Arithmetic Operations and Chained Matrix Multiplication

Appendix D, Ch 4.2 (pp. 75-83), Ch 30/30.1 (pp 898-901)

Problem 2-3 (page 41), Problem 30-1 (pp. 920), Section 15-2 (pp. 370-378)

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2019; distribution outside this group of students is NOT allowed.*

1. Arithmetic Operations: Models of computation. When we perform arithmetic operations, sometime we do not distinguish one operation from the other. Thus an operation count might include additions, subtractions, multiplications, etc. Sometimg we distinguish between additions and subtractions on one hand, and multiplications on the other, and we do not allow divisions. Such models for arithmetic operations are called **word models**. We use such models when we talk about operations on polynomials and matrices or generic algorithms. When we talk about operations on say integers, the bit length of integers becomes important. Then we talk about a **bit model** of computation. Then insteading of talking about additions, subtractions or multiplications, we analyze the bit count of such operations, and thus the end result is expressed in terms of **bit operations**. In the word model, we sometime employ what is know as a **straight line program (SLP)** model of computation: in simplistic terms think of programs that do not allow control structures (i.e. if-then-else, and goto statements), or iterative structures (i.e. for-loops, while-loops) and thus consist of straighforward (assignment) statements.

2. Bit model fundamentals. The arithmetic model that we adapt for the remainder of this discussion on integer multiplication is the bit model. An operation involving two integers costs a number of bit operations proportional to the number of bits of the two integers. If the input consists of two numbers a, b one may assume that $a > b$ in general. Thus the input might consist of an n-bit integer a and m-bit integer b and $a > b$ also implies $n \geq m$. Some other times we would just say that the input consists of two n-bit integers a, b. The former case can be reduced to the latter if we pad b with some leading zeroes to make b, with the leading zeroes have as many bits as a.

3. Fact Floors/Ceilings. For any x, we have that $x \leq \lceil x \rceil$ and $x \geq \lfloor x \rfloor$.

4. Theorem: *Bit Representation.* The number of bits `bit(n)` to represent integer $n > 1$ is $bit(n) = \lceil \lg(n+1) \rceil = \lfloor \lg n \rfloor + 1$.
Proof. See PS1.Problem 2 for details.

5 Example. Integer 3 requires 2 bits, 15 requires 4 bits, but 16 requires 5 bits for its representation.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

5 Example. Integer 3 requires 2 bits, 15 requires 4 bits, but 16 requires 5 bits for its representation.

6 Internal Representation: Little endian. Suppose we have integer $a = 177$. By default it is written in denary (i.e. base-10, or radix-10). Sometimes we say so with a subscript i.e. $a = 177_{10}$. The binary representation of a is $a = 10110001_2 = 10110001 = [10110001]$. The former with the subscript 2 is preferable and non confusing, the bracketed one is preferable to the middle one; one can read the middle one as an integer about 10 million. We can represent the bits of an integer internally in the form of an array A. Thus A is of length 8 for this example where $8 = \lceil \lg(177+1) \rceil$. Then $A[0] = 1, A[1] = 0, A[2] = 0$. Thus the internal representation represents in the lowest index the least-significant (i.e. rightmost) bit of the integer. Thus A can be summarily written $A = [1, 0, 0, 0, 1, 1, 0, 1]$. This is the little-endian representation of the bits of a right to left i.e. from least-significant (right-most) to most-significant (left-most). In a big-endian representation stored in A', we would have $A' = [1, 0, 1, 1, 0, 0, 0, 1]$ and we would store the bits left to right, most-significant to least-significant.

7 Multiplication by 2^k, and division by 2^k. Let $b = 5_{10}$. Then $b = [101] = 101_2$. How much is $b \times 2^3$. Obviously $b \times 2^3 = 5 \times 8 = 40$. In binary this is [101000] i.e. b shifted left 3 positions, and that 3 is the exponent of 2^3. Likewise $b/2^2$ is equal to $5/4 = 1.25$. For $b = [101]$ the decimal point is omitted but is to the right of the least significant digit i.e. $b = [101.]$. Then $b/2^2$ is equivalent to a shift right by two positions and thus $b/2^2 = [1.01]$. If bit positions on the left of the decimal point are to zero or positive powers, to the right of the decimal point would be to negative powers. Thus $b/2^2 = [1.01] = 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} = 1 + 1/4 = 1.25$.

8 Divide and Conquer. Integer $a = 177_{10}$ has $n = 8$ bits. We can split a into two parts or halves, the most-significant half and the least-significant half. We can also call them leftmost and righmost halves, or high and low halves. Thus using the latter terminology, $a = [10110001]$ becomes $a = a_h a_l$ where $a_h = [1011]$ and $a_l = [0001]$. Converting into denary, $a_h = 11_{10}$ and $a_l = 1_{10}$. How does a, a_l, a_h relate to each other? Well, $a = a_h \times 2^4 + a_l$. Since $a_h = [1011]$ then $a_h \times 2^4 = [10110000]$. add to it the a_l term and the answer follows.

In general, $a = a_h \times 2^{n/2} + a_l$.

9 Polynomials: Degree and Degree-bound. A polynomial such as $f(x) = x^7 + x^5 + x^4 + 1$ is of degree 7 as the largest exponent of x is a 7. We write $degree(f) = d(f) = 7$. The degree-bound i.e. $db(f) = 8$, that is $db(f) = d(f) + 1$. Why so? This is the amount of space required to store f into F, i.e. the length of F. Adapting the little-endian representation $F = [1, 0, 0, 0, 1, 1, 0, 1]$.

10 Duality of storing the coefficient of polynomials and the bits of integers. We notice that F of f is identical to the A of a of paragraph 8 and earlier. If you wonder why, evaluate $f(2) = f(x)|_{x=2}$ which gives $f(2) = 177$! Thus there is a connection between the bit-model for integer operationd and the word-model for polynomial operations. Some results can translate from one model to the other!

11 Addition and Subtraction of polynomials. Let $f(x) = x^7 + x^5 + x^4 + 1$ with $F = [1, 0, 0, 0, 1, 1, 0, 1]$ and $g(x) = x^7 + 1$ with $G = [1, 0, 0, 0, 0, 0, 0, 1]$. How much is $f(x) + g(x)$? Well obviously $h(x) = f(x) + g(x) = 2x^7 + x^5 + x^4 + 2$, and thus $H = [2, 0, 0, 0, 1, 1, 0, 2]$. Note that there is no duality anymore between $h(x)$ or H and binary integers. There are 2s in H in addition to the binary digits 0 and 1! How long does it take to add or subtract two polynomials? It depends on the degree bound or the maximum degree bound i.e. it is dependent on $\max\{n, m\}$. The latter is always $\max\{n, m\} \leq n + m$. Thus we prefer to write that in the word model addition or subtraction of two polynomial of degrees n, m or degree bounds n, m is $O(n + m)$. (The big-Oh takes care the difference by one between degree and degree bound.)

```
f:  d(f) = n , db(f) = n+1    then    d(f+g)=max(n,m) ,  db(f+g)=max(n,m)+1
g:  d(g) = m , df(g) = m+1    then    d(f*g)= n+m     ,  db(f*g)= n+m+1
```

12 Multiplication of polynomials. For $g(x) = x^7 + 1$, to compute $g(x) \times x^2$ we have $c(x) = x^9 + x^2$, and thus $C = [0, 0, 1, 0, 0, 0, 0, 0, 0, 1]$. Thus C is G shifted right two positions. Thus multiplication of a generic polynomial $f(x)$ with monomial x^k will cost (see table above) $O(n + k)$ since the F of f with $db(f) = n + 1$ will be shifted k positions and thus the result would be of degree $n + k$ and degree-bound $n + k + 1$. This is similar to paragraph 7. If we multiply $f(x) * g(x)$ we need to multiply all term of f with all terms of g. This will cost us $db(f) * db(g)$ multiplications plus no more the same number of additions to combine same power monomials (eg. x^7 as the product of x^7 of f and 1 of g or the other way around.) We sometimes right that the cost of polynomial multiplication if done the high-school way is $O(d(f)d(g))$ thus simplifying the $O(db(f)db(g)) = O((d(f) + 1)(d(g) + 1))$.

(c) Copyright A. Gerbessiotis.

13 Multiple Polynomial Multiplication: Example. Say we have three polynomical f, g, h with degree $3, 4, 5$. Let the cost of multiplying two polynomial of degrees n, m be $n \times m$. (If this is too simplistic, make the cost $(n+1)(m+1)$ for further practice.)

What is the best way to multiply the three polynomials. Polynomial and integer multiplication is associative which means $(f(x)g(x))h(x) = f(x)(g(x)h(x))$.

Say we multiply $p_1(x) = f(x)g(x)$. This is at a cost of $3 \times 4 = 12$. Then $d(p_1) = 7$. Finally we multiply $p_1(x)h(x)$ at a cost of $7 \times 5 = 35$. The total cost is $35 + 12 = 47$ and $f(x)g(x)h(x)$ has degree $7 + 5 = 12$.

Say we multiply $q_1(x) = g(x)h(x)$. This is at a cost of $4 \times 5 = 20$. Then $d(q_1) = 9$. Finally we multiply $q_1(x)f(x)$ at a cost of $9 \times 3 = 27$. The total cost is $20 + 27 = 47$ and $f(x)g(x)h(x)$ has degree $7 + 5 = 12$.

Same answer, same degree and same cost though individual costs differ. Do you have a proof?

We derive the first elementary results.

14 Divide and Conquer. Let $f(x) = x^7+x^5+x^4+1$ with $F = [1, 0, 0, 0, 1, 1, 0, 1]$. We can split $f(x)$ in the high-half and low-half the same way we did this for integers in paragraph 8. But before we do so we need to write the missing terms of $f(x)$. Thus $f(x) = x^7+0\cdot x^6+x^5+x^4+0\cdot x^3+0\cdot x^2+0\cdot x^1+1$ with $F = [1, 0, 0, 0, 1, 1, 0, 1]$. Then $f(x) = f_h(x)\times x^{n/2}+f_l(x) = f_h(x)\times x^4+f_l(x)$, where $f_h(x) = x^3 + 0 \cdot x^2 + x^1 + x^0$ and $f_l(x) = 0 \cdot x^3 + 0 \cdot x^2 + 0 \cdot x^1 + 1$. Moreover $F = [F_l, F_h]$ with $F_l = [1, 0, 0, 0]$ and $F_h = [1, 1, 0, 1]$.

15 Integer and Polynomial Operations.

```
  f, d(f)=n , g, d(g)=m          a, bit(a)=n , b, bit(b)=m

f(x)+g(x)     O(n+m)            a+b          O(n+m)          [ also O(max(n,m))]
f(x)-g(x)     O(n+m)            a-b          O(n+m)          [ also O(max(n,m))]
f(x)*g(x)     O(n*m)            a*b          O(n*m)
f(g)*(x**k)   O(n+k)            a*(2**k)     O(n+k)          [ for integers: also a/(2**k)]
```

Another way to formulate 15 for integer bit operations (and also polynomial) is to adjust the two integers to the size of the larger one (and we assumed early one in this discussion $a > b$).

15.1 Addition/Subtraction of n-bit integers. The addition or subtraction of two n-bit numbers requires $O(n)$ bit operations by the trivial elementary school algorithm.

Another way to formulate 15 for integer bit operations is if we have no info about n, m. This implicitly uses Theorem 4 that relates n to $bit(n)$ or in general a to $bit(a)$, and b to $bit(b)$.

15.2 Addition/Subtraction: alternative description. The addition/subtraction of a and b requires $O(bit(a) + bit(b))$ bit operations i.e. $O(\lg a + \lg b)$ bit operations.

Given the assumption that $a > b$ we can also state.

15.3 Addition/Subtraction: alternative description. The addition/subtraction of a and b, given that $a > b$ requires $O(bit(a))$ bit operations i.e. $O(\lg a)$ bit operations.

Likewise for integer multiplication we have the following alternative formualtions.

15.4 Multiplication of n-bit integers. The multiplication of two n-bit (positive) integer numbers requires $O(n^2)$ bit operations by the trivial elementary school algorithm. The result can be up to $2n$ bit long.

15.5 Multiplication a and b positive integers. The naive multiplication of a and b such that $c = ab$ requires $O(bit(a) \cdot bit(b)) = O(\lg a \cdot \lg b)$ bit operations. The bit representation of c requires $O(\lg a + \lg b)$ bits.

15.6 Multiplication by 2^k The multiplication of a by 2^k requires $O(bit(a) + k) = O(\lg a + k)$ bit operations.

This is by writing down a which requires $bit(a)$ bits and multiplication by 2^k is a left-shift operation of k bit positions, filling with trailing zeroes.

15.7 Integer Division by 2^k The integer division of a by 2^k requires $O(bit(a) + k) = O(\lg a + n)$ bit operations.

Integer division by 2^k is equivalent to a right-shift operation. (Note that we ignore what happens to the right of the decimal point.)

15.8 Integer Division. If a and b are positive integers, the division of a by b, $a > b$ creates an integer quotient q and integer remainder r such that $a = qb + r$, where $0 \leq r < b$, and $q \leq a$. Since a is $bit(a)$ bits, the quotient q is $bit(a)$ long as well, Since $r < b$, r is $bit(b)$ long.

(c) Copyright A. Gerbessiotis.

16 Theorem: Karatsuba-Ofman algorithm (for integer multiplication). The Karatsuba-Ofman algorithm for multiplying two n bit numbers requires $O(n^{\lg 3})$ bit operations, faster than the algorithm of 15 that requires $O(n^{\lg 4}) = O(n^2)$ bit operations.

The idea behind the Karatsuba-Ofman algorithm is equivalent to reducing the number of multiplications in multiplying two degree 1 polynomials from 4 into 3. Consider for example the product $f(x) \cdot g(x)$ where $f(x) = ax + b$ and $g(x) = cx + d$, where a, b, c, d are the scalar coefficients of the two polynomials. The naive polynomial multiplication requires $f(x)g(x) = acx^2 + (ad + bc)x + bd$ 4 scalar multiplications, involving $a \cdot c$, $a \cdot d$, $b \cdot c$ and $b \cdot d$. This can be reduced to 3 if we observe that $ad + bc = (a + b)(c + d) - ac - bd$. Therefore if the first two multiplications are ac, bd then in order to compute the coefficient of x in $f(x)g(x)$ we don't need to compute the two terms ad, bc separately in two multiplications but rather their sum. This can be done in a single third multiplication that computes $(a + b)(c + d)$ and the subtracts from it the results of the first two multiplications, i.e. ac and bd already computed and stored in auxiliary storage (variables). We thus manages to reduce the number of multiplication from 4 to 3; the negative side effect is that the number of additions/subtractions has increased from 1 to 4.

16.1. Theorem: Karatsuba-Offman algorithm (for polynomials). Two degree bound n polynomial can be multiplied in the word model within $O(n^{\lg 3})$ operations.

Proof (of Theorem 16). In order to simplify exposition, without loss of generality, let n be an even power of two. Let a, b be the two n-bit numbers. The n bit number a consists of $n/2$ least significant bits and $n/2$ most significant bits respectively. Let a_h, a_l be the integer representation of these $n/2$ bits. Then $a = a_h 2^{n/2} + a_l$ and similarly $b = b_h 2^{n/2} + b_l$ Then

$$ab = (a_h 2^{n/2} + a_l)(b_h 2^{n/2} + b_l) = a_h b_h 2^n + (a_h b_l + a_l b_h)2^{n/2} + a_l b_l = a_h b_h 2^n + ((a_h + a_l)(b_h + b_l) - a_l b_l - a_h b_h)2^{n/2} + a_l b_l$$

```
   KaratsubaOfman(a,b)
1. (a_h a_l) =a;
2. (b_h b_l) =b;
3. t1 = KaratsubaOfman((a_h + a_l ),(b_h + b_l));
4. t2 = KaratsubaOfman(a_h , b_h);
5. t3 = KaratsubaOfman(a_l , b_l);
6. t4 = t1 - t2 -t3;
7. t2 = t2 << n; // t2 =t2 * 2**n
8. t4 = t4 << (n/2); // t4 = t4 * 2**(n/2)
6. return ( t2 + t4 + t3);
```

Karatsuba-Ofman Multiplication: A missing step

Therefore in order two multiply two n bit numbers a and b we group these two n-bit numbers into groups of $n/2$ bits and thus get a_h, b_h, a_l, b_l. We then perform three multiplications, one is $a_l b_l$ (line 5), the other $a_h b_h$ (line 4) and the third is the $((a_h + a_l)((b_h + b_l)$ (line 3) of $n/2$-bit integers. In addition, the latter operation requires 2 additions of $n/2$ bit numbers in $O(n)$ time and to get ab we need to perform two additional addition and two subtractions for a total of $O(n)$ as well plus two shift operations. Multiplication by 2^n or $2^{n/2}$ is a shift left operation requiring $O(n)$ operations. Therefore the running time $M(n)$ to multiply a and b is $M(n) = 3M(n/2) + \Theta(n)$, where $3M(n/2)$ is due to the three multiplications of $n/2$ bit numbers and $O(n)$ includes additions and shifts. A solution of the recurrence by the master method yields $M(n) = \Theta(n^{\lg 3}) = \Theta(n^{1.58496})$.

17 Detail. $(a_h + a_l)$ and $(b_h + b_l)$ may not be $n/2$ bit integers any more when step 3 is performed. In fact they may be $n/2 + 1$-bit integers. How one can fix this "tiny" problem? This is easily fixable as follows. Write $a_h + a_l$ as follows. $a_h + a_l = a_h 2^{n/2} + a_1$ (similarly for $b_h + b_l$). Note that $a_h + a_l$ can be represented by up to $n/2 + 1$ bits: a_h is either a 0 or 1 i.e. a bit. Then the product $(a_h + a_l)(b_h + b_l) = a_h b_h 2^n + (a_h b_1 + b_h a_1)2^{n/2} + b_1 a_1$. The product $a_h b_h$ is easy to find as it involves two 1-bit numbers. So are the products $a_h b_1$ (the product is either b_1 or 0 depending on whether a_h is 1 or 0) and $b_h a_1$ (either a_1 or 0). The only complication arises from $a_1 b_1$: both numbers are $n/2$ bit integers and the product is a KaratsubaOfman multiplication of two $n/2$-bit integers. The fix thus consists of one $n/2$-bit multiplication plus $O(n)$ bit operations. An easier way to resolve this problem is by rewriting the product as follows.

$$ab = (a_h 2^{n/2} + a_l)(b_h 2^{n/2} + b_l) = a_h b_h 2^n + (a_h b_l + a_l b_h)2^{n/2} + a_l b_l = a_h b_h 2^n + ((a_h - a_l)(b_l - b_h) + a_l b_l + a_h b_h)2^{n/2} + a_l b_l$$

For the polynomial version of the Karatsuba-Ofman algorithms let $a(x) = a_h x^{n/2+1} + a_l$ and $b(x) = b_h x^{n/2+1} + b_l$, where a_h, b_h, a_l, b_l are now polynomials of degree $n/2$ (or less).

18 Example $a = 20$ **and** $b = 200$**.**

```
 a= 20 = [ 0001 0100]   ah = [0001] =  1 , al = [0100] = 4
 b=200 = [ 1100 1000]   bh = [1100] = 12 , bl = [1000] = 8
  ah*bh = 1x12  = [1100]   ah*bh * (2**n) = ah*bh*(2**8) =              [1100 0000 0000]
  al*bl = 4x 8  = [0010 0000]                             =              [0000 0010 0000]
((ah+al)*(bl+bh)-ah bh - al bl)*(2**n/2)
((1+4)  *(12+8) - 12   - 32   )*(2**4)=56*(2**4)=
   [ 0011 1000] * (2**4)                                  =          +   [0011 1000 0000]
                              Total                       =              [1111 1010 0000] = 4000
```

19 Question. For a given input x and (positive) integer n, how fast can we raise x to the power of n, i.e. compute x^n?
We are going to answer this question in the word model.
This topic is discussed slightly modified in Corollary 31.35 of page 878-880 of CLRS.

19.1 Theorem Exponentiation x^n can be evaluated in at most $2\lg n$ multiplications.

Proof. Let $k = \lfloor \lg n \rfloor$. We first compute all power of two products

$$x, y_1 = x^2, y_2 = y_1^2 = x^4, y_3 = y_2^2 = x^8, \ldots y_k = y_{k-1}^2 = x^{2^k}$$

This requires k multiplications (`Count1`) by performing repeated doubling. Note that $y_k \leq x^n \leq y_k^2$. We then find the binary representation of n i.e. $n = b_0 + b_1 2^1 + \ldots + b_k 2^k$, where $b_i = 0, 1$. Then

$$x^n = x^{b_0 + b_1 2^1 + \ldots + b_k 2^k} = x^{b_0} x^{b_1 2^1} \ldots x^{b_k 2^k}$$

In general, let m of the b_i's be 1 (i.e. the remaining b_i's are zero), where $m \leq k+1$. Let these m b_i's be $b_{l_1}, b_{l_2}, \ldots b_{l_m}$. Then $x_n = x^{2^{l_1}} x^{2^{l_2}} \ldots x^{2^{l_m}} = y_{l_1} y_{l_2} \ldots y_{l_m}$, i.e. x_n requires $m - 1 \leq k + 1 - 1 = k$ multiplications (`Count2`) for a total of $k + k = 2k = 2\lfloor \lg n \rfloor \leq 2 \lg n$ M's, if we combine `Count1` and `Count2`.

```
   Power( x, n)  // Iterative Version
1.  k = floor(lgn); X[0]=x;
2.  for (i=1;i<=k;i++)
3.   X[i] = X[i-1]*X[i-1] // X[i]=x**(2^i)
4.  n= B[k]B[k-1]...B[0] //binary representation of n
5.  if (B[0] == 1)  P=X;
6.   else           P=1;
7.  for(i=1;i<=k;i++);
8.   if (B[i]==1)  P= P* X[i];
9.  return(P);
```

```
   Power(x,n)  // Recursive version
1. if (n==0) return(1);
2. if (n==1) return(x);
3. if mod(n,2) ==1
4.    return(x*Power(x*x,(n-1)/2));
5. else
6.    return(  Power(x*x,n/2));
```

Example. Let $n = 13$. Compute x^n for some x. Since $n = 13$, then $k = 3$. We first compute, $y_1 = x^2$, $y_2 = y_1^2 = x^4$, and $y_3 = y_2^2 = x^8$ with three M's. Given that the binary representation of 13 is 1101, then $x^{13} = x^{1\cdot 8 + 1\cdot 4 + 0\cdot 2 + 1\cdot 1} = y_3 \cdot y_2 \cdot x$, i.e. x^n can be computed with 2 additional M's for a total of 5 multiplications.

Polynomial Evaluation: Brute-force

This topic is discussed in section 30.1 (page 824) of CLRS but it is problem 2-3 on page 39.

20 Polynomial Evaluation. Suppose we are given a polynomial $f(x)$ such that

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$$

where the indeterminate is x and the coefficients $a_i \in R$. Depending on the circumstances a_i could belong to the set of integer numbers (denoted by Z) or the set of rational numbers (denoted by Q). Suppose we are given $c \in R$. Polynomial evaluation deals with finding

$$f(x)|_{x=c} = f(c).$$

In order to compute $f(c)$ the naive method for evaluation first finds

$$1, c, c^2, \ldots, c^n \text{ for a total of } (n-1) \text{ multiplications}$$

then computes all products of c^i with a_i

$$a_0, a_1 c, a_2 c^2, \ldots, a_n c^n \text{ for a total of } n \text{ multiplications}$$

and finally adds up the partial products to derive $f(c)$

$$f(c) = a_0 + a_1 c + a_2 c^2 + \ldots + a_n c^n \text{ for a total of } n \text{ additions}$$

```
   Naive(F[0..n],n,c)  /F[0..n] stores f(x)  ; we shall compute f(c)
1. result = F[0]; xn=1;
2. for(j=1;j<=n;j++) {
3.  xn *= c;
4.  result = result + xn * F[j];
5. }
6. return(result);
```

Total Operation Count. The total count of the Naive method is : $(2n-1)$ M's (for multiplications) and n A's (for additions). (Note that the first time line 3 is executed `xn` is equal to 1 and thus no multiplication is required.)

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

21 Horner's Rules There is a faster way to perform polynomial evaluation however using a method that is known as Horner's method/rule or Horner-Newton rule. According to this method

$$f(x) = ((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0.$$

i.e. $f(x) = a_3x^3 + a_2x^2 + a_1x + a_0 = ((a_3x + a_2)x + a_1)x + a_0$. This method requires n M's and n A's, i.e. it is faster than the naive method.

```
   Horner(F[0..n],n,c)
1. result = F[n];
2. for(j=n-1;j>=0;j--) {
3.  result = result * c + F[j];
4. }
5. return(result);
```

Total Operation Count. Line 3 uses overall n As and n Ms.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

In the next few pages we review some definitions from matrix theory. This roughly corresponds to the material covered in the textbook from page 724 (Chapter 28, Matrix Operations) through page 734.

A matrix is a rectangular array of numbers. For example, A is a 3×4 matrix, 3 indicating the number of rows and 4 the number of columns.

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \\ 1 & 3 & 9 & 27 \end{pmatrix}$$

In A, a_{ij} represents the entry in row i and column j. The transpose of A denoted by A^T is A with rows becoming columns and columns becoming rows.

$$A^T = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 4 & 9 \\ 1 & 8 & 27 \end{pmatrix}$$

A (column) vector x is an $n \times 1$ matrix. A row vector x^T is an $1 \times n$ matrix. A unit vector e_i is one whose i-th row is 1 and all other entries 0. A zero matrix is a matrix with all elements zeros. A square matrix is an $n \times n$ matrix.

$$x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, x^T = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}.$$

Special cases of matrices are (a) a diagonal (only a_{ii} elements are non-zero), (b) the identity matrix whose i-th column is e_i, $i = 1, \ldots, n$, (c) a tridiagonal where $a_{ij} = 0$ for $\mid i - j \mid > 1$, (d) an upper triangular ($a_{ij} = 0, i > j$ and a lower triangular ($a_{ij} = 0, i < j$, (e) a permutation matrix whose entries are zero except for a single one in every row and column.

A matrix A is symmetric if and only if $A = A^T$.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

22 Matrix Addition For two matrices A and B of dimension $n \times m$, matrix C of dimension also $n \times m$ is called the sum of A and B and we write $C = A + B$ if and only if $c_{ij} = a_{ij} + b_{ij}$.

22.1 Matrix Addition Operation Count. If A, B, C are $n \times m$ matrices, then addition requires nm additions and thus $O(nm)$ time.

23 Matrix Subtraction. Subtraction is defined similarly and requires as much time as well.

24 Matrix Multiplication. For two matrices A, B, matrix $C = A \times B$ is defined as long as the column size of A equals the row size of B, i.e. if A is an $n \times m$ and B an $m \times p$ matrix, then C is an $n \times p$ matrix such that $c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$.

24.1 Matrix Multiplication Operation Count. Matrix multiplication thus requires mnp multiplications and $np(m-1)$ additions.

25 Inverse of Matrix. The inverse of a $n \times n$ matrix A is a matrix A^{-1} such that $AA^{-1} = A^{-1}A = I$. If a matrix has an inverse it is called nonsingular or invertible.

26 Linear Independence. A set of n vectors are linearly independent if $\sum_i c_i x_i = 0$ implies $c_i = 0$ for all i.
Otherwise the n vectors are called dependent.
For example, the n vectors of an identity matrix are linear independent.

(c) Copyright A. Gerbessiotis.

Strassen's algorithm

Suppose we view matrix multiplication not the traditional way but devise for it a divide and conquer algorithm. Let us assume we are given an $n \times n$ matrix A, and let us assume without loss of generality that n is a power of two. Then we can view A as consisting of four $n/2 \times n/2$ matrices $A_{12}, A_{11}, A_{21}, A_{22}$. Then the product of two matrices A and B can be viewed as a blocked matrix product.

$$C = A \times B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

where, $C_{11} = A_{11}B_{11} + A_{12}B_{21}$, $C_{12} = A_{11}B_{12} + A_{12}B_{22}$, $C_{21} = A_{21}B_{11} + A_{22}B_{21}$, $C_{22} = A_{21}B_{12} + A_{22}B_{22}$. It is clear that for the calculation of C, 8 $n/2 \times n/2$ multiplications are required and 4 additions. Each addition requires $n^2/4$ additions for a total of n^2 operations. Then the total number of operations (additions and mutltiplications)

$$T(n) = 8T(n/2) + n^2$$

By the master method we derive that $T(n) = O(n^3)$. In the late 60s Strassen reformulated the calculations for C so that only 7 matrix multiplications (including its additions), and 7 additions and 8 subtractions are required. For Strassen's method we need

$$T(n) = 7T(n/2) + 15n^2/4$$

The solution to this recurrence is $T(n) = O(n^{\lg 7}) = O(n^{2.80})$. There has been much activity in reducing the exponent of matrix multiplication to something close to 2. The best exponent is an algorithm due to Coppersmith and Winograd where the exponent $\omega = 2.376$ giving a running time of $T(n) = O(n^{\omega})$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Original scheduling

Strassen's original method involved 18 additions thus giving a recurrence of

$$T(n) = 7T(n/2) + 18n^2/4$$

for matrix multiplication. For Strassen's method to work as claimed n must be a power of two and the base case is for $n = 2$. The number of operations required to multiply two 2×2 matrices is given either by the traditional method (known also as Cannon's method) of 12 operations (8 multiplications, 4 additions) or Strassen's method itself of 25 operations (7 multiplications, and 18 additions).

$$\begin{aligned}
m_1 &= (A_{12} - A_{22}) * (B_{21} + B_{22}) \\
m_2 &= (A_{11} + A_{22}) * (B_{11} + B_{22}) \\
m_3 &= (A_{11} - A_{21}) * (B_{11} + B_{12}) \\
m_4 &= (A_{11} + A_{12}) * B_{22} \\
m_5 &= A_{11} * (B_{12} - B_{22}) \\
m_6 &= A_{22} * (B_{21} - B_{11}) \\
m_7 &= (A_{21} + A_{22}) * B_{11} \\
C_{11} &= m_1 + m_2 - m_4 + m_6 \\
C_{21} &= m_6 + m_7 \\
C_{12} &= m_4 + m_5 \\
C_{22} &= m_2 - m_3 + m_5 - m_7.
\end{aligned}$$

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Dynamic programming, like the divide-and-conquer method, solves problems by dividing a problem into subproblems, solving the subproblems, and combining the solutions to subproblems. In d-and-c subproblems are usually independent of each other. DP however is used in cases where subproblems are not independent. A DP algorithm solves subproblems once, saves the solutions in a table and avoids recomputing the answer for a problem that has already been solved before. The development of a DP algorithm is broken into a sequence o steps.

(1) Characterize the structure of an optimal solution.

(2) Recursively define the value of an optimal solution.

(3) Compute the value of an optimal solution in a bottom-up fashion.

(4) Construct an optimal solution from computed information.

We are going to use dp techniques in solving the following problem.
Problem. Generalized matrix multiplication of q matrices.
Input. A matrix product $M = M_1 \times M_2 \times \ldots \times M_q$, where M_i is an $r_i \times r_{i+1}$ matrix.
Output. Compute M using the fewest number of operations.

Note that matrix multiplication is an associative operation i.e. $A \times (B \times C) = (A \times B) \times C$.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Method. Dynamic Programming allows for the solution of this problem in $O(q^3)$ time.

Description. Let M_{ij} denote the result of the consecutive product chain $M_i \times \ldots \times M_j$ and let $c[i,j]$ be the cost of OPTIMALLY computing such a product. Let $k[i,j]$ be such that the optimal solution of M_{ij} requires the computation of $M_{ik} \times M_{k+1j}$, where $k = k[i,j]$. Therefore in an optimal computation of M_{ij}, M_{ik} and M_{k+1j} are first computed and then M_{ij} as the product of the two partial results.

Then we obtain the following dynamic programming formula.

$$c[i,i] = 0 \text{ for all } 1 \leq i \leq q.$$

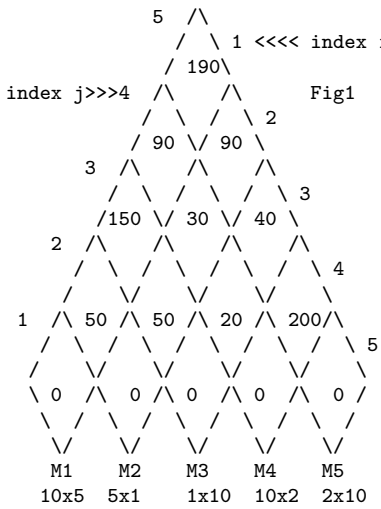
$$c[i,i+1] = r_i r_{i+1} r_{i+2} \quad \text{and} \quad k[i,i+1] = i, \text{ for all } 1 \leq i < q.$$

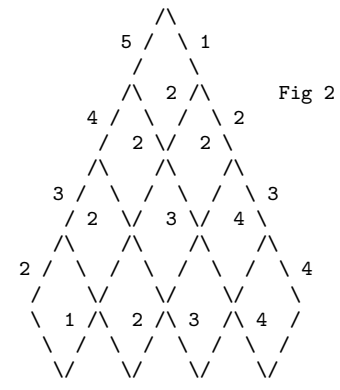
$$c[i,i+s] = \min_{i \leq k < i+s} \{c[i,k] + c[k+1,i+s] + r_i r_{k+1} r_{i+s+1}\} \text{ for all } 1 \leq i \leq q-s \text{ and } 1 < s < q \text{ with}$$

$$k[i,i+s] = \text{value of k that obtains the above minimum}$$

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

An example





1. STANDARD VIEW

2. ALTERNATIVE VIEW

i \ j	1	2	3	4	5
1	0	50	150	90	190
2		0	50	30	90
3			0	20	40
4				0	200<----------c[i,i+1]
5					0 <----------c[i,i]

An example (continued)

With reference to the left figure, $c[1,1] = c[2,2] = c[3,3] = c[4,4] = c[5,5] = 0$, and thus the first row of zeroes is filled. In order to fill the second line from the bottom $c[1,2] = 10 \times 5 \times 1 = 50$, and similarly $c[2,3] = 50$, $c[3,4] = 20$, and $c[4,5] = 200$. At the same time we record i in Figure 2 (in the corresponding box).

In order to fill the third line from the bottom we note that $c[1,3] = \min\{c[1,1]+c[2,3]+10\ 5\ 10, c[1,2]+c[3,3]+10\ 1\ 10\} = \min\{0+50+500, 50+0+100\} = 150$ and record $k = 2$ in Fig 2. Similarly $c[2,4] = \min\{c[2,2]+c[3,4]+5\ 1\ 2, c[2,3]+c[4,4]+5\ 10\ 2\} = \min\{0+20+10, 50+100\} = 30$ and record $k = 2$. $c[3,5] = \min\{c[3,3]+c[4,5]+1\ 10\ 10, c[3,4]+c[5,5]+1\ 2\ 10\} = \min\{0+200+100, 20+0+20\} = 40$ and record $k = 4$.

Then we fill the fourth row from the bottom. $c[1,4] = \min\{c[1,1]+c[2,4]+10\ 5\ 2, c[1,2]+c[3,4]+10\ 1\ 2, C[1,3]+c[4,4]+10\ 10\ 10\} = \min\{0+30+100, 50+20+20, 150+0+1000\} = 90$ and record $k = 2$ in Fig 2. $c[2,5] = \min\{c[2,2]+c[3,5]+5\ 1\ 10, c[2,3]+c[4,5]+5\ 10\ 10, C[2,4]+c[5,5]+5\ 2\ 10\} = \min\{0+40+50, 50+200+500, 30+0+100\} = 90$ and record $k = 2$ in Fig 2.

Finally, $c[1,5] = \min\{c[1,1]+c[2,5]+10\ 5\ 10, c[1,2]+c[3,5]+10\ 1\ 10, C[1,3]+c[4,5]+10\ 10\ 10, c[1,4]+c[5,5]+10\ 2\ 10\} = \min\{0+90+500, 50+40+100, 150+200+1000, 90+0+200\} = 190$ and record $k = 2$ in Fig 2.

Summarizing, in order to compute $M_1 \ldots M_5$ we need to perform 190 operations. We first compute $M_1 \times M_2$ and then $M_3 \times \times M_4 \times M_5$. The first product is computed directly in 50 multiplications by reading entry $c[1,2]$ from Figure 1. For M_{35} we obtain that we need to perform 40 operations since $c[3,5] = 40$ in Figure 1, and the way is by finding $M_3 \times M_4$ first and then multiplying the result with M_5 as derived from the $[3,5]$ entry of Figure 2. The first product $M_3 \times M_4$ requires 20 operations, and the second the remaining twenty multiplications. Therefore $M_{15} = (M_1 \times M_2) \times ((M_3 \times M_4) \times M_5)$.

(c) Copyright A. Gerbessiotis.