Chapter 1: Asymptotics

1. Big Oh

1.1 Definition

> $f(n) = O(g(n))$ *if* $\exists c, n_0 > 0$ *such that* $0 \leq f(n) \leq cg(n)$ *for all* $n \geq n_0$

1.2 Methods to prove

1.2.1 Definition method

Example: prove $\frac{1}{2}n^2 - 3n = O(n^2)$

Answer:

$$\Leftrightarrow 0 \leq \frac{1}{2}n^2 - 3n \leq c \cdot n^2$$
$$\Leftrightarrow \frac{1}{2} - \frac{3}{n} \leq c \quad (n \geq 6)$$

Therefore, if $c = \frac{1}{2}$ and $n_0 = 6$, $0 \leq f(n) \leq \frac{1}{2}g(n)$ is true for all $n \geq n_0$

1.2.2 Limit method

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = O(g(n))$$

Example: prove $2^{n+1} = O(4^n)$

Answer:

$$
\begin{aligned}
lg(\lim_{n\to\infty}\frac{f(n)}{g(n)}) &= \lim_{n\to\infty} lg\frac{f(n)}{g(n)} \\
&= \lim_{n\to\infty}(lg2^{n+1} - lg4^n) \\
&= \lim_{n\to\infty}(n+1-2n) \\
&= -\infty
\end{aligned}
$$

Therefore, $\lim_{n\to\infty}\frac{2^{n+1}}{4^n} = 0$, which implies that $2^{n+1} = O(4^n)$

1.3 Special cases

- $n! = O(n^n)$

$$
\begin{aligned}
n! &= 1\cdot 2\cdot 3\cdots 4 \\
&\leq n\cdot n\cdots n \\
&= n^n
\end{aligned}
$$

$\Rightarrow lg(n!) = O(nlgn)$

- $f(n) = O(n^k) \quad iFF \quad lg(f(n)) = O(lgn)$

For the forward direction, assume $f(n) = O(n^k)$ for some k. We have $c_1, n_0 > 0$ such that for all $n \geq n_0$:

$$
\begin{aligned}
f(n) &\leq c_1 n^k \\
\lg(f(n)) &\leq \lg\left(c_1 n^k\right) \\
&= \lg c_1 + k\lg n \\
&\leq c_2 \lg n
\end{aligned}
$$

So $\lg(f(n)) = \mathcal{O}(\lg n)$.

For the reverse direction, assume $lg(f(n)) = O(lgn)$. We have $c_3, n_0 > 0$ such that for all $n \geq n_0$:

$$
\begin{aligned}
\lg(f(n)) &\leq c_3 \lg n \\
\lg(f(n)) &\leq \lg(n^{c_3}) \\
f(n) &\leq n^{c_3}
\end{aligned}
$$

So $f(n)$ is polynomially-bounded by n^{c3}.

2. Big Omega

2.1 Definition

> $f(n) = \Omega(g(n))$ *if* $\exists c, n_0 > 0$ *such that* $0 \leq cg(n) \leq f(n)$ *for all* $n \geq n_0$

2.2 Methods to prove

2.2.1 Definition method

Example: prove $1 + 2 + \cdots + n = \Omega(n^2)$

Answer:

$$\begin{aligned} 1 + 2 + \cdots + n &\geq [\frac{n}{2}] + [\frac{n}{2} + 1] + \cdots + n \\ &\geq [\frac{n}{2}] + [\frac{n}{2}] + \cdots + [\frac{n}{2}] \\ &\geq \frac{n}{2} \cdot \frac{n}{2} \\ &= \frac{n^2}{4} \end{aligned}$$

Therefore, if $c = \frac{1}{4}$ and $n_0 = 1, 0 \leq \frac{1}{4}n^2 \leq 1 + 2 + \cdots + n$ is true for all $n \geq n_0$

2.2.2 Limit method

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \Omega(g(n))$$

Example: prove $2^{n^2} = \Omega(3^n)$

Answer:

$$\begin{aligned} lg(\lim_{n \to \infty} \frac{f(n)}{g(n)}) &= \lim_{n \to \infty} lg\frac{f(n)}{g(n)} \\ &= \lim_{n \to \infty} (lg2^{2^n} - lg3^n) \\ &= \lim_{n \to \infty} (n^2 - nlg3) \\ &= \infty \end{aligned}$$

Therefore, $\lim_{n \to \infty} \frac{2^{2^n}}{3^n} = \infty$, which implies that $2^{n^2} = \Omega(3^n)$

3. Big Theta

3.1 Definition

> $f(n) = \Theta(g(n))$ *if* $\exists c_1, c_2, n_0 > 0$ *such that* $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ *for all* $n \geq n_0$

3.2 Methods to prove

3.2.1 Definition method

Example: prove $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$

Answer:

$O(n^{k+1})$:

$$\sum_{i=1}^{n} i^k \leq \sum_{i=1}^{n} n^k = n \cdot n^k = n^{k+1}$$

$\Rightarrow c_2 = 1$

$\Omega(n^{k+1})$:

$$\begin{aligned} 2\sum_{i=1}^{n} i^k &= \sum_{i=1}^{n} i^k + \sum_{i=1}^{n} (n-i+1)^k \\ &= \sum_{i=1}^{n} i^k + (n-i+1)^k \\ &\geq \sum_{i=1}^{n} (\frac{n}{2})^k \\ &= \frac{n^{k+1}}{2^{k+1}} \\ &= \Omega(n^{k+1}) \end{aligned}$$

$\Rightarrow c_1 = \frac{1}{2^{k+1}}$

Therefore, if $c_1 = \frac{1}{2^{k+1}}$ and $c_2 = 1, 0 \leq \frac{1}{2^{k+1}} n^{k+1} \leq \sum_{i=1}^{n} i^k \leq n^{k+1}$ is true for all $n \geq n_0$

3.2.2 Limit method

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = constant \Rightarrow f(n) = \Theta(g(n))$$

Example: prove $lg(n^2) = \Theta(lgn + 5)$

Answer:

$$\begin{aligned}\lim_{n\to\infty}\frac{lg(n^2)}{lgn+5} &= \lim_{n\to\infty}\frac{\frac{2}{nln2}}{\frac{1}{nln2}} \\ &= 2\end{aligned}$$

Therefore, $\lim_{n\to\infty}\frac{lg(n^2)}{lgn+5} = 2$, which implies that $lg(n^2) = \Theta(lgn + 5)$

4. Properties

4.1 Transitivity

> $f(n) = \Theta(g(n)) \quad \& \quad g(n) = \Theta(h(n)) \quad \Rightarrow \quad f(n) = \Theta(h(n))$

> $f(n) = O(g(n)) \quad \& \quad g(n) = O(h(n)) \quad \Rightarrow \quad f(n) = O(h(n))$

> $f(n) = \Omega(g(n)) \quad \& \quad g(n) = \Omega(h(n)) \quad \Rightarrow \quad f(n) = \Omega(h(n))$

4.2 Symmetry

> $f(n) = \Theta(g(n)) \quad iFF \quad g(n) = \Theta(f(n))$

4.3 Transpose

> $f(n) = O(g(n)) \quad iFF \quad g(n) = \Omega(f(n))$

4.4 Cookbook

> $n^a = O(n^b) \quad iFF \quad a \leq b$

> $log_a n = O(log_b n) \quad \forall a, b$

> $c^n = O(d^n) \quad iFF \quad c \leq d$

> $f(n) = O(f_1(n)) \quad \& \quad g(n) = O(g_1(n))$
>
> > $\Rightarrow f(n) \cdot g(n) = O(f_1(n) \cdot g_1(n))$
> >
> > $\Rightarrow f(n) + g(n) = O(max(f_1(n), g_1(n)))$

Chapter 2: Recurrences

---

1. Merge sort

$$T(n) = 2T(\lceil \frac{n}{2} \rceil) + \Theta(n)$$

2. Master theorem

> *It is a method to solve recurrences of the form*
>
> $$T(n) = aT(\frac{n}{b}) + f(n)$$
>
> *where $a \geq 1, b \geq 1$, and $f(n)$ is asymptotically positive.*

2.1 Case 1

> *If $\exists \epsilon > 0$, making $f(n) = O(n^{\log_b a - \epsilon}) \quad \Rightarrow \quad T(n) = \Theta(n^{\log_b a})$*

Example: $T(n) = 7T(\frac{n}{2}) + n^2$

$$\begin{gathered} \log_b a = \lg 7 \quad (\text{between 2 and 3}) \\ f(n) = n^2 = O(n^{\lg 7 - \epsilon}) = O(n^{\log_b a - \epsilon}) \quad \text{for} \quad \epsilon = \lg 7 - 2 \end{gathered}$$

This means that we have case 1, and $T(n) = \Theta(n^{\lg 7})$

2.2 Case 2

> *If* $f(n) = \Theta(n^{\log_b a}) \quad \Rightarrow \quad T(n) = \Theta(n^{\log_b a} \lg n)$

Example: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$

$$f(n) = \Theta(n) = \Theta(n^{\log_b a})$$

This means that we have case 2, and $T(n) = \Theta(n \lg n)$

2.3 Case 3

> *If* $\exists \epsilon > 0$, *making* $f(n) = \Omega(n^{\log_b a + \epsilon})$
>
> *and* $\exists c < 1$, *making* $af(\frac{n}{b}) \leq cf(n)$ *for all* $n > n_0$ *(regularity condition)*
>
> $\Rightarrow \quad T(n) = \Theta(f(n))$

Example: $T(n) = 4T(\frac{n}{2}) + n^2\sqrt{n}$

$$log_b a = 2$$
$$f(n) = n^{\frac{5}{2}} = O(n^{2+\epsilon}) = O(n^{log_b a + \epsilon}) \quad \text{for} \quad \epsilon = \frac{1}{2}$$

If the regularity condition holds:

$$af(\frac{n}{b}) \leq cf(n)$$
$$4f(\frac{n}{2}) \leq cf(n)$$
$$4(\frac{n}{2})^{\frac{5}{2}} \leq cn^{\frac{5}{2}}$$
$$\sqrt{\frac{1}{2}} \leq c$$

So $\exists c = \sqrt{\frac{1}{2}}$, making regularity condition hold.

This means that we have case 3, and $T(n) = \Theta(f(n)) = \Theta(n^2\sqrt{n})$

3. Substitution method

> *The idea is to guess a solution and then prove it by induction.*

Example:

Prove that $T(n) = 2T(n/2) + n = \Theta(n \lg n)$

Guess: $T(n) = O(n \lg n)$

Induction hypothesis: For all $k < n, T(k) \leq ck \lg k$

Induction:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + n \\ &\leq 2c\frac{n}{2}\lg(\frac{n}{2}) + n \\ &= cn \lg n + (1-c)n \end{aligned}$$

which implies that $T(n) \leq cn \lg n$ if:

$$(1-c)n \leq 0$$

This is true when $c \geq 1$. So, $T(n) = O(n \lg n)$.

Guess: $T(n) = \Omega(n \lg n)$

Induction hypothesis: For all $k < n, T(k) \geq dk \lg k$

Induction:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + n \\ &\geq 2d\frac{n}{2}\lg(\frac{n}{2}) + n \\ &= dn \lg n + (1-d)n \end{aligned}$$

which implies that $T(n) \geq dn \lg n$ if:

$$(1-d)n \geq 0$$

This is true when $0 < d \leq 1$. So, $T(n) = \Omega(n \lg n)$.

Therefore, $T(n) = \Theta(n \lg n)$ is proved.

4. Recursion tree

> *The idea is to visualize a divide-and-conquer computation where each node represents a subproblem.*

Example:

$T(n) = 3T(\frac{n}{4}) + n^2, T(1) = \Theta(1)$

(a) The depth

$$h = O(\log_4 n)$$

This is because: $n/4^h = 1$

(b) The work

The 0 level: node = 3^0 + unit work = $n^2 = (\frac{1}{16})^0 n^2$

The 1 level: node = 3^1 + unit work = $(\frac{1}{4})^2 n^2 = (\frac{1}{16})^1 n^2$

The 2: node = 3^2 + unit work = $(\frac{1}{4})^4 n^2 = (\frac{1}{16})^2 n^2$

.....

The i level: node = 3^i + unit work = $(\frac{1}{4})^(2i) n^2 = (\frac{1}{16})^i n^2$

The work at level i is:

$$(\frac{3}{16})^i n^2$$

The total amount of work in the recursion tree is:

$$\begin{aligned}\sum_{i=0}^{\log_4 n} (\frac{3}{16})^i n^2 &< \sum_{i=0}^{\infty} (\frac{3}{16})^i n^2 \\ &= \frac{1}{1-\frac{3}{16}} n^2 \\ &= \frac{16}{13} n^2 \\ &= O(n^2)\end{aligned}$$

Therefore, we guess that $T(n) = O(n^2)$.

Chapter 3: Sorting

1. Heapsort

1.1 Heaps

1.1-1 Index

(a) Given the index i of a certain node:

- $\text{Index}[\text{parent}] = i/2$
- $\text{Index}[\text{left}] = 2i$
- $\text{Index}[\text{right}] = 2i + 1$

(b) For an n-element heap, the leaf nodes are indexed by:

$$\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$$

This is because $\text{Index}[\text{left}(\lfloor n/2 \rfloor + 1)] = 2(\lfloor n/2 \rfloor + 1) > 2(n/2 - 1) + 2 = n$.

Therefore, the leaf nodes correspond to the second half of the heap array (plus the middle element if n is odd).

1.1-2 Count

In a heap of height h:

The minimum number of elements: 2^h.

The maximum number of elements: $2^{h+1} - 1$.

This is because a complete binary tree of depth $h - 1$ has $\sum_{i=0}^{h-1} 2^i = 2^h - 1$ elements.

In an n-element heap:

The maximum number of elements at height h is: $\lceil n/2^{h+1} \rceil$.

This is because:

$$n_h = \lceil \frac{n_{h-1}}{2} \rceil \leq \lceil \frac{1}{2} \lceil \frac{n}{2^{(h-1)+1}} \rceil \rceil = \lceil \frac{1}{2} \lceil \frac{n}{2^h} \rceil \rceil = \lceil \frac{n}{2^{h+1}} \rceil$$

1.1-3 Height

An n-element heap has height $\lfloor \lg n \rfloor$.

This is because in 1-2, we have: $2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1} \Rightarrow h \leq \lg n < h + 1$.

1.2 MAX_HEAPIFY

1.2-1 Preconditions

The binary trees rooted at $\text{Index}[\text{left}(i)]$ and $\text{Index}[\text{right}(i)]$ are already max-heaps.

1.2-2 Algorithms

1.2-2-1 Recursion

```
MAX_HEAPIFY(A, i) (official)
    # 在 A[i], A[left(i)], A[right(i)] 中选出最大的
    l = LEFT(i)
    r = RIGHT(i)
    if l <= A.heap_size and A[l] > A[i]
        largest = l
    else largest = i
    if r <= A.heap_size and A[r] > A[largest]
        largest = r
    # 如果 A[i] 是最大的，直接退出
    # 如果 A[i] 不是最大的，交换 A[i] 和 largest，继续递归
    if largest != i
        exchange A[i] with A[largest]
        MAX_HEAPIFY(A, largest)
```

1.2-2-2 While

```
MAX_HEAPIFY(A, i)
    while true
        # 在 A[i], A[left(i)], A[right(i)] 中选出最大的
        l = LEFT(i)
        r = RIGHT(i)
        if l <= A.heap_size and A[l] > A[i]
            largest = l
```

```
        else largest = i
        if r <= A.heap_size and A[r] > A[largest]
            largest = r
        # 如果 A[i] 是最大的，直接退出
        if largest == i
            return
        # 如果 A[i] 不是最大的，交换 A[i] 和 largest，继续循环
        exchange A[i] with A[largest]
        i = largest
```

1.2-3 Analysis

It takes $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, $A[\text{LEFT}(i)]$, $A[\text{RIGHT}(i)]$, plus the time to run recursion on a subtree (assuming that the recuisive call occurs) while the children's subtrees each have size at most $2n/3$. Therefore:

$$T(n) \leq T(2n/3) + \Theta(1)$$

Using the case 2 of the master theorem, the solution is $T(n) = O(\lg n)$.

1.3 BUILD_MAX_HEAP

1.3-1 Algorithms

```
BUILD_MAX_HEAP(A)
    A.heap_size = A.length
    # 自底向上构造最大堆
    for i = floor(A.length / 2) downto 1
        MAX_HEAPIFY(A, i)
```

We need to note that nodes indexed at $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ are leaves. This is why the loop starts at $\lfloor n/2 \rfloor$.

1.3-2 Analysis

We know the lemma that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in the heap.

And the cost of MAX_HEAPIFY at height h is $O(h)$.

Therefore, the cost of BUILD_MAX_HEAP is:

$$\begin{aligned}\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) &= O(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}) \\ &= O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) \\ &= O(n)\end{aligned}$$

1.4 HEAPSORT

1.4-1 Algorithms

```
HEAPSORT(A)
    # 构建最大堆
    BUILD_MAX_HEAP(A)
    for i = A.length downto 2
        # 将 A[i] 和 A[1] 交换
        exchange A[1] and A[i]
        A.heap_size = A.heap_size - 1
        # 将 A[1] 下沉
        MAX_HEAPIFY(A, 1)
```

1.4-2 Analysis

The cost of BUILD_MAX_HEAP is $O(n)$.

The cost of MAX_HEAPIFY for $n-1$ times is $(n-1)O(\lg n) = O(n \lg n)$.

Therefore, the cost of HEAPSORT is $O(n \lg n)$.

2. Quicksort

2.1 Algorithms

2.1-1 PARTITION

```
PARTITION(A, p, r)
    # x is the pivot element
    x = A[r]
    i = p - 1
    for j = p to r-1
        if A[j] <= x
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i+1] with A[r]
    return i+1
```

2.1-2 QUICKSORT

```
QUICKSORT(A, p, r)
    if p < r
        q = PARTITION(A, p, r)
        # A[p..q-1] 和 A[q+1..r] 有序
        QUICKSORT(A, p, q-1)
        QUICKSORT(A, q+1, r)
```

2.2. Analysis

2.2-1 Worst case

The worst case occurs when the partition produces one subproblem with $n-1$ elements and one with 0 element.

$\Rightarrow$ The array is sorted or reverse sorted.

$$\begin{aligned} T(n) &= \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

Therefore, in the worst case, the cost is $\Theta(n^2)$.

2.2-2 Best case

The best case occurs when the partition produces two subproblems with equal elements.

$$\begin{aligned} T(n) &= \min_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) \\ &= 2T(\frac{n}{2}) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

Therefore, in the best case, the cost is $O(n \log n)$.

2.2-3 Average case

In the average case, the running time is $O(n \log n)$ whenever the split has constant proportionality.

3. Randomized Quicksort

3.1 Algorithms

```
RANDOMIZED_PARTITION(A, p, r)
    i = RANDOM(p, r)
    exchange A[r] with A[i]
    return PARTITION(A, p, r)
```

```
RANDOMIZED_QUICKSORT(A, p, r)
    if p < r
        q = RANDOMIZED_PARTITION(A, p, r)
        RANDOMIZED_QUICKSORT(A, p, q-1)
        RANDOMIZED_QUICKSORT(A, q+1, r)
```

3.2 Analysis

3.2-1 Comparison-based

We rename elements in A as $z_1, z_2, \ldots, z_n$ in which z_i is the i-th smallest element in A.

We define that $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ contains elements between z_i and z_j.

We also define that $X_{ij} = I\{z_i \text{ and } z_j \text{ are compared}\}$ so that $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$.

The probability that z_i and z_j can be compared equals to the probablity that z_i or z_j is firstly selected as the pivot from Z_{ij}. It is easy to think about because once $z_i < z_k < z_j$ is selected as the pivot, z_i and z_j goes to two different partitions. Therefore:

$$\begin{aligned}\Pr\{z_i \text{ is compared with } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is firstly selected as the pivot from } Z_{ij}\} \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\ &= \frac{2}{j-i+1}\end{aligned}$$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} = \sum_{i=1}^{n-1} O(\log n) = O(n \log n)$$

3.2-2 Expectation-based

Like what we did in part (a), we define indicator random variables:

$$X_i = I\{i\text{th largest element is chosen as } x^*\}$$

Since x^* is selected as a random element in the array with a size of n, the probabilities of any particular element being selected are all equal. Therefore:

$$E[X_i] = \Pr[i\text{th largest is picked}] = \frac{1}{n}$$

We can apply linearity of expectation over all of the events X_i, and the recurrence for the expected running time of RANDOMIZED_QUICKSORT is as follows:

$$\begin{aligned}E[T(n)] &= E[\sum_{i=1}^{n} X_i (T(i-1) + T(n-q) + \Theta(n))] \\ &= \sum_{i=1}^{n} E[X_i (T(i-1) + T(n-q) + \Theta(n))] \\ &= \frac{1}{n} \sum_{i=1}^{n} (E[T(i-1) + T(n-q)] + \Theta(n)) \\ &= \Theta(n) + \frac{2}{n} \sum_{i=1}^{n} E[T(i-1)] \\ &= \Theta(n) + \frac{2}{n} \sum_{i=1}^{n-1} E[T(i)]\end{aligned}$$

In class, we have proved the lemma that:

$$\sum_{k=1}^{n-1} k \log k \leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2$$

This is because:

$$\begin{aligned}
\sum_{k=1}^{n-1} k \log k &= \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k \log k + \sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} k \log k \\
&\leq (\log n - 1) \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k + \log n \sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} k \\
&= \log n \sum_{k=1}^{n} k - \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k \\
&\leq \frac{1}{2} n(n-1) \log n - \frac{1}{2} (\frac{n}{2} - 1) \frac{n}{2} \\
&\leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2
\end{aligned}$$

Assume that for any $k < n, T(k) \leq k \log k + \Theta(n)$,

$$\begin{aligned}
E[T(n)] &= \Theta(n) + \frac{2}{n} \sum_{i=1}^{n-1} E[T(i)] \\
&\leq \Theta(n) + \frac{2}{n} \sum_{i=1}^{n-1} (i \log i + \Theta(n)) \\
&\leq \frac{2}{n} \sum_{i=1}^{n-1} i \log i + \frac{2}{n} \Theta(n) + \Theta(n) \\
&\leq \frac{2}{n} (\frac{1}{2} n^2 \log n - \frac{1}{8} n^2) + \Theta(n) \\
&\leq n \log n - \frac{1}{8} n + \Theta(n) \\
&\leq n \log n + \Theta(n)
\end{aligned}$$

Therefore, $E[T(n)] = \Theta(n \log n)$.

4#. Lower Bound in C-B Sorting

4#.1 Theorem

> *Any sorting algorithm on n-elements that uses comparisons to sort elements of unlimited range ($-\infty$ to $+\infty$) makes no better than $\Omega(n \log n)$.*

In other words, $\Omega(n \log n)$ is the lower bound for any comparison-based sorting algorithm on an unlimited range in the worst case scenario.

Counting sort, Radix sort and Bucket sort are sorting algorithms in linear time. They do not violate the lower bound rule because they sort numbers in a range.

4#.2 Proof

Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements. Because each of the $n!$ permutations of the input appears as some leaf, we have $n! \leq l$. Since a binary tree of height h has no more than 2^h leaves, we have:

$$n! \leq l \leq 2^h$$

which, by taking logarithms, implies

$$\begin{aligned} h &\geq \log(n!) \\ &= \Omega(n \log n) \end{aligned}$$

4. Counting Sort

4.1 Algorithms

Definition: $A[1...n]$ is the input; $B[1...n]$ is the output; $C[0...k]$ provides temporary working storage. k is the range of n numbers.

```
COUNTING_SORT(A, B, k)
    for i = 0 to k
        C[i] = 0
    # C[i] contains the number of elements = i O(n)
    for j = 1 to A.length
        C[A[j]] = C[A[j]] + 1
    # C[i] contains the number of elements <= i O(k)
    for i = 1 to k
        C[i] = C[i] + C[i-1]
    for j = A.length downto 1
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] - 1
```

4.2 Analysis

The running time is $O(n+k)$. If $k = O(n)$, then $O(n+k) = O(n)$.

The last loop makes Counting Sort a Stable algorithm.

5. Radix Sort

5.1 Algorithms

Definition: $A[1...n]$ is the input; d is the number of digits in n numbers; k is the range of the digit d.

```
RADIX_SORT(A, d)
    for i = 1 to d
        Use Counting Sort to Sort Array A on Digit d
```

Radix Sort sorts from LSB to MSB.

5.2 Analysis

For one pass, it takes $\Theta(n+k)$

For all passes, it takes $d \cdot \Theta(n+k) = \Theta(dn+dk)$.

It we assume that d and k are constants, the running time is $\Theta(n)$.

6. Bucket Sort

6.1 Algorithms

Assumption: Elements are uniformly distributed over $[0, 1)$.

Definition: $A[1...n]$ is the input and $0 \leq A[i] < 1$; $B[0...n-1]$ is an array that works as n buckets.

```
BUCKET_SORT(A)
    n = A.length
    for i = 0 to n-1
        make B[i] an empty list
    for i = 1 to n
        insert A[i] into list B[floor(nA[i])]
    for i = 0 to n-1
        sort list B[i] with insertion sort
    concatenate the lists B[0], B[1],..., B[n-1] together in
order
```

6.2 Analysis

Definition: n_i is the number elements in $B[i]$.

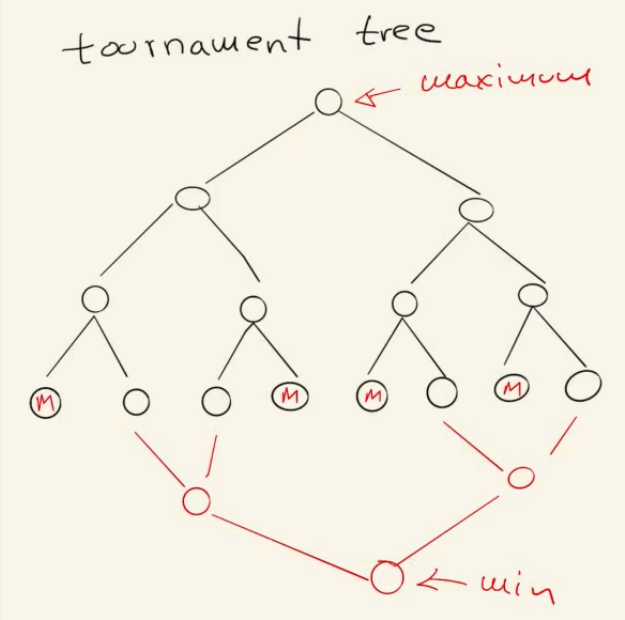
$$
\begin{aligned}
T(n) &= \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \\
&= \Theta(n) + O(2 - \frac{1}{n}) \\
&= \Theta(n) + O(n) \\
&= \Theta(n)
\end{aligned}
$$

7. Selection Algorithms

7.1 Tournament Tree

7.1-1 Find the max

We can see from the chart that it takes $(n-1)$ comparisons to find the max.

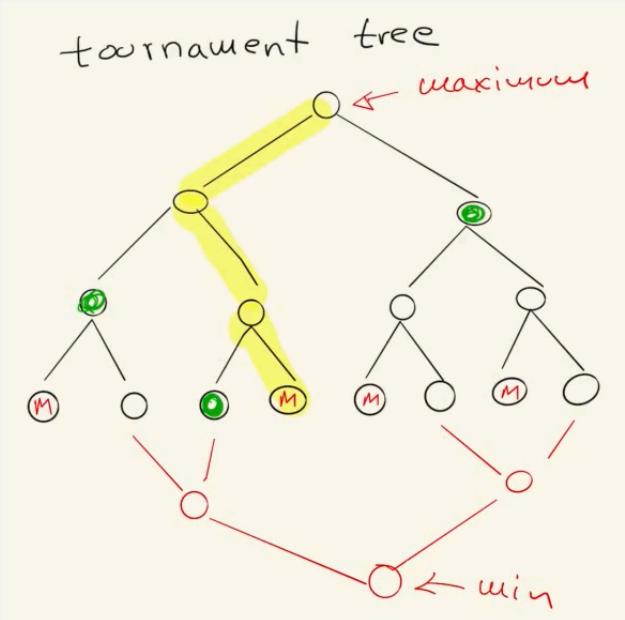


7.1-2 Find the min

Assuming we have found the max, it takes additional $(n/2 - 1)$ comparisons to find the min.

7.1-3 Find the kth max

Assuming we have found the max, it takes additional $O(\log n)$ comparisons to find the 2^{rd} max.



Therefore, selecting the k^{th} max takes $O(n + k \log n)$ running time.

7.2 Randomized Select

7.2-1 Algorithms

Definition: It returns the ith smallest element in $A[p \ldots r]$.

```
RANDOMIZED_SELECT(A, p, r, i)
    if p == r
        return A[p]
    q = RANDOMIZED_PARTITION(A, p, r)
    k = q - p + 1
    if i == k
        return A[q]
    else if i < k
        return RANDOMIZED_SELECT(A, p, q-1, i)
    else
        return RANDOMIZED_SELECT(A, q+1, r, i-k)
```

7.2-2 Expected time

We assume that the i-th smallest element is always bigger than the pivot.

$$
\begin{aligned}
E[T(n)] &\leq E[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)] \\
&= \sum_{k=1}^{n} E[X_k \cdot T(\max(k-1, n-k))] + O(n) \\
&= \sum_{k=1}^{n} E[X_k] \cdot E[T(\max(k-1, n-k))] + O(n) \\
&= \sum_{k=1}^{n} \frac{1}{n} \cdot E[T(\max(k-1, n-k))] + O(n) \\
&= O(n)
\end{aligned}
$$

7.3 Medium Select

7.3-1 Algorithms

- Divide the n elements into $\lfloor n/5 \rfloor$ groups with 5 elements in the groups other than the last one.
- Find the median of each of the $\lceil n/5 \rceil$ groups by first insertion-sorting the

elements and then picking.
- Find the median x of the $\lceil n/5 \rceil$ medians found.
- Partition the n elements around the median-of-medians x. Suppose there are $k-1$ elements in low side and $n-k$ elements in high side.
- If $i = k$, then return x. If $i < k$, return the i-th smallest in low side. If $i > k$, return the $(i-k)$-th smallest in high side.

7.3-2 Worst time

In the worst case, it runs in $O(n)$ time.

Chapter 4: Data Structure

---

1. Hashing Tables

1.1 Hash Tables

> *Memory:* $O(n)$
>
> *Time:* $O(n)$

U = Universe of n—keys.

m = Size of the hash table.

$\alpha = n/m$ is the load factor.

1.2 Hash Collisions

1.2-1 Expected Collisions

For successful search $(\alpha = n/m \leq 1)$, the number of expected collisions:

Chaining: $1 + \frac{\alpha}{2}$.

Linear Probe: $\frac{1}{2}(1 + \frac{1}{1-\alpha})$.

Double Hashing: $\frac{1}{2}\ln\frac{1}{1-\alpha}$

1.2-2 Hashing with Chains

> α *can* > 1.

For $j = 0, 1, \ldots, m-1$, denote the length of list $T[j]$ is n_j.

- $n = n_0 + n_1 + \ldots + n_{m-1}$.
- $E[n_j] = \alpha = n/m$.

1.2-2-1 Unsuccessful Search

> *Unsuccessful search takes expected time* $\Theta(1+\alpha)$.

Simple uniform hasing to have an unsuccessful search for key k needs to go to the end of $T[h(k)]$.

Since $E[n_{h(k)}] = \alpha$,

The expected time is $\Theta(\alpha)$ plus computing $h(k)$, which is $\Theta(1)$.

1.2-2-2 Successful Search

> *Successful search takes expected time* $\Theta(1+\alpha)$.

Need to find the number of elements inserted into x_j's list after x_j was inserted.

Note that $P_r\{h(k_i) = h(k_j)\} = 1/m \, (\, i \neq j \,) \quad \Rightarrow \quad E[X_{ij}] = 1/m$.

$$
\begin{aligned}
E[\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{ij})] &= \frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{ij}) \\
&= \frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}\frac{1}{m}) \\
&= 1+\frac{\alpha}{2}-\frac{\alpha}{2n} \\
&= \Theta(1+\alpha)
\end{aligned}
$$

1.2-3 Hashing with Open Addressing

> $0 < \alpha \leq 1$

Linear probing:

$$h(k, i) = (h'(k) + i) \bmod m$$

Double hashing:

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

1.3 Hash Functions

1.3-1 Division

$$h(k) = k \bmod m$$

- Bad values: $m = 2^p$
- Good values: $m = \text{prime} \cdot \#\text{search}$

1.3-2 Multiplication

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

Normally, $A = (\sqrt{5} - 1)/2$

- The value of m is not important here, normally $m = 2^p$

1.3-3 Universial

Randomly select or create h.

2. Binary Search Tree

2.1 Traverse

2.1-1 Inorder_tree_walk

```
INORDER_TREE_WALK(x)
    if x != NULL
        INORDER_TREE_WALK(x.left)
        print x.key
        INORDER_TREE_WALK(x.right)
```

If x is the root of an n-node subtree, then it takes $\Theta(n)$ time.

2.2 Search

Time of these operations is proportional to height h of the binary search tree, which is $O(h)$.

2.2-1 Tree_search

```
TREE_SEARCH(x, k)
    if x == NULL or k == x.key
        return x
    if k < x.key
        return TREE_SEARCH(x.left, k)
    else
        return TREE_SEARCH(x.right, k)
```

2.2-2 Tree_minimum

```
TREE_MINIMUM(x)
    while x.left != NULL
        x = x.left
    return x
```

2.2-3 Tree_maximum

```
TREE_MAXIMUM(x)
    while x.right != NULL
        x = x.right
    return x
```

2.2-4 Tree_successor

There are 2 cases:

- If x's right subtree is not empty, its successor is the leftmost node in x's right subtree.
- If x's right subtree is empty, we simply go up until we find a node that is a left child.

```
TREE_SUCCESSOR(x)
    if x.right != NULL
        return TREE_MINIMUM(x.right)
    y = x.parent
    while y != NULL and x == y.right
        x = y
        y = y.parent
    return y
```

2.3 Insert

Time is proportional to height h of the binary search tree, which is $O(h)$.

2.3-1 Tree_insert

```
TREE_INSERT(T, z)
    if T.root == NULL
        T.root = z
    y = NULL
    x = T.root
    while x != NULL
        y = x
        if z.key < x.key
            x = x.left
        else
```

```
            x = x.right
    z.parent = y
    if z.key < y.key
        y.left = z
    else
        y.right = z
```

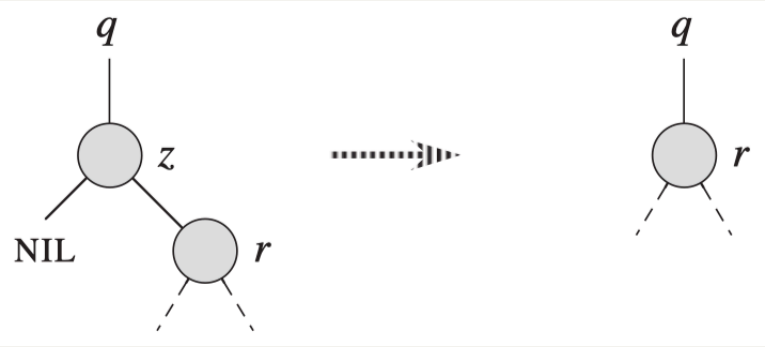
2.4 Delete

2.4-1 Tree_delete

Time is proportional to height h of the binary search tree, which is $O(h)$.

There are 4 cases:

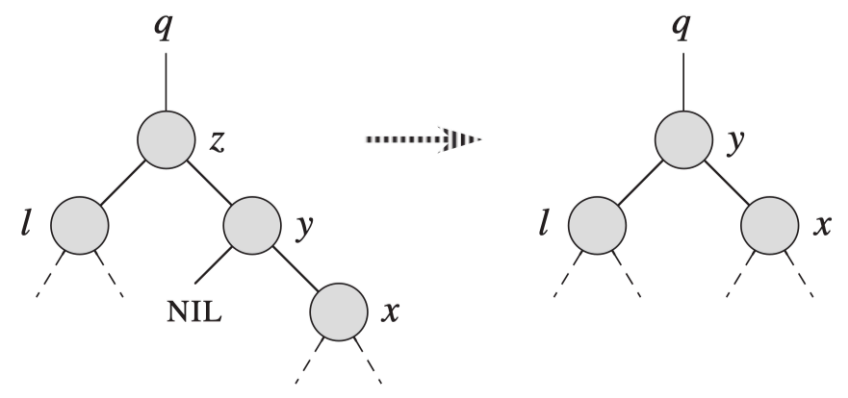
- If z has no left child. Transplant z with $z.right$.



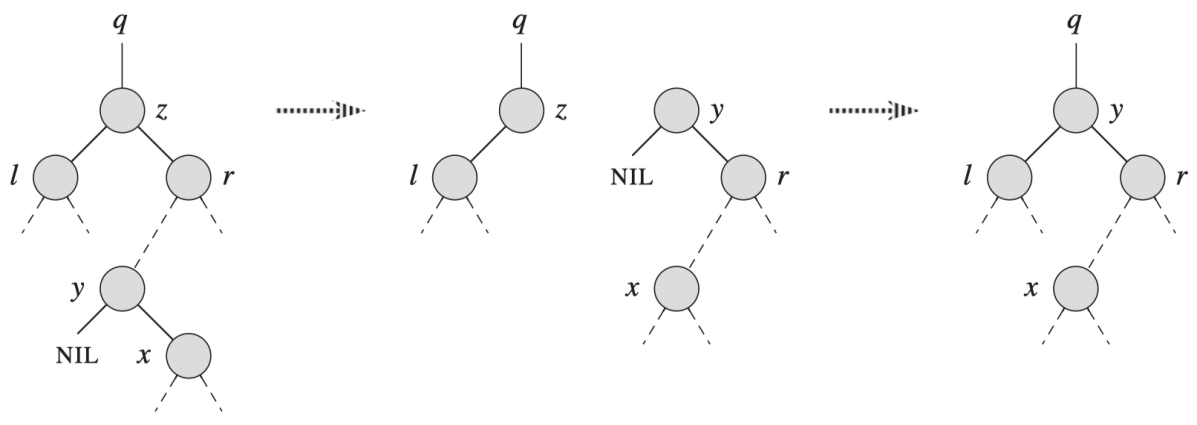
- If z has no right child. Transplant z with $z.left$.

If z has both a left and a right child. We find z's successor y, which lies in z's right subtree and has no left child.

- If y is z's right child. Transplant z with y.



- If y is not z's right child. Firstly transplant y with y.right, then transplant z with y




```
TREE_DELETE(T, z)
    if z.left == NULL
        TRANSPLANT(T, z, z.right)
    else if z.right == NULL
        TRANSPLANT(T, z, z.left)
    else
        y = TREE_MINIMUM(z.right)
        if y.parent != z
            TRANSPLANT(y, y.right)
            y.right = z.right
            y.right.parent = y
        TRANSPLANT(T, z, y)
        y.left = z.left
        y.left.parent = y
```

3. Red-Black Tree

3.1 Properties

3.1-1 Balance

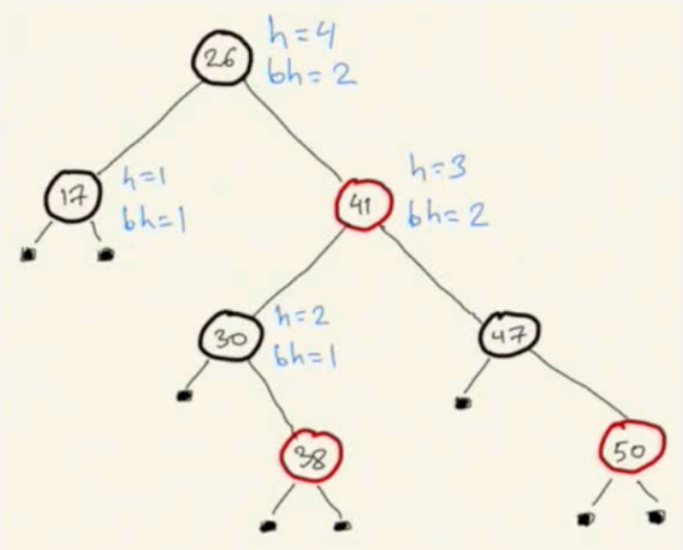
The Red-Black Tree is approximately balanced $\Rightarrow O(h) = O(\log n)$.

3.1-2 5 Properties

- Every node is either red or black.
- The root is black.
- Every leaf (NULL) is black.
- If a node is red, then both its children are black.
- For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

3.1-3 Height

- Height of node: the number of edges in longest path to a leaf.
- Black-height of node x: the number of black nodes on path from x to a leaf (exclude x).



3.1-3-1 Claim 1

- Any node x with height h has black-height $bh(x) \geq h/2$.

This is because by property 4, there are less than $h/2$ red nodes on path from node x to a leaf $\Rightarrow bh(x) \geq h/2$.

3.1-3-2 Claim 2

- The subtree rooted at any node x contains $\geq 2^{bh(x)} - 1$ internal nodes.

This can be proved by induction.

3.1-4 1 Lemma

- A red-black tree with n internal nodes has height $\leq 2\log(n+1)$.

Consider x being the root of the red-black tree. By Claim 1 and Claim 2,

$$n \geq 2^{bh(x)} - 1 \geq 2^{\frac{h}{2}} - 1$$

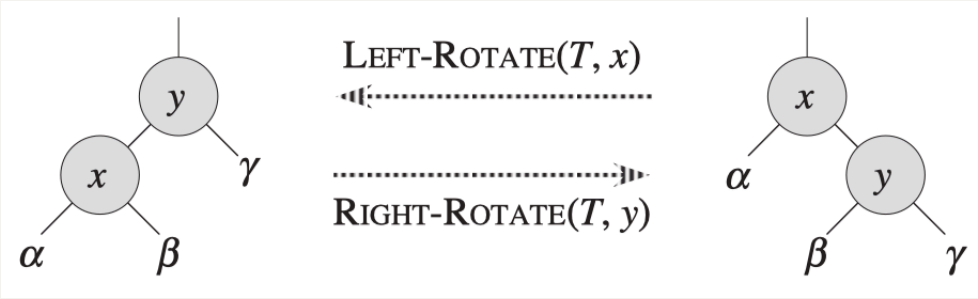
Therefore: $h \leq 2\log(n+1)$.

3.2 Operations

Operations include Tree_search, Tree_minimum, Tree_maximum, Tree_successor are the same in BST ($O(\log n)$).

3.2-1 Rotation

Time is $O(\log n)$.



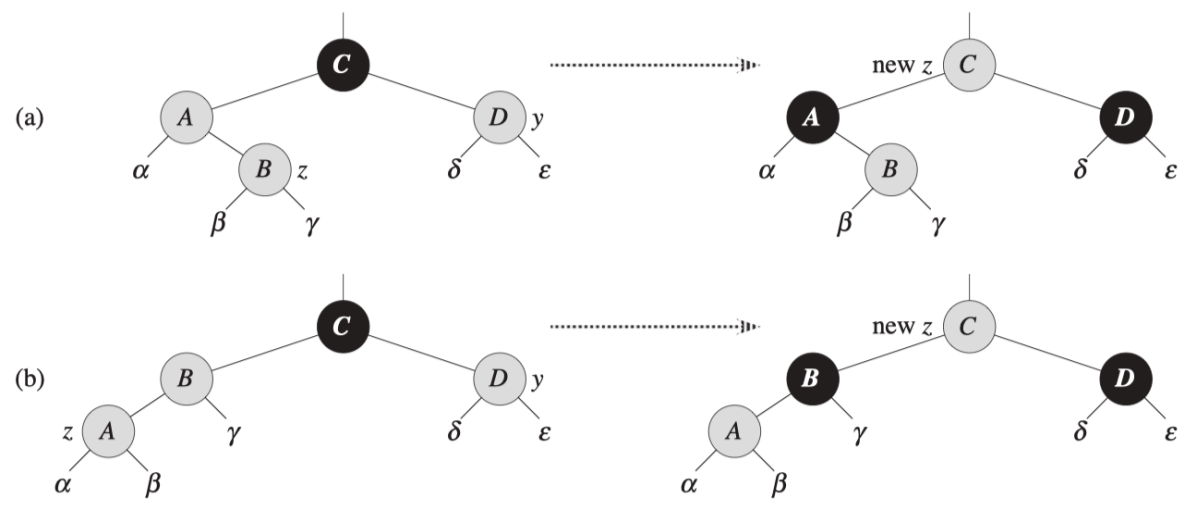

```
LEFT_ROTATE(T, x)
    y = x.right
```

```
    x.right = y.left
    if y.left != T.nil
        y.left.parent = x
    y.parent = x.parent
    if x.parent == T.nil
        T.root = y
    else if x == x.parent.left
        x.parent.left = y
    else
        x.parent.right = y
    y.left = x
    x.parent = y
```

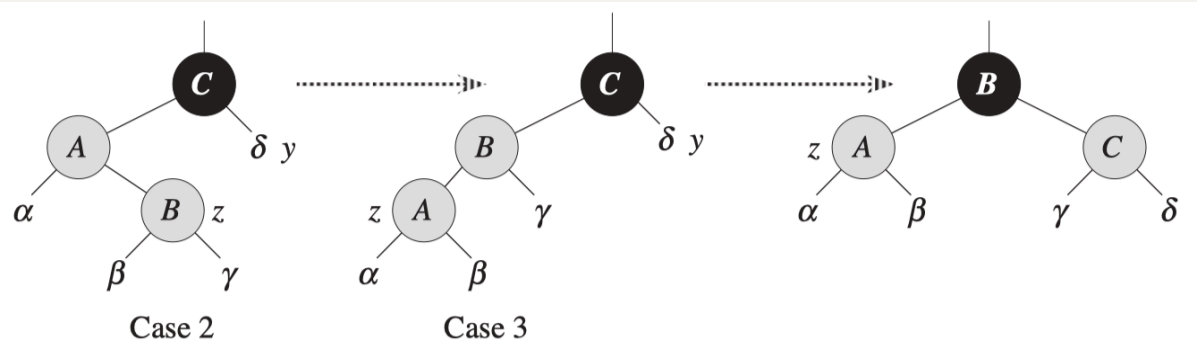
3.2-2 Insert

Time is $O(\log n)$.

- If z's uncle node y is red



- If z's uncle node y is black and z is a right child.
- If z's uncle node y is black and z is a left child.



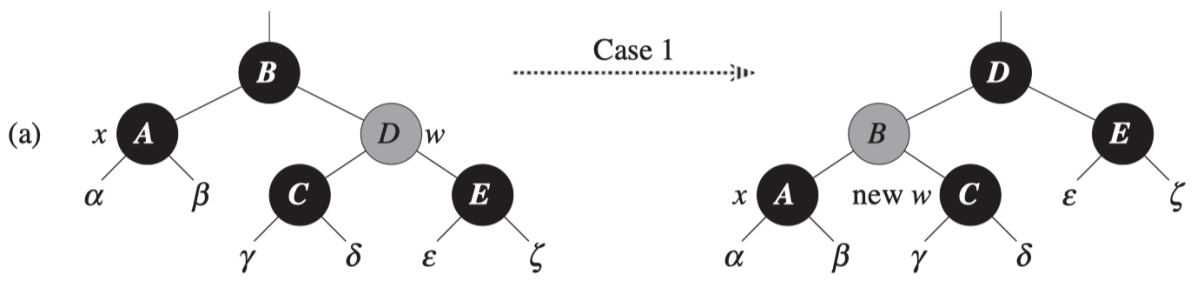
```
RB_INSERT(T, x)
    y = T.nil
    x = T.root
    while x != T.nil
        y = x
        if z.key < x.key
            x = x.left
        else
            x = x.right
    z.parent = y
    if y == T.nil
        T.root = z
    else if z.key < y.key
        y.left = z
    else
        y.right = z
    z.left = T.nil
    z.right = T.nil
    z.color = RED
    RB_INSERT_FIXUP(T, z)
```

```
RB_INSERT_FIXUP(T, z)
    while z.parent.color == RED
        if z.parent == z.parent.parent.left
            y = z.parent.parent.right
            # case 1
            if y.color == RED
                z.parent.color = BLACK
                y.color = BLACK
                z.parent.parent.color = RED
                z = z.parent.parent
            # case 2
            else if z == z.parent.right
                z = z.parent
                LEFT_ROTATE(T, z)
            # case 3
            z.parent.color = BLACK
            z.parent.parent.color = RED
            RIGHT_ROTATE(T, z.parent.parent)
        else (same as then clause with "right" and "left"
exchanged)
    T.root.color = BLACK
```

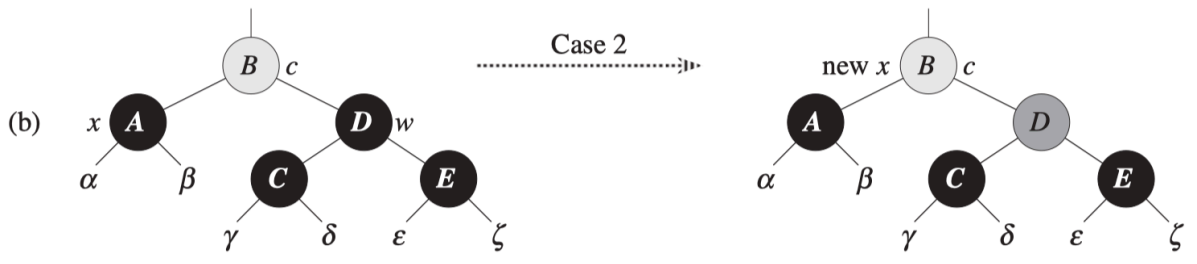
3.2-3 Delete

Time is $O(\log n)$.

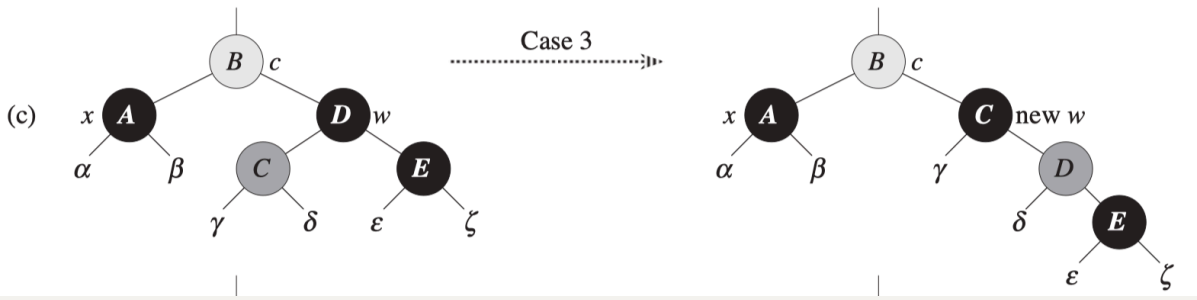
- x's sibling node w is red.



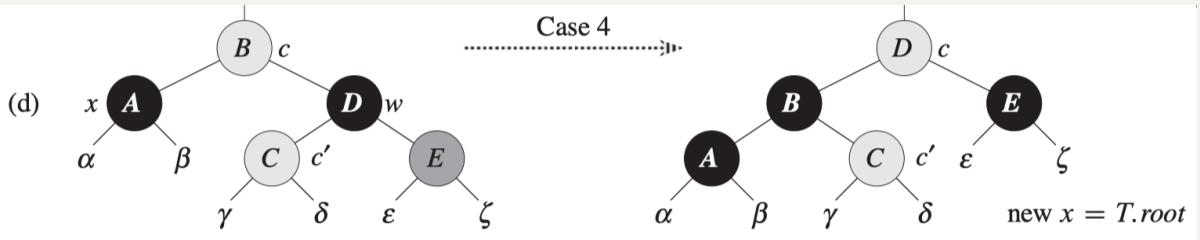
- x's sibling node w is black and both of w's children are black.



- x's sibling node w is black, w's left child is red and w's right child is black.



- x's sibling node w is black and w's right child is red.




```
RB_DELETE(T, z)
    y_original_color = y.color
    if z.left == T.nil
        x = z.right
```

```
        RB_TRANSPLANT(T, z, z.right)
    else if z.right == T.nil
        x = z.left
        RB_TRANSPLANT(T, z, z.left)
    else
        y = TREE_MINIMUM(z.right)
        y_original_color = y.color
        x = y.right
        if y.parent == z
            x.parent = y
        else
            RB_TRANSPLANT(T, y, y.right)
            y.right = z.right
            y.right.parent = y
        RB_TRANSPLANT(T, z, y)
        y.left = z.left
        y.left.parent = y
        y.color = z.color
    if y_original_color == BLACK
        RB_DELETE_FIXUP(T, x)
```

```
RB_DELETE_FIXUP(T, x)
    while x != T.root and x.color == BLACK
        if x == x.parent.left
            w = x.parent.right
            if w.color == RED
                w.color = BLACK
                x.parent.color = RED
                LEFT_ROTATE(T, x.parent)
                w = x.parent.right
            if w.left.color == BLACK and w.right.color == BLACK
                w.color = RED
                x = x.parent
            else if w.right.color == BLACK
                w.left.color = BLACK
                w.color = RED
                RIGHT_ROTATE(T, w)
                w = x.parent.right
            w.color = x.parent.color
            x.parent.color = BLACK
            w.right.color = BLACK
            LEFT_ROTATE(T, x.parent)
            x = T.root
```

```
        else (same as then clause with "right" and "left"
exchanged)
    x.color = BLACK
```

Chapter 5: Dynamic Programming

---

1. Two Characteristics

1-1 Optimal Substructure

> *Optimal solution to the "big" problem entails optimal solutions to similar-type subproblems.*

1-2 Overlapping Subproblems

> *As you break the large problem into smaller subproblems, you often need to re-calculate the same thing. Store intermeidate solutions & Reuse them.*

2. Matrix Chain Multiply Problem

```
MATRIX_CHAIN_ORDER(p)
    n = p.length - 1
    let m[1..n,1..n] be a new table
    for i=1 to n
        m[i,i] = 0
    for l=2 to n
        for i=1 to n-l+1
            j = i + l - 1
            m[i,j] = inf
            for k=i to j-1
                q = m[i,k] + m[k+1,j] + p{i-1}p{k}p{j}
                m[i,j] = min(m[i,j], q)
```

The running time is $O(n^3)$.

3. LCS Problem

```
LCS(X, Y)
    m = X.length
    n = Y.length
    let c[1..m,1..n] be a new table
    for i = 1 to m
        c[i,0] = 0
    for j = 0 to n
        c[0,j] = 0
    for i=1 to m
        for j=1 to n
            if x{i} == y{j}
                c[i,j] = c[i-1,j-1] + 1
            else
                c[i,j] = max(c[i-1,j], c[i,j-1])
    return c[m,n]
```

The running time is $\Theta(mn)$.

Chapter 6: Greedy Algorithms

---

1. Two Principles

1-1 Greedy Principle

> *A global optimal is reached by doing local greedy choices.*

1-2 Optimal Sub-Structure

> *Like dynamic planning.*

2. Activity Selection Problem

> *There is a set of possibly overlapping classes but only one classroom.*
>
> *How to maximize the number of classes to fit in the classroom ?*

- Sort by finish time
- Schedule with earliest finish time possible

It takes running time.

3. The Knapsack Problem

> *A thief enters a store that has items. Item values and weights . He can only carry weights.*

3-1 Fractional version

Greedy !

- Sort items per .
- Keep on taking in descending order, maximizing the value for the he can take.

It takes running time.

3-2 0-1 version

Dynamic !

Define be the maximum value taking from items with "leftover" weight .

4. Huffman Codes

```
HUFFMAN(C)
    n = |C|
    Q = C
    for i=1 to n-1
        allocate a new node z
        z.left = x = EXTRACT_MIN(Q)
        z.right = y = EXTRACT_MIN(Q)
        z.freq = x.freq + y.freq
        INSERT(Q, z)
    return EXTRACT_MIN(Q)
```

Chapter 7: Amortized Analysis

> *The goal of amortized analysis is to compute the average cost of each operation in a sequence of operations.*

In detail, we find the worst-case runtime of n operations and amortize the total over those n operations.

1. Aggregate Analysis

We compute the worst case runtime $T(n)$ of any sequence of n operations, and then divide by n to get the amortized cost.

1.1 Stack

Naive:

OPERATION	COST
Pop	$O(1)$
Push	$O(1)$
Multipop	$O(n)$

Therefore, n operations can cost up to $n \cdot O(n) = O(n^2)$. (Not a good up bound)

Aggregate:

You cannot pop more than what you push and you can push at most n items.

Therefore, the total amount of cost is $O(n)$ or $O(1)$ per operation.

1.2 Binary Counter

Naive:

Cost comes from flipping a bit. If we increase $O(n)$ times on a binary counter that has n bits, this may cost $n \cdot O(n) = O(n^2)$ in total.

Aggregate:

For n increments, the total number of filps is:

$$\sum_{i=1}^{n} \frac{n}{2^n} \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n = O(n)$$

1.3 Example

Suppose we perform a sequence of n operations on a data structure in which operation i takes i time if i is a power of 2 and takes 1 time otherwise. Use aggregate analysis to determine the amortized cost per operation.

$$c_i = \begin{cases} i & \text{if } i \text{ is a power of } 2 \\ 1 & \text{otherwise} \end{cases}$$

n operations cost

$$\begin{aligned} \sum_{i=1}^{n} c_i &\leq n + \sum_{j=0}^{\lg n} 2^j \\ &= n + (2n - 1) \\ &< 3n \end{aligned}$$

Therefore, the amortized cost is less than $3n = 3$. Thus, by aggregate analysis, the amortized cost is $O(1)$.

2. Accounting Method

The accounting method charges a certain amount for each operation (potentially different for different operations). The amount that exceeds the cost is credit, which can be spent to pay for other operations. The credit in an object must always be greater than or equal to 0.

2.1 Stack

Accounting:

OPERATION	COST	CHARGE
Push	$O(1)$	2$
Pop	$O(1)$	0$
Multipop	$O(1)$	0$

Charge push for 2 dollars. 1 dollar pays for the push and 1 dollar gets deposited to pay for the pop.

For n operations, it uses at most $2n = O(n)$ dollars in total or $O(1)$ amortized time and the credit never goes negative, so we've proven the amortized runtime is constant.

2.2 Binary Counter

Charge every $0 \to 1$ filp for 2 dollars. 1 dollars pays for the flip and 1 dollar gets deposited to pay for $1 \to 0$ flip.

For n increments, we need $2n = O(n)$ dollars or $O(1)$ amortized time and the credit never goes negative, so we've proven the amortized runtime is constant.

2.3 Example

Suppose we perform a sequence of n operations on a data structure in which operation i takes i time if i is a power of 2 and takes 1 time otherwise. Use the accounting method to determine the amortized cost per operation.

Charge 3 dollars for each operation i.

If i is not a power of 2, use 1 dollar to pay for the operation and store 2 dollars as credit. If i is a power of 2, pay the i dollars using stored credit.

Clearly credit never goes negative as there are 2^{j-1} steps between steps 2^{j-1} and 2^j , each of which stores 2 dollars, for a total of 2^j dollars stored.

The amortized cost of each operation is $O(1)$ and the credit never goes negative, so we've proven the amortized runtime is constant.

Chapter 8: Graph Algorithms

1. Graph

1.1 Graphs

1.1.1 Directed graph

> *A directed graph G is a pair (V, E) where V is a finite set and E is a binary relation on V.*

1.1.2 Undirected graph

> *An undirected graph $G = (V, E)$, the edge set E consists of unordered pairs of vertices.*

1.2 Special undirected graphs

1.2.1 Bipartite graph

> *V can be divided into 2 sets V_1 and V_2*
>
> *s.t.* $V_1 \cap V_2 = \varnothing$
>
> $V_1 \cup V_2 = V$

1.2.2 Simple graph

> *$G = (V, E)$ is a simple graph*
>
> *if* *no circle*
>
> *no repeated edge*

1.2.3 Complete graph

> *$G = (V, E)$ is a complete graph if every pair of vertices is adjacent.*

- Edge numbers: $C_n^2 = n(n-1)/2$

1.2.4 Subgraph

> *$G' = (V', E')$ is a subgraph of $G = (V, E)$*
>
> *if* $V' \subseteq V$
>
> $E' \subseteq E$

- Induced subgraph

> *$G' = (V', E')$ is an induced subgraph of $G = (V, E)$*
>
> *if* $V' \subseteq V$
>
> $E' = \{(u, v) \in E : u, v \in V'\}$

1.3 Isomorphism

1.3.1 Difinition

> *Two graphs $G = (V, E)$ and $G' = (V, E)$ are isomorphic if there exists a bijection*
>
> *$f : V \to V'$ such that $(u, v) \in E$ if and only if $(f(u), f(v)) \in E'$*

1.3.2 Necessary conditions:

- $|V| = |V'|$
- $|E| = |E'|$
- The vertices of the same degree have the same number of edges

1.4 Degrees

1.4.1 Definition

> *The degree of a vertex in an undirected graph is the number of edges incident on it.*

> *The degree of a vertex in a directed graph is its in-degree plus its out-degree.*

1.4.2 Handshaking lemma

> *if $G = (V, E)$ is an undirected graph, then $\sum_{v \in V} \text{degree}(V) = 2|E|$*

1.5 Connectivity

1.5.1 Definition

> *An undirected graph is connected if every vertex is reachable from all other vertices.*

> *A directed graph is strongly connected if every two vertices are reachable from each other.*

2. Tree

2.1 Free trees

2.1.1 Definition

> *A free tree is a "connected", "acyclic", "undirected" graph*

2.1.2 Properties

Let $G = (V, E)$ be a free tree.

- Any two vertices in G are connected by a unique simple path.
- G is connected.

- If any edge is removed, it will become disconnected.
- G is acyclic.
- If any edge is added, it will contain a cycle.
- $|E| = |V| - 1$.

3. Elementary Graph Algorithms

3.1 BFS

```
BFS(V, E, s):
    for u in V - s:
        d[u] = inf
    d[s] = 0
    enqueue(Q, s)
    while Q is not empty
        u = dequeue(Q)
        for v in Adj[u]:
            if d[v] = inf
                d[v] = d[u] + 1
                enqueue(Q, v)
```

Time cost is $O(V + E)$:

- Initialization: $O(V)$
- Scanning adjacent lists: $O(E)$
- Operations on queue: $O(V)$

3.2 DFS

```
DFS(V, E):
    for u in V:
        color[u] = white
    time = 0
    for u in V:
        if color[u] == white:
            DFS_VISIT(u)
DFS_VISIT(u):
    color[u] = gray
    time = time + 1
    d[u] = time
    for v in Adj[u]:
```

```
        if color[u] == white:
            DFS_VISIT(v)
    color[u] = black
    time = time + 1
    f[u] = time
```

Time cost is $\Theta(V + E)$:

- Initialization: $\Theta(V)$
- Calling recursion: $\Theta(V)$
- DFS_VISIT: $\Theta(E)$

3.3 Topological Sort

```
TOPO_SORT(V, E):
    DFS(v, E)
    return vertices in decreasing order of finishing time
```

Time cost is $\Theta(V + E)$

4. Minimum Spanning Trees

4.1 Prim

Adding the minimum "accessible" edge to the spanning tree each time.

- $w(u, v)$ is the weight between vertice u and vertice v.
- r is the root of the spanning tree.

```
PRIM(V, E, w, r):
    for u in V:
        key[u] = inf
        parent[u] = nill
        insert(Q, u)
    assign_key(r, 0)
    while Q is not empty:
        u = extract_min(Q) based on key
        for v in Adj[u]:
            if v in Q and w[u,v] < key[v]:
                parent[v] = u
                assign_key(v, w[u,v])
```

If Q is a binary heap, the time cost would be $O(E \log V)$:

- $|V| \cdot$ extract_min $\Rightarrow O(V \log V)$.
- $|E| \cdot$ assign_key $\Rightarrow O(E \log V)$.

If Q is a fibonacci heap, the time cost would be $O(V \log V + E)$:

- $|V| \cdot$ extract_min $\Rightarrow O(V \log V)$.
- $|E| \cdot$ assign_key $\Rightarrow O(E)$.

5. Shortest Path

```
INIT_SINGLE_SOURCE(V, s):
    for v in V:
        d[v] = inf
        parent[v] = nill
    d[s] = 0
```

```
RELAX(u, v, w):
    if d[v] > d[u] + w[u,v]:
        d[v] = d[u] + w[u,v]
        parent[v] = u
```

5.1 Bellman-Ford

It is OK with negative weights.

The algorithm is to relax all the edges in $G.E$ for $|G.V| - 1$ times.

It is edge-based.

```
BELLMAN_FORD(G, w, s):
    INIT_SINGLE_SOURCE(G, s)
    for i = 1 to |G.V| - 1:
        for edge(u,v) in G.E:
            RELAX(u, v, w)
    for each edge(u,v) in G.E:
        if d[v] > d[u] + w(u,v):
            return False
    return True
```

The time cost would be $O(VE)$.

5.2 Dijkstra

It is a greedy algorithm with no negative weights allowed.

The algorithm each time selects the vertice in $Q - S$ with the minimum $d[v]$ and relaxes all the edges to its neighbors.

It is vertice-based.

```
DIJKSTRA(G, w, s):
    INIT_SINGLE_SOURCE(G, s)
    S = {}
    Q = G.V
    while Q is not empty:
        u = extract_min(Q) based on d[]
        S = S + {u}
        for v in Adj[u]:
            RELAX(u, v, w)
```

Like Prim, if binary queue is used, the time cost would be $O(E \log V)$.

```
DIJKSTRA(Matrix, s_idx):
```

```
    S = [s_idx]
    Q = [x for x in range(len(matrix)) if x != s_idx]
    dis = Matrix[s_idx]
    while Q:
        # 找出Q中d最小的下标
        min_idx = Q[0]
        for i in Q:
            if dis[i] < dis[min_idx]:
                min_idx = i
        S.append(min_idx)
        Q.remove(min_idx)
        # 更新距离
        for i in Q:
            dis[i] = min(dis[i], dis[min_idx] + Matrix[min_idx]
[i])
    return dis
```

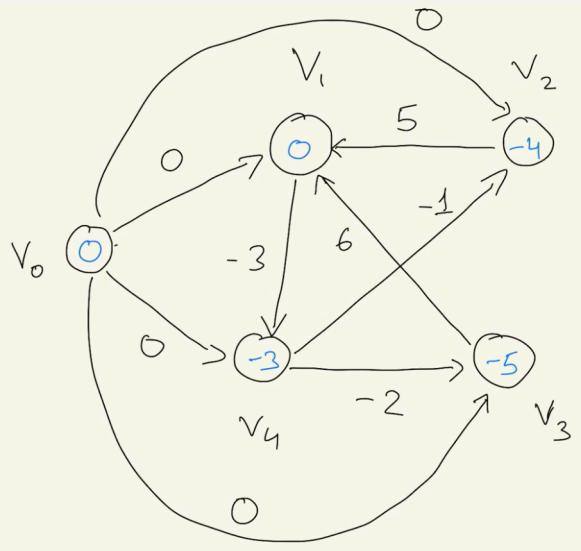
Reference: https://www.zhihu.com/question/20630094/answer/758191548

5.3 Difference Constraints

Example:

$$
\begin{aligned}
x_1 - x_2 &\leq 5 \\
x_1 - x_3 &\leq 6 \\
x_2 - x_4 &\leq -1 \\
x_3 - x_4 &\leq -2 \\
x_4 - x_1 &\leq -3
\end{aligned}
$$

The graph would be:



Run Bellman-Ford:

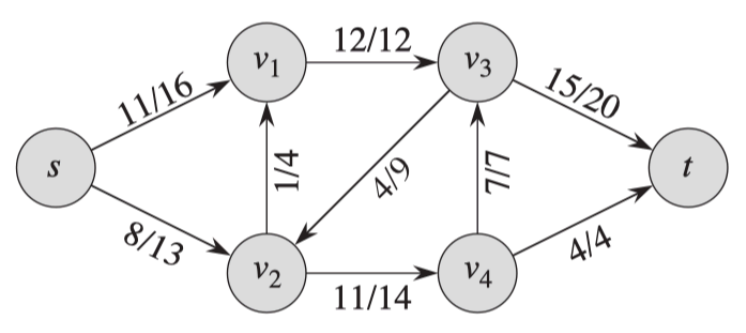
- If there is a negative weight cycle $\Rightarrow$ No solution
- Otherwise, output the solution

The time cost would be:

$O(m+n)$ to build the graph + $O(V \cdot E) = O((n+1) \cdot (m+n))$ to run the Bellman-Ford = $O(n^2 + mn)$

6. Maximum Flow

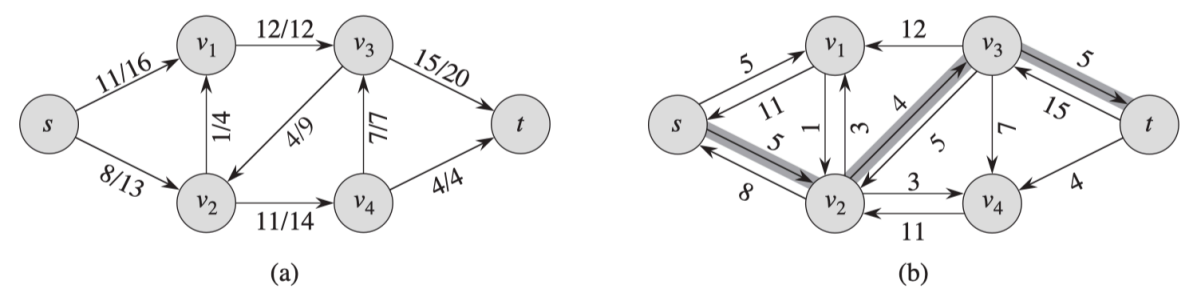
6.1 Definition



- Capacity constraint: For all , we require .
- Flow conservation: For all , we require .

The maximum flow problem is to find the maximum value .

6.2 Residual Network



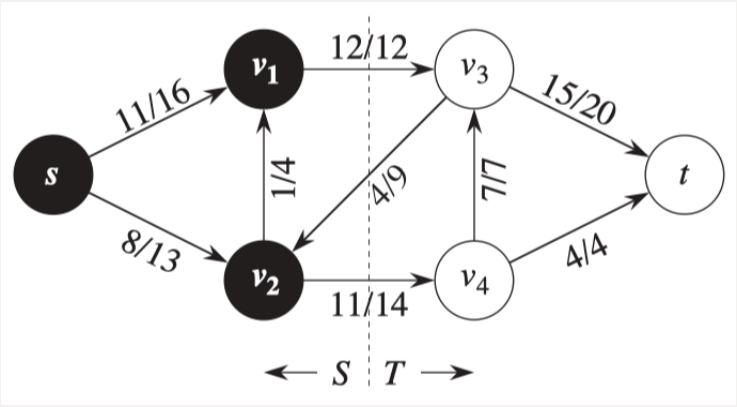
is the amount of additional flow that can pass through edge .

- Augmenting path a simple path from to in .

Intuition: AP is a sequence of edges where capacity exceeds existing flow and more flow can be pushed in.

The residual capacity of augmenting path is:

- Cut is a partition of the vertices in a disjoint set such that and , .



In this case, while

- Max flow min cut theorem: they are equivalent

1. is a maximum flow in .
2. The residual network contains no augmenting paths.
3. for some cut of .

6.3 Ford Fulkerson Algorithm

- Initialize for all the .
- While augmenting path in the residual network :

  Increase flow in by adding , the residual capacity of .

```
FORD_FULKERSON(G,s,t):
    for (u,v) in E:
        f(u,v) = 0
    while augmenting path p in residual network Gf:
        cf(p) = min{cf(u,v):(u,v) is in p}
        for (u,v) in p:
            if (u,v) in E:
                f(u,v) = f(u,v) + cf(p)
            else:
                f(v,u) = f(v,u) - cf(p)
```

Suppose we use BFS to look for an augmenting path, which takes , the running time would be

6.4 Edmonds Karp Algorithm

Always choose the augmenting path with the minimum number of edges (shortest path).

The running time would be.

Chapter 9: NP-Complete Problems

---

1. P

> *It is the class of problems that can be solved in polynomial time.*

2. NP

> *It is the class of problems that can be verified in polynomial time given a certificate.*

3. NPC

Language $L \in \text{NPC}$ if and only if:

- $L \in \text{NP}$
- $\forall L' \in \text{NP} \rightarrow L' \leq_p L$ ($L \in$ NP-hard)

Methodology to prove NPC:

(a) Prove that a solution can be verified in polynomial time.

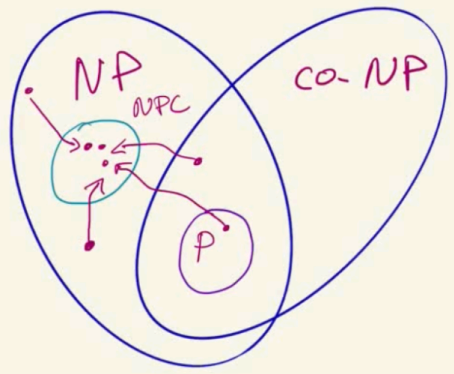
(b) Select known $L' \in \text{NPC}$

(b1) Describe algorithm $f()$ that given instance $x \in L'$, it translates it to $f(x)$ such that $(x \in L') \leq_p (f(x) \in L)$.

(b2) Show $f()$ takes polynomial time to compute.

4. Euler Diagram

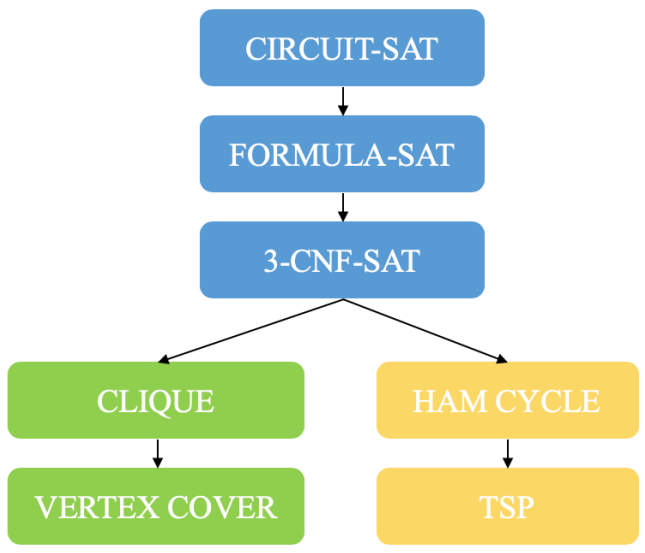
The relationship among class P, class NP and class NPC is:



5. Reducibility

We say that $A \leq_p B$ (A is polynomially reducible to B) if and only if there is an polynomial-time algorithm that given an instance of A, we can transform it to some instance of B, solve B and then obtain a solution for A in polynomial time.

6. NP-Complete Problems



Appendix

1. Summations

1.1 Arithmetic series

> $\sum_{k=1}^{n} k = \frac{n(n+1)}{2} = \Theta(n^2)$

1.2 Geometric series

> $\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1}-1}{x-1}$

1.3 Infinite series

> $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \quad if \quad |x| < 1$

Example: prove $\sum_{k=0}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2}$ when $|x| < 1$

Answer:

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

$$\text{Differentiate both sides} \Leftrightarrow \sum_{k=0}^{\infty} k \cdot x^{k-1} = \frac{1}{(1-x)^2}$$

$$\text{Multiply both sides by x} \Leftrightarrow \sum_{k=0}^{\infty} k \cdot x^{k} = \frac{x}{(1-x)^2}$$

1.4 Telescoping

> $\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_0$

> $\sum_{k=0}^{n-1} (a_k - a_{k+1}) = a_0 - a_n$

Example:

$\sum_{k=1}^{n-1} \frac{1}{k \cdot (k+1)} = \sum_{k=1}^{n} (\frac{1}{k} - \frac{1}{k+1}) = 1 - \frac{1}{n}$

1.5 Binomial coefficient

> $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
>
> $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$

2. Counting

2.1 Permutation

Case 1

> *$P(n,r)$ is the number of ways to arrange r out of n distinct objects where the order is important.*

$$P(n,r) = \frac{n!}{(n-r)!}$$

Example:

In how many ways n-people can sit to from a ring ?

Answer:

$P(n,n)/n = (n-1)!$ ways.

Case 2

> *If not all objects are distinct but ($q_1 + q_2, \ldots, + q_t = r$)*
>
> *1^{st} kind: q_1 objects,*
>
> *2^{nd} kind: q_2 objects,*
>
> *.....*
>
> *t^{th} kind: q_t objects*

$$P(n,r) = \frac{n!}{q_1! \cdot q_2! \cdot \ldots \cdot q_t!}$$

Example 1:

how many ways that 5 dashes and 8 dots can be arranged into ?

Answer:

$13!/(5! \cdot 8!)$ ways.

Example 2:

Prove that $(k!)!$ is disivible by $(k!)^{(k-1)!}$.

Answer:

$\Rightarrow$ Use a combinatorial argument

Consider $k!$ objects as follows:

1^{st} kind: k objects,

2^{nd} kind: k objects,

.....

$(k-1)!$ kind: k objects

So, the number ways to permute them is:

$$\frac{(k!)!}{k! \cdot k! \cdot \ldots \cdot k!} = \frac{(k!)!}{(k!)^{(k-1)!}}$$

Therefore, QED.

2.2 Combination

> *$C(n,r)$ is the number of ways to arrange r out of n distinct objects where the order is not important.*

$$C(n,r) = \frac{P(n,r)}{r!} = \frac{n!}{r! \cdot (n-r)!}$$

Example 1:

How many diagonals a decagon has ?

Answer:

$C(10, 2) - 10 = 35$ diagonals.

Example 2:

In how many ways can three numbers can be selected from 1 to 300 so that their unique sum is divisible by 3 ?

Answer:

$C(100, 3) + C(100, 3) + C(100, 3) + 100^3 = 1485100$ ways.

Example 3:

11 scientists work on a secret project. They lock the project documents into a cabinet so that the cabinet can be opened if and only if at least 6 scientists are present with their keys.

(a) What is the smallest number of locks needed ?

$\Rightarrow$ For $\forall$ group of 5 scientists, $\exists$ 1 lock that they don't have.

$\because$ There are totally $C(11, 5)$ ways to form a group of 5 scientists.

$\therefore C(11, 5)$ locks are needed in total.

(b) What is the smallest number of keys each scientist needs to have ?

$\Rightarrow$ For $\forall$ group of 5 other scientists, $\exists$ 1 lock that one has but they don't.

$\because$ There are totally $C(10, 5)$ ways to form a group of 5 other scientists.

$\therefore$ Each scientist needs to have $C(10, 5)$ keys.

3. Probability

3.1 Probability axioms

> - $P(a \in S) \geq 0$
> - $P(S) = 1$
> - $P(A \cup B) = P(A) + P(B) - P(A \cap B)$
> - $P(A \cap B) = P(A) \cdot P(B)$ *if* A *and* B *are independent*
> - *If* $\forall$ *events* $a \in S$*, we have* $P(a) = 1/|S|$*, then it has a Uniform Probability Distribution*

Example:

What is the probability of having exactly k heads when flipping a fair coin n times.

Answer:

$P = C(n, k)/2^n$

3.2 Bayes theorem

> $$P(A|B) = \frac{P(A) \cdot P(B|A)}{P(B)} = \frac{P(A) \cdot P(B|A)}{P(A) \cdot P(B|A) + P(\overline{A}) \cdot P(B|\overline{A})}$$

Example:

We have two coins. One is fair, the other one is biased and brings H all the time.

We filp one coin twice and have HH. What is the probability of picking the biased one ?

Answer:

Assume picking the biased coin as event A and having HH as event B.

$$\begin{aligned} P(A|B) &= \frac{P(B|A) \cdot P(A)}{P(A) \cdot P(B|A) + P(\overline{A}) \cdot P(B|\overline{A})} \\ &= \frac{1 \cdot 0.5}{0.5 \cdot 1 + 0.5 \cdot 0.25} \\ &= 0.8 \end{aligned}$$

3.3 Discrete random variables

Expected value

> $$E[X] = \sum_{x \in X} x \cdot P(X = x)$$

- $E[X + Y] = E[X] + E[Y]$
- $E[aX] = aE[X]$
- $E[X \cdot Y] = E[X] \cdot E[Y]$ if X and Y are independent

Variance

> $$Var[X] = E[(X - E[X])^2] = E[X^2] - E^2[X]$$

3.4 Discrete probability distribution

Bernoulli trial

> *A random experiment with only two possible outcomes.*

Geometric distribution

> *The discrete probability distribution of the number X of Bernoulli trials needed to have a success.*

> $$P(X = k) = (1 - p)^{(k-1)} \cdot p$$
>
> $$E[X] = \sum_{k=1}^{\infty} k \cdot (1 - p)^{(k-1)} \cdot p = 1/p$$
>
> $$Var[X] = (1 - p)/p^2$$

Binomial distribution

> *The discrete probability distribution of the number X of successes over n Bernoulli trials.*

> $$P(X = k) = C(n, k) \cdot p^k \cdot (1 - p)^{n-k}$$
>
> $$E[X] = \sum_{k=0}^{n} k \cdot P(X = k) = np$$
>
> $$Var[X] = n \cdot p \cdot (1 - p)$$

4. Logarithms

4.1 Definition

> $a = b^c \quad \Leftrightarrow \quad log_b a = c$

4.2 Properties

> $log_b a = \frac{1}{log_a b}$

> $a^{log_b n} = n^{log_b a}$

The iterated logarithm function:

> $lg^{(i)}(n) = \begin{cases} n & \text{if } i = 0 \\ lg\left(lg^{(i-1)}(n)\right) & \text{if } i > 0 \end{cases}$
>
> > $\lg^* n = \min\left\{i \geq 0 : \lg^{(i)} n \leq 1\right\}$
> >
> > $lg^* 2 = 1$
> >
> > $lg^* 2^2 = 1 + lg^* 2 = 2$
> >
> > $lg^* 2^{2^2} = 1 + lg * 2^2 = 3$
> >
> > $lg^* 2^{2^{2^2}} = 1 + lg^* 2^{2^2} = 4$
> >
> > $\cdots$

5. Induction

Example

Prove that the sum of the first n odd positive integers is n^2

Basis:

When $n = 1$, $1 = 1^2$.

Hypothesis:

Assume $n = k$, it works: $1 + 3 + 5 + \cdots + (2k - 1) = k^2$

Inductive Step:

When $n = k + 1$:

$1 + 3 + 5 + \cdots + (2k - 1) + (2k + 1) = k^2 + (2k + 1) = (k + 1)^2$

Therefore, the statement is proved. Q.E.D.

6. Contradiction

Example

Show that $\sqrt{2}$ is irrational

Assume towards a contradiction that $\sqrt{2}$ is rational.

$\Rightarrow \sqrt{2} = a/b$, where a and b have no common factors.

$\Rightarrow a^2 = 2b^2$.

$\Rightarrow a^2$ is even $\Rightarrow^{(*)}$ a is even and suppose $a = 2c$.

$\Rightarrow b^2 = 2c$.

$\Rightarrow b^2$ is even $\Rightarrow^{(*)}$ b is even.

This is a contradiction because we assume that a and b have no common factors.

Proove: $(*)$

Assume towards a contradiction that a^2 is even $\Rightarrow a$ is odd.

Suppose $a = 2k + 1$

$\Rightarrow a^2 = (2k + 1)^2 = 4k^2 + 2k + 1 = 2(2k^2 + k) + 1$, which is odd.

This is a contradiction because we assume that a^2 is even $\Rightarrow a$ is odd.