به نام خدا

دانشکده علوم ریاضی

آنالیز الگوریتم ها
نیم‌سال دوم ۰۱-۰۲

تمرین ۱ - امیرحسین محمودی_ ۹۸۱۰۸۷۷۹

1

1.1

we show these:
h=O(g), g=O(f), f=$\Theta(t), t = O(r), r = O(q), q = O(p), p = O(s)$

1.1.1

assuming c=1:

$$log^4 n \leq c(\sqrt{n}) = \sqrt{n} \rightarrow$$

say $n_0 = 2^{100}$

$$log^4 2^{100} = 100^4 log^4 2 = 10^8 log^4 2 \leq \sqrt{2^{100}} = 2^{50}$$

we now that:

$$\lim_{n\to\infty} \frac{\log^4 n}{n} = 0$$

1.1.2

assume $c = 1$:

$$\sqrt{n} \leq c(n + 1000) = n + 1000 \rightarrow$$
$$0 \leq n - \sqrt{n} + 1000 \rightarrow n_0 \geq 0$$
$$\rightarrow \sqrt{n} = O(n + 1000)$$

assume we have an c and n_0 that :

$$n + 1000 \leq c(\sqrt{n}) \rightarrow$$

$$n - c(\sqrt{n}) + 1000 \leq 0;$$

if we chose n:

$$n \geq max(n_0, (\frac{c + \sqrt{c^2 - 4000}}{2})^2)$$

so it's a contradiction.

1.1.3

taking c=2 and $n_0 = 1000$ we have:

$$n + 1000 < 2n < 2t(n) \rightarrow f = O(t)$$

On the other side we have:

$$t(n) = 7t(n) + O(n)$$

using master theorem we have t=O(f) so $t(n) = \theta(f(n))$.

1.1.4

using master theorem we have $r(n) = \Theta(n^{\log_3 28})$.so there is a c and n_0 that $n^{\log_3 28} < c_0 r(n)$.we Also now that $\lim_{n\to\infty} \frac{n}{n^{\log_3 28}} = 0$ so t=O(r)

1.1.5

$$\sum_{x=0}^{n} x^3 < \int_1^{n+1} x^3 dx = \frac{(n+1)^4 + 1}{4} < 4n^4$$

so $q = \Theta(n^4) = O(p(n))$

1.1.6

for $n > 4$ we have:

$$n! > 6n \rightarrow n(n!)! > (6n)! > n^6$$

we also have $l \lim_{n\to\infty} \frac{q}{p} = 0$ so q=O(p).

1.1.7

we know $n^n > n!$ so we have :

$$\lim_{n\to\infty} \frac{((n!)!)}{n^n} < \frac{(n!)!}{\frac{(n^n)!}{(n^n - n!)!}} < \lim_{n\to\infty} \frac{n!}{n^n}$$

SO p=O(s).

1.2

1.2.1

if $(\alpha+\beta)=1$ then we have an incomplete tree.assume $\alpha<\beta$.in each level total cost of all nodes(assume they all exists) equals n.the shortest path to any leaf comes from following αs (length of $\Theta(\log_\alpha)$) and longest path comes by following βs(length of $\Theta(\log_\beta)$).
Now we consider two balanced trees:
τ_α is a subgraph of τ(our tree) and its height is $\log_\alpha$.
τ_β is a supergraph of $\tau$and its height is $\log_\beta$.
Assuming the total cost of each level is n in these trees we can calculate total cost of these trees:

$$S_\alpha(n)=n\times\Theta(\log_\alpha)=\Theta(_\alpha)$$

$$S_\beta(n)=n\times\Theta(\log_\beta)=\Theta(_\beta)$$

note that $S_\alpha\le T(n)\le S_\beta$.so we have:

$$T(n)=\Theta(n\log n)$$

1.2.2

Top $T(n)$ takes cn Times.Then calls to $T(\alpha)$ and $T(\beta)$, which take a total of $cn(\alpha+\beta)$ and so on. if $(\alpha+\beta)<1$ summing infinitely:

$$\sum_{i=0}^{\infty}cn(\alpha+\beta)^i=\frac{cn}{1-(\alpha+\beta)}=O(n)$$

1.2.3

draw the tree.In each level we have total cost $c\cdot a^i(\frac{n}{b^i}^d(\log\frac{n}{b^i}))$.the height of the tree is $\log_b n$.the total cost is:

$$T(n)=\sum_{i=0}^{\log_b n}c\cdot a^i(\frac{n}{b^i}^d(\log\frac{n}{b^i}))=cn^d\log n\sum_{i=0}^{\log_b n}(\frac{a}{b^d})^i-cn^d\log b\sum_{i=0}^{\log_b n}i(\frac{a}{b^d})^i$$

if $\frac{a}{b^d}<1$, then $\sum_{i=0}^{\log_b n}(\frac{a}{b^d})^i<\frac{1}{1-(\frac{a}{b^d})}$ and $\sum_{i=0}^{\log_b n}i(\frac{a}{b^d})^i<\frac{(\frac{a}{b^d})}{(1-(\frac{a}{b^d}))^2}$. so for c_1 and c_2 we have::

$$T(n)\le c_1n^d\log n-c_2n^d\log n=O(n^d\log n)$$

if $\frac{a}{b^d}=1$:

$$T(n)=cn^d\log n\sum_{i=0}^{\log_b n}(1)^i-cn^d\log b\sum_{i=0}^{\log_b n}i=n^d\log n(\log_b n+1)-n^d(\frac{\log_b n+1}{2})(\log n)=O(n^d\log^2 n)$$

if $\frac{a}{b^d} > 1$:

$$T(n) = cn^d \log n \sum_{i=0}^{\log_b n} (\frac{a}{b^d})^i - cn^d \log b \sum_{i=0}^{\log_b n} i(\frac{a}{b^d})^i = n^d \cdot (\log b) \sum_{i=0}^{\log_b n} (\frac{a}{b^d})^i (\log_b n - i) \leq n^d((\frac{a}{b^d})^{\log_b n}) = O(n^{\log_b a})$$

2

2.1

there are three loops with constant complexity.each loop runs n times so the time complexity of the Floyd-Warshall algorithm is $O(n^3)$.

2.2

using potential method to analyze n operation we have:

$$\phi(T) = 2 \cdot T.num - T.size$$

This function is 0 immediately after Table expansion because T.size=2 T.num .Also it is T.num just before expansion.its alwaus nonnegative cause the Table is Always at least half full.so it is our desired function.
The initial value of the function and variables are 0.
if the i-th operation contains an expansion we have:

$$\hat{c_i} = c_i + \Phi_i - \phi_{i-1}$$

$$= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1})$$

$$= num_i + (2 \cdot num_i - 2 \cdot (num_i - 1)) - (2(num_i - 1) - (num_i - 1))$$

$$= 3$$

if the i-th operation does not contain an expansion we have:

$$\hat{c_i} = c_i + \Phi_i - \phi_{i-1}$$

$$= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1})$$

$$= 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i)$$

$$= 3$$

So the amortized cost of a single operation is 3 or $O(1)$.

2.3

We have two loops.For loop runs m-1 times.we show that the inner loop is also runs at most m-1 times:
k is always positive.
k could only increase by line which runs at most m-1 times.
The only way to decrease k is line 7($\pi[q] < q$ and $k < q$so the while loop should decrease k).so

this line could be run at most m-1 times (else k would be not positive.)
So this function runs in $\Theta(m)$

3 Gogol Sort

3.1

Algorithm 1: Gogol-Sort(X)

```
if X.size=1 then
    Return X
end
else
    for ( i=0 to ⌊X.size/2⌋ ) {
        l=X[i]
        r=X[ ⌊X.size/2⌋ + i]
        if l>r then
            Swap(l,r)
        end
    }
    Gogol-Sort(X[0,...,⌊X.size/2⌋ − 1])
    Gogol-Sort(X[⌊X.size/2⌋, ..., X.size − 1])
end
```

Proof by induction:
For $X.size = 2$ it works. assume it works for $X.size = 2^n$.now we prove it for $X.size = 2^{n+1}$: if we run the algorithm on the array, it splits the array into A and B.then it starts to compare elements each by each.if $B_i > A_i$ then they swap.we show they will be Gogoli or they can be converted to Gogoli. then for an given index p, we all elements after them swaps and all elements before them remains.So we get two Gogol array and they are 2^n long so we can sort them concatenate them and its done.
now for time complexity we have:

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

by master theorem it is O(nlog n).

3.2

we add mode to gogol-sort which indicate to sort ascending or descending.now we define some kind of merge sort: now for time complexity we have:

$$T(n) = 2T(\frac{n}{2}) + O(n \log n)$$

by master theorem it is O(nlog^2 n).

Algorithm 2: Merge2(X,mode)

if *X.size=1* **then**
  Return X
end
else
  Gogol-Sort2(X[0,...,$\lfloor \frac{X.size}{2} \rfloor - 1], 1$)
  Gogol-Sort2(X[$\lfloor \frac{X.size}{2} \rfloor, ..., X.size - 1, 0$])
end

To prove it works we act like last part and use proof by induction on merge function.for proving for and array with length 2^{n+1} we sort each part with one mode and the result is a gogoli array which we know we can sort.

3.3

this algorithm needs no extra memory and it can be paralalized .

4

We just need some kind of quick sort.we first chose a pivot from bolts and partition nuts by it.and then we partition the bolts with the pivot.then we recursively use match on each partition.
Time complexity is just like quick sort with a different constant.
Partition algorithm is $O(n)$(cause the loop runs n times) and the we use two partition in each Match.we call Match *logn* times.so the Match Algorithm is of $O(n \log n)$.

Algorithm 3: Partition(A,p,r,pivot)

$i = p$;
$j = p$;
while $j < pivot$ **do**
  if $A[j] \leq pivot$ **then**
    exchange $A[i]$ with $A[j]$;
    $i = i + 1$;
  end
  else if $A[j] = pivot$ **then**
    exchange $A[j]$ with $A[r]$;
    $j = j - 1$;
  end
  $j = j + 1$;
end
exchange $A[i]$ with $A[r]$;
Return i

Algorithm 4: Match(bolts,nuts,p,r)

```
if p<r then
    pivot = Partition(nuts,p,r,bolts[r]);
    Partition(bolts,p,r,nuts[pivot]);
    Match(nuts,bolts,p,pivot-1);
    Match(nuts,bolts,pivot+1,r);
end
```

5

assume we have n points. and we draw a vertical line which split the points into two equal parts with n/2 points.we make each part stable recursively.now we want to make two stable parts stable to each other.
We show that for every point if we add its projection on the seperating line then this point will be stable to all points on the other part: for each point $p = (x, y)$ and its projection $p' = (x', y')$ in right part we have for all points $q_i = (x_i, y_i)$in left part:

$$x_i < x' < x$$

$$y_i < y' \ or \ y + i \geq y' = y$$

so its stable to all points on other part.also lines added on the separating line are stable to other points.also new points are stable to each other cause they have same x.
So we can do this recursively.we need to add O(n log n) points(draw the tree and use height and levels).
for time complexity we have:

$$T(n) = 2T(n/2) + O(n)$$

applying master theorem we have :$T(n) = O(nlogn)$