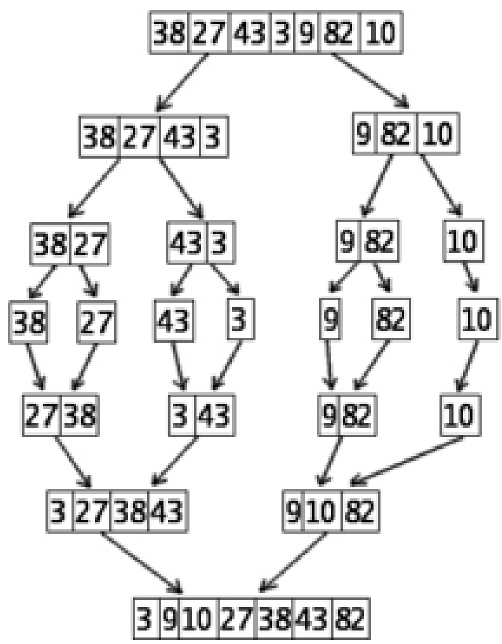
Computer Science 577 Notes
Introduction to Algorithms

Mendel C. Mayr

February 4, 2015



Contents

1 Recurrence Relations

1.1 Recursive Analysis of Insertion and Merge Sort

Insertion sort: let $M(n)$ be the comparisons required to sort a list of size n
Analysis: note that $M(1) = 0$ and $M(n) = M(n-1) + n$ for $n > 1$

(i) $M(n) = M(n-1) + n$

(ii) $M(n) = M(n-2) + n + (n-1)$ ...

(iii) $M(n) = M(n-k) + n + (n-1) + \ ... \ + (n-k+1)$

(iv) Let $k = n-1$, $M(n) = M(1) + n(n-n+1) + \sum_{i=1}^{n-1} i$

(v) $M(n) = 0 + n + (n-1)(n-2)/2 \approx n^2/2$

Merge sort: let $M(n)$ be the comparison required to sort a list of size n

Analysis: for simplcity, consider only n such that $n = 2^a$ for some integer a
Note that $M(1) = 0$ and $M(n) = 2M(n/2) + n$ for $n > 1$

(i) $M(n) = 2M(n/2) + n$

(ii) $M(n) = 2M(n/4) + n + (n/2)$

(iii) $M(n) = 2M(n/2^k) + n + (n/2) + \ .... \ + (n/2^{k-1})$

(iv) Let $k = a$, $2M(1) + \sum_{i=1}^{k-1} n/2^i$ (unclarified point)

(v) Mergesort is $O(n \log n)$

1.2 Recursive Linear Selection

Recursive linear selection algorithm: given $x_1, x_2, ..., x_n$ distinct keys, find x_k (i.e. the kth smallest element) without using sorting
Note: the rank of an element (i.e. the number of keys greater than it) can be found in linear time
Linear selection algorithm is as folows:

(i) Remove keys of known rank, to make $n = 5(mod 10)$

(ii) Divide elements into groups of 5, denoted $S[i]$ for i from 1 to $n/5$

(iii) Recursively find the median of each group, denoted $x[i]$

(iv) Let M^* be the median of the set $x[i]$ for i from 1 to $n/5$

(v) Divide keys into groups of keys less than (call this L), equal to, or greater than (call this R) M^*

(vi) Recursiveley process one of L or R

Analysis: Note that steps 1, 2, and 5 are $O(n)$, so the number of computations for this algorithm, $T(n) = T(n/5) + T(7n/10) + O(n)$
Guessing and proving: supposed that $T(n) = O(n)$, which can be proven via strong induction, i.e. $T(n) < An$ for some constant A for all n
Recall that strong induction relies on proving two claims:

(i) The statement holds for all $n \geq 1$, $\forall n \leq n_0$ (base case)

(ii) If the statmenet holds for all $i < n$, it holds for n

Proof of second part of strong inductive proof:

(i) Suppose that $T(n) = O(n)$ for $i < n$

(ii) We seek and A such that $A(n/5) + A(7n/10) + cn \leq An$

(iii) Thus, $A \geq 10c$ is sufficient for this part

Proof of first part of strong inductive proof: need A such that $n(n-1)/2 \leq An$ for $1 \leq n \leq 10$
So $A \geq 9/10$ is sufficient for this part

Conclusion: $A = max\{9/2, 10c\}$ will suffice to show that $T(n) = O(n)$

1.3 Recursive Quadratic Closest-Pair

Recursive quadratic algorithm: find closest pair of points

i (Supposing that $n = 2^k$) into 2 equal groups, denoted L and R

ii Recurisvely find the closest pair in L and R

iii Report closest pair form testing elements of L against elements of R

iv Report best pair out of those from steps (ii) and (iii)

Analysis: $T(n) = 2(T/n) + O(n^2)$ for $n = 2^k \geq 4$, $T(2) = 1$
The $O(n^2)$ in the recursive case comes from step (iii)

Consider the recursion tree, which is full binary tree: at the first level, the problem size is $n, n/2, n/4, ...$ at the first, second, third, etc. levels. Thus, the number of computations required is n^2 at the first level, $2(n/2)^2 = n^2/2$ at the second level, $2(n/4)^2 = n^2/4$ at the third, etc.

Thus, the maximum number of computations is $\sum_{k=1}^{\infty} n^2/2^k = 2n^2$, thus $T(n) = O(n^2)$

1.4 Divide and Conquer Recurrences and Master Theorem

Master theorem: if $T(n) = aT(n/b) + O(n^d)$ for some constants $a > 0, b > 1$, and $d \geq 0$, then:

(i) $T(n) = O(n^d)$ if $d > \log_b a$

(ii) $T(n) = O(n^d \log n)$ if $d = \log_b a$

(iii) $T(n) = O(n^{\log_b a})$ if $d < \log_b a$

Proof: consider the recursion tree for such a problem
Notice that a is the branching factor of the problem. At the ith level (starting at index 0), there are a^i subproblems of size n/b^i. which means the computation that must be done at that level is $a^i O((n/b^i)^d)$
The number of levels in the recusion tree is $k = \log_b n$
As such: $T(n) = \sum_{i=0}^{k} a_i O((n/b^i)^d) = O(n^d) \sum_{i=0}^{k} (a/b^d)^i$. Now consider the cases

(i) If $d > \log_b a$, then $a/b^d < 1$
$\sum_{i=0}^{\infty} (a/b^d)^i < \infty$ (i.e. series converges)
$\sum_{i=0}^{\infty} (a/b^d)^i = O(1)$, so $T(n) = O(n^d)$

(ii) If $d = \log_b a$, then $a/b^d = 1$
$\sum_{i=0}^{k}(a/b^d)^i = \sum_{i=0}^{k} 1 = k+1$, since $k = \log_b n = \log n / \log b = O(\log n)$
Therefore, $T(n) = O(n^d \log n)$

(iii) If $d < \log_b a$, then $a/b^d > 1$
$\sum_{i=0}^{k}(a/b^d)^i = O((a/b^d)^k)$, so $T(n) = O(n^d)O(a^k)/b^{dk}$
Since $k = \log_b n, n = b^k$, $T(n) = O(n^d)O(a^k)/n^d = O(a^k)$
$a^k = a^{\log_b n} = n^{\log_b a}$, so $T(n) = O(n^{\log_b a})$

1.5 Asymptotics

Notation for asymptotics:

(i) f and g are real value functions on $x \geq 0$. $f(x)$, $g(x) \geq 0$ for sufficiently large x
Sufficently large: $\exists x_0 > 0$ such that $f(x) \geq 0$ when $x \geq x_0$

(ii) $f = O(g)$ means that for some $c > 0$ and $x_0 > 0$, $f(x) \leq cg(x)$ for al $x \geq x_0$

(iii) $f = \Omega(g)$ means that for some $c > 0$ and $x_0 \geq 0$, $f(x) \geq xg(x)$ for all $x \geq x_0$
This is equivalent to saying that $g = O(f)$

(iv) $f = \Theta(g)$ means that $f = O(g)$ and $f = \Theta(g)$

Demonstrating that $f = O(g)$ can be done algebraically, or via L'Hopital's rule

Additional definitions:

(i) $f = o(g)$ means $\lim_{x\to\infty} f(x)/g(x) = 0$

(ii) $f \sim g$ means $\lim_{x\to\infty} f(x)/g(x) = 1$

Polynomial growth: f is polynomially bounded if $f(x) = O(x^k)$ for some $k > 0$, (efficiently computable)
Exponential growth: f is exponential growth if $f(x) = O(\alpha^x)$ for some $\alpha > 1$

1.6 Arithmetic Algorithms

Addition: elementary (i.e. sum and carry bits), adding two n-bit numbers has $O(n)$ complexity
Subtraction: inverse of addition, similarly requires $O(n)$ times
Multiplication: elementary algorithm requires $O(n)$ times
There exists an $O(n^a)$ algorithm for multiplication, where $a < 2$:
For multiplying n-bit numbers, where $n = 2^k$:

(i) $x = 2^{n/2}x_1 + x_0$, $y = 2^{n/2}y_1 + y_0$

(ii) $xy = 2^n x_1 y_1 + 2^{n/2}(x_1 y_0 + x_0 y_1) + x_0 y_0$

(iii) Let $a = x_1 y_1$, $c = x_0 y_0$, $d = (x_1 + x_2)(y_1 + y_2)$

(iv) Let $b = x_1 y_0 + x_o y_1$, note that $b = D - A - C$

Analysis: let $k(n)$ be the complexity of this algorithm
$k(n) = 3k(n/2) + O(n)$ when $n > 1$, $K(n) = O(1)$ when $n = 1$
By the master theorem: $k(n) = O(n^{\log_2 3}) \approx O(n^{1.59})$

Using Newton interation, division reducible to multiplication: similar complexity
Open question: does there exist $O(n)$ algorithm for multiplication

Matrix multiplication: let $A = (a_{i,j}), B = (b_{i,j})$ for $1 \leq i, j \leq n$
Then $C = (c_{i,j})$, where $c_{i,j} = \sum_{k=1}^{n} a_{i,k} n_{k,j}$
Recursive algorithm for matrix multiplication: subblocks of matricies can be multipled as follows

$$\text{Since } X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ and } Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}, \text{ then } XY = \begin{bmatrix} AE + BG & CE + DG \\ AF + BH & CF + DH \end{bmatrix}$$

The 8 products required tocan be found recursively, resulting in $T(n) = 8T(n/2) + O(n^2) = O(n^3)$
A matrix decomposition can be used to make a faster algorithm, requiring only 7 products:

$$\begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

The products $P_1, P_2, ..., P_7$ are defined respectively as:
$A(F - H), (A + B)H, (C + D)E, D(G - E), (A + D)(E + H), (B - D)(G + H), (A - C)(E + F)$
The complexity, by the master theorem, is: $T(n) = 7T(n/2) + O(n^2) = O(n^{\log_2 7}) \approx O(n^{2.81})$

A Review of Basic Mathematical Concepts

A.1 Properties of Logarithms

Change of base: $\log_a x = \log_b x / \log_b a$
Basic properties:

(i) $\log_a(uv) = \log_a u + \log_a v$

(ii) $\log_a(u/v) = \log_a u - \log_a v$

(iii) $\log_a u^n = n \log_a u$