NHẬP MÔN PHÂN TÍCH ĐỘ PHỨC TẠP THUẬT TOÁN

Project

Phân tích năm thuật toán yêu thích

1. Quick sort
2. Linear deterministic quick find (một thuật toán của Blum, Floyd, Pratt, Rivest, and Tarjan)
3. Thuật toán Tortoise and Hare
4. Fast fourier transform (phiên bản của Cooley và Tukey)
5. Subset sum

1. Quick sort

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

Hàm quicksort là một hàm đệ quy gồm 2 bước, bước một gọi hàm partition để phân chia mảng A thành 2 mảng con có độ dài p và $n - p - 1$ (với p là index của biến pivot), bước 2 gọi đệ quy quicksort để sort tiếp 2 mảng con này. Độ dài của mảng trong một mỗi lần gọi quick sort bằng $n = hi - low + 1$.

Dễ thấy hàm partition có độ phức tạp $\theta(n)$. Bởi biến chạy j chắc chắn chạy qua mỗi phần tử đúng một lần duy nhất và các phép toán trong vòng for là một hằng số nhỏ hơn 5.

Suy ra nếu gọi T(n) là độ phức tạp của thuật toán khi sort n phần tử thì

$$T(n) = T(p) + T(n - p - 1) + \theta(n)$$

Từ công thức truy hồi trên cho thấy độ phức tạp của quick sort phụ thuộc rất nhiều vào vị trí của biến pivot.

Worst case:

Trường hợp xấu nhất xảy ra khi ta luôn chọn được biến pivot là phần tử lớn nhất hoặc nhỏ nhất, lúc đó việc partition sẽ rất mất cân bằng giữa $T(p)$ và $T(n - p - 1)$

$$T(n) = T(n - 1) + \theta(n)$$

Dùng master theorem ta được $T(n) = \theta(n^2)$

Best case:

Trường hợp lý tưởng nhất của quick sort xảy ra ta luôn phân hoạch được thành 2 mảng con rất đều nhau, tức chọn được pivot nằm chính giữa mảng

$$T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$$

Dùng master theorem ta được $T(n) = \theta(n \log_2 n)$

Average case:

Khi dữ liệu đầu vào hoàn toàn ngẫu nhiên hoặc ta có thể chọn một cách ngẫu nhiên giá trị cho biến pivot đưa tới trường hợp trung bình cho thuật toán. Một cách tổng quát, biến pivot của chúng ta được chọn ngẫu nhiên trong n phần tử mảng với xác suất như nhau, suy ra xác suất để chọn được pivot nằm ở vị trí p bất kỳ là $\frac{1}{n}$

Từ đó ta có kỳ vọng cho $T(n)$

$$T(n) = \sum_{p=0}^{n-1} \frac{1}{n} T(p) + \sum_{p=0}^{n-1} \frac{1}{n} T(n - p - 1) + \theta(n)$$

Vì $\sum_{p=0}^{n-1} \frac{1}{n} T(p)$ và $\sum_{p=0}^{n-1} \frac{1}{n} T(n - p - 1)$ thực chất là 2 tổng giống nhau với thứ tự các phần tử ngược nhau nên ta có thể viết lại $T(n)$ như sau

$$T(n) = 2 * \sum_{p=0}^{n-1} \frac{1}{n} T(p) + \theta(n) = \frac{2}{n} \sum_{p=0}^{n-1} T(p) + \theta(n)$$

Vì xác suất pivot bằng một phần tử nào đó trong mảng là như nhau nên kỳ vọng của pivot sẽ bằng trung bình cộng các phần tử trong A, từ đây ta đưa ra suy đoán $T(n)$ trong trường hợp trung bình sẽ không khác quá xa với $T(n)$ trong trường hợp tốt nhất. Tức ta dự đoán $T(n) = O(n \log_2 n)$ rồi dùng quy nạp để chứng minh.

Chứng minh tồn tại một hằng số b sao cho $T(n) \leq bn \log_2 n$

Base case: n = 0 và n = 1. Khi đó $lo \geq hi$ nên thuật toán dừng chỉ sau một phép so sánh.

Giả xử $T(p) \leq bp \log_2 p$ với mọi $p < n$, ta có

$$T(n) = \frac{2}{n}\sum_{p=0}^{n-1} T(p) + cn = \frac{2}{n} * 2 + \frac{2}{n}\sum_{p=2}^{n-1} T(p) + cn$$

$$T(n) \leq \frac{4}{n} + \frac{2}{n}\sum_{p=2}^{n-1} bp \log_2 p + cn = \frac{4}{n} + \frac{2b}{n}\sum_{p=2}^{n-1} p \log_2 p + cn$$

Ta có $p \log_2 p \leq \int_p^{p+1} x \log_2 x \, dx$ suy ra

$$T(n) \leq \frac{4}{n} + \frac{2b}{n}\int_2^n x \log_2 x \, dx + cn$$

Với $\int_2^n x \log_2 x \, dx = \frac{1}{\ln(2)}\left(\frac{n^2 \ln(n)}{2} - \frac{n^2}{4} - \frac{2^2 \ln(2)}{2} + \frac{2^2}{4}\right) = \frac{n^2 \log_2 n}{2} - \frac{n^2}{2\ln(2)} + C$ thay vào ta được

$$T(n) \leq \frac{4}{n} + \frac{2b}{n}\left(\frac{n^2 \log_2 n}{2} - \frac{n^2}{2\ln(2)} + C\right) + cn$$

$$T(n) \leq \frac{4}{n} + 2b\left(\frac{n \log_2 n}{2} - \frac{n}{2\ln(2)} + \frac{C}{n}\right) + cn$$

$$T(n) \leq \frac{4}{n} + bn \log_2 n - \frac{bn}{\ln(2)} + \frac{bC}{n} + cn = bn \log_2 n - n\left(\frac{b}{\ln(2)} - c\right) + \frac{4 + bC}{n}$$

$T(n) \leq bn \log_2 n$ với b đủ lớn (cụ thể b > c)

Suy ra $T(n) = O(n \log_2 n)$ mà ta biết rằng độ phức tạp trong trường hợp tốt nhất là $\theta(n \log_2 n)$ nên trong trường hợp trung bình một lần nữa $T(n) = \theta(n \log_2 n)$

2. Deterministic linear Quick find by Blum, Floyd, Pratt, Rivest, and Tarjan

Mục tiêu thuật toán là tìm ra phần tử lớn thứ k của một mảng n phần tử chưa sắp xếp trong thời gian tuyến tính

```
LinearSelection(A[1, 2, ..., n], k):
  if n ≤ 25
    use brute force
  else
    m ← ⌈n/5⌉
    for i ← 1 to m do
      M[i] ← MedianOfFive(A[5i − 4, ..., 5i])
    A[p] ← LinearSelection(M[1, 2, ..., m], ⌊m/2⌋)
    r ← Partition(A[1, 2, ..., p], ⌊m/2⌋)
    if k < r
      return LinearSelection(A[1, 2, ..., r − 1], k)
    else if k > r
      return LinearSelection(A[r + 1, 2, ..., n], k − r)
    else
      return A[r]
```

Thuật toán dùng cùng cách phân hoạch như quick sort, đầu tiên chọn ra phần tử pivot, rồi chia mảng ban đầu thành 2 mảng con, từ đây ta biết được vị trí sau khi sắp xếp của pivot, gọi vị trí này là r, nếu phần tử cần tìm là $k < r$ thì ta lại dùng đệ quy để tìm k trong $r - 1$ phần tử của mảng con đầu tiên, còn nếu $k > r$ thì ta tìm phần tử có vị trí $k - r$ trong $n - r$ phần tử của mảng con thứ 2. Còn nếu may mắn $k = r$ ta có thể return kết quả ngay lập tức.

Điểm khác biệt lớn nhất dẫn tới đột phá của thuật toán nằm ở cách chọn pivot. Việc chọn pivot chỉ áp dụng khi $n \geq 25$, khi đó mảng ban đầu được chia thành $\left\lceil \frac{n}{5} \right\rceil$ nhóm, mỗi nhóm có tối đa 5 phần tử, sau đó thuật toán đi tính median của mỗi nhóm 5 phần tử này lập thành mảng M. Rồi lại dùng chính hàm LinearSelection để tìm median cho mảng M.

Nếu $n < 25$, tức nhỏ hơn một hằng số, ta coi như có thể tìm ra r trong thời gian hằng số C (có thể dùng chính quick find với cách chọn pivot bình thường hoặc một thuật toán bất kỳ khác).

Nếu gọi $T(n)$ là độ phức tạp thời gian của thuật toán khi chạy trên mảng n phần tử, suy ra việc tìm median trên mảng M tốn thời gian $T\left(\frac{n}{5}\right)$.

Sau đó phần phân hoạch diễn ra, cũng như quick sort, phần phân hoạch chắc chắn chạy trong thời gian $\theta(n)$ bởi phải duyệt qua tất cả các phần tử đúng một lần.

Sau đó thuật toán gọi đệ quy để giải bài toán với số phần tử là $r - 1$ hoặc $n - r$, nhưng với cách chọn pivot như trên, ta sẽ chứng minh phần phân hoạch đã chia mảng A thành hai phần đều có số phần tử chắc chắn lớn hơn hoặc bằng $\frac{3n}{10}$

Gọi pivot A[r] là p. Vì p là median của $\left\lceil \frac{n}{5} \right\rceil$ phần tử trong M (cũng nằm trong A) nên có ít nhất $\left\lceil \frac{n}{10} \right\rceil$ phần tử nhỏ hơn p, hơn nữa các phần tử nhỏ hơn p này đều là median của các nhóm 5 phần tử, nên ta lại có thêm ít nhất $2\left\lceil \frac{n}{10} \right\rceil \leq p$ nữa. Tổng lại ta có ít nhất $\left\lceil \frac{3n}{10} \right\rceil$ phần tử nhỏ hơn hoặc bằng p. Tương tự ta cũng có $\left\lceil \frac{3n}{10} \right\rceil$ phần tử lớn hơn hoặc bằng p nên việc phân hoạch tệ nhất là chia thành 2 nhóm $\frac{3n}{10}$ và $\frac{7n}{10}$.

Từ đó trong trường hợp tệ nhất ta có công thức truy hồi của $T(n)$ như sau

$$T(n) = \theta(n) + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right)$$

Vì $\frac{n}{5} + \frac{7n}{10} < n$ nên sau mỗi lần đệ quy kích thước bài toán được giảm đi tỉ lệ tuyến tính với n cộng thêm một phần cn các phép toán tính thêm, ta ước đoán rằng thuật toán chạy trong thời gian $O(n)$ rồi dùng quy nạp để chứng minh.

Chứng minh $T(n) \leq bn$

Base case: Với $n < 25$ ta có $T(n) = c_1$

Giả xử $T(n_0) \leq bn_0$ với mọi $n_0 < n$, ta được

$$T(n) \leq cn + b\frac{n}{5} + b\frac{7n}{10} = \frac{(10c + 9b)n}{10}$$

Suy ra

$$T(n) \leq bn$$

với b đủ lớn, $b \geq 10c$ chẳng hạn

Suy ra $T(n) = O(n)$, mà trong trường hợp tốt nhất (tìm thấy A[k] trong lần phân hoạch đầu tiên chẳng hạn, hoặc việc phân hoạch luôn cân bằng) thì thuật toán chạy trong thời gian $T(n) = \theta(n)$ nên ta có thể kết luận thuật toán có độ phức tạp $T(n) = \theta(n)$

3. Tortoise and Hare (một thuật toán của Floyd check xem danh sách liên kết có vòng hay không)

Bình thường khi check một danh sách liên kết có vòng hay không, ta thường duyệt qua mỗi phần tử trong danh sách rồi kiểm tra xem phần tử này đã được duyệt qua hay chưa. Cách giải này có chi phí thời gian $\theta(n^2)$ và bộ nhớ $\theta(n)$. Nhưng thuật toán của Floyd giải bài toán này chỉ trong thời gian $\theta(n)$ và bộ nhớ $\theta(1)$

```
CycleDiscovery(L):
    ptr1 ← head(L)
    ptr2 ← head(L)
    while ptr2 ≠ Null and next(ptr2) ≠ Null
        ptr1 ← next(ptr1)
        ptr2 ← next(next(ptr2))
        if ptr1 = ptr2
            return Yes
    return No
```

Ý tưởng của Floyd là dùng 2 con trỏ cùng duyệt danh sách cần kiểm tra L, con trỏ ptr1 (rùa) duyệt danh sách với tốc độ 1, con trỏ ptr2 (thỏ) duyệt danh sách với tốc độ 2.

Việc duyệt qua mỗi phần tử chạy trong thời gian $\theta(1)$ bao gồm các bước trong vòng lặp while (gán ptr và so sánh).

Trong trường hợp L không có vòng, ptr2 hoặc next(ptr2) sẽ bằng Null nên thuật toán dừng lại, chạy trong thời gian $\theta(n)$ vì ptr1 và ptr2 đều đã duyệt được $\frac{n}{2}$ phần tử. Đây cũng là best case của thuật toán.

Trong trường hợp L có vòng, vì ptr2 chạy nhanh gấp đôi ptr1 nên chắc chắn sẽ quay về ngay sau ptr1. ptr2 nhảy cóc 2 bước một nên có thể sẽ nhảy qua ptr1 mà không so sánh, nhưng điều đó chắc chắn không thể xảy ra vì muốn ptr2 nhảy qua ptr1 một bước thì trước đó ptr2 đã phải bằng ptr1 rồi, tiếp theo đây ta sẽ chứng minh ptr2 sau vài vòng loop chắc chắn sẽ gặp ptr1.

Sau khi ptr1 duyệt vào loop (nơi L có vòng), vì ptr2 chạy nhanh gấp đôi nên sẽ nhanh chóng đuổi kịp đằng sau ptr1 sau nhiều nhất là m bước duyệt. (m là số phần tử của loop, trong trường hợp xấu nhất khi ptr1 bước vào loop, ptr2 nằm ngay trước ptr1 nên ptr2 phải duyệt hết loop 1 lần mới có thể đuổi kịp ptr1, gọi x là số phần tử duyệt qua sau mỗi bước, ta có $2x - m = x =>$ ptr2 phải duyệt qua nhiều nhất là m phần tử để có thể đuổi ptr1 kịp, khi này ptr1 đã đi đến cuối loop).

Sau khi tiếp cận ptr1, ptr2 sẽ nằm ngay sau ptr1 2 hoặc 1 phần tử. Nếu ptr2 nằm ngay sau ptr1 chỉ một phần tử thì chỉ cần một bước duyệt tiếp theo ptr2 sẽ bằng ptr1. Còn nếu ptr2 cách ptr1 2 phần tử thì sau một bước duyệt nữa ptr2 sẽ nằm ngay sau ptr1 và thêm một bước duyệt nữa để bằng ptr1.

Vì $m \leq n$ nên khi bị bắt kịp, cùng lắm là ptr1 đã đi đến cuối L.

Tổng kết lại ta có kết luật nếu L có vòng thì ptr2 chắn chắn sẽ bằng ptr1 trong nhiều nhất là n bước duyệt. Suy ra thuật toán có độ phức tạp thời gian $\theta(n)$ và bộ nhớ $\theta(1)$.

4. Fast fourier transform

Biến đổi fourier rời rạc là biến đổi cực kỳ quan trọng với các công nghệ số hiện tại.

Theo định nghĩa của wikipedia, biến đổi fourier rời rạc biến đổi một tập complex samples $\{x_n\} := x_0, x_1, \dots, x_{N-1}$ (dịch là tập mẫu phức nhưng em thấy để nguyên tiếng anh thì đúng nghĩa hơn) thành một tập complex point $\{X_k\} := X_0, X_1, \dots, X_{N-1}$ khác theo công thức

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{i2\pi}{N}kn}$$

Hoặc có thể dùng công thức của Euler để biến đổi công thức trên từ dạng mũ phức về dạng số phức $a + bi$

$$X_k = \sum_{n=0}^{N-1} x_n \left[\cos\left(\frac{2\pi}{N}kn\right) - i * sin\left(\frac{2\pi}{N}kn\right)\right]$$

Nếu tính trực tiếp từng X_k một ta sẽ mất thời gian $\theta(N)$ cho mỗi X_k, vậy nhìn chung một thuật toán kiểu này sẽ mất thời gian $\theta(N^2)$ để tính hết tập X.

Một thuật toán biến đổi Fourier nhanh của Cooley và Tokey thực hiện biến đổi Fourier rời rạc trên chỉ trong thời gian $\theta(N \log_2 N)$.

```
X_0,...,N-1 ← ditfft2(x, N, s):
    if N = 1 then
        X_0 ← x_0
    else
        X_0,...,N/2-1 ← ditfft2(x, N/2, 2s)
        X_N/2,...,N-1 ← ditfft2(x+s, N/2, 2s)
        for k = 0 to N/2-1 do
            t ← X_k
            X_k ← t + exp(-2πi k/N) X_k+N/2
            X_k+N/2 ← t - exp(-2πi k/N) X_k+N/2
        end for
    end if
```

Thuật toán mặc định input đầu vào N phải là một lũy thừa của 2, nếu không input sẽ được xử lý trước khi biến đổi (bằng cách thêm các biến 0 vào sample input).

Việc tính tổng X_k sẽ được chia thành 2 phần như sau

$$X_k = \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-\frac{i2\pi}{N}2mk} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-\frac{i2\pi}{N}(2m+1)k}$$

$$X_k = \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-\frac{i2\pi}{N/2}mk} + e^{-\frac{i2\pi}{N}k} \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-\frac{i2\pi}{N/2}mk}$$

$$X_k = E_k + e^{-\frac{i2\pi}{N}k} O_k$$

Với E_k là tổng trên các phần tử chẵn của x, O_k là tổng trên các phần tử lẻ

$$E_k = \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-\frac{i2\pi}{N/2}mk}, O_k = \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-\frac{i2\pi}{N/2}mk}$$

Nhờ tính chất tuần hoàn của số phức, giá trị của $X_{k+\frac{N}{2}}$ cũng tính được qua E_k và O_k

$$X_{k+\frac{N}{2}} = \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-\frac{i2\pi}{N}2m(k+\frac{N}{2})} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-\frac{i2\pi}{N}(2m+1)(k+\frac{N}{2})}$$

$$X_{k+\frac{N}{2}} = \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-\frac{i2\pi}{N/2}mk} e^{-2\pi mi} + e^{-\frac{i2\pi}{N}k} e^{-\pi i} \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-\frac{i2\pi}{N/2}mk} e^{-2\pi mi}$$

$$X_{k+\frac{N}{2}} = \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-\frac{i2\pi}{N/2}mk} - e^{-\frac{i2\pi}{N}k} \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-\frac{i2\pi}{N/2}mk}$$

$$X_{k+\frac{N}{2}} = E_k - e^{-\frac{i2\pi}{N}k} O_k$$

Mà E_k và O_k đều là biến đổi Fourier rời rạc của mỗi chuỗi x có độ dài bằng $\frac{N}{2}$

Nên thuật toán đã gọi đệ quy để tính E_k và O_k rồi sau đó dùng $\theta(N)$ phép tính để khôi phục lại dãy X.

Từ đó nếu đặt $T(N)$ là độ phức tạp của thuật toán thì

$$T(N) = 2T\left(\frac{N}{2}\right) + \theta(N)$$

Dùng Master theorem ta được $T(N) = \theta(N \log_2 N)$

5. Subset sum

Bài toán: Cho mảng X có n phần tử, và số thực T. Xác định xem trong trong X có tập con nào có tổng bằng T hay không.

Một thuật toán vét cạn giải quyết bài toán này thường có độ phức tạp $\theta(2^n)$

Dưới đây là một thuật toán quy hoạch động giải quyết bài toán trên

```
DynamicSubsetSum(X[1, 2, ..., n], T):
  S[0, 0] ← True
  for t ← 1 to T
    S[0, t] ← False
  for i ← 1 to n
    S[i, 0] = True
    for t ← 1 to X[i − 1]
      S[i, t] ← S[i − 1, t]              [[avoid the case t < 0]]
    for t ← X[i] to T
      S[i, t] ← S[i − 1, t] ∨ S[i − 1, t − X[i]]
  return S[n, T]
```

Dựa vào một tính chất của tổng, ta có thể giảm bớt được kích thước bài toán.

Xét một phần tử $x \in X$, tồn tại một tập con của X có tổng bằng T nếu một trong 2 điều kiện sau là đúng

- Tồn tại tập con $X \setminus \{x\}$ có tổng bằng $T - x$

- Tồn tại tập con $X \setminus \{x\}$ có tổng bằng T

Gọi SS(n) là độ phức tạp thời gian của thuật toán

Từ đó thuật toán có công thức đệ quy

$$SS(n) = \begin{cases} True, & if\ T = 0 \\ False, & if\ T < 0\ or\ i < 0 \\ SS(n-1, T - X[n]) \cup SS(n-1, T), & otherwise \end{cases}$$

Sau đó thuật toán lưu lại các kết quả của các bài toán con vào một mảng 2 chiều SS $n * T$ phần tử. Giá trị tại vị SS[i][t] là True nếu tồn tại mảng con i phần tử có tổng bằng t, ngược lại SS[i][t] bằng False.

Từ đó ta có thể tính độ phức tạp SS(n) qua việc đếm số phần tử trong mảng SS

Việc tính giá trị cho một phần tử trong SS chỉ mất thời gian $\theta(1)$. Suy ra thuật toán có độ phức tạp $\theta(nT)$.

Thực ra subset sum là một bài toán thuộc lớp NP-complete khi n và T là những số rất lớn, biến đầu vào T có thể biểu diễn dưới dạng $l = \log_2 T$ bit đầu vào. Độ phức tạp của thuật toán khi đó là hàm mũ $\theta(n2^l)$ chưa kể các phép tính trên số cực lớn.