Algoritmos

Maratona de Programação

Sumário

Capítulo 1

Introdução

1.1 Bugs do Milênio

Erros teóricos:

- Não ler o enunciado do problema com calma.
- Assumir algum fato sobre a solução na pressa.
- Não reler os limites do problema antes de submeter.
- Quando adaptar um algoritmo, atentar para todos os detalhes da estrutura do algoritmo, se devem (ou não) ser modificados (ex:marcação de vértices/estados).
- O problema pode ser NP, disfarçado ou mesmo sem limites especificados. Nesse caso a solução é bronca mesmo. Não é hora de tentar ganhar o prêmio nobel.

Erros com valor máximo de variável:

- Verificar com calma (fazer as contas direito) para ver se o infinito é tão infinito quanto parece.
- Verificar se operações com infinito estouram 31 bits.
- Usar multiplicacao de *int*'s e estourar 32 bits (por exemplo, checar sinais usando $a * b > 0$).

Erros de casos extremos:

- Testou caso $n = 0$? $n = 1$? $n = MAXN$? Muitas vezes tem que tratar separado.
- Pense em todos os casos que podem ser considerados casos extremos ou casos isolados.
- Casos extremos podem atrapalhar não só no algoritmo, mas em coisas como construir alguma estrutura (ex: lista de adj em grafos).
- Não esquecer de self-loops ou multiarestas em grafos.
- Em problemas de caminho Euleriano, verificar se o grafo é conexo.

Erros de desatenção em implementação:

- Errar ctrl-C/ctrl-V em código. Muito comum.
- Colocar igualdade dentro de *if*? ($if(a = 0)continue$;)
- Esquecer de inicializar variável.
- Trocar *break* por *continue* (ou vice-versa).
- Declarar variável global e variável local com mesmo nome (é pedir pra dar merda...).

Erros de implementação:

- Definir variável com tipo errado (*int* por *double*, *int* por *char*).
- Não usar variável com nome *max* e *min*.
- Não esquecer que $.size()$ é unsigned.
- Lembrar que 1 é *int*, ou seja, se fizer $long\ long\ a = 1 << 40$;, não irá funcionar (o ideal é fazer $long\ long\ a = 1LL << 40$;).

Erros em limites:

- Qual o ordem do tempo e memória? 10^8 é uma referência para tempo. Sempre verificar rapidamente a memória, apesar de que o limite costuma ser bem grande.
- A constante pode ser muito diminuída com um algoritmo melhor (ex: húngaro no lugar de fluxo) ou com operações mais rápidas (ex: divisões são lentas, bitwise é rápido)?
- O exercício é um caso particular que pode (e está precisando) ser otimizado e não usar direto a biblioteca?

Erros em doubles:

- Primeiro, evitar (a não ser que seja necessário ou mais simples a solução) usar *float/double*. E.g. conta que só precisa de 2 casas decimais pode ser feita com inteiro e depois %100.
- Sempre usar *double*, não *float* (a não ser que o enunciado peça explicitamente).
- Testar igualdade com tolerância (absoluta, e talvez relativa).
- Cuidado com erros de imprecisão, em particular evitar ao máximo subtrair dois números praticamente iguais.

Outros erros:

- Evitar (a não ser que seja necessário) alocação dinâmica de memória.

- Não usar STL desnecessariamente (ex: vector quando um array normal dá na mesma), mas usar se facilitar (ex: nomes associados a vértices de um grafo - $map < string, int >$) ou se precisar (ex: um algoritmo $O(nlogn)$ que usa $< set >$ é necessário para passar no tempo).
- Não inicializar variável a cada teste (zerou vetores? zerou variável que soma algo? zerou com zero? era pra zerar com zero, com -1 ou com INF?).
- Saída está formatada corretamente?
- Declarou vetor com tamanho suficiente?
- Cuidado ao tirar o módulo de número negativo. Ex.: $x\%n$ não dá o resultado esperado se x é negativo, fazer $(x\%n + n)\%n$.

1.2 Recomendações gerais

Cortesia da PUC-RJ.

ANTES DA PROVA

- Revisar os algoritmos disponíveis na biblioteca.
- Revisar a referência STL.
- Reler este roteiro.
- Ouvir o discurso motivacional do técnico.

ANTES DE IMPLEMENTAR UM PROBLEMA

- Quem for implementar deve relê-lo antes.
- Peça todas as clarifications que forem necessárias.
- Marque as restrições e faça contas com os limites da entrada.
- Teste o algoritmo no papel e convença outra pessoa de que ele funciona.
- Planeje a resolução para os problemas grandes: a equipe se junta para definir as estruturas de dados, mas cada pessoa escreve uma função.

DEBUGAR UM PROGRAMA

- Ao encontrar um bug, escreva um caso de teste que o dispare.
- Reimplementar trechos de programas entendidos errados.
- Em caso de RE, procure todos os [, / e %.

1.3 Os 1010 mandamentos

Também cortesia da PUC-RJ.

0. Não dividirás por zero.
1. Não alocarás dinamicamente.
2. Compararás números de ponto flutuante usando EPS.
3. Verificarás se o grafo pode ser desconexo.
4. Verificarás se as arestas do grafo podem ter peso negativo.
5. Verificarás se pode haver mais de uma aresta ligando dois vértices.
6. Conferirás todos os índices de uma programação dinâmica.
7. Reduzirás o branching factor da DFS.
8. Farás todos os cortes possíveis em uma DFS.
9. Tomarás cuidado com pontos coincidentes e com pontos colineares.

1.4 Limites da representação de dados

tipo	bits	mínimo	..	máximo	precisão decimal
char	8	0	..	127	2
signed char	8	-128	..	127	2
unsigned char	8	0	..	255	2
short	16	-32.768	..	32.767	4
unsigned short	16	0	..	65.535	4
int	32	-2×10^9	..	2×10^9	9
unsigned int	32	0	..	4×10^9	9
long long	64	-9×10^{18}	..	9×10^{18}	18
unsigned long long	64	0	..	18×10^{18}	19

tipo	bits	expoente	precisão decimal
float	32	38	6
double	64	308	15
long double	80	19.728	18

1.5 Quantidade de números primos de 1 até 10^n

É sempre verdade que $n/ln(n) < pi(n) < 1.26 * n/ln(n)$.

$pi(10^1) = 4$	$pi(10^2) = 25$	$pi(10^3) = 168$
$pi(10^4) = 1.229$	$pi(10^5) = 9.592$	$pi(10^6) = 78.498$
$pi(10^7) = 664.579$	$pi(10^8) = 5.761.455$	$pi(10^9) = 50.847.534$

1.6 Triângulo de Pascal

$n \backslash p$	0	1	2	3	4	5	6	7	8	9	10
0	1										
1	1	1									
2	1	2	1								
3	1	3	3	1							
4	1	4	6	4	1						
5	1	5	10	10	5	1					
6	1	6	15	20	15	6	1				
7	1	7	21	35	35	21	7	1			
8	1	8	28	56	70	56	28	8	1		
9	1	9	36	84	126	126	84	36	9	1	
10	1	10	45	120	210	252	210	120	45	10	1

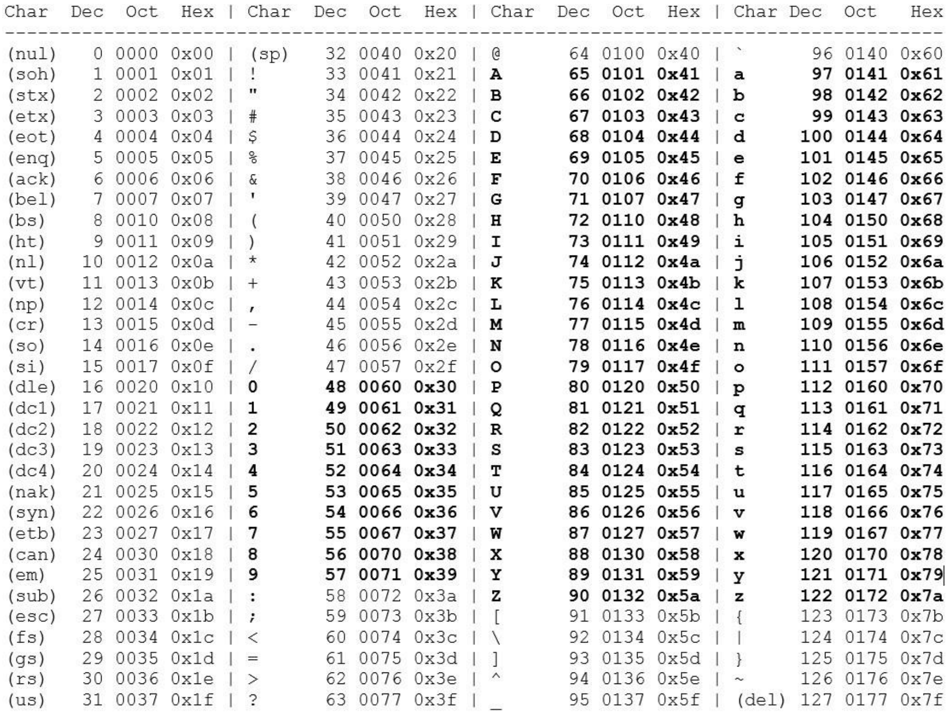
$C(33, 16)$	1.166.803.110	limite do int
$C(34, 17)$	2.333.606.220	limite do unsigned int
$C(66, 33)$	7.219.428.434.016.265.740	limite do long long
$C(67, 33)$	14.226.520.737.620.288.370	limite do unsigned long long

1.7 Fatoriais

Fatoriais até 20 com os limites de tipo.

0!	1	
1!	1	
2!	2	
3!	6	
4!	24	
5!	120	
6!	720	
7!	5.040	
8!	40.320	
9!	362.880	
10!	3.628.800	
11!	39.916.800	
12!	479.001.600	limite do unsigned int
13!	6.227.020.800	
14!	87.178.291.200	
15!	1.307.674.368.000	
16!	20.922.789.888.000	
17!	355.687.428.096.000	
18!	6.402.373.705.728.000	
19!	121.645.100.408.832.000	
20!	2.432.902.008.176.640.000	limite do unsigned long long

1.8 Tabela ASCII



```
Char   Dec  Oct  Hex | Char   Dec  Oct  Hex | Char   Dec  Oct  Hex | Char  Dec  Oct   Hex
------------------------------------------------------------------------------------------
(nul)     0 0000 0x00 | (sp)     32 0040 0x20 | @        64 0100 0x40 | `        96 0140 0x60
(soh)     1 0001 0x01 | !        33 0041 0x21 | A        65 0101 0x41 | a        97 0141 0x61
(stx)     2 0002 0x02 | "        34 0042 0x22 | B        66 0102 0x42 | b        98 0142 0x62
(etx)     3 0003 0x03 | #        35 0043 0x23 | C        67 0103 0x43 | c        99 0143 0x63
(eot)     4 0004 0x04 | $        36 0044 0x24 | D        68 0104 0x44 | d       100 0144 0x64
(enq)     5 0005 0x05 | %        37 0045 0x25 | E        69 0105 0x45 | e       101 0145 0x65
(ack)     6 0006 0x06 | &        38 0046 0x26 | F        70 0106 0x46 | f       102 0146 0x66
(bel)     7 0007 0x07 | '        39 0047 0x27 | G        71 0107 0x47 | g       103 0147 0x67
(bs)      8 0010 0x08 | (        40 0050 0x28 | H        72 0110 0x48 | h       104 0150 0x68
(ht)      9 0011 0x09 | )        41 0051 0x29 | I        73 0111 0x49 | i       105 0151 0x69
(nl)     10 0012 0x0a | *        42 0052 0x2a | J        74 0112 0x4a | j       106 0152 0x6a
(vt)     11 0013 0x0b | +        43 0053 0x2b | K        75 0113 0x4b | k       107 0153 0x6b
(np)     12 0014 0x0c | ,        44 0054 0x2c | L        76 0114 0x4c | l       108 0154 0x6c
(cr)     13 0015 0x0d | -        45 0055 0x2d | M        77 0115 0x4d | m       109 0155 0x6d
(so)     14 0016 0x0e | .        46 0056 0x2e | N        78 0116 0x4e | n       110 0156 0x6e
(si)     15 0017 0x0f | /        47 0057 0x2f | O        79 0117 0x4f | o       111 0157 0x6f
(dle)    16 0020 0x10 | 0        48 0060 0x30 | P        80 0120 0x50 | p       112 0160 0x70
(dc1)    17 0021 0x11 | 1        49 0061 0x31 | Q        81 0121 0x51 | q       113 0161 0x71
(dc2)    18 0022 0x12 | 2        50 0062 0x32 | R        82 0122 0x52 | r       114 0162 0x72
(dc3)    19 0023 0x13 | 3        51 0063 0x33 | S        83 0123 0x53 | s       115 0163 0x73
(dc4)    20 0024 0x14 | 4        52 0064 0x34 | T        84 0124 0x54 | t       116 0164 0x74
(nak)    21 0025 0x15 | 5        53 0065 0x35 | U        85 0125 0x55 | u       117 0165 0x75
(syn)    22 0026 0x16 | 6        54 0066 0x36 | V        86 0126 0x56 | v       118 0166 0x76
(etb)    23 0027 0x17 | 7        55 0067 0x37 | W        87 0127 0x57 | w       119 0167 0x77
(can)    24 0030 0x18 | 8        56 0070 0x38 | X        88 0130 0x58 | x       120 0170 0x78
(em)     25 0031 0x19 | 9        57 0071 0x39 | Y        89 0131 0x59 | y       121 0171 0x79
(sub)    26 0032 0x1a | :        58 0072 0x3a | Z        90 0132 0x5a | z       122 0172 0x7a
(esc)    27 0033 0x1b | ;        59 0073 0x3b | [        91 0133 0x5b | {       123 0173 0x7b
(fs)     28 0034 0x1c | <        60 0074 0x3c | \        92 0134 0x5c | |       124 0174 0x7c
(gs)     29 0035 0x1d | =        61 0075 0x3d | ]        93 0135 0x5d | }       125 0175 0x7d
(rs)     30 0036 0x1e | >        62 0076 0x3e | ^        94 0136 0x5e | ~       126 0176 0x7e
(us)     31 0037 0x1f | ?        63 0077 0x3f | _        95 0137 0x5f | (del)   127 0177 0x7f
```

1.9 Primos até 10.000

Existem 1.229 números primos até 10.000.

2	3	5	7	11	13	17	19	23	29	31
37	41	43	47	53	59	61	67	71	73	79
83	89	97	101	103	107	109	113	127	131	137
139	149	151	157	163	167	173	179	181	191	193
197	199	211	223	227	229	233	239	241	251	257
263	269	271	277	281	283	293	307	311	313	317
331	337	347	349	353	359	367	373	379	383	389
397	401	409	419	421	431	433	439	443	449	457
461	463	467	479	487	491	499	503	509	521	523
541	547	557	563	569	571	577	587	593	599	601
607	613	617	619	631	641	643	647	653	659	661
673	677	683	691	701	709	719	727	733	739	743
751	757	761	769	773	787	797	809	811	821	823
827	829	839	853	857	859	863	877	881	883	887
907	911	919	929	937	941	947	953	967	971	977
983	991	997	1009	1013	1019	1021	1031	1033	1039	1049
1051	1061	1063	1069	1087	1091	1093	1097	1103	1109	1117
1123	1129	1151	1153	1163	1171	1181	1187	1193	1201	1213
1217	1223	1229	1231	1237	1249	1259	1277	1279	1283	1289
1291	1297	1301	1303	1307	1319	1321	1327	1361	1367	1373
1381	1399	1409	1423	1427	1429	1433	1439	1447	1451	1453
1459	1471	1481	1483	1487	1489	1493	1499	1511	1523	1531
1543	1549	1553	1559	1567	1571	1579	1583	1597	1601	1607
1609	1613	1619	1621	1627	1637	1657	1663	1667	1669	1693
1697	1699	1709	1721	1723	1733	1741	1747	1753	1759	1777
1783	1787	1789	1801	1811	1823	1831	1847	1861	1867	1871
1873	1877	1879	1889	1901	1907	1913	1931	1933	1949	1951
1973	1979	1987	1993	1997	1999	2003	2011	2017	2027	2029
2039	2053	2063	2069	2081	2083	2087	2089	2099	2111	2113
2129	2131	2137	2141	2143	2153	2161	2179	2203	2207	2213
2221	2237	2239	2243	2251	2267	2269	2273	2281	2287	2293
2297	2309	2311	2333	2339	2341	2347	2351	2357	2371	2377
2381	2383	2389	2393	2399	2411	2417	2423	2437	2441	2447
2459	2467	2473	2477	2503	2521	2531	2539	2543	2549	2551
2557	2579	2591	2593	2609	2617	2621	2633	2647	2657	2659
2663	2671	2677	2683	2687	2689	2693	2699	2707	2711	2713
2719	2729	2731	2741	2749	2753	2767	2777	2789	2791	2797
2801	2803	2819	2833	2837	2843	2851	2857	2861	2879	2887
2897	2903	2909	2917	2927	2939	2953	2957	2963	2969	2971
2999	3001	3011	3019	3023	3037	3041	3049	3061	3067	3079
3083	3089	3109	3119	3121	3137	3163	3167	3169	3181	3187
3191	3203	3209	3217	3221	3229	3251	3253	3257	3259	3271
3299	3301	3307	3313	3319	3323	3329	3331	3343	3347	3359
3361	3371	3373	3389	3391	3407	3413	3433	3449	3457	3461
3463	3467	3469	3491	3499	3511	3517	3527	3529	3533	3539
3541	3547	3557	3559	3571	3581	3583	3593	3607	3613	3617
3623	3631	3637	3643	3659	3671	3673	3677	3691	3697	3701
3709	3719	3727	3733	3739	3761	3767	3769	3779	3793	3797
3803	3821	3823	3833	3847	3851	3853	3863	3877	3881	3889
3907	3911	3917	3919	3923	3929	3931	3943	3947	3967	3989
4001	4003	4007	4013	4019	4021	4027	4049	4051	4057	4073
4079	4091	4093	4099	4111	4127	4129	4133	4139	4153	4157

4159	4177	4201	4211	4217	4219	4229	4231	4241	4243	4253
4259	4261	4271	4273	4283	4289	4297	4327	4337	4339	4349
4357	4363	4373	4391	4397	4409	4421	4423	4441	4447	4451
4457	4463	4481	4483	4493	4507	4513	4517	4519	4523	4547
4549	4561	4567	4583	4591	4597	4603	4621	4637	4639	4643
4649	4651	4657	4663	4673	4679	4691	4703	4721	4723	4729
4733	4751	4759	4783	4787	4789	4793	4799	4801	4813	4817
4831	4861	4871	4877	4889	4903	4909	4919	4931	4933	4937
4943	4951	4957	4967	4969	4973	4987	4993	4999	5003	5009
5011	5021	5023	5039	5051	5059	5077	5081	5087	5099	5101
5107	5113	5119	5147	5153	5167	5171	5179	5189	5197	5209
5227	5231	5233	5237	5261	5273	5279	5281	5297	5303	5309
5323	5333	5347	5351	5381	5387	5393	5399	5407	5413	5417
5419	5431	5437	5441	5443	5449	5471	5477	5479	5483	5501
5503	5507	5519	5521	5527	5531	5557	5563	5569	5573	5581
5591	5623	5639	5641	5647	5651	5653	5657	5659	5669	5683
5689	5693	5701	5711	5717	5737	5741	5743	5749	5779	5783
5791	5801	5807	5813	5821	5827	5839	5843	5849	5851	5857
5861	5867	5869	5879	5881	5897	5903	5923	5927	5939	5953
5981	5987	6007	6011	6029	6037	6043	6047	6053	6067	6073
6079	6089	6091	6101	6113	6121	6131	6133	6143	6151	6163
6173	6197	6199	6203	6211	6217	6221	6229	6247	6257	6263
6269	6271	6277	6287	6299	6301	6311	6317	6323	6329	6337
6343	6353	6359	6361	6367	6373	6379	6389	6397	6421	6427
6449	6451	6469	6473	6481	6491	6521	6529	6547	6551	6553
6563	6569	6571	6577	6581	6599	6607	6619	6637	6653	6659
6661	6673	6679	6689	6691	6701	6703	6709	6719	6733	6737
6761	6763	6779	6781	6791	6793	6803	6823	6827	6829	6833
6841	6857	6863	6869	6871	6883	6899	6907	6911	6917	6947
6949	6959	6961	6967	6971	6977	6983	6991	6997	7001	7013
7019	7027	7039	7043	7057	7069	7079	7103	7109	7121	7127
7129	7151	7159	7177	7187	7193	7207	7211	7213	7219	7229
7237	7243	7247	7253	7283	7297	7307	7309	7321	7331	7333
7349	7351	7369	7393	7411	7417	7433	7451	7457	7459	7477
7481	7487	7489	7499	7507	7517	7523	7529	7537	7541	7547
7549	7559	7561	7573	7577	7583	7589	7591	7603	7607	7621
7639	7643	7649	7669	7673	7681	7687	7691	7699	7703	7717
7723	7727	7741	7753	7757	7759	7789	7793	7817	7823	7829
7841	7853	7867	7873	7877	7879	7883	7901	7907	7919	7927
7933	7937	7949	7951	7963	7993	8009	8011	8017	8039	8053
8059	8069	8081	8087	8089	8093	8101	8111	8117	8123	8147
8161	8167	8171	8179	8191	8209	8219	8221	8231	8233	8237
8243	8263	8269	8273	8287	8291	8293	8297	8311	8317	8329
8353	8363	8369	8377	8387	8389	8419	8423	8429	8431	8443
8447	8461	8467	8501	8513	8521	8527	8537	8539	8543	8563
8573	8581	8597	8599	8609	8623	8627	8629	8641	8647	8663
8669	8677	8681	8689	8693	8699	8707	8713	8719	8731	8737
8741	8747	8753	8761	8779	8783	8803	8807	8819	8821	8831
8837	8839	8849	8861	8863	8867	8887	8893	8923	8929	8933
8941	8951	8963	8969	8971	8999	9001	9007	9011	9013	9029
9041	9043	9049	9059	9067	9091	9103	9109	9127	9133	9137
9151	9157	9161	9173	9181	9187	9199	9203	9209	9221	9227
9239	9241	9257	9277	9281	9283	9293	9311	9319	9323	9337
9341	9343	9349	9371	9377	9391	9397	9403	9413	9419	9421
9431	9433	9437	9439	9461	9463	9467	9473	9479	9491	9497
9511	9521	9533	9539	9547	9551	9587	9601	9613	9619	9623
9629	9631	9643	9649	9661	9677	9679	9689	9697	9719	9721
9733	9739	9743	9749	9767	9769	9781	9787	9791	9803	9811
9817	9829	9833	9839	9851	9857	9859	9871	9883	9887	9901
9907	9923	9929	9931	9941	9949	9967	9973			

Capítulo 2

C++ e Biblioteca STD

2.1 Complex

Exemplo: **#include <complex>**, **complex<double> point**;

Funções: **real**, **imag**, **ab**s, **arg**, **norm**, **conj**, **polar**

2.2 Pair

#include <utility>
pair<tipo1, tipo2> P;
tipo1 first, tipo2 second

2.3 List

list<Elem> c //Cria uma lista vazia.
list<Elem> c1(c2) //Cria uma cópia de uma outra lista do mesmo tipo (todos os elementos são copiados).
list<Elem> c(n) //Cria uma lista com n elementos definidos pelo construtor default.
list<Elem> c(n,elem) //Cria uma lista inicializada com n cópias do elemento $elem$.
list<Elem> c(beg,end) //Cria uma lista com os elementos no intervalo $[beg, end)$.
c.˜list<Elem>() //Destrói todos os elementos e libera a memória.

Membros de list:
begin, **end**, **rbegin**, **rend**, **size**, **empty**, **clear**, **swap**.
front //Retorna o primeiro elemento.
back //Retorna o último elemento.
push_back //Coloca uma cópia de $elem$ no final da lista.
pop_back //Remove o último elemento e não retorna ele.
push_front//Insere uma cópia de $elem$ no começo da lista.
pop_front //Remove o primeiro elemento da lista e não retorna ele.
swap //Troca duas list's em $O(1)$.
erase (it)//Remove o elemento na posição apontada pelo iterador it e retorna a posição do próximo elemento.
erase (beg,end)//Remove todos os elementos no range $[beg, end)$ e retorna a posição do próximo elemento;
insert (it, pos)//Insere o elemento pos na posição anterior à apontada pelo iterador it.

2.4 Vector

#include <vector>
vector<tipo> V;

Membros de vector:
begin, **end**, **rbegin**, **rend**, **size**, **empty**, **clear**, **swap**.
reserve //Seta a capacidade mínima do vetor.
front //Retorna a referência para o primeiro elemento.
back //Retorna a referência para o último elemento.
erase //Remove um elemento do vetor.
pop_back //Remove o último elemento do vetor.
push_back //Adiciona um elemento no final do vetor.
swap //Troca dois vector's em $O(1)$.

2.5 Deque

#include <queue>
deque<tipo> Q;
Q[50] //Acesso randômico.

Membros de deque:
begin, **end**, **rbegin**, **rend**, **size**, **empty**, **clear**, **swap**.
front //Retorna uma referência para o primeiro elemento.
back //retorna uma referência para o último elemento.
erase //Remove um elemento do deque.
pop_back //Remove o último elemento do deque.
pop_front //Remove o primeiro elemento do deque.
push_back //Insere um elemento no final do deque.
push_front//Insere um elemento no começo do deque.

2.6 Queue

#include <queue>
queue<tipo> Q;

Membros de queue:
back //Retorna uma referência ao último elemento da fila.
empty //Retorna se a fila está vazia ou não.
front //Retorna uma referência ao primeiro elemento da fila.
pop //Retorna o primeiro elemento da fila.

push //Insere um elemento no final da fila.
size //Retorna o número de elementos da fila.

2.7 Stack

#include <stack>
stack<tipo> P;

Membros de stack:
empty //Retorna se pilha está vazia ou não.
pop //Remove o elemento no topo da pilha.
push //Insere um elemento na pilha.
size //retorna o tamanho da pilha.
top //Retorna uma referÊncia para o elemento no topo da pilha.

2.8 Map

#include <map>
#include <string>
map<string, int> si;

Membros de map:
begin, **end**, **rbegin**, **rend**, **size**, **empty**, **clear**, **swap**, **count**.
erase //Remove um elemento do mapa.
find //retorna um iterador para um elemento do mapa que tenha a chave.
lower_bound //Retorna um iterador para o primeiro elemento maior que a chave ou igual à chave.
upper_bound //Retorna um iterador para o primeiro elemento maior que a chave.

Map é um set de pair, ao iterar pelos elementos de map, $i- > first$ é a chave e $i- > second$ é o valor.

Map com comparador personalizado: Utilizar **struct** com **bool operator<(tipoStruct s) const** . Cuidado pra diferenciar os elementos!

2.9 Set

#include <set>
set<tipo> S;

Membros de set:
begin, **end**, **rbegin**, **rend**, **size**, **empty**, **clear**, **swap**.
erase //Remove um elemento do set.
find //Retorna um iterador para um elemento do set.
insert //Insere um elemento no set.
lower_bound //Retorna um iterador para o primeiro elemento maior que um valor ou igual a um valor.
upper_bound //Retorna um iterador para o primeiro elemento maior que um valor.

Criando set com comparador personalizado: Utilizar **struct cmp** com **bool operator()(tipo, tipo) const** e declarar **set<tipo, vector<tipo>, cmp()> S**. Cuidado pra diferenciar os elementos!

2.10 Priority Queue

#include <queue>
priority_queue<tipo> pq

Membros: **empty**, **size**, **top**, **push**, **pop**.

Utilizar **struct cmp** com **bool operator ()(tipo, tipo)** e declarar **priority_queue<tipo, vector<tipo>, cmp()> pq**
.

Maior vem antes!

2.11 Bitset

#include <bitset>
bitset<MAXN> bs

Membros: **empty**, **size**, **count**, **to_string**, **to_ulong**, **to_ullong**.
set //Seta todos os elementos para 1.
reset //Seta todos os elementos para 0.
flip(n) //Alterna o bit n.
flip //Alterna todos os bits.

2.12 Algorithm

#include <algorithm>
int a = max(5,13); //Retorna o maior valor (templatizado).
int a = min(5,13); //Retorna o menor valor.
vector <int> v;
sort(v.begin(), v.end()); //Ordena de acordo com o operador <.
int vv[MAXN];
stable_sort(&vv[0], &vv[MAXN]); //Ordena mantendo a ordem de elementos iguais.
reverse(v.begin(), v.end()); //Inverte a ordem.

Nas funções abaixo pode-se mandar uma função de comparação $comp$ customizada mandando $\&comp$ como último parâmetro.
next_permutation(v.begin(), v.end()); //Reordena v para a próxima permutação segundo a ordenação lexicográfica. $O(n)$. Retorna $false$ se não existir.
prev_permutation(v.begin(), v.end()); //Reordena v para a permutação anterior segundo a ordenação lexicográfica. $O(n)$. Retorna $false$ se não existir.
nth_element(v.begin(), v.begin()+n, v.end()); //Coloca o n-ésimo elemento na posição correta, todos os menores que ele antes e todos os maiores depois. $Expected\ O(n)$.

Capítulo 3

Estruturas de dados

.

3.1 Heap

Árvore de prioridade ou priority_queue. Suporta o update e ordena pelo menor dist[u]. Comparador equivale a 'menor que'. O vetor heap é 1-indexed.

```
#define swap(a, b) { int _x = a; a=b; b=_x; }
#define MAXN 100009

int dist[MAXN];
bool comp(int a, int b){
    return dist[a] < dist[b];
}

class Heap{
private:
    int heap[MAXN];
    int inv[MAXN];
    int heapsize;
    void sifup(int n){
        int k = n << 1;
        while (k <= heapsize){
            if (k < heapsize && comp(heap[k+1], heap[k
                ])) k++;
            if (comp(heap[k], heap[n])){
                swap(heap[n], heap[k]);
                inv[heap[n]] = n;
                n = inv[heap[k]] = k;
                k <<= 1;
            }
            else break;
        }
    }
    void sifdown(int n){
        int k = n >> 1;
        while (k){
            if (comp(heap[n], heap[k])){
                swap(heap[n], heap[k]);
                inv[heap[n]] = n;
                n = inv[heap[k]] = k;
                k >>= 1;
            }
            else break;
        }
    }
public:
    Heap() { heapsize = 0; }
    void clear() { heapsize = 0; }
    bool empty() { return heapsize==0; }
    void update(int n){
        if (inv[n]>heapsize) return;
        sifup(inv[n]);
        sifdown(inv[n]);
    }
    void push(int n){
        heap[++heapsize] = n;
        inv[n] = heapsize;
        sifdown(heapsize);
    }
    bool count(int n){
        int k = inv[n];
        return k <= heapsize && k > 0 && heap[k] == n;
    }
    int top(){
        if (heapsize <=0) return -1;
        return heap[1];
    }
    void pop(){
        if (heapsize <=0) return;
        heap[1] = heap[heapsize--];
        inv[heap[1]] = 1;
        sifup(1);
    }
};
```

3.2 Union-Find

Disjoint sets em tempo $O(logn)$

```
#include <cstdio>
#include <vector>
using namespace std;

class UnionFind {
private:
    vector<int> parent, rank;
public:
    UnionFind(int N) {
        rank.assign(N+1, 0);
        parent.assign(N+1, 0);
        for (int i = 0; i <= N; i++) parent[i] = i;
    }
    int find(int i) {
        while (i != parent[i]) i = parent[i];
        return i;
    }
    bool isSameSet(int i, int j) {
        return find(i) == find(j);
    }
    void unionSet(int i, int j) {
        if (isSameSet(i, j)) return;
        int x = find(i), y = find(j);
        if (rank[x] > rank[y]) parent[y] = x;
        else {
            parent[x] = y;
            if (rank[x] == rank[y]) rank[y]++;
        }
    }
};
```

3.3 Binary Indexed Tree/Fenwick Tree

Resolve queries do tipo RSQ de 1 a n (1-indexed) em $O(logn)$. Update pontual em $O(logn)$.

```
#include <vector>
using namespace std;

const int neutral = 0;
int comp(int a, int b) {
    return a+b;
}

class FenwickTree {
private:
    vector<int> ft;
public:
    FenwickTree(int n) {
        ft.assign(n + 1, 0); //1-indexed
    }
    int rsq(int i) { // returns RSQ(1, i)
        int sum = neutral;
        while (i > 0) {
            sum = comp(sum, ft[i]);
            i -= (i & -i);
        }
        return sum;
    }
    int rsq(int i, int j) {
        return rsq(j) - rsq(i - 1);
    }
    void update(int i, int v) {
        while (i < (int)ft.size()) {
            ft[i] = comp(v, ft[i]);
            i += (i & -i);
        }
    }
};
```

3.4 Binary Indexed Tree/ Fenwick Tree com range updates e queries

Resolve queries do tipo RSQ de i a j (1-indexed) em $O(logn)$. Range updates ($a[i...j] += v$) em $O(logn)$.

```
#include <vector>
using namespace std;

class FenwickTree {
private:
    vector<int> ft1, ft2;
    int rsq(vector<int> &ft, int i) {
        int sum = 0;
        while (i > 0) {
            sum += ft[i];
            i -= (i & -i);
        }
        return sum;
    }
    void update(vector<int> &ft, int i, int v) {
        while (i < (int)ft.size()) {f
            t[i] += v;
            i += (i & -i);
        }
    }
public:
    FenwickTree(int n) {
        ft1.assign(n + 1, 0);    //1-indexed
        ft2.assign(n + 1, 0);    //1-indexed
    }
    void update(int i, int j, int v) { update(
        ft1, i, v);
        update(ft1, j+1, -v);
        update(ft2, i, v * (i-1));
        update(ft2, j+1, -v*j);
    }
    int rsq(int i) {
        return rsq(ft1, i) * i - rsq(ft2, i);
    }
    int rsq(int i, int j) {
        return rsq(j) - rsq(i-1);
    }
};
```

3.5 Segment Tree

Árvore de segmentos em 1D.

```
#include <cstdio>
#include <vector>
#include <algorithm>
#define INF (1<<30)
using namespace std;

const int neutral = 0;
int comp(int a, int b){
    return a+b;
}

class SegmentTree {
private:
    vector<int> st, pos;
    int size;
#define parent(p) (p >> 1)
#define left(p) (p << 1)
#define right(p) ((p << 1) + 1)
    void build(int p, int l, int r, int* A){
        if (l == r){
            st[p] = A[l];
            pos[l] = p;
        }
        else {
            build(left(p), l, (l + r) / 2, A);
            build(right(p), (l + r) / 2 + 1, r, A);
            st[p] = comp(st[left(p)], st[right(p)]);

        }
    }
    int find(int p, int l, int r, int a, int b) {
        if (a > r || b < l) return neutral;
        if (l >= a && r <= b) return st[p];
        int p1 = find(left(p), l, (l + r) / 2, a, b);
        int p2 = find(right(p), (l + r) / 2 + 1, r, a, b);
        return comp(p1, p2);
    }
public:
    SegmentTree(int* begin, int* end) { size = (int)(end - begin);
        st.assign(4 * size, neutral);
        pos.assign(size + 9, 0);
        build(1, 0, size - 1, begin);
    }
    int query(int a, int b) { return find(1, 0, size - 1, a, b);}
    void update(int n, int num){
        st[pos[n]] = num;
        n = parent(pos[n]);
        while (n>0){
            st[n] = comp(st[left(n)], st[right(n)]);
            n = parent(n);
        }
    }
};
```

3.6 Segment Tree com Lazy Propagation

```
#include <cstdio>
#include <vector>
#include <algorithm>
#define INF (1<<30)
using namespace std;

const int neutral = 0; //comp(x, neutral) = x
int comp(int a, int b){
    return a + b;
}

class SegmentTree {
private:
    vector<int> st, lazy;
    int size;
#define parent(p) (p >> 1)
#define left(p) (p << 1)
#define right(p) ((p << 1) + 1)
    void build(int p, int l, int r, int* A){
        if (l == r){
            st[p] = A[l];
        }
        else {
            build(left(p), l, (l + r) / 2, A);
            build(right(p), (l + r) / 2 + 1, r, A);
            st[p] = comp(st[left(p)], st[right(p)]);
        }
    }
    void push(int p, int l, int r) {
        st[p] += (r - l + 1) * lazy[p]; //Caso RSQ
        //st[p] += lazy[p];             //Caso RMQ
        if (l != r) {
            lazy[right(p)] += lazy[p];
            lazy[left(p)] += lazy[p];
        }
        lazy[p] = 0;
    }
    void update(int p, int l, int r, int a, int b, int k) {
        push(p, l, r);
        if (a > r || b < l) return;
        else if (l >= a && r <= b) {
            lazy[p] = k; push(p, l, r);
        }
        else {
            update(left(p), l, (l + r) / 2, a, b, k);
            update(right(p), (l + r) / 2 + 1, r, a, b, k);
            st[p] = comp(st[left(p)], st[right(p)]);
        }
    }
    int query(int p, int l, int r, int a, int b) {
        push(p, l, r);
        if (a > r || b < l) return neutral;
        if (l >= a && r <= b) return st[p];
        int p1 = query(left(p), l, (l + r) / 2, a, b);
        int p2 = query(right(p), (l + r) / 2 + 1, r, a, b);
        return comp(p1, p2);
    }
public:
    SegmentTree(int* begin, int* end) { size = (int)(end - begin);
        st.assign(4 * size, neutral);
        lazy.assign(4 * size, 0);
        build(1, 0, size - 1, begin);
    }
    int query(int a, int b) { return query(1, 0, size - 1, a, b);}
    void update(int a, int b, int k) { update(1, 0, size - 1, a, b, k); }
};
```

3.7 2D Binary Indexed Tree/Fenwick Tree

```
#include <vector>
using namespace std;
const int neutral = 0;
int comp(int a, int b){
    return a+b;
}
class FenwickTree2D {
private:
    vector< vector<int> > ft;
public:
    FenwickTree2D(int n, int m){
        ft.assign(n + 1, vector<int>(m + 1, 0)); //1-
            indexed
    }
    int rsq(int i, int j) { // returns RSQ((1,1), (i,
        j))
        int sum = 0, _j = j;
        while (i > 0) { j = _j;
            while (j > 0) {
                sum = comp(sum, ft[i][j]);
                j -= (j & -j);
            }
            i -= (i & -i);
        }
        return sum;
    }
    void update(int i, int j, int v) {
        int _j = j;
        while (i < (int)ft.size()) { j = _j;
            while (j < (int)ft[i].size()) {
                ft[i][j] = comp(v, ft[i][j]);
                j += (j & -j);
            }
            i += (i & -i);
        }
    }
};
```

3.8 2D Segment Tree

```
#include <vector>
using namespace std;
#define INF (1<<30)
const int neutral = 0;
int comp(int a, int b) {
    return a+b;
}
class SegmentTree2D {
    int sizex, sizey, ix, jx, iy, jy, x, y, v; vector
    < vector<int> > st;
#define parent(p) (p >> 1)
#define left(p) (p << 1)
#define right(p) ((p << 1) + 1)
    void buildx(int px, int lx, int rx, vector< vector
        <int> > &A) {
        if(lx != rx){
            int mx = (lx + rx) / 2;
            buildx(left(px), lx, mx, A);
            buildx(right(px), mx+1, rx, A);
        }
        buildy(px, lx, rx, 1, 0, sizey-1, A);
    }
    void buildy(int px, int lx, int rx, int py, int
        ly, int ry, vector< vector<int> > &A) {
        if(ly == ry){
            if(lx == rx) st[px][py] = A[lx][ly];
            else st[px][py] = comp(st[left(px)][py], st
                [right(px)][py]);
        }
        else {
            int my = (ly + ry) / 2;
            buildy(px, lx, rx, left(py), ly, my, A);
            buildy(px, lx, rx, right(py), my+1, ry, A);
            st[px][py] = comp(st[px][left(py)], st[px][
                right(py)]);
        }
    }
    int queryx(int px, int lx, int rx) {
        if(lx > jx || rx < ix) return neutral;
        if(lx >= ix && rx <= jx) return queryy(px, 1,
            0, sizey-1);
        int mx = (lx + rx)/2;
        int p1x = queryx(left(px), lx, mx); int
        p2x = queryx(right(px), mx+1, rx);
        return comp(p1x, p2x);
    }
    int queryy(int px, int py, int ly, int ry) {
        if(ly > jy || ry < iy) return neutral;
        if(ly >= iy && ry <= jy) return st[px][py];
        int my = (ly + ry) / 2;
        int p1y = queryy(px, left(py), ly, my);
        int p2y = queryy(px, right(py), my+1, ry);
        return comp(p1y, p2y);
    }
    void updatex(int px, int lx, int rx) {
        if(lx > x || rx < x) return;
        if(lx < rx){
            int mx = (lx + rx) / 2;
            updatex(left(px), lx, mx);
            updatex(right(px), mx+1, rx);
        }
        updatey(px, lx, rx, 1, 0, sizey-1);
    }
    void updatey(int px, int lx, int rx, int py, int
        ly, int ry) {
        if(ly > y || ry < y) return;
        if(ly == ry){
            if(lx == rx) st[px][py] = v;
            else st[px][py] = comp(st[left(px)][py], st[
                right(px)][py]);
        }
        else {
            int my = (ly + ry) / 2;
            updatey(px, lx, rx, left(py), ly, my); updatey
            (px, lx, rx, right(py), my+1, ry);
            st[px][py] = comp(st[px][left(py)], st[px][
                right(py)]);
        }
    }
public:
    SegmentTree2D(vector< vector<int> > &A) { siz
        ex = A.size();
        sizey = A[0].size();
        st.assign(4*sizex+9, vector<int>(4*sizey+9));
        buildx(1, 0, sizex-1, A);
    }
    void update(int _x, int _y, int _v) {
        x = _x; y = _y; v = _v;
        updatex(1, 0, sizex - 1);
    }
    int query(int lx, int rx, int ly, int ry) {
        ix = lx; jx = rx; iy = ly; jy = ry;
        return queryx(1, 0, sizex - 1);
    }
};
```

3.9 Persistent Segment Tree

Segment Tree Persistente. Ao começar a usar a árvore, chamar o construtor com o tamanho exato. *update* retorna o número da nova versão. $MAXS$ deve ser da ordem de $2N + QlogN$. As versões são indexadas em 0, sendo 0 a versão original.

```
#include <vector>
#include <algorithm>
#define INF (1<<30)
using namespace std;
#define MAXS 2000009

const int neutral = 0; //comp(x, neutral) = x
int comp(int a, int b){
    return a+b;
}

class PersistentSegmentTree {
private:
    int st[MAXS], vroot[MAXS];
    int left[MAXS], right[MAXS];
    int size, nds, nv;
    int newnode(){
        left[nds] = right[nds] = -1;s
        t[nds++] = neutral;
        return nds-1;
    }
    void build(int p, int l, int r, int* A){
        if (l == r){
            st[p] = A ? A[l] : neutral;
        }
        else {
            left[p] = newnode();
            right[p] = newnode();
            int m = (l + r) / 2;
            build(left[p], l, m, A);
            build(right[p], m+1, r, A);
            st[p] = comp(st[left[p]],st[right[p]]);
        }
    }
    void update(int prv, int p, int l, int r, int i ,
         int k) {
        if (i > r || i < l || l > r) return ;
        int m = (l + r) / 2;
        if (l == r) st[p] = k;
        else if (i <= m){
            right[p] = right[prv];
            left[p] = newnode();
            update(left[prv], left[p], l, m, i, k);
            st[p] = comp(st[left[p]],st[right[p]]);
        }
        else {
            left[p] = left[prv];
            right[p] = newnode();
            update(right[prv], right[p], m+1, r, i,k);
            st[p] = comp(st[left[p]],st[right[p]]);
        }
    }
    int query(int p, int l, int r, int a, int b) {
        if (a > r || b < l || l > r) return neutral;
        if (l >= a && r <= b) return st[p];
        int p1 = query(left[p], l, (l + r) / 2, a, b);
        int p2 = query(right[p], (l + r) / 2 + 1, r, a
            , b);
        return comp(p1, p2);
    }
public:
    PersistentSegmentTree(){size = nds = nv = 0;}
    PersistentSegmentTree(int* begin, int* end){
        nds = nv = 0; size = (int)(end-begin);
        vroot[nv++] = newnode();
        build(vroot[0], 0, size-1, begin);
    }
    PersistentSegmentTree(int _size) {
        nds = nv = 0; size = _size; vro
        ot[nv++] = newnode();
        build(vroot[0], 0, size-1, NULL);
    }
    int query(int a, int b, int v) { return query(vroot
        [v], 0, size-1, a, b); }
    int update(int i, int v, int k) {
        vroot[nv++] = newnode();
        update(vroot[v], vroot[nv - 1], 0, size-1, i, k)
            ;
        return nv-1;
    }
    int nver() { return nv; }
};
```

3.10 Sparse Table

Resolve queries do tipo RMQ de l a r em $O(1)$. Pré-processamento $O(nlogn)$.

```
#include <cstdio>
#include <cmath>
#define MAXN 100009
#define MAXLOGN 20

// comparison function, can reclace with min, max, gcd
int comp (int a, int b){
    return min(a,b);
}

class SparseTable {
private:
    int SpT[MAXN][MAXLOGN];
    int size;
public:
    SparseTable(int* begin, int* end) {
        size = end - begin;
        for (int i=0; i<size; i++){
            SpT[i][0] = begin[i];
        }
        for (int j = 1; 1 << j <= size; j++){
            for (int i=0; i + (1<<j) <= size; i++){
                SpT[i][j] = comp(SpT[i][j-1], SpT[i
                    +(1<<(j-1))][j - 1]);
            }
        }
    }
    int query(int l, int r){
        int k = (int)floor(log((double)r-l+1) / log
            (2.0)); // 2^k <= (j-i+1)
        return comp(SpT[l][k], SpT[r-(1<<k)+1][k]);
    }
};
```

3.11 Treap/Cartesian Tree

Treap simples, suporta operações da BST (*insert, count, erase, nth_element*). Propriedades:

- É 10 vezes mais lento que Red-Black Tree, só usar se for estritamente necessário.
- Se os valores de y forem os valores de uma array e x a posição, as queries de máximo se tornam queries de LCA.
- IMPORTANTE: *split* separa entre k-1 e k.
- *nth_element* é 1-indexed.
- Não suporta valores repetidos de *x*.

```
#include <cstdio>
#include <set>
#include <algorithm>
using namespace std;

struct node {
    int x, y, size;
    node *l, *r;
    node (int _x) {
        x = _x;
        y = rand(); s
        ize = 1;
        l = r = NULL;
    }
};

class Treap{
private:
    node *root;
    void refresh(node *t) {
        if (t == NULL) return;
        t->size = 1;
        if (t->l != NULL)
            t->size += t->l->size;
        if (t->r != NULL)
            t->size += t->r->size;
    }
    void split(node *&t, int k, node *&a, node *&b) {
        node *aux;
        if (t == NULL) {
            a = b = NULL;
            return;
        }
        else if (t->x < k) {
            split(t->r, k, aux, b);
            t->r = aux;
            refresh(t);
            a = t;
        }
        else {
            split(t->l, k, a, aux);
            t->l = aux;
            refresh(t);
            b = t;
        }
    }
    node *merge(node *&a, node *&b) {
        node *aux;
        if (a == NULL) return b;
        else if (b == NULL) return a;
        if (a->y < b->y) {
            aux = merge(a->r, b);
            a->r = aux;
            refresh(a);
            return a;
        }
        else {
            aux = merge(a, b->l);
            b->l = aux;
            refresh(b);
            return b;
        }
    }
    node *count(node *t, int k) {
        if (t == NULL) return NULL;
        else if (k < t->x) return count(t->l, k);
        else if (k == t->x) return t;
        else return count(t->r, k);
    }
    int size(node *t) {
        if (t == NULL) return 0;
        else return t->size;
    }
    node *nth_element(node *t, int n) {
        if (t == NULL) return NULL;
        if (n <= size(t->l)) return nth_element(t->l, n);
        else if (n == size(t->l) + 1) return t;
        else return nth_element(t->r, n-size(t->l)-1);
    }
    void del(node *&t) {
        if (t == NULL) return;
        if (t->l != NULL) del(t->l);
        if (t->r != NULL) del(t->r);
        delete t;
        t = NULL;
    }
public:
    Treap() { root = NULL; }
    ~Treap() { clear(); }
    void clear() { del(root); }
    int size() { return size(root); }
    bool count(int k) { return count(root, k) != NULL; }
    bool insert(int k) {
        if (count(root, k) != NULL) return false;
        node *a, *b, *c, *d;
        split(root, k, a, b);
        c = new node(k);
        d = merge(a, c);
        root = merge(d, b);
        return true;
    }
    bool erase(int k) {
        node *f = count(root, k);
        if (f == NULL) return false;
        node *a, *b, *c, *d;
        split(root, k, a, b);
        split(b, k+1, c, d);
        root = merge(a, d);
        delete f;
        return true;
    }
    int nth_element(int n) {
        node *ans = nth_element(root, n);
        if (ans == NULL) return -1;
        else return ans->x;
    }
};
```

3.12 Treap/Cartesian Tree implícita

Suporta operações do vector, da SegmentTree e reverse em $O(logn)$ (*insertAt, erase, at, query, reverse*). Propriedades:

- Pode trocar trechos do vetor de lugar com *split* e *merge*.
- IMPORTANTE: *split* separa em árvores de tamanho k e $size - k$.
- *at* é 0-indexed.

```
#include <cstdio>
#include <algorithm>
#define INF (1 << 30)
using namespace std;

const int neutral = 0; //comp(x, neutral) = x
int comp(int a, int b) {
    return a + b;
}

struct node {
    int y, v, sum, size;
    bool swap; node
    *l, *r; node(
    int _v) {
        v = sum = _v;
        y = rand();
        size = 1;
        l = r = NULL;
        swap = false;
    }
};

class ImplicitTreap {
private:
    node *root;
    void refresh(node * t) {
        if(t == NULL) return;
        t->size = 1;
        t->sum = t->v;
        if (t->l != NULL) {
            t->size += t->l->size;
            t->sum = comp(t->sum, t->l->sum);
            t->l->swap ^= t->swap;
        }
        if (t->r != NULL) {
            t->size += t->r->size;
            t->sum = comp(t->sum, t->r->sum);
            t->r->swap ^= t->swap;
        }
        if (t->swap) {
            swap(t->l, t->r);
            t->swap = false;
        }
    }
    void split(node * &t, int k, node * &a, node * &b) { r
        efresh(t);
        node * aux;
        if(t == NULL) {
            a = b = NULL;
            return;
        }
        else if(size(t->l) < k) {
            split(t->r, k-size(t->l)-1, aux, b);
            t->r = aux;
            refresh(t);
            a = t;
        }
        else{
            split(t->l, k, a, aux);
            t->l = aux;
            refresh(t);
            b = t;
        }
    }
    node* merge(node* &a, node* &b){
        refresh(a);
        refresh(b);
        node * aux;
        if(a == NULL) return b;
        else if(b == NULL) return a;
        if(a->y < b->y) {
            aux = merge(a->r, b);
            a->r = aux;
            refresh(a);
            return a;
        }
        else{
            aux = merge(a, b->l);
            b->l = aux;
            refresh(b);
            return b;
        }
    }
    node * at(node * t, int n) {
        if (t == NULL) return NULL;
        refresh(t);
        if(n < size(t->l)) return at(t->l, n);
        else if(n == size(t->l)) return t; else
        return at(t->r, n-size(t->l)-1);
    }
    int size(node * t) {
        if (t == NULL) return 0;
        else return t->size;
    }
    void del(node * &t) {
        if (t == NULL) return;
        if (t->l != NULL) del(t->l);
        if (t->r != NULL) del(t->r);
        delete t;
        t = NULL;
    }
public:
    ImplicitTreap() { root = NULL; }
    ~ImplicitTreap() { clear(); }
    void clear() { del(root); }
    int size() { return size(root); }
    bool insertAt(int n, int v) {
        node *a, *b, *c, *d;
        split(root, n, a, b);
        c = new node(v);
        d = merge(a, c);
        root = merge(d, b);
        return true;
    }
    bool erase(int n) {
        node *a, *b, *c, *d; sp
        lit(root, n, a, b);
        split(b, 1, c, d); r
        oot = merge(a, d);
        if (c == NULL) return false;
        delete c;
        return true;
    }
    int at(int n) {
        node * ans = at(root, n);
```

```
        if (ans == NULL) return -1;
        else return ans->v;
    }
    int query(int l, int r){ if
        (l>r) swap(l, r); node
        *a, *b, *c, *d;
        split(root, l, a, d);
        split(d, r-l+1, b, c);
        int ans = (b != NULL ? b->sum : neutral);
        d = merge(b, c);
        root = merge(a, d);
        return ans;
    }
    void reverse(int l, int r){
        if(l>r) swap(l, r); node
        *a, *b, *c, *d; split(
        root, l, a, d);
        split(d, r-l+1, b, c);
        if(b != NULL) b->swap ^= 1;
        d = merge(b, c);
        root = merge(a, d);
    }
};
```

3.13 Splay Tree

Árvore de busca binária em que, para todas as operações, rotaciona-se a raíz até o elemento desejado chegar na raíz.

```
struct node {
    int key;
    node *left, *right;
    node(int k){
        key = k; left = right = 0;
    }
};
class SplayTree {
private:
    node *root;
    node *rotateright(node *p){
        node* q = p->left;
        p->left=q->right;
        q->right = p;
        return q;
    }
    node *rotateleft(node *q){
        node* p = q->right;
        q->right=p->left;
        p->left = q;
        return p;
    }
    node *splay(node *p, int key) {
        if (p== NULL || p->key == key) return p;
        if (p->key > key){
            if (!p->left) return p;
            if (p->left->key > key){
                p->left->left = splay(p->left->left, key
                    );
                p= rotateright(p);
            }
            else if(p->left->key < key) {
                p->left->right = splay(p->left->right,
                    key);
                if (p->left->right)
                    p->left = rotateleft(p->left);
            }
            return (p->left == NULL) ? p : rotateright(
                p);
        }
        else {
            if (!p->right) return p;
            if (p->right->key > key) {
                p->right->left= splay(p->right->left,
                    key);
                if (p->right->left)
                    p->right = rotateright(p->right);
            }
            else if (p->right->key < key) {
                p->right->right = splay(p->right->right,
                    key);
                p = rotateleft(p);
            }
            return (p->right == NULL) ? p : rotateleft(
                p);
        }
    }
    void del(node *p){
        if (!p) return;
        del(p->left); del(p->right);
        delete p;
    }
public:
    SplayTree() { root = 0; }
    ~SplayTree() { del(root); }
    bool empty() { return root == NULL; }
    void clear() {
        del(root);
        root = 0;
    }
    void insert(int key) {
        if (!root){
            root = new node(key);
            return;
        }
        node *p = splay(root, key);
        if (p->key == key) return;
        root= new node(key);
        if (p->key > key) {
            root ->right = p;
            root ->left= p->left;
            p->left = NULL;
        }
        else {
            root ->left = p;
            root ->right= p->right;
            p->right= NULL;
        }
    }
    void erase(int key) {
        node * p= splay(root, key);
        if (p != NULL && p->key != key) return;
        if (!p->right) {
            root= p->left;
            delete p;
            return;
        }
        node *q = splay(p->right, key);
        q->left = p->left;
        root =q;
        delete p;
    }
    bool count(int key) {
        if(!root) return false; ro
        ot = splay(root, key);
        return root ->key == key;
    }
};
```

3.14 Link Cut Tree

Estrutura de dados semelhante a disjoint sets que permite conectar vértices sem pai a algum outro vértice, cortar relação com o pai e queries de raíz da árvore e LCA. Tudo em $O(log^2 n)$.

```
#include <vector >
using namespace std;
#define INF (1<<30)

struct node
{    int size, id, w;
     node *par, *ppar, *left, *right;
     node() {
        par = ppar = left = right = NULL; w
        = size = INF;

     }
};

class LinkCutTree
{
    vector <node> lct;

    void update(node * p) {

        p->size = p->w;
        if (p->left) p->size += p->left->size;
        if (p->right) p->size += p->right->size;
    }

    void rotateright(node * p) {
        node *q, *r;
        q = p->par, r = q->par;
        if((q->left = p->right)) q->left->par = q;
        p->right = q, q->par = p;
        if ((p->par = r)) {
            if (q == r->left) r->left = p;
            else r->right = p;
        }
        p->ppar = q->ppar;

        q->ppar = NULL;
        update(q);
    }


    void rotateleft(node * p) {
        node *q, *r;
        q = p->par, r = q->par;
        if((q->right = p->left)) q->right->par = q;
        p->left = q, q->par = p;
        if ((p->par = r))
        {
            if (q == r->left) r->left = p;
            else r->right = p;
        }
        p->ppar = q->ppar;
        q->ppar = 0;
        update(q);
    }

    void splay(node * p) {
        node *q, *r;
        while (p->par != NULL) {
            q = p->par;
            if (q->par == NULL) {
                if (p == q->left) rotateright(p);
                else rotateleft(p);
            }
            else {
                r = q->par;
                if (q == r->left) {
                    if (p == q->left) rotateright(q),
                        rotateright(p);
                    else rotateleft(p), rotateright(p);
                }
                else {
                    if (p == q->right) rotateleft(q),
                        rotateleft(p);
                    else rotateright(p), rotateleft(p);
                }
            }
        }
        update(p);
    }


    node* access(node* p){
        splay(p);
        if (p->right != NULL) {
            p->right->ppar = p;
            p->right->par = NULL;
            p->right = NULL;
            update(p);
        }

        node* last = p;
        while (p->ppar != NULL) {
            node * q = p->ppar;
            last = q;
            splay(q);
            if(q->right != NULL){
                q->right->ppar = q;
                q->right->par = NULL;
            }
            q->right = p;
            p->par = q;
            p->ppar = NULL;
            update(q);
            splay(p);
        }

        return last;
    }


public:
    LinkCutTree() {}
    LinkCutTree(int n) {
        lct.resize(n+ 1);
          for(int i = 0; i <= n; i++){
            lct[i].id = i;
                update(&lct[i]);
          }
    }

    void link(int u, int v, int w) { //u becomes
        child of v
        node *p = &lct[u], *q = &lct[v];a
        ccess(p);
        access(q);
        p->left = q;
        q->par = p;
        p->w = w;
        update(p);
    }

    void link(int u, int v) { //unweighted
        link(u, v, 1);
    }

    void cut(int u) {node
        * p = &lct[u];acc
        ess(p);
        p->left->par = NULL;
        p->left = NULL;
        update(p);
```

```
    }

    int findroot(int u){
      node * p = &lct[u];
      access(p);
      while (p->left) p = p->left;
      splay(p);
      return p->id;
    }

  bool IsSameTree(int u, int v) {
    return findroot(u) == findroot(v);
  }

   int depth(int u) {
     access(&lct[u]);
     return lct[u].size - lct[u].w;
   }

   int LCA(int u, int v) {a
     ccess(&lct[u]);
     return access(&lct[v])->id;
   }
};
```

3.15 Link Cut Tree não direcionada

Semelhante a *LinkCutTree*, porém permite o *link* e o *cut* de quaisquer vértices de forma não direcionada.. Tudo em $O(log^2 n)$.

```
struct node { ... };
class LinkCutTree { ...};

class Undirected Link Cut Tree
{
    LinkCutTree lct;
    vector <int> par;

    void invert(int u) {
        if (par[u] == -1) return;
        int v = par[u];
        invert(v);
        lct.cut(u);
        par[u] = -1;
        lct.link(v, u);
        par[v] = u;
    }
public:
    Undirected Link Cut Tree() {}
    Undirected Link Cut Tree(int n) {
        lct = LinkCutTree(n);
        par.assign(n+1, -1);
    }
    void link(int u, int v) {
        if (lct.depth(u) < lct.depth(v)) {
            invert(u);
            lct.link(u,v);
            par[u] = v;
        }
        else{
            invert(v);
            lct.link(v,u);
            par[v] = u;
        }
    }
    void cut(int u, int v){
        if (par[v] == u) u = v;
        lct.cut(u);
        par[u] = -1;
    }
    bool Is Same Tree(int u, int v) {
        return lct.Is Same Tree(u, v);
    }
};
```

3.16 AVL Tree

```
struct node {
    int key, height, size;
    node *left, *right;
    node(int k) {
        key = k; left = right = 0;
        height = size = 1;
    }
};

class AVLtree {
private:
    node *root;
    int size_;
    int height(node *p) {
        return p ? p->height : 0;
    }
    int size(node *p) {
        return p ? p->size : 0;
    }
    int bfactor(node *p) {
        return height(p->right) - height(p->left);

    }
    void fixheight(node *p) {
        int hl = height(p->left);
        int hr = height(p->right);
        p->height = (hl>hr ? hl : hr) + 1;
        p->size = 1 + size(p->left) + size(p->right);
    }
    node *rotateright(node *p) {
        node* q = p->left;
        p->left = q->right;
        q->right = p;
        fixheight(p);
        fixheight(q);
        return q;
    }
    node *rotateleft(node *q) {
        node* p = q->right;
        q->right = p->left;
        p->left = q;
        fixheight(q);
        fixheight(p);
        return p;
    }
    node *balance(node *p) { f
        ixheight(p);
        if (bfactor(p) == 2) {
            if (bfactor(p->right) <0)
             p->right = rotateright(p->right);
            return rotateleft(p);
        }
        if (bfactor(p) == -2) {
            if (bfactor(p->left) >0)
               p->left = rotateleft(p->left);
            return rotateright(p);
        }
        return p;
    }
    node* build(node *p, int k) {
        if (!p) return new node(k);
        if (p->key == k) return p;
        else if (k<p->key) p->left = build(p->left,k)
            ;
        else p->right = build(p->right, k);
        return balance(p);
    }
    node *findmin(node *p) {
        return p->left ? findmin(p->left): p;
    }
    node *removemin(node *p) {
        if(p->left==0) return p->right;
        p->left = removemin(p->left);
        return balance(p);
    }
    node *remove(node *p, int k) {
        if (!p) return 0;
        if (k< p->key) p->left = remove(p->left, k);
        else if (k > p->key) p->right = remove(p->
            right, k);
        else {
            node *l = p->left;
            node *r = p->right;
            delete p;
            if (!r) return l;
            node *min = findmin(r);
            min->right = removemin(r);
            min->left = l;
            return balance(min);
        }
        return balance(p);
    }

    bool find(node* p, int k) {
        if (!p) return false;
        if (p->key == k) return true;
        else if(k<p->key) return find(p->left, k);
        else return find(p->right,k);
    }
    void del(node* p) {
        if (!p) return; de
        l(p->left);
        del(p->right);
        delete p;
    }
    node *nth(node *p, int n) {
        if (!p) return p;
        if (size(p->left) + 1 > n) return nth(p->left,
            n);
        if (size(p->left) + 1 < n) return nth(p->right
            , n - size(p->left) - 1);
        else return p ;
    }

public:
    AVLtree() { root = 0; size_ = 0; }
    ~AVLtree() { del(root); }
    bool empty() { return size_ == 0;}
    int size() { return size_;}
    void clear() {
        size_ = 0;
        del(root);
        root = 0;
    }
    void insert(int key) { s
        ize_++;
        root = build(root, key);
    }
    void erase(int key) {
        size_--;
        root = remove(root, key);
    }
    bool count(int key) {
        return find(root,key);
    }
    int nth_element(int n) { node
        *p = nth(root,n); if (p
        ) return p->key; else
        return -1;
    }   //1-indexed
};
```

3.17 Heavy-Light Decomposition

Decomposição Heavy-light de uma árvore em $O(n)$. Query de LCA em $O(logn)$. $chain[i]$ é a i-ésima cadeia. $nchs$ é o número de cadeias. $nchain[u]$ é o índice da cadeia a qual u pertence. $up[i]$ é o índice do nó do qual a i-ésima cadeia é filha. $id[u]$ é o índice de nó u dentro de sua cadeia. $fson[u]$ é o filho de u por onde a cadeia atual prossegue. $depth[i]$ é a profundidade do nó mais alto da i-ésima cadeia. Para queries de distância e outras, montar SegTrees, BIT's ou somas parciais por cadeia.

```
#include <vector>
using namespace std;
#define MAXN 100009

int par[MAXN], size[MAXN];
vector<int> adjList[MAXN];
int root, N, up[MAXN], fson[MAXN];
vector<int> chain[MAXN];
int nchs, nchain[MAXN], id[MAXN], depth[MAXN];

int sizedfs(int u, int p) {
    size[u] = 1; fson[u] = −1; par[u] = p;
    int msz = 0;
    for(int i=0; i<(int)adjList[u].size(); i++) {
        int v = adjList[u][i];
        if (v == p) continue;
        size[u] += sizedfs(v, u);
        if (size[v] > msz){
            fson[u] = v; msz = size[v];
        }
    }
    return size[u];
}

void builddfs(int u, int ch, int h){
    nchain[u] = ch; id[u] = chain[ch].size(); chain
    [ch].push_back(u);
    for(int i=0; i<(int)adjList[u].size(); i++){
        int v = adjList[u][i];
        if (v == par[u]) continue;
        if (v == fson[u]) builddfs(v, ch, h+1);
        else {
            up[nchs] = u; depth[nchs] = h;
            chain[nchs].clear();
            builddfs(v, nchs++, h+1);
        }
    }
}

void heavylightdecomposition(int _root) {
    root = _root;
    sizedfs(root, −1);
    nchs = 0; chain[0].clear();
    up[nchs] = −1; depth[nchs] = 0;b
    uilddfs(root, nchs++, 1);
}

int LCA(int u, int v){
    int cu = nchain[u], cv = nchain[v];
    while (cu != cv){
        if (depth[cu] > depth[cv]) u = up[cu];
        else v = up[cv];
        cu = nchain[u]; cv = nchain[v];
    }
    if (id[u] < id[v]) return u;
    else return v;
}
```

3.18 Centroid Decomposition

Realiza a decomposição em $O(nlogn)$ e retorna a raíz da decomposição. $csons[i]$ são os filhos do i-ésimo nó segundo a decomposição. $par[i]$ é o pai do i-ésimo nó segundo a decomposição. $label[i]$ é a profundidade do i-ésimo nó na decomposição, iniciando em 0. $size[i]$ no final do algoritmo contém o tamanho da subárvore de centróides com raíz i.
CUIDADO: o pai da raiz da árvore centróide é ele mesmo.

```
#include <cstring>
#include <vector>
using namespace std;
#define MAXN 100009

typedef long long ll;
typedef pair<ll, int> ii;
int clevel[MAXN], cpar[MAXN], csize[MAXN];
vector<int> csons[MAXN];
vector<ii> adjList[MAXN];
int N, K;

int subsize(int u, int p){
    csize[u]=1;
    for (int i=0; i<(int)adjList[u].size(); i++){
        int v = adjList[u][i].second;
        if (v != p && clevel[v] < 0)
            csize[u] += subsize(v, u);
    }
    return csize[u];
}
int findcentroid(int u, int p, int nn){
    for (int i=0; i<(int)adjList[u].size(); i++){
        int v = adjList[u][i].second;
        if (v != p && clevel[v] < 0 && csize[v] > nn
            /2)
            return findcentroid(v, u, nn);
    }
    return u;
}
int decompose (int root, int par){
    subsize(root, −1);
    int u = findcentroid(root, −1, csize[root]);
    cpar[u] = par;
    clevel[u] = par >= 0 ? clevel[par]+1 : 0;c
    size[u] =1;
    for (int i=0; i<(int)adjList[u].size(); i++){
        int v = adjList[u][i].second;
        if(v != par && clevel[v] < 0){
            v = decompose(v, u);
            csons[u].push_back(v);
            csize[u] += csize[v];
        }
    }
    return u;
}
int centroiddecomposition(int root){
    memset(&clevel, −1, sizeof clevel);
    for (int i=0; i<=N; i++) csons[i].clear();
    return decompose(root, −1);
}
```

3.19 Convex Hull Trick

Para queries de mínimo, sete $maxCH = false$ e insira retas do tipo $y = mx + n$ na ordem decrescente de m. Resolve queries de $min(y(x))$ para todas as retas inseridas em tempo $O(logn)$.
Para queries de máximo, sete $maxCH = true$ e insira retas do tipo $y = mx + n$ na ordem crescente de m. Resolve queries de $max(y(x))$ para todas as retas inseridas em tempo $O(logn)$.

```
#include <cstdio>
#include <vector>
#define INF (1<<30)
#define MAXN 1009
using namespace std;

typedef long long int ll;

class CHTrick{
private:
    vector<ll> m, n;
    vector<double> p;
    bool maxCH;
public:
    CHTrick(bool _maxCH) { maxCH = _maxCH; }
    void clear() {
        m.clear(); n.clear(); p.clear();
    }
    double inter(double nm, double nn, double lm,
        double ln) {
        return (ln - nn) / (nm - lm);
    }
    void push(ll nm, ll nn) {
        while (!p.empty() &&
            ((nm == m.back() && (maxCH? -1:1) *nn <= (
                maxCH? -1:1) *n.back()) ||
            p.back() >= inter(nm, nn, m.back(), n.
                back()))) {
            m.pop_back(); n.pop_back(); p.pop_back()
                ;
        }
        p.push_back(p.empty() ? -INF : inter(nm, nn,
            m.back(), n.back()));
        m.push_back(nm); n.push_back(nn);
    }
    ll query(ll x) {
        if (p.empty()) return (maxCH? -1:1) *INF;
        double dx = (double) x;
        ll high = p.size() -1, low = 0, mid;
        if (dx >= p[high]) return m[high] * x + n[high
            ];
        while (high > low +1){ mid
            = (high+low) / 2;
            if (dx < p[mid]) high = mid;
            else low = mid;
        }
        return m[low] * x + n[low];
    }
};
```

3.20 Lowest Commom Ancestor (LCA) e distância na árvore

$P[i][j]$ = o 2^j-ésimo pai do i-ésimo nó. *computeP* (*root*) computa a matriz P em $O(nlogn)$. A partir de P, pode-se calcular o LCA de dois nós em $O(log^2 n)$ para uma árvore qualquer com raiz em *root*. *dist* usa *LCA* pra calcular a query de distância em $O(log^2 n)$.

```
#include <vector>
#include <iostream>
using namespace std;
#define MAXN 100009
#define MAXLOGN 20

typedef pair<int, int> ii;
vector<ii> adjList[MAXN];
int depth[MAXN], level[MAXN];
int P[MAXN][MAXLOGN], N;

void depthdfs(int u) {
    for (int i=0; i<(int)adjList[u].size(); i++){
        int v = adjList[u][i].first;
        int w = adjList[u][i].second;
        if (v == P[u][0]) continue;
        P[v][0] = u;
        level[v] = 1 + level[u];
        depth[v]=w+depth[u];d
        epthdfs(v);
    }
}
void computeP(int root) {
    level[root]=depth[root]=0;
    P[root][0]=root;
    depthdfs(root);
    for (int j = 1; j < MAXLOGN; j++)
        for (int i = 1; i <= N; i++)
            P[i][j] = P[P[i][j-1]][j-1];
}
int LCA(int a, int b) {
    if(level[a] > level[b]) swap(a, b);
    int d = level[b] - level[a];
    for (int i=0; i<MAXLOGN; i++){
        if((d & (1<<i)) != 0) b = P[b][i];
    }
    if(a == b) return a;
    for (int i = MAXLOGN-1; i>=0; i--)
        while (P[a][i] != P[b][i]) {
            a=P[a][i]; b=P[b][i];
        }
    return P[a][0];
}
int dist(int u, int v) {
    return depth[u] + depth[v] - 2* depth[LCA(u, v)];
}
```

3.21 Merge Sort Tree

Constrói a árvore de recursão do merge-sort. $O(nlogn)$ em espaço e em tempo de contrução. Queries:

- O número de elementos menores que k em um intervalo (a, b) em $O(log^2 n)$.
- O n-ésimo elemento 0-indexed em um intervalo (a, b) em $O(log^3 n)$.

```
#include <vector>
#define INF (1<<30)
using namespace std;

class MergeSortTree {
private:
    vector< vector<int> > st;
    int size;
    #define parent(p) (p>>1)
    #define left(p) (p<<1)
    #define right(p) ((p<<1) + 1)
    void build(int p, int l, int r, int* A) { //O(n)
        st[p].resize(r-l+1);
        if (l == r) {
            st[p][0] = A[l];
        }
        else {
            int pl = left(p), pr = right(p), m = (l+r)
                /2;
            build(pl, l, m, A);
            build(pr, m+1, r, A);
            unsigned int i=0, j=0, k=0;
            while(i< st[pl].size() && j < st[pr].size()
                ){
                if (st[pl][i] < st[pr][j]) st[p][k++] =
                    st[pl][i++];
                else st[p][k++] = st[pr][j++];
            }
            while(i < st[pl].size()) st[p][k++] = st[pl
                ][i++];
            while(j < st[pr].size()) st[p][k++] = st[pr
                ][j++];
        }
    }
    int less(int p, int l, int r, int a, int b, int k
        ) { // O(log n)
        if (st[p][0] >= k || a > r || b < l) return 0;
        if (l >= a && r <= b){
            l = 0; r = (int)st[p].size();
            int m;
            while(r > l + 1){
                m = (r+l)/2;
                if (st[p][m] < k) l = m;
                else r = m;
            }
            return r;
        }
        int p1 = less(left(p), l, (l+r)/2, a, b, k);
        int p2 = less(right(p), (l+r)/2+1, r, a, b, k)
            ;
        return p1+p2;
    }
public:
    MergeSortTree(int* begin, int* end) { si
        ze = (int)(end-begin);
        st.assign(4*size, vector<int>());
        build(1, 0, size-1, begin);
    }
    int less(int a, int b, int k){
        return less(1, 0, size-1, a, b,k);
    }
    int nth_element(int a, int b, int n){
        int l = -INF, r = INF, m;
        while(r > l+1){
            m = (r+l)/2;
            if (less(a, b, m) <= n) l = m;
            else r = m;
        }
        return l;
    }
};
```

3.22 Código de Huffman

Constrói o autômato de Huffman: dado um conjunto de elementos, montar uma árvore cujas folhas são os elementos desse conjunto, o pai é a soma dos filhos e a soma de todos os nós é mínima. Propriedades:

- O caminho da raíz até a folha é o código de Huffman (filho esquerdo -> menor nó -> 0, filho direito -> maior nó -> 1);
- Nenhum código é o prefixo de outro;
- Essencialmente ele minimiza: $cost = \sum_{i=0}^{n-1} a[i] \times depth[i]$

```
#include <queue>
#include <vector>
using namespace std;
#define MAXN 100009
#define INF (1LL<<60)

typedef long long ll;
typedef pair<ll, int> ii;

class HuffmanTree{
    vector<int> left, right, par;
    int root, size;ll
    cost;

public:
    HuffmanTree(int* begin, int* end){
        size = (int)(end - begin);
        int u, v;
        cost = 0;
        ll cu, cv;
        left.assign(2*size+9, -1);
        right.assign(2*size+9, -1);
        par.assign(2*size+9, -1);
        priority_queue<ii> pq;
        for(int i=0; i<size; i++){
            pq.push(ii(-begin[i], i));
        }
        while(pq.size() > 1){
            u = pq.top().second;
            cu = -pq.top().first;
            pq.pop();
            v = pq.top().second; cv
            = -pq.top().first; pq
            .pop();
            root = size++;
            left[root] = u; par[u] = root;
            right[root] = v; par[v] = root;
            cost += cu + cv;
            pq.push(ii(-cu-cv, root));
        }
    }
    ll getCost() { return cost; }
    int getSize() { return size; }
    void getCode(int u, char* buffer){
        vector<int> s;
        while(par[u] >= 0){
            if (left[par[u]] == u) s.push_back(0);
            else s.push_back(1);
            u = par[u];
        }
        while(!s.empty()){
            *buffer = s.back() + '0';
            buffer++; s.pop_back();
        }
        *buffer = '\0';
    }
};
```

Capítulo 4

Paradigmas

4.1 Merge Sort

Algoritmo $O(nlogn)$ para ordenar o vetor em $[a, b]$. *inv* conta o número de inversões do bubble-sort nesse trecho.

```
#include <cstdio>
#define MAXN 100009

typedef long long int ll;ll
inv;
int N, vet[MAXN], aux[MAXN];

void mergesort(int a, int b){
    if (a==b) return;
    int mid = (a+b)/2;
    mergesort(a, mid);
    mergesort(mid+1, b);
    int p=a, q=mid+1, k=a;
    while(p<=mid && q<=b){
        if (vet[p]>vet[q]){
            aux[k++]=vet[q++];
            inv += (ll)(q-k);
        }
        else aux[k++]=vet[p++];
    }
    while(p<=mid) aux[k++]=vet[p++];
    while(q<=b) aux[k++]=vet[q++];
    for(int i=a; i<=b; i++) vet[i]=aux[i];
}
```

4.2 Quick Sort

Algoritmo *Expected* $O(nlogn)$ para ordenar o vetor em $[a, b]$. É o mais rápido conhecido.

```
#include <cstdio>
#include <algorithm>
using namespace std;

void quicksort(int* arr, int l, int r){
    if (l >= r) return;
    int mid = l + (r - l) / 2;
    int pivot = arr[mid];
    swap(arr[mid], arr[l]);
    int i = l + 1, j = r;
    while (i <= j) {
        while(i <= j && arr[i] <= pivot) i++;
        while(i <= j && arr[j] > pivot) j--;
        if (i < j) swap(arr[i], arr[j]);
    }
    swap(arr[i-1], arr[l]);
    quicksort(arr, l, i-2);
    quicksort(arr, i, r);
}
```

4.3 Longest Increasing Subsequence (LIS)

$O(nlogn)$. $M[i]$ representa o índice do menor elemento tal que existe uma sequência de tamanho i que termina nele.

```
#define MAXN 100009

int LIS(int* arr, int n){ int
   M[n+1], L=1, l, h, m; M[1]
   = 0;
    for(int i=1; i<n; i++){
        if (arr[i]<arr[M[1]]){       //estritamente
             crescente
        //if (arr[i]<=arr[M[1]]){   //crescente
            M[1] = i; continue;
        }
        l = 1; h = L+1;
        while(h>l+1){
            m = (l+h)/2;
            if (arr[M[m]]<arr[i]) l = m;   //
                estritamente crescente
            //if (arr[M[m]]<=arr[i]) l = m;   //
                crescente
            else h = m;
        }
        if(h>L) L=h;
        M[h] = i;
    }
    return L;
}
```

4.4 Maximum Sum Increasing Subsequence

$O(nlogn)$. $MIS[k]$ é o maior soma de uma subsequência crescente que termina em k. $rank[i]$ é o valor do índice de $arr[i]$ quando ordenada. $A[i] = MIS[k]$, $rank[k] = i$.

```
#include <vector>
#include <algorithm>
#include <cstring>
#define MAXN 100009
using namespace std;

int comp(int a, int b){
    return max(a, b);
}

class FenwickTree { ... };

int arr[MAXN], A[MAXN];
int rank[MAXN], invrank[MAXN], N;

bool rankcomp(int a, int b){
    //if (arr[a] == arr[b]) return a<b; //crescente
    if (arr[a] == arr[b]) return a>b;    //
        estritamente crescente
    else return arr[a] < arr[b];
}

int MSIS(){
    for(int i=1; i<=N; i++) invrank[i] = i;s
    ort(invrank+1, invrank+1+N, &rankcomp);
    for(int i=1; i<=N; i++) rank[invrank[i]] = i;
    memset(&A, 0, sizeof A);
    FenwickTree ft(N);
    for(int i=1, j; i<=N; i++){
        j = rank[i];
        A[j] = arr[i] + ft.rsq(j-1);
        ft.update(j, A[j]);
    }
    return ft.rsq(N);
}
```

4.5 Problema dos Pares mais Próximos

Algoritmo $O(nlogn)$ para achar os pares mais próximos segundo a distância euclidiana em uma array de pontos no espaço.

```
#include <cstdio>
#include <algorithm>
#include <iostream>
#include <cmath>
#include <set>
#define EPS 1e-9
#define INF 1e+9
#define MAXN 100009
using namespace std;

struct point{
    double x, y;
    point() { x=y= 0.0;}
    point(double _x, double _y) : x(_x), y(_y) {}
};

double dist(point p1, point p2) {
    return hypot(p1.x - p2.x, p1.y - p2.y);
}

typedef pair<point, point> pp;

bool compx(point a, point b){
    if (fabs(a.x - b.x)<EPS) return a.y < b.y;
    else return a.x < b.x;
}

bool compy(point a, point b){
    if(fabs(a.y - b.y)<EPS) return a.x < b.x;
    else return a.y < b.y;
}

pp closestPair(point* points, point* aux, int a, int
    b){
    pp pl, pr, ans = make_pair(point(-INF, 0), point
        (0, INF));
    if (a == b) return ans;
    int mid = (a+b)/2;
    pl = closestPair(points, aux, a, mid);
    pr = closestPair(points, aux, mid+1, b);
    double dr = dist(pr.first,pr.second); double
    dl = dist(pl.first,pl.second); double d
    = min(dl, dr);
    double midx = 0.5*(points[mid].x + points[mid
        +1].x);
    int kl=0, kr;
    for(int i=a; i<=mid; i++){
        if (points[i].x >= midx - d - EPS) aux[kl++]
            =points[i];
    }
    kr = kl;
    for(int i=mid+1; i<=b; i++){
        if (points[i].x <= midx + d + EPS) aux[kr++]
            =points[i];
        else break;
    }
    sort(aux, aux+kl, &compy);
    sort(aux+kl, aux+kr, &compy);
    for(int i=0, k=kl; i<kl; i++){
        for(int j=k; j<kr; j++){
            if (aux[i].y - aux[j].y > d + EPS) k=j
                +1;
            else if (aux[j].y - aux[i].y > d + EPS)
                break;
            else if (dist(ans.first,ans.second) > di
                st(aux[i], aux[j])) ans =
                make_pair(aux[i], aux[j]);
        }
    }
    if (dr< dist(ans.first, ans.second)) ans= pr;
    if (dl < dist(ans.first,ans.second)) ans= pl;
    return ans;
}

pp closestPair(point* points, int n){
    point aux[n];
    sort(points, points+n, &compx);
    return closestPair(points, aux, 0, n-1);
}
```

4.6 Otimização de Dois Ponteiros

Reduz a complexidade de $O(n^2k)$ para $O(nk)$ de PD's da seguinte forma (e outras variantes):

$$dp[i][j] = 1 + \min_{1\le k\le i}(\max(dp[k-1][j-1], dp[i-k][j])), \quad caso\ base: \ dp[0][j], dp[i][0] \tag{4.1}$$

- $A[i][j] = k$ ótimo que minimiza $dp[i][j]$.
- É necessário que $dp[i][j]$ seja crescente em i: $dp[i][j] \le dp[i+1][j]$.
- Este exemplo é o problema dos ovos e dos prédios.

```
#include <algorithm>
using namespace std;
#define MAXN 1009
#define MAXK 19
#define INF (1<<30)

int dp[MAXN][MAXK], A[MAXN][MAXK], N, K;

void twopointer() {
    for (int i=0; i<=N; i++) dp[i][0] = INF;
    for (int j=0; j<=K; j++) dp[0][j] = 0, A[0][j] = 1;
    dp[0][0] = 0;
    for (int i=1; i<=N; i++) {
        for (int j=1; j<=K; j++) {
            dp[i][j] = INF;
            for (int k=A[i-1][j]; k<=i; k++) {
                int cur = 1 + max(dp[k-1][j-1], dp[i-k][j]);
                if (dp[i][j] > cur) {
                    dp[i][j] = cur;
                    A[i][j] = k;
                }
                if (dp[k-1][j-1] > dp[i-k][j]) break;
            }
        }
    }
}
```

4.7 Otimização de Convex Hull Trick

Reduz a complexidade de $O(n^2k)$ para $O(nklogn)$ de uma PD da seguinte forma (e outras variantes):

$$dp[i][j] = \min_{0\le k<i}(A[k] * x[i] + dp[k][j-1]), \quad caso\ base: \ dp[0][j], dp[i][0] \tag{4.2}$$

É necessário que A seja decrescente: $A[i] \ge A[i+1]$.

```
#include <vector>
#define INF (1<<30)
#define MAXN 1009
using namespace std;

typedef long long ll;
class CHTrick{ ... };

ll x[MAXN], A[MAXN], dp[MAXN][MAXN];
int N, K;

void solve() {
    for (int i=0; i<=N; i++) dp[i][0] = 0;
    CHTrick cht(false);
    for (int j=1; j<=K; j++){
        dp[0][j] = 0;
        cht.clear();
        for (int i=1; i<=N; i++){
            cht.push(A[i-1], dp[i-1][j-1]);
            dp[i][j] = cht.query(x[i]);
        }
    }
}
```

4.8 Otimização de Slope Trick

Reduz a complexidade de $O(nS^2)$ para $O(nlogn)$ da seguinte PD, onde $f[i] = \min_j(dp[i][j])$ e $opt[i] = j$ que otimiza $f[i]$:

$$dp[i][j] = \min_{k\le j}(dp[i-1][k] + |a[i]-k|), \quad caso\ base: \ dp[0][j] = max(0, a[i]-j) \tag{4.3}$$

```
#include <queue>
#include <algorithm>
using namespace std;
#define MAXN 3009

typedef long long ll;
int N;
ll a[MAXN], f[MAXN], opt[MAXN];

ll slope(){
    priority_queue<ll> pq; opt[0] = a[0]; f[0] =0;
    pq.push(a[0]);
    for (int i=1; i<N; i++){
        pq.push(a[i]);
        f[i] = f[i-1] + abs(a[i] - pq.top());
        if (a[i] < pq.top()){
            pq.pop(); pq.push(a[i]);
        }
        opt[i] = pq.top();
    }
    return f[N-1];
}
```

4.9 Otimização de Divisão e Conquista

Reduz a complexidade de $O(n^2k)$ para $O(nklogn)$ de PD's das seguintes formas (e outras variantes):

$$dp[i][j] = \min_{0 \leq k < i} (dp[k][j-1] + C[k][i]), \ caso\ base: \ dp[0][j], dp[i][0] \tag{4.4}$$

- $C[i][k]$ = custo que só depende de i e de k.
- $A[i][j] = k$ ótimo que minimiza $dp[i][j]$.

É necessário que A seja crescente ao longo de cada coluna: $A[i][j] \leq A[i+1][j]$.

```
#include <cstdio>
#include <algorithm>
using namespace std;
#define MAXN 1009
#define INF (1 << 30)

int dp[MAXN][MAXN], C[MAXN][MAXN], N, K;

void calculatedp(int min_i, int max_i, int j, int
     min_k, int max_k) {
    if (min_i > max_i) return; int
    i = (min_i + max_i) / 2; int
    ans = INF, opt;
    for (int k=min_k; k<=min(max_k, i-1); k++){
        if (ans > dp[k][j-1] + C[k][i]) {
            opt = k;
            ans = dp[k][j-1] + C[k][i];
        }
    }
    dp[i][j] = ans;
    calculatedp(min_i, i-1, j, min_k, opt);
    calculatedp(i+1, max_i, j, opt, max_k);
}

void solve(){
    for (int i=0; i<=N; i++) dp[i][0] = 0;
    for (int j=0; j<=K; j++) dp[0][j] = 0; for
    (int j=1; j<=K; j++){
        calculatedp(1, N, j, 0, N-1);
    }
}
```

4.10 Otimização de Knuth

Reduz a complexidade de $O(n^3)$ para $O(n^2)$ de PD's das seguintes formas (e outras variantes):

$$dp[i][j] = C[i][j] + \min_{i<k<j} (dp[i][k] + dp[k][j]), \ caso\ base: \ dp[i][j], j \_ i < S \tag{4.5}$$

$$dp[i][j] = \min_{i<k<j} (dp[i][k] + C[i][k]), \ caso\ base: \ dp[i][j], j \_ i < S \tag{4.6}$$

- S é uma constante definida, normalmente 1 (caso base $dp[i][i]$).
- $C[i][j]$ = custo que só depende de i e de j.
- $A[i][j] = k$ ótimo que minimiza $dp[i][j]$.

É necessário que se satisfaçam as seguintes condições:

- Desigualdade quadrangular sobre C: $C[a][c] + C[b][d] \leq C[a][d] + C[b][c], a \leq b \leq c \leq d$.
- Monotonicidade sobre C: $C[b][c] \leq C[a][d], a \leq b \leq c \leq d$.

Ou a seguinte condição:

- A crescente nas linhas e nas colunas: $A[i][j-1] \leq A[i][j] \leq A[i+1][j]$.

```
#include <cstdio>
#define MAXN 1009
#define INF (1LL << 60)

typedef long long 11;

11 dp[MAXN][MAXN], C[MAXN][MAXN];
int A[MAXN][MAXN], N, S;

void knuth() {
    11 cur;
    for (int s = 0; s < N; s++){
        for (int i=0, j; i + s<N; i++){
            j = i + s;
            if (s < S) { // Caso base
                dp[i][j] = 0;
                A[i][j] = i;
                continue;
            }
            dp[i][j] = INF;
            for (int k = A[i][j - 1]; k <= A[i + 1][j]; k++)
              {
                cur = C[i][j] + dp[i][k] + dp[k][j];
                if (dp[i][j] > cur) {
                    dp[i][j] = cur;
                    A[i][j] = k;
                }
            }
        }
    }
}
```

Capítulo 5

Grafos

5.1 DFS e BFS

```
#include <stack>
#include <queue>
using namespace std;
#define MAXN 1009

int vis[MAXN];
vector <int> adjList[MAXN];

void dfs(int u){
    vis[u]= true;
    /* atividades neste no */
    int v;
    for(int i=0; i<(int)adjList[u].size(); i++){
        v= adjList[u][i];
        if (!vis[v]) dfs(v);
    }
}

void dfsstack(){
    memset(&vis, false, sizeof vis); stack
    <int> s;
    s.push(root);
    int u, v;
    while(!s.empty()){
        u= s.top();
        s.pop();
        /* atividades neste no */
        for(int i=0; i<(int)adjList[u].size(); i++){
            v = adjList[u][i];
            if (!vis[v]){
                vis[v]= true;
                s.push(v);
            }
        }
    }
}

void bfs(){
    memset(&vis, false, sizeof vis);
    queue<int> q;
    q.push(root);
    int u, v;
    while(!q.empty()){
        u = q.top();
        q.pop();
        /* atividades neste no */
        for(int i=0; i<(int)adjList[u].size(); i++){
            v = adjList[u][i];
            if (!vis[v]){
                vis[v]= true;
                q.push(v);
            }
        }
    }
}
```

5.2 DFS Spanning Tree

```
#include <vector>
using namespace std;
#define MAXN 1009
#define UNVISITED -1
#define EXPLORED -2
#define VISITED -3

int vis[MAXN], parent[MAXN];
vector <int> adjList[MAXN];

void graphCheck(int u){ // DFS for checking graph edge pro
    perties
    vis[u] = EXPLORED;
    for (int j = 0, v; j < (int)adjList[u].size(); j
        ++) {
        v = adjList[u][j];
        if (vis[v] == UNVISITED){
            printf("␣Tree␣Edge␣(%d,␣%d)\n", u, v);
            parent[v] = u; // parent of this children
                is me
            graphCheck(v);
        }
        else if (vis[v] == EXPLORED) {
            printf("␣Back␣Edge␣(%d,␣%d)␣(Cycle)\n", u,
                v);
        }
        else if (vis[v] == VISITED)
            printf("␣Forward/Cross␣Edge␣(%d,␣%d)\n", u,
                v);
    }
    vis[u] = VISITED;
}
```

5.3 Pontos de articulação e Pontes

```
#include <vector>
#include <algorithm>
#include <cstring>
using namespace std;
#define MAXN 1009
#define UNVISITED −1

int num[MAXN], N, low[MAXN], parent[MAXN], counter,
     rootChildren, articulationVertex[MAXN], root;
vector<int> adjList[MAXN];

void tarjan(int u) {
    low[u] = num[u] = counter++;
    for (int j = 0, v; j < (int)adjList[u].size();j
        ++) {
        v = adjList[u][j];
        if (num[v] == UNVISITED) {
            parent[v] = u;
            if (u == root) rootChildren++;
            tarjan(v);
            if (low[v] >= num[u]) articulationVertex[u]
                = true;
            if (low[v] > num[u]) printf("␣Edge␣(%d,␣%d)
                ␣is␣a␣bridge\n", u, v);
            low[u] = min(low[u], low[v]);
        }
        else if (v != parent[u])
            low[u] = min(low[u], num[v]);
    }
}

int main() { co
    unter = 0;
    memset(&num, UNVISITED, sizeof num);
    memset(&low, 0, sizeof low);
    memset(&parent, 0, sizeof parent);
    memset(&articulationVertex, 0, sizeof
        articulationVertex);
    printf("Bridges:\n");
    for (int i = 0; i < N; i++)
        if (num[i] == UNVISITED) {
            root = i; rootChildren = 0; tarjan(i);
            articulationVertex[root] = (rootChildren >
                1);
        } // special case
    printf("Articulation␣Points:\n");
    for (int i = 0; i < N; i++)
        if (articulationVertex[i])
            printf("␣Vertex␣%d\n", i);
    return 0;
}
```

5.4 Ordenação Topológica

Inicializar vis como false. toposort guarda a ordenação na ordem inversa!

```
#include <vector>
using namespace std;
#define MAXN 1009

int vis[MAXN];
vector<int> adjList[MAXN];
vector<int> toposort;    //Ordem reversa !

void ts(int u) {
    vis[u] = true;
    for (int j = 0, v; j < (int)adjList[u].size();j
        ++) {
        v = adjList[u][j];
        if (!vis[v]) ts(v);
    }
    toposort.push_back(u);
}
```

5.5 Componentes Fortemente Conexos: Algoritmo de Tarjan

```
#include <vector>
#include <stack>
#include <algorithm>
#include <cstring>
using namespace std;
#define MAXN 100009
#define UNVISITED −1

int num[MAXN], vis[MAXN], component[MAXN], N, M, low
      [MAXN], counter, root, numSCC;
stack<int> S;
vector<int> adjList[MAXN];

void dfs(int u) {
    low[u] = num[u] = counter++; S
    .push(u);
    vis[u] = 1;
    int v;
    for (int j = 0; j < (int)adjList[u].size(); j++)
        {
        v = adjList[u][j];
        if (num[v] == UNVISITED) dfs(v);
        if (vis[v]) low[u] = min(low[u], low[v]);
    }
    if (low[u] == num[u]) {
        while (true) {
            v = S.top(); S.pop(); vis[v] = 0;
            component[v] = numSCC;
            if (u == v) break;
        }
        numSCC++;
    }
}

void tarjan() {
    counter = numSCC = 0;
    memset(&num, UNVISITED, sizeof num);
    memset(&vis, 0, sizeof vis); memset(&low
    , 0, sizeof low);
    for (int i = 0; i < N; i++){
        if (num[i] == UNVISITED)
            dfs(i);
    }
}
```

5.6 Componentes Fortemente Conexos: Algoritmo de Kosaraju

```
#include <vector>
#include <cstring>
#define MAXN 100009
using namespace std;

vector<int> adjList[MAXN], revAdjList[MAXN],
    toposort;
bool num[MAXN];
int component[MAXN], parent = 0, N, M, numSCC;

void revdfs(int u){
    num[u] = true;
    for(int i=0, v; i<(int)revAdjList[u].size(); i
        ++){
      v = revAdjList[u][i];
        if(!num[v]) revdfs(v);
    }
    toposort.push_back(u);
}

void dfs(int u) {
    num[u] = true;
    component[u] = parent;
    for(int i=0, v; i<(int)adjList[u].size(); i++){
      v = adjList[u][i];
        if(!num[v]) dfs(v);
    }
}

void kosaraju(){
     memset(&num, false, sizeof num);
    for (int i=0; i<N; i++)  {
        if(!num[i]) revdfs(i);
    }
    memset(&num, false, sizeof num);
   numSCC = 0;
    for (int i=N-1; i>=0; i--){
        if(!num[toposort[i]]) {
            parent = toposort[i];
            dfs(toposort[i]);
            numSCC++;
        }
    }
}
```

5.7 Caminho mínimo: Algoritmo de Dijkstra

Caminho mínimo entre dois nós. Funciona para arestas negativas sem ciclos negativos. $O(V\,logV + E)$.

```
#include <set>
#include <vector>
#include <cstring>
#include <iostream>
using namespace std;
#define MAXN 100009
#define INF (1<<30)

typedef pair<int, int> ii;
vector<ii> adjList[MAXN];
int dist[MAXN], n, m;

int dijkstra(int s, int t)
{
    for (int i=1; i<=n; i++) dist[i] = INF;
    dist[s]=0;
    set<pair<int, int> > nodes;
    nodes.insert(ii(0, s));
    while (!nodes.empty()) {
        int u = nodes.begin()->second;
        nodes.erase(nodes.begin());
        for (int i=0; i<(int)adjList[u].size(); i++){
            int v = adjList[u][i].second;
            int w = adjList[u][i].first;
            if (dist[v] > dist[u] + w){
                if (dist[v] < INF){
                    nodes.erase(ii(dist[v], v));
                }
                dist[v] =dist[u] + w;
                nodes.insert(ii(dist[v], v));
            }
        }
    }
    return dist[t];
}
```

5.8 Caminho mínimo: Algoritmo de Floyd-Warshall

Caminho mínimo em $O(V^3)$. Muito rápido de codar, pode calcular caminho mínimo para ir e voltar de cada nó.

```
#define MAXN 409

int adjMat[MAXN][MAXN], N;

void floydwarshall(){
    for (int k = 0; k < N; k++)
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                adjMat[i][j] = min(adjMat[i][j],
                    adjMat[i][k] + adjMat[k][j]);
}
```

5.9 Caminho mínimo: Algoritmo de Bellman-Ford

Caminho mínimo em grafos com ciclo negativo. $O(VE)$.

```
#include <vector >
#include <cstring >
#include <iostream>
#define MAXN 1009
using namespace std;

int dist[MAXN] , N;
typedef pair <int , int> ii;
vector<ii> adjList[MAXN];

int bellmanford(int s, int t)
    memset(&dist, 1<<20, sizeof dist);
    dist[s] = 0;
    bool hasNegativeWeightCycle = false;
    for (int i = 0, v, w; i < N; i++){
        for (int u = 0; u < N; u++) {
            for (int j = 0; j < (int)adjList[u].size();
                j++) {
                v = adjList[u][j].first;
                w = adjList[u][j].second;
                if(i==N−1 && dist[v] > dist[u] + w)
                    has Negative Weight Cycle = true;
                else dist[v] = min(dist[v], dist[u] + w)
                    ;
            }
        }
    }
    return dist[t];
}
```

5.10 Caminho mínimo: Shortest Path Faster Algorithm (SPFA)

Caminho mínimo em grafos com ciclo negativo. *Worst case* $O(VE)$, caso médio igual a dijkstra.

```
#include <queue>
#include <vector >
#include <cstring >
#include <iostream>
using namespace std;
#define MAXN 100009
#define INF (1<<30)

typedef pair <int , int> ii;
vector<ii> adjList[MAXN];
int dist[MAXN], vis[MAXN], N, M;
bool inq[MAXN];

int spfa(int s, int t){
    for(int i=0; i<=N; i++) dist[i] = INF;
    memset(&inq, false, sizeof inq);
    memset(&vis, 0, sizeof vis);
    queue<int> q;
    q.push(s); dist[s]=0;
    inq[s] = true;
    while (!q.empty()) {
        int u=q.front(); q.pop();i
        f(vis[u]> N) return −1;inq
        [u] = false;
        for (int i = 0; i < (int)adjList[u].size(); i
            ++) {
            int v = adjList[u][i].second;
            int w = adjList[u][i].first;
            if(dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                if (!inq[v]) {
                    vis[v]++; q.push(v);
                    inq[v] = true;
                }
            }
        }
    }
    return dist[t];
}
```

5.11 Árvore Geradora Mínima: Algoritmo de Kruskal

$O(ElogV)$.

```
#include <vector >
#include <algorithm >
using namespace std;

class UnionFind {
    ...
};

typedef pair <int, int> ii;
typedef long long ll;

int N, M;
vector< pair<ll, ii> > edgeList; // (weight, two
    vertices) of the edge

ll kruskal(){
    ll cost = 0;
    UnionFind UF(N); pair
    <int, ii> edge;
    sort(edgeList.begin(), edgeList.end());
    for(int i = 0; i < M; i++) {
        edge = edgeList[i];
        if(!UF. is Same Set(edge.second.first, edge.
            second.second)) {
            cost += edge.first;
            UF. union Set(edge.second.first, edge.second.
                second);
        }
    }
    return cost;
}
```

5.12 Árvore Geradora Mínima: Algoritmo de Prim

$O(E \log E)$.

```
#include <vector>
#include <queue>
#include <cstring>
#define MAXN 10009
using namespace std;

typedef long long ll;
typedef pair<int, int> ii;
vector<ii> adjList[MAXN];
int N, M;

ll prim() {
    bool taken[MAXN];
    memset(&taken, false, sizeof taken); taken
    [0] = true;
    priority_queue<ii> pq;
    ii v, front; int u, w; ll cost = 0;
    for (int j = 0; j < (int)adjList[0].size(); j++)
        {
        v = adjList[0][j];
        pq.push(ii(-v.second, -v.first));
    }
    while (!pq.empty()) {
        front = pq.top(); pq.pop();
        u = -front.second; w = -front.first;
        if (!taken[u]) {
            cost += (ll)w; taken[u] = true;
            for (int j = 0; j < (int)adjList[u].size();
                j++) {
                v = adjList[u][j];
                if (!taken[v.first]) pq.push(ii(-v.
                    second, -v.first));
            }
        }
    }
    return cost;
}
```

5.13 2-SAT

twosat() retorna se existe configuração ou não. N é o dobro do número de variáveis. Para $0 \le i < N/2$, $value[i] = x_i$ e $value[i + N/2] = !x_i$. $O(N + M)$.

```
#include <vector>
#include <stack>
#include <set>
#include <algorithm>
#include <cstring>
using namespace std;
#define MAXN 400009
#define UNVISITED -1
#define EXPLORED -2
#define VISITED -3

int num[MAXN], vis[MAXN], component[MAXN], low[MAXN
    ];
int value[MAXN], N, M, counter, root, numSCC; stack
<int> S;
vector<int> adjList[MAXN], toposort, C[MAXN];
set<int> adjComp[MAXN];

void dfs(int u) { ...}

void tarjan() { ... }

void ts(int u) {
    vis[u] = VISITED;
    set<int>::iterator it;
    for (it = adjComp[u].begin(); it!=adjComp[u].end()
        ; it++){
        if (vis[*it] == UNVISITED) dfs(*it);
    }
    toposort.push_back(u);
}

bool ffill(int u, int k) { valu
    e[u] = k;
    int v = (u>=N/2 ? u-N/2 : u+N/2);
    if (value[v] == value[u]) return false;
    if (value[v] == -1 && !ffill(v,1-k)) return false
        ;
    for (int i=0; i<(int)adjList[u].size(); i++){
        v = adjList[u][i];
        if (value[v] == 0 && k == 1) return false;
        if (value[v] == -1 && (component[u] ==
            component[v]
            || k == 1) && !ffill(v, k)) return false ;
    }
    return true;
}

bool twosat() {
    tarjan();
    for (int i=0; i<N; i++){
        C[i].clear(); adjComp[i].clear();
    }
    for (int i=0; i<N/2; i++){
        if (component[i]==component[i+N/2]) return
            false;
    }
    for (int u=0, v; u<N; u++){
        C[component[u]].push_back(u);
        for (int i=0; i<(int)adjList[u].size(); i++){
            v = adjList[u][i];
            if (component[u] != component[v])
                adjComp[component[u]].insert(component[v
                    ]);
        }
    }
    memset(&vis, UNVISITED, sizeof vis); to
    posort.clear();
    for (int i=0; i<numSCC; i++){
        if (vis[i] == UNVISITED) ts(i);
    }
    memset(&value, -1, sizeof value);
    for (int i=0, c, u; i<(int)toposort.size(); i++){
        c = toposort[i];
        for (int j=0; j<(int)C[c].size(); j++){ u
            = C[c][j];
            if (value[u] == -1 && !ffill(u, 1))
                return false;
        }
    }
    return true;
}
```

5.14 Fluxo Máximo: Algoritmo de Edmonds-Karp

Fluxo máximo de s a t. $O(VE^2)$. Usar *add* (precisa adicionar ida e volta na lista de adjacência).

```
#inc lude <queue> lude
#inc <cstring > ine
#def INF  (1<<30)
#def ine MAXN 103000
#def ine MAXM    900000
using namespace std;

int  N, M,  ned,  prv [MAXN] ,  first [MAXN];
int  cap [MAXM] ,  to [MAXM] ,  nxt [MAXM] ,  dist [MAXN];

void  init(){
     nemset (first, −1, sizeof  first); ed
    n= 0;
}
void  add (int u , int  v ,  int  f ) { to
      [ ned ]  = v ,  cap [ ned ]  = f ;
      nxt [ ned ]  =  first [ u ];
      first [ u ]  = ned++;
      to [ ned ]  = u ,  cap [ ned ]    =  0 ;
      nxt [ ned ]  =  first [ v ];
      first  [ v ]   = ned++;

}
      augment ( int  v ,  int mi    nEdge , int s ) {
int  nt  e  =  prv [ v ];
    if  ( e  ==   −1)  return  mi  nEdge ;
    int  f  =  augment ( to [ e ^ 1  ] ,  min ( minEdge ,  cap [ e ] ) ,  s )
    i      ;
     ap [ e ]  −=  f ;
     ap [ e ^ 1 ] +=  f ;
    c
    c
```

```
    retur n   f ;
}
bool bfs ( int  s ,  int  t ) {
    int u   , v ;
    memset(& dist ,  −1, si zeof dist ) ;
    dist[s]  =  0 ;
    queue <int> q ; q . push ( s ) ;
    memse t(&prv , −1,  siz qeof  prv ) ;
    while    ( !. empty () )  {
        u  = q  . front () ;  q . pop () ;
        if   (u  == t ) break    ;
        for (int  e  =   first [ u ];  e != −1; e  =  nxt [ e ] ) {
            v  =  to [ e ];
            if   ( dist [ v ]   < 0  &&  cap [ e ]  >  0 ) {
                dist [ v ] =  d ist [ u ]  +  1 ;
                q . push (v) ;
                prv [ v ]  =  e ;
            }
        }
    }
    return   dist [ t ] >= 0 ;
}
int   edmondskarp ( int s ,
    int result  =  0 ;          int t ) {
    while   ( b  fs ( s ,   t ) )  {
        result  +=  augment
    }                                   ( t ,  INF ,   s ) ;
     return  result;
}
```

5.15 Fluxo Máximo: Algoritmo de Dinic

Fluxo máximo de s a t. $O(V^2E)$. Usar *add* (precisa adicionar ida e volta na lista de adjacência).

```
#include <queue>
#include <cstring >
#define  INF  (1<<30)
#define  MAXN   103000
#define  MAXM   900000
using  namespace  std ;

int N, M,   ned ,   first [MAXN];
int cap [MAXM] ,  to [MAXM] ,  nxt [MAXM] ,   dist [MAXN];

void  init(){
    memset (first, −1, sizeof first); ned
    = 0;
}
void  add ( int   u , int   v , int   f ) { to
      [ ned ] = v , cap [ ned ] = f ; nxt [
      ned ]  =  first [ u ];
       first [ u ]  = ned++;
      to [ ned ] = u , cap [ ned ] =  0 ; nxt
      [ ned ]  =  first [ v ];
       first [ v ]   = ned++;
}

int  dfs ( int  u ,  int  f ,  int  s ,  int  t )  {
    if  ( u  ==  t )  return  f ;
    int v ,  df ;
    for ( int  e  =  first [ u ] ; e != −1;  e  =  nxt [ e ] ) {
          v =  to [ e ];
        if ( dist [ v ] ==  dist [ u ]  + 1 &&  cap [ e ]  > 0 )  {
            df =  dfs ( v ,  min ( f ,  cap [ e ] ) ,  s ,  t ) ;
            if  ( df  >  0 )  {
                cap [ e ]  −=  df ;
```

```
                cap [ e ^ 1 ]  +=  df ;
                return   df ;
            }
        }
    }
    return   0 ;
}
bool bfs ( in   t  s ,   int  t ) {
    int u , v     ;
    memset(& dist ,  −1, sizeof     dist ) ;
    dist [ s ] =  0 ;
    queue<in  t> q ; q . push ( s ) ;
    while    ( ! q . empty () ) {
        u  =  q . front () ;  q . pop () ;
        for ( int  e  =  first [ u ] ;    e != −1;  e  = nxt [ e ] ) {
            v  =  to [ e ] ;
            if  ( dist [ v ] <  0  &&  cap [ e ] >  0 ) {
                dist [ v ]  =  dist [u ]  +  1 ;
                q . push ( v ) ;
            }
        }
    }
    return   dist [ t ]  >=  0 ;
}
int dinic ( int  s ,   int  t )   { l
    int resut  =  0 ,  f ;
    while   ( b fs ( s ,   t ) ) {
         while   ( f =  dfs ( s , INF ,   s ,   t ) )   result += f ;
    }
     return  result;
}
```

5.16 Maximum Matching: Algoritmo húngaro

Emparelhamento máximo em grafo bipartido em $O(VE)$. Vértices enumerados de 1 a m em U e de 1 a n em V. Mais rápido de codar do que Hopcroft-Karp.

```
#include <cstdio>
#include <cstring>
#include <vector>
using namespace std;
#define MAXN 1009

vector<int> adjU[MAXN];
int pairU[MAXN], pairV[MAXN];
bool vis[MAXN];
int m, n;

bool dfs(int u)
{
    vis[u] = true;
    if (u == 0) return true;
    int v;
    for (int i=0; i!=(int)adjU[u].size(); ++i) { v =
        adjU[u][i];
        if (!vis[pairV[v]] && dfs(pairV[v])) {
            pairV[v] = u; pairU[u] = v;
            return true;
        }
    }
    return false;
}

int hungarian()
{
    memset(&pairU, 0, sizeof pairU);
    memset(&pairV, 0, sizeof pairV); int result = 0;
    for (int u = 1; u <= m; u++){
        memset(&vis, false, sizeof vis);
        if (pairU[u]==0 && dfs(u))
            result++;
    }
    return result;
}
```

5.17 Maximum Matching: Algoritmo de Hopcroft-Karp

Emparelhamento máximo em grafo bipartido em $O(\sqrt{V}E)$. Vértices enumerados de 1 a m em U e de 1 a n em V.

```
#include <vector>
#include <queue>
#include <cstdio>
#include <cstring>
#define INF (1<<30)
#define MAXN 1009
using namespace std;

vector<int> adjU[MAXN];
int pairU[MAXN], pairV[MAXN], dist[MAXN];
int m, n;

bool bfs()
{
    queue<int> q;
    for (int u=1; u<=m; u++){
        if (pairU[u]==0) {
            dist[u] = 0;
            q.push(u);
        }
        else dist[u] = INF;
    }
    dist[0] = INF;
    int u, v;
    while (!q.empty()) {
        u = q.front(); q.pop();
        if (dist[u] < dist[0]) {
            for (int i=0; i<(int)adjU[u].size(); ++i
                ) {
                v = adjU[u][i];
                if (dist[pairV[v]] == INF) {
                    dist[pairV[v]] = dist[u] + 1;
                    q.push(pairV[v]);
                }
            }
        }
    }
    return (dist[0] != INF);
}

bool dfs(int u)
{
    if (u == 0) return true;
    int v;
    for (int i=0; i!=(int)adjU[u].size(); ++i) { v =
        adjU[u][i];
        if (dist[pairV[v]] == dist[u]+1){
            if (dfs(pairV[v])) {
                pairV[v] = u; pairU[u] = v;
                return true;
            }
        }
    }
    dist[u] = INF;
    return false;
}

int hopcroftKarp()
{
    memset(&pairU, 0, sizeof pairU);
    memset(&pairV, 0, sizeof pairV); int result = 0;
    while (bfs()) {
        for (int u=1; u<=m; u++){
            if (pairU[u]==0 && dfs(u))
                result++;
        }
    }
    return result;
}
```

5.18 Maximum Matching: Algoritmo Blossom

Recebe a matriz de adjacência de um grafo qualquer e preenche o vetor *pair* com os devidos pares de cada nó. $O(n^4)$.

```
#include <cstdio>
#include <cstring>
#define MAXN 79

int n , pair [MAXN] , head , tail , queue [MAXN] , start ,

    finish , newBase , parent [MAXN] , base [MAXN] ;
bool adjMat [MAXN][MAXN] , inQueue [MAXN] , in Path [MAXN
    ] , in Blossom [MAXN] ;

void push ( int u ) {
    queue [ tail ++] = u ;
    inQueue [ u ] = true ;
}

int findCommonAncestor ( int u , int v ) {
    memset ( inPath , 0 , sizeof ( in Path ) ) ;
    while ( true ) {
        u = base [ u ] ; in
        Path [ u ] = true ;
        if ( u == start ) break ;
        u = parent [ pair [ u ] ] ;
    }
    while ( true ) {
        v = base [ v ] ;
        if ( in Path [ v ] ) break ;
        v = parent [ pair [ v ] ] ;
    }
    return v ;

}

void resetTrace ( int u ) {
    int v ;
    while ( base [ u ] != newBase ) { v
        = pair [ u ] ;
        in Blossom [ base [ u ] ] = 1 ; in
        Blossom [ base [ v ] ] = 1 ; u =
        parent [ v ] ;
        if ( base [ u ] != newBase ) parent [ u ] = v ;
    }
}

void blossom Contract ( int u , int v ) { newBase
    = findCommonAncestor ( u , v ) ;
    memset(inBlossom , 0, sizeof(inBlossom)) ;
    resetTrace(u) ;
    resetTrace(v) ;
    if ( base [ u ] != newBase ) parent [ u]= v ; if (
    base [ v ] != newBase ) parent [ v]= u ; for (
    u=1;u<=n ; u++)
        if ( in Blossom [ base [ u ] ] ) {
            base [ u]= newBase ;
            if ( ! inQueue [ u ] ) push ( u ) ;
        }
}


void find Augmenting Path () {
    int u , v ;
    memset ( inQueue , false , sizeof ( inQueue ) ) ;
    memset ( parent , 0 , sizeof ( parent ) ) ;
    for ( u = 1 ; u <= n ; u++) base [ u]=u ; head
    = 1; tail = 1;
    push(start);
    finish = 0;
    while ( head < tail ) {
        u = queue [ head ++];
        for ( v = 1; v <= n ; v++)
            if ( adjMat [ u ] [ v ] && base [ u ] != base [ v ] && p
                air [ u ] != v )
                if ( v == start | | ( pair [ v ] > 0 &&
                    parent [ pair [ v ] ] > 0 ) ) blossom
                    Contract ( u , v ) ;
                else if ( parent [ v ] == 0 ) {
                    parent [ v ] = u ;
                    if (pair[v] > 0) push(pair[v]);
                    else { finish = v ; return ; }
                }
    }
}


void augmentPath ( ) {
    int u , v , w;
    u = finish;
    while ( u > 0 ) {
        v = parent [ u ] ; w = pair [ v ] ;
        pair[v] = u; pair[u] = v; u = w;
    }
}

void edmondskarpmaxmatch () {
    int u ;
    memset ( pair , 0 , sizeof ( pair ) ) ;
    for ( u = 1; u <= n; u++)
        if (pair[u] == 0)
        {
            start = u ;
            findAugmentingPath();
            if(finish > 0) augmentPath () ;
        }
}
```

5.19 Corte Mínimo Global: Algoritmo de Stoer-Wagner

Calcula o corte mínimo global em $O(V^3)$: dado um grafo representado com matriz de adjacência, calcula o custo mínimo para desconectar o grafo. Esta implementação modifica o grafo, logo se precisar dele depois deve-se fazer uma cópia.

```
#include <cstdio>
#include <vector >
#include <cstring>
using namespace std ;
#define MAXN 509
#define INF (1<<30)

int n , adj Matrix [MAXN] [MAXN] ;
vector <int> bestCut ;

int mincut ( ) {
    int best Cost = INF ;
    vector <int> v [MAXN] ;
    for (int i =0; i <n; ++i)
        v[i].assign (1, i);
    int w[MAXN] , sel;
    bool exist [MAXN] , added [MAXN] ;
    memset (exist, true, sizeof exist);
    for ( int phase =0; phase<n−1; ++p hase ) {
        memset ( added , false , sizeof added ) ;
```

```
        memset(w, 0, sizeof w);
        for (int j=0, prev; j<n-phase; ++j) {
            sel = -1;
            for (int i=0; i<n; ++i) {
                if (exist[i] && !added[i] && (sel == -1 || w[i] > w[sel]))
                    sel = i;
            }
            if (j == n-phase-1) {
                if (w[sel] < bestCost) {
                    bestCost = w[sel];
                    bestCut = v[sel];
                }
                v[prev].insert(v[prev].end(), v[sel].begin(), v[sel].end());
                for (int i=0; i<n; ++i) adjMatrix[prev][i] = adjMatrix[i][prev] += adjMatrix[sel][i];
                exist[sel] = false;
            }
            else {
                added[sel] = true;
                for (int i=0; i<n; ++i) w[i] += adjMatrix[sel][i];
                prev = sel;
            }
        }
    }
    return bestCost;
}
```

5.20 Min Cost Max Flow

$O(VE(VlogV + E))$. Usar função *add* para adiconar as arestas. K é o fluxo desejado, ao final do algoritmo K retorna quanto de fluxo não foi possível repassar. Para alguns problemas, pode ser necessário duplicar os nós em nó de entrada e de saída e colocar uma aresta de capacidade infinita e custo zero entre eles. Ex: problema bidirecionado. Usa dijstra modificado com função potencial.

```
#include <cstdio>
#include <climits>
#include <cstring>
#include <queue>
#include <set>

using namespace std;
#define MAXN 103000
#define MAXM 900000
#define INF (1LL << 60)
typedef long long ll;

int N, M, ned, prv[MAXN], first[MAXN];
ll cap[MAXM], cost[MAXM], to[MAXM], nxt[MAXM], dist[MAXN], pot[MAXN], K;

void init(){
    memset(first, -1, sizeof first); ned = 0;
}

void add(int u, int v, ll f, ll c){
    to[ned] = v, cap[ned] = f;
    cost[ned] = c, nxt[ned] = first[u];
    first[u] = ned++;

    to[ned] = u, cap[ned] = 0;
    cost[ned] = -c, nxt[ned] = first[v]; first[v] = ned++;
}

bool dijkstra(int s, int t){
    memset(prv, -1, sizeof prv);
    for(int i = 0; i < N; i++) dist[i] = INF;
    set< pair<ll, int> > q;
    q.insert(make_pair(0LL, s));
    dist[s] = prv[s] = 0;
    int u, v;
    while(!q.empty()){
        u = q.begin()->second;
        q.erase(q.begin());
        for(int e = first[u]; e != -1; e = nxt[e]){
            if(cap[e]<=0) continue;
            v = to[e];
            ll new_dist = dist[u] + cost[e] + pot[u] - pot[v];
            if(new_dist < dist[v]){
                q.erase(make_pair(dist[v], v));
                dist[v] = new_dist;
                prv[v] = e;
                q.insert(make_pair(new_dist, v));
            }
        }
    }
    return prv[t] != -1;
}

ll augment(int s, int t){
    ll flow = K;
    for(int i = t; i != s; i = to[prv[i]^1]) flow = min(flow, cap[prv[i]]);
    for(int i = t; i != s; i = to[prv[i]^1]){ cap[prv[i]] -= flow;
        cap[prv[i]^1] += flow;
    }
    K -= flow;
    ll flowCost = flow*(dist[t] - pot[s] + pot[t]);
    for(int i = 0; i < N; i++)
        if(prv[i] != -1) pot[i] += dist[i];
    return flowCost;
}

ll mincostmaxflow(int s, int t){ ll flowCost = 0;
    memset(pot, 0, sizeof(pot));
    while(dijkstra(s, t)){
        flowCost += augment(s, t);
    }
    return flowCost;
}
```

5.21 Euler Tour: Algoritmo de Fleury

Computa o Euler Tour em $O(V + E)$. Grafo indexado em 0. Retorna array vazia se não existe caminho. Procura automaticamente o início caso haja algum nó ímpar. Usa a preferência passada pro caso em que todos os nós são ímpares ou 0 se não há preferência.

```
#include <vector>
#include <stack>
#define MAXN 400009
using namespace std;

vector<int> adjList[MAXN];
int N, M;

vector<int> euler(int s = 0) {
    vector<int> work(N, 0), in(N, 0), out(N, 0), tour
        ;
    for(int u = 0; u < N; u++) {
        for(int i=0; i<(int)adjList[u].size(); i++) {
            int v = adjList[u][i];
            out[u]++; in[v]++;
        }
    }
    int cntin = 0, cntout = 0;
    for (int u = 0; u < N; u++) {
        if(in[u] == out[u]+1) {
            cntout++;
            if (cntout == 2) return tour;
        }
        else if (out[u] == in[u]+1) {
            cntin++; s = u;
            if (cntin == 2) return tour;
        }
        else if (out[u] != in[u] || in[u]==0)
            return tour;
    }
    stack<int> dfs;
    dfs.push(s);
    while (!dfs.empty()) {
        int u = dfs.top();
        if (work[u] < (int)adjList[u].size()) {
            dfs.push(adjList[u][work[u]++]);
        }
        else {
            tour.push_back(u);
            dfs.pop();
        }
    }
    int n = tour.size();
    for (int i=0; 2*i<n; i++)
        swap(tour[i], tour[n-i-1]);
    return tour;
}
```

5.22 Dominator Tree

Computa a Dominator Tree em $O((V + E)logV)$. Precisa da lista de adjacência reversa. $sdom[i]$ é o semi-dominator no grafo mapeado por num, $dom[i]$ é o immediate-dominator no grafo mapeado por num. $idom[i]$ é o immediate-dominator real. $dtree$ é a dominator tree. A origem sempre é 1.

```
#include <vector>
using namespace std;
#define MAXN 200009

vector<int> adjList[MAXN], revAdjList[MAXN];
vector<int> dtree[MAXN];
int sdom[MAXN], dom[MAXN], idom[MAXN], N, M;
int dsu[MAXN], best[MAXN]; //auxiliares
int par[MAXN], num[MAXN], rev[MAXN], cnt; //dfs

int find(int x) {
    if (x == dsu[x]) return x;
    int y = find(dsu[x]);
    if(sdom[best[x]] > sdom[best[dsu[x]]]) best[x] =
        best[dsu[x]];
    return dsu[x] = y;
}

void dfs(int u) {
    num[u] = cnt; rev[cnt++] = u;
    for (int i=0; i<(int)adjList[u].size(); i++) {
        int v = adjList[u][i];
        if(num[v] >= 0) continue;
        dfs(v);
        par[num[v]] = num[u];
    }
}

void dominator() {
    for (int u = 1; u <= N; u++) {
        num[u] = -1; dtree[u].clear();
        dsu[u] = best[u] = sdom[u] = u;
    }
    cnt = 0;
    dfs(1);
    for (int j = cnt - 1, u; u = rev[j], j > 0; j--)
        {
        for (int i = 0; i<(int)revAdjList[u].size(); i
            ++) {
            int y = num[revAdjList[u][i]];
            if (y == -1) continue;
            find(y);
            if (sdom[best[y]] < sdom[j]) sdom[j] = sdom[
                best[y]];
        }
        dtree[sdom[j]].push_back(j);
        int x = dsu[j] = par[j];
        for (int i=0; i<(int)dtree[x].size(); i++) {
            int z = dtree[x][i];
            find(z);
            if (sdom[best[z]] < x) dom[z] = best[z];
            else dom[z] = x;
        }
        dtree[x].clear();
    }
    idom[1] = -1;
    for (int i = 1; i < cnt; i++) {
        if(sdom[i] != dom[i]) dom[i] = dom[dom[i]];
        idom[rev[i]] = rev[dom[i]];
        dtree[rev[dom[i]]].push_back(rev[i]);
    }
}
```

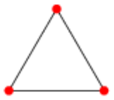
5.23 Grafos notáveis

singleton graph

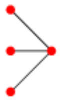
2-path graph

3-path graph

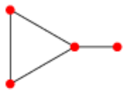
triangle graph



claw graph



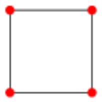
paw graph



4-path graph

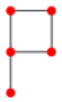
diamond graph

square graph

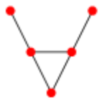


tetrahedral graph

banner graph

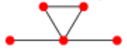


bull graph

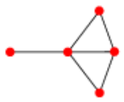


butterfly graph

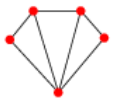
cricket graph



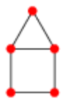
dart graph



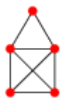
gem graph



house graph



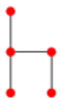
house X graph



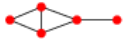
pentatope graph

5-graph 31

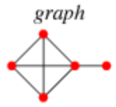
fork graph



kite graph



(4,1)-lollipop graph



5-path graph

5-star graph

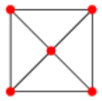
(2,3)-complete bipartite graph

5-cycle graph

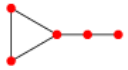
(3,2)-fan graph

Johnson solid skeleton 12

5-wheel graph



(3,2)-tadpole graph



Capítulo 6

Matemática

6.1 Aritmética Modular

MDC, MMC, euclides extendido, inverso modular $a^{-1}(modm)$, divisão modular $(a/b)(modm)$, exponenciação modular $a^b(modm)$, solução inteira da equação de Diophantine $ax + by = c$. $modMul$ calcula $(a * b)\%m$ sem overflow. Triângulo de Pascal até 10^6.

```
#include <cstdio>
#define MAXN 1000009
#define MOD 1000000007LL

template <typename T>
T gcd (T a, T b){
    return b == 0 ? a : gcd(b, a % b);
}

template <typename T>
T lcm (T a, T b) {
    return a * (b / gcd(a, b));
}

template <typename T>
T extGcd (T a, T b, T& x, T& y){
    if (b == 0){
        x = 1; y = 0;
        return a;
    }
    else {
        T g = extGcd(b, a % b, y, x);
        y -= a / b * x;
        return g;
    }
}

template <typename T>
T modInv (T a, T m){
    T x, y;
    extGcd(a, m, x, y);
    return (x % m + m) % m;
}

template <typename T>
T modDiv (T a, T b, T m){
    return ((a % m) * modInv(b, m)) % m;
}

template<typename T>
T modMul(T a, T b, T m){
    T x = 0, y = a % m;
    while (b > 0){
        if(b % 2 == 1) x = (x + y) % m;
        y = (y * 2) % m;
        b /= 2;
    }
    return x % m;
}

template<typename T>
T modExp(T a, T b, T m){
    if (b == 0) return (T)1;
    T c = modExp(a, b / 2, m);
    c = (c * c) % m;
    if (b % 2 != 0) c = (c * a) % m;
    return c;
}

template<typename T>
void diophantine(T a, T b, T c, T& x, T& y) {
    T d = extGcd(a, b, x, y);
    x *= c / d;
    y *= c / d;
}

ll fat[MAXN];
void preprocessfat(){
    fat[0] = 1;
    for(ll i=1; i<MAXN; i++){
        fat[i] = (i * fat[i - 1])%MOD;
    }
}

template<typename T>
T pascal(int n, int k, T m){
    return modDiv(fat[n], (fat[k] * fat[n-k])%m, m);
}
```

6.2 Números primos

Diversas operações com números primos. Crivo de Eristótenes, número de divisores, totiente de Euler e número de diferentes fatores primos. *isPrimeSieve* funciona em $O(\sqrt{n}/logn)$ se os fatores estiverem em *primes*.

```
#include <bitset>
#include <cstdio>
#include <vector>
#include <cstring>
using namespace std;
#define MAXN 10000009

typedef long long ll;
ll sievesize, numDiffPF[MAXN];
bitset<MAXN> bs;
vector<ll> primes;

void sieve(ll n){
    sievesize=n+1;
    bs.set();
    bs[0] = bs[1] = 0;
    for (ll i = 2; i <= sievesize; i++){
        if (bs[i]) {
            for (ll j = i * i; j <= (ll)sievesize; j += i) bs[j] = 0;
            primes.push_back(i);
        }
    }
}

bool isPrimeSieve(ll N) {
    if (N <= (ll)sievesize) return bs[N];
    for (int i = 0; i < (int)primes.size() && primes[i] * primes[i] <= N; i++)
        if (N % primes[i] == 0) return false;
    return true;
}


//O(sqrt(n))
bool isPrime(ll N){
    if (N < 0) return isPrime(-N);
    for(ll i=2; i*i <= N; i++){
        if (N % i == 0) return false;
    }
    return true;
}

//O(sqrt(n)) cortesia do Fabinho
bool isPrimeFast(ll n){
    if (n < 0) n = -n;
    if (n < 5 || n % 2 == 0 || n % 3 == 0)
        return (n == 2 || n == 3);
    ll maxP = sqrt(n) + 2;
    for (ll p = 5; p < maxP; p += 6){
        if (p < n && n % p == 0) return false;
        if (p+2 < n && n % (p+2) == 0) return false;

    }
    return true;
}

vector<ll> primeFactors(ll N) {
    vector<int> factors;
    ll PF_idx = 0, PF = primes[PF_idx];
    while (PF * PF <= N) {
        while (N % PF == 0) {
            N /= PF;
            factors.push_back(PF);
        }
        PF = primes[++PF_idx];
    }
    // special case if N is a prime
    if (N != 1) factors.push_back(N);
    return factors;
}

ll numDiv(ll N){
    ll PF_idx = 0, PF = primes[PF_idx], ans = 1; //start from ans = 1
    while (PF * PF <= N) {
        ll power = 0; // count the power
        while (N % PF == 0) { N /= PF; power++; } ans *= (power + 1); // according to the formula
        PF = primes[++PF_idx];
    }
    // (last factor has pow = 1, we add 1 to it)
    if (N != 1) ans *= 2;
    return ans;
}


ll EulerPhi(ll N){
    ll PF_idx = 0, PF = primes[PF_idx], ans = N;
    while (PF * PF <= N) {
        if (N % PF == 0) ans -= ans / PF;
        while (N % PF == 0) N /= PF;
        PF = primes[++PF_idx];
    }
    if (N != 1) ans -= ans / N;
    return ans;
}

void numDiffPf(){
    memset(numDiffPF, 0, sizeof numDiffPF);
    for (int i = 2; i < MAXN; i++)
        if (numDiffPF[i] == 0)
            for (int j = i; j < MAXN; j += i)
                numDiffPF[j]++;
}
```

6.3 Fórmula de Legendre

Dados um inteiro *n* e um primo *p*, calcula o expoente da maior potência de *p* que divide *n*! em $O(logn)$.

```
#include <cstdio>
#include <cstring>

typedef long long ll;

ll legendre(ll n, ll p){
    int ans = 0;
    ll prod = p;
    while (prod <= n) {
        ans += n/prod;
        prod *= p;
    }
    return ans;
}
```

6.4 Números de Catalan

Números de Catalan podem ser computados pela recursão:

$$Cat(n) = \frac{4n-2}{n+1}Cat(n-1) \qquad Cat(0) = 1 \tag{6.1}$$

- $Cat(n)$ = número de arvores binárias completas de $n+1$ folhas ou $2n+1$ elementos;
- $Cat(n)$ = número de combinações válidas para n pares de parêntesis;
- $Cat(n)$ = número de formas que o parentesiamento de $n+1$ elementos pode ser feito;
- $Cat(n)$ = número de triangulações de um polígono convexo de $n+2$ lados; e
- $Cat(n)$ = número de caminhos monotônicos discretos para ir de $(0, 0)$ a (n, n).

6.5 Números de Stirling de primeira espécie

Números de Stirling de primeira espécie $s(n,m), n \geq m$ podem ser calculados pela recursão $s(n,m) = s(n-1,m-1) - (n-1)s(n-1,m)$, com $s(n,n) = 1$ e $s(n,0) = 0, n > 0$

Propriedades:

- $s(n,m)$ = coeficiente de x^m em $P(x) = x(x-1)\cdots(x-n+1)$.
- $|s(n,m)|$ = número de permutações de tamanho n com exatamente m ciclos.

6.6 Números de Stirling de segunda espécie

Números de Stirling de segunda espécie $S(n,m)$, $n \geq m$ podem ser calculados pela recursão: $S(n,m) = S(n-1,m-1) + mS(n-1,m)$, com $S(n,n) = 1$ e $S(n,0) = 0$, $n > 0$, ou pelo princípio da inclusão-exclusão:

$$S(n,m) = \frac{1}{m!}\sum_{i=0}^{m}(-1)^i\binom{m}{i}(n-i)^n \tag{6.2}$$

Propriedades:

- $S(n,m)$ = número de formas de alocar n objetos em exatamente m conjuntos não vazios.
- $S(n,m)m!$ = número de funções sobrejetoras de um conjunto de n elementos em um de m elementos.

6.7 Lemma de Burnside

Seja X um conjunto e tt um conjunto de transformações de elementos de X em outros elementos de X. Para cada transformação $g \in tt$, tem-se X^g elementos $x \in X$ tal que $g(x) = x$. Incluir a transformação identidade $id(x) = x \forall x \in X$. Então o subconjunto X/tt de elementos de X que não podem ser obtidos dois a dois por transformações em tt é tal que:

$$|X/tt| = \frac{1}{|tt|}\sum_{g \in G}|X^g| \tag{6.3}$$

6.8 Algoritmo de Pollard-Rho

Retorna um fator de n, usar para $n > 9 \times 10^{13}$.

```
#include <algorithm >
using namespace std ;

template <typename T>
T gcd (T a, T b) { ... }
template<typename T>
T modMul(T a, T b, T m) { ... }

template<typename T>
T pollard(T n) {
    int i = 0, k = 2, d;
    T x = 3, y = 3;
    while (++i) {
        x = (mulmod(x, x, n) + n - 1) % n;
        d = gcd (abs (y - x), n);
        if (d != 1 && d != n) return d;
        if (i == k) y = x, k *= 2;
    }
}
```

6.9 Baby-Step Giant-Step para Logaritmo Discreto

Resolve a equação $a^x = b(mod m)$ em $O(\sqrt{m} \log m)$. Retorna -1 se não há solução.

```
#include <cstdio>
#include <map>
using namespace std;

template <typename T>
T baby (T a, T b, T m) {
    a %= m; b %=m;
    T n = (T) sqrt (m + .0) + 1;
    T an = 1;
    for (T i =0; i <n; ++i)
        an = (an * a) % m;
    map<T, T> vals;
    for (T i =1, cur=an; i <=n; ++i) {
        if (!vals.count(cur))
            vals[cur] = i;
        cur = (cur * an) % m;
    }
    for (T i =0, cur=b; i <=n; ++i) {
        if (vals.count(cur)) {
            T ans = vals[cur] * n - i;
            if (ans < m)
                return ans;
        }
        cur = (cur * a) % m;
    }
    return -1;
}
```

6.10 Código de Gray

Converte para o cógido de gray, ida $O(1)$ e volta $O(\log n)$.

```
int g (int n) {
    return n ^ (n >> 1);
}
int rev_g (int g) {
    int n = 0;
    for (; g; g>>=1)
        n ^= g;
    return n;
}
```

6.11 Triplas Pitagóricas

Todas as triplas pitagóricas (a, b, c), $a^2 + b^2 = c^2$ podem ser geradas a partir das equações:

$$a = k(m^2 - n^2), \quad b = 2kmn, \quad c = k(m^2 + n^2) \tag{6.4}$$

Triplas primiticas são geradas por $k = 1$.

- *tripleFromHypot*, $O(f\sqrt{c})$, onde f é o número de divisores de c ($f \leq 450$ para $c \leq 10^7$), retorna os possíveis pares (a, b) dado c.
- *primitiveTripleFromHypot*, $O(\sqrt{c})$, retorna os possíveis pares primitivos (a, b) dado c.
- *tripleFromSide*, $O(a)$, retorna os possíveis de pares (b, c) dado a.

```
#include <cstdio>
#include <cmath>
#include <set>
#include <iostream>
using namespace std;

typedef pair<int, int> ii;

inline void addSides(set<ii>& ans, int k, int s) {
    for (int n=1, m, ms, a, b; n*n<=s/2; n++){
        ms = s - n*n;
        m = floor(sqrt(ms));
        if (m*m == ms && m!=n) {
            a = m*m - n*n;
            b = 2*m*n;
            ans.insert(ii(k*a,k*b));
            ans.insert(ii(k*b, k*a));
        }
    }
}
set<ii> tripleFromHypot(int c){
    set<ii> ans;
    for (int k = 1, s; k*k <= c; k++){
        if (c%k != 0) continue;
        s = c/k;
        addSides(ans, s, k);
        addSides(ans, k, s);
    }
    return ans;
}
set<ii> primitiveTripleFromHypot(int c) {
    set<ii> ans;
    addSides(ans, 1, c);
    return ans;
}
set<ii> tripleFromSide(int a){
    set<ii> ans;
    for (int n = 1, m, b, c; n < a;n++){
        if (a%n != 0) continue; m
        = a*a/ n;
        if (m%2 != n%2) continue; b
        = (m-n) / 2;
        c = (m+n) / 2;
        ans.insert(ii(b, c));
    }
    return ans;
}
```

6.12 Teorema Chinês dos Restos

Resolve em $O(nlogn)$ o sistema $x = a[i] (mod\ p[i]), 0 \leq i < n, gcd(a[i], a[j]) = 1$ para todo $i \neq j$.

```
template <typename T>
T extGcd(T a, T b, T& x, T& y) { ...}

template <typename T>
T modInv(T a, T m) { ... }

template <typename T>
T modDiv(T a, T b, T m) { ... }

template<typename T>
T chinesert(T* a, T* p, int n, T m){
    T P = 1;
    for(int i=0; i<n;i++) P = (P * p[i]) % m;
    T x = 0, pp;
    for(int i=0; i<n; i++){
        pp = modDiv(P, p[i], m);
        x = (x + (((a[i] * pp) % m) * modInv(pp, p[i]))) % m;
    }
    return x;
}
```

6.13 Matrizes

```
#include <vector>
#include <cmath>
#define EPS 1e-5
using namespace std;

typedef long long ll;
typedef vector <vector<double> > matrix;

matrix operator +(matrix a, matrix b){
    int n = (int)a.size();
    int m = (int)a[0].size();
    matrix c;
    c.resize(n);
    for (int i=0; i<n; i++){
        c[i].resize(m);
        for (int j=0; j<m; j++){
            c[i][j] = a[i][j] + b[i][j];
        }
    }
    return c;
}

matrix operator *(matrix a, matrix b){
    int n = (int)a.size();
    if (a[0].size() != b.size()) printf("fail\n");
    int m = (int)b.size();
    int p = (int)b[0].size();
    matrix c;
    c.resize(n);
    for (int i=0; i<n; i++){
        c[i].assign(p, 0);
        for (int j=0; j<p; j++){
            for (int k=0; k<m; k++){
                c[i][j] += a[i][k]*b[k][j];
            }
        }
    }
    return c;
}

matrix operator *(double k, matrix a){
    int n = (int)a.size();
    int m = (int)a[0].size();
    for (int i=0; i<n; i++){
        for (int j=0; j<m;j++){ a[
          i][j] *= k;
        }
    }
    return a;
}

matrix operator -(matrix a, matrix b){
    return a + ((-1.0) * b);
}
```

6.14 Exponenciação de matrizes e Fibonacci

Calcula o n-ésimo termo de fibonacci em tempo $O(logn)$. Calcula uma matriz elevado a n em $O(m^3 logn)$.

```
matrix matrixExp(matrix a, int n){ if(
    n == 0) return id(a.size()); matrix
    c = matrixExp(a, n/2);
    c = c*c;
    if(n%2 != 0) c = c*a;
    return c;
}

matrix fibo(){
    matrix c; c.resize(2);
    c[0].assign(2, 1);
    c[1].assign(2, 1);
    c[1][1] = 0;
    return c;
}

double fibo(int n){
    matrix f = matrixExp(fibo(), n);
    return f[0][1];
}
```

6.15 Sistemas Lineares: Determinante e Eliminação de Gauss

Função para o determinante da matriz A pelo algoritmo de Chió em $O(n^3)$. $gauss(A, B)$ retorna se o sistema $Ax = B$ possui solução e executa a eliminação de Gauss em A e B.

```
bool comlumnHasNonZero(matrix & a, int j) {
    int n = (int)a.size();
    for (int i = 0; i < n; i++){
        if (fabs(a[i][j]) > EPS) return true;
    }
    return false;
}

bool lineHasNonZero(matrix &a, int i) {
    int m = (int)a[0].size();
    for (int j = 0; j < m; j++){
        if (fabs(a[i][j]) > EPS) return true;
    }
    return false;
}

bool hasNonZero(matrix & a) {
    int n = (int)a.size();
    for (int i = 0; i < n; i++){
        if (lineHasNonZero(a, i)) return true;
    }
    return false;
}

void switchColumns(matrix & a, int i, int j) {
    double tmp;
    int n = (int)a.size();
    for (int k=0; k<n; k++){
        tmp = a[k][i];
        a[k][i] = a[k][j];
        a[k][j] = tmp;
    }
}

void switchLines(matrix & a, int i, int j) {
    double tmp;
    int m = (int)a[i].size();
    for (int k=0; k<m; k++){
        tmp = a[i][k];
        a[i][k] = a[j][k];
        a[j][k] = tmp;
    }
}

bool fix(matrix & a, int i, int j) {
    int n = (int)a.size();
    int m = (int)a[0].size();
    int l = i, c = j;
    bool switched = false;
    while (l < n && !lineHasNonZero(a, l)) l++;
    if (l == n) return false;
    if (i != l) {
        switched = !switched;
        switchLines(a, i, l);
    }
    while (c < m && fabs(a[i][c])<EPS) c++;
    if (j != c) {
        switched = !switched;
        switchColumns(a, j, c);
    }
    return switched;
}

matrix id(int n) {
    matrix c; c.resize(n);
    for (int i=0; i<n; i++){
        c[i].assign(n, 0);
        c[i][i] = 1;
    }
    return c;
}

double det(matrix a) {
    int n = a.size();
    if(n == 1) return a[0][0];
    double sig = (fix(a,0,0) ? -1 : 1);
    matrix b = id(n-1);
    for (int i=0; i<n-1; i++){
        for (int j=0; j<n-1; j++){
            b[i][j] = a[i+1][j+1]*a[0][0] - a[i
                +1][0] * a[0][j+1];
        }
    }
    double d = det(b)/pow(a[0][0], n-2);
    return sig*d;
}

void lineSumTo(matrix & a, int i, int j, double c) {
    int m = (int)a[0].size();
    for (int k=0; k<m; k++){
        a[j][k] += c*a[i][k];
    }
}

void columnSumTo(matrix & a, int i, int j, double c)
    {
    int n = (int)a.size();
    for (int k=0; k<n; k++){
        a[k][j] += c*a[k][i];
    }
}

bool gauss(matrix & a, matrix & b) {
    int n = (int)a.size();
    int m = (int)a[0].size();
    double p;
    for (int i = 0, l; i < min(n, m); i++){
        l = i;
        while (l < n && fabs(a[l][i])<EPS) l++;
        if (l == n) return false; s
        witchLines(a, i, l);
        switchLines(b, i, l);
        for (int j=i+1; j<n; j++){ p
            = -a[j][i]/a[i][i];
            lineSumTo(a, i, j, p);
            lineSumTo(b, i, j, p);
        }
    }
    for (int i=min(n, m)-1; i>=0; i--){
        for (int j=0; j<i; j++){ p
            = -a[j][i]/a[i][i];
            lineSumTo(a, i, j, p);
            lineSumTo(b, i, j, p);
        }
    }
    return true;
}
```

6.16 Multiplicação de matriz esparsa

Multiplica duas matrizes em $O(n^2m)$, onde m é o mínimo do número médio de números não nulos em cada linha e coluna.

```
vector < vector <int> > adjA, adjB;

matrix sparsemult(matrix a, matrix b) {
    int n = (int)a.size();
    if (a[0].size() != b.size()) printf("fail\n");
    int m = (int)b.size();
    int p = (int)b[0].size();
    adjA.resize(n);
    for (int i=0; i<n; i++) {
        adjA[i].clear();
        for (int k=0; k<m; k++) {
            if (fabs(a[i][k]) > EPS)
                adjA[i].push_back(k);
        }
    }
    adjB.resize(p);
    for (int j=0; j<p; j++) {
        adjB[j].clear();
        for (int k=0; k<m; k++) {
            if (fabs(b[k][j]) > EPS)
                adjB[j].push_back(k);
        }
    }
    matrix c;
    c.resize(n);
    for (int i=0; i<n; i++){
        c[i].assign(p, 0);
        for (int j=0; j<p; j++){
            for (int u=0, v=0, k; u<(int)adjA[i].size()
              && v<(int)adjB[j].size();) {
                if (adjA[i][u] > adjB[j][v]) v++;
                else if (adjA[i][u] < adjB[j][v]) u++;
                else {
                    k = adjA[i][u];
                    c[i][j] += a[i][k]*b[k][j];
                    u++; v++;
                }
            }
        }
    }
    return c;
}
```

6.17 Método de Gauss-Seidel

Resolve o sistema linear iterativamente com complexidade $O(n^2logP\ REC^{-1})$. É necessário que a diagonal principal seja dominante.

```
matrix gaussSeidel(matrix& a, matrix& b, double
    PREC) {
    int n = (int)a.size();
    matrix x = b, xp = b;
    double error;
    do {
        error = 0.0;
        for (int i=0; i<n; i++) { xp
            [i][0] = b[i][0];
            for (int j=0; j<n; j++) {
                if (i < j) xp[i][0] -= a[i][j]*xp[j][0];
                if (i > j) xp[i][0] -= a[i][j]*x[j][0];
            }
            xp[i][0] /= a[i][i];
            error = max(error, fabs(xp[i][0]-x[i][0]));
        }
        x = xp;
    } while (error > PREC);
    return xp;
}
```

6.18 Eliminação de Gauss com o XOR

gaussxor retorna o valor máximo de xor que é possível se obter fazendo xor entre os elementos da array. $O(NlogS)$.

```
#include <cstdio>
#include <algorithm>
using namespace std;
#define MAXN 100009

typedef long long ll;

ll gaussxor(ll* arr, int N) {l
    l cur, sig = (1LL << 62);
    for (int j=0, t=0, i; sig > 0; sig >>= 1) {
        i=t;
        while (i<N && (arr[i] & sig)==0) i++;
        if (i >= N) continue;
        swap(arr[i], arr[t]);
        for (int i=0; i<N; i++){
            if(i!=t && (arr[i] & sig)!=0)
                arr[i] ^= arr[t];
        }
        t++;
    }
    cur = 0;
    for (int i=0; i<N; i++){
        cur = cur^arr[i];
    }
    return cur;
}
```

6.19 Fast Fourier Transform (FFT)

Usar em caso de double. Em caso de inteiro converter com $fa[i].real() + 0.5$. Não dá overflow.

```
#include <vector>
#include <cstdio>
#include <complex>
using namespace std;

typedef complex<double> base;

void fft(vector<base> &a, bool invert){
    int n = (int)a.size();
    for (int i=1, j=0; i<n; ++i) {
        int bit = n >> 1;
        for (; j>=bit; bit>>=1)
            j -= bit;
        j += bit;
        if (i < j) swap(a[i], a[j]);
    }

    for (int len=2; len<=n; len<<=1) {
        double ang = 2*M_PI/len * (invert ? -1 : 1);
        base wlen(cos(ang), sin(ang));
        for (int i=0; i<n; i+=len) { base
            w(1);
            for (int j=0; j<len/2; ++j) {
                base u = a[i+j], v = a[i+j+len/2] * w;
                a[i+j] = u + v;
                a[i+j+len/2] = u - v;
                w *= wlen;
            }
        }
    }
    if (invert)
        for (int i=0; i<n; ++i)
            a[i] /= n;
}

void convolution(vector<base> a, vector<base> b,
    vector<base> & res) {
    int n = 1;
    while(n < max(a.size(), b.size())) n <<= 1;
    n <<= 1;
    a.resize(n), b.resize(n);
    fft(a, false); fft(b, false);r
    es.resize(n);
    for(int i=0; i<n; ++i) res[i] = a[i] * b[i];
    fft(res, true);
}
```

6.20 Number Theoretic Transform (NTT)

Usar long long. Cuidado com overflow. m é o primo selecionado. O resultado é calculado $mod[m]$. Limite de $n = 2^{21}$.

```
#include <vector>
#include <cstdio>
#include <cstring>
#define MAXN 100009
using namespace std;

typedef long long ll;

template <typename T>
T extGcd(T a, T b, T& x, T& y) { ...}

template <typename T>
T modInv(T a, T m) { ... }

const ll mod[2] = {1004535809LL, 1092616193LL};
const ll root[2] = {12289LL, 23747LL};
const ll root_1[2] = {313564925LL, 642907570LL};
const ll root_pw = 1LL<<21;

void ntt(vector<ll> &a, bool invert, int m) {
    ll n = (ll)a.size();
    for (ll i=1, j=0; i<n; ++i) {
        ll bit = n>> 1;
        for (; j>=bit; bit>>=1) j -= bit;
        j += bit;
        if (i < j) swap(a[i], a[j]);
    }
    for(ll len=2,wlen;len<=n; len<<=1) { wlen
        = invert ? root_1[m] : root[m]; for (ll
        i=len; i<root_pw; i<<=1)
            wlen = (wlen * wlen % mod[m]);
        for (ll i=0; i<n; i+=len){
            for (ll j=0, w=1; j<len/2; ++j){
                ll u = a[i+j], v = a[i+j+len/2] *w %
                    mod[m];
                a[i+j] = (u+v < mod[m] ? u+v : u+v-mod[m
                    ]);
                a[i+j+len/2] = (u-v >= 0 ? u-v : u-v+mod
                    [m]);
                w = w * wlen % mod[m];
            }
        }
    }
    if(invert) {
        ll nrev = modInv(n, mod[m]);
        for (ll i=0; i<n; ++i)
            a[i] = a[i] * nrev % mod[m];
    }
}

void convolution(vector<ll> a, vector<ll> b, vector<
    ll> & res, int m){
    ll n = 1;
    while (n < max(a.size(), b.size())) n <<= 1;
    n <<= 1;
    a.resize(n), b.resize(n);
    ntt(a, false, m); ntt(b, false,m);res
    .resize(n);
    for(int i=0; i<n; ++i) res[i] = (a[i] * b[i])%mod[m
        ];
    ntt(res, true, m);
}
```

6.21 Convolução circular

Utiliza FFT/NTT para computar em $O(nlogn)$: $res[i] = \sum_{j=0}^{n-1} a[j] * b[(i-j+n)\%n]$

```
template <typename T>
void circularconvolution(vector<T> a, vector<T> b,
    vector<T> & res) {
    int n = a.size();
    b.insert(b.end(), b.begin(), b.end());
    convolution(a, b, res);
    res = vector<T>(res.begin()+n, res.begin()+(2*n))
        ;
}
```

6.22 Convolução com CRT

Utiliza o teorema chinês dos restos e duas NTT's para calcular a resposta módulo $mod[0] * mod[1] = 1,097,572,091,361,755,137$. Este número é normalmente grande o suficiente para calcular os valores exatos se as arrays originais tiverem cada elemento menor que aproximadamente 10^6 e $n \leq 2^{20}$. Implementação do teorema chinês dos restos por cortesia do IME.

```
template<typename T>
T modMul(T a, T b, T m) { ... }

// convolution mod 1,097,572,091,361,755,137
void modConv(vector<ll> a, vector<ll> b, vector<ll> & r
    es) {
    vector<ll> r0, r1;
    convolution(a, b, r0, 0);
    convolution(a, b, r1, 1);
    ll x, y, s, r, p = mod[0] * mod[1];
    extGcd(mod[0], mod[1], r, s);
    res.resize(r0.size());
    for (int i=0; i<(int)res.size(); i++) {
        res[i] = (modMul((s*mod[1]+p)%p, r0[i], p)
        + modMul((r*mod[0]+p)%p, r1[i], p) + p) % p;
    }
}
```

6.23 Convolução com Decomposição SQRT

Se os números forem menores que aproximadamente 10^6, separa a primeira metade de bits da segunda em cada array e executa 4 FFT's com números menores que aproximadamente 10^3. Isso permite a FFT complexa com double ter precisão suficiente pra calcular de forma exata. Depois basta juntar.

```
#include <cmath>
#define MOD 1000003LL
#define SMOD 1024LL // ~ sqrt(MOD)
typedef long long ll;

void sqrtConv(vector<ll> a, vector<ll> b, vector<ll> & c
    ) {
    vector<base> ca[2], cb[2], cc[2][2];
    ca[0].resize(a.size());
    ca[1].resize(a.size());
    for (int i=0; i<(int)a.size(); i++) {
        ca[0][i] = base(a[i] % SMOD, 0);
        ca[1][i] = base(a[i] / SMOD, 0);
    }
    cb[0].resize(b.size());
    cb[1].resize(b.size());
    for (int i=0; i<(int)b.size(); i++) {
        cb[0][i] = base(b[i] % SMOD, 0);
        cb[1][i] = base(b[i] / SMOD, 0);
    }
    for (int l=0; l<2; l++) for (int r=0; r<2; r++)
        convolution(ca[l], cb[r], cc[l][r]);
    c.resize(cc[0][0].size());
    for (int i=0; i<(int)c.size(); i++) {
        c[i] =
        (((ll)round(cc[1][1][i].real()))%MOD*(SMOD*
            SMOD)%MOD)%MOD +
        ((ll)round(cc[0][1][i].real()))%MOD*SMOD%MOD +
        ((ll)round(cc[1][0][i].real()))%MOD*SMOD%MOD +
        ((ll)round(cc[0][0][i].real()))%MOD;
        c[i] %= MOD;
    }
}
```

6.24 Ciclos em sequências: Algoritmo de Floyd

```
#include <iostream>
using namespace std;

typedef pair<int, int> ii;

ii floydCycleFinding(int x0) {
    // 1st part: finding k*start, hares speed is 2x tor
        toises
    int tortoise = f(x0), hare = f(f(x0)); // f(x0) i
        s the node next to x0
    while (tortoise != hare) { tortoise = f(tortoise)
        ; hare = f(f(hare)); }
    // 2nd part: finding start, hare and tortoise move at
        the same speed
    int start = 0; hare = x0;
    while (tortoise != hare) { tortoise = f(tortoise)
        ; hare = f(hare); start++; }
    // 3rd part: finding period, hare moves, tortoise stays
    int period = 1; hare = f(tortoise);
    while (tortoise != hare) { hare = f(hare); period
        ++; }
    return ii(start, period);
}
```

6.25 Bignum em C++

print imprime o número. *fix* remove os zeros à frente. *str2bignum* converte de string para para bignum. *int2bignum* gera um bignum a partir de um inteiro menor que a base. *bignum2int* só funciona se não der overflow. A divisão por inteiros só funciona para inteiros menores que a base. Soma, subtração, shift left e shift right em $O(n)$, multiplicação, divisão e resto em $O(n^2)$. Divisão e resto em uma única operação, é lenta para bases muito grandes. A subtração só funciona para $a \geq b$.

```
#include <vector>
#include <algorithm>
#include <cstring>
using namespace std;

typedef vector<int> bignum;
const int base = 1000 * 1000 * 1000;

void print(bignum& a) {
    printf("%d", a.empty() ? 0 : a.back());
    for (int i=(int)a.size()-2; i>=0; --i) { p
        rintf("%09d", a[i]);
    }
}
void fix(bignum& a) {
    while (a.size() > 1u && a.back() == 0) {
        a.pop_back();
    }
}
bool comp(bignum a, bignum b) { f
    ix(a); fix(b);
    if (a.size() != b.size()) return a.size() < b.s
        ize();
    for (int i=(int)a.size()-1; i>=0; i--){
        if (a[i] != b[i]) return a[i] < b[i ];
    }
    return false;
}
void str2bignum(char* s, bignum& a) { a.
    clear();
    for (int i=(int)strlen(s); i>0; i-=9) { s
        [i] = 0;
        a.push_back (atoi (i>=9 ? s+i-9 : s));
    }
    fix(a);
}
void int2bignum(int n, bignum& a) { a.
    clear();
    if(n== 0) a.push_back(0);
    while (n> 0) {
        a.push_back(n%base);
        n /= base;
    }
}
int bignum2int(bignum& a) {
    int ans = 0, p=1;
    for (int i=0; i<(int)a.size(); i++){
        ans += a[i] * p;
        p *= base;
    }
    return ans;
}
void sum(bignum& a, bignum& b, bignum& c) {
    int carry = 0, n = max(a.size(), b.size());
    c.resize(n);
    for (int i=0, ai, bi; i<n; i++) {
        ai = i < (int)a.size() ? a[i] :0;
        bi= i < (int)b.size() ? b[i]: 0;c
        [i] = carry + ai + bi;
        carry = c[i] / base;c
        [i] %= base;
    }
    if (carry> 0) c.push_back(carry);
    fix(c);
}
```

```
void subtract(bignum& a, bignum& b, bignu int cm & c) {
    arry = 0, n =max(a.size(), b.size(c.res));
    ize(n);
    for (int i=0, ai, bi; i<n; i++) {
        ai = i < (int)a.size() ? a[i] :0;
        bi=i < (int)b.size() ? b[i]: 0;c
        [i] =carry+ai - bi ;
        carry = c[i] < 0 ? 1 : 0;
        if (c[i] < 0) c[i] += base;
    }
    fix(c);
}
void shiftL(bignum& a, int b, bignum& c) { c
    .resize((int)a.size() + b);
    for (int i=(int)c.size()-1; i>=0; i--){
        if(i>=b) c[i] = a[i-b];
        else c[i] = 0;
    }
    fix(c);
}
void shiftR(bignum& a, int b, bignum& c) {
    if (((int)a.size()) <= b) {
        c.clear(); c.push_back(0);
        return;
    }
    c.resize((int)a.size() - b);
    for (int i=0; i<(int)c.size();i++){
        c[i] = a[i+b];
    }
    fix(c);
}

//Multiplica bignum b por int a<base
void multiply(int a, bignum& b, bignum &c ) {
    int carry = 0;
    c.resize(b.size());
    for (int i=0; i<(int)b.size() || carry; i++) {
        if (i == (int)b.size()) c.push_back(0 );
        long long cur = carry + a * 1ll * b[i];
        c[i] = int(cur % base);
        carry = int(cur / base);
    }
    fix(c);
}
void multiply(bignum a, bignum b, bignum& c) {
    int n = a.size()+b.size();
    long long carry = 0, acum;c
    .resize(n);
    for (int k=0; k<n || carry; k++) {
        if (k == n) c.push_back(0);
        acum= carry; carry = 0;
        for (int i=0, j=k; i <= k && i<(int)b.size();
            i++, j--) {
            if (j >= (int)b.size()) continue;
            acum += a[i] * 1ll * b[j];
            carry += acum / base;
            acum %= base;
        }
        c[k] = acum;
    }
    fix(c);
}
void divide(bignum& a, int b, bignum& c) {
    int carry = 0;
    c.resize(a.size());
```

```
    for (int i=(int)a.size()-1; i>=0; --i){
        long long cur = a[i] + carry * 1ll * base;
        c[i] = int(cur / b);
        carry = int(cur % b);
    }
    fix(a);
}

void divide(bignum a, bignum b, bignum& q, bignum& r)
    {
    bignum z, p;
    int2bignum(0,z);
    int2bignum(1,p);
    int2bignum(0,q);
    while(comp(b, a)){
        shiftL(p, max(1u, a.size()-b.size()), p);
        shiftL(b, max(1u, a.size()-b.size()), b);
    }
    while(true){
        while(comp(a, b) && comp(z, p)){
            shiftR(p, 1, p); shiftR(b, 1, b);
        }

        if(!comp(z, p)) break;
        subtract(a, b, a);
        sum(q, p, q);
    }
    swap(a,r);
}
```

6.26 BigInteger em Java

Sintaxe da classe BigInteger do Java.

```
import java.util.Scanner;
import java.math.BigInteger;

class Main { /* UVa 10925 - Krakovia */
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int caseNo = 1;
        while (true) {
            int N = sc.nextInt(), F = sc.nextInt();
            if (N == 0 && F == 0) break;
            BigInteger sum = BigInteger.ZERO;
            for (int i = 0; i < N; i++) {
                BigInteger V = sc.nextBigInteger();
                sum = sum.add(V);
            }
            System.out.println("Bill␣#" +
                (caseNo++) + "␣costs␣" + sum +
            ":␣each␣friend␣should␣pay␣" +
            sum.divide(BigInteger.valueOf(F)));
            System.out.println();
        }
    }
}
```

6.27 Jogo de Nim

Determina se o primerio jogador tem a estratégia em um jogo de Nim. Para o jogo de Nim simples (a partir de uma pilha de tamanho u, pode-se deixar a pilha de qualquer tamanho $v \geq 0$), a soma de Nim vale $S_1 \oplus S_2 \oplus ... \oplus S_n$. Para jogos mais complexos, deve-se montar o grafo de posições alcançáveis para cada pilha. $adjPos(i, u)$ deve retornar as posições alcançáveis a partir da posição u na pilha i. Para problemas que envolvem a separação em vários subproblemas, montar uma PD para calcular o mex de cada estado.

```
#include <vector>
#include <algorithm>
using namespace std;
#define MAXN 109
#define MAXM 1009

int N, S[MAXN];

vector<int> adjPos(int i, int u){
    ...
}

bool nim(){
    int ans = 0, g[MAXM];
    vector<int> F, G;
    for(int i=0; i<N; i++){ g
        [0] = 0;
        for(int u=1; u<=S[i]; u++){ F
            = adjPos(i, u);
            G.clear();
            for(int j=0; j<(int)F.size(); j++){
                G.push_back(g[F[j]]);
            }
            sort(G.begin(), G.end());
            g[u] = 0;
            for(int j=0; j<(int)G.size() && G[j] <= g[u
                ]; j++){
                if (G[j] == g[u]) g[u]++;
            }
        }
        ans ^= g[S[i]];
    }
    return ans != 0;
}

//Mex dynamic programming

void stevehalim(){
    for(int n=1; n<MAXN; n++){
        if (n<=2) { mex[n] = 0; continue; }
        set<int> jog;
        for(int i=3; i<=n-2; i++){
            jog.insert(mex[i-1]^mex[n-i]);
        }
        int cnt = 0;
        while(jog.count(cnt)) cnt++;
        mex[n] = cnt;
    }
    //for(int n=1; n<MAXN; n++)
    //    printf("%d,", mex[n]);
}
```

Capítulo 7

Processamento de Strings

7.1 Knuth-Morris-Pratt (KMP)

String matching em $O(n + m)$. Inicializar a classe com a string a ser procurada e usar match para receber o vector com as posições de matches.

```
#include <cstdio>
#include <string>
#include <vector>
#include <cstring>
using namespace std;
#define MAXN 100009

class KMP{
private:
    char P[MAXN];
    int m, n, b[MAXN];
public:
    KMP(const char * _P) {
        strcpy(P, _P);
        b[0] = -1;
        m = strlen(P);
        for (int i = 0, j = -1; i < m;) {
            while (j >= 0 && P[i] != P[j]) j = b[j];
            i++; j++;
            b[i] = j;
        }
    }
    vector<int> match(const char * T) {
        n = strlen(T);
        vector<int> ans;
        for (int i=0, j=0; i < n;) {
            while (j >= 0 && T[i] != P[j]) j = b[j];
            i++; j++;
            if (j == m) {
                ans.push_back(i - j);
                j = b[j];
            }
        }
        return ans;
    }
};
```

7.2 Rabin-Karp

String matching em $O(n+m)$. Retorna o vector com as posições de matches. AVISO: Usa *strncmp*, não *strcmp*. CUIDADO: Roda em $O(nm)$ para casos do tipo $AAAAAAAAAAAAAA - AAAA$ ou $ABABABABABABABAB - ABABABA$.

```
#include <cstdio>
#include <cstring>
#include <vector>
using namespace std;
#define d 256 //Tamanho do alfabeto
#define MOD 40031

vector<int> rabinkarp(const char * P, const char * T) {
    vector<int> ans;
    int n = strlen(T);
    int m = strlen(P);
    int p = 0, t = 0, h = 1;
    for (int i = 0; i < m-1; i++) h = (h*d)%MOD;
    for (int i = 0; i < m; i++){
        p = (d*p + P[i])%MOD;
        t = (d* t + T[i])%MOD;
    }
    for (int i = 0; i <= n-m; i++){
        if (p == t && strncmp(T+i, P, m) == 0){
            ans.push_back(i);
        }
        if (i+m < n){
            t = (d * (t - T[i] * h) + T[i+m])%MOD;
            if (t < 0) t += MOD;
        }
    }
    return ans;
}
```

7.3 Repetend: menor período de uma string

Menor período da string em $O(n)$.

```
#include <cstdio>
#include <cstring>
#define MAXN 100009

int repetend(char * s) { int
    n = strlen(s); int
    nxt[n+1];
    nxt[0] = −1;
    for (int i = 1; i <= n; i++) {
        int j = nxt[i − 1];
        while (j >= 0 && s[j] != s[i − 1])
            j = nxt[j];
        nxt[i] = j + 1;
    }
    int a = n − nxt[n];
    if (n % a == 0)
        return a;
    return n;
}
```

7.4 Suffix Array

Algoritmo $O(nlogn)$ para computar a Suffix Array SA.

```
#define MAXN 100009
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

char str[MAXN];
int n; // the length of input string
int RA[MAXN], tempRA[MAXN];
int SA[MAXN], tempSA[MAXN]; //SA: suffix array
int c[MAXN];

void countingSort(int k) { // O(n)
    int i, sum, maxi = max(300, n);
    memset(c, 0, sizeof c);
    for (i = 0; i < n; i++)  c[i + k < n ? RA[i + k]
        : 0]++;
    for (i = sum = 0; i < maxi; i++) {
        int t = c[i];
        c[i] = sum;
        sum += t;
    }
    for (i = 0; i < n; i++)  tempSA[c[SA[i]+k < n ?
        RA[SA[i]+k] : 0]++] = SA[i];
    for (i = 0; i < n; i++)  SA[i] = tempSA[i];
}

void constructSA() { //O(nlogn)
    int i, k, r;
    for (i = 0; i < n; i++) RA[i] = str[i];
    for (i = 0; i < n; i++) SA[i] = i;
    for (k = 1; k < n; k <<= 1) {
        countingSort(k);
        countingSort(0);
        tempRA[SA[0]] = r = 0;
        for (i = 1; i < n; i++)  tempRA[SA[i]] =
            (RA[SA[i]] == RA[SA[i −1]] && RA[SA[i]+k] ==
                RA[SA[i −1]+k]) ? r : ++r;
        for (i = 0; i < n; i++)  RA[i] = tempRA[i];
        if (RA[SA[n − 1]] == n−1) break;
    }
}
```

7.5 Suffix Array: string matching

String matching em $O(mlogn)$. Requer construção da SA.

```
ii stringMatching(char * P) {
    int lo = 0, hi = n−1, mid = lo;
    while (lo < hi) { // find lower bound
        mid = (lo + hi) / 2;
        int res = strncmp(str + SA[mid], P, m);
        if (res >= 0) hi = mid;
        else lo = mid + 1;
    }
    if (strncmp(str + SA[lo], P, m) != 0) return ii(
        −1, −1);
    ii ans; ans.first = lo;
    lo = 0; hi = n − 1; mid = lo;
    while (lo < hi) { // if lower bound is found,
        find upper bound
        mid = (lo + hi) / 2;
        int res = strncmp(str + SA[mid], P, m);
        if (res > 0) hi = mid;
        else lo = mid + 1;
    }
    if (strncmp(str + SA[hi], P, m) != 0) hi −−; //s
        pecial case
    ans.second = hi;
    return ans;
} // return lower/upperbound as first/second item of the pair,
    respectively
```

7.6 Suffix Array: Longest Common Prefix

LCP [*i*] guarda o tamanho do maior prefixo comum entre *SA*[*i*] e *SA*[*i* − 1].

```
int Phi[MAXN];
int LCP[MAXN], PLCP[MAXN];

// Longest Common Prefix
 SA[i] and SA[i -1]
void computeLCP() { //O(n)
    int i, L;
    Phi[SA[0]] = -1;
    for (i = 1; i < n; i++)  Phi[SA[i]] = SA[i-1];
    for (i = L = 0; i < n; i++) {
        if (Phi[i] == -1) {
            PLCP[i] = 0; continue;
        }
        while (str[i+L]==str[Phi[i]+L]) L++;
        PLCP[i] = L;
        L = max(L-1, 0);
    }
    for (i = 0; i < n; i++) LCP[i] = PLCP[SA[i]];
}
```

7.7 Suffix Array: Longest Repeated Substring

É o valor do maior LCP.

```
int LRS(){  //O(n)
    int lrs = 0;
    for (int i=0; i<n; i++) if (LCP[i] > LCP[lrs]) lrs
        = i;
    return lrs;
}
```

7.8 Suffix Array: Longest Common Substring

Retorna o tamanho da maior substring comum a *str*1 e *str*2.

```
int LCS(char* str1, char* str2){
    int n1 = strlen(str1);
    strcpy(str, str1); strcat(str, "$");
    strcat(str, str2); strcat(str, "#");
    n = strlen(str);
    constructSA(); computeLCP();
    int ans = 0;
    for (int i=1; i<n; i++){
        if (LCP[i] > LCP[ans] &&
            ((SA[i]<n1 && SA[i-1]>n1) || (SA[i]>n1 &&
                SA[i-1]<n1))){
            ans = i;
        }
    }
    return LCP[ans];
}
```

7.9 Longest Palindromic Substring: Algoritmo de Manacher

Usado para achar a maior substring palíndromo. A função *manacher* calcula a array *L*. *L*[*i*] é o máximo valor possível tal que $str[i + j] = str[i - j]$, $0 \leq j \leq L[i]$. Pra calcular os palíndromos pares, basta adicionar '|' entre todos os caracateres e calcular o maior valor de *L* da string.

```
#include <cstdio>
#include <cstring>
#include <algorithm>
#define MAXN 3009
using namespace std;

void manacher(char* str, int* L) {
    int n = strlen(str), c = 0, r = 0;
    for (int i = 0; i < n; i++) {
        if(i < r && 2*c >= i) L[i] = min(L[2*c-i], r
            -i);
        else L[i] = 0;
        while (i-L[i]-1 >= 0 && i+L[i]+1 < n &&
            str[i-L[i]-1] == str[i+L[i]+1]) L[i]++;
        if(i+L[i]>r) { c=i; r=i+L[i]; }


    }
}

int LPS(char* text){
    int n=2*strlen(text)+1;
    char temp[n+1];
    for (int i=0, k=0; text[i]; i++) { temp
        [k++]='|'; temp[k++]=text[i];
    }
    temp[n-1]='|'; temp[n]='\0';
    int L[n], ans=1;
    manacher(temp, L);
    for (int i=0; i<n; i++)
        ans = max(ans, L[i]);
    return ans;
}
```

7.10 Aho Corasick

Resolve o problema de achar ocorrências de um dicionário em um texto em $O(n)$, onde n é o comprimento do texto. Pré-processamento: $O(m)$, onde m é a soma do número de caracteres de todas as palavras do dicionário. Cuidado: o número de matches pode ser $Worst\ case\ O(n^2)$! Guardar apenas o número de matches, se for o que o problema pedir. Pode ser usada pra achar uma submatriz dentro de uma matriz (usar pra achar a ocorrência de cada linha dentro da outra, depois usar KMP nas colunas com a array de ids).

```
#include <cstdio>
#include <cstring>
#include <map>
#include <queue>
#include <vector>
using namespace std;
#define ALFA 256
#define MOD 1000000009
#define MAXK 70
#define MAXN 100009

typedef pair<int, int> ii;

struct node {
    node *fail, *next[ALFA];
    vector<char> adj;
    vector<int> pats;
    int nid;
    node(int _nid) {
        fail = NULL;
        nid = _nid;
        memset(&next, 0, sizeof next);
    }
};

class AhoCorasick
{
private:
    node *trie;
    map<int, int> sizes;
    int size;
    node *suffix(node *x, char c) {
        while (x != trie && x->next[c] == 0) x = x->fail;
        return (x->next[c] ? x->next[c] : trie);
    }
public:
    AhoCorasick() { trie = new node(0); size = 1;}
    void clear() {
        node *x, *y;
        queue<node*> q; q.push(trie);
        while (!q.empty()) {
            x = q.front(); q.pop();
            for (int i = 0; i < (int)x->adj.size();i++) {
                y = x->next[x->adj[i]];
                if (y != NULL && y != trie) q.push(y);
            }
            delete x;
        }
        trie = new node(0); size = 1;
    }
    void setfails() {
        queue<node*> q;
        node *x, *y;
        for (int i = 0; i < ALFA; i++) {
            x = trie->next[i];
            if (x != NULL && x != trie) {
                x->fail = trie;
                q.push(x);
            }
        }
        while (!q.empty()) {
            x = q.front(); q.pop();
            for (int i = 0; i < (int)x->adj.size();i++) {
                y = x->next[x->adj[i]];
                y->fail = suffix(x->fail, x->adj[i]);
                y->pats.insert(y->pats.end(),
                    y->fail->pats.begin(), y->fail->pats.end());
                q.push(y);
            }
        }
    }
    void insert(const char *s, int id) {
        int len = strlen(s);
        sizes[id] = len;
        node *x = trie, *y;
        for (int i = 0; i < len; i++) {
            y = x->next[s[i]];
            if (y == NULL || y == trie) {
                x->next[s[i]] = new node(size++);
                x->adj.push_back(s[i]);
            }
            x = x->next[s[i]];
        }
        x->pats.push_back(id);
    }
    vector<ii> match(const char *s) {
        node *x = trie;
        int len = strlen(s), id;
        vector<ii> ans;
        for (int i = 0; i < len; i++) {
            x = suffix(x, s[i]);
            for (int j = 0; j < (int)x->pats.size(); j++) {
                id = x->pats[j];
                ans.push_back(ii(id, i - sizes[id]));
            }
        }
        return ans;
    }

//Dynamic Programming (size left, appeared, node id)
private:
    int dp[MAXK][MAXK][MAXN];
    inline int nOnes(int mask) {
        int ans = 0;
        while (mask > 0) {
            ans++;
            mask -= (mask & -mask);
        }
        return ans;
    }
    int DP(const int s, const int mask, node *x, const int K) {
        if (x == NULL) return 0;
        if (dp[s][mask][x->nid] >= 0) return dp[s][mask][x->nid];
        int nmask = mask;
        for (int i = 0; i < (int)x->pats.size(); i++) {
            nmask |= (1 << x->pats[i]);
        }
```

```
        if (nOnes(nmask) > K) return dp[s][mask][x->
            nid] = 0;
        if (s == 0) return dp[s][mask][x->nid] = 1;
        int ans = 0;
        for (char c = 'a'; c <= 'z'; c++) {
            ans = (ans + DP(s - 1, nmask, suffix(x, c),
                K)) % MOD;
        }
        return dp[s][mask][x->nid] = ans;
    }

public:
    int DP(int sz, int K) {
        return DP(sz, 0, trie, K);
    }
    void initDP() {
        for (int i = 0; i < MAXK; i++)
            for (int j = 0; j < MAXK; j++)
                for (int s = 0; s < size; s++) dp
                    [i][j][s] = -1;
    }
};
```

7.11 Longest Common Subsequence

Técnica com PD para calcular a maior subsequência comum a duas strings. Chamar *DP* (0, 0). $O(n^2)$, pode ser feito em $O(nlogn)$ com Suffix Array.

```
#include <cstdio>
#define MAXN 109

int dp[MAXN][MAXN], n, m;
char str1[MAXN], str2[MAXN];

int DP(int i, int j) {
    if (dp[i][j]>=0) return dp[i][j];
    if (i==n && j==m) return dp[i][j] = 0;
    if (i==n) return dp[i][j] = DP(i, j+1);
    if (j==m) return dp[i][j] = DP(i+1,j);
    dp[i][j] = DP(i+1,j);
    if (DP(i,j+1)>dp[i][j]) dp[i][j]=dp[i][j+1];
    if (str1[i]==str2[j] && 1+DP(i+1,j+1)>dp[i][j])
        dp[i][j]=1+dp[i+1][j+1];
    return dp[i][j];
}
```

7.12 Função Z e Algoritmo Z

Função Z é uma função tal que $z[i]$ é máximo e $str[j] = str[i + j]$, $0 \leq j \leq z[i]$. O algoritmo Z computa todos os $z[i]$, $0 \leq i < n$, em $O(n)$. $z[0] = 0$.

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

void zfunction(char* s, int* z) {
    int n = strlen(s);
    fill(z, z+n, 0);
    for (int i=1, l=0, r=0; i<n; ++i) {
        if (i <= r)
            z[i] = min(r-i+1, z[i-l]);
        while (i+z[i] < n && s[z[i]] == s[i+z[i]])
            ++z[i];
        if (i+z[i]-1 > r)
            l = i, r = i+z[i]-1;
    }
}
```

Capítulo 8

Geometria Computacional

8.1 Ponto 2D

Ponto com double em 2D com algumas funcionalidades: distância, produto interno, produto vetorial (componente z), teste counter-clockwise, teste de pontos colineares, rotação em relação ao centro do plano, projeção de u sobre v.

```
#include <cmath>
#include <vector >
using namespace std;
#define EPS 1e-9

struct point{
    double x, y;
    point() { x=y= 0.0;}
    point(double _x, double _y) : x(_x), y(_y) {}
    double norm() { return hypot(x, y); }
    point normalized() {
        return point(x,y) * (1.0/norm());
    }
    double angle() { return atan2(y, x); }
    double polarAngle() {
        double a = atan2(y, x);
        return a < 0 ? a + 2*M_PI : a;
    }
    bool operator < (point other) const{
        if (fabs(x - other.x) > EPS)
            return x < other.x;
        else return y < other.y;
    }
    bool operator == (point other) const {
        return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS));
    }
    point operator +(point other) const{
        return point(x + other.x, y + other.y);
    }
    point operator -(point other) const{
        return point(x - other.x, y - other.y);
    }
    point operator *(double k) const{
        return point(x*k, y*k);
    }
};

double dist(point p1, point p2) {
    return hypot(p1.x - p2.x, p1.y - p2.y);

}

double inner(point p1, point p2) {
    return p1.x* p2.x + p1.y* p2.y;
}

double cross(point p1, point p2) {
    return p1.x* p2.y - p1.y* p2.x;
}

bool ccw(point p, point q, point r) {
    return cross(q-p,r-p) > 0;
}

bool collinear(point p, point q, point r) {
    return fabs(cross(p-q, r-p)) < EPS;
}

point rotate(point p, double rad) {
    return point(p.x * cos(rad) - p.y * sin(rad),
    p.x * sin(rad) + p.y * cos(rad));
}

double angle(point a, point o, point b) {
    return acos(inner(a-o, b-o) / (dist(o,a)*dist(o,b)));
}

point proj(point u, point v){
    return v*(inner(u,v)/inner(v,v));
}

bool between(point p, point q, point r) {
     return collinear(p, q, r) && inner(p - q, r - q) <= 0;
}

int leftmostIndex(vector<point> &P){
    int ans = 0;
    for (int i=1; i<(int)P.size();i++){
        if (P[i] < P[ans]) ans = i;
    }

    return ans;
}
```

8.2 Linha 2D

Algumas funções de reta e segmento de reta no plano 2D: dois pontos para reta, projeção e distância ponto-reta e lonto-segmento de reta. Reta representada da forma $ax + by + c = 0$. Se possivel fazemos $b = 1$.

```
struct line{
   double a, b, c;
   line() { a = b = c =NAN; }
   line(double _a, double _b, double _c) : a(_a), b(
      _b), c(_c) {}
};

line pointsToLine(point p1, point p2) {
   line l;
   if (fabs(p1.x - p2.x) < EPS && fabs(p1.y - p2.y)
      < EPS) {
      l.a = l.b = l.c =NAN;
   }
   else if (fabs(p1.x - p2.x) <EPS){
      l.a = 1.0; l.b = 0.0; l.c =-p1.x;
   }
   else {
      l.a = -(p1.y - p2.y) / (p1.x - p2.x);
      l.b = 1.0;
      l.c= -(l.a * p1.x) - p1.y;
   }
   return l;
}

bool areParallel(line l1, line l2) {
   return (fabs(l1.a - l2.a) < EPS) && (fabs(l1.b -
      l2.b)<EPS);
}
bool areSame(line l1, linel2) {
   return areParallel(l1, l2) && (fabs(l1.c - l2.c)
      < EPS);

}

point intersection(line l1, line l2) {
   if (areParallel(l1, l2)) return point(NAN, NAN);
   point p;
   p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b
      - l1.a * l2.b);
   if (fabs(l1.b) >EPS) p.y= -(l1.a * p.x+ l1.c);
   else p.y= -(l2.a * p.x+ l2.c);
   return p;
}

point projPointToLine(point u, line l) {
   point a, b;
   if (fabs(l.b - 1.0) <EPS) {
      a = point(- l.c / l.a, 0.0);
      b= point(- l.c / l.a, 1.0);
   }
   else {
      a = point(0, -l.c / l.b);
      b= point(1, -(l.c + 1.0) / l.b);
   }
   return a + proj(u - a, b - a);
}

double distToLine(point p, line l) {
   return dist(p, projPointToLine(p, l));
}

point closestToLineSegment(point p, point a, point b
   ) {
   double u= inner(p-a, b-a) / inner(b-a, b-a);
   if (u< 0.0) return a;
   if (u> 1.0) return b;
   return a + ((b - a)*u);
}


double distToLineSegment(point p, point a, point b)
   {
   return dist(p, closestToLineSegment(p, a, b));
}
```

8.3 Círculo 2D

```
struct circle{
   pointc;
   double r;
   circle(){ c = point(); r = 0; }
   circle(point _c, double _r) : c(_c), r(_r) {}
   double area() { return M_PI*r*r; }
   double chord(double rad) { return  2*r*sin(rad
      /2.0);}
   double sector(double rad) { return 0.5 * rad * area()

      /M_PI; }
   bool intersects(circle other){
      return dist(c, other.c) < r + other.r;
   }
   bool contains(pointp){ return dist(c, p) <= r +
      EPS; }
   bool isEnclosingCircle(vector<point> &p, int n){
      for(int i =0; i <n; i ++){
         if (!contains(p[i])) return false;
      }
      return true;
   }
};

circle circumcircle(point a, point b, point c){
   circle ans;
   point u=point((b-a).y, -(b-a).x);
   point v = point((c-a).y, -(c-a).x);
   point n = (c-b)*0.5;
   double t=cross(u,n)/cross(v,u);
   ans.c = ((a+c)*0.5) + (v*t);
   ans.r = dist(ans.c, a);
   return ans;
}


int insideCircle(point p, circle c) {
   if (fabs(dist(p , c.c) - c.r)<EPS) return 1;
   else if (dist(p , c.c) < c.r) return 0;
   else return 2;
} //0 = inside/1 = border/2 = outside

circle incircle( point p1, point p2, point p3 ){
    double m1=dist(p2, p3);
    double m2=dist(p1, p3);
    double m3=dist(p1, p2);
    point c = (p1*m1+p2*m2+p3*m3)*(1/(m1+m2+m3));
    double s = 0.5 * (m1+m2+m3);
    double r= sqrt(s * (s-m1) * (s-m2) * (s-m3))/ s;
    return circle(c, r);
}
```

8.4 Triângulo 2D

```
struct triangle{
    point a, b, c;
    triangle() { a = b = c = point(); }
    triangle(point _a, point _b, point _c): a(_a), b(_b), c(_c) {}
    double perimeter() { return dist(a,b) + dist(b,c) + dist(c,a); }
    double semiPerimeter() { return perimeter()/2.0; }
    double area() {
        double s = semiPerimeter(), ab = dist(a,b), bc = dist(b,c), ca = dist(c,a);
        return sqrt(s * (s-ab) * (s-bc) * (s-ca));
    }
    double rInCircle() {
        return area()/semiPerimeter();
    }
    circle inCircle() {
        return incircle(a,b,c);

    }
    double rCircumCircle() {
        return dist(a,b)*dist(b,c)*dist(c,a)/(4.0*area());
    }
    circle circumCircle(){
        return circumcircle(a,b,c);
    }
    int isInside(point p){
        double u = cross(b-a,p-a)*cross(b-a,c-a);
        double v = cross(c-b,p-b)*cross(c-b,a-b);
        double w = cross(a-c,p-c)*cross(a-c,b-c);
        if (u > 0.0 && v > 0.0 && w > 0.0) return 0;
        if (u < 0.0 || v < 0.0 || w < 0.0) return 2;
        else return 1;
    } //0 = inside/ 1 = border/ 2 = outside
};

double rInCircle(point a, point b, point c) {
    return triangle(a,b,c).rInCircle();
}

double rCircumCircle(point a, point b, point c) {
    return triangle(a,b,c).rCircumCircle();
}


int isInsideTriangle(point a, point b, point c, point p){
    return triangle(a,b,c).isInside(p);
} //0 = inside/ 1 = border/ 2 = outside
```

8.5 Polígono 2D

Algumas funcionalidades do polígono 2D. IMPORTANTE: o último ponto de P é igual ao primeiro.

```
#include <vector>
#include <algorithm>
using namespace std;

point lineIntersectSeg(point p, point q, point A, point B) {
    double a = B.y - A.y;
    double b = A.x - B.x;
    double c = B.x * A.y - A.x * B.y; double u = fabs(a * p.x + b * p.y + c); double v = fabs(a * q.x + b * q.y + c);
    return point((p.x * v + q.x * u) / (u+v), (p.y * v + q.y * u) / (u+v));
}

typedef vector<point> polygon;

double perimeter(polygon & P) {
    double result = 0.0;
    for (int i = 0; i < (int)P.size()-1; i++) result += dist(P[i], P[i+1]);
    return result;
}
double signedArea(polygon & P) {
    double result = 0.0;
    for (int i = 0; i < (int)P.size()-1; i++){
        result += cross(P[i], P[i+1]);
    }
    return result/2.0;
}
double area(polygon & P) {
    return fabs(signedArea(P));
}
bool isConvex(polygon & P) {
    int sz = (int)P.size();
    if (sz <= 3) return false;
    bool isLeft = ccw(P[0], P[1], P[2]);
    for (int i = 1; i < sz-1; i++){
        if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)
            return false;
    }
    return true;
}
bool inPolygon(polygon &P, point p){
    if(P.size() == 0u) return false;
    double sum = 0.0;
    for (int i = 0; i < (int)P.size()-1; i++) {
        if (ccw(p, P[i], P[i+1])) sum += angle(P[i],p, P[i+1]);
        else sum -= angle(P[i], p, P[i+1]);
    }
    return fabs(fabs(sum) - 2*M_PI) < EPS;
}
polygon make_polygon(vector<point> P) {
    if (!P.empty() && !(P.back() == P.front()))
        P.push_back(P[0]);
    if (signedArea(P) < 0.0){
        for (int i = 0; 2*i < (int)P.size(); i++){ swap(P[i], P[P.size()-i-1]);
        }
    }
    return P;
}
polygon cutPolygon(polygon P, point a, point b) {
    vector<point> R;
    double left1, left2;
    for (int i = 0; i < (int)P.size(); i++) {
        left1 = cross(b-a, P[i]-a);
        if (i != (int)P.size()-1) left2 = cross(b-a, P[i+1]-a);
        else left2 = 0;
        if (left1 > -EPS) R.push_back(P[i]);
        if (left1 * left2 < -EPS)
            R.push_back(lineIntersectSeg(P[i], P[i+1], a, b));
    }
    return make_polygon(R);
}
```

8.6 Convex Hull

Dado um conjunto de pontos, retorna o menor polígono que contém todos os pontos. Retorna o primeiro ponto repetido.

```
#include <algorithm >
using namespace std; p

oint pivot(0, 0);

bool angleCmp(point a, point b){
    if (collinear(pivot, a, b)) return dist(pivot, a)
         < dist(pivot, b);
    double d1x = a.x-pivot.x, d1y = a.y-pivot.y;
    double d2x = b.x-pivot.x, d2y = b.y-pivot.y;
    return atan2(d1y, d1x) - atan2(d2y, d2x) < 0;
}

vector<point> convexHull(vector<point> P) {
    int i, j, n = (int)P.size();
    if (n <= 3) {
        if (!(P[0] == P[n - 1])) P.push_back(P[0]);
        return P;
    }
    int P0 = leftmostIndex(P);
    swap(P[0], P[P0]);
    pivot = P[0];
    sort(++P.begin(), P.end(), angleCmp);
    vector<point> S;
    S.push_back(P[n-1]);
    S.push_back(P[0]);
    S.push_back(P[1]);
    i = 2;
    while (i < n) {
        j = (int)S.size() -1;
        if (ccw(S[j - 1], S[j], P[i])) S.push_back(P[i
            ++]);
        else S.pop_back();
    }
    return S;
}
```

8.7 Ponto dentro de polígono convexo

Dado um polígono convexo com o primeiro ponto igual ao último e um novo ponto q, verifica se o ponto está dentro (inclui borda) em $O(logn)$. O ponto índice 1 (pivot) deve ser um pivô válido para não quebrar *atan2*. Pode usar o *vector* retornado direto da função *convexHull*.

```
bool query(vector<point> &CH, point q) {
    int i = 2, j = CH.size() -1, m;

    pivot = CH[1];
    while (j > i+1){
        int m = (i+j) / 2;
        if (angleCmp(q, CH[m])) j = m;
        else i = m;
    }

    return isInsideTriangle(pivot, CH[i], CH[j], q)
         != 2;
}
```

8.8 Soma de Minkowski

Determina o polígono que contorna a soma de Minkowski de duas regiões delimitadas por polígonos regulares. A soma de Minkowski de dois conjuntos de pontos A e B é o conjunto $C = \{u \in R^2 | u = a + b, a \in A, b \in B\}$. Algumas aplicações interessantes:

- Para verificar se A e B possuem intersecção, basta verificar se $(0, 0) \in minkowski(A, -B)$.
- $(1/n) * minkowski(A_1, A_2, ..., A_n)$ representa todos os baricentros possíveis de pontos em $A_1, A_2, ..., A_n$.

```
polygon minkowski(polygon A, polygon B) {
    polygon P;
    A.pop_back(); B.pop_back();
    int s1 = leftmostIndex(A), n1 = A.size(); int s
    2 = leftmostIndex(B), n2 = B.size(); P.
    push_back(A[s1]+B[s2]);
    int i=s1+1, j=s2+1, c1=0, c2=0;
    point v1, v2;
    while (c1 < n1 || c2 < n2) {
        v1 = A[i]-A[(i-1+n1)%n1];
        v2 = B[j] -B[(j-1+n2)%n2];
        if (c2 == n2 || (c1 < n1 && cross(v1, v2) > 0)
            ) {
            P.push_back(P.back() + v1);
            i = (i+1)%n1; c1++;
        }
        else if (c1 == n1 || (c2 < n2 && cross(v1, v2)
            < 0)) {
            P.push_back(P.back() + v2);
            j = (j+1)%n2; c2++;
        }
        else {
            P.push_back(P.back() + (v1+v2));
            i = (i+1)%n1; c1++;
            j = (j+1)%n2; c2++;
        }
    }
    return make_polygon(P);
}
```

8.9 Minimum Enclosing Circle

Computa o círculo de raio mínimo que contém um conjunto de pontos. Baseado em permutação aleatória. Complexidade: *expected* $O(n)$.

```
circle minimumCircle(vector<point> p) {
    random_shuffle(p.begin(), p.end());ci
    rcle C = circle(p[0], 0.0);
    for (int i = 0; i < (int)p.size();i++){
        if(C.contains(p[i])) continue;
        C = circle(p[i], 0.0);
        for (int j = 0; j < i; j++){
            if (C.contains(p[j])) continue;
            C = circle((p[j] + p[i])*0.5, 0.5*dist(p[j
                ], p[i]));
            for (int k = 0; k < j;k++){
                if (C.contains(p[k])) continue;
                C = circumcircle(p[j], p[i], p[k]);
            }
        }
    }
    return C;
}
```

8.10 Ponto inteiro

Algumas funções de ponto usando apenas inteiros: *insideCircle* e o comparador polar *polarCmp* para $x, y > 10^9$.

```
typedef long long ll;

struct point_i{
    ll x, y;
    point_i() { x = y = 0; }
    point_i(ll _x, ll _y) : x(_x), y(_y) {}
    bool operator < (point_i other) const { if(
        x != other.x) return x < other.x; else
        return y < other.y;
    }
    bool operator == (point_i other) const{
        return (x == other.x && y == other.y);
    }
};
 ll cross(point_i a, point_i b){
    return a.x*b.y - a.y*b.x;
}
int insideCircle(point_i p, point_i c, int r) {l
    l dx = p.x - c.x, dy = p.y - c.y;
    ll Euc = dx * dx + dy * dy, rSq = r * r;
    return Euc < rSq ? 0 : Euc == rSq ? 1 : 2;
} //0 = inside/1 = border/2 = outside
bool polarCmp(point_i a, point_i b){
    if (b.y*a.y > 0) return cross(a, b) > 0;
    else if(b.y == 0 && b.x > 0) return false; else i
    f(a.y == 0 && a.x > 0) return true; else
    return b.y < a.y;
}
```

8.11 Ponto 3D

```
#include <cstdio>
#include <cmath>
#define EPS 1e-9

struct point{
    double x, y, z;
    point() { x = y = z = 0.0; }
    point(double _x, double _y, double _z) : x(_x), y
        (_y), z(_z) {}
    double norm() { return hypot(x, y, z); }
    point normalized() {
        return point(x,y,z)*(1.0/norm());
    }
    bool operator < (point other) const{
        if (fabs(x - other.x) > EPS) return x < other.
            x;
        else if (fabs(y - other.y) > EPS) return y < o
            ther.y;
        else return z < other.z;
    }
    bool operator == (point other) const {
        return (fabs(x - other.x) < EPS && fabs(y -
            other.y) < EPS && fabs(z - other.z) < EPS)
            ;
    }
    point operator +(point other) const{
        return point(x + other.x, y + other.y, z +
            other.z);
    }
    point operator -(point other) const{
        return point(x - other.x, y - other.y, z -
            other.z);
    }
    point operator *(double k) const {
        return point(x*k, y*k, z*k);
    }
};
double dist(point p1, point p2) {
    return (p1-p2).norm();
}
double inner(point p1, point p2) {
    return p1.x*p2.x + p1.y*p2.y + p1.z*p2.z;
}
point cross(point p1, point p2) {
    point ans;
    ans.x = p1.y*p2.z - p1.z*p2.y;
    ans.y = p1.z*p2.x - p1.x*p2.z;
    ans.z = p1.x*p2.y - p1.y*p2.x;
    return ans;
}
bool collinear(point p, point q, point r) {
    return cross(p-q, r-p).norm() < EPS;
}
double angle(point a, point o, point b) {
    return acos(inner(a-o, b-o) / (dist(o,a)*dist(o,b
        )));
}
point proj(point u, point v){
    return v*(inner(u,v)/inner(v,v));
}
```

8.12 Triângulo 3D

```
struct triangle{
    point a, b, c;
    triangle() { a = b = c = point(); }
    triangle(point _a, point _b, point _c): a(_a), b
        (_b), c(_c) {}
    double perimeter() { return dist(a,b) + dist(b,c)
        + dist(c,a); }
    double semiPerimeter() { return perimeter()/2.0;
        }
    double area(){
        double s = semiPerimeter(), ab = dist(a,b),
            bc = dist(b,c), ca = dist(c,a);
        return sqrt(s*(s-ab)*(s-bc)*(s-ca));
    }
    double rInCircle() {
        return area()/semiPerimeter();
    }
    double rCircumCircle() {
        return dist(a,b)*dist(b,c)*dist(c,a)/(4.0*area
            ());
    }
    point normalVector(){
        return cross(y-x, z-x).normalized();
    }
    int isInside(point p){
        point n = normalVector();
        double u = proj(cross(b-a,p-a), n).normalized
            ()*proj(cross(b-a,c-a), n).normalized();
        double v = proj(cross(c-b,p-b), n).normalized
            ()*proj(cross(c-b,a-b), n).normalized();
        double w = proj(cross(a-c,p-c), n).normalized
            ()*proj(cross(a-c,b-c), n).normalized();
        if (u > 0.0 && v > 0.0 && w > 0.0) return 0;
        else if (u < 0.0 || v < 0.0 || w < 0.0) return
            2;
        else return 1;
    } //0 = inside/ 1 = border/ 2 = outside
    int isProjInside(point p){
        return isInside(p + proj(a-p, normalVector()))
            ;
    } //0 = inside/ 1 = border/ 2 = outside
};

double rInCircle(point a, point b, point c) {
    return triangle(a,b,c).rInCircle();
}

double rCircumCircle(point a, point b, point c) {
    return triangle(a,b,c).rCircumCircle();
}

int isProjInsideTriangle(point a, point b, point c,
    point p){
    return triangle(a,b,c).isProjInside(p);
} //0 = inside/ 1 = border/ 2 = outside
```

8.13 Linha 3D

Desta vez a linha é implementada com um ponto de referência e um vetor base. *distVector* é um vetor que é perpendicular a ambas as linhas e tem como comprimento a distância entre elas. *distVector* é a "ponte"de *a* a *b* de menor caminho entre as duas linhas. *distVectorBasePoint* é o ponto da linha *a* de onde sai o *distVector*. *distVectorEndPoint* é o ponto na linha *b* onde chega a ponte.

```
struct line{
    point r;
    point v;
    line(point _r, point _v){
        v = _v; r = _r;
    }
    bool operator == (line other) const {
        return fabs(cross(r-other.r,v).norm()) < EPS
            && fabs(cross(r-other.r,other.v).norm
            ()) < EPS;
    }
};

point distVector(line l, point p){
    point dr = p - l.r;
    return dr - proj(dr,l.v);
}

point distVectorBasePoint(line l, point p){
    return proj(p - l.r, l.v) + l.r;
}

point distVectorEndPoint(line l, point p){
    return p;
}

point distVector(line a, line b){
    point dr = b.r - a.r;
    point n = cross(a.v, b.v);
    if (n.norm() < EPS){
        return dr - proj(dr, a.v);
    }
    else return proj(dr,n);
}

double dist(line a, line b){
    return distVector(a, b).norm();
}

point distVectorBasePoint(line a, line b){
    if (cross(a.v, b.v).norm() < EPS) return a.r;
    point d = distVector(a, b);
    double lambda;
    if (fabs(b.v.x*a.v.y - a.v.x*b.v.y) > EPS)
        lambda = (b.v.x*(b.r.y-a.r.y-d.y) - b.v.y*(b
            .r.x-a.r.x-d.x))/(b.v.x*a.v.y - a.v.x*b.
            v.y);
    else if(fabs(b.v.x*a.v.z - a.v.x*b.v.z) > EPS)
        lambda = (b.v.x*(b.r.z-a.r.z-d.z) - b.v.z*(b
            .r.x-a.r.x-d.x))/(b.v.x*a.v.z - a.v.x*b.
            v.z);
    else if(fabs(b.v.z*a.v.y - a.v.z*b.v.y) > EPS)
        lambda = (b.v.z*(b.r.y-a.r.y-d.y) - b.v.y*(b
            .r.z-a.r.z-d.z))/(b.v.z*a.v.y - a.v.z*b.
            v.y);
    return a.r + (a.v*lambda);
}

point distVectorEndPoint(line a, line b){
    return distVectorBasePoint(a, b) + distVector(a,
        b);
}
```

8.14 Grande Círculo

Dado o raio da Terra e as coordenadas em latitude e longitude de dois pontos p e q, retorna o ângulo pOq. Retorna a distância mínima de viagem pela superfície.

```
#include <cstdio>
#include <cmath>

double gcTheta(double pLat, double pLong, double
    qLat, double qLong) {
  pLat *= PI / 180.0; pLong *= PI / 180.0; //
      convert degree to radian
  qLat *= PI / 180.0; qLong *= PI / 180.0;
  return acos(cos(pLat)*cos(pLong)*cos(qLat)*cos(
      qLong) +
      cos(pLat)*sin(pLong)*cos(qLat)*sin(qLong) +
      sin(pLat)*sin(qLat));
}

double gcDistance(double pLat, double pLong, double
    qLat, double qLong, double radius) {
  return radius * gcTheta(pLat, pLong, qLat, qLong);
}
```

8.15 Coordenadas polares, cilíndricas e esféricas

Coordenadas polares:

$$x = rcos\varphi \qquad y = rsen\varphi \qquad dS = rdrd\varphi \tag{8.1}$$

Coordenadas cilíndricas:

$$x = rcos\varphi \qquad y = rsen\varphi \qquad z = z \tag{8.2}$$

$$d\dot\gamma = dr\hat r + rd\varphi\hat\varphi + dz\hat z \qquad dV = rdrd\varphi dz \tag{8.3}$$

$$d\dot S_r = rd\varphi dz\hat r \qquad d\dot S_\varphi = drdz\hat\varphi \qquad d\dot S_z = rdrd\varphi\hat z \tag{8.4}$$

$$\nabla f = \frac{\partial f}{\partial r}\hat r + \frac{1}{r}\frac{\partial f}{\partial\varphi}\hat\theta + \frac{\partial f}{\partial z}\hat z \tag{8.5}$$

$$\dot\nabla.\dot F \quad \frac{1}{}\frac{\partial}{} \quad {}_r) + \frac{1}{r}\frac{\partial F_\varphi}{\partial} + \frac{\partial F_z}{\partial z} \tag{8.6}$$

$$\frac{}{r}\frac{}{\partial r} \quad F = (\frac{1}{r}\frac{\partial F_z}{\partial\varphi} - \frac{\partial F_\varphi}{\partial z})\hat r +$$

$$(\frac{\partial F_r}{\partial z} - \frac{}{\partial r})\hat\varphi + \frac{}{}(\frac{1}{r}\frac{\partial}{\partial r}(rF_\varphi) - \frac{}{\partial\varphi})\hat z$$

Coordenadas esféricas:

$$x = rcos\varphi sen\theta \qquad y = rsen\varphi sen\theta \qquad z = rcos\theta \tag{8.8}$$

$$d\dot\gamma = dr\hat r + rd\theta\hat\theta + rsen\theta d\varphi\hat\varphi \tag{8.9}$$

$$d\dot S_r = r^2 sen\theta d\theta d\varphi\hat r \qquad d\dot S_\theta = rsen\theta d\varphi dr\hat\theta \qquad d\dot S_\varphi = rdrd\theta\hat\varphi \tag{8.10}$$

$$dV = r^2 sen\theta drd\theta d\varphi \qquad d\Omega = \frac{dS_r}{r^2} = sen\theta d\theta d\varphi \tag{8.11}$$

$$\nabla\boldsymbol{f} = \frac{\partial f}{\partial r}\hat{\boldsymbol r} + \frac{1}{r}\frac{\partial f}{\partial\theta}\boldsymbol\theta + \frac{1}{}\frac{\partial f}{}\boldsymbol\varphi \tag{8.12}$$

$$\dot\nabla.\dot F = \frac{1}{}\frac{\partial}{}{}^2 \; r\,F_r) + \frac{}{rsen\theta}\frac{}{\partial\theta}(sen\theta F_\theta) + \frac{}{rsen\theta}\frac{\partial F_\varphi}{\partial\varphi} \tag{8.13}$$

$$\nabla\times F = \frac{1}{rsen\theta}(\frac{\partial}{\partial\theta}(F_\varphi sen\theta) - \frac{}{\partial\varphi})\hat r + \tag{8.14}$$

$$\frac{}{r}(\frac{1}{sen\theta}\frac{}{\partial\varphi} - \frac{}{}(rF_\varphi))\hat\theta + \frac{}{}(\frac{}{}(rF_\theta) - \frac{}{\partial\theta})\hat\varphi$$

8.16 Cálculo Vetorial 2D

Curva regular no plano e comprimento do arco:

$$\dot\gamma(t),\ C^1,\ \dot\gamma^r(t) \neq 0 \qquad L(\gamma) = \int_a^b ||\dot\gamma^r(t)||dt \tag{8.15}$$

Reta tangente e normal:

$$T : X = \dot\gamma(t_0) + \lambda.\dot\gamma^r(t_0) \tag{8.16}$$

$$N : \{X \in R^2 :< X - \dot\gamma(t_0), \dot\gamma^r(t_0) >= 0\} \tag{8.17}$$

Curva de orientação invertida:

$$\dot\gamma^-(t) = \dot\gamma(a + b - t) \tag{8.18}$$

$$R(t) = \frac{1}{|k(t)|} \qquad \dot C(t) = \dot\gamma(t) + \frac{\dot N(t)}{K(t)} \tag{8.21}$$

Equações de Frenet:

$$\dot T^r(t) = K(s).\dot N(t) \qquad \dot N^r(t) = -K(t).\dot T(t) \tag{8.22}$$

Referencial de Frenet:

Teorema de Gauss no plano:

$$\vec{T}(t) = \frac{\vec{\gamma}'(t)}{||\vec{\gamma}'(t)||} \qquad \vec{N}(t) = (-T_y(t), T_x(t)) \tag{8.19}$$

Curvatura, raio e centro de curvatura:

$$K(t) = \frac{\vec{\gamma}''(t).\vec{N}(t)}{||\vec{\gamma}'(t)||^2} \tag{8.20}$$

$$\oint_{\partial S} \vec{F}.\vec{N}d\gamma = \iint_S \nabla.\vec{F}dxdy \tag{8.23}$$

Teorema de Green:

$$\oint_{\partial\Omega} Pdx + Qdy = \iint_\Omega \left(\frac{\partial Q}{\partial x} - \frac{\partial P}{\partial y}\right)dxdy \tag{8.24}$$

8.17 Cálculo Vetorial 3D

Referencial de Frenet:

$$\vec{T}(t) = \frac{\vec{\gamma}'(t)}{||\vec{\gamma}'(t)||} \quad \vec{B}(t) = \frac{\vec{\gamma}'(t) \times \vec{\gamma}''(t)}{||\vec{\gamma}'(t) \times \vec{\gamma}''(t)||} \quad \vec{N}(t) = \vec{B}(t) \times \vec{T}(t) \tag{8.25}$$

Curvatura e torção:

$$\tau(t) = \frac{< \vec{\gamma}'(t) \times \vec{\gamma}''(t), \vec{\gamma}'''(t) >}{||\vec{\gamma}'(t) \times \vec{\gamma}''(t)||^2} \quad K(t) = \frac{||\vec{\gamma}'(t) \times \vec{\gamma}''(t)||}{||\vec{\gamma}'(t)||^3} \tag{8.26}$$

Plano normal a $\vec{\gamma}(t_0)$:

$$< X - \vec{\gamma}(t_0), T(t_0) >= 0 \tag{8.27}$$

Equações de Frenet:

$$\vec{T}'(t) = K(t).\vec{N}(t) \tag{8.28}$$

$$\vec{N}'(t) = -K(t).\vec{T}(t) - \tau(t).\vec{B}(t) \tag{8.29}$$

$$\vec{B}'(t) = -\tau(t).\vec{N}(t) \tag{8.30}$$

Integral de linha de um campo escalar:

$$\int_\gamma f d\gamma = \int_a^b \mathbf{f}(\vec{\gamma}(t))||\vec{\gamma}'(t)||dt \tag{8.31}$$

Integral de linha de um campo vetorial:

$$\int_\gamma \vec{F}.d\vec{\gamma} = \int_a^b \vec{\mathbf{f}}(\vec{\gamma}(t)).\vec{\gamma}'(t)dt \tag{8.32}$$

Operador nabla:

$$\vec{\nabla} = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}) \tag{8.33}$$

Campo gradiente:

$$\vec{F}(x,y,z) = \vec{\nabla}\mathbf{f}(x,y,z) = (\frac{\partial \mathbf{f}}{\partial x}, \frac{\partial \mathbf{f}}{\partial y}, \frac{\partial \mathbf{f}}{\partial z})(x,y,z) \tag{8.34}$$

Campo conservativo:

$$\vec{F}(x,y,z) = \vec{\nabla}\mathbf{f}(x,y,z) \Leftrightarrow \int_\gamma \vec{F}.d\vec{\gamma} = 0 \tag{8.35}$$

Campo rotacional:

$$\vec{\nabla} \times \vec{F} = (\frac{\partial R}{\partial y} - \frac{\partial Q}{\partial z}, \frac{\partial P}{\partial z} - \frac{\partial R}{\partial x}, \frac{\partial Q}{\partial x} - \frac{\partial P}{\partial y}) \tag{8.36}$$

Superfície parametrizada:

$$\vec{S}(u,v) = (x(u,v), y(u,v), z(u,v)) \tag{8.38}$$

$$S_u(u,v) = (\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u}), \; \vec{S}_v(u,v) = (\frac{\partial x}{\partial v}, \frac{\partial y}{\partial v}, \frac{\partial z}{\partial v}) \tag{8.39}$$

Vetor normal à superfície:

$$\vec{N}(u,v) = (\vec{S}_u \times \vec{S}_v)(u,v) \neq \vec{0} \tag{8.40}$$

Superfície diferenciável:

$$\vec{N}(u,v) \neq \vec{0} \tag{8.41}$$

Plano tangente à superfície:

$$< (x,y,z) - \vec{S}(u_0,v_0), \vec{N}(u_0,v_0) >= 0 \tag{8.42}$$

Área da superfície:

$$A(S) = \iint_U ||\vec{N}(u,v)||dudv \tag{8.43}$$

Integral de superfície de um campo escalar:

$$\iint_S f dS = \iint_U \mathbf{f}(\vec{S}(u,v))||\vec{N}(u,v)||dudv \tag{8.44}$$

Integral de superfície de um campo vetorial:

$$\iint_S \vec{F}.d\vec{S} = \iint_U \vec{F}(\vec{S}(u,v)).\vec{N}(u,v)dudv \tag{8.45}$$

Massa e centro de massa:

$$M = \int_\gamma \rho(t)d\vec{\gamma}, \qquad \bar{x}M = \int_\gamma \vec{x}(t)\rho(t)d\vec{\gamma} \tag{8.46}$$

$$M = \iint_S \rho(u,v)ds \quad \bar{x}M = \iint_S \vec{x}(u,v)\rho(u,v)ds \tag{8.47}$$

$$M = \iiint_V \rho(x,y,z)dv \quad \bar{x}M = \iint_S \vec{x}(x,y,z)\rho(z,y,z)dv \tag{8.48}$$

Teorema de Pappus para a área, d = distância entre $\bar{x}$ e o eixo de rotação:

$$A(S) = 2\pi d L(C) \tag{8.49}$$

Teorema de Pappus para o volume, d = distância entre $\bar{x}$ e o eixo de rotação:

$$V(\Omega) = 2\pi d A(S) \tag{8.50}$$

$\dot{F}$ é conservativo → $\dot{\nabla} \times \dot{F} = 0$ Campo divergente:

$$\nabla.F = (\frac{\partial P}{\partial x}, \frac{\partial Q}{\partial y}, \frac{\partial R}{\partial z}) \tag{8.37}$$

- $\dot{F}$ é solenoidal quando $\dot{\nabla}.\dot{F} = 0$
- Existe $\dot{t}$ tal que $\dot{F} = \times \dot{t} \Rightarrow .\dot{F} = 0$
- $\nabla. \qquad \nabla.$

$\nabla.F \neq 0 \Rightarrow$ não existe tt tal que $F = \nabla \times tt$

Teorema de Gauss no espaço:

$$_{\partial V} \dot{F}.\dot{N} d\gamma = {}_V \nabla.F dxdydz \tag{8.51}$$

Teorema de Stokes:

$$_{\partial S} \dot{F}.d\dot{\gamma} = {}_S \nabla \times \dot{F}.\dot{N} ds \tag{8.52}$$

8.18 Geometria Analítica

Pontos de intersecção de dois círculos:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{8.53}$$

$$l = \frac{r_1^2 - r_2^2 + d^2}{2d} \tag{8.54}$$

$$h = \sqrt{r_1^2 - l^2} \tag{8.55}$$

$$x = \frac{l}{d}(x_2 - x_1) \pm \frac{h}{d}(y_2 - y_1) + x_1 \tag{8.56}$$

$$y = \frac{l}{d}(y_2 - y_1) \mp \frac{h}{d}(x_2 - x_1) + y_1 \tag{8.57}$$

8.19 Ponto ótimo numa linha

Dado um conjunto de pontos $x[i]$, $0 \le i < N$, o ponto x que minimiza

$$\sum_{i=0}^{N-1} |x - x[i]|$$

é o ponto médio do vetor ordenado (mediana): $x[N/2]$, se N é ímpar ou qualquer ponto em $[x[N/2 - 1], x[N/2]]$, se N é par.

8.20 Equação da reta

A equação da reta que passa pelos pontos (x_1, y_1) e (x_2, y_2) é dada por:

$$(y_2 - y_1)x + (x_1 - x_2)y + (y_1 x_2 - x_1 y_2) = 0 \tag{8.58}$$

Capítulo 9

Miscelânea

9.1 Algoritmo de Mo

Resolve queries offline em $O((N + Q)\sqrt{N})$. É necessário que o update de queries seja $O(1)$.

Pra usar como árvore chamar $treemo(root)$. $arr[i]$ guarda a propriedade dos nós em pre-pos ordem, $inv[i]$ guarda de qual nó veio $arr[i]$. Para usar com as arestas colocar a propriedade em $c[u]$ a propriedade de $e(u, parent[u])$.

CUIDADO: no modo árvore ele muda o valor de N.

```
#include <cmath>
#include <cstring>
#include <iostream>
#include <algorithm>
using namespace std;
#define MAXN 200009

int arr[MAXN], N, sn, Q;
bool appeared[MAXN];
int inv[MAXN]; //mo tree

struct query {
    int l, r, id, lca;
    ll ans;
    query() { l=r=id=lca=-1; }

    query(int _id, int _l, int _r) {
        id = _id; l = _l; r = _r; lca = -1;
    }
} qrs[MAXN];
bool lrcomp(query a, query b) {
    if(a.l/sn != b.l/sn) return a.l/sn < b.l/sn;
    return a.r > b.r;
}
bool idcomp(query a, query b) {
    return a.id < b.id;
}
ll freq[MAXN], curAns;
void check(int i) {
    if (i < 0 || i >= N) return;
    //if (appeared[i]) { //mo array
    if (appeared[inv[i]]) { //mo tree
        freq[arr[i]]--;
        if (freq[arr[i]] == 0) curAns--;
    }
    else {
        if (freq[arr[i]] == 0) curAns++;
        freq[arr[i]]++;
    }
    appeared[inv[i]] = !appeared[inv[i]];   //mo tree
    //appeared[i] = !appeared[i];           //mo array
}
void mo() {
    sn = sqrt(N);
    sort(qrs, qrs+Q, &lrcomp);
    memset(&freq, 0, sizeof freq);
    memset(&appeared, false, sizeof appeared);
    int l = 1, r = 0;
    curAns = 0;
    for (int i=0; i<Q; i++){
        query & q = qrs[i];
        while (r > q.r) check(r--);
        while (r < q.r) check(++r);
        while (l < q.l) check(l++);
        while (l > q.l) check(--l);
        check(q.lca);
        q.ans = curAns;
        check(q.lca);
    }
    sort(qrs, qrs+Q, &idcomp);
}


/* Codigo pra LCA aqui */

//Mo para arvores
int st[MAXN], en[MAXN], cnt, c[MAXN];
void prepos(int u, int p) {
    arr[cnt] = c[u]; inv[cnt] = u; st[u] = cnt++;
    for (int i=0; i<(int)adjList[u].size(); i++) {
        int v = adjList[u][i].first;
        if (v != p) prepos(v, u);
    }
    arr[cnt] = c[u]; inv[cnt] = u; en[u] = cnt++;
}
void treemo(int root) {
    cnt = 0;
    prepos(root, -1); computeP(root);
    N = cnt;
    for (int i=0, u, v, lca; i<Q; i++) {
        query &q = qrs[i];
        u = q.l; v = q.r; lca = LCA(u,v);
        if (st[u] > st[v]) swap(u, v);
        //if (lca == u) q.l = st[u]+1, q.lca = -1;
              //propriedade na aresta
        //else q.l = en[u], q.lca = -1;
        if(lca == u) q.l = st[u], q.lca = -1; //
            propriedade no noh
        else q.l = en[u], q.lca = st[lca]; q
        .r = st[v];
    }
    mo();
}
```

9.2 Algoritmo de Mo sem remoção

Algoritmo de Mo $O((N+Q)\sqrt{N})$ para responder queries sem suporte da operação de remoção (Ex: Union-find). Preencher as funções: *add*(*i*), adiciona o elemento *i* à estrutura; *save*() cria um ponto de retorno no tempo da estrutura; *reset*(), retorna ao último ponto salvo; *clear*(), limpa a estrutura. Atualizar e limpar *curAns*.

```
#include <algorithm>
#include <vector>
using namespace std;
#define MAXN 100009
#define MAXS 320

typedef pair<int, int> ii;
int arr[MAXN], N, sn, ans[i], curAns;
vector<ii> queries[MAXN][MAXS];

void add(int i) { ... }
void save() { ... }
void reset() { ... }
void clear() { ... }

void solvequeries(int l, int r, int bl) {
    for (int k=0; k<(int)queries[l][bl].size(); k++) {
        ii cur = queries[l][bl][k];
        while (r < cur.first) add(++r); ans
        [cur.second] = curAns;
    }
}

void solveblock(int bl) {
    clear();
    int r = min(bl * sn, N);
    for (int l=r; l > 0; l--) {
        add(l);
        save();
        solvequeries(l, r, bl);
        reset();
    }
}

void blockquery(int l, int r, int id) {
    clear();
    for (int i=l; i<=r; i++) add(i); ans
    [id] = curAns;
}

void mo() {
    sn = sqrt(N);
    for (int q=1, u, v; q<=Q; q++) {
        scanf("%d␣%d", &u, &v);
        if (u/sn == v/sn) blockquery(u, v, q);
        else queries[u][v/sn].push_back(ii(v, q));
    }
    for (int i=1; i<=N; i++) {
        for (int j=0; j<MAXS; j++) {
            sort(queries[i][j].begin(),
                queries[i][j].end());
        }
    }
    for (int j=1; j<MAXS; j++) solveblock(j);
}
```

9.3 Iteração sobre polyominos

Itera sobre todas as possíveis figuras de polyominos em uma grade. Posições com used 1 são proibidas. Colocar borda com 1's. Neste algoritmo em específico, calcula a soma máxima. used = 2 representa as posições usadas e path guarda tais posições.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <cstring>
#define MAXN 500
using namespace std;

typedef pair<int, int> ii;
ii coord[MAXN], path[MAXN];
int used[MAXN][MAXN];
int mat[MAXN][MAXN], maxsum, N, M, nused;

vector<ii> neightbors(ii pos){
    vector<ii> v;
    v.push_back(ii(pos.first+1, pos.second));
    v.push_back(ii(pos.first-1, pos.second));
    v.push_back(ii(pos.first, pos.second+1));
    v.push_back(ii(pos.first, pos.second-1));
    return v;
}

//0->unused, 1->to be used, 2->used
void dfs(int sum, int h, int n){
    used[coord[n].first][coord[n].second] = 2;
    path[h] = coord[n];
    if (h == M) {
        maxsum = max(maxsum, sum);
        used[coord[n].first][coord[n].second] = 1;
        return;
    }
    vector<ii> nei = neightbors(coord[n]);
    ii next;
    int nadded = 0;
    for (int i=0; i<4; i++){ next
        = nei[i];
        if (!used[next.first][next.second]) { used
            [next.first][next.second] = 1; coord[
            nused] = next;
            nadded++; nused++;
        }
    }
    for (int i = n+1, nextsum; i<nused; i++){
        nextsum = mat[coord[i].first][coord[i].second
            ];
        dfs(sum+ nextsum, h+1, i);
    }
    while (nadded-->0){
        nused--;
        next = coord[nused];
        used[next.first][next.second] = 0;
    }
    used[coord[n].first][coord[n].second] = 1;
}
```

9.4 Quadrádo Mágico Ímpar

Gera uma matriz quadrática $n \times n$ em $O(n^2)$, n ímpar, tal que a soma dos elementos ao longo de todas as linhas, de todas as colunas e das duas diagonais é a mesma. Os elementos vão de 1 a n^2. A matriz é indexada em 0.

```
#include <cstring>
#define MAXN 1009

int mat[MAXN][MAXN], n; //0-indexed

void magicsquare () {
    int i=n-1, j=n / 2;
    memset(&mat, 0, sizeof mat); for
    (int k=1; k<=n*n; k++){
        mat[i][j] = k;
        if (mat[(i+1)%n][(j+1)%n] > 0) {
            i = (i-1+n)%n;
        }
        else{
            i = (i+1)%n;
            j = (j+1)%n;
        }
    }
}
```

9.5 Expressão Parentética para Polonesa

Dada uma expressão matemática na forma parentética, converte para a forma polonesa e retorna o tamanho da string na forma polonesa. Ex.: $(2 * 4/a \wedge b)/(2 * c) \rightarrow 24 * ab \wedge /2c * /$.

```
#include <map>
#include <stack>
using namespace std;

inline bool isOp (char c) {
    return c=='+' || c=='-' || c=='*' || c=='/' || c
        =='^';
}

inline bool isCarac (char c) {
    return (c>='a' && c<='z') || (c>='A' && c<='Z')
        || (c>='0' && c<='9');

}

int paren2polish (char* paren, char* polish) {
    map<char, int> prec;
    prec['('] = 0;
    prec['+'] = prec['-'] = 1;
    prec['*'] = prec['/'] = 2; prec
    ['^'] = 3;
    int len = 0;
    stack<char> op;
    for (int i=0; paren[i]>0; i++){
        if (isOp(paren[i])) {
            while (!op.empty() && prec[op.top()]>=
                prec[paren[i]]) {
                polish[len++] = op.top();
                op.pop();
            }
            op.push(paren[i]);
        }
        else if (paren[i]=='(')
            op.push('(');
        else if (paren[i]==')') {
            while (op.top() != '(') {
                polish[len++] = op.top();
                op.pop();
            }

            op.pop();
        }
        else if (isCarac(paren[i]))
            polish[len++] = paren[i];
    }
    while (!op.empty()) {
        polish[len++] = op.top();
        op.pop();
    }
    polish[len] = 0;
    return len;
}
```

9.6 Problema do histograma

Algoritmo $O(n)$ para resolver o problema do retângulo máximo em um histograma.

```
#include <stack>
#define MAXN 100009
#define INF (1LL<<60)
using namespace std;

typedef long long ll;

ll histogram (ll* vet, int n) { stack
    <ll> s;
    ll ans = 0, tp, cur;
    int i = 0;
    while (i < n || !s.empty()) {
        if (i < n && (s.empty() || vet[s.top()] <= vet[
            i])) s.push(i++);
        else {
            tp = s.top();
            s.pop();
            cur = vet[tp] * (s.empty() ? i : i - s.top
                () - 1);
            if (ans < cur) ans = cur;
        }
    }
    return ans;
}
```

9.7 Problema do casamento estável

Resolve o problema do casamento estável em $O(n^2)$. m é o número de homens, n é o número de mulheres, $L[i]$ é a lista de preferências do i-ésimo homem (melhor primeiro), $R[i][j]$ é a nota que a mulher i dá ao homem j. $R2L[i] == j$ retorna se a mulher i está casada com o homem j, $L2R[i] == j$ retorna se o homem i está casado com a mulher j.

```
#include <cstring>
#define MAXN 1009

int m, n, p[MAXN];
int L[MAXN][MAXN], R[MAXN][MAXN];
int R2L[MAXN], L2R[MAXN];

void stableMarriage() {
    memset(R2L, -1, sizeof(R2L));
    memset(p, 0, sizeof(p));
    for (int i = 0, wom, hubby; i < m; i++){
        for (int man = i; man >= 0;) {
            while (true) {
                wom = L[man][p[man]++];
                if (R2L[wom] < 0 || R[wom][man] > R[
                    wom][R2L[wom]]) break;
            }
            hubby = R2L[wom];
            R2L[L2R[man] = wom] = man;
            man = hubby;
        }
    }
}
```

9.8 Teoremas e Fórmulas

- Fórmula de Cayley: existem n^{n-2} árvores geradoras em um grafo completo de n vértices.
- Desarranjo: o número $der(n)$ de permutações de n elementos em que nenhum dos elementos fica na posição original é dado por: $der(n) = (n-1)(der(n-1) + der(n-2))$, onde $der(0) = 1$ e $der(1) = 0$.
- Teorema de Erdos Gallai: é condição suficiente para que uma array represente os graus dos vértices de um nó: $d_1 \geq d_2 \geq ... \geq d_n$, $\sum_{i=1}^{n} d_i = 2k$, $\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1} min(d_i, k)$.
- Fórmula de Euler para grafos planares: $V - E + F = 2$, onde F é o número de faces.
- Círculo de Moser: o número de peças em que um círculo pode ser divido por cordas ligadas a n pontos tais que não se tem 3 cordas internamente concorrentes é dada por: $g(n) = C^n_4 + C^n_2 + 1$.
- Teorema de Pick: se I é o número de pontos inteiros dentro de um polígono, A a área do polígono e b o número de pontos inteiros na borda, então $A = i + b/2 - 1$.
- O número de árvores geradores em um grafo bipartido completo é $m^{n-1} \times n^{m-1}$.
- Teorema de Kirchhoff: o número de árvores geradoras em um grafo é igual ao cofator da sua matriz laplaciana L. $L = D - A$, em que D é uma matriz diagonal em que $a_{ii} = d_i$ e A é a matriz de adjacência.
- Teorema de Konig: a cobertura mínima de vértices em um grafo bipartido (o número mínimo de vértices a serem removidos para se remover todas as arestas) é igual ao pareamento máximo do grafo.
- Teorema de Zeckendorf: qualquer inteiro positivo pode ser representado pela soma de números de Fibonacci que não inclua dois números consecutivos. Para achar essa soma, usar o algoritmo guloso, sempre procurando o maior número de fibonacci menor que o número.
- Teorema de Dilworth: em um DAG que representa um conjunto parcialmente ordenado, uma cadeia é um subconjunto de vértices tais que todos os pares dentro dele são comparáveis; uma anti-cadeia é um subconjunto tal que todos os pares de vértices dele são não comparáveis. O teorema afirma que a partição mínima em cadeias é igual ao comprimenton da maior anti-cadeia. Para computar, criar um grafo bipartido: para cada vértice x, duplicar para u_x e v_x. Uma aresta $x \rightarrow y$ é escrita como $u_x \rightarrow v_y$. O tamanho da partição mínima, também chamada de largura do conjunto, é N − o emparelhamento máximo.
- Teorema de Mirsky: semelhante ao teorema de Dilworth, o tamanho da partição mínima em anti-cadeias é igual ao comprimento da maior cadeia.