CSOR W4231.002 Lecture 5: Feb 24, 2014 Spring 2014

In today's lecture, we will first discuss an optimal greedy algorithm for the problem of cache maintenance. We will then introduce the dynamic programming principle and apply it to matrix chain multiplication.

1 Cache maintenance

We consider the following problem. There is a main memory with n pages and a cache memory that can store at most k pages at any time. A sequence of requests for memory pages $r_1, r_2, \ldots, r_m$ is received.

In order to service a request, the corresponding page must be in the cache. Note that after the first k requests for distinct pages the cache will be full. At this point, a request for a page that is not in the cache results in a *cache miss*. Since the cache can store only k pages at any time, we will have to *evict* a page from the cache in order to bring in the requested page. We assume that a request is received and serviced within the same time step.

Therefore, at each time $1 \leq t \leq m$, we need to decide which page (if any) to evict from the cache so that all m requests are serviced at time m. We call such a sequence of eviction decisions a *schedule*; an algorithm that provides such a schedule is called a scheduling algorithm.

Our **goal** is to find the schedule that minimizes the total number of cache misses. To this end we need to come up with an algorithm that **schedules** the evictions in such a way that the number of cache misses after all requests are serviced is minimal.

For example, consider $n = 3$ and $k = 2$. Let a, b, c be the pages in the main memory and suppose we receive requests a, b, c, b, c, a, b. Let "$-$" stand for "no eviction"; then we may apply schedule S depicted below to satisfy this sequence of requests.

time t:	1	2	3	4	5	6	7
requests:	a,	b,	c,	b,	c,	a,	b
eviction schedule S:	$-$,	$-$,	a,	$-$,	$-$,	c,	$-$
cache contents:	$\{a\}$	$\{a, b\}$	$\{b, c\}$	$\{b, c\}$	$\{b, c\}$	$\{b, a\}$	$\{b, a\}$

Schedule $S = \{-, -, a, -, -, c, -\}$ evicts page a from the cache at time 3 and page c at time 6; this results in two cache misses, which is the best we can do here.

Note that this problem is an *offline* problem: the whole sequence of requests is input to the scheduling algorithm. However this problem naturally arises in an *online* setting. That is, requests arrive one at a time; at any time $1 \leq t < m$ request r_t needs to be serviced at time t, *before* requests $r_{t+1}, \ldots, r_m$ are received. Therefore, an algorithm that provides a schedule for the *online* problem can only base its eviction decision at time t on the requests it has seen so far and the decisions it has made so far. However analyzing the *offline* setting, i.e., looking for the optimal offline algorithm, is useful because it yields valuable insight for the online problem as we will discuss later.

1.1 Farthest-into-Future algorithm

Consider the following rule, which we call Farthest-into-Future:

Definition 1 *Farthest-into-Future: When the page requested at time i is not in the cache, evict from the cache the page that is needed the farthest into the future and bring in the requested page.*

In the example above the eviction schedule S is what Farthest-into-Future produces; we will use the shorthand FF and denote its schedule by S_{FF}.

A priori, it is not clear why this greedy algorithm is optimal. In fact, even less is clear: may we focus on algorithms that evict pages only when the current request is for a page that is not in the cache? One might think of bringing into the cache pages even when there is no cache miss and therefore no immediate reason for an eviction; this could perhaps allow for fewer cache misses in the future. To make the above concrete, we introduce the notion of a **reduced** schedule.

Definition 2 *A reduced schedule brings a page into the cache at time i only if the page is requested at time i and is not already in the cache.*

Such a schedule is called reduced because in some sense it performs the least amount of work at every step i. Note that FF is a reduced schedule.

Fact 1 *We can transform a non-reduced schedule into a reduced one that is at least as good, i.e., incurs at most the same number of evictions.*

The main idea of the proof is as follows. Let S' be a schedule that is not reduced and solves an instance of cache maintenance. We will transform it into a reduced schedule S as follows. Each time S' brings a page a from the main memory into the cache that is **not** requested at time i, we "pretend" that S brings in a as well. But in reality S leaves a in main memory and only brings it in at the first step $t = j > i$ where a is requested (so, as soon as a is requested). We can then charge the cache miss of S' at time $j > i$ to the eviction of S at the earlier time i. It follows that S performs at most as many evictions as S'.

Hence to find the optimal schedule we need consider only reduced schedules. Note that in a reduced schedule, an eviction occurs only when there is a cache miss, so the number of cache misses and the number of evictions is the same.

1.2 Optimality of Greedy

We will now use an exchange argument to transform any reduced schedule S into the schedule given by S_{FF} so that the quality (number of cache misses) of the original schedule S is not decreased.

Claim 1 *Let S be a reduced schedule that makes the same eviction decisions as S_{FF} through time $t = i$, i.e., through the first i items of the sequence of requests, for some i. Then there is a reduced schedule S' that makes the same eviction decisions as S_{FF} until time $t = i + 1$, i.e., for the first $i + 1$ items, and incurs no more cache misses than S does.*

First we will show that, **if** Claim 1 is true, **then** optimality of S_{FF} follows. We will basically apply the claim for every $0 \leq i \leq m - 1$, thus creating a new schedule $S' = S_{i+1}$ for every i.

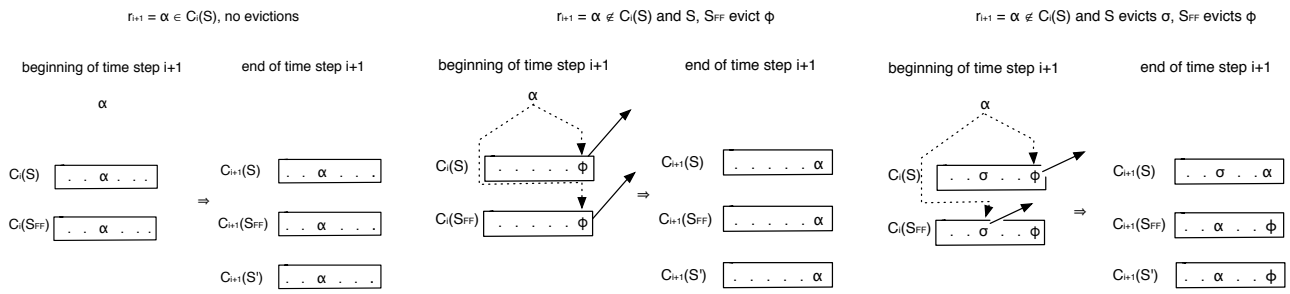
Proposition 1 *The schedule S_{FF} provided by the Farthest-into-Future algorithm is optimal.*

Proof. Let $cm(S)$ denote the **total** number of cache misses of schedule S. Let S^* be an optimal reduced schedule. Trivially, S^* follows (that is, makes the same eviction decisions as) S_{FF} up to time $t = 0$. By Claim 1, we can construct a reduced schedule S_1 that follows S_{FF} up to $t = 1$ and $cm(S_1) \leq cm(S^*)$.

Now S_1 is a reduced schedule that follows S_{FF} up to time $t = 1$. By Claim 1, we can construct a reduced schedule S_2 that follows S_{FF} up to $t = 2$ and $cm(S_2) \leq cm(S_1)$.

Now S_2 is a reduced schedule that follows S_{FF} up to time $t = 2$. By Claim 1, we can construct a reduced schedule S_3 that follows S_{FF} up to $t = 3$ and $cm(S_3) \leq cm(S_2)$.

Applying the claim for all subsequent $3 \leq i \leq m - 2$, we obtain that S_{m-1} is a reduced schedule that follows S_{FF} up to time $m - 1$. Then by Claim 1, we can construct a reduced schedule S_m that follows S_{FF} up to time m (hence $S_m = S_{FF}$), and $cm(S_m) \leq cm(S_{m-1})$. Tracing back all the inequalities, we conclude that $cm(S_m) \leq cm(S^*)$. Hence S_m is optimal. Since $S_m = S_{FF}$, it follows that S_{FF} is optimal. □



(a) Case 1 (b) Case 2a (c) Case 2b

Figure 1: Cache contents of S, S' and S_{FF} during time step $i+1$. Dots indicate that the rest of the cache contents are the same in all caches.

We will now prove Claim 1. Recall that $cm(S)$ is the **total** number of cache misses of schedule S.

Proof. Let $C_i(S)$ denote the contents of the cache of schedule S at time i. Since S and S_{FF} have made the same scheduling decisions up to time $t = i$, it follows that at the end of time step i, the contents of their caches are identical, i.e., $C_i(S) = C_i(S_{FF})$; also, the number of cache misses of S and S_{FF} so far is the same.

Consider the $i+1$-st request, $r_{i+1} = \alpha$. There are two cases.

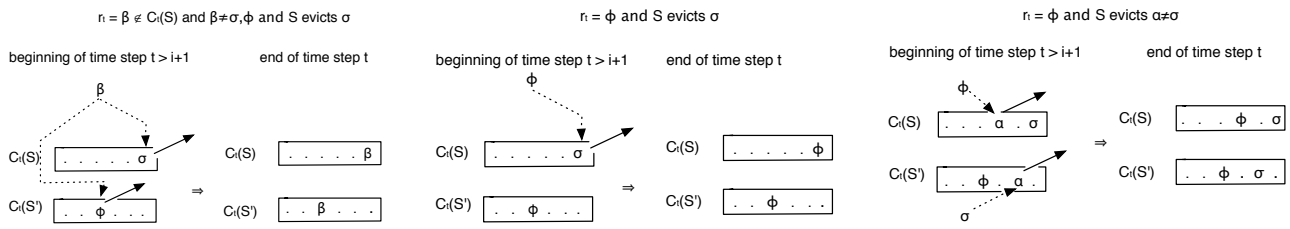
1. If $\alpha \in C_i(S)$ (hence α is also in $C_i(S_{FF})$), there is no cache miss for either schedule (see Figure 1(a)). Thus no eviction is necessary and $C_{i+1}(S) = C_{i+1}(S_{FF})$.

   In this case we can simply set $S' = S$. Then S' follows S_{FF} up to time $i+1$ (since S does), and trivially $cm(S') \leq cm(S)$.

2. If $\alpha \notin C_i(S)$ (hence α also not in $C_i(S_{FF})$), both schedules need bring α into their caches.

   (a) If S and S_{FF} both evict the same page ϕ from their caches (see Figure 1(b)), then we have $C_{i+1}(S) = C_{i+1}(S_{FF})$. In this case, we can again set $S' = S$: the same argument as above shows that S' follows S_{FF} up to time $i+1$ and $cm(S') \leq cm(S)$.

   (b) Now suppose that S evicts ϕ but S_{FF} evicts σ. (By definition of S_{FF}, this happens because σ is requested later in the future than ϕ.) Then at the end of time step $i+1$ the cache contents for the two schedules will differ in exactly one item (see Figure 1(c)): S_{FF} will have ϕ but S will have σ, so
   $$C_{i+1}(S_{FF}) = C_{i+1}(S) - \{\sigma\} + \{\phi\}.$$
   Since we want S' to agree with S_{FF} through time $t = i+1$, S' evicts σ from its cache as well. Hence
   $$C_{i+1}(S') = C_{i+1}(S_{FF}) = C_{i+1}(S) - \{\sigma\} + \{\phi\}.$$
   Now we need to ensure that S' will not incur more misses than S. The easiest way to guarantee this is by setting $S' = S$ *once* the cache contents of the two schedules are the same again (at the end of time step $i+1$ they differ in exactly one item). Thus, for $t > i+1$, we want to make $C_t(S')$ equal to $C_t(S)$ as soon as possible, while not incurring unnecessary misses. Once $C_t(S') = C_t(S)$ for some $t > i+1$, we can have S' behave like S for all future requests; then if between steps $i+2$ and t S' has not incurred more misses than S, we also have that $cm(S') \leq cm(S)$.
   How do we achieve this? From time $t = i+2$ onward, S' **behaves exactly like** S—thus the number of cache misses of S and S' up to time t is the same—**until** one of the following things happens for the first time.



(a) Case i. (b) Case A (c) Case B

Figure 2: Cache contents of S and S' during the first time step that one of i., A, B happens. Dots stand for pages that are the same in both caches.

i. Request r_t is for page $\beta \neq \phi, \sigma$, **and** $\beta \notin C_t(S)$ **and** S evicts σ.
Since $\beta \notin C_t(S)$, and $C_t(S)$ and $C_t(S')$ only differ in σ and ϕ (recall that S' behaves just like S after time $i+1$), it follows that $\beta \notin C_t(S')$ either.
In this case we have S' evict ϕ and bring in β as well (see Figure 2(a)). So now $C_t(S') = C_t(S)$ and we make S' follow S henceforth. Thus S' follows S_{FF} up to time $i+1$ and $cm(S') \leq cm(S)$.

ii. Request r_t is for page ϕ.

A. If S evicts σ then we're all set (see Figure 2(b)): in this case S' can simply access ϕ from its cache, no eviction is made and we have S' follow S from now on. Thus, S' follows S_{FF} up to time $i+1$ and, in this case, $cm(S') < cm(S)$.

B. If S evicts some other page, say $\alpha \neq \sigma$ from its cache, then we have S' evict α as well from its cache and we bring in σ from the main memory (see Figure 2(c)). Now $C_t(S') = C_t(S)$ and we can set $S' = S$. Since S' and S have the same number of evictions up to time t, and $S' = S$ henceforth, it follows that $cm(S') \leq cm(S)$.
However S' is no longer reduced: we brought in σ even though there was no request for σ at time t. So we now reduce S'. By Fact 1, the reduction of S' will have the same number of evictions as S' **and** still agree with S_{FF} up to time $i+1$: all the real evictions of the reduced S' will happen **after** time $i+1$. We return the reduced S' as the S' of the Claim.

iii. There is a request for σ. In fact this cannot occur: this is because S_{FF} evicted σ and not ϕ, hence one of cases i., ii. will happen first by the definition of S_{FF}.

□

1.3 The online problem

As mentioned at the end of Section 1, this problem has a natural **online** version. Experimentally the best scheduling algorithms for the online problem are variants of the Least Recently Used (LRU) principle: evict the page that was requested the longest ago. This is basically Farthest-into-Future reversed in time. Since LRU is a deterministic algorithm one can devise a sequence of online requests that will cause very bad performance of LRU as compared to the performance of the optimal offline algorithm (how?). However, LRU behaves well on average. The intuition behind the principle is locality of reference: a running program will generally keep accessing the things it's just been accessing.

Finally note that even though it is easy to come up with a greedy algorithm, showing optimality can be more challenging. We will discuss more greedy algorithms when we move into graph theory.

2 Matrix chain multiplication

We now move on to a different design principle, namely *dynamic programming*. Consider the following problem: we want to multiply three matrices, A_1, A_2 and A_3 of dimensions 6×1, 1×5, 5×2. How many arithmetic operations are necessary?

In general, let $C = A \cdot B$, where A is an $m \times n$ matrix and B is an $n \times q$ matrix; for $1 \leq i \leq m$ and $1 \leq j \leq q$ we have

$$C[i, j] = \sum_{k=1}^{n} A[i, k] \cdot B[k, j].$$

Entry $C[i, j]$ requires n scalar multiplications and $n - 1$ additions. Thus the number of arithmetic operations to compute $C[i, j]$ is dominated by the number of scalar multiplications; hence we will focus on the latter from now on. Since C has mq entries, the total number of scalar multiplications required to compute C is $n \cdot mq$.

Back to our example. We can compute the product $A_1A_2A_3$ in two ways:

1. First compute $A_1 \cdot A_2$, then multiply the resulting matrix by A_3. We need $6 \cdot 1 \cdot 5$ scalar multiplications to compute the first product, which is a 6×5 matrix. So we need $6 \cdot 5 \cdot 2$ more scalar multiplications to multiply this matrix by A_3, thus a total of 90 scalar multiplications.

2. Alternatively, first compute $A_2 \cdot A_3$, then multiply A_1 by the resulting matrix. We need $1 \cdot 5 \cdot 2$ scalar multiplications to compute the product $A_2 \cdot A_3$, which is a 1×2 matrix. So we need $6 \cdot 1 \cdot 2$ more scalar multiplications to multiply A_1 by this matrix. Thus we have a total of 22 scalar multiplications.

Note that the second method requires approximately 75% fewer operations than the first. Thus the order in which we decide to parenthesize the product of matrices is important. Formally, the matrix chain multiplication problem is as follows.

Definition 3 *A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrices, surrounded by parentheses.*

For example, $((A_1A_2)A_3)$ and $(A_1(A_2A_3))$ are fully parenthesized.

Input: n matrices $A_1, A_2, \ldots, A_n$, with dimensions $p_{i-1} \times p_i$, for $1 \leq i \leq n$. (The dimensions are defined so that the multiplication of adjacent matrices is always valid.)

Output: the optimal parenthesization of the input and its cost; that is, the parenthesization that yields the minimum number of scalar multiplications, as well as that number.

In the example above, the optimal parenthesization is $(A_1(A_2A_3))$ and its cost is 22. Note that we might want both the optimal parenthesization and its cost, or just the cost. Also, we do not compute the actual product of the matrices here.

2.1 A first attempt: brute-force

We might be tempted to go over every possible parenthesization and find the optimal one by **brute force**. Let $P(n)$ be the number of possible parenthesizations of a product of n matrices. By definition, for some $1 \leq k \leq n - 1$, we can write the product as

$$(A_1A_2 \cdots A_k)(A_{k+1} \cdots A_n).$$

The number of possible parenthesizations for the above product is the number of ways to combine every possible parenthesization of the first subproduct with every possible parenthesization of the second

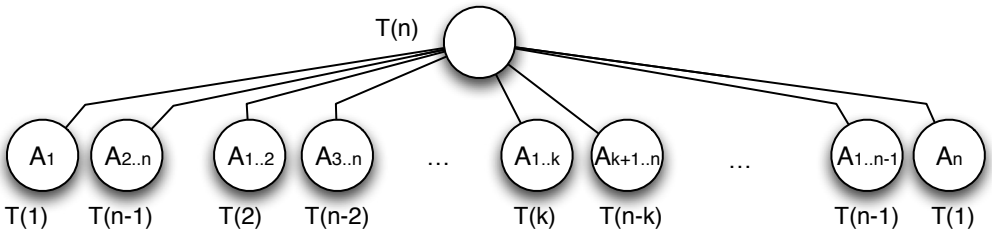


Figure 3: Levels 0 and 1 of the tree of recursive calls for the recursive algorithm of Section 2.2.

subproduct. Hence it is given by $P(k) \cdot P(n-k)$. Since we can break the input in two parts in $n-1$ ways (there are $n-1$ choices for k), we have the following recurrence:

$$P(n) = \sum_{k=1}^{n-1} P(k) \cdot P(n-k), P(2) = 1, P(1) = 1, P(0) = 0.$$

To get a quick and very crude (yet sufficient for our purposes) lower bound for $P(n)$ note that

$$P(n) \geq P(1) \cdot P(n-1) + P(2) \cdot P(n-2) \geq P(n-1) + P(n-2) \tag{1}$$

Now by strong induction on n we can show that $P(n) \geq F_n$, the n-th Fibonacci number. Thus (e.g., by Problem 6a in Homework 1), we have $P(n) = \Omega(2^{n/2})$. Hence brute force requires exponential time.

2.2 A second attempt: divide and conquer

Given the algorithmic techniques we have seen so far, divide and conquer seems like a good candidate for this problem. Let $A_{1..n}$ be the matrix that results from multiplying A_1 through A_n. By definition, the fully parenthesized product on n matrices is the product of two fully parenthesized subproducts, that is, for some $1 \leq k \leq n-1$,

$$A_1 \cdots A_n = (A_1 \cdots A_k)(A_{k+1} \cdots A_n) = A_{1..k} A_{k+1..n}$$

Now consider the optimal parenthesization of $A_{1..n}$, i.e., the parenthesization that results in the minimal number of scalar multiplications. This parenthesization can also be expressed as the product of two fully parenthesized subproducts of matrices. Say $A_{1..k^*} A_{k^*+1..n}$. is the optimal parenthesization, for some $1 \leq k^* \leq n-1$; then, given k^*, each of the subproblems $A_{1..k^*}, A_{k^*+1..n}$ must be an optimal parenthesization itself: if either subproblem is not optimal, we can replace it by a better one and thus get a better overall solution (since, given k^*, the cost of the overall solution is completely determined by the costs of the subproblems and the cost of multiplying $A_{1..k^*}$ by $A_{k^*+1..n}$).

Thus we can compute the cost of the optimal parenthesization recursively. However, unlike the recursive algorithms we saw so far (think Mergesort), where we had a clear idea where the division point should be, here we do not know where the division point is. Hence we need consider all possible division points: in order to find the optimal parenthesization of $A_{1..n}$ we need to solve $n-1$ subproblems, arising for every $1 \leq k \leq n-1$. Each subproblem involves solving $A_{1..k}$, and $A_{k+1..n}$, and combining them.

This is bad: unlike Mergesort, where all the work goes into combining the subproblems, here all the work goes into generating subproblems. Almost always, this leads to exponential time recursive algorithms. Indeed, let's look into the tree of recursive calls of our recursive algorithm more carefully. The root and first-level of this tree are depicted in Figure 3. Let $T(n)$ be the time required to parenthesize a product of n matrices, where $T(1) \geq 1$, $T(2) \geq 2$. Since finding the optimal of the $n-1$ subproblems

at level 0 takes time $\Omega(1)$ and combining the subproblems at level 1 also takes time $\Omega(1)$ we obtain the following recurrence for $T(n)$

$$T(n) \geq \sum_{k=1}^{n-1}[T(k) + T(n-k) + 1] + 1,\ T(1) \geq 1,\ T(2) \geq 2.$$

Again, we can lower bound $T(n)$ as $T(n) \geq T(n-1) + T(n-2)$ and show by strong induction that $T(n) \geq F_n$. Thus the running time of the recursive algorithm is $\Omega(2^{n/2})$.

2.3 Elements of dynamic programming

However, we are not that far from a polynomial time solution. Recall the Fibonacci problem from Homework 1: even though the recursive algorithm was exponential time, you came up with an iterative approach that only required quadratic time in n. This was possible because of the following properties of the problem:

1. **Overlapping subproblems:** The recursion tree of the recursive algorithm presents a spectacular redundancy in computations. Subproblems are **overlapping** in different branches of the tree and the same subproblems are computed again and again.
2. **Easy to compute recurrence for combining the smaller subproblems:** using $F_n = F_{n-1} + F_{n-2}$ we only spend a constant amount of work to combine the solutions to the smaller subproblems into a solution for a bigger subproblem.
3. **Iterative, bottom-up computations:** we computed the subproblems from smallest (F_0, F_1) to largest (F_n), iteratively.
4. **Small number of subproblems:** we only needed to solve $n-1$ subproblems.

We will show that the matrix chain multplication problem exhibits similar properties; that is, we will

1. Develop a **recurrence** for the cost of the optimal parenthesization of the input in terms of the costs of the optimal parenthesizations of appropriate subproblems (we already argued that the optimal solution to the problem implies optimal solutions to the subproblems); this will allow us to solve the subproblems in a **bottom-up** fashion.
2. Show that the total number of subproblems is polynomial in n.
3. Spend polynomial work for combining the subproblems.
4. Solve the subproblems in a **bottom-up** fashion, from smallest to largest.

2.3.1 The recurrence for the cost of the optimal solution

Let's start with developing the recurrence for the cost of the optimal solution. First note that the number of scalar multiplications required to multiply matrices $A_{i..k} \cdot A_{k+1..j}$ is $p_{i-1} \cdot p_k \cdot p_j$ since $A_{i..k}$ is a matrix of dimensions $p_{i-1} \times p_k$, while $A_{k+1..j}$ is a $p_k \times p_j$ matrix.

First attempt: let $c(1,j)$ be the optimal cost for computing $A_{1..j}$ for $1 \leq j \leq n$. Then

$$c(1,j) = \begin{cases} 0 & \text{, if } j = 1 \\ \min_{1 \leq k \leq j-1} \left\{ \overbrace{c(1,k)}^{A_{1..k}} + \overbrace{c(k+1,j)}^{A_{k+1..j}} + \overbrace{p_0 p_k p_j}^{A_{1..k} \cdot A_{k+1..j}} \right\} & \text{, if } j > 1 \end{cases}$$

This does not quite work: the subproblems are not both of the same form as the original problem (why?). Hence we need to introduce one more variable and consider more subproblems.

Second attempt: let $c(i,j)$ be the optimal cost for computing $A_{i..j}$ for $1 \leq i \leq j \leq n$. We have

$$c(i,j) = \begin{cases} 0 & \text{, if } i = j \\ \min_{i \leq k \leq j-1} \left\{ c(i,k) + c(k+1,j) + p_{i-1}p_k p_j \right\} & \text{, if } i < j \end{cases}$$

Here the subproblems are of the same form as the original problem. Note that there are only $\Theta(n^2)$ subproblems. Also, **if** we compute the subproblems from smaller to larger, then we only need $\Theta(j-i) = \Theta(n)$ work to compute a subproblem from smaller subproblems: each term inside the minimum operand requires constant time.

2.3.2 Bottom-up computation of subproblems

So what we need to do is show how to compute the subproblems in a **bottom-up** fashion, from smaller to larger. We define matrices m and s, of dimensions $n \times n$ and $(n-1) \times (n-1)$ respectively, such that

$$\begin{aligned} m[i,j] &= c(i,j) \text{ for } 1 \leq i \leq j \leq n \\ s[i,j] &= k \textbf{ iff} \text{ the optimal parenthesization of } A_{i..j} \text{ is } A_{i..k}A_{k+1..j} \text{ for } 1 \leq i \leq n-1 \text{ and } 2 \leq j \leq n \end{aligned}$$

We will fill up the upper triangle of matrices m and s, starting from the main diagonal and filling diagonal by diagonal moving upwards. The last entries to be filled are $m[1,n]$, $s[1,n]$; $m[1,n]$ is the cost of the optimal parenthesization of $A_{1..n}$. There are $\Theta(n^2)$ entries to fill in, and each entry requires $\Theta(j-i) = O(n)$ work, hence this algorithm runs in $\Theta(n^3)$ time (see your textbook for pseudocode). The total space required for m and s is $\Theta(n^2)$.

Note that if we are only interested in the *cost* of the optimal solution then we do not have to store s. However if we want the optimal solution as well, then s allows us to reconstruct the optimal solution faster: given s, we recover the optimal parenthesization for every pair (i,j) in $O(1)$.

2.3.3 Memoized recursion

Instead of computing the subproblems iteratively as described above, we could use the recursive algorithm we came up with originally, together with the $n \times n$ matrix m to store optimally solved subproblems. In detail, we could initialize m to ∞ (i.e., the largest number in the system) above the main diagonal and to 0 on the main diagonal. When we need to compute a subproblem, we look up its value in matrix m: if it is ∞, then we proceed with computing the subproblem; we store its value once computed; else, we directly use its value from m.

The technique of storing the values of the solved subproblems is referred to as *memoization*. The memoized recursive algorithm described above avoids solving over and over subproblems it has already solved, which is the reason why the original recursive algorithm in Section 2.2 was exponential time. The running time of this algorithm (see your textbook for pseudocode) is again $O(n^3)$.

Usually, when we think of a dynamic programming algorithm, we think of an iterative implementation.

2.3.4 Dynamic Programming vs Divide and Conquer

To conclude, note that dynamic programming is similar to divide and conquer, in that it combines solutions to subproblems in order to generate a solution to the whole problem. However, divide and conquer starts with a large problem and divides it into small pieces, while dynamic programming works from the bottom up, solving the smallest subproblems first and building optimal solutions to steadily larger problems.