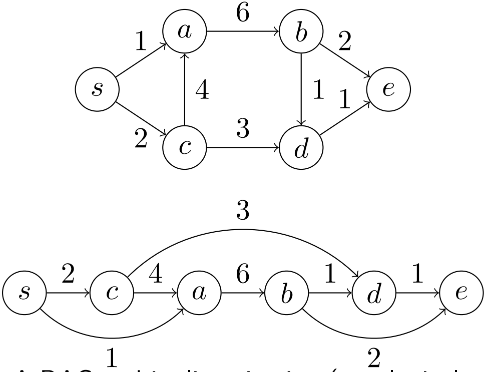


Figure: A DAG and its linearization (topological ordering)

Why this helps with shortest paths

Example. $s \to d$: the only way get to d is through its predecessors b or c, so we need only compare these two routes:

$$\mathsf{dist}(s,d) = \min\{\mathsf{dist}(s,b)+1, \mathsf{dist}(s,c)+3\}$$

Why this helps with shortest paths

Example. $s \to d$: the only way get to d is through its predecessors b or c, so we need only compare these two routes:

$$\text{dist}(s, d) = \min\{\text{dist}(s, b) + 1, \text{dist}(s, c) + 3\}$$

A similar relation can be written for every node.

- Computing these dist values in the left-to-right order $\Rightarrow$ before getting to a node v, we already have all the information to compute dist(s, v) $\Rightarrow$ computing all the distance in a single pass

Algorithm for Shortest Paths in DAG

Algorithm 1: ShortestPath(V, E)

initialize dist$(s, v) = \infty$ for $s \neq v$ and dist$(s, s) = 0$;
for $v \in V \backslash s$ *in linearized order* **do**
    dist$(s, v) = \min_{(u,v) \in E}\{$dist$(s, u) + e(u, v)\}$
end

Algorithm for Shortest Paths in DAG

Algorithm 2: ShortestPath(V, E)

```
initialize dist(s, v) = ∞ for s ≠ v and dist(s, s) = 0;
for v ∈ V\s in linearized order do
    dist(s, v) = min_{(u,v)∈E} {dist(s, u) + e(u, v)}
end
```

Two methods to estimate computation complexity:

- Analyze the algorithm: there are at most $|E|$ times comparisions $\Rightarrow O(|E|)$
- Analyze the storage: size of table dist is $|V|$, computing each item requires at most $|V|$ times comparisions $\Rightarrow O(|V|^2)$
  - the second estimation could be too coarse when the graph is sparse, since in that case $|E| \ll |V|^2$

Recap

The above algorithm solves a collection of subproblems

$$\{\mathsf{dist}(s, u)\}_{u \in V}$$

- start from the smallest of them, $\mathsf{dist}(s, s)$
- then proceed to solve progressively "larger" subproblems: distances to vertices that are further along the linearization
- large subproblems can be solved by previously solved smaller subproblems

Recap

The above algorithm solves a collection of subproblems

$$\{\text{dist}(s, u)\}_{u \in V}$$

- start from the smallest of them, $\text{dist}(s, s)$
- then proceed to solve progressively "larger" subproblems: distances to vertices that are further along the linearization
- large subproblems can be solved by previously solved smaller subproblems

This is a very generic technique.

- $\text{dist}(\cdot, \cdot)$ in our particular case computing the *minimum* of sums, we could just as well make it to be *maximum*.
- Or we could use a product instead of a sum.

Key Property of Dynamic Programming

Iterative optimal substructure

$\exists$ an ordering on the subproblems and an iterative relation:

- subproblems appear in the ordering
- iterative relation shows how to solve a subproblem P using the answers to "smaller" subproblems P', a.k.a. optimal solution for P can be derived from optimal solutions for $P' \subset P$

$\rightsquigarrow$ admits iteration in a single pass

DP Paradigm

Dynamic programming is a very powerful algorithmic paradigm: a problem is solved by identifying a collection of subproblems and tackling them one by one

- smallest first
- using answers to small problems to solve larger ones
- until reaching the original problem

In dynamic programming, the DAG is *implicit* and should always be kept in mind

- node $\leftrightarrow$ subproblem/state (associated with an optimal function value)
- edge $a \rightarrow b$ represents dependencies between a and b, in other words, if to solve subproblem b we need to the answer to subproblem a, then there is a (conceptual) edge from a to b $\Rightarrow$ a is thought of as a smaller subproblem than b

Outline

All Pairs Shortest Paths

Life is complicated. Real world needs algorithm for general directed weighted graph: G could have negative edge weights (but with no negative cycles).

- Dijkstra's algorithm fails to handle negative edge weights.
- Bellman-Ford algorithm works correctly with SSSP in general directed graph with higher complexity $O(|V||E|)$.

What if we want to find the shortest path not just from a single-source s but all sources?

Naive idea: invoking Bellman-Ford algorithm $|V|$ times, once for each starting node $\rightsquigarrow$ running time $O(|V|^2|E|)$

- typically, $|E| > |V|$

Better algorithm?

Floyd-Warshall Algorithm

Floyd-Warshall algorithm: a better dynamic-programming algorithm with better complexity $O(|V|^3)$

Basic idea. the shortest path $u \to w_1 \to \cdots \to w_l \to v$ between (u, v) uses some number of intermediate nodes — possibly none.

- Suppose we disallow intermediate nodes altogether $\rightsquigarrow$ solve all-pairs shortest paths at once: dist$(u, v) = e(u, v)$.

What if we gradually expand the set S of permissible intermediate nodes?

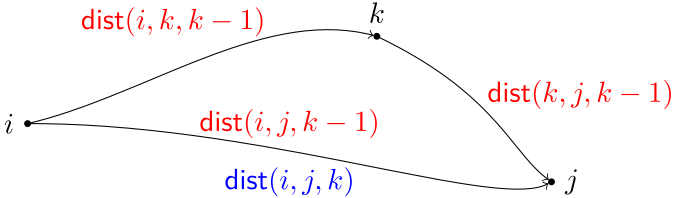
We can do this one node at a time, updating the shortest path lengths at each stage.

- Eventually S grows to V $\Rightarrow$ at this point all vertices are allowed to be on all paths $\rightsquigarrow$ find the true shortest paths between vertices of the graph.

Dynamic Programming on Intermediates

Number the vertices in V as $\{1, 2, \ldots, n\}$, and let dist(i, j, k) denote the length of the shortest path from i to j in which only nodes $\{1, 2, \ldots, k\}$ can be used as intermediates.

- Initially, dist$(i, j, 0)$ is the length of the direct edge between i and j if it exists and is ∞ otherwise.



Gradually increase the number of admissble intermediate node. The initial value of dist(i, j, k) is dist$(i, j, k-1)$.

Using k gives us shorter path from i to j if and only if

$$\text{dist}(i, k, k-1) + \text{dist}(k, j, k-1) < \text{dist}(i, j, k-1)$$

In this case, dist(i, j, k) should be updated accordingly.

Floyd-Warshall Algorithm

Algorithm 3: FloydWarshall($G = (V, E)$)

```
for i = 1 to n do
    for j = 1 to n do
        dist(i, j, 0) = ∞
    end
end
for (i, j) ∈ E do  dist(i, j, 0) = e(i, j) ;
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            dist(i, j, k) = min{dist(i, k, k − 1) + dist(k, j, k −
             1), dist(i, j, k − 1)}
        end
    end
end
```

Outline

Longest Increasing Subsequences

Input: a sequence of numbers $a_1, \ldots, a_n$.

- A *subsequence* is any subset of these numbers taken in order, of the form $a_{i_1}, \ldots, a_{i_k}$ where $1 \leq i_1 \leq \cdots \leq i_k \leq n$.
- An *increasing* subsequence is one in which the numbers are getting strictly larger.

Goal: find the increasing subsequence of greatest length.

Example

The arrow denotes transitions between consecutive elements of the optimal solution in the original sequence.

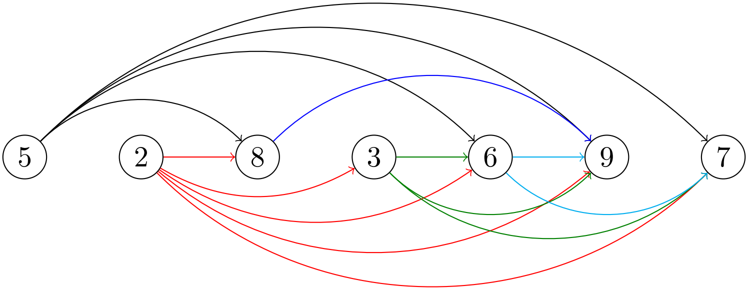
The DAG of Increasing Subsequence

Goal: find the optimal soultion from the solution space (all increasing subsequences) $\Rightarrow$ create a graph of all permissible transitions for increasing subsequence

- Establish a node i for each element a_i, add directed edges (i, j) whenever it is possible for a_i and a_j to be consecutive elements in an increasing subsequence, i.e., $i < j \wedge a_i < a_j$

$G = (V, E)$ is a DAG, since $(i, j) \in E$ is only possible when $i < j$

- there is a one-to-one correspondence between increasing subsequences and paths in this DAG



Dynamic Programming

Our goal translates LIS to finding the longest path (each edge with weight 1) in the DAG.

Our goal translates LIS to finding the longest path (each edge with weight 1) in the DAG.

Define $L(j)$: number of nodes on the longest path (the longest increasing subsequence) ending at j

- interpret $L(j)$ as the length of the longest path $+1$ with j as destination node from all possible source node

$$\ell = \max_{j \in [n]} L(j)$$

Our goal translates LIS to finding the longest path (each edge with weight 1) in the DAG.

Define $L(j)$: number of nodes on the longest path (the longest increasing subsequence) ending at j

- interpret $L(j)$ as the length of the longest path $+1$ with j as destination node from all possible source node

$$\ell = \max_{j \in [n]} L(j)$$

To solve LIS, we defined a collection of subproblems $\{L(j)\}_{j \in [n]}$ with the optimal sub-structure property that allows them to be solved in a single pass.

Algorithm and Complexity Analysis

Algorithm 4: LIS(A)

for $j = 1$ *to* n **do** $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$;
return $\max_j\{L(j)\}$

- Note that $(i, j) \in E$ is possible only when $i < j$.

Algorithm 5: LIS(A)

1: **for** $j = 1$ *to* n **do** $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$;
2: **return** $\max_j\{L(j)\}$

- Note that $(i, j) \in E$ is possible only when $i < j$.

The algorithm requires the predecessors of j to be known

- Construct the adjacency list of the reverse graph G^R (typically in linear time)

Algorithm and Complexity Analysis

Algorithm 6: LIS(A)

for $j = 1$ *to* n **do** $L(j) = 1 + \max\{L(i) : (i,j) \in E\}$;
return $\max_j\{L(j)\}$

- Note that $(i,j) \in E$ is possible only when $i < j$.

The algorithm requires the predecessors of j to be known

- Construct the adjacency list of the reverse graph G^R (typically in linear time)

The computation of $L(j)$ then takes time proportional to the indegree of $j \leadsto$ overall running time linear in $|E|$

- The maximum being when the input array is sorted in increasing order $\leadsto W(n) = |E| = O(n^2)$

Algorithm and Complexity Analysis

Algorithm 7: LIS(A)

for $j = 1$ *to* n **do** $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$;
return $\max_j\{L(j)\}$

- Note that $(i, j) \in E$ is possible only when $i < j$.

The algorithm requires the predecessors of j to be known

- Construct the adjacency list of the reverse graph G^R (typically in linear time)

The computation of $L(j)$ then takes time proportional to the indegree of $j \rightsquigarrow$ overall running time linear in $|E|$

- The maximum being when the input array is sorted in increasing order $\rightsquigarrow W(n) = |E| = O(n^2)$

LIS computes longest path in reverse DAG for each “source” node then selects the max, while ShortestPath computes the shortest path in DAG from one given source node to all other nodes.

Trace Solution

There is one last issue to be cleared up.

The L-values only tell us the length of the optimal subsequence, how to recover the subsequence itself?

- This is easily managed with bookkeeping device
  - when computing $L(j)$, note down prev(j), the next-to-last node on the longest path to j (think how?)
- The optimal subsequence can then be reconstructed by the following these backpointers.

Recursion? No, thanks.

Returning to our discussion of longest increasing subsequences

- The formula for $L(j)$ also suggests an alternative, recursive algorithm. Wouldn't that be even simpler?

Recursion? No, thanks.

Returning to our discussion of longest increasing subsequences

- The formula for $L(j)$ also suggests an alternative, recursive algorithm. Wouldn't that be even simpler?

Actually, recursion is a very bad idea: the resulting procedure would require exponential time.

- Suppose the given numbers are sorted. Clearly, this is the worse case. The formula for subproblem $L(j)$ becomes:

$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}$$

Recursion? No, thanks.

Returning to our discussion of longest increasing subsequences

- The formula for $L(j)$ also suggests an alternative, recursive algorithm. Wouldn't that be even simpler?

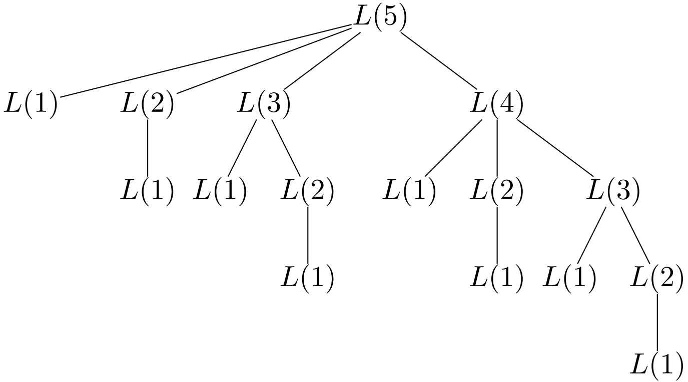
Actually, recursion is a very bad idea: the resulting procedure would require exponential time.

- Suppose the given numbers are sorted. Clearly, this is the worse case. The formula for subproblem $L(j)$ becomes:

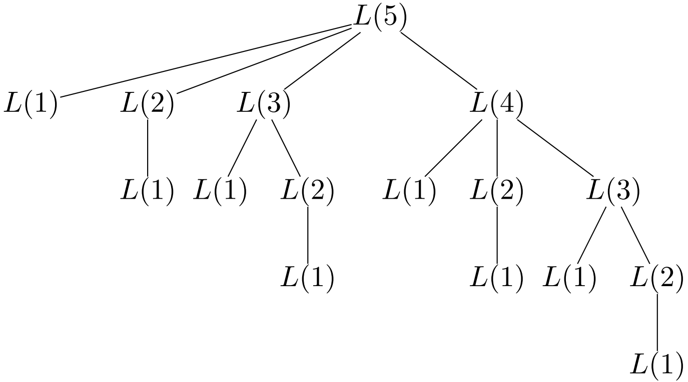
$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}$$

The following figure unravels the recursion for $L(5)$. Notice the same subproblems get solved over and over again.

Why Recursion is Not Good?

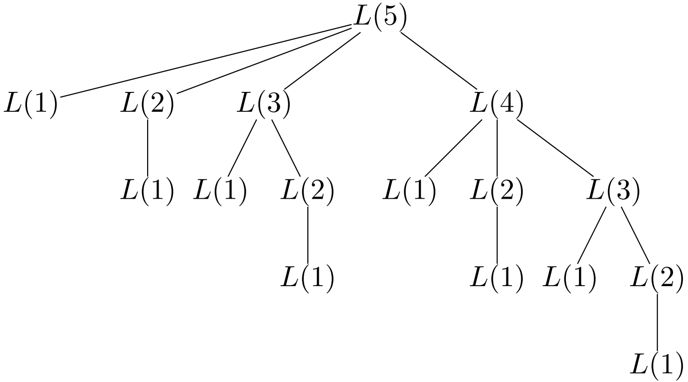


Why Recursion is Not Good?



Nodes correspond to the computation cost. Let $C(n)$ be the number of nodes on the tree for $L(n)$. We have $T(n) = C(n)$.

Why Recursion is Not Good?



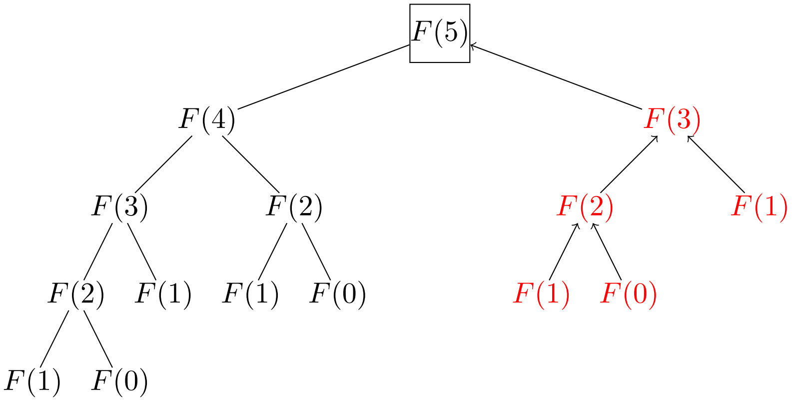
Nodes correspond to the computation cost. Let $C(n)$ be the number of nodes on the tree for $L(n)$. We have $T(n) = C(n)$.

Clearly, we have the following iterative relation:

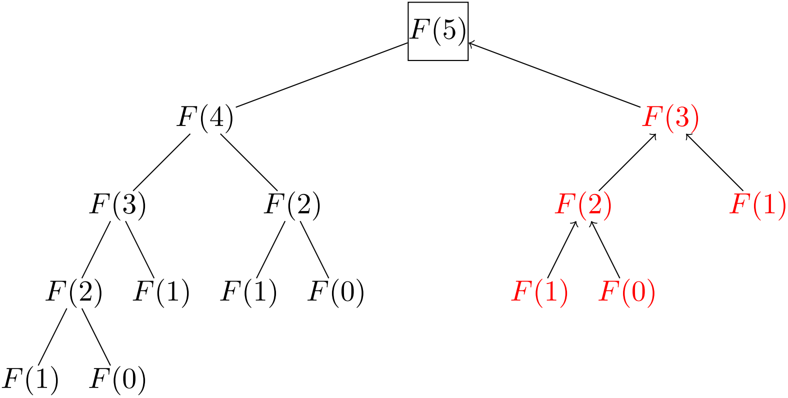
$$C(n) = C(n-1) + \cdots + C(2) + C(1)$$

- $C(n)$ is exponentially in $n \leadsto$ a recursive solution is disastrous

Similar Case for Fibonacci Number



Similar Case for Fibonacci Number

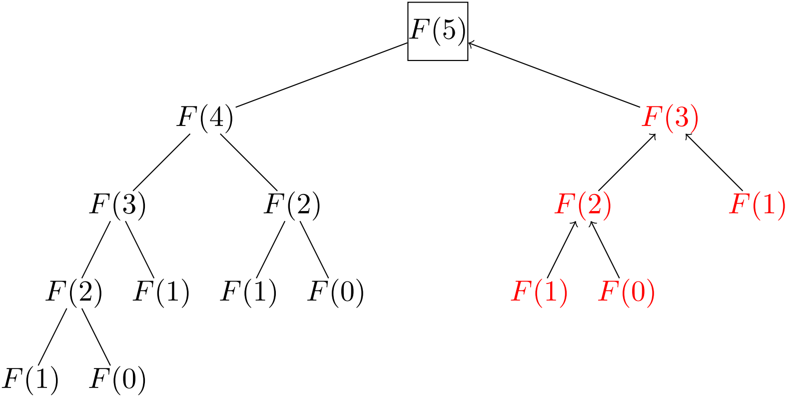


Recursive approach: complexity is $F(n)$.

- Let $C(n)$ be the the nodes on the tree for $F(n)$, we have:

$$C(n) = C(n-1) + C(n-2) = F(n)$$

Similar Case for Fibonacci Number



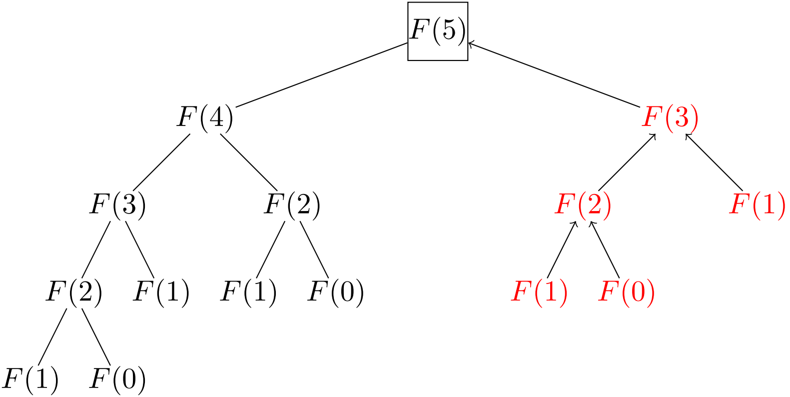
Recursive approach: complexity is $F(n)$.

- Let $C(n)$ be the the nodes on the tree for $F(n)$, we have:

$$C(n) = C(n-1) + C(n-2) = F(n)$$

Iterative approach: complexity is $O(n)$.

Similar Case for Fibonacci Number



Recursive approach: complexity is $F(n)$.

- Let $C(n)$ be the the nodes on the tree for $F(n)$, we have:

$$C(n) = C(n-1) + C(n-2) = F(n)$$

Iterative approach: complexity is $O(n)$.

Divide-and-conquer approach: complexity is $O(\log n)$.

Dynamic Programming vs. Divide-and-Conquer

In the realm of divide-and-conquer, a problem is expressed in terms of subproblems that are *substantially smaller*, say half the size.

- For instance, MergeSort sorts an array of size n by recursively sorting two subarrays of size $n/2$.
- The sharp drop in problem size, the full recursion tree has only logarithmic depth and a polynomial number of nodes.

Dynamic Programming vs. Divide-and-Conquer

In the realm of divide-and-conquer, a problem is expressed in terms of subproblems that are *substantially smaller*, say half the size.

- For instance, MergeSort sorts an array of size n by recursively sorting two subarrays of size $n/2$.
- The sharp drop in problem size, the full recursion tree has only logarithmic depth and a polynomial number of nodes.

In dynamic programming, the problem is reduced to subproblems that are only slightly smaller. Thus the full recursion tree generally has polynomial depth and exponentially number of nodes.

- However, most of these nodes are repeated $\rightsquigarrow$ not too many distinct subproblems among them.
- Efficiency is therefore obtained by explicitly enumerating the distinct subproblems and solving them in the right order.

Outline

Maximum Interval Sum (最大子段和)

Problem. Given an integer array (possibly negative) $A[n]$

$$(a_1, a_2, \ldots, a_n)$$

Goal. Find the maximum interval sum:

$$\mathsf{MIS} = \max\{0, \max_{1 \le i \le j \le n} \sum_{k=i}^{j} a_k\}$$

Example. $(-2, 11, -4, 13, -5, -2)$

Solution: $\mathsf{MIS} = a_2 + a_3 + a_4 = 20$

Possible Algorithms

Brute Force: enumerate all possible (i, j) pairs $(i \leq j)$, compute the sum $a_i + \cdots + a_j$ and find the largest.

Divide-and-Conquer: Split the array into left halve and right halve, compute max interval in left halve, right halve and cross one, then find the largest

Dynamic Programming

Brute Force Algorithm

Algorithm 8: Enumerate($A[n]$)

```
Output: MIS, i*, j*
MIS ← 0;
for i ← 1 to n do
    for j ← i to n do                  //enumerate all possible (i, j)
        sum ← 0;
        for k ← i to j do              //compute sum of A[i, j]
            sum ← sum + A[k];
        end
        if sum > MIS then              //update max interval sum
            MIS ← sum, i* ← i, j* ← j;
        end
    end
end
```

Brute Force Algorithm

Algorithm 9: Enumerate($A[n]$)

Output: MIS, i^*, j^*

MIS $\leftarrow 0$;
for $i \leftarrow 1$ *to* n **do**
    for $j \leftarrow i$ *to* n **do** //enumerate all possible (i, j)
        $sum \leftarrow 0$;
        for $k \leftarrow i$ *to* j **do** //compute sum of $A[i, j]$
            $sum \leftarrow sum + A[k]$;
        end
        if $sum >$ MIS **then** //update max interval sum
            MIS $\leftarrow sum$, $i^* \leftarrow i$, $j^* \leftarrow j$;
        end
    end
end

Complexity: $n^2 \times O(n) = O(n^3)$

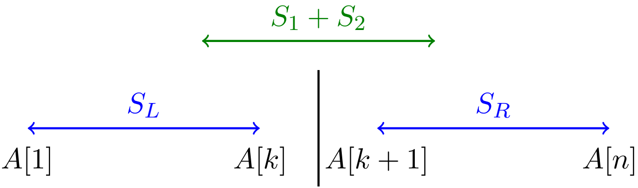
Divide-and-Conquer

Break $A[n]$ into left halve $A[1,k]$ and right halve $A[k+1,n]$, with median k

- Recursively compute S_L for A_L
- Recursively compute S_R for A_R

Compute the max sum S_1 with k as the right boundary, compute the max sum S_2 with $k+1$ as the left boundary,

Output $\max\{S_L, S_R, S_1 + S_2\}$



Pseudocode of Divide-and-Conquer Algorithm

Algorithm 10: MaxIntervalSum$(A[i,j])$

Output: max interval MIS and left/right boundary

if $i = j$ **then return** $\max\{A[i], 0\}$ and boundaries; $//|A| = 1$

$k \leftarrow \lfloor (i+j)/2 \rfloor$;

$S_L \leftarrow$ MaxIntervalSum(A, i, k) ;

$S_R \leftarrow$ MaxIntervalSum$(A, k+1, j)$;

$S_1 \leftarrow$ MaxOneside$(A, i, k, \leftarrow)$;

$S_2 \leftarrow$ MaxOneside$(A, k+1, j, \rightarrow)$;

return $\max\{S_L, S_R, S_1 + S_2\}$ and boundaries;

- If $A[i] \leq 0$, set the left and right boundary as 0
- The complexity of MaxOneside is $O(n)$.

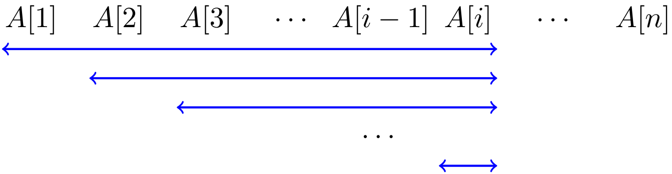
$$\left.\begin{array}{c} T(n) = 2T(n/2) + O(n) \\ T(1) = O(1) \end{array}\right\} \Rightarrow T(n) = O(n \log n)$$

Dynamic Programming

Subproblem: left boundary is 1, right boundary is i

Optimized function: $\text{OPT}(i)$ — maximum interval sum in $A[1,\ldots,i]$ that must include $A[i]$, with i as the right boundary

$$\text{OPT}(i) = \max_{1\le k\le i}\left\{\sum_{j=k}^{i} A[j]\right\}$$



$\text{OPT}(i)$: MIS with i as right boundary

Directly compute $\text{OPT}(i)$ according to the definition will be rather inefficient.

Iterative Relation of Optimized Function

Iterative relation of $\mathsf{OPT}(i)$: depending on the contribution of $\mathsf{OPT}(i-1)$

- $\mathsf{OPT}(i-1) < 0$: the interval only consists of $A[i]$
- $\mathsf{OPT}(i-1) \geq 0$: the interval connects to previous interval

Iterative Relation of Optimized Function

Iterative relation of $\mathsf{OPT}(i)$: depending on the contribution of $\mathsf{OPT}(i-1)$

- $\mathsf{OPT}(i-1) < 0$: the interval only consists of $A[i]$
- $\mathsf{OPT}(i-1) \geq 0$: the interval connects to previous interval

$$\mathsf{OPT}(i) = \max\{\mathsf{OPT}(i-1) + A[i], A[i]\}, i = 2, \ldots, n$$
$$\mathsf{OPT}(1) = A[1]$$

$$\mathsf{MIS} = \max_{1 \leq i \leq n} \{\mathsf{OPT}(i)\}$$

Pseudocode

Algorithm 11: DPMaxIntervalSum($A[n]$)

```
MIS ← 0, i* ← 0, j* ← 0;
OPT(1) = A[1], L(1) = 1 //L(i) records the real left boundary of OPT(i);
for i = 2 to n do                    //i: right boundary of subproblem
    if OPT(i − 1) > 0 then
        OPT(i) ← OPT(i − 1) + A[i];
        L(i) ← L(i − 1);
    end
    else OPT(i) ← A[i], L(i) = i;
    if OPT(i) > MIS then
        MIS ← OPT(i), i* ← L(i), j* ← i
    end
end
return MIS, i*, j*;
```

Time and space complexity: $O(n)$ (think why?)

Outline

Compress Grayscale Image

Grayscale image can be viewed as a sequence of pixels (each pixel ranges from $0 \sim 255$, 8-bit/1-byte)

$$\{a_1, a_2, \ldots, a_n\},\ a_i \text{ is the gray value of the } i\text{-th pixel}$$

- a good test image because of its detail, flat regions, shading, and texture.
- Lena Forsén was also guest of honor at the banquet of IEEE ICIP 2015, delivered a speech and chaired the best paper award ceremony.

Compress Grayscale Image

Grayscale image can be viewed as a sequence of pixels (each pixel ranges from $0 \sim 255$, 8-bit/1-byte)

$$\{a_1, a_2, \ldots, a_n\},\ a_i \text{ is the gray value of the } i\text{-th pixel}$$

- a good test image because of its detail, flat regions, shading, and texture.
- Lena Forsén was also guest of honor at the banquet of IEEE ICIP 2015, delivered a speech and chaired the best paper award ceremony.

Fixed-length image storage. Sequentialize pixels and store: each pixel takes 8-bit, an n pixels image takes $8n$-bit/n-byte

Compress Grayscale Image

Grayscale image can be viewed as a sequence of pixels (each pixel ranges from $0 \sim 255$, 8-bit/1-byte)

$$\{a_1, a_2, \ldots, a_n\},\ a_i \text{ is the gray value of the } i\text{-th pixel}$$

- a good test image because of its detail, flat regions, shading, and texture.
- Lena Forsén was also guest of honor at the banquet of IEEE ICIP 2015, delivered a speech and chaired the best paper award ceremony.

Fixed-length image storage. Sequentialize pixels and store: each pixel takes 8-bit, an n pixels image takes $8n$-bit/n-byte

Observe that image usually has some local pattern. Any better storage method?

Variable-Length Compression

Format of variable-length compression. Encoding grayscale values with variable-length to save storage: divide $\{a_1, a_2, \dots, a_n\}$ into m segments: $S_1, S_2, \dots, S_m$

S_1	S_2	$\cdots$	S_m

S_k contains ℓ_k number of pixels, pixels in S_k take at most b_k-bit

$$b_k = \max_{a \in S_k} \{\lceil \log(a+1) \rceil\}$$

- fix the maximal length of S_k be $256 \Rightarrow \ell_k$ can be represented by 8-bit
- b_k of S_k is among $[1, 8] \Rightarrow b_i$ can be represented by 3-bit
- header of S_k: $\ell_k + b_k = 11$ bit $\rightsquigarrow$ necessary for decoding

$$\text{total storage} = \sum_{k=1}^{m} (b_k \cdot \ell_k + 11)$$

Compress Grayscale Image

Constraint:

- the length of k-th segment: $\ell_k \leq 256$
- the k-th segment takes: $b_k \times \ell_k + 11$
- $b_k = \lceil \log(\max_{a \in S_k} + 1) \rceil \leq 8$

Goal: given $\{a_1, a_2, \ldots, a_n\}$, find the optimal partition:

$$\min_P \left\{ \sum_{k=1}^{m} (b_k \times \ell_k + 11) \right\}$$

$P = \{S_1, S_2, \ldots, S_m\}$ is a partition

Example

Sequence of grayscale values

$$\{10, 12, 15, 255, 1, 2, 1, 1, 2, 2, 1, 1\}$$

1. $S_1 = \{10, 12, 15\}$, $S_2 = \{255\}$, $S_3 = \{1, 2, 1, 1, 2, 2, 1, 1\}$

$$11 \times 3 + 4 \times 3 + 8 \times 1 + 2 \times 8 = 69$$

2. $S_1 = \{10, 12, 15, 255, 1, 2, 1, 1, 2, 2, 1, 1\}$

$$11 \times 1 + 8 \times 12 = 107$$

3. $S_1 = \{10\}$, $S_2 = \{12\}$, $S_3 = \{15\}$, $S_4 = \{255\}$, $S_5 = \{1\}$, $S_6 = \{2\}$, $S_7 = \{1\}$, $S_8 = \{1\}$, $S_9 = \{2\}$, $S_{10} = \{2\}$, $S_{11} = \{1\}$, $S_{12} = \{1\}$,

$$11 \times 12 + 4 \times 3 + 8 \times 1 + 1 \times 5 + 2 \times 3 = 163$$

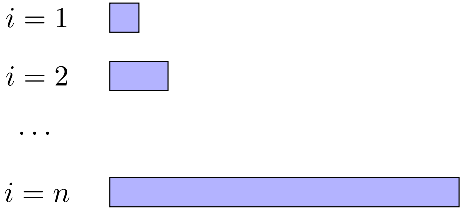
Conclusion: the first partition is better

Dynamic Programming Method

Subproblem: left boundary is always 1, right boundary is i

- Pixel sequences: $\{a_1, a_2, \ldots, a_i\}$
- Optimized function: $\mathrm{OPT}(i)$ is the minimal storage bits for $\{a_1, \ldots, a_i\}$

Computation order



Algorithm Design

OPT(i): the optimal storage for $\{a_1, a_2, \ldots, a_i\}$. Let S_m be the last segment, ℓ_m be its length. The iterative relation of OPT is:

$$\mathsf{OPT}(i) = \min_{1 \le \ell_m \le \min\{i, 256\}} \{\mathsf{OPT}(i - \ell_m) + \ell_m \times b_m + 11\}$$

$$b_m = \left\lceil \log(\max_{a \in S_m} \{a\}) \right\rceil \le 8$$

$$\mathsf{OPT}(0) = 0$$

$a_1, a_2, \ldots, a_{i-\ell}$ | $a_{i-\ell+1}, a_2, \ldots, a_i$

the first $i - \ell_m$ pixels
$\mathsf{OPT}(i - \ell_m)$
$S_1, \ldots, S_{m-1}$

m-th segment: ℓ_m pixels
$\ell_m \times b_m + 11$
S_m

Algorithm 12: Compress(I, n) //compute OPT(n)

OPT$(0) \leftarrow 0$;
for $i \leftarrow 1$ *to* n **do** //right boundary of subproblem
  OPT$(i) \leftarrow +\infty$, $L(i) \leftarrow 0$;
  for $\ell_m \leftarrow 1$ *to* $\min\{i, 256\}$ **do**
    $b_m =$ length$(i - \ell_m + 1, i)$;
    if OPT$(i) >$ OPT$(i - \ell_m) + \ell_m \times b_m + 11$ **then**
     update OPT(i), $L(i) \leftarrow \ell_m$;
  end
end

- ℓ_m denote is length of the last candidate segment S_m
- length(α, β) is the function that computes b_{max} for $I[\alpha, \beta]$
- $L(i)$ is the length of the last segment S_m with i as the right boundary (last segment in optimal partition for subproblem $[1, i]$): used for trace back partition.
- OPT$(i) \leftarrow +\infty$: simply trigger the iteration

Complexity: $O(256n)$

Demo

Input: $I = \{10, 12, 15, 255, 1, 2\}$. Suppose we have finish the computation of subproblems up to right boundary $i = 5$.

i	1	2	3	4	5	6
OPT(i)	15	19	23	42	50	?
$L(i)$	1	2	3	1	2	?

Demo

10	12	15	255	1	2

$\mathsf{OPT}(5) = 50$ $1 \times 2 + 11$ 63

Demo

10	12	15	255	1	2

$\mathsf{OPT}(5) = 50$ $\qquad 1 \times 2 + 11 \qquad 63$

10	12	15	255	1	2

$\mathsf{OPT}(4) = 42$ $\qquad 2 \times 2 + 11 \qquad 57$

Demo

10	12	15	255	1	2

$\mathsf{OPT}(5) = 50$ $\quad 1 \times 2 + 11 \quad 63$

10	12	15	255	1	2

$\mathsf{OPT}(4) = 42$ $\quad 2 \times 2 + 11 \quad 57$

10	12	15	255	1	2

$\mathsf{OPT}(3) = 23$ $\quad 3 \times 8 + 11 \quad 58$

Demo

10	12	15	255	1	2

$\text{OPT}(5) = 50$ $\quad 1 \times 2 + 11 \quad 63$

10	12	15	255	1	2

$\text{OPT}(4) = 42$ $\quad 2 \times 2 + 11 \quad 57$

10	12	15	255	1	2

$\text{OPT}(3) = 23$ $\quad 3 \times 8 + 11 \quad 58$

10	12	15	255	1	2

$\text{OPT}(2) = 19$ $\quad 4 \times 8 + 11 \quad 62$

Demo

10	12	15	255	1	2

$\mathsf{OPT}(5) = 50$ $\quad 1 \times 2 + 11 \quad 63$

10	12	15	255	1	2

$\mathsf{OPT}(4) = 42$ $\quad 2 \times 2 + 11 \quad 57$

10	12	15	255	1	2

$\mathsf{OPT}(3) = 23$ $\quad 3 \times 8 + 11 \quad 58$

10	12	15	255	1	2

$\mathsf{OPT}(2) = 19$ $\quad 4 \times 8 + 11 \quad 62$

10	12	15	255	1	2

$\mathsf{OPT}(1) = 15$ $\quad 5 \times 8 + 11 \quad 66$

Demo

10	12	15	255	1	2

$\mathsf{OPT}(5) = 50$ $\quad 1 \times 2 + 11 \quad 63$

10	12	15	255	1	2

$\mathsf{OPT}(4) = 42$ $\quad 2 \times 2 + 11 \quad 57$

10	12	15	255	1	2

$\mathsf{OPT}(3) = 23$ $\quad 3 \times 8 + 11 \quad 58$

10	12	15	255	1	2

$\mathsf{OPT}(2) = 19$ $\quad 4 \times 8 + 11 \quad 62$

10	12	15	255	1	2

$\mathsf{OPT}(1) = 15$ $\quad 5 \times 8 + 11 \quad 66$

10	12	15	255	1	2

$6 \times 8 + 11 \quad 59$

Trace Optimal Solution

Algorithm 13: Traceback$(L(n))$ (input is the trace table)

Output: optimal partition P

$k \leftarrow 1$; **while** $n \neq 0$ **do**

    $P(k) \leftarrow L(n)$;

    $n \leftarrow n - L(n)$;

    $k \leftarrow k + 1$;

end

reverse P;

- $P(k)$: the length of k-th segment
- Complexity: $O(n)$